Tutorial

REALbasic®

Create your own software.

REALbasic 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 5.2, June, 2003.

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's integrated development environment is made up of a rich set of *graphical user interface* objects (commonly referred to as GUI), an object-oriented language, an object browser, and a debugger.

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 Mac OS and Windows 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 SimpleText, the text editor included with Macintosh computers or NotePad, the text editor that is included with Windows. Using REALbasic, you will be able to compile the application for Mac OS "classic", Mac OS X, and Windows 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 (SimpleText is built from over 20,000 lines of C/C++ programming code).

Who Should Use this Manual

The tutorial is written for someone who is new to programming. You do not need any knowledge of programming in order to complete this tutorial.

If you have some 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.

NOTE: 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 both the Windows and Macintosh versions of REALbasic. The interface design and feature set are identical on both platforms, so the differences between platforms are cosmetic and have to do with the differences between the Macintosh's "Aqua" interface and Windows XP's standard appearance setting.

Italic type is used to emphasize the first time a new term is used, and to highlight import 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". This is equivalent to "choose New 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 Macintosh, the Command key is the modifier; on Windows, the Ctrl key is the modifier. For example, the shortcut "⌘-O" is the Macintosh keyboard equivalent. It means to

hold down the Command key, press the "O" key, and then release the Command key. The shortcut "Ctrl+O" is the Windows keyboard equivalent and means to hold down the Ctrl key, press the "O" key, and release the Ctrl key.

NOTE: When you see a paragraph with a large "i" to its left, you will know that the information provided will enhance your understanding of REALbasic.

NOTE: 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 is of code too long to fit on one line in the printed manual. When this happens, the 'overflow' text is indented halfway across the page. It appears like this:

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

You should enter this text as one line of code in the Code Editor.

Whenever you run your application, REALbasic first checks your code for syntax errors as described in the section, "Automatic Debugging Features" on page 106. 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 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:

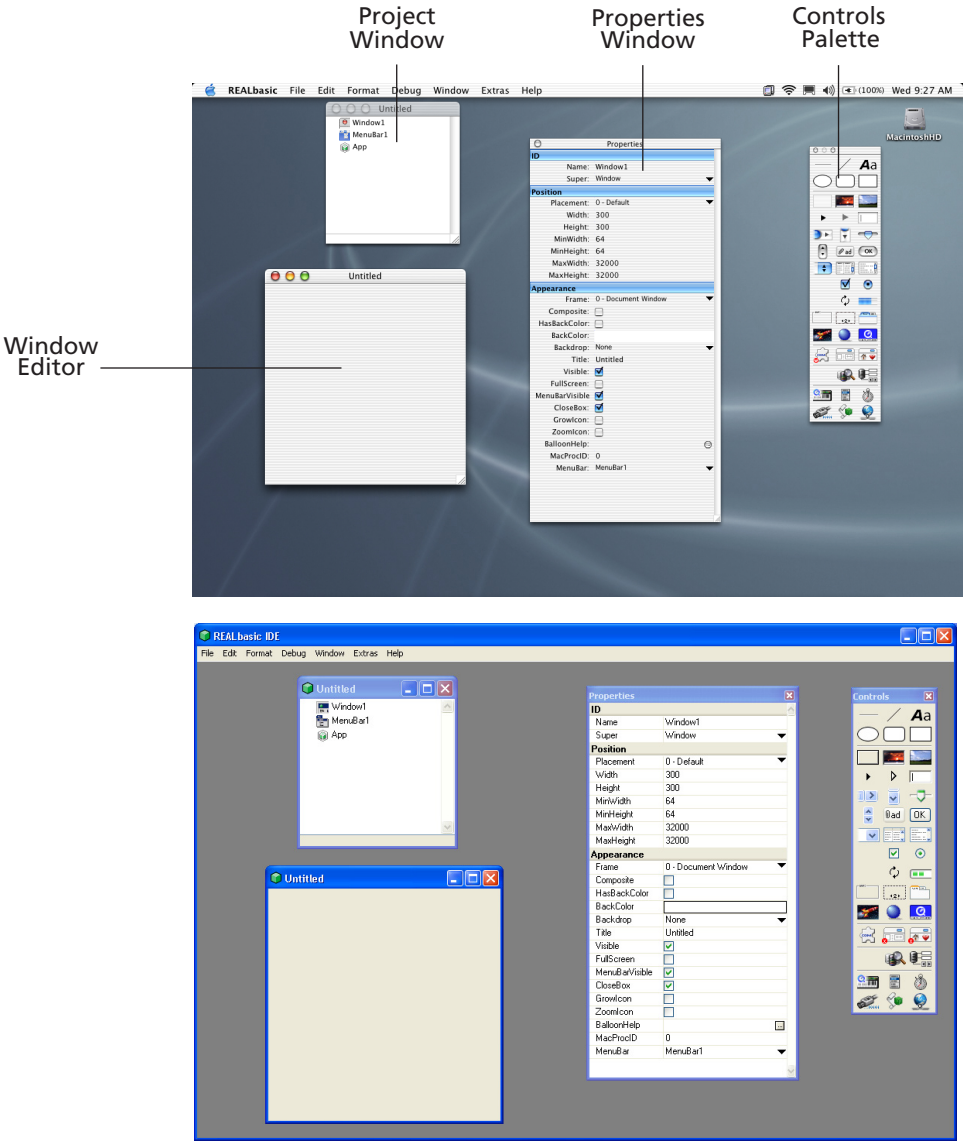
- Start Up REALbasic
- Identify REALbasic's windows
- Build a document window that will hold the text of your text editor
- Run your application

Starting Up REALbasic

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

After REALbasic has started up, your screen should look like Figure 1:

Figure 1. The REALbasic Development environment.



NOTE: In Figure 1 some windows have been moved from their default locations so you can see them better.

On Windows, the Window Editor and Project Window are inside the REALbasic IDE window and the Controls Palette and Properties Window float on top of the REALbasic IDE window.

REALbasic's Windows

As you can see in Figure 1 on page 12, there are four windows that open when you start up REALbasic:

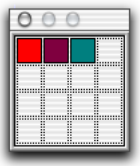
- The *Project Window* contains a list of all of the parts that make up your REALbasic application. For example, the Project Window includes items for all the windows that your application uses, the menu bars, and objects such as sounds, pictures, databases, and QuickTime movies. By default, the Project Window includes an item for the application's main window, *Window1*, its main menu bar, *MenuBar1*, and an item for code associated with the application as a whole, *App*. You double-click an item in the Project Window to edit or view it.
- The *Window Editor* is where you build all the windows, dialog boxes, alert boxes, and palettes for the application. Each such window is opened in its own Window Editor and all the windows in the application are listed in the Project Window. The Window Editor that opens when you create a new project is for the window listed in the Project Window, *Window1*.
- The *Controls Palette* contains icons representing interface objects that you can drag and drop onto the Window Editor to create your application's interface. Interface objects are referred to as *controls* in REALbasic.
- The *Properties Window* contains the list of the names of properties and their values for the *currently selected* object in your application. When you select a different object, the Properties Window changes to show the properties of that object. If no object is selected, the Properties Window is empty.

In addition, you can open the following windows:

- The *Colors Window* is used to store colors that you have defined for use in your REALbasic application. It consists of a palette of up to 16 colors. You can use the Colors Window to assign a color to a property that accepts a color. To assign a color to a palette element, click it to display the Apple Color Picker. To assign a color to an object property, drag a color from the Colors Window to a property value that accepts a color (such as the BackColor property of a Window object) or drag it to a line of code that assigns a color to an object that can store a color.

  The Colors Window with three colors is shown in Figure 2 on page 13.

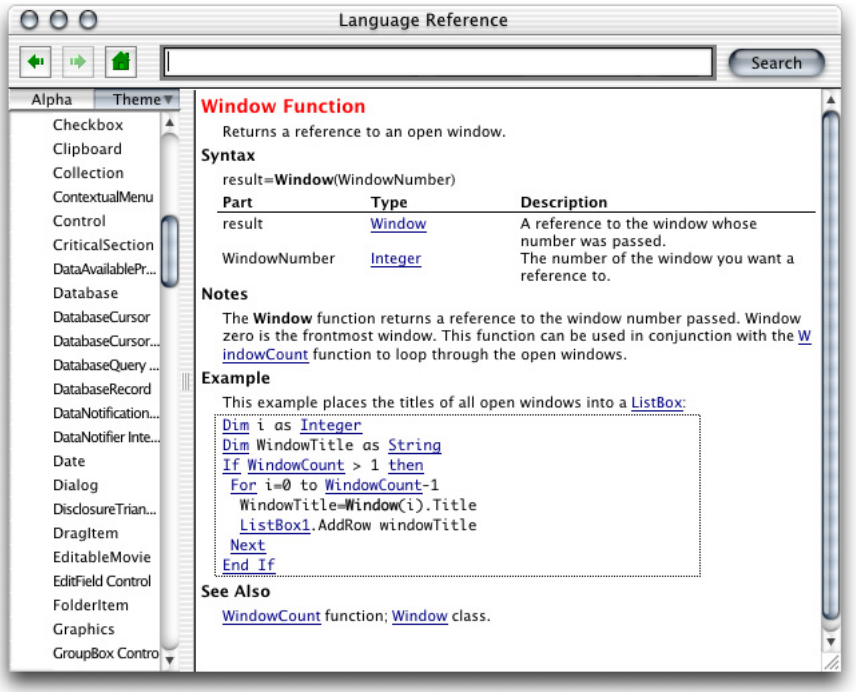
Figure 2. The Colors Window with three colors assigned.



- The *Online Language Reference Window* contains the REALbasic *Language Reference* (Choose Help ▶ Language Reference to display the online reference). Use it as a

convenient alternative to the printed or electronic (PDF) version of the *Language Reference*.

Figure 3. . The Language Reference window.



(The REALbasic *User's Guide* is not part of this help file, but you can access it onscreen by opening the User's Guide pdf file with Adobe Acrobat.)

You will learn more about the features of REALbasic as you progress through the Tutorial.

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, it opens an untitled window in a Window Editor with the name *Window1*. The name of the window is listed in the Project Window and its properties are shown in the Properties Window. This is as shown in Figure 1 on page 12.

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:

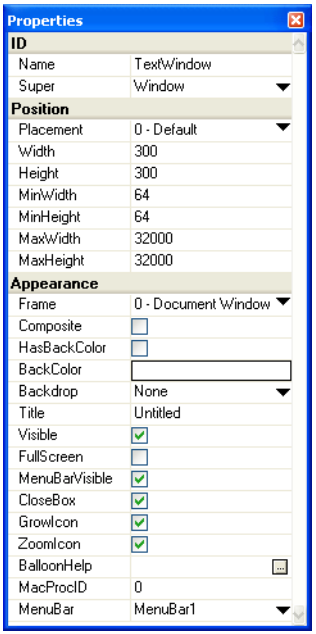
1 Click on Window1's name in the Project Window.

The Properties Window now shows Window1's current properties.

2 Change Window1's Name property to **TextWindow** and press the Return key.
When you press Return, Window1's name in the Project Window changes to TextWindow.
Next, we need to tell REALbasic to add the standard Grow Box and Zoom Icon to TextWindow so the user can resize the window.

3 In TextWindow's Properties Window, check the GrowIcon and ZoomIcon properties.
The Properties Window should now look like Figure 4.

Figure 4. TextWindow's Properties Window.



Later in the Tutorial, you will refer to TextWindow by name, so be sure to change it as shown here.

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 tool in the Controls Palette is shown in Figure 5.

Figure 5. The EditField Tool in the Controls Palette.

To add an EditField to TextWindow, do this:

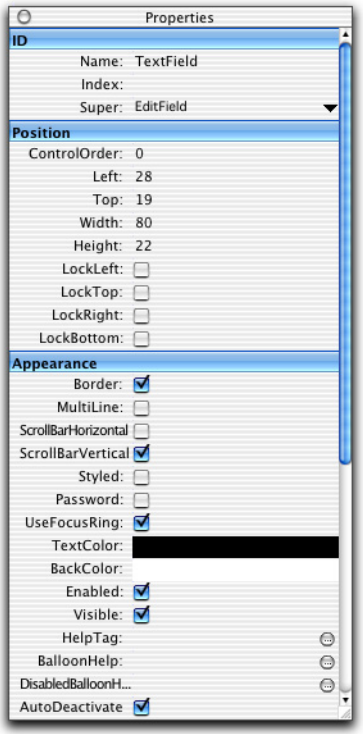
1. If TextWindow isn't already open in the Window Editor, double-click its name in the Project Window to open it.
2. Locate the EditField control in the Controls Palette and drag it anywhere onto TextWindow.

   Since the EditField is the currently selected object, the Properties Window now shows its properties.

   You use the controls in the Controls Palette as templates for your interface objects. When you drag a control from the Controls Palette 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 Window right now.
3. Use the Properties Window to change the Name property of the EditField from EditField1 to **TextField**.

   The Properties Window should look like Figure 6.

Figure 6. The Properties Window after renaming EditField1.



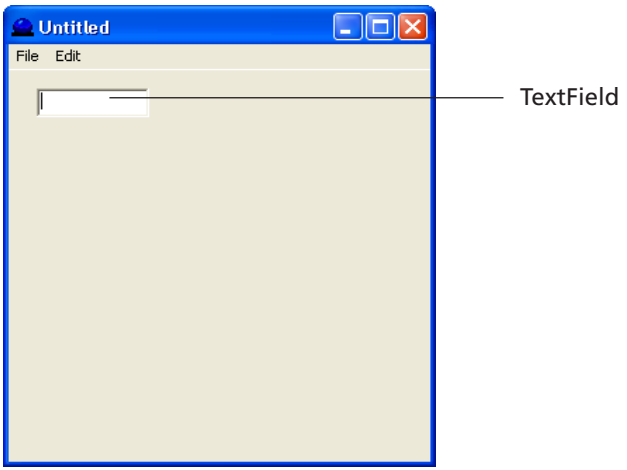
Although you have just started building your application, you may want to run it now, just to see what happens.

To run your application, do this:

1. Choose Debug ▶ Run (⌘-R on Macintosh or Ctrl+R on Windows).

TextWindow appears and should look similar to that shown in Figure 7.

Figure 7. The first run of your application.



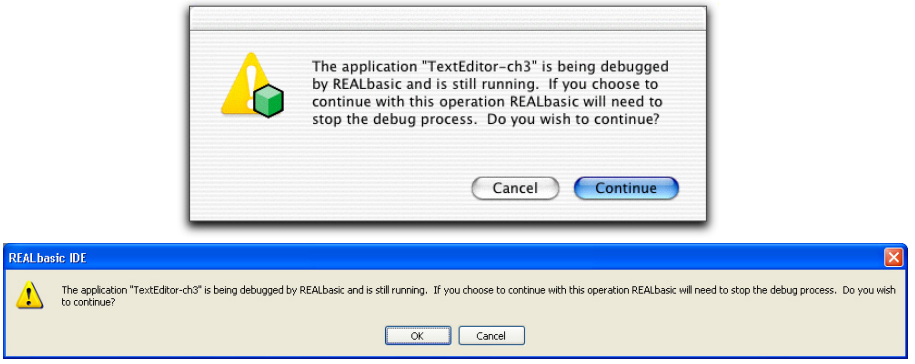
2 Type something in the TextField to try it out.

3 After you are done exploring, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development Environment (or use the command key equivalent, ⌘-Q or Ctrl+Q).

When you choose Run to launch your application, REALbasic compiles the code you've written and switches you to the *Runtime Environment (provided it finds no syntax errors)*. The Runtime Environment is where you do "test runs" and debug your application. After you quit your application, you return to the Development Environment.

In order to resume work within REALbasic's Development environment, you must quit out of the Runtime environment. You can't use the Development environment while the Runtime environment is active. If you try to modify your REALbasic application in the Development environment while the Runtime environment is active, REALbasic will display one of the messages shown in Figure 8 on page 18.

Figure 8. Message boxes indicating that you are trying to modify your application while it is running.



Click Continue (or OK, depending on your operating system) to quit out of the Runtime environment and resume editing your application in the Development environment.

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 Window to fix the left and top sides of TextField in the top-left corner of TextWindow.

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

1 Click TextField to select it, if it isn't already selected.

Refer to the Properties Window and locate the Top and Left properties. Click to the right of the value corresponding to the Left property and type **-1**. Press the Return key to set the property value.

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 Window 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 9 on page 19.

Figure 9. The Document Window with moved Textfield.

3 Now, drag the exposed (lower right) resizing handle of the TextField 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.

The window should now look as shown in Figure 10:

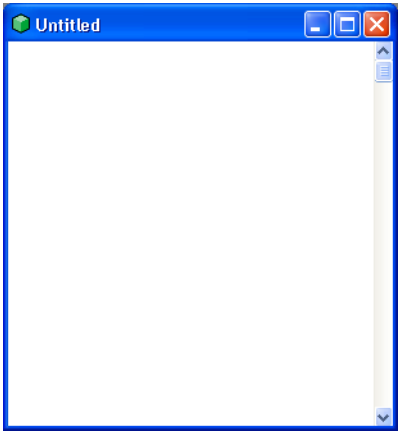
Figure 10. The TextWindow after resizing.

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 done simply by assigning the MultiLine property to the TextField.

5 Select the MultiLine property of the TextField using the checkbox in the Properties Window.

The TextField now has a scrollbar. The Document Window should look similar to Figure 11.

Figure 11. The TextWindow after adding the MultiLine property.



To run the application again, do this:

1. Choose Debug ▶ Run (⌘-R on Macintosh or Ctrl+R on Windows).
   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.
3. When you're done typing, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development Environment.
   Save your project now. Choose File ▶ Save (⌘-S). Save your project file with the name **TextEditor-ch2.rb**.
   The title of the Project Window changes to "TextEditor-ch2.rb".

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 Runtime environment and resizing the window. You'll see something like Figure 12. Then choose Edit ▶ Undo to restore the window to its original state.

Figure 12. Resizing the document window with a fixed-sized EditField.

In Figure 12, the EditField remains the size you specified in its Properties window, 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. Select the TextField. Locate the LockLeft, LockTop, LockRight, and LockBottom properties in the Properties Window and select them using their checkboxes.
2. Run your application (⌘-R on Macintosh or Ctrl+R on Windows) to test the resizing feature.
3. Choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.
4. Save your project once again to save the settings of the four Lock... properties.

At this point, you have created a very useful REALbasic object, TextWindow. 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, 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.

NOTE: The easiest way to reuse TextWindow in another project is to drag it from the Project Window to the Desktop (or any Finder directory). This gesture saves it as an exported REALbasic object. When you want to reuse it in another application, simply drag TextWindow from the Finder into the Project Window of another project.

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 Development environment	*REALbasic User's Guide*
REALbasic commands and language	*REALbasic Language Reference*

CHAPTER 3

Creating Menu Items

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.rb" in the Tutorial Files folder on the REALbasic CD.

Adding a Select All Menu Item

In this exercise, you will add a Select All menu item to the Edit menu. There are two required steps for implementing a menu item:

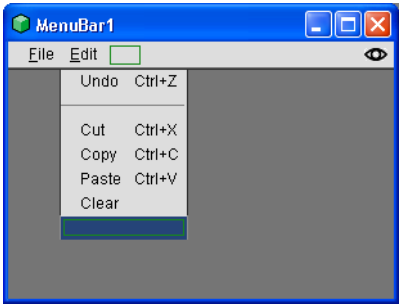
- Adding the menu item itself using the Menu Editor,
- Adding a *menu handler* method that tells REALbasic what to do when the user selects the menu item. The menu handler can call other methods.

Adding the Menu Item

To add a Select All item to the Edit menu, do this:

1. If it is not already visible, bring the Project Window to the front by choosing Window ▶ Project (⌘-0 on Macintosh or Ctrl+0 on Windows).
2. Double-click the MenuBar1 item in the Project Window to open the Menu Editor. MenuBar1 is the default menu bar that automatically applies to the whole application. You can add other menu bars and associate them with individual windows.
3. Click on the Edit menu in the Menu Editor.
4. Select the blank menu item at the end of the list, as shown in Figure 13.

Figure 13. The blank menu item selected.

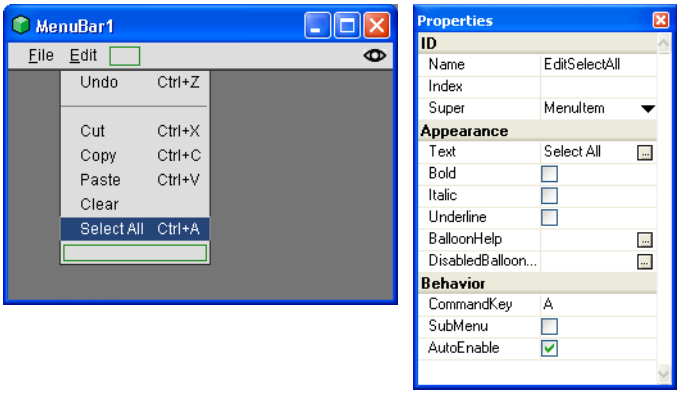


In the Menu Editor, there is always a blank menu item on each menu. You use it to create new menu items; it is not really a part of the menu and does not appear when you run your application.

NOTE: If you add a menu item by mistake, you can remove it by selecting it and pressing Delete.

5. In the Properties Window, enter **Select All** in the Text property area and press **Return**.
   The Name property is automatically filled in as EditSelectAll. No spaces are allowed in the Name property.
6. Assign the letter **A** to the Command Key property and press **Return**.
   Your Menu Editor and Properties window should look like those shown in Figure 14.

Figure 14. Menu Editor and Properties Window.



In the Menu Editor, the eye on the right side of the menu bar enables you to preview the menu and menubar for other platforms. It's a pop-up menu for all the platforms on which REALbasic applications run. If you wish, choose another platform from the menu and check out the menubar for the platform you aren't currently running.

The Properties window shows that the Super class (i.e., the class the object is cloned from) of this object is MenuItem. This means that it gets all the properties of this class and you can use the properties and methods of the MenuItem class to manage the Select All menu item.

7 Close the Menu Editor.

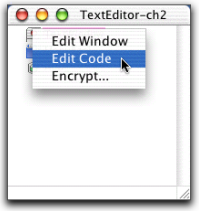
Assigning a Function to the Menu Item

The next thing we need to do is to tell REALbasic what to do when the user selects this menu item. This is called the *menu handler*. The menu handler is a method that runs automatically when the user chooses the Select All menu item. To write the menu handler, you need to open the Code Editor for the TextWindow. Each window has its own Code Editor, which holds the code that manages that window. Since the Select All menu item pertains to the contents of TextWindow, we want to put that menu item's menu handler there.

To open the Code Editor for TextWindow, do this:

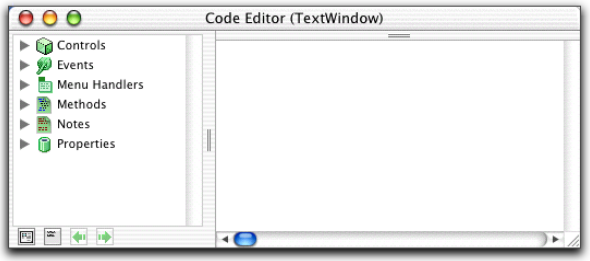
- On Macintosh, highlight the TextWindow item in the Project Window and press Option-Tab or Control-click on the TextWindow item and choose Edit Code from the contextual menu. On Windows, right-click on the TextWindow item and choose Edit Code from the contextual menu.

Figure 15. Opening the Code Editor for TextWindow.



The Code Editor for TextWindow appears. Notice that its Title bar indicates that it belongs to TextWindow. Later in the Tutorial, you will work with other windows and other items that also have their own Code Editors. Be sure you add your code to the correct Code Editor.

Figure 16. The Code Editor for TextWindow.

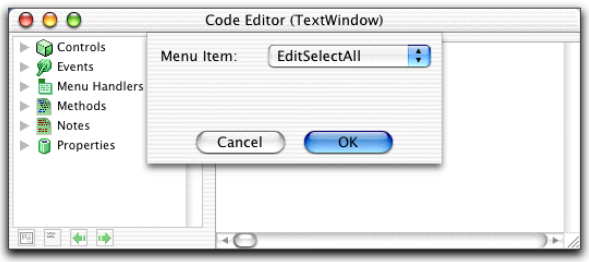


In this case, the menu handler performs the text selection.

To add the menu handler for the Select All menu item, do this:

1. With the Code Editor for TextWindow as the frontmost window, choose Edit ▶ New Menu Handler…
   A New Menu Handler dialog box appears. On Mac OS X, it is a Sheet Window; on Windows, it is a dialog box.
2. Choose EditSelectAll from the pop-up menu and click OK.

Figure 17. Creating a Menu Handler.



A new method called EditSelectAll appears in the TextWindow Code Editor in the Menu Handlers category, and the Code Editor displays the function declaration.

3 Type the following:

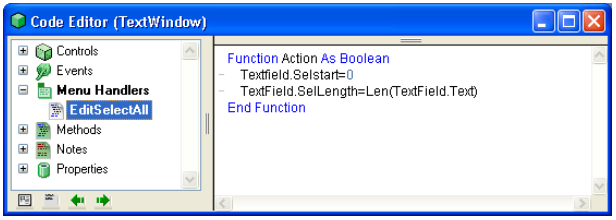
```
TextField.SelStart=0
TextField.SelLength=Len(TextField.Text)
```

This code uses two properties of an EditField control, SelStart and SelLength, to determine which text to select. The SelStart property sets the position of the first highlighted character in the selection and SelLength sets the length of the highlighted text, beginning at SelStart. The Len function is a global function that returns the length of the string of characters passed to it. In this case, it is passed all the text in TextField.

This code sets the start of the selection at the beginning of the text and sets the length of the selection to the length of the text in the TextField. (The Text property of an EditField contains the text in the EditField.)

The TextWindow Code Editor looks as shown in Figure 18.

Figure 18. The EditSelectAll menu handler with code entered.



Now that you've entered code for more than one event handler, the Left arrow below the Browser panel is enabled. The arrows are Back and Forward navigation buttons that work the same way as the navigation buttons in a Web browser. You can jump to the EnableMenuItems event handler by clicking the Back button and return to the menu handler by clicking the Forward button.

4 Save your project as **TextEditor-ch3.rb**.

5 Run your application by choosing Debug ▶ Run ((⌘-R or Ctrl+R).

NOTE: If you have any trouble compiling your application, check to see that you have renamed the EditField to TextField and the document window to TextWindow. If the control hasn't been renamed, REALbasic won't recognize references to TextField's properties. Also, be sure you are working with the Code Editor for TextWindow. The Title bar for the Code Editor should say "TextWindow," as in Figure 18.

6 Type some text into the text editor and try out the Select All menu item.
You should be able to select text using either the keyboard equivalent or the menu item.

When you're done, choose REALbasic ▶ Quit on Mac OS X (⌘-Q) or File ▶ Exit on Windows to quit your application and return to the Development environment.

Review

In this chapter you learned how to add menu items to your application, to enable them, and to handle their events.

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

CHAPTER 4

Working with Documents

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

- Create menu items for creating, 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 last chapter ("TextEditor-ch3.rb"). 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.

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. As you learned in the previous chapter, implementing a menu item involves two steps:

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

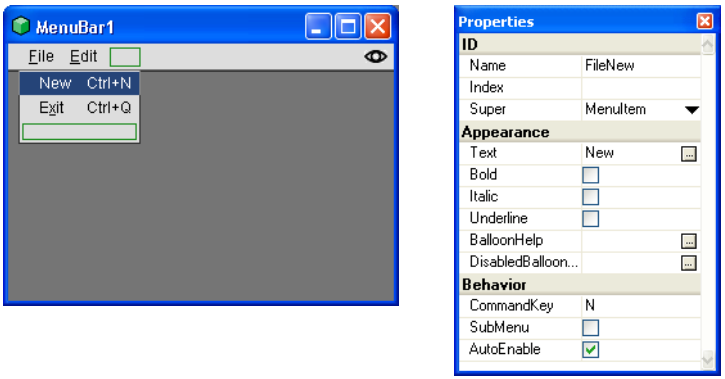
Creating the New Menu Item

To create the New menu item, do this:

1. Double-click the MenuBar1 item in the Project window and select the blank menu item in the File menu.
2. In the Properties Window, enter **New** in the Text area and **N** in the CommandKey property area.
   REALbasic automatically assigns the Name FileNew.
3. In the Menu Editor, drag the New menu item to the top of the menu.

   Your Menu Editor should look like Figure 19. When you select the New menu item, the Properties window for that Item should look as shown in Figure 19.

Figure 19. The Updated File menu.



4. Close the Menu Editor.

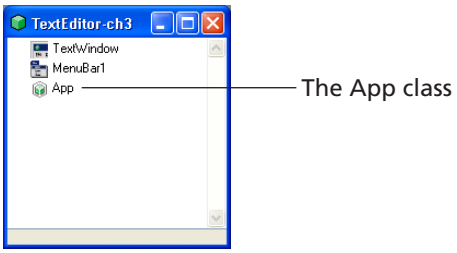
Handling the New Menu Item

After you add a menu item, you need to add a menu handler that tells REALbasic what to do when a user chooses the item. Without the menu handler, the menu item would do nothing.

The New menu item should be available at any time the application is running. That is, the user should be able to create a new text document even if no windows are open. For that reason, it would be a mistake to place the menu handler for the New menu item in the Code Editor for TextWindow. That would mean that the menu handler would be available only when a document window is already 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.

Figure 20. The App object in the Project Window.



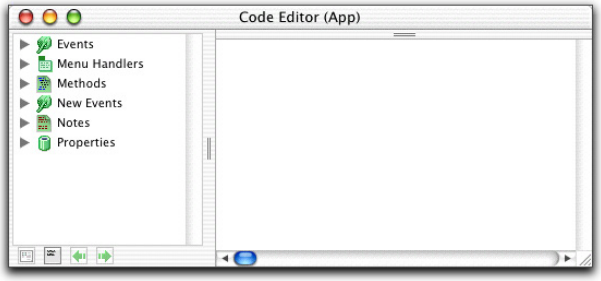
That place is the Application object. It is represented in the Project Window as the "App" class that appears below MenuBar1. The App class has a Code Editor of its own, but code that is placed in this Code Editor is available to the application as a whole—not just a particular window.

To write the Menu Handler for the New menu item, first open the Code Editor for the App class.

To handle the New menu item, do this:

1 Double-click App class in the Project Window.
The Code Editor for App appears. Notice how the Title Bar identifies the owner of this Code Editor as "App."

Figure 21. The Code Editor for the App class.



2 With the Code Editor for App in front, choose Edit ▶ New Menu Handler… to create a menu handler for the New menu item.

3 Select the FileNew menu handler and click OK.

4 Enter the following code into the Code Editor for the menu handler:

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

The Dim statement creates a new variable of type TextWindow but does not actually create the clone of the template. That is done in the next line with the New

operator. The New operator creates the clone and returns it in w. The new copy, w, is displayed immediately because the Visible property of TextWindow was set to True in Chapter 2.

5 Save your project as **TextEditor-ch4.rb** and then switch to the Runtime environment to test the New menu item.

NOTE: If the application doesn't compile correctly, be sure you have renamed Window1 to TextWindow.

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 Windows, it may create the new window directly on top of the existing window. On Macintosh, try it out even if no text window is open.

6 When you are finished, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit to return to the Development environment.

File Types

In order for your application to recognize specific types of files, you can 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 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 Types dialog box to specify valid file types. You display the File Types dialog box by choosing Edit ▶ File Types.

The TextEditor application must be able to open, modify, and save the files it creates. On Macintosh, it uses the TEXT file type and on Windows it uses RTF (Rich Text Format). The following procedure adds this capability to the application. TEXT is included as the default file type, but the RTF file type must be added.

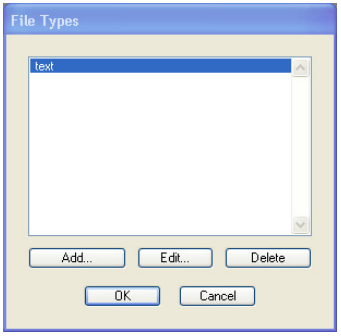
Adding a File Type

To add the RTF file type, do this:

1 Choose Edit ▶ File Types.

The File Types dialog box appears, with the TEXT File Type already defined.

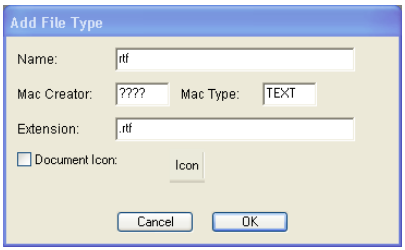
Figure 22. The File Types dialog box with the default file type defined.



2 Click the Add button to display the File Types editor.

3 Name the file type **rtf**, with a Mac Creator of **????**, Mac Type of **TEXT**, and Extension **.rtf** as shown in Figure 23 on page 33. Creator and Type are case-sensitive.

Figure 23. The RTF file type.



The Creator 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.

Saving Documents

In this section, you add the Save menu item to the application. Since saving a document can occur only when a document is already open, we will let the text editor window —rather than the application as a whole—manage this task. This means that you will add the menu handler for the Save menu item to the TextWindow's Code Editor. 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 and Select All menu items.
- **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 Bring the Menu Editor to the front or, if it is not open, double-click the MenuBar1 item in the Project Window.

2 Select the blank menu item in the File menu and use the Properties window to assign it the Text **Save** and CommandKey **S**.

REALbasic automatically assigns the name FileSave.

3 Deselect the AutoEnable property in the Behavior category.
The AutoEnable property enables the menu item all the time; this is not appropriate for a Save menu. It should not be enabled if there have been no changes to the document since it was last saved or when there are no document windows open.

4 Drag the Save menu item between the New and Quit/Exit items.
The Menu Editor should now look like Figure 24 on page 34.

Figure 24. The updated Menu Editor.

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

Adding Properties to TextWindow

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. A *property* of a window is a variable that can hold a value that describes some attribute of the window. 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 has changed since the last Save.

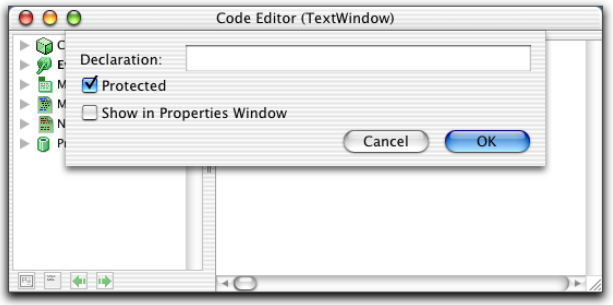
It makes sense to add these properties to TextWindow, since each copy of TextWindow that is opened in the application is associated with a specific file.

To add the properties to TextWindow, do this:

1 Select TextWindow in the Project Window and press Option-Tab to open TextWindow's Code Editor or, on Windows, right-click and choose Edit Code.

2 Choose Edit ▶ New Property…
The Property Declaration dialog box appears.

Figure 25. The Property Declaration dialog box.



3 Enter **Document as FolderItem**, deselect the **Protected** property, and then click OK (a *FolderItem* is the name of the REALbasic object that refers to files and folders).

4 Choose Edit ▶ New Property….
The Property Declaration dialog appears.

5 Enter **TextHasChanged as Boolean**, deselect the **Protected** property, and then click OK (Boolean is a data type that can take two values: *True* or *False*). In the section "Managing the TextHasChanged Property" on page 41 you will use this property to keep track of changes to the contents of the TextField.

6 Click the disclosure triangle next to the Properties category label in the Code Editor for TextWindow.
The Document and TextHasChanged properties are now listed. The Properties category label itself is now in boldface, indicating that properties have been added.

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 function as a flag to 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. Since the Save menu should only be enabled when there is a document window open, we need to use the EnableMenuItems event handler for TextWindow.

To enable the menu item, do this:

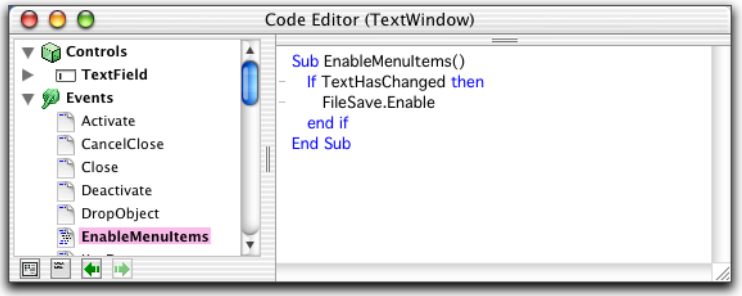
1 Click the disclosure triangle next to the Events category label in TextWindow's Code Editor to reveal the events (or double-click the Events label).

2 Select EnableMenuItems and enter the following code:

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

The Code Editor should look as shown in Figure 26.

Figure 26. The EnableMenuItems Event Handler.



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

3 Save your project.

Adding a SaveFile Method

Next, you need to specify the actions to be taken when the Save menu item is chosen by the user. This is done in the SaveFile method. This method will be called by the menu handler for the Save menu item as well as the menu handler for the Save As menu item that you will add in the section "Adding a Save As Menu Item" on page 42. This method will manage two 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.

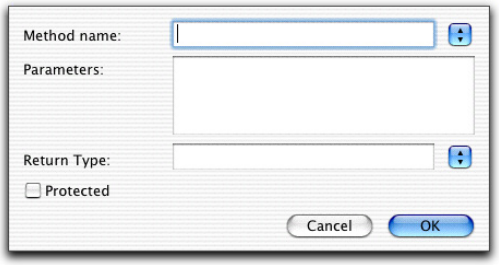
  In the latter case, the application must first present a save-file dialog box that lets the user enter a filename. In the former case, the application saves the document using the existing filename.

  In this method, the Boolean parameter DisplaySaveDialog is used to force the method to 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 With the Code Editor for the TextWindow window in the front, choose Edit ▶ New Method.
The New Method dialog box appears.

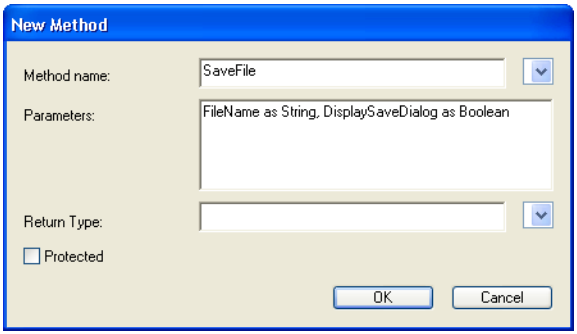
Figure 27. The New Method dialog box.



The New Method dialog 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. The parameters are values that are input to the method and the Return Type is the output from the method. This method has values that are input to the method but nothing is output.

2 Enter **SaveFile** as the method name and **FileName as String, DisplaySaveDialog As Boolean** in the Parameters area. Leave the Return Type area blank.
The dialog box should now look like Figure 28.

Figure 28. The New Method dialog box.



3 Click OK to close the dialog box.

The Code Editor for the SaveFile method appears. Note that the method name and parameters have been added. If you need to change the name or parameters, you can double-click the SaveFile method name in the Browser panel 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
 f=GetSaveFolderItem("rtf",FileName)
 #else
 f=GetSaveFolderItem("text",FileName)
 #endif
 If f <> nil then //if the user clicked Save
  Title=f.Name
  Document=f
 End if
End if
If Not DisplaySaveDialog then //user chose Save
  If Document <> Nil then
    Document.SaveStyledEditField TextField
    TextHasChanged=False
  End if
Elseif DisplaySaveDialog then //user chose SaveAs or New doc
  If Document <> Nil and f <> Nil then
    Document.SaveStyledEditField TextField
    TextHasChanged=False
  End if
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.

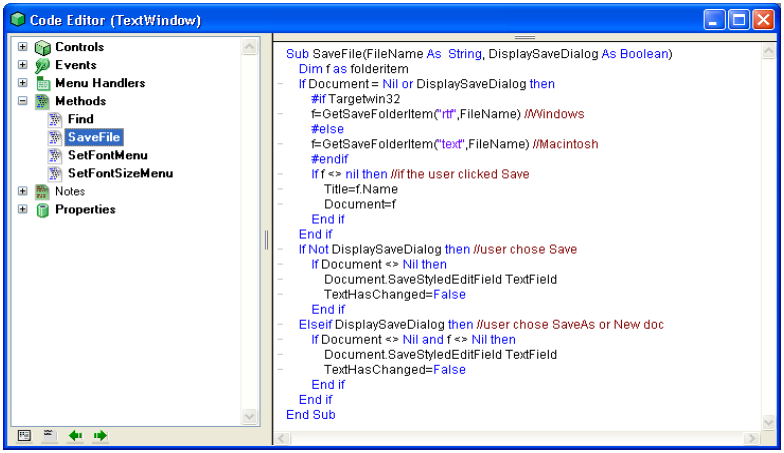
The logic of this method is as follows: If the Document property is undefined (i.e., its value is "Nil"), the document has not been saved, so the Save File dialog box must be presented. The GetSaveFolderItem function does this.

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 29 on page 39:

Figure 29. Code entered for SaveFile method.



We are not quite ready to call 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 the section "Managing the TextHasChanged Property" on page 41.

Using The Online Language Reference

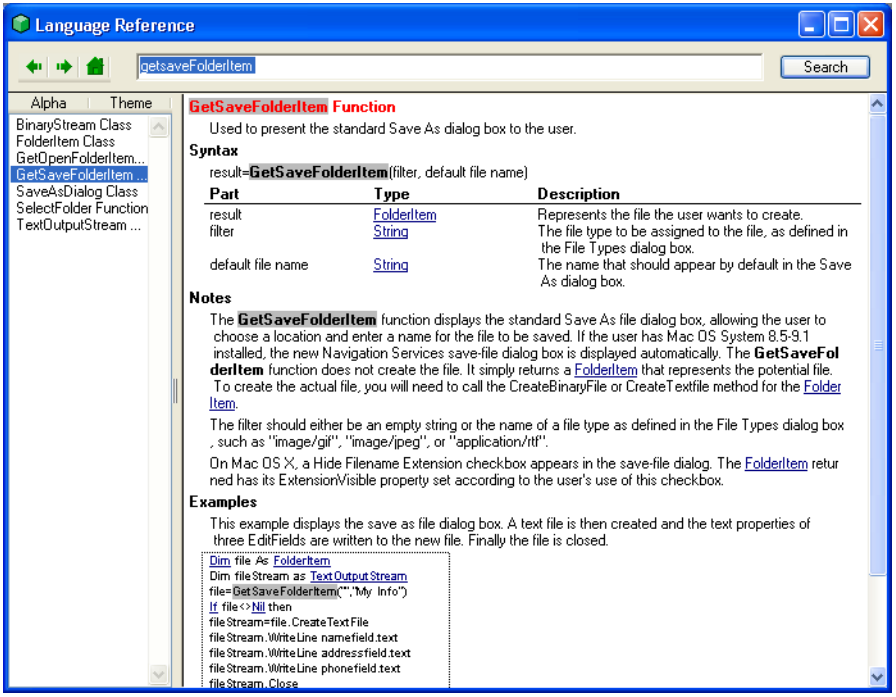
The SaveFile method uses two built-in methods to do the hard work: It calls the global method GetSaveFolderItem to present the save-file dialog box and the SaveStyledEditField method of the FolderItem class to save the contents of the EditField that is part of TextWindow. If you wish, you can look up these methods in the REALbasic *Language Reference* or, more conveniently, in the online version of the reference.

To look up GetSaveFolderItem, do this:

1 Choose Help ▶ Language Reference (⌘-1 or F1 on Windows).
The online help dialog box appears. The browser on the left lists all the main entries in the *Language Reference*, sorted by theme (category) or alphabetically. The default sort order is by theme, but you can list the items alphabetically by clicking the Alpha tab at the top. The header area contains a search field that you can use to find language elements or any other terms used in the reference.

2 Enter GetSaveFolderItem in the search field and click Search.
As usual, REALbasic tries to guess what you are typing as you are entering characters. Press Tab at any point to display a contextual menu or accept REALbasic's guess.
The window should look like Figure 30 on page 40. Notice that all instances of the term you searched for are highlighted.

Figure 30. The GetSaveFolderItem online documentation.



The main panel in Figure 30 presents the documentation for GetSaveFolderItem; hypertext links to related entries are in blue and are underlined. If you wish, you can click on a reference to FolderItem and then scroll down to read about the SaveStyledEditField method in the Methods table. The arrows in the header area are Back and Forward buttons that work the same as in an Internet browser.

Also, code examples that are shown in dotted rectangles in the online reference can be dragged into your Code Editor window.

3 When you are finished, click the Close box to put away the online reference.

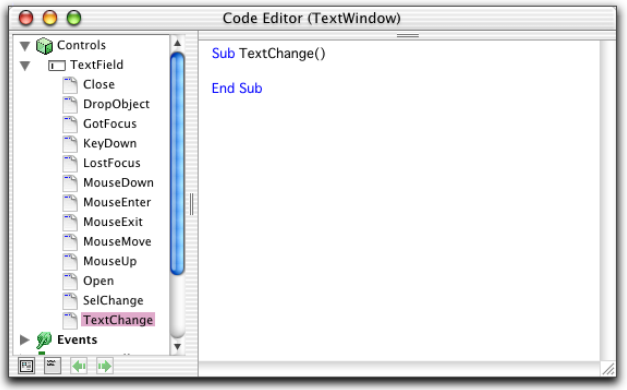
Managing the TextHasChanged Property

The TextHasChanged property that is used in the SaveFile method 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* is a method that runs automatically when 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. 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, bring the Code Editor for TextWindow to the front 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.

Figure 31. TextField's Event Handlers.



To manage the TextHasChanged flag, do this:

1 Highlight the TextChange event.

2 Add the following line of code to the blank 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.

NOTE: 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 installed and passes the value of False to the DisplaySavedDialog 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 since it will save the contents of a particular document window.

To handle the menu item, do this:

1. With the Code Editor for TextWindow in front, choose Edit ▶ New Menu Handler, select FileSave from the pop-up menu, and click OK.
   A new menu handler named FileSave is added to the Code Editor Browser.
2. Enter the following code:

```
SaveFile Title, False
```

   This menu handler calls another method, SaveFile, that does the work. The terms 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, a string, is the default name of the file to be saved and the second is a Boolean that tells SaveFile whether it needs to display a "save changes" dialog box[1]. The term "Title" is the Title property of the Window class—the text that appears in the window's Title bar.

   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.
3. Save your project.
4. Choose Debug ▶ Run (⌘-R or Ctrl+R) to test your application.
   Notice that the Save menu item is disabled initially.
5. 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.
6. Choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to quit your application and return to the Development Environment.

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. Double-click the MenuBar1 item in the Project Window and add a new menu item to the File menu with the Text **Save As...**.Use three periods instead of the ellipsis character (which consists of three dots).

1. …which we haven't created yet! It will be added in the next chapter.

REALbasic automatically sets the Name property to FileSaveAs.

2 Drag it between the Save and Quit (Exit on Windows) menu items.

3 From TextWindow's Code Editor, choose Edit ▶ New Menu Handler and choose FileSaveAs.

4 Enter the following code:

```
SaveFile Title, True
```

This menu handler also 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.

5 Save your project and test the Save and Save As commands by choosing Debug ▶ Run. 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.

6 When you are finished, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

Adding an Open Menu Item

Now that you have implemented the Save and Save As commands, 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.

Creating the Open Menu Item

To create the Open menu item, do this:

1 Double-click the MenuBar1 item in the Project window and select the blank menu item in the File menu.

NOTE: If the Menu Editor does not open, check to see if the Debug ▶ Kill menu item is active. If it is, choose it.

2 In the Properties Window, enter **Open...** in the Text property area and **O** in the CommandKey property area.

3 Drag the Open menu item between the New and Save menu items.

Like the New menu item, the Open menu item should be available even if there are no open document windows. Therefore you need to use the App object to manage the menu item.

Handling the Menu Item

Since the Open menu item must work when no windows are open, its menu handler belongs in the Application object.

To handle the Open File menu item, do this:

1 With the Code Editor for the App class in front, choose Edit ▶ New Menu Handler… to create a menu handler for Open.

If the Code Editor for the App class is not open, double-click the App item in the Project Window.

2 Select the FileOpen menu handler in the New Menu Handler dialog and enter the following code into the FileOpen menu handler in the Code Editor:

```
Dim f as FolderItem
Dim w as TextWindow
f=GetOpenFolderItem("text;rtf") //displays open-file dialog
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 which displays the open-file dialog box and returns a reference to the document that the user selects. The parameter ("text;rtf") instructs the GetOpenFolderItem method to display only text documents. The "text" file type was added to the project using the File Types dialog box (see "Adding a File Type" on page 32).

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.

3 Save your project.

4 Choose Debug ▶ Run 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.rb project on your REALbasic CD and compare your project to that one.

5 When you are finished, quit the application to return to the Development environment.

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*

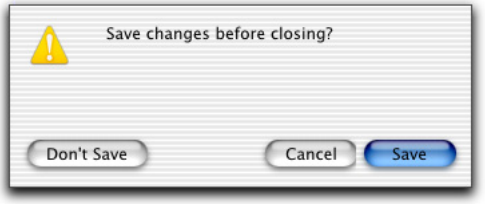
CHAPTER 5

Adding a "Save Changes" Dialog Box

In "well-behaved" applications, the application gives the user a chance to save changes to all open documents whenever he closes a window or chooses Quit 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.



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.rb"). Launch REALbasic and open the project file. If you need to, you can use the "TextEditor-ch4.rb"file that is in the Tutorial Files folder on the REALbasic CD.

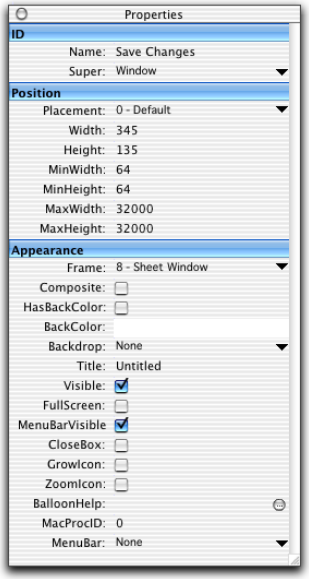
Creating the Dialog Box

You create the dialog box by adding a new window to the project and adding the icon, button, and text controls to the window. The controls are added to a window by dragging them from the Controls Palette (as you added the EditField in Chapter 2).

To create the dialog box, do this:

1. With the Project Window as the frontmost window, choose File ▶ New Window.
   REALbasic adds a window to the project and names it Window1.
2. Use Window1's Properties Window to change its name to **SaveChanges**.
3. Change the window's Width to **345** and Height to **135**.
4. Change its Frame Property to **Sheet Window**.
   A "sheet window" is the official name for those new modal windows in Mac OS X that appear to drop down from the parent window's Title bar. On other operating systems, a Sheet Window looks like a regular modal dialog box on that operating system.
5. Deselect the CloseBox, GrowIcon, and ZoomIcon checkboxes.
   This window will be used as a fixed-size dialog box.
   The window's Properties Window should now look like this:

Figure 33. Properties of the SaveChanges window.



The following steps add the controls to the empty dialog box.

1. Use the Controls Palette to drag a Canvas control  to the top-left area of the SaveChanges window. This control will display the caution icon shown in Figure 32 on page 47.
   The Canvas control is a blank drawing canvas. It comes equipped with a set of drawing tools that you use programmatically to customize its appearance.
2. Click on the Canvas control and, using the Properties Window, assign it the properties shown in Table 1.

Table 1: Properties of the Canvas Control.

Property	Value
Left	13
Top	13
Width	46
Height	46

3. Open the Code Editor for SaveChanges by selecting SaveChanges in the Project Window and pressing Option-Tab (Macintosh) or right-clicking and choosing Edit Code.
4. Expand the Controls item and then expand the Canvas1 item.
   The list of event handlers for a Canvas control appears. To create the caution icon, you will add code to the Paint event handler. This event handler runs whenever REALbasic detects that the Canvas control needs to be redrawn.
5. Highlight the Paint event.
   Notice that the parameter line for the Paint event handler contains one parameter—"g as Graphics." The Graphics class in REALbasic contains the methods that allow you to completely customize the appearance of the Canvas control. They are your drawing tools.
6. Enter the following line of code:

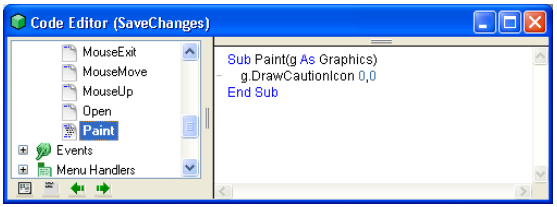
```
g.DrawCautionIcon 0,0
```

The syntax "g.DrawCautionIcon" indicates that DrawCautionIcon is a method within the Graphics class. This is a built-in method in the Graphics class that draws the icon.

NOTE: If you wish, check out the drawing tools available in the Graphics class using the online reference. You will see that the DrawCautionIcon method takes two parameters — the x and y coordinates where the top-left corner of the caution icon is to be positioned.

The Code Editor should look like Figure 34 on page 50.

Figure 34. Code for the Caution Icon.



Next, you need to add the text that appears to the right of the Caution icon. This is done by placing a StaticText control in the dialog box.

To add static text to the dialog box, do this:

1 Click on the StaticText control Aa in the Controls Palette and drag it to the right of the Canvas control in the SaveChanges Window. Notice how alignment lines appear and it snaps in alignment with the top of the Canvas control.

2 With the StaticText control selected, assign it the properties shown below.

Table 2: Properties of the StaticText Control.

Property	Value
Left	72
Top	13
Width	255
Height	20
Text	Save changes before closing?
MultiLine	Checked (Yes)

The next series of instructions adds the buttons that are placed below the Canvas and StaticText controls.

To add buttons to the dialog box, do this:

1 Using the Pushbutton control OK in the Controls Palette, drag three pushbuttons into the approximate positions shown in Figure 32 on page 47.

Drag the Don't Save pushbutton first and let REALbasic align it with the left edge of the Canvas control. Then drag the Cancel and Save pushbuttons into place, using the horizontal alignment line to align them with the bottom of the Don't Save button.

Next, you will assign properties to each button by successively selecting each button and changing its properties using the Properties Window. As with the other controls, you use the Properties Window to set the exact position of the object.

2 Select each pushbutton and make the property assignments shown in Table 3.

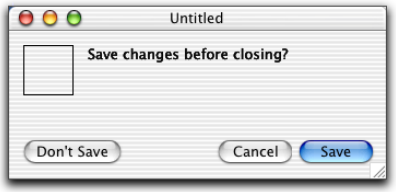
Table 3: Properties of the Don't Save, Cancel, and Save Pushbuttons.

Property	Pushbutton		
	Don't Save	**Cancel**	**Save**
Name	DontSave	Cancel	Save
Left	13	191	265
Top	100	100	100
Width	90	69	69
Height	20	20	20
Caption	Don't Save	Cancel	Save
Default	Not checked	Not checked	Checked
Cancel	Not Checked	Checked	Not checked
Enabled	Checked	Checked	Checked
Visible	Checked	Checked	Checked

As with all controls, the Name property is the internal name of the object that you use to refer to the object. The Caption property is the label that appears in the Pushbutton.

The Save Changes dialog box should now look like Figure 35.

Figure 35. The Save Changes dialog box in the Development environment.



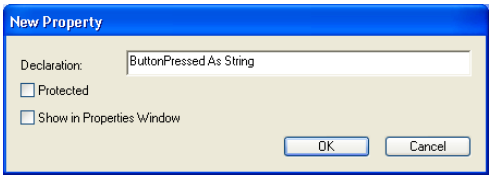
In the next series of steps, you assign a value to a property that identifies the button that is pressed. First you will declare the variable as a property of the SaveChanges window.

1 With the Code Editor for the SaveChanges window in front, choose Edit ▶ New Property.

Reminder: Every window has its own Code Editor. The Code Editor for the Save Changes dialog has "Code Editor (Save Changes)" in its title bar. You may have at least two Code Editors open right now. You always need to put your code in the correct Code Editor.

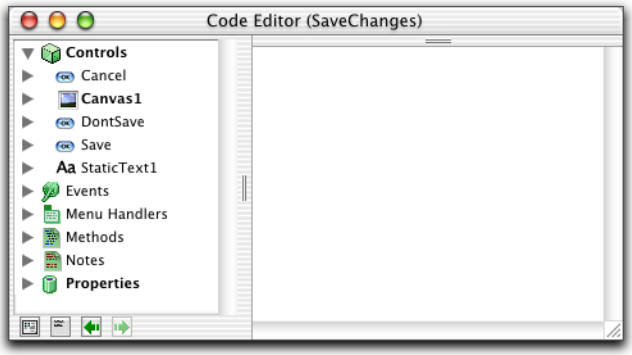
2 Enter **ButtonPressed as String** in the dialog box. Deselect the Protected property and click OK.
The New Property dialog box should look like this.

Figure 36. The ButtonPressed property.



3 Expand the Controls item in the Code Editor for the Save Changes window.
The three Pushbutton controls are listed by name, along with the Canvas control, as shown in Figure 37.

Figure 37. Controls in the Save Changes dialog box.



4 Expand the Cancel PushButton control and click the Action item.
The Action event handler runs when the user clicks a button.

5 Enter the following code:

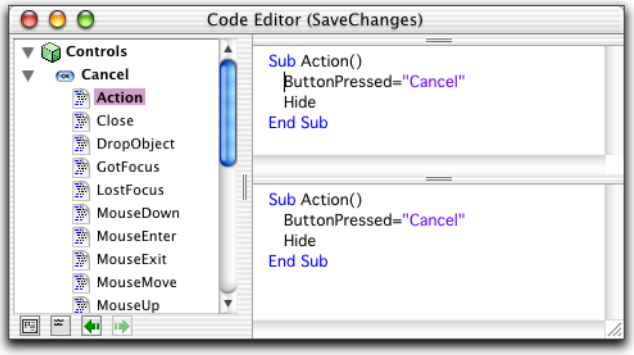
```
ButtonPressed="Cancel"
Hide
```

The code assigns a string to the ButtonPressed property so that we can later determine which button was pressed. The Hide statement is a method of the Window class that, not surprisingly, makes the window disappear.

You now need to add similar code to the Action events of the DontSave and Save buttons. One way to do this is copy and paste this code into the two other Action events and then edit the pasted code. You'll do this by splitting the Code Editor area into two panes, each showing a different event.

6 To split the code editing area into two panes, move the mouse to the divider, which is indicated by two horizontal lines above the editing area and just below the window's title bar. Drag downward when the pointer changes to this ‡.
Your Code Editor should now look like this.

Figure 38. The Code Editor after creating a second pane.

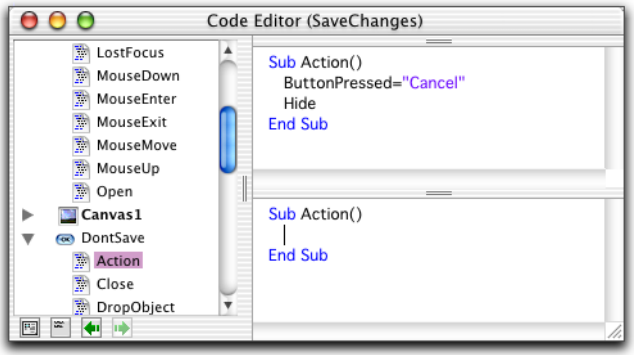


At present, the second pane shows a copy of the Cancel button's Action event. We need to use it to show the Action event for the DontSave button.

7 Place the insertion point in the bottom pane and then expand the DontSave button's events and click Action.

The bottom pane now shows the (empty) Action event for DontSave.

Figure 39. The Code Editor after selecting the DontSave button's Action event.



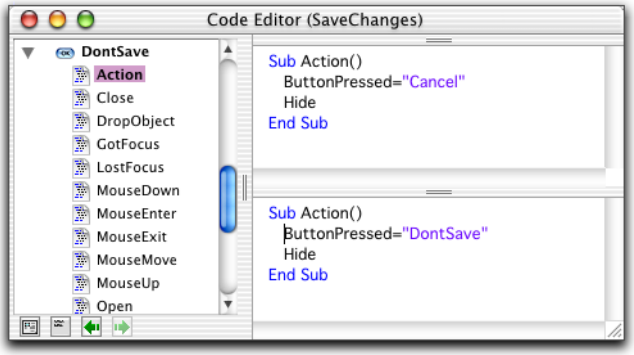
8 Select the text of the Cancel button's Action event and drag it into the Action event for DontSave.

9 Change the text of the code to the following:

```
ButtonPressed="DontSave"
Hide
```

The Code Editor should now look as shown in Figure 40:

Figure 40. The Code Editor after adding the DontSave button's Event handler.



When working with two or more Code Editor panes, place the insertion point in a pane and then click an Event in the browser area to assign that Event's code to the pane.

10 Create a third pane by dragging the divider from the top to the middle of the top pane, placing the new pane in the middle.

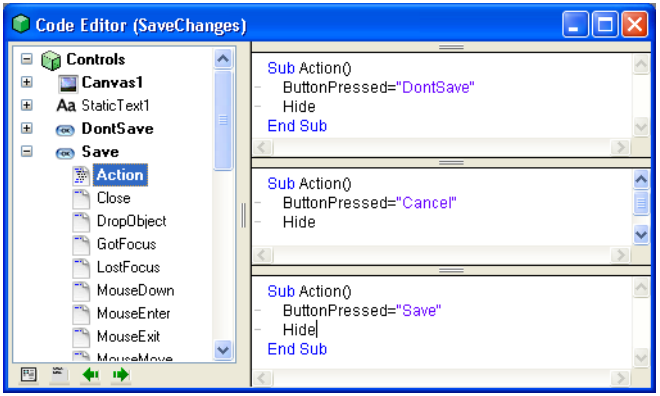
11 Click in the middle pane and then expand the Save button's Events and click its Action event handler.

12 Drag a copy of the code to the Save button's event handler and modify it so that it reads as follows:

```
ButtonPressed="Save"
Hide
```

With all three Events on-screen, the Code Editor looks as shown in Figure 41.

Figure 41. The Code Editor after adding Action event handlers for all three PushButtons.



13 Note that the Event handler that is highlighted in the Browser area is the one that contains the insertion point. In Figure 41 on page 54, the Action event for the DontSave button contains the insertion point.

14 Close the bottom two panes by dragging their dividers to the top.

Displaying the Save Changes Dialog Box

The final step is to add code that displays the SaveChanges dialog box when the user chooses the Quit menu item and there are unsaved changes to the contents of any open window. This is done in the CancelClose event handler in TextWindow.

The 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 it 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 and the application will not quit.

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 In the Code Editor for TextWindow, expand the Events item and highlight the CancelClose event.
Reminder: You are now back to the Code Editor for TextWindow, not SaveChanges.

2 Enter the following code:

```
If TextHasChanged then
 SaveChanges.ShowModal //display dialog & wait for input
 Select Case SaveChanges.ButtonPressed
 case "DontSave"
 case "Cancel"
  Return True //cancel the quit
 case "Save" //call SaveFile to save the document
  TextWindow(Window(1)).SaveFile Window(1).Title, False
 End Select
 SaveChanges.Close //close the dialog
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 SaveChanges.ShowModal statement runs the ShowModal method of the Window class (the Save Changes dialog box is an instance of the Window class and, therefore, inherits all its properties and methods). This method stops code execution at this statement until the user clicks one of the three buttons in the dialog.

The Select Case structure determines which button the user clicks. It looks at the value of the ButtonPressed property of the SaveChanges window. If the Don't Save button is 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, cancelling 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 Changes dialog box (since we're already in a Save Changes dialog box!).

The syntax:

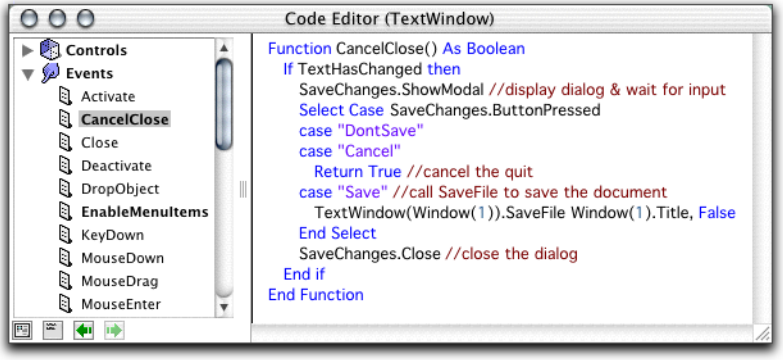
```
TextWindow(Window(1)).SaveFile
```

is a reference to the second window (Window zero is the dialog box). That is, the instance of TextWindow that was in front when the Save Changes dialog appeared.

"Window(1)" is the Window function, which returns a reference to the specified window. Since you can have more than one TextWindow open, REALbasic needs to know which instance of TextWindow you are working with at the moment.

3 Save your project as **TextEditor-ch5.rb**.
The Code Editor for CancelClose should now looks like Figure 42.

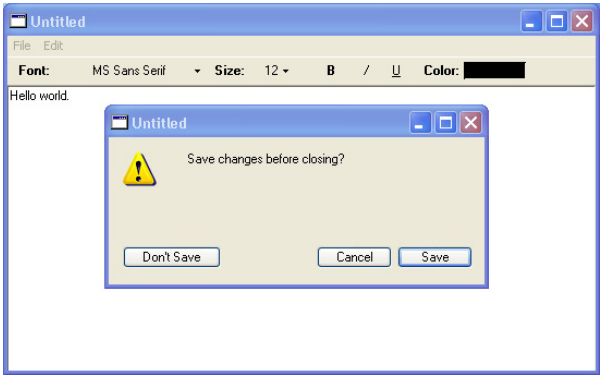
Figure 42. The CancelClose method.



4 Test the SaveChanges dialog box by switching to the Runtime environment, creating a new document, saving it to disk, modifying its contents, and then quitting to the Development environment.

The Save Changes dialog box should appear when you choose Quit. On Mac OS X, it appears as a Sheet window, but on other platforms it will appear as a 'regular' dialog box.

Figure 43. The SaveChanges dialog box.



If you click Don't Save, the application will go ahead and quit; if you click Save, a save-file dialog box will appear before the quit, and if you click Cancel, the Quit 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

Adding Drag and Drop to TextEditor

In this chapter you will add the ability to open text documents in TextEditor simply by dragging them from the desktop to an open window. You can drag several documents at once and the target window can either be blank or can contain an existing document.

You will learn how to:

- Configure an EditField control to accept dragged files,
- Handle multiple dragged items.

EditFields automatically support drag and drop within REALbasic, provided the MultiLine property is set; the steps in this chapter are necessary to add the ability to drag and drop external text documents.

Getting Started

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

Configuring TextField to Accept Dragged Documents

The first step is to tell TextField to accept dragged text files. REALbasic also allows you to drag and drop pictures and data types that you define. But before a control can accept dragged items, you must tell it which file type or types to accept. You use the file types defined in the File Types dialog box that was discussed in the section File Types" on page 32.

Dragging a text file to an EditField is supported only on Macintosh.

Since we want TextField to be ready to accept dragged files at any time, we do this in TextField's Open event handler. This event runs when the document window first opens.

To add the text file types, do this:

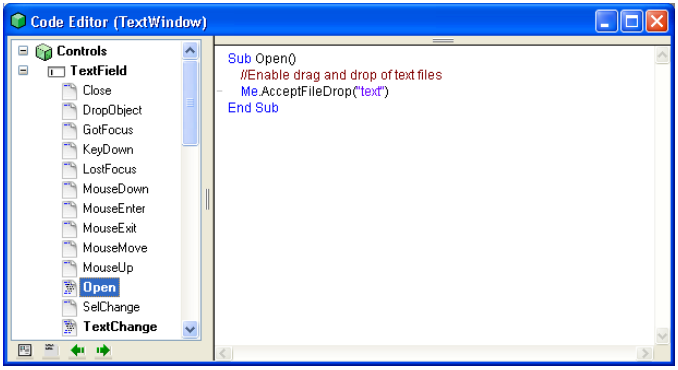
1 In TextWindow's Code Editor, expand the TextField item in the Controls category, and highlight the Open event handler.

2 Add the following code to this method:

```
//Enable drag and drop of text files
Me.AcceptFileDrop("text")
```

The AcceptFileDrop method lets you regulate the types of files that can be dropped on a control. The parameter, text, is the name of the File Type you defined in Chapter 4 for plain text files. The Code Editor should look like this:

Figure 44. Enabling drag and drop of text files.



The only remaining step is to tell TextField what to do when the user drags one or more files of this type. We do this in the control's DropObject event handler.

To process dragged files, do this:

1. Click TextField's DropObject event handler in the Code Editor browser.

   Notice that it takes one parameter, obj as DragItem. The properties of the DragItem class let you determine what types of data have been dragged to the object. If acceptable data types are dragged, you use other properties to extract the data.

2. Add the following code to the DropObject event handler:

```
Dim textStream as TextInputStream
If obj.FolderItemAvailable then
  Do
    textStream=obj.FolderItem.OpenasTextFile
    me.text=me.text+TextStream.ReadAll+EndOfLine
  loop until not obj.NextItem
End if
```

3. Save the project as **TextEditor-ch6.rb.**

   The FolderItemAvailable property of the DragItem object is True if one or more FolderItems (i.e., documents) have been dragged.

   If a FolderItem is available, the code uses the OpenAsTextFile method of the FolderItem class to open the text file. This method returns an object of type TextInputStream, which will contain the contents of the text file. The ReadAll method is then used to add the contents of the FolderItem to the Text property of TextField. The EndOfLine function adds a Return character to the end of the text.

   Since the user can drag more than one file at a time, the Do…Loop is used to continue this process until no more acceptable FolderItems are available. The NextItem method assigns the next eligible item's values to the properties of the DragItem object and returns False when no more eligible items remain.

Testing the Application

Now that the drag and drop capability has been added, you can test the application.

1. Choose Debug ▶ Run (⌘-R or Ctrl+R) and experiment with different text files and text clippings.

   If you are using a Macintosh, you can experiment with dragging files to the TextEditor. You'll find that TextEditor will reject dragged items of the wrong file type.

   You can also select text in the TextEditor and drag it to the desktop to create a text clipping file.

2. When you are finished testing, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

Review

In this chapter you learned how to add drag and drop capabilities to TextEditor.

To Learn More About:	*Go to:*
REALbasic drag and drop	*REALbasic User's Guide: Chapter 5.*
REALbasic commands and language	*REALbasic Language Reference*

CHAPTER 7

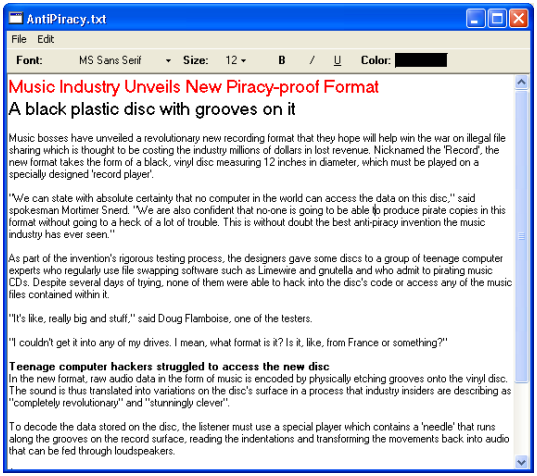
Working with Styled Text

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 45.

Figure 45. The finished TextEditor application window.



As you can see, the controls appear inside the window. The Font and Font Size controls are a 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-ch6.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch6.rb" that is in the Tutorial Files folder on the REALbasic CD.

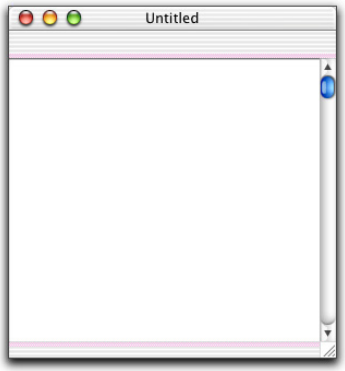
Configuring TextField for Styled Text

Before implementing the Style and Size menus, you need to tell REALbasic to allow the TextField to accept multiple font styles and sizes. You do this by setting the Styled property of the TextField. We also 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. Double-click the TextWindow item in the Project Window or right-click on it and choose Edit Window.
   The window opens in a Window Editor. Its properties are now displayed in the Properties Window.
2. Click on the TextField in the window and use the Properties Window to set the Styled property. The Styled property is listed in the Appearance group.
   Unless you do this, your font, font size, and style properties will affect all the text in TextField; that is, you won't be able to apply different styles to different text selections.
3. Set the Top property of TextField to **25**.
4. Reduce the value of the Height property by 26.
   Your TextWindow should now look like Figure 46.

Figure 46. The TextWindow after adding space for the Font and Style controls.



5 Save the project as **TextEditor-ch7.rb**.

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 Palette to the top area of TextWindow, above TextField.
It will serve as the label for the Size menu.

2 Using the Properties Window for the StaticText control, set its properties as shown in the following table.

Table 4: 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 Macintosh and Windows versions of the application (and sometimes between Mac OS X and Mac OS 'classic'), this option is provided so that you can easily get an attractive font size without adding code of your own.

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 pictures.
4. Using the Properties Window for the BevelButton, set its properties as shown in Table 5.

Table 5: Properties of the Size Menu Control.

Property	Value
Name	SizeMenu
Left	221
Top	4
Width	43
Height	16
Caption	(leave blank)
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. In TextWindow's Code Editor, open the SizeMenu item in the Controls category and highlight the Open event handler.
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)
```

The first seven lines call the AddRow method of the BevelButton class that adds an item to its popup 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 Str function converts the (integer) font size of the selected text in TextField 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 Enter the following code into SizeMenu's Action event handler:

```
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. 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.

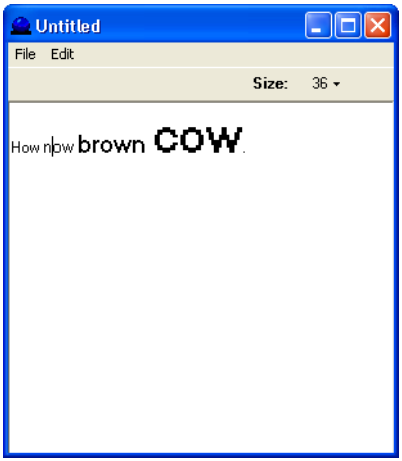
Trying out the Size Menu

Save your project and try it out (Debug ▶ Run). 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 47 on page 68.

Figure 47. The Size Menu Isn't Being Updated.



In Figure 47, 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. It 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 item and highlight the SelChange event handler.
2. Enter the following code into this event handler:

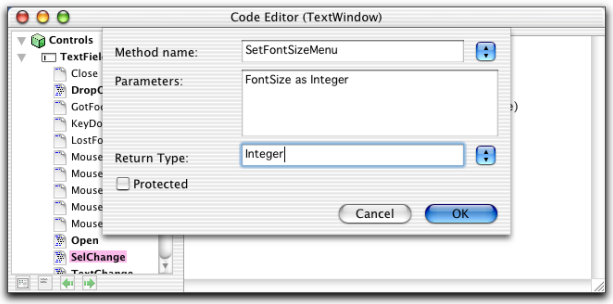
```
//update Font Size menu
If Str(Me.SelTextSize) <> SizeMenu.Caption then
  SizeMenu.Caption=Str(Me.SelTextSize)
  SizeMenu.MenuValue=SetFontSizeMenu(Me.SelTextSize)
End if
```

This If statement 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 in front, chose Edit ▶ New Method. The New Method dialog box appears.
4. Enter the name **SetFontSizeMenu**, parameter **FontSize as Integer**, and Return type of **Integer**.

The dialog box should now look like Figure 48.

Figure 48. SetFontSizeMenu's parameters.



When you click OK, the Code Editor for TextWindow opens the SetFontSizeMenu method.

5 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 last line returns the contents of the variable s.

6 Try out the application (Debug ▶ Run).
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 45 on page 63 that allow the user to apply the Bold, Italic, and Underline

styles. REALbasic also supports the Outline, Shadowed, Condensed, and Extended styles (on Macintosh "classic" only). In the tutorial, we will implement only the standard three style variations available on all platforms.

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. 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.
2. Drag a Bevelbutton control  from the Controls Palette to TextWindow's header area, just to the right of the Size menu.
3. Using its Properties Window, set its properties as follows:

Table 6: 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 (⌘-D or Ctrl+D), move the new buttons to the right of the Bold buttons—to the approximate positions of the Italic and Underline

buttons in Figure 45 on page 63 and use the Properties Window to set their properties as follows:

Table 7: 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
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. Do 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:

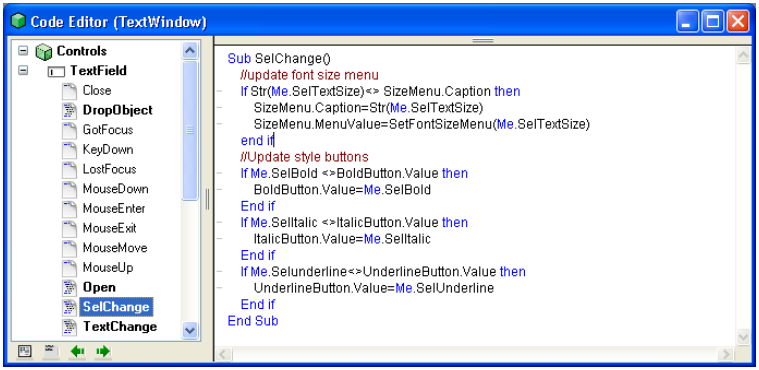
1. In the Code Editor for TextWindow, expand the Controls item and then expand the TextField item.
2. Click on SelChange and add the following code below the existing code:

```
//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 49.

Figure 49. 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.

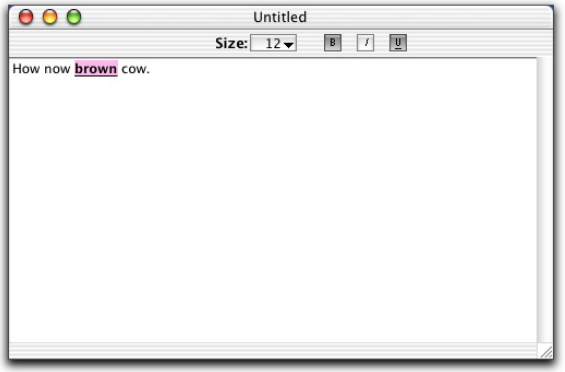
To use the styled text editor, do this:

1. Choose Debug ▶ Run and enter some text in the text editor.
2. 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 50. Bold and Underline styles applied to text.



3 When you are finished testing, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

Implementing the Color Control

In this section, you will add a Color control that enables 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.

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 (⌘-D or Ctrl+D).
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. (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 Drag a Canvas control  from the Controls Palette to the right of this StaticText object (don't worry about the fact that its default size is way too big) and use the Properties Window to assign it the following properties:

Table 8: 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 the 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.

Note that the method for the Paint event is passed one parameter, g as Graphics. You use this parameter to gain access to the Graphics class's drawing tools.

2 Enter the following code into the Paint event:

```
g.ForeColor=RGB(0,0,0) //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 RGB function uses the Red-Green-Blue color model to assign a color to the ForeColor property. Its parameters are the amounts of red, green, and blue in the color. The ForeColor property itself doesn't draw anything.

The first line of code sets the 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. 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=rgb(255,255,255) //default color
b=SelectColor(c,"Select a Text Color")
If b then
  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 routine 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 online or printed Language Reference for more information about ByRef.

Updating the Color Control

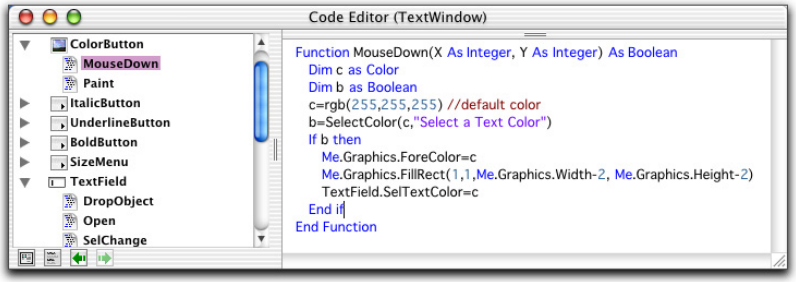
By now you must have guessed that we need to add some code to the SelChange event of TextField to update the color of the Canvas control when the user moves the insertion point into text that has a different color attribute.

To add the update code, do this:

1 In the Code Editor for TextWindow, click the second icon below the Browser area  , which is called "Hide Empty Methods."

The Browser changes to show only methods that have code. This makes it easier to navigate to methods that you need to modify.

Figure 51. The Browser with blank methods hidden.

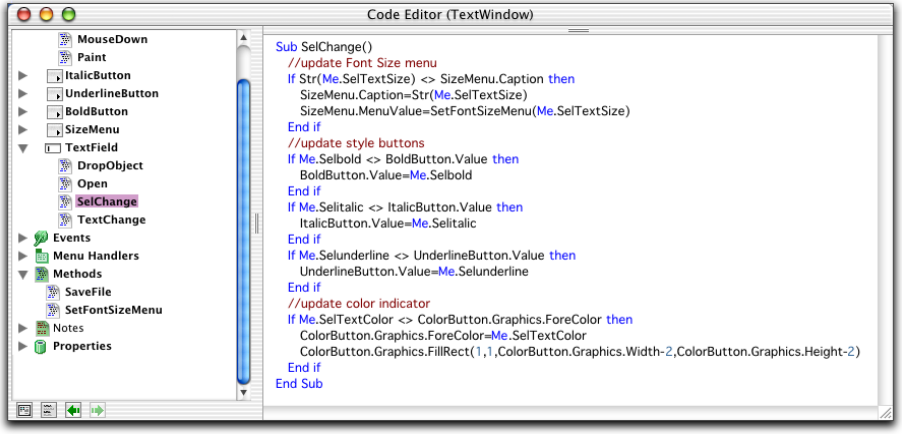


2 Click on SelChange in the TextField item and add the following code to the existing code:

```
//update color indicator
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). The SelChange event handler should now look like Figure 52.

Figure 52. 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.

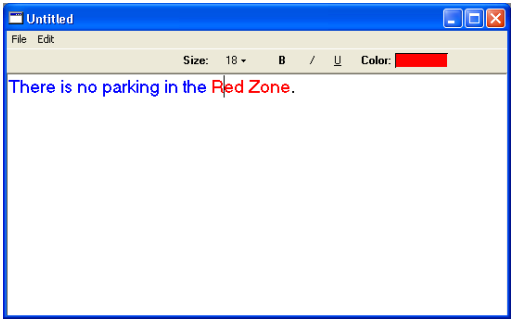
Testing the Color Control

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. Choose Debug ▶ Run and enter some text in the text editor.
2. Select some text and try changing the color.

Figure 53. Colors applied to text.



3. When you are finished testing, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

Review

In this chapter you learned how to work with StaticText, Separator, BevelButton, and Canvas controls and use 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: Chapters 14.*
REALbasic commands and language	*REALbasic Language Reference*

CHAPTER 8

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-ch7.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch7.rb" that is in the Tutorial Files folder on the REALbasic CD.

Implementing the Font Menu

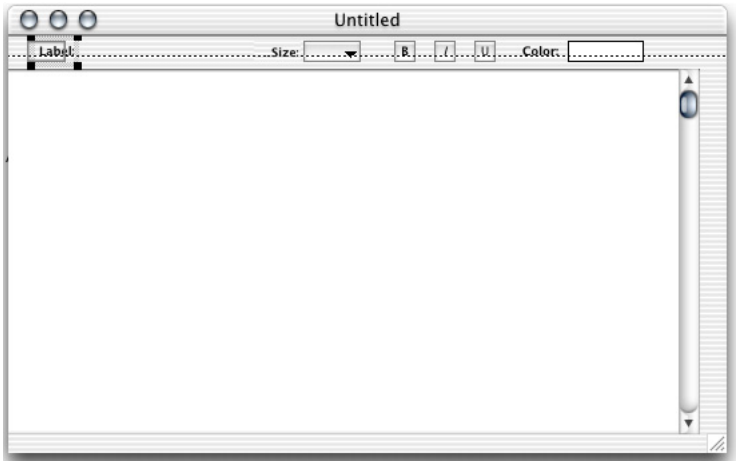
Implementing the Font menu involves the same basic steps for menu creation that you learned in the previous chapter. The key difference here is that you will add a method that loads the names of existing fonts into the menu items. This method runs when the window opens.

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

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

1 Drag a StaticText control Aa from the Controls Palette to the left side of the header area of TextWindow and align its baseline with the baselines of the other StaticText controls, as shown in Figure 54.

Figure 54. Aligning the StaticText control with the other labels.



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

Table 9: 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 Drag a BevelButton control  from the Controls Palette just to the right of the Font label and align it with the tops of the other controls.

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

Table 10: Properties of the Font Menu Control.

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

5 Save your project as **TextEditor-ch8.rb**.

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 MouseUp event for FontMenu selected.

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

```
Dim i, nFonts as Integer
nFonts=FontCount-1
For i=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 and the Font function returns the name of the i^{th} font. 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.

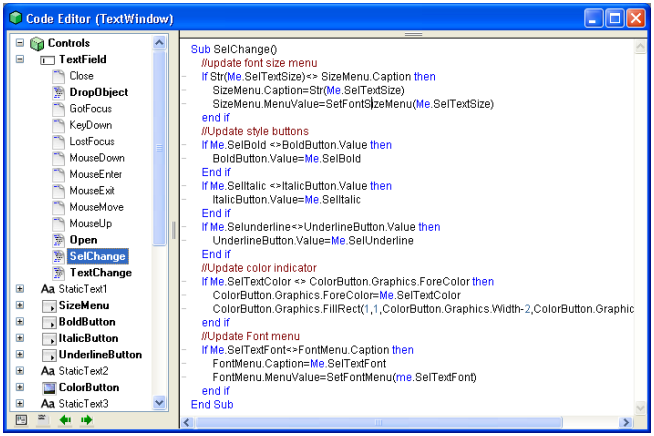
To update the Font menu, do this:

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 55 on page 83:

Figure 55. 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.

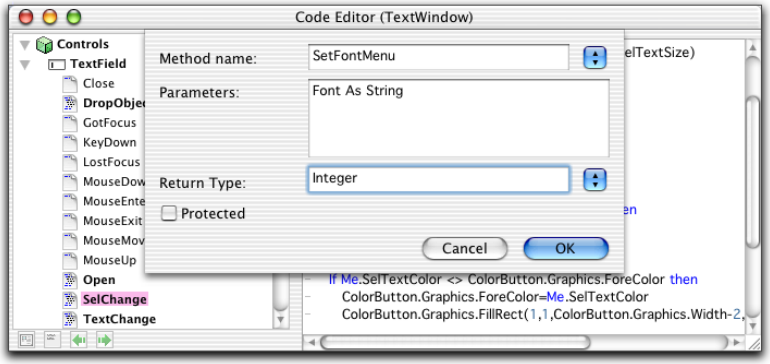
To add the SetFontMenu method, do this:

1 With the Code Editor for TextWindow in front, chose Edit ▶ New Method. The New Method dialog box appears.

2 Enter the name **SetFontMenu**, parameter **Font as String**, and Return type of **Integer**.

The dialog box should now look like Figure 56 on page 83.

Figure 56. The SetFontMenu dialog box.



3 Click OK.

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.

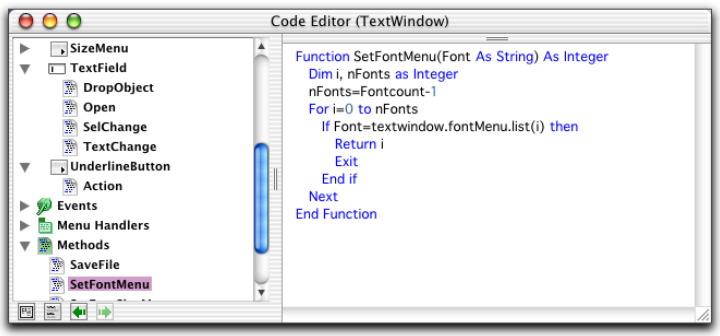
4 Add the following code to this method:

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

5 Save the project.

The new method should look like Figure 57.

Figure 57. The SetFontMenu method.



In the SelChange event handler, the name of the current font is passed to this function in the parameter Font. 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.

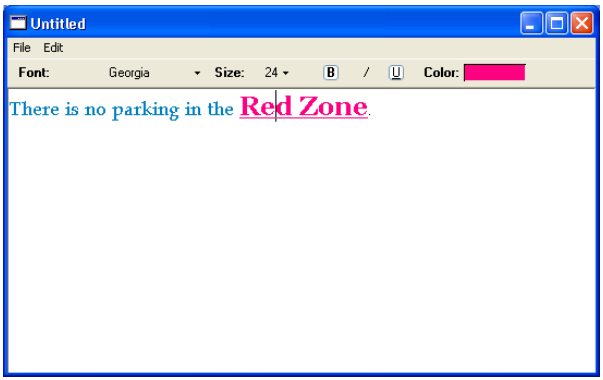
Once the SelChange event handler has this sequential number, it can assign it to the Font menu's MenuValue property. This resets the checkmark in the Font menu.

Testing the Application

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

1 Choose Debug ▶ Run (⌘-R or Ctrl+R) and experiment with different fonts, font sizes, styles, and colors.

Figure 58. Colors, Fonts, and Styles on Windows.



2 When you are finished testing, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

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 9

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-ch8.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch8.rb" 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. Double-click the MenuBar1 item in the Project Window, click on the File menu, and select the blank menu item.
2. In the Properties Window, enter **Page Setup...** in the Text area and press Return.
   REALbasic automatically assigns the Name "FilePageSetup". If you used an ellipsis (…) instead of three dots, remove the ellipsis from the name.
3. Deselect the AutoEnable property.
4. Next, select the blank menu item in the File menu and enter **Print...** in the Text area and press Return.
5. Assign **P** to the CommandKey property.
6. Deselect the AutoEnable property.
7. Position the two new menu items between the Save As and Quit menus (Exit menu on Windows).
8. Select the empty menu item at the bottom of the File menu and enter a dash "**-**" as its Text property.
   This creates a separator between groups of menu items.
9. Drag the separator between the Save As and Page Setup menu items.
10. Create another separator and drag it between the Print and Quit (or Exit) menu items.
11. Close the Menu Editor.
12. Save the project as **TextEditor-ch9.rb**.

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, so the AutoEnable property should not be used for these menu items.

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

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

2. 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.

To add the property, do this:

1. With TextWindow's Code Editor in front, choose Edit ▶ New Property and enter **PageSetup as String** in the Property definition dialog box.
2. Click OK to close the window.

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

To add the Page Setup menu handler, do this:

1. Choose Edit ▶ New Menu Handler and choose FilePageSetup from the pop-up menu.
2. Enter the following code in the Page Setup menu handler:

```
Dim ps as PrinterSetup
ps=New PrinterSetup
If PageSetup <> "" then
  ps.SetupString=PageSetup
End if
If ps.PageSetupDialog then
  PageSetup=ps.setupstring
End if
```

The PrinterSetup property, SetupString, contains the page setup selections that the user makes in the Page Setup dialog box. 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. Choose Edit ▶ New Menu Handler 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 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 a 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. Choose Debug ▶ Run and enter some text in the text 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, choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit on Windows to return to the Development environment.

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 10

Communicating Between Windows

In this chapter you will work with object communication features 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 searches from the beginning 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-ch9.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch9.rb" 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:

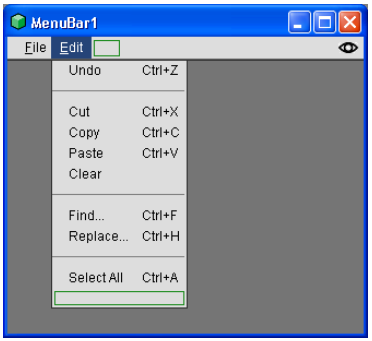
1. Double-click the MenuBar1 item in the Project Window.
2. Select the Edit menu in the menu bar.
3. Select the empty menu item at the bottom of the Edit menu and enter **Find...** as its Text property.

   The Name property automatically is filled in as "EditFind" in the Properties Window.
4. Type an **F** for the CommandKey property.
5. Deselect the AutoEnable property
6. Select the empty menu item at the bottom of the Edit menu and enter **Replace...** as its Text property.
7. Type an **H** for the CommandKey property.
8. Deselect the AutoEnable property.
9. Select the empty menu item at the bottom of the Edit menu and enter a dash "**-**" as its Text property.

   This creates a separator between groups of menu items.
10. Drag the separator between the Clear and Select All menu items.
11. Move the Select All menu item to the bottom and add another separator and place it between the Replace and Select All items.

    The Edit menu should look like Figure 59.

Figure 59. The completed Edit menu.



12 Close the Menu Editor and save your project as **TextEditor-ch10.rb**.

Enabling the Find and Replace Menu Items

The Find and Replace menus should be enabled only when a document window is open, so you enable it in TextWindow's Code Editor. Since the AutoEnable property is disabled, these menu items will be disabled when no document windows are open.

To enable the menu item, do this:

1 Open the Code Editor for TextWindow by right-clicking on the TextWindow item in the Project Window and choosing Edit Code or pressing Option-Tab.

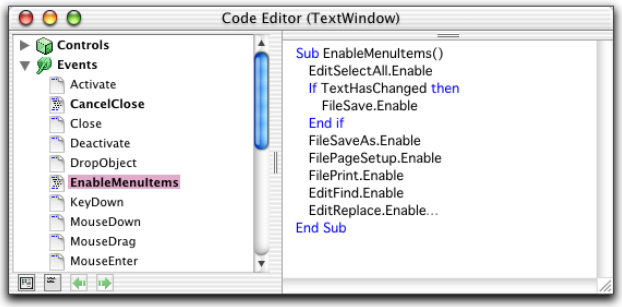
2 Highlight the EnableMenuItems event handler in the Events item.

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 60 on page 95.

Figure 60. Updated Code Editor for EnableMenuItems.

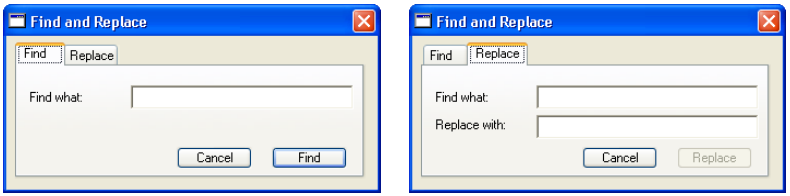


Creating the Find and Replace Dialog Box

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 function after opening the dialog box. When you are finished, the dialog box will look like Figure 61:

Figure 61. 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. With the Project Window as the frontmost window, choose File ▶ New Window.
   REALbasic adds a window to the project and names it Window1.
2. Use Window1's Properties Window to change its name to **FindWindow**.
3. Change its Title property to **Find and Replace**.
4. Change the window's Width to **340** and Height to **140**.
5. Deselect the GrowIcon and ZoomIcon properties.
   These properties are 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 Palette 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**.
   The Edit Tab dialog box appears. Use this dialog to enter the label for each tab.
3. Click the tab with the three dots to display the Tab Panel editor.
4. Click on Tab 0 and click the Edit button.
5. Change its name to **Find**, click OK, and then highlight Tab 1 in the Tab Panel editor and click the Edit button.
6. Change the name of the second tab to **Replace** and click OK.
7. 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. Click the Find tab and drag a StaticText control Aa 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 Palette to the right of the StaticText control and assign it the properties shown in Table 11 using the Properties Window.

Table 11: Properties of the EditField Control.

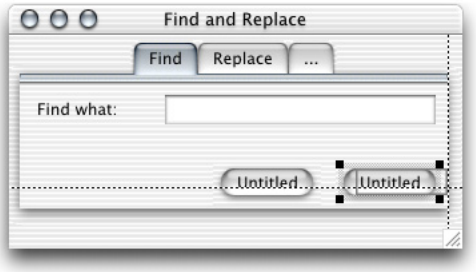
Property	Value
Name	FindText
Left	112
Top	43
Width	204
Height	22

4 Drag a PushButton from the Controls Palette to the bottom area of the TabPanel control, as shown in Figure 62, below.

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

6 Drag the two pushbuttons to their approximate locations, letting REALbasic align their baselines, as shown in Figure 62.

Figure 62. Aligning the Find button.



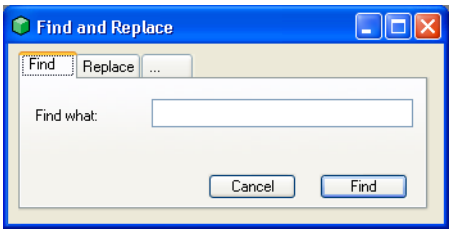
7 Select each pushbutton and make the property assignments shown in Table 12.

Table 12: 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 63. The Find panel of the Find and Replace dialog box.



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 Aa 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 palette to the right of the StaticText tool.
5. Click on the EditField control and, using the Properties Window, assign it the properties shown in Table 13.

Table 13: Properties of the EditField Control.

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

6. Duplicate the StaticText and Editfield Tools (⌘-D or Ctrl+D) 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 14: Properties of the EditField Control.

Property	Value
Name	ReplaceText
Left	112

Table 14: Properties of the EditField Control.

Property	Value
Top	70
Width	204
Height	22

9 Next, drag a PushButton control  from the Controls Palette to the area occupied by the Replace button in Figure 61 on page 96.

10 Select the Pushbutton control and choose Edit ▶ Duplicate (⌘-D) to create the Cancel pushbutton.

11 Drag the Cancel pushbutton into place, letting REALbasic align it with the baseline of the Replace button using the alignment line.

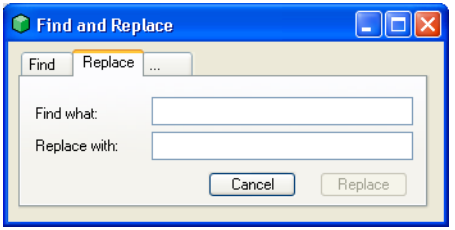
12 Select each pushbutton and make the property assignments shown in Table 15.

Table 15: 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 64. 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.

1 Click on FindWindow in the Project Window and press Option-Tab or Right-click and select Edit Code to open its Code Editor.

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. 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. Add 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.

1 Select the TextWindow item in the Project Window and press Option-Tab to open its Code Editor or right-click and select Edit Code.

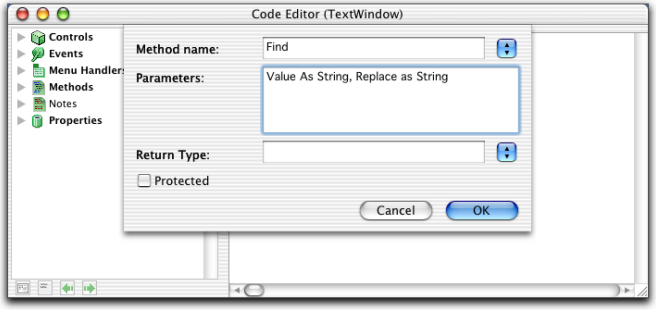
2 Choose Edit ▶ New Method to create the Find method.

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

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

The dialog should look like that shown in Figure 65:

Figure 65. The method declaration dialog box for Find.



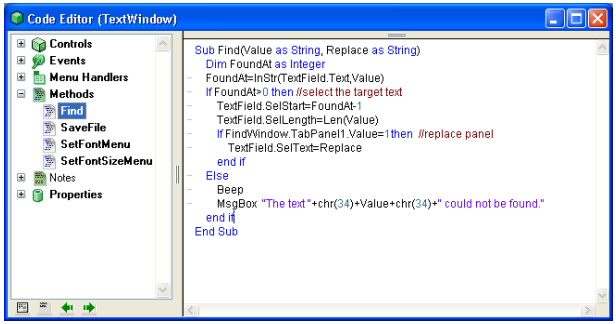
4 Click OK to display the Code Editor for the Find method.

5 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 66.

Figure 66. The Find method in the Code Editor.



This method locates the string to be searched for (the parameter *Value*) using the InStr function. InStr takes two parameters, the text to search and the text to search for. 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 page of the dialog box.

1 With TextWindow's Code Editor as the frontmost window, choose Edit ▶ New Menu Handler.

2 Choose EditFind from the Menu Handler pop-up menu and enter the following code into the menu handler method.

```
FindWindow.Show
```

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

3 Choose Edit ▶ New Menu Handler again and choose EditReplace.

4 Add the following code to the Replace menu item's menu handler:

```
FindWindow.Show
FindWindow.TabPanel1.Value=1
```

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).

Testing the Find and Replace Functions

You are now ready to test the new features. Choose Debug ▶ Run, 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.

1 Return to the Development environment and expand the TabPanel1 item in FindWindow's Code Editor.

2 Highlight the Change event handler.

This is the event that runs when the user clicks on a tab.

3 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 TabPanel1.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.

4 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 11

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-ch10.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch10.rb" 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

A portion of the REALbasic Debugger is active whenever you enter code in your application. 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 choose Debug ▶ Run, 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
```

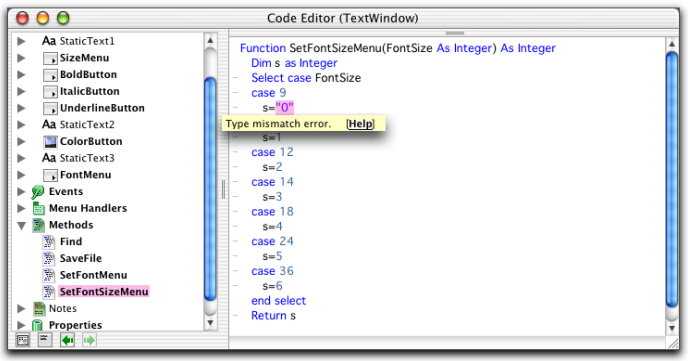
to

```
s="0"
```

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

4. Now, choose Debug ▶ Run (⌘-R or Ctrl+R).
A "Type Mismatch" error message appears and the offending line of code is highlighted. Your Code Editor should look like that shown in Figure 67.

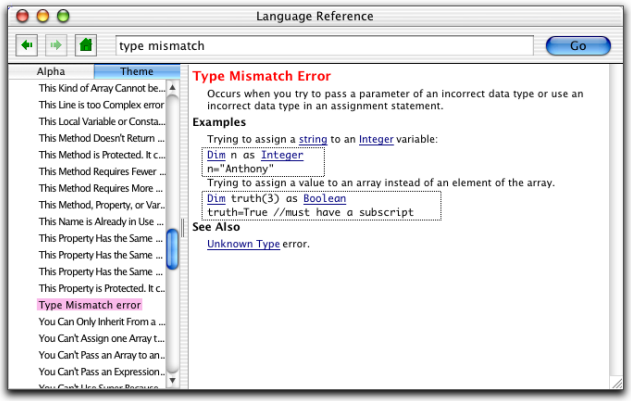
Figure 67. Syntax error message in the Code Editor.



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.

The syntax error message has a Help button to the right of the error. When you click Help, it opens the Online Reference to the entry for the error.

Figure 68. The "Type Mismatch Error" error in the Online Reference.



5 To fix the error, simply 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.

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.

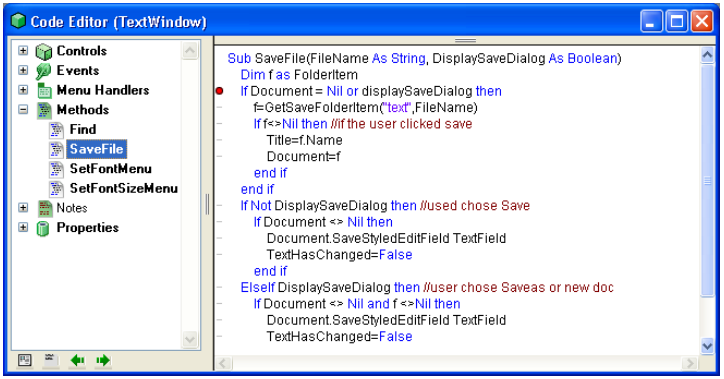
To see how the REALbasic Debugger works, do this:

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

2 In the Browser, expand the Methods item and select the SaveFile method.
The SaveFile method is displayed. 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 icon appears in the margin of the Code Editor, signalling a breakpoint. The Code Editor should look as shown in Figure 69 on page 108.

Figure 69. Setting a Breakpoint in the Code Editor.

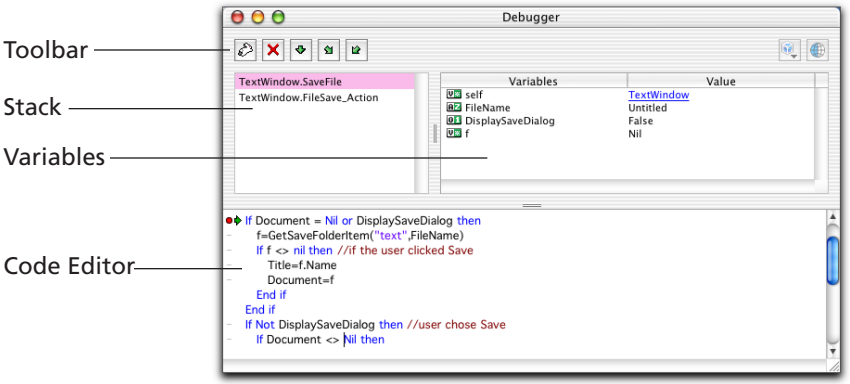


This breakpoint will cause REALbasic to pause when you try to save a document in the Runtime Environment. When you try to save a new document, the Debugger will appear instead of the save-file dialog box.

4 Choose Debug ▶ Run (⌘-R or Ctrl+R) to start your application in the Runtime Environment.

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

Figure 70. The Debugger stopped at the breakpoint.

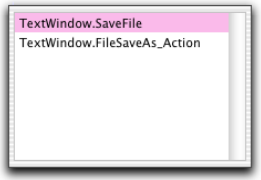


The Debugger window is divided into three sections. The Code Editor section shows the method that is currently executing. Execution has stopped right at the breakpoint line. The red dot indicates the breakpoint and the green arrow shows the line that is executing.

The Stack Pane 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.

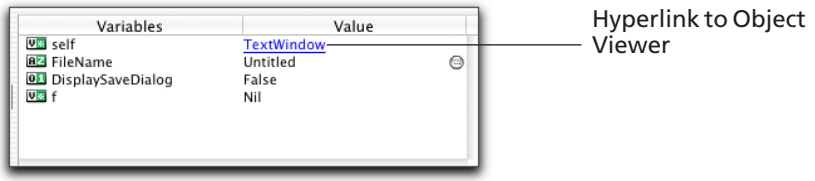
You can check the Stack Pane to verify that methods are actually called when you expect them to be called.

Figure 71. The Stack pane shows the order in which methods were called.



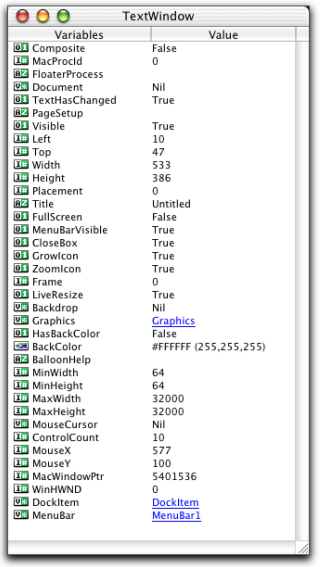
Variables Pane contains a list of all the variables local to the method containing the breakpoint, along with their current values (if any). The data type of each variable is indicated by a small icon in the left column.

Figure 72. The Variables pane.



Any objects (rather than variables) that are defined in the method are shown as hyperlinks rather than values. If you click a link, a window called the *Object Viewer* opens, containing the list of current values for the object's properties.

Figure 73. The Object Viewer for TextWindow.



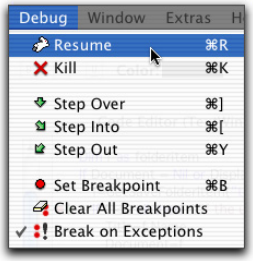
Each property of the object is shown in the Object Viewer in the same format as the Variables Pane. The data type of the property is indicated via an icon in the first column and the property's value is shown in the second column. If a value is also an object, then a hyperlink is shown instead of a value. For example, the Graphics property is shown as a hyperlink to its Object Viewer. You can open as many Object Viewer windows as you wish.

The important feature of the Object Viewer and Variables pane is that it is interactive. As you execute code line by line in the Debugger, the Object Viewer 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:

Figure 74. The Debug menu while the Debugger window is active.



- **Resume** (normally **Run**): Continues execution from the breakpoint line without further interruption. This exits from the Debugging environment. It does not remove the breakpoints that you've set in your code.
- **Kill**: 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.

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.

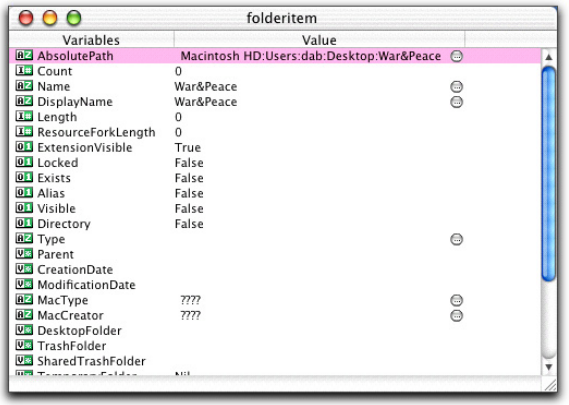
When you are in the Debugger, you can execute code line by line and monitor the contents of the Variables Window. You do this using the Step Into or Step Over buttons (or menu items).

6 Click the Step Over button  until the save-file dialog box appears.
Each time you select this menu item, the current line of code is executed and the green arrow shown in Figure 70 on page 108 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.

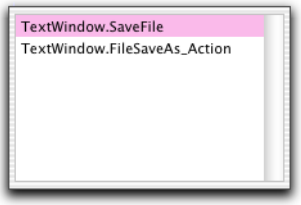
Click the FolderItem hyperlink to see the current value of f in the Object Viewer. The Object Viewer will show the absolute pathname to the document and the filename that you just gave it.

Figure 75. The Object Viewer for the FolderItem after saving the document.



8 In the Variables pane, click the TextWindow object to see the parent window's properties.
You can open several Object Viewers at the same time and keep them open as you step through your code.
When you execute the line "Title=f.Name", notice that the Title property in TextWindow's Object Viewer updates to show the document name that you entered.
The Stack panelists the current method and should look like that shown in Figure 76.

Figure 76. The Stack Pane.



This is as it should be, i.e., the SaveFile method was called when the Save or Save As menu handler was executed. In Figure 76, you can see that the user chose Save As rather than Save.

9 To resume execution of your application, click the Resume icon in the Debugger.

10 Choose REALbasic ▶ Quit on Mac OS X or File ▶ Exit (on Windows) to exit the Runtime 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 that errors in your code only manifest themselves 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.

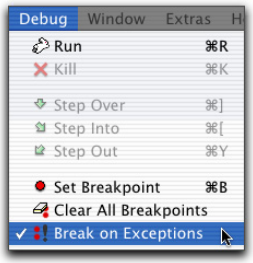
The existence of a potential runtime error does not prevent REALbasic from successfully compiling the application and the application may run without problems for a long while as long as the line of code containing the error is not actually executed. For example, if the line containing the error is in 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 Debug 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 Pull down the Debug menu and verify that Break on Exceptions is selected. It should have a check mark beside it, as shown here.

Figure 77. The Break on Exceptions option is selected.



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

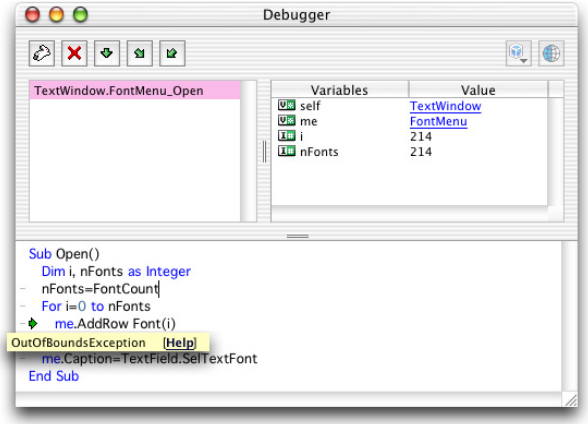
2 Highlight FontMenu's Open event.
You will see the code:

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

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

4 Choose Debug ▶ Run (⌘-R or Ctrl+R).
Instead of seeing a new document window, execution will stop and you will see the error shown in Figure 78.

Figure 78. 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.

When a runtime exception occurs in a standalone application, REALbasic, obviously, can't display the line of code that caused the error. 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.

5 Correct the deliberate error in the line:

```
nFonts=FontCount
```

by changing it back to:

```
nFonts=FontCount-1
```

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 12

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 Mac OS "classic", Mac OS X, and Windows applications.

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-ch11.rb"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch11.rb" 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 Mac OS or Windows application.

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

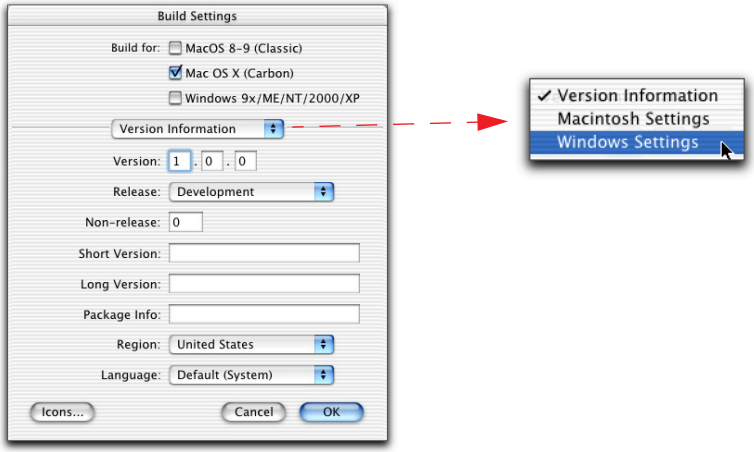
The File Menu has two menu commands that you use for building standalone applications: Build Settings and Build Application. The Build Settings command displays a dialog box in which you set parameters for the build process and the Build Application command actually builds the standalone application, using the current parameters that you've set in the Build Settings dialog.

When you use the Build Application command without specifying any custom settings, REALbasic builds the application for the platform you are currently running and gives the a default name, "My Application" or "My Application (Mac OS X)", for Mac OS X builds.

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

1 Choose File ▶ Build Settings….
The dialog box shown in Figure 79 appears.

Figure 79. The Build Settings dialog box.



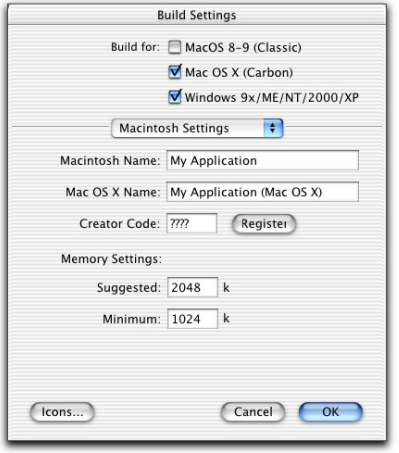
In the top area of the dialog box, you select the target platform (or platforms) for the build. Your choices are the Mac OS "classic" environment (pre-Mac OS X), the Mac OS X environment or "classic" with the CarbonLib extension installed, and/or any flavor of Windows from Win95 to NT/2000/XP. You can build as many as three targets simultaneously.

The Version Information screen enables you to enter version information for the build. This information is saved in the application for all platforms. See the *User's Guide* for a description of where each setting appears in the standalone application.

2 The platform you are currently running is preselected. If you have another operating system available, check it as well.

3 If you are building for a Macintosh OS, choose Macintosh Settings from the pop-up menu.
The following screen appears.

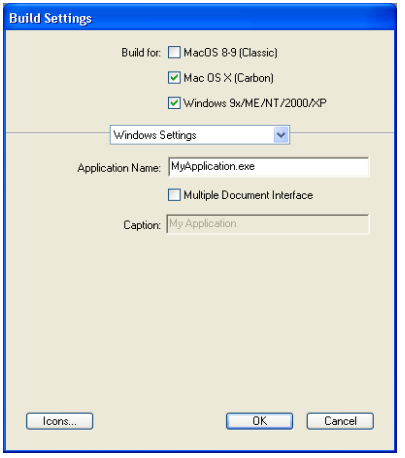
Figure 80. The Macintosh Settings panel.



4 Enter the name **TextEditor** in either the Macintosh Name (for Macintosh Classic builds) and/or Mac OS X name (for Mac OS X builds).

5 Choose Windows Settings from the pop-up menu.

Figure 81. The Windows Settings panel.



6 If you haven't already replaced the default application name for the Windows build, enter **TextEditor.exe** in the Application Name area.

7 Click OK to save your settings.

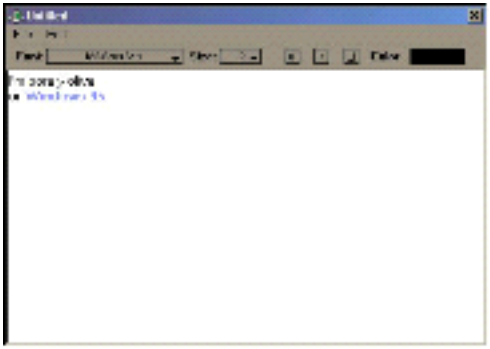
8 Choose File ▶ Build Application.
REALbasic builds both Mac OS and Windows standalone applications, places them in the same folder as your project, and brings that Finder window to the front.

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

Figure 82. The Application icon for the standalone application.

If you are running a Windows computer, try out the standalone application under Windows as well.

Figure 83. TextEditor running on Windows.



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