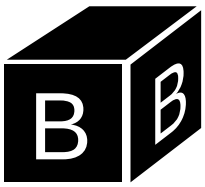


Chapter 5

# *Dialog Boxes*

In a graphical user interface, the user typically decides what action to execute by selecting an option from a list of menu items at the top of the screen. If the action to be performed requires a data value to be processed, the user is prompted to enter the data value by entering it in a dialog box. Menu items and dialog boxes are the predominant GUI methods for interacting with human users. This chapter shows how to construct dialog boxes that manage the input, execution, and output of the program. The dialog boxes in this chapter are activated by the commander button. Later chapters will describe how to install menu items, which, when selected by the user, can initiate procedures or activate dialog boxes.

## Numeric input from a dialog box

This section shows how to link elements in a Component Pascal module with a dialog box. The dialog box allows the user to input a value in a rectangular input area called a field. A button will be provided in the dialog box, which will cause an exported procedure to be executed when pressed.

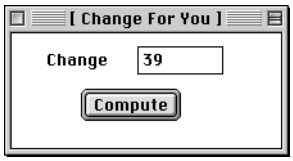
The process of programming a dialog box requires the following steps:

*The design process for programming with dialog*

- Decide which elements belong in the dialog box.
- For each input/output element, determine the type of the corresponding variable.
- For each button, determine what processing must be performed.
- Write a module with each of the variables and procedures exported. Compile the module.
- Select Controls→New Form… and link the module to the dialog box.
- Fine tune the layout of the elements of the form.
- Write a Docu document with a commander to activate and test the dialog box.

These steps will be illustrated by an example similar to the module in Figure 4.7 in the previous chapter. Procedure MakeChange in that program assigns the value 39 to the variable cents, then computes the corresponding number of dimes, nickels, and pennies, which it displays on the Log. Such a program is not too useful because the same output is produced every time it executes. To get the number of coins for a different value of cents you would have to modify the source program by changing 39 to whatever value you want, recompile the program, then execute it again.

It would be much better if you could present the user with a dialog box in which she could enter the number of cents in change. Figure 5.1 shows the desired dialog box for the user. The rectangular box, called a text field, is the input area. The user enters the number of cents in change into the box then clicks the button labeled Compute. Clicking the button with the mouse causes the program to print the number of dimes, nickels, and pennies onto the Log.




**(a)** MacOS.


**(b)** MSWindows.

**Figure 5.1**
The dialog box for inputting an integer value shown in mask mode.

In this problem, we want three elements in the dialog box—the label Change that identifies the text field, the text field for the user input, and the button that causes the computation to commence. The next step in the design process is to write a program that contains variables and procedures that correspond to the elements of the desired dialog box. In the dialog box of Figure 5.1, the label Change and the input text field correspond to a single element of type INTEGER. It is an integer element because the user should not enter a value with a decimal point. The button labeled Compute corresponds to a procedure.

The module in Figure 5.2 illustrates the next step in the design process. It is a program containing an exported variable named change that corresponds to the label Change and the input text field, and an exported procedure named MakeChange that corresponds to the button labeled Compute. Later in the design process, we will link the variable change to the text field of the dialog box. When the user enters an integer in the text field of the dialog box, the BlackBox framework automatically assigns that integer value to change. We will also link the procedure MakeChange to the button labeled Compute in Figure 5.1. When the user clicks the button with the mouse, MakeChange will execute.

Variable change is located within module Pbox05A but outside procedure MakeChange, unlike variable cents, which is located within procedure MakeChange. Variables that are located within a module and not within a procedure are called *global* variables, in contrast to those that are within a procedure, which are called *local* variables. Variable cents is local to procedure MakeChange, while variable change is global.

*Global and local variables*

In Component Pascal, variables that correspond to elements of a dialog box must be global. An attempt to export cents will cause the compiler to protest with the error message, Illegally marked identifier. The reason for the restriction on the exporting of variables is that local variables exist only during execution of their procedures, while global variables exist as long as their modules are *loaded* into main memory. When the dialog box of Figure 5.1 is activated, the framework loads the module linked to it into main memory. When the user enters a value, and before she clicks the Compute button, change gets the value she enters. At this point in time, variable cents does not even exist, because procedure MakeChange is not executing.

*Loading modules into main memory*

It is only when the user clicks the Compute button triggering the execution of MakeChange that variables cents, dimes, nickels, and pennies come into existence.

```
MODULE Pbox05A;
   IMPORT StdLog;

   VAR
      change*: INTEGER;

   PROCEDURE MakeChange*;
      VAR
         cents: INTEGER;
         dimes, nickels, pennies: INTEGER;
   BEGIN
      cents := change;
      dimes := cents DIV 10;
      cents := cents MOD 10;
      nickels := cents DIV 5;
      pennies := cents MOD 5;
      StdLog.String("You have "); StdLog.Int( change);
      StdLog.String(" cents in change."); StdLog.Ln;
      StdLog.String("Dimes: "); StdLog.Int(dimes); StdLog.Ln;
      StdLog.String("Nickels: "); StdLog.Int(nickels); StdLog.Ln;
      StdLog.String("Pennies: "); StdLog.Int(pennies); StdLog.Ln
   END MakeChange;

BEGIN
   change := 0
END Pbox05A.
```

**Figure 5.2**
A module for constructing a dialog box to input an integer value.

Module Pbox05A has an initialization part with an assignment statement

```
change := 0
```

This statement belongs to the module and is not part of a procedure. The question naturally arises, When does this statement execute? Until now, all executable statements belonged to procedures, and an exported procedure began execution in response to the click of a commander by the user. Such is not the case with the statements in the initialization part of a module. Instead, they are executed once when the module is loaded from disk into main memory. Figure 5.3 illustrates the loading process for this program.

Before a procedure can execute, the entire module in which it is contained must be placed in the main memory of the computer. Figure 5.3(a) shows the situation after the source program has been written and compiled. The compiler creates the object program, which it stores on disk in the Code folder, and the interface, which it stores on the disk in the Sym folder. When the user first activates the dialog box linked to a module, the framework loads the object program for the module into main memory. Figure 5.3(b) shows the loading into main memory that must take


**(a)** Before first execution.


**(b)** Activating dialog box triggers load.


**(c)** Subsequent executions do not require load.

**Figure 5.3**
Dynamic loading in the BlackBox framework.

place before the first execution of any procedure. Subsequent executions of the procedure do not require loading, as shown in Figure 5.3(c).

The statement in the initialization part of the module is executed once when the module is loaded and before any procedures in the module are executed. The statement

```
change := 0
```

assigns zero to change so it will have a default value when the dialog box is displayed for the first time


**(a)** MacOS.

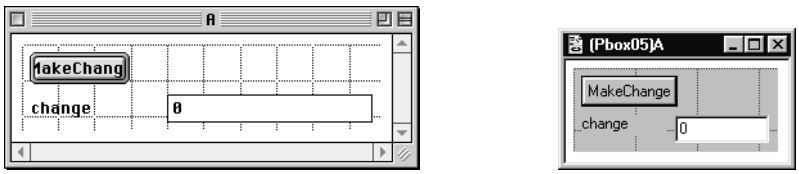

**(b)** MSWindows.

**Figure 5.4**
The result of selecting Controls→New Form… .

The fourth step in the design process is to select Controls→New Form…, which you should attempt only after you get your program compiled. A dialog box will appear as shown in Figure 5.4. It asks the programmer for a link to the desired dialog box. In this example, the programmer entered Pbox05A. If you have not com-

piled your program, the framework will have no object file with the information it needs to make the link to the exported record. After you enter the link, click the Create button to create the dialog box for the program.

Figure 5.5 shows the box as it appears in layout mode when it is created by selecting Controls→New Form… . Layout mode is for constructing a dialog box instead of using it. For example, if the dialog box is in mask mode, as it is in Figure 5.1, and you click the button then the procedure that the button is linked to will execute. But if the dialog box is in layout mode and you click the button then the button is simply selected and can be repositioned or resized by dragging it with the mouse. You can distinguish visually between the modes by the background grid that appears in layout mode but is absent in mask mode.



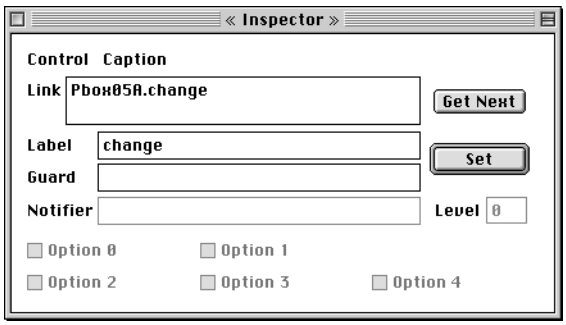

**(a)** MacOS. **(b)** MSWindows.

**Figure 5.5**
The dialog box shown in layout mode.

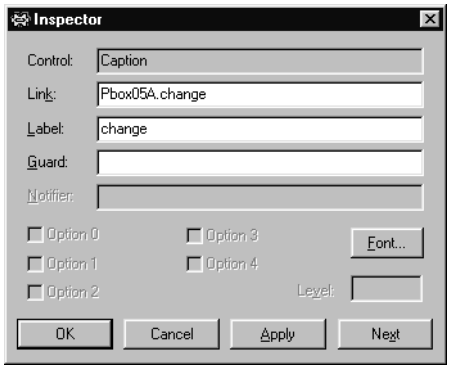
When you select Controls→New Form… and click the Create button the BlackBox framework inspects the interface of the module to which the new dialog box is linked. It inserts objects into the dialog box based on the types of the items exported by the module. The objects in a dialog box are called controls, because they allow the user to control the action of the computer by interacting with them via the mouse and keyboard. In this example, the framework inspected two exported items and as a result inserted three controls into the dialog box. The exported variable, change of type INTEGER, caused the framework to insert two controls—a caption control that appears as the word change in the dialog box of Figure 5.5, and a text field control that appears as the rectangular input area. The exported procedure MakeChange caused the framework to insert one control—a command button that appears at the top of the dialog box.

A new dialog box created by the framework is not usually suitable for immediate use. In this example, you would need to spruce up the appearance by capitalizing the c in change for the caption control and changing the wording in the command button control. In layout mode the attributes of a control are inspected and changed by selecting the control then choosing Edit→Part Info… (Macintosh) or Edit→Object Properties… (Windows) from the menu. Figure 5.6 shows the resulting dialog box, called the Inspector, when the caption control is selected.

The five primary attributes of a control as displayed in Figure 5.6 are (a) the control type, (b) how the control is linked, (c) the label for the control, (d) the guard for the control, and (e) the notifier for the control. The figure shows that (a) the type of the control is caption, (b) the control is linked to Pbox05A.change, (c) the control's label is change, (d) there is no guard, and (e) there is no notifier. The label of the caption is what appears to the user. To spruce up this control you would capitalize c




**(a)** MacOS.




**(b)** MSWindows.

**Figure 5.6**
The dialog box for editing a control's attributes.

in change in the label field and click the Set button to set the change. Because the purpose of a caption control is to simply display the label in the dialog box, the link field of a caption control serves no apparent purpose. You could eliminate the link field in this caption control with no adverse affect on your program.

A similar inspection of the attributes of the rectangular input area shows that it is of type text field, it is linked to Pbox05A.change, and it has no label, guard, or notifier. With this control, the link is crucial. When the user enters text into the rectangular input area and clicks the compute button the framework recognizes the link between the dialog box and the exported variable. It converts the text value entered by the user into an integer value, which it gives to the integer change. The label attribute does not apply to a text field control.

The push button, which is a command button control, uses both the link and the label attributes. The link specifies the procedure to be executed when the button is pressed and the label specifies the text to be displayed on the face of the button. To spruce up this control you would change the label to Compute.

Another way of sprucing up the box when it is in layout mode is to use the commands of the Layout menu, which permit you to align the controls in various ways. Normally, a dialog box cannot be resized when it is in mask mode and the user is entering data. In layout mode you can resize the window that contains the dialog box, but the size of the window in layout mode does not affect the size of the dialog box. To change the size of the dialog box you must be in layout mode and choose Edit→Select Document. The entire dialog box is then selected with handles for resizing with the mouse.

*Always save dialog forms in layout mode.*

When you have finished sprucing up your dialog select File→Save As… to save it. Dialog forms should always be saved in layout mode. You must save it in the Rsrc folder of your project. Rsrc stands for resource, and the folder contains various programming resources for a project. As an example, the dialog box of Figure 5.5 was saved with name DlgA in the Rsrc folder that was itself in the project folder named Pbox05. Dialog boxes are stored as standard BlackBox compound documents. If you ever want to modify the dialog box simply open the document as you would any other file and edit it. And when you save it, be sure that it is in layout mode.

The last step in the design process is to create documentation, which is stored in

the Docu folder as usual. The commander should be followed by a procedure from the StdCmds module instead of a procedure from your module. Figure 5.7 shows the documentation for this program.

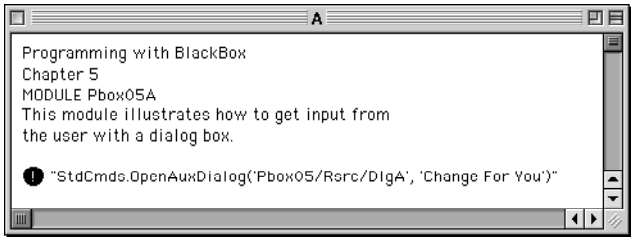



**Figure 5.7**
The documentation for the module in Listing 5.2

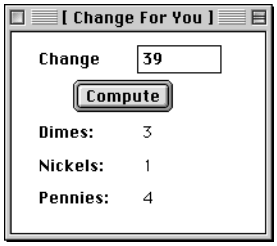
In this example the commander was followed with the string

"StdCmds.OpenAuxDialog('Pbox05/Rsrc/DlgA', 'Change For You')"

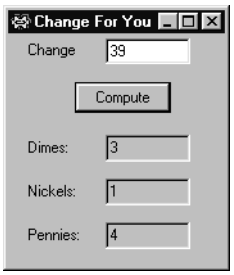
The StdCmds module contains the procedure OpenAuxDialog, whose purpose is to open a dialog box in mask mode. It requires two parameters, each of which is a string. The first parameter 'Pbox05/Rsrc/DlgA' specifies the file that is to be opened, and the second parameter 'Change For You' specifies the title to appear in the title bar of the dialog box. Figure 5.1 shows this title.

### Numeric output to a dialog box

The previous example showed how to input an integer value into a dialog box. The output was sent to the Log. The Log is for development and debugging. A commercial application would not use the Log for output. It is more common for the results of a computation to be shown in a dialog box or in some other document such as a spreadsheet document or a word processing document. Figure 5.8 shows a dialog box that displays the output from the computation.




**(a)** MacOS.




**(b)** MSWindows.

**Figure 5.8**
A dialog box that displays output as well as input.

The dialog box in Figure 5.8 is a bit more complex than the one in Figure 5.1. It has nine controls, eight of which are linked to four variables and one of which is linked to a procedure. Figure 5.9 shows the corresponding module.

```
MODULE Pbox05B;
   IMPORT Dialog;

   VAR
      d*: RECORD
         change*: INTEGER;
         dimes-, nickels-, pennies-: INTEGER
      END;

   PROCEDURE MakeChange*;
      VAR
         cents: INTEGER;
   BEGIN
      cents := d.change;
      d.dimes := cents DIV 10;
      cents := cents MOD 10;
      d.nickels := cents DIV 5;
      d.pennies := cents MOD 5;
      Dialog.Update(d)
   END MakeChange;

BEGIN
   d.change := 0;
   d.dimes := 0; d.nickels := 0; d.pennies := 0
END Pbox05B.
```

**Figure 5.9**
Sending output to a dialog box.

## Records

It would be possible for the module in Figure 5.9 to declare four global variables using the same technique as does the previous module. However, when more than one variable is to be linked to a dialog, the preferred programming style is to group them together in a record. A *record* is similar to an array in that both are collections of values. They are different in that the collection of values in an array must all have the same type. For example, variable message in Figure 4.13 is an array of characters. Every element of the array must be a character because that is a property of arrays.

*A record is a collection of values, which need not have the same type.*

The module in Figure 5.9 has a single exported record named d for dialog box. As is the case for all records, the type of this variable begins with the reserved word RECORD and ends with the word END. Between these words is a collection of four *fields*, each with a name and a type. The first field is named change, has type integer, and is exported with the familiar asterisk * export mark. The next three fields are named dimes, nickels, and pennies, also have type integer, but have a different kind of export mark. The hyphen - indicates *read-only export* and is for displaying values in a dialog box that the user is not allowed to change. You can see from the dialog

*The fields of a record*

*Read-only export*

box in Figure 5.8 that the user enters a value for Change and the computer calculates the proper number of dimes. Because the user does not enter a value for dimes, field dimes is exported read only. Note that export marks are required for both the variable d and its fields. To emphasize the difference between exporting with * and with -, exporting with * is called *read/write export*.

*Read/write export*

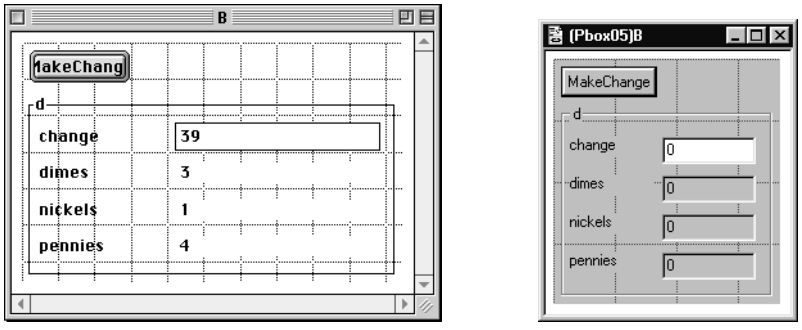
You refer to a field in a record in the same way that you refer to a procedure in a module. Namely, you write the name of the record followed by a period followed by the name of the field in the record. For example, you refer to field change in record d by writing d.change. Records are used to group values other than those for dialog boxes. When used for values in a dialog box, the record is known as an interactor, because of its role in the interaction between the user and the program.

*Referencing a field in a record*

*An interactor is a record linked to a dialog box.*

As before, the button labeled Compute is linked to MakeChange. In this program, procedure MakeChange computes the coins for the change and assigns those values to d.dimes, d.nickels, and d.pennies. However, simply changing the value of an interactor's field does not automatically transmit that change to the visual appearance of an open dialog box on the screen. Procedure Dialog.Update has the ability to transmit the change to the screen. It is only when Dialog.Update(d) executes that the changes to the values of d.dimes, d.nickels, and d.pennies are displayed in the dialog box.

Figure 5.10 shows the initial dialog box in layout mode when Controls→New Form… is selected and linked to Pbox05B. When you group variables together into a record the BlackBox forms creator inserts a control element called a group box that surrounds the elements. The default value for the label of the group box is the name of the record, which is d in the figure. Because a group box was not desired in the final dialog box it was simply deleted while in layout mode. If you want a group box in your dialog box you can use the Inspector to change the text that appears to the user. You can also resize the box, and move the controls around to include or exclude any controls you wish.




**(a)** MacOS. **(b)** MSWindows.

**Figure 5.10**
The default form produced by the forms creator for the module of Figure 5.9.

You should experiment with the Layout menu options, which provide many useful tools for creating precisely positioned elements. For example, if you highlight several controls and select Layout→Align Left all the controls you highlighted will shift horizontally to the left until their left sides are all aligned. Another handy fea-

ture that is not evident from the Layout menu selections is called Drag and Pick. Say you have resized several controls and you now want them to be all the same size. Highlight several controls that have different sizes, hold down the command key (MacOS) or the alt key (MSWindows), and then drag to another destination control. When you release the mouse, all selected views will be made the same size as the destination control.

### String output to a dialog box

In the previous example, the interactor contained an integer field change for input and integer fields nickels, dimes, and pennies for output. Input and output with real variables is similar. Sometimes it is desirable to use string output to display the result in a way that is more conventional for the user.

For example, suppose you want to include a dollar amount as the output. You want to prefix the value with a dollar sign $ and have the value displayed to the nearest cent, which is two places past the decimal point. To make the output look this nice requires you to convert the real value to a string with the desired format. Figure 5.11 shows a dialog box for such a scenario.

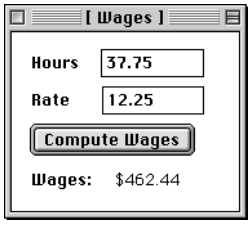



**Figure 5.11**
A dialog box with string output for Listing 5.12.

Figure 5.12 (page 85) shows the program corresponding to the dialog box of Figure 5.11. Even though the output appears to be a real value, the program shows that it is a string, because the type of d.result is an array of 16 characters. When procedure ComputeWages executes, it calculates wage, which is a real variable, as the product of d.hours and d.rate. With the input shown in Figure 5.11, wage would get the real value 462.4375. The program then calls procedure RealToString from module PboxStrings to convert the real value to a string value with two places past the decimal point. In the actual parameter list of RealToString, 1 is the field width, which will expand to accommodate any real value, and 2 is the number of places past the decimal. This procedure call gives the string value "462.44" to d.result. The next statement concatenates the dollar sign to the beginning of d.result, which finally gets displayed with the call to Dialog.Update.

### Problems

**1.** Design a dialog box to input an integer value for the number of feet and a real value for the number of inches. When the user clicks a button in the dialog box, compute the equivalent length in meters and output the results of the computation to the Log. One inch is exactly 0.0254 meters and one foot is exactly 12 inches. Here is a sample output to the Log.

```
Feet: 4
Inches: 3.8
Meters: 1.31572
```

**2.** Work Problem 1, but display the computed value for meters in the dialog box.

**3.** Work Problem 1, but display the computed value for meters in the dialog box to two places past the decimal point. Use string output as in the program of Figure 5.12.

```
MODULE Pbox05C;
   IMPORT Dialog, PboxStrings;

   VAR
      d*: RECORD
         hours*, rate*: REAL;
         result-: ARRAY 16 OF CHAR
      END;

   PROCEDURE ComputeWages*;
      VAR
         wage: REAL;
   BEGIN
      wage := d.hours * d.rate;
      PboxStrings.RealToString(wage, 1, 2, d.result);
      d.result :="$" + d.result;
      Dialog.Update(d)
   END ComputeWages;

BEGIN
   d.hours := 0.0; d.rate := 0.0;
   d.result := ""
END Pbox05C.
```

**Figure 5.12**
A program that produces string output to a dialog box.

**4.** Design a dialog box to input a real value for the temperature in Fahrenheit. When the user clicks a button in the dialog box, compute the equivalent temperature in Celsius and show both temperatures on the Log with their values identified appropriately as are the values in Problem 1.

**5.** Work Problem 4, but display the computed value for the Celsius temperature in the dialog box.

**6.** Work Problem 4, but display the computed value for the Celsius temperature in the dialog box to one place past the decimal point. Use string output as in the program of Figure 5.12.

**7.** Design a dialog box to input two real values for the lengths of two perpendicular sides of a right triangle. When the user clicks a button in the dialog box, compute the length of the hypotenuse and show all three lengths on the Log with their values identified appropriately as are the values in Problem 1.

**8.** Work Problem 7, but display the computed value for the length of the hypotenuse in the dialog box.

**9.** Work Problem 7, but display the computed value for the length of the hypotenuse in the dialog box to one place past the decimal point. Use string output as in the program of Figure 5.12.

**10.** Design a dialog box to input a real value for the radius of a circle. When the user clicks a button in the dialog box, compute the circumference and area and show all three mea-

sures on the Log with their values identified appropriately as are the values in Problem 1. Import the value of π from module Math for your computations.

**11.** Work Problem 10, but display the computed values for the circumference and area in the dialog box.

**12.** Work Problem 10, but display the computed values for the circumference and area in the dialog box to one place past the decimal point. Use string output as in the program of Figure 5.12.

**13.** Design a dialog box to input an integer value for the total number of hours. When the user clicks a button in the dialog box, compute the equivalent number of days, weeks, and hours. Show all time measures on the Log as in the following sample.

```
Total hours: 4123
Number of weeks: 24
Number of days: 3
Number of hours: 19
```

**14.** Work Problem 13, but display the computed values for the number of weeks, days, and hours in the dialog box.

**15.** Construct a five-function integer calculator as shown in the dialog box of Figure 5.13. The result from the dialog box is shown just after the user pressed the button labeled *. You will need to export five procedures, one for each button.

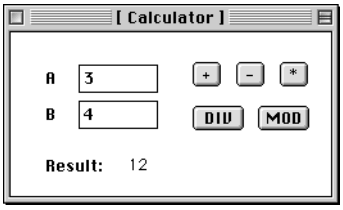



**Figure 5.13**
The dialog box for Problem 15.

**16.** Construct a four-function real number calculator similar to the integer calculator of Problem 15. Allow the user to add, subtract, multiply or divide two real numbers. You will need to export four procedures, one for each button.