Object REXX for Windows

OODialog Method Reference

Version 2.1

SH12-6727-00

Object REXX for Windows

OODialog Method Reference

Version 2.1

SH12-6727-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Appendix. Notices" on page 525.

First Edition, March 2001

This edition applies to Version 2.1 of IBM Object REXX for Windows Development Edition (5639-M68), and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

This edition replaces SH12-6224-02.

Contents

About This Book

This book describes the OODialog class library in Object REXX for Windows Version 2.1, and how to use it to program user interfaces.

Who Should Use This Book

This book is intended for Object REXX programmers who want to design graphical user interfaces for their applications.

How This Book is Structured

This book is structured in two parts.

Part 1 is a tutorial. It is organized as a guided tour through different functions and classes and uses examples that are enhanced from chapter to chapter.

Part 2 is intended as reference material. It describes the classes and methods in detail.

Related Information

Object REXX for Windows: Programming Guide, SH12-6726

Object REXX for Windows: Reference, SH12-6725

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other REXX documentation:

- Visit our home page at `http://www.ibm.com/software/ad/obj-rexx/support.html#Buy or get support`. There you can access the Internet Online Form where you can enter comments and send them.
- Send your comments by e-mail to `swsdid@de.ibm.com`. Be sure to include the name of the book, the part number of the book, the version of REXX, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative. The mailing address is on the back of the Readers' Comments form. The fax number is +49-(0)7031-16-4892.

How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▸▸— symbol indicates the beginning of a statement.

  The —▸ symbol indicates that the statement syntax is continued on the next line.

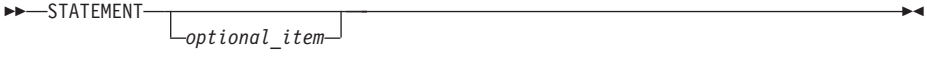
  The ▸— symbol indicates that a statement is continued from the previous line.

  The —▸◂ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ▸— symbol and end with the —▸ symbol.

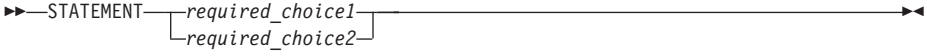
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.



- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.



- If choosing one of the items is optional, the entire stack appears below the main path.

- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

- An arrow returning to the left above the main line indicates an item that can be repeated.

  A repeat arrow above a stack indicates that you can repeat the items in the stack.
- A set of vertical bars around an item indicates that the item is a *fragment*, a part of the syntax diagram that appears in greater detail below the main diagram.

fragment:

- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:

Part 1. Developing Graphical User Interfaces with OODialog

This part is a tutorial that demonstrates how to use the Resource Workshop and the classes defined by OODialog to design and control graphical user interfaces for your Object REXX programs.

This tutorial does not cover all available OODialog methods. However, it introduces many of them in the form of small samples so that most of the commonly required dialog functionalities are covered by this book. A detailed description of all classes and methods is given in the second part of this book.

To learn more about GUI programming with OODialog, check out the sample programs delivered with OODialog in the `OODIALOG\SAMPLES` directory.

Chapter 1. Conceptual Overview

This chapter gives you a conceptual overview of how OODialog is designed and builds a bridge between native Windows objects and Object REXX objects. It also introduces the standard dialog classes, which provide an easy graphical user interface for smaller applications.

The Design of OODialog

The GUI builder consists of three basic parts: the Object REXX interface to the Windows API – the external functions – written in C, an IBM resource editor called *Resource Workshop*, and the Object REXX dialog classes. The external functions provided by OODIALOG.DLL are for internal use only and, with a few exceptions, must not be registered and called directly. The registration and calls are managed by the OODialog classes. The other two components are fundamental for creating a professional front end for your application. You can also create dialogs completely dynamically without using the Resource Workshop, which is also used in the standard dialogs that are described next.

The following topic ″Standard Dialogs″ gives you an introduction to producing frequently used dialogs and to help making the development of simple user interfaces as easy as possible. However, if you plan to develop more complex user interfaces and want to learn more about the concept of OODialog, skip this topic and concentrate on the “IBM Resource Workshop” on page 9 and “Object REXX Dialog Classes” on page 11.

Standard Dialogs

OODialog provides dialogs that can be used to implement user interfaces easily without defining dialog layouts and subclasses. These dialogs are called *standard dialogs* and are defined by the `OODPLAIN.CLS`. Most of these dialogs can be used by writing a two-liner, and they are useful when you need a quick interface to the user. The design of the dialog classes and the way they can be used is described in detail after the introduction of the standard dialogs.

Timed Message Box

The *timed message box* is used to display a message to the user for a specified time. The first argument contains the message, the second the box title, and the third the time, in milliseconds, during which the message box is visible. As with the other dialogs, you have to call Execute to run the dialog. See Figure 1 for an example.


```
dlg = .TimedMessage~new("This is a timed message!","Hello !", 3000)
dlg~execute
::requires "OODPLAIN.CLS"
```

Figure 1. Timed Message Box

The ::requires "OODPLAIN.CLS" statement is required for the rest of the examples in this topic but not separately listed.

Input Box, Integer Box, Password Box

The *input box* is used to read a text string from the keyboard, similar to what the REXX *PULL* instruction does in a DOS window. The first argument contains a message, and the second the box title. The Execute method runs the dialog and returns the text string, as shown in Figure 2.


```
dlg = .InputBox~new("This is an input dialog,",
"please enter some data","InputBox")
say "Your InputBox data :" dlg~execute
```

Figure 2. Input Box Dialog

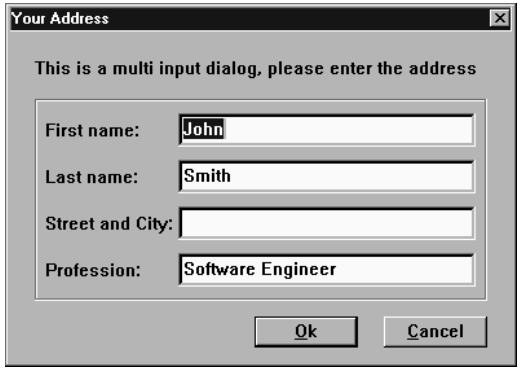
The *integer box* and the *password box* are similar to the input box, with one exception: In the integer box you can return numerical data only, and in the password box the input characters are displayed as asterisks (*).

```
dlg = .IntegerBox~new("This is an integer dialog,",
"please enter numerical data","IntegerBox")
say  "Your IntegerBox data: " dlg~execute

dlg = .PasswordBox~new("Please enter your password","Security")
say  "Your PasswordBox data: " dlg~execute
```

Multiple Input Box

The *multiple input box* is used to read more than one text string from the keyboard. The first argument contains a message, the second the dialog title, the third a stem containing the labels for the entry lines, and the fourth a stem

containing the initialization values for the entry lines. The first stem containing the labels must start with 1 and continue in increments of 1. The second stem must start with 101 and continue in increments of 1. The Execute method runs the dialog. The data is placed into the second stem and into the object attributes that have the same names as the labels, with ampersands (&), colons (:), and blanks removed as shown in Figure 3.

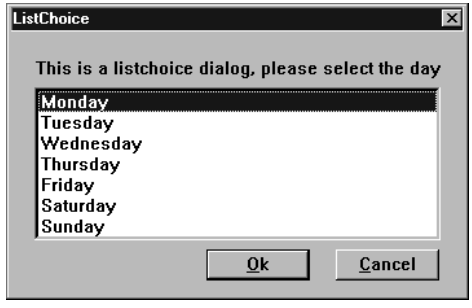



```
lab.1 = "First name: "
lab.2 = "Last name: "
lab.3 = "Street and City: "
lab.4 = "Profession: "

addr.101 = "John" ; addr.102 = "Smith" ; addr.103 = ""
addr.104 = "Software Engineer"

dlg = .MultiInputBox~new("This is a multi input dialog,",
"please enter the address","Your Address",lab., addr.)
if dlg~execute = 1 then
do
say dlg~FirstName ; say dlg~LastName
say dlg~StreetandCity ; say dlg~Profession
end
```

Figure 3. Multiple Input Box Dialog

List Choice

The *list choice* dialog is used to select one entry of a list. The first argument contains a message, the second the dialog title, and the third a stem containing the entries for the list. The stem suffixes must be 1 to n in increments of 1. The Execute method runs the dialog and returns the selected text string as shown in Figure 4.




```
lst.1  = "Monday" ; lst.2  = "Tuesday"
lst.3  = "Wednesday" ; lst.4  = "Thursday"
lst.5  = "Friday" ; lst.6  = "Saturday"
lst.7  = "Sunday"

dlg = .ListChoice~new("This is a listchoice dialog,",
"please select the day","ListChoice",lst.)
say "Your ListChoice data:" dlg~execute
```

Figure 4. List Choice Dialog

Multiple List Choice

The *multiple list choice* dialog is similar to the *list choice* dialog, except that the user can select more than one list entry and that the return value is not the selected strings but the list index of the selected list entries separated by blanks. The Execute method runs the dialog and returns the numbers of the selected entries, separated by blanks as shown in Figure 5.


```
lst.1  = "Monday" ; lst.2  = "Tuesday" ;
lst.3  = "Wednesday" ; lst.4  = "Thursday"
lst.5  = "Friday" ; lst.6  = "Saturday";
lst.7  = "Sunday"

dlg = .MultiListChoice~new("This is a multiple list choice dialog,",
                                    "please select the days",,
                                    "MultipleListChoice",lst.)
s = dlg~execute
if s <> 0 then do while s <> ""
  parse var s res s
  if res <> "" then say lst.res
end
```

Figure 5. Multiple List Choice Dialog

The `do-while` loop at the end of this sample is to parse the individual indexes from the return value and to display its corresponding string.

Check List

The *check list* dialog is similar to the multiple list choice dialog. Instead of displaying the alternatives in a listed form, check boxes are used. See Figure 6 for an example.

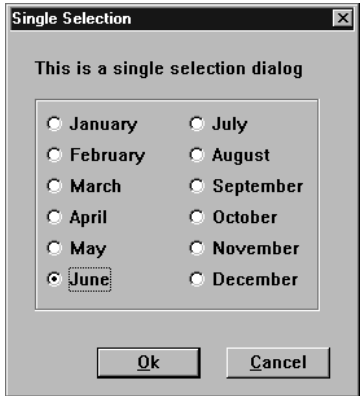

```
lst.1  = "Monday" ; lst.2  = "Tuesday"
lst.3  = "Wednesday" ; lst.4  = "Thursday"
lst.5  = "Friday" ; lst.6  = "Saturday"
lst.7  = "Sunday"

do i = 101 to 107
   chk.i = 0
end
chk.101 = 1

dlg = .CheckList~new("This is a checklist dialog",,
                           "Checklist",lst., chk.)
if dlg~execute = 1 then do
   say "Your CheckList data: "
     do i = 101 to 107
       a = i-100
       if chk.i = 1 then say lst.a
     end
end
```

Figure 6. Check List Dialog

The first argument contains a message, the second the dialog title, the third a stem containing the alternatives, and the fourth a stem containing the initialization values of the check boxes. Optional parameters are the length of the check boxes in dialog units and the maximum number of check boxes in one column. The first stem suffix must start with 1 and continue in increments of 1. The second stem suffix must start with 101 and continue in increments of 1. To preselect a check box, the corresponding stem entry must be assigned to 1. The Execute method runs the dialog. The data is returned in the second stem and in the object attributes named after the check box labels. For example, `chk.102` and `dlg~tuesday` represent the same check box. If a check box has been selected, its stem entry and the relative object attribute are 1, otherwise they are 0.

Single Selection

The *single selection* dialog is similar to the list choice dialog. Instead of displaying the alternatives in a listed form, radio buttons are used. The arguments are the same as in the check list dialog, except that fourth argument number is not a stem but the number of the radio button to be preselected; in the example, June is preselected. The Execute method runs the dialog and returns the number of the selected radio button as shown in Figure 7.




```
mon.1 = "January" ; mon.2 = "February" ;
mon.3 = "March"   ; mon.4 = "April"
mon.5 = "May"      ;mon.6 = "June"     ;
mon.7 = "July"    ; mon.8 = "August"
mon.9 = "September";mon.10= "October"  ;
mon.11= "November"; mon.12= "December"

dlg = .SingleSelection~new("This is a single selection dialog",,
                           "Single Selection",mon.,6,,6)
s = dlg~execute
say  "Your SingleSelection data: " mon.s
```

Figure 7. Single Selection Dialog

If you need more complex user interfaces or customized layouts, you can define your dialog using the Resource Workshop and then write an Object REXX program that loads, runs, and controls this dialog.

IBM Resource Workshop

With the Resource Workshop you can create and manipulate Windows resources. This enables you to create graphical user interfaces for your programs. For a tutorial to help you get started on creating your user interfaces in this way, see “Chapter 2. Creating your User Interface” on page 13.

.

In Windows, a resource is, among other things, a file or a part of a file that describes the layout of a window. You can use resources to compose the dialogs you want to execute using Object REXX. Within the Resource Workshop you can determine the size, frame type, and style of the dialog, and you can place text, control items, and data fields that the dialog should contain. The various control items that are supported by the Resource Workshop are shown in Figure 8.

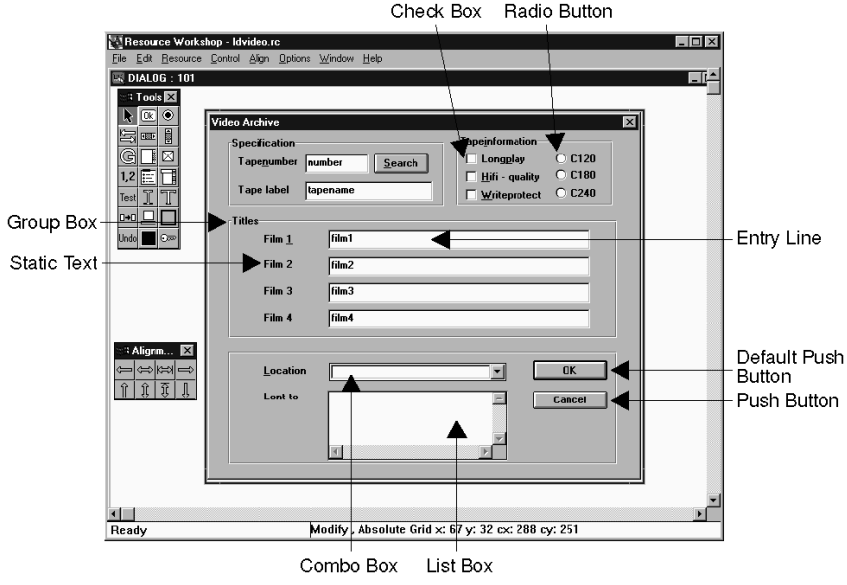


Figure 8. The Resource Workshop

With the Resource Workshop you can add and place one or more of these items into the dialog. When you have finished your dialog design, save the data to a resource script (.RC), which you need to execute the dialog with Object REXX.

Resources

A resource can be simply a single bitmap (.BMP) file, or an icon (.ICO), either of which you can create on your own. A bitmap file is a binary representation of a graphic image in a program. Each bit, or group of bits, in the bitmap represents one pixel of this image on your screen. Icons are small bitmapped images. Windows programs typically use customized icons to represent minimized windows.

A resource can also be a complex project that contains many kinds of graphic elements. These kinds of resource projects begin the creation of a .RC file.

After you create this initial file, you populate it with different kinds of elements. From within the Resource Workshop you have access to the Bitmap Editor, which enables you to create your own bitmaps and icons.

You can also base your resource project on a binary resource file (.RES). This type of file can contain one or more compiled resources. Typically, you compile all the resources for an application into a single .RES file, and then bind the .RES file to an executable file.

Note: The Resource Workshop is a resource editor only, and cannot be equated with a graphical programming environment. You cannot assign program code to a dialog or its items within the Resource Workshop. Also, the Resource Workshop supports features that are not supported by OODialog.

Object REXX Dialog Classes

Dialog classes are your interface to Windows. There is more than one class, because there is more than one way of executing a dialog with Object REXX, and there are dialogs with different behaviors.

The class of interest at the moment is the UserDialog class. This class contains the methods to execute a dialog that has been created with the Resource Workshop and stored into a resource script (.RC).

Object REXX Objects and Windows Objects

To use the UserDialog class, you must create an object, which is the case for every other class also. OODialog differentiates between two kinds of objects, the Object REXX object, which must be an instance of a subclass of the BaseDialog or PlainBaseDialog, and the dialog window itself, which is an object of the Windows system itself. After you created an instance of the UserDialog you have an Object REXX object but not a Windows object, which means that Windows has not created a dialog window yet nor allocated memory for one.

The next step is loading the resource script. You then have a template describing the layout of the dialog in a Windows internal format, but still not a Windows object.

When you call the method to execute the dialog in Object REXX, a real Windows object is created, data is transferred from the Object REXX object to the Windows object, and the dialog is displayed. Now you can enter data and work (communicate) with the items of the dialog or the dialog itself.

When you finish the dialog, its data is received in Object REXX, and the Windows object is deleted.

Separate Data Areas

Why is it necessary to transfer data to and from the Windows object?

In a dialog you give information to the user and receive feedback from the user. You can get feedback via push buttons, entry lines (also called entry fields), list or combo boxes, radio buttons, or check boxes. Although you can assign Object REXX object attributes to each of these data items (this is done automatically in most cases when loading the resource script or dynamically adding dialog items), the memory needed for the dialog data items is allocated and managed by the Windows object itself.

Whenever the user changes the state of one of these dialog items, the internal data buffer of the Windows dialog is updated, but the corresponding Object REXX object is not aware of any modification at that point and so the object attributes remain unchanged.

To reflect the state of the Windows dialog in the Object REXX object, the UserDialog class defines methods to exchange data of the Object REXX dialog object with the Windows dialog, and vice versa. Before executing the Windows dialog, data is automatically copied from the Object REXX object to the Windows dialog, and after dialog execution, data is copied from the Windows dialog back to the Object REXX dialog object. Keep in mind that the Object REXX dialog attributes are separated from the Windows dialog data. To keep the original data of the Object REXX dialog object after the Windows dialog has been finished, press Cancel or Esc.

Methods Dealing with Windows Objects

Most of the methods provided by the BaseDialog deal with real Windows objects and therefore are first applicable when the Window dialog was created. These methods can either be used within InitDialog, within a method called by a dialog event, or after calling ExecuteAsync.

The “Summary of User Dialog Processing” on page 60 gives you an overview of all the methods and their calling sequence that are involved in running a UserDialog object.

Chapter 2. Creating your User Interface

This chapter contains a tutorial that you can use to get started on creating the graphical elements of the user interfaces for your programs. These elements are the familiar items – dialog boxes, command buttons, and so on – that will make your programs attractive and easy to use for Windows-oriented users.

When you are creating the user interface for a program of your own, it will be important to plan its design carefully. Consider how best to arrange the interface so that the user can work with it as simply and intuitively as possible. To help you to do this, make diagrams of your dialogs. Use a flowchart, if necessary, to help you to create the dialogs – and the connections between them – that lead the users securely to what they want to do. If possible, consult some potential users.

If you are interested in working more intensively with graphical user interface development, you will find a wide variety of published guides on the subject to help you.

The graphic element you will use most frequently in your user interfaces will probably be the dialog box, because of its versatility and the wide range of command options and user-input devices it can use. In this tutorial, you will create a dialog box and populate it with some standard user-interface elements.

To create your dialog, you use the IBM Resource Workshop. You perform the following steps:

1. Create a new project
2. Create a new dialog
3. Configure the Resource Workshop
4. Add control items to your dialog

Creating a New Resource Project

To begin with, you need to create the project in which you will place all the resources for your interface. Resources are the graphic elements that make up the interface.

To create a new project:

1. Open the IBM Resource Workshop. You can do this in several ways:

On the Windows taskbar, select **Start/Programs/Object REXX for Windows/IBM Resource Workshop**.

Alternatively, type `Workshop` at the **Command Prompt**.

Or, if the Object REXX Workbench is already open, select **Tools/OODialog/Resource Workshop** from the menu bar.

The Resource Workshop opens.

2. In the Resource Workshop menu bar, select **File/New resource project**.

   The **New resource project** dialog appears.

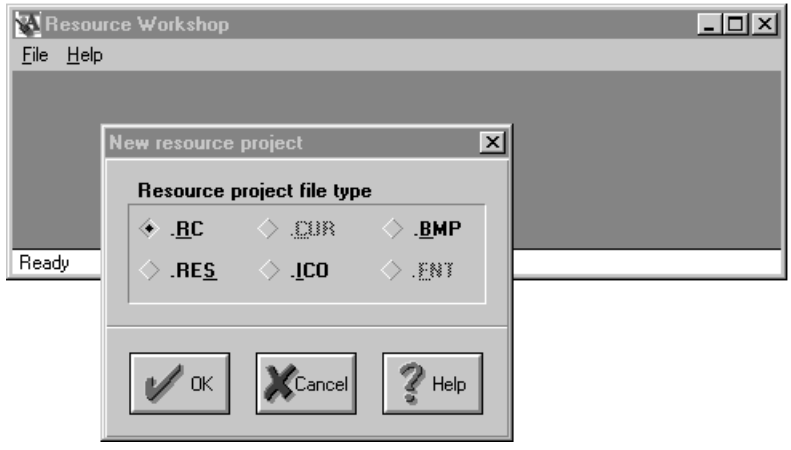


*Figure 9. The **New resource project** dialog*

3. Select the .RC type, if it is not already selected. This creates a project that can include all the different resources you create.

   If you would like information about the other resource types that you can create, click the **Help** button. See also "Resources" on page 10.

4. Click **OK**.

   An empty window appears, representing your new project. You must now name your project and decide a location for it, so that the resources you create will automatically be saved in the right location.

5. In the Resource Workshop menu bar, select **File/Save resource project**.

   The **Save file as** dialog appears.

6. Use the **Directories** list to choose the location where your project will be saved. Double-click on a drive letter to raise a list of available directories above the list of drives, and then select the directory you want.

   In the **File type** edit field, choose the **RC resource script** file type.

   In the **New file name** field, add the name you wish to give to your project. The name can have no more than eight letters; use the suffix `.rc`. For now, name your project `newproj.rc`.

7. Click **OK**. Your new project dialog is now renamed to reflect the new name you have given to your project.

Creating a New Dialog

You now create the first dialog box in your new project.

1. In the Resource Workshop menu bar, select **Resource/New**.

   The **New resource** dialog appears.

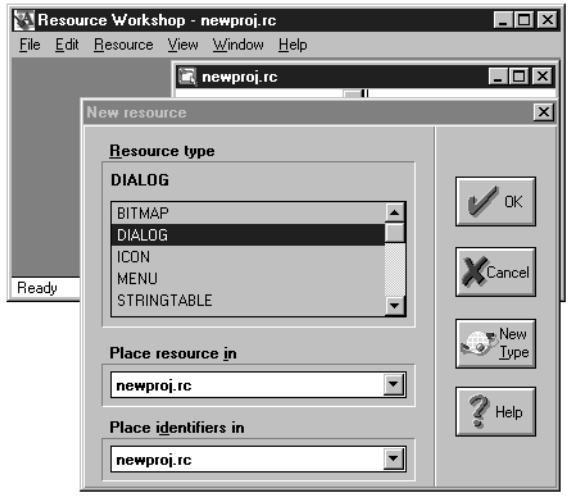


*Figure 10. The **New Resource** dialog*

2. From the **Resource type** list, select **Dialog**. Ensure that your new project name appears in both of the entry fields **Place resource in** and **Place identifiers in**.
3. Click **OK**.

   The **DialogExpert** dialog appears.

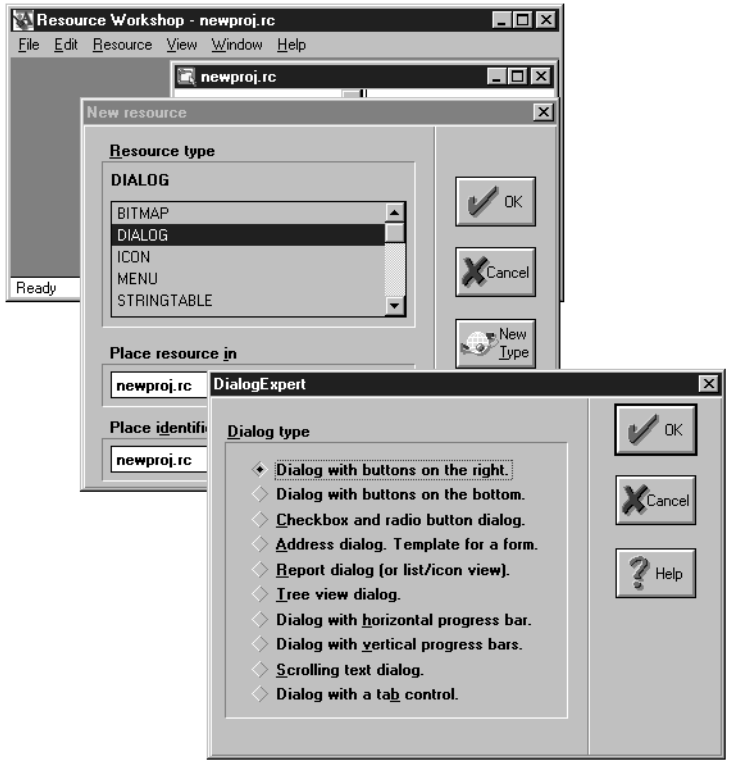


*Figure 11. The **DialogExpert** dialog*

4. You can now choose the basic appearance of your dialog from the list of dialog types. The names are self-explanatory; if you need more information, click the **Help** button.

   For now, choose the first type, **Dialog with buttons on the right**.

5. Click **OK**.

   The **Dialog Editor** opens, and displays your new dialog box, as well as some toolbars that you can use later to create controls in your dialog.

6. You now need to name your dialog. To do this:

   From the **Resource** menu, select **Rename**.

   The **Rename resource** dialog appears.

*Figure 12. The **Rename resource** dialog*

7. In the **New name** field, type the name you wish to give to your new dialog.

   For now, use `MyNewDialog`.
8. Click **OK**.

   The **Resource Workshop** dialog appears.

*Figure 13. The **Resource Workshop** dialog*

9. Click **Yes** to confirm that you wish to create a new identifier for the dialog.

   The **New identifier** dialog appears.

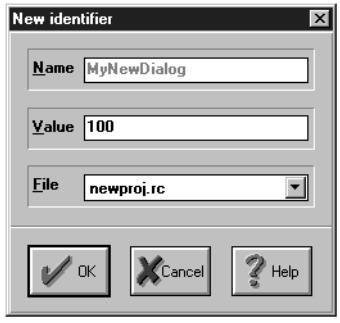


*Figure 14. The **New identifier** dialog*

10. You now need to allot to your dialog a number that will identify it. The number can be any number greater than 10.

    In the **Value** entry field, enter the number 100.

11. Click **OK**. The new name and numerical identifier are added to your dialog.

12. Next you need to modify the appearance of your dialog. To do this, call up the **Window style** dialog. You can do this in several ways:

    Double-click the dialog's title bar; double-click an empty space within the dialog; or right-click an empty space within the dialog, and select **Style** in the ensuing dialog.

    Alternatively, select the dialog by clicking its title bar or its outer border, and then select **Control/Style** from the menu bar.

    The **Window style** dialog appears.

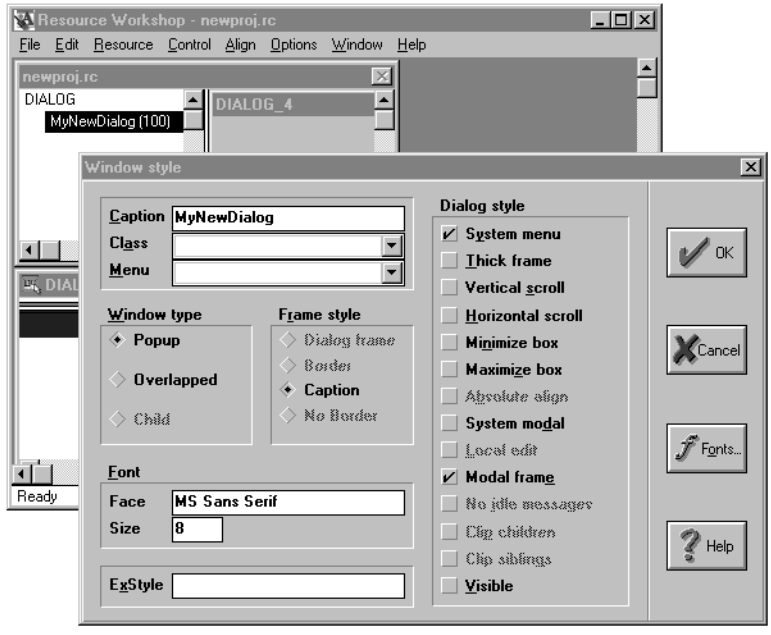


*Figure 15. The **Window style** dialog*

13. In the **Caption** entry field, enter the name you wish the user to see as the caption of your dialog.

    Leave the **Class** and **Menu** fields empty.

    For information about the other options presented, click the **Help** button.

14. When you are ready, click **OK** to complete setting the appearance options for your dialog.

15. You can now specify the size of your dialog, and where it will appear on the user's monitor screen.

    To do this, first maximize the Dialog Editor window. Drag the dialog to the desired location, and drag its sides and/or corners to resize it.

    Alternatively, select the dialog box, and from the menu bar choose **Align/Size**. In the ensuing **Size dialog**, you can set precise values for the X and Y coordinates of the dialog and for its dimensions.

    For more information about the values you can specify, click **Help**. Click **OK** to complete specifying your dialog size and position.

Configuring the Resource Workshop

If this is the first time that you are using the Resource Workshop to create dialogs after a new installation of Object REXX for Windows, you will need to set some configuration options for the Dialog Editor. You will need to do this only once. Skip this step if the configuration is done already.

1. With your new dialog box visible, select **Options/Preferences** from the menu bar.

   The **Preferences** dialog opens.

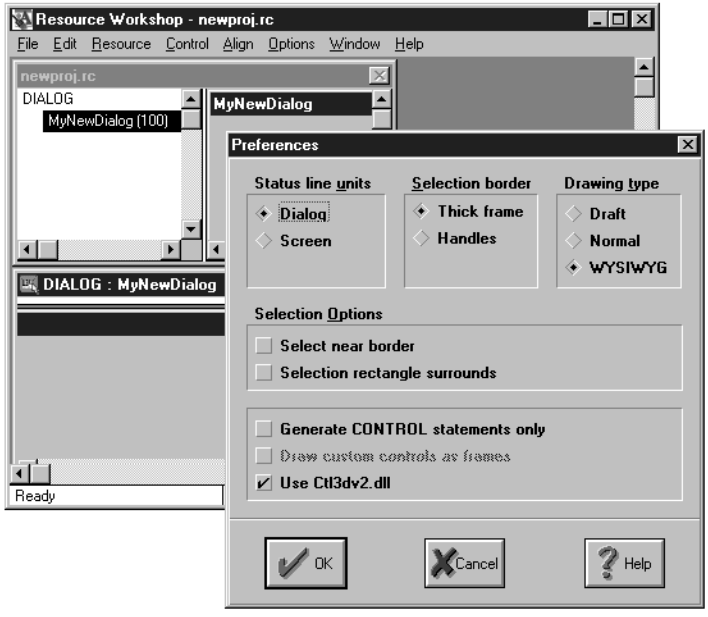


*Figure 16. The **Preferences** dialog*

2. For information about the options in the **Preferences** dialog, click the **Help** button.

   For now, set the following options:

   In the **Status line units** area, select **Dialog**.

   In the **Selection border** area, select **Thick frame**.

   In the **Drawing type** area, select **WYSIWYG**.

   In the **Selection Options** area, deselect both options.

   Select the **Use Ctl3dv2.dll** option, and deselect the others in this area.
3. Click **OK** to confirm your preferences.

Adding Control Items to your Dialog

You now add some control items to your new dialog box. Three such controls are already in place, the command buttons **OK**, **Cancel**, and **Help**.

You will now enable your users to provide information to the program by entering some text, for example, their names and addresses. You make this possible by placing text entry fields in your dialog.

Beside your new dialog box are two floating toolbars, **Tools** and **Alignment**. (If they don't appear, select **Options/Show Tools** and **Options/Show Alignment** from the menu bar.) You can move these toolbars around as you wish within the Resource Workshop window.

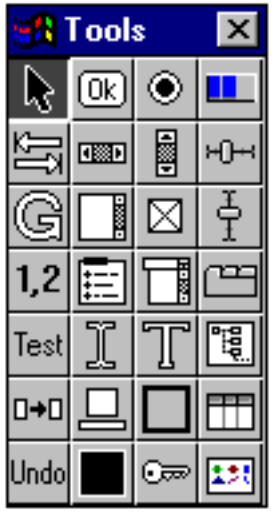


Figure 17. The Tools toolbar

To add a text entry field to your dialog box:

1. In the **Tools** toolbar, click the text cursor icon  .

   Alternatively, select **Control/Edit Text** in the menu bar.
2. Place the pointer inside the dialog. The pointer has now changed appearance; it now shows the text cursor icon with a crosshair indicator in the top left corner.

   Position the crosshair where you want the top left-hand corner of your text entry field to appear, and drag to where you want the opposite corner to appear.

   Your new text field appears in your dialog.

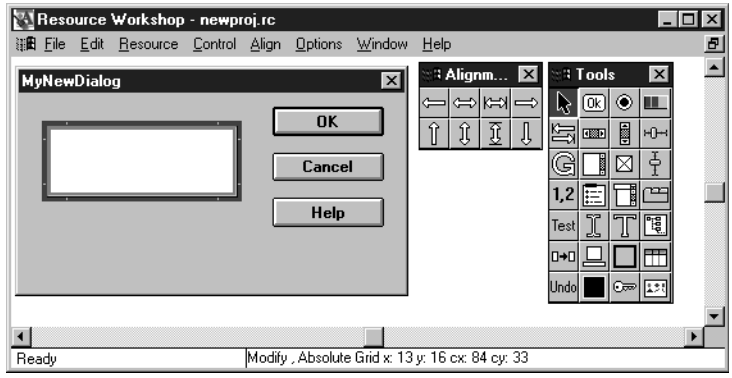


Figure 18. Creating a new text entry field

3. You now need to modify the text field to suit your needs. To alter the style of the text field, either double-click it, or select it and choose **Control/Style** from the menu bar.

   The **Edit text style** dialog appears.

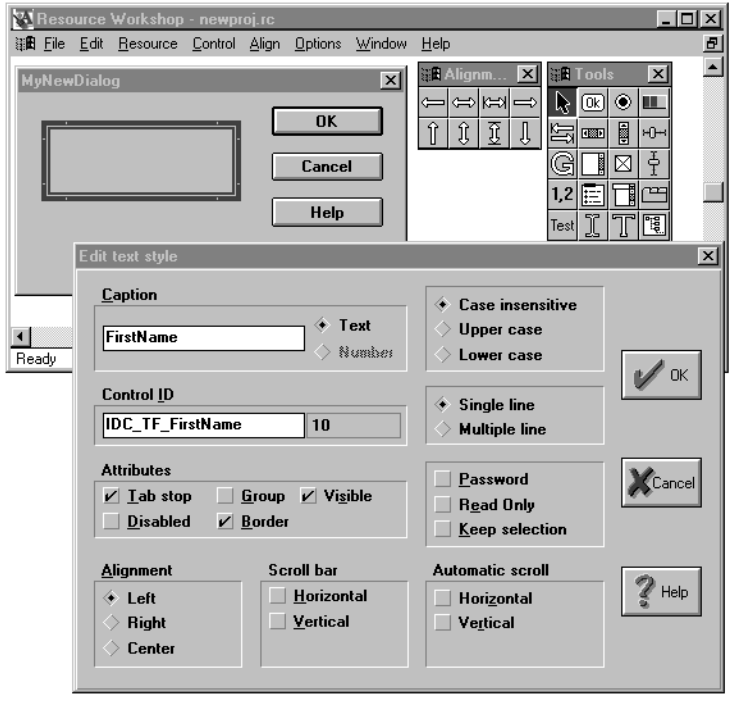


*Figure 19. The **Edit text style** dialog*

4. In the **Caption** entry field, enter the text for the caption that the user will see for the text field.

   In the **Control ID** field, enter an appropriate identifier that will help you to identify this text field later. For example, if you create a number of text fields to contain users' names and addresses, you might name them `IDC_TF_FirstName`, `IDC_TF_LastName`, `IDC_TF_StreetName`, and so on. This will make it easy for you to identify the various elements later when you want to assign their functionality to them in the REXX program.

   For information about the other options in the **Edit text style** dialog, click the **Help** button. When you are finished modifying the text field, click **OK**.

   You have now successfully placed a text field element into your new dialog.

Continue to experiment by placing other control elements in your dialog. Use the icons in the two middle columns of the **Tools** toolbar to choose controls that you can add to your dialog. (For a description of the functions of the icons in the **Tools** toolbar, see the next section, ″The Tools Toolbar.″ Alternatively, choose new controls from the list given in the **Control** menu.

Use the **Alignment** toolbar to locate controls precisely in your dialogs. In the case of each new element, modify its style as you did in step 4.

Add at least one of each type of dialog element to your sample dialog.

When you are familiar with the process of creating the graphic elements of your user interfaces, go on to Chapter 3, ″Using a Dialog with Object REXX″, to learn how to add functionality to these elements.

The Tools Toolbar

All of the control items that you can add to your dialogs are available from the Tools toolbar. To add a control item, simply click the appropriate icon for the feature, and then draw it in your dialog by using the pointer. (An alternative way to select control items is to use the **Control** menu in the Resource Workshop.)

The control items and related options that are available from the Tools toolbar are as follows:

Table 1. The Tools toolbar

 Select button	 OK push button	 Auto radio button	 Progress bar
 Tab set tool, to mark items for tab stops	 Horizontal scroll bar	 Vertical scroll bar	 Horizontal slider
 Group tool	 Text scroll bar	 Check box	 Vertical slider
 Set order tool. Sets the tab stop sequence.	 Group box	 Combo box. Enlarge it downward for long lists.	 Tab control
 Test your dialog. Activates your dialog in a new window.	 Dynamic text field	 Static text field	 Tree view
 Duplicate	 [This is a non-functional button.]	 Black frame. Puts an empty rectangular frame into your dialog.	 Report view
 Undo	 Black rectangle. Puts a rectangle into your dialog.	 New custom control	 List view

Chapter 3. Using a Dialog with Object REXX

Once you are familiar with the Resource Workshop, you can either continue with this chapter using the dialog that you created, or take the one that is provided in the tutorial directory (EMPLOYE1.RC) as shown in Figure 20.

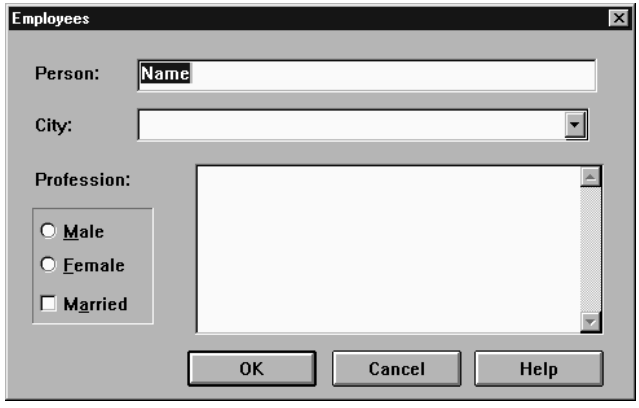


Figure 20. The Tutorial Dialog

The following sections demonstrate how to use the Workbench to create an Object REXX program that executes this dialog.

Using the Object REXX Workbench OODialog Template Generator

The "Template Generator" of the Workbench creates a program template that is based on a resource script created with the Resource Workshop. All the statements that are needed to execute the dialog, to define the new dialog class, to define its methods, to connect the dialog events to methods and the dialog data items to attributes is automatically added to the generated program template by the template generator.

To use the template generator in the *OODialog* popup menu under the *Tools* menu:

1. Start the Workbench.
2. Select menu item *Template Generator* in the *OODialog* popup menu under the *Tools* menu. The following dialog appears on your screen:

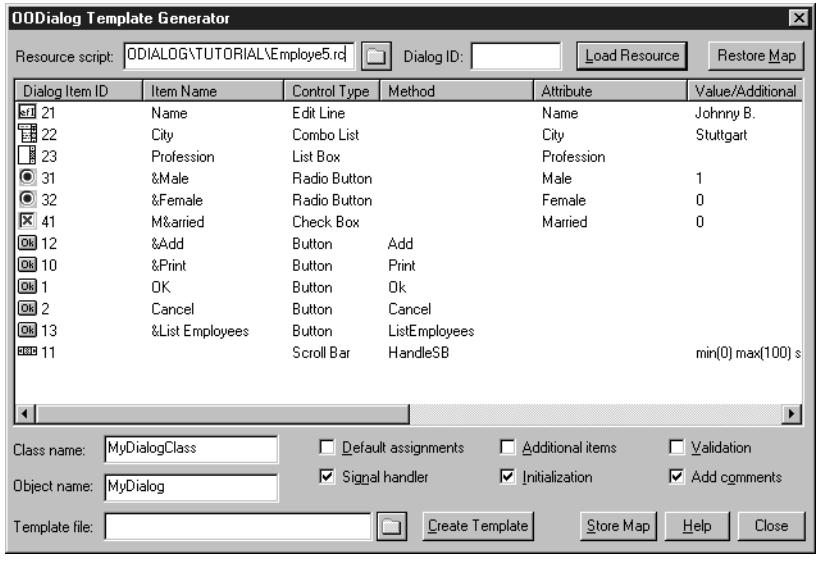


Figure 21. The OODialog Template Generator Panel

3. In the *Resource Script* field, enter the name of the resource script you previously created or the name of one of the tutorial resource scripts. The *Dialog ID* field is only required if more than one dialog is defined in the given resource script.
4. After you specified the dialog resource, you can press the *Load Resource* button to load the dialog definition into the template generator panel. If an error occurs, check whether you are using type "Text" for the caption field.

   All non-static dialog control items (ID not equal to -1) are now listed in the list box. You can see that for data items the field in the *Item Name* column, which you specified in the caption field of the Resource Workshop, is taken as default for the *Attribute* column, and for buttons it is taken as default for the *Method* column.

When you double-click on the *Dialog Item ID* field of a data item, a dialog appears on the screen as shown in Figure 22 on page 27.

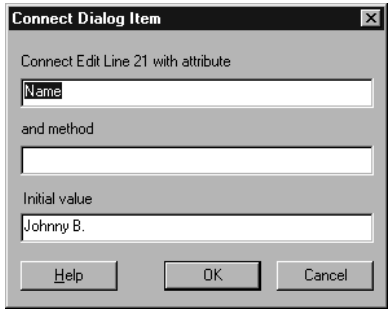


Figure 22. The Connect Data Item Panel

You can change the name of the attribute that is assigned to the data item and add a method that is called each time an event occurs for the data item. If, for example, you connect a method with a radio button, this method is called each time the radio button is selected. In this method you can, for example, check if the constellation of the radio button group is valid. The third field in this dialog is to set the initial value for the data item.

If you look at the radio buttons and check boxes, you see that the ampersand (&) has been removed for the attribute names. Blanks and colons (:) are removed from the names to make the attribute name a valid REXX symbol. The *Method* field is empty for data items when loading the resource. If you want to assign a method to a radio button or a check box or change the attribute name, double-click on the *Dialog Item ID* field as you did before. If you entered an invalid Object REXX symbol for the *Attribute* field or this attribute was already defined, the data item is connected to the object attribute DATAXXX, where XXX stands for the ID of the dialog item. If, for example, you assigned `Name` to dialog item 21 and `Name` was already defined for the dialog object, the data of this dialog item is exchanged with attribute DATA21.

When you double-click on the *Dialog Item ID* field of a button, the panel that appears looks as shown in Figure 23 on page 28.

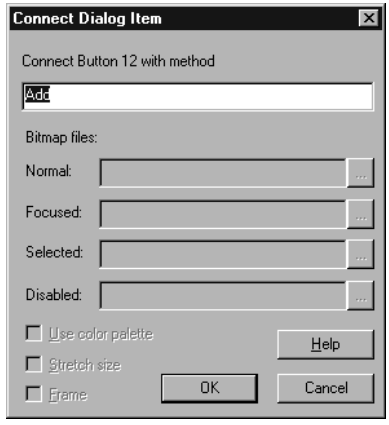


Figure 23. The Connect Button Panel

Most of the fields contained by this dialog are disabled for the buttons that are added to the dialog resource. These fields are needed to create buttons that contain a bitmap (bitmap button), which are discussed later. All you can set for normal buttons is the method that is called each time the button is pressed. To "owner-drawn" buttons you can assign several bitmaps stored in a file. The *Use Color Palette*, the *Stretch Size*, and the *Frame* options influence the appearance of the bitmaps. For further information, see "ConnectBitmapButton" on page 112.

The ID constants IDOK, IDCANCEL, and IDHELP, which are assigned to the buttons created automatically by the Resource Workshop, are automatically replaced by 1, 2, and 9.

If you double-click on the ID of a scroll bar, a panel appears that looks as shown in Figure 24.

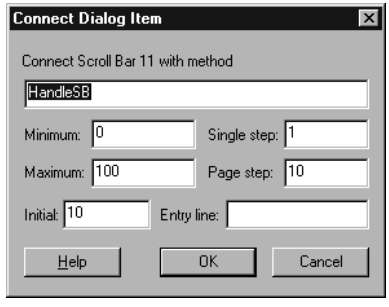


Figure 24. The Connect Scroll Bar Panel

In the scroll bar panel you can enter the name of the method that is intended to handle all scroll bar events, the minimum, maximum, and initial position values for the scroll bar, the number the scroll bar is increased or decreased each time the down or up (single step) or the page down or page up (page step) key is used on a focused scroll bar. The last field is to specify an entry line that you want to associate with the scroll bar that then operates as a spin control. The numerical value of the entry line reflects the position of the scroll bar, and vice versa. Leave this field empty if you do not want to use this functionality.

After you have done all the necessary modifications in the item association list, specify the name of the Object REXX program that you want to create in the *Template File* field. In the *Class Name* field you can specify the name of the OODialog subclass and in the *Object Name* you can specify the name of the class instance to be used in the program template.

The template generator supports the following options:

Default assignments
: Specifies whether attribute names are taken from the resource script (enabled) or explicitly assigned to the names listed in the item list. If you did not change an attribute name listed in the item list, you can leave this option disabled to reduce the size of the generated program.

Signal handler
: Specifies whether you want a signal handler set at the beginning of your script. If a signal handler is set, all conditions raised are handled by the signal handler, for example, to close a dialog in case of an error before the program is terminated.

Additional items
: Enable this option if you want to extend your dialog resource by adding dynamic items to the dialog. If this option is enabled, the DefineDialog method is overridden.

Initialization
: Enable this option if you want to initialize dialog items before the dialog pops up. If this option is enabled, the InitDialog method is overridden. You must enable this option, if you use scroll bars or want to initialize combo boxes or list boxes.

Validation
: Enable this option if you want to validate the dialog before the dialog is closed. If this option is enabled, the Validate method is overridden.

Add comments
: Enable this option to let the template generator add comments to the generated script.

You can store the dialog item settings in a file by pressing the *Store Map* button. With the *Restore Map* button, you can retrieve the same list again. To create the Object REXX program template, press the *Create Template* button and then close the template generator panel. The newly generated template is automatically loaded into the Workbench.

Here is how the program template could look like:

```
/**************************************************************************/
/* Name: testtg3.rex                                                      */
/* Type: Object REXX Script using OODialog                                */
/* Author:                                                                */
/* Resource: Employe1.rc                                                  */
/*                                                                        */
/* Description:                                                           */
/* This file has been created by the Object REXX Workbench OODIALOG       */
/* template generator.                                                    */
/*                                                                        */
/* Coypright (C) _____, 199_. All Rights Reserved.                        */
/*                                                                        */
/**************************************************************************/

/* Install signal handler to catch error conditions and clean up */
signal on any name CleanUp

EmployeeInput = .EmployeeInputClass~new
if EmployeeInput~InitCode = 0 then do
  rc = EmployeeInput~Execute("SHOWTOP")
end

/* Add program code here */

exit   /* leave program */


/* ---- signal handler to destroy dialog if error condition was raised  ----*/
CleanUp:
   call ErrorMessage "Error" rc "occurred at line" sigl":" errortext(rc),
                     || "a"x || condition("o")~message
   if EmployeeInput~IsDialogActive then EmployeeInput~StopIt

::requires "OODIALOG.CLS"    /* This file contains the OODIALOG classes */
```

Figure 25. A Generated Program Template (Part 1 of 3)

```
/* ---------------------------- Directives ---------------------------------*/
::class EmployeeInputClass subclass UserDialog

/* All connections are done explicitly */
::method InitAutoDetection
   self~NoAutoDetection    /* disable autodetection */


::method Init
  use arg InitStem.
  if Arg(1,"o") = 1 then
     InitRet = self~Init:super
  else
     InitRet = self~Init:super(InitStem.)   /* Initialization stem is used */

  if self~Load("P:\SAMPLES\WIN\OODIALOG\TUTORIAL\Employe1.rc", ) \= 0 then do
     self~InitCode = 1
     return
  end  /* Connect dialog control items to class methods */
  self~ConnectButton(1,"OK")
  self~ConnectButton(2,"Cancel")
  self~ConnectButton(9,"Help")

  /* Connect dialog data items to object attributes */
  /* These attributes are created dynamically */
  self~ConnectEntryLine(21,"Name")                      /* attribute Name */
  self~ConnectComboBox(22,"City", "LIST")               /* attribute City */
  self~ConnectListBox(23,"Profession")                  /* attribute Profession */
  self~ConnectRadioButton(31,"Male")                    /* attribute Male */
  self~ConnectRadioButton(32,"Female")                  /* attribute Female */
  self~ConnectCheckBox(41,"Married")                    /* attribute Married */
```

Figure 25. A Generated Program Template (Part 2 of 3)

```
  /* Initial values that are assigned to the object attributes */
  self~Name= ''                                      /* Entry Line */
  self~City= 'Stuttgart'                             /* Combo List */
  self~Profession= ''                                /* List Box */
  self~Male=1                                        /* Radio Button */
  self~Female=0                                      /* Radio Button */
  self~Married=0                                     /* Check Box */

  /* Add your initialization code here */
  return InitRet


::method InitDialog
  InitDlgRet = self~InitDialog:super
  /* Initialization Code (e.g. fill list and combo boxes) */
  return InitDlgRet

/* --------------------- message handler ----------------------------------*/


  /* Method OK is connected to item 1 */
::method OK
  resOK = self~OK:super  /* make sure self~Validate is called and
                                    self~InitCode is set to 1    */
  self~Finished = resOK  /* 1 means close dialog, 0 means keep open */
  return resOK


  /* Method Cancel is connected to item 2 */
::method Cancel
  resCancel = self~Cancel:super  /* make sure self~InitCode is set to 2 */
  self~Finished = resCancel      /* 1 means close dialog, 0 means keep open */
  return resCancel


  /* Method Help is connected to item 9 */
::method Help
  self~Help:super
```

Figure 25. A Generated Program Template (Part 3 of 3)

You can directly run the generated program. The dialog that you previously designed using the Resource Workshop appears on the screen, ready to retrieve data input. If you want to change the behavior of the dialog you can do the required modifications within this template. Some of the comments guide you to find the right location for your modifications.

The following sections demonstrate how to use OODialog to run and control this dialog within Object REXX step by step without using the template generator.

The PlainUserDialog Class

Start the Object REXX Workbench to create a new Object REXX script. First, add the statement *::requires "OODPLAIN.CLS"*, which loads a file in your script with all the class definitions necessary to use OODialog. Place the statement at the end of your script.

For the first step, you are going to display the dialog, enter some data, and quit the dialog. Because you will not implement any further functions, you will not have to define a new dialog subclass; you can use the existing class. The class you need for working with dialogs defined through a resource script is called the PlainUserDialog class. The `EMPLOYE1.RC` in the tutorial directory that you are going to use is shown in Figure 26.

```
/*****************************************************************************

employe1.rc

produced by IBM Object REXX Resource Workshop

*****************************************************************************/

#define DIALOG_1        1

100 DIALOG 6, 15, 241, 141
STYLE DS_MODALFRAME ¦ WS_POPUP ¦ WS_CAPTION ¦ WS_SYSMENU
CAPTION "Employees"
FONT 8, "System"{
CONTROL "Name", 21, "EDIT", WS_BORDER ¦ WS_TABSTOP, 50, 11, 177, 12
CONTROL "City", 22, "COMBOBOX", CBS_DROPDOWNLIST ¦ WS_CHILD ¦
                    WS_VISIBLE ¦ WS_TABSTOP, 50, 30, 174, 55
CONTROL "Profession", 23, "LISTBOX", LBS_STANDARD, 73, 52, 156, 65
AUTORADIOBUTTON "&Male", 31, 12, 71, 28, 12
AUTORADIOBUTTON "&Female", 32, 12, 84, 39, 12
AUTOCHECKBOX "M&arried", 41, 12, 99, 37, 12
DEFPUSHBUTTON "OK", IDOK, 69, 123, 50, 14
PUSHBUTTON "Cancel", IDCANCEL, 125, 123, 50, 14
PUSHBUTTON "Help", IDHELP, 180,123, 50, 14
LTEXT "Person:", -1, 10, 12, 34, 8
LTEXT "City:", -1, 10, 32, 34, 8
LTEXT "Profession:", -1, 10, 53, 42, 8
CONTROL "", -1, "static", SS_BLACKFRAME ¦ WS_CHILD ¦ WS_VISIBLE, 9, 68, 47, 45
}
```

Figure 26. The Resource Script of the Employee Dialog

Figure 27 on page 34 shows the Object REXX script that you need to execute your dialog.

```
dlg = .PlainUserDialog~new
if dlg~InitCode <> 0 then exit
if dlg~Load("EMPLOYE1.RC", 100) ¬= 0 then exit
if dlg~Execute("SHOWTOP") = 1 then do
   say dlg~Name
   say dlg~City
   say dlg~Profession
   say dlg~Male
   say dlg~Female
   say dlg~Married
end
dlg~deinstall
::requires "OODPLAIN.CLS"
```

Figure 27. Simple Object REXX Script for the Employee Dialog (EMPLOYE1.REX)

The first line of the script instantiates an object of the predefined PlainUserDialog class and assigns the object to the symbol *dlg*. The second line checks for an initialization error which could be caused by not finding the dynamic-link library (DLL) that defines the external functions required by OODialog. The third line loads your resource script and creates a template for the Windows dialog. You do not have a Windows object yet. The Windows dialog is first created within the Execute method. After the loading, you can set the object attributes for the data items of your dialog. If you want to predefine values for the data items of your dialog before the dialog pops up, use PlainUserDialog methods to assign these values to the attributes. If you want to set the default value for the *City* combo box to "New York", for example, and the default sex to male, add the following three lines:

```
dlg~City = "New York"
dlg~Male = 1      /* radio button is selected */
dlg~Female = 0    /* radio button is not selected (default) */
```

after the Load method, where *City*, *Male*, and *Female* are the captions entered in the Resource Workshop. You can skip the comments if you want.

If the *AutoDetect* attribute is 1, which is the default for an instance of the PlainUserDialog class, an attribute is dynamically created within the object — not within the class — for each of the input items added to the dialog. This is done within the Load method. The attribute has the same name that was assigned to the dialog items in the *Caption* field, except that ampersands (&), colons, and blanks are filtered. If the caption is not a valid Object REXX symbol, the attribute is named *DATAx*, where *x* stands for the identification number of the dialog item.

If you use symbolic IDs, an attribute with the name of the ID is assigned to the dialog item. If, for example, a check box has the ID "CHECK_1", an attribute "self~CHECK_1" is assigned to the check box.

If you want to change the value of a dialog item, or retrieve the value, it is not sufficient to assign a new value to the attribute that belongs to the item, or retrieve the value of the attribute. This is because the data within the Object REXX object is separated from the data within the internal buffer of the Windows dialog. Each time an object attribute is changed, only the state of the object is modified, but not the state of the Windows dialog. To do that, exchange the data of the Object REXX object with the Windows dialog using the SetData method (see page 125). To get the data from the Windows dialog, use the GetData method (see page 125). The entire data of the Windows dialog is then copied to the Object REXX object and is accessible by the attributes.

There are several methods for exchanging only a single dialog item. See the methods beginning with Set or Get followed by the name of the dialog item type, such as SetEntryLine or GetRadioButton. If you want to set the check mark for the *Married* check box only, for example, and you know the ID of the dialog item, you can either call `dlg~SetCheckBox(41, 1)` or `dlg~SetValue(41, 1)`. To call `dlg~SetCheckBox(41, 1)` you must know the dialog item type. To call `dlg~SetValue(41, 1)`, the dialog item must have been registered, which is done automatically if *AutoDetect* is 1. If you do not know the dialog ID, you can assign 1 to the attribute *Married* and then call `dlg~SetAttrib("Married")`. This method copies the data from attribute `Married` to the assigned dialog item.

If you are using symbolic IDs and the check box has the ID "Check_1", for example, you can call `dlg~SetCheckBox("Check_1",1)` to set the check mark for that check box.

It is possible to use the DO OVER loop on a dialog object to loop through all the data attributes that are contained by the dialog object.

Deinstall in the last line of the program is used to remove the external functions needed for OODialog from the memory. It is a cleanup method.

Deinstall automatically detects whether there is still a dialog running on your system and only removes the external functions if they are really obsolete at the time of calling Deinstall.

Changing the Dialog Behavior

Most of the methods defined by OODialog are used to specify the behavior of the Windows dialog. The previous program does not implement a useful user interface. There is no selectable entry in either the list of the combo box or the list box. The list box and the combo box only make sense when list entries are selectable, otherwise you can replace them with a normal entry line. To fill these lists with strings, you can use the AddListEntry and AddComboEntry

methods. The following program extends the previous one to support list selection for the list box and the combo box as shown in Figure 28.

```
dlg = .MyDialogClass~new
if dlg~InitCode <> 0 then exit
if dlg~Load("EMPLOYE1.RC", 100) ¬= 0 then exit
if dlg~Execute("SHOWTOP") = 1 then do
   say dlg~Name
   say dlg~City
   say dlg~Profession
   say dlg~Male
   say dlg~Female
   say dlg~Married
end
dlg~deinstall
::requires "OODPLAIN.CLS"
::class MyDialogClass subclass PlainUserDialog
::method InitDialog
   self~City = "New York"
   self~Male = 1
   self~Female = 0
   self~AddComboEntry(22, "Munich")
   self~AddComboEntry(22, "New York")
   self~AddComboEntry(22, "San Francisco")
   self~AddComboEntry(22, "Stuttgart")
   self~AddListEntry(23, "Business Manager")
   self~AddListEntry(23, "Software Developer")
   self~AddListEntry(23, "Broker")
   self~AddListEntry(23, "Police Man")
   self~AddListEntry(23, "Lawyer")
```

Figure 28. Extended Dialog Using Subclassing (EMPLOYE2.REX)

In the program in Figure 28, *MyDialogClass* is defined, which is a subclass of the PlainUserDialog class. The definition of this class must begin directly after the *::requires* statement. *MyDialogClass* redefines one method called InitDialog which is already defined in the PlainUserDialog class. The InitDialog method is called by the PlainUserDialog class within the Execute method after the Windows object has been created and before it is displayed. This method is the best place to use the AddComboEntry and AddListEntry methods. All the other methods that directly manipulate Windows objects to initialize the dialog like SetEntryLine, SetListLine, SetListTabulators, or SetTitle, can also be called within InitDialog.

In the preceding code, the InitDialog method first sets the three attributes *City, Male, and Female* to default values and then puts four city names into the combo box specified by ID 22, and five professions into the list box specified by ID 23. The Object REXX object attributes are automatically exchanged with the Windows object within the Execute method before the dialog is displayed.

If you would have used symbolic IDs for the list box and the combo box you would have to specify the symbolic IDs instead of 22 and 23. In the previous example you would have to call *self~AddComboEntry("CB_CITY", "Munich")* or *self~AddListEntry("LB_PROF", "Broker")* given that you were using CB_CITY instead of 22 and LB_PROF instead of 23.

There are several other methods of working with list boxes and combo boxes. For a list box, there is AddListEntry InsertListEntry, DeleteListEntry, ChangeListEntry, FindListEntry, GetListEntry, GetCurrentListIndex, SetListTabulators, ListAddDirectory and ListDrop. For the combo box there are the same methods except that you have to use *Combo* instead of *List* for the method names, and there is no *SetComboTabulators*. Run this program to see the different behavior.

Enhance your dialog now by adding a Print button. You can give it any identification number that is greater than 9 and not already used for another item. The reason for the ID greater than 9 is that ID 1 and two are reserved for the OK and Cancel buttons. Both buttons terminate the dialog execution and call their corresponding methods. ID 9 is reserved for a Help button that calls the Help method. The IDs between 3 and 8 are reserved for other standard buttons. All IDs from 10 to 9999 are available.

When you execute the new dialog and press the Print button you will notice that nothing happens. To react on a button event, you must provide a method and connect it to the button using the ConnectButton method. See Figure 29 for an example that demonstrates how to implement a Print method that reacts on the Print button and displays the data of the employee dialog.

```
dlg = .MyDialogClass~new
if dlg~InitCode <> 0 then exit
if dlg~Load("EMPLOYE2.RC", 100) ¬= 0 then exit
dlg~Execute("SHOWTOP")
dlg~deinstall
::requires "OODPLAIN.CLS"
```

Figure 29. Providing a Method to Handle a Button Event (EMPLOYE3.REX) (Part 1 of 3)

```
::class MyDialogClass subclass PlainUserDialog
::method InitDialog
   self~City = "New York"
   self~Male = 1
   self~Female = 0
   self~AddComboEntry(22, "Munich")
   self~AddComboEntry(22, "New York")
   self~AddComboEntry(22, "San Francisco")
   self~AddComboEntry(22, "Stuttgart")
   self~AddListEntry(23, "Business Manager")
   self~AddListEntry(23, "Software Developer")
   self~AddListEntry(23, "Broker")
   self~AddListEntry(23, "Police Man")
   self~AddListEntry(23, "Lawyer")
   self~ConnectButton(10, "Print")  /* connect button 10 with a method */
```

Figure 29. Providing a Method to Handle a Button Event (EMPLOYE3.REX) (Part 2 of 3)

```
::method Print
   if self~Male = 1 then title = "Mr."; else title = "Ms."
   if self~Married = 1 then addition = " (married) "
   else addition = ""
   say title self~Name addition
   say "City:" self~City
   say "Profession:" self~Profession
```

Figure 29. Providing a Method to Handle a Button Event (EMPLOYE3.REX) (Part 3 of 3)

"Employe2.rc" is the same as "Employe1.rc" except for the Print button. In addition to the previous sample, the Print method is defined, and there is one more line, `self~ConnectButton(10, "Print")`, in the InitDialog method to connect the newly created button ID 10 with the Print method. Connect... methods only take effect when placed at the correct location. One possible location is the InitDialog method used in the previous example. Other possible locations will be discussed later. Another way to connect buttons with Object REXX methods is the "CONNECTBUTTONS" option, which can be passed to the Load method as the third parameter `dlg~Load("EMPLOYE2.RC",100,"CONNECTBUTTONS")`. If this option is specified, all buttons in the resource are connected to a method that is named after the caption of the button. Ampersands and blanks are filtered.

Because of the Print method, which displays the dialog data, the output processed by the program header is no longer needed. The output of the Print method is formatted so that the correct title depending on the sex is placed in front of the name and "(married)" is placed behind it, when appropriate. You can see that the output is displayed in the output window, provided by the workbench or in the console window, if started by the REXX command.

If you want to display the output in a message box, you can take one of the callable external functions provided by OODialog. Callable external functions are InfoMessage, ErrorMessage, YesNoMessage, GetScreenSize, GetFileNameWindow, PlaySoundFile, PlaySoundFileInLoop, StopSoundFile, SleepMs, and WinTimer. The function used for displaying the dialog data is InfoMessage as shown in Figure 30.

When you run "Employe3.rex", notice that Print method does not display the right values. This is because of the separation between dialog-internal data and the object attributes. To retrieve the data from the dialog within the corresponding object attributes, you must call GetData, which is done in Print method in Figure 30. GetData is also called implicitly when the dialog is closed with OK.

```
::method Print
    self~GetData
    if self~Male = 1 then title = "Mr."; else title = "Ms."
    if self~Married = 1 then addition = " (married) "
    else addition = ""
    call InfoMessage title self~Name addition || "A"x || "City:" ||,
                         self~City ||, "A"x ||,
                         "Profession:" self~Profession
```

Figure 30. Modified Print Method to Display Output in a Message Box (EMPLOYE4.REX)

InfoMessage takes one argument, which is a string that is displayed in the message box. The string passed to the function in the previous example is created from various variables and contains two line breaks that are specified by the hexadecimal value `'A'x`. The comma at the end of line 6 is an Object REXX line-continuation sign.

Dialog Data Validation

In several cases the data already entered or to be entered into the dialog data fields must conform to specific conditions. OODialog provides a predefined mechanism to check the dialog data for consistency. The Validate method is called when the OK button is pressed. The default implementation of Validate returns 1, which means that everything is all right and the dialog can be closed. If you override Validate you can change this and check whether the data entered conforms to your conditions. If not, you can return 0, which means that the dialog cannot be closed. The following listing shows the definition of Validate that denies the closing of the dialog if the name field for the employee is an empty entry line. See Figure 31 on page 40 for an example.

```
   ⋮

::method Validate
    if self~GetValue(21)~strip = "" then
    do
        call InfoMessage "An unnamed employee is not accepted!"
        return 0       /* dialog cannot be closed */
    end; else
        return 1       /* dialog can be closed */
```

Figure 31. Data Validation by Overriding Method Validate (EMPVALID.REX)

When you add these lines to the previous sample, the dialog can be closed only when the name field contains some data. Otherwise a message box pops up.

Advanced Dialog Programming

The following sections show how to implement an Object REXX program that can handle the data of more than one employee and allows the user to scroll through the different employees. This tutorial is not intended to deal with normal Object REXX programming, so the data will be stored in the memory and not to a file. The basis for this program is a dialog like the one shown in Figure 32.

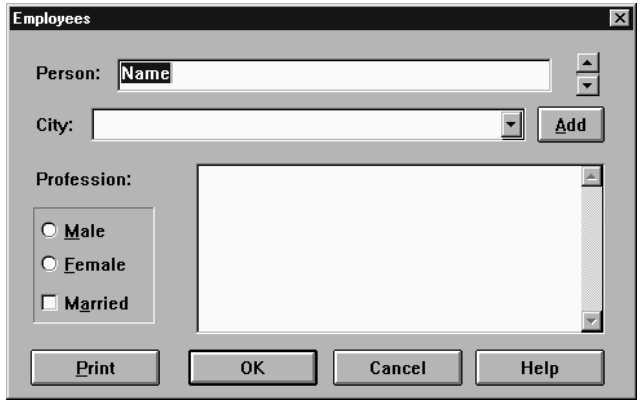


Figure 32. Extending the Dialog for Multidata Handling

If you do not want to modify the resource script yourself, you can take the "EMPLOYE3.RC" from the tutorial directory.

```
signal on any name CleanUp

dlg = .MyDialogClass~new
if dlg~InitCode <> 0 then exit
dlg~Execute("SHOWTOP")
dlg~deinstall
exit
/* -- signal handler to destroy dialog if condition was raised---*/
CleanUp:
   call ErrorMessage "Error" rc "occurred at line" sigl":",
        || errortext(rc), || "a"x || condition("o")~message
   if dlg~IsDialogActive then dlg~StopIt

::requires "OODPLAIN.CLS"

::class MyDialogClass subclass PlainUserDialog

::method Employees attribute
::method Emp_count attribute
::method Emp_current attribute
::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("EMPLOYE3.RC", 100)
    if ret = 0 then self~Employees = .array~new(10)
    self~Emp_count = 1
    self~Emp_current = 1
    self~ConnectButton(10, "Print")
                                /* connect button 10 with a method */
    self~ConnectButton(12, "Add")
                                /* connect button 12 with a method */
    self~InitCode = ret
    return ret
```

Figure 33. Handling the Add Button (EMPLOYE5.REX) (Part 1 of 2)

```
::method InitDialog
    self~City = "New York"
    self~Male = 1
    self~Female = 0
    self~AddComboEntry(22, "Munich")
    self~AddComboEntry(22, "New York")
    self~AddComboEntry(22, "San Francisco")
    self~AddComboEntry(22, "Stuttgart")
    self~AddListEntry(23, "Business Manager")
    self~AddListEntry(23, "Software Developer")
    self~AddListEntry(23, "Broker")
    self~AddListEntry(23, "Police Man")
    self~AddListEntry(23, "Lawyer")
::method Print
    self~GetData
    if self~Male = 1 then title = "Mr."; else title = "Ms."
    if self~Married = 1 then addition = " (married) "
                              else addition = ""
    call InfoMessage(title self~Name addition ¦¦ "A"x ¦¦,
                          "City:" self~City ¦¦ "A"x ¦¦
                          "Profession:" self~Profession)
::method Add
    self~Employees[self~Emp_count] = .directory~new
    self~Employees[self~Emp_count]['NAME'] = self~GetValue(21)
    self~Employees[self~Emp_count]['CITY'] = self~GetValue(22)
    self~Employees[self~Emp_count]['PROFESSION'] = self~GetValue(23)
    if self~GetValue(31) = 1 then sex = 1; else sex = 2
    self~Employees[self~Emp_count]['SEX'] = sex
    self~Employees[self~Emp_count]['MARRIED'] = self~GetValue(41)
    self~Emp_count = self~Emp_count +1
    self~Emp_current = self~Emp_count
    self~SetValue(21, "");
::method Set
    self~SetValue(21, self~Employees[self~Emp_current]['NAME'])
    self~SetValue(22, self~Employees[self~Emp_current]['CITY'])
    self~SetValue(23, self~Employees[self~Emp_current]['PROFESSION'])
    if self~Employees[self~Emp_current]['SEX'] = 1 then do
       self~SetValue(31, 1);self~SetValue(32, 0); end
    else do
       self~SetValue(31, 0);self~SetValue(32, 1); end
    self~SetValue(41, self~Employees[self~Emp_current]['MARRIED'])
```

Figure 33. Handling the Add Button (EMPLOYE5.REX) (Part 2 of 2)

The first line of the program is new. "`signal on any name CleanUp`" is used to install a signal handler that handles runtime errors. The signal handler itself is defined after the *CleanUp* label in line 8. This label is called when a condition was raised that would cause the program to be terminated. When an OODialog program is executed and the dialog is active when the error condition is raised, the program is interrupted without deleting the Windows object. When you run your program from a command console with `REXX` this is no problem because the dialog is deleted by the closed process. When you

run the program from the workbench, the dialog is still up after the Object REXX program is interrupted, but it does not react to an event. This is because the control program – the Object REXX program you wrote – was interrupted by an error or failure condition. The code defined for label *CleanUp* displays the Object REXX error message in a message box and then deletes the Windows object before the Object REXX program is terminated. You can take this code and copy it for all your other OODialog programs.

To handle the data of more than one employee the `MyDialogClass` class defines three attributes. `Employees` is assigned to an array that is created in the Init method and stores the data fields for each employee. `Emp_count` is to hold the number of employees stored in the `Employees` array and `Emp_current` stores the array index of the currently displayed employee.

In addition to the previous example, the two `Add` and `Set` methods are defined. The `Add` method is called each time the Add button (with ID 12) is pressed. The first line in the `Add` method assigns a directory object to the array slot for the current Employee. The directory object holds the 5 entries NAME, CITY, PROFESSION, SEX, and MARRIED. The values for these entries are directly taken from the Windows dialog via the GetValue, which expects the ID of the dialog item that the data is required from. The entry SEX is either 1 if the “Male” or 2 if the “Female” radio button is selected. The `Set` method is defined to prepare the next step. It copies the entries of the directory object at the `Emp_current` index directly to the Windows dialog by using the SetValue method which expects the ID of the dialog item where the data is to be copied to, and a value.

The next step provides the necessary extensions to browse through the stored employees by using the scroll bar, as shown in Figure 34 on page 44.

```
::requires "OODIALOG.CLS"
   :
   :
::class MyDialogClass subclass UserDialog
   :
   :

::method InitDialog
   :
   :
self~ConnectScrollBar(11, "Emp_Previous", "Emp_Next")

   :
   :
::method Add
   :
   :
    self~SetSBRange(11, 1, self~Emp_count)
    self~SetSBPos(11, self~Emp_count)
::method Emp_Previous
   if self~Emp_count = 1 then return
   if self~Emp_current > 1 then do
       self~Emp_current = self~Emp_current - 1
       self~SetSBPos(11, self~Emp_current)
       self~Set
   end; else
       call TimedMessage "You have reached the top!","Info",1000
::method Emp_Next
   if self~Emp_count = 1 then return
   if self~Emp_current < self~Emp_count-1 then do
       self~Emp_current = self~Emp_current + 1
       self~SetSBPos(11, self~Emp_current)
       self~Set
   end; else
       call TimedMessage "You have reached the bottom!","Info",1000
```

Figure 34. Handling Scroll Bar Events (EMPLOYE6.REX)

This program excerpt shows the additional lines of code that are required to handle the events of the scroll bar. Because scroll bars are not supported by the PlainUserDialog, the UserDialog class must be subclassed, which extends the PlainUserDialog by several more methods. Because the UserDialog is defined within OODIALOG.CLS, this file must be requested. The advantage of splitting the functionality of OODialog into these two classes and into two files is to save memory when you only need the functionality provided by the PlainUserDialog or any of the Standard Dialogs. In the InitDialog method, the UP event of the scroll bar with ID 11 is connected to method `Emp_Previous`, and the DOWN event is connected to method `Emp_Next` via the ConnectScrollBar method. The `Emp_Previous` method decreases the attribute `Emp_current` and calls the `Set` method to copy the values of the directory object to the Windows dialog. The `Emp_Next` method does the same except that `Emp_current` is increased. Both methods check if the top or the bottom is

reached. The SetSBPos method is called to set the position of the scroll bar. The SetSBRange method is called within `Add` to extend the range of the scroll bar each time an employee is added.

TimedMessage is one of the standard dialog functions that creates an object of TimedMessage class and executes it.

Nesting Dialogs

Nesting dialogs allows you to execute a dialog within another running dialog. OODialog has a maximum nesting level of 10 dialogs.

Instantiate a new dialog object, create the Windows dialog, and execute it. OODialog does the dialog management for you. Running a child dialog from a parent dialog causes the parent dialog to be disabled, which means that you can no longer access control items of this dialog. This behavior is called *application-modal*. The dialog is disabled automatically. The usual way is to execute the newly created child dialog with *SHOWTOP*, which causes the dialog to be the topmost window. After the dialog is finished, the parent dialog is enabled again automatically, but becomes the topmost window only after you called the ToTheTop method in the parent dialog. To make sure that the parent dialog is not locked by an unsuccessful execution of the nested dialog, you can invoke the Enable method before the parent dialog receives control again. The following example includes nested dialogs.

Formatted Lists

In the following example a new button is added to the employee dialog that is named “List Employees”. When you press this button, a second dialog pops up and lists all the employees that are currently stored in the memory. The list contains the title, the name, the profession, and the city. The “List Employees” button should only be enabled when at least one employee is stored in memory. The example is shown in Figure 35 on page 46.

```
101 DIALOG 6, 15, 278, 144
STYLE DS_MODALFRAME ¦ WS_POPUP ¦ WS_CAPTION ¦ WS_SYSMENU
CAPTION "List of Employees"
FONT 8, "System"
{
 DEFPUSHBUTTON "OK", IDOK, 226, 127, 50, 14
 CONTROL "List", 101, "LISTBOX",
        LBS_NOTIFY ¦ WS_BORDER ¦ WS_BORDER ¦ WS_VSCROLL, 3, 16, 272, 103
 LTEXT "Name", -1, 5, 7, 26, 8
 LTEXT "Profession", -1, 101, 7, 60, 8
 LTEXT "City", -1, 201, 7, 60, 8
}
```

Figure 35. Resource Definition of the Employee List Dialog

```
   :

::method Init
   :
        self~ConnectButton(13, "Emp_List")
   :


::methodInitDialog
   :
    self~DisableItem(11)
    self~DisableItem(13)

   :
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 1 of 6)

```
::method Add
   :
    self~EnableItem(11)
    self~EnableItem(13)

   :

::method Emp_List
   ldlg = .EmployeeListClass~new(self)
   ldlg~Execute("SHOWTOP")
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 2 of 6)

```
::method FillList
   use arg subdlg, id
   column1 = 24;column2 = 29
   do i = 1 to self~Emp_count-1
       if self~Employees[i]['SEX'] = 1 then
       title = "Mr."; else title = "Ms."
       addstring = title self~Employees[i]['NAME']
       spacebetween = column1 - self~Employees[i]['NAME']~length - 5
       if spacebetween > 0 then addstring = addstring,
       || " "~copies(spacebetween)
       addstring = addstring || " "self~Employees[i]['PROFESSION']
       spacebetween = column2 - self~Employees[i]
       ['PROFESSION']~length - 5
       if spacebetween > 0 then addstring = addstring,
       || " "~copies(spacebetween)
       addstring = addstring || " "self  ~Employees[i]['CITY']
       subdlg~AddListEntry(id, addstring)
   end
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 3 of 6)

```
::class EmployeeListClass subclass UserDialog

::method parent attribute
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 4 of 6)

```
::method Init
   use arg ParentDlg
   self~parent = ParentDlg
   ret = self~init:super
   if ret = 0 then ret = self~Load("EMPLOYE5.RC", 101)
   self~InitCode = ret
   return ret
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 5 of 6)

```
::method InitDialog
   self~parent~FillList(self, 101)
   font = self~CreateFont("Courier", 10)
   self~SetItemFont(101, font)
```

Figure 36. Nesting a List Dialog (EMPLOYE8.REX) (Part 6 of 6)

Figure 35 on page 46 shows the resource definition of the dialog containing the list. Figure 36 on page 46 shows the parts of the `MyDialogClass` definition that are new, and the definition of the `EmployeeListClass`. In addition to the

previous sample, dialog 100 contains the "List Employees" button instead of the Help button. This button (ID 13) is connected to method `Emp_List` in the Init method. The two additional lines in the InitDialog method are to initially disable the "List Employees" button and the scroll bar. Disabled dialog items are grayed. At the end of the `Add` method the disabled dialog items are enabled again, because after `Add` was called once, an employee is stored in memory and therefore the list can be displayed and the scroll bar can be used to display data of the employee.

`EmployeeListClass` defines the new attribute `Parent` which stores the OODialog object that is passed on to the Init method. Init also calls the method of its superclass and loads the dialog resource 101. Notice that both dialog resources are stored in one .RC file.

The `Emp_List` method is called when the "List Employees" button is pressed. `Emp_List` instantiates an object of class `EmployeeListClass` and passes on `self` to the Init method. `Emp_List` then calls the Execute method of this dialog instance to execute the dialog. Before this dialog pops up, the InitDialog method of `EmployeeListClass` is automatically called. In the InitDialog method, the `FillList` method of the Parent object is called. `Parent` was set to *self* of `MyDialogClass` within the `Emp_List` method and therefore the `FillList` method of `MyDialogClass` is called. Arguments for these methods are *self* of the `EmployeeListClass` object and the ID of the list box.

`FillList` processes all employees stored in memory and formats a data string that is added to the list by calling AddListEntry. In this method blanks are inserted between the data fields so that the columns start at the same character position. If you were to run this sample without the two last program lines – within the InitDialog method – you could see that the data fields are not in a row when pressing the "List Employees" button. This is because the default font for the list box is the same as for the dialog, and that is a proportional font. Therefore, in the InitDialog method, a monospaced font (Courier) is created by calling CreateFont and assigned to the list box by calling SetItemFont. In this way the data fields are lined up.

Instead of using a monospaced font, you can also use a list box that has the "Multicolumn" and the "Tab stops" options checked. The following excerpt shows the differences when using tab stops instead of a monospaced font and blanks. See Figure 37 on page 49 for details.

```
101 DIALOG 6, 15, 278, 144
   :
 CONTROL "List", 101, "LISTBOX", LBS_NOTIFY ¦ WS_BORDER ¦
                LBS_USETABSTOPS ¦ LBS_MULTICOLUMN ¦ WS_BORDER ¦
                 WS_VSCROLL, 3, 16, 272, 103
   :

   :
```

Figure 37. Resource Definition of a Multicolumn List

```
::method FillList
   use arg subdlg, id
   do i = 1 to self~Emp_count-1
       if self~Employees[i]['SEX'] = 1 then title = "Mr.";
                    else title = "Ms."
       addstring = title self~Employees[i]['NAME']
       addstring = addstring ¦¦ "9"x ¦¦ self~Employees[i]['PROFESSION']
       addstring = addstring ¦¦ "9"x ¦¦ self~Employees[i]['CITY']
       subdlg~AddListEntry(id, addstring)
   end
```

Figure 38. Formatting the List Data Using Tab Stops (EMPLOYE9.REX) (Part 1 of 2)

```
::class EmployeeListClass subclass UserDialog

   :

::method InitDialog
   self~parent~FillList(self, 101)
   self~SetListTabulators(101, 98, 198)
```

Figure 38. Formatting the List Data Using Tab Stops (EMPLOYE9.REX) (Part 2 of 2)

In the InitDialog method, the tabulator stops are set to 98 and 198 (in dialog units) for the list box by calling SetListTabulators. This causes the `FillList` method to be smaller because no data field length calculation must be done but a tabulator character ("9"x) is needed to separate the columns. Tabulator formatting works with proportional fonts and monospaced fonts.

Using Menus within Your Dialogs

OODialog provides several methods that allow to use menu resources within your dialog or to create menus dynamically and add them to the dialog.

You can use the Resource Workshop to create a menu for your dialog by selecting menu item *Resource – New* and choosing MENU as the type for the

resource. If you close the dialog with OK the menu editor appears in the Resource Workshop. In the menu editor you can add the popup menus and menu items that your menu shall contain. Remember to rename the menu resource to a numerical ID. OODialog does not allow the definition of accelerator keys for the menu items although the Resource Workshop supports this.

To load a menu from the resource script, you can use LoadMenu. You can either add more menu items to the loaded menu or set the menu to the dialog by calling SetMenu. SetMenu adds the menu to an existing Windows dialog. Therefore the right place to call SetMenu is the InitDialog method because InitDialog is automatically called after the Windows object was created and before the dialog pops up. For an example on how to use menus within dialogs, see Figure 39.

```
   ⋮

::method Init
   ret = self~init:super;
   if ret = 0 then ret = self~Load("EMPLOYE7.RC", 100)
   if ret = 0 then self~Employees = .array~new(10)
   if ret = 0 then do
       self~Emp_count = 1
       self~Emp_current = 1
       self~ConnectButton(10, "Print")
       self~ConnectButton(12, "Add")
       self~ConnectButton(13, "Emp_List")
       if self~LoadMenu("EMPLOYE7.RC", 200) = 0 then do
           self~ConnectMenuItem(201, "Add")
           self~ConnectMenuItem(202, "Print")
           self~ConnectMenuItem(203, "Emp_List")
           self~ConnectMenuItem(204, "About")
       end
   end
   self~InitCode = ret
   return ret
```

Figure 39. Adding a Menu Resource to a Dialog (EMP_MENU.REX) (Part 1 of 2)

```
::method InitDialog
   ⋮
    self~SetMenu

::method Add
   ⋮
    self~EnableMenuItem(203)

   ⋮

::method About
   call InfoMessage "This is a sample About message!"
```

Figure 39. Adding a Menu Resource to a Dialog (EMP_MENU.REX) (Part 2 of 2)

The first part of the Init method is the same as for all the previous samples except that the dialog resource is loaded from `EMPLOYE7.RC`. The reason for this is, that the size of the dialog must be larger than before because all dialog items are moved downward when the menu is added to the top of the dialog. The highlighted statements are necessary to load and connect the menu resource. `LoadMenu("EMPLOYE7.RC", 200)` loads the menu resource 200 from the resource script. If LoadMenu was successful (return value = 0), the menu item IDs are connected to Object REXX methods by using ConnectMenuItem. The menu items “Exit” and “Cancel” have not been connected to any method because the IDs of the two menu items are 1 and 2 and these events are already by default connected with the OK and Cancel methods. If you use for a menu item the same ID as for a button, you must connect either the item or the button with a method because they send the same event if selected or pressed.

At the end of the InitDialog method, SetMenu is called to add the loaded menu resource to the dialog. SetMenu must be called from InitDialog because the Windows object must exist if you use SetMenu. The `Add` method has been overridden to enable menu item “List”, which is initially grayed. The `About` method is called when the “About” menu item is selected and gives you information about the author of this tutorial.

In the following sample the same menu is added to the dialog, but this time no resource is defined. Instead all the popup menus, separators, and menu items that the menu consists of, are added dynamically. See Figure 40 on page 52.

```
   :
::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("EMPLOYE7.RC", 100)
    if ret = 0 then self~Employees = .array~new(10)
    if ret = 0 then do
        self~Emp_count = 1
        self~Emp_current = 1
        self~ConnectButton(10, "Print")
        self~ConnectButton(12, "Add")
        self~ConnectButton(13, "Emp_List")
        self~ConnectMenuItem(201, "Add")
        self~ConnectMenuItem(202, "Print")
        self~ConnectMenuItem(203, "Emp_List")
        self~ConnectMenuItem(204, "OK")
        self~ConnectMenuItem(205, "Cancel")
        self~ConnectMenuItem(206, "About")
   end

   self~InitCode = ret
   return ret
```

Figure 40. Adding a Dynamically Created Menu to a Dialog (EMP_MEND.REX) (Part 1 of 2)

```
::method InitDialog
   :
    self~SetMenu
::method DefineDialog
    forward class(super) continue
    self~CreateMenu
    self~AddPopupMenu("&Employees")
    self~AddMenuItem("&Add", 201)
    self~AddMenuItem("&Print", 202)
    self~AddMenuSeparator
            /* last item in popup */
    self~AddMenuItem("&List", 203, "GRAYED END")
            /* last popup in menu */
    self~AddPopupMenu("&Control", "END")
    self~AddMenuItem("E&xit", 204)
    self~AddMenuItem("Cancel", 205)
    self~AddMenuSeparator
            /* last item in popup */
    self~AddMenuItem("&About", 206, "END")
```

Figure 40. Adding a Dynamically Created Menu to a Dialog (EMP_MEND.REX) (Part 2 of 2)

The Init method is similar to the previous one. ConnectMenuItem is used to connect the menu items with a method. The “Exit” and “Cancel” menu items must be connected because these two menu items are created with ID 204 and 205, which are not handled by default. There is no *LoadMenu* method because the menu resource is to be created dynamically. A good place for the dynamic

creation of a dynamic control is the DefineDialog method, which is overridden in this sample to define the menu.

To start the creation of a menu you must call CreateMenu. Then you can add popup menus by using AddPopupMenu, separators by using AddMenuSeparator, and menu items by calling AddMenuItem. The menu components are added in the same order as these methods are called. The END option must be specified for every last menu item in a popup menu and for the last popup menu in the menu. The option GRAYED is used for the "List" menu item, which causes this menu item initially to be displayed grayed. The `Add` and `About` methods are equal to those in the previous sample. See also "DisableMenuItem" on page 175, "CheckMenuItem" on page 175, "UncheckMenuItem" on page 175, "GrayMenuItem" on page 175, "SetMenuItemRadio" on page 176, and "GetMenuItemState" on page 176.

Creating Graphics with OODialog

If you want to make your dialogs more attractive, you can display a bitmap as the background of a dialog or use your own, so-called owner-drawn, graphical push buttons. *Owner-drawn* means that you can use the area reserved for the push button to draw your own graphics, display bitmaps, or write text. All graphic methods require a device context for the button, which is a memory for graphics.

To use a bitmap as the background you need the method BackgroundBitmap (see page 165) or TiledBackgroundBitmap (see page 164), depending on whether you want to show one bitmap or tile the entire dialog background with a bitmap. To provide a bitmap button for owner-drawn bitmaps you use the ConnectBitmap method (see page 112).

The source code excerpt in Figure 41 on page 54 shows the statements necessary to display a bitmap button instead of the "Add" button and to have the Object REXX logo displayed in the background.

```
  ⋮

::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("EMPLOYE8.RC", 100)
    if ret = 0 then self~Employees = .array~new(10)
    if ret = 0 then do
        self~Emp_count = 1
        self~Emp_current = 1
        self~ConnectButton(10, "Print")
                                  /* connect button 10 with a method */
        self~ConnectBitmapButton(12, "Add", "add.bmp",,,,"FRAME")
        self~BackgroundBitmap("logo.bmp")
    end
    self~InitCode = ret
    return ret
```

Figure 41. Using Bitmaps (EMPLOYE7.REX)

The two highlighted lines provide a bitmap button and a background bitmap. The drawing and refreshing of bitmap buttons and the background bitmap is handled by OODialog. The rest of the graphic methods are so-called "top window" methods, because the drawing is not displayed persistently. Each time a window is drawn upon the button and then removed, the repainted area is erased and not restored with the original drawing.

The following example demonstrates how to use the graphic methods together with the device context methods to display a circle in a dialog.

```
   ⋮

::class MyDialogClass subclass UserDialog

::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("Drawings.RC", 100)
    self~ConnectButton(11, "Circle")
    self~ConnectButton(12, "MyRectangle")
    self~InitCode = ret
    return ret

::method Circle
    dc = self~GetButtonDC(10)
    pen = self~CreatePen(5, "SOLID", 1)
    oldpen = self~ObjectToDc(dc, pen)
    self~DrawArc(dc, 10, 10, 320, 200)
    self~ObjectToDc(dc, oldpen)
    self~DeleteObject(pen)
    self~FreeButtonDC(10, dc)
```

Figure 42. Display a Drawing in the Dialog (DRAWING1.REX) (Part 1 of 2)

```
::method MyRectangle
    dc = self~GetButtonDC(10)
    pen = self~CreatePen(8, "SOLID", 2)
    oldpen = self~ObjectToDc(dc, pen)
    self~Rectangle(dc, 10, 10, 320, 200)
    self~ObjectToDc(dc, oldpen)
    self~DeleteObject(pen)
    self~FreeButtonDC(10, dc)
```

Figure 42. Display a Drawing in the Dialog (DRAWING1.REX) (Part 2 of 2)

This program shows the definition of a class that connects the two buttons 11 and 12 with the methods `Circle` and `MyRectangle`. The name “MyRectangle” was chosen because OODialog already defines a method Rectangle. Naming the method that is connected to button 12 “Rectangle” would cause an endless loop.

The Init method loads the resource script and connects the button. The first line in method `Circle` calls GetButtonDC to retrieve the device context of the owner-drawn button. The return value `DC` is required by the other methods to connect the graphic operations with the appropriate dialog item. In the second line a pen is created by calling CreatePen. A pen is used to draw pixels and lines (the outline part of a graphic). The other two objects that can be used in a device context are a brush, which is used to draw the filling part of a graphic, and a font. ObjectToDC is used to load the newly created pen into the device context. The return value of `ObjectToDC` is the old object that was

used before. In this case, this is the pen that was assigned to the device context of the owner-drawn button before loading the new one.

The next line does the drawing. For the method `Circle` DrawArc is used, which draws an ellipse or part of it. In the method `MyRectangle`, Rectangle is used to draw the rectangle. After the drawing the old pen is assigned back to the device context. Because you no longer need the pen, you can delete it using DeleteObject. The last method, FreeButtonDC, is the counterpart of GetButtonDC. Each time you call a "Get..DC" method you should also call the "Free..DC" method to release the device context again. If you do not free the device context, your Windows system might eventually run out of resources.

The overhead of using a graphic method seems to be large, but once you have retrieved your device context and created and assigned your graphic objects, it is easy to draw various graphic elements. When you run the program you should also move another window onto the dialog. If you remove that window again from the top of the dialog, your graphic disappears as mentioned before. The next sample shows how to make graphics within an owner-drawn button persistent. See Figure 43.

```
   ⋮

::class MyDialogClass subclass UserDialog

::method GraphicObject attribute

::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("Drawings.RC", 100)
    self~ConnectButton(11, "Circle")
    self~ConnectButton(12, "MyRectangle")
    self~GraphicObject = "NONE"
    if ret = 0 then self~ConnectDraw(10, "DrawIt")
    self~InitCode = ret
    return ret
```

Figure 43. Making Graphics Persistent (DRAWING2.REX) (Part 1 of 4)

```
::method DrawIt
    if self~GraphicObject = "NONE" then return 0
    dc = self~GetButtonDC(10)
    if self~GraphicObject = "CIRCLE" then do
        size = 5
        color = 1
        x = 60
    end; else do
        size = 8
        color = 2
        x = 20
    end;
```

Figure 43. Making Graphics Persistent (DRAWING2.REX) (Part 2 of 4)

```
    pen = self~CreatePen(size, "SOLID", color)
    oldpen = self~ObjectToDc(dc, pen)
    font = self~CreateFont("Arial", 24, "BOLD ITALIC")
    oldfont = self~FontToDC(dc, font)
    self~TransparentText(dc)
    if self~GraphicObject = "CIRCLE" then
        self~DrawArc(dc, 10, 10, 300, 200)
    else
        self~Rectangle(dc, 10, 10, 320, 200)
    self~WriteDirect(dc,x,100,self~GraphicObject)
    self~FontToDC(dc, oldfont)
    self~DeleteFont(font)
    self~ObjectToDc(dc, oldpen)
    self~DeleteObject(pen)
    self~OpaqueText(dc)
    self~FreeButtonDC(10, dc)
    return 1
```

Figure 43. Making Graphics Persistent (DRAWING2.REX) (Part 3 of 4)

```
::method Circle
    self~GraphicObject = "CIRCLE"
    self~RedrawButton(10, 1)

::method MyRectangle
    self~GraphicObject = "RECTANGLE"
    self~RedrawButton(10, 1)
```

Figure 43. Making Graphics Persistent (DRAWING2.REX) (Part 4 of 4)

Figure 43 on page 56 shows not only the parts that were necessary to make persistent graphics, but also how to write text into the device context with a user-defined font. The two drawing methods of the previous sample have

been combined into one method. The methods `Circle` and `MyRectangle` that handle the button events now only set the new attribute `GraphicObject` to either “CIRCLE” or “RECTANGLE”.

The Init method contains two additional lines. The first one initializes attribute `GraphicObject` to “NONE” and the second one connects method `DrawIt` with the Windows message *WM_DRAWITEM* to get graphic persistence. Message WM_DRAWITEM is automatically sent each time a dialog item must be drawn or redrawn.

ConnectDraw expects two arguments. The first is the ID of the button that the device context uses to draw the graphic, and the second is the name of the method for processing the drawing instructions. If you omit the button ID, the specified method is called for all owner-drawn buttons contained by the current dialog. This means that, in this case, it makes no difference whether or not the ID is specified because this dialog contains only one owner-drawn button.

Method `DrawIt` manipulates the device context. It loads a user-defined font into the device context of the button by calling FontToDC, which is similar to ObjectToDC. The user-defined font was created using CreateFont. It also displays text in the drawing area of the button using WriteDirect. `FontToDC` returns the old font — like `ObjectToDC`. Load the old font back into the device context after the write statements were processed. The counterpart to `CreateFont` is DeleteFont which frees the Windows resource allocated by calling CreateFont.

The TransparentText and OpaqueText methods used to set the text mode. Figure 44 helps to clarify the difference between the two modes.

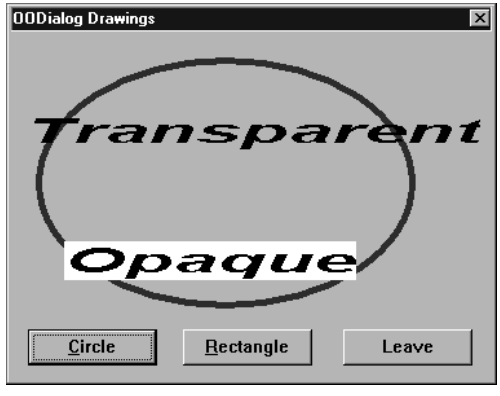


Figure 44. The Two Text Modes: Transparent and Opaque

Scrolling Text and Bitmaps

The ScrollInButton method displays a given text with a given font within an owner-drawn button from right to left. Figure 45 shows how to use this method.

```
   ⋮

::class MyDialogClass subclass UserDialog

::method Init
    ret = self~init:super;
    if ret = 0 then ret = self~Load("Textscrl.RC", 100)
    self~fontname = "Arial"
    self~text = "This is a scrolling text demonstration"
    self~fontsize = 24
    self~ConnectButton(11, "Display")
    self~InitCode = ret
    return ret


::method Display
    self~GetData
    self~ScrollInButton(10, self~text, self~fontname,,
                             self~fontsize,"BOLD")
```

Figure 45. Scrolling Text from Right to Left (TEXTSCRL.REX)

The beginning of the program equals all previous examples. A signal handler is installed and the instance is created and executed. The user-defined class only overrides the Init method and defines one additional method. In Init, the default values for the dialog data items are assigned (this is possible only after Load is called because the attributes are automatically added to the object within Load) and button 11 is connected to method `Display`. Method `Display` calls GetData to retrieve the values from the dialog data item and calls the ScrollInButton method to scroll the text stored in `self~text` from right to left within the owner-drawn button 10 in the font that is specified by `self~fontname` and `self~fontsize`.

OODialog also allows you to move bitmaps within owner-drawn buttons. To get more information on this topic, see "ScrollBitmapFromTo" on page 164 and "DisplaceBitmap" on page 165 and look at the `OODGRAPH.REX` example in the `OODIALOG\SAMPLES` directory. `OODGRAPH.REX` executes a dynamically created dialog that was composed by calling UserDialog methods.

More about Event Handling

OODialog provides a list of methods that help handling dialog events or dialog commands, such as ConnectButton and ConnectBitmapButton.

ConnectScrollBar can be used to catch the events of a scroll bar and route them to different methods. Another way of catching scroll bar events is by calling ConnectAllSBEvents which causes all scroll bar events to be routed to one method. In this context it is also recommended to get familiar with DetermineSBPosition and ConnectELwithSB. ConnectControl can be used to catch events from any dialog control like a radio button, a combo box, or even an entry field.

ConnectList catches the event of a list box. The corresponding method is called each time the list selection changes.

ConnectListLeftDoubleClick can be used to catch the double-click event within a list box, ConnectDraw catches the WM_DRAWITEM message. If you need to catch any other event, you can use AddUserMsg to connect the specified Windows message number with a class method. To use AddUserMsg you must know the hexadecimal value of the message to be caught. You can specify filters to determine whether the values of the `WPARAM` and `LPARAM` arguments must be equal to the specified ones. Within the method that is connected to all events, you can use the "Use Arg wParam lParam" statement to retrieve the additional message values. As an example, you can connect method `DrawIt` with an owner-drawn button. If method `DrawIt` is defined as follows:

```
   ⋮

::method DrawIt
use arg ID
   ⋮
```

`ID` is set to the button ID that is to be drawn.

All other OODialog "Connect..." methods are to connect object attributes with dialog data items.

Summary of User Dialog Processing

Figure 46 on page 61 describes in which way the methods of a UserDialog object must be called to create and execute a Windows dialog.

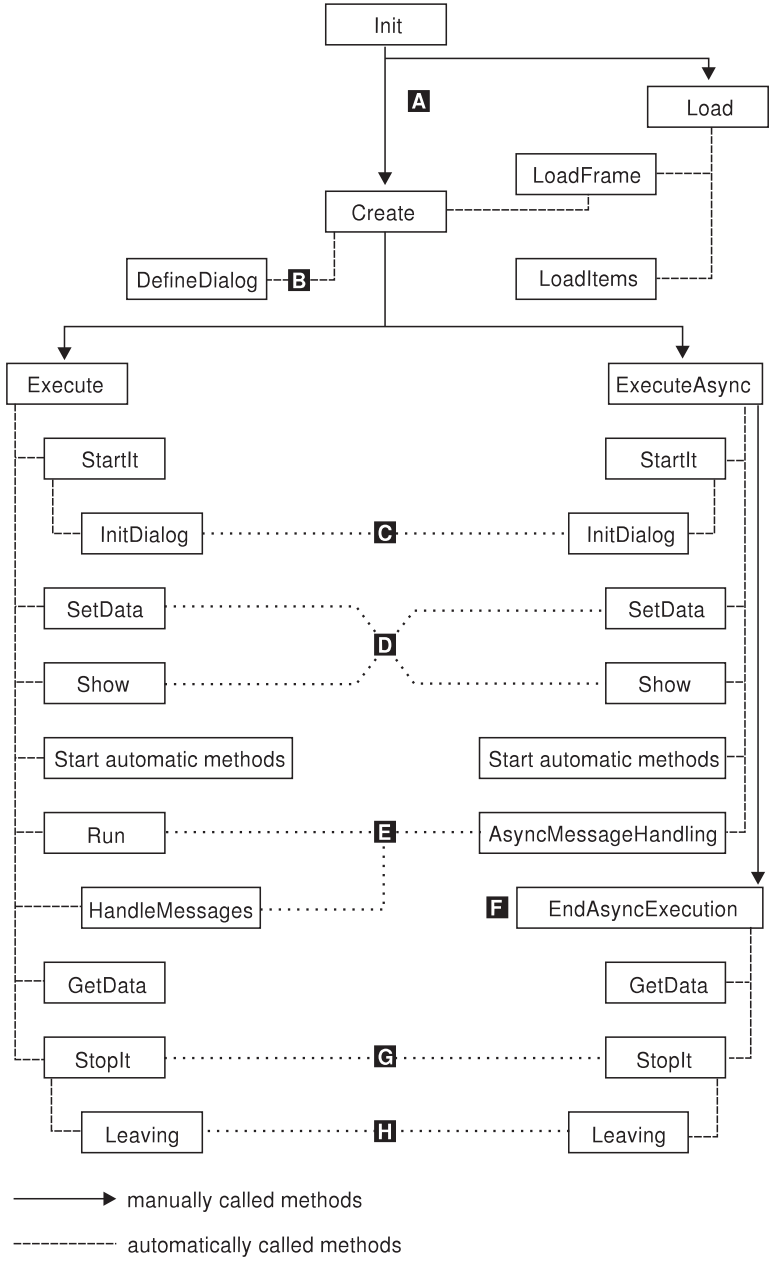


Figure 46. How the Methods Work Together

The first method called is Init. This is done automatically by the new class method when an object is instantiated. Once the object has been created, the user can choose (**A**) between defining the dialog manually – using the *Create*

method – and loading the dialog from a resource script – using the Load method. The choice is indicated by solid lines.

If you want to load the dialog from a resource script, the Load method first calls LoadFrame and then LoadItems. LoadItems can also be used for a CategoryDialog to load all dialog items from a dialog resource into a category page. LoadFrame, however, calls Create, which means that choosing to load the dialog ends in the same branch as choosing to define the dialog manually.

The Create method allocates memory for a dialog template to which the dialog items can be added. The DefineDialog method (**B**), which is called by Create, is the right place to add items to the dialog. If you choose to define the dialog manually, UserDialog should be subclassed and all Add... messages (AddText, AddEntryLine, AddButton) be placed into DefineDialog. The new attributes are added to the OODialog object by LoadItems or the Add... methods. Therefore, you can assign values to dialog items after DefineDialog is executed. You can choose whether to send the Execute message or the ExecuteAsync message after Create has been processed. At this time, no Windows dialog exists, only a dialog template that contains all information about the dialog's appearance.

Once you send the Execute or ExecuteAsync message, the StartIt method is called. Method Startit creates a real Windows dialog and calls InitDialog (**C**). The InitDialog method is the right place to fill the combo boxes and list boxes – using AddListEntry or AddComboEntry – and to invoke other methods that deal with real dialog items (Windows objects) like SetItemFont, DisableItem, or GetButtonDC. All Connect... methods, except the ConnectScrollBar and ConnectAllSBEvents methods, can also be placed in this method, but the Init method is the recommended place because these Connect... methods do not deal with Windows objects.

However, ConnectScrollBar and ConnectAllSBEvents must be specified within InitDialog because they deal with a real Windows object.

In the Execute and ExecuteAsync methods, the next method called is SetData to transfer the data from the attributes – or the stem if given – to the Windows dialog. This implies that modifying a dialog data item with a method that directly deals with the dialog item like SetCurrentListIndex or SetRadioButton within InitDialog is obsolete because the dialog items are reset by SetData with the values that are either stored in the data stem or in the corresponding object attributes.

After this, all the methods that were added by using AddAutoStartMethod are executed asynchronously. This feature is for animated buttons. Next, the dialog is displayed by sending Show (**D**).

At this point there is a difference between Execute and ExecuteAsync (**E**). For Execute dialogs, the Run method is invoked next, to dispatch the messages until the user closes the dialog. For ExecuteAsync dialogs the AsyncMessageHandling message is started to handle the incoming messages and dispatch them to the dialog object. Because this method is executed asynchronously, ExecuteAsync returns while the dialog is still up.

In this way it is possible to let your Object REXX program continue processing while the dialog is up and waiting for user interaction.

When the dialog is closed with OK (ID=1), the data is transferred from the Windows dialog to the OODialog object through GetData, and the dialog is removed from memory by StopIt. A second Execute is not possible. For ExecuteAsync, EndAsyncExecution (**F**) must be called manually to wait until the user closes the dialog and to transfer the data to the object.

After StopIt (**G**) was called, method Leaving (**H**) is invoked to allow you to do any dialog post-processing. After StopIt is called, you can no longer use methods that deal with Windows objects. But you still have access to the attributes of the dialog object.

Chapter 4. Other OODialog Classes

There is more than one way to execute your dialogs with OODialog. Until now, the UserDialog and PlainUserDialog classes have been used, which are required when the dialogs are stored in a resource script. In some of the OODialog samples other classes are used that are briefly described in the following sections.

The ResDialog Class

The ResDialog class is used when the dialogs are stored in a dynamic-link library (DLL). If you save your resources as a .RES file in binary format, which you can do with the Resource Workshop, you can link this file to a DLL. Windows provides functions to load resources directly from a DLL, to be used by the ResDialog class. The advantage of using DLL resources is that they are faster. Because you can also store bitmaps in the DLL, you no longer need to handle a large number of files, and your dialog resources are better protected.

To link a binary resource file to a DLL, OODialog includes the IBM VisualAge® C++ linker, ILINK. This linker lets you create a DLL from a RES file. The linker must be executed within a command window. Here is the syntax to invoke the linker:

```
ilink MyDialog.res /DLL -out:MyDialog.dll
```

This command creates `MYDIALOG.DLL` by converting the binary resource file `MYDIALOG.RES`. To minimize the keystrokes, you can also use `MAKEDLL.BAT`, which contains the same command. To create the same DLL, just enter `MAKEDLL MyDialog` at the command prompt. The names of the DLL and the binary resource file are the same, except for a different extension.

Whether you use binary resources stored in a DLL or resource scripts depends on what you want to do. If your application uses dialogs with many bitmap buttons, or if you use a lot of complex dialogs, it is better to store the resources in a DLL so you have one file instead of many .BMP and .RC files. If your application is not too big, resources stored in a DLL are easy to link and modify. Note, however, that bitmaps stored in a DLL do not support different color palettes. To support different color palettes, you must use bitmap files.

The following list shows the differences between the UserDialog class and the ResDialog:

- The Init method needs two additional arguments, the name of the DLL and the ID of the dialog resource. For example, to instantiate a dialog object that can execute dialog 100, which is stored in `MYDLG.DLL`, use the following statement:

  ```
  dlg = .ResDialog~Init("MYDLG.DLL", 100)
  ```

- You must not call Load or Create.
- The DefineDialog method is not called.
- You can pass the resource ID of bitmaps stored in your DLL to ConnectBitmapButton and ChangeBitmapButton instead of the file names or bitmap handles.

All other methods of the ResDialog class are equal to those of the UserDialog class.

The CategoryDialog Class

The *CategoryDialog* class is a subclass of the UserDialog class. It lets you use more than one dialog within one window and is comparable to the OS/2® notebook. Figure 47 on page 67 gives you an example.

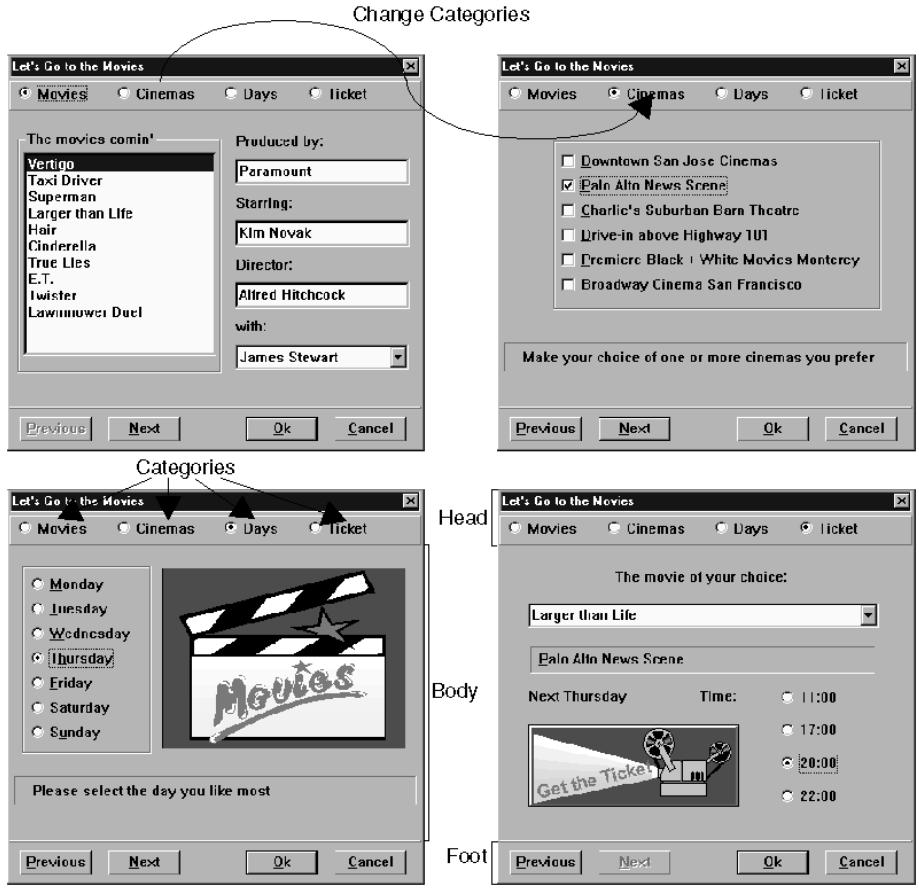


Figure 47. A Category Dialog

The dialogs displayed are the four pages of the `OOTICKET.REX` sample in the `OODIALOG\SAMPLES` directory. To move to a different page, select the corresponding radio button. The dialog is split into three parts, which are named head, body, and foot. Only the body must be defined by you, the head and foot are created by OODialog itself. The Object REXX instructions to execute this category dialog are shown in Figure 48.

```
   :
dlg = .TicketDialog~new(data.,,4,,"TOPLINE")
if dlg~InitCode ¬= 0 then do; say "Init did not work"; exit; end

dlg~createcenter(200, 180, "Come to the movies !")
dlg~execute("SHOWTOP")
```

Figure 48. Execution of a Category Dialog

The first line instantiates an object of TicketDialog, which is a subclass of the CategoryDialog. The initialization data is passed to the dialog through the stem data. The dialog consists of four categories and is of the style TOPLINE.

After InitCode has been checked, the dialog template (see "Summary of User Dialog Processing" on page 60) is created by using CreateCenter. Using Create or CreateCenter means that no resource script is used to retrieve the layout of the dialog. This is why the size of the dialog must be passed to CreateCenter in dialog units, not in screen pixels. The third argument of CreateCenter is the title of the dialog. The dialog must be large enough to fit all the dialog pages, including the head and the foot, which are added automatically.

The individual dialog pages can be defined dynamically or by using the Resource Workshop. You cannot use the Resource Workshop to define the category dialog itself. Figure 49 shows the layout of a CategoryDialog.

```
::class TicketDialog subclass CategoryDialog

   ⋮

::method InitCategories
   self~catalog['names'] = .array~of("Movies", "Cinemas", "Days", "Ticket")
   ⋮
::method Days
   self~AddText(10,self~SizeY - 65,0,0, "Please select the day you like most")
   self~AddRadioGroup(31, 5, 5,0, "&Monday  &Tuesday &Wednesday",
       "&Thursday &Friday &Saturday &Sunday")
   self~AddBlackRect(1, self~SizeY -68, self~SizeX -6, 14)
   self~AddBitmapButton(145, 73, 10, 125, 100, ,,"bmp\movie.bmp")
::method Ticket
   self~loaditems("ticket.rc")
   self~connectBitmapButton("", 45,"bmp\ticket.bmp",,,,"FRAME USEPAL")
```

Figure 49. Defining the Layout of a CategoryDialog

InitCategories is one of the CategoryDialog methods that must be overridden. To use CategoryDialog you must subclass it. InitCategories is used to specify which categories are used. The information must be assigned to the Object REXX directory object catalog. The entry names in this directory must be assigned to an array containing the names of the categories.

The DefineDialog method is called by Create to add dialog items to the dialog template. You must define a dialog for each category. A method with the same name as the category is called to define the particular dialog page.

For example, the Days method is called to define the dialog page that is displayed when the Days radio button is selected. The Ticket method is called

to define the `Ticket` page. This means that the define methods have the same name as the category names that are added to the array in `catalog['names']`.

In the `Days` method the dialog items are added manually. The `Add...` methods need the positions, in dialog units, where the dialog items are to be located. These methods enable you to create dialogs without using the Resource Workshop. Some of them add a whole group of items, which is much faster than adding the items with the Resource Workshop.

If you look at the `Ticket` method, LoadItems is called with the name of a resource script. Because the CategoryDialog is created with Create and the pages contain only dialog items, it is not necessary to retrieve the dialog frame, only the items of the dialog, which is done by LoadItems.

It is also possible to load items from a resource script and then manually add more. Once you have defined your dialog pages, tasks such as creating the dialog, switching the pages, and transferring the data, are handled by OODialog. Data is handled in the same way as in the UserDialog class. Object attributes are added to all data items. It is also possible to use a stem variable to set or retrieve the dialog values. All data items of all pages are copied from or into one stem.

To summarize, to use a CategoryDialog you must to subclass it, override InitCategories with a method that creates an array containing the category names and assign it to `catalog['names']`. In addition, you must define the methods to load or add the dialog items to the pages using method names that match the names stored in the array.

Because the dialog items are spread over more than one dialog, all of the methods that must communicate with real Windows objects, such as GetValue, SetValue, AddComboEntry and FindListEntry, must know on which page the corresponding dialog is. Therefore you must use the CategoryDialog methods and add the dialog page number starting with 1. Some of the CategoryDialog methods are used in the next example.

```
   ⋮
   ⋮
::method InitDialog
   expose films
   self~InitDialog:super
   films = .array~of("Disclosure","Bad Boys","Drop Zone","Twister",
            ,"Hair","Cinderella","True lies",
             "Nasty guy in your bathroom","Grassmower")

   do i = 1 to 9
      self~addCategoryListEntry(31, films[i], 1)
   end
   self~addCategoryComboEntry(35,"Jack Nicholson"            , 1)
   self~addCategoryComboEntry(35,"Jackie's Girls"            , 1)
   self~addCategoryComboEntry(35,"Linda Moore"               , 1)

   self~setCategoryStaticText(42, date(), 4)
   self~setCategoryStaticText(43, time(), 4)
   self~setCategoryStaticText(44, "Charlie's Suburb Barn Theatre", 4)
   ⋮
   ⋮
```

Figure 50. Examples of CategoryDialog Methods (Part 1 of 2)

```
::method changePage
   expose films
   NewPage = self~GetSelectedPage
   if (NewPage = 4) then do
     self~ChangePage:super(NewPage)
     self~CategoryComboDrop(41, 4)
     Lines = self~getCategoryValue(31, 1)
     do while Lines ¬= ''
       parse var Lines Line Lines
       self~addCategoryComboEntry(41, films[Line] ,4)
     end
     self~setCategoryComboLine(41, films[line] ,4)
     end
   else
     self~ChangePage:super(NewPage)
```

Figure 50. Examples of CategoryDialog Methods (Part 2 of 2)

The methods have the same name as their equivalent in the UserDialog class with the prefix `Category` added. The arguments are the same except that it is also necessary to determine the page on which the dialog item is located.

When defining the resource scripts or the layout definition methods, ensure that you do not use the same item ID twice. All methods that allow you to manually add a group of dialog items expect a starting ID for the first item, which is increased for each item. The other methods are the same as for the UserDialog class.

Another sample to demonstrate the use of CategoryDialog is EM_CATEG.REX in the OODIALOG\TUTORIAL directory. This example combines the employee input dialog and the employee list dialog of the previous examples in one category dialog. See Figure 51 for details.

```
dlg = .EmployeeDialog~new(,,,,"TOPLINE WIZARD")
if dlg~InitCode ¬= 0 then do; say "Dialog init did not work"; exit; end
dlg~createcenter(280, 160, "Employee Dialog")
   :

::class EmployeeDialog subclass CategoryDialog
   :

::method InitDialog
   :
   self~AddCategoryComboEntry(22, "Munich", 1)
   :
   self~AddCategoryListEntry(23, "Business Manager", 1)
   :
   self~DisableCategoryItem(44, 1)
   self~SetCategoryListTabulators(101, 98, 198, 2)
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 1 of 6)

```
::method InitCategories
   self~catalog['names'] = .array~of("Input", "List")
        /* set the width of the button row at the bottom to 35 */
   self~catalog['page']['btnwidth'] = 35
        /* change name of wizzard buttons,
                    default is &Backward and &Forward */
   self~catalog['page']['leftbtntext'] = "&Input"
   self~catalog['page']['rightbtntext'] = "&List"
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 2 of 6)

```
::method Input                                          /* page 1 */
   self~loaditems("em_categ.rc", 100)
   self~ConnectButton(40, "Print")
   self~ConnectButton(42, "Add")
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 3 of 6)

```
::method List                                          /* page 2 */
   self~loaditems("em_categ.rc", 101)

   :

::method Add
   :
     self~Employees[self~Emp_count]['NAME'] = self~GetCategoryValue(21, 1)
   :
     self~EnableCategoryItem(44, 1)
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 4 of 6)

```
   :

::method FillList
   use arg id
   do i = 1 to self~Emp_count-1
       if self~Employees[i]['SEX'] = 1 then title = "Mr."; else title = "Ms."
       addstring = title self~Employees[i]['NAME']
       addstring = addstring ¦¦"9"x ¦¦self~Employees[i]['PROFESSION']
       addstring = addstring ¦¦"9"x ¦¦self~Employees[i]['CITY']
       self~AddCategoryListEntry(id, addstring, 2)
   end
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 5 of 6)

```
::method PageHasChanged
   NewPage = self~CurrentCategory
   if NewPage = 1 then do
      self~Emp_current = self~GetCurrentCategoryListIndex(101, 2)
      if self~Emp_current > 0 then do
         self~SetSBPos(44, self~Emp_current)
         self~Set
      end
   end
   else do
      self~CategoryListDrop(101, 2)
      self~FillList(101)
   end
```

Figure 51. Code Excerpt of EM_CATEG.REX (Part 6 of 6)

In the first line, the category dialog is instantiated with the `TOPLINE` and the `WIZARD` options. CreateCenter is used to create a dialog in the given size and center it on the screen. The third argument is the dialog caption.

In the InitCategories method the category catalog is set. Two categories, Input and List, are added and the page layout of the wizard dialog is modified. Method `Input` is automatically called by CreateCenter to define the layout for the first page and `List` is called to define the second page. Both methods call LoadItems to load the items from the `EM_CATEG.RC` file. In `Input`, the two buttons "Add" and "Print" are connected to the corresponding methods. Method `Print` has not changed, while in the `Add` method GetCategoryValue and EnableCategoryItem are called, where the second argument specifies the category page. Method `Set` has changed such that SetCategoryValue is used. `Emp_Previous` and `Emp_Next` have not changed.

Method `FillList` changed because AddCategoryListEntry must be used instead of AddListEntry and the argument `subdlg` is no longer needed.

The last method, `PageHasChanged` is an overridden method that is automatically called each time the category page has changed. In this method the algorithm that is processed depends on the newly selected page. If page 2 is selected, the list must be created or refreshed. If page 1 is selected, the data entry that is currently selected in the list is displayed in the input dialog.

Chapter 5. Tokenizing OODialog Scripts

Larger REXX scripts should be tokenized to improve the performance. Make sure that your GUI scripts using OODialog are tokenized as well. Otherwise, the load time for your interactive programs can be very long. You can tokenize your programs by using `REXXC.EXE`.

The required OODialog files are in the internal format to minimize the time required to parse them. The `SCRIPTS` subdirectory contains the source scripts to make some modifications. After you have changed the source files, you must execute `REXX BUILD` to tokenize the modified source scripts. Notice that you must not name your modified OODialog classes `OODIALOG.CLS`, `OODPLAIN.CLS`, or `OODWIN32.CLS`.

`BUILD.REX` creates one file containing all the classes defined by the OODialog source scripts.

Chapter 6. OODialog External Functions

OODialog provides the following callable functions that can be used in your Object REXX programs.

InfoMessage

Displays an information message window:

```
call InfoMessage "some message text"
ret = InfoMessage("another text")
```

ErrorMessage

Displays an error message window:

```
call ErrorMessage "some error message text"
ret = ErrorMessage("another error message")
```

YesNoMessage

Displays a message and ask the user for a YES or NO answer:

```
ret = YesNoMessage("press Yes or No")
if ret=1 then /* this is yes */
```

GetScreenSize

Queries the monitor size in dialog units and pixels:

```
val = GetScreenSize()
parse var val dunitx dunity pixelx pixely
```

PlaySoundFile

Plays a sound file (.WAV):

```
call PlaySoundFile "d:\wav\sound.wav"
ret = PlaySoundFile("d:\wav\sound.wav","YES")
```

The optional second parameter YES plays the file asynchronously, that is the program continues execution. See also the routine “Play” on page 177.

PlaySoundFileInLoop

Plays a sound file (.WAV) continuously and asynchronously:

```
call PlaySoundFileInLoop "d:\wav\sound.wav"
```

StopSoundFile

Stops playing an asynchronous sound file (.WAV):

```
call StopSoundFile
```

GetFileNameWindow

Displays an Open File window:

```
file = GetFileNameWindow(filename,handle,filter,loadorsave,title,,
                         defExtension,multiSelect,sepChar)
```

Parameters (optional):

filename
A preselected name.

handle
The parent window handle.

filter
A file mask specification.

loadorsave
If you specify 1, which is the default, you get the file name for a load operation. If you specify 0, you get the file name for a save operation.

title
The window title. The default is "Open a File" or "Save File As", depending on what you specify for *loadorsave*.

defExtension
The default extension that is added if no extension was specified. The default is TXT.

multiSelect
If you specify 1, you can select several files. In this case, *loadorsave* must also be 1. The result is then *path file1 file2 file3 ....*

If you specify 0 or omit this parameter, you get the selected file name or an empty string when the Open File window is canceled.

sepChar
Specifies which character should be used for separating the file names when *multiSelect* = 1.

This is needed for file names with blank characters. If this argument is omitted, the separation character is a blank. If the argument is specified, the returned path and file name uses this separation character. For example, if you specify "#" as the separation character, the return string might look as follows:

```
C:\WINNT#file with blank.ext#fileWithNoBlank.TXT
```

Example:

```
"Text files (*.txt)"||'0'x||"*.TXT"||'0'x|| ,
"All files (*.*)"||'0'x||"*.*"
```

SleepMS

Sleeps for a given time interval, in milliseconds:

```
call SleepMS(3000) /* 3 seconds */
```

WinTimer

Starts, stops, and waits for a windows timer:

```
tid = WinTimer("START",300)   /* 0,3 seconds */
call  WinTimer("WAIT,tid)     /* wait... */
ret = WinTimer("STOP",tid)    /* stop premature */
```

OODialog functions are registered automatically when the first dialog is initialized. If no dialog has been created, register individual functions with:

```
call RxFuncAdd functionname, "OODialog", functionname
```

To register all OODialog functions:

```
call RxFuncAdd InstMMFuncs, "OODialog", InstMMFuncs
call InstMMFuncs
```

A more convenient way to call these functions is provided by the "Public Routines" on page 176.

The standard dialog classes can also be executed as callable functions. These functions are described with their respective classes in "Chapter 14. Standard Dialog Classes and Functions" on page 293.

Part 2. OODialog Method Reference

The classes provided by OODialog form a hierarchy as shown in Figure 52.

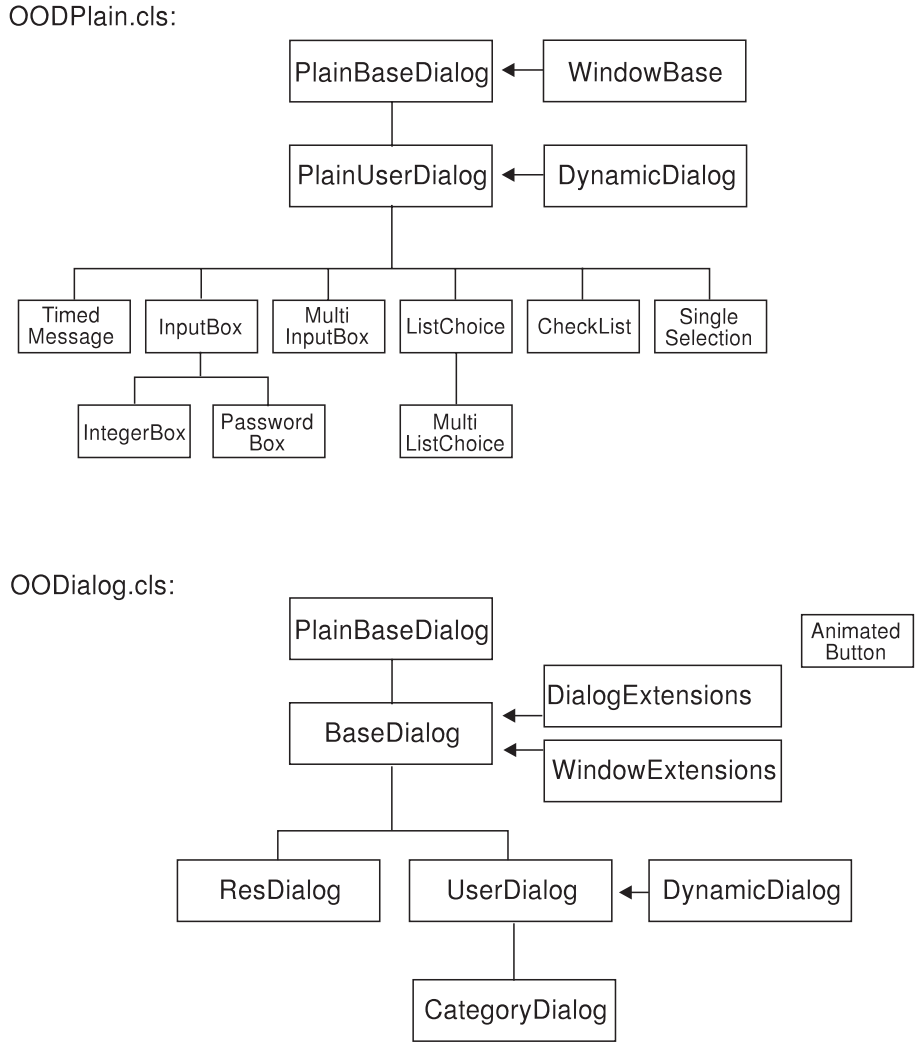


Figure 52. The Hierarchy of OODialog Classes (Part 1 of 3)

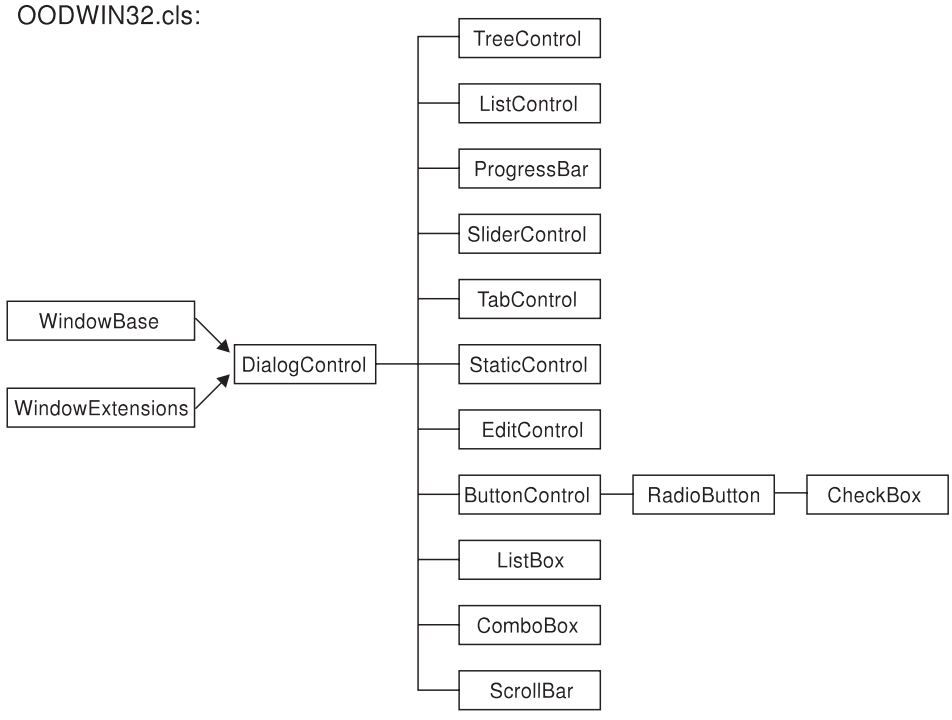


Figure 52. The Hierarchy of OODialog Classes (Part 2 of 3)

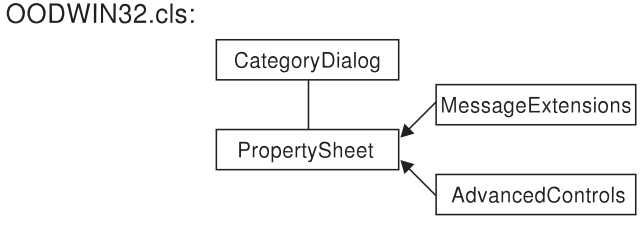


Figure 52. The Hierarchy of OODialog Classes (Part 3 of 3)

The classes are:

PlainBaseDialog, BaseDialog
Base methods regardless of whether the dialog is implemented as a binary resource, a script, or dynamically. PlainBaseDialog provides limited functionality.

PlainUserDialog
Subclass of PlainBaseDialog used to create a dialog with all its control elements or to execute a dialog stored in a resource script (.RC). This class has limited functionality.

DynamicDialog, DialogExtensions, WindowBase, WindowExtensions
Internal mixin classes used to extend PlainBaseDialog, PlainUserDialog, BaseDialog, UserDialog, and DialogControl. The methods provided by these classes are not listed separately but are listed in BaseDialog or UserDialog.

UserDialog
Subclass of BaseDialog used to create a dialog with all its control elements, such as push buttons, check boxes, radio buttons, entry lines, and list boxes.

ResDialog
Subclass of BaseDialog for dialogs within a binary (compiled) resource file (.DLL).

CategoryDialog
Subclass of UserDialog used to create a dialog with several pages that overlay each other.

TimedMessage
Class to show a message window for a defined duration.

InputBox
Class to dynamically define a dialog with a message, one entry line, and two push buttons (OK, Cancel).

PasswordBox
Similar to InputBox, but keystrokes in the entry line are shown as asterisks (*).

IntegerBox
Similar to InputBox, but only numeric data can be entered in the entry line.

MultiInputBox
Similar to InputBox, but with multiple entry lines.

ListChoice
Class to dynamically define a dialog with a list box, where one line can be selected and returned to the caller.

MultiListChoice
Similar to ListChoice, but more than one line can be selected and returned to the caller.

CheckList
Class to dynamically define a dialog with a group of check boxes, which can be selected and returned to the caller.

SingleSelection
Class to dynamically define a dialog with a group of radio buttons, where one can be selected and returned.

Dialog
Subclass of UserDialog for simple dialogs. You can change the default dialog style from UserDialog to ResDialog.

AnimatedButton
Class to implement an animated button within a dialog.

DialogControl
Class to implement methods that are common to all dialogs and dialog controls.

TreeControl
Class to implement a tree to display the list of items in a dialog in a hierarchy.

ListControl
Class to implement a list view to display the items in a dialog as a collection.

ProgressBar
Class to implement a progress indicator within a dialog.

SliderControl
Class to implement a slider or trackbar within a dialog.

TabControl
Class to implement tabs, which can be compared to dividers in a notebook or labels in a file cabinet.

StaticControl
Class to query and modify static controls, such as static text, group boxes, and frames.

EditControl
Class to query and modify edit controls, which are also called entry lines.

ButtonControl
Class to implement push buttons within a dialog.

RadioButtonControl
Class to implement radio buttons within a dialog.

CheckBoxControl
Class to implement check boxes within a dialog.

ListBoxControl
Class to implement list boxes within a dialog.

ComboBoxControl
Class to implement a combo box, which combines a list box with an edit control.

ScrollBarControl
Class to implement a scroll bar within a dialog.

PropertySheetControl
Class to implement a property sheet, which is similar to a category dialog that spreads its dialog items over several pages (categories), where the individual pages are controlled by a tab control instead of radio buttons or combo box lists.

Chapter 7. Definition of Terms

id The identification number of a dialog item. An ID is assigned by the user the dialog item is created using the Resource Workshop or dynamically. IDs 1, 2, and 9 are reserved for the OK, Cancel, and Help push buttons. An ID can be either numerical (for example, 100) or symbolic (for example, "Bankaccount_Entry").

handle
A unique reference to a Windows object assigned by the system. It can be a reference to a dialog, a particular dialog item, or a graphic object (pen, brush, font). Handles are required for certain methods; they can be retrieved from the system when needed.

device context
Stores information about the graphic objects that are displayed, such as bitmaps, lines, and pixels, and the tools used to display them, such as pens, brushes, and fonts. A device context can be acquired for a dialog or a button; it must be explicitly freed when the text or graphic operations are completed.

pixel Individual addressable point within a window. VGA screens support 640 by 480 pixels, SVGA screens support higher resolutions, such as 800 by 600, 1024 by 768, 1280 by 1024, and 1600 by 1200. Pixel values in a dialog start at the top left corner and include the window title and border.

dialog unit
Used within dialog box templates to define the size and position of the dialog box and its controls. There is a horizontal and a vertical dialog base unit to convert width and height of dialog boxes and controls from dialog units to pixels and vice versa. The value of these base units depend on the screen resolution and the active system font; they are stored in attributes of the UserDialog class.

```
xPixels = xDialogUnits * self~FactorX
```

color Each color supported by the Windows operating system is assigned a number. Sample color indexes are 0 (black), 1 (dark red), 2 (dark green), 3 (dark yellow), 4 (dark blue), 5 (purple), 6 (blue grey), 7 (light grey), 8 (pale green), 9 (light blue), 10 (white), 11 (grey), 12 (dark grey), 13 (red), 14 (light green), 15 (yellow), 16 (blue), 17 (pink), 18 (turquoise).

color palette

An array that contains color values identifying the colors that can currently be displayed or drawn on the output device.

Color palettes are used by devices that can generate many colors but can only display or draw a subset of them at a time. For such devices, Windows maintains a system palette to track and manage the current colors of the device.

Applications do not have direct access to this system palette. Instead, Windows associates a default palette with each device context. Applications can use the colors in the default palette.

The default palette is an array of color values identifying the colors that can be used with a device context by default. Windows associates the default palette with a context whenever an application creates a context for a device that supports color palettes. The default palette ensures that colors are available for use by an application without any further action. The default palette typically has 20 entries (colors), but the exact number of entries can vary from device to device. The colors in the default palette depend on the device. Display devices, for example, often use the 16 standard colors of the VGA display and 4 other colors defined by Windows.

Chapter 8. BaseDialog Class

The *BaseDialog* class implements base methods for all dialogs regardless of whether the dialog is implemented as a binary resource, a resource script, or created dynamically. Binary (compiled) resources are stored in a DLL. A dialog is created dynamically by using Add... methods. Dialogs that are implemented using a resource script (.RC) are generated semi-dynamically.

BaseDialog is an abstract class. You cannot use it to execute a Windows dialog but have to use one of its subclasses.

See the subclasses in "Chapter 10. UserDialog Class" on page 227 and "Chapter 12. ResDialog Class" on page 269 for additional information.

Requires:

`BaseDlg.cls` is the source file of this class. Use the tokenized version of OODialog, `oodialog.cls`, to shorten the dialog startup time.

```
::requires oodialog.cls
```

Attributes:

Instances of the BaseDialog class have the following attributes:

AutoDetect
Automatic data field detection on (=1, default) or off (=0). For the UserDialog subclass the default is off and Connect... methods or a resource script are usually used.

AutomaticMethods
A queue containing the methods that are started concurrently before the execution of the dialog.

BkgBitmap
The handle to a bitmap that is displayed in the dialog's background.

BkgBrushBmp
The handle to a bitmap that is used to draw the dialog's background.

ConstDir
A directory containing the numerical values assigned to symbolic IDs (#define-statements in the resource script).

DataConnection
A protected attribute to store connections between dialog items and the attributes of the dialog instance.

DlgHandle
The handle to the dialog.

Finished
0 if dialog is executing, 1 if terminated with OK, and 2 if canceled.

InitCode
The result of the Init method. If Init fails, its value is 1.

IsExtended
A protected attribute that is true (=1) if the graphics extension is installed.

UseStem
A protected attribute that is true (=1) if a stem variable was passed to Init.

Routines:
See "Public Routines" on page 176 for a description of the audio Play routine.

Methods:
Instances of the *BaseDialog* class implement the methods listed in Table 2.

Table 2. BaseDialog Instance Methods

Method...	...on page
AbsRect2LogRect	200
AddAttribute	124
AddAutoStartMethod	171
AddComboEntry	135
AddListEntry	142
AddUserMsg	122
AssignWindow	186
AsyncMessageHandling	106
AutoDetection	109
BackgroundBitmap	165
BackgroundColor	152
Cancel	134
CaptureMouse	207
Center	158
ChangeBitmapButton	162

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
ChangeComboEntry	139
ChangeListEntry	145
CheckMenuItem	175
Clear	197
ClearButtonRect	161
ClearMessages	107
ClearRect	161
ClearWindowRect	162
ClientToScreen	201
CombineELwithSB	149
ComboAddDirectory	139
ComboDrop	140
ConnectAllSBEvents	121
ConnectAnimatedButton	172
ConnectBitmapButton	112
ConnectButton	111
ConnectCheckBox	117
ConnectComboBox	117
ConnectControl	114
ConnectDraw	114
ConnectEntryLine	116
ConnectList	115
ConnectListBox	118
ConnectListLeftDoubleClick	115
ConnectMenuItem	174
ConnectMouseCapture	111
ConnectMove	110
ConnectMultiListBox	118
ConnectPosChanged	110
ConnectRadioButton	117
ConnectResize	109
ConnectScrollBar	119

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
CreateBrush	219
CreateFont	216
CreatePen	220
Cursor_AppStarting	205
Cursor_Arrow	205
Cursor_Cross	206
Cursor_No	206
CursorPos	203
Cursor_Wait	206
DeInstall	135
DeleteComboEntry	136
DeleteFont	217
DeleteListEntry	142
DeleteObject	221
DetermineSBPosition	149
Disable	191
DisableItem	153
DisableMenuItem	175
DisplaceBitmap	165
Display	192
Draw	197
DrawAngleArc	225
DrawArc	223
DrawBitmap	163
DrawButton	159
DrawLine	222
DrawPie	225
DrawPixel	223
Dump	180
Enable	190
EnableMenuItem	174
EnableItem	153

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
EndAsyncExecution	102
Execute	100
ExecuteAsync	101
FillDrawing	225
FindComboEntry	137
FindListEntry	143
FocusItem	152
FontColor	218
FontToDC	218
ForegroundWindow	194
FreeButtonDC	167
FreeDC	210
FreeWindowDC	167
Get	150
GetArcDirection	224
GetAttrib	131
GetBitmapSizeX	162
GetBitmapSizeY	163
GetBmpDisplacement	165
GetButtonDC	166
GetButtonRect	151
GetClientRect	189
GetCheckBox	130
GetComboEntry	137
GetComboItems	137
GetComboLine	129
GetCurrentComboIndex	138
GetCurrentListIndex	145
GetData	125
GetDataStem	133
GetDC	209
GetEntryLine	126

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
GetFocus	190
GetID	187
GetItem	151
GetListEntry	143
GetListItemHeight	144
GetListItems	144
GetListLine	127
GetListWidth	140
GetMenuItemState	176
GetMouseCapture	207
GetMultiList	128
GetPixel	223
GetPos	151
GetRadioButton	129
GetRect	187
GetSBPos	148
GetSBRange	147
GetSize	189
GetTextSize	215
GetValue	130
GetWindowDC	166
GetWindowRect	152
GrayMenuItem	175
HandleMessages	106
HScrollPos	202
Help	134
Hide	191
HideFast	191
HideItem	153
HideItemFast	154
HideWindow	154
HideWindowFast	155

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
Init	98
InitAutoDetection	108
InitDialog	98
IsMouseButtonDown	208
Leaving	135
InsertComboEntry	136
InsertListEntry	142
IsDialogActive	104
ItemTitle	126
ListAddDirectory	146
ListDrop	146
LoadBitmap	208
LogRect2AbsRect	199
Maximize	193
Minimize	193
Move	195
MoveItem	158
NoAutoDetection	108
ObjectToDC	220
OK	133
OpaqueText	212
PeekDialogMessage	107
Popup	102
PopupAsChild	103
Rectangle	221
Redraw	198
RedrawClient	199
RedrawButton	160
RedrawRect	159
RedrawWindow	156
RedrawWindowRect	160
ReleaseMouseCapture	207

Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
RemoveBitmap	209
Resize	193
ResizeItem	157
RestoreCursorShape	204
Run	99
ScreenToClient	200
Scroll	201
ScrollBitmapFromTo	164
ScrollButton	170
ScrollInButton	170
ScrollText	168
SendMessageToItem	107
SetArcDirection	224
SetAttrib	132
SetCheckBox	130
SetComboLine	129
SetCursorPos	203
SetCurrentComboIndex	138
SetCurrentListIndex	145
SetData	125
SetDataStem	132
SetEntryLine	126
SetFocus	190
SetFont	216
SetHScrollPos	202
SetVScrollPos	203
SetItemFont	170
SetListColumnWidth	141
SetListItemHeight	144
SetListLine	127
SetListWidth	141
SetListTabulators	146

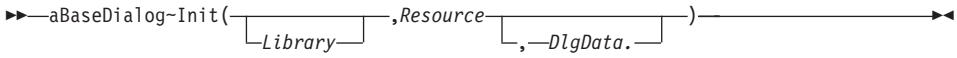
Table 2. BaseDialog Instance Methods (continued)

Method...	**...on page**
SetMenuItemRadio	176
SetMultiList	128
SetRadioButton	130
SetRect	188
SetSBPos	148
SetSBRange	147
SetStaticText	126
SetTitle	197
SetValue	131
SetWindowRect	155
SetWindowTitle	159
Show	105
ShowFast	191
ShowItem	154
ShowItemFast	154
ShowWindow	155
ShowWindowFast	155
StopIt	104
TiledBackgroundBitmap	164
Title	196
Title=	196
TransparentText	212
ToTheTop	105
UncheckMenuItem	175
Update	196
Validate	134
VScrollPos	202
Write	167
WriteDirect	212
WriteToButton	214
WriteToWindow	213

Preparing and Running the Dialog

This section presents the methods used to prepare and initialize a dialog, show it, run it, and stop it.

Init



The constructor of the class installs the necessary C functions for the Object REXX API manager and prepares the dialog management for a new dialog.

Protected:

This method is protected. You cannot create an instance of BaseDialog. You can only create instances of its subclasses.

Arguments:

The arguments are:

Library

The file name of a .DLL file. Pass an empty string if you do not use binary resources.

Resource

The ID, or the symbolic name, of the dialog within the resource file.

DlgData.

A stem variable (remember the period!) that contains initialization data for the dialog. For example, if you assign the string "Hello world" to `DlgData.103`, where `103` is the ID of an entry field, it is initialized with this string. If the dialog is terminated with OK, the data of the dialog is copied into this stem variable.

Example:

The following example shows how the ResDialog class is implemented, overriding the Init method. If your subclass overrides the Init method, ensure that it calls the Init method of its superclass:

```
::class ResDialog subclass BaseDialog
::method Init
  expose Library Resource DlgData.
  use arg Library, Resource, DlgData.
  return self~init:super(Library, Resource, DlgData.)
```

InitDialog

The *InitDialog* method is called after the Windows dialog has been created. It is useful for setting data fields and initializing combo and list boxes. Do not use *Set...* methods because the SetData method is executed automatically afterwards and sets the values of all dialog items from the attributes.

Protected:

The method is designed to be overwritten in subclasses; it cannot be called from outside the class.

Example:

The following example shows how to use InitDialog to initialize dialog items; in this case a list box:

```
::class MyDialog subclass Userdialog
::method InitDialog
  self~InitDialog:super
  AddListEntry(501, "this is the first line")
  AddListEntry(501, "and this one the second")
```

Run

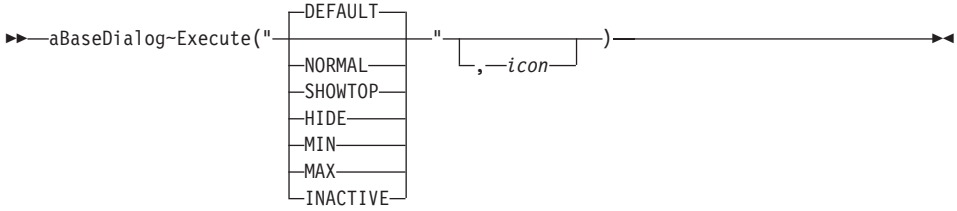
The *Run* method dispatches messages from the Windows dialog until the user terminates the dialog by one of the following actions:

- Press the OK button (the push button with ID 1)
- Press the Cancel button (the push button with ID 2)
- Press the Enter key (if OK or Cancel is the default button)
- Press the Esc key

Protected:

Run is a protected method. You cannot call this method directly; it is called by Execute.

Execute



The *Execute* method creates the dialog, shows it (see "Show" on page 105), starts automatic methods, and destroys the dialog. The data is passed to the Windows dialog before execution and received from it after the dialog is terminated.

Note: If another dialog has already been started in the same process, it is disabled by Execute.

Arguments:

The arguments are:

show See "Show" on page 105.

icon The resource ID of the dialog's icon.

Return value:

0 The dialog was not executed.

1 You terminated the method using the OK button.

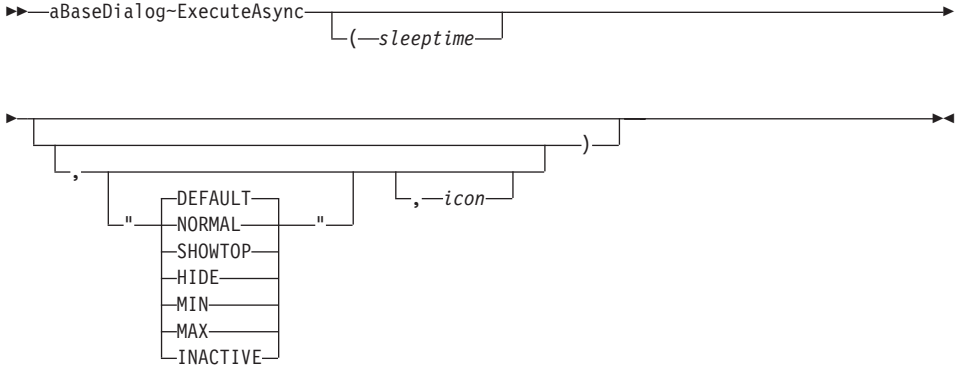
2 You terminated the method using the Cancel button.

Example:

The following example instantiates a new dialog object (remember that it is not possible to instantiate an object of the BaseDialog class), creates a dialog template, and runs the dialog as the topmost window:

```
MyDialog = .UserDialog~new(...)
MyDialog~Create(...)
MyDialog~Execute("SHOWTOP")
```

ExecuteAsync



The *ExecuteAsync* method does the same as Execute, except that it dispatches messages asynchronously. Therefore the ExecuteAsync method returns immediately after the dialog has been started.

Arguments:

The arguments are:

sleeptime
The time slice, in milliseconds, until the next message is processed.

show See "Show" on page 105.

icon The resource ID of the dialog's icon.

Return value:

0 The dialog was started.

1 An error occurred. Do not call the EndAsyncExecution method in this case.

Example:

The following example starts a dialog and runs the statements between ExecuteAsync and EndAsyncExecution asynchronously to the dialog:

```
ret = MyDialog~ExecuteAsync(1000, "SHOWTOP")
if ret = 0 then do
   ...
   /* Object REXX statements to run while the dialog is executing */
   ...
   MyDialog~EndAsyncExecution
   end
else call ErrorMessage("Could not start dialog")
```

EndAsyncExecution

The *EndAsyncExecution* method is used to complete the asynchronous execution of a dialog. It does not terminate the dialog but waits until the user terminates it.

Return value:

0 The dialog was not executed.

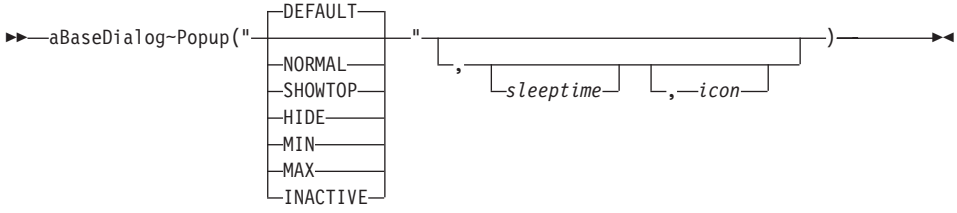
1 The dialog was terminated using the OK button.

2 The dialog was terminated using the Cancel button.

Example:

See the example in "ExecuteAsync" on page 101.

Popup



The *Popup* method starts a dialog, dispatches messages asynchronously, and returns immediately after the dialog is started.

A dialog started with Popup is independent of any other dialog. This means that a dialog already started in the same process is not disabled by Popup. You can therefore use Popup to produce nonmodal dialogs.

Arguments:

The arguments are:

show See "Show" on page 105.

sleeptime

The time, in milliseconds, until the next message is processed.

icon The resource ID of the dialog's icon.

Return value:

This method does not return a value.

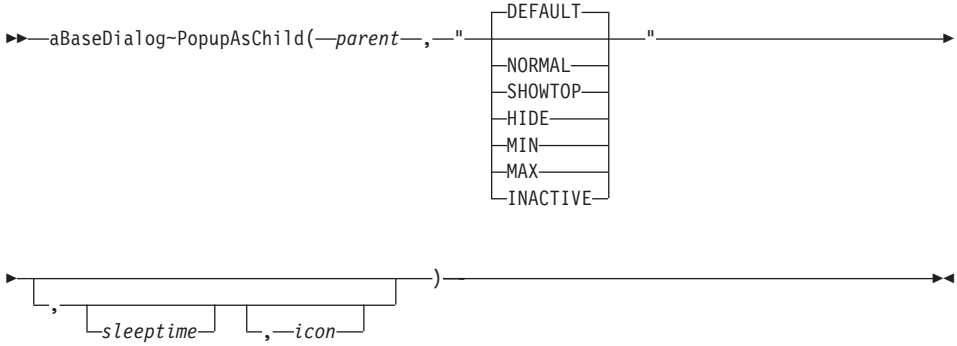
Example:

The following example starts a dialog and runs the statements after Popup asynchronously to the dialog. This means that the dialog reacts to an event like pressing a button and calls the connected method while the DO loop is being processed:

```
MyDialog~Popup("SHOWTOP", 250)
do i = 1 to 1000
   say "Iteration" i
   call SleepMs 100
end
```

This example could also be part of a method handling an event of a dialog, for example dialog A. The newly started dialog MyDialog is independent of dialog A. If dialog A is closed, MyDialog remains unaffected and active.

PopupAsChild



The *PopupAsChild* method starts a dialog as a child dialog of another dialog, dispatches messages asynchronously, and returns immediately after the dialog is started.

A dialog started with PopupAsChild and its parent dialog can be active at the same time. This means that the parent dialog is not disabled by the child dialog. You can therefore use PopupAsChild to produce nonmodal dialogs. However, the child dialog is automatically terminated when the parent dialog is closed.

Arguments:

The arguments are:

parent An object of the PlainBaseDialog class or one of its descendants that is the parent of the newly started dialog.

show See “Show” on page 105.

sleeptime
The time, in milliseconds, until the next message is processed.

icon The resource ID of the dialog's icon.

Return value:
This method does not return a value.

Example:
The following example starts a dialog and runs the statements after PopupAsChild asynchronously to the dialog. This means that the dialog reacts to an event like pressing a button and calls the connected method while the DO loop is being processed. The new dialog is started as a child of MyParent and is therefore closed when the MyParent dialog is closed:

```
MyParent = .UserDialog~new
...
MyParent~Popup("SHOWTOP")
...
MyDialog~PopupAsChild(MyParent, "SHOWTOP", 250)
do i = 1 to 1000
   say "Iteration" i
   call SleepMs 100
   if i = 800 then MyParent~Finished = 1  /* close both dialogs when i = 800 */
end
```

This example could also be part of a method handling an event of a dialog, for example dialog A. The newly started dialog MyDialog is independent of dialog A. If dialog A is closed, MyDialog remains unaffected and active.

IsDialogActive

The *IsDialogActive* method returns 1 if the Windows dialog still exists.

Example:
The following example tests whether the dialog is active:

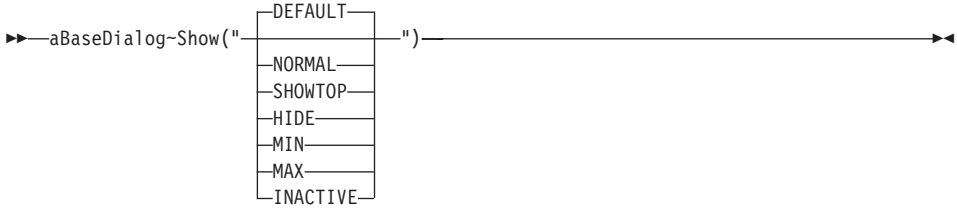
```
if MyDialog~IsDialogActive then ...
```

StopIt

The *StopIt* method removes the Windows dialog from the memory. It is called by Execute, after the user terminates the dialog.

Protected:
This method is protected and for internal use only.

Show



The *Show* method shows the dialog; it is usually called by Execute or ExecuteAsync.

Argument:
The argument must be one of the following:

DEFAULT
Makes the dialog visible with the default window size. This is the default.

NORMAL
Same as default.

SHOWTOP
Makes the dialog the topmost dialog.

HIDE Makes the dialog invisible.

MIN Minimizes the dialog.

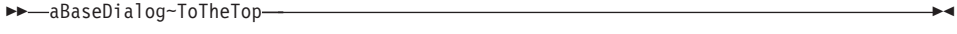
MAX Maximizes the dialog.

INACTIVE
Deactivates the dialog.

Example:
The following statement hides the dialog:

```
MyDialog~Show("HIDE")
```

ToTheTop



The *ToTheTop* method makes the dialog the topmost dialog.

Example:

The following example uses the ToTheTop method to make the user aware of an alarm event:

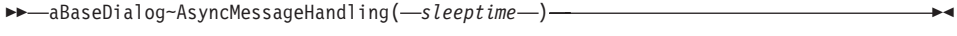
```
aDialog = .MyDialog~new
msg = .Message~new(aDialog, 'Remind')
a = .Alarm~new('17:30:00', msg)

::class MyDialog subclass UserDialog
   :
::method Remind
  self~SetStaticText(102, "Don't forget to go home!")
  self~ToTheTop
```

Note: The Message and Alarm classes are built-in classes of Object REXX. See the *Object REXX for Windows: Reference* for further information.

HandleMessages

The *HandleMessages* method handles dialog messages synchronously. It is called by Execute. HandleMessages is a dispatcher that receives Windows events and posts the message that is set to handle the event.

AsyncMessageHandling



The *AsyncMessageHandling* method starts the asynchronous handling of dialog messages. It is invoked automatically by ExecuteAsync with the Start method of the Object class. A message in this context is the name of an object method that is processed whenever the corresponding event occurs. You can set the messages that should be sent by using *Connect...* methods (see page 107).

Protected:

This method is protected and for internal use only.

Arguments:

The only argument is:

sleeptime

The time slice, in milliseconds, unitl the next message is processed.

PeekDialogMessage

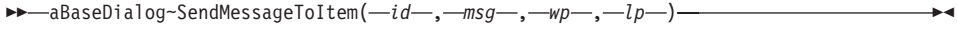
The *PeekDialogMessage* method returns the first pending message of the dialog's message queue without removing it from the message queue.

Return value:
The first pending message or an empty string.

ClearMessages

The *ClearMessages* method clears all pending dialog messages.

SendMessageToItem



The *SendMessageToItem* method sends a Windows message to a dialog item. It is used to influence the behavior of dialog items.

Arguments:
The arguments are:

id The ID of the dialog item.

msg The Windows message (you need a Windows SDK to look up these numbers).

wp The first message parameter (wParam).

lp The second message parameter (lParam).

Example:
The following example sets the marker to radio button 9001:

```
MyDialog~SendMessageToItem(9001, "0x000000F1", 1, 0)
```

Connect Methods

The following methods create a connection between a dialog control and an Object REXX attribute or method. The behavior of the connections differ with the dialog control.

- For push buttons you connect a method to the button. The connected method is called each time the button is pressed.

- For data items, such as an entry line, list box, or combo box, an attribute is created and added to the dialog object. The attribute is used as an interface to the data of the entry line, list box, or combo box.
- Check boxes and radio buttons are also data items and are therefore connected to an attribute. The only valid values for these attributes are 1 for selected and 0 for not selected.
- List boxes, multiple list boxes, and combo boxes can also be connected to a method that is called each time a line in the box is selected.
- For a scroll bar you can specify different methods that are called depending on the user action. The user can click on the arrow buttons, drag the thumb, or use direction keys.

In a UserDialog the *Connect...* methods are called automatically from the *Add...* methods. The proper place for *Connect...* methods is the InitDialog method.

Note: The method name that is to be sent when the specified event occurs must be less than 256 characters.

InitAutoDetection

```
►►—aBaseDialog~InitAutoDetection———————————————————————————————►◄
```

The *InitAutoDetection* method is called by the Init method to change the default setting for the automatic data field detection.

Automatic *data field detection* means that for every dialog data item a corresponding Object REXX attribute is created automatically. If you disable automatic detection, you have to use the *Connect...* methods to assign a dialog item to an Object REXX attribute.

Protected:

This method is protected. You can override this method within your subclass to change the standard behavior.

Example:

The following example overrides the method to switch off auto detection:

```
::class MyDialog subclass UserDialog
::method InitAutoDetection
   self~NoAutoDetection
```

NoAutoDetection

```
►►—aBaseDialog~NoAutoDetection———————————————————————————————►◄
```

The *NoAutoDetection* method switches off auto detection.

AutoDetection

The *AutoDetection* method switches on auto detection.

ConnectResize

The *ConnectResize* method connects a dialog resize event with a method. It is called each time the size of the dialog is changed.

Arguments:

The only argument is:

msgToRaise

The message that is to be sent each time the dialog is resized. Provide a method with a matching name.

Return value:

This method does not return a value.

Example:

```
::class MyDlgClass subclass UserDialog

::method Init
  selfinit:super(...)
  selfConnectResize("OnSize")

::method OnSize
  use arg dummy, sizeinfo    /* sizeinfo contains information about the
                                new width and height */
  msg = self~PeekDialogMessage
  if msg~left(6) = "ONSIZE" then return    /* not the last size
                                                event message */
  w = binaryand(data, "0x0000FFFF") / self~FactorX
  h = binaryand(data, "0xFFFF0000") % X2D("FFFF") / self~FactorY
  say "New width=" w ", new height=" h
```

Note: Connections are usually placed in the Init or InitDialog method. If both methods are defined, use Init as the place for this connection – but not before *init:super* has been called.

ConnectMove

The *ConnectMove* method connects a dialog move event with a method. It is called each time the position of the dialog is changed.

Arguments:

The only argument is:

msgToRaise

The message that is to be sent each time the dialog is moved. Provide a method with a matching name.

Return value:

This method does not return a value.

Example:

```
::class MyDlgClass subclass UserDialog

::method Init
  selfinit:super(...)
  selfConnectResize("OnMove")

::method OnMove
  use arg dummy, posinfo     /* posinfo contains information
                                   about the new position */
  msg = self~PeekDialogMessage
  if msg~left(6) = "ONMOVE" then return    /* not the last move
                                                event message */
  w = binaryand(data, "0x0000FFFF") / self~FactorX
  h = binaryand(data, "0xFFFF0000") % X2D("FFFF") / self~FactorY
  say "New x=" x ", new y=" y
```

Note: Connections are usually placed in the Init or InitDialog method. If both methods are defined, use Init as the place for this connection – but not before *init:super* has been called.

ConnectPosChanged

The *ConnectPosChanged* method connects a change regarding the dialog coordinates with a method. It is called each time the size, position, or place in the Z order of the dialog is changed.

Arguments:

The only argument is:

msgToRaise
The message that is to be sent each time the coordinates of the dialog are changed. Provide a method with a matching name.

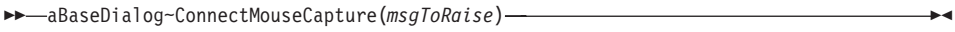
Return value:
This method does not return a value.

Example:

```
::class MyDlgClass subclass UserDialog

::method Init
  self~init:super(...)
  self~ConnectPosChanged("OnNewPos")

::method OnNewPos
  say "The new rectangle is" self~GetWindowRect(self~DlgHandle)
```

Note: Connections are usually placed in the Init or InitDialog method. If both methods are defined, use Init as the place for this connection – but not before *init:super* has been called.

ConnectMouseCapture



The *ConnectMouseCapture* method connects a method with the lose-mouse-capture event. It is called each time the dialog loses the mouse capture. This can happen, for example, when you move a dialog with the mouse and release the left mouse button.

Arguments:
The only argument is:

msgToRaise
The message that is to be sent each time the mouse capture is lost in the dialog. Provide a method with a matching name.

Return value:
This method does not return a value.

ConnectButton

The *ConnectButton* method connects a push button with a method.

Arguments:

The arguments are:

id The ID of the dialog element.

msgToRaise
The message that is sent each time the button is clicked. You should provide a method with the matching name.

Return value:

–1 The specified symbolic ID could not be resolved.

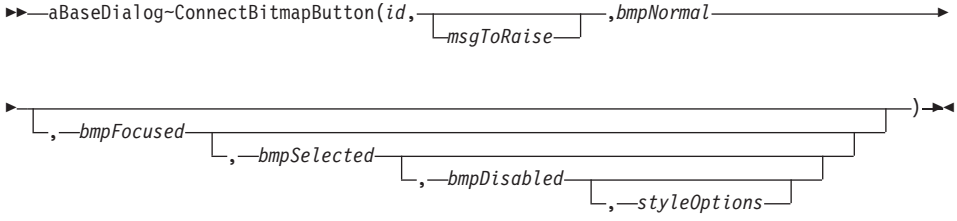
0 No error.

Example:

```
::class MyDlgClass subclass UserDialog

::method Init
  self~init:super(...)
  self~ConnectButton(203, "SayHello")

::method SayHello
  say "Hello"
```

Note: Connections are usually placed in the Init or InitDialog method. If both methods are defined, use Init as the place for this connection – but not before *init:super* has been called.

ConnectBitmapButton



The *ConnectBitmapButton* method connects a bitmap and a method with a push button. The given bitmaps are displayed instead of a Windows push button.

Arguments:

The arguments are:

id The ID of the button.

msgToRaise
The message that is to be sent to this object when the button is clicked.

bmpNormal
The name (alphanumeric), resource ID (numeric), or handle (*INMEMORY* option) of a bitmap file. This bitmap is displayed when the button is not selected, not focused, and not disabled. It is used for the other button states in case the other arguments are omitted.

bmpFocused
This bitmap is displayed when the button is focused. The focused button is activated when the Enter key is pressed.

bmpSelected
This bitmap is displayed while the button is clicked and held.

bmpDisabled
This bitmap is displayed when the button is disabled.

styleOptions
One of the following keywords:

FRAME
Draws a frame around the button. When using this option, the bitmap button behaves like a normal Windows button, except that a bitmap is shown instead of a text.

USEPAL
Stores the colors of the bitmap file as the system color palette. This option is needed when the bitmap was created with a palette other than the default Windows color palette. Use it for one button only, because only one color palette can be active at any time. *USEPAL* is invalid for a bitmap loaded from a DLL.

INMEMORY
This option must be used if the named bitmaps are already loaded into memory by using the LoadBitmap method. In this case, *bmpNormal*, *bmpFocused*, *bmpSelected*, and *bmpDisabled* specify a bitmap handle instead of a file.

STRETCH
If this option is specified and the extent of the bitmap is smaller than the extent of the button rectangle, the bitmap is adapted to match the extent of the button. *STRETCH* has no effect for bitmaps loaded through a DLL.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

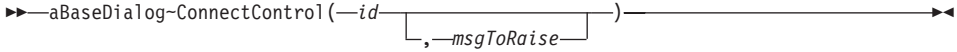
Example:

The following example connects a button with four bitmaps and a method:

```
   ⋮
::method InitDialog
  self~ConnectBitmapButton(204, "BmpButtonClicked",,
                           "AddBut_n.bmp", "AddBut_f.bmp",,
                           "AddBut_s.bmp", "AddBut_d.bmp", "FRAME")
::method BmpButtonClicked
   ⋮
```

See also method "ChangeBitmapButton" on page 162.

ConnectControl



The *ConnectControl* method connects a dialog control with a method.

Arguments:

The arguments are:

id The ID of the dialog element.

msgToRaise

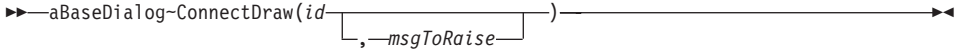
The message that is to be sent each time the button is clicked. Provide a method with the matching name.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectDraw



The *ConnectDraw* method connects the WM_DRAWITEM event with a method. A WM_DRAWITEM message is sent for owner-drawn buttons each time they are to be redrawn.

Arguments:

The arguments are:

id The ID of the dialog control. If the ID is omitted, all drawing events of all owner-drawn buttons are routed to the method.

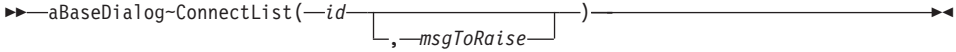
msgToRaise
The message that is to be sent each time the WM_DRAWITEM event occurs. Provide a method with the matching name. You can use `USE ARG ID` to retrieve the ID of the item that is to be redrawn.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectList



The *ConnectList* method connects a list box, multiple list box, or combo box with a method. The method is called each time the user selects a new item from the list.

Arguments:
The arguments are:

id The ID of the dialog element.

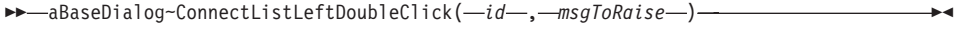
msgToRaise
The message that is to be sent each time the button is pressed. Provide a method with the matching name.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectListLeftDoubleClick



The *ConnectListLeftDoubleClick* method combines a left double-click within the list box with a method.

Arguments:
The arguments are:

id The ID of the list box.

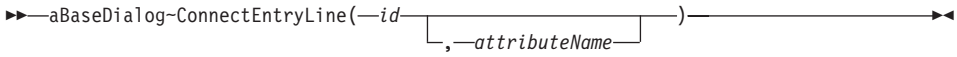
msgToRaise
The name of the method that is to be called.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectEntryLine



The *ConnectEntryLine* method creates a new attribute and connects it to the entry line *id*. The attribute has to be synchronized with the entry line manually. This can be done globally with the SetData and GetData methods (see page 125), or for only one item with the SetEntryLine and GetEntryLine methods (see page 126). It is done automatically by Execute when the dialog starts and after it terminates. If AutoDetection is enabled, or if the dialog is created dynamically (manually or based on a resource script), you do not have to use this method or any other *Connect...* methods that deal with dialog controls).

Arguments:
The arguments are:

id The ID of the entry field you want to connect.

attributeName
An unused valid REXX symbol because an attribute with exactly this name is added to the dialog object with this method. Blank spaces, ampersands (&), and colons (:) are removed from the *attributeName*. If this argument is omitted, is not valid, or already exists, and the ID is numeric, an attribute with the name *DATAid* is used, where *id* is the value of the first argument.

Return value:

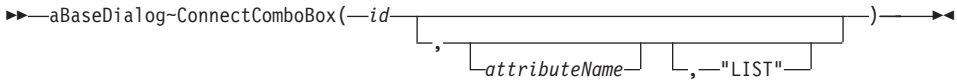
–1 The specified symbolic ID could not be resolved.

0 No error.

Example:
In the following example, the entry line with ID 202 is associated with the attribute *Name*. "Put your name here!" is assigned to the newly created attribute. Then the dialog is executed. After the dialog has terminated, the data of the entry line, which the user might have changed, is copied back to the attribute *Name*.

```
MyDialog~ConnectEntryLine(202, "Name")
MyDialog~Name="Put your name here!"
MyDialog~Execute("SHOWTOP")
say MyDialog~Name
```

ConnectComboBox



The *ConnectComboBox* method creates an attribute and connects it to a combo box. The value of the combo box, that is, the text in the entry line or the selected list item, is associated with this attribute. See "ConnectEntryLine" on page 116 for a more detailed description.

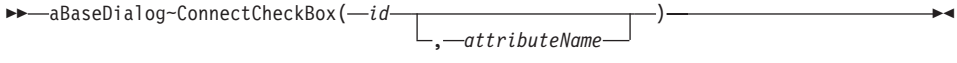
If the combo box is of type "Drop down list", you must specify "LIST" to connect an attribute with the combo box.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectCheckBox



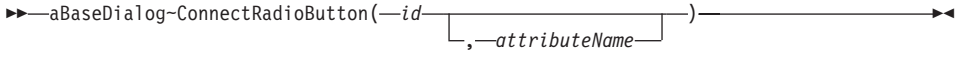
The *ConnectCheckBox* method connects a check box control to a newly created attribute. A check box attribute has only two valid values: 1 if the box has a check mark, and 0 if it has not. See "ConnectEntryLine" on page 116 for a more detailed description.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectRadioButton



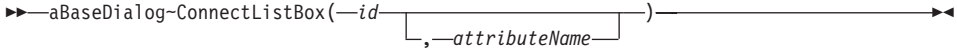
The *ConnectRadioButton* method connects a radio button control to a newly created attribute. A radio button attribute has only two valid values: 1 if the radio button is marked and 0 if it is not. See "ConnectEntryLine" on page 116 for a more detailed description.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectListBox



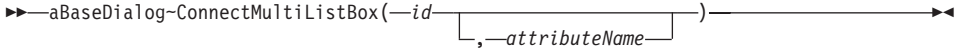
The *ConnectListBox* method connects a list box to a newly created attribute. The value of the attribute is the number of the selected line. Therefore, if the attribute value is 3, the third line is currently selected or will be selected, depending on whether you set data to the dialog or receive it. See "ConnectEntryLine" on page 116 for a more detailed description.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

ConnectMultiListBox



The *ConnectMultiListBox* method connects a list box to a newly created attribute. The list box has the multiple-selection style enabled (by setting the *MULTI* option when adding this list box), that is, you can select more than one item at the same time. The value of the attribute is a string containing the numbers of the selected lines. The numbers are separated by blank spaces. Therefore, if the attribute value is 3 5 6, the third, fifth, and sixth item are currently selected, or will be selected if SetData is executed. See "ConnectEntryLine" on page 116 for a more detailed description.

Return value:

–1 The specified symbolic ID could not be resolved.

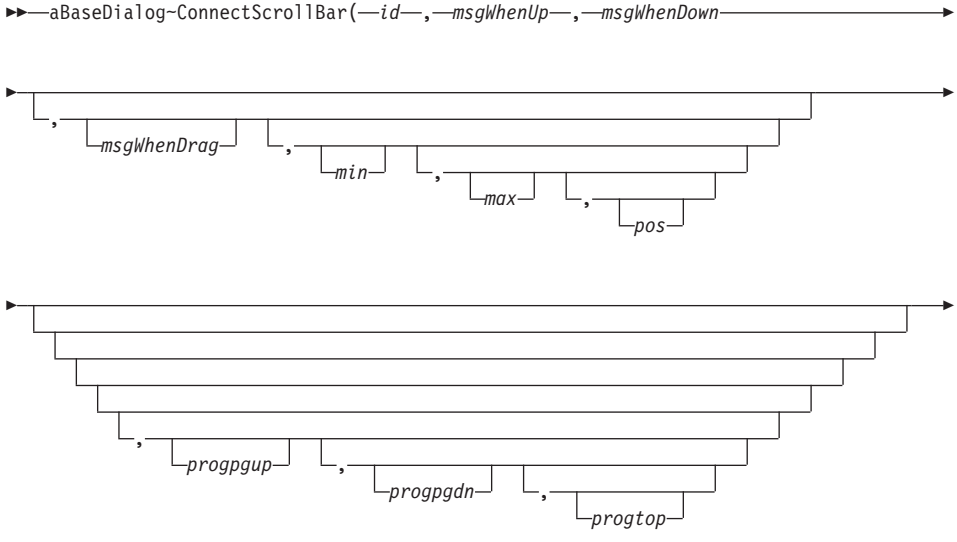
0 No error.

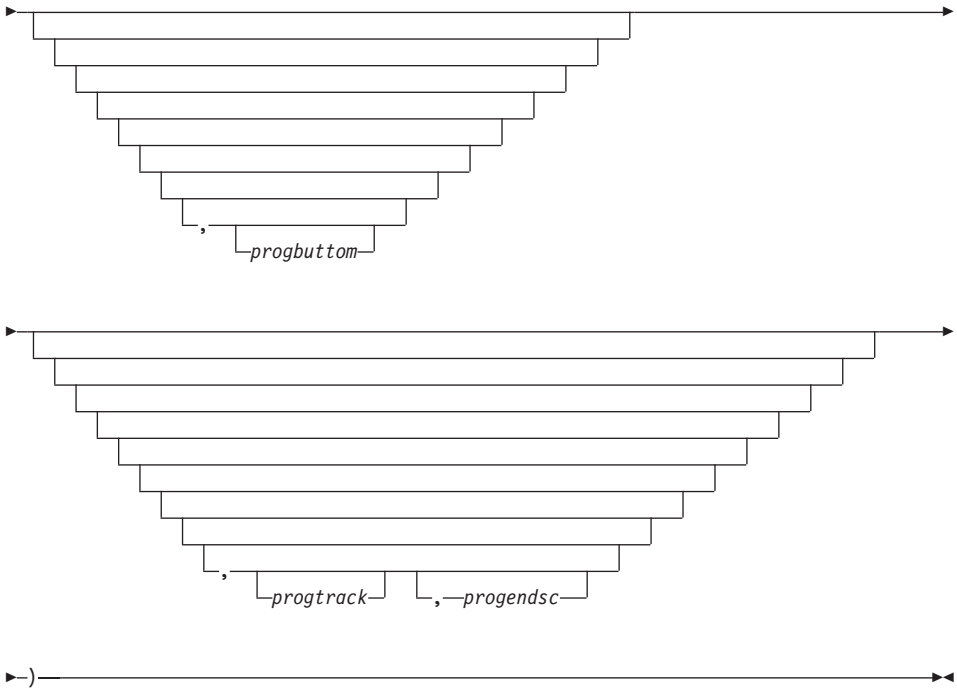
Example:

The following example defines a list box with the name of the four seasons. It then preselects the items *Summer* and *Winter*. After execution of the dialog, it parses the value of the attribute.

```
MyDialog = .ResDialog~new(...)
MyDialog~NoAutoDetection
MyDialog~AddListBox(205, ..., "MULTI")
MyDialog~ConnectMultiListBox(205, "ListBox")
seasons.1="Spring"
seasons.2="Summer"
seasons.3="Autumn"
seasons.4="Winter"
do season over seasons
  MyDialog~AddListEntry(205, season)
end
MyDialog~ListBox="2 4"
MyDialog~Execute("SHOWTOP")
selItems = MyDialog~ListBox
do until anItem =""
   parse var selItems anItem selItems
   say "You selected: "seasons.anItem
end
```

ConnectScrollBar





The *ConnectScrollBar* method initializes and connects a scroll bar to an Object REXX object. Use this method in the InitDialog method.

Protected:

This method is protected.

Arguments:

The arguments are:

id The ID of the scroll bar.

msgWhenUp

The method that is called each time the scroll bar is incremented.

msgWhenDown

The method that is called each time the scroll bar is decremented.

msgWhenDrag

The method that is called each time the scroll bar is dragged with the mouse.

min, max

The minimum and maximum values for the scroll bar.

pos The current or preselected value.

progpgup
The method that is called each time the scroll bar is focused and the PgUp key is pressed.

progpgdn
The method that is called each time the scroll bar is focused and the PgDn key is pressed.

progtop
The method that is called each time the scroll bar is focused and the Home key is pressed.

progbottom
The method that is called each time the scroll bar is focused and the End key is pressed.

progtrack
The method that is called each time the scroll box is dragged.

progendsc
The method that is called each time the scroll box is released after the dragging.

Return value:

–1 The specified symbolic ID could not be resolved.

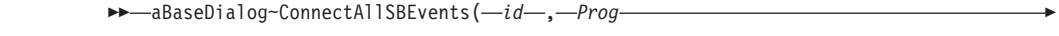
0 No error.

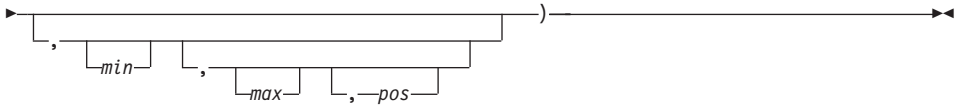
Example:

In the following example, scroll bar 255 is connected to three methods and initialized with 1 as the minimum, 20 as the maximum, and 6 as the current value:

```
::class MyDialog subclass UserDialog
   :
::method DefineDialog
self~ConnectScrollBar(255,"Increase","Decrease","Drag",1,20,6)
   :
::method Increase
   :
::method Decrease
   :
::method Drag
   :
/* see CombineElWithSB below for continuation */
```

ConnectAllSBEvents





Connects all scroll bar events to one method.

Protected:
This method is protected.

Arguments:
The arguments are:

id The ID of the scroll bar

Prog The method that is called for all events sent by the scroll bar.

min, max
The minimum and maximum values for the scroll bar.

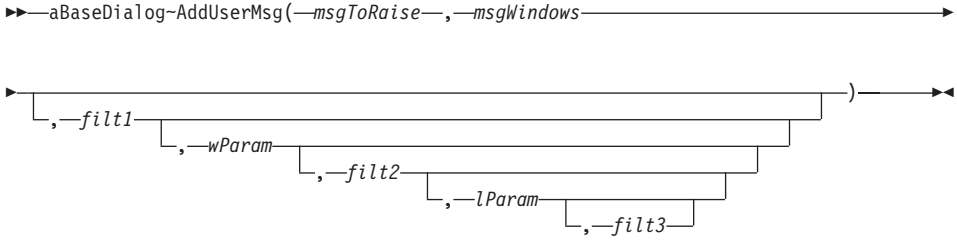
pos The current or preselected value.

Return value:

–1 The specified symbolic ID could not be resolved.

0 No error.

AddUserMsg



The *AddUserMsg* method connects a Windows message with an Object REXX method. This message is designed to be used by Windows programmers who are familiar with the Windows API.

You have to pass the Windows message ID and the two message parameters (*wParam* and *lParam*) to specify the exact event you want to catch. In addition, you can specify filters for each parameter. Filters are useful for catching more than one message or one parameter with one method.

Protected:

This method is protected. You can use it only within the scope of the BaseDialog class or its subclasses.

Arguments:

The arguments are:

msgToRaise
: The message that is to be sent to the Object REXX dialog object each time the specified Windows message is caught. Provide a method with the same name. The maximum size for a message is limited to 256 characters.

msgWindows
: The message in the Windows environment that is to be caught.

filt1 This filter is used to binary AND the incoming Windows message.

wParam
: This is the first parameter that must be passed with the Windows message.

filt2 This filter is used to binary AND the *wParam* argument.

lParam
: This is the second message parameter.

filt3 This is the filter for *lParam*.

Example:

The following example shows an implementation of the *ConnectList* method:

```
::class BaseDialog
  :
::method ConnectList
  use arg msgToRaise, id
  self~AddUserMsg(msgToRaise, '0x00000111', '0xFFFFFFFF',,
                  '0x0001'||id~d2x(4), '0xFFFFFFFF', 0, 0)
```

Assume that this method is called with ID=254 and msgToRaise="ListChanged". After the ConnectList is executed, the ListChanged message is sent to the Object REXX dialog object if the following conditions are true:

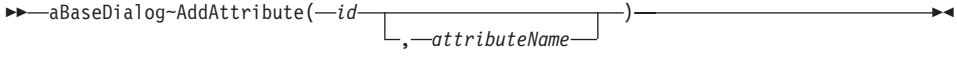
- Message "0x00000111" (WM_COMMAND) is generated by Windows in answer to an event (for example, a button is clicked or a list has changed). The filter "0xFFFFFFFF" ensures that only that message is caught; if the filter were "0xFFFFEFFF", the message "0x00001111" would be caught as well.

- The first message parameter is "0x000100FF". The first part, "0x0001", specifies the event, and the second part, "0x00FE" (equals decimal 254), specifies the dialog control where the event occurred. By using another filter it is possible to make more than one event a trigger for the *ListChanged* method; for example, filter "0xFFFFFFFE" would ignore the last bit of the ID, and this the same event for dialog item 255 would call ListChanged as well.
- The second message parameter and its filter are ignored.

The following example invokes a user-defined method `DoubleClick` each time the left mouse button is double-clicked:

```
self~AddUserMsg('DoubleCick','0x00000203','0xFFFFFFFF',0,0,0,0)
```

AddAttribute



The *AddAttribute* method adds an attribute to the dialog object. The attribute is associated with the dialog control *id*.

Protected:

This method is for internal use only.

Arguments:

The arguments are:

id The ID of the dialog control.

attributeName

The name you want to give to the attribute. This name must comply with the conventions of Object REXX for valid symbols. *AddAttribute* checks whether the argument is valid. In case of an invalid argument, an attribute with the name DATA*id* is created, where *id* is the value of the first argument. This method automatically removes blanks, ampersands (&), and colons (:).

Example:

The first and second lines generate the attributes `Add` and `List all items`. The third line generates the assembled attribute `DATA34` because `ListALLitems` already exists. The fourth line creates attribute `DATA35` because `Update+Refresh` is not a valid symbol name.

```
self~AddAttribute(32, "&Add")
self~AddAttribute(33, "List all items")
self~AddAttribute(34, "ListALLitems:")
self~AddAttribute(35, "Update+Refresh")
```

Get and Set Methods

Get methods are used to retrieve the data from all or individual controls of a dialog. Set methods are used to set the values of all or individual controls, without changing the associated Object REXX attributes.

GetData

The *GetData* method receives data from the Windows dialog and copies it to the associated object attributes.

Example:

The following example continues the SetData example:

```
   :
MyDialog~ConnectEntryLine(102, "ENTRYLINE_1")
MyDialog~ConnectCheckBox(201, )
MyDialog~ConnectListbox(203, "LISTBOX_DAYS")
   :
/* process the dialog */
   :
MyDialog~GetData            /* retrieve dialog item value */
say MyDialog~ENTRYLINE_1
say MyDialog~DATA201
say MyDialog~LISTBOX_DAYS
```

SetData

The *SetData* method transfers data from the Object REXX attributes to the Windows dialog.

Example:

Dialog items with ID 102, 201 and 203 are connected to the attributes `ENTRYLINE_1`, `DATA201`, and `LISTBOX_DAYS`. Attribute `DATA201` is generated by the ConnectCheckBox method. Then the attributes are initialized with some values. This does not change the dialog window, unless you run the SetData method.

```
   :
MyDialog~ConnectEntryLine(102, "ENTRYLINE_1")
MyDialog~ConnectCheckBox(201,)
MyDialog~ConnectListbox(203, "LISTBOX_DAYS")
   :
MyDialog~ENTRYLINE_1="Memorial Day"
```

```
MyDialog~DATA201=1
MyDialog~LISTBOX_DAYS="Monday"

MyDialog~SetData
```

ItemTitle

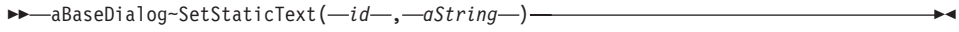
The *ItemTitle* method returns the title of the given dialog item.

Arguments:

The only argument is:

id The ID of the dialog item.

SetStaticText



The *SetStaticText* method changes the text of a static text control.

Arguments:

The arguments are:

id The ID of the static text control for which you want to change the text.

aString

The new text for the static text control.

GetEntryLine

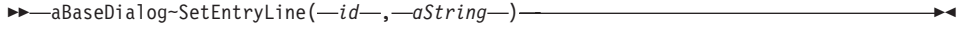
The *GetEntryLine* method returns the value of the given entry line.

Arguments:

The only argument is:

id The ID of the entry line.

SetEntryLine



The *SetEntryLine* method puts the value of a string into an entry line.

Arguments:

The arguments are:

id The ID of the entry line.

aString
The value to be assigned to the entry line.

Example:

Assume that three methods are connected to a push button. The SetToDefault method overrides the value in the Windows dialog entry line 234 with the value 256 but does not change its associated attribute. Using SetEntryLine has the same effect as a change to the entry line made by the user. The associated attribute in the Object REXX object (DATA234) still has the original value. Thus it is possible to undo the changes or confirm them.

```
::method SetToDefault
   self~SetEntryLine(234, "256")

::method AcceptValues
   self~GetAttrib(DATA234)

::method UndoChanges
   self~SetAttrib(DATA234)
```

GetListLine

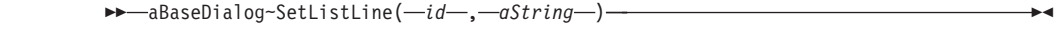
The *GetListLine* method returns the value of the currently selected list item. If you need the index of the item, use the GetCurrentListIndex method. If no item is selected, a null string is returned.

Arguments:

The only argument is:

id The ID of the list box.

SetListLine



The *SetListLine* method assigns the value of a string to the list box. Thus the item with the value of *aString* becomes selected. The first item is selected if the string is not found in the list box. This method does not apply to a multiple selection list box (see "SetMultiList" on page 128).

Arguments:

The arguments are:

id The ID of the list box.

aString
The value of the item to be selected.

Example:

The following example selects item "New York" in list box 232:

```
MyBaseDialog~SetListLine(232, "New York")
```

GetMultiList

The *GetMultiList* method can be applied to a multiple-selection list box. It returns a string containing the indexes of up to 1000 selected items. The numbers are separated by blanks.

Arguments:

The only argument is:

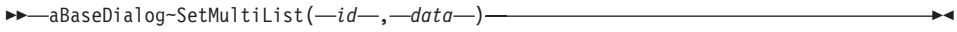
id The ID of the multiple-selection list box.

Example:

The following example shows how to handle a multiple-selection list box. It parses the returned string as long as it contains an index.

```
selLines = MyDialog~GetMultiList(555)
do until selLines = ""
   parse var selLines aLine selLines
   say aLine
end
```

SetMultiList



The *SetMultiList* method selects one or more lines in a multiple-selection list box.

Arguments:

The arguments are:

id The ID of the multiple-selection list box.

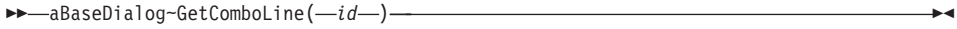
data The indexes (separated by blanks) of the lines to be selected.

Example:

The following example selects the lines 2, 5, and 6 of list box 345:

```
MyDialog~SetMultiList(345, "2 5 6")
```

GetComboLine



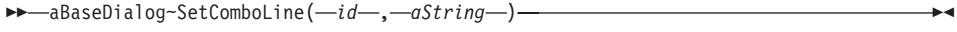
The *GetComboLine* method returns the value of the currently selected list item of a combo box. If you need the index of the item, use the GetCurrentComboIndex method. If no item is selected, a null string is returned.

Arguments:

The only argument is:

id The ID of the combo box.

SetComboLine



The *SetComboLine* method assigns a string to the given combo box. Thus the item with the value of *aString* becomes selected. If not found in the combo box, the first item selected is the string.

Arguments:

The arguments are:

id The ID of the combo box.

aString

The value of the item to be selected.

GetRadioButton

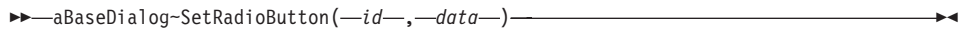
The *GetRadioButton* method returns 1 if the radio button is selected, 0 if it is not selected.

Arguments:

The only argument is:

id The ID of the radio button.

SetRadioButton



The *SetRadioButton* method marks the radio button if the given data value is 1, and removes the mark if the value is 0.

Arguments:

The arguments are:

id The ID of the radio button.

data 1 to select the button or 0 to deselect it.

GetCheckBox

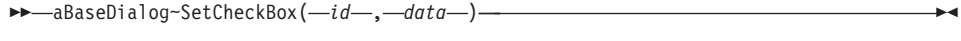
The *GetCheckBox* method returns the value of a check box: 1 if the check box is selected (has a check mark), 0 if it is not selected.

Arguments:

The only argument is:

id The ID of the check box.

SetCheckBox



The *SetCheckBox* method puts a check mark in the check box if the given data value is 1 and removes the check mark if the value is 0.

Arguments:

The arguments are:

id The ID of the check box.

data The value 1 to check the box or 0 to remove the check mark.

GetValue

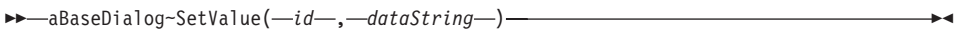
The *GetValue* method gets the value of a dialog item, regardless of its type. The item must have been connected before.

Arguments:

The only argument is:

id The ID of the dialog item.

SetValue



The *SetValue* method sets the value of a dialog item. You do not have to know what kind of item it is. The dialog item must have been connected before.

Arguments:

The arguments are:

id The ID of the dialog item.

dataString

The value that is assigned to the item. It must be a valid value.

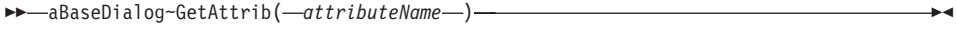
Example:

The following example sets dialog item 123 to (string) value "1 2 3". This is meaningful if 123 is an entry field, or if it is a list box that contains the line "1 2 3". However, it is an error to apply this against a check box.

```
MyDialog~SetValue(123, "1 2 3")
```

Note: If it is a multiple-selection list box, the SetValue method does not look for an item with "1 2 3" as value but highlights the first, second, and third line.

GetAttrib



The *GetAttrib* method assigns the value of a dialog item to the associated Object REXX attribute. It does not return a value. You do not have to know the ID or the type of the dialog item.

Arguments:

The only argument is:

attributeName

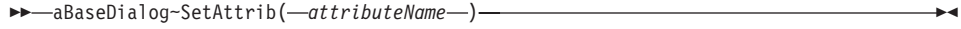
The name of the attribute.

Example:

The following example shows how to get the data value of a dialog item without knowing its ID:

```
MyDialog~GetAttrib("FirstName")
if MyDialog~FirstName¬="" then ...
```

SetAttrib



The *SetAttrib* method copies the value of an attribute to the associated dialog item. You do not have to know the ID or the type of the dialog item.

Arguments:

The only argument is:

attributeName

The name of the attribute.

Example:

The following example copies the value of the attribute *DATA101* to the associated dialog item:

```
MyBaseDialog~SetAttrib("DATA101")
```

SetDataStem

The *SetDataStem* method sets all Windows dialog items to the values within the given stem; the suffixes of the stem variable are the dialog IDs.

Protected:

This method is protected.

Arguments:

The only argument is:

dataStem.

A stem variable containing initialization data. Remember the trailing period.

Example:

The following example initializes the dialog items with ID 21, 22, and 23:

```
  ⋮
```

```
dlgStem.21="Windows 95"
dlgStem.22="0"
dlgStem.23="1 2 3"
self~SetDataStem(dlgStem.)
```

GetDataStem

The *GetDataStem* method gets the values of all dialog items and copies them to the given stem.

Protected:
This method is protected.

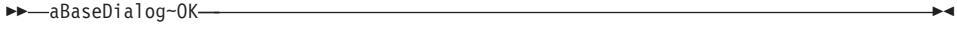
Arguments:
The only argument is:

dataStem.
The name of a stem variable to which the data is returned. Remember the trailing period.

Standard Event Methods

The following methods are abstract methods that are called each time a push button with ID 1, 2, or 9 is pressed.

OK



The *OK* method is called in response to a pressed OK button. It calls Validate to get its return code. The default return code is the *self~finished* attribute, which is usually 1, and the dialog is terminated.

Protected:
This method is protected. You might want to override it in your subclass. If you do, forward the OK message to the parent class after processing has finished. Set the *self~finished* attribute to 1 or 0 and return it. The dialog continues executing if you return the value 0. See also "Validate" on page 134.

Example:
The following example shows how to override the OK method:

```
::method OK
...
self~ok:super()
self~finished = 1
return self~finished
```

Cancel

The *Cancel* method is called in response to a pressed Cancel button. The default return code is the *self~finished* attribute, which is usually 1 and the dialog is terminated. The *InitCode* attribute is set to 2 if the dialog is terminated.

Protected:

This method is protected. You might want to override it in your subclass. If you do, forward the Cancel message to the parent class after processing has finished. Set the *self~finished* attribute to 1 or 0 and return it. The dialog continues executing if you return the value 0.

Help

The *Help* method is called in response to a pressed Help button.

Protected:

This method is protected. You might want to override it in your subclass.

Validate

The *Validate* method is an abstract method that is called to determine whether the dialog can be closed. This method is called by the OK method. The standard implementation is that Validate returns 1 and the dialog is closed. The dialog is not closed if Validate returns 0.

Protected:

The method is designed to be defined in a subclass.

Example:

In the following example *Validate* checks whether entry line 203 is empty. If it is empty, *Validate* returns 0, which indicates that the dialog cannot be closed.

```
::class MyDialog subclass UserDialog
::method Validate
   if self~GetEntryLine(203) = "" then return 0
   else return 1
```

Leaving

The *Leaving* method is called when the dialog was closed.

DeInstall

The *DeInstall* method removes the external functions from the Object REXX API manager. It should be called at the end of each dialog. The installed functions are freed when all dialogs are finished.

Combo Box Methods

The following methods belong to combo boxes.

AddComboEntry

The *AddComboEntry* method adds a string to the list of a combo box. The new item becomes the last one, if the list does not have the *SORT* flag set. In the case of a sorted list, the new item is inserted at the proper position.

Arguments:

The arguments are:

id The ID of a combo box.

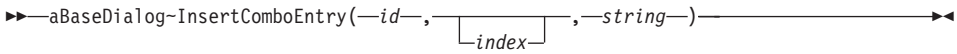
aString
The data to be inserted as a new line.

Example:

The following example adds the new line, `Another item`, to the list of combo box 103:

```
MyDialog~AddComboEntry(103, "Another item")
```

InsertComboEntry



The *InsertComboEntry* method inserts a string into the list of a combo box.

Arguments:

The arguments are:

id The ID of the combo box.

index The index (line number) where you want to insert the new item. If this argument is omitted, the new item is inserted after the currently selected item.

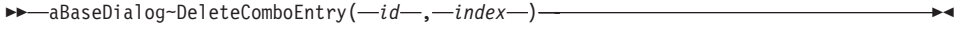
string The data string to be inserted.

Example:

This statement inserts `The new third line` after the second line into the list of combo box 103:

```
MyDialog~InsertComboEntry(103, 2, "The new third line")
```

DeleteComboEntry



The *DeleteComboEntry* method deletes a string from the combo box.

Arguments:

The arguments are:

id The ID of the combo box.

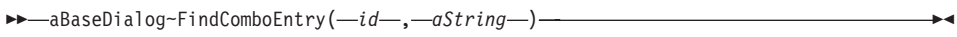
index The line number of the item to be deleted. Use the FindComboEntry method (see page137) to retrieve the index of an item.

Example:

The following example shows a method that deletes the item that is passed to the method in the form of a text string from combo box 203:

```
   ⋮
::method DeleteFromCombo
   use arg delStr
   idx = self~FindComboEntry(203, delStr)
   self~DeleteComboEntry(203, idx)
```

FindComboEntry



The *FindComboEntry* method returns the index corresponding to a given text string in the combo box.

Arguments:

The arguments are:

id The ID of the combo box

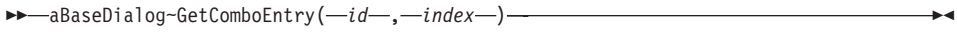
aString

The string of which you search the index in the combo box.

Example:

See "DeleteComboEntry" on page 136 for an example.

GetComboEntry



The *GetComboEntry* method returns the string at index *index* of the combo box.

Arguments:

The arguments are:

id The ID of the combo box

index The index of the list entry to be retrieved

Example:

```
if dlg~GetComboEntry(203,5)="JOHN"
then ...
```

GetComboItems

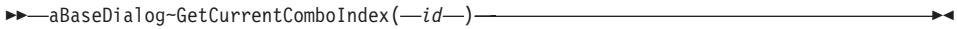
The *GetComboItems* method returns the number of items in the combo box.

Arguments:

The only argument is:

id The ID of the combo box

GetCurrentComboIndex



The *GetCurrentComboIndex* method returns the index of the currently selected item within the list. See "GetComboLine" on page 129 for information on how to retrieve the selected combo box item.

Arguments:

The only argument is:

id The ID of the combo box.

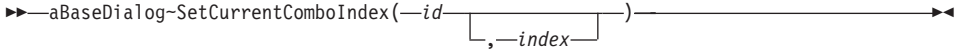
Example:

The following example displays the line number of the currently selected combo box item within entry line 240:

```
::class MyListDialog subclass UserDialog
   :
::method Init
   self~Init:super
   self~ConnectList(230, "ListSelected")
   :
::method ListSelected
   line = self~GetCurrentComboIndex(230)
   SetEntryLine(240, line)
```

Method `ListSelected` is called each time the selected item within the combo box changes.

SetCurrentComboIndex



The *SetCurrentComboIndex* method selects the item with the given index within the list. If called without an index, all items in the list are deselected. See "SetComboLine" on page 129 for information on how to select a combo box item using a data value.

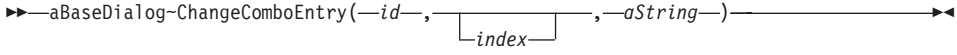
Arguments:

The arguments are:

id The ID of the combo box.

index The index within the combo box.

ChangeComboEntry



The *ChangeComboEntry* method changes the value of a given entry in a combo box to a new string.

Arguments:

The arguments are:

id The ID of the combo box

index The index number of the item you want to replace. To retrieve the index, use the FindComboEntry or GetCurrentComboIndex method (see page 137 or 138).

aString

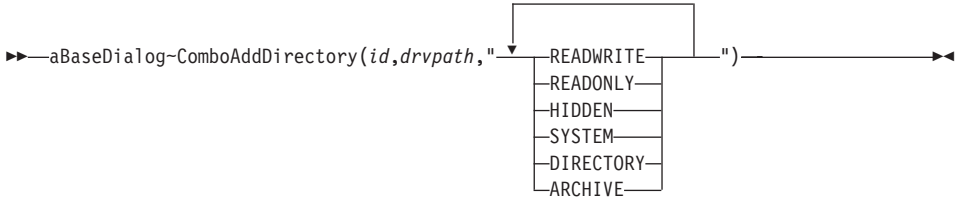
The new text.

Example:

In the following example, method `ChangeButtonPressed` changes the currently selected line of combo box 230 to the value in entry line 250:

```
   ⋮
::method ChangeButtonPressed
   idx = self~GetCurrentComboEntry(230)
   str = self~GetEntryLine(250)
   self~ChangeComboEntry(230, idx, str)
```

ComboAddDirectory



The *ComboAddDirectory* method adds all or selected file names in the given directory to the combo box.

Arguments:

The arguments are:

id The ID of the combo box.

drvpath
The drive, path, and name pattern.

fileAttributes
Specify the file attributes that the files must have in order to be added:

READWRITE
Normal read/write files (same as none).

READONLY
Files that have the read-only bit.

HIDDEN
Files that have the hidden bit.

SYSTEM
Files that have the system bit.

DIRECTORY
Files that have the directory bit.

ARCHIVE
Files that have the archive bit.

Example:
The following example fills the combo box list with the names of all read/write files with extension .REX in the given directory:

```
MyDialog~ComboAddDirectory(203, drive":\"path"\*.rex", "READWRITE")
```

ComboDrop

The *ComboDrop* method deletes all items from the list of the given combo box.

Arguments:
The only argument is:

id The ID of the combo box.

List Box Methods

The following methods deal with list boxes.

GetListWidth

The *GetListWidth* method returns the scrollable width of a list box, in dialog units.

Arguments:

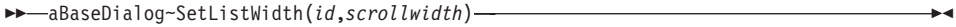
The only argument is:

id The ID of the list box of which you want to know the scrollable width.

Return value:

The width of the scrollable area of the list box, in dialog units.

SetListWidth



The *SetListWidth* method sets the scrollable width of a list box, in dialog units. If the scrollable width is greater than the width of the list box and the "HSCROLL" (WS_HSCROLL in the resource script) style is defined for the list box (see “AddListBox” on page 244), a horizontal scroll bar is displayed.

Arguments:

The arguments are:

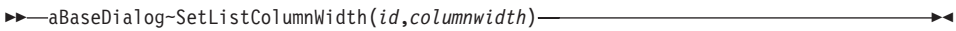
id The ID of the list box for which you want to set the scrollable width.

scrollwidth
The width of the scrollable area of the list box, in dialog units.

Return value:

This method does not return a value.

SetListColumnWidth



The *SetListColumnWidth* method sets the width of all columns in a list box, in dialog units.

Arguments:

The arguments are:

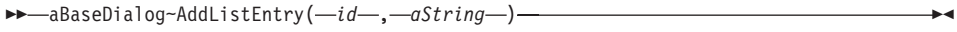
id The ID of the list box for which you want to set the column width.

columnwidth
The width of the columns in the list box, in dialog units.

Return value:

This method does not return a value.

AddListEntry



The *AddListEntry* method adds a string to the given list box. See also "AddComboEntry" on page 135. The line is added at the end (by default), or in sorted order if the list box was defined with the sorted flag.

Arguments:

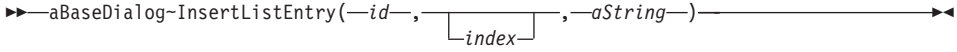
The arguments are:

id The ID of a list box.

aString

The data to be inserted as a new line.

InsertListEntry



The *InsertListEntry* method inserts a string into the given list box. See also "InsertComboEntry" on page 136.

Arguments:

The arguments are:

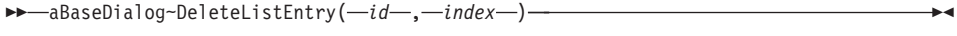
id The ID of the list box.

index The index (line number starting with 1) of the item after which the new item is inserted. If this argument is omitted, the new item is inserted after the currently selected item.

aString

The text string to be inserted.

DeleteListEntry



The *DeleteListEntry* method deletes an item from a list box. See also "DeleteComboEntry" on page 136.

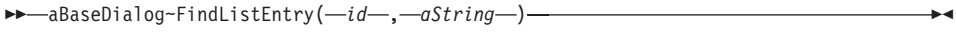
Arguments:

The arguments are:

id The ID of the list box.

index The line number of the item to be deleted. Use "FindListEntry" to retrieve the index of an item. If this argument is omitted, the currently selected item is deleted.

FindListEntry



The *FindListEntry* method returns the index of the given string within the given list box. The first item has index 1, the second has index 2, and so forth. If the list box does not contain the string, 0 is returned.

Arguments:

The arguments are:

id The ID of the list box.

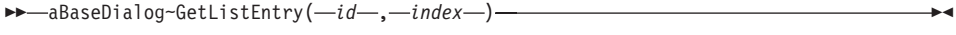
aString
The item text you are looking for.

Example:

The following example shows a method that adds the contents of an entry line (214) to the list box (215) if no item with the same value is already contained in it:

```
   ⋮
::method PutEntryInList
   str = self~GetEntryLine(214)
   if self~FindListEntry(215, str) = 0 then
      self~AddListEntry(215, str)
```

GetListEntry



The *GetListEntry* method returns the string at index *index* of the list.

Arguments:

The arguments are:

id The ID of the list box.

index The index of the list entry to be retrieved.

Example:

```
if dlg~GetListEntry(203,5)="JOHN"
then ...
```

GetListItems

The *GetListItems* method returns the number of items in the list box.

Arguments:

The only argument is:

id The ID of the list box.

GetListItemHeight

The *GetListItemHeight* method returns the height of the items in a list box, in dialog units.

Arguments:

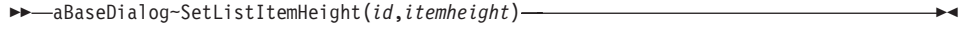
The only argument is:

id The ID of the list box of which you want to know the item height.

Return value:

The height of the list box items, in dialog units.

SetListItemHeight



The *SetListItemHeight* method sets the height for all items in a list box, in dialog units. It determines the space between the individual list box items.

Arguments:

The arguments are:

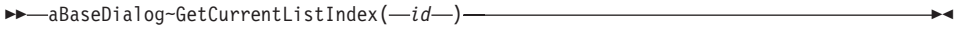
id The ID of the list box for which you want to set the item height.

itemheight

The height of the items in the list box, in dialog units.

Return value:
A number smaller than 0 if the height that you specify is not valid.

GetCurrentListIndex

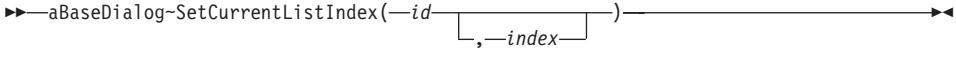


The *GetCurrentListIndex* method returns the index of the currently selected list box item, or 0 if no item is selected. See "GetListLine" on page 127 for information on how to retrieve the selected list box item.

Arguments:
The only argument is:

id The ID of the list box.

SetCurrentListIndex



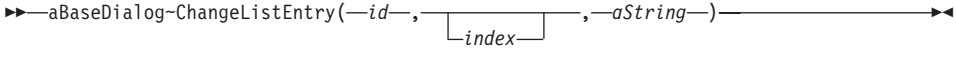
The *SetCurrentListIndex* selects the item with the given index in the list. If called without an index, all items in the list are deselected. See "SetListLine" on page 127 for information on how to select a list box item using a data value.

Arguments:
The arguments are:

id The ID of the list box.

index The index within the list box.

ChangeListEntry



The *ChangeListEntry* method changes the contents of a line in a list box.

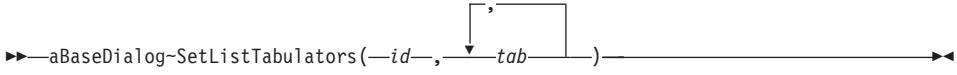
Arguments:
The arguments are:

id The ID of the list box.

index The index of the item that you want to replace. If this argument is omitted, the currently selected item is changed.

aString
The new text of the item.

SetListTabulators



The *SetListTabulators* method sets the tabulators for a list box. Thus you can use items containing tab characters ('09'x), which is useful for formatting the list in more than one column.

Arguments:
The arguments are:

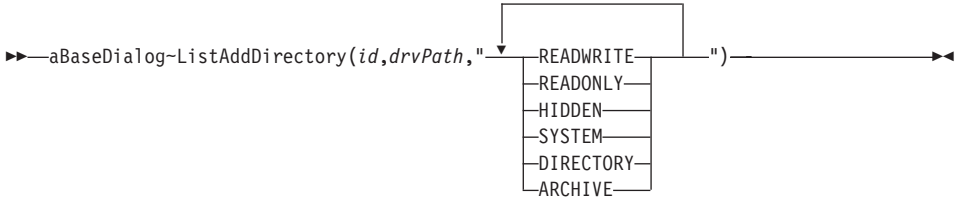
id The ID of the list box.

tab The positions of the tabs relative to the left edge of the list box.

Example:
The following example creates a four-column list and adds a tab-formatted row to the list. The tabulator positions are 10, 20, and 30.

```
MyDialog~SetListTabulators(102, 10, 20, 30)
MyDialog~AddListEntry(102, var1 || '09'x || var2 || '09'x || ,
                           var3 || '09'x || var4)
```

ListAddDirectory



The *ListAddDirectory* method adds all or selected file names of a given directory to the list box. See "ComboAddDirectory" on page 139 for more information.

ListDrop

The *ListDrop* method removes all items from the list box.

Arguments:

The only argument is:

id The ID of the list box.

Scroll Bar Methods

The following methods are used to set or get the behavior of a scroll bar. You can connect scroll bars with numerical entry fields to edit the value with the mouse.

GetSBRange

The *GetSBRange* method returns the range of a scroll bar control. It returns the two values (minimum and maximum) in one string, separated by a blank.

Protected:

This method is protected.

Arguments:

The only argument is:

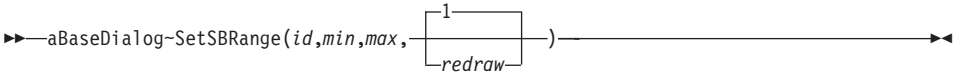
id The ID of the scroll bar.

Example:

The following example demonstrates how to get the minimum and the maximum values of the scroll bar:

```
   :
::method DumpSBRange
SBrange = self~GetSBRange(234)
parse var SBrange SBmin SBmax
say SBmin " - " SBmax
```

SetSBRange



The *SetSBRange* method sets the range of a scroll bar control. It sets the minimum and maximum values.

Protected:
This method is not intended to be used outside of the BaseDialog class.

Arguments:
The arguments are:

id The ID of a scroll bar control.

min The minimum value.

max The maximum value.

redraw
A flag indicating whether (1) or not (0) the scroll bar should be redrawn. The default is 1.

Example:
The following example allows the scroll bar to take values between 1 and 10:

```
MyDialog~SetSBRange(234, 1, 10, 1)
```

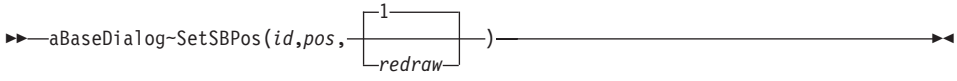
GetSBPos

The *GetSBPos* method returns the current value of a scroll bar control.

Arguments:
The only argument is:

id The ID of the scroll bar.

SetSBPos



The *SetSBPos* method sets the current value of a scroll bar control.

Protected:
This method is protected.

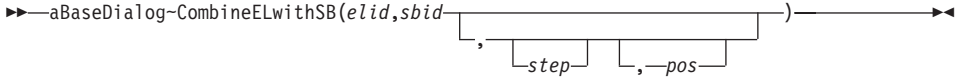
Arguments:
The arguments are:

id The ID of the scroll bar.

pos The value to which you want to set the scroll bar. It must be within the defined range.

redraw
A flag indicating whether (1) or not (0) the scroll bar should be redrawn. The default is 1.

CombineELwithSB



The *CombineELwithSB* method connects an entry line with a scroll bar such that each time the slider of the scroll bar is moved, the value of the entry field is changed. This method must be used in a method registered with ConnectScrollBar.

Arguments:
The arguments are:

elid The ID of the entry line.

sbid The ID of the scroll bar.

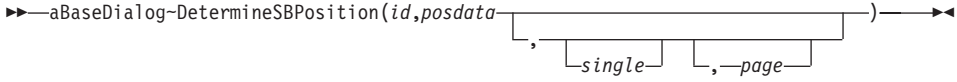
step The size of one step. If, for example, *step* is 3 and the current position is 4, the next position is 7.

pos If the *step* value is zero, this sets the position of the scroll bar and entry line. Use it in the method registered for *drag*.

Example:
The following example continues the example of *ConnectScrollBar*. In the registered methods an entry line (251) is combined with the scroll bar (255).

```
::method Increase
self~CombineELwithSB(251,255,+20)
::method Decrease
self~CombineELwithSB(251,255,-20)
::method Drag
use arg wparam, lparam                  /* wparam=position */
self~CombineELwithSB(251,255,0,wparam)
```

DetermineSBPosition



The *DetermineSBPosition* method calculates and sets the new scroll bar position based on the position data retrieved from the scroll bar and the step information.

Protected:
This method is protected.

Arguments:
The arguments are:

id The ID of the scroll bar.

posdata
The position information sent with the connected scroll bar event.

single This number is added (or subtracted if negative) to the current position for a single step. If omitted, the single step size is 1.

page This number is added (or subtracted if negative) to the current position for a page step. If omitted, the page step size is 10.

Return value:
The new scroll bar position.

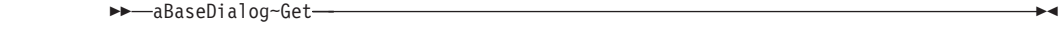
Example:
The following example demonstrates how to update the scroll bar position. Each time the ScrollBarEventHandler is called by an event for scroll bar SB_SIZE, the position of the scroll bar is calculated and updated. `posdata` is sent along with the scroll bar event.

```
   /* Method ScrollBarEventHandler is connected to item SB_SIZE */
::method ScrollBarEventHandler
    use arg posdata, sbwnd
    pos = self~DetermineSBPosition("SB_SIZE",posdata,1,25)
    return pos
```

Methods for Window Handles, Sizes, and Positions

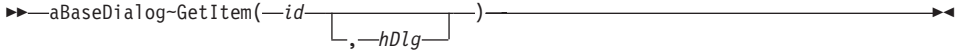
The following methods return information about the dialog or a single dialog control.

Get



The *Get* method returns the handle of the current Windows dialog. A handle is a unique reference to a particular Windows object. Handles are used within some of the methods to work on a particular Windows object.

GetItem



The *GetItem* method returns the handle of a particular dialog item.

Arguments:

The arguments are:

id The ID of the dialog element.

hDlg The handle of the dialog. If it is omitted, the main dialog handle is used.

Example:

The following example returns the handle of a push button:

```
hndPushButton = MyDialog~GetItem(101)
```

GetPos

The *GetPos* method returns the dialog window's position in pixels. The values are returned in a string separated by blanks for parsing.

Example:

The following example moves the Window towards the top left of the screen. *GetScreenSize* is an external function of OODialog.

```
parse value self~GetPos with px py
self~Move( px%self~FactorX - 10, py%self~FactorY - 10)
```

GetButtonRect

The *GetButtonRect* method returns the size and position of the given button. The four values (left, top, right, bottom) are returned in one string separated by blanks.

Arguments:

The only argument is:

id The ID of the button

GetWindowRect

```
►►—aBaseDialog~GetWindowRect(—hwnd—)————————————————————————►◄
```

The *GetWindowRect* method returns the size and position of the given window. The four values (left, top, right, bottom) are returned in one string separated by blanks.

Arguments:

The only argument is:

hwnd The handle of the window. Use the Get method to retrieve the window handle.

Appearance Modification Methods

The methods listed below are to change the appearance of the dialog itself or one of its items. The list contains methods to change the size, position, visibility, and title (header).

Some of the methods come in two flavors, normal (for example, *ShowWindow*) and fast (for example, *ShowWindowFast*). The *fast* extension indicates that the method does not redraw the item or window immediately. After modifying several items, invoke the Update method (see page 196) to redraw the dialog.

BackgroundColor

```
►►—aBaseDialog~BackgroundColor(color)————————————————————————►◄
```

The *BackgroundColor* method sets the background color of a dialog.

Arguments:

The only argument is:

color A color-palette index specifying the background color. For information on the color numbers, refer to"Chapter 7. Definition of Terms" on page 87.

Return value:

This method does not return a value.

FocusItem

```
►►—aBaseDialog~FocusItem(—id—)————————————————————————————►◄
```

The *FocusItem* method sets the input focus to a particular dialog item.

Arguments:

The only argument is:

id The ID of the dialog item to set the focus to

Return value:

A handle to the window that previously had the input focus

EnableItem

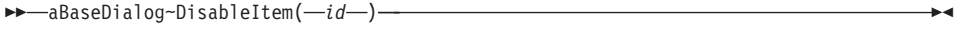
The *EnableItem* method enables the given dialog item.

Arguments:

The only argument is:

id The ID of the item

DisableItem



The *DisableItem* method disables the given dialog item. A disabled dialog item is usually indicated by a gray instead of a black title or text; it cannot be changed by the user.

Arguments:

The only argument is:

id The ID of the item

HideItem

The *HideItem* method makes the given item disappear from the screen and thus unavailable to the user. In fact, the item is still in the dialog and you can transfer its data.

Arguments:

The only argument is:

id The ID of the item

HideItemFast

The *HideItemFast* method hides an item without redrawing its area. It is similar to the HideItem method, but it is faster because the item's area is not redrawn. The *HideItemFast* method is used when more than one item state is modified. After the operations, you can manually redraw the dialog window, using the "Update" on page 196 method.

Arguments:

The only argument is:

id The ID of the item

ShowItem

The *ShowItem* method makes the given dialog item reappear on the screen.

Arguments:

The only argument is:

id The ID of the item

ShowItemFast

The *ShowItemFast* method shows an item without redrawing its area. It is the counterpart to the HideItemFast method.

HideWindow

The *HideWindow* method hides a whole dialog window or a dialog item.

Arguments:

The only argument is:

hwnd A handle to the window or dialog item. Use the Get or GetItem method to get a handle.

Example:

The following example hides the whole dialog:

```
hwnd = MyDialog~Get
MyDialog~HideWindow(hwnd)
```

HideWindowFast

The *HideWindowFast* method is similar to the HideWindow method, but it is faster because the window's or item's area is not redrawn. The *HideWindowFast* method is used when more than one state is modified. After the operations, you can manually redraw the dialog window, using the Update method.

Arguments:

The only argument is:

hwnd A handle to the window or dialog item

ShowWindow

The *ShowWindow* method shows the window or item again.

Arguments:

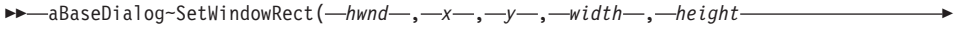
The only argument is:

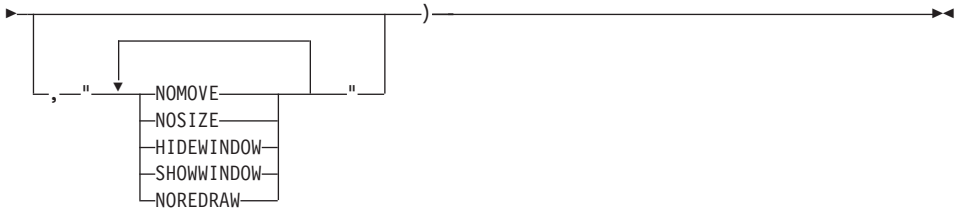
hwnd The handle of a window or an item

ShowWindowFast

The *ShowWindowFast* method is the counterpart to the HideWindowFast method.

SetWindowRect





The *SetWindowRect* method sets new coordinates for a specific window.

Arguments:

The arguments are:

hwnd The handle to the dialog that is to be repositioned.

x, y The new position of the upper left corner, in screen pixels.

width The new width of the window, in screen pixels.

height The new height of the window, in screen pixels.

showOptions

This argument can be one or more of the following keywords, separated by blanks:

NOMOVE

The upper left position of the window has not changed.

NOSIZE

The size of the window has not changed.

HIDEWINDOW

The window is to be made invisible.

SHOWWINDOW

The window is to be made visible.

NOREDRAW

The window is to be repositioned without redrawing it.

Return value:

0 Repositioning was successful.

1 Repositioning failed.

RedrawWindow

The *RedrawWindow* method redraws a specific dialog.

Arguments:

The only argument is:

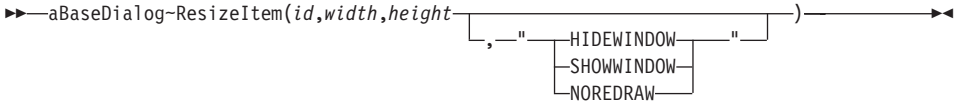
hwnd The handle to the dialog that is to be redrawn.

Return value:

0 Redrawing was successful.

1 Redrawing failed.

ResizeItem



The *ResizeItem* method changes the size of a dialog item.

Arguments:

The arguments are:

id The ID of the dialog item you want to resize

width, height
The new size in dialog units

showOptions
This argument can be one of the following keywords:

HIDEWINDOW
Hides the item

SHOWWINDOW
Shows the item

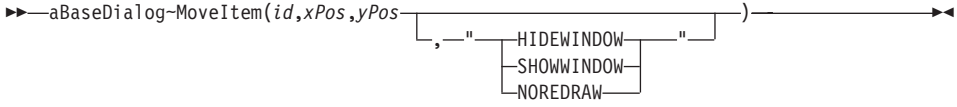
NOREDRAW
Resizes the item without updating the display. Use the Update method to manually update the display.

Example:

The following example resizes a dialog item:

```
MyDialog~ResizeItem(123, 40, 30, "SHOWWINDOW")
```

MoveItem



The *MoveItem* method moves a dialog item to another position within the dialog window.

Arguments:

The arguments are:

id The ID of the dialog item you want to move

xPos, yPos
The new position in dialog units relative to the dialog window

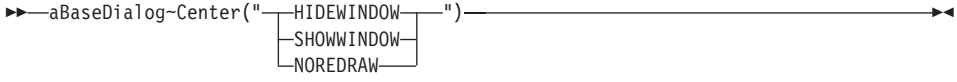
showOptions
This argument can be one of the following keywords:

HIDEWINDOW
Hides the dialog

SHOWWINDOW
Shows the dialog

NOREDRAW
Moves the dialog item without updating the display. Use the "Update" on page 196 method to manually update the display.

Center



The *Center* method moves the dialog to the screen center.

Arguments:

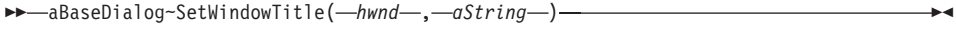
The only argument can be one of:

HIDEWINDOW
Hides the dialog

SHOWWINDOW
Shows the dialog

NOREDRAW
Center the dialog without updating the display. Use the "Update" on page 196 method to manually update the display.

SetWindowTitle



The *SetWindowTitle* method changes the title of a window.

Arguments:
The arguments are:

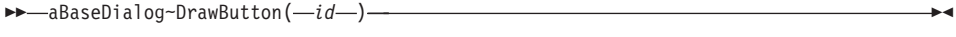
hwnd The handle of the window whose title you want to change

aString
The new title text

Window Draw Methods

The methods listed below are used to draw, redraw, and clear window areas.

DrawButton



The *DrawButton* method draws the given button.

Arguments:
The only argument is:

id The ID of the button

RedrawRect

The *RedrawRect* method immediately redraws a rectangle within the client area of a dialog. You can specify whether the background of the dialog is to be erased before repainting.

Arguments:
The arguments are:

hwnd The handle to the dialog in which parts of the client area are

to be redrawn. See “Get” on page 150 or “GetItem” on page 151 for information on how to get a window handle. If you omit this argument, the handle of the dialog is used.

left, top
The upper left corner of the rectangle relative to the client area, in screen pixels.

right, bottom
The lower right corner of the rectangle relative to the client area, in screen pixels.

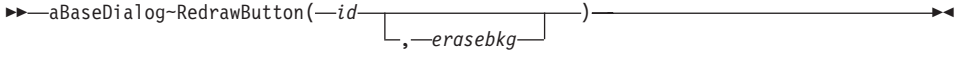
erasebkg
If this argument is 1 or 0, the background of the dialog is erased before redrawing. The default is 0.

Return value:

0 Redrawing was successful.

1 Redrawing failed.

RedrawButton



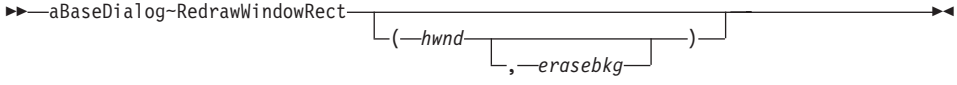
The *RedrawButton* method redraws the given button.

Arguments:
The arguments are:

id The ID of the button

erasebkg
Determines whether (1) or not (0) the background of the drawing area should be erased before redrawing. The default is 0.

RedrawWindowRect



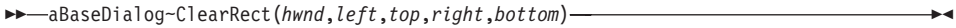
The *RedrawWindowRect* method redraws the given window rectangle.

Arguments:
The arguments are:

hwnd The handle to the window. See "Get" on page 150 or "GetItem" on page 151 for information on how to get a window handle. If you omit this argument, the handle of the dialog is used.

erasebkg
Determines whether (1) or not (0) the background of the drawing area should be erased before redrawing. The default is 0.

ClearRect



The *ClearRect* method clears the given rectangle of a window. The values are in pixels.

Arguments:
The arguments are:

hwnd The handle of the window. See Get or GetItem for how to get a window handle.

left The horizontal value of the upper-left corner of the rectangle

top The vertical value of the upper left corner

right The horizontal value of the lower right corner

bottom
The vertical value of the lower right corner

Example:
The following example clears a rectangle of the size 20 by 20:

```
hwnd=MyDialog~Get
MyDialog~ClearRect(hwnd, 2, 4, 22, 24)
```

ClearButtonRect

The *ClearButtonRect* method erases the draw area of the given button.

Arguments:
The only argument is:

id The ID of the push button

ClearWindowRect

The *ClearWindowRect* method erases the draw area of the given window.

Arguments:

The only argument is:

hwnd The handle of the window. See Get or GetItem for how to get a window handle.

Example:

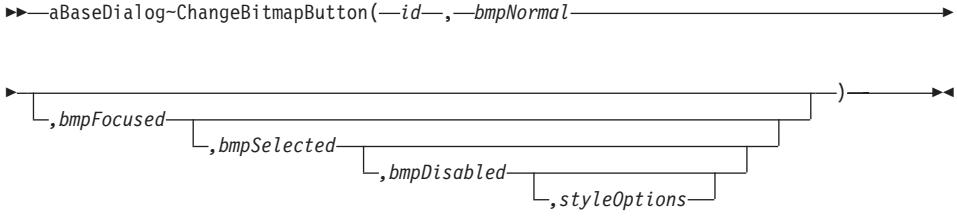
The following example gets the window handle and then clears the window:

```
hwnd = MyDialog~GetItem(211)
MyDialog~ClearWindowRect(hwnd)
```

Bitmap Methods

The methods listed below deal with bitmaps.

ChangeBitmapButton



The *ChangeBitmapButton* method changes the bitmaps of a bitmap button.

Arguments:

The arguments are the same as for ConnectBitmapButton, except for the first argument (msgToRaise), which is skipped in this method.

Example:

The following example replaces the current bitmap with a new bitmap:

```
MyDialog~ChangeBitmapButton(501, "NewBB.bmp")
```

GetBitmapSizeX

The *GetBitmapSizeX* method returns the horizontal bitmap extension.

Arguments:

The only argument is:

id The ID of the bitmap button

GetBitmapSizeY

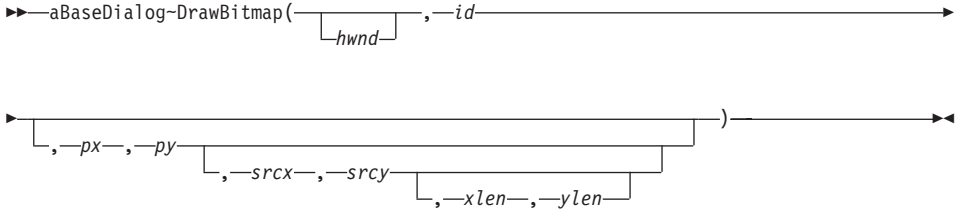
The *GetBitmapSizeY* method returns the vertical bitmap extension.

Arguments:

The only argument is:

id The ID of the bitmap button

DrawBitmap



The *DrawBitmap* method draws the bitmap of a button. You can also use this method to move a bitmap or a part of it.

Arguments:

The arguments are:

hwnd The handle to the window. If this argument is omitted, the handle for the button is used automatically.

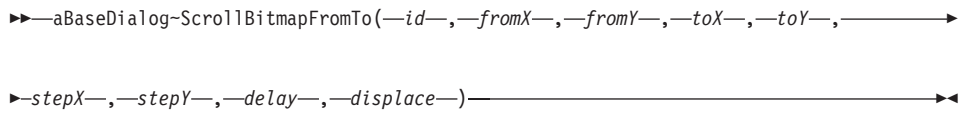
id The ID of the button that has the owner-draw option set

px, py The upper-left corner of the target space within the button (default is 0)

srcx, srcy
The upper-left corner within the bitmap (default is 0)

xlen, yLen
The extension of the bitmap or a part of it (default is the whole bitmap)

ScrollBitmapFromTo



The *ScrollBitmapFromTo* method scrolls a bitmap from one position to another within an owner-drawn button.

Arguments:
The arguments are:

id The ID of the button

fromX, fromY
The starting position

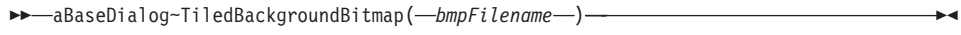
toX, toY
The target position

stepX, stepY
The width of one step

delay The time in milliseconds this method waits after each move before doing the next move. This determines the speed at which the bitmap moves.

displace
If set to 1 the internal position of the bitmap (bitmap displacement) is updated after each incremental move. DisplaceBitmap is called after each step to adjust the bitmap position. If the dialog is redrawn, the bitmap is shown at the correct position, but the drawing is slower.

TiledBackgroundBitmap

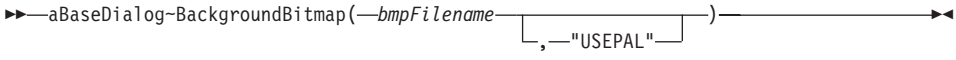


The *TiledBackgroundBitmap* method sets a bitmap as the background brush (Windows NT only). If the bitmap size is less than the size of the background, the bitmap is drawn repetitively.

Arguments:
The only argument is:

bmpFilename
The name of a bitmap file

BackgroundBitmap



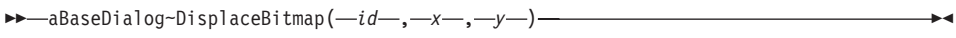
The *BackgroundBitmap* method sets a bitmap as the dialog's background picture.

Arguments:
The arguments are:

bmpFilename
The name of a bitmap file

option Set the last argument to *USEPAL* if you want to use the color palette of the bitmap. See "ConnectBitmapButton" on page 112 for more information.

DisplaceBitmap



The *DisplaceBitmap* method sets the position of a bitmap within a button.

Arguments:
The arguments are:

id The ID of a button

x The horizontal displacement in screen pixels. A negative value can be used.

y The vertical displacement (negative allowed)

Example:
The following example moves the bitmap within a button four screen pixels to the right and three pixels upward:

```
MyBaseDialog~DisplaceBitmap(244, 4, -3)
```

GetBmpDisplacement

The *GetBmpDisplacement* method gets the position of a bitmap within a button.

Arguments:

The only argument is:

id The ID of the button

Example:

The following example shows how to use the GetButtonRect and GetBmpDisplacement methods:

```
bRect = MyBaseDialog~GetButtonRect(244)
parse var bRect left top right bottom
bmpPos = MyBaseDialog~GetBmpDisplacement(244)
parse var bmpPos x y
```

Device Context Methods

The methods listed below are used to retrieve and release a device context.

A device context is associated with a window, a dialog, or a push button, and is a drawing area managed by a window. A device context stores information about the graphic objects (bitmaps, lines, pixels, ...) that are displayed and the tools (pen, brush, font, ...) that are used to display them.

GetWindowDC

The *GetWindowDC* method returns the device context of a window. Do not forget to free the device context after you have completed the operations (see FreeWindowDC).

Arguments:

The only argument is:

hwnd The handle of the window

GetButtonDC

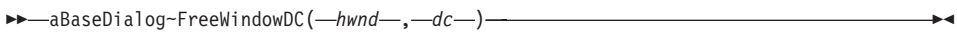
The *GetButtonDC* method returns the device context of a button. Do not forget to free the device context after you have completed the operations (see FreeButtonDC).

Arguments:

The only argument is:

id The ID of the button

FreeWindowDC



The *FreeWindowDC* method frees the device context of a window.

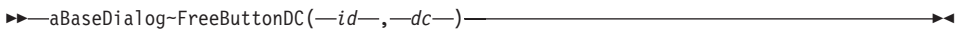
Arguments:

The arguments are:

hwnd The window handle

dc The device context previously received by the GetWindowDC method

FreeButtonDC



The *FreeButtonDC* method releases the device context of a button.

Arguments:

The arguments are:

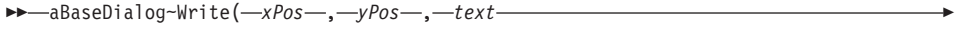
id The ID of the button

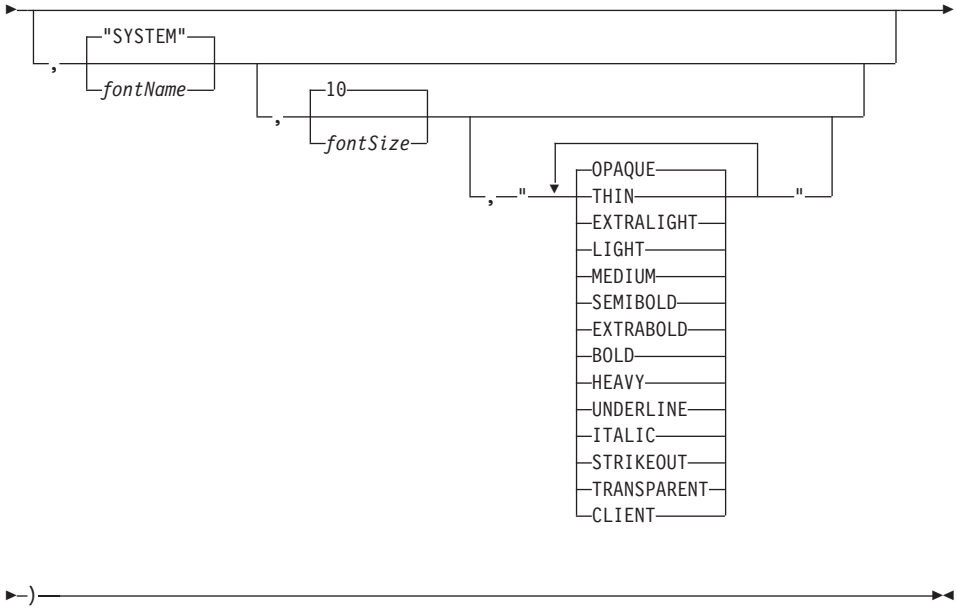
dc The device context previously received by the GetButtonDC method

Text Methods

The following methods are used to display text dynamically in a window area and to modify the state of a device context. See "GetWindowDC" on page 166, "GetDC" on page 209, and "GetButtonDC" on page 166 for information on how to retrieve a device context.

Write



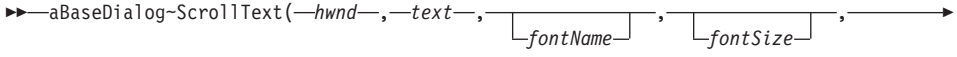


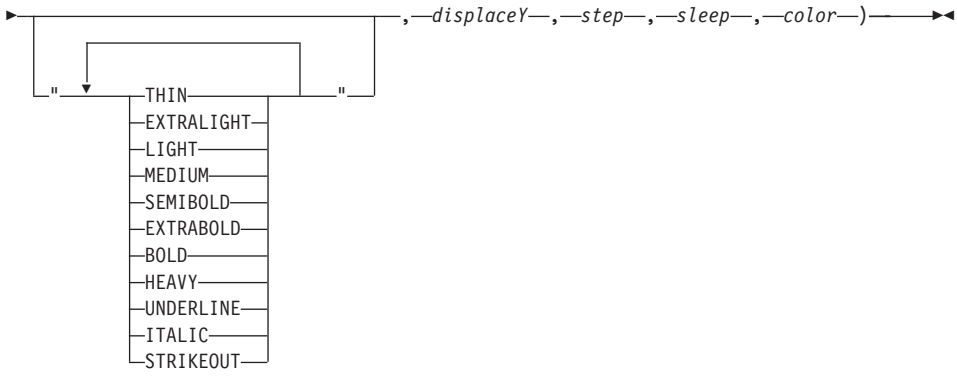
The *Write* method enables you to write text to the dialog in the given font and size, to the given position. This method does not take a handle or an ID; it always writes to the dialog window.

Arguments:

See "WriteToWindow" on page 213 for a description of the other arguments.

ScrollText





The *ScrollText* method scrolls text in a window with the given size, font, and color. The text is scrolled from right to left. If the method is started concurrently, call it a second time to stop scrolling.

Arguments:

The arguments are:

hwnd The handle of the window in which the text is scrolled

text A text string that is scrolled

displaceY
The vertical displacement of the text relative to the top of the window's client area (default 0)

step The size of one step in screen pixels (default 4)

sleep The time in milliseconds that the program waits after each movement (default 10). This determines the speed.

color The color of the text (default 0, black)

See WriteToWindow for a description of the other arguments.

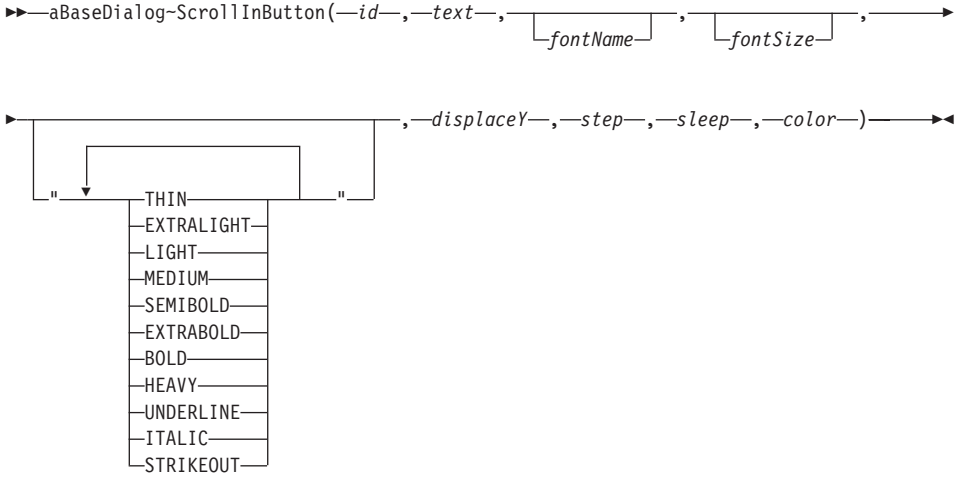
Example:

The following example scrolls the string "Hello world!" from left to right within the given window. The text is located two pixels below the top of the client area, one move is 3 screen pixels, and the delay time after each movement is 15 ms.

```
MyDialog~ScrollText(hwnd, "Hello world!", , , , 2, 3, 15)
```

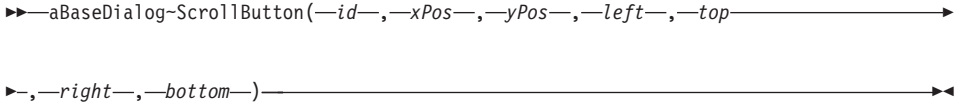
Note: Only one sleep interval can be set for multiple scrolling texts within one process. All scrolling text in one process is synchronized with the first given interval.

ScrollInButton



The *ScrollInButton* method scrolls text within a button. It is similar to the ScrollText method, except that you have to pass an ID instead of a window handle.

ScrollButton



The *ScrollButton* method moves the rectangle within a button. It is used to move bitmaps within buttons.

Arguments:

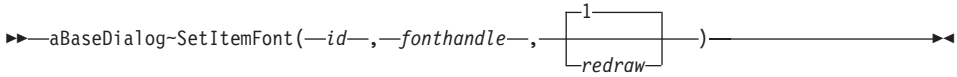
The arguments are:

id The ID of the button

xPos, yPos
The new position of the rectangle (in pixels)

left, top, right, bottom
The extension of the rectangle

SetItemFont



The *SetItemFont* method changes the font for a particular dialog item.

The best place to call this method is within InitDialog. If the font is no longer needed, for instance, when the dialog is closed or another font has been assigned to the dialog item, you should free the font resource by calling DeleteFont. A good place to do this is the Leaving method.

Arguments:

The arguments are:

id The ID of the dialog item.

fonthandle
The handle returned by CreateFont.

redraw

0 Do not redraw the item.

1 Redraw the item, which is the default.

Example:

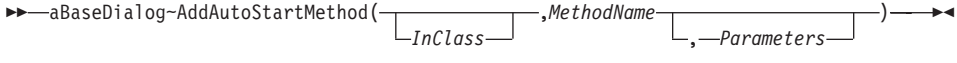
The following example sets a 12-point Arial font for item 101.

```
::method InitDialog
:
   hFont=self~CreateFont("Arial",12)
   self~SetItemFont(101,hFont,0)
```

Animated Buttons

The methods listed below work with animated buttons.

AddAutoStartMethod



The *AddAutoStartMethod* method adds a method name and parameters to a special internal queue. All methods in this queue will be started automatically and run concurrently when the dialog is executed. The given method (*MethodName*) in the given class (*InClass*) is started concurrently with the dialog when the dialog is activated using the Execute or ExecuteAsync method. This is useful for processing animated buttons.

Arguments:

The arguments are:

InClass
The class where the method is defined. If this argument is omitted, the method is assumed to be defined in the dialog class.

MethodName
The name of the method

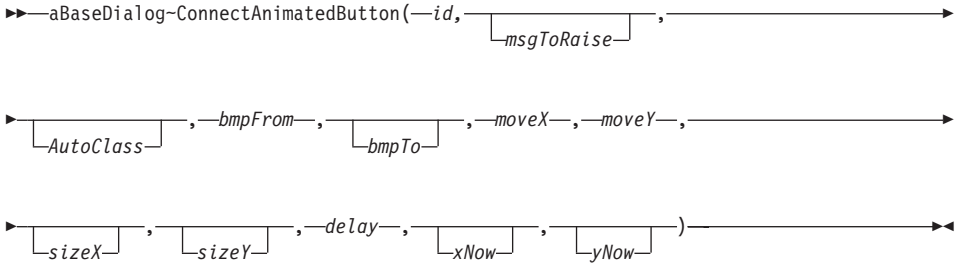
Parameters
All parameters that are passed to this method

Example:

The following example installs the *ExecuteB* method of the *MyAnimatedButton* class so that it is processed concurrently with the dialog execution:

```
MyDialog~AddAutoStartMethod("MyAnimatedButton", "ExecuteB")

::class MyAnimatedButton
::method ExecuteB
  :
```

ConnectAnimatedButton



The *ConnectAnimatedButton* method installs an animated button and runs it concurrently with the main activity.

Arguments:

The arguments are:

id The ID of the button

msgToRaise
The name of a method within the same class. This method is called each time the button is clicked.

AutoClass
The class that controls the animation (default is Chapter 15. AnimatedButton Class)

bmpFrom
The ID of the first bitmap in the animation sequence within a binary resource. It can also be the name of an array stored in the .local directory containing handles of bitmaps to be animated and *bmpTo* is omitted. See LoadBitmap for how to get bitmap handles. The array starts at index 1.

bmpTo
The ID of the last bitmap in the animation sequence within a binary resource. If omitted, *bmpFrom* is expected to be the name of an array stored in .local that holds the bitmap handles of the bitmaps that are to be animated.

moveX, moveY
Size of one move (in pixels)

sizeX, sizeY
Size of the bitmaps (if omitted, the size of the bitmaps is retrieved)

delay The time in milliseconds the method waits after each move

xnow, ynow
The starting position of the bitmap

Example:

The following example defines and runs an animated button. The example loads ten bitmaps ("anibmp1.bmp" to "anibmp10.bmp") into memory and stores them into the array ″My.Bitmaps″ that is stored in the .local directory. The name ″My.Bitmaps″ is specified as the `bmpfrom` and `bmpto` is omitted. After the dialog execution the bitmaps are removed from memory again. The sample also uses a different animation class (″.MyAnimation″) which subclasses from .AnimatedButton and overrides method HitRight which plays a tune each time the animated bitmap hits the right border.

```
/* store array in .local */
.Local["My.Bitmaps"] = .array~new(10)
/* load 10 bitmaps into .local array */
do i= 1 to 10
  .Local["My.Bitmaps"][i] = Dialog~LoadBitmap("anibmp"i".bmp")
  /* you could also use .My.Bitmaps[i] =  ... */
end

/* connect bitmap sequence and .MyAnimated class with button IDANI */
Dialog~ConnectAnimatedButton("IDANI",,.MyAnimation,"My.Bitmaps",,1,1,,,100)

...
```

```
Dialog~Execute
...

/* Free the bitmap previously loaded */
do bmp over .Local["My.Bitmaps"] /* You could also use do bmp over .My.Bitmaps */
   Dialog~RemoveBitmap(bmp)
end


::class MyAnimation subclass AnimatedButton

/* play sound.wav whenever the bitmap hits the right border */

::method HitRight
   ret = Play("sound.wav", yes)
   return self~super:hitright
```

Menu Methods

The methods listed below manipulate a menu connected to the dialog.

ConnectMenuItem

The *ConnectMenuItem* method is called to connect a menu item selection with a method.

Arguments:

The arguments are:

id The ID of the menu item.

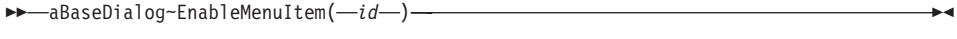
msgToRaise
The name of the method that is to be called.

Example:

See ConnectButton.

Do not use one of the menu item methods below, prior to the *SetMenu* method. If you call one of these methods before SetMenu has been called, the intended action will not be processed and the return code is unpredictable.

EnableMenuItem



The *EnableMenuItem* method is called to enable a menu item.

Arguments:

The only argument is:

id The ID of the menu item to be enabled.

DisableMenuItem

The *DisableMenuItem* method is called to disable a menu item.

Arguments:

The only argument is:

id The ID of the menu item to be disabled.

CheckMenuItem

The *CheckMenuItem* method is called to set the check mark for a menu item.

Arguments:

The only argument is:

id The ID of the menu item to be checked.

UncheckMenuItem

The *UncheckMenuItem* method is called to remove the check mark from a menu item.

Arguments:

The only argument is:

id The ID of the menu item to be unchecked.

GrayMenuItem

The *GrayMenuItem* method is called to disable a menu item. The menu item is grayed.

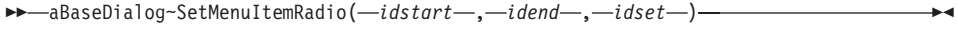
EnableMenuItem is used to reset the grayed state.

Arguments:

The only argument is:

id The ID of the menu item to be grayed.

SetMenuItemRadio



The *SetMenuItemRadio* method is used to change the selection for a radio button menu group.

Arguments:

The arguments are:

idstart The ID of the first menu item in the group.

idend The ID of the last menu item in the group.

idset The ID of the menu item that is to be selected.

Example:

The following example shows how to change the selection within a radio button group. Menu item 102 gets the radio button.

```
self~SetMenuItemRadio(101, 105, 102)
```

GetMenuItemState

The *GetMenuItemState* method returns the state of a given menu item

Arguments:

The only argument is:

id The ID of the menu item whose state is of interest.

Return values:

CHECKED DISABLED GRAYED HIGHLIGHTED

Public Routines

The routines listed below are useful additional functions.

Play

The routine listed below is used to play audio sounds.

The *Play* routine can be used to play an audio file using the Windows multimedia capabilities. See also the *PlaySoundFile* function in "Chapter 6. OODialog External Functions" on page 77.

The file name is looked up in the current directory and in the directories of the *SOUNDPATH* environment variable.

Arguments:

The arguments are:

fileName

The file name of an audio (.WAV) file. The file name is looked up in the directories of the SOUNDPATH environment variable. If this argument is omitted, the currently played sound file is stopped.

option You can set the last argument to:

YES This plays the audio file asynchronously.

LOOP Plays the audio file asynchronously and repeats it in a loop. You can stop the loop by calling Play again omitting all arguments.

Example:

The following example plays a welcoming message:

```
rc = play('Welcome.wav')
```

InfoDialog

Pops up a message box containing the specified text and an OK button.

Argument:

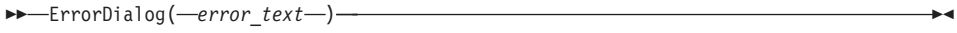
The only argument is:

info_text

Text to be displayed in the message box.

ErrorDialog

Pops up a message box containing the specified text, an OK button, and an error symbol.



Argument:

The only argument is:

error_text
Text to be displayed in the message box.

AskDialog

Pops up a message box containing the specified text, a Yes button, and a No Button.

Arguments:

The only argument is:

question
Text to be displayed in the message box.

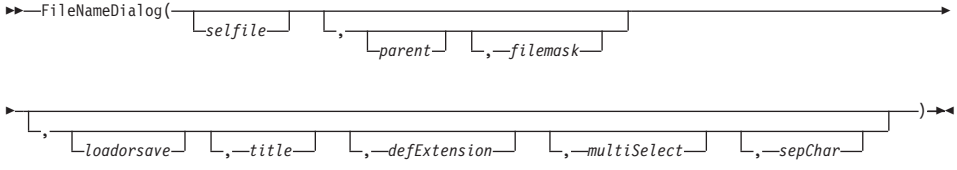
Return Values:

0 The No button has been selected.

1 The Yes button has been selected.

FileNameDialog

Causes a file selection dialog box to appear.



Arguments:

The arguments are:

selfile Preselected file name.

parent Handle to the parent window.

filemask
Pairs of null-terminated filter strings.

The first string in each pair is a display string that describes the filter (for example, "Text Files"), and the second string specifies the filter pattern (for example, "*.TXT"). To specify multiple filter patterns for a single display string, use a semicolon to separate the patterns (for example, "*.TXT;*.DOC;*.BAK"). A pattern string can be a combination of valid file name characters and the asterisk (*) wildcard character. Do not include spaces in the pattern string.

If omitted, `"Text Files (*.TXT)"¦¦'0'x¦¦"*.TXT"¦¦'0'x¦¦"All Files (*.*)"¦¦'0'x¦¦"*.*"` is used as the filter

loadorsave
Specifies which dialog should be displayed.

LOAD
Display the File Open Dialog (default).

SAVE Display the File Save Dialog.

title The window title. The default is ″Open a File″ or ″Save File As″, depending on what is specified for `loadorsave`.

defExtension
The default extension that is added if no extension was specified. The default is TXT.

multiSelect
Specifies if the dialog allows selection of multiple files.

MULTI
Multiple file selection allowed. In this case, `loadorsave` must also be ″LOAD″ or omitted. The result is then path file1 file2 file3 ... If this argument is omitted, you get the selected file name or an empty string when the ″Open File″ window is canceled.

sepChar
Specifies the separation character for the returned path and file names. This is needed for file names with blank characters. If this argument is omitted, the separation character is a blank. If the argument is specified, the returned path and file name uses this separation character. For example, if you specify ″#″ as the separation character, the return string might look as follows:

```
C:\WINNT#file with blank.ext#fileWithNoBlank.TXT
```

Return value:
Returns the selected file name or an empty string when canceled.

FindWindow

Searches the Windows application list for a specific window and returns its handle.

Argument:

The only argument is:

Caption

Caption of the window that is to be searched.

Return value:

Handle of the window or 0 if the window was not found.

Debugging Method

The following method displays the internal setting of the dialog administration table.

Dump

The *Dump* method displays the internal settings of the dialog administration table. This method can be helpful for debugging OODialog programs.

Argument:

The only argument is:

dialogadmin

A pointer to a particular dialog administration record. If you specify this argument, you get detailed information on this record. If you omit this argument, all administration records (one for each dialog of the active process) are listed.

Return value:

This method does not return a value.

Chapter 9. DialogControl Class

The *DialogControl* class provides methods that are common to all dialogs and dialog controls. It is a generic class that serves as a superclass to all dialog-control-specific classes.

Attributes:

Instances of the *DialogControl* class have the following attributes:

FactorX
The horizontal size of one dialog unit, in pixels.

FactorY
The vertical size of one dialog unit, in pixels.

SizeX The width of the dialog, in dialog units.

SizeY The height of the dialog, in dialog units.

Requires:

The *DialogControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *DialogControl* class implement the methods listed in Table 3.

Table 3. DialogControl Instance Methods

Method...	...on page
AbsRect2LogRect	200
AssignFocus	185
AssignWindow	186
CaptureMouse	207
Clear	197
ClearRect	197
ClientToScreen	201
CreateBrush	219
CreateFont	216
CreatePen	220
Cursor_AppStarting	205
Cursor_Arrow	205

Table 3. DialogControl Instance Methods (continued)

Method...	**...on page**
Cursor_Cross	206
Cursor_No	206
CursorPos	203
Cursor_Wait	206
DeleteFont	217
DeleteObject	221
Disable	191
Display	192
Draw	197
DrawAngleArc	225
DrawArc	223
DrawLine	222
DrawPie	225
DrawPixel	223
Enable	190
FillDrawing	225
FontColor	218
FontToDC	218
ForegroundWindow	194
FreeDC	210
GetArcDirection	224
GetClientRect	189
GetDC	209
GetFocus	190
GetID	187
GetMouseCapture	207
GetPixel	223
GetPos	189
GetRect	187
GetSize	189
GetTextSize	215
HScrollPos	202

Table 3. DialogControl Instance Methods (continued)

Method...	**...on page**
Hide	191
HideFast	191
IsMouseButtonDown	208
LoadBitmap	208
LogRect2AbsRect	199
Maximize	193
Minimize	193
Move	195
ObjectToDC	220
OpaqueText	212
ProcessMessage	184
Rectangle	221
Redraw	198
RedrawClient	199
RedrawRect	198
ReleaseMouseCapture	207
RemoveBitmap	209
Resize	193
RestoreCursorShape	204
ScreenToClient	200
Scroll	201
SetArcDirection	224
SetColor	194
SetCursorPos	203
SetFocus	190
SetFont	216
SetHScrollPos	202
SetVScrollPos	203
SetRect	188
SetTitle	197
Show	185
ShowFast	191

Table 3. DialogControl Instance Methods (continued)

Method...	...on page
Title	196
Title=	196
TransparentText	212
Update	196
Value	185
Value=	185
VScrollPos	202
Write	210
WriteToButton	214
WriteToWindow	213

Preparing and Running the Dialog Control

The following methods are used to prepare (initialize) a dialog control, show it, run it, and stop it.

ProcessMessage

```
►►─aDialogControl~ProcessMessage(message,firstParam,secondParam)──────────────►◄
```

The *ProcessMessage* method sends a Windows message to a dialog control.

Arguments:

The arguments are:

message
The number of the message to be sent to the dialog control.

firstParam,secondParam
Additional arguments specific to the message.

Return value:

The return values are message-specific.

Example:

The following example erases the background of the NAME edit control:

```
dlgc = MyDialog~GetEditControl("NAME")
WM_ERASEBACKGROUND = "14"~x2d
hdc = dlgc~GetDc
dlgc~ProcessMessage(WM_ERASEBACKGROUND, hdc, 0)
dlgc~FreeDC(hdc)
```

Note: The Window message numbers are not documented.

AssignFocus

The *AssignFocus* method sets the input focus to the associated dialog control.

Show

The *Show* method makes a dialog control visible and activates it.

Example:

The following example makes the edit control NAME visible and activates it:

```
MyDialog~GetEditControl("NAME")~Show
```

Value

The *Value* method retrieves the current value of a dialog control.

Return value:

The current value set in the dialog control.

Note: See "GetValue" on page 130 for more information.

Value=

The *Value*= method sets a value for a dialog control.

Arguments:

The only argument is:

new_value
The value assigned to the dialog control.

Example
The following example selects check box RESTART and deselects check box VERIFY:

```
MyDialog~GetCheckControl("RESTART")~Value=1
MyDialog~GetCheckControl("VERIFY")~Value=0
```

Note: See "SetValue" on page 131 for more information.

Connect Method

The following method creates a connection between a dialog or dialog control and an object of another class.

AssignWindow

The *AssignWindow* method connects a dialog or dialog control with an existing object of the PlainBaseDialog or DialogControl class. Note that the connected dialog or dialog control might not support all methods provided by the DialogControl class.

Arguments:
The only argument is:

hwnd The handle to the dialog or dialog control that you want to assign to the DialogControl object.

Return value:
The handle to the dialog or dialog control that has been assigned, or 0 if the connection failed.

Example:
The following example searches the desktop for a dialog with the title "Monitoring Applications", connects it to the object `dlgc` of the DialogControl class, and then minimizes the dialog:

```
dlgc = .DialogControl~new
...
whnd = FindWindow("Monitoring Application")
if whnd \= 0 then do
   dlgc~AssignWindow(whnd)
   dlgc~Display("MIN")
end
```

Get and Set Methods

Get methods are used to retrieve the data from all or individual controls of a dialog. Set methods are used to set the values of all or individual controls, without changing the associated Object REXX attributes.

GetID

The *GetID* method retrieves the identification number of the associated dialog or dialog control.

Return value:
The numeric ID.

Example:
The following example displays 1 in most cases:

```
say MyDialog~GetButtonControl("IDOK")~GetID
```

GetRect

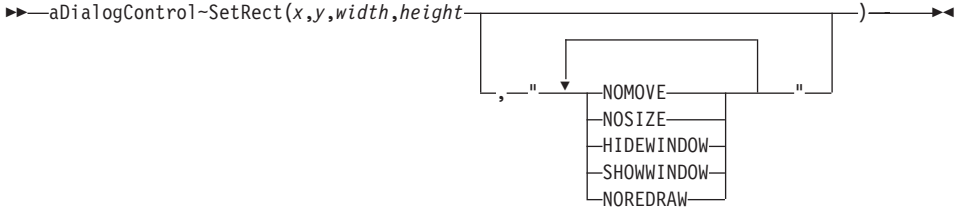
Retrieves the dimensions of the rectangle surrounding the associated dialog or dialog control. The coordinates are relative to the upper left corner of the screen and are specified in screen pixels. The order is: left, top, right, bottom; where 'left' and 'top' are the x and y coordinates of the upper left-hand corner of the rectangle, and 'right' and 'bottom' are the coordinates of the bottom right-hand corner.

Return value:
The coordinates of the dialog or dialog control, separated by blanks.

Example:
The following example calculates the width and height of an entry line:

```
parse value MyDialog~GetEditControl("Name")~GetRect with left top,
right bottom
width = right - left
height = bottom - top
```

SetRect



The *SetRect* method sets new coordinates for the associated dialog or dialog control.

Arguments:

The arguments are:

x,y The new position of the upper left corner, in screen pixels.

width The new width of the dialog or dialog control, in screen pixels.

height The new height of the dialog or dialog control, in screen pixels.

showOptions

One or more of the following keywords, separated by blanks:

NOMOVE

The upper left position of the dialog or dialog control is not changed.

NOSIZE

The size of the dialog or dialog control is not changed.

HIDEWINDOW

The dialog or dialog control is to be made invisible.

SHOWWINDOW

The dialog or dialog control is to be made visible.

NOREDRAW

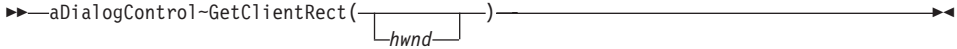
The dialog or dialog control is to be repositioned without redrawing it.

Return value:

0 Repositioning was successful.

1 Repositioning failed.

GetClientRect



The *GetClientRect* method returns the client rectangle of a dialog or dialog control in screen pixels. The client coordinates specify the upper left and lower right corners of the client area. Because the client coordinates are relative to the upper left corner of the client area of a dialog or dialog control, the coordinates of the upper left corner are (0,0).

Arguments:

The only argument is:

hwnd The handle to a dialog or dialog control. If this argument is omitted, the dimensions of the associated dialog or dialog control are returned.

Return value:

The client rectangle in the format "*left top right bottom*", separated by blanks.

GetPos

The *GetPos* method returns the coordinates of the upper left corner of the dialog or dialog control, in dialog units.

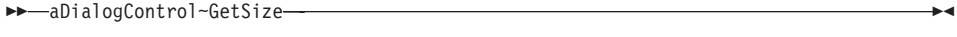
Return value:

The horizontal and vertical position, separated by a blank.

Example:

For an example, see "Move" on page 195.

GetSize



The *GetSize* method returns the width and height of the dialog or dialog control, in dialog units.

Return value:

The width and height, separated by a blank.

Example:

For an example, see "Resize" on page 193.

GetFocus

```
►►—aDialogControl~GetFocus———————————————————————————————►◄
```

The *GetFocus* method returns the handle to the dialog or dialog control that has currently the input focus.

Return value:

The handle to the dialog or dialog control that has the input focus, or 0 if this method failed.

SetFocus

```
►►—aDialogControl~SetFocus(hwnd)—————————————————————————►◄
```

The *SetFocus* method sets the input focus to a dialog or dialog control.

Arguments:

The only argument is:

hwnd The handle to the dialog or dialog control that is to receive the input focus.

Return value:

The handle to the dialog or dialog control that had the focus before, or 0 if this method failed.

Appearance Modification Methods

The following methods are to change the appearance of the dialog itself or one of its controls. The list contains methods to change the size, position, visibility, and title (header).

Enable

```
►►—aDialogControl~Enable—————————————————————————————————►◄
```

The *Enable* method enables a dialog or dialog control to accept user interaction.

Example:

```
MyDialog~GetEditControl("Name")~Enable
```

Disable

The *Disable* method disables a dialog or dialog control.

Example:

```
MyDialog~GetEditControl("Name")~Disable
```

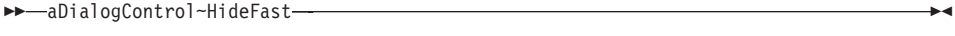
Hide

The *Hide* method makes a dialog or dialog control invisible and activates another dialog or dialog control.

Example:

```
MyDialog~GetEditControl("NAME")~Hide
```

HideFast



The *HideFast* method marks a dialog or dialog control as invisible but does not redraw it. Send the Update method (see page 196) to the dialog or dialog control to force a redraw.

Example:

```
MyDialog~GetEditControl("NAME")~HideFast
...
MyDialog~Update
```

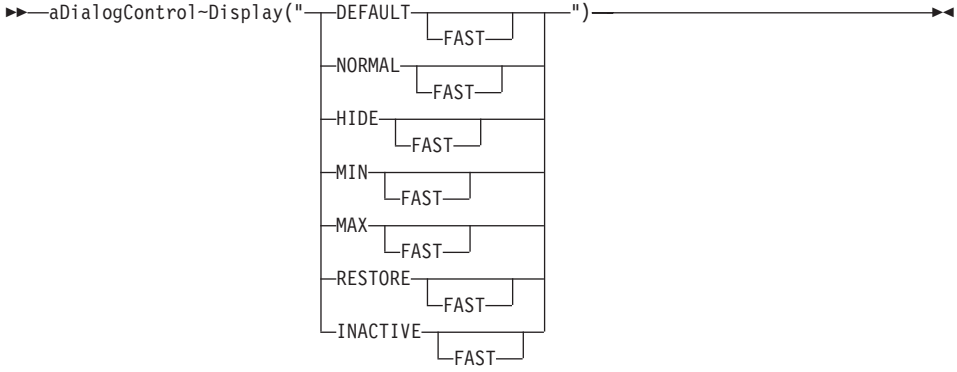
ShowFast

The *ShowFast* method marks a dialog or dialog control as visible but does not redraw it. Send the Update method (see page 196) to the dialog or dialog control to force a redraw.

Example:

```
MyDialog~GetEditControl("NAME")~ShowFast
...
MyDialog~Update
```

Display



The *Display* method displays a dialog or dialog control as specified.

Argument:

This argument must be one of the following keywords:

DEFAULT
Displays the dialog or dialog control in its default state.

NORMAL
Same as calling the Show method (see page 105).

HIDE Same as calling the Hide method (see page 191).

MIN Minimizes the dialog.

MAX Maximizes the dialog.

RESTORE
Activates and displays the dialog. If the dialog is minimized or maximized, it is restored to its original size and position.

INACTIVE
Displays the dialog or dialog control in its current state. The active dialog or dialog control remains active.

To each keyword you can add **FAST**, separated by a blank, to suppress the redrawing of the dialog or dialog control.

Example:

The following statement minimizes the dialog without redrawing it.

```
MyDialog~GetTreeControl("FILES")~Display("MIN FAST")
```

Minimize

The *Minimize* method minimizes the dialog.

Return value:

0 Minimizing was successful.

1 Minimizing failed.

Maximize

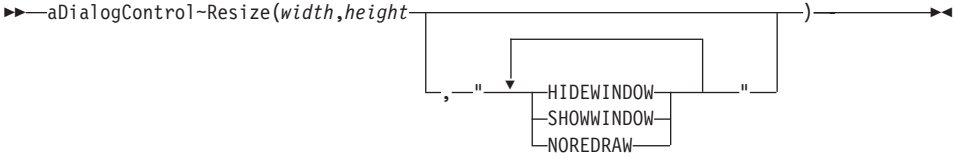
The *Maximize* method maximizes the dialog.

Return value:

0 Maximizing was successful.

1 Maximizing failed.

Resize



The *Resize* method resizes a dialog or dialog control.

Arguments:

The arguments are:

width The new width of the dialog or dialog control, in dialog units.

height The new height of the dialog or dialog control, in dialog units.

showOptions

One or more of the following keywords, separated by blanks:

HIDEWINDOW

The dialog or dialog control is to be made invisible.

SHOWWINDOW

The dialog or dialog control is to be made visible.

NOREDRAW
The dialog or dialog control is to be repositioned without redrawing it.

Return value:

0 Resizing was successful.

1 Resizing failed.

Example:

The following example resizes the tree view control FILES almost to the size of the window and displays the new size:

```
obj = MyDialog~GetTreeControl("FILES")
if obj = .Nil then return
obj~Resize(MyDialog~SizeX -10, MyDialog~SizeY -20)
parse value obj~GetSize with width height
say "New width of window is" width "and new height is" height
```

SetColor

```
►►—aDialogControl~SetColor(background,foreground)————————————————————►◄
```

The *SetColor* method sets the background and foreground colors of the dialog control.

Arguments:

The arguments are:

background
The color number of the background color.

foreground
The color number of the foreground color.

Return value:

0 The color has been assigned.

1 The selected color was already assigned.

Example:

The following example sets the background color of list box FILES to blue:

```
MyDialog~GetListBox("FILES")~SetColor(4, 15)
```

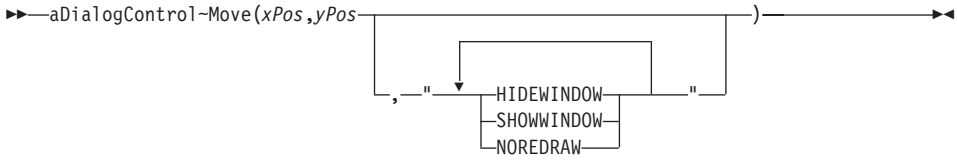
ForegroundWindow

```
►►—aDialogControl~ForegroundWindow————————————————————————————————►◄
```

The *ForegroundWindow* method returns the handle to the current foreground window.

Return value:
The handle to the foreground window, or 0 if this method failed.

Move



The *Move* method moves the associated dialog or dialog control to the specified position.

Arguments:
The arguments are:

xPos The new horizontal position of the dialog or dialog control, in dialog units.

yPos The new vertical position of the dialog or dialog control, in dialog units.

showOptions
One or more of the following keywords, separated by blanks:

HIDEWINDOW
The dialog or dialog control is to be made invisible.

SHOWWINDOW
The dialog or dialog control is to be made visible.

NOREDRAW
The dialog or dialog control is to be repositioned without redrawing it.

Return value:

0 Moving was successful.

1 Moving failed.

Example:
The following example repositions the tree view control FILES to the upper left corner of the window and displays the new position:

```
obj = MyDialog~GetTreeControl("FILES")
if obj = .Nil then return
obj~Move(1,1)
parse value obj~GetPos with x y
say "New horizontal position of window is" x "and new vertical position is" y
```

Update

The *Update* method makes the contents of the dialog or dialog control invalid and therefore forces it to be updated.

Title

The *Title* method retrieves the title of the dialog or dialog control.

Return value:

0 The title was retrieved.

1 Retrieving the title failed.

Title=

The *Title*= method sets the title of the dialog or dialog control.

Arguments:

The only argument is:

new_text

A text string that is to be used as the title or text of the dialog or dialog control.

Return value:

0 The title was set.

1 Setting the title failed.

Example:

The following example changes the label of radio button CHOICE2:

```
MyDialog~GetRadioControl("CHOICE2")~Title="&Object REXX (preferred choice)"
MyDialog~Redraw
```

SetTitle

The *SetTitle* method sets the title of the dialog or dialog control. It is equal to "Title=" on page 196.

Draw Methods

The following methods are used to draw, redraw, and clear a dialog or dialog control.

Draw

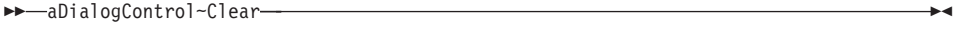
The *Draw* method draws the dialog or dialog control.

Return value:

0 Drawing was successful.

1 Drawing failed.

Clear



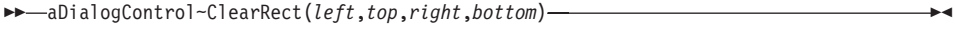
The *Clear* method draws the dialog or dialog control using the background brush.

Return value:

0 Clearing was successful.

1 Clearing failed.

ClearRect



The *ClearRect* method draws the dialog or dialog control using the background brush.

Arguments:

The arguments are:

left,top
The upper left corner of the rectangle, in screen pixels.

right,bottom
The lower right corner of the rectangle, in screen pixels.

Return value:

0 Drawing was successful.

1 Drawing failed.

Redraw

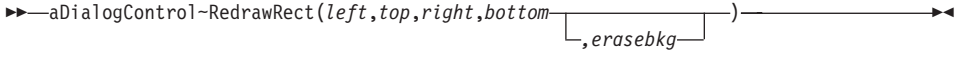
The *Redraw* method redraws the dialog or dialog control immediately.

Return value:

0 Redrawing was successful.

1 Redrawing failed.

RedrawRect



The *RedrawRect* method immediately redraws the rectangle of the client area of the associated dialog. You can specify whether the background of the dialog is to be erased before repainting.

Arguments:

The arguments are:

left,top
The upper left corner of the rectangle relative to the client area, in screen pixels.

right,bottom
The lower right corner of the rectangle relative to the client area, in screen pixels.

erasebkg
If this argument is 1 or "Y", the background of the dialog is erased before repainting.

Return value:

0 Redrawing was successful.

1 Redrawing failed.

RedrawClient

The *RedrawClient* method immediately redraws the entire client area of the dialog or dialog control. You can specify whether the background of the dialog or dialog control is to be erased before redrawing.

Arguments:

The only argument is:

erasebkg

If you specify 1 or "Y", the background of the dialog is erased before redrawing.

Return value:

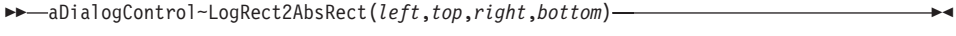
0 Redrawing was successful.

1 Redrawing failed.

Conversion Methods

The following methods are used to convert and map coordinates of a dialog or dialog control.

LogRect2AbsRect



The *LogRect2AbsRect* method converts the coordinates from dialog units to screen pixels.

Arguments:

The arguments are:

left,top

The position of the upper left corner, in dialog units.

right,bottom

The position of the lower right corner, in dialog units.

Return value:

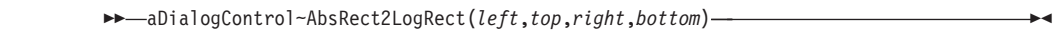
A compound variable that stores the four screen pixel coordinates.

The position of the upper left corner is stored in RetStem.left and RetStem.top. The position of the lower right corner is stored in RetStem.right and RetStem.bottom. The tails left, top, right, and bottom must be uninitialized symbols.

Example:

```
absrect. = MyDialog~LogRect2AbsRect(5, 5, 10, 10)
say "Screen pixel rectangle=" absrect.left "," absrect.top ",",
absrect.right "," absrect.bottom
```

AbsRect2LogRect



The *AbsRect2LogRect* method converts the coordinates from screen pixels to dialog units.

Arguments:

The arguments are:

left,top
The position of the upper left corner, in screen pixels.

right,bottom
The position of the lower right corner, in screen pixels.

Return value:

A compound variable that stores the four screen dialog units. The position of the upper left corner is stored in RetStem.left and RetStem.top. The position of the lower right corner is stored in RetStem.right and RetStem.bottom. The tails left, top, right, and bottom must be uninitialized symbols.

Example:

```
rectdunit. = MyDialog~AbsRect2LogRect(20, 20, 40, 40)
say "Dialog unit rectangle=" rectdunit.left "," rectdunit.top ",",
rectdunit.right "," rectdunit.bottom
```

ScreenToClient

The *ScreenToClient* method maps the coordinates relative to the upper left corner of the screen, to a location within the client area relative to the upper left corner of the dialog's client area.

Arguments:

The arguments are:

x The horizontal position, in screen pixels.

y The vertical position, in screen pixels.

Return value:
The horizontal and vertical positions of the specified location relative to the upper left corner of the client area, separated by a blank.

ClientToScreen

The *ClientToScreen* method maps the coordinates relative to the dialog's client area to the coordinates relative to the upper left corner of the screen.

Arguments:
The arguments are:

x The horizontal position in screen pixels.

y The vertical position in screen pixels.

Return value:
The horizontal and vertical positions of the specified location relative to the location (0,0) of the screen, separated by a blank.

Scroll Methods

The following methods are used to scroll a dialog or dialog control and to set scroll bars.

Scroll

The *Scroll* method scrolls the contents of the associated dialog or dialog control by the amount specified.

Arguments:
The arguments are:

cx The number of screen pixels the content of the dialog or dialog control is to be scrolled to the right or to the left, if negative.

cy The number of screen pixels the content of the dialog or dialog control is to be scrolled downward or upward, if negative.

Return value:

0 Scrolling was successful.

1 Scrolling failed.

HScrollPos

The *HScrollPos* method returns the position of the horizontal scroll bar in the associated dialog or dialog control.

Return value:

The position of the horizontal scroll bar.

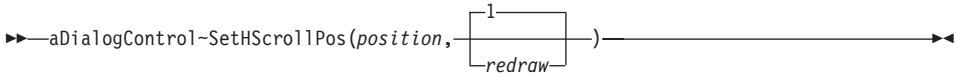
VScrollPos

The *VScrollPos* method returns the position of the vertical scroll bar in the associated dialog or dialog control.

Return value:

The position of the vertical scroll bar.

SetHScrollPos



The *SetHScrollPos* method sets the thumb position of the horizontal scroll bar contained in the associated dialog or dialog control.

Arguments:

The arguments are:

position

The new thumb position of the horizontal scroll bar.

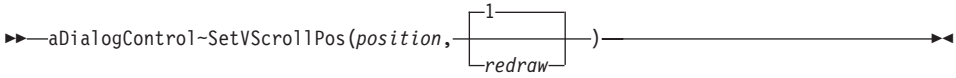
redraw

If this argument is 1 (the default), the display of the scroll bar is updated.

Return value:

The previous position of the horizontal scroll bar, or 0 if this method failed.

SetVScrollPos



The *SetVScrollPos* method sets the thumb position of the vertical scroll bar contained in the associated dialog or dialog control.

Arguments:

The arguments are:

position

The new thumb position of the vertical scroll bar.

redraw

If this argument is 1 (the default), the display of the scroll bar is updated.

Return value:

The previous position of the vertical scroll bar, or 0 if this method failed.

Mouse and Cursor Methods

The following methods are used to position and shape the mouse cursor and to capture the mouse.

CursorPos

The *CursorPos* method returns the current position of the mouse cursor.

Return value:

The horizontal and vertical position of the mouse, separated by a blank.

Example:

See "SetCursorPos" for an example on how to use this method.

SetCursorPos

The *SetCursorPos* method moves the mouse cursor to the specified position. This method can be used to force the repainting of the mouse cursor or to keep the mouse cursor within a specific rectangle.

Arguments:

The arguments are:

x The horizontal position of the mouse cursor, in screen pixels.

y The vertical position of the mouse cursor, in screen pixels.

Return value:

0 Moving the mouse cursor was successful.

1 Moving the mouse cursor failed.

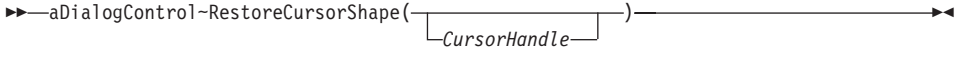
Example

The following example shows two methods: one indicating that processing has started and one indicating that processing has completed. The method IndicateBeginProcessing changes the shape of the mouse cursor to the WAIT cursor and IndicateEndProcessing restores the original mouse cursor shape. Both methods retrieve the current position of the mouse cursor and move it by one screen pixel in each direction to force the repainting of the mouse cursor.

```
::method IndicateBeginProcessing
  self~Current_Cursor = self~Cursor_Wait
  parse value self~CursorPos with curx cury
  self~SetCursorPos(curx+1, cury+1)

::method IndicateEndProcessing
  self~RestoreCursorShape(self~Current_Cursor)
  parse value self~CursorPos with curx cury
  self~SetCursorPos(curx-1, cury-1)
```

See "DefListDragHandler" on page 321 for another example on how to use the mouse methods.

RestoreCursorShape



The *RestoreCursorShape* method restores the shape of the mouse cursor.

Arguments:

The only argument is:

CursorHandle

The handle to the mouse cursor shape returned by the

Cursor_Arrow, Cursor_AppStarting, Cursor_Cross, Cursor_No, or Cursor_Wait method.

If you omit this argument, the cursor shape is set to an arrow. Therefore, it is recommended that you store the original mouse cursor shape by specifying its handle when you change its shape.

Return value:

The handle to the current cursor shape, that is, the shape that was used before the cursor was restored to the given shape.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

Cursor_Arrow

The *Cursor_Arrow* method sets the shape of the mouse cursor to the standard arrow. The new shape is only used when the mouse cursor is within the rectangle of the associated dialog or dialog control.

Return value:

The handle to the current cursor shape, that is, the shape that was used before the arrow shape was set.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

Cursor_AppStarting

The *Cursor_AppStarting* method sets the shape of the mouse cursor to the standard arrow with a small hourglass. The new shape is only used when the mouse cursor is within the rectangle of the associated dialog or dialog control.

Return value:

The handle to the current cursor shape, that is, the shape that was used before the arrow shape with the hourglass was set.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

Cursor_Cross

```
►►—aDialogControl~Cursor_Cross————————————————————————————————————►◄
```

The *Cursor_Cross* method sets the shape of the mouse cursor to a crosshair. The new shape is only used when the mouse cursor is within the rectangle of the associated dialog or dialog control.

Return value:

The handle to the current shape, that is, the shape that was used before the crosshair cursor shape was set.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

Cursor_No

```
►►—aDialogControl~Cursor_No———————————————————————————————————————►◄
```

The *Cursor_No* method sets the shape of the mouse cursor to a slashed circle to deny access. The new shape is only used when the mouse cursor is within the rectangle of the associated dialog or dialog control.

Return value:

The handle to the current shape, that is, the shape that was used before the slashed-circle cursor shape was set.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

Cursor_Wait

```
►►—aDialogControl~Cursor_Wait————————————————————————————————————►◄
```

The *Cursor_Wait* method sets the shape of the mouse cursor to the hourglass. The new shape is only used when the mouse cursor is within the rectangle of the associated dialog or dialog control.

Return value:

The handle to the current shape, that is, the shape that was used before the hourglass shape was set.

Example:

See "SetCursorPos" on page 203 for an example on how to use this method.

GetMouseCapture

The *GetMouseCapture* method retrieves the dialog or dialog control that captured the mouse. This dialog or dialog control receives the entire mouse input regardless of whether the mouse cursor is within the borders of the dialog or dialog control.

Return value:

The handle to the dialog or dialog control that captures the mouse, or 0 if the mouse is not captured.

CaptureMouse

The *CaptureMouse* method captures the mouse. This means that the associated dialog or dialog control receives the entire mouse input regardless of whether the mouse cursor is within the borders of the dialog or dialog control.

Return value:

The handle to the dialog or dialog control that previously captured the mouse, or 0 if the mouse was not captured before.

Note: If you change the cursor shape while the mouse is being captured, this change is ignored.

ReleaseMouseCapture

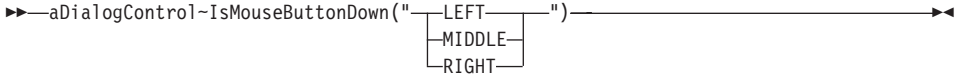
The *ReleaseMouseCapture* method releases the mouse capture from a dialog or dialog control and restores normal mouse input processing. This means that the mouse input is then received by another dialog or dialog control that captured the mouse.

Return value:

0 The mouse capture was released.

1 Releasing the mouse capture failed.

IsMouseButtonDown



The *IsMouseButtonDown* method retrieves information on whether a mouse button is pressed.

Arguments:

The only argument is:

button

The location of the mouse button you are interested in.

Return value:

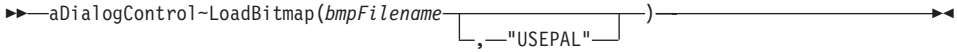
0 The button is not being pressed.

1 The button is being pressed.

Bitmap Methods

The following methods load and release bitmaps.

LoadBitmap



The *LoadBitmap* method loads a bitmap from a file into memory and returns the handle to the bitmap.

Arguments:

The arguments are:

bmpFilename

The name of a bitmap file. The name can also include a relative or absolute path.

USEPAL

Sets the color palette of the bitmap as the system color palette.

Example:

The following example loads into memory the bitmap file, `Walker.bmp`, which is located in the `BMP` subdirectory. *hBmp* is the handle to this in-memory bitmap.

```
hBmp = MyDialog~LoadBitmap("bmp\Walker.bmp", "USEPAL")
```

Note: Do not forget to call the RemoveBitmap method to free memory when the bitmap is no longer in use. You have to specify the INMEMORY option when using the ConnectBitmapButton or ChangeBitmapButton method.

RemoveBitmap

The *RemoveBitmap* method frees an in-memory bitmap that was loaded by LoadBitmap.

Arguments:

The only argument is:

hBitmap
The bitmap handle.

Device Context Methods

The following methods are used to retrieve and release a device context (DC).

A device context (DC) is associated with a dialog or dialog control, and is a drawing area managed by a dialog or dialog control. It stores information about graphic objects (such as bitmaps, lines, and pixels that are displayed) and the tools (such as pens, brushes, and fonts) that are used to display them.

GetDC

The *GetDC* method reserves drawing resources and returns the handle to the display device context of a dialog or dialog control.

Return value:

The handle to the device context, or 0 if this method failed.

Example:

The following example retrieves the device context of button DRAWINGS, processes the drawing commands, and frees the device context resources:

```
obj = MyDialog~GetButtonControl("DRAWINGS")
if obj = .Nil then return -1
dc = obj~GetDC
if dc = 0 then return -1
... /* draw something */
obj~FreeDC(dc)
```

Note: When you have finished with the device context, call FreeDC.

FreeDC

The *FreeDC* method releases the device context resources that were reserved for GetDC.

Arguments:

The only argument is:

dc The handle to the device context that is to be released.

Return value:

0 The device context resources were released.

1 Releasing the device context resources failed.

Example:

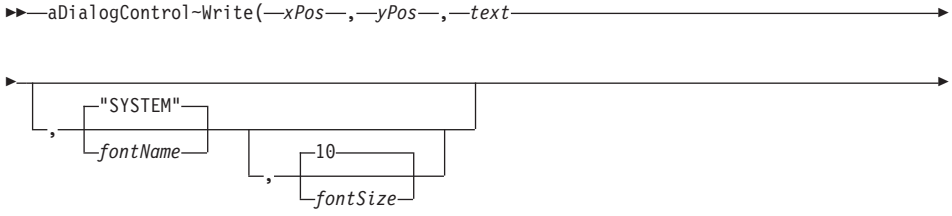
See "GetDC" on page 209 for an example.

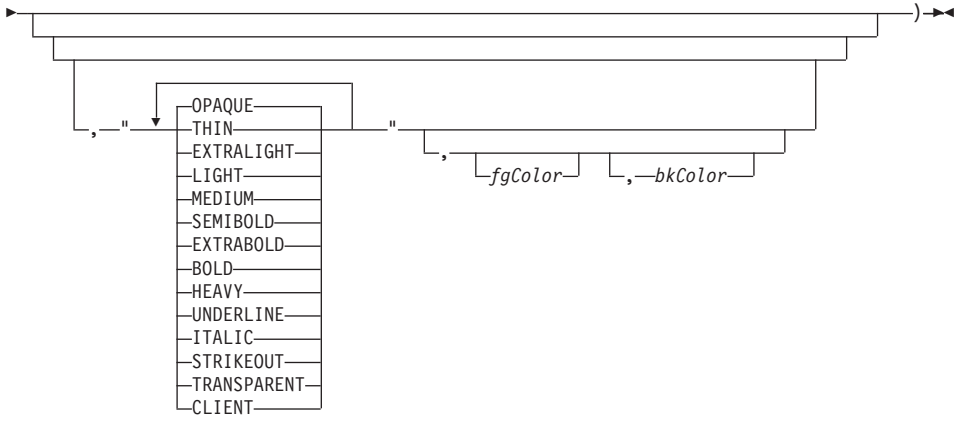
Note: Always call this method when you have finished with the device context.

Text Methods

The following methods are used to display text dynamically in a window area and to modify the state of a device context. See "GetWindowDC" on page 166, "GetDC" on page 209, and "GetButtonDC" on page 166 for information on how to retrieve a device context.

Write





The *Write* method writes the specified text to the device context associated with the dialog or dialog control in the given font, style, and color at the given position.

Arguments:

The arguments are:

xPos, yPos
The starting position of the text, in pixels.

text The string that you want to write to the dialog or dialog control.

fontName
The name of a font. If omitted, the SYSTEM font is used.

fontSize
The size of the font. If omitted, the standard size (10) is used.

fontStyle
One or more of the keywords listed in the syntax diagram, separated by blanks:

TRANSPARENT
This style writes the text without clearing the background.

OPAQUE
This style, which is the default, clears the background with the given background color, or with white if the background color is omitted, before writing the text.

CLIENT
The device context of the client area of the dialog or

dialog control is used instead of the device context of the entire dialog or dialog control.

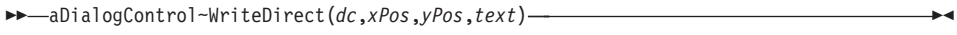
fgcolor
The color index of the text color.

bkColor
The color index of the background color. The background color is not used in transparent mode.

Example:
The following example writes the string "Hello world!" to the dialog using a blue 24pt Arial font in bold and transparent, italic style:

```
MyDialog~Write(5, 5, "Hello world!", "Arial", 24,,
"BOLD ITALIC TRANSPARENT CLIENT", 4)
```

WriteDirect



The *WriteDirect* method enables you to write text to a device context at a given position.

Arguments:
The arguments are:

dc A device context.

xPos, yPos
The position where the text is placed, in pixels.

text The string you want to write to the dialog or dialog control.

TransparentText

The *TransparentText* method enables you to write text to a device context using WriteDirect in transparent mode, that is, without a white background behind the text. Restore the default mode using “OpaqueText”.

Arguments:
The only argument is:

dc A device context.

OpaqueText

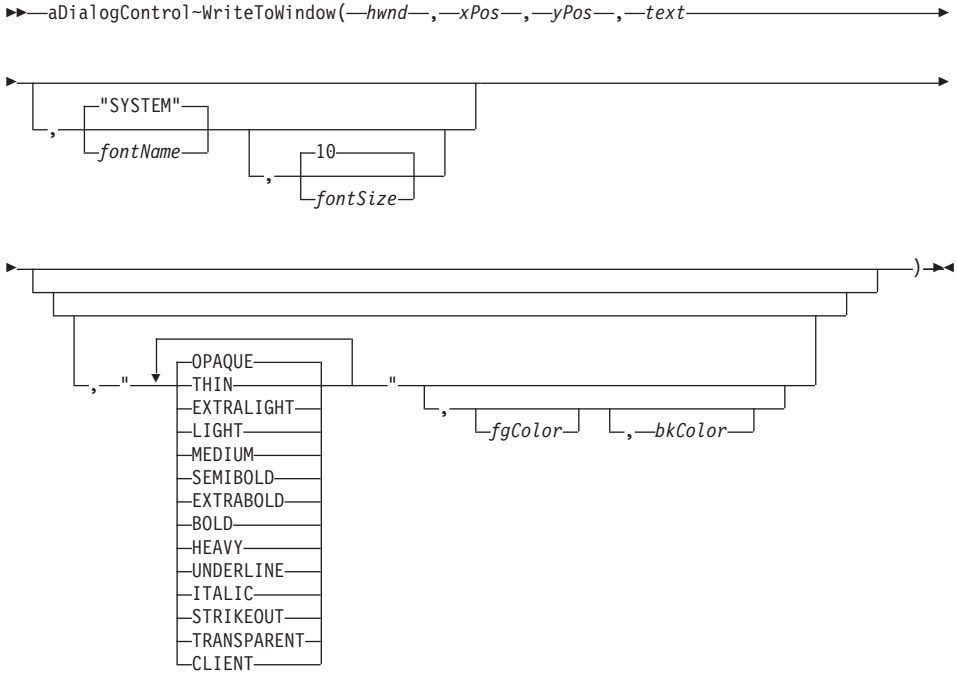
The *OpaqueText* method restores the default text mode, that is, with a white background behind the text, which overlays whatever is at that position in the dialog or dialog control. Use this method after transparent mode was set using "TransparentText" on page 212.

Arguments:

The only argument is:

dc A device context.

WriteToWindow



The *WriteToWindow* method enables you to write text to a dialog or dialog control in the given font and size to the given position.

Arguments:

The arguments are:

hwnd The handle of the dialog or dialog control. See "Get" on page 150 for how to get a valid handle.

xPos, yPos
The starting position of the text, in pixels.

text The string you want to write to the dialog or dialog control.

fontName
The name of a font. If omitted, the SYSTEM font is used.

fontSize
The size of the font. If omitted, the standard size (10) is used.

fontStyle
One or more of the keywords listed in the syntax diagram, separated by blanks:

TRANSPARENT
This style writes the text without clearing the background.

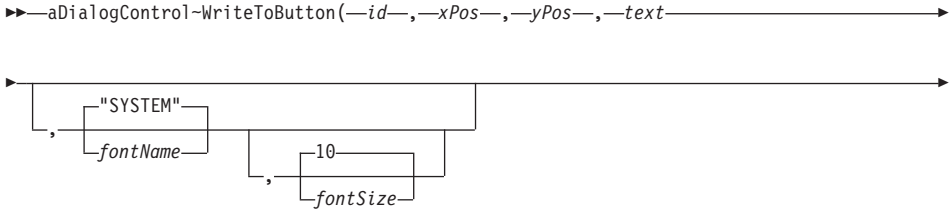
OPAQUE
This style, which is the default, clears the background with the given background color, or with white if the background color is omitted, before writing the text.

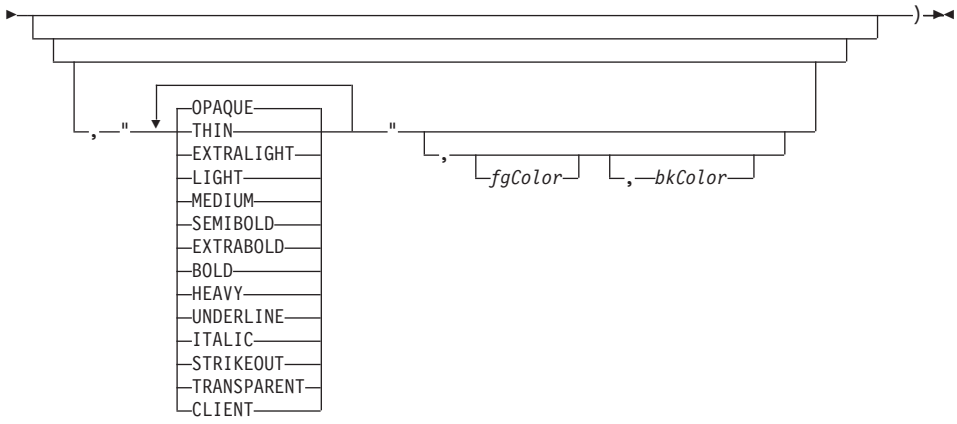
CLIENT
The device context of the dialog's client area is used instead of the device context of the entire dialog.

Example:
This example writes the string ″Hello world!″ to the dialog window using a 24pt Arial font in bold and italic style:

```
hwnd=MyDialog~Get
MyDialog~WriteToWindow(hwnd, 23, 15, "Hello world!", ,
                       "Arial", 24, "BOLD ITALIC")
```

WriteToButton





The *WriteToButton* method enables you to write text to a button in the given font and size to the given position.

Arguments:

The arguments are:

id The ID of a button.

See "WriteToWindow" on page 213 for a description of the other arguments.

GetTextSize

```
aDialogControl~GetTextSize(text,fontname,fontsize,hwnd)
```

The *GetTextSize* method returns the width and height that the specified text requires in the font and size given.

Arguments:

The arguments are:

text The text for which the dimensions are to be returned.

fontname
The name of the font used in the device context (DC).

fontsize
The size of the font used in the device context (DC).

hwnd The handle to the dialog or dialog control that is the owner of the device context (DC).

Return value:

The width and height of the text, in dialog units, separated by a blank.

Example:

The following example stores the space required by the specified text in the device context of MyButton, in cx and cy:

```
parse value MyButton~GetTextSize("This is the output!") with cx cy
```

SetFont

The *SetFont* method assigns another font to the text in a dialog or dialog control.

Arguments:

The arguments are:

fontHandle

The handle to the font that is to be used by the dialog or dialog item. Use "CreateFont" to get this handle.

redraw

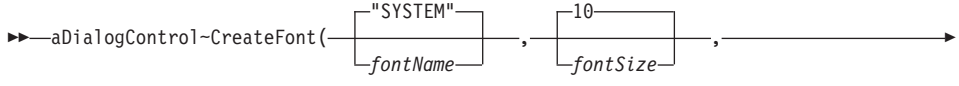
If you specify 1, the dialog or dialog item is redrawn.

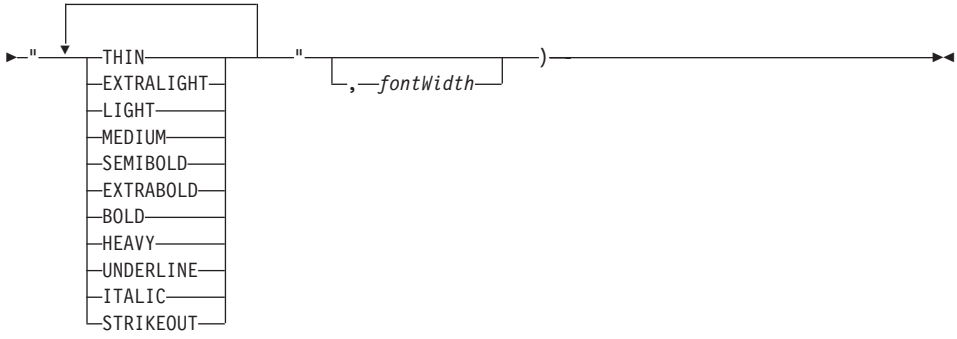
Example:

The following example creates the font Arial with a pitch size of 14 and assigns it to the tree view control FILES, which is forced to be redrawn.

```
hfnt = Mydialog~CreateFont("Arial", 14)
MyDialog~GetTreeControl("FILES")~SetFont(hfnt, 1)
```

CreateFont





The *CreateFont* method creates a font. It returns a handle that you can use in the FontToDC method (see page 218) to activate the font in a device context or in the SetItemFont method (see page 170) to change the font of a dialog or dialog item.

Arguments:

The arguments are:

fontName
The name of a font. You can look for valid fonts in the *Fonts* folder of your Windows Control Panel. If omitted, the SYSTEM font is used.

fontSize
The size of the font. If omitted, the standard size (10) is used.

fontStyle
One or more of the keywords listed in the syntax diagram, separated by blanks.

fontWidth
This argument is optional if it differs from fontSize.

Example:

The following example creates a 16-point italic Arial font:

```
hfnt = MyDialog~CreateFont("Arial", 16, "ITALIC")
```

DeleteFont

The *DeleteFont* method deletes a font. This method is to be used to delete a font created with the CreateFont method (see page 216).

Arguments:

The only argument is:

hFont The handle of a font.

FontToDC

```
►►—aDialogControl~FontToDC(dc,hFont)———————————————————————————►◄
```

The *FontToDC* method loads a font into a device context and returns the handle of the previous font. Use the GetWindowDC, GetDC, or GetButtonDC method to retrieve a device context, and the CreateFont method to get a font handle. To reset the font to the original state, use another FontToDC call with the handle of the previous font. To release the device context, use the FreeWindowDC, FreeDC, or FreeButtonDC method.

Arguments:

The arguments are:

dc The device context of a dialog or button.

hFont The handle of a font.

Example:

This example loads an Arial font into the current dialog window:

```
hfnt = MyDialog~CreateFont("Arial", 16, "ITALIC")
dc   = MyDialog~GetDC
oldf = MyDialog~FontToDC(dc,hfnt)  /* activate font */
...
MyDialog~FontToDC(dc,oldf)         /* restore previous font */
MyDialog~FreeDC(dc)
```

FontColor

```
►►—aDialogControl~FontColor(color,dc)———————————————————————————►◄
```

The *FontColor* method sets the font color for a device context.

Arguments:

The arguments are:

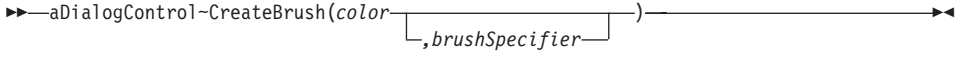
color The index of a color in the system's color palette.

dc The device context.

Graphic Methods

These methods deal with drawing graphics within the device context of a window. See "GetWindowDC" on page 166, "GetDC" on page 209, and "GetButtonDC" on page 166 for information on how to retrieve a device context.

CreateBrush



The *CreateBrush* method creates a color brush or a bitmap brush. It returns the handle to a brush object. To remove the brush, use "DeleteObject" on page 221. The brush is used to fill rectangles.

Arguments:

The arguments are:

color The color number. For a list of color numbers, refer to "Chapter 7. Definition of Terms" on page 87.

brushSpecifier

The name of a bitmap file or one of the following keywords to create a hatched brush:

UPDIAGONAL
A 45-degree upward, left-to-right hatch

CROSS
A horizontal and vertical crosshatch

DIAGCROSS
A 45-degree crosshatch

DOWNDIAGONAL
A 45-degree downward, left-to-right hatch

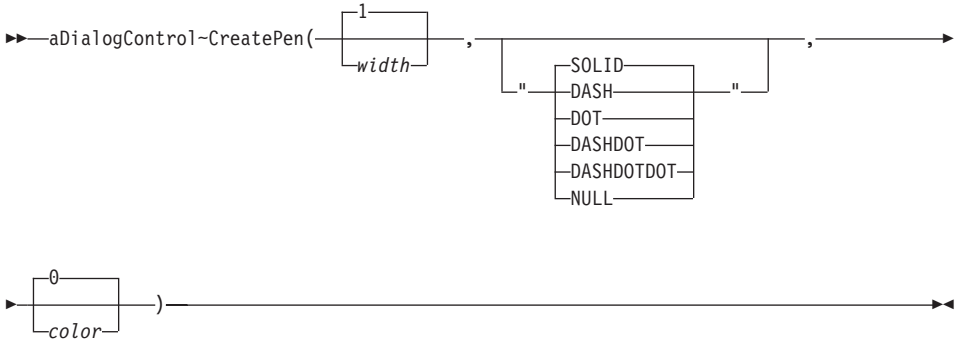
HORIZONTAL
A horizontal hatch

VERTICAL
A vertical hatch

If CreateBrush is sent to a dialog object (subclass of ResDialog), *brushSpecifier* can also be an integer resource ID for a bitmap stored in the DLL that also stores the resource.

If this argument is omitted, a solid brush with the specified color is created.

CreatePen



The *CreatePen* method creates a pen in the specified color and style. It returns the handle to a pen object. To remove the pen, use "DeleteObject" on page 221.

Arguments:

The arguments are:

width The width of the lines that the pen will draw. If omitted, 1 is used as default.

style One of the keywords listed in the syntax diagram. Values other than SOLID or NULL have no effect on pens with a width greater than 1. SOLID is the default.

color The color number of the pen. If omitted, 0 is used as default. For a list of color numbers, refer to "Chapter 7. Definition of Terms" on page 87.

Example:

The following example creates a dotted pen object with a width of 1:

```
hPen = MyDialog~CreatePen(1, "DOT", 13)
```

ObjectToDC

The *ObjectToDC* method loads a graphic object, namely a pen or a brush, into a device context. Subsequent lines, rectangles, and arcs are drawn using the pen and brush.

Arguments:

The arguments are:

dc The device context.

obj The object: a pen or a brush.

Return value:
The handle of the previous active pen or brush. It can be used to restore the previous environment.

Example:
The following example activates a pen for drawing:

```
dc = MyBaseDialog~GetDC
hpen = MyDialog~CreatePen(2, "SOLID", 4)
MyDialog~ObjectToDC(dc,hpen)
... /* do lines, rectangles, ... */
MyDialog~deleteObject(hpen)
```

DeleteObject

The *DeleteObject* method deletes a graphic object, namely a pen or a brush. See "CreatePen" on page 220 and "CreateBrush" on page 219 for information on how to get the handle of a pen or brush.

Arguments:
The only argument is:

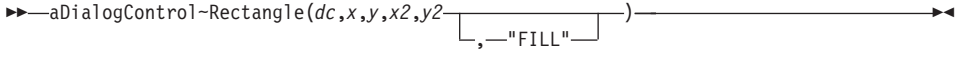
obj The handle of a pen or brush.

Graphic Drawing Methods

The following methods are used to draw rectangles, lines, pixels, and arcs in a device context. See "GetWindowDC" on page 166, "GetDC" on page 209, and "GetButtonDC" on page 166 for how to retrieve a device context. A pen and a brush can be activated using ObjectToDC before invoking the drawing methods.

Note: Because the pixel values include the title bar in a dialog it is easier to define a button filling the window, and then draw on the button.

Rectangle



The *Rectangle* method draws a rectangle to the given device context. The appearance is determined by the graphics objects currently active in the

device context. The active pen draws the outline and, optionally, the active brush fills the inside area. The default pen is thin black and the default brush is white.

Arguments:

The arguments are:

dc The device context.

x, y The position of the upper left corner of the rectangle, in pixels.

x2, y2 The position of the lower right corner.

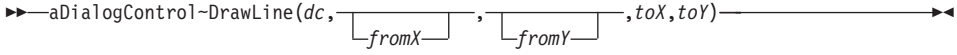
"FILL" The rectangle is filled with the active brush.

Example:

The following example draws a red rectangle filled with yellow, surrounded by a black rectangle:

```
dc = self~getButtonDC(100)
brush = self~createBrush(15)        /* yellow */
pen = self~createPen(10,'solid',13) /* thick red */
oldb = self~objectToDC(dc,brush)
oldp = self~objectToDC(dc,pen)
self~Rectangle(dc, 50, 50, 200, 150, "FILL")
self~objectToDC(dc,oldp); self~deleteObject(pen)
self~objectToDC(dc,oldb); self~deleteObject(brush)
self~Rectangle(dc, 40, 40, 210, 160) /* default */
```

DrawLine



The *DrawLine* method draws a line within the device context using the active pen.

Arguments:

The arguments are:

dc The device context.

fromX, fromY
The starting position, in pixels. If omitted, the previous end point of a line or arc is used.

toX, toY
The target position.

DrawPixel

The *DrawPixel* method draws a pixel within the device context.

Arguments:

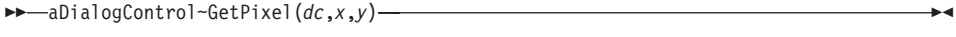
The arguments are:

dc The device context.

x, y The position, in pixels.

color The color number for the pixel. For a list of color numbers, refer to "Chapter 7. Definition of Terms" on page 87.

GetPixel



The *GetPixel* method returns the color of a pixel within the device context.

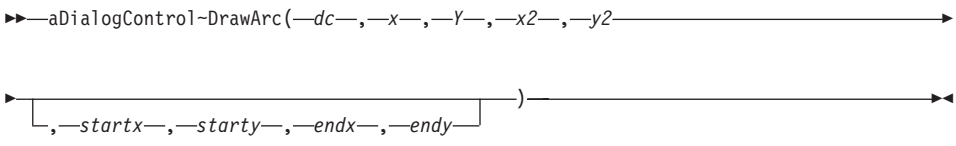
Arguments:

The arguments are:

dc The device context.

x, y The position, in pixels.

DrawArc



The *DrawArc* method draws a circle or ellipse on the given device context using the active pen for the outline. The circle or ellipse is drawn within the boundaries of an imaginary rectangle whose coordinates are given. A partial figure can be drawn by giving the end points of two radials. By default, the figure is drawn counterclockwise, but the direction can be modified using "SetArcDirection" on page 224.

Arguments:

The arguments are:

dc The device context.

x, y The position of the upper left corner of the imaginary rectangle, in pixels.

x2, y2 The position of the lower right corner of the imaginary rectangle, in pixels.

startx, starty, endx, endy
The end points of the starting and ending radials for drawing the figure. A full circle or ellipse is drawn if no start and end are given. Omitted values default to 0. Imaginary radials are drawn from the center to the start and end points. The circle or ellipse is then drawn between the intersections of these lines with the full circle or ellipse.

Example:

This example draws a full ellipse and a quarter circle:

```
dc = self~getButtonDC(100)
pen = self~createPen(4,'solid',13)
oldp = self~objectToDC(dc,pen)
self~drawArc(dc,50,50,200,150)                 /* full ellipse */
self~drawArc(dc,100,100,150,150, 200,50,75,75) /* quarter circle */
self~objectToDC(dc,oldp); self~deleteObject(pen)
```

GetArcDirection

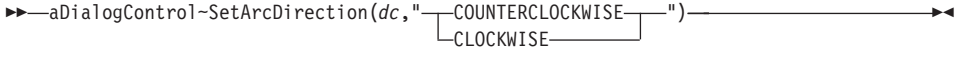
The *GetArcDirection* method returns the current drawing direction for the DrawArc method.

Arguments:

The only argument is:

dc The device context.

SetArcDirection



The *SetArcDirection* method changes the drawing direction for the DrawArc and DrawPie methods.

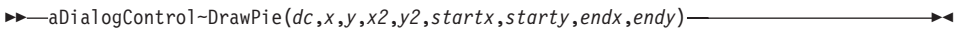
Arguments:

The arguments are:

dc The device context.

direction
The new drawing direction.

DrawPie



The *DrawPie* method draws a pie of a circle or ellipse on the given device context using the active pen for the outline and the active brush to fill the pie. The circle or ellipse is drawn within the boundaries of an imaginary rectangle whose coordinates are given. The arc is drawn between start and end radials in the direction specified by "SetArcDirection" on page 224.

Arguments:
The arguments are:

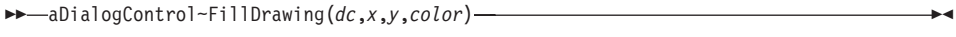
dc The device context.

x, y The position of the upper left corner of the imaginary rectangle, in pixels.

x2, y2 The position of the lower right corner of the imaginary rectangle.

startx, starty, endx, endy
The end points of the two radials (same as for DrawArc).

FillDrawing



The *FillDrawing* method fills an outline figure in the given device context using the active brush.

Arguments:
The arguments are:

dc The device context.

x, y The inside starting position for filling the outline figure with the color of the brush, in pixels.

color The color number of the outline figure whose inside will be filled. For a list of color numbers, refer to "Chapter 7. Definition of Terms" on page 87.

DrawAngleArc

```
►►—aDialogControl~DrawAngleArc(dc,xs,ys,x,y,radius,startangle,sweepangle)————————►◄
```

The *DrawAngleArc* method draws a partial circle (arc) and a line connecting the start drawing point with the start of the arc on the given device context using the active pen for the outline. The circle is drawn counterclockwise with the given radius between the given angles.

Arguments:

The arguments are:

dc The device context.

xs, ys The start draw position, in pixels.

x, y The center of the circle, in pixels.

radius The radius of the circle, in pixels.

startangle, sweepangle
The starting and ending angles for the partial circle in degrees (0 is the x-axis).

Chapter 10. UserDialog Class

The *UserDialog* class extends the *BaseDialog* class. It provides methods to create a dialog with these control elements:

- Entry lines
- Push buttons
- Check boxes
- Radio buttons
- List boxes
- Combo boxes
- Frames and rectangles

Note: The class also inherits the methods of its parent class.

There are two ways of creating a dialog:

- Load the dialog from a resource script using the Load method. A resource script can be created with a graphical resource editor such as the Resource Workshop.
- Invoke *Add...* methods to an instance of this class or a subclass and create the dialog step by step, one method for one dialog item. The best place to invoke these *Add...* methods is to override the *DefineDialog* method. The *DefineDialog* method is called automatically when the instance is created.

  There are also methods that enable you to define a group of the same dialog elements together. The names of these methods end with *Group* or *Stem*.

You can also combine loading a dialog from a resource script and adding elements dynamically.

Requires:

`UserDlg.cls` is the source file of this class. Use the tokenized version of OODialog, `oodialog.cls`, to shorten your dialog's startup time:

```
::requires oodialog.cls
```

Subclass:

The *UserDialog* class is a subclass of *BaseDialog*.

Attributes:

Instances of the *UserDialog* class have the following attributes:

AktPtr
An attribute for internal use

BasePtr
An attribute for internal use

DialogItemCount
An attribute for internal use

FactorX
Horizontal size of one dialog unit (in pixels)

FactorY
Vertical size of one dialog unit (in pixels)

SizeX Width of the dialog in dialog units

SizeY Height of the dialog in dialog units

Methods:

Instances of the *UserDialog* class implement the methods listed in the following table.

Method...	...on page
AddBitmapButton	238
AddBlackFrame	258
AddBlackRect	258
AddButton	237
AddButtonGroup	256
AddCheckBox	246
AddCheckBoxStem	254
AddCheckGroup	249
AddComboBox	245
AddComboInput	252
AddEntryLine	241
AddGrayFrame	258
AddGrayRect	258
AddGroupBox	240
AddInput	249
AddInputGroup	251
AddInputStem	252

Method...	...on page
AddListBox	244
AddMenuItem	261
AddMenuSeparator	262
AddOkCancelLeftBottom	259
AddOkCancelLeftTop	260
AddOkCancelRightBottom	259
AddOkCancelRightTop	259
AddPasswordLine	243
AddPopupMenu	261
AddRadioButton	246
AddRadioGroup	248
AddRadioStem	255
AddScrollBar	255
AddText	240
AddWhiteFrame	258
AddWhiteRect	257
Create	231
CreateCenter	232
CreateMenu	261
DefineDialog	233
Init	230
InitAutoDetection	230
Load	234
LoadFrame	235
LoadItems	236
LoadMenu	262
SetMenu	262
StartIt	260
StopIt	260

Init

The *Init* method initializes a new dialog object.

Arguments:

The only argument is:

DlgData.

A stem variable that is used to initialize the data fields of the dialog. If the dialog is terminated by means of the **OK** button, the values of the dialog's data fields are copied to this variable. The ID of the dialog items is used to name the entry within the stem.

Example:

The following example creates a new dialog object:

```
MyDialog=.UserDialog~new(aStem.)
```

InitAutoDetection

The *InitAutoDetection* method is called by the *Init* to determine whether or not automatic data field detection should be used. For a *UserDialog*, autodetection is disabled.

Protected:

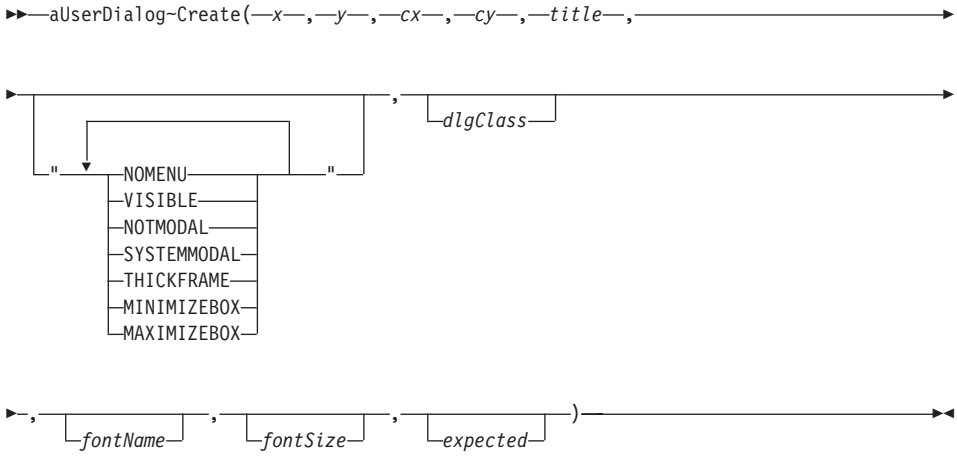
This method is protected. It is called by the class itself and can be overwritten.

Example:

The following example overrides the method to switch off auto detection:

```
::class MyClass subclass UserDialog

::method InitAutoDetection
self~NoAutoDetection
```

Create



The *Create* method creates the Windows dialog you previously defined with *Add...* methods. You can set the size, title, and style of the dialog. If the return code of Create is 0 the dialog creation failed and you should not try to execute the dialog object.

Arguments:

The arguments are:

x, y The position of the upper-left edge of the dialog in dialog units

cx, cy The extent (width and height) of the dialog in dialog units

title The dialog's title that is displayed in the title bar

options

One or more of the keywords listed in the syntax diagram, separated by blanks:

NOMENU
Creates a dialog without a system menu

VISIBLE
Creates a visible dialog

NOTMODAL
Creates a dialog with a normal window frame

SYSTEMMODAL
Creates a dialog that blocks all other windows

THICKFRAME
Creates a dialog with a thick frame

MINIMIZEBOX
The dialog is minimized

MAXIMIZEBOX
The dialog is maximized

dlgClass
Name of the window class used for the dialog. This argument must be omitted or an empty string.

fontName
The name of a font used by the dialog for all text. The default font is *System*.

fontSize
The size of the font used by the dialog. The default value is 8.

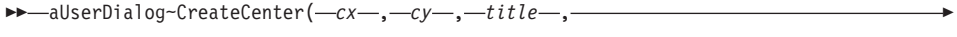
expected
This argument determines the maximum number of dialog elements (static text, entry lines, list boxes, and the like) the dialog can handle. The default value is 200. If your dialog has more than 200 elements, you must set this value; otherwise, the dialog fails.

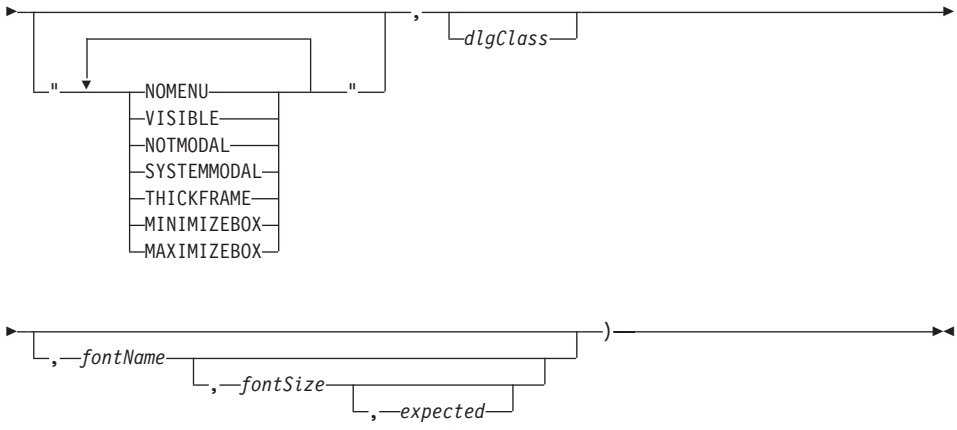
Example:

The following example creates a dialog with a size of 300 by 200 dialog units. The dialog has no system menu in its upper-left corner. It has a thick frame (therefore the dialog will get resizable) and a 12-point font. The dialog has capabilities for up to 100 elements.

```
rc = MyDialog~Create(20, 20, 300, 200, "My first Dialog",,
  "THICKFRAME NOMENU",, "Courier", 12, 100)
if rc <> 0 then MyDialog~Execute
```

CreateCenter





The *CreateCenter* method creates a dialog and centers its position. See "Create" on page 231 for a description of all arguments and an example.

DefineDialog

The *DefineDialog* method is called by *Create*. It is designed to be overwritten in a subclass of *UserDialog*. You should do all or additional dialog definitions, such as adding dialog items to the dialog, within this method. See also "Summary of User Dialog Processing" on page 60.

Protected:

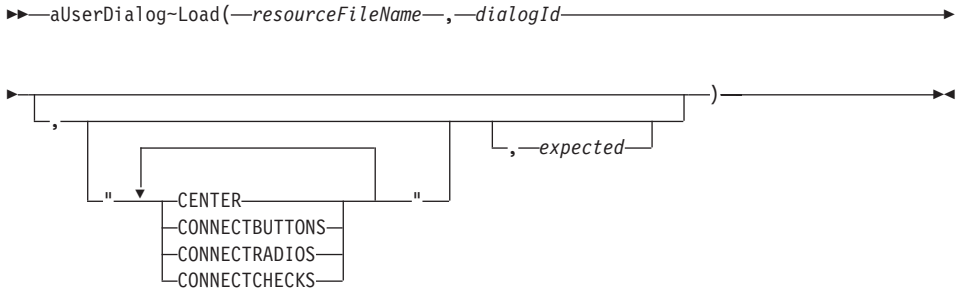
This method is protected. There is no need to call this method from anywhere else than *Create*.

Example:

When the dialog is created, a push button and an entry line are added to its client area:

```
::class MyDialog subclass UserDialog
   :
::method DefineDialog
self~AddButton(401, 20, 235, 40, 15, "&More...")
self~AddEntryLine(402, INPUT_1, 20, 170, 150)
   :
```

Load



The Load method creates the dialog based on the data of a given resource script (a file with the extension .RC). It calls the *LoadFrame* and *LoadItems* methods to retrieve the dialog data from the file. See also "Summary of User Dialog Processing" on page 60.

Return code:

The return code is 0 for a successful load and 1 otherwise

Arguments:

The arguments are:

resourceFileName

The name of the resource script of the dialog

dialogId

The ID (number) of the dialog. Note that each dialog has a unique ID assigned to it. There can be more than one dialog definition in one resource file. If there is only one dialog resource in the resource file, you do not have to indicate the ID.

options

One or more of the keywords listed in the syntax diagram, separated by blanks:

CENTER

The dialog is positioned in the center.

CONNECTBUTTONS

For each button a connection to an object method is established automatically. See ConnectControl for a description of connecting buttons to a method.

CONNECTRADIOS

Similar to *CONNECTBUTTONS,* this option enforces the method to connect the radio buttons.

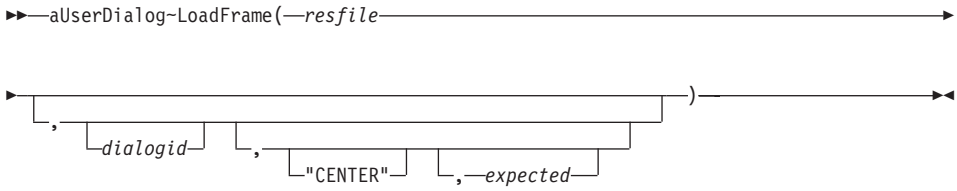
CONNECTCHECKS
This option connects the check box controls.

expected
This is the maximum number of dialog elements the dialog object can handle. See Create.

Example:
The following example creates a dialog based on the values for dialog 100 in Dialog1.rc. It also connects the push and radio buttons to a message named after the buttons' title.

```
MyDlg = .UserDialog~new()
MyDlg~Load("Dialog1.rc", 100, "CONNECTBUTTONS CONNECTRADIOS")
```

LoadFrame



The *LoadFrame* method creates the window frame using the data of the given dialog resource with *dialogid* in file *resfile*. It is usually called by the Load method. See also "Summary of User Dialog Processing" on page 60.

Protected:
This method is protected. It can only be used internally within a class method.

Arguments:
The arguments are:

resfile The name of the resource file

dialogid
The ID of the dialog. It can be omitted if there is just one dialog; otherwise it has to be specified.

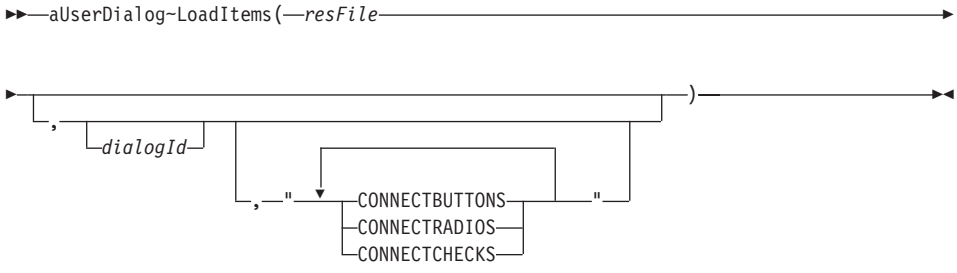
expected
The number of expected dialog items

Example:
The following example overrides the Load method, so it loads the dialog window (just the frame) but not its contents:

```
::class WindowOnlyDialog subclass UserDialog
  :
```

```
   .
::method Load
self~LoadFrame("Dialog2.rc", 100, "CENTER", 20)
```

LoadItems



The *LoadItems* method creates the dialog items, using the data of the given resource script. It is either called by the Load method, or it can be used in the context of a category dialog. See also "Summary of User Dialog Processing" on page 60.

Protected:
This method cannot be called from outside the class.

Arguments:
See Load for a description.

Example:
In the following example the dialog is created either with the items of dialog 200 or dialog 300, depending on the argument:

```
::class MyDialog subclass UserDialog
   :
::method Load
   use arg view
   self~LoadFrame("Dialog2.rc", 200, "CENTER", 200)
   if view="special" then
      self~LoadItems("Dialog2.rc", 200, "CONNECTBUTTONS")
   else
      self~LoadItems("Dialog2.rc", 300, "CONNECTBUTTONS")
```

Add... Methods

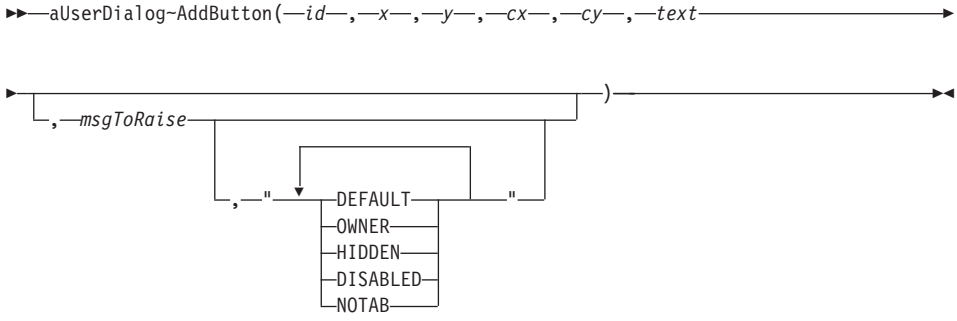
The methods listed below (all starting with *Add*) can be used to create a dialog dynamically without any resource script (.RC file). They can also be used in addition to a *loaded* dialog.

The recommended way to create a dialog is to subclass from *UserDialog* and put all *Add...* statements into *DefineDialog* method, which is executed when the dialog is about to be created.

Add... methods call the matching *Connect...* methods to create the associated Object REXX attribute. *Add...* methods cannot be used after Execute has started.

Note: The coordinates are usually set in dialog units, if not mentioned explicitly.

AddButton



The *AddButton* method adds a push button to the dialog and connects it with a method that is processed whenever the button is clicked.

Arguments:

The arguments are:

id A unique number you have to assign to the button. You need the ID to refer to this control in other methods.

x, y The position of the button's upper-left corner relative to the dialog measured in dialog units

cx, cy The size of the button in dialog units

text The button's title that is displayed on the button

msgToRaise

The name of a method that is processed whenever the button is clicked

options

The last argument can be one or more of:

DEFAULT

The button becomes the default button in the dialog.

OWNER
The button is *owner-drawn*. This option is used for bitmap buttons.

HIDDEN
The button is not visible at startup time.

DISABLED
The button is disabled at startup time.

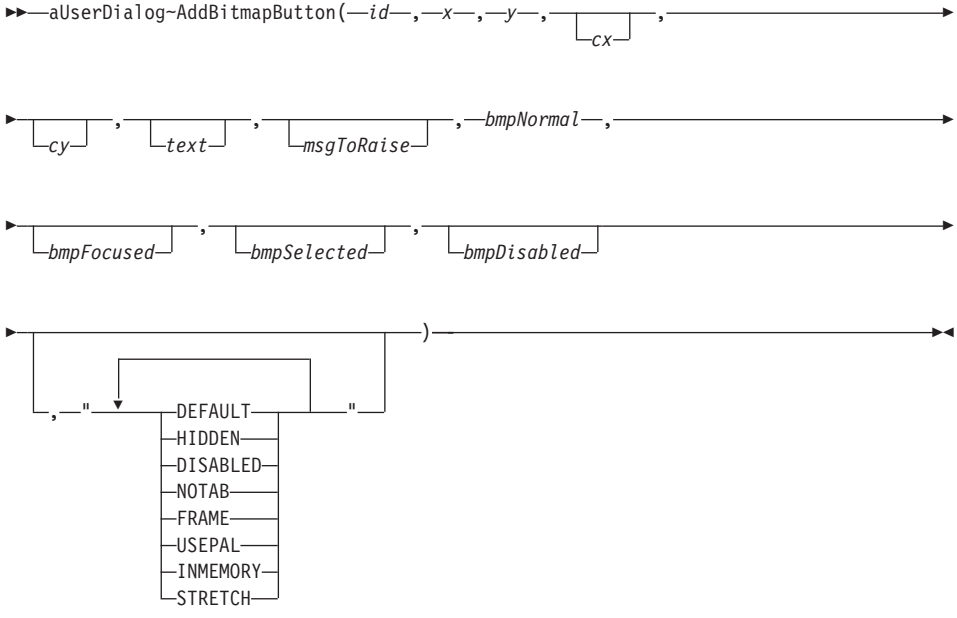
NOTAB
There is no tabstop at the button, so you cannot get to the button by using just the keyboard (tab key).

Example:

The following example creates a push button entitled *Get new Info* at position x=100/y=80 and size width=40/height=15. The button's ID is 555, and if the button is clicked, the *getInfo* message is sent to the dialog object.

```
MyDialog~AddButton(555, 100, 80, 40, 15, "&Get new Info",,
"getInfo", "NOTAB")
```

AddBitmapButton



The *AddBitmapButton* method adds a push button with a bitmap (instead of plain text) to the dialog. You can provide four different bitmaps representing the four states of a button.

The bitmaps can be specified by either a file name or a bitmap handle. You can retrieve a bitmap handle by loading a bitmap stored in a file into memory, using the method "LoadBitmap" on page 208). If you pass a bitmap handle to the method, you must use the *INMEMORY* option.

Arguments:

The arguments are the same as for *AddButton*, with the changes listed below:

bmpNormal
A bitmap that is displayed

bmpFocused
A bitmap that is displayed if the button is focused. Having the focus means that the button is clicked by using the Enter key. Normally the focused button is surrounded by a dashed frame.

bmpSelected
A bitmap that is displayed while the button is clicked and held

bmpDisabled
A bitmap that is displayed if the button is disabled

options
In addition to *AddButton*, there are four more options:

FRAME
The button has a 3D frame. This gives your bitmap the same behavior as a standard Windows button.

USEPAL
The color palette of the bitmap is loaded and used. This argument should be specified for just one of the dialog buttons, because only one color palette can be active at any time.

INMEMORY
Specifies that the bitmap was loaded into memory before. If you switch often between different bitmaps within one button, the loading of all bitmaps into memory increases performance.

STRETCH
If this option is specified and the extent of the bitmap is smaller than the extent of the button rectangle, the bitmap is adapted to match the extent of the button.

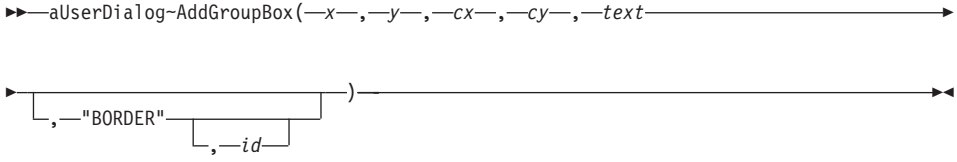
See AddButton for a description of the other arguments.

Example:

The following example defines a button with ID 601. The bitmap in the `Button1.bmp` file is displayed for the push button instead of a black text on a grey background. If the button is disabled (by using the *DisableItem* method, see page 153), the bitmap is exchanged and *Button1D.bmp* is shown instead. If the button is clicked, the *BmpPushed* message is sent.

```
MyDialog~AddBitmapButton(601, 20, 317, 80, 30, , "BmpPushed",,
                                "Button1.bmp",,,"Button1D.bmp","FRAME USEPAL")
```

AddGroupBox



The *AddGroupBox* method adds a group box to the dialog. A group box has a frame and a title.

Arguments:

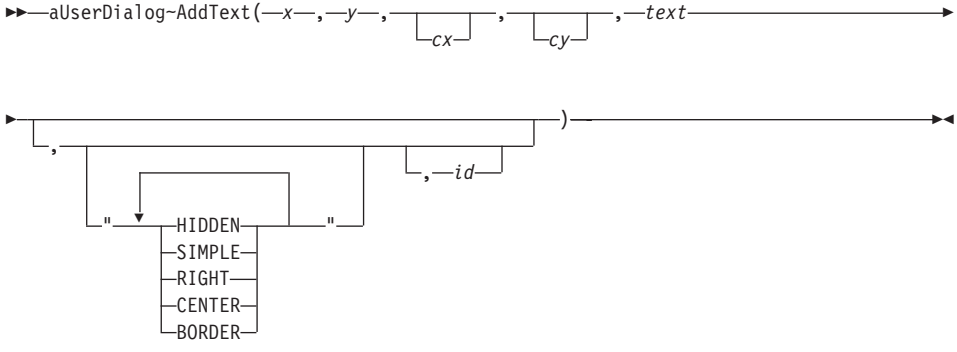
The arguments are the same as for AddButton, with the changes listed below:

text The title of the group box

options

BORDER
A rectangle is drawn around the group box

AddText



The *AddText* method adds a static text element to the dialog.

Arguments:

The arguments are the same as for AddButton, with the changes listed below:

text The text string to be displayed

options

This argument can be one or more of:

HIDDEN
The text is not visible at startup time

SIMPLE
Simple text field

RIGHT
The text is aligned to the right

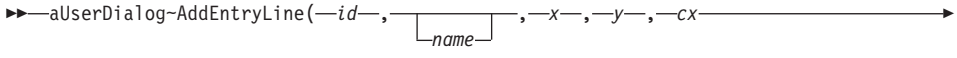
CENTER
The text is centered. If neither RIGHT or CENTER is specified, the text is aligned to the left.

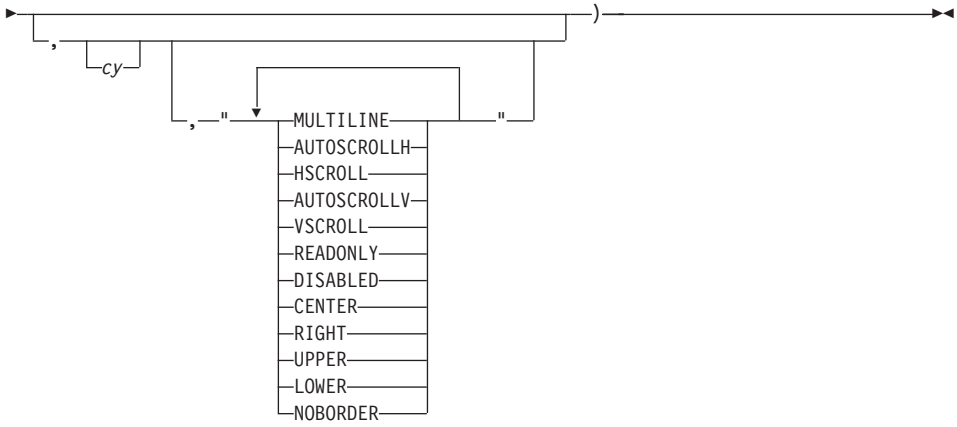
BORDER
A rectangle is drawn around the text

If not specified RIGHT or CENTER, the text is aligned to the left.

id If omitted, ID -1 is used.

AddEntryLine





The *AddEntryLine* method adds an entry line to the dialog.

Arguments:

The arguments are:

id This must be a unique number.

name This is the name of the entry line. An attribute with exactly this name is added to the object and provides data for the dialog item automatically. See "ConnectEntryLine" on page 116.

x, y The position of the upper-left corner relative to the dialog's client area measured in dialog units

cx The length of the entry line in dialog units

cy The height of the entry line. If this argument is omitted or equal to 0, the height is calculated to fit the font's height.

Options

The last argument can be one or more (separated by a blank) of:

MULTILINE

Designates a multiple-line edit control. (The default is single line.)

AUTOSCROLLH

Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.

HSCROLL
Combines a horizontal scroll bar with the entry line.

AUTOSCROLLV
Automatically scrolls text up one page when the user presses ENTER on the last line.

VSCROLL
Combines a vertical scroll bar with the entry line.

READONLY
Prevents the user from entering or editing text in the edit control.

DISABLED
Initially disables the entry line.

CENTER
Centers text in a multiline edit control.

RIGHT
Aligns text flush right in a multiline edit control.

UPPER
Converts all characters to uppercase as they are typed into the edit control.

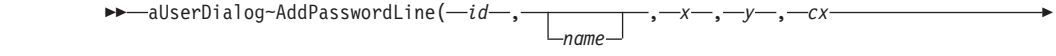
LOWER
Converts all characters to lowercase as they are typed into the edit control.

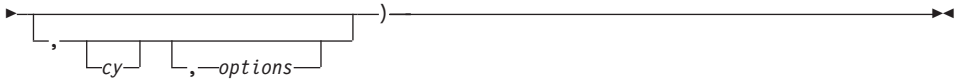
NOBORDER
The rectangle is not drawn around the entry field.

Example:
The following example puts the entry line with ID 201 and length of 150 dialog units close to the upper-left corner of the dialog's client area. The *FIRSTNAME* attribute is created and connected to the dialog item. If the entered data is longer than 150 dialog units, the entryline is scrolled horizontally.

```
MyDialog~AddEntryLine(201, "FIRSTNAME", 12, 14, 150,,"AUTOSCROLLH")
```

AddPasswordLine



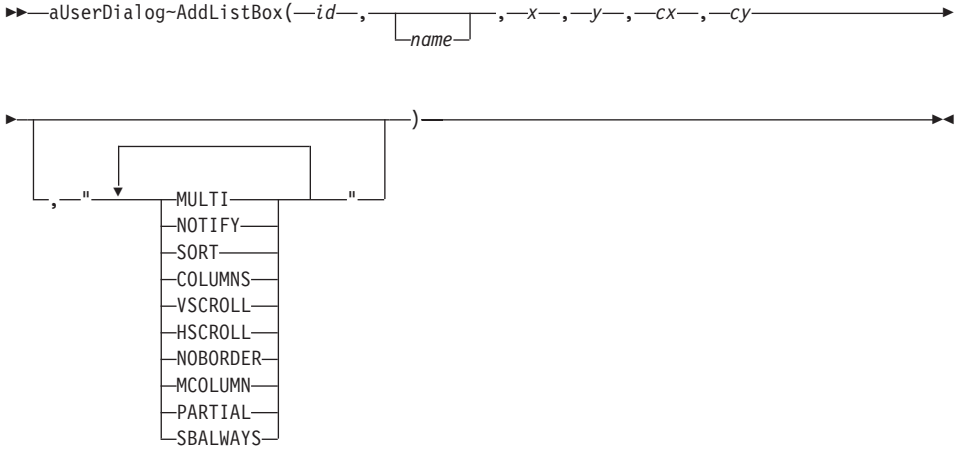


The *AddPasswordLine* method adds a password entry line that does not echo the characters entered but displays asterisks (*) instead.

Arguments:

See AddEntryLine for a description of the arguments.

AddListBox



Adds a list box to the dialog.

Arguments:

The arguments are the same as for AddEntryLine, with the changes listed below:

Options

The last argument can be one or more of:

MULTI

Makes the list box a multiple choice list box, that is, you can select more than one line.

NOTIFY

A message is posted whenever the user selects an item of the list box. To use this feature you have to connect the list to a method (see "ConnectList" on page 115).

SORT The items in the dialog are listed in the noted order.

COLUMNS
The list box can handle tab characters ('09'x). Use this option together with the SetListTabulators method (see page 146) to have more than one column in a list.

VSCROLL
Adds a vertical scroll bar to the list box. Scroll bars appear only if the list contains more lines than can fit in the available space.

HSCROLL
Adds a horizontal scroll bar to the list box. See also "SetListWidth" on page 141.

NOBORDER
Draw list without drawing a rectangle around it.

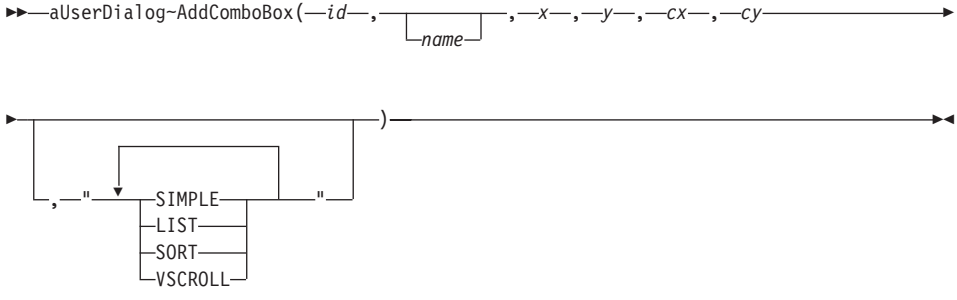
MCOLUMN
Makes the list box a multicolumn list box that can be scrolled horizontally. "SetListColumnWidth" on page 141 sets the width of the columns.

PARTIAL
The size of the list box equals the size specified by the application when it created the list box. Windows usually sizes a list box such that the list box does not display partial items.

SBALWAYS
The list box shows a disabled scroll bar if there is no need to scroll. If you do not specify this option, the scroll bar is hidden when the list box does not contain enough items.

Note: A list box does not support a horizontal scroll bar.

AddComboBox



The *AddComboBox* method adds a combo box to the dialog. A combo box is a combination of an entry line and a list box.

Arguments:

The arguments are the same as for AddEntryLine, with the changes listed below:

options

The last argument can be one or more of:

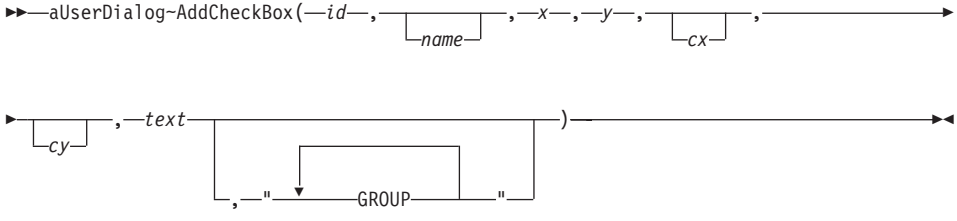
SIMPLE
Displays the list box all the time.

LIST No free text can be entered in the entry line; the list contains selectable items only.

SORT The items in the list are sorted by the combo box itself.

VSCROLL
Adds a vertical scroll bar to the combo box.

AddCheckBox



The *AddCheckBox* method adds a check box to the dialog.

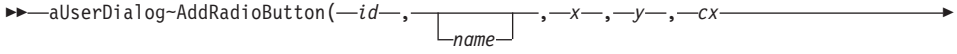
Arguments:

The arguments are the same as for AddEntryLine, with the changes listed below:

name The name of the check box. If omitted, *text* is used.

text The text that is displayed next to the check box.

AddRadioButton



The *AddRadioButton* method adds a radio button to the dialog.

Arguments:

The arguments are the same as for AddEntryLine, with the changes listed below:

name The name of the radio button

text The text that is displayed next to the radio button

options

Valid values for the last argument are:

GROUP

Makes the radio button the beginning of a new group. Use this option just for the first radio button if you want to make all radio buttons dependent. In each group if you select a radio button, the previously selected button is automatically deselected.

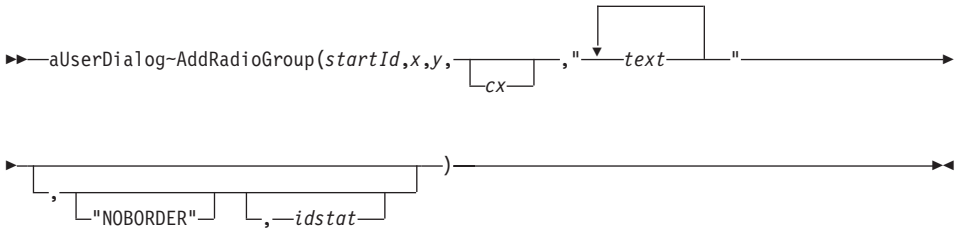
Example:

The following example defines seven radio buttons with IDs 501 through 507:

```
RText.1="Monday"
RText.2="Tuesday"
RText.3="Wednesday"
RText.4="Thursday"
RText.5="Friday"
RText.6="Saturday"
RText.7="Sunday"

do i=1 to 7
 MyDialog~AddRadioButton(500+i, , 20, i*15+13, 40, 14, RText.i)
end
```

Note: There are also methods that create a whole group automatically (see the AddRadioGroup method below and AddRadioStem).

AddRadioGroup



The *AddRadioGroup* method creates a group of radio buttons.

Arguments:

The arguments are:

startId The ID of the first radio button. The *startId* is increased by 1 for each additional radio button and then assigned to the dialog item.

x, y The position of the first radio button control. The other radio buttons are positioned automatically.

cx The length of the radio button plus text. If omitted, the space needed is calculated.

text The text string for each radio button. Single strings have to be separated by blank spaces. This argument determines the number of radio buttons in total.

options

The only option is *NOBORDER,* which prevents the method from placing a group box around the group.

idstat This argument is used to set the static frame ID.

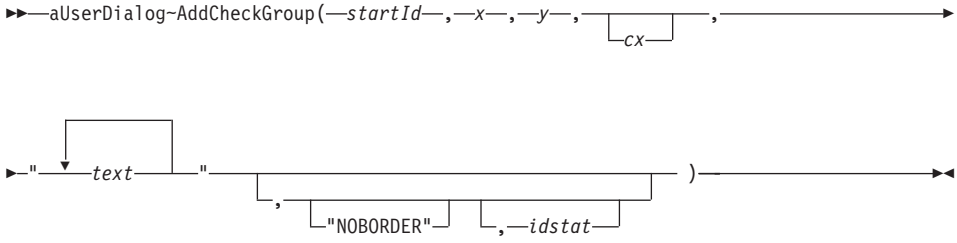
Example:

The following example adds a group of three radio buttons with IDs 301, 302, and 303 to the dialog (see Figure 53):

```
MyDialog = .UserDialog~new
MyDialog~Create(100,100,80,60,"Radio Button Group")
MyDialog~AddRadioGroup(301, 23, 18, ,"Fast Medium Slow")
MyDialog~fast = 1
MyDialog~Execute
```

Figure 53. Sample Radio Button Group

AddCheckGroup



The *AddCheckGroup* method creates a group of check boxes. See AddRadioGroup for a full description.

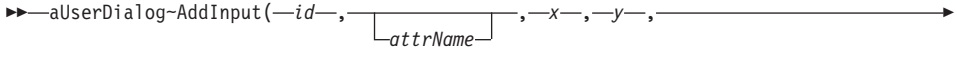
Example:

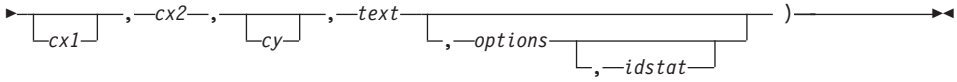
The following example adds a group with four check boxes to the dialog. Two check boxes are preselected (see Figure 54):

```
MyDialog~AddCheckGroup(401, 23, 18, ,"Smalltalk C++ ObjectREXX OO-COBOL")
MyDialog~smalltalk = 1
MyDialog~objectrexx = 1
```

Figure 54. Sample Check Box Group

AddInput





The *AddInput* method adds an entry line with a label (a static text) to the dialog.

Arguments:

The arguments are:

id The unique ID of the entry line

attrName The *attrName* is used to create an attribute in the dialog object that reflects the contents of the entry line (see "AddEntryLine" on page 241). If it is skipped, the *text* label is used as the attribute name.

x, y The position of the upper-left edge of the label. The entry line is aligned automatically.

cx1 The length of the label. If omitted, the length is calculated.

cx2 The length of the entry field

cy The height of the entry field. If omitted, the height is calculated.

text The label displayed in front of the entry field

options Possible options are:

HIDDEN
Makes the input group invisible

PASSWORD
Displays asterisks instead of the typed-in characters

For further options see AddEntryLine.

idstat An ID for the label

Example:

The following example creates an entry field and the label *Your e-mail address* (placed on the entry field's left side). It also creates an attribute with the name *YOUREMAILADDRESS*. The height of the elements is calculated. (See Figure 55.)

```
MyUserDialog~AddInput(402, , 20, 30, , 150, , "Your eMail address")
```

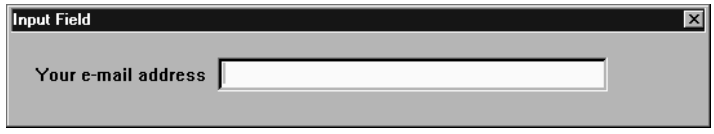
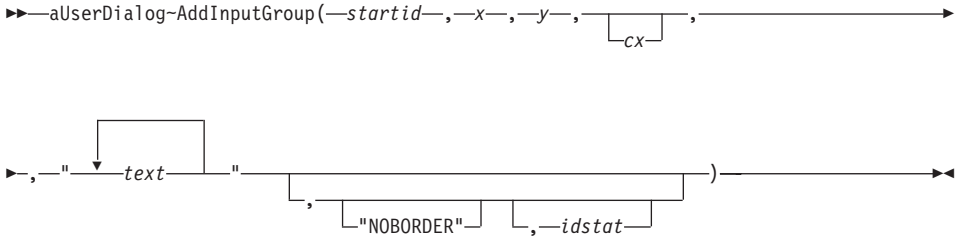


Figure 55. Sample Input Field

AddInputGroup



The *AddInputGroup* method creates a group of one or more entry lines.

Arguments:

The arguments are:

startid An ID that is assigned to the first entry line. Consecutive numbers are assigned to the other entry fields.

x, y Position of the input group's upper-left corner

cx1 Length of the entry field label. If omitted, the length is calculated.

cx2 Length of the entry field in dialog units

text The text strings used for each entry field's label. The single strings are to be separated by blank spaces. This argument determines the number of entry fields in total.

options

In addition to the options of AddInput, *NOBORDER* can be used to prevent the method from placing a group box around the group.

idstat The ID of the first label. Usually you do not have to specify this value because labels are static controls.

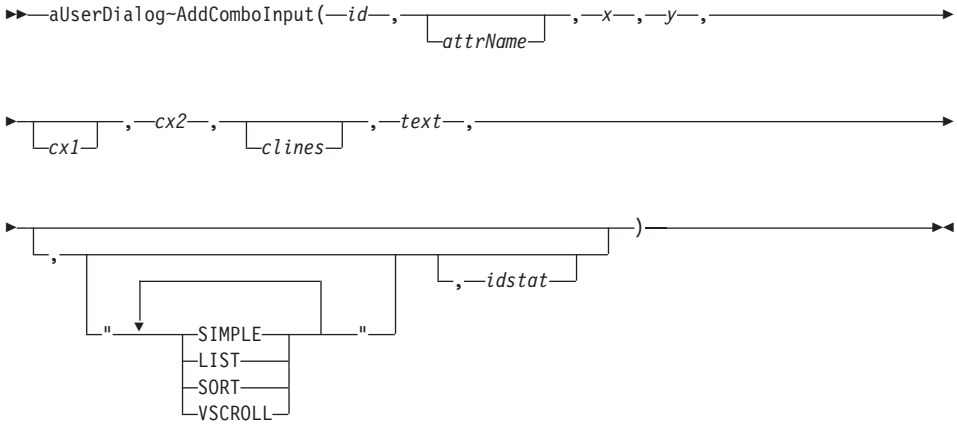
Example:

The following example creates a four-line input group. The single entry lines are accessible by IDs 301 through 304.

```
MyDialog~AddInputGroup(301, 20, 20, ,130, "Name FirstName Street City")
```

Note: If you want to use labels that include blanks (for example, "First Name" instead of "FirstName"), use the AddInputStem method.

AddComboInput



The *AddComboInput* method adds a combo box and a label string to the dialog.

Arguments:

The arguments are:

id The ID of the combo box

attrName
The name of the combo box. This name is used as an object attribute name.

x, y Position of the group (text string of combo box)

cx1 Length of the text string

cx2 Width of the combo box

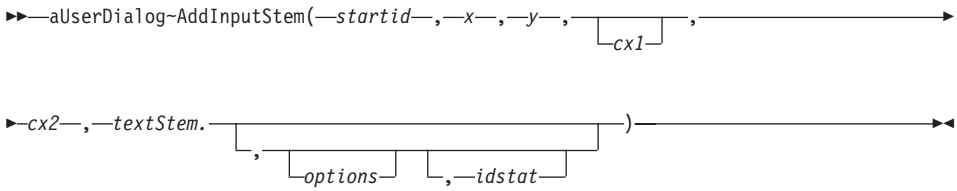
clines Vertical length of the combo box in number of lines

text Label being displayed on the left-hand side of the combo box

options
See "AddComboBox" on page 245

idstat The ID of the label. Usually you do not have to specify this value because labels are static controls.

AddInputStem



The *AddInputStem* method adds a group of input fields to the dialog. The difference between this method and the AddInputGroup method is that the titles (and names) of the single lines are passed to the method in a stem variable. Thus it is possible to use strings containing blank spaces.

Arguments:

The arguments are:

startid The ID of the first entry line

x, y The position of the whole group (upper-left corner)

cx1 The length of the text strings. If omitted, the size is calculated.

cx2 The width of the entry fields

textStem.
A stem variable containing all labels for the entry fields. The object attribute for each field is created on the basis of this string.

options
In addition to the options of the AddInput method, *NOBORDER* can be used to prevent the method from placing a group box around the group.

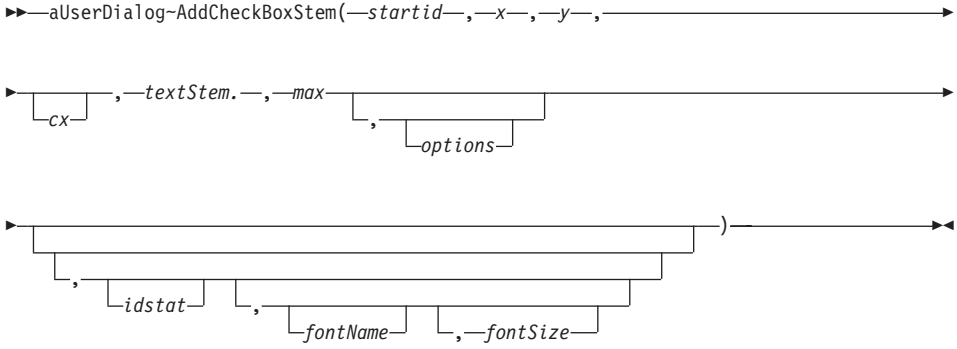
idstat The ID of the first label. Usually you do not have to specify this value because labels are static controls.

Example:

The following example shows how to use *AddInputStem*. It creates a four-line input group. For each entry line (with IDs 401 through 404) an object attribute is provided. The names might be different from the title because not all characters can be used for Object REXX symbols. In this example the *NAME*, *FIRSTNAME*, *STREETNUMBER*, and *CITYZIP* attributes are added to the object.

```
FNames.1="Name"
FNames.2="First Name"
FNames.3="Street & Number"
FNames.4="City & ZIP"

MyDialog~AddInputStem(401, 20, 20, , 150, FNames.)
```

AddCheckBoxStem



The *AddCheckBoxStem* method creates a group of check box controls. Unlike the AddCheckGroup method you pass the titles of the check boxes in a stem variable instead of using a string. Thus you can use labels including blanks.

Arguments:

See AddCheckGroup for a description of the arguments. The new arguments are:

textStem.
A stem variable containing all labels for the check boxes. The object attribute for each check box is created on the basis of this string.

max The maximum number of check box items in one column. If *textStem* has more items than *max,* a second column is created.

fontName
The name of the font used within the dialog

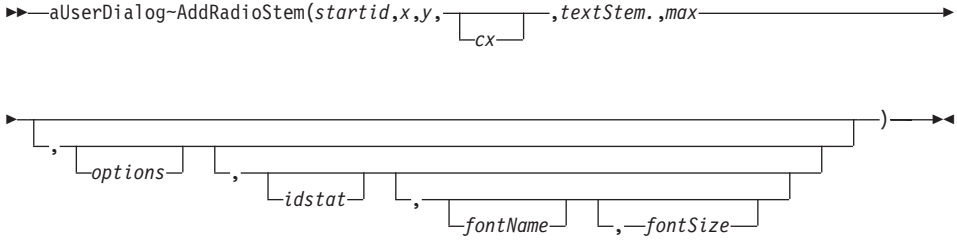
fontSize
The size of the font used within the dialog

Example:

The following example creates a three-column check box group:

```
CBNames.1="C"
CBNames.2="Pascal"
CBNames.3="Cobol"
CBNames.4="REXX"
CBNames.5="Basic"
CBNames.6="Fortran"

MyDialog~AddCheckBoxStem(501, 20, 20, ,CBNames, 2,,
"NOBORDER", 551, "Courier New", 12)
```

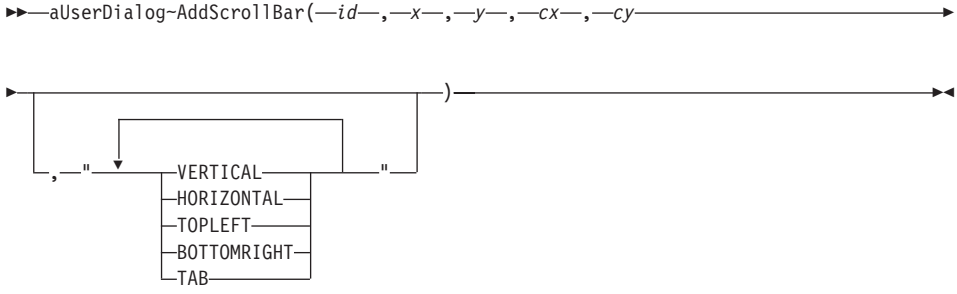
AddRadioStem



The *AddRadioStem* method adds a group of radio button controls to the dialog.

See AddRadioGroup for a description of the arguments and an example.

AddScrollBar



The *AddScrollBar* method adds a scroll bar to the dialog.

Arguments:

The arguments are:

id This must be a unique number

x, y The position of the upper-left corner relative to the dialog's client area measured in dialog units

cx The horizontal size of the scroll bar in dialog units

cy The vertical size of the scroll bar

options

The last argument can be one or more of:

VERTICAL

The scroll bar is positioned vertically (default)

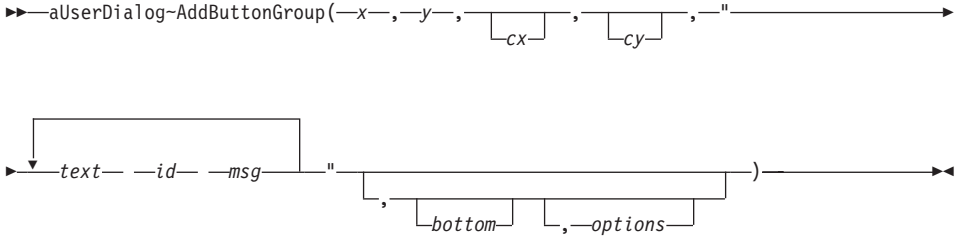
HORIZONTAL
The scroll bar is positioned horizontally

TOPLEFT
The scroll bar is aligned to the top left of the given rectangle and has a predetermined width (if vertical) or height (if horizontal)

BOTTOMRIGHT
The scroll bar is aligned to the bottom right of the given rectangle and has a predetermined width (if vertical) or height (if horizontal)

TAB The scroll bar is assigned a tab stop

AddButtonGroup



Use the *AddButtonGroup* method to add more than one push button at once to the dialog. The buttons are arranged in a row or in a column.

Arguments:
The arguments are:

x, y The position of the entire button group

cx, cy The size of a single button. One or both arguments can be skipped. If so, the default values (cx=40, cy=12) are taken.

text ID msg
These arguments are interpreted as *one* string containing *three* words (separated by blanks) for each button. The first word is the text that is displayed on the button, the second is the ID of the button, and the third is the name of a message that is sent to the object whenever the button is clicked. The fourth to sixth words are for the next button, and so forth.

bottom
This is a flag to switch between a vertical (=0) or horizontal (=1) placement of the buttons.

options
If *DEFAULT* is used, the first button becomes the default button. For the other options, see AddButton.

Example:

The following example creates three buttons (**Add**, **Delete**, and **Update**):

```
MyDialog~AddButtonGroup(20, 235, , , "&Add 301 AddItem" ¦¦ ,
"&Delete 302 DeleteItem" ¦¦ ,
"&Update 303 UpdateItem")
```

Frames and Rectangles

The methods listed below add simple graphical elements to the dialog. They are useful for giving the dialog a nice finish. Use Figure 56 to help you find the right element.

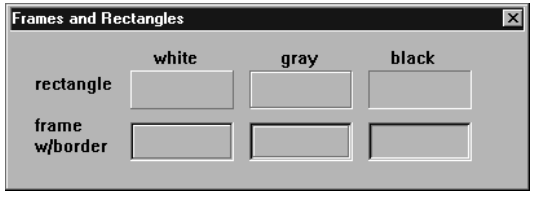
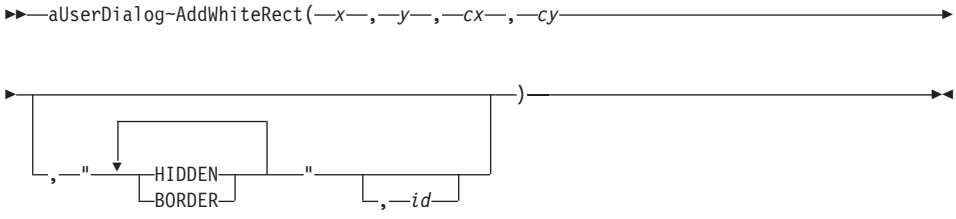


Figure 56. Frames and Rectangles in 3D Style

Note: There is currently no difference between rectangles and frames.

AddWhiteRect



The *AddWhiteRect* method adds a white rectangle to the dialog.

Arguments:

The arguments are:

x, y The position of the rectangle's upper-left corner relative to the dialog measured in dialog units

cx, cy The size of the rectangle in dialog units

options

The options can be:

HIDDEN
The frame or rectangle is not visible at startup time

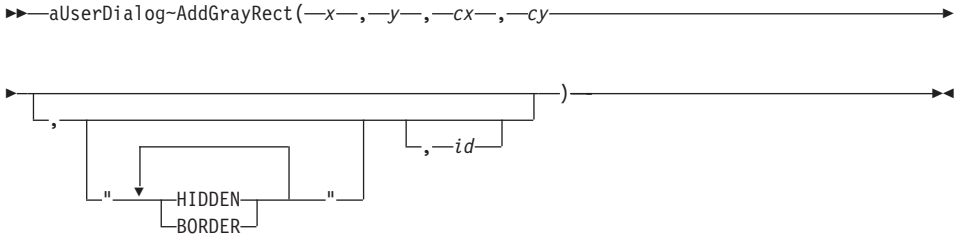
BORDER
A border is drawn around the rectangle or frame

id The ID of the item, -1 is used by default

AddWhiteFrame

The *AddWhiteFrame* method is currently identical to the *AddWhiteRect* method.

AddGrayRect



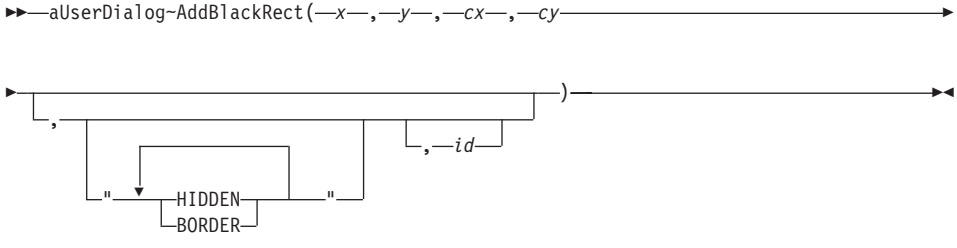
The *AddGrayRect* method adds a gray rectangle to the dialog.

Arguments:
See AddWhiteRect for a description of the arguments.

AddGreyFrame

The *AddGreyFrame* method is currently identical to the *AddGreyRect* method.

AddBlackRect



The *AddBlackRect* method adds a black rectangle to the dialog.

Arguments:
See AddWhiteRect for a description of the arguments.

AddBlackFrame

The *AddBlackFrame* method is currently identical to the *AddBlackRect* method.

OK and Cancel Push Buttons

The four methods described in this section add **OK** and **Cancel** push buttons to the dialog. The standard IDs (1 for OK and 2 for Cancel) are assigned to the buttons.

AddOkCancelRightBottom

The *AddOkCancelRightBottom* method adds an **OK** and a **Cancel** push button to the lower-right edge of the dialog.

Example:

The following example adds the two push buttons to the bottom of the dialog. It further overrides the standard *OK* and *Cancel* methods.

```
::class MyClass subclass UserDialog

::method DefineDialog
   :
   self~AddOkCancelRightBottom

::method OK
   ret = MessageBox("Are you sure?", "Please confirm", "OK")
   if ret=1 then self~OK:super

::method Cancel
   ret = MessageBox("Do you really want to quit?", "Please confirm", "OK")
   if ret=1 then self~CANCEL:super
```

AddOkCancelLeftBottom

The *AddOkCancelLeftBottom* method adds an **OK** and a **Cancel** push button to the lower-left edge of the dialog.

AddOkCancelRightTop

The *AddOkCancelRightTop* method adds an **OK** and a **Cancel** push button vertically to the upper-right edge of the dialog.

AddOkCancelLeftTop

The *AddOkCancelLeftTop* method adds an **OK** and a **Cancel** push button vertically to the upper-left edge of the dialog.

Dialog Control Methods

The methods described in this section control the execution of the dialog; they are for internal use only.

StartIt

The *StartIt* method is for internal use only. It is necessary to create a real Windows object based on the dialog template.

Protected:

This method is protected and cannot be called from outside the instance. It can be overwritten, although this is not recommended.

Arguments:

There is only one argument:

icon This argument is currently not used.

StopIt

The *StopIt* method is for internal use only. It is the counterpart to the *BaseDialog* class *StopIt* method to remove the Windows object.

Protected:

This method is protected and cannot be called from outside the instance. It can be overwritten, although this is not recommended.

Menu Methods

The methods described in this section are for creating dialog menus.

CreateMenu

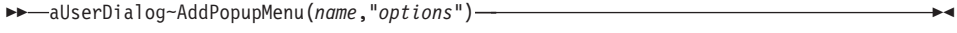
This method initializes the creation of a menu. When you have finished adding menu items, call SetMenu to combine the menu with the dialog.

Arguments:

There is only one argument:

count Maximum number of menu items that can be added

AddPopupMenu



This method adds a popup menu to the menu.

The last popup menu must have the option ″END″ specified.

Arguments:

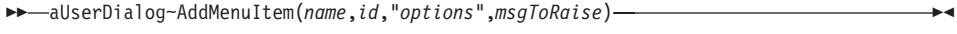
The arguments are:

name The name of the popup menu

options

GRAYED DISABLED END

AddMenuItem



This method adds a new menu item after the last added item.

The last menu item in a popup menu must have the option ″END″ specified.

Arguments:

The arguments are:

name The name of the menu item. The name can include &.

id The ID of the menu item

options

GRAYED DISABLED END CHECKED

msgToRaise
Method to be called when the menu item is selected. See ConnectMenuItem.

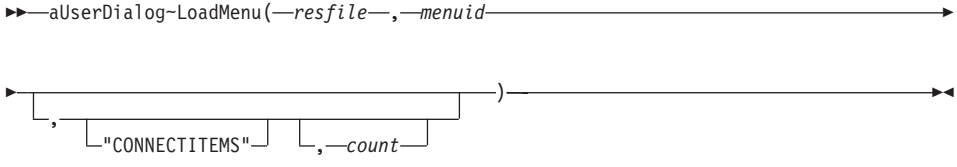
AddMenuSeparator

This method adds a menu separator to the menu after the last added item.

SetMenu

This method adds the menu that was created or loaded for the current dialog to the dialog window. Note that the menu needs additional space and therefore displaces the rest of the dialog items.

LoadMenu



This method loads a menu resource out of a resource script.

To combine the menu with the dialog, you must call SetMenu.

Arguments:
The arguments are:

resfile The name of a resource script.

menuid
The ID of the menu resource.

CONNECTITEMS
The menu items are automatically connected to methods named after the name of the menu item. See LoadItems for further details.

count The maximum number of menu items that can be loaded.

Chapter 11. PlainUserDialog Class and PlainBaseDialog Class

The PlainUserDialog class subclasses from PlainBaseDialog class and provides all the methods that normally are required to execute a dialog that is either created dynamically or loaded from a resource script (.RC). In other words it is a limited version of Chapter 10. UserDialog Class. Use ::requires `"OODPLAIN.CLS"` in your script to get access to the PlainUserDialog class.

UserDialog includes all the methods PlainUserDialog does plus all the methods defined in the DialogExtensions class. These are more specific methods that cover asynchronous dialog execution, scroll bar support, resizing and repositioning, bitmaps, graphics (device context related methods), scrolling text, and menus (action bars).

The reason for splitting the functionality into two classes is to provide a smaller package which requires less system resources for ordinary user interfaces like the “Standard Dialogs” on page 3.

The following table lists all the methods that are provided by the PlainUserDialog class. The individual methods are documented in “Chapter 8. BaseDialog Class” on page 89 or “Chapter 10. UserDialog Class” on page 227.

Attributes:

Instances of the *PlainUserDialog* class have the following attributes:

AutoDetect
Automatic data field detection on (=1, default) or off (=0). For the *UserDialog* subclass the default is off and *Connect...* methods or a resource script are usually used.

AutomaticMethods
A queue containing the methods that are started concurrently before the execution of the dialog

ConstDir
A directory string storing the numerical values assigned to symbolic IDs (#define-statements in the resource script)

DataConnection
Protected attribute to store connections between dialog items and the attributes of the dialog instance

DlgHandle
A handle to the dialog

Finished
0 if dialog is executing, 1 if terminated with OK, and 2 if canceled

InitCode
Result of the *init* method; in case *init* failed, its value is 1.

UseStem
Protected attribute that is true (=1) if a stem variable was passed to *init*

Methods:
Instances of the *PlainUserDialog* class implement the methods listed in Table 4.

Table 4. PlainUserDialog and PlainBaseDialog Class Methods

Method...	...on page
AddAttribute	124
AddBitmapButton	238
AddBlackFrame	258
AddBlackRect	258
AddButton	237
AddButtonGroup	256
AddCheckBox	246
AddCheckBoxStem	254
AddCheckGroup	249
AddComboBox	245
AddComboEntry	135
AddComboInput	252
AddEntryLine	241
AddGrayFrame	258
AddGrayRect	258
AddGroupBox	240
AddInput	249
AddInputGroup	251
AddInputStem	252
AddListBox	244
AddListEntry	142
AddMenuItem	261

Table 4. PlainUserDialog and PlainBaseDialog Class Methods (continued)

Method...	**...on page**
AddMenuSeparator	262
AddOkCancelLeftBottom	259
AddOkCancelLeftTop	260
AddOkCancelRightBottom	259
AddOkCancelRightTop	259
AddPasswordLine	243
AddPopupMenu	261
AddRadioButton	246
AddRadioGroup	248
AddRadioStem	255
AddScrollBar	255
AddText	240
AddUserMsg	122
AddWhiteFrame	258
AddWhiteRect	257
AssignWindow	186
AutoDetection	109
Cancel	134
Center	158
ChangeComboEntry	139
ChangeListEntry	145
ClearMessages	107
ComboAddDirectory	139
ComboDrop	140
ConnectButton	111
ConnectCheckBox	117
ConnectComboBox	117
ConnectControl	114
ConnectEntryLine	116
ConnectList	115
ConnectListBox	118
ConnectListLeftDoubleClick	115

Table 4. PlainUserDialog and PlainBaseDialog Class Methods (continued)

Method...	**...on page**
ConnectMultiListBox	118
ConnectRadioButton	117
Create	231
CreateCenter	232
CreateMenu	261
DefineDialog	233
DeInstall	135
DeleteComboEntry	136
DeleteListEntry	142
Disable	191
DisableItem	153
Enable	190
EnableItem	153
Execute	100
FindComboEntry	137
FindListEntry	143
FocusItem	152
Get	150
GetAttrib	131
GetButtonRect	151
GetCheckBox	130
GetComboEntry	137
GetComboItems	137
GetComboLine	129
GetCurrentComboIndex	138
GetCurrentListIndex	145
GetData	125
GetDataStem	133
GetEntryLine	126
GetID	187
GetItem	151
GetListEntry	143

Table 4. PlainUserDialog and PlainBaseDialog Class Methods (continued)

Method...	**...on page**
GetListItems	144
GetListLine	127
GetMultiList	128
GetPos	189
GetRadioButton	129
GetSize	189
GetTextSize	215
GetValue	130
HandleMessages	106
Help	134
Hide	191
HideItem	153
HideWindow	154
Init	98
InitAutoDetection	108
InitDialog	98
InsertComboEntry	136
InsertListEntry	142
IsDialogActive	104
ItemTitle	126
Leaving	135
ListAddDirectory	146
ListDrop	146
Load	234
LoadFrame	235
LoadItems	236
LoadMenu	262
Move	195
NoAutoDetection	108
OK	133
Resize	193
Run	99

Table 4. PlainUserDialog and PlainBaseDialog Class Methods (continued)

Method...	**...on page**
SetAttrib	132
SetCheckBox	130
SetComboLine	129
SetCurrentComboIndex	138
SetCurrentListIndex	145
SetData	125
SetDataStem	132
SetEntryLine	126
SetListLine	127
SetListTabulators	146
SetMenu	262
SetMultiList	128
SetRadioButton	130
SetStaticText	126
SetTitle	197
SetValue	131
SetWindowTitle	159
Show	105
ShowItem	154
ShowWindow	155
StartIt	260
StopIt	104
Title	196
Title=	196
Update	196
Validate	134

Chapter 12. ResDialog Class

The *ResDialog* class is designed to be used together with a binary (compiled) resource. A binary dialog resource is linked to a DLL (that is, a file with the extension .DLL).

Requires:

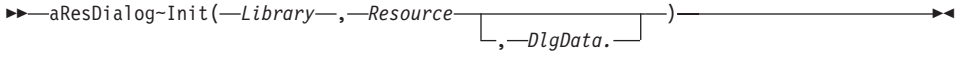
`ResDlg.cls` is the source file of this class. Use the tokenized version of OODialog, `oodialog.cls`, to shorten your dialog's startup time:

```
::requires oodialog.cls
```

Subclass:

The *ResDialog* class is a subclass of *BaseDialog*.

Init



The *Init* method of the parent class, *BaseDialog*, has been overwritten.

Arguments:

The arguments you have to pass to the *new* method of the class when creating a new dialog instance are:

Library

The file name of the DLL where the resource is located

Resource

The ID of the resource. This is a unique number you assigned to the (dialog) resource while creating it.

DlgData.

A stem variable (don't forget the trailing period) that contains initialization data. See Init for more details.

Example:

This sample code creates a new dialog object from the *ResDialog* class. It uses dialog resource 100 in the `MYDLG.DLL` file. The dialog is initialized with the values of the *MyDlgData.* stem variable.

```
MyDlgData.101="1"
MyDlgData.102="Please enter your password."
MyDlgData.103=""

dlg = ResDialog~new("MYDLG.DLL", 100, MyDlgData.)
```

StartIt

The *StartIt* method is for internal use only. It is necessary to create a real Windows object based on the dialog template.

Protected:

This method is protected and cannot be called from outside the instance. It can be overwritten, although that is not recommended.

Arguments:

There is only one argument:

icon This argument is currently not used.

SetMenu

The *SetMenu* method adds a menu resource, that is stored in the same DLL, to the dialog. Note that the menu needs additional space and therefore displaces the rest of the dialog items.

SetMenu can be called in the InitDialog method only.

Arguments:

There is only one argument:

resid ID of the menu resource stored in the same DLL as the dialog.

Chapter 13. CategoryDialog Class

The *CategoryDialog* class creates and controls a dialog that has more than one panel. It is similar to the *notebook* control available in OS/2 or the *property sheet* available in the Windows 95 user interface.

Depending on the style you choose, you can switch among different pages by either clicking radio buttons or selecting an item from a drop down list. Each page has its own window controls.

Requires:

`CatDlg.cls` is the source file of this class. Use the tokenized version of OODialog, `oodialog.cls`, to shorten your dialog's startup time:

```
::requires oodialog.cls
```

Subclass:

The *CategoryDialog* class is a subclass of *UserDialog* (see "Chapter 10. UserDialog Class" on page 227).

Attributes:

Instances of the *CategoryDialog* class have the following attributes:

Catalog

A directory describing the layout and behavior of the dialog. This directory is usually set up in the *InitCategories* method of the dialog (see "InitCategories" on page 275 for more information).

StaticID

An internal counter

Methods:

Instances of the *CategoryDialog* class implement the methods listed in the following table.

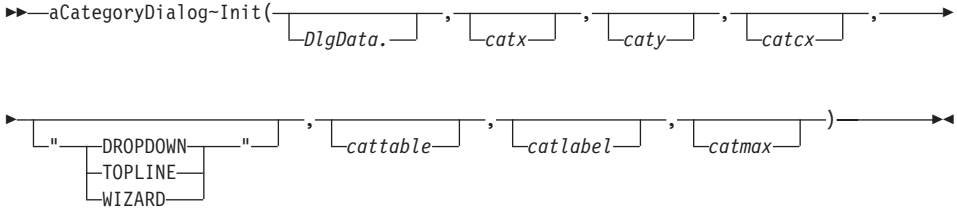
Note: In fact, most of the methods do the same as the methods in the parent class, *UserDialog*, except that they are enabled to work with a category dialog.

Method...	...on page
AddCategoryComboEntry	286
AddCategoryListEntry	288
CategoryComboAddDirectory	287
CategoryComboDrop	288

Setting Up the Dialog

The following methods are used to set up the pages of the dialog and start it.

Init



The *Init* method initializes the category dialog object.

Arguments:

The arguments are:

DlgData.
A stem variable (do not forget the trailing period) that contains initialization data for some or all dialog items. If the dialog is terminated by means of the **OK** button, the values of the dialog's data fields are copied to this variable. The ID of the dialog items is used to name the entry within the stem.

catx, caty
The position of the category selection control group (radio buttons or combo box). The defaults are 10 and 4.

catcx This argument sets the length of one item of the control group (calculated if omitted)

style This argument determines the style of the category dialog. Without one of the following keywords, the category selection is done by a vertical radio button group:

DROPDOWN
Creates a drop-down list at the top (useful if there are many categories)

TOPLINE
Draws a horizontal radio button group at the top of the client area

WIZARD
Adds *Backward* and *Forward* buttons with IDs 11 and 12 to switch between category pages

Without *DROPDOWN* and *TOPLINE* the default category selection is done by a vertical radio button group, with the dialog pages to the right of the radio buttons.

cattable
This argument can be used to set the category names separated by blanks. If omitted, set the category names in the InitCategories method.

catlabel
This argument defines the label for the combo box in *DROPDOWN* style (default is "Page:")

catmax
This argument sets the split point of the radio button group in default style, or the number of entries in the combo box drop-down list.

Example:
The following example creates a category dialog, using a combo box as the selection control:

```
dlg = MyCategoryDialog~new(MyData.,,,,"DROPDOWN", ,
"Movies Cinemas Days Ticket","Dialog panel:")
dlg~createCenter(200,180,"Let's go to the movies")
dlg~execute("SHOWTOP")
...
::class MyCategoryDialog subclass CategoryDialog

::method Movies        /* define the Movies page  */
...
::method Cinemas       /* define the Cinemas page */
...
```

InitCategories

The *InitCategories* method is called by *Init* to set the characteristics of the category dialog.

Protected:
This method is protected.

Catalog directory:
The *InitCategories* should set up the *Catalog* directory with information about the layout and the behavior of the dialog. The directory entries are:

names An array containing the names of the categories. The array is initialized with the names given in the *Init* method (argument

cattable). These names are used as labels for page selection control and as messages sent to the object to define the single pages.

Your subclass must provide a method for each category page—with the same name as the label in this directory—to define the dialog page using LoadItems to load the dialog items from a resource script or *Add...* methods. Notice that blanks are removed when you call the *Define...* methods.

If your subclass provides methods with the prefix Init followed by the name of the categories (blanks removed), these methods are called by InitDialog to initialize the dialog (each page is a dialog) that contains the corresponding category.

Unless you already specified the categories with the *Init* method, you must assign an array to this *Catalog* entry.

count Number of categories

handles
For internal use only

id For internal use only

category
For internal use only

page A directory with the following entries:

font Name of the font used for the dialog

fsize Size of the font

style Style of the dialog (see "Create" on page 231)

expected
Total number of expected dialog items of all category pages (200)

btnwidth
Width of *Backward* and *Forward* push buttons (see *WIZARD* in *Init* method)

leftbtntext
Alternate label of *Backward* button

rightbtntext
Alternate label of *Forward* button

The next four entries should not be modified:

x Horizontal position of the category pages relative to the parent dialog

y Vertical position of the category pages relative to the parent dialog

w Width of the category pages

h Height of the category pages

Example:

The following example sets the category names to *Text Editor, Compiler, Linker*, and *Debugger*. The subclass of *CategoryDialog* must define four methods named after them.

```
::class MyCategoryDialog subclass CategoryDialog
::method InitCategories
   self~catalog['names']= .array~of("Text Editor","Compiler","Linker",,
   "Debugger")
   self~catalog['page']['leftbtntext'] = '&Previous'
   self~catalog['page']['rightbtntext'] = '&Next'

::method TextEditor       /* blanks are removed when the message is sent
                             from DefineDialog */
                          /* add control to the first page */
   :
::method Compiler
   :
::method InitTextEditor   /* called by InitDialog */
                          /* initialize the first dialog */
   :
```

DefineDialog

The *DefineDialog* method is called after the main dialog has been created. This method must not be overwritten in a subclass because it defines the layout of the frame window and calls the definition methods for each category page.

Protected:

This method is protected.

Example:

Assume that you have the categories "Common Data", "Company Data", and "Special". The following methods are automatically called by *DefineDialog* to add dialog items to the associated page:

CommonData to add controls to the first category page.

CompanyData to add controls to the second category page.

Special to add controls to the third category page.

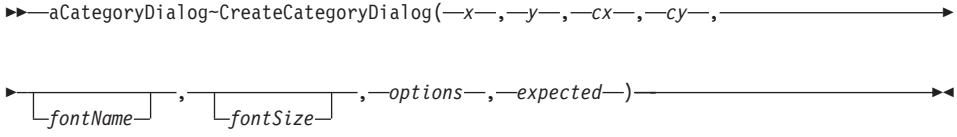
See "InitCategories" on page 275 for more information.

CategoryPage

The *CategoryPage* method adds controls to the base window of a Category Dialog. It is used to define the layout of the parent dialog that contains the single pages.

Protected:
This method is protected and should not be overwritten or called. Use the InitCategories method to set up the dialog.

CreateCategoryDialog



The *CreateCategoryDialog* method creates the category dialog.

Protected:
This method is protected. It is called by another method and usually does not have to be called manually.

InitDialog

The *InitDialog* method is called after the Windows dialog has been created and before the category dialog is to be displayed.

Do not override this method to initialize your category dialog pages but define an *Init...* method for each of your pages that you want to initialize like adding combo and list box items. If your subclassed dialog defines a method that has the prefix "Init" followed by the name of the category (without blanks), this method is called by *InitDialog* to handle the initialization of the corresponding page. If you use a method that requires a category specifier,

such as AddCategoryComboEntry or GetTreeControl, and you omit the category, OODialog assumes that the dialog item addressed is part of the page that contains the current category.

Protected:
This method is protected.

Example:
Assume that you have the categories "Common Data", "Company Data", and "Special". The following methods are automatically called when provided by the subclass:

InitCommonData	to initialize the first category.
InitCompanyData	to initialize the second category.
InitSpecial	to initialize the third category.

See "InitCategories" on page 275 for more information.

GetSelectedPage

The *GetSelectedPage* method is used internally to return the currently selected page using the combo box or radio buttons (1 indicates the first page).

CurrentCategory

The *CurrentCategory* method returns the number of the current dialog page. The first page is numbered 1.

Example:
The following example tests the current page number:

```
if MyCategoryDialog~CurrentCategory=2 then do ...
```

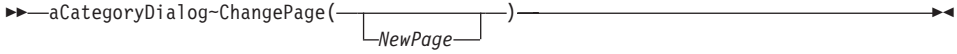
NextPage

The *NextPage* method switches the dialog to the next category page.

PreviousPage

The *PreviousPage* method switches the dialog to the previous category page.

ChangePage



The *ChangePage* method switches the dialog to another page and returns the new page number. It is also called by selection control to activate the selected page. *ChangePage* invokes PageHasChanged after the new page is activated.

Arguments:

The only argument is:

NewPage

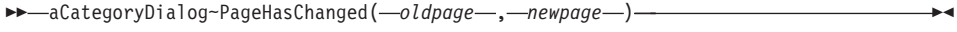
The page number of the new page (default is the page selected by the combo box or radio button)

Example:

The following example activates the second category page:

```
MyCategoryDialog~ChangePage(2)
```

PageHasChanged



The *PageHasChanged* method is invoked by ChangePage when a new page is activated. The default implementation returns without an action. The user can override this method to react to page changes.

Arguments:

The arguments are:

oldpage

The page number of the previous page

newpage

The page number of the new page

StartIt

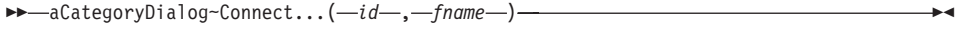
The *StartIt* method is called by Execute to create a real Windows object based on the dialog template. You might override it in your subclass, but be sure to forward the message to the parent method.

Protected:
This method is protected.

Example:
This is a template for overwriting base methods:

```
::class MyCatDlg subclass CatergoryDialog
::method StartIt
   say "this is method 'StartIt' !"
   self~StartIt:super()
```

Connect... Methods



The *Connect...* methods connect data dialog items of certain types with the dialog object. The *Connect...* methods should be placed into the user-defined methods with the names of the categories defined in the InitCategories method. The *Connect...* methods are defined for the *BaseDialog* class. For more information, see "Chapter 8. BaseDialog Class" on page 89.

Arguments:
The arguments are:

id The ID of the dialog item

fname The name of the object attribute

Example:
The following example connects an entry line in the *Movies* page with the *FIRSTNAME* object attribute:

```
::method InitCategories
   self~catalog['names'] = .array~of("Movies",...)
...
::method Movies
  self~ConnectEntryLine(101, "FIRSTNAME")
```

Note: IDs for dialog elements need not be unique across all pages. However, IDs for buttons and list boxes that are connected to methods must be unique for the whole category dialog.

Methods for Dialog Items

The methods listed in this section deal with individual dialog items on one of the pages of the category dialog.

The methods correspond to methods with similar names of the *BaseDialog* class; the word *Category* is inserted between the verb and the dialog item in the method name. For example, *AddCategoryComboEntry* for the *CategoryDialog* class has the same function as *AddComboEntry* of the *BaseDialog* class.

Note: The methods listed here have the same parameters as the corresponding methods of the *BaseDialog* class, with the number of the category page as an extra parameter.

Another way to directly address dialog items of a category dialog is to retrieve an object of the DialogControl class (see page 181) or one of its derivates that is associated with the requested dialog control. To retrieve such an object, you can call one of the following methods depending on the requested control:

- "GetStaticControl" on page 338
- "GetEditControl" on page 339
- "GetButtonControl" on page 340
- "GetRadioControl" on page 341
- "GetCheckControl" on page 342
- "GetListBox" on page 342
- "GetComboBox" on page 343
- "GetScrollBar" on page 345
- "GetTreeControl" on page 345
- "GetListControl" on page 346
- "GetProgressBar" on page 347
- "GetSliderControl" on page 348
- "GetTabControl" on page 349

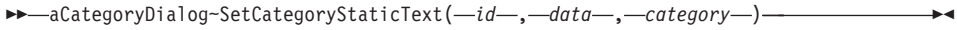
To use these methods and the resulting objects, your category dialog must inherit from the mixin class *AdvancedControls*. You can do this by adding the keyword "inherit" followed by the mixin class name "AdvancedControls". For example:

```
::class MyCategory subclass CategoryDialog public inherit AdvancedControls
```

Get and Set Methods

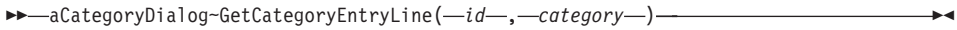
The following sections describe the individual Get and Set methods.

SetCategoryStaticText



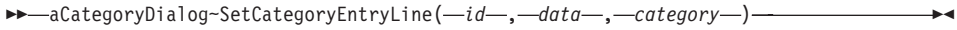
For more information, see “SetStaticText” on page 126.

GetCategoryEntryLine



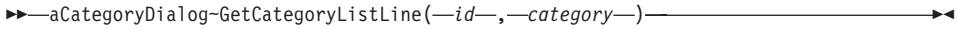
For more information, see “GetEntryLine” on page 126.

SetCategoryEntryLine



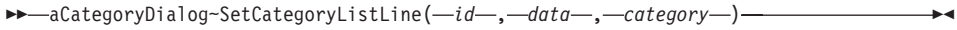
For more information, see “SetEntryLine” on page 126.

GetCategoryListLine



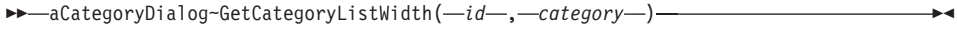
For more information, see “GetListLine” on page 127.

SetCategoryListLine



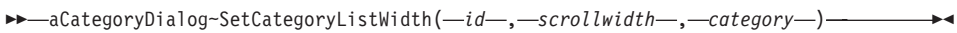
For more information, see “SetListLine” on page 127.

GetCategoryListWidth



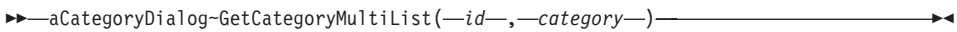
For more information, see “GetListWidth” on page 140.

SetCategoryListWidth



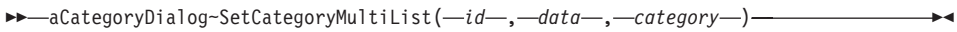
For more information, see "SetListWidth" on page 141.

GetCategoryMultiList



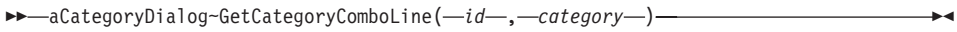
For more information, see "GetMultiList" on page 128.

SetCategoryMultiList



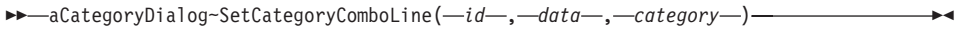
For more information, see "SetMultiList" on page 128.

GetCategoryComboLine



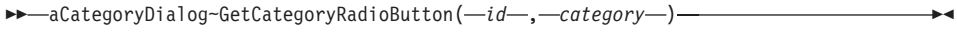
For more information, see "GetComboLine" on page 129.

SetCategoryComboLine



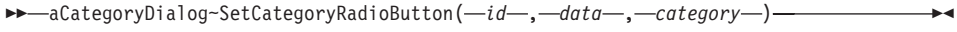
For more information, see "SetComboLine" on page 129.

GetCategoryRadioButton



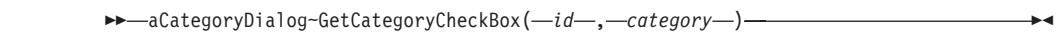
For more information, see "GetRadioButton" on page 129.

SetCategoryRadioButton



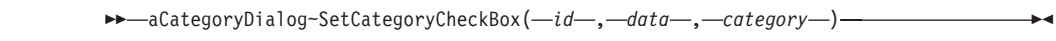
For more information, see “SetRadioButton” on page 130.

GetCategoryCheckBox



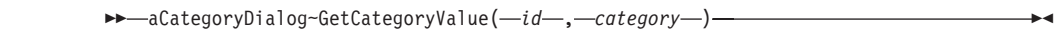
For more information, see “GetCheckBox” on page 130.

SetCategoryCheckBox



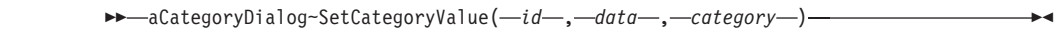
For more information, see “SetCheckBox” on page 130.

GetCategoryValue



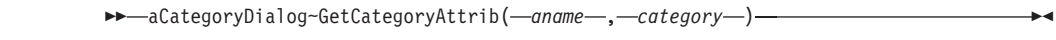
For more information, see “GetValue” on page 130.

SetCategoryValue



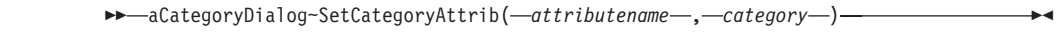
For more information, see “SetValue” on page 131.

GetCategoryAttrib



For more information, see “GetAttrib” on page 131.

SetCategoryAttrib

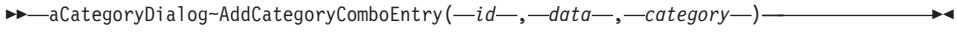


For more information, see “SetAttrib” on page 132.

Combo Box Methods

The following sections describe the individual combo box methods.

AddCategoryComboEntry



For more information, see "AddComboEntry" on page 135.

Arguments:

The arguments are:

id The ID of the combo box

data The text string that is added to the combo box

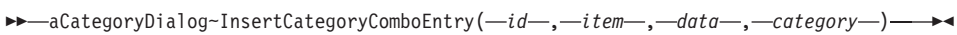
category
The category page number where the combo box is located

Example:

The following example adds a text string to the list of the combo box 101 in the third category page.

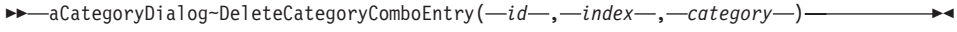
```
MyCategoryDialog~AddCategoryComboEntry(101, "I'm one of the choices", 3)
```

InsertCategoryComboEntry



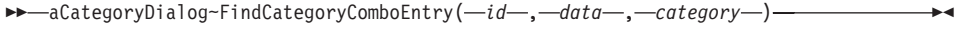
For more information, see "InsertComboEntry" on page 136.

DeleteCategoryComboEntry



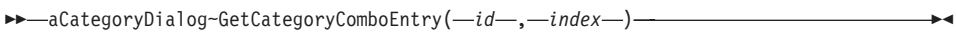
For more information, see method "DeleteComboEntry" on page 136.

FindCategoryComboEntry



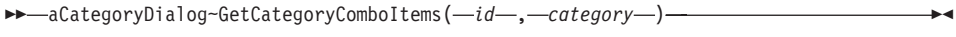
For more information, see "FindComboEntry" on page 137.

GetCategoryComboEntry



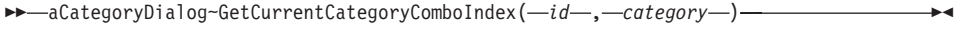
For more information, see “GetComboEntry” on page 137.

GetCategoryComboItems



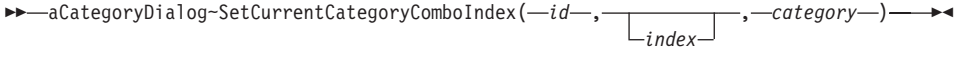
For more information, see “GetComboItems” on page 137.

GetCurrentCategoryComboIndex



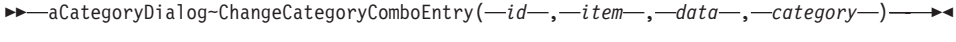
For more information, see “GetCurrentComboIndex” on page 138.

SetCurrentCategoryComboIndex



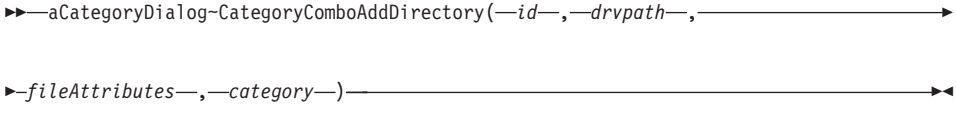
For more information, see “SetCurrentComboIndex” on page 138.

ChangeCategoryComboEntry



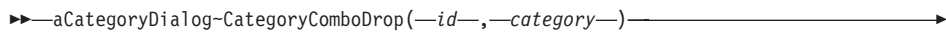
For more information, see “ChangeComboEntry” on page 139.

CategoryComboAddDirectory



For more information, see “ComboAddDirectory” on page 139.

CategoryComboDrop

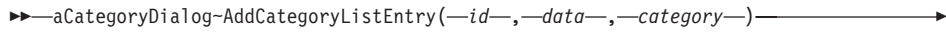


For more information, see "ComboDrop" on page 140.

List Box Methods

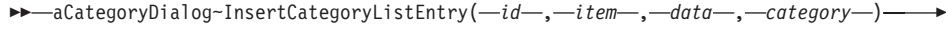
The following sections describe the individual list box methods.

AddCategoryListEntry



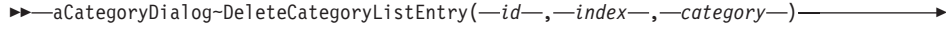
For more information, see "AddListEntry" on page 142.

InsertCategoryListEntry



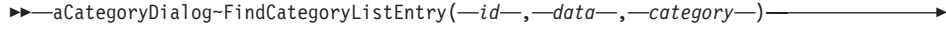
For more information, see "InsertListEntry" on page 142.

DeleteCategoryListEntry



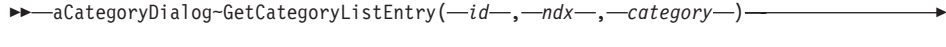
For more information, see "DeleteListEntry" on page 142.

FindCategoryListEntry



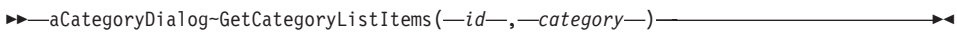
For more information, see "FindListEntry" on page 143.

GetCategoryListEntry



For more information, see "GetListEntry" on page 143.

GetCategoryListItems



For more information, see “GetListItems” on page 144.

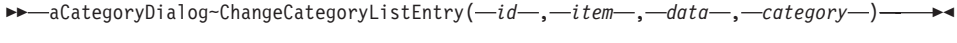
GetCurrentCategoryListIndex

For more information, see “GetCurrentListIndex” on page 145.

SetCurrentCategoryListIndex

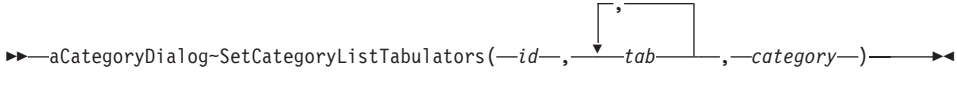
For more information, see “SetCurrentListIndex” on page 145.

ChangeCategoryListEntry



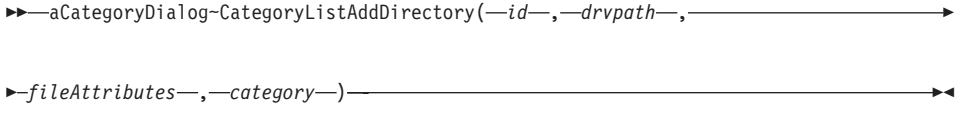
For more information, see “ChangeListEntry” on page 145.

SetCategoryListTabulators



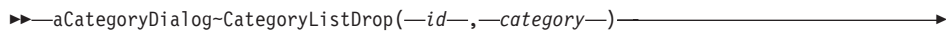
For more information, see “SetListTabulators” on page 146.

CategoryListAddDirectory



For more information, see “ListAddDirectory” on page 146.

CategoryListDrop

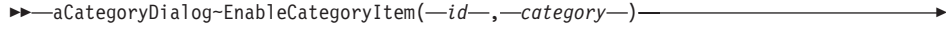


For more information, see “ListDrop” on page 146.

Appearance Modification Methods

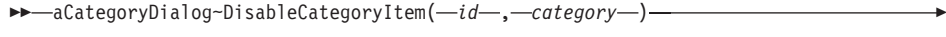
The following sections describe the methods affecting the appearance of the item.

EnableCategoryItem



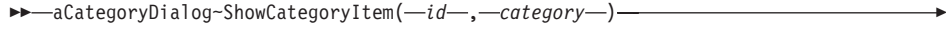
For more information, see “EnableItem” on page 153.

DisableCategoryItem



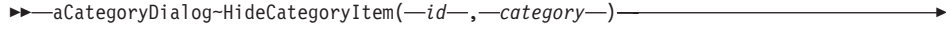
For more information, see “DisableItem” on page 153.

ShowCategoryItem



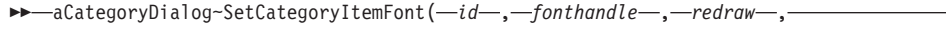
For more information, see “ShowItem” on page 154.

HideCategoryItem



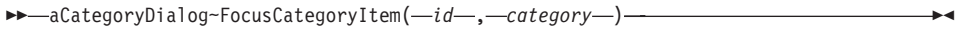
For more information, see “HideItem” on page 153.

SetCategoryItemFont



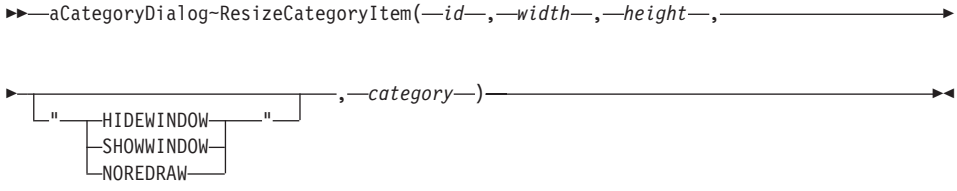
For more information, see "SetItemFont" on page 170.

FocusCategoryItem



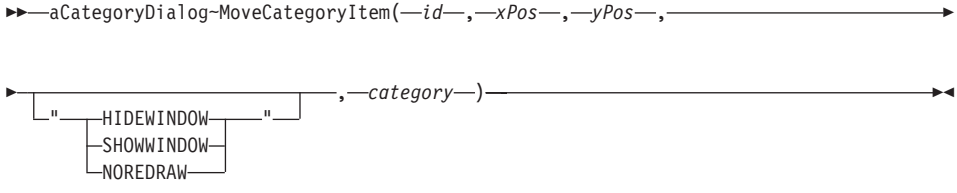
For more information, see "FocusItem" on page 152.

ResizeCategoryItem



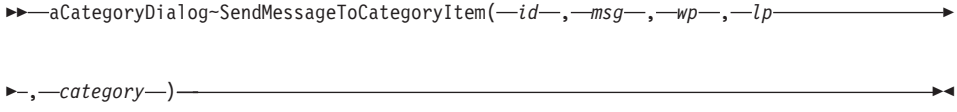
For more information, see "ResizeItem" on page 157.

MoveCategoryItem



For more information, see "MoveItem" on page 158.

SendMessageToCategoryItem



For more information, see "SendMessageToItem" on page 107.

Chapter 14. Standard Dialog Classes and Functions

The standard dialog classes are:
- TimedMessage
- InputBox
- PasswordBox
- IntegerBox
- MultiInputBox
- ListChoice
- MultipleListChoice
- CheckList
- SingleSelection

Requires:

`StdDlg.cls` is the source file for the standard dialog classes. Use the tokenized version of OODialog, `OODPLAIN.CLS`, to shorten your dialog's startup time:

```
::requires "OODPLAIN.CLS"
```

Preparation:

Standard dialogs are prepared by using the *new* method of the class, which in turn invokes the *Init* method. The parameters are described for the *Init* method of each class.

Execution:

The dialog is then run by using the *Execute* method. *Execute* returns the user's input if the **OK** button is clicked and the null string if the **Cancel** button is clicked to terminate the dialog. If there is more than one return value, *Execute* returns the value 1 and stores the results in an attribute.

Functions:

Each standard dialog is also available as a callable function.

TimedMessage Class

The *TimeMessage* class shows a message window for a defined duration.

Requires:

`OODPLAIN.CLS` is required to use this class

Subclass:

This class is a subclass of the "PlainUserDialog" class

Execute:

Returns 1

The methods listed below are defined by this class.

Init

```
►►—aTimedMessage~Init(message,title,sleeping)———————————————————————————►◄
```

The *Init* method prepares the dialog.

Arguments:

The arguments are:

message
A string that is displayed inside the window as a message. The length of the message determines the horizontal size of all standard dialogs.

title A string that is displayed as the window title in the title bar of the dialog

sleeping
A number that determines how long (in milliseconds) the window is shown

Example:

The following example shows a window with the *Information* title for a duration of 3 seconds:

```
dlg = .TimedMessage~New("Application will be started, please wait",,
                        "Information", 3000)
dlg~Execute
drop dlg
```

DefineDialog

```
►►—aTimedMessage~DefineDialog———————————————————————————————————————————►◄
```

The *DefineDialog* method is called by the *Create* method of the parent class, *PlainUserDialog*, which in turn is called at the very beginning of *Execute*. You do not have to call it. However, you may want to override it in your subclass to add more dialog controls to the window. If you override it, you have to forward the message to the parent class by using the keyword *super*.

Example:

The following example shows how to subclass the *TimedMessage* class and how to add a background bitmap to the dialog window:

```
::class MyTimedMessage subclass TimedMessage inherit DialogExtensions

::method DefineDialog
   self~BackgroundBitmap("mybackg.bmp", "USEPAL")
   self~DefineDialog:super()
```

Execute

The *Execute* method creates and shows the message window. After the given time (see *Init* method), it destroys the dialog automatically.

TimedMessage Function

OODialog provides a shortcut function to invoke a *TimedMessage* dialog as a function:

```
ret = TimedMessage("We are starting...","Please wait",3000)
```

The parameters are described in the *Init* method.

InputBox Class

The *InputBox* class provides a simple dialog with a title, a message, one entry line, an **OK**, and a **Cancel** push button.

Requires:
OODPLAIN.CLS is required to use this class

Subclass:
This class is a subclass of the "PlainUserDialog" class

Execute:
Returns the user's input

The methods listed below are defined by this class.

Init

The *Init* method prepares the input dialog.

Arguments:
The arguments are:

message
A text string that is displayed in the dialog

title A string that is displayed as the dialog's title in the title bar

preval A string to initialize the entry line. If you do not want to put any text in the entry line, just pass an empty string.

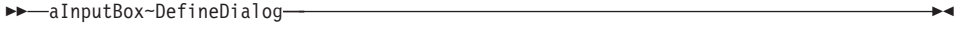
len The width of the entry line in dialog units

Example:

The following example shows a dialog with the *Input* title and an entry line:

```
dlg = .InputBox~New("Please enter your email address", ,
"Input", "user@host.domain", 150)
value = dlg~Execute
say "You entered:" value
drop dlg
```

DefineDialog



The *DefineDialog* method is called by the *Create* method of the parent class, *PlainUserDialog*, which in turn is called at the very beginning of *Execute*. You do not have to call it. However, you may want to override it in your subclass to add more dialog controls to the window. If you override it, you have to forward the message to the parent class by using the keyword *super*.

AddLine

The *AddLine* method is used internally to add one entry line to the dialog.

Execute

The *Execute* method creates and shows the dialog. After termination, the value of the entry line is returned if the user clicks the **OK** button; a null string is returned if the user clicks on **Cancel**.

InputBox Function

OODialog provides a shortcut function to invoke an *InputBox* dialog as a function:

```
say "Your name:" InputBox("Please enter your name","Personal Data")
```

The parameters are described in the *Init* method.

PasswordBox Class

The *PasswordBox* class is an *InputBox* dialog with an entry line that echoes the keys with asterisks (*) instead of characters.

Requires:
OODPLAIN.CLS is required to use this class

Subclass:
This class is a subclass of the InputBox Class

Execute:
Returns the user's password

The methods are the same as for the InputBox Class, with the exception of *AddLine*.

AddLine

```
►►—aPasswordBox~AddLine(—x—,—y—,—l—)————————————————————————————►◄
```

The *AddLine* overrides the same method of the parent class, *InputBox*, by using a password entry line instead of a simple entry line.

PasswordBox Function

OODialog provides a shortcut function to invoke a *PasswordBox* dialog as a function:

```
pwd = PasswordBox("Please enter your password","Security")
```

The parameters are described in the *Init* method of the *InputBox* class.

IntegerBox Class

The *IntegerBox* class is an *InputBox* dialog whose entry line allows only numerical data.

Requires:
OODPLAIN.CLS is required to use this class

Subclass:
This class is a subclass of the InputBox Class class

Execute:
Returns the user's numeric input

The methods are the same as for the InputBox Class class, with the exception of *Validate*.

Validate

The only method this subclass overrides is *Validate*, which is one of the automatically called methods of *PlainUserDialog*. It is invoked by the *OK* method, which in turn is called in response to a push button event. This method checks whether or not the entry line contains a valid numerical value. If the value is invalid, a message window is displayed.

IntegerBox Function

OODialog provides a shortcut function to invoke an *IntegerBox* dialog as a function:

```
say "Your age:" IntegerBox("Please enter your age","Personal Data")
```

The parameters are described in the *Init* method of the *InputBox* class.

MultiInputBox Class

The *MultiInputBox* class is a dialog that provides a title, a message, and one or more entry lines. After execution of this dialog you can access the values of the entry lines.

Requires:

OODPLAIN.CLS is required to use this class

Subclass:

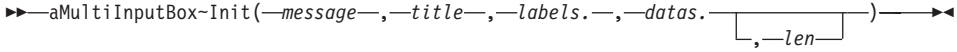
This class is a subclass of the "PlainUserDialog" class

Execute:

Returns 1 (if **OK** was clicked). The values entered by the user are stored in attributes matching the labels of the entry lines.

The methods are the same as for the InputBox Class class, with the exception of *Init*.

Init



The *Init* method is called automatically whenever a new instance of this class is created. It prepares the dialog.

Arguments:

The arguments are:

message
A text string that is displayed on top of the entry lines. Use it to give the user advice on what to do.

title A text string that is displayed in the title bar.

labels. A stem variable containing strings that are used as labels on the left side of the entry lines. *Labels.1* becomes the label for the first entry line, *labels.2* for the second, and so forth.

datas. A stem variable (do not forget the trailing period) containing strings that are used to initialize the entry lines. The entries must start with 101 and continue in increments of 1.

len The length of the entry lines. All entry lines get the same length.

Example:

The following example creates a four-line input box. The data entered is stored in the object attributes that are displayed after dialog execution.

```
lab.1 = "First name"  ;     lab.2 = "Last name "
lab.3 = "Street and City" ; lab.4 = "Profession:"

addr.101 = "John" ; addr.102 = "Smith" ; addr.103 = ""
addr.104 = "Software Engineer"

dlg = .MultiInputBox~new("Please enter your address", ,
"Your Address", lab., addr.)
if dlg~execute = 1 then do
  say "The address is:"
  say dlg~firstname dlg~lastname
  say dlg~StreetandCity
  say dlg~Profession
end
```

MultiInputBox Function

OODialog provides a shortcut function to invoke a *MultiInputBox* dialog as a function:

```
res = MultiInputBox('Enter your address','Personal Data', ,
                    .array~of("&First name","Last &name","&City"), ,
                    .array~of("John","Smith",'San Jose'), 100)
if res ¬= .Nil then do entry over res
                       say 'Address-line[]= ' entry
                    end
```

The parameters are described in the *Init* method, but, instead of stems, arrays are passed into and returned from the function.

ListChoice Class

The *ListChoice* class provides a dialog with a list box, an **OK**, and a **Cancel** button. The selected item is returned if the **OK** push button is used to terminate the dialog.

Requires:

OODPLAIN.CLS is required to use this class

Subclass:

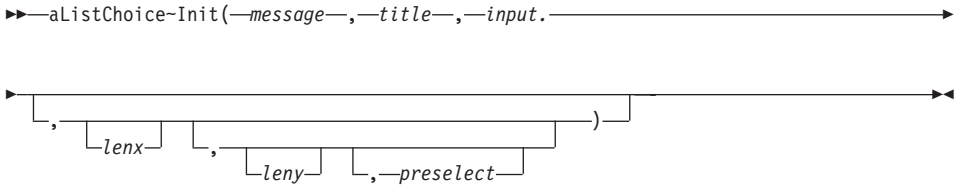
This class is a subclass of the "PlainUserDialog" class

Execute:

Returns the user's choice or a null string

The method listed below is defined by this class.

Init



The *Init* method is used to initialize a newly created instance of this class.

Arguments:

The arguments are:

message
A text string that is displayed on top of the list box. Use it to give the user advice on what to do.

title A text string for the dialog's title

input. A stem variable (do not forget the trailing period) containing string values that are inserted into the list box

lenx, leny
The size of the list box in dialog units

preselect
Entry that is selected when list pops up

Example:

The following example creates a list choice dialog box where the user can select exactly one dessert:

```
lst.1 = "Cookies"; lst.2 = "Pie"; lst.3 = "Ice cream"; lst.4 = "Fruit"

dlg = .ListChoice~new("Select the dessert please","YourChoice",lst.,,,"Pie")
say "Your ListChoice data:" dlg~execute
```

ListChoice Function

OODialog provides a shortcut function to invoke a *ListChoice* dialog as a function:

```
day = ListChoice('Select a day','My favorite day', ,
                 .array~of("Monday","Tuesday","Wednesday","Thursday",,
                           "Friday","Saturday","Sunday") ,,,"Thursday")
say "Your favorite day is" day
```

The parameters are described in the *Init* method, but, instead of an input stem an array is passed into the function.

MultiListChoice Class

The *MultiListChoice* class is an extension of the *ListChoice* class. It makes it possible for the user to select more than one line at a time. The *Execute* method returns the selected items' indexes separated by blank spaces. The first item has index 1.

Requires:

OODPLAIN.CLS is required to use this class

Subclass:

This class is a subclass of the ListChoice Class

Execute:

Returns the index numbers of the entries selected

Preselect:

Indexes of entries, separated by a blank, that are to be preselected. The first entry has index 1 and the rest are increments of one.

The methods are the same as for the ListChoice Class class, except that *Execute* returns the index numbers of the selected entries.

Example:

The following example creates a multiple list choice box where the user can select multiple entries:

```
lst.1 = "Monday" ;   lst.2 = "Tuesday" ; lst.3 = "Wednesday"
lst.4 = "Thursday" ; lst.5 = "Friday" ;  lst.6 = "Saturday"
lst.7 = "Sunday"

dlg = .MultiListChoice~new("Select the days you are working this week", ,
                           "YourMultipleChoice",lst.,,,"2 5")
s = dlg~execute
```

```
if s ¬= '' then do while s ¬= ''
                  parse var s res s
                  say lst.res
                end
```

MultiListChoice Function

OODialog provides a shortcut function to invoke a *MultiListChoice* dialog as a function:

```
days = MultiListChoice('Select days','My TV Days', ,
                       .array~of("Monday","Tuesday","Wednesday", ,
                              "Thursday","Friday","Saturday","Sunday"),,,"2 5")
if days ¬= .Nil then do day over days
                      say 'TV day =' day
                    end
```

The parameters are described in the *Init* method, but, instead of stems, arrays are passed into and returned from the function.

CheckList Class

The *CheckList* class is a dialog with a group of one or more check boxes.

Requires:

OODPLAIN.CLS is required to use this class.

Subclass:

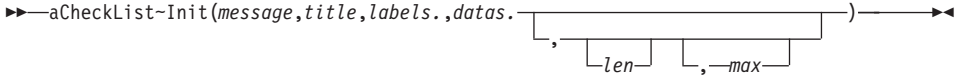
This class is a subclass of the "PlainUserDialog" class.

Execute:

Returns 1 (if **OK** was clicked). The check boxes selected by the user are marked in a stem variable with the value 1.

The method listed below is defined by this class.

Init



Arguments:

The arguments are:

message
A text string that is displayed on top of the check box group. Use it to give the user advice on what to do.

title A text string for the dialog's title

labels. A stem variable (do not forget the trailing period) containing all the labels for the check boxes

datas. This argument is a stem variable (do not forget the trailing period) that you can use to preselect the check boxes. The first check box relates to stem item 101, the second to 102, and so forth. A value of 1 indicates *selected*, and a value of 0 indicates *deselected*.

For example, *Datas.103=1* indicates that there is a check mark on the third box.

len Determines the length of the check boxes and labels. If omitted, the size is calculated to fit the largest label.

max The maximum number of check boxes in one column. If there are more check boxes than *max* – that is, *labels.* has more items than the value of *max* – this method continues with a new column.

Example:

The following example creates and shows a dialog with seven check boxes:

```
lst.1 = "Monday";   lst.2 = "Tuesday"; lst.3 = "Wednesday"
lst.4 = "Thursday"; lst.5 = "Friday";  lst.6 = "Saturday"
lst.7 = "Sunday"

do i = 101 to 107
   chk.i = 0
end

dlg = .CheckList~new("Please select a day!","Day of week",lst., chk.)
if dlg~execute = 1 then do
   say "You selected the following day(s): "
   do i = 101 to 107
      a = i-100
      if chk.i = 1 then say lst.a
   end
end
```

CheckList Function

OODialog provides a shortcut function to invoke a *CheckList* dialog as a function:

```
weekdays = .array~of("Monday","Tuesday","Wednesday", ,
                     "Thursday","Friday","Saturday","Sunday")
days = CheckList('Check the days','Working Days',weekdays)
if days ¬= .Nil then do i = 1 to days~items
                        if days[i] then say 'Working day =' weekdays[i]
                     end
```

The parameters are described in the *Init* method, but, instead of stems, arrays are passed into and returned from the function.

SingleSelection Class

The *SingleSelection* class shows a dialog that has a group of radio buttons. The user can select only one item of the group.

Requires:

OODPLAIN.CLS is required to use this class

Subclass:

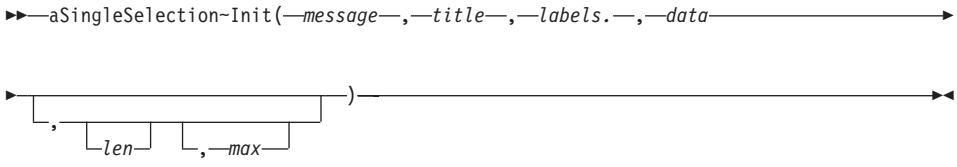
This class is a subclass of the "PlainUserDialog" class

Execute:

Returns the number of the radio button selected

The method listed below is defined by this class.

Init



Arguments:

The arguments are:

message
A text string that is displayed on top of the radio button group. Use it to give the user advice on what to do.

title A text string for the title bar

labels. This argument is a stem variable containing all labels for the radio buttons

data You can use this argument to preselect one radio button. A value of 1 selects the first radio button, a value of 2 selects the second, and so on.

len Determines the length of the check boxes and labels. If omitted, the size is calculated to fit the largest label.

max The maximum number of radio buttons in one column. If there are more radio buttons than *max* – that is, *labels.* has more items than the value of *max* – this method continues with a new column.

Example:

The following example creates and executes a dialog that contains a two-column radio button group. The fifth radio button (the button with the label *May*) is preselected.

```
mon.1 = "January" ; mon.2 = "February" ; mon.3 = "March"
mon.4 = "April"   ; mon.5 = "May"      ; mon.6 = "June"
mon.7 = "July"    ; mon.8 = "August"   ; mon.9 = "September"
mon.10= "October" ; mon.11= "November" ; mon.12= "December"

dlg = .SingleSelection~new("Please select a month!", ,
                           "Single Selection", mon., 5, , 6)
s = dlg~execute
say  "You Selected the month: " mon.s
```

SingleSelection Function

OODialog provides a shortcut function to invoke a *SingleSelection* dialog as a function:

```
months = .array~of("Jan","Feb","Mar","Apr","May","Jun", ,
                   "Jul","Aug","Sep","Oct","Nov","Dec")
m = SingleSelection('Check it','Born in',months,12,,6)
say "Born in month" m '=' months[m]
```

The parameters are described in the *Init* method, but instead of an input stem, an array is passed into the function.

Chapter 15. AnimatedButton Class

The *AnimatedButton* class provides the methods to implement an animated button within a dialog. The attributes and methods are only described briefly in this document. An example program, `oowalker.rex`, is provided with the OODialog sample programs.

ParentDlg
Attribute holding the handle of the parent dialog

Stopped
Animation ends when set to 1 (see *Stop* method)

Init Initialize the animation parameters:

```
but = .AnimatedButton~new(buttonid,from,to,,
                          movex,movey,sizex,sizey,delay,,
                          startx,starty,parentdialog)
```

The values are stored in a stem variable:

sprite.buttonid
ID of animation button

sprite.from
Array of in-memory bitmap handles, or a bitmap resource ID in a DLL, or the name of an array in the .local directory containing handles to bitmaps loaded with "LoadBitmap" on page 208. The array has to start with 1 and continue in increments by 1.

sprite.to
0 if *sprite.from* is an array, or the name of an array stored in `.local`, or a bitmap resource ID in a DLL

sprite.movex
Size of one move horizontally (pixels)

sprite.movey
Size of one move vertically

sprite.sizex
Horizontal size of all bitmaps (pixels)

sprite.sizey
Vertical size of all bitmaps

sprite.delay
Time delay between moves (ms)

Startx and *starty* are the initial bitmap position, and *parentdialog* is stored in the *ParentDlg* attribute.

Two more values are initialized in the stem variable:

sprite.smooth
Set to 1 for smooth edge change (can be changed to 0 for a bouncy edge change)

sprite.step
Set to 1 as the step size between *sprite.from* and *sprite.to* for bitmaps in a DLL

SetSprite
Set all the sprite. animation values using a stem:

```
mysprite.from = .array~of(bmp1,bmp2,...)
mysprite.to = 0
mysprite.movex = ...
...
self~setSprite(mysprite.)
```

GetSprite
Retrieve the animation values into a stem:

```
self~getSprite(mysprite.)
```

SetFromTo
Set bitmap information (*sprite.from* and *sprite.to*):

```
self~setFromTo(bmpfrom,bmpto)
```

SetMove
Set size of one move (*sprite.movex* and *sprite.movey*):

```
self~setMove(movex,movey)
```

SetDelay
Set delay between moves in milliseconds (*sprite.delay*):

```
self~setDelay(delay)
```

SetSmooth
Set smooth (1) or bouncy (0) edges (*sprite.smooth*):

```
self~setSmooth(smooth)  /* 1 or 0 */
```

SetStep
Set the step size (*sprite.step*) between *sprite.from* and *sprite.to* for bitmaps in a DLL, for example, if bitmap resources are numbered 202, 204, 206, etc:

```
self~setFromTo(202,210)
self~setStep(2)
```

Run Run the animation by going through all the bitmaps repetitively until dialog is stopped; invokes *MoveSeq*:

```
self~run
```

MoveSeq
Animate one sequence through all the bitmaps in the given move steps; invokes *MovePos*:

```
self~moveSeq
```

MovePos
Move the bitmaps by the arguments:

```
self~movePos(movex,movey)
```

MoveTo
Move the bitmaps in the predefined steps to the given position; invokes *MoveSeq*:

```
self~moveTo(posx,posy)
```

SetPos
Set the new starting position of the bitmaps:

```
self~setPos(newx,newy)
```

GetPos
Retrieve the current position into a stem:

```
self~getPos(pos.)
say 'pos=' pos.x pos.y
```

ParentStopped
Check the parent dialog window and return its finished attribute (1 means finished)

Stop Stop animation by setting the stopped attribute to 1

HitRight
Invoked by run when the bitmap hits the right edge (returns 1 and bitmap starts at left again; you can return 0 and set the new position yourself)

HitLeft
Invoked when the bitmap hits the left edge (default action is to start at right again)

HitBottom
Invoked when the bitmap hits the bottom edge (default action is to start at top again)

HitTop
Invoked when the bitmap hits the top edge (default action is to start at bottom again)

To use an animated button a dialog has to:

- Define a button in a resource file (owner-drawn)
- Load the bitmaps of the animation into memory using an array
- Initialize the animated button with the animation parameters
- Invoke the run method of the animated button
- Stop the animation and remove the bitmaps from memory

The dialog may also dynamically change the parameters (for example, the size of a move, or the speed) and override actions, such as hitting an edge.

See the `oowalker.rex` and `oowalk2.rex` examples in `OODIALOG\SAMPLES`.

For further information see "ConnectAnimatedButton" on page 172.

Chapter 16. MessageExtensions Class

To use the methods defined by the mixin class *MessageExtensions* you must inherit from this class by specifying the INHERIT option for the ::CLASS directive in the class declaration. For example:

```
::class MyExtendedDialog SUBCLASS UserDialog INHERIT MessageExtensions
```

Requires:

The *MessageExtensions* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

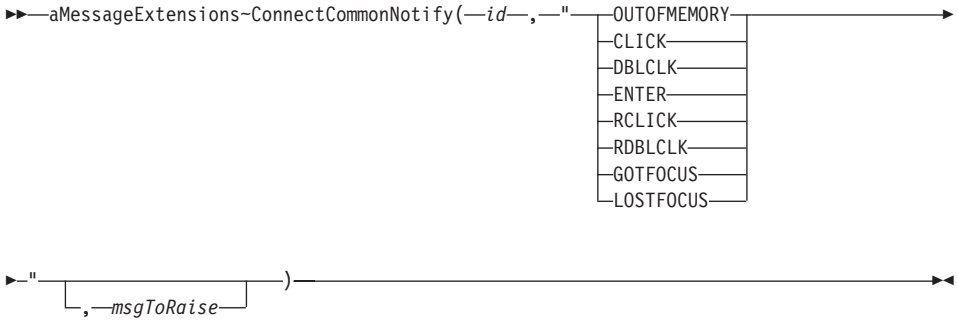
Methods:

Instances of the *MessageExtensions* class implement the methods listed in Table 5.

Table 5. MessageExtensions Instance Methods

Method...	...on page
ConnectButtonNotify	322
ConnectComboBoxNotify	328
ConnectCommonNotify	312
ConnectEditNotify	324
ConnectListBoxify	326
ConnectListNotify	318
ConnectScrollBarNotify	330
ConnectSliderNotify	330
ConnectTabNotify	332
ConnectTreeNotify	313
DefListDragHandler	321
DefTreeDragHandler	316

ConnectCommonNotify



The *ConnectCommonNotify* method connects a particular WM_NOTIFY message for a dialog control with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to a dialog control.

If a specific Connect...Notify method exists for the dialog control, use this method instead of the ConnectCommonNotify method because the system might send a specific notification instead of a common one.

Arguments:

The arguments are:

id The ID of the dialog control of which a notification is to be connected to a method.

event The event to be connected with a method:

OUTOFMEMORY
The dialog control went out of memory.

CLICK
The left mouse button was clicked on the dialog control.

DBLCLK
The left mouse button was double-clicked on the dialog control.

ENTER
The return key was pressed in the dialog item.

RCLICK
The right mouse button was clicked on the dialog item.

RDBLCLK
The right mouse button was double-clicked on the dialog control.

GOTFOCUS
The dialog item got the input focus.

LOSTFOCUS
The dialog item lost the input focus.

msgToRaise
The message that is to be sent whenever the specified notification is received. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

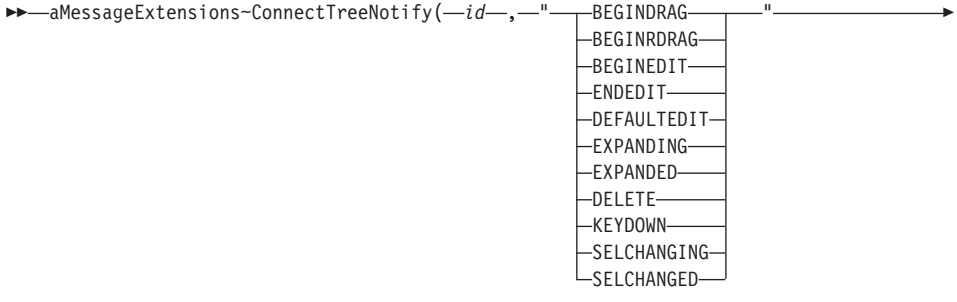
Return value:
This method does not return a value.

Example:
The following example connects the double-click of the left mouse button on dialog control DLGITEM1 with method OnDblClk:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectCommonNotify(DLGITEM1, "DBLCLK")

::method OnDblClk
  use arg id
  say "Item" id " has been double-clicked!"
```

Note: Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.

ConnectTreeNotify



The *ConnectTreeNotify* method connects a particular WM_NOTIFY message for a tree view control with a method. The WM_NOTIFY message informs the dialog that an event has occurred in the tree view.

Arguments:

The arguments are:

id The ID of the tree view control of which a notification is to be connected to a method.

event The event to be connected with a method:

BEGINDRAG
A drag-and-drop operaton was initiated. See "DefTreeDragHandler" on page 316 for information on how to implement a drag-and-drop handler.

BEGINRDRAG
A drag-and-drop operaton involving the right mouse button was initiated. See "DefTreeDragHandler" on page 316 for information on how to implement a drag-and-drop handler.

BEGINEDIT
Editing a label has been started.

ENDEDIT
Label editing has ended.

DEFAULTEDIT
This event connects the notification that label editing has been started and ended with a predefined event-handling method. This method extracts the newly entered text from the notification and modifies the item of which the label was edited. If this event is not connected you must provide your own event-handling method and connect it with the BEGINEDIT and ENDEDIT events. Otherwise, the edited text is lost and the item remains unchanged.

When you specify this event, omit the *msgToRaise* argument.

EXPANDING
An item is about to expand or collapse. This notification is sent before the item has expanded or collapsed.

EXPANDED
An item has expanded or collapsed. This notification is sent after the item expanded or collapsed.

DELETE
An item has been deleted.

KEYDOWN
A key was pressed inside the tree view. This notification is not sent while a label is being edited.

SELCHANGING
Another item is about to be selected. This notification is sent before the selection has changed.

SELCHANGED
Another item was selected. This notification is sent after the selection was changed.

msgToRaise
The message that is to be sent whenever the specified notification is received from the tree view control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example connects the selection-changed event for the tree view FileTree with method NewTreeSelection and displays the text of the new selection:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectTreeNotify("FileTree", "SELCHANGED", "NewTreeSelection")

::method NewTreeSelection
  tc = self~GetTreeControl("FileTree")
  info. = tc~ItemInfo(tc~Selected)
  say "New selection is:" info.!text
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.

2. The event-handling method connected to ENDEDIT receives two arguments: the item handle of which the label has been edited and the newly entered text. Example:

```
::method OnEndEdit
  use arg item, newText
```

3. The event-handling method connected to KEYDOWN receives two arguments: the control ID of the tree view control and the virtual key code pressed. Use the method "KeyName" on page 517 of the VirtualKeyCodes class to get the name of the key. Note that your class must inherit from the VirtualKeyCodes class to use the KeyName method. Example:

```
::method OnKeyDown
  use arg id, vkey
  say "Key" self~KeyName(vkey) "was pressed."
```

4. The event-handling method connected to EXPANDED or EXPANDING receives three arguments: the control ID of the tree view control, the tree item expanded or collapsed, and a string that indicates whether the item was expanded or collapsed. Example:

```
::method OnExpanding
  use arg id, item, what
  say "Item with handle" item "is going to be" what
```

5. The event-handling method connected to BEGINDRAG or BEGINRDRAG receives three arguments: the control ID of the tree view control, the tree item to be dragged, and the point where the mouse cursor was pressed (x and y positions, separated by a blank). Example:

```
::method OnBeginDrag
  use arg id, item, where
  say "Item with handle" item "is in drag-and-drop mode"
  parse var where x y
  say "The drag operation started at point ("x","y")"
```

DefTreeDragHandler

```
►►—aMessageExtensions~DefTreeDragHandler———————————————————————————►◄
```

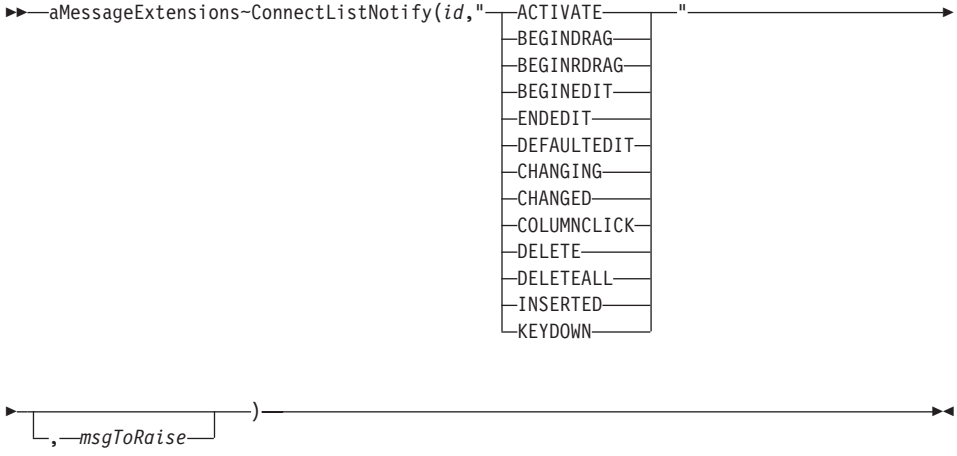
A tree view control cannot handle a drag-and-drop operation within the tree view. Therefore, you can connect the *DefTreeDragHandler* method with the BEGINDRAG or BEGINRDRAG notification message (see "ConnectTreeNotify" on page 313) to allow the moving of an item or a node with all its subitems from one parent node to another within a tree view. The cursor shape is changed to a crosshair during the drag operation. If the cursor is moved over the item dragged, the cursor shape is changed to a slashed circle. You can cancel the drag operation by clicking the other mouse button while holding the button that started the drag operation.

The *DefTreeDragHandler* is implemented as follows:

```
::method DefTreeDragHandler
   use arg id, item, pt
   tc = self~GetTreeControl(id)
   hc = tc~Cursor_Cross   /* change cursor and store current */
   parse value tc~GetRect with left top right bottom
   oldItem = 0
   nocurs = 0
   lmb = self~IsMouseButtonDown("LEFT")
   rmb = self~IsMouseButtonDown("RIGHT")
   call time "R"
   do while (lmb \= 0 | rmb \= 0) & \(lmb \= 0 & rmb \= 0)
     pos = self~CursorPos
     parse var pos x y
     parse value tc~ScreenToClient(x, y) with newx newy
      ht = tc~HitTest(newx, newy)
      if ht \= 0 & ht~wordpos("ONITEM") > 0 then do
          parse var ht newParent where
          /* check if droptarget is the current parent or one of the dragged
             item's children */
          if newParent \= Item & newParent \= tc~Parent(Item) & tc~IsAncestor,
             (Item,newParent) = 0
          then do
             is. = tc~ItemInfo(newParent)
             if is.!State~Wordpos("INDROP") = 0 then
             do
                call time "R"
                tc~DropHighlight(newParent)
                if nocurs \= 0 then do
                  tc~RestoreCursorShape(nocurs) /*restore old cursor (cross)*/
                  nocurs = 0
                end
             end
             else if time("E") > 1 then do  /* expand node after 1 second */
                 if is.!Children \= 0 & is.!State~Wordpos("EXPANDED") = 0 then
                 tc~expand(newParent)
             end
          end
          else do
             if nocurs = 0 then do
                nocurs = tc~Cursor_No  /* set no cursor and retrieve
                                          current cursor (cross) */
                tc~DropHighlight  /* remove drop highlight */
             end
          end
      end
      else do
          if newParent \= 0 then do
             /* necessary to redraw cursor when moving on a valid item again */
              tc~DropHighlight  /* remove drop highlight */
              newParent = 0
          end
          if nocurs = 0 then nocurs = tc~Cursor_No  /* set no cursor and
                                     retrieve current cursor (cross) */
          /* handle scrolling */
```

```
        fvItem = tc~FirstVisible
        if (ybottom) & (tc~NextVisible(fvItem) \= 0) then do
            tc~MakeFirstVisible(tc~NextVisible(fvItem))
            if y-bottom < 200 then call sleepms 200-(y-bottom)
        end
    end
    lmb = self~IsMouseButtonDown("LEFT")
    rmb = self~IsMouseButtonDown("RIGHT")
end
if ht~wordpos("ONITEM") > 0 & lmb = 0 & rmb = 0 then do /* if mouse on item
                                                          and both mouse buttons up */
     item = tc~MoveItem(Item, newParent, 1)  /* move item under newParent */
end
tc~DropHighlight(0)  /* remove drop highlight */
tc~select(item)      /* select item */
tc~EnsureVisible(item)
tc~RestoreCursorShape(hc)  /* restore old cursor */
pos = self~CursorPos
parse var pos x y
self~SetCursorPos(x+1, y+1)  /* move cursor to force redraw */
```

ConnectListNotify



The *ConnectListNotify* method connects a particular WM_NOTIFY message for a list view control with a method. The WM_NOTIFY message informs the dialog that an event has occurred in the list view.

Arguments:

The arguments are:

id The ID of the list view control of which a notification is to be connected to a method.

event The event to be connected with a method:

ACTIVATE
An item is activated by double-clicking the left mouse button.

BEGINDRAG
A drag-and-drop operaton was initiated. See "DefListDragHandler" on page 321 for information on how to implement a drag-and-drop handler.

BEGINRDRAG
A drag-and-drop operaton involving the right mouse button was initiated. See "DefListDragHandler" on page 321 for information on how to implement a drag-and-drop handler.

BEGINEDIT
Editing a label has been started.

ENDEDIT
Label editing has ended.

DEFAULTEDIT
This event connects the notification that label editing has been started and ended with a predefined event-handling method. This method extracts the newly entered text from the notification and modifies the item of which the label was edited. If this event is not connected you must provide your own event-handling method and connect it with the BEGINEDIT and ENDEDIT events. Otherwise, the edited text is lost and the item remains unchanged.

When you specify this event, omit the *msgToRaise* argument.

CHANGING
An item is about to change. This notification is sent before the item is changed.

CHANGED
An item has changed. This notification is sent after the item changed.

COLUMNCLICK
A column has been clicked.

DELETE
An item has been deleted.

DELETEALL
All items have been deleted.

INSERTED
A new item has been inserted.

KEYDOWN
A key was pressed inside the list view. This notification is not sent while a label is being edited.

msgToRaise
The message that is to be sent whenever the specified notification is received from the list view control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example connects the column-clicked event for the list view EMPLOYEES with method ColumnAction and changes the style of the list view from REPORT to SMALLICON:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectListNotify("EMPLOYEES", "COLUMNCLICK", "ColumnAction")

::method ColumnAction
  use arg id, column
  lc = self~GetListControl("EMPLOYEES")
  lc~ReplaceStyle("REPORT", "SMALLICON EDIT SINGLESEL ASCENDING")
  if column > 0 then ...
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling method connected to ENDEDIT receives two arguments: the item ID of which the label has been edited and the newly entered text. Example:

   ```
   ::method OnEndEdit
     use arg item, newText
   ```

3. The event-handling method connected to COLUMNCLICK receives two arguments: the control ID of the list view control and the zero-based column number of which the header button was pressed. Example:

   ```
   ::method OnColumnClick
     use arg id, column
   ```

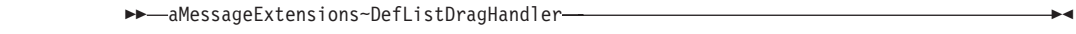
4. The event-handling method connected to KEYDOWN receives two arguments: the control ID of the list view control and the virtual key code pressed. Use the method "KeyName" on page 517 of the VirtualKeyCodes class to get the name of the key. Note that your class must inherit from the VirtualKeyCodes class to use the KeyName method. Example:

```
::method OnKeyDown
  use arg id, vkey
  say "Key" self~KeyName(vkey) "was pressed."
```

5. The event-handling method connected to BEGINDRAG or BEGINRDRAG receives three arguments: the control ID of the list view control, the index of the list item to be dragged, and the point where the mouse cursor was pressed (x and y positions, separated by a blank). Example:

```
::method OnBeginDrag
  use arg id, item, where
  say "Item at index" item "is in drag-and-drop mode"
  parse var where x y
  say "The drag operation started at point ("x","y")"
```

DefListDragHandler



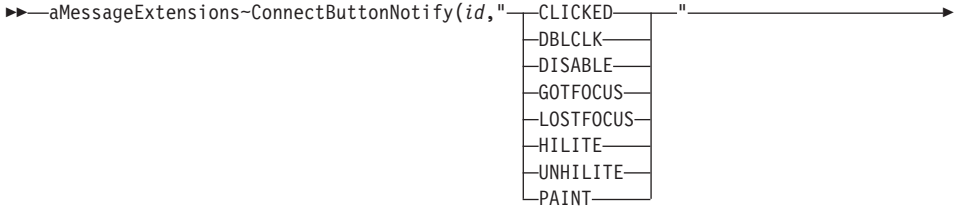
A list view control cannot handle a drag-and-drop operation within the list view. Therefore, you can connect the *DefListDragHandler* method with the BEGINDRAG or BEGINRDRAG notification message (see "ConnectListNotify" on page 318) to allow the dragging of an item from one location to another within an icon view and a smallicon view. The cursor shape is changed to a crosshair during the drag operation. You can cancel the drag operation by clicking the other mouse button while holding the button that started the drag operation. Note that the item position is not flexible when the list view control has the AUTOARRANGE style.

The *DefListDragHandler* is implemented as follows:

```
::method DefListDragHandler
   use arg id, item, pt
   lc = self~GetListControl(id)
   hc = lc~Cursor_Cross   /* change cursor and store current */
   parse value lc~GetRect with left top right bottom
   parse var pt oldx oldy
   origin = lc~ItemPos(item)
   lmb = self~IsMouseButtonDown("LEFT")
   rmb = self~IsMouseButtonDown("RIGHT")
   do while (lmb \= 0 | rmb \= 0) & \(lmb \= 0 & rmb \= 0)
     pos = self~CursorPos
     parse var pos x y
     parse value lc~ScreenToClient(x, y) with newx newy
```

```
  hs = lc~HScrollPos; vs = lc~VScrollPos
  sx = x-right
  sy = y-bottom
   in_rx = (sx <= 30) & (newx >= -30)
   in_ry = (sy <= 30) & (newy >= -30)
   if (in_rx & in_ry) then do    /* is the mouse cursor inside the drag
                                                        rectangle */
       if xright then sx = sx + 30; else sx = 0
       if ybottom then sy = sy + 30; else sy = 0
       newx = newx+hs;  newy = newy +vs;
       if newx < 0 then newx = 0
       if newy < 0 then newy = 0
       if (in_rx & oldx \= newx) | (in_ry & oldy \= newy) then do
        lc~SetItemPos(item, newx, newy)
          oldx = newx
          oldy = newy
          if sx \= 0 | sy \= 0 then do
             lc~Scroll(sx, sy)
             call sleepms 30
          end
     end
   end
   else do    /* no, so force the mouse cursor back inside the rectangle */
      if newx < -30 then newx = -30
      if sx > 30 then newx = (right-left) + 28
      if newy < -30 then newy = -30
      if sy > 30 then newy = (bottom-top) + 28
      parse value lc~ClientToSCreen(newx, newy) with x y
      self~SetCursorPos(x, y)
   end
   lmb = self~IsMouseButtonDown("LEFT")
   rmb = self~IsMouseButtonDown("RIGHT")
end
if (lmb \= 0 & rmb \= 0) then do  /* if both buttons pressed restore
                                     original pos */
   parse var origin x y
   lc~SetItemPos(item, x, y)
end
lc~RestoreCursorShape(hc)  /* restore old cursor */
pos = self~CursorPos
parse var pos x y
self~SetCursorPos(x+1, y+1)  /* move cursor to force redraw */
```

ConnectButtonNotify



The *ConnectButtonNotify* method connects a particular WM_NOTIFY message for a button control (push button, radio button, or check box) with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to the button.

Arguments:

The arguments are:

id The ID of the button control of which a notification is to be connected to a method.

event The event to be connected with a method:

CLICKED
The button has been clicked.

DBLCLK
The button has been double-clicked.

DISABLE
The button has been disabled.

GOTFOCUS
The button got the input focus.

LOSTFOCUS
The button lost the input focus.

HILITE
The button has been selected.

UNHILITE
The highlighting is to be removed (lost selection).

PAINT
The button is to be repainted. This notification is only sent for owner-drawn buttons.

msgToRaise
The message that is to be sent whenever the specified notification is received from the button control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:

This method does not return a value.

Example:

The following example displays a message whenever the OK button is selected:

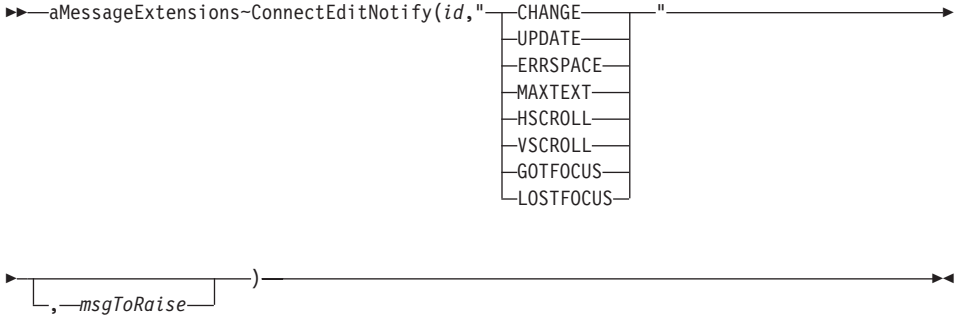
```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectButtonNotify("OK", "HILITE")

::method OnHilite
  say "The OK button has been selected"
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling methods receive two arguments: the ID of the button (extract the low-order word) and the handle to the button. Example:

```
::method Handler
  use arg ev_id, handle
  id = BinaryAnd(ev_id, "0x0000FFFF")
  ...
```

ConnectEditNotify



The *ConnectEditNotify* method connects a particular WM_NOTIFY message for an edit control with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to the edit control.

Arguments:

The arguments are:

id The ID of the edit control of which a notification is to be connected to a method.

event The event to be connected with a method:

CHANGE
The text has been altered. This notification is sent after the screen has been updated.

UPDATE
The text has been altered. This notification is sent before the screen is updated.

ERRSPACE
An out-of-memory problem has occurred.

MAXTEXT
The text inserted exceeds the specified number of characters for the edit control. This notification is also sent when:

- An edit control does not have the ES_AUTOHSCROLL or AUTOSCROLLH style and the number of characters to be inserted would exceed the width of the edit control.
- The ES_AUTOVSCROLL or AUTOSCROLLV style is not set and the total number of lines resulting from a text insertion would exceed the height of the edit control.

HSCROLL
The horizontal scroll bar has been used.

VSCROLL
The vertical scroll bar has been used.

GOTFOCUS
The edit control got the input focus.

LOSTFOCUS
The edit control lost the input focus.

msgToRaise
The message that is to be sent whenever the specified notification is received from the edit control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example verifies the input of entry line AMOUNT and resets it to 0 when a nonnumeric value was entered:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
```

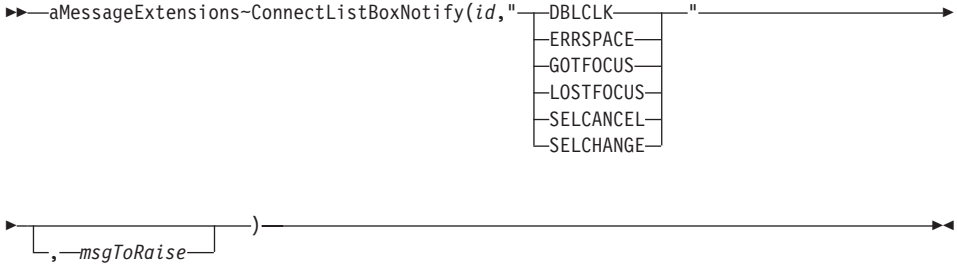
```
    self~init:super(...)
    self~ConnectEditNotify("AMOUNT", "CHANGE")

  ::method OnChange
    ec = self~GetEditControl("AMOUNT")
    if ec~GetText~Space(0) \= "" & ec~GetText~DataType("N") = 0 then do
      ec~SetModified(0)
      ec~Select
      ec~ReplaceSelText("0")
     end
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling methods receive two arguments: the ID of the edit control (extract the low-order word) and the handle to the edit control. Example:

```
::method Handler
  use arg ev_id, handle
  id = BinaryAnd(ev_id, "0x0000FFFF")
  ...
```

ConnectListBoxNotify



The *ConnectListBoxNotify* method connects a particular WM_NOTIFY message for a list box with a method. The WM_NOTIFY message informs the dialog that an event has occurred in the list box.

Arguments:

The arguments are:

id The ID of the list box of which a notification is to be connected to a method.

event The event to be connected with a method:

DBLCLK
An entry in the list box has been selected with a double click.

ERRSPACE
An out-of-memory problem has occurred.

GOTFOCUS
The list box got the input focus.

LOSTFOCUS
The list box lost the input focus.

SELCANCEL
The selection in the list box has been canceled.

SELCHANGE
Another list box entry has been selected.

msgToRaise
The message that is to be sent whenever the specified notification is received from the list box. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example displays the text of the selected list box entry:

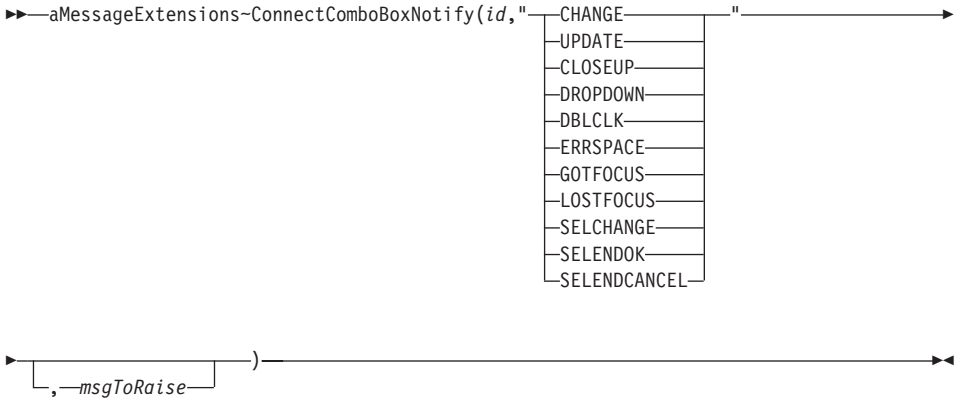
```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectListBoxNotify("MYLIST", "SELCHANGE", "SelectionChanged")

::method SelectionChanged
  li = self~GetListBox("MYLIST")
  say "New selection is:" li~Selected
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling methods receive two arguments: the ID of the list box (extract the low-order word) and the handle to the list box. Example:

```
::method Handler
  use arg ev_id, handle
  id = BinaryAnd(ev_id, "0x0000FFFF")
  ...
```

ConnectComboBoxNotify



The *ConnectComboBoxNotify* method connects a particular WM_NOTIFY message for a combo box with a method. The WM_NOTIFY message informs the dialog that an event has occurred in the combo box.

Arguments:

The arguments are:

id The ID of the combo box of which a notification is to be connected to a method.

event The event to be connected with a method:

CHANGE
The text in the edit control has been altered. This notification is sent after Windows updated the screen.

UPDATE
The text in the edit control has been altered. This notification is sent before Windows updates the screen.

CLOSEUP
The list of the combo box has been closed.

DROPDOWN
The list of the combo box is about to be made visible.

DBLCLK
An entry in the combo box list has been selected with a double click.

ERRSPACE
An out-of-memory problem has occurred.

GOTFOCUS
The combo box got the input focus.

LOSTFOCUS
The combo box lost the input focus.

SELCHANGE
Another entry in the combo box list has been selected.

SELENDOK
The list was closed after another entry was selected.

SELENDCANCEL
After the selection of another entry, another control or dialog was selected, which canceled the selection of the entry.

msgToRaise
The message that is to be sent whenever the specified notification is received from the combo control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example invokes method PlaySong whenever the list of the combo box PROFESSIONS is about to be made visible:

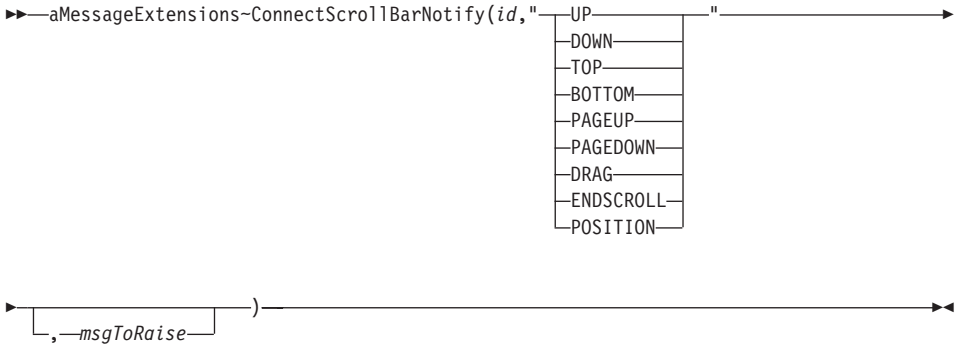
```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method InitDialog
  self~init:super(...)
  self~ConnectComboBoxNotify("PROFESSIONS", "DROPDOWN", "PlaySong")
```

Notes:

1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling methods receive two arguments: the ID of the combo box (extract the low-order word) and the handle to the combo box. Example:

```
::method Handler
  use arg ev_id, handle
  id = BinaryAnd(ev_id, "0x0000FFFF")
  ...
```

ConnectScrollBarNotify



The *ConnectScrollBarNotify* method connects a particular WM_NOTIFY message for a scroll bar with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to the scroll bar.

Arguments:

The arguments are:

id The ID of the scroll bar of which a notification is to be connected to a method.

event The event to be connected with a method:

UP The scroll bar was scrolled to the left or up by one unit.

DOWN
The scroll bar was scrolled to the right or down by one unit.

TOP The scroll bar was scrolled to the upper left.

BOTTOM
The scroll bar was scrolled to the lower right.

PAGEUP
The scroll bar was scrolled to the left or up by one page size.

PAGEDOWN
The scroll bar was scrolled to the right or down by one page size.

DRAG
The scroll bar has been dragged.

ENDSCROLL
Scrolling has been ended, that is, the appropriate key or mouse button has been released.

POSITION
The scroll bar was scrolled to an absolute position (the left mouse button has been released).

msgToRaise
The message that is to be sent whenever the specified notification is received from the scroll bar. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:
This method does not return a value.

Example:
The following example connects the POSITION event with method OnPosition, which extracts the new position from the notification arguments and stores it for the scroll bar. It also displays the new position and the event type for POSITION, which is to be 4:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method InitDialog
  self~InitDialog:super(...)
  self~ConnectScrollBarNotify("MYSCROLL", "POSITION")

::method OnPosition
  use arg ev_pos, hnd
  pos = ev_pos % "10000"~x2d
  self~GetScrollBar("MYSCROLL")~SetPos(pos,1)   /* redraw scroll bar
                                                   with new position */
  say "Pos=" pos ", event code to verify=" BinaryAnd(ev_pos,,
    "0x0000FFFF") "(expected 4)"
```

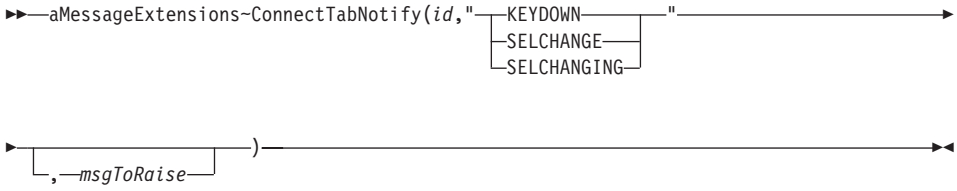
Notes:

1. The method can only be called after the scroll bar was created by Windows. A good location for this connection is the InitDialog method.
2. The event-handling methods receive two arguments: an event-position pair and the handle to the scroll bar. You can retrieve the scroll bar position by extracting the high-order word. Example:

   ```
   ::method Handler
     use arg ev_pos, handle
     position = ev_pos % "10000"~x2d
   ```

   If the user changed the scroll bar position, you must set the scroll bar position with “SetPos” on page 454 to keep the selected position.

ConnectTabNotify



The *ConnectTabNotify* method connects a particular WM_NOTIFY message for a tab control with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to the tab control.

Arguments:

The arguments are:

id The ID of the tab control of which a notification is to be connected to a method.

event The event to be connected with a method:

KEYDOWN
A key has been pressed while the tab control was focused.

SELCHANGE
Another tab has been selected in the tab control. This method is called after the selection was changed.

SELCHANGING
Another tab has been selected in the tab control. This method is called before the selection is changed.

msgToRaise
The message that is to be sent whenever the specified notification is received from the tab control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:

This method does not return a value.

Example:

The following example invokes method OnSelChange whenever another tab is selected in the tab control PAGE:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method Init
  self~init:super(...)
  self~ConnectTabNotify("PAGE", "SELCHANGE")
```

Notes:

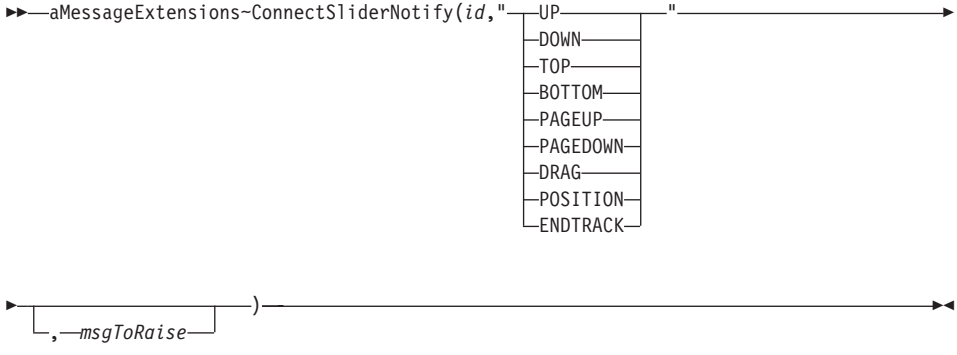
1. Connections are usually placed in the Init or InitDialog method. If both methods are defined, use *init* as the place for this connection – but not before *init:super* has been called.
2. The event-handling method that is connected to KEYDOWN receives two arguments: the control ID of the tab control and the virtual key code that has been pressed. Use the method "KeyName" on page 517 of the VirtualKeyCodes class to get the name of the key. Note that your class must inherit from the VirtualKeyCodes class to use the KeyName method. Example:

   ```
   ::method OnKeyDown
     use arg id, vkey
     say "Key" self~KeyName(vkey) "was pressed."
   ```

3. All other event-handling methods receive two arguments: the ID of the tab control (extract the low-order word) and the handle to the tab control. Example:

   ```
   ::method Handler
     use arg ev_id, handle
     id = BinaryAnd(ev_id, "0x0000FFFF")
   ```

ConnectSliderNotify



The *ConnectSliderNotify* method connects a particular WM_NOTIFY message for a slider control, which is also called a track bar, with a method. The WM_NOTIFY message informs the dialog that an event has occurred with regard to the slider control.

Arguments:

The arguments are:

id The ID of the slider control of which a notification is to be connected to a method.

event The event to be connected with a method:

UP The Up or right key has been pressed.

DOWN
The Down or left key has been pressed.

TOP The Home key has been pressed.

BOTTOM
The End key has been pressed.

PAGEUP
The PgUp key has been pressed.

PAGEDOWN
The PgDn key has been pressed.

DRAG
The slider has been moved.

POSITION
The left mouse button has been released, following a DRAG notification.

ENDTRACK
The slider movement is completed, that is, the appropriate key or mouse button has been released.

msgToRaise
The message that is to be sent whenever the specified notification is received from the slider control. Provide a method with a matching name. If you omit this argument, the event is preceded by `On`.

Return value:

This method does not return a value.

Example:

The following example connects the POSITION event (release mouse button after dragging) with method PosSet, which extracts the new slider position from the notification arguments and displays it together with the event type for POSITION, which is to be 4:

```
::class MyDlgClass subclass UserDialog inherit MessageExtensions

::method InitDialog
  self~InitDialog:super(...)
```

```
  self~ConnectSliderNotify("MYSLIDER", "POSITION", PosSet)

::method PosSet
  use arg ev_pos, hnd
  pos = ev_pos % "10000"~x2d
  say "Pos=" pos ", event code to verify=" BinaryAnd(ev_pos,,
    "0x0000FFFF") "(expected 4)"
```

Notes:

1. The method can only be called after the slider was created by Windows. A good location for this connection is the InitDialog method.
2. The event-handling methods receive two arguments: an event-position pair and the handle to the slider control. For some events, you can retrieve the slider position by extracting the high-oder word. Example:

```
::method Handler
  use arg ev_pos, handle
  position = ev_pos % "10000"~x2d
```

Chapter 17. AdvancedControls Class

The *AdvancedControls* class provides methods to add and use the new Win32 controls tree view control, list view control, tab control, slider control, and progress bar. It also provides methods to retrieve a specific object for any dialog control.

To use the methods defined by this mixin class, you must inherit from this class by specifying the INHERIT option for the ::CLASS directive in the class declaration. For example:

```
::class NewWin32Dialog SUBCLASS UserDialog INHERIT AdvancedControls
```

Requires:

The *AdvancedControls* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *AdvancedControls* class implement the methods listed in Table 6.

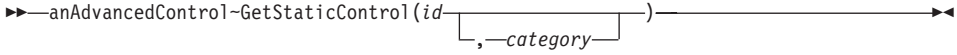
Table 6. AdvancedControls Instance Methods

Method...	...on page
AddListControl	354
AddProgressBar	358
AddSliderControl	359
AddTabControl	361
AddTreeControl	352
ConnectListControl	351
ConnectSliderControl	351
ConnectTreeControl	350
GetButtonControl	340
GetCheckControl	342
GetComboBox	343
GetEditControl	339
GetListBox	342
GetListControl	346
GetProgressBar	347

Table 6. AdvancedControls Instance Methods (continued)

Method...	...on page
GetRadioControl	341
GetScrollBar	345
GetSliderControl	348
GetStaticControl	338
GetTabControl	349
GetTreeControl	345

GetStaticControl



The *GetStaticControl* method returns an object of the StaticControl class that is assigned to the static dialog item with the specified ID. The StaticControl class provides methods to query and manipulate static dialog items like static text, group boxes, or frames. The static controls must have a positive ID.

Arguments:

The arguments are:

id The ID of the static dialog item.

category
The number of the category dialog page containing the requested dialog item. This argument must only be specified for category dialogs.

Return value:

An object of the StaticControl class or .Nil if the requested dialog item does not exist.

Example:

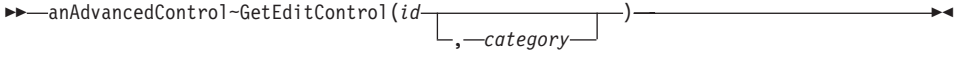
The following example requests an object of dialog item ITEM7 and, if the dialog item exists, resizes it, changes the displayed text, and sets another background and foreground color:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method ReArrange
  di = self~GetStaticControl("ITEM7")
  if di == .Nil then return
  di~Resize(100, 25, "HIDE")
```

```
di~Title="Processing layout update!"
di~SetColor(7,4)
di~Show
...
```

Note: GetStaticControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetEditControl



The *GetEditControl* method returns an object of the EditControl class that is assigned to the entry line with the specified ID. The EditControl class (see page 367) provides methods to query and manipulate edit controls.

Arguments:

The arguments are:

id The ID of the edit control.

category

The number of the category dialog page containing the requested edit control. This argument must only be specified for category dialogs.

Return value:

An object of the *StaticControl* class or .Nil if the requested edit control does not exist.

Example:

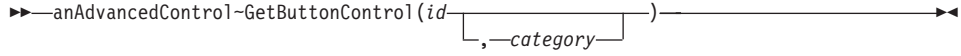
The following example gets an object of the *EditControl* class and checks whether the NAME entry line is empty. If the name field is empty, the dialog is not valid.

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method Validate
  di = self~GetEditControl("NAME")
  if di == .Nil then return 0
  if di~Title~space(0) \="" then return 1
```

Note: GetEditControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this

method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetButtonControl



The *GetButtonControl* method returns an object of the ButtonControl class that is assigned to the push button with the specified ID. The ButtonControl class (see page 415) provides methods to query and manipulate push buttons.

Arguments:

The arguments are:

id The ID of the push button.

category

The number of the category dialog page containing the requested push button. This argument must only be specified for category dialogs.

Return value:

An object of the ButtonControl class or .Nil if the requested push button does not exist.

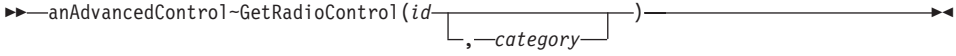
Example:

The following example displays the current state of the OK button by retrieving an object of the Button class and calling the State method:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method CurrentState
  di = self~GetButtonControl(1)
  if di == .Nil then return 0
  say "State is" di~State
```

Note: GetButtonControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetRadioControl



The *GetRadioControl* method returns an object of the RadioButton class that is assigned to the radio button with the specified ID. The RadioButton class (see page 427) provides methods to query and manipulate radio buttons.

Arguments:

The arguments are:

id The ID of the radio button.

category

The number of the category dialog page containing the requested radio button. This argument must only be specified for category dialogs.

Return value:

An object of the RadioButton class or .Nil if the requested radio button does not exist.

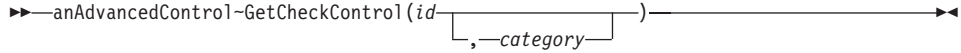
Example:

The following example displays a message when radio button CHOICE1 is selected:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method CurrentState
  di = self~GetRadioControl("CHOICE1")
  if di == .Nil then return 0
  id di~IsChecked = "CHECKED" then say "The radio button is selected!"
```

Note: GetRadioControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetCheckControl



The *GetCheckControl* method returns an object of the CheckBox class that is assigned to the check box with the specified ID. The CheckBox class (see page 429) provides methods to query and manipulate check boxes.

Arguments:

The arguments are:

id The ID of the check box.

category

The number of the category dialog page containing the requested check box. This argument must only be specified for category dialogs.

Return value:

An object of the CheckBox class or .Nil if the requested check box does not exist.

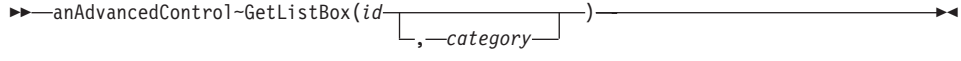
Example:

The following example displays a message when check box CHOICE1 is checked:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method CurrentState
  di = self~GetCheckControl("CHOICE1")
  if di == .Nil then return 0
  if di~IsChecked = "CHECKED" then say "The check box is checked!"
```

Note: GetCheckControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetListBox



The *GetListBox* method returns an object of the ListBox class that is assigned to the list box with the specified ID. The ListBox class (see page 431) provides methods to query and manipulate list boxes.

Arguments:

The arguments are:

id The ID of the list box.

category
The number of the category dialog page containing the requested list box. This argument must only be specified for category dialogs.

Return value:

An object of the ListBox class or .Nil if the requested list box does not exist.

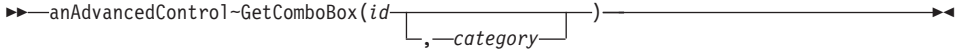
Example:

The following example removes all entries of list box AREAS and adds several new entries. Entry City will be preselected. Object "di" is connected to list box AREAS.

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method UpdateList
  di = self~GetListBox("AREAS")
  if di == .Nil then return 0
  di~DeleteAll
  di~Add("Town")
  di~Add("City")
  di~Add("Green")
  di~Add("Forest")
  di~Select("City")
```

Note: GetListBox connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetComboBox



The *GetComboBox* method returns an object of the ComboBox class that is assigned to the list box with the specified ID. The ComboBox class (see page 445) provides methods to query and manipulate combo boxes.

Arguments:

The arguments are:

id The ID of the combo box.

category
The number of the category dialog page containing the requested combo box. This argument must only be specified for category dialogs.

Return value:

An object of the ComboBox class or .Nil if the requested combo box does not exist.

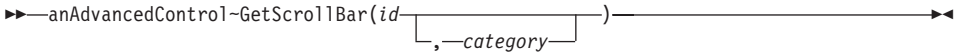
Example:

The following example removes all entries of combo box AREAS and adds several new entries. Entry "City" will be preselected. Object "di" is connected to combo box AREAS.

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method UpdateList
  di = self~GetComboBox("AREAS")
  if di == .Nil then return 0
  di~DeleteAll
  di~Add("Town")
  di~Add("City")
  di~Add("Green")
  di~Add("Forest")
  di~Select("City")
```

Note: GetComboBox connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of “StartIt” on page 260). For more information on this issue, refer to “Summary of User Dialog Processing” on page 60.

GetScrollBar



The *GetScrollBar* method returns an object of the ScrollBar class that is assigned to the scroll bar with the specified ID. The ScrollBar class (see page 453) provides methods to query and manipulate scroll bars.

Arguments:

The arguments are:

id The ID of the scroll bar.

category

The number of the category dialog page containing the requested scroll bar. This argument must only be specified for category dialogs.

Return value:

An object of the ScrollBar class or .Nil if the requested scroll bar does not exist.

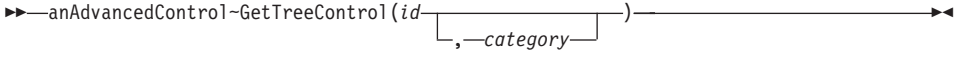
Example:

The following example sets a new range and a new position for scroll bar HORSB. Object "di" is connected to scroll bar HORSB.

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method FocusPage
  di = self~GetScrollBar("HORSB")
  if di == .Nil then return 0
  di~SetRange(0, 1000, 0)
  di~SetPos(500, 1)
```

Note: GetScrollBar connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of “StartIt” on page 260). For more information on this issue, refer to “Summary of User Dialog Processing” on page 60.

GetTreeControl



The *GetTreeControl* method returns an object of the TreeControl class that is assigned to the tree view with the specified ID. The TreeControl class (see page 491) provides methods to query and manipulate tree views.

Arguments:

The arguments are:

id The ID of the tree view.

category

The number of the category dialog page containing the requested tree view. This argument must only be specified for category dialogs.

Return value:

An object of the TreeControl class or .Nil if the requested tree view does not exist.

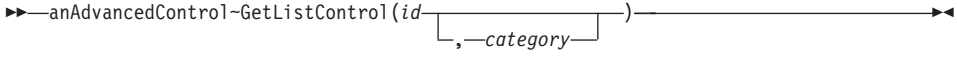
Example:

The following example initializes tree view 101 by sending message ADD to object "tc", which is assigned to 101 by using GetTreeControl:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method InitDialog
  tc = self~GetTreeControl(101)
  if tc == .Nil then return
  tc~Add("Root 1")
  tc~Add(  ,"Item 1")
  tc~Add(  ,"Item 2")
  tc~Add(  ,"Item 3")
  tc~Add("Root 2",,,"EXPANDED")
  tc~Add(  ,"Item 4",,,"BOLD")
  tc~Add(  ,"Item 5")
  tc~Add(  ,"Subroot")
  tc~Add(  ,    ,"Item 6",3)
```

Note: GetTreeControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetListControl



The *GetListControl* method returns an object of the ListControl class that is assigned to the list view with the specified ID. The ListControl class (see page 379) provides methods to query and manipulate list views.

Arguments:

The arguments are:

id The ID of the list view.

category
The number of the category dialog page containing the requested list view. This argument must only be specified for category dialogs.

Return value:

An object of the ListControl class or .Nil if the requested list view does not exist.

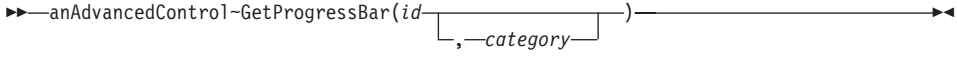
Example:

The following example initializes list view 101 by sending message ADD to object lc, which is assigned to 101 by using GetListControl:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method InitDialog
  lc = self~GetListControl(101)
  if lc == .Nil then return
  lc~~Add(101222)~~Add(,"Smith")~~Add(,,"John")
  lc~~Add(101223)~~Add(,"Michael")~~Add(,,"Carl")
```

Note: GetListControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetProgressBar



The *GetProgressBar* method returns an object of the ProgressBar Control class that is assigned to the progress bar with the specified ID. The ProgressBarControl class (see page 461) provides methods to query and manipulate progress bars.

Arguments:

The arguments are:

id The ID of the progress bar.

category
The number of the category dialog page containing the requested progress bar. This argument must only be specified for category dialogs.

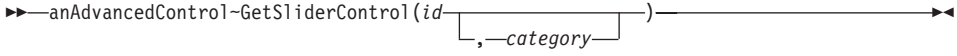
Return value:
An object of the ProgressBar class or .Nil if the requested progress bar does not exist.

Example:
The following example initializes and modifies progress bar PROGRESS by sending messages to the object that is returned by GetProgressBar:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method InitDialog
  pb = self~GetProgressBar("PROGRESS")
  if pb == .Nil then return
  pb~setstep(50)
  pb~setrange(,500)

::method UpdateProgress
  use arg amount
  self~GetProgressBar("PROGRESS")~SetPos(amount)
```

Note: GetProgressBar connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetSliderControl



The *GetSliderControl* method returns an object of the SliderControl class that is assigned to the track bar with the specified ID. The SliderControl class (see page 465) provides methods to query and manipulate track bars.

Arguments:
The arguments are:

id The ID of the track bar.

category
The number of the category dialog page containing the requested track bar. This argument must only be specified for category dialogs.

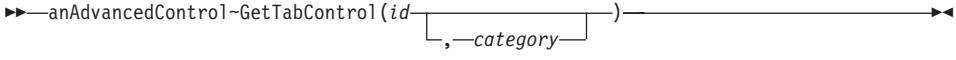
Return value:
An object of the SliderControl class or .Nil if the requested track bar does not exist.

Example:
The following example initializes track bar 103:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method InitTheSlider
  sl = self~GetSliderControl(103)
  if sl == .Nil then return
  no = 0; yes = 1
  sl~ClearSelRange(no)
  sl~SetMax(200,no)
  sl~SetTickFrequency(50)
  sl~SetTickAt(75)
  sl~SetSelStart(20, no)
  sl~SetSelEnd(180, yes)
  sl~Pos = 167
```

Note: GetSliderControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

GetTabControl



The *GetTabControl* method returns an object of the TabControl class that is assigned to the tab control with the specified ID. The TabControl class (see page 479) provides methods to query and manipulate tab controls.

Arguments:
The arguments are:

id The ID of the tab control.

category

The number of the category dialog page containing the requested tab control. This argument must only be specified for category dialogs.

Return value:

An object of the TabControl class or .Nil if the requested tab control does not exist.

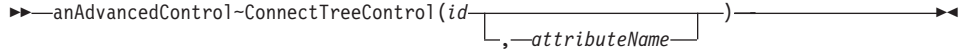
Example:

The following example initializes tab control PAGES to have five tabs:

```
::class MyDlgClass subclass UserDialog inherit AdvancedControls

::method InitDialog
  self~GetTabControl("PAGES")~AddSequence("Design","Implementation",,
  "Test","Review","Release")
```

Note: GetTabControl connects an Object REXX object with a Windows object. If the object does not exist, the NIL object is returned. Therefore, this method can only be applied after the Windows dialog has been created (after the invocation of "StartIt" on page 260). For more information on this issue, refer to "Summary of User Dialog Processing" on page 60.

ConnectTreeControl



The *ConnectTreeControl* method creates a new attribute and connects it to the tree view *id*. The attribute has to be synchronized manually with the tree view. You can do this globally using the SetData and GetData methods (see page 125) or methods provided by the TreeControl class. A tree view can contain many items. When the dialog data is set, the first tree view item containing the same text as the text stored in the connected attribute, is selected. When the data is received, the attribute receives the text of the selected tree view item. Usually, the connection is made automatically and you do not have to use this method.

Arguments:

The arguments are:

id The ID of the tree view that you want to connect.

attributeName

An unused valid REXX symbol because an attribute with exactly this name is added to the dialog object by this

method. Blank spaces, ampersands (&), and colons (:) are removed from the *attributeName*.

If this argument is omitted, is not valid, or already exists, the following occurs:

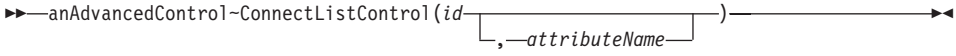
- If the ID is numeric, an attribute with the name *DATAid* is used, where *id* is the value of the first argument.
- If the ID is symbolic, the attribute is named as the ID.

Example:

In the following example, the tree view with ID 202 is associated with the attribute FileName. Then TEST.REX is assigned to the newly created attribute. Then the dialog is executed, which preselects TEST.REX in the tree view, if it exits. After the dialog is terminated, the selected entry of the tree view is copied to the attribute FileName.

```
MyDialog~ConnectTreeControl(202, "FileName")
MyDialog~FileName="TEST.REX"
MyDialog~Execute("SHOWTOP")
say MyDialog~FileName
```

ConnectListControl



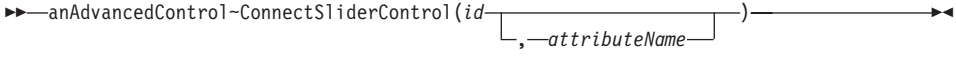
The *ConnectListControl* method creates a new attribute and connects it to the list view *id*. The *attributeName* is a string containing the numbers of the selected lines. The numbers are separated by blanks. Therefore, if value of the attribute after GetData is ″3 5 6″, the third, fifth, and sixth items are currently selected, or will be selected when SetData is executed. For further information, refer to “ConnectTreeControl” on page 350.

Example:

In the following example, the list view with ID 202 is associated with the attribute Customers. The first, 14th, and 29th entries in the list are preselected.

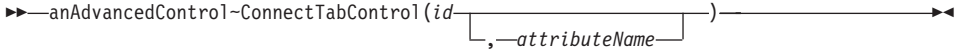
```
MyDialog~ConnectListControl(202, "Customers")
MyDialog~Customers="1 14 29"
```

ConnectSliderControl



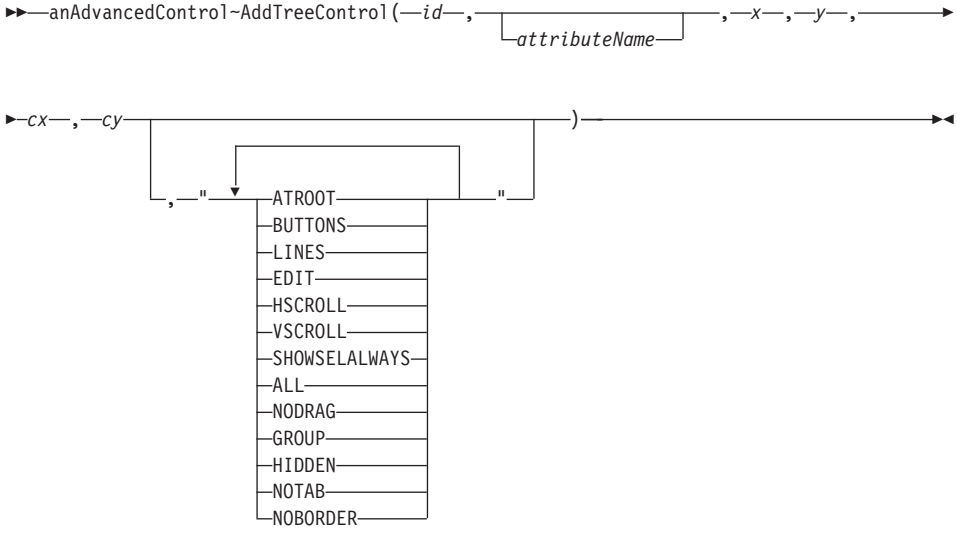
The *ConnectSliderControl* method creates a new attribute and connects it to the track bar *id*. The *attributeName* is the numerical position of the slider. For further information, refer to "ConnectTreeControl" on page 350.

ConnectTabControl



The *ConnectTabControl* method creates a new attribute and connects it to the tab control *id*. The *attributeName* is the text of the active tab. For further information, refer to "ConnectTreeControl" on page 350.

AddTreeControl



The *AddTreeControl* method adds a tree view to the dialog and connects it with a data attribute. For further information on tree view controls, refer to "Chapter 31. TreeControl Class" on page 491.

Arguments:

The arguments are:

id A unique identifier assigned to the control. You need the ID to refer to this control in other methods.

attributeName
The name of the data attribute associated with the dialog item. See page 350 to get information on what happens when this argument is omitted.

x, y The position of the upper left corner of the control relative to the dialog, in dialog units.

cx, xy The width and height of the dialog item, in dialog units.

options
This argument determines the behavior and style of the dialog item and can be one or more of the following, separated by blanks:

ATROOT
The tree view has lines linking child items to the root of the hierarchy.

BUTTONS
The tree view adds a button to the left of each parent item.

LINES
The tree view has lines linking child items to their corresponding parent items.

EDIT The tree view allows the user to edit the labels of tree view items. To store the edited text, you must connect a method to the ENDEDIT notification or connect the DEFAULTEDIT event handler (see "ConnectTreeNotify" on page 313).

HSCROLL
The tree view supports a horizontal scroll bar.

VSCROLL
The tree view supports a vertical scroll bar.

SHOWSELALWAYS
A selected item remains selected when the tree view loses focus.

ALL The options ATROOT, BUTTONS, LINES, EDIT, HSCROLL, and SHOWSELALWAYS are all applied.

NODRAG
The tree view is prevented from sending "begin drag" notifications.

GROUP
The first control of a group of controls in which the user can move from one control to the next with the arrow key.

HIDDEN
The control is initially hidden.

NOTAB
The tab key cannot be used to move to this control.

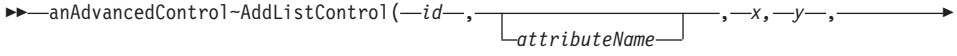
NOBORDER
No border is drawn around the control.

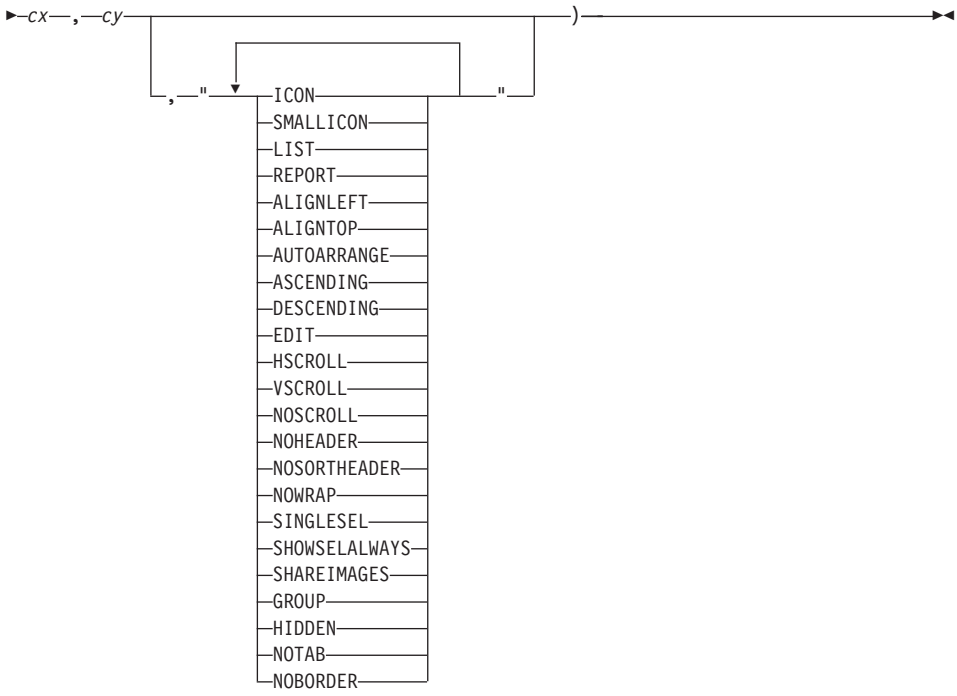
Example:

The following example creates a tree view at position x=100 and y=80 and with a size of width=40 and height=120. The ID of the tree is 555 and its data is associated with attribute BRANCH. The tree view uses lines for child items and roots, displays a button to the left of each parent item, and supports item editing.

```
MyDialog~AddTreeControl(555, "Branch", 100, 80, 40, 120,,
"LINES BUTTON EDIT ATROOT")
```

AddListControl





The *AddListControl* method adds a list view to the dialog and connects it with a data attribute. For further information on list view controls, refer to "Chapter 20. ListControl Class" on page 379.

Arguments:

The arguments are:

id A unique identifier assigned to the control. You need the ID to refer to this control in other methods.

attributeName

The name of the data attribute associated with the dialog item. See page 350 to get information on what happens when this argument is omitted.

x, y The position of the upper left corner of the control relative to the dialog, in dialog units.

cx, xy The width and height of the dialog item, in dialog units.

options

This argument determines the behavior and style of the dialog item and can be one or more of the following, separated by blanks:

ICON Use the icon view. Each item appears as a full-sized icon with a label below it. The user can drag the items to any location in the list view control.

SMALLICON
Use the small-icon view. Each item appears as a small icon with a label to the right of it. The user can drag the items to any location in the list view control.

LIST Use the list view. Each item appears as a small icon with a label to the right of it. Items are arranged in columns and cannot be moved by the user.

REPORT
Use the report view. Each item appears on a separate line with information arranged in columns. The leftmost column contains the small icon and label, and subsequent columns contain subitems.

ALIGNLEFT
In icon and small-icon views, the items are left-aligned.

ALIGNTOP
In icon and small-icon views, the items are aligned with the top of the control.

AUTOARRANGE
In icon and small-icon views, the icons are always automatically arranged.

ASCENDING
Sorts items by item text in ascending order.

DESCENDING
Sorts items by item text in descending order.

EDIT The list view allows the user to edit the list view items. To store the edited text, you must connect a method to the ENDEDIT notification or you connect the DEFAULTEDIT event handler (see "ConnectTreeNotify" on page 313.

HSCROLL
The list view supports a horizontal scroll bar.

VSCROLL
The list view supports a vertical scroll bar.

NOSCROLL
Disables scrolling.

NOHEADER
No column header is displayed in the report view. By default, columns have headers in the report view.

NOSORTHEADER
Specifies that column headers do not work like buttons. This option is useful if clicking a header in the report view does not carry out an action.

NOWRAP
Displays item text on a single line in the icon view. By default, the item text can wrap in the icon view.

SINGLESEL
Allows only one item to be selected at a time. By default, several items can be selected.

SHOWSELALWAYS
Specifies that a selected item remains selected when the list view loses focus.

DEFAULTEDIT
Connects the notification that label editing has been started and ended with a predefined event-handling method. This method extracts the newly entered text from the notification and modifies the item of which the label was edited. If this event is not connected you must provide your own event-handling method and connect it with the BEGINEDIT and ENDEDIT events. Otherwise, the edited text is lost and the item remains unchanged.

SHAREIMAGES
The control does not take ownership of the image lists assigned to it. This option enables an image list to be used with several list controls.

GROUP
Specifies the first control of a group of control in which the user can move from one control to the next with the arrow keys.

HIDDEN
The control is initially hidden.

NOTAB
The tab key cannot be used to move to this control.

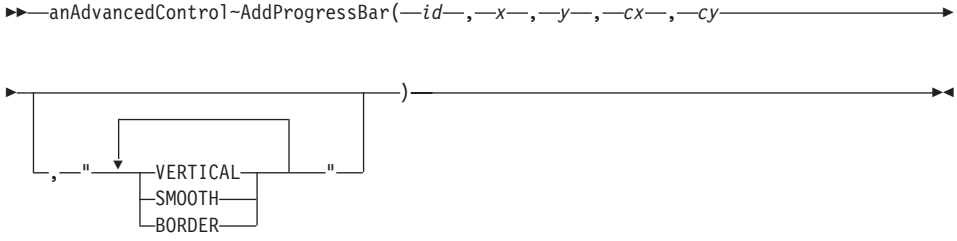
NOBORDER
No border is drawn around the control.

Example:

The following example creates a list view at position x=100 and y=80 and with a size of width=40 and height=120. The list view with ID 555 is a report view with items sorted in ascending order. It supports item editing and column headers do not behave like buttons. Its data is associated with attribute EMPLOYEES.

```
MyDialog~AddListControl(555, "EMPLOYEES", 100, 80, 40, 120,,
"REPORT ASCENDING EDIT NOSORTHEADER")
```

AddProgressBar



The *AddProgressBar* method adds a progress bar to the dialog and connects it with a data attribute. For further information on progress bar controls, refer to "Chapter 28. ProgressBarControl Class" on page 461.

Arguments:

The arguments are:

id A unique identifier assigned to the control. You need the ID to refer to this control in other methods.

x, y The position of the upper left corner of the control relative to the dialog, in dialog units.

cx, xy The width and height of the dialog item, in dialog units.

options

This argument determines the behavior and style of the dialog item and can be one or more of the following:

VERTICAL

The progress bar is oriented vertically.

SMOOTH

The progress bar is incremented smoothly.

BORDER

A border is drawn around the progress bar.

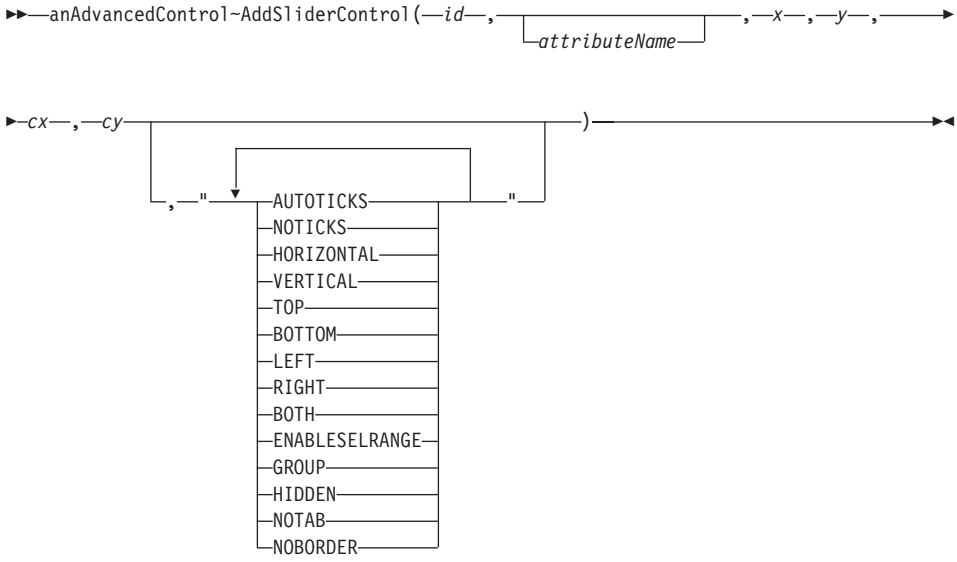
If you omit this argument, the progress bar is oriented horizontally.

Example:

The following example creates a progress bar with ID DONE at the bottom of the dialog. The progress bar is as wide as the dialog.

```
MyDialog~AddProgressBar("DONE",10,MyDialog~sizey-16,MyDialog~sizex=20,12)
```

AddSliderControl



The *AddSliderControl* method adds a slider control (track bar) to the dialog and connects it with a data attribute. For further information on slider controls, refer to "Chapter 29. SliderControl Class" on page 465.

Arguments:

The arguments are:

id A unique identifier assigned to the control. You need the ID to refer to this control in other methods.

attributeName
The name of the data attribute associated with the dialog item. See page 350 to get information on what happens when this argument is omitted.

x, y The position of the upper left corner of the control relative to the dialog, in dialog units.

cx, xy The width and height of the dialog item, in dialog units.

options

This argument determines the behavior and style of the dialog item and can be one or more of the following:

AUTOTICKS
Creates a slider that has a tick mark for each increment in its range of values. These tick marks are automatically added when an application calls the InitRange method. You cannot use the SetTickAt and SetTickFrequency methods to specify the position of the tick marks when you use this option.

NOTICKS
Creates a slider that does not display tick marks.

HORIZONTAL
Orients the slider horizontally. This is the default orientation.

VERTICAL
Orients the slider vertically.

TOP Displays tick marks along the top of a horizontal slider.

BOTTOM
Displays tick marks along the bottom of a horizontal slider. This option can be used together with the TOP option to display tick marks on both sides of the slider control.

LEFT Displays tick marks along the left of a vertical slider.

RIGHT
Displays tick marks along the right of a vertical slider. This option can be used together with the LEFT option to display tick marks on both sides of the slider control.

BOTH Displays tick marks on both sides of the slider in any direction.

ENABLESELRANGE
Displays a selection range. If you set this option, the tick marks at the starting and ending positions of a selection range are displayed as triangles and the selection range is highlighted. This can be used, for example, to indicate a preferred selection.

GROUP
Specifies the first control of a group of control in which the user can move from one control to the next with the arrow keys.

HIDDEN
The control is initially hidden.

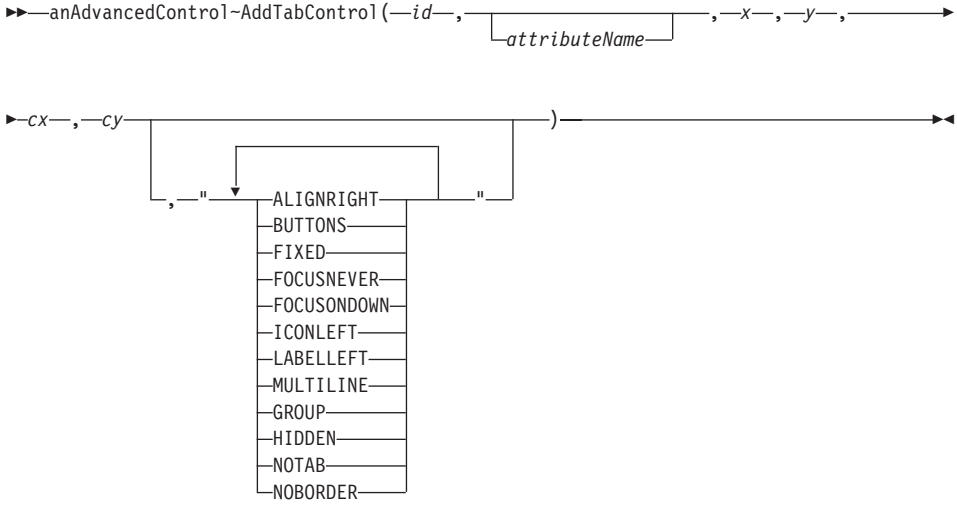
NOTAB
The tab key cannot be used to move to this control.

NOBORDER
No border is drawn around the control.

Example:
The following example creates a vertical slider control at the right border of the dialog. The slider with ID PRESSURE displays automatic tick marks on both sides. Its position is associated with attribute PRESSURE.

```
MyDialog~AddSliderControl("PRESSURE",, MyDialog~sizeX-30, 15, 20,,
MyDialog~sizeY-30, "VERTICAL BOTH AUTOTICKS")
```

AddTabControl



The *AddTabControl* method adds a tab control to the dialog and connects it with a data attribute. For further information on tab controls, refer to "Chapter 30. TabControl Class" on page 479.

Arguments:

The arguments are:

id A unique identifier assigned to the control. You need the ID to refer to this control in other methods.

attributeName
The name of the data attribute associated with the dialog item. See page 350 to get information on what happens when this argument is omitted.

x, y The position of the upper left corner of the control relative to the dialog, in dialog units.

cx, xy The width and height of the dialog item, in dialog units.

options
This argument determines the behavior and style of the dialog item and can be one or more of the following:

ALIGNRIGHT
Right-justifies tabs. By default, tabs are left-justified within a row.

BUTTONS
Modifies the appearance of the tabs to look like buttons.

FIXED
Makes all tabs equal in width. You cannot use this option with the ALIGNRIGHT option.

FOCUSNEVER
A tab never receives the input focus.

FOCUSONDOWN
A tab receives the input focus when clicked (typically with option BUTTONS).

ICONLEFT
Forces the icon to the left, but leaves the tab label centered. By default, the control centers the icon and label, with the icon being to the left of the label.

LABELLEFT
Left-aligns both the icon and the label.

MULTILINE
Causes a tab control to display several rows of tabs, enabling all tabs to be displayed at the same time.

GROUP
Specifies the first control of a group of control in which the user can move from one control to the next with the arrow keys.

HIDDEN
The control is initially hidden.

NOTAB
The tab key cannot be used to move to this control.

NOBORDER
No border is drawn around the control.

Example:

The following example creates a tab control with ID PAGES and multiline capability. Its data (the selected tab) is associated with attribute CURRENTPAGE.

```
MyDialog~AddTabControl("PAGES","CURRENTPAGE", 10, 120, 200, 20,,
"MULTILINE FIXED")
```

Chapter 18. StaticControl Class

The *StaticControl* class provides methods to query and modify static controls, such as static text, group boxes, and frames. It inherits all methods of the DialogControl class (see page 181).

The *StaticControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Use the GetStaticControl method (see page 338) to retrieve an object of the *StaticControl* class. To use this method, the static control must have a positive ID.

Chapter 19. EditControl Class

The *EditControl* class provides methods to query and modify edit controls, which are also called entry lines. It inherits all methods of the DialogControl class (see page 181).

Use the GetEditControl method (see page 339) to retrieve an object of the *EditControl* class.

Requires:

The *EditControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *EditControl* class implement the methods listed in Table 7.

Table 7. EditControl Instance Methods

Method...	...on page
EnsureCaretVisibility	370
FirstVisibleLine	375
GetLine	377
IsModified	371
LineFromIndex	373
LineIndex	372
LineLength	373
LineScroll	370
Lines	372
Margins	377
PasswordChar	375
PasswordChar=	374
ReplaceSelText	374
ScrollCommand	369
Select	368
Selected	368
SetLimit	374
SetMargins	376

Table 7. EditControl Instance Methods (continued)

Method...	...on page
SetModified	371
SetReadOnly	376

Selected

```
►►—anEditControl~Selected————————————————————————————————————►◄
```

The *Selected* method retrieves the indexes of the starting and ending character of the text selected. If the starting index equals the ending index, no text is selected and the index specifies the current cursor position. If the ending index is 0 and the starting index is 1, the entire text is selected.

Return value:

The one-based starting and ending index of the current selection, separated by a blank.

Example:

The following example displays the starting and ending index of the text selection of the edit control NAME. It then selects the entire text of the edit control.

```
edit = MyDialog~GetEditControl("NAME")
if edit == .Nil then return
parse value edit~Selected with start end
say "Starting index of selection is" start
say "Ending index of selection is" end
edit~Select(1,0)
```

Select

```
►►—anEditControl~Select(start,end)———————————————————————————►◄
```

The *Select* method selects the text or sets the cursor position for the associated edit control. If the starting index equals the ending index, no text is selected and the cursor is set to the character at the specified index. If the ending index is 0 and the starting index is 1, the entire text is selected.

Arguments:

The arguments are:

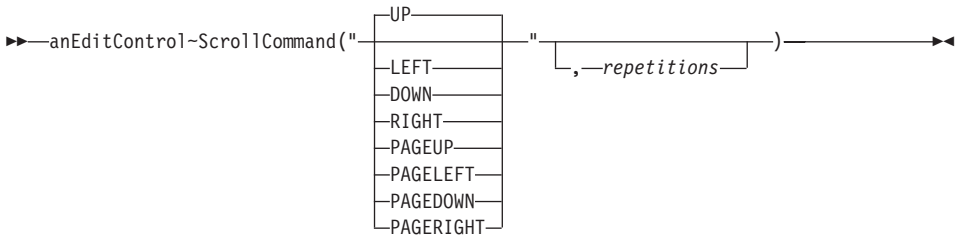
start A one-based index where the selection begins or the cursor is to be set.

end A one-based index where the selection ends.

Example:

See "Selected" on page 368.

ScrollCommand



The *ScrollCommand* method scrolls the associated edit control in a given direction.

Arguments:

The arguments are:

command

Specifies the direction and the step size of the scroll command. Possible values are:

UP or LEFT

Scrolls up one line. UP is the default.

DOWN or RIGHT

Scrolls down one line.

PAGEUP or PAGELEFT

Scrolls up one page.

PAGEDOWN or PAGERIGHT

Scrolls down one page.

repetitions

The number of times the scroll command is to be issued. If this argument is omitted, the scroll command is issued once.

Example:

The following example scrolls down 3 lines in the edit control:

```
edit = MyDialog~GetEditControl("NAME")
if edit == .Nil then return
edit~ScrollCommand("DOWN",3)
```

LineScroll

The *LineScroll* method scrolls the text in a multiline edit control vertically or horizontally by the specified number of characters and lines.

Arguments:

The arguments are:

sChars The number of characters to be scrolled horizontally.

sLines The number of lines to be scrolled vertically.

Return value:

0 if the message is sent to a multiline edit control, or a non-zero value if the message is sent to a single-line edit control.

Example:

The following example scrolls an edit control by 50 characters and 35 lines:

```
edit~Scroll(50, 35)
```

Note: The edit control does not scroll vertically past the last line of text. If the current line, plus the number of lines specified by *sLines*, exceeds the total number of lines in the edit control, the last line of the edit control is scrolled to the top. The LineScroll message can, however, be used to scroll horizontally past the last character of a line.

EnsureCaretVisibility

The *EnsureCaretVisibility* method scrolls the edit control until the caret (cursor) is visible.

Return value:

0 if the object is associated with an existing edit control.

IsModified

```
►►—anEditControl~IsModified————————————————————————————————————►◄
```

The *IsModified* method retrieves information on whether the edit control has been modified.

Return value:

1 The text in the edit control has been altered.

0 For all other cases.

Example:

```
if edit~IsModified = 1 then MyDialog~Save
```

SetModified

```
►►—anEditControl~SetModified(bool)—————————————————————————————►◄
```

The *SetModified* method sets the flag to indicate whether the edit control has been modified.

Arguments:

The only argument is:

bool

1 The flag indicates that the text has been altered.

0 For all other cases.

Example:

In the following example, the Save method stores the dialog contents in a file and clears the modified flag:

```
::method Save
    /* write contents to file */
    edit = MyDialog~GetEditControl("TEXT")
    ...
    edit~SetModified(0)
```

Lines

```
►►─anEditControl~Lines──────────────────────────────────────────────────►◄
```

The *Lines* method retrieves the number of text lines of a multiline edit control.

Return value:
The number of text lines.

Example:
For an example, refer to "LineIndex".

LineIndex

```
►►─anEditControl~LineIndex(line)────────────────────────────────────────►◄
```

The *LineIndex* method retrieves the one-based character index of the beginning of the *line* in the associated edit control.

Arguments:
The only argument is:

line The number of the line of which the starting index is to be retrieved. Line numbers are incremented by 1, starting with 1.

Return value:
The character index of the specified line. The first line starts at index 1.

Example:
The following example sets a text for edit control TEXT and displays the number of text lines (3), the starting index of the second line (carriage return and line feed, which mark a line break, are also considered to be characters), and the length of the third line:

```
edit = MyDialog~GetEditControl("TEXT")
if edit == .Nil then return

"It is easy to learn and easy to use." || 13~d2c || 10~d2c ||,
"Have fun with it!"
say "Number of lines:" edit~lines
     say "Line 2 begins at index" edit~LineIndex(2)
     say "Length of 3rd line:" edit~LineLength(3)

Result: The number of lines: 3
        Line 2 begins at index 37
        Length of 3rd line: 17
```

LineLength

The *LineLength* method retrieves the number of characters contained in the *line* in the associated edit control.

Arguments:

The only argument is:

line The number of the line of which the number of characters is to be retrieved. Line numbers are incremented by 1, starting with 1.

Return value:

The length of the given line.

Example:

For an example, refer to "LineIndex" on page 372.

LineFromIndex

The *LineFromIndex* method retrieves the one-based line number that contains the character index *index*.

Arguments:

The only argument is:

index The one-based character index contained in the line whose number is to be retrieved.

Return value:

The line number containing the specified character index. The first line starts at index 1. If the specified index exceeds the number of characters contained in the edit control or an invalid character index was specified, 0 is returned.

Example:

The following example displays the line in which character 55 is contained:

```
edit = MyDialog~GetEditControl("TEXT")
if edit == .Nil then return

"It is easy to learn and easy to use." || 13~d2c || 10~d2c ||,
"Have fun with it!"
```

```
say "Character 55 is contained in line" edit~LineFromIndex(55)

Result: Character 55 is contained in line 2
```

ReplaceSelText

```
►►—anEditControl~ReplaceSelText(text)————————————————————————————►◄
```

The *ReplaceSelText* method replaces the selected text in the associated edit control with a new one.

Arguments:

The only argument is:

text The text string that is to replace the currently selected text.

Example:

```
edit = MyDialog~GetEditControl("TEXT")
if edit == .Nil then return
edit~Title = "Object REXX is a hybrid language."
edit~Select(17,25)
    edit~ReplaceSelText("n interpreted")
    say edit~Title

Result: Object REXX is an interpreted language.
```

SetLimit

```
►►—anEditControl~SetLimit(chars)—————————————————————————————————►◄
```

The *SetLimit* method sets the maximum numbers of characters that the associated edit control can contain.

Arguments:

The only argument is:

chars The number of characters that the edit control can contain.

PasswordChar=

```
►►—anEditControl~PasswordChar=char———————————————————————————————►◄
```

The *PasswordChar=* method sets the character that is displayed in an edit control for which the PASSWORD or ES_PASSWORD flag is set.

Arguments:

The only argument is:

char The character that is displayed for the typed characters.

Example:

The following example ensures that if the PASSWORD style was chosen for the edit control in the resource workshop, the dollar sign ($) is displayed for each character typed in the edit control:

```
edit = MyDialog~GetEditControl("TEXT")
if edit == .Nil then return
edit~PasswordChar = "$"
say "The new password character is" edit~PasswordChar
```

PasswordChar

```
►►—anEditControl~PasswordChar————————————————————————————————————►◄
```

The *PasswordChar* method retrieves the character that is displayed in an edit control for which the PASSWORD option or ES_PASSWORD style was set in the resource workshop.

Return value:

The character that is displayed instead of the characters contained in the edit control. If the edit control is no password field or no password character was set, an empty string is returned.

Example:

For an example, refer to "PasswordChar=" on page 374.

FirstVisibleLine

```
►►—anEditControl~FirstVisibleLine————————————————————————————————►◄
```

The *FirstVisibleLine* method retrieves the one-based line number of the first line visible in a multiline edit control.

Return value:

The number of the first visible line, starting with 1.

Example:

For an example, refer to "PasswordChar=" on page 374.

SetReadOnly

The *SetReadOnly* method sets or unsets the read-only flag for the associated edit control. If the read-only flag is set, the user can no longer modify the text of the edit control.

Arguments:

The only argument is:

bool 1 if the edit control is to be marked as a read-only field (the default), or 0 if new text can be typed into the edit control.

Return value:

0 if this method was successful.

SetMargins

The *SetMargins* method sets the left and right margins for the associated edit control. The margins determine the spacing to the left and right of the edit control.

Arguments:

The arguments are:

left The left margin, specified in screen pixels.

right The right margin, specified in screen pixels.

Example:

The following example sets the margins for edit control TEXT such that the left indent is 10 screen pixels and on the right there is a spacing of 5 pixels between the text and the frame of the edit control:

```
edit = MyDialog~GetEditControl("TEXT")
if edit == .Nil then return
edit~SetMargins(10, 5)
     parse value edit~Margins with left right
     say "The new left margin is" left" and the new right margin is" right
```

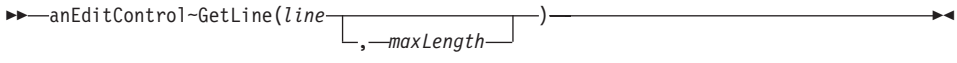
Margins

The *Margins* method retrieves the left and right margins of the associated edit control.

Return value:
The left and right margins, in screen pixels, separated by a blank.

Example:
For an example, refer to "SetMargins" on page 376.

GetLine



The *GetLine* method retrieves the text string contained in the specified line.

Arguments:
The arguments are:

line The one-based line number to be retrieved.

maxLength
The maximum number of characters to be retrieved. Object REXX allocates the appropriate amount of memory to store the text string. If the line consists of more characters than fit in the memory, the text string is truncated. If you omit this argument, the maximum number of characters retrieved is 255.

Return value:
A text string or an empty string.

Example:
The following example stores all lines contained in edit control EDITOR in a stem. If a line consists of more than 1024 characters, it is truncated.

```
edit = MyDialog~GetEditControl("EDITOR")
if edit == .Nil then return
do i = 1 to edit~Lines
   lines.i = edit~GetLine(i, 1024)
end
```

Note: The carriage return and line-feed characters are not included in the returned text string. To get the contents of a single line edit control, use the Title method (see page 196) or call GetLine(i, 1024).

Chapter 20. ListControl Class

A list view control is a window that displays a collection of items, with each item consisting of an icon and a label. It provides several ways of arranging and displaying items. Refer to OODLIST.REX in the OODIALOG\SAMPLES directory for an example.

Requires:

The *ListControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *ListControl* class implement the methods listed in Table 8.

Table 8. ListControl Instance Methods

Method...	...on page
Add	393
AddRow	394
AddStyle	383
AlignLeft	409
AlignTop	409
Arrange	408
BkColor	412
BkColor=	412
ColumnInfo	387
ColumnWidth	388
Delete	395
DeleteAll	395
DeleteColumn	385
Deselect	399
DropHighlighted	400
Edit	410
EndEdit	411
EnsureVisible	404
Find	407
FindNearestXY	408

Table 8. ListControl Instance Methods (continued)

Method...	**...on page**
FindPartial	407
FirstVisible	401
Focus	400
Focused	400
Insert	389
InsertColumn	385
ItemInfo	396
ItemPos	409
ItemState	398
ItemText	398
Items	395
ItemsPerPage	411
Last	396
LastSelected	399
Modify	390
ModifyColumn	386
Next	402
NextLeft	402
NextRight	402
NextSelected	401
Prepare4nItems	396
Previous	402
PreviousSelected	401
RedrawItems	403
RemoveImages	406
RemoveStyle	384
RemoveSmallImages	406
ReplaceStyle	382
RestoreEditClass	411
Scroll	411
Select	398
Selected	399

Table 8. ListControl Instance Methods (continued)

Method...	...on page
SelectedItems	396
SetColumnWidth	388
SetImages	405
SetItemPos	410
SetItemState	392
SetItemText	391
SetSmallImages	405
SmallSpacing	403
SnapToGrid	408
Spacing	403
StringWidth	389
SubclassEdit	411
TextBkColor	413
TextBkColor=	414
TextColor	413
TextColor=	413
Update	404
UpdateItem	404

View Styles

List view controls can display their contents in different views. The current view is specified by the window style of the control. Additional window styles define the alignment of the items and the functionality of the list view control. The different views are:

Icon view
Each item appears as a full-sized icon with a label below it. The user can drag the items to any location in the list view.

Small-icon view
Each item appears as a small icon with a label to the left of it. The user can drag the items to any location.

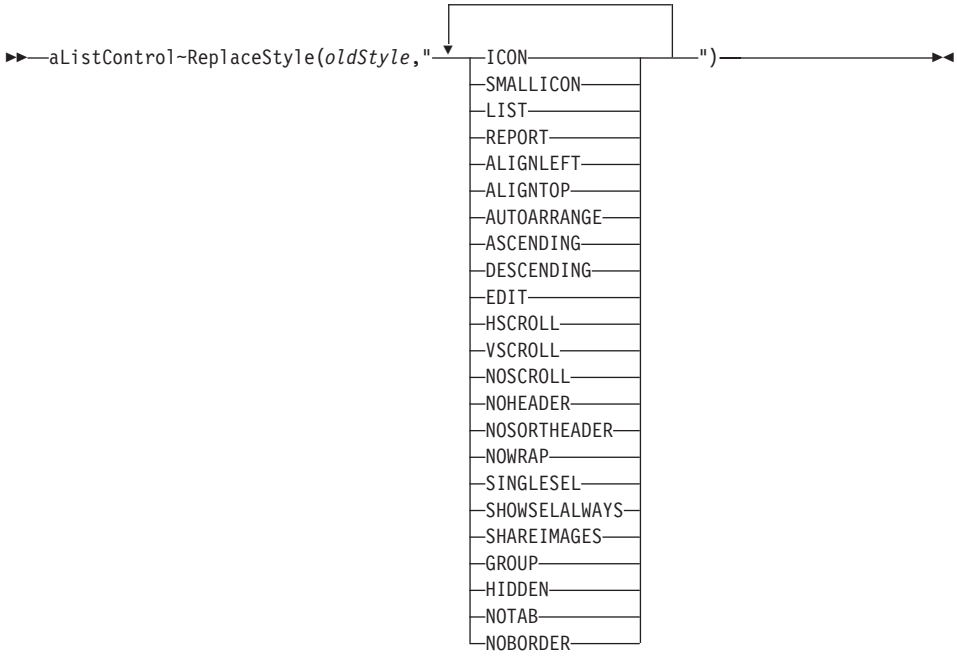
List view
Each item appears as a small icon with a label to the left of it. The user cannot drag the items.

Report view
Each item appears on a separate line with information arranged in columns. The leftmost column contains the small icon and the label. All following columns contain subitems as specified by the application.

Methods of the ListControl Class

The following sections describe the individual methods.

ReplaceStyle



The *ReplaceStyle* method removes a window style of a list view control and sets new styles.

Arguments:
The arguments are:

oldStyle
The window style to be removed.

newStyle
The new window styles to be set, which is one or more of the

styles listed in the syntax diagram, separated by blanks. For an explanation of the different styles, refer to "AddListControl" on page 354.

Return value:

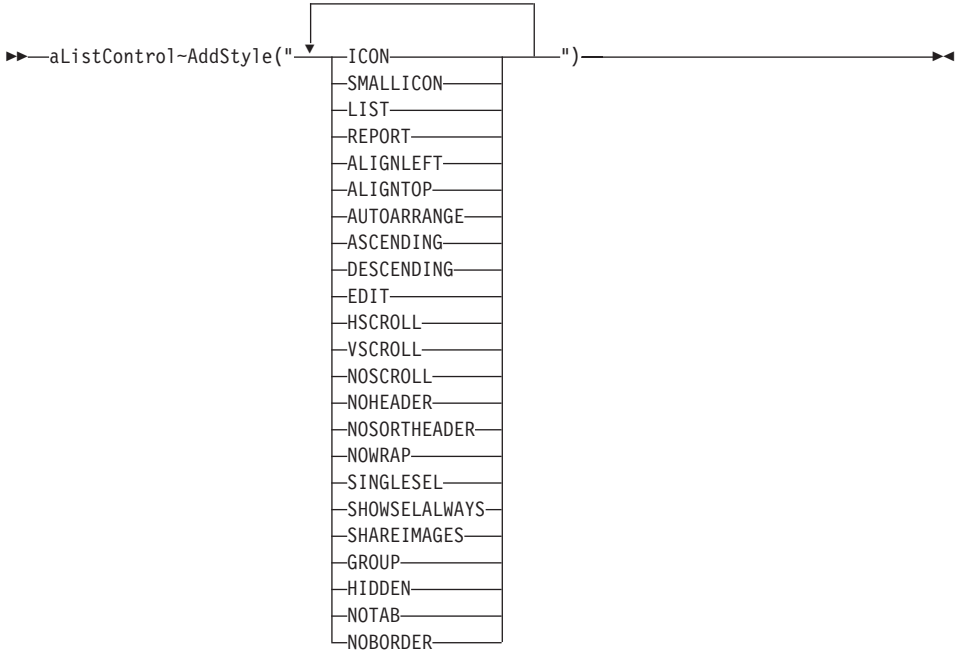
0 if this method fails.

Example:

The following example replaces a small-icon list with an icon list and connects the new bitmap file:

```
::method Icon
  curList = self~GetListControl(104)
  curList~SetImages("ilist.bmp",16,12)
  curList~ReplaceStyle("SMALLICON","ICON")
```

AddStyle



The *AddStyle* method adds new window styles to a list view control.

Arguments:

The only argument is:

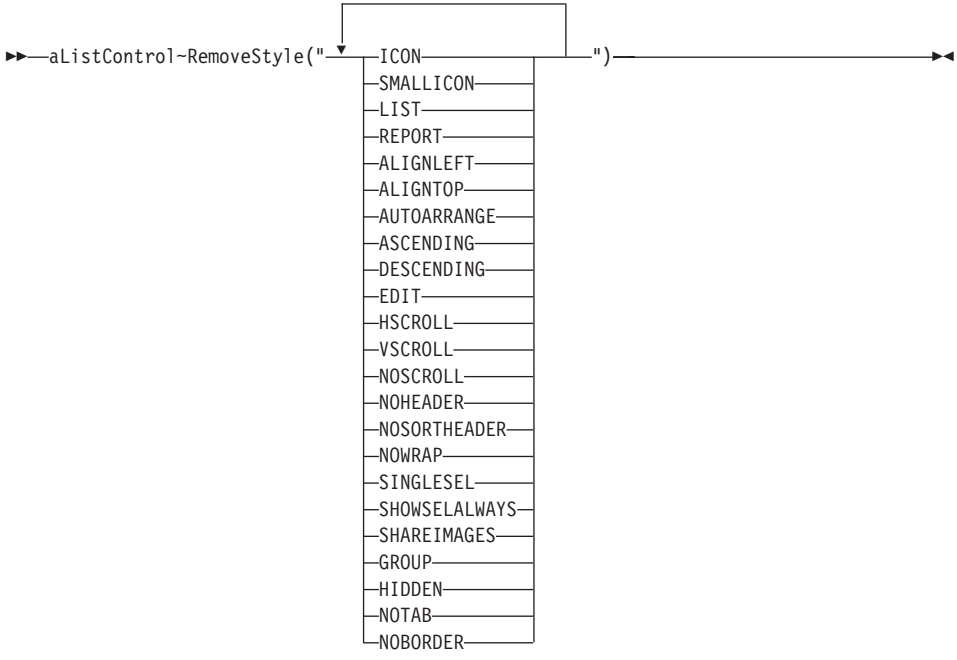
style The window styles to be added, which is one or more of the

styles listed in the syntax diagram, separated by blanks. For an explanation of the different styles, refer to "AddListControl" on page 354.

Return value:

0 if this method fails.

RemoveStyle



The *RemoveStyle* method removes one or more window styles of a list view control.

Arguments:

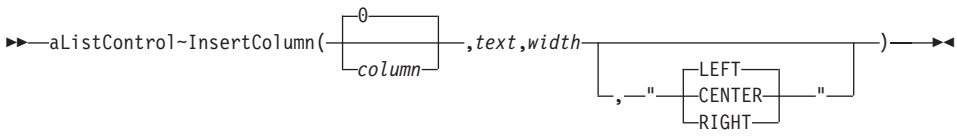
The only argument is:

style The window styles to be removed, which is one or more of the styles listed in the syntax diagram, separated by blanks. For an explanation of the different styles, refer to "AddListControl" on page 354.

Return value:

0 if this method fails.

InsertColumn



The *InsertColumn* method sets the attributes of a report list view column.

Arguments:

The arguments are:

column
The number of the column. 0 is the first column and the default.

text The text of the column heading.

width The width of the column, in pixels.

align The alignment of the column heading and the subitem text within the column. It can be one of the following values:

CENTER The text is centered.

LEFT The text is left-aligned, which is the default.

RIGHT The text is right-aligned.

Return value:

The number of the new column, or 0 if this method fails.

Example:

The following example adds three columns to a report list:

```
::method InitReport
  curList = self~GetListControl(102)
  if curList \= .Nil then
  do
    curList~InsertColumn(0,"First Name",50)
    curList~InsertColumn(1,"Last Name",50)
    curList~InsertColumn(2,"Age",50)
  end
```

DeleteColumn

The *DeleteColumn* method removes a column from a list view control.

Arguments:

The only argument is:

column

The number of the column to be deleted. The first column must be deleted last.

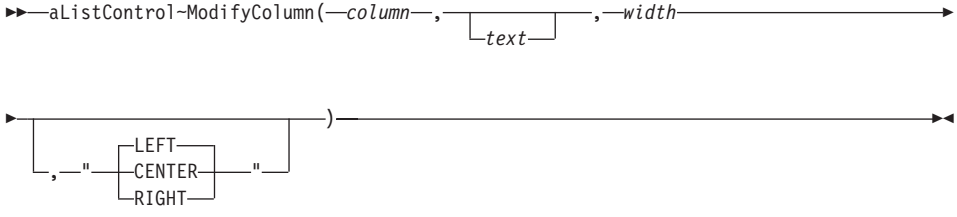
Return value:

0 The column was deleted.

–1 You did not specify *column*.

1 For all other cases.

ModifyColumn



The *ModifyColumn* method sets new attributes for a column of a list view control.

Arguments:

The arguments are:

column

The number of the column. 0 is the first column.

text The text of the column heading. If you omit this argument, the heading is not changed.

width The width of the column, in pixels.

align The alignment of the column heading and the subitem text within the column. It can be one of the following values:

CENTER The text is centered.

LEFT The text is left-aligned, which is the default.

RIGHT The text is right-aligned.

Return value:

0 The column was modified.

–1 You did not specify *column*.

1 For all other cases.

Example:

The following example changes the title, size, and alignment of the first column in a report list:

```
::method ChangeColumn
  curList = self~GetListControl(102)
  curList~ModifyColumn(0,"New Title",100,"RIGHT")
```

ColumnInfo

The *ColumnInfo* method retrieves the attributes of a column of a list view control.

Arguments:

The only argument is:

column

The number of the column of which the attributes are to be retrieved. 0 is the first column.

Return value:

A compound variable that stores the attributes of the item, or –1 if this method fails. The attributes are:

RetStem.!TEXT

The heading of the column.

RetStem.!COLUMN

The column number.

RetStem.!WIDTH

The width of the column.

RetStem.!ALIGN

The alignment of the column: "LEFT", "RIGHT", or "CENTER".

Example:

The following example displays the column attributes in an information box when the column is clicked on:

```
::method OnColumnClick
  use arg id, column
  curList = self~GetListControl(102)
  info. = curList~ColumnInfo(column)
  call InfoDialog("Column Title : " info.!Text"d"x,
                  "Column Number: " info.!Column"d"x,
                  "Column Width : " info.!Width"d"x,
                  "Alignment    : " info.!Align)
```

ColumnWidth

The *ColumnWidth* method retrieves the width of a column in a report or list view.

Arguments:

The only argument is:

column

The number of the column of which the width is to be retrieved. 0 is the first column.

Return value:

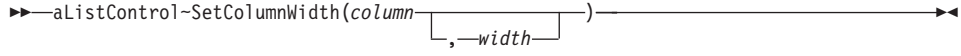
The column width, or –1 if you did not specify *column*, or 0 in all other cases.

Example:

The following example displays the column width in an information box when the column is clicked on:

```
::method OnColumnClick
  use arg id, column
  curList = self~GetListControl(102)
  call InfoDialog(curList~ColumnWidth(column))
```

SetColumnWidth



The *SetColumnWidth* method sets the width of a column in a report or list view.

Arguments:

The arguments are:

column

The number of the column of which the width is to be set. 0 is the first column.

width The width of the column, in pixels. If you omit this argument, the column is sized automatically.

Return value:

0 The column width was set.

–1 You did not specify *column*.

1 For all other cases.

Example:

The following example enlarges the selected column by 10:

```
::method OnColumnClick
  use arg id, column
  curList = self~GetListControl(102)
  curList~SetColumnWidth(column,curList~ColumnWidth(column)+10)
```

StringWidth

The *StringWidth* method determines the width of a specified string using the current font of the list view control.

Arguments:

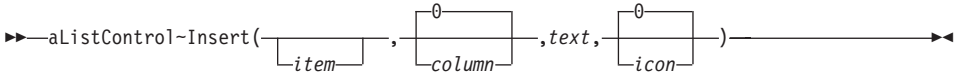
The only argument is:

text The text string of which the width is to be determined.

Return value:

The string width, or –1 if you did not specify a *text*, or 0 in all other cases.

Insert



The *Insert* method inserts a new item in a list view control.

Arguments:

The arguments are:

item The number of the item. If you omit this argument, the number of the last item is increased by 1.

column
The number of the column. If you omit this argument, 0 is assumed. This argument only applies to report views.

text The text of the item.

icon The index of the icon of the list view item within the bitmap file, set with the SetImages or SetSmallImages method (see

page 405). The SetImages method must be used for the icon view and the SetSmallImages for the list, report, and small-icon views.

In a report view, this argument can only be used for the first column.

If you omit this argument, 0 is assumed.

Return value:

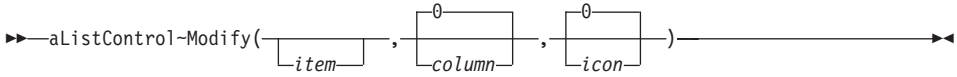
The index of the new item, or −1 in all other cases.

Example:

The following example inserts items in a list:

```
::method InitReport
  curList = self~GetListControl(102)
  if curList \= .Nil then
  do
    curList~Insert(,,"First")
    curList~Insert(,,"Second")
    curList~Insert(,,"Third")
  end
```

Modify



The *Modify* method sets some or all attributes of a list view item.

Arguments:

The arguments are:

item The number of the item. If you omit this argument, the selected item is used.

column
The number of the column. If you omit this argument, 0 is assumed. This argument only applies to report views.

text The new text for the item.

icon The new index for the icon of the list view item within the bitmap file, set with the SetImages or SetSmallImages method (see page 405). The SetImages method must be used for the icon view and the SetSmallImages method for the list, report, and small-icon views.

In a report view, this argument can only be used for the first column.

If you omit this argument, 0 is assumed.

Return value:

0 The modification was successful.

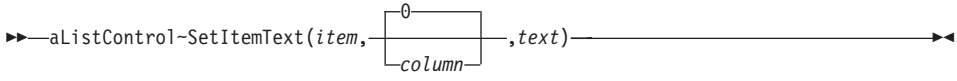
1 For all other cases.

Example:

The following example modifies the icon of the item that is double-clicked:

```
::method OnActivate
  curList = self~GetListControl(102)
  if curList \= .Nil then
  do
    si = curlist~Focused
    curList~Modify(si,,,2)
  end
```

SetItemText



The *SetItemText* method changes the text of a list view item or a column.

Arguments:

The arguments are:

item The number of the item.

column

The number of the column. If you omit this argument, 0 is assumed. This argument only applies to report views.

text The text of the item or a column.

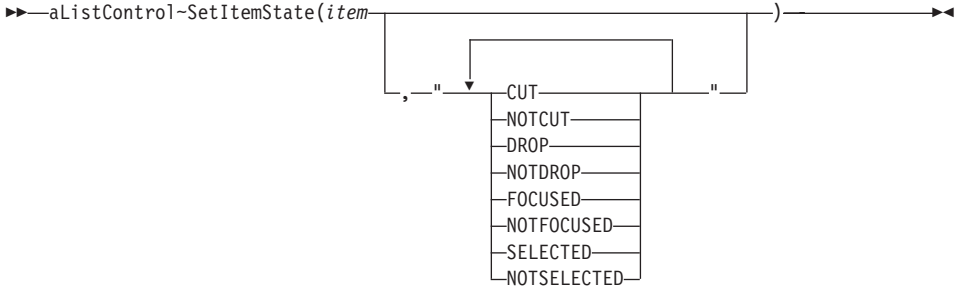
Return value:

0 The change was successful.

–1 You did not specify *column*.

1 For all other cases.

SetItemState



The *SetItemState* method sets the state of a list view item.

Arguments:

The arguments are:

item The number of the item.

state The state of the item, which can be one or more of the following values, separated by blanks:

CUT The item is marked for a cut-and-paste operation.

NOTCUT The item cannot be used for a cut-and-paste operation.

DROP The item is highlighted as a drag-and-drop target.

NOTDROP The item is not highlighted as a drag-and-drop target.

FOCUSED The item has the focus and is therefore surrounded by the standard focus rectangle. Only one item can have the focus.

NOTFOCUSED
The item does not have the focus.

SELECTED The item is selected. Its appearance depends on whether it has the focus and on the system colors used for a selection.

NOTSELECTED
The item is not selected.

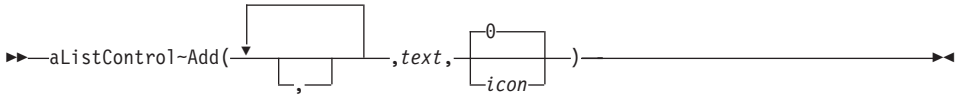
Return value:

0 The state was set successfully.

–1 You did not specify *item*.

1 For all other cases.

Add



The *Add* method adds a new item to the report view. It can be used to fill a list view or report view sequentially.

Arguments:

The arguments are:

, The number of commas specifies in which column the text of the item is to be placed. For example, one comma specifies that the text of the item is to be placed in the first column.

text The text for the item.

icon The index of the icon of the list view item within the bitmap file, set with the SetImages or SetSmallImages method (see page 405). The SetImages method must be used for the icon view and the SetSmallImages method for the list, report, and small-icon views.

In a report view, this argument can only be used for the first column.

If you omit this argument, 0 is assumed.

Example:

The following example adds three columns and two items to a report list control. To get the following result:

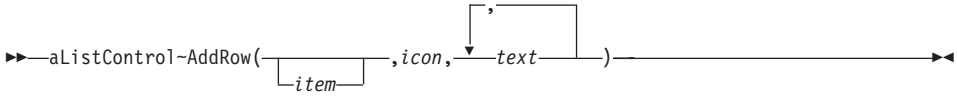
First Name	Last Name	Age
Mike	Miller	30
Sue	Thaxtor	29

you must specify the following:

```
::method InitDialog
  InitDlgRet = self~InitDialog:super

  curList = self~GetListControl("IDC_LIST_REP")
  if curList \= .Nil then
  do
```

```
  curList~SetSmallImages("E:\oodlist\oodlist.BMP",16,12)
  curList~InsertColumn(0,"First Name",50)
  curList~InsertColumn(1,"Last Name",50)
  curList~InsertColumn(2,"Age",50)
  curList~Add("Mike")
  curList~Add(,"Miller")
  curList~Add(,,"30")
  curList~~Add("Sue")~~Add(,"Thaxton")~~Add(,,"29")
end
return InitDlgRet
```

AddRow



The *AddRow* method adds a new item to a list.

Arguments:

The arguments are:

item The number of the item. If you omit this argument, the number of the last item is increased by 1.

icon The index of the icon of the list view item within the bitmap file, set with the SetImages or SetSmallImages method (see page 405). The SetImages method must be used for the icon view and the SetSmallImages method for the list, report, and small icon views.

In a report view, this argument can only be used for the first column.

text Any number of text strings. The first is used for the first column, the second for the second column, and so on. If you specify more text entries than there are columns, the extra entries are ignored.

Return value:

The index of the new item, or –1 in all other cases.

Example:

The following example adds three items to a report list with two columns:

```
::method InitList
  curList = self~GetListControl(101)
  curList~AddRow(,,"Mike","Miller")
  curList~AddRow(,,"Sue","Muller")
  curList~AddRow(,,"Chris","Watson")
```

Delete

The *Delete* method removes an item from a list view control.

Arguments:

The only argument is:

item The number of the item.

Return value:

0 The item was deleted.

–1 You did not specify *item*.

1 For all other cases.

Example:

The following example deletes the selected item in a list control:

```
::method DeleteSelectedItem
  curList = self~GetListControl(102)
  curList~Delete(curList~Selected)
```

DeleteAll

The *DeleteAll* method removes all items from a list view control.

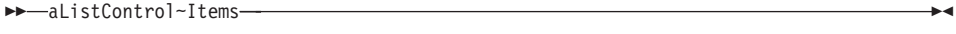
Return value:

0 The items were deleted.

–1 No item was available.

1 For all other cases.

Items



The *Items* method retrieves the number of items in a list view control.

Return value:

The number of items.

Last

The *Last* method retrieves the number of the last item in a list view control.

Return value:
The number of the last item.

Prepare4nItems

The *Prepare4nItems* method prepares a list view control for adding a large number of items.

Arguments:
The only argument is:

items The number of the items to be added later.

Return value:

0 The list view control was prepared.

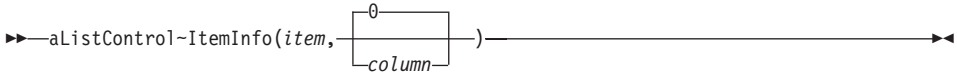
–1 You did not specify *items*.

SelectedItems

The *SelectedItems* method determines the number of selected items in a list view control.

Return value:
The number of selected items.

ItemInfo



The *ItemInfo* method retrieves the attributes of a list view item.

Arguments:

The arguments are:

item The number of the item.

column

The number of the column. If you omit this argument, 0 is assumed.

Return value:

A compound variable that stores the attributes of the item, or –1 in all other cases. The compound variable can be:

RetStem.!TEXT

The item text.

RetStem.!IMAGE

The index of the icon of the list view item within the bitmap file, set with the SetImages or SetSmallImages method (see page 405). The SetImages method must be used for the icon view and the SetSmallImages method for the list, report, and small icon views.

In a report view, this argument can only be used for the first column.

RetStem.!STATE

One or more of the following values:

CUT The item is marked for a cut-and-paste operation.

DROPPED

The item is highlighted as a drag-and-drop target.

FOCUSED

The item has the focus and is therefore surrounded by the standard focus rectangle. Only one item can have the focus.

SELECTED

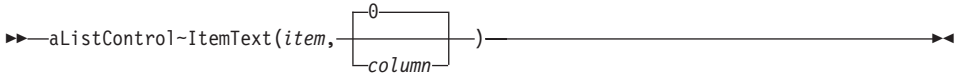
The item is selected. Its appearance depends on whether it has the focus and on the system colors used for a selection.

Example:

The following example displays the item text, icon index, and the item state in a message box:

```
::method DisplayItemInfo
  curList = self~GetListControl(104)
  itemInfo. = curList~ItemInfo(curList~Selected)
  call InfoDialog("Item Text   : " itemInfo.!Text"d"x,
                  "Image Index : " itemInfo.!Image"d"x,
                  "Item State  : " itemInfo.!State)
```

ItemText



The *ItemText* method retrieves the text of a list view item or a column.

Arguments:

The arguments are:

item The number of the item.

column

The number of the column. If you omit this argument, 0 is assumed.

Return value:

The item or column text, or –1 if you did not specify *item*.

ItemState

The *ItemState* method retrieves the state of a list view item.

Arguments:

The only argument is:

item The number of the item.

Return value:

The state of the item, which can be one or more of the following values:

CUT The item can be used in a cut-and-paste operation.

DROPPED The item is highlighted as a drag-and-drop target.

FOCUSED The item has the focus and is therefore surrounded by the standard focus rectangle. Only one item can have the focus.

SELECTED The item is selected. Its appearance depends on whether it has the focus and on the system colors used for a selection.

Select

The *Select* method selects an item.

Arguments:

The only argument is:

item The number of the item.

Return value:

0 The item was selected.

–1 You did not specify *item*.

1 For all other cases.

Deselect

The *Deselect* method deselects an item.

Arguments:

The only argument is:

item The number of the item.

Return value:

0 The item was selected.

–1 You did not specify *item*.

1 For all other cases.

Selected

The *Selected* method returns the number of the selected item.

Return value:

The number of the item selected last, or –1 in all other cases.

LastSelected

The *LastSelected* method returns the number of the item selected last.

Return value:
The number of the item selected last, or –1 in all other cases.

Focused

The *Focused* method retrieves the number of the item that has currently the focus.

Return value:
The number of the item with the focus, or –1 in all other cases.

Focus

The *Focus* method assigns the focus to the specified item, which is then surrounded by the standard focus rectangle. Although more than one item can be selected, only one item can have the focus.

Arguments:
The only argument is:

item The number of the item to receive the focus.

Return value:

0 The specified item received the focus.

–1 You did not specify *item*.

1 For all other cases.

DropHighlighted

The *DropHighlighted* method retrieves the item that is highlighted as a drag-and-drop target.

Return value:
The number of the selected item, or –1 in all other cases.

FirstVisible

The *FirstVisible* method retrieves the number of the first item visible in a list or report view.

Return value:
The number of the first item visible, or 0 if the list view control is in icon or small-icon view.

NextSelected

The *NextSelected* method retrieves the selected item that follows, or is to the right of, *item*.

Arguments:
The only argument is:

item The number of the item at which the search is to start. The specified item itself is excluded from the search.

Return value:
The number of the selected item, or –1 in all other cases.

PreviousSelected

The *PreviousSelected* method retrieves the selected item that precedes, or is to the left of, *item*.

Arguments:
The only argument is:

item The number of the item at which the search is to start. The specified item itself is excluded from the search.

Return value:
The number of the selected item, or –1 in all other cases.

Next

```
►►—aListControl~Next(item)————————————————————————————————————————►◄
```

The *Next* method retrieves the item that follows, or is to the right of, *item*.

Arguments:
The only argument is:

item The number of the item at which the search is to start.

Return value:
The number of the following item, or –1 in all other cases.

Previous

```
►►—aListControl~Previous(item)————————————————————————————————————►◄
```

The *Previous* method retrieves the item that precedes, or is to the left of, *item*.

Arguments:
The only argument is:

item The number of the item at which the search is to start.

Return value:
The number of the previous item, or –1 in all other cases.

NextLeft

```
►►—aListControl~NextLeft(item)————————————————————————————————————►◄
```

The *NextLeft* method retrieves the item left to *item*.

Arguments:
The only argument is:

item The number of the item at which the search is to start. The specified item itself is excluded from the search.

Return value:
The number of the next item to the left, or –1 in all other cases.

NextRight

```
►►—aListControl~NextRight(item)———————————————————————————————————►◄
```

The *NextRight* method retrieves the item right to *item*.

Arguments:

The only argument is:

item The number of the item at which the search is to start. The specified item itself is excluded from the search.

Return value:

The number of the next item to the right, or –1 in all other cases.

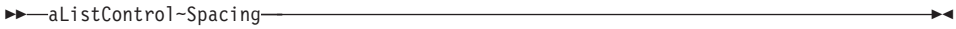
SmallSpacing

The *SmallSpacing* method determines the spacing between items in a small-icon list view control.

Return value:

The amount of spacing between the items.

Spacing

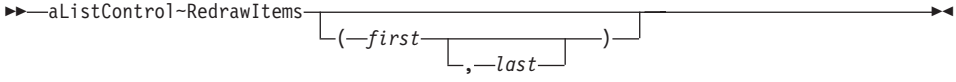


The *Spacing* method determines the spacing between items in an icon list view control.

Return value:

The amount of spacing between the items.

RedrawItems



The *RedrawItems* method forces a list view control to redraw a range of items.

Arguments:

The arguments are:

first The number of the first item to be redrawn. The default is 0.

last The number of the last item to be redrawn. The default is 0.

Return value:

0 The specified range of items was redrawn.

1 For all other cases.

UpdateItem

The *UpdateItem* method updates a list view item.

Arguments:

The only argument is:

item The number of the item to be updated.

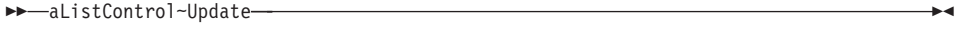
Return value:

0 The item was updated.

–1 You did not specify *item*.

1 For all other cases.

Update

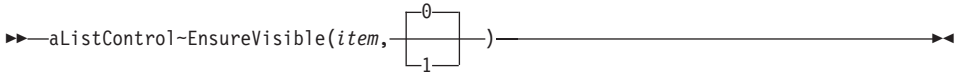


The *Update* method updates a list view control.

Return value:

0.

EnsureVisible



The *EnsureVisible* method ensures that a list view item is entirely or partially visible by scrolling the list view control, if necessary.

Arguments:

The arguments are:

item The number of the item visible.

partial Specifies whether the item must be entirely visible:

1 The list view control is not scrolled if the item is at least partially visible.

0 The list view control is scrolled if the item is only partially visible. This is the default.

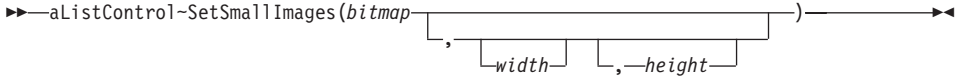
Return value:

0 The item is visible.

–1 You did not specify *item*.

1 For all other cases.

SetSmallImages



The *SetSmallImages* method assigns an image list to a small-icon list view control.

Arguments:

The arguments are:

bitmap
The name of, or handle to, a bitmap file that has already been loaded using the LoadBitmap method.

width The width of each image, in pixels. If you do not specify this argument or specify 0, the width of the image in the image file is used.

height The height of each image, in pixels. If you do not specify this argument or specify 0, the height of the image in the image file is used.

Return value:

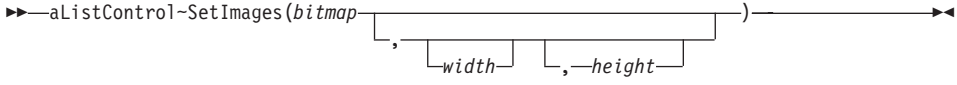
The handle to the image list, or –1 if you did not specify *bitmap*, or 0 in all other cases.

Example:

The following example connects a bitmap file with a small-icon list:

```
::method InitSmallIconList
  curList = self~GetListControl(104)
  curList~SetSmallImages("oodlist.BMP",16,12)
```

SetImages



The *SetImages* method assigns an image list to an icon list view control.

Arguments:

The arguments are:

bitmap
The name of, or handle to, a bitmap file that has already been loaded using the LoadBitmap method.

width The width of each image, in pixels. If you do not specify this argument or specify 0, the width of the image in the image file is used.

height The height of each image, in pixels. If you do not specify this argument or specify 0, the height of the image in the image file is used.

Return value:

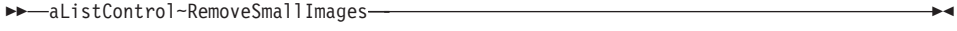
The handle to the image list, or –1 if you did not specify *bitmap*, or 0 in all other cases.

Example:

The following example connects a bitmap file with an icon list:

```
::method InitIconList
  curList = self~GetListControl(104)
  curList~SetImages("oodlist.BMP",16,12)
```

RemoveSmallImages



The *RemoveSmallImages* method erases an image list of a small-icon list view control.

Return value:

0 The image list was erased.

1 For all other cases.

RemoveImages

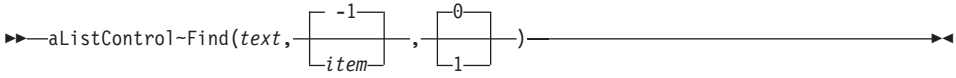
The *RemoveImages* method erases an image list of an icon list view control.

Return value:

0 The image list was erased.

1 For all other cases.

Find



The *Find* method searches for a list view item containing *text*. The text of this item must exactly match *text*.

Arguments:

The arguments are:

text The text of the item to be searched for.

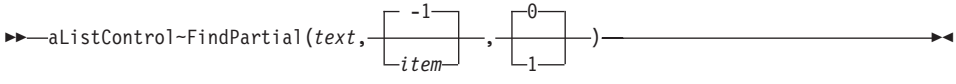
item Specify the number of the item at which the search is to be started. Specify –1 or omit this argument to start the search at the beginning.

wrap Specify 1 if the search is to be continued at the beginning if no match is found. Specify 0 or omit this argument if the search is to stop at the end of the list.

Return value:

The number of the item, or –1 in all other cases.

FindPartial



The *FindPartial* method searches for a list view item containing *text*. An item matches if its text begins with *text*.

Arguments:

The arguments are:

text The text of the item to be searched for.

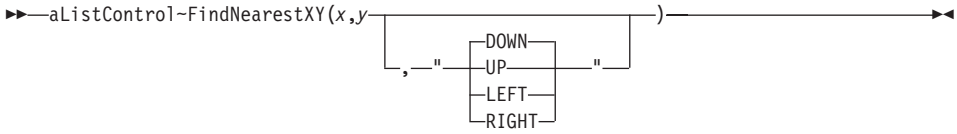
item Specify the number of the item at which the search is to be started. Specify –1 or omit this argument to start the search at the beginning.

wrap Specify 1 if the search is to be continued at the beginning if no match is found. Specify 0 or omit this argument if the search is to stop at the end of the list.

Return value:

The number of the item, or –1 in all other cases.

FindNearestXY



The *FindNearestXY* method searches, in the specified direction, for the item nearest to the specified position.

Arguments:

The arguments are:

x The x-coordinate of the position at which the search is to be started.

y The y-coordinate of the position at which the search is to be started.

direction

The direction in which the search should proceed.

Return value:

The index of the item, or –1 in all other cases.

Arrange

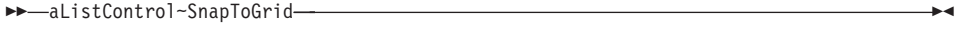
The *Arrange* method aligns items according to the current alignment style of the list view control.

Return value:

0 The items were aligned.

1 For all other cases.

SnapToGrid



The *SnapToGrid* method snaps all icons to the nearest grid position.

Return value:

0 The items were snapped.

1 For all other cases.

AlignLeft

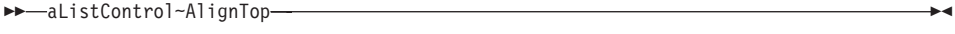
The *AlignLeft* method aligns items along the left window border.

Return value:

0 The items were aligned.

1 For all other cases.

AlignTop



The *AlignTop* method aligns items along the upper window border.

Return value:

0 The items were aligned.

1 For all other cases.

ItemPos

The *ItemPos* method retrieves the position of the upper left corner of the item.

Arguments:

The only argument is:

item The number of the item.

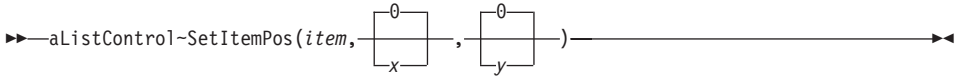
Return value:

The x- and y-coordinates of the upper left corner of the item, or –1 if you did not specify *item*, or 0 in all other cases.

Note: Use "DefListDragHandler" on page 321 to support default dragging:

```
self~ConnectListNotify(104,"BEGINDRAG","DefListDragHandler")
```

SetItemPos



The *SetItemPos* method moves an item to a specified position in a list view control, which must be in icon or small-icon view.

Arguments:

The arguments are:

item The number of the item.

x The x-coordinate of the new position of the upper left corner of the item, in view coordinates. The default is 0.

y The y-coordinate of the new position of the upper left corner of the item, in view coordinates. The default is 0.

Return value:

0 The item was moved.

–1 You did not specify *item*.

1 For all other cases.

Note: Use "DefListDragHandler" on page 321 to support default dragging:

```
self~ConnectListNotify(104,"BEGINDRAG","DefListDragHandler")
```

Edit

The *Edit* method begins editing of the text of the specified list view item.

Arguments:

The only argument is:

item The number of the item.

Return value:

The handle of the edit control used to edit the item text, or 0 in all other cases.

EndEdit

The *EndEdit* method cancels editing of the list view item that is being edited.

SubclassEdit

The *SubclassEdit* method is used by the DefListEditHandler to correct an operating system problem if the Esc or Enter key was pressed in an active edit control.

RestoreEditClass

The *RestoreEditClass* method is used by the DefListEditHandler to correct an operating system problem if the Esc or Enter key was pressed in an active edit item.

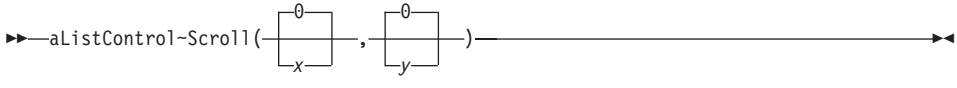
ItemsPerPage

The *ItemsPerPage* method calculates the number of items that vertically fit the visible area of a list view control that is in list or report view. Only fully visible items are counted.

Return value:

The number of fully visible items. If the current view is an icon or small-icon view, the return value is the total number of items in the list view control.

Scroll



The *Scroll* method scrolls the content of a list view control.

Arguments:

The arguments are:

x An integer value specifying the amount of horizontal scrolling. If the control is in icon, small-icon, or report view, this value specifies the number of pixels to be scrolled. If it is in list view, this value specifies the number of columns to be scrolled. The default value is 0.

y An integer value specifying the amount of vertical scrolling. If the control is in icon, small-icon, or report view, this value specifies the number of pixels to be scrolled. If it is in list view, this value specifies the number of lines to be scrolled. The default value is 0.

Return value:

0 Scrolling was successful.

1 Scrolling failed.

BkColor

The *BkColor* method retrieves the background color for a list view control.

Return value:

The color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

BkColor=

The *BkColor*= method sets the background color of a list view control.

Arguments:

The only argument is:

color The new background color. Specify the color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

Example:

The following example sets the background color of a list control to yellow:

```
::method Yellow
  curList = self~GetListControl(104)
  curList~BkColor = 15
  curList~Update
```

TextColor

The *TextColor* method retrieves the text color of a list view control.

Return value:
The color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

TextColor=

The *TextColor=* method sets the text color of a list view control.

Arguments:
The only argument is:

color The new text color. Specify the color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

Example:
The following example sets the text color of a list control to light blue:

```
::method LightBlue
  curList = self~GetListControl(104)
  curList~BkColor = 9
  curList~Update
```

TextBkColor

The *TextBkColor* method retrieves the background color of the text in a list view control.

Return value:
The color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

TextBkColor=

```
►►—aListControl~TextBkColor=(color)———————————————————————————————►◄
```

The *TextBkColor=* method sets the background color for the text in a list view control.

Arguments:

The only argument is:

color The new background color for text. Specify the color-palette index specifier (0 to 18). For more information on color palettes, refer to "Chapter 7. Definition of Terms" on page 87.

Notification Messages

The list view control sends notification messages to notify about events. For more information on notification messages, refer to "ConnectListNotify" on page 318.

The following example shows how to connect the list view notification messages with the corresponding message:

```
::method Init
  use arg InitStem.
  if Arg(1,"o") = 1 then
     InitRet = self~Init:super
  else
     InitRet = self~Init:super(InitStem.)

  if self~Load("list.rc", ) \= 0 then do
     self~InitCode = 1
    return
  end

  /* Connect dialog control items to class methods */
  self~ConnectListNotify("IDC_LIST","Changing","OnChanging_IDC_LIST")
  self~ConnectListNotify("IDC_LIST","Changed","OnChanged_IDC_LIST")
  self~ConnectListNotify("IDC_LIST","DefaultEdit")
  self~ConnectListNotify("IDC_LIST","Delete","OnDelete_IDC_LIST")
  self~ConnectListNotify("IDC_LIST","KeyDown","OnKeyDown_IDC_LIST")
  self~ConnectButton("IDC_PB_NEW","IDC_PB_NEW")
  self~ConnectButton("IDC_PB_DELETE","IDC_PB_DELETE")
  self~ConnectButton(2,"Cancel")
  self~ConnectButton(9,"Help")
  self~ConnectButton(1,"OK")
  return InitRet
```

Chapter 21. ButtonControl Class

The *ButtonControl* class provides methods to query and modify push button controls. It inherits all methods of the DialogControl class (see page 181).

Use the GetButtonControl method (see page 340) to retrieve an object of the *ButtonControl* class.

Requires:

The *ButtonControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *ButtonControl* class implement the methods listed in Table 9.

Table 9. ButtonControl Instance Methods

Method...	...on page
ChangeBitmap	419
DimBitmap	425
DisplaceBitmap	420
DrawBitmap	424
GetBitmapSizeX	423
GetBitmapSizeY	423
GetBmpDisplacement	421
Scroll	421
ScrollBitmapFromTo	425
ScrollText	422
State	416
State=	416
Style=	417

State

The *State* method retrieves the current state of the associated button control.

Return value:

A text string that can contain one or more of the following keywords, separated by blanks:

"CHECKED"
The radio button is selected or the check box is checked.

"UNCHECKED"
The radio button is not selected or the check box is unchecked.

"INDETERMINATE"
The Auto-3–State button is neither checked nor unchecked, but grayed.

"PUSHED"
The cursor is positioned on the button and the left mouse button is pressed and held.

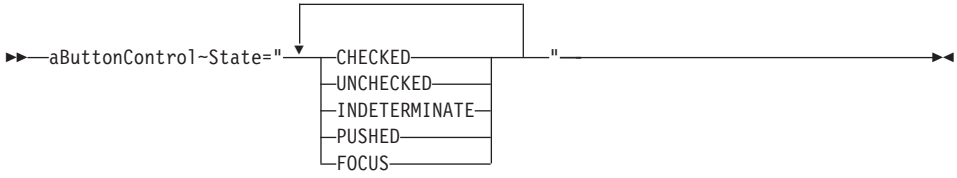
"FOCUS"
The button has the keyboard focus.

Example:

```
button = MyDialog~GetButtonControl("IDOK")
if button == .Nil then return
say button~State
```

The result could be "UNCHECKED FOCUS".

State=



The *State*= method sets the state for the associated button control.

Arguments:

The only argument is:

new_State

A text string that contains one or more of the following keywords, separated by a blank:

CHECKED

The radio button is to be selected or the check box is to be checked.

UNCHECKED

The radio button is not to be selected or the check box is to be unchecked.

INDETERMINATE

The Auto-3–State button is to be set to the grayed state.

PUSHED

The button is to be set to the "pushed" state.

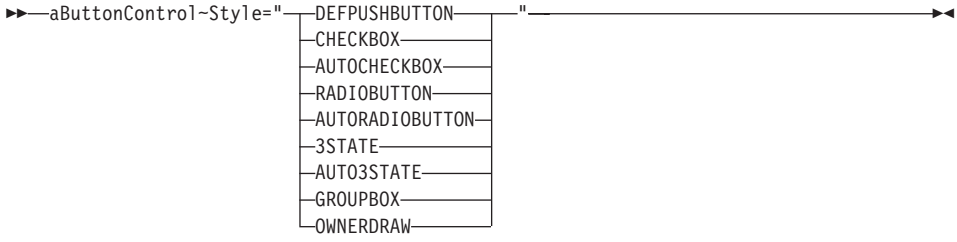
FOCUS

The button is to be set to the "focused" state.

Example:

```
button = MyDialog~GetButtonControl("IDOK")
if button == .Nil then return
button~State="FOCUS PUSHED"
```

Style=



The *Style*= method changes the style of the associated button control.

Arguments:

The only argument is:

new_Style

A text string that contains one of the following keywords:

DEFPUSHBUTTON
A default push button that is pushed when the Enter key is pressed.

CHECKBOX
A check box the state of which has to be maintained by the program.

AUTOCHECKBOX
A check box the check state of which toggles between checked and unchecked each time the user selects the check box.

RADIOBUTTON
A radio button the state of which has to be maintained by the program.

AUTORADIOBUTTON
A radio button that sets the check state of the button to checked and the check state for all other buttons in the same group to unchecked each time the user selects this radio button.

3STATE
A button that is equal to a check box except that the check box can be grayed as well as checked or unchecked.

AUTO3STATE
A button that is equal to a three-state check box except that the check box changes its state when the user selects it. The state cycles through checked, grayed, and unchecked.

GROUPBOX
A rectangle in which other controls can be grouped. A label is displayed in the upper left corner of the rectangle.

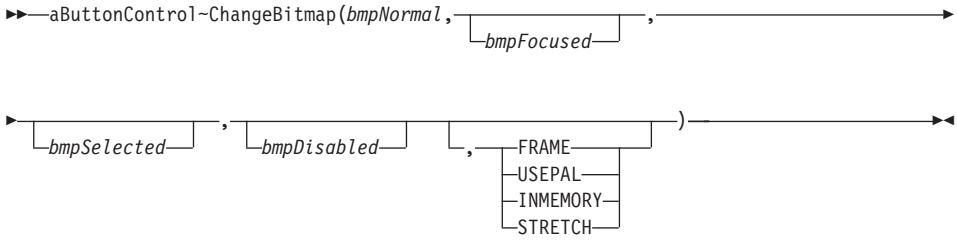
OWNERDRAW
An owner-drawn button that can be used to display a bitmap or graphics. If none of the previous values is specified, the push-button style is set for the associated button control.

Example:

The following example makes the OK button the default button:

```
button = MyDialog~GetButtonControl("IDOK")
if button == .Nil then return
button~Style="DEFPUSHBUTTON"
```

ChangeBitmap



The *ChangeBitmap* method changes the bitmap of a bitmap button.

Arguments:

The arguments are:

bmpNormal

The (alphanumeric) name, (numeric) resource ID, or handle of a bitmap that is displayed when the button is neither selected, nor focused, nor disabled. If you specify the bitmap handle, the INMEMORY option must be specified.

This option is used if none of the other arguments is specified.

bmpFocused

The (alphanumeric) name, (numeric) resource ID, or handle of a bitmap that is displayed when the button is focused. The focused button is activated when the Enter key is pressed. If you specify the bitmap handle, the INMEMORY option must be specified.

bmpSelected

The (alphanumeric) name, (numeric) resource ID, or handle of a bitmap that is displayed when the button is clicked and held. If you specify the bitmap handle, the INMEMORY option must be specified.

bmpDisabled

The (alphanumeric) name, (numeric) resource ID, or handle of a bitmap that is displayed when the button is disabled. If you specify the bitmap handle, the INMEMORY option must be specified.

styleOptions

The last argument can be one of the following:

FRAME

Draws a frame around the button. When you use this

option, the bitmap button behaves like a normal Windows button except that a bitmap is shown instead of text.

USEPAL
Takes the colors of the bitmap file and stores them as the system color palette. This option is needed when the bitmap was created with a palette other than the default Windows color palette. Use it for one button only because only one color palette can be active at a time.

This option is not valid for a bitmap loaded from a dynamic-link library.

INMEMORY
This option must be used if the named bitmaps are already loaded into memory by using the LoadBitmap method (see page 208). In this case, you must specify a bitmap handle instead of a file name or ID.

STRETCH
If this option is specified and the extent of the bitmap is smaller than the extent of the button rectangle, the bitmap is adapted to match the extent of the button. This option has no effect on bitmaps loaded from a dynamic-link library.

Example:

```
button = MyDialog~GetButtonControl("IDOK")
if button == .Nil then return
button~ChangeBitmap("AddBut_n.bmp", "AddBut_f.bmp", "AddBut_s.bmp",,
"AddBut_d.bmp", "FRAME")
```

See also “ConnectBitmapButton” on page 112.

DisplaceBitmap

```
►►—aButtonControl~DisplaceBitmap(x,y)————————————————————————————►◄
```

The *DisplaceBitmap* method sets the position of a bitmap within a bitmap button.

Arguments:
The arguments are:

x The horizontal displacement, in screen pixels. A negative value can also be used.

y The vertical displacement, in screen pixels. A negative value can also be used.

Example:

The following example moves the bitmap within the associated bitmap button 4 screen pixels to the right and 3 pixels upward:

```
button = MyDialog~GetButtonControl("IDOK")
if button == .Nil then return
parse value button~GetBmpDisplacement with dx dy
button~DisplacementBitmap(244, dx+4, dy-3)
```

GetBmpDisplacement

The *GetBmpDisplacement* method retrieves the position of a bitmap within a bitmap button.

Return value:

The horizontal and vertical positions of the bitmap, in screen pixels and separated by a blank.

Example:

See "DisplaceBitmap" on page 420.

Scroll

The *Scroll* method moves the rectangle within the associated button and redraws the uncovered area with the button background color. It is used to move bitmaps within bitmap buttons.

Arguments:

The arguments are:

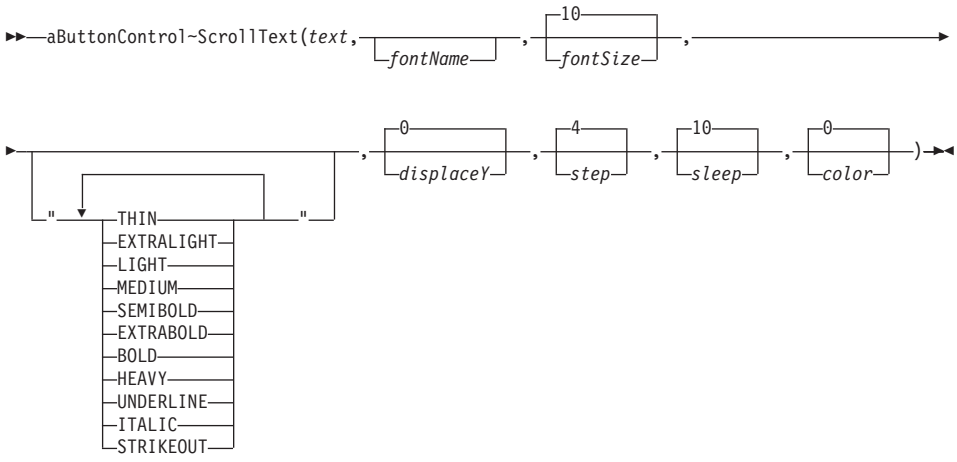
xPos, yPos

The new position of the rectangle, in screen pixels.

left, top, right, bottom

The upper left and lower right corner of the rectangle to be moved.

ScrollText



The *ScrollText* method scrolls text in the associated button with the given font, size, and color. The text is scrolled from right to left. If the method is started concurrently, call it a second time to stop scrolling. The associated button must have the OWNERDRAWN style.

Arguments:

The arguments are:

text A text string that is displayed and scrolled.

fontName
The name of the font used to write the text. If omitted, the system font is used.

fontSize
The size of the font used to write the text. If omitted, the standard size (10) is used.

fontStyle
This argument can be one or more of the keywords listed in the syntax diagram. If you use more than one keyword, put them in one string, separated by blanks.

displaceY
The vertical displacement of the text relative to the top of the client area of the window. The default is 0.

step The amount of screen pixels that the text is moved in each cycle. The default is 4.

sleep The time, in milliseconds, that the program waits after each movement to determine the scrolling speed. The default is 10.

color The color index used for the text. The default is 0, which is black.

Example:

The following example scrolls the string ″Hello world!″ from left to right within the associated button. The text is located 2 pixels below the top of the client area, one move is 3 screen pixels, and the delay time after each movement is 15 ms.

```
button = MyDialog~GetButtonControl("IFOK")
if button == .Nil then return
button~ScrollText("Hello world!", "Arial", 36, "BOLD ITALIC", 2, 3, 15, 4)
```

GetBitmapSizeX

The *GetBitmapSizeX* method retrieves the width of the bitmap that is set for the associated button.

Return value:

The width of the bitmap, in screen pixels.

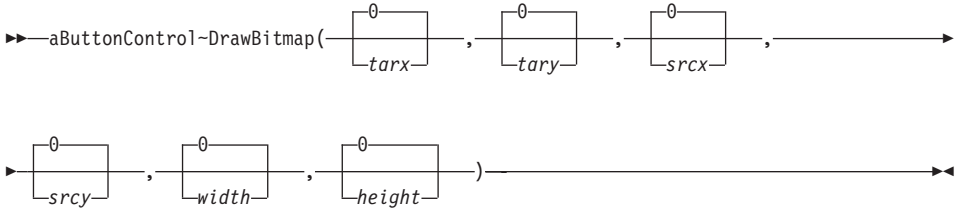
GetBitmapSizeY

The *GetBitmapSizeY* method retrieves the height of the bitmap that is set for the associated button.

Return value:

The height of the bitmap, in screen pixels.

DrawBitmap



The *DrawBitmap* method draws the bitmap of the associated bitmap button. You can also use this method to move a bitmap or part of it.

Contrary to the method "DisplaceBitmap" on page 420, which sets a permanent position for the bitmap, DrawBitmap immediately draws the bitmap, or part of it, at the specified position. If the button must be refreshed, the bitmap is drawn at the position set with DisplaceBitmap. DrawBitmap is used by ScrollBitmapFromTo, for example.

Arguments:

The arguments are:

tarx,tary
The position relative to the client area of the button where the bitmap is to be displayed. The default is 0,0.

srcx,srcy
The offsets that specify the first pixel in the bitmap to be displayed. If you omit these arguments, the pixel at position 0,0 is the first pixel displayed.

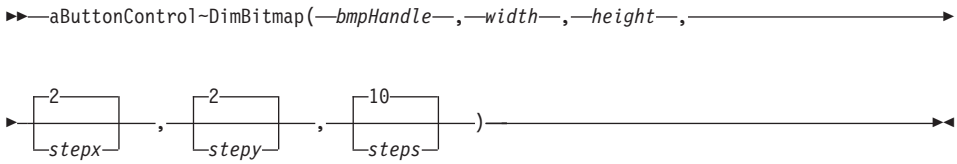
width,height
The width and height of the bitmap. If you omit these arguments or specify 0, the entire bitmap is displayed at the specified position. You can use these arguments to display only parts of the bitmap.

Return value:

0 if the bitmap could be drawn.

Note: You can use the DrawBitmap method to animate a bitmap by providing a bitmap that contains several images and use the offset and extension arguments to display a single image of the bitmap.

DimBitmap



The *DimBitmap* method draws a bitmap step by step.

Arguments:

The arguments are:

bmpHandle
A handle to the bitmap loaded with "LoadBitmap" on page 208.

width, height
The extensions of the bitmap.

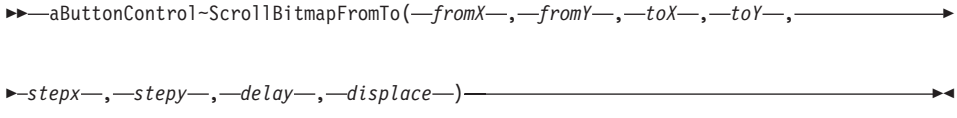
stepx, stepy
The number of incremental pixels displayed at each step. The default is 2,2.

steps The number of iterations used to display the bitmap. The default is 10.

Return value:

This method does not return a value.

ScrollBitmapFromTo



The *ScrollBitmapFromTo* method scrolls the bitmap within the associated bitmap button from one position to another.

For an explanation of the arguments, refer to "ScrollBitmapFromTo" on page 164.

Chapter 22. RadioButton Class

The *RadioButton* class provides methods to query and modify radio button controls. It inherits all methods of the ButtonControl and DialogControl classes (see pages 415 and 181).

Use the GetRadioControl (see page 341) to retrieve an object of the *RadioButton* class.

Requires:

The *RadioButton* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *RadioControl* class implement the methods listed in Table 10.

Table 10. RadioButton Instance Methods

Method...	...on page
Check	428
Indeterminate	428
IsChecked	427
Uncheck	428

IsChecked

```
►►—aRadioButton~IsChecked————————————————————————————————————►◄
```

The *IsChecked* method retrieves the current state of the associated button control.

Return value:

One of the following values:

"CHECKED"

The radio button is selected or the check box is checked.

"UNCHECKED"

The radio button is not selected or the check box is unchecked.

"INDETERMINATE"
The Auto–3–State button is grayed to indicate an indeterminate state.

"UNKNOWN"
No information on the current state available.

Check

The *Check* method marks the associated button control as checked.

Uncheck

The *Uncheck* method deletes the check mark of the button control.

Indeterminate

The *Indeterminate* method grays the check box of an Auto–3–State button to indicate an indeterminate state.

Chapter 23. CheckBox Class

The *CheckBox* class provides methods to query and modify check box controls. It inherits all methods of:

- The RadioButton class (see page 427)
- The ButtonControl class (see page 415)
- The DialogControl class (see page 181)

The *CheckBox* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Use the GetCheckControl method (see page 342) to retrieve an object of the *CheckBox* class.

Chapter 24. ListBox Class

The *ListBox* class provides methods to query and modify list box controls. It inherits all methods of the DialogControl class (see page 181).

Use the GetListControl method (see page 346) to retrieve an object of the *ListBox* class.

Requires:

The *ListBox* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *ListBox* class implement the methods listed in Table 11.

Table 11. ListBox Instance Methods

Method...	...on page
Add	432
AddDirectory	441
ColumnWidth	388
Delete	433
DeleteAll	433
DeSelectIndex	435
DeselectRange	437
Find	433
GetFirstVisible	439
GetText	439
Insert	432
ItemHeight	442
ItemHeight=	443
Items	437
MakeFirstVisible	438
Modify	439
Select	435
Selected	434
SelectedIndex	434

Table 11. ListBox Instance Methods (continued)

Method...	...on page
SelectedIndexes	438
SelectedItems	438
SelectIndex	435
SelectRange	436
SetTabulators	440
SetWidth	442
Width	442

Add

The *Add* method adds a new item to the list. If the list is not sorted, the new item is added to the end of the list.

Arguments:

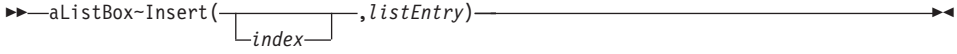
The only argument is:

listEntry
A text string added to the list.

Return value:

A one-based index that specifies the position at which the entry has been added, or 0 or a value less than 0 to indicate an error.

Insert



The *Insert* method inserts a new item into the list after the specified item.

Arguments:

The arguments are:

index The index (starting with 1) of the list item after which the new item is to be added. If this argument is omitted, the list entry is added after the currently selected item.

listEntry
A text string added to the list.

Return value:
A one-based index that specifies the position at which the entry has been added, or 0 or a value less than 0 to indicate an error.

Delete

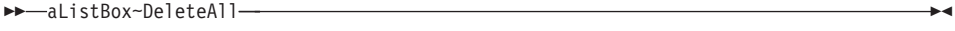
The *Delete* method removes a list item from the associated list box.

Arguments:
The only argument is:

index The index (starting with 1) of the list item to be removed from the list. If this argument is omitted, the currently selected item is deleted.

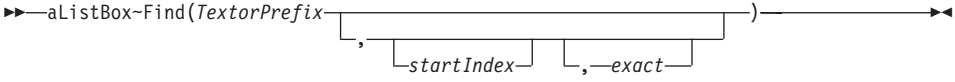
Return value:
The number of remaining list items, or 0 to indicate an error.

DeleteAll



The *DeleteAll* method removes all list items from the associated list box.

Find



The *Find* method searches the list box for a list entry containing the specified text or prefix. The search is caseless.

Arguments:
The arguments are:

TextorPrefix
The text or prefix for which the list is searched.

startIndex
The first list item at which the search is to be started. When the search reaches the bottom of the list, it is continued backward. If you omit this argument or specify 0, the entire list is searched.

exact If you specify 1 or E, the text of the list item must exactly match the text specified for *TextorPrefix*. If you omit this argument or specify 0, the list entries are searched for a prefix that matches *TextorPrefix*.

Return value:
The one-based index of the list entry that matches the search text, or 0 if not found.

SelectedIndex

```
►►—aListBox~SelectedIndex————————————————————————————————————►◄
```

The *SelectedIndex* method retrieves the index of the currently selected list entry. This entry is highlighted and surrounded by a dotted border. To get the selected items for a multiple selection listbox, use GetMultiList; see "GetMultiList" on page 128.

Return value:
The one-based index of the currently selected list entry, or 0 if none is selected.

Selected

```
►►—aListBox~Selected—————————————————————————————————————————►◄
```

The *Selected* method retrieves the text of the currently selected list entry.

Return value:
The text of the currently selected list entry, or an empty string if none is selected.

SelectIndex

The *SelectIndex* method selects the list entry at the specified position. The currently selected list item is highlighted and surrounded by a dotted border. To select multiple items of a multiple selection listbox, use SetMultiList; see "SetMultiList" on page 128.

Arguments:

The only argument is:

index The index (starting with 1) of the list item to be selected. If you specify 0 for this argument, the list box must not contain any selection.

Return value:

0 if an error occurred or you specified 0 for *index* to remove the selection.

DeSelectIndex

The *DeSelectIndex* method deselects the list entry at the specified position for multiple selection listboxes. The currently selected list item is highlighted and surrounded by a dotted border.

Arguments:

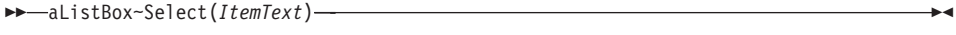
The only argument is:

index The index (starting with 1) of the list item to be deselected. If you specify no index or 0 for this argument, all items are deselected.

Return value:

–1 if an error occurred.

Select



The *Select* method selects the list entry that matches the specified text.

Arguments:

The only argument is:

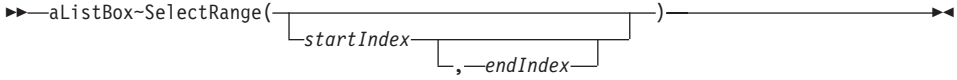
ItemText

The text that the list box is searched for.

Return value:

0 if an error occurred or a matching list entry was not found.

SelectRange



The *SelectRange* method selects one or more consecutive items of the associated multiselection list box.

Arguments:

The arguments are:

startIndex

The position of the first list item to be selected. If this argument is omitted, the selection range starts with the first item in the list.

endIndex

The position of the last list item to be selected. If this argument is omitted, the selection range ends with the last item in the list.

If *startIndex* is equal to *endIndex*, a single item of the list is selected.

Return value:

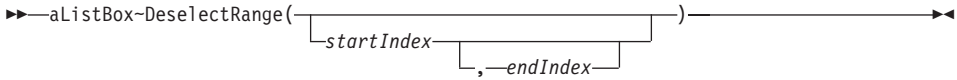
–1 if an error occurred. This can happen, for example, if the list box is no multiselection list box.

Example:

The following example selects the items 5 through 9:

```
lb = self~GetListBox("Offers")
if lb == .Nil then return
lb~SelectRange(5,9)
```

DeselectRange



The *DeselectRange* method removes the selection for one or more consecutive items of the associated multiselection list box.

Arguments:

The arguments are:

startIndex

The position of the first list item to be deselected. If this argument is omitted, the deselection range starts with the first item in the list.

endIndex

The position of the last list item to be deselected. If this argument is omitted, the deselection range ends with the last item in the list.

If *startIndex* is equal to *endIndex*, a single item of the list is deselected.

Return value:

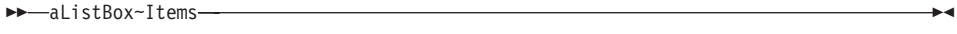
–1 if an error occurred. This can happen, for example, if the list box is no multiselection list box.

Example:

The following example deselects the items 1 through 5:

```
lb = self~GetListBox("Offers")
if lb == .Nil then return
lb~DeselectRange(,6)
```

Items



The *Items* method retrieves the number of items in the list box.

Return value:

The number of items in the list box.

SelectedItems

```
►►—aListBox~SelectedItems————————————————————————————————————►◄
```

The *SelectedItems* method retrieves the number of items that are currently selected in the associated multiselection list box.

Return value:

The number of selected items, or –1 if this method fails, for example if the list box is no multiselection list box.

Example:

For an example, refer to "SelectedIndexes".

SelectedIndexes

```
►►—aListBox~SelectedIndexes———————————————————————————————————►◄
```

The *SelectedIndexes* method retrieves the indexes of all items that are currently selected in the associated multiselection list box.

Return value:

A text string containing the indexes of the selected items in the list, separated by blanks.

Example:

The following example lists all items selected in the associated multiselection list box:

```
lb = self~GetListBox("Offers")
if lb == .Nil then return
sit = lb~SelectedItems
if sit > 0 then do
   say "You ordered:"
   sndx = lb~SelectedIndexes
   do i = 1 to sit
      parse var sndx order sndx
      say "1 x" lb~GetText(order)
   end
end
```

MakeFirstVisible

```
►►—aListBox~MakeFirstVisible(index)———————————————————————————►◄
```

The *MakeFirstVisible* method makes the list entry at the specified position the first visible list item when you scroll up or down.

Arguments:

The only argument is:

index The one-based index of the list box item to be made first visible.

Return value:

–1 if an error occurred.

GetFirstVisible

The *GetFirstVisible* method retrieves the index of the first list item visible in the list box.

Return value:

The one-based index of the list box item visible first.

GetText

The *GetText* method gets the text of the list item at the specified position in the list box.

Arguments:

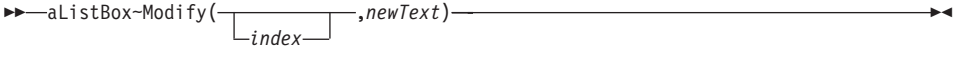
The only argument is:

index The one-based index of the list box item containing the text you are interested in.

Return value:

The text of the list box item at the given position, or an empty string if the *index* does not refer to an item or an error occurred.

Modify



The *Modify* method changes the text of the list item at the specified position in the list box.

Arguments:

The arguments are:

index The one-based index of the list box item of which the text is to be changed. If you omit this argument, the currently selected item is modified.

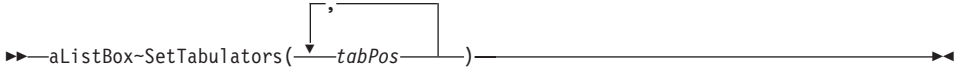
newText

The new text string to be displayed at the given position.

Return value:

The one-based index of the modified list box item at the given position. The return value is 0 if an error occurred, or –1 if the *index* does not refer to an item.

SetTabulators



The *SetTabulators* method sets the tabulators for the associated list box control. This enables you to use items containing tab characters ('09'x), which is useful if you want to format the list in more than one column when using proportional fonts.

Arguments:

The only argument is:

tabPos

The position or positions of the tabs relative to the left edge of the list box, in dialog units.

Return value:

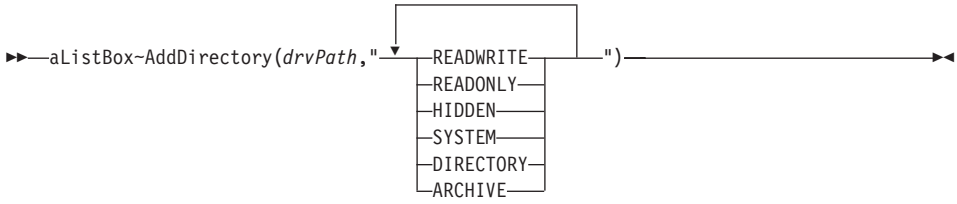
1 if an error occurred.

Example:

The following example creates a list that can handle up to three tabulators in a list entry. The tabulator positions are 10, 20, and 30.

```
lb = MyDialog~GetListBox(102)
if lb == .Nil then return
lb~SetTabulators(10, 20, 30)
lb~Add(textcol1 || '09'x || textcol2 || '09'x || textcol3 || '09'x ||,
   textcol4)
```

AddDirectory



The *AddDirectory* method adds all or selected file names of a given directory to the list box.

Arguments:

The arguments are:

drvpath

The drive, path, and name pattern.

fileAttributes

The attributes that the files must have in order to be added:

READWRITE

Normal read/write files (same as none).

READONLY

Files that have the read-only bit.

HIDDEN

Files that have the hidden bit.

SYSTEM

Files that have the system bit.

DIRECTORY

Files that have the directory bit.

ARCHIVE

Files that have the archive bit.

Return value:

The one-based index of the file added last to the list, or 0 if an error occurred.

Example:

The following example puts the names of all read/write files with extension .REX in the given directory of the list box:

```
MyDialog~AddDirectory(203, drive":\"path"\*.rex", "READWRITE")
```

SetWidth

The *SetWidth* method sets the internal width of the list box, in dialog units. If the internal width exceeds the width of the list box control and the list box has the HSCROLL style, the list box provides a horizontal scroll bar.

Arguments:

The only argument is:

width The width of the list box, in dialog units.

Example:

The following example sets the internal width twice the width of the list box control:

```
lb = MyDialog~GetListBox(102)
if lb == .Nil then return
lb~SetWidth(lb~SizeX*2)
```

Width

The *Width* method retrieves the internal width of the associated list box.

Return value:

The internal width of the list box, in dialog units, or 0 if an error occurred.

ItemHeight

The *ItemHeight* method retrieves the height of the items in the list box, in dialog units.

Return value:

The height of the list box items, in dialog units.

ItemHeight=

The *ItemHeight*= method sets the height of the items in the list box, in dialog units.

Arguments:

The only argument is:

height The height of all list box items, in dialog units.

ColumnWidth=

The *ColumnWidth*= method sets the width of the list box columns in a multicolumn list box control.

Arguments:

The only argument is:

width The width of all list box columns, in dialog units.

Chapter 25. ComboBox Class

The *ComboBox* class provides methods to query and modify combo box controls. It inherits all methods of the DialogControl class (see page 181).

Use the GetComboBox method (see page 343) to retrieve an object of the *ComboBox* class.

Requires:

The *ComboBox* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *ComboBox* class implement the methods listed in Table 12.

Table 12. ComboBox Instance Methods

Method...	...on page
Add	446
AddDirectory	450
CloseDropDown	451
Delete	446
DeleteAll	447
EditSelection	452
Find	447
GetText	449
Insert	446
IsDropDownOpen	452
Items	449
Modify	450
OpenDropDown	451
Select	449
SelectIndex	448
Selected	448
SelectedIndex	448

Add

The *Add* method adds a new item to the list of the combo box. If the list is not sorted, the new item is added to the end of the list.

Arguments:

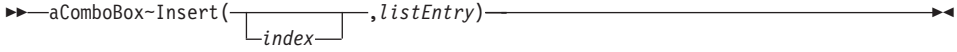
The only argument is:

listEntry
A text string added to the list.

Return value:

A one-based index that specifies the position at which the entry has been added, or 0 or a value less than 0 to indicate an error.

Insert



The *Insert* method inserts a new item into the list of the combo box after the specified item.

Arguments:

The arguments are:

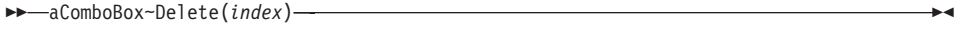
index The index (starting with 1) of the list item after which the new item is to be added. If this argument is omitted, the list entry is added after the currently selected item.

listEntry
A text string added to the list.

Return value:

A one-based index that specifies the position at which the entry has been added, or 0 or a value less than 0 to indicate an error.

Delete



The *Delete* method removes a list item from the associated combo box.

Arguments:

The only argument is:

index The index (starting with 1) of the list item to be removed from the list. If this argument is omitted, the currently selected item is deleted.

Return value:

The number of remaining list items, or 0 to indicate an error.

DeleteAll

The *DeleteAll* method removes all list items from the associated combo box.

Arguments:

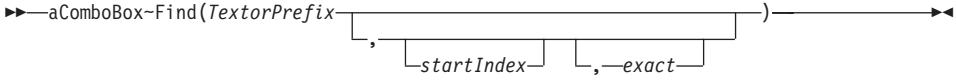
The only argument is:

index The index (starting with 1) of the list item to be removed from the list. If this argument is omitted, the currently selected item is deleted.

Return value:

The number of remaining list items, or 0 to indicate an error.

Find



The *Find* method searches the combo box for a list entry containing the specified text or prefix. The search is caseless.

Arguments:

The arguments are:

TextorPrefix

The text or prefix for which the list is searched.

startIndex

The first list item at which the search is to be started. When the search reaches the bottom of the list, it is continued backward. If you omit this argument or specify 0, the entire list is searched.

exact If you specify 1 or E, the text of the list item must exactly match the text specified for *TextorPrefix*. If you omit this argument or specify 0, the list entries are searched for a prefix that matches *TextorPrefix*.

Return value:

The one-based index of the list entry that matches the search text, or 0 if not found.

SelectedIndex

```
►►—aComboBox~SelectedIndex———————————————————————————————►◄
```

The *SelectedIndex* method retrieves the index of the currently selected entry of the combo box list. This entry is highlighted and surrounded by a dotted border.

Return value:

The one-based index of the currently selected list entry, or 0 if none is selected.

Selected

```
►►—aComboBox~Selected————————————————————————————————————►◄
```

The *Selected* method retrieves the text of the currently selected entry of the combo box list.

Return value:

The text of the currently selected list entry, or an empty string if none is selected.

SelectIndex

```
►►—aComboBox~SelectIndex(index)———————————————————————————►◄
```

The *SelectIndex* method selects the list entry at the specified position. The currently selected list item in the combo box is highlighted and surrounded by a dotted border.

Arguments:

The only argument is:

index The index (starting with 1) of the list item to be selected. If you specify 0 for this argument, the combo box must not contain any selection.

Return value:
0 if an error occurred or you specified 0 for *index* to remove the selection.

Select

The *Select* method selects the list entry that matches the specified text.

Arguments:
The only argument is:

ItemText
The text that the combo box is searched for.

Return value:
0 if an error occurred or a matching list entry was not found.

Items

The *Items* method retrieves the number of items in the combo box.

Return value:
The number of items in the combo box list.

GetText

The *GetText* method gets the text of the list item at the specified position in the combo box.

Arguments:
The only argument is:

index The one-based index of the combo box item containing the text you are interested in.

Return value:
The text of the combo box item at the given position, or an empty string if the *index* does not refer to an item or an error occurred.

Modify

The *Modify* method changes the text of the list item at the specified position in the combo box.

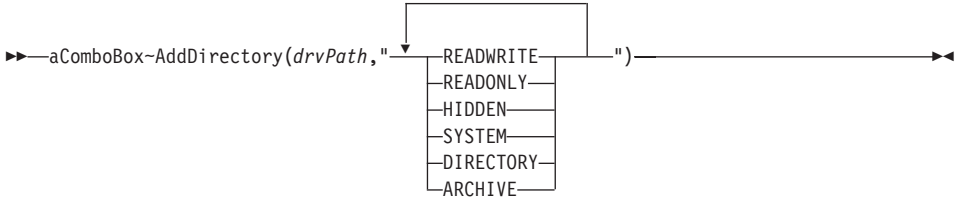
Arguments:
The arguments are:

index The one-based index of the combo box item of which the text is to be changed. If you omit this argument, the currently selected item is modified.

newText
The new text string to be displayed at the given position.

Return value:
The one-based index of the modified combo box item at the given position. The return value is 0 if an error occurred, or –1 if the *index* does not refer to an item.

AddDirectory



The *AddDirectory* method adds all or selected file names of a given directory to the combo box.

Arguments:
The arguments are:

drvpath
The drive, path, and name pattern.

fileAttributes
Specify the file attributes the files must possess in order to be added:

READWRITE
Normal read/write files (same as none).

READONLY
Files that have the read-only bit.

HIDDEN
Files that have the hidden bit.

SYSTEM
Files that have the system bit.

DIRECTORY
Files that have the directory bit.

ARCHIVE
Files that have the archive bit.

Return value:
The one-based index of the file name added last to the list, or 0 if an error occurred.

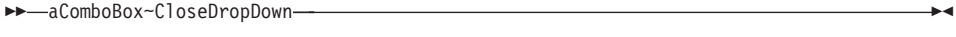
Example:
The following example puts the names of all read/write files with extension .REX in the given directory of the list box:

```
MyDialog~AddDirectory(203, drive":\"path"\*.rex", "READWRITE")
```

OpenDropDown

The *OpenDropDown* method opens the list box of the associated combo box.

CloseDropDown



The *CloseDropDown* method closes the list box of the associated combo box.

IsDropDownOpen

```
►►—aComboBox~IsDropDownOpen———————————————————————————————————►◄
```

The *IsDropDownOpen* method retrieves whether the list box of the associated combo box is open, that is, visible.

Return value:

1 The list box is open.

0 For all other cases.

EditSelection

```
►►—aComboBox~EditSelection(startNdx,endNdx)———————————————————————►◄
```

The *EditSelection* method selects the specified text range in the edit control of the associated combo box.

Arguments:

The arguments are:

startNdx

The one-based index of the first character in the edit control to be selected. If you omit this argument or specify 0, the selection is removed.

endNdx

The one-based index of the last character in the edit control to be selected. If you omit this argument or specify 0, the selection is removed.

Return value:

0 The selection was successful.

1 An error occurred.

Chapter 26. ScrollBar Class

The *ScrollBar* class provides methods to query and modify scroll bars. It inherits all methods of the DialogControl class (see page 181).

Use the GetScrollBar method to retrieve an object of the *ScrollBar* class.

Requires:

The *ScrollBar* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

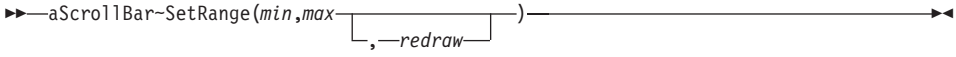
Methods:

Instances of the *ScrollBar* class implement the methods listed in Table 13.

Table 13. ScrollBar Instance Methods

Method...	...on page
DeterminePosition	455
Position	455
Range	454
SetRange	453
SetPos	454

SetRange



The *SetRange* method sets the minimum and maximum positions for the associated scroll bar.

Arguments:

The arguments are:

min The minimum position to which the scroll bar can be moved.

max The maximum position to which the scroll bar can be moved.

redraw

If you specify 1 or Y, the scroll bar is redrawn using the new

range. Otherwise, the range is set but the scroll bar display is not updated. The default value is 1.

Return value:

0 Setting the range was successful.

1 For all other cases.

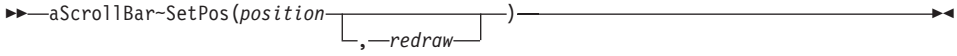
Range

The *Range* method retrieves the minimum and maximum positions of the associated scroll bar.

Return value:

The minimum and maximum positions, separated by a blank.

SetPos



The *SetPos* method sets the position of the scroll box for the associated scroll bar.

Arguments:

The arguments are:

position

The position to which the scroll box is to be moved.

redraw

If you specify 1 or Y, the scroll bar is redrawn using the new position. Otherwise, the new position is set, but the scroll bar display is not updated. The default value is 1.

Return value:

0 Setting the position was successful.

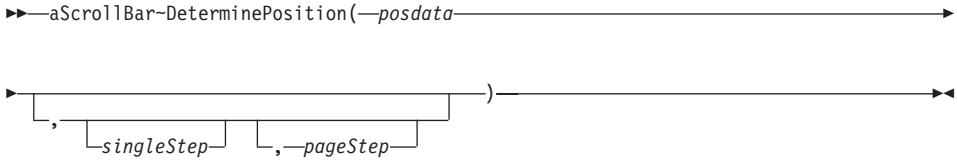
1 For all other cases.

Position

The *Position* method retrieves the position of the scroll box in the associated scroll bar.

Return value:
The position of the scroll box.

DeterminePosition



The *DeterminePosition* method determines the new position of the scroll box based on the position sent with the scroll bar notification messages.

Arguments:
The arguments are:

posdata
The position sent with the scroll bar notification messages.

singleStep
The value by which the scroll box position is increased or decreased when the user performs a single-step event like using the Down or Up arrow keys or clicking on the arrow buttons of the scroll bar.

pageStep
The value by which the scroll box position is increased or decreased when the user performs a page-step event like using the PgDn or PgUp arrow keys or clicking on an area in the scroll bar that is not occupied by the scroll box or the arrow buttons.

Return value:
The resulting position based on *posdata* and the current position.

Note: The position of a scroll bar cannot be modified directly by a user. The Object REXX program must react to the notification that is the result of

the user interaction and set the resulting position of the scroll box using "SetPos" on page 454. Use the DeterminePosition method to determine the resulting position within your notification handler for the scroll bar notification messages.

Chapter 27. PropertySheet Class

The *PropertySheet* class provides methods to control a property sheet. It is a subclass of the CategoryDialog class (see page 271). A property sheet is similar to a category dialog that spreads its dialog items over several pages (categories), where the individual pages are controlled by a tab control instead of radio buttons or combo box lists.

Refer to PROPDEMO.REX in the OODIALOG\SAMPLES directory for an example.

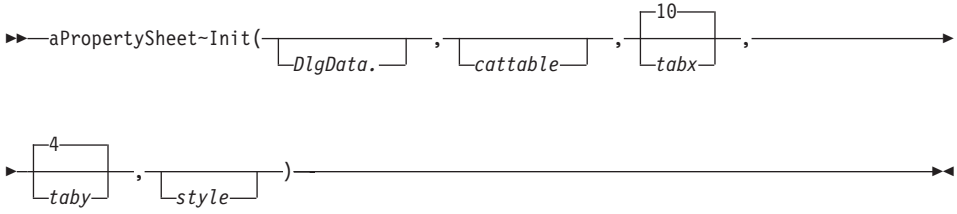
The *PropertySheet* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Use an object of the *DialogControl* class or one of its subclasses to work with an individual dialog item of the sheets. To retrieve such an object, you can call one of the following methods depending on the requested control:

- "GetStaticControl" on page 338
- "GetEditControl" on page 339
- "GetButtonControl" on page 340
- "GetRadioControl" on page 341
- "GetCheckControl" on page 342
- "GetListBox" on page 342
- "GetComboBox" on page 343
- "GetScrollBar" on page 345
- "GetTreeControl" on page 345
- "GetListControl" on page 346
- "GetProgressBar" on page 347
- "GetSliderControl" on page 348
- "GetTabControl" on page 349

Init



The *Init* method is called when a new instance of the PropertySheet class is created and initializes the property sheet object.

Arguments:

The arguments are:

DlgData.

An optional stem variable containing the initial values for some or all dialog items. If the dialog is terminated with the OK button, the values in the data fields of the dialog are copied to this variable. The ID of the dialog items is used to name the entry within the stem.

cattable

You can use this argument to set the category names, separated by blanks. Your class, which is a subclass of PropertySheet, must provide a method for each of the categories in the string to define the dialog items of the related category page. The name of the method is equal to the category name in the string.

If you omit this argument, set the category names to the catalog directory in the InitCategories method (see page 275).

tabx, taby

The position of the tab control that is used to select the category. The default is 10,4.

style Determines the style of the property sheet and the tab control used to select the visible category. The style must be specified as a text string that can contain tab control options (see "AddTabControl" on page 361) or the option WIZARD, which adds a backward and a forward button with ID 11 and 12 to the dialog to switch between the category pages.

Example:

The following example creates a property sheet dialog with 4 sheets:

```
dlg = MyProperty~new(MyData.,"Movies Cinemas Days Ticket",,,"WIZARD")
dlg~createCenter(200,180,"Let's go to the movies")
dlg~execute("SHOWTOP")
   :
::class MyProperty subclass PropertySheet
::method Movies    /* define the Movies page */
::method Cinemas   /* define the Cinemas page */
   :
```

Chapter 28. ProgressBarControl Class

A progress bar is a window that an application can use to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled, from left to right, with the system highlight color as an operation progresses. It has a range and a current position. The range represents the entire duration of the operation, and the current position represents the progress that the application has made toward completing the operation.

The *ProgressBarControl* class provides methods to change the range and the current position of the progress bar. The window procedure uses the range and the current position to determine the percentage of the progress bar. The highest possible range or current position value is 65,535.

Refer to OODPBAR.REX and PROPDEMO.REX in the OODIALOG\SAMPLES directory for an example.

The *ProgressBarControl* class does not send any notification messages. For information about notification messages, refer to "ConnectListNotify" on page 318.

Requires:

The *ProgressBarControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

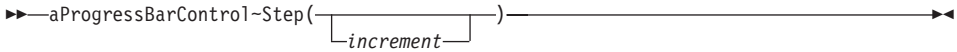
Methods:

Instances of the *ProgressBarControl* class implement the methods listed in Table 14.

Table 14. ProgressBarControl Instance Methods

Method...	...on page
SetPos	462
SetRange	463
SetStep	463
Step	462

Step



The *Step* method advances the current position of a progress bar or the increment set with the SetStep method (see page 463), and redraws the bar to reflect the new position.

Arguments:

The only argument is:

increment

The value for advancing the current position. If you do not specify this argument, the position is advanced by the value set with the SetStep method. If the position exceeds the maximum range value, the progress indicator starts at the beginning again.

Return value:

The previous position.

SetPos

The *SetPos* method sets the new position for a progress bar and redraws the bar to reflect the new position.

Arguments:

The only argument is:

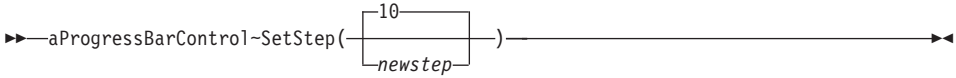
newpos

The new position.

Return value:

The previous position, or –1 if you do not specify *newpos*.

SetStep



The *SetStep* method specifies the step increment for a progress bar. The step increment is the amount by which the progress bar increases its current position whenever the method is called.

Arguments:

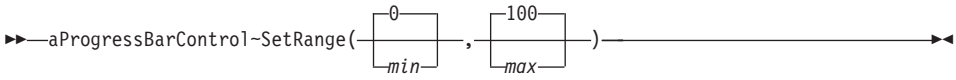
The only argument is:

newstep
The new step increment. The default step increment is 10.

Return value:

The previous position, or –1 if you do not specify *newpos*.

SetRange



The *SetRange* method sets the minimum and maximum values for a progress bar and redraws the bar to reflect the new range.

Arguments:

The arguments are:

min The minimum range value. The default is 0.

max The maximum range value. The default is 100.

Return value:

A string containing the minimum and maximum value of the previous range, separated by a blank, or 0 in all other cases.

Chapter 29. SliderControl Class

A slider control, which is also called a trackbar, is a window that contains a slider and optional tick marks. When the user moves the slider, using either the mouse or the direction keys, the slider sends notification messages to indicate the change.

Sliders are useful if you want the user to select a specific value or a set of consecutive values within a specific range. For example, you might use a slider to enable the user to set the repeat rate of the keyboard by moving the slider to a given tick mark.

The slider in a trackbar moves in increments that you specify when you create it. For example, if you specify that the trackbar should have a range from 0 to 10, the slider can occupy only eleven positions: a position at the left side of the slider and one position for each increment in the range. Typically, each of these positions is identified by a tick mark.

After you have created a slider, you can use the *SliderControl* methods to set and retrieve its properties. Refer to PROPDEMO.REX in the OODIALOG\SAMPLES directory for an example.

The *SliderControl* class sends notification messages to notify about the event. For information about notification messages, refer to "ConnectListNotify" on page 318.

Requires:

The *SliderControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *SliderControl* class implement the methods listed in Table 15.

Table 15. SliderControl Instance Methods

Method...	...on page
ClearSelRange	476
ClearTicks	470
CountTicks	470
GetLineStep	472
GetPageStep	472

Table 15. SliderControl Instance Methods (continued)

Method...	...on page
GetTick	471
InitRange	468
InitSelRange	474
Pos	467
Pos=	466
Range	469
SelRange	477
SetLineStep	473
SetMax	469
SetMin	468
SetPageStep	474
SetPos	466
SetSelEnd	476
SetSelStart	475
SetTickAt	471
SetTickFrequency	471

Pos=

The *Pos=* method sets the new logical position of the slider and redraws the slider.

Arguments:

The only argument is:

value The new logical position. A valid position is an integer value within the range of the minimum and maximum positions of the slider. If you specify a value outside this range, the position is set to the maximum or minimum position.

SetPos

The *SetPos* method sets the new logical position of the slider and redraws the slider if required.

Arguments:

The arguments are:

pos The new logical position. A valid position is an integer value within the range of the minimum and maximum positions of the slider. If you specify a value outside this range, the position is set to the maximum or minimum position.

redraw

The redraw flag. If you specify 1, the control is redrawn with the slider at the position given by *pos*. If you specify 0 or omit this argument, the control is not redrawn. However, the new position is set regardless of the redraw argument.

Example:

The following example sets the slider to the maximum position, with the range of the slider already been set to 0 to 100 using the SetRange method:

```
::method SetToMax
  ctrl=self~GetSliderControl("IDC_1")
  ctrl~SetPos(100,1)
```

Pos

The *Pos* method retrieves the current logical position of the slider.

Return value:

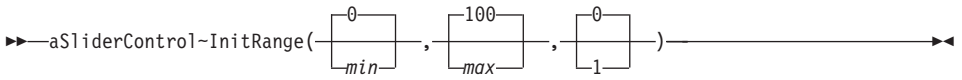
The current logical position of the slider.

Example:

The following example displays the current slider position:

```
::method DisplayPos
  ctrl=self~GetSliderControl("IDC_1")
  pos = ctrl~Pos
  say pos
```

InitRange



The *InitRange* method sets the minimum and maximum positions of the slider and redraws the slider, if required.

Arguments:

The arguments are:

min The minimum position of the slider. The default is 0.

max The maximum position of the slider. The default is 100.

redraw
The redraw flag. If you specify 1, the slider is redrawn after the range is set. If you specify 0 or omit this argument, the slider is not redrawn.

Return value:

0 The range was set.

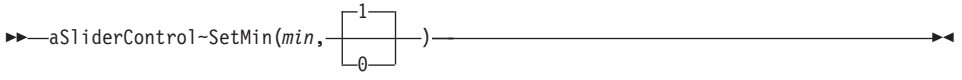
–1 The minimum you specified is greater than the maximum.

Example:

The following example sets the range, the line step, the page step, and the tick frequency of the slider:

```
::method InitDialog
  curSL = self~GetSliderControl("IDC_1")
  if curSL \= .Nil then do
    curSL~InitRange(0,100)
    curSL~SetLineStep(1)
    curSL~SetPageStep(10)
    curSL~SetTickFrequency(10)
  end
```

SetMin



The *SetMin* method sets the minimum logical position for a slider and redraws the slider, if required.

Arguments:

The arguments are:

min The minimum position of the slider.

redraw

The redraw flag. If you specify 1 or omit this argument, the slider is redrawn after the minimum position is set. If you specify 0, the slider is not redrawn.

Return value:

0 The minimum position was set.

–1 You omitted *min*.

SetMax

The *SetMax* method sets the maximum logical position for a slider and redraws the slider, if required.

Arguments:

The arguments are:

min The maximum position of the slider.

redraw

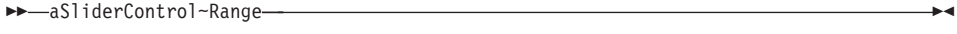
The redraw flag. If you specify 1 or omit this argument, the slider is redrawn after the maximum position is set. If you specify 0, the slider is not redrawn.

Return value:

0 The maximum position was set successfully.

–1 You omitted *min*.

Range



The *Range* method retrieves the minimum and maximum positions of the slider.

Return value:

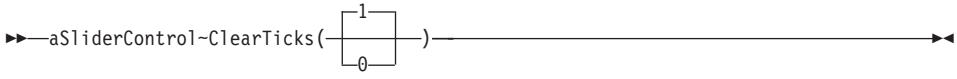
The minimum and maximum positions of the slider, separated by a blank.

Example:

The following example displays the range of a slider:

```
::method DisplayRange
  ctrl=self~GetSlidercontrol("IDC_1")
  range = ctrl~Range
  parse var range min max
  say min max
```

ClearTicks



The *ClearTicks* method removes the current tick marks from a slider. It does not, however, remove the first and last tick marks because they are created automatically by the slider.

Arguments:

The arguments are:

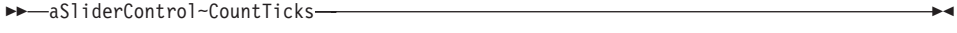
redraw

The redraw flag. If you specify 1 or omit this argument, the slider is redrawn after the tick marks are removed. If you specify 0, the slider is not redrawn.

Return value:

0.

CountTicks



The *CountTicks* method retrieves the number of tick marks in a slider, including the first and last tick marks, which are created automatically by the slider.

Return value:

The number of tick marks.

GetTick

The *GetTick* method retrieves the logical position of a tick mark in a slider. A valid position is an integer value within the minimum and maximum positions of the slider.

Arguments:

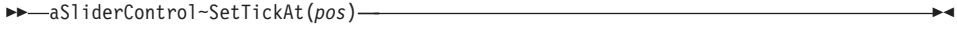
The only argument is:

tic A zero-based index identifying a tick mark. A valid index is in the range of 0 to 2 ticks less than the tick count returned by the CountTicks method (see page 470).

Return value:

The logical position of the specified tick mark, or –1 if you did not specify a valid index for *tic*.

SetTickAt



The *SetTickAt* method sets a tick mark at the specified logical position in the slider.

Arguments:

The only argument is:

pos An integer value within the minimum and maximum positions of the slider.

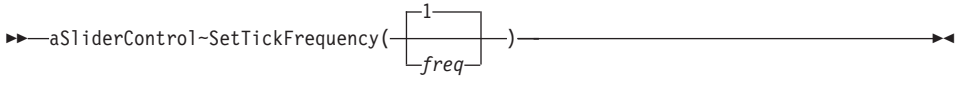
Return value:

0 The tick mark was set.

–1 You omitted *pos*.

1 For all other cases.

SetTickFrequency



The *SetTickFrequency* method sets the interval frequency for tick marks in a slider. For example, if you set the frequency to 2, a tick mark is displayed for every other increment in the slider's range.

Arguments:

The only argument is:

freq The frequency of the tick marks. The default is 1, that is, every increment in the range is associated with a tick mark.

Return value:

0 The tick mark frequency was set.

–1 You omitted *freq*.

1 For all other cases.

Example:

The following example sets the range, the line step, the page step, and the tick frequency of the slider:

```
::method InitDialog
  curSL = self~GetSliderControl("IDC_1")
  if curSL \= .Nil then do
    curSL~InitRange(0,100)
    curSL~SetLineStep(1)
    curSL~SetPageStep(10)
    curSL~SetTickFrequency(10)
  end
```

GetLineStep

```
►►—aSliderControl~GetLineStep————————————————————————————►◄
```

The *GetLineStep* method retrieves the number of logical positions that the slider moves in response to keyboard input from the arrow keys, such as the Right arrow or the Down arrow keys. The logical positions are the integer increments within the minimum and maximum positions of the slider.

Return value:

The line size for the slider.

GetPageStep

```
►►—aSliderControl~GetPageStep————————————————————————————►◄
```

The *GetPageStep* method retrieves the number of logical positions that the slider moves in response to keyboard input, such as the PageUp or PageDown keys, or mouse input, such as clicks in the slider's channel. The logical positions are the integer increments within the minimum and maximum positions of the slider.

Return value:
The page size for the slider.

SetLineStep

The *SetLineStep* method sets the number of logical positions that the slider moves in response to keyboard input from the arrow keys, such as the Right arrow or the Down arrow keys. The logical positions are the integer increments within the minimum and maximum positions of the slider.

Arguments:
The only argument is:

step The new line size.

Return value:
The previous line size, or –1 if you omit *step*.

Example:
The following example sets the range, the line step, the page step, and the tick frequency of the slider:

```
::method InitDialog
  curSL = self~GetSliderControl("IDC_1")
  if curSL \= .Nil then do
    curSL~InitRange(0,100)
    curSL~SetLineStep(1)
    curSL~SetPageStep(10)
    curSL~SetTickFrequency(10)
  end
```

SetPageStep

The *SetPageStep* method sets the number of logical positions that the slider moves in response to keyboard input, such as the PageUp or PageDown keys, or mouse input, such as clicks in the slider's channel. The logical positions are the integer increments within the minimum and maximum positions of the slider.

Arguments:

The only argument is:

step The new page size.

Return value:

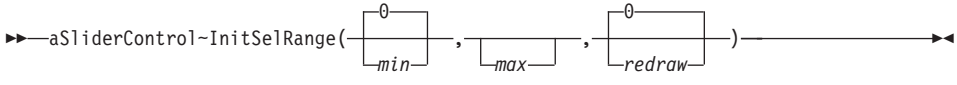
The previous page size.

Example:

The following example sets the range, the line step, the page step, and the tick frequency of the slider:

```
::method InitDialog
  curSL = self~GetSliderControl("IDC_1")
  if curSL \= .Nil then do
    curSL~InitRange(0,100)
    curSL~SetLineStep(1)
    curSL~SetPageStep(10)
    curSL~SetTickFrequency(10)
  end
```

InitSelRange



The *InitSelRange* method sets the starting and ending logical positions for the current selection range in a slider. It is ignored if the slider does not have a selection range.

Arguments:

The arguments are:

min The logical starting position of the selection range. The default is 0.

max The logical ending position of the selection range. If you omit this argument, the maximum position of the slider's range is assumed.

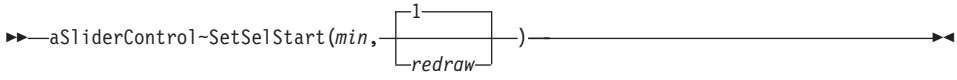
redraw
The redraw flag. If you specify 1, the slider is redrawn after the selection range is set. If you specify 0 or omit this argument, the selection range is set but the slider is not redrawn.

Return value:

–1 The minimum you specified is greater than the maximum.

0 In all other cases.

SetSelStart



The *SetSelStart* method sets the starting logical position for the current selection range in a slider. It is ignored if the slider does not have a selection range.

Arguments:
The arguments are:

min The logical starting position of the selection range.

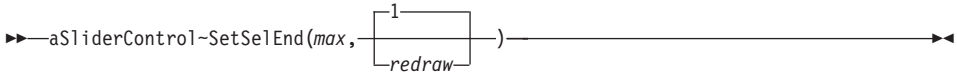
redraw
The redraw flag. If you specify 1 or omit this argument, the slider is redrawn after the starting position has been set. If you specify 0, the starting position is set but the slider is not redrawn.

Return value:

–1 You omitted *min*.

0 In all other cases.

SetSelEnd



The *SetSelEnd* method sets the logical ending position for the current selection range in a slider. It is ignored if the slider does not have a selection range.

Arguments:

The arguments are:

min The logical ending position of the selection range.

redraw

The redraw flag. If you specify 1 or omit this argument, the slider is redrawn after the ending position has been set. If you specify 0, the ending position is set but the slider is not redrawn.

Return value:

–1 You omitted *max*.

0 In all other cases.

ClearSelRange

The *ClearSelRange* method clears the current selection range in a slider.

Arguments:

The only argument is:

redraw

The redraw flag. If you specify 1, the slider is redrawn after the selection is cleared.

Return value:

0.

SelRange

The *SelRange* method retrieves the starting position of the current selection range in a slider.

Return value:

The starting and ending positions of the current selection range, separated by a blank.

Chapter 30. TabControl Class

A tab control can be compared to a divider in a notebook or a label in a file cabinet. By using a tab control, an application can define several pages for the same area of a dialog or dialog control. Each page consists of a set of information or a group of controls that the application displays when the user selects the corresponding tab.

A special type of tab control displays tabs that look like buttons. Clicking a button immediately performs a command instead of displaying a page.

You can apply specific characteristics to tab controls by specifying tab control styles. For example, you can specify the alignment and general appearance of the tabs in a tab control.

By default, a tab control displays only one row of tabs. If not all tabs can be shown at once, the tab control displays an up-and-down control so that the user can scroll to view additional tabs.

Refer to PROPDEMO.REX in the OODIALOG\SAMPLES directory for an example.

The *TabControl* class sends notification messages to notify about the event. For information about notification messages, refer to "ConnectListNotify" on page 318.

Requires:

The *TabControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *TabControl* class implement the methods listed in Table 16.

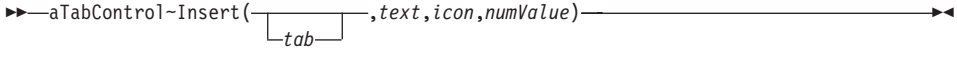
Table 16. TabControl Instance Methods

Method...	...on page
AddFullSeq	482
AddSequence	482
AdjustToRectangle	489
Delete	484
DeleteAll	485

Table 16. TabControl Instance Methods (continued)

Method...	...on page
Focus	486
Focused	487
Insert	480
ItemInfo	484
Items	483
Last	485
Modify	481
PosRectangle	489
RemoveImages	488
RequiredWindowSize	490
Rows	483
Select	486
SelectIndex	486
Selected	485
SelectedIndex	486
SetImages	487
SetPadding	488
SetSize	488

Insert



The *Insert* method inserts a new tab in a tab control.

Arguments:

The arguments are:

tab The number of the tab. If you omit this argument, the number of the last tab is increased by 1, starting with 0.

text The label text for the inserted tab.

icon The index of the icon in the image list of the tab control, set with the SetImages method (see page 487).

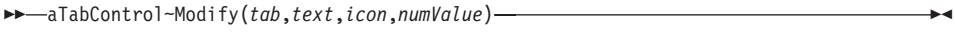
numValue
An integer value stored together with the tab to save information.

Return value:
The number of the new tab, or –1 for all other cases.

Example:
The following example inserts three tabs in a tab control with the specified text and a specific item:

```
::method InitDialog
  InitDlgRet = self~InitDialog:super
  curTab = self~GetTabControl("ID_TAB")
  if curTab \= .Nil then do
    curTab~SetImages("oodtab.BMP",16,16)
    curTab~Insert(,"First Tab",0)
    curTab~Insert(,"Second Tab",1)
    curTab~Insert(,"Third Tab",2)
  end
return InitDlgRet
```

Modify



The *Modify* method sets some or all of the attributes of a tab.

Arguments:
The arguments are:

tab The number of the tab.

text The label text for the tab.

icon The index of the icon in the image list of the tab control, set with the SetImages method (see page 487).

numValue
An integer value stored together with the tab to save information.

Return value:

0 The attributes were set.

–1 You did not specify *tab*.

1 In all other cases.

AddSequence

```
►►—aTabControl~AddSequence(text1,text2,text3,...)———————————————————►◄
```

The *AddSequence* method inserts a sequence of tabs in a tab control for which you can only specify the label text. The number of the tab inserted last is increased by 1.

Arguments:

The only argument is:

text The label text for the inserted tab.

Return value:

The number of the tab inserted last, or –1 for all other cases.

Example:

The following example inserts three tabs in a tab control:

```
::method InitDialog
  InitDlgRet = self~InitDialog:super
  curTab = self~GetTabControl("ID_TAB")
  if curTab \= .Nil then do
    curTab~AddSequence("First Tab","Second Tab","Third Tab")
  end
return InitDlgRet
```

AddFullSeq

```
►►—aTabControl~AddFullSeq(text_1,icon_1,numValue_1,text_2,icon_2,numValue_2,...)———►◄
```

The *AddFullSeq* method inserts a sequence of tabs in a tab control for which you can specify the number, label text, and integer value.

Arguments:

The arguments are:

text The label text for the inserted tab.

icon The index of the icon in the image list of the tab control, set with the SetImages method (see page 487).

numValue
An integer value stored together with the tab to save information.

Return value:

The number of the tab inserted last, or –1 for all other cases.

Example:

The following example adds a sequence of tabs and sets their text and icon:

```
::method InitDialog
  InitDlgRet = self~InitDialog:super
  curTab = self~GetTabControl("ID_TAB")
  if curTab \= .Nil then do
    curTab~SetImages("oodtab.BMP",16,16)
    curTab~AddFullSeq("s11", 0,, "s12", 1,, "s13", 2,, "s14", 3)
  end
return InitDlgRet
```

Items

The *Items* method retrieves the number of tabs in a tab control.

Return value:

The number of the tabs, or 0 for all other cases.

Example:

The following example displays the number of tabs:

```
::method DisplayTabNum
  curTab = self~GetTabControl("ID_TAB")
  if curTab \= .Nil then do
    say curTab~Items
  end
```

Rows

The *Rows* method retrieves the current number of rows of tabs in a tab control. Only tab controls with multiline style can have several rows of tabs.

Return value:

The number of the tab rows.

ItemInfo

```
►►—aTabControl~ItemInfo(tab)—————————————————————————————————►◄
```

The *ItemInfo* method retrieves information about a tab in a tab control.

Arguments:

The only argument is:

text The number of the tab.

Return value:

A compound variable that stores the attributes of the tab, or –1 in all other cases. The compound variable can be:

RetStem.!TEXT

The label text for the tab.

RetStem.!IMAGE

The index of the tab in the image list of the tab control, or –1 if the tab does not have an image.

RetStem.!PARAM

An integer value stored together with the tab to save information:

Example:

The following example displays the text of all tabs:

```
::method DisplayText
  curTab = self~GetTabControl("ID_TAB")
  if curTab \= .Nil then do
    do i = 0 to curTab~Items - 1
       ItemInfo. = curTab~ItemInfo(i)
       say ItemInfo.!Text
    end
  end
```

Delete

```
►►—aTabControl~Delete(tab)———————————————————————————————————►◄
```

The *Delete* method removes a tab from a tab control.

Arguments:

The only argument is:

tab The number of the tab to be removed.

Return value:

0 The tab was removed.

–1 You did not specify *tab* or there is no tab available.

1 For all other cases.

DeleteAll

The *DeleteAll* method removes all tabs from a tab control.

Return value:

0 The tabs were removed.

1 For all other cases.

Last

The *Last* method retrieves the number of the last tab in a tab control.

Return value:
The number of the last tab, or 0 in all other cases.

Selected

The *Selected* method retrieves the label text of the currently selected tab.

Return value:
The label text of the currently selected tab, or 0 in all other cases.

SelectedIndex

```
►►—aTabControl~SelectedIndex————————————————————————————————►◄
```

The *SelectedIndex* method retrieves the number of the currently selected tab.

Return value:
The number of the currently selected tab, or 0 in all other cases.

Select

```
►►—aTabControl~Select(text)——————————————————————————————————►◄
```

The *Select* method selects the tab with the specified label text.

Arguments:
The only argument is:

text The label text of the tab to be selected.

Return value:
The number of the selected tab, or 0 in all other cases.

SelectIndex

```
►►—aTabControl~SelectIndex(tab)———————————————————————————————►◄
```

The *SelectIndex* method selects the specified tab in a tab control.

Arguments:
The only argument is:

tab The number of the tab to be selected.

Return value:
The number of the previously selected tab, or –1 in all other cases.

Focus

```
►►—aTabControl~Focus(tab)—————————————————————————————————————►◄
```

The *Focus* method sets the focus to the specified tab in a tab control.

Arguments:

The only argument is:

tab The number of the tab to receive the focus.

Return value:

0.

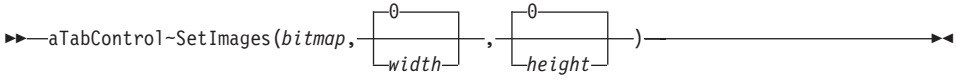
Focused

The *Focused* method returns the number of the tab that has the focus. The tab with the focus can differ from the selected tab.

Return value:

The number of the tab having the focus.

SetImages



The *SetImages* method assigns an image list to a tab control.

Arguments:

The arguments are:

bitmap

The name of, or the handle to, a bitmap file that is already loaded with LoadBitmap method (see page 208).

width The width, in pixels, of each image. If you specify 0 or omit this argument, the width of the image in the image file is used.

height The height, in pixels, of each image. If you specify 0 or omit this argument, the height of the image in the image file is used.

Return value:

The handle to the previous image list, or 0 for all other cases.

RemoveImages

```
►►─aTabControl~RemoveImages──────────────────────────────────────────►◄
```

The *RemoveImages* method deletes an image list of a tab control.

Return value:

0 The image list was removed.

1 In all other cases.

SetPadding

```
►►─aTabControl~SetPadding(padX,padY)─────────────────────────────────►◄
```

The *SetPadding* method sets the amount of space (padding) around the icon and the label of a tab.

Arguments:

The arguments are:

padX The amount of horizontal padding, in pixels.

padY The amount of vertical padding, in pixels.

Return value:

0.

SetSize

```
►►─aTabControl~SetSize(width,height)─────────────────────────────────►◄
```

The *SetSize* method sets the width and height of tabs in a fixed-width or owner-drawn tab control.

Arguments:

The arguments are:

width The new width, in pixels.

height The new height, in pixels.

Return value:

The old width and height as a string, in pixels.

PosRectangle

```
►►—aTabControl~PosRectangle(tab)———————————————————————————►◄
```

The *PosRectangle* method retrieves the rectangle around a tab in a tab control.

Arguments:

The only argument is:

tab The number of the tab.

Return value:

A string containing the coordinates of the rectangle, or an empty string. The coordinates are separated by blanks and are in the following order:

- X-coordinate of the upper left corner of the rectangle
- Y-coordinate of the upper left corner of the rectangle
- X-coordinate of the lower right corner of the rectangle
- Y-coordinate of the lower right corner of the rectangle

AdjustToRectangle

```
►►—aTabControl~AdjustToRectangle(left,top,right,bottom)———————————►◄
```

The *AdjustToRectangle* method calculates the window rectangle of a tab control that corresponds to the specified display rectangle.

Arguments:

The arguments are:

left The x-coordinate of the upper left corner of the display rectangle.

top The y-coordinate of the upper left corner of the display rectangle.

right The x-coordinate of the lower right corner of the display rectangle.

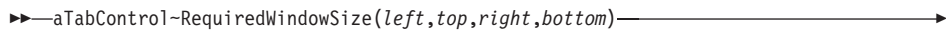
bottom
The y-coordinate of the lower right corner of the display rectangle.

Return value:

A string containing the coordinates of the window rectangle, or an empty string. The coordinates are separated by blanks and are in the following order:

- X-coordinate of the upper left corner of the rectangle
- Y-coordinate of the upper left corner of the rectangle
- X-coordinate of the lower right corner of the rectangle
- Y-coordinate of the lower right corner of the rectangle

RequiredWindowSize



The *RequiredWindowSize* method calculates the display rectangle of a tab control that corresponds to the specified window rectangle.

Arguments:

The arguments are:

left The x-coordinate of the upper left corner of the window rectangle.

top The y-coordinate of the upper left corner of the window rectangle.

right The x-coordinate of the lower right corner of the window rectangle.

bottom
The y-coordinate of the lower right corner of the window rectangle.

Return value:

A string containing the coordinates of the display rectangle, or an empty string. The coordinates are separated by blanks and are in the following order:

- X-coordinate of the upper left corner of the rectangle
- Y-coordinate of the upper left corner of the rectangle
- X-coordinate of the lower right corner of the rectangle
- Y-coordinate of the lower right corner of the rectangle

Chapter 31. TreeControl Class

The tree view control is a dialog that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. Each item consists of a label and an optional image bitmap, and can have a list of subitems associated with it. By clicking on an item, the user can expand and collapse the associated list of subitems.

Refer to OODTREE.REX in the OODIALOG\SAMPLES directory for an example.

Requires:

The *TreeControl* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods:

Instances of the *TreeControl* class implement the methods listed in Table 17.

Table 17. TreeControl Instance Methods

Method...	...on page
Add	494
Child	501
Collapse	505
CollapseAndReset	506
Delete	504
DeleteAll	505
DropHighlight	510
DropHighlighted	502
Edit	508
EndEdit	509
EnsureVisible	507
Expand	506
FirstVisible	502
HitTest	512
Indent	508
Indent=	508
Insert	492

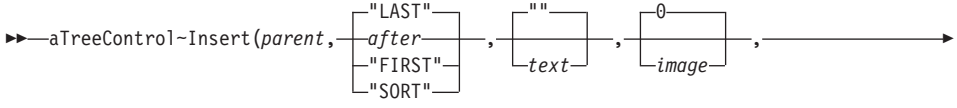
Table 17. TreeControl Instance Methods (continued)

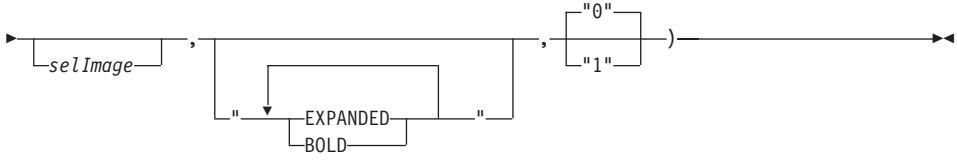
Method...	...on page
IsAncestor	514
ItemInfo	499
Items	500
MakeFirstVisible	510
Modify	496
MoveItem	513
Next	503
NextVisible	503
Parent	501
Previous	504
PreviousVisible	504
RemoveImages	512
RestoreEditClass	509
Root	501
Select	509
Selected	502
SetImages	511
SortChildren	511
SubclassEdit	509
Toggle	507
VisibleItems	500

Methods of the TreeControl Class

The following sections describe the individual methods of the TreeControl class.

Insert





The *Insert* method inserts a new item in a tree view control.

Arguments:

The arguments are:

parent The handle to the parent item. If you specify "ROOT", the item is inserted at the root of the tree view control.

after The handle to the item after which the new item is to be inserted or one of the following values:

"FIRST" Inserts the item at the beginning of the list.

"LAST" Inserts the item at the end of the list. This is the default.

"SORT" Inserts the item into the list in alphabetical order.

text The text for the item. If you omit this argument, "" is assumed.

image The index of the icon image in the tree view control's bitmap file to be used when the item is in the non-selected state. If you omit this argument, the icon with index 0 is used.

selImage

The index of the icon image in the tree view control's bitmap file to be used when the item is in the selected state. If you omit this argument, the icon specified for *image* is used.

state Specifies the appearance and functionality of the item. It can be a combination of the following values, separated by a blank:

EXPANDED The item is currently expanded with all child items visible. This only applies to parent items.

BOLD The item is shown in bold.

If you omit this argument, the item is inserted in collapsed and normal form.

children

Indicates whether the item has child items associated with it:

"0" The item has no child items, which is the default.

"1" The item has one or more child items.

You can use this argument to show an item that does not have any child items, with an expand button. This allows you to save memory usage by dynamically loading and displaying the child items only when the user expands the item.

Return value:

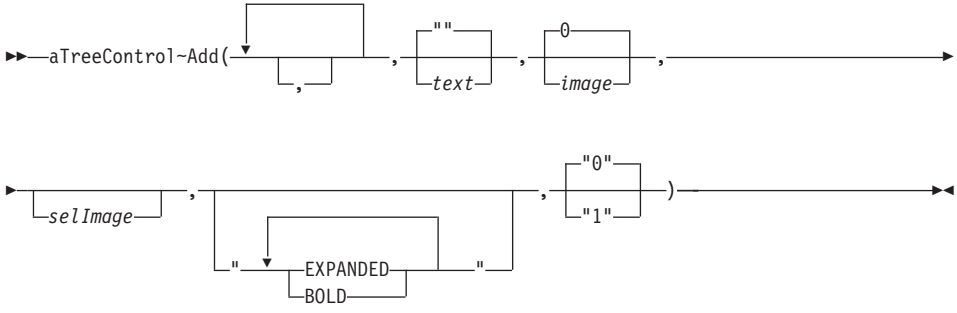
The handle to the new item, or 0 for all other cases.

Example:

The following example inserts child items when a specific root item is expanded. The text strings for the items are loaded from a file.

```
::method OnExpanding_IDC_TREE
  use arg tree, item
  itemFile = "root6.inp"
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(item)

  if itemInfo.!TEXT = "Root 6" & \itemInfo.!STATE~POS("EXPANDED") then
  do
    do while lines(itemFile)
      line = linein(itemFile)
      command = "curTree~Insert(item,,"||line||")"
      interpret command
    end
    curTree~Expand(item)
  end
```

Add



The *Add* method adds a new item to a tree view control.

Arguments:

The arguments are:

, The number of commas specifies at which parent the item is to be inserted. If you omit this argument, the item is inserted as a root item. Each additional comma inserts the item one level deeper than the item inserted previously. See the example in the following.

text The text for the item. If you omit this argument, "" is assumed.

image The index of the icon image in the tree view control's bitmap file to be used when the item is in the non-selected state. If you omit this argument, the icon with index 0 is used.

selImage
The index of the icon image in the tree view control's bitmap file to be used when the item is in the selected state. If you omit this argument, the icon specified for *image* is used.

state Specifies the appearance and functionality of the item. It can be a combination of the following values, separated by a blank:

EXPANDED The item is currently expanded with all child items visible. This only applies to parent items.

BOLD The item is shown in bold.

If you omit this argument, the item is inserted in collapsed and normal form.

children
Indicates whether the item has child items associated with it:

"0" The item has no child items, which is the default.

"1" The item has one or more child items.

You can use this argument to show an item that does not have any child items, with an expand button. This allows you to save memory usage by dynamically loading and displaying the child items only when the user expands the item.

Return value:

The handle to the new item, or 0 for all other cases.

Example:

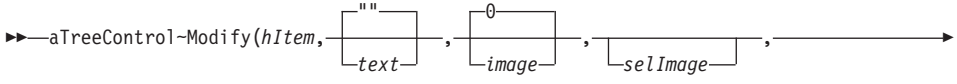
To get the following tree view:

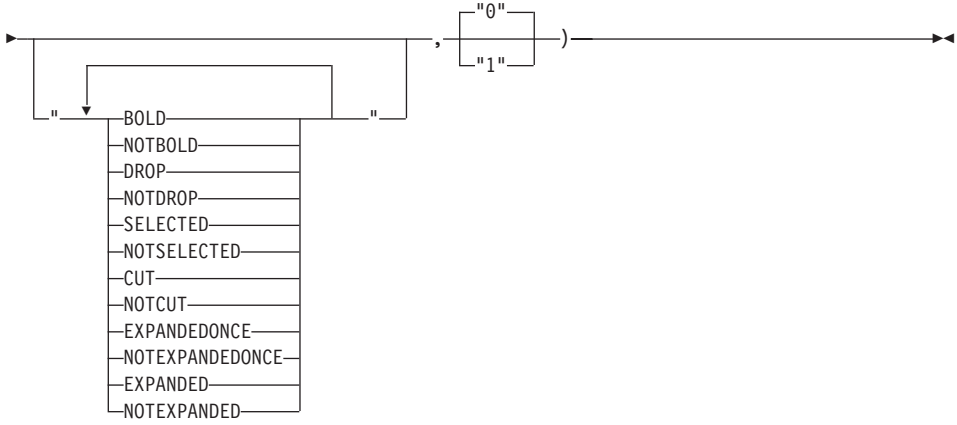
- Peter
  - Mike
    - George
    - Monique
      - John
    - Chris
  - Maud
  - Ringo
- Paul
  - Dave
  - Sam
  - Jeff
- Mary
  - Helen
  - Michelle
  - Diana

your example must look as follows:

```
::method InitDialog
  InitDlgRet = self~InitDialog:super
  curTree = self~GetTreeControl("IDC_TREE")
  if curTree \= .Nil then
  do
    curTree~Add("Peter",,,"BOLD EXPANDED")
    curTree~Add(,"Mike",,,"EXPANDED")
    curTree~Add(,,"George")
    curTree~Add(,,"Monique")
    curTree~Add(,,,"John")
    curTree~Add(,,"Chris")
    curTree~Add(,"Maud")
    curTree~Add(,"Ringo")
    curTree~Add("Paul",,,"BOLD EXPANDED")
    curTree~Add(,"Dave")
    curTree~Add(,"Sam")
    curTree~Add(,"Jeff")
    curTree~Add("Mary",,,"BOLD EXPANDED")
    curTree~Add(,"Helen")
    curTree~Add(,"Michelle")
    curTree~Add(,"Diana")
end
```

Modify





The *Modify* method sets some or all attributes of an item of a tree view control.

Arguments:

The arguments are:

hItem The handle to the item to be modified.

text The text for the item. If you omit this argument, "" is assumed.

image The index of the icon image in the tree view control's bitmap file to be used when the item is in the non-selected state. If you omit this argument, the icon with index 0 is used.

selImage
The index of the icon image in the tree view control's bitmap file to be used when the item is in the selected state. If you omit this argument, the icon specified for *image* is used.

state Specifies the appearance and functionality of the item. It can be one or more of the following values, separated by blanks:

BOLD The item is shown in bold.

NOTBOLD The item is not bold.

DROP The item is selected as a drag-and-drop target.

NOTDROP The item is not selected as a drag-and-drop target.

SELECTED The item is selected. Its appearance depends on whether it has the focus and whether the system colors are used for the selection.

NOTSELECTED
The item is not selected.

CUT The item is selected as part of a cut-and-paste operation.

NOTCUT The item is not selected as part of a cut-and-paste operation.

EXPANDEDONCE
The item's list of child items has been expanded at least once.

NOTEXPANDEDONCE
The item's list of child items has not been expanded at least once.

EXPANDED The item's list is currently expanded with all child items visible. This only applies to parent items.

NOTEXPANDED
The item's list is currently not expanded.

children
Indicates whether the item has child items associated with it:

"0" The item has no child items, which is the default.

"1" The item has one or more child items.

You can use this argument to show an item that does not have any child items, with an expand button. This allows you to save memory usage by dynamically loading and displaying the child items only when the user expands the item.

Return value:

0 The item has been modified.

–1 For all other cases.

Example:
The following example changes the text of the item to bold when it is selected:

```
::method OnSelChanging_IDC_TREE
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~Modify(curTree~selected,,,,"BOLD")
```

ItemInfo

The *ItemInfo* method retrieves some or all attributes of an item of a tree view control.

Arguments:

The only argument is:

hItem The handle to the item of which attributes are to be retrieved.

Return value:

A compound variable that stores the attributes of the item, or –1 in all other cases. The compound variable can be:

RetStem.!TEXT
The text of the item.

RetStem.!CHILDREN

1 The item has children.

0 The item has no children.

RetStem.!IMAGE
The index of the icon image in the tree view control's bitmap file used when the item is in the non-selected state.

RetStem.!SELECTEDIMAGE
The index of the icon image in the tree view control's bitmap file used when the item is in the selected state.

RetStem.!STATE
An empty string or one or more of the following strings, separated by blanks:

EXPANDED
The item's list is currently expanded with all child items visible. This only applies to parent items.

BOLD The item is in bold.

SELECTED
The item is selected.

EXPANDEDONCE
The item's list has been expanded at least once. This only applies to parent items.

INDROP
The item is selected as a drag-and-drop target.

Example:

The following example displays the attributes of the selected item:

```
::method Info
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~Selected)
  say itemInfo.!TEXT
  say itemInfo.!CHILDREN
  say itemInfo.!IMAGE
  say itemInfo.!STATE
```

Items

The *Items* method retrieves the number of items in a tree view control.

Return value:

The number of items.

Example:

The following example counts all items in a tree view control:

```
::method Count
  curTree = self~GetTreeControl("IDC_TREE")
  say curTree~Items
```

VisibleItems

The *VisibleItems* method obtains the number of items that can be fully visible in a tree view control. This number can be greater than the number of items in the control. The control calculates this value by dividing the height of the client window by the height of an item.

Return value:

The number of items that can be fully visible. For example, if you can see all of 19 items and part of another item, the return value is 19, not 20.

Example:

The following example returns the number of items that can be fully visible:

```
::method Visible
  curTree = self~GetTreeControl("IDC_TREE")
  say curTree~VisibleItems
```

Root

The *Root* method retrieves the first or topmost item of the tree view control.

Return value:
The handle to the first item, or 0 in all other cases.

Example:
The following example displays the name of the root item:

```
::method RootName
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~Root)
  say ItemInfo.!Text
```

Parent

The *Parent* method retrieves the parent of the specified item.

Arguments:
The only argument is:

hItem The handle to the item for which the parent is to be retrieved.

Return value:
The handle to the parent item, or –1 if *hItem* is not specified or is 0, or 0 in all other cases.

Example:
The following example displays the name of the selected item's parent:

```
::method Parent
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~Parent(curTree~Selected))
  say ItemInfo.!Text
```

Child

The *Child* method retrieves the first child item of *hItem*.

Arguments:
The only argument is:

hItem The handle to the item of which the first child is to be retrieved.

Return value:
The handle to the first child item, or –1 if you omitted *hItem*, or 0 in all other cases.

Example:
The following example displays the name of parent of the selected item:

```
::method Child
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~Child(curTree~Selected))
  say ItemInfo.!Text
```

Selected

```
►►—aTreeControl~Selected————————————————————————————————————————►◄
```

The *Selected* method retrieves the currently selected item.

Return value:
The handle to the currently selected item, or 0 in all other cases.

Example:
The following example displays the name of the selected item:

```
::method SelectedName
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~Selected)
  say ItemInfo.!Text
```

DropHighlighted

```
►►—aTreeControl~DropHighlighted—————————————————————————————————►◄
```

The *DropHighlighted* method retrieves the item that is the target of a drag-and-drop operation.

Return value:
The handle to the item, or 0 in all other cases.

FirstVisible

```
►►—aTreeControl~FirstVisible————————————————————————————————————►◄
```

The *FirstVisible* method retrieves the first visible item in the client window of a tree view control.

Return value:
The handle to the first visible item, or 0 in all other cases.

Example:
The following example displays the name of the first visible item:

```
::method FirstVisibleName
  curTree = self~GetTreeControl("IDC_TREE")
  itemInfo. = curTree~ItemInfo(curTree~FirstVisible)
  say ItemInfo.!Text
```

Next

The *Next* method retrieves the sibling item next to *hItem*.

Arguments:
The only argument is:

hItem The handle to the item next to the sibling item to be retrieved.

Return value:
The handle to the next sibling item, or –1 if you omitted *hItem*, or 0 in all other cases.

Example:
The following example displays the name of the selected item and its siblings:

```
::method SiblingNames
  curTree = self~GetTreeControl("IDC_TREE")
  nextItem = curTree~Selected
  do while nextItem \= 0
    itemInfo. = curTree~ItemInfo(nextItem)
    say ItemInfo.!Text
    nextItem = curTree~Next(nextItem)
  end
```

NextVisible

The *NextVisible* method retrieves the visible item following *hItem*.

Arguments:
The only argument is:

hItem The handle to the item that precedes the visible item to be retrieved. *hItem* must also be visible.

Return value:
The handle to the next visible item, or –1 if you omitted *hItem*, or 0 in all other cases.

Previous

The *Previous* method retrieves the sibling item preceding *hItem*.

Arguments:
The only argument is:

hItem The handle to the item that follows the sibling item to be retrieved.

Return value:
The handle to the previous sibling item, or –1 if *hItem* is not specified or is 0, or 0 in all other cases.

PreviousVisible

The *PreviousVisible* method retrieves the visible item preceding *hItem*.

Arguments:
The only argument is:

hItem The handle to the item that follows the visible child item to be retrieved. *hItem* must also be visible.

Return value:
The handle to the previous visible child item, or –1 if *hItem* is not specified or is 0, or 0 in all other cases.

Delete

The *Delete* method removes an item from a tree view control.

Arguments:
The only argument is:

hItem The handle to the item to be deleted.

Return value:

0 The item was deleted.

1 An error occurred.

–1 *hItem* is 0 or is not a valid value.

Example:

The following example deletes the selected item and all its children, if any:

```
::method IDC_PB_DELETE
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~Delete(curTree~Selected)
```

DeleteAll

The *DeleteAll* method removes all items from a tree view control.

Return value:

0 The items were removed.

1 For all other cases.

Collapse

The *Collapse* method collapses the list of child items associated with the specified parent item.

Arguments:

The only argument is:

hItem The handle to the parent item to collapse.

Return value:

0 The list of child items has collapsed.

–1 *hItem* is not specified or is 0.

1 For all other cases.

Example:

The following example collapses the selected item:

```
::method CollapseSelected
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~Collapse(curTree~Selected)
```

CollapseAndReset

The *CollapseAndReset* method collapses the list of child items associated with the specified parent item and removes the child items.

Arguments:

The only argument is:

hItem The handle to the parent item to collapse.

Return value:

0 The list of child items has collapsed and the child items have been removed.

–1 *hItem* is not specified or is 0.

1 For all other cases.

Example:

The following example collapses the selected item and removes all its child items:

```
::method CollapseSelectedAndReset
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~CollapseAndReset(curTree~Selected)
```

Expand

The *Expand* method expands the list of child items associated with the specified parent item.

Arguments:

The only argument is:

hItem The handle to the parent item to be expanded.

Return value:

0 The parent item was expanded.

–1 *hItem* is not specified or is 0.

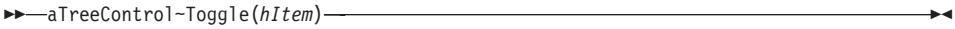
1 For all other cases.

Example:

The following example expands the selected item:

```
::method ExpandSelected
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~Expand(curTree~Selected)
```

Toggle



The *Toggle* method collapses the list of the specified item if it was expanded, or expands it if it was collapsed.

Arguments:

The only argument is:

hItem The handle to the item to be expanded or collapsed.

Return value:

0 The item was expanded or collapsed.

–1 *hItem* is not specified or is 0.

1 For all other cases.

Example:

The following example toggles between expanding and collapsing a selected item:

```
::method ToggleSelected
  curTree = self~GetTreeControl("IDC_TREE")
  curTree~Toggle(curTree~Selected)
```

EnsureVisible

The *EnsureVisible* method ensures that a tree view item is visible, expanding the parent item or scrolling the tree view control, if necessary.

Arguments:

The only argument is:

hItem The handle to the item to be visible.

Return value:

0 The items in the tree view control were scrolled to ensure that the specified item is visible.

–1 *hItem* is not specified or is 0.

1 For all other cases.

Indent

The *Indent* method retrieves the amount, in pixels, by which the child items are indented relative to their parent item.

Return value:

The amount indented, in pixels.

Indent=

The *Indent*= method sets the width of indentation for a tree view control and redraws the control to reflect the new width.

Arguments:

The only argument is:

indent The width of the indentation, in pixels. If you specify a width that is smaller than the system-defined minimum, it is set to the system-defined minimum.

Return value:

–1 if *indent* is 0.

Edit

The *Edit* method starts editing the text of the specified item by replacing the text with a single-line edit control containing this text. It implicitly selects and focuses the specified item.

Arguments:

The only argument is:

hItem The handle to the item to be edited.

Return value:

The handle to the edit control used to edit the item text, or –1 if *hItem* is not specified or is 0, or 0 in all other cases.

EndEdit

The *EndEdit* method ends the editing of the item label of the tree view.

Arguments:

The only argument is:

cancel Indicates whether editing is canceled without being saved to the label. If you specify "1" or "YES", editing is canceled. Otherwise, the changes are saved to the label, which is the default.

Return value:

0 Editing has ended successfully.

1 For all other cases.

SubclassEdit

The *SubclassEdit* method is used by the DefTreeEditHandler to correct the problem occurring when Esc or the Enter key is pressed in an active edit item.

RestoreEditClass

The *RestoreEditClass* method is used by the DefTreeEditHandler to correct the problem occurring when Esc or the Enter key is pressed in an active edit item.

Select

The *Select* method selects a specific item.

Arguments:

The only argument is:

hItem The handle to the item to be selected.

Return value:

0 The item was selected.

–1 *hItem* was not specified or is 0.

1 For all other cases.

MakeFirstVisible

The *MakeFirstVisible* method ensures that *hItem* is visible and displays it at the top of the control's dialog, if possible. If the specified item is near the end of the control's hierarchy of items, it might not become the first visible item depending on how many items fit in the dialog.

Arguments:

The only argument is:

hItem The handle to the item to be visible first.

Return value:

0 The item is visible first.

–1 *hItem* was not specified or is 0.

1 For all other cases.

DropHighlight

The *DropHighlight* method redraws *hItem* in the style used to indicate the target of a drag-and-drop operation.

Arguments:

The only argument is:

hItem The handle of the item to be redrawn.

Return value:

0 The item was redrawn.

–1 *hItem* was not specified or is 0.

1 For all other cases.

SortChildren

The *SortChildren* method sorts the child items of the specified parent item in a tree view control.

Arguments:

The only argument is:

hItem The handle to the parent item the child items of which are to be sorted.

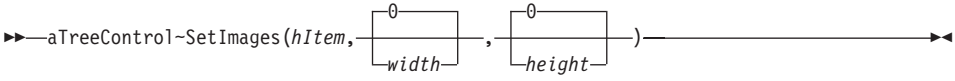
Return value:

0 The child items were sorted.

–1 *hItem* was not specified or is 0.

1 For all other cases.

SetImages



The *SetImages* method sets the image list for a tree view control and redraws the control using the new images.

Arguments:

The arguments are:

bitmap

The name of, or handle to, a bitmap file that is already loaded using the LoadBitmap method (see page 208).

width The width of each image, in pixels. If you specify 0 or omit this argument, the height of the image in the image file is used as width.

height The height of each image, in pixels. If you specify 0 or omit this argument, the height of the image in the image file is used.

Return value:

The handle to the previous image list, or –1 if you did not specify *bitmap*, or 0 in all other cases.

Example:

The following example sets the image list during dialog initialization:

```
::method InitDialog
  expose bmpFile
  InitDlgRet = self~InitDialog:super
  curTree = self~GetTreeControl("IDC_TREE")
  if curTree \= .Nil then
  do
    curTree~SetImages(bmpFile,16,12)
  end
  return InitDlgRet
```

RemoveImages

```
►►—aTreeControl~RemoveImages————————————————————————————————————►◄
```

The *RemoveImages* method destroys the image list of the tree view.

Return value:

0 The image list was destroyed.

1 For all other cases.

HitTest

```
►►—aTreeControl~HitTest(x,y)—————————————————————————————————————►◄
```

The *HitTest* method determines the location of the specified point relative to the client area of a tree view control.

Arguments:

The arguments are:

x The x-coordinate of the point.

y The y-coordinate of the point.

Return value:

0 if no item occupies the point, or one or more of the following strings if an item occupies the specified point:

handle
The handle to the item that occupies the specified point.

ABOVE
Above the client area.

BELOW
Below the client area.

NOWHERE
In the client area but below the last item.

ONITEM
On the bitmap or label associated with an item.

ONBUTTON
On the button associated with an item.

ONICON
On the icon associated with an item.

ONINDENT
In the indentation associated with an item.

ONLABEL
On the label (string) associated with an item.

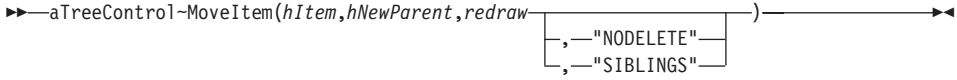
ONRIGHT
In the area to the right of an item.

ONSTATEICON
On the state icon for a tree view item that is in a user-defined state.

TOLEFT
To the left of the client area.

TORIGHT
To the right of the client area.

MoveItem



The *MoveItem* method moves an item to a new location.

Arguments:
The arguments are:

hItem The handle to the item to be moved.

hNewParent
The handle to the new parent to which the item is to be moved.

redraw
The tree view control is updated.

options
One of the following options:

"NODELETE" The item is copied to another location.

"SIBLINGS" Siblings are moved together with the item.

Return value:
The handle to the new parent, or 0 in all other cases.

IsAncestor

The *IsAncestor* method checks if an item is an ancestor of another item.

Arguments:
The arguments are:

hParent
The ancestor.

hItem The item to be checked.

Return value:
1 if *hParent* is an ancestor of *hItem*.

Notification Messages

The tree view control sends notification messages to notify about events. For more information on notification messages, refer to "ConnectListNotify" on page 318.

The following example shows how to connect the tree view notification messages with the corresponding message:

```
::method Init
  use arg InitStem.
  if Arg(1,"o") = 1 then
     InitRet = self~Init:super
  else
     InitRet = self~Init:super(InitStem.)

  if self~Load("tree.rc", ) \= 0 then do
     self~InitCode = 1
    return
  end

  /* Connect dialog control items to class methods */
  self~ConnectTreeNotify("IDC_TREE","SelChanging","OnSelChanging_IDC_TREE")
  self~ConnectTreeNotify("IDC_TREE","SelChanged","OnSelChanged_IDC_TREE")
  self~ConnectTreeNotify("IDC_TREE","Expanding","OnExpanding_IDC_TREE")
```

```
self~ConnectTreeNotify("IDC_TREE","Expanded","OnExpanded_IDC_TREE")
self~ConnectTreeNotify("IDC_TREE","DefaultEdit")
self~ConnectTreeNotify("IDC_TREE","Delete","OnDelete_IDC_TREE")
self~ConnectTreeNotify("IDC_TREE","KeyDown","OnKeyDown_IDC_TREE")
self~ConnectButton("IDC_PB_NEW","IDC_PB_NEW")
self~ConnectButton("IDC_PB_DELETE","IDC_PB_DELETE")
self~ConnectButton(2,"Cancel")
self~ConnectButton(9,"Help")
self~ConnectButton(1,"OK")
return InitRet
```

Chapter 32. VirtualKeyCodes Class

The methods of the *VirtualKeyCodes* class can be used for all objects of the WindowsProgramManager and WindowObject classes. These classes inherit from the *VirtualKeyCodes* class. The *VirtualKeyCodes* class cannot be used as a standalone class.

The *VirtualKeyCodes* class requires the class definition file `oodwin32.cls`:

```
::requires oodwin32.cls
```

Methods of the VirtualKeyCodes Class

Instances of the VirtualKeyCodes class implement the methods described in the following sections.

VCode

The *VCode* method returns the decimal value of a symbolic key name.

Arguments:

The only argument is:

keyname
The symbolic key name. See "Symbolic Names for Virtual Keys" on page 518 for a list of key names.

Return value:

The decimal value of the symbolic key name. If the symbolic name is not found, 255 is returned.

KeyName

The *KeyName* method returns the symbolic key name of the specified hexadecimal code.

Arguments:

The only argument is:

vcode The hexadecimal code of the key.

Return value:

The symbolic key name of the specified code.

Example:

The following example deletes or inserts an item in a tree view depending on the selected key:

```
::method OnKeyDown_IDC_TREE
  use arg treeId, key
  curTree = self~GetTreeControl(treeId)
  /* if the Delete key is pressed, delete the selected item */
  if self~KeyName(key) = "DELETE" then
     curTree~Delete(curTree~Selected)
  else
  /* if the Insert key is pressed, simulate pressing the New button */
  if self~KeyName(key) = "INSERT" then
     self~IDC_PB_NEW
```

Symbolic Names for Virtual Keys

Table 18 shows the symbolic names and the keyboard equivalents for the virtual keys used by Object REXX:

Table 18. Symbolic Names for Virtual Keys

Symbolic Name	Mouse or Keyboard Equivalent
LBUTTON	Left mouse button
RBUTTON	Right mouse button
CANCEL	Control-break processing
MBUTTON	Middle mouse button (three-button mouse)
BACK	BACKSPACE key
TAB	TAB key
CLEAR	CLEAR key
RETURN	ENTER key
SHIFT	SHIFT key
CONTROL	CRTL key
MENU	ALT key
PAUSE	PAUSE key
CAPITAL	CAPS LOCK key
ESCAPE	ESC key
SPACE	SPACEBAR
PRIOR	PAGE UP key
NEXT	PAGE DOWN key

Table 18. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
END	END key
HOME	HOME key
LEFT	LEFT ARROW key
UP	UP ARROW key
RIGHT	RIGHT ARROW key
DOWN	DOWN ARROW key
SELECT	SELECT key
EXECUTE	EXECUTE key
SNAPSHOT	PRINT SCREEN key
INSERT	INS key
DELETE	DEL key
HELP	HELP key
0	0 key
1	1 key
2	2 key
3	3 key
4	4 key
5	5 key
6	6 key
7	7 key
8	8 key
9	9 key
A	A key
B	B key
C	C key
D	D key
E	E key
F	F key
G	G key
H	H key
I	I key
J	J key

Table 18. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
K	K key
L	L key
M	M key
N	N key
O	O key
Q	Q key
R	R key
S	S key
T	T key
U	U key
V	V key
W	W key
X	X key
Y	Y key
Z	Z key
NUMPAD0	Numeric keypad 0 key
NUMPAD1	Numeric keypad 1 key
NUMPAD2	Numeric keypad 2 key
NUMPAD3	Numeric keypad 3 key
NUMPAD4	Numeric keypad 4 key
NUMPAD5	Numeric keypad 5 key
NUMPAD6	Numeric keypad 6 key
NUMPAD7	Numeric keypad 7 key
NUMPAD8	Numeric keypad 8 key
NUMPAD9	Numeric keypad 9 key
MULTIPLY	Multiply key
ADD	Add key
SEPARATOR	Separator key
SUBTRACT	Subtract key
DECIMAL	Decimal key
DIVIDE	Divide key
F1	F1 key

Table 18. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
F2	F2 key
F3	F3 key
F4	F4 key
F5	F5 key
F6	F6 key
F7	F7 key
F8	F8 key
F9	F9 key
F10	F10 key
F11	F11 key
F12	F12 key
F13	F13 key
F14	F14 key
F15	F15 key
F16	F16 key
F17	F17 key
F18	F18 key
F19	F19 key
F20	F20 key
F21	F21 key
F22	F22 key
F23	F23 key
F24	F24 key
NUMLOCK	NUM LOCK key
SCROLL	SCROLL LOCK key

Part 3. Appendixes

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100
70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

IBM
OS/2
VisualAge

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

Readers' Comments — We'd Like to Hear from You

Object REXX for Windows
OODialog Method Reference
Version 2.1

Publication No. SH12-6727-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SH12-6727-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Str. 220
71032 Boeblingen
Germany

Fold and Tape | **Please do not staple** | Fold and Tape

SH12-6727-00

Cut or Fold
Along Line

Part Number: CT81FIE
Program Number: 5639-M68 Development Edition

Printed in Denmark by IBM Danmark A/S

(1P) P/N: CT81FIE

SH12-6727-00