

DrDialog Reference

Table of Contents

Introduction	1
Asking for help	3
Editing a dialog	5
<u>Container controls</u>	5
<u>Tabbing order</u>	6
<u>Selecting controls</u>	6
<u>Moving controls</u>	7
<u>Moving controls between dialogs</u>	8
<u>Copying controls</u>	8
<u>Copying controls between dialogs</u>	8
<u>Sizing controls</u>	9
<u>Editing a control's style information</u>	9
<u>Editing a control's REXX code</u>	9
<u>Editing a control's attributes using the pop-up menu</u>	10
<u>Keyboard shortcuts</u>	11
Tools window	15
<u>Tools window</u>	15
<u>Help tool</u>	15
<u>About tool</u>	15
<u>View tool</u>	16
<u>Grab tool</u>	17
<u>Dialog load tool</u>	17
<u>Dialog save tool</u>	18
<u>Stop tool</u>	18
<u>Controls window</u>	19
<u>Controls window</u>	19
<u>Group window</u>	19
<u>Group window</u>	20
<u>Left align controls</u>	20
<u>Bottom align controls</u>	21
<u>Right align controls</u>	22
<u>Top align controls</u>	23
<u>Horizontally center controls</u>	24
<u>Vertically center controls</u>	25
<u>Horizontally space controls</u>	27
<u>Vertically space controls</u>	28
<u>Equal width controls</u>	30
<u>Equal height controls</u>	31
<u>Same style controls</u>	33
<u>Hide controls</u>	34
<u>Show controls</u>	34
<u>Delete controls</u>	34
<u>Size window</u>	34
<u>Size window</u>	35
<u>ID window</u>	35
<u>ID window</u>	35
<u>Name window</u>	36

Table of Contents

<u>Tools window</u>	
<u>Name window</u>	36
<u>Text window</u>	37
<u>Text window</u>	37
<u>Color window</u>	38
<u>Color window</u>	38
<u>DrRexx window</u>	39
<u>DrRexx window</u>	39
<u>Drop-down menu window</u>	40
<u>Drop-down menu window</u>	40
<u>Run-time window</u>	42
<u>Run-time window</u>	43
<u>Dialog select window</u>	44
<u>Dialog select window</u>	44
<u>Background window</u>	45
<u>Managing your DrDialog workspace</u>	47
<u>Invoking DrDialog</u>	49
<u>DrDialog and the Workplace Shell</u>	51
<u>REStoPgm</u>	51
<u>REStoEXE</u>	52
<u>DrRexx</u>	53
<u>The DrRexx notebook</u>	53
<u>Events section</u>	53
<u>Events section</u>	54
<u>Global procedures section</u>	55
<u>Global procedures section</u>	56
<u>Notepad section</u>	56
<u>Notepad section</u>	56
<u>Using the DrRexx editor</u>	57
<u>Using your own editor</u>	58
<u>Writing REXX code for DrRexx</u>	59
<u>The DrRexx execution model</u>	60
<u>DrRexx subcommand environments</u>	61
<u>Error handling in DrRexx</u>	62
<u>Invoking DrRexx</u>	62
<u>Getting started: Your first DrRexx application</u>	63
<u>DrRexx programming techniques</u>	64
<u>Creating a modal dialog</u>	65
<u>Associating data with dialogs and controls</u>	66
<u>Adjusting controls when a dialog is resized</u>	67
<u>Creating and displaying pop-up menus</u>	68
<u>Signaling that data entry is complete</u>	68
<u>Working with dynamic controls</u>	69
<u>Putting user hints into your DrRexx application</u>	70
<u>Preventing dialogs from initially flashing</u>	71

Table of Contents

<u>DrRexx</u>	
<u>DrRexx sample programs</u>	71
<u>DrRexx example programs</u>	72
<u>DrRexx window functions</u>	72
<u>Open</u>	74
<u>Close</u>	75
<u>Owner</u>	76
<u>Frame</u>	77
<u>Hide</u>	77
<u>Show</u>	78
<u>Visible</u>	78
<u>Top</u>	78
<u>Bottom</u>	79
<u>Enable</u>	79
<u>Disable</u>	80
<u>Enabled</u>	80
<u>Focus</u>	80
<u>Position</u>	81
<u>Text</u>	81
<u>Hint</u>	82
<u>Add</u>	83
<u>Add (for a list box or combo box)</u>	83
<u>Add (for a notebook)</u>	83
<u>Add (for a container)</u>	85
<u>Delete</u>	86
<u>Delete (for a list box or combo box)</u>	86
<u>Delete (for a notebook)</u>	86
<u>Delete (for a container)</u>	87
<u>Item</u>	87
<u>Item (for a list box or combo box)</u>	87
<u>Item (for a notebook)</u>	88
<u>Item (for a value set)</u>	89
<u>Item (for a slider)</u>	90
<u>Item (for a container)</u>	91
<u>Select</u>	91
<u>Select (for a list box or combo box)</u>	92
<u>Select (for a single line edit control)</u>	92
<u>Select (for a horizontal or vertical scroll bar)</u>	93
<u>Select (for a spinbutton)</u>	93
<u>Select (for a push button, check box, radio button or bagbutton)</u>	94
<u>Select (for a notebook)</u>	94
<u>Select (for a value set)</u>	95
<u>Select (for a slider)</u>	95
<u>Select (for a container)</u>	96
<u>Range</u>	97
<u>Range (for a dialog)</u>	97
<u>Range (for a single-line edit control)</u>	98
<u>Range (for a horizontal or vertical scroll bar)</u>	98
<u>Range (for a spinbutton)</u>	98
<u>Range (for a value set)</u>	99

Table of Contents

<u>DrRexx</u>	
<u>Range (for a slider)</u>	100
<u>Style</u>	100
<u>Font</u>	101
<u>Color</u>	101
<u>ID</u>	103
<u>Drag</u>	103
<u>Drag (for a container)</u>	106
<u>Drop</u>	107
<u>Drop (for a container)</u>	108
<u>IsDefault</u>	109
<u>Timer</u>	110
<u>View</u>	111
<u>SetStem</u>	113
<u>GetStem</u>	115
<u>Controls</u>	116
<u>Classes</u>	116
<u>DrRexx menu functions</u>	117
<u>MenuPopUp</u>	118
<u>MenuChecked</u>	119
<u>MenuDisabled</u>	119
<u>MenuText</u>	120
<u>DrRexx concurrency functions</u>	120
<u>Start</u>	121
<u>Stop</u>	122
<u>Result</u>	122
<u>Notify</u>	123
<u>Use</u>	124
<u>Val</u>	125
<u>Sleep</u>	126
<u>Concurrent programming example</u>	126
<u>DrRexx miscellaneous functions</u>	129
<u>ModalFor</u>	130
<u>EventData</u>	131
<u>Event</u>	131
<u>Control</u>	131
<u>Class</u>	132
<u>Dialog</u>	132
<u>Dialogs</u>	133
<u>FilePrompt</u>	133
<u>Clipboard</u>	134
<u>ScreenSize</u>	134
<u>DrRexxVersion</u>	135
<u>DrDialog controls</u>	137
<u>Dialog control</u>	137
<u>Push button control</u>	139
<u>Check box control</u>	139
<u>Radio button control</u>	140
<u>Text control</u>	141

Table of Contents

<u>DrDialog controls</u>	
<u>Notebook control</u>	142
<u>Container control</u>	142
<u>List box control</u>	144
<u>Single line edit control</u>	145
<u>Multi-line edit control</u>	146
<u>Combo box control</u>	146
<u>Spin button control</u>	147
<u>Value set control</u>	148
<u>Vertical scroll bar control</u>	149
<u>Horizontal scroll bar control</u>	150
<u>Slider control</u>	150
<u>Group box control</u>	151
<u>Frame control</u>	152
<u>Rectangle control</u>	153
<u>Billboard control</u>	153
<u>Canvas control</u>	154
<u>Paint control</u>	155
<u>Bitmap button control</u>	155
<u>Bagbutton control</u>	156
<u>Turtle control</u>	157
<u>Bitmap control</u>	158
<u>User defined control</u>	158
<u>Marquee control</u>	159
<u>Drop event</u>	160
<u>DrDialog specific controls</u>	163
<u>Billboard controls</u>	163
<u>Canvas controls</u>	164
<u>Paint controls</u>	165
<u>Bitmap button controls</u>	165
<u>Bagbutton controls</u>	166
<u>Turtle controls</u>	168
<u>Turtle control commands</u>	168
<u>Marquee controls</u>	170
<u>DrsAide</u>	173
<u>The DrsAide extension mechanism</u>	173
<u>The DrsAide tool</u>	173
<u>The default DrsAide tool</u>	174
<u>Invoking the default DrsAide tool from DrDialog</u>	174
<u>Invoking the default DrsAide tool from the Workplace Shell</u>	176
<u>DrDialog function</u>	177
<u>DrDialog 'Init' subcommand</u>	178
<u>DrDialog 'Owner' subcommand</u>	178
<u>DrDialog 'Focus' subcommand</u>	179
<u>DrDialog 'GetRES' subcommand</u>	179
<u>DrDialog 'SetRES' subcommand</u>	179
<u>DrDialog 'Filename' subcommand</u>	180
<u>DrDialog 'Modified' subcommand</u>	180

Table of Contents

<u>DrsAide</u>	
<u>DrDialog 'Dialogs' subcommand</u>	180
<u>DrDialog 'Controls' subcommand</u>	180
<u>DrDialog 'Events' subcommand</u>	181
<u>DrDialog 'Globals' subcommand</u>	181
<u>DrDialog 'Global' subcommand</u>	181
<u>DrDialog 'NewDialog' subcommand</u>	181
<u>DrDialog 'NewControl' subcommand</u>	182
<u>DrDialog 'DropDialog' subcommand</u>	182
<u>DrDialog 'DropControl' subcommand</u>	182
<u>DrDialog 'Dialog' subcommand</u>	183
<u>DrDialog 'Control' subcommand</u>	183
<u>DrDialog 'Select' subcommand</u>	183
<u>DrDialog 'Name' subcommand</u>	184
<u>DrDialog 'Text' subcommand</u>	184
<u>DrDialog 'Hint' subcommand</u>	184
<u>DrDialog 'Position' subcommand</u>	185
<u>DrDialog 'Style' subcommand</u>	185
<u>DrDialog 'Font' subcommand</u>	186
<u>DrDialog 'Color' subcommand</u>	186
<u>DrDialog 'Event' subcommand</u>	187
<u>DrDialog 'Class' subcommand</u>	188
<u>DrsAide tools</u>	188
<u>Array tool</u>	189
<u>REView tool</u>	190
<u>RexxUse tool</u>	190
<u>RexxLib tool</u>	190
<u>BMPList tool</u>	191
<u>Writing your own DrsAide tool</u>	192
<u>Adding hints to your DrsAide tool</u>	195
<u>Utilities</u>	197
<u>BMPtoDLL</u>	197
<u>REStoRXX</u>	197
<u>REView</u>	198
<u>REVise</u>	198
<u>User preferences</u>	199
<u>Related packages</u>	201
<u>Acknowledgements</u>	203
<u>Footnote</u>	205
<u>Footnote</u>	207
<u>Footnote</u>	209

Table of Contents

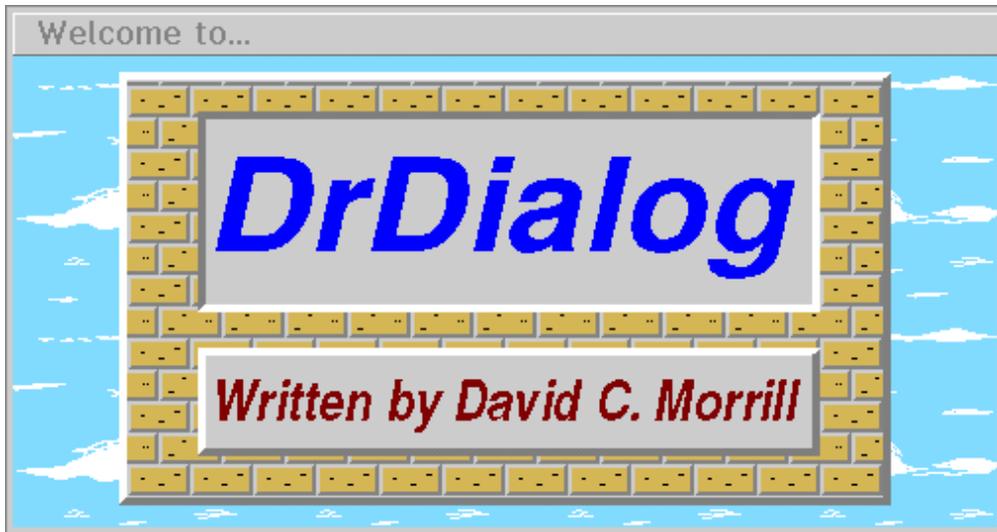
<u>Footnote</u>	211
<u>Footnote</u>	213
<u>Footnote</u>	215
<u>Footnote</u>	217
<u>Footnote</u>	219
<u>Footnote</u>	221
<u>Footnote</u>	223
<u>Footnote</u>	225
<u>Footnote</u>	227
<u>Footnote</u>	229
<u>Footnote</u>	231
<u>Footnote</u>	233
<u>Footnote</u>	235
<u>Footnote</u>	237
<u>Footnote</u>	239
<u>Footnote</u>	241
<u>Footnote</u>	243
<u>Footnote</u>	245
<u>Footnote</u>	247
<u>Footnote</u>	249
<u>Footnote</u>	251
<u>Footnote</u>	253
<u>Footnote</u>	255
<u>Footnote</u>	257

Table of Contents

<u>Footnote</u>	259
<u>Footnote</u>	261
<u>Footnote</u>	263
<u>Footnote</u>	265
<u>Footnote</u>	267
<u>Footnote</u>	269
<u>Footnote</u>	271
<u>Footnote</u>	273
<u>Footnote</u>	275
<u>Footnote</u>	277
<u>Footnote</u>	279
<u>Footnote</u>	281
<u>Footnote</u>	283
<u>Footnote</u>	285
<u>Footnote</u>	287
<u>Footnote</u>	289
<u>Footnote</u>	291
<u>Footnote</u>	293

Introduction

(c) Copyright International Business Machines Corporation 1993.
All rights reserved.



DrDialog is a tool for creating and editing OS/2 Presentation Manager dialogs . It is both powerful and easy to use.

With its **DrRexx** feature, it is also a complete visual programming environment for REXX based applications.

DrDialog can create new dialogs from scratch, as well as import dialog files previously created with the standard OS/2 dialog editor (i.e. **DLGEDIT**). It also has the unique ability to **grab** dialogs, menus and controls right off the screen and import them directly into the editor.

DrDialog creates standard **.RES** and **.DLG** files for use with the OS/2 resource compiler (i.e. **RC**). It can also create equate files for use with various high-level languages (i.e. **.H** files for C or C++, and **.DEF** files for Oberon).

DrDialog has a number of tools to help you create and edit dialogs. Most of the tools can also be invoked in several ways: from a menu bar, pop-up menu, or a toolbar, whichever is most convenient to your style of working.

The available tools include:

Tools This toolbar window contains iconic buttons to display all of the other tool windows and perform global actions for the editor.

ID The ID tool window. This window controls the type of information displayed in each control while the editor is in **ID** mode.

Controls The controls tool window. This window contains icons for each control type supported by the editor. Use button 2 to drag and drop an icon from the controls tool into the edit dialog to create a control of the

selected type .

Group The group tools window. This window contains iconic buttons which perform various operations on the currently selected group of controls within the current edit dialog.

Size The size tool window. This window allows you to display and edit the size and location of the currently active control in the current edit dialog.

Text The text tool window. This window allows you to edit the text and font for the currently active control in the current edit dialog.

Color The color tool window. This window allows you to edit the colors for the currently active control in the current edit dialog.

Name The name tool window. This window allows you to display and edit the names and associated ID's of all controls in the current edit dialog.

DrRexx The DrRexx tool window. This window displays the set of events associated a control in the current edit dialog. It also allows editing the REXX code associated with the current edit dialog.

Menu The drop-down menu tool window. This window allows you to edit the drop-down menu associated with the current edit dialog.

Run-time The run-time tool window. This window allows you to control the execution of the DrRexx application being edited.

Asking for help

This document describes how to use **DrDialog**. It can be read sequentially or in a random order, using the imbedded *hypertext* links. It can also be accessed contextually, directly from **DrDialog**.

To request contextual help for any **DrDialog** tool or window, do the following :

1. Make sure the desired window has the focus (click on its title bar with button 1 if necessary).
2. Position the pointer over the part of the window you want help with.
3. Press **F1** to request help.

If specific help about the icon or control you are pointing at is available, it will be displayed. Otherwise, more general help about the particular **DrDialog** tool or window will be displayed.

Help is also provided by the two controls at the bottom of the DrDialog background window. As the pointer is moved around the screen, the leftmost control displays a description of the DrDialog window the pointer is currently in, while the rightmost control describes the function of the control the pointer is currently over . If the pointer is over an edit dialog, the rightmost control displays the ID number, name, type and hint text for the control pointed at in the form: **ID = name [type] 'hint'**, while the leftmost control indicates whether the edit dialog containing it is active (i.e. the *current* dialog) or inactive.

Editing a dialog

DrDialog is designed to make creating and editing dialogs as simple and intuitive as possible. Wherever possible, it attempts to follow CUA guidelines for selecting and manipulating controls and their attributes.

The editor also allows you to edit more than one dialog at a time. In fact, any number of dialogs can be on the screen at the same time. However, at any given instant, only one dialog is considered to be the *current* edit dialog. This is an important distinction to remember, because all DrDialog tools operate on the current edit dialog. The current, or active, dialog appears *normal*, while all inactive dialogs appear to be subdued, or *greyed out*.

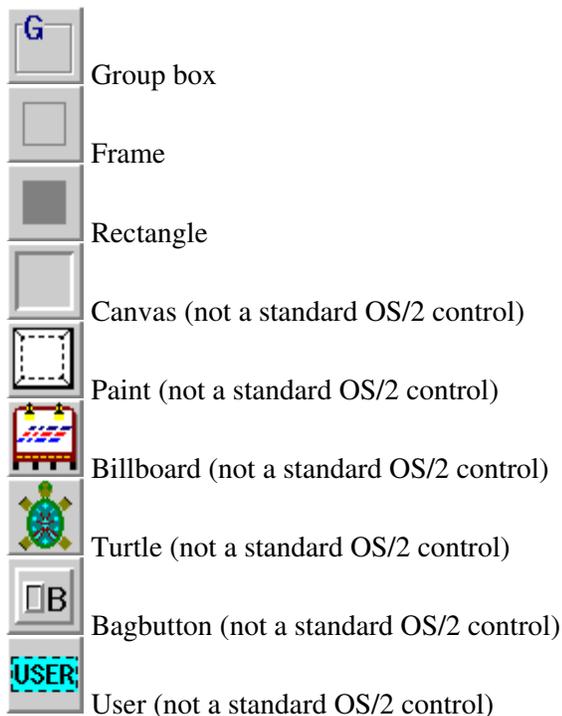
You can make any inactive dialog the current edit dialog by clicking on it with either the left or right mouse button. You can also use the Dialog select tool to select the new current dialog from a list of all dialogs being edited. This is especially handy if the dialog you wish to edit is not currently visible .

Container controls

Controls may be considered to be of two types:

- oControls which convey information or interact with the user (e.g. text fields and push buttons)
- oControls which help to visually organize the first type of control (e.g . group boxes)

In **DrDialog** the second type of control is called a *container* control and has certain special attributes. The control types recognized by the editor as container controls are:



Note: **Billboard**, **turtle**, and **user** controls can either be container or non-container controls, depending upon the setting of the container check box in their respective style dialogs.

Note:

Container controls differ from non-container controls in the following ways :

- oContainer controls are always visually below non-container controls.
- oContainer controls *contain* things (non-container controls and other container controls). When a container control is moved or copied, the controls it contains are moved or copied with it.

Note: For a control to be *contained* within a container control, it must be completely within the bounds of the container control. If any part of the control is outside of the container, then it is not contained within that container .

The notion of being *contained* is important because many **DrDialog** tools perform actions on the controls contained within a container as well as on the container.

Tabbing order

DrDialog automatically determines the tabbing order between controls based on their spatial arrangement within a dialog and on whether or not they are contained within a container control or not:

- oFor controls contained within a container control the tabbing order is top- to-bottom and left-to-right.
- oFor controls not contained within a container control, the tabbing order is left-to-right and top-to-bottom.
- oFor nested containers, each nested container behaves as if it were a single large control for the purpose of determining its tab order within the container it is nested in. Once its position in the tab order is reached, tabbing precedes among the controls it contains using the first rule above. When the last control in the container is reached, tabbing continues with the first control in the next nested container.

Only the lower left hand corner of a control is used in determining the tabbing order of controls.

Note: The exception is a **combo box** control, where the *drop-down* portion of the control is ignored for the purpose of determining the tab order.

Selecting controls

All controls within the current edit dialog are in one of three states:

- oInactive
- oSelected
- oActive

These three states are indicated visually within the editor by the appearance of the *grab handles* drawn on each control when in *edit* mode, and illustrated below :





Most **DrDialog** tools operate on the set of currently selected controls. Selection is performed using button 1 of the mouse. Whenever there is one or more selected controls, one of the selected controls has the additional status of being the **active** control. Normally, this is the first control selected when performing a selection operation.

To select a single control: Click and release button 1 anywhere within the control to be selected. All other previously selected controls revert to the **inactive** state, and the control clicked on is selected and becomes the **active** control.

To select several controls: Press button 1 and drag the pointer over all controls to be selected. As soon as button 1 is pressed, all previously selected controls revert to the **inactive** state. The first control the pointer touches becomes the **active** control, and each subsequent control becomes a **selected** control.

Note: If the pointer starts in a control contained within a container control and later passes into the container, the container will **not** be selected. Conversely, if the pointer starts within a container and later passes over controls contained within the container, the contained controls will **not** be selected. This complicated sounding, yet simple, rule makes it easy to select either containers or the controls contained within them.

To extend the current control selection: Once button 1 of the mouse has been released, **inactive** controls can be added to the current set of selected controls using either of the following methods:

- o Press the **Ctrl** or **Shift** key on the keyboard, then select the additional controls using either of the techniques described above (i.e. either click or perform a **drag** operation with button 1 pressed).
- o Position the pointer over the lower left hand **grab handle** of the first inactive control to be added, then either click or begin a **drag** operation with button 1.

To change the active control: An already **selected** control can be made the **active** control by clicking (i.e. pressing and releasing) button 1 in the lower left hand grab handle of the control. The previous **active** control will switch to the **selected** state.

Note: The dialog itself can only be selected using the first method described above. It will **never** be selected using any of the other methods.

Note: Double-clicking a control with button 1 will both make it the active control and invoke the DrRexx window so that you can edit or view the REXX event handlers associated with the control.

Moving controls

Any control can be moved around within its dialog by placing the pointer over the control and then using button 2 of the mouse to drag the control to its new location.

If the control to be moved is in the **selected** or **active** state, all other selected or active controls are also moved with it. This fact is indicated visually by the size of the tracking rectangle that appears while the mouse is being dragged.

If any of the controls being moved is a container control, all of the controls it contains are also moved.

Note: The entire dialog can also be moved using this technique. Just position the pointer over any part of the dialog not covered by a control (e.g. the **title bar**) and use button 2 to drag the dialog to its new position as described above.

Moving controls between dialogs

Using the technique described in the previous section, controls can only be moved around within the current edit dialog. However, if desired, it is also possible to move controls *between* dialogs.

To do this, first make sure that both dialogs involved in the operation are visible on the screen, and that the dialog containing the controls to be moved is the current edit dialog. Then press the **Shift** key on the keyboard and perform a move operation exactly as described before, making sure that the pointer ends up somewhere over the dialog to which the controls are being moved.

Pressing the **Shift** key removes the bounds keeping the move operation within the current edit dialog and allows you to drag the controls anywhere on the screen. When you release button 2 of the pointer, the editor checks to see which edit dialog is directly under the pointer and moves the controls to that dialog. If the pointer is not over any edit dialog, the editor beeps to indicate an error and does not move the controls.

Copying controls

Any control can be copied by placing the pointer over the control, pressing the **Ctrl** key on the keyboard, and then using button 2 of the mouse to drag a copy of the control to its new location.

If the control to be copied is in the **selected** or **active** state, all other selected or active controls are also copied. This fact is indicated visually by the size of the tracking rectangle that appears while the mouse is being dragged.

If any of the controls being copied is a container control, all of the controls it contains are also copied.

Note: The entire dialog can also be copied using this technique. Just position the pointer over any part of the dialog not covered by a control (e.g. the **title bar**) and use button 2 with the **Ctrl** key pressed to drag a complete copy of the dialog to its new position as described above.

Copying controls between dialogs

Using the technique described in the previous section, controls can only be copied within the current edit dialog. However, if desired, it is also possible to copy controls *between* dialogs.

To do this, first make sure that both dialogs involved in the operation are visible on the screen, and that the dialog containing the controls to be copied is the current edit dialog. Then press the **Shift** key on the keyboard and perform a copy operation exactly as described before, making sure that the pointer ends up somewhere over the dialog to which the controls are being copied.

Pressing the **Shift** key removes the bounds keeping the copy operation within the current edit dialog and allows you to drag the controls anywhere on the screen . When you release button 2 of the pointer, the editor checks to see which edit dialog is directly under the pointer and copies the controls to that dialog. If the pointer is not over any edit dialog, the editor beeps to indicate an error and does not copy any controls.

Sizing controls

The currently **active** control can be resized by placing the mouse pointer over one of the eight *grab handles* and using button 2 of the mouse to drag a tracking rectangle into the desired shape. The fact that the mouse is over one of the *grab handles* is indicated by a change in the shape of the mouse pointer to reflect the directions the control can be resized in.

If the control being resized is a container control, only the container itself is affected by the resize operation. However, if the **Ctrl** key on the keyboard is being pressed at the start of the resizing operation, all controls contained within the container are resized proportionally also.

Editing a control's style information

Each control has associated with it style information that can affect the appearance and behavior of the control. The style information for a control can be edited by selecting the **Style...** option from the pop-up menu that appears after clicking button 2 while the pointer is over the control (the control need not be selected first).

Changes made to the style information using the pop-up dialog that appears will be reflected immediately in the appearance of the control. Once all changes have been made, they can be finalized either by pressing the **Enter** key or clicking the **OK** button at the bottom of the dialog. Alternatively, the previous style information can be restored by clicking on the **Cancel** button. In either case, the pop- up dialog will be removed from the display.

Note: Only one pop-up dialog exists per control type. If the pop-up dialog for a particular control type is already being displayed when a new control of the same type is selected, the style information for the newly selected control will replace the previous style information within the pop-up dialog. This allows the style information for a collection of identical control types to be edited very quickly and easily. The same is true if button 2 is clicked within a control of the type corresponding to the pop-up dialog.

Editing a control's REXX code

Each control optionally has associated with it REXX code to process various events that occur for the control (e.g. a push button *click* event). The REXX code for each control is edited using the **DrRexx** tool window. The editor provides a quick path to this tool by **double-clicking** a control in the edit dialog. The DrRexx tool window will appear with the set of event pages for the clicked on control already displayed. To select the REXX code for a particular event, click on its corresponding page tab and the REXX code associated with the event will be displayed for editing.

Note: If the number of events defined for a particular control is large , it may be necessary to use the scroll buttons on the side of the DrRexx notebook in order to scroll all of the event page tabs into view.

Editing a control's attributes using the pop-up menu

While all of a control's attributes can be edited using the various tools provided by DrDialog, the editor also provides a means to quickly and easily modify most of a control's attributes using a context sensitive pop-up menu.

To use the pop-up menu, first position the pointer over the control whose attributes are to be changed, then click (i.e. press and release) button 2 of the mouse.

The pop-up menu's options are divided into three categories:

- oGlobal
- oControl sensitive
- oGroup sensitive

The **global** options at the top of the pop-up menu are not specific to the control over which the pointer is positioned. The two global options are:

Tools This submenu contains icons for each of the DrDialog tool windows. Selecting an icon will display the corresponding tool window. This submenu contains the same set of icons displayed in the **Tools** submenu of the DrDialog menu bar.

Controls This submenu contains icons for each type of DrDialog control. Selecting an icon will create a corresponding control centered at the point where the pointer was when the pop-up menu was invoked. This submenu contains the same set of icons displayed in the **Controls** submenu of the DrDialog menu bar and in the Control window.

The **control sensitive** options in the middle of the pop-up menu operate on the control the pointer was positioned over when the pop-up menu was invoked. The control sensitive options are:

Remove This submenu contains two icons. The  icon hides the specified control. The control can be made visible again using the  button available in the Group window or pop-up menu option. The  icon deletes the specified control. If the specified control is the dialog itself, you will be prompted whether you wish to delete the entire dialog or not.

Events This submenu lists all DrRexx events defined for the specified control . Selecting an option from the submenu will cause the DrRexx window to appear with the specified event page already displayed.

Hint Displays a pop-up dialog that allows you to change the hint text for the specified control. The hint text will be displayed at run-time whenever the pointer passes over the specified control.

Text Displays a pop-up dialog that allows you to change the text for the specified control.

Style Displays a pop-up dialog that allows you to change the style for the specified control.

Name Displays a pop-up dialog that allows you to change the name of the specified control.

ID Displays a pop-up dialog that allows you to change the numeric ID of the specified control.

Color Displays a pop-up dialog that allows you to change the foreground and background color of the specified control. If you need to change more than the foreground or background color of the control, you must use the Color window.

Font Displays a pop-up dialog that allows you to change the font for the specified control.

Adjust Displays a pop-up dialog that allows you to change the position or size of the specified control one pel at a time. Whether the size or position of the control is adjusted depends upon the position of the pointer at the time

the pop-up menu was invoked. If the pointer was over one of the *grab handles* for the control, the size of the control will be adjusted. If the pointer was anywhere else over the control, the position of the control will be adjusted.

Note: If the pointer is over a control that is selected, all other selected controls are also adjusted at the same time. This makes it easy to move an entire group of controls around one pixel at a time.

The **group sensitive** option at the bottom of the pop-up menu operates on the group of currently selected controls and is enabled only if the control the pointer is positioned over is selected or active. The group sensitive option is:

Group Displays a pop-up dialog containing iconic buttons for aligning, spacing, sizing, hiding, showing, deleting and setting the styles of the current group of selected controls. These are the same buttons that can also be found in the Group window.

Any pop-up dialog that appears as a result of selecting a pop-up menu option will automatically be removed when any other DrDialog window is given the focus or a new pop-up menu is requested. It can also be removed explicitly by clicking the  or  buttons in the dialog.

Keyboard shortcuts

Many of the DrDialog editing operations that can be performed using the pointer can also be performed using the keyboard. These keyboard *shortcuts* can be separated into several different groups based on the type of operation they perform :

Navigation and selection

< Move the pointer left.

> Move the pointer right.

^ Move the pointer up.

| Move the pointer down.

Enter Select the pointed at control (and unselect all other controls).

Ctrl-Enter Add the pointed at control to the current selection. If the control is already selected, then make it the active control.

Tab Select the next control in the same *container* as the currently active control. The newly selected control is made the active control and all previously selected controls are unselected. If the current active control is the dialog frame, the next dialog in the ring of dialogs being edited is made the current edit dialog . If no control is currently selected, no action is performed.

Shift-Tab Select the previous control in the same *container* as the currently active control.

Ctrl-Tab Add the next control in the same *container* as the currently active control to the selection and make it the new active control. If no control is currently selected, no action is performed.

Ctrl-Shift-Tab Add the previous control in the same *container* as the currently active control to the selection and make it the new active control. If no control is currently selected, no action is performed.

Home Select the current dialog's frame (and unselect all other controls) .

Page Up Select the control *containing* the current active control (and unselect all other controls). If no control is currently selected, or the dialog frame is currently selected, no action is performed.

Page Down Select the first control contained within the currently active control (and unselect all other controls). If no control is currently selected, or the currently active control does not contain any controls, no action is performed .

Esc Unselect all currently selected controls.

Editing

Ctrl-< Move the pointed at control left. If the control is part of the current selection, all selected controls are moved. If the pointed at control is the dialog frame, the entire dialog is moved.

Ctrl-> Move the pointed at control right.

Ctrl-^ Move the pointed at control up.

Ctrl-| Move the pointed at control down.

Alt-< Left align all currently selected controls with the current active control .

Alt-> Right align all currently selected controls with the current active control.

Alt-^ Top align all currently selected controls with the current active control .

Alt-| Bottom align all currently selected controls with the current active control.

| Horizontally center align all currently selected controls with the current active control.

_ (Underscore) Vertically center align all currently selected controls with the current active control.

H Evenly space horizontally all currently selected controls with respect to the current active control (see [Horizontally space controls](#) for more information on this operation).

V Evenly space vertically all currently selected controls with respect to the current active control (see [Vertically space controls](#) for more information on this operation).

X Make all currently selected controls have the same width as the current active control.

Y Make all currently selected controls have the same height as the current active control.

\ Make all currently selected controls have the same style as the current active control.

= Make all currently selected controls have the same height, width and style as the current active control.

? Display a pop-up dialog allowing the hint text for the pointed at control to be edited.

T Display a pop-up dialog allowing the text for the pointed at control to be edited.

N Display a pop-up dialog allowing the name of the pointed at control to be edited.

I Display a pop-up dialog allowing the numeric ID of the pointed at control to be edited.

C Display a pop-up dialog allowing the foreground and background colors for the pointed at control to be edited.

F Display a pop-up dialog allowing the font for the pointed at control to be edited.

S Display a pop-up dialog allowing the style of the pointed at control to be edited.

A Display a pop-up dialog allowing the position or size of the pointed at control to be adjusted. If the pointer is over one of the control's *grab* handles, the size of the control will be adjusted; otherwise its position will be adjusted.

G Display a pop-up dialog permitting operations that work on the currently selected group of controls to be performed.

E Display a pop-up menu of all events defined for the pointed at control. Selecting an item will display the corresponding event handling code for the pointed at control.

Enter (the **Enter** key on the numeric keypad). Display a pop-up menu of all events defined for the pointed at control. Selecting an item will display the corresponding event handling code for the pointed at control.

Space Make a copy of the pointed at control and select it (and unselect all other controls). If the control pointed at is the dialog frame, no action is performed (use Ctrl-Ins if you wish to copy an entire dialog). Note that the copied control will appear on top of the original control.

End Display the DrDialog context-sensitive pop-up menu (see the section on editing a control's attributes using the [pop-up](#) menu for more information).

B Display a pop-up menu of all DrDialog tools. Selecting an item from the menu will invoke the selected tool.

Ins Display a pop-up menu of all controls. Selecting an item from the menu will insert a new control of the selected type centered on the position originally pointed at.

Ctrl-Ins Make a copy of the pointed at control and all controls it contains. If the pointed at control is part of the current selection, all other controls (and the controls they contain) are also copied. If the pointed at control is the dialog frame, the entire dialog is copied. Note that the copy will appear on top of the original controls or dialog.

Del Deletes all selected controls. The pointer must be over one of the controls in the current selection for the operation to proceed. If the pointed at control is the dialog frame, you will be prompted first if you wish to delete the entire dialog. If any of the selected controls have associated event handlers, you will also be prompted first if you wish to delete all of the controls.

- Hide the pointed at control. If the control pointed at is the dialog frame , the entire dialog is hidden, and the next dialog in the ring of dialogs being edited is selected for editing. If the pointed at control is part of the current selection, all selected controls are hidden.
- + Show all currently hidden controls.

Tool selection

- Ctrl-F1 Display the Tools window.
- Ctrl-F2 Display the Controls window.
- Ctrl-F3 Display the Group window.
- Ctrl-F4 Display the Dialog select window.
- Ctrl-F5 Display the Run-time window.
- Ctrl-F6 Display the DrRexx window.
- Ctrl-F7 Display the Menu window.
- Ctrl-F8 Display the ID window.
- Ctrl-F9 Display the Text window.
- Ctrl-F10 Display the Color window.
- Ctrl-F11 Display the Size window.
- Ctrl-F12 Display the DrsAide window.
- F12 Display the current edit dialog (i.e. bring it to the front).

Control creation

When any of the keys in this section are pressed, a new control of the specified type is created centered on the current pointer position. Note that the uppercase character in the description of the control is the *mnemonic* for the control .

- Alt-A pAint control
- Alt-B Billboard control
- Alt-C Check box control
- Alt-D combo-box control (i.e. Drop-down listbox control)
- Alt-E single-line Edit control
- Alt-F Frame control
- Alt-G Group box control
- Alt-H canvas (i.e. Holder) control
- Alt-I Icon button control
- Alt-J turtle control
- Alt-K bitmap control
- Alt-L List box control
- Alt-M Multi-line edit control
- Alt-O cOntainer control
- Alt-P Push button control
- Alt-Q marQueue control
- Alt-R Radio button control
- Alt-S Spin button control
- Alt-T Text control
- Alt-U User defined control
- Alt-V Value set control
- Alt-W bagbutton control
- Alt-X horizontal scroll bar control
- Alt-Y vertical scroll bar control
- Alt-Z slider control

Alt-[rectangle control

Miscellaneous

F2 Save the current set of dialogs in the file they were loaded from.

Alt-F2 Save the current set of dialogs in a file you select using the file prompt dialog.

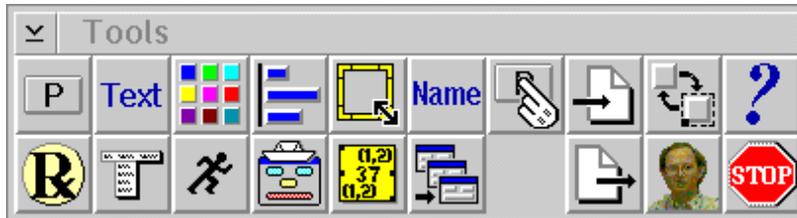
F3 Load an existing set of dialogs into the editor using the file prompt dialog to select the file to be loaded. If any changes have been made to the current set of dialogs, you will be prompted whether you wish to save the changes prior to loading the new file.

F5 Run the current DrRexx application under control of DrDialog. The Run-time window will appear and the application will automatically begin execution. If the **Auto-save** option has been enabled, the current set of dialogs will be saved to a file prior to beginning execution of the application.

q Quit (i.e. exit) DrDialog. If any changes have been made to the current set of dialogs, you will be prompted whether you wish to save the changes prior to terminating DrDialog.

/ Toggle the current viewing mode between *edit* and *view* mode (see the View tool for more information).

Tools window



Tools window

The **tools** window is a *toolbar* (i.e. array of iconic buttons). Each button in the toolbar either activates another editor tool window or performs an action global to the operation of the editor.

When a button representing a tool window is clicked, the corresponding tool window is displayed in the position it last had.

As an alternative to using the tool window, all of the same functions are also available using the **Tools** submenu of either the DrDialog menu bar or the pop-up menu.

To learn more about the function of a particular tool window button, double-click on the image of the button displayed in the other window.

Help tool

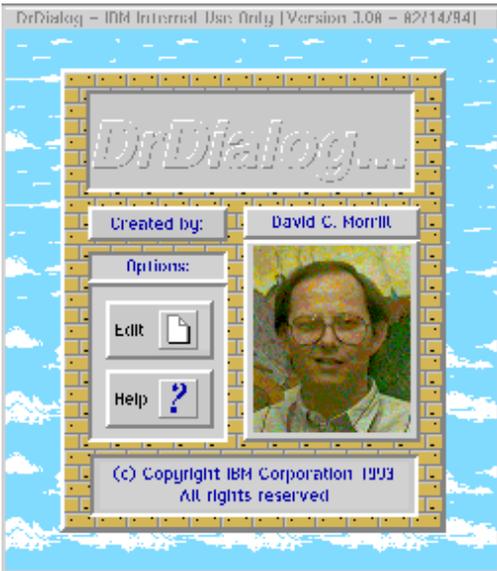


Clicking the **help** button causes this document to be displayed.

About tool



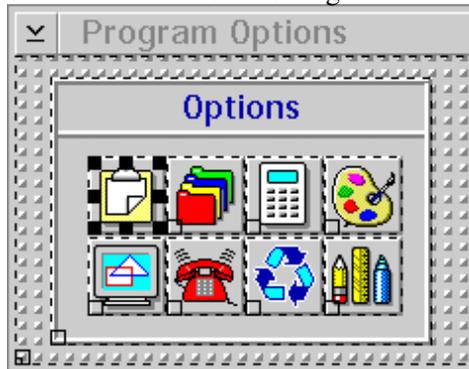
Clicking the **about** button causes the **DrDialog** logo window to be displayed :



View tool



Clicking the **view** button toggles all dialogs being edited between *edit* and *view* mode. In edit mode (the initial mode of the editor), each dialog control is drawn with a dashed line surrounding it to indicate the bounds of the control. In addition, one or more *grab handles* are drawn for use in sizing and selecting the control. A typical



dialog in edit mode might appear as follows:

In view mode, each dialog control appears as it would in normal use, with no additional lines or grab handles drawn over it. Note however that the grab handles are still logically there, and may be used in exactly the same manner as edit mode. A typical dialog in view mode might appear as follows:



Grab tool



Clicking the **grab** button activates *grab* mode. In grab mode the mouse pointer changes to a pointing hand. Positioning the pointer over any dialog frame, menu or control currently on the screen and dragging it with button 2 into the editor will import the specified object directly into the editor.

Based on what you grab, there are several possible results:

oIf you grab the frame of a dialog, the entire dialog and all of its controls , menus and submenus will be imported into the editor as the new current edit dialog.

oIf you grab a control, the control will be added to the current edit dialog at the point where you release button 2. If the drag outline does not intersect the current edit dialog when button 2 is released, a beep sounds and the control is not added to the edit dialog.

oIf you grab a menu bar, the menu and all of its submenus will become the current edit dialog's menu. If the current edit dialog already has a menu, you will first be prompted if you wish to replace it with the menu you just grabbed. If the grabbed menu becomes the current edit dialog's menu, the Menu window is automatically displayed to allow you to edit the menu just imported.

If you decide not to grab anything once in grab mode, click button 1 of the mouse to exit grab mode without taking any action. Note however that you will receive an informational message indicating that you should use button 2 to drag the desired object.

Note: If you attempt to grab something that cannot be copied, the editor will beep and exit grab mode without taking any action.

Dialog load tool



The **dialog load** button loads a resource (i.e. **.RES**) file containing one or more dialogs into the editor. If any changes have been made to the current set of dialogs, you will be asked if you wish to discard the changes and proceed with loading a new resource file.

A standard file dialog will be displayed. Enter the name of the file to be edited and press **Enter**, or click on the **Open** button to continue. Alternatively, you can click on the **Cancel** button to return to the editor without loading a new resource file.

If the selected file exists and is a valid resource file, the dialogs it contains will be loaded into the editor for further editing. The editor will automatically select the first dialog found in the file as the current dialog.

If, in addition, the resource file was last edited using **DrDialog**, any symbols assigned to controls or dialogs will automatically be loaded into the editor . If the file was last edited using a different dialog editor (e.g. **DLGEDIT**), DrDialog will check for an include file with a **.H** extension in the same directory as the resource file being loaded. If the file exists, the editor will use any *#define* statements found in the **.H** file to define symbolic names for the controls and dialogs found in the resource file.

If no corresponding **.H** file is found, and the *Prompt for .H file (if necessary)* option was checked in the file open dialog, DrDialog will prompt you for the name of the include file containing the symbolic names. If the option was not checked, the resource file will be loaded with no symbolic names initially defined.

Note: An existing resource file can also be loaded into the editor by selecting the **Open...** option from the **File** submenu of the DrDialog window menu bar.

Dialog save tool



The **dialog save** button saves information about the current set of dialogs into one or more files.

A standard file dialog will be displayed. Enter the name of the file to be used for saving the dialogs. You should also specify what type of dialog information is to be saved, and for what language, using the check boxes and radio buttons located near the bottom of the file dialog. The available choices for type of information to be saved are:

Resource Save the current dialogs in the OS/2 standard binary **.RES** file format.

Dialog Save the current dialogs in the OS/2 standard text **.DLG** file format .

Equates Save the name and ID information for the current dialogs in a format suitable for use with the selected language. For C or C++, this is a **.H** file containing statements of the form: **#define name ID**. For Oberon, this is a **.DEF** file containing **CONST**'s of the form: **name* = ID;**.

WinProc Not currently implemented (has no effect).

Note: When entering the name of the file to be saved, it is not necessary to specify the file extension. A file extension will be provided for each type of information to be saved:

.RES For resource files

.DLG For dialog files

.H For C or C++ equate files

.DEF For Oberon equate files

Once the file name has been entered, and the type of information to be saved has been specified, press **Enter** or click on the **Save** button to continue saving the requested files. Alternatively, you can click on the **Cancel** button to return to the editor without writing any files.

Note: The names associated with the dialog controls are save as a special resource type within the **.RES** file. This allows the names to be recovered the next time the resource file is loaded into the editor. Editing the dialog with a different dialog editor may cause the names assigned to the various controls and dialogs to be lost.

Note: The current set of dialogs can also be saved to a file using the **Save** or **Save as...** options in the **File** submenu of the DrDialog window menu bar.

Stop tool



Clicking the **stop** button terminates **DrDialog**.

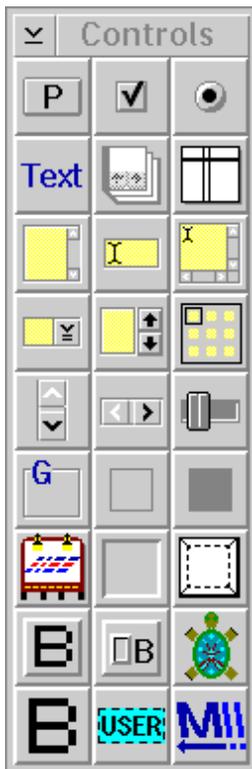
If a resource file with unsaved changes is being edited at the time the stop button is clicked, you will be prompted first to verify that you wish to discard the current file before terminating the editor.

Controls window

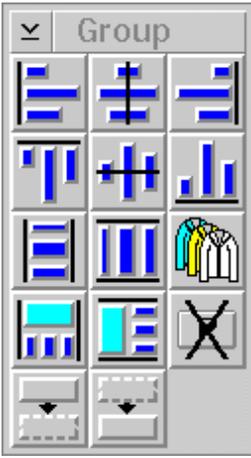
The **controls** window shows all the controls that can be created using **DrDialog**. To create a control of a particular type, use button 2 of the mouse to drag and drop from the icon representing the desired control to the location in the edit dialog where you want to place the control. Once the control has been created, it will automatically be made active for sizing or other editing operations.

To learn more about the individual controls that can be created using the controls window, double-click on the image of the icon displayed in the other window.

Controls window



Group window



Group window

The **group** window is a collection of tools for aligning, sizing, spacing, hiding, showing, deleting and setting the styles of groups of controls. It consists of an array of iconic buttons which operate either on the group of currently selected controls or controls contained within the currently selected control.

Note: In the case of the **Show controls** button, it operates on the group of currently hidden controls.

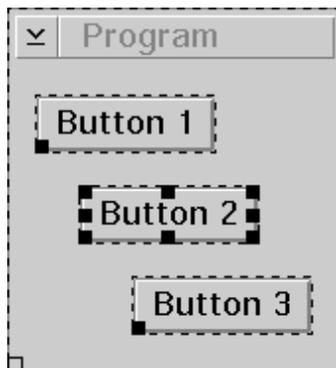
The set of functions available through the group window is also available through the **Group** submenu of either the DrDialog window or the pop-up menu.

To learn more about the function of a particular group window button, double-click on the image of the button displayed in the other window.

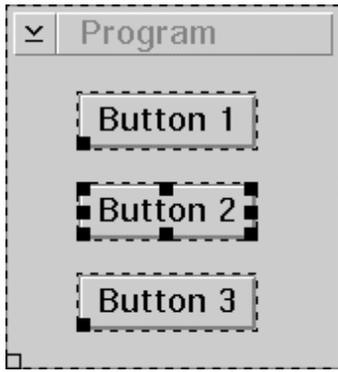
Left align controls



Clicking the **left align** button aligns all currently selected controls flush with the left edge of the current active control.

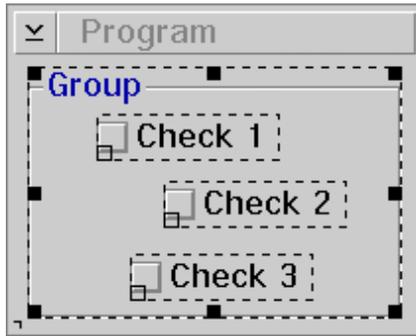


For example, before:

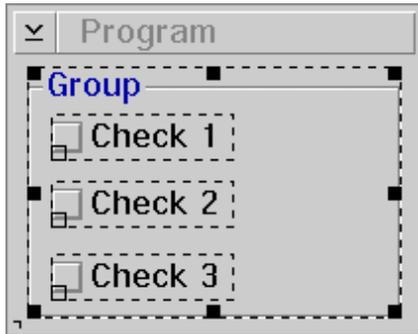


and after:

If only one control is selected, and it is a container control, all the controls it contains are aligned with the left margin of the container. The left margin of the container is inset from the left edge of the control by an amount dependent upon the type of container control.



For example, before:



and after:

Bottom align controls



Clicking the **bottom align** button aligns all currently selected controls flush with the bottom edge of the current active control.

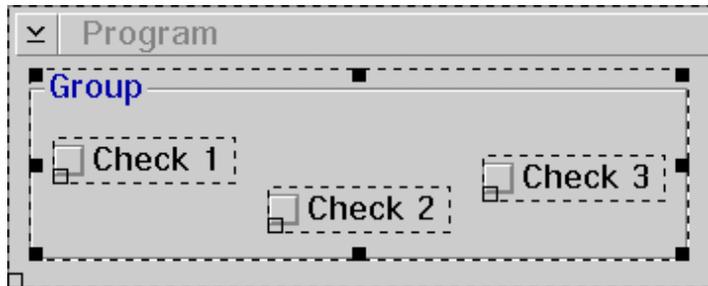


For example, before:

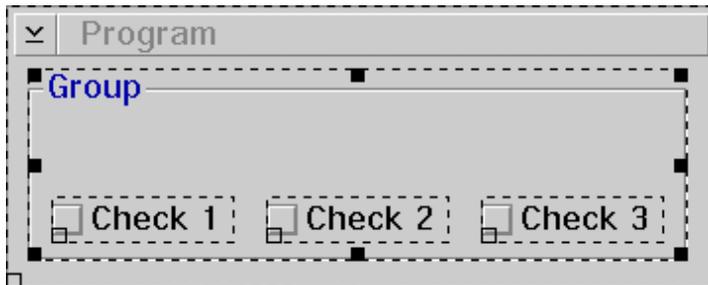


and after:

If only one control is selected, and it is a container control, all the controls it contains are aligned with the bottom margin of the container. The bottom margin of the container is inset from the bottom edge of the control by an amount dependent upon the type of container control.



For example, before:

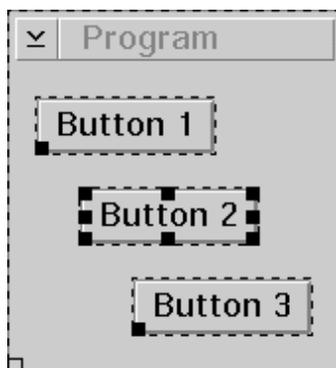


and after:

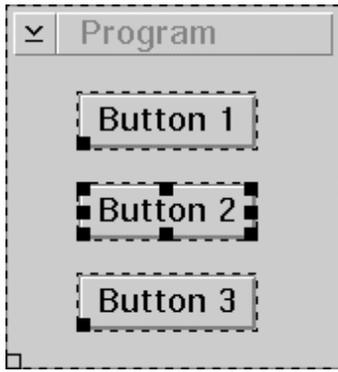
Right align controls



Clicking the **right align** button aligns all currently selected controls flush with the right edge of the current active control.

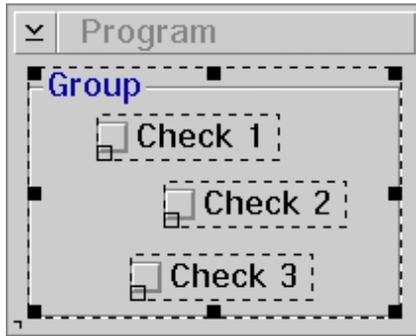


For example, before:

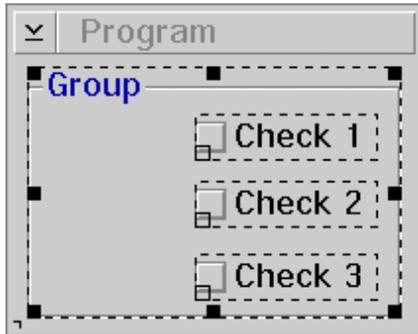


and after:

If only one control is selected, and it is a container control, all the controls it contains are aligned with the right margin of the container. The right margin of the container is inset from the right edge of the control by an amount dependent upon the type of container control.



For example, before:



and after:

Top align controls



Clicking the **top align** button aligns all currently selected controls flush with the top edge of the current active control.

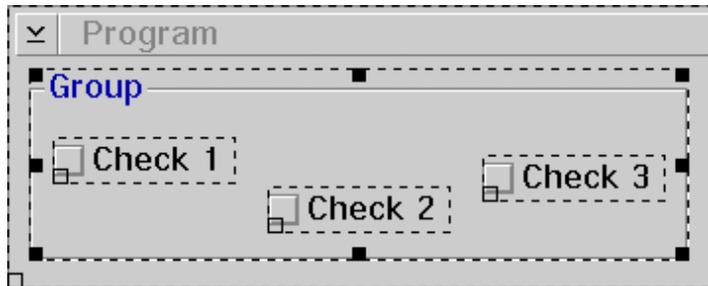


For example, before:

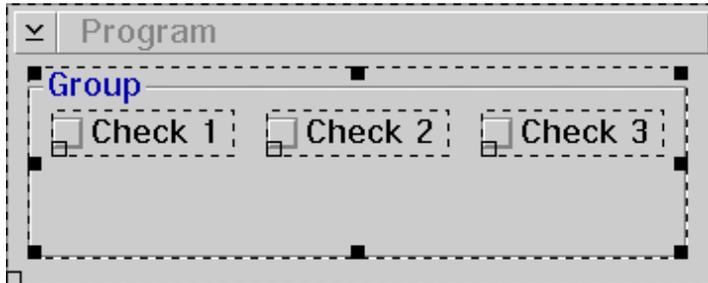


and after:

If only one control is selected, and it is a container control, all the controls it contains are aligned with the top margin of the container. The top margin of the container is inset from the top edge of the control by an amount dependent upon the type of container control.



For example, before:

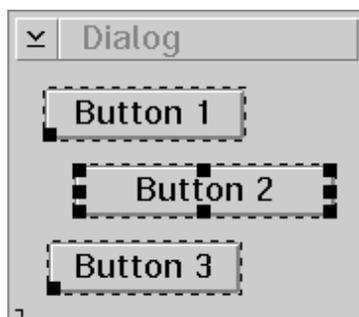


and after:

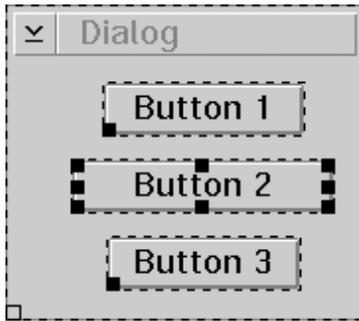
Horizontally center controls



Clicking the **horizontally center** button aligns the horizontal midpoint of all currently selected controls with the horizontal midpoint of the current active control .

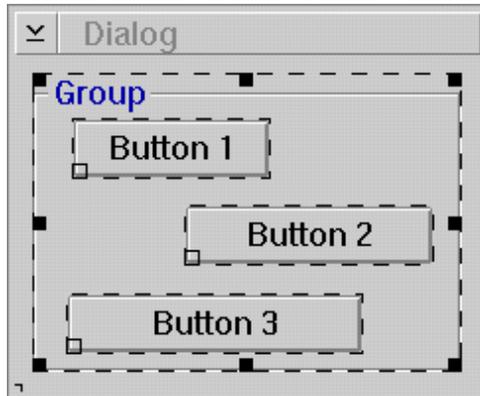


For example, before:

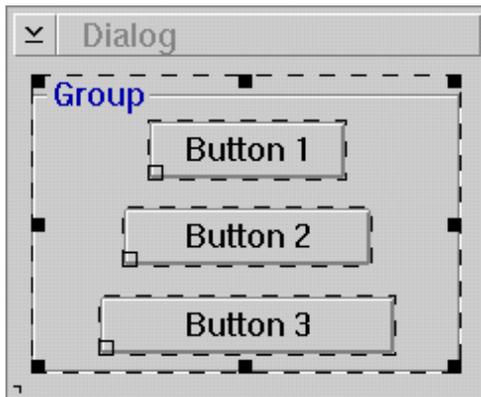


and after:

If only one control is selected, and it is a container control, the horizontal midpoints of all controls it contains are aligned with the horizontal midpoint of the container.



For example, before:

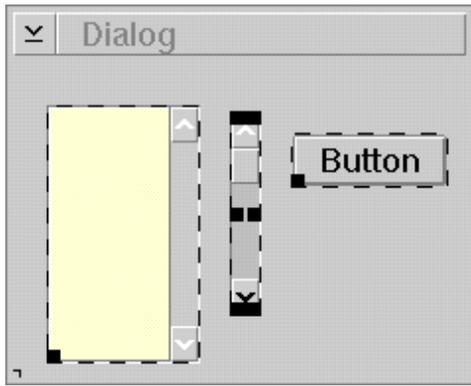


and after:

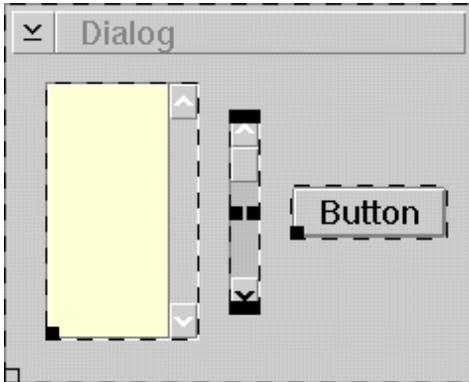
Vertically center controls



Clicking the **vertically center** button aligns the vertical midpoint of all currently selected controls with the vertical midpoint of the current active control.

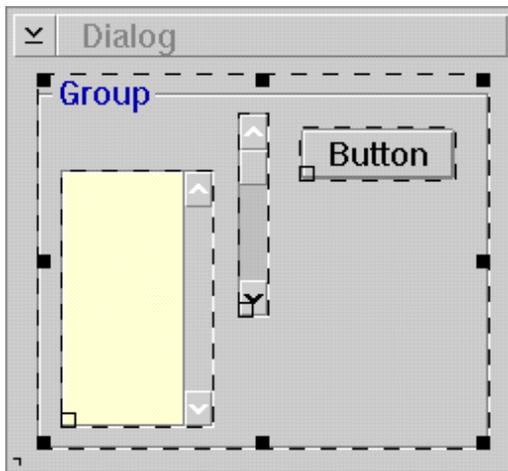


For example, before: ▢

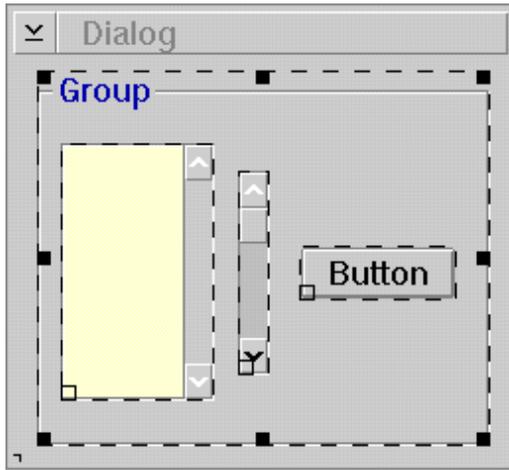


and after: ▢

If only one control is selected, and it is a container control, the vertical midpoints of all controls it contains are aligned with the vertical midpoint of the container.



For example, before: ▢



and after: ▾

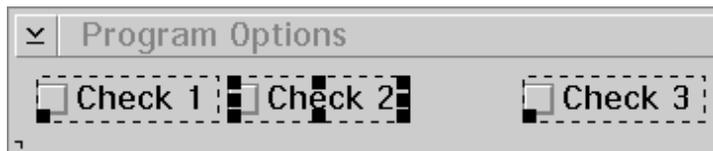
Horizontally space controls



Clicking the **horizontally space** button spaces all currently selected controls evenly horizontally. *Evenly* means that the amount of space between each control is the same.

There are three different cases to consider:

Case 1: There exists a horizontal line that intersects both the active control and at least one other selected control. In this case the leftmost and rightmost controls act as *anchors* and all the other controls (including the active control) are spaced evenly between them. The spacing will preserve the original relative ordering between the controls.

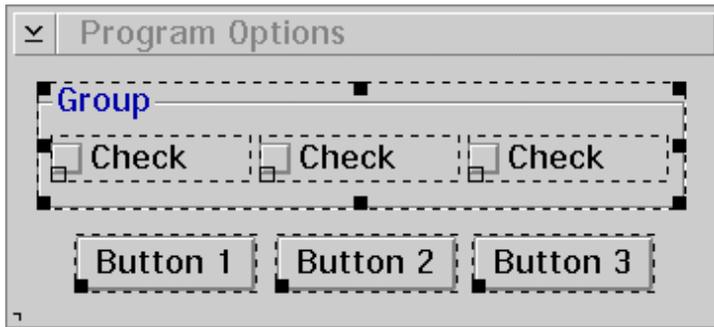


For example, before: ▾

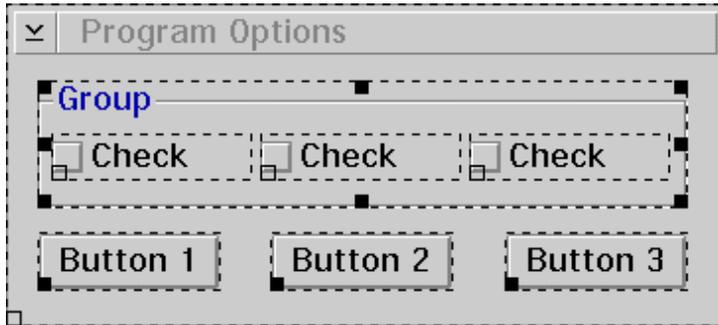


and after: □

Case 2: There is no horizontal line that intersects both the active control and at least one other selected control. In this case the active control acts as an *anchor* and all the other controls are spaced evenly between the left and right ends of the active control. The spacing will preserve the original relative ordering between the selected controls.

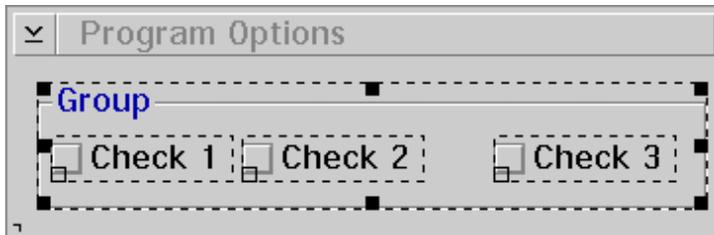


For example, before:

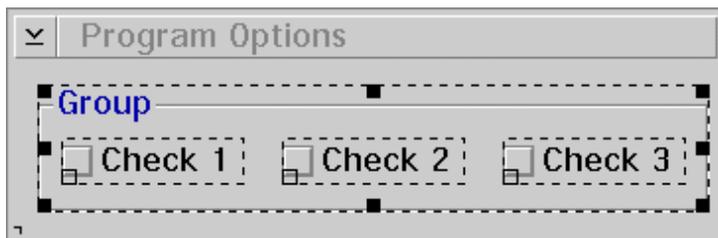


and after:

Case 3: **Only one control is selected, and it is a container control** . In this case, all the controls inside of the container are spaced evenly between the left and right margins of the container. The left and right margins of the container are inset from the left and right edges of the container by an amount dependent upon the type of container control. Note that the spacing is performed separately for each group of vertically aligned controls within the container. While this may sound complicated, it actually facilitates the rapid alignment of arrays of controls within a container (try it, you'll like it!).



For example, before:



and after:

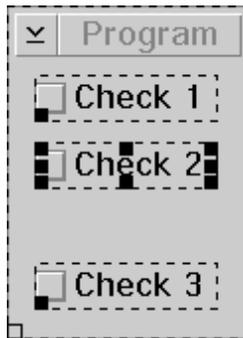
Vertically space controls



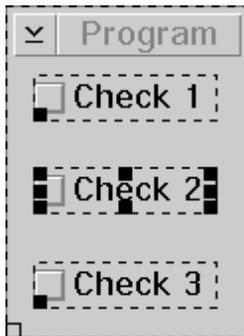
Clicking the **vertically space** button spaces all currently selected controls evenly vertically. *Evenly* means that the amount of space between each control is the same.

There are three different cases to consider:

Case 1: **There exists a vertical line that intersects both the active control and at least one other selected control.** In this case the topmost and bottommost controls act as *anchors* and all the other controls (including the active control) are spaced evenly between them. The spacing will preserve the original relative ordering between the controls.

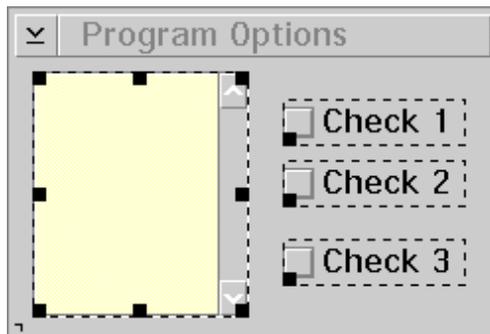


For example, before:

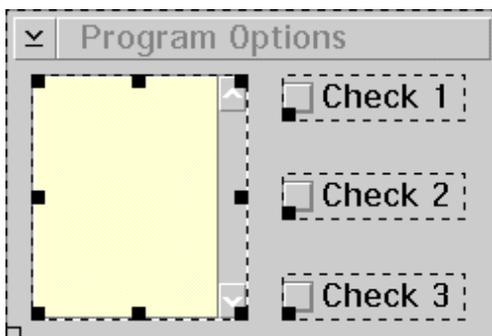


and after:

Case 2: **There is no vertical line that intersects both the active control and at least one other selected control.** In this case the active control acts as an *anchor* and all the other controls are spaced evenly between the top and bottom ends of the active control. The spacing will preserve the original relative ordering between the selected controls.

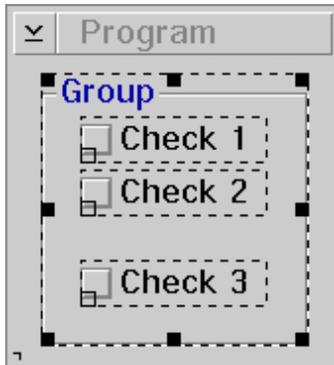


For example, before:

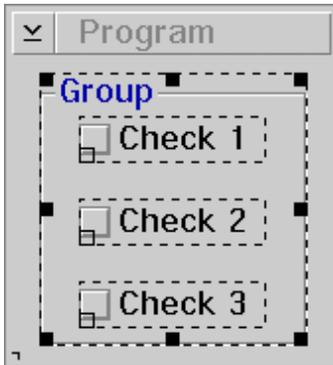


and after:

Case 3: **Only one control is selected, and it is a container control** . In this case, all the controls inside of the container are spaced evenly between the top and bottom margins of the container. The top and bottom margins of the container are inset from the top and bottom edges of the container by an amount dependent upon the type of container control. Note that the spacing is performed separately for each group of horizontally aligned controls within the container. While this may sound complicated, it actually facilitates the rapid alignment of arrays of controls within a container (try it, you'll like it!).



For example, before:

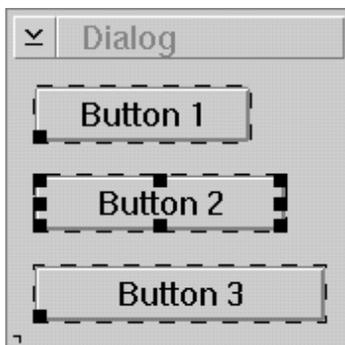


and after:

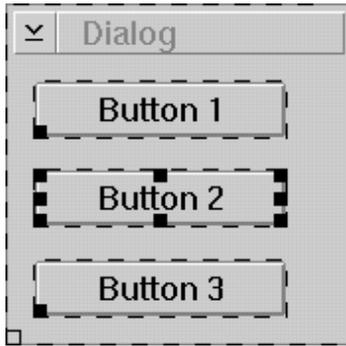
Equal width controls



Clicking the **equal width** button makes all currently selected controls the same width as the current active control.

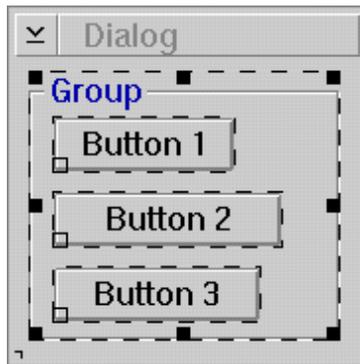


For example, before:

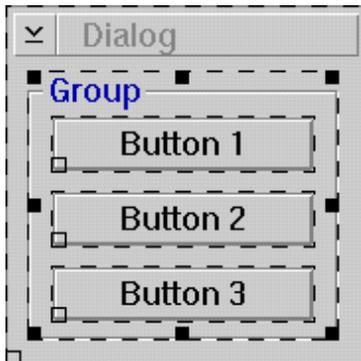


and after:

If only one control is selected, and it is a container control, all the controls it contains are made the same width as the distance between the left and right margins of the container control. The left and right margins of the container are inset from the left and right edges of the control by an amount dependent upon the type of container control.



For example, before:

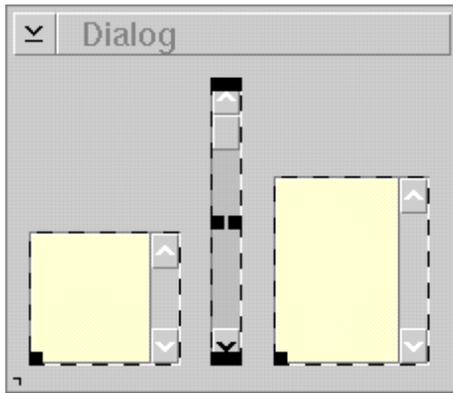


and after:

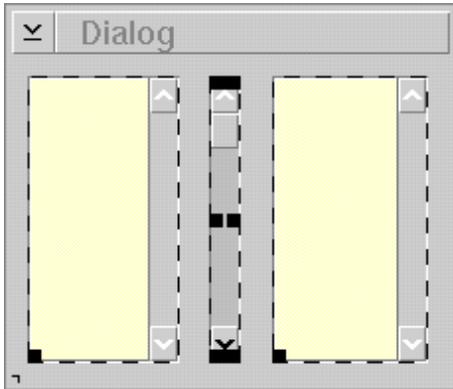
Equal height controls



Clicking the **equal height** button makes all currently selected controls the same height as the current active control.

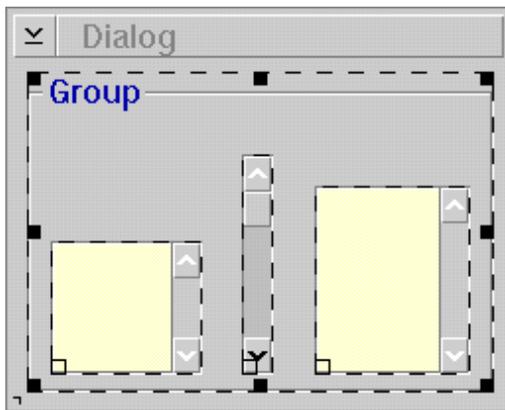


For example, before: ▢

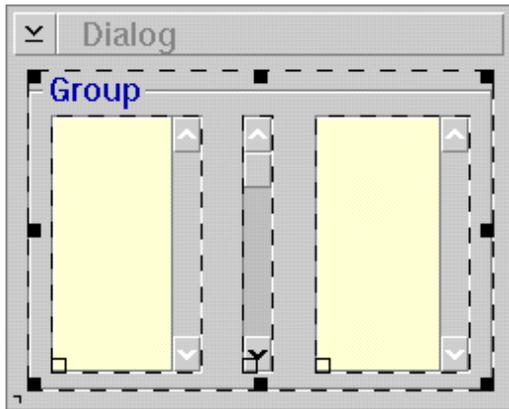


and after: ▢

If only one control is selected, and it is a container control, all the controls it contains are made the same height as the distance between the top and bottom margins of the container control. The top and bottom margins of the container are inset from the top and bottom edges of the control by an amount dependent upon the type of container control.



For example, before: ▢



and after:

Same style controls

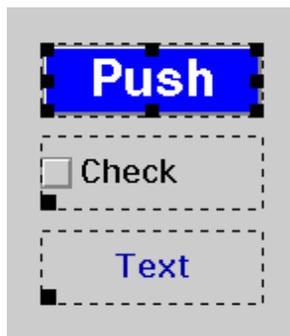


Clicking the **same style** button makes all currently selected controls have the same attributes as the current active control.

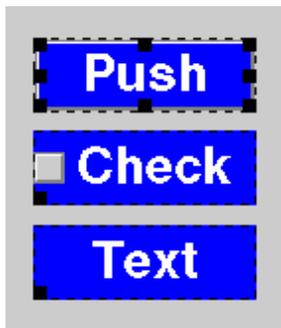
The attribute information copied from the active control to each selected control consists of:

- oColor
- oFont
- oStyle (as set in the active control's pop-up style dialog)

Note: The **style** information is copied to a selected control only if it is the same type of control as the active control. The **color** and **font** information is copied to a selected control only if the selected control can accept it.



For example, before:



and after:

Hide controls



Clicking the **hide** button hides the currently selected controls in the edit dialog. Note that the controls are not deleted, but are simply hidden. This can be useful if you are trying to operate on a control which is being obscured by other controls in front of it.



All hidden controls can be made visible again by clicking on the  button.

Note: If the currently selected control is the dialog itself, the entire dialog is hidden. To make it visible again, it must be selected for editing using the Dialog select window.

Show controls



Clicking the **show** button shows all controls within the current edit dialog that were previously hidden using the



 button.

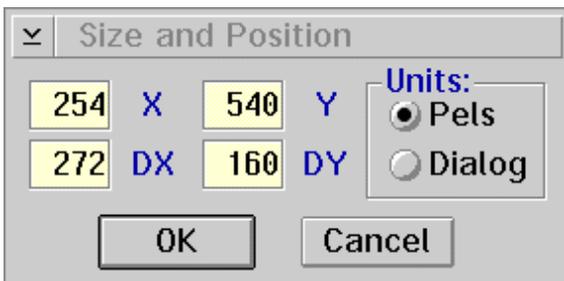
Delete controls



Clicking the **delete** button deletes the currently selected controls from the edit dialog. If no controls are currently selected, no action is taken. If the dialog itself is selected, you will be prompted whether you wish to delete the entire dialog or not.

Pressing the **Del** key on the keyboard while the current edit dialog has the focus also has the same effect as pressing the **delete** button.

Size window



Size window

The **size** window displays the location and size of the currently active control. The location displayed is of the lower-left corner of the control.

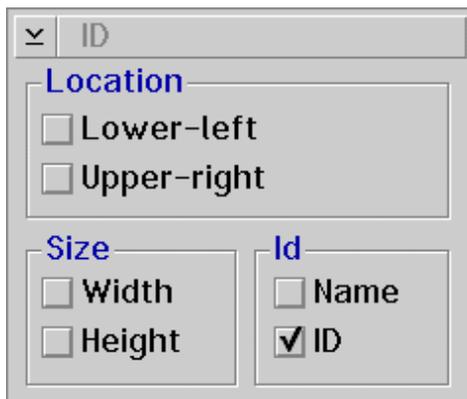
You can specify the units as either **pels** or **dialog units** by checking the appropriate radio button.

You can also change the location and/or size of the control by editing the contents of the appropriate fields and either pressing **Enter** or clicking on the **OK** button.

Clicking the **Cancel** button will copy the active control's current location and size information back into the entry fields, discarding any changes you may have made.

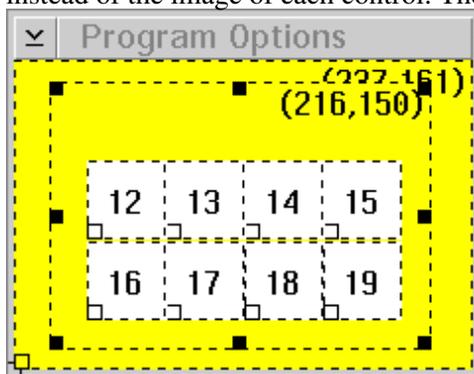
If no control is active, the size window will be disabled.

ID window



ID window

Selecting the **ID** tool modifies the display of all dialogs being edited to show information about each control instead of the image of each control. The resulting display will look something like the following:



Container controls are shown in yellow, and all other controls are shown in white.

All normal editing actions (e.g. moving, sizing, changing view mode) are still available while the ID tool is active. All edit dialogs will return to their normal display mode when the ID tool window is closed.

The check boxes in the ID tool window specify what information is to be displayed within each control. The choices are:

- Lower-left Display the coordinate of the lower-left corner of the control in the lower-left corner of the control.
- Upper-right Display the coordinate of the upper-right corner of the control in the upper-right corner of the control.
- Width Display the width of the control in square brackets in the center of the control.
- Height Display the height of the control in square brackets in the center of the control.
- Name Display the name of the control in the center of the control.
- ID Display the ID of the control in the center of the control.

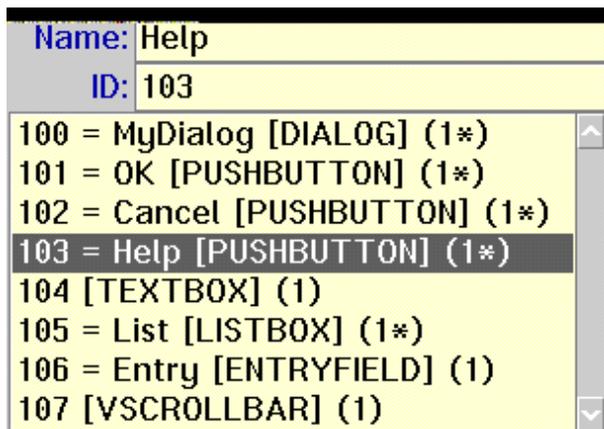
If both **width** and **height** are checked, they are displayed in the form: [**width,height**].

If both **name** and **ID** are checked, they are shown in the form: **name=ID** .

If there is not enough room in a particular control to display all of the requested information, only the information that will fit is displayed. The information is prioritized so that lower priority information is removed first. From high to low, the priority order is as follows:

- oID
- oName
- oWidth/Height
- oLower-left
- oUpper-right

Name window



Name window

The **name** window lists the ID and optional name of each control in the edit dialog, including the dialog itself. The name window is actually the first page in the [Events](#) section of the DrRexx notebook. The list box displays each control in the form: **ID = name [type] (count*)**, where **name** is optional, **type** is the type of control (e.g. **PUSHBUTTON**), and **count** is the number of controls in the resource file with the same ID. If a '*' is present

after *count*, it indicates that one or more control specific REXX handlers have been written for the control.

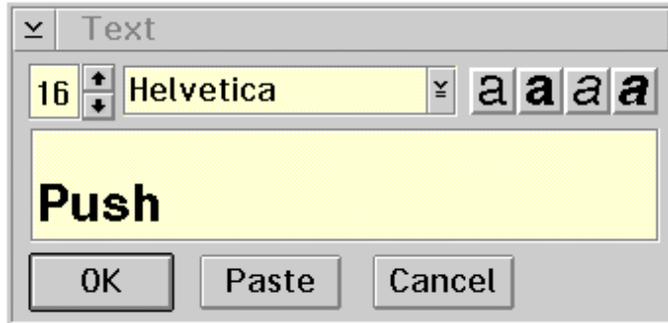
The currently active control (if any) is always selected in the list box. Selecting a new entry in the list box will also select a new active control in the edit dialog. The name and ID of the active control can also be changed by editing the values in the two entry fields at the top of the name window and then pressing **Enter** to make the change.

Note: ID's must be unique within a dialog. Names must be unique within a resource file. If you enter a non-unique name, the editor will automatically append a numerical suffix to make it unique. If you enter a non-unique ID, the editor will automatically replace it by the next available unique ID.

If you enter an ID used in a different dialog, and the name field is blank, the control will be assigned whatever name is currently attached to the ID in the other dialog. If the name field is not blank, all controls with the same ID will be assigned the new name.

The name and ID of a control can also be set using the **Name...** and **ID...** options of the pop-up menu.

Text window



Text window

The **text** window allows you to specify the window text and font for the currently active control. When a control is made active, its current window text and font information is copied to the edit fields of the text window.

To change the control's window text, edit the entry field and press **Enter** or click on the **OK** button.

To change the control's font:

- oSelect the font family from the drop-down list in the middle
- oSelect the point size using the spin button on the left
- oSelect the style (i.e. *normal*, *bold*, *italic* or *bold italic*) using the style buttons on the right
- oClick the **OK** button or press **Enter** to make the font change.

Clicking the **Cancel** button will copy the active control's current window text and font information back into the text tool's edit fields, discarding any changes you may have made.

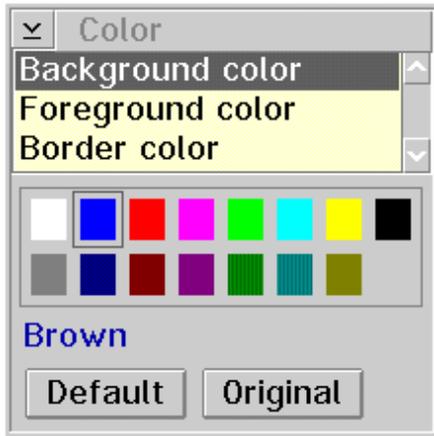
Clicking the **Paste** button will copy the current contents of the clipboard into the entry field and also make it the control's new window text. If no text is in the clipboard, the editor will beep.

If no control is active, or the active control does not have a text field, the text window will be disabled.

If the active control has text, but no font information, the font selection controls will be disabled.

The text and font of a control can also be set using the **Text...** and **Font...** options of the pop-up menu.

Color window



Color window

The **color** window allows you to specify the colors to use for the current active control. When a control is made active, its list of color attributes and color information is copied to the edit fields of the color window.

To change the active control's colors:

- oSelect an attribute from the list of color attributes defined for the active control (e.g. **Background color**). The color palette will be updated to show the color currently being used for that attribute (if it is known).
- oSelect a new color from the palette of available colors

To restore the active control's colors back to their default values, click the **Default** button. To restore the active control's colors back to the values they had when the control was first selected, click the **Original** button.

Note: Not all control types have the same color attributes. Some control types do not have any defined color attributes (e.g. ICONBUTTON). If a control with no color attributes is made active, the color tool is disabled .

Normally the color tool displays a color palette consisting of 16 default colors. However, if the color tool is sized large enough vertically, an additional 40 colors are added to the palette (for a total of 56 colors). The new colors are the colors used by PM to draw specific display items (e.g. **Window static text**), and correspond to the colors that can be specified using the system **scheme editor**. If you select colors from this extended portion of the color palette, your controls will automatically use whatever color is in effect for that particular **logical** color. That is, if another user's color scheme is different than yours, your controls will automatically use the color scheme in effect for that user when your dialog is displayed on that user's system).

The text control located below the color palette displays the **name** of the color currently being pointed at, and can be used to locate a particular logical color in the palette.

When used in conjunction with the [same style](#) tool, the color tool can be used to set the colors for an entire group of controls quickly and easily:

- oFirst, select the group of controls whose colors are to be set.
- oSecond, use the color tool to set the colors for the active control in the group.

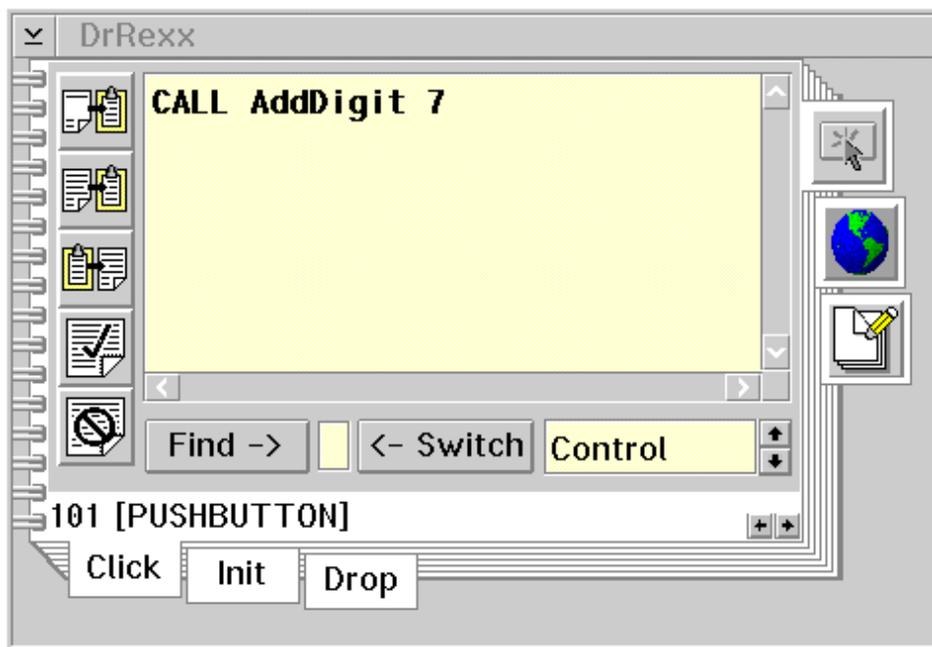


- oFinally, click the button to copy the color information from the active control to all other selected controls in the group.

Note: The size of the color tool, and hence the number of colors displayed in the palette, is a user preference item and is automatically saved across DrDialog sessions.

The foreground and background colors of a control can also be set using the **Colors...** option of the [pop-up](#) menu.

DrRexx window



DrRexx window

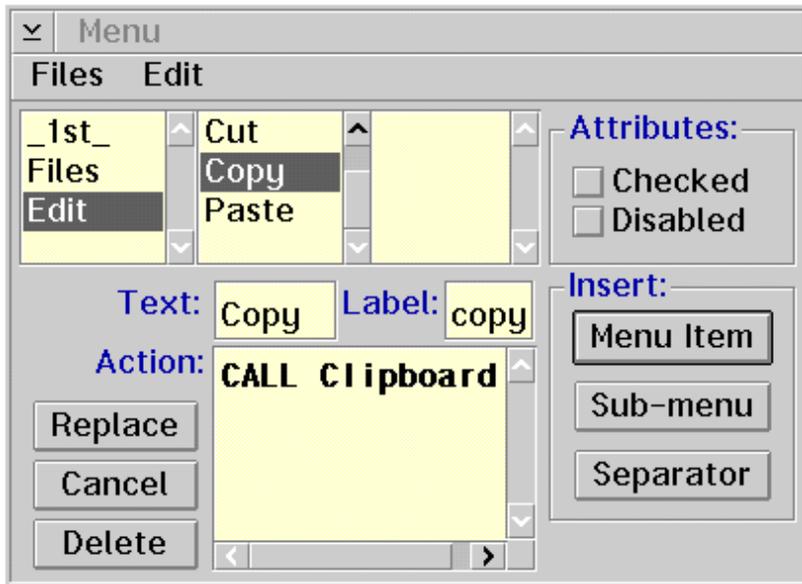
The DrRexx window is actually a notebook with several major sections:

- oREXX event handlers for each defined control
- oREXX procedures defined globally for the current resource file
- oA notepad of REXX code fragments or other useful pieces of information

For more detailed information on the use of the DrRexx notebook, refer to the [DrRexx](#) section.

For more information on the events defined for each type of control, refer to the [controls](#) section.

Drop-down menu window



Drop-down menu window

The drop-down menu editor window allows you to define and edit the contents and actions for the drop-down menu, if any, associated with the current dialog.

The drop-down menu editor uses a series of listboxes to display the structure of the drop-down menu being edited. It also displays a version of the actual menu at the top of the editor window so you can verify that the menu has the desired structure and appearance.

The left-most menu editor listbox displays the items at the top-most level of the drop-down menu structure (i.e. the items on the *menu bar*). Only submenus can be added to the menu bar.

The middle listbox displays the menu items for the drop-down menu currently selected in the left-most listbox, while the right-most listbox displays the menu items for the submenu currently selected in the middle listbox. If the left-most or middle listboxes do not have a submenu selected, then no items are listed in the middle or right-most listbox respectively.

The menu item currently being edited is always the right-most selected entry within the three listboxes.

Note: The topmost entry in each listbox is always **_1st_**. This entry is not an actual menu item, and so cannot be modified. Its purpose is to provide a way of inserting a menu item as the first entry in a menu bar or submenu.

There are three types of entries that can be inserted into a drop-down menu structure:

- oMenu item
- oSubmenu
- oSeparator

A **menu item** can be either selectable or non-selectable (i.e. static) . A menu item is selectable if it has some

REXX code associated with it. The REXX code associated with a menu item is entered into the **Action** multi-line edit control. If a menu item has no REXX code associated with it, then it is static and cannot be selected.

A **submenu** defines the entry point to a submenu. It has no REXX code associated with it, but is selectable (selecting it causes its submenu to be displayed) . A submenu can also be used as a pop-up menu when invoked using the MenuPopUp menu function.

A **separator** is simply a horizontal bar that separates other menu items. It has no REXX code associated with it and it cannot be selected.

Initially, a dialog has no drop-down menu. All of the listboxes are empty except for the left-most, which has a **_1st_** entry. The only type of menu entry that can be added to the top-level of a drop-down menu is a submenu.

To create a new submenu:

- oSelect the entry in the listbox you want the submenu inserted *after* (initially this can only be the **_1st_** entry).
- oType the text describing the submenu into the **Text** field.

Note: A '~' character preceding a character in the text defines that character as a keyboard accelerator for that menu entry.

- oEnter an optional label for the submenu entry into the **Label** field. Labels are used to identify menu entries when using the DrRexx menu functions.

Note: Menu labels need not be unique within a drop-down menu. A menu function applied to a particular label operates on all menu items with that label.

- oSet the **Check** and **Disabled** check boxes to indicate the desired initial state of the submenu entry (i.e. checked or not checked, disabled or not disabled) .

- oClick the **submenu** button to insert the submenu entry into the drop-down menu after the currently selected menu entry.

The process for inserting menu items or separators is very similar.

To insert a new menu item:

- oSelect the entry in the listbox you want the menu item inserted *after*. Menu items can only be inserted after menu entries in the middle or right-most listbox.

- oType the text describing the menu item into the **Text** field.

Note: A '~' character preceding a character in the text defines that character as a keyboard accelerator for that menu entry.

- oEnter an optional label for the menu item into the **Label** field. See the previous description of inserting submenus for an explanation of the use of labels.

- oSet the **Check** and **Disabled** check boxes to indicate the desired initial state of the menu item (i.e. checked or not checked, disabled or not disabled).

- oEnter the REXX code to be executed when the menu item is selected into the **Action** multi-line edit control. If no REXX code is entered, the menu item will be static (i.e. non-selectable).

- oClick the **Menu Item** button to insert the menu item into the drop-down menu after the currently selected menu entry.

To insert a new separator:

- oSelect the entry in the listbox you want the separator inserted *after*. Separators can only be inserted after menu

entries in the middle or right-most listbox .

oClick the **Separator** button to insert the separator into the drop-down menu after the currently selected menu entry.

Once a menu entry has been added to the drop-down menu, it can modified or deleted.

To modify an existing menu entry:

oSelect the entry to be modified in the appropriate listbox.

Note: In the case of a second or third level menu entry, this may require selecting its parent submenus first. You can also *back up* to an already selected parent submenu by double-clicking its listbox entry.

The **Text**, **Label**, **Action**, **Checked** and **Disabled** fields will be updated to reflect the current values for the selected menu entry.

Alternatively, if the desired entry is a menu item, it can be selected directly from the actual version of the menu at the top of the menu editor window.

oChange the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields to reflect their new values.

oClick the **Replace** button to make the changes to the menu entry. Alternatively, you can simply select another menu entry; the previously selected menu entry will automatically be updated with the values of the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields prior to displaying the new menu entry's values.

To delete an existing menu entry:

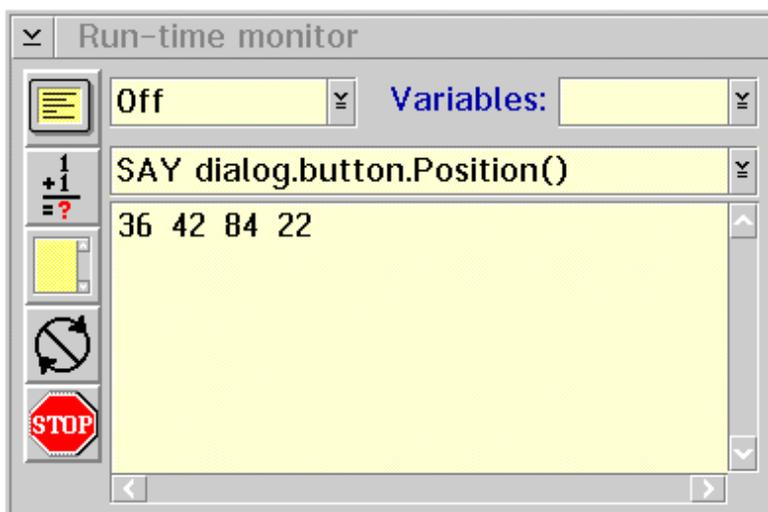
oSelect the entry to be deleted (see the previous description of modifying menu entries for more information on how to do this).

oClick the **Delete** button to delete the menu entry.

To correct an editing error:

oClick the **Cancel** button and the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields will be restored to their last saved values.

Run-time window



Run-time window

The run-time window allows you to run the DrRexx application currently being edited under control of the editor.

DrDialog operates in one of two modes: *edit* mode or *run* mode. In edit mode you can freely make changes to any of the dialogs contained in the current resource file. This is the mode in which most of the DrDialog tools operate.

Using the run-time window, you can tell the editor to enter run mode. In run mode, all of the editing functions of the editor are disabled so that you can test the actual behavior of the your application.



Run mode is entered by clicking the  button in the run-time window. When you do this several things happen:

- oAll editing tool windows are hidden, including the current dialog being edited.
- oAll of the controls in the run-time window are enabled.
- oYour DrRexx application is started.



You can stop execution of your DrRexx application at any time by clicking on the  button (which replaces the  button when you enter run mode.

While you are in run mode, your application behaves exactly the way it would when running outside of the DrDialog environment (i.e. when running under control of the stand-alone DrRexx environment). In addition, you can use the various tools of the run-time window to help you debug your application. For more information on the function of the various run-time window tools, double-click on the appropriate areas of the figure to the left.

If your application requires command line arguments, you can specify them by entering them into the entry field of the run-time window prior to starting execution of your application. Alternatively, if you have previously entered the arguments, you can also select them into the entry field using the drop-down list. Whatever text is in the entry field when the application begins execution is passed to it as its command line arguments.

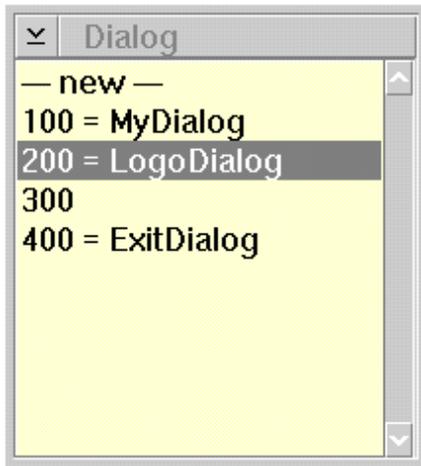
Note: If you wish, you can have DrDialog automatically save the current set of dialogs you are editing to a file



each time you click the  button. This can prevent you from losing editing changes if your application causes DrDialog to crash (e.g. your application calls an external function with invalid arguments).

To enable this feature, first select **Options...** from the **Options** submenu of the DrDialog menu bar. Then check the **Save before running program** check box in the **Auto save** section of the dialog that appears. You may also specify the name of the file to save the current set of dialogs in before running your program. If you do not specify a file name, the dialogs will be saved in the same file they would have been saved in if you had selected the **Save** option from the **File** submenu of the DrDialog menu bar.

Dialog select window



Dialog select window

The **dialog select** window lists the ID and optional name of each dialog in the current resource file being edited. The list box displays each dialog in the form : **ID = name**, where *name* is optional.

Double clicking an entry in the list box selects the corresponding dialog for editing. Double clicking the **--- new** entry at the top of the list box creates a new, empty dialog.

Note: A dialog can be given a symbolic name by selecting the dialog for editing and either changing its entry in the name window or by using the **Name..** option of the pop-up menu.

Background window

The **background** window is a backdrop for all other DrDialog windows. It can be made full screen to separate DrDialog's windows from all other application windows; or it can be made small to allow quick access to other applications on the desktop.

The background window also allows DrDialog to be minimized if desired.

At the top of the background window is the DrDialog menu bar. The menu bar provides access to the following editor functions:

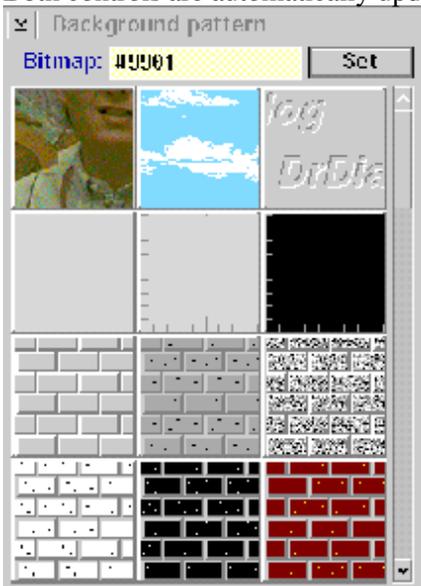
File This submenu provides options for loading and saving new or existing resource files

Tools This submenu provides access to the same set of editor tools available through the Tools window and the **T**ools submenu of the pop-up menu.

Controls This submenu allows any DrDialog control type to be added to the current edit dialog. It provides the same palette of controls available through the Controls window and **C**ontrols submenu of the pop-up menu.

Group This submenu provides access to the same set of group manipulation tools available through the Group window and the **G**roup submenu of the pop-up menu .

At the bottom of the background window are two controls that describe the DrDialog window and control currently pointed at. The leftmost control gives a brief description of the window the pointer is currently in, while the rightmost control describes the function of the control the pointer is currently over. If the pointer is over an edit dialog, the rightmost control displays the ID number, name, type and hint text for the control pointed at in the form: **ID = name [type] ' hint'**, while the leftmost control identifies the edit dialog as being active or inactive. Both controls are automatically updated as you move the pointer around the screen.



The design displayed in the background window is user selectable. To change it, click button 2 while the pointer is anywhere over the current background window pattern. A pop-up window will appear displaying a list of available background patterns. Scroll through the list until an interesting design appears, then click on it with button 1 to make it the new background. If you are not satisfied with the result, you may scroll through the list

and select another design.

If the background pattern you would like to use is in a separate **.DLL** or **.BMP** file, you may also type its name into the entry field at the top of the dialog, then click the **Set** button to make it the new background pattern. For a bitmap in a **.DLL**, the name should be of the form: **dllName:#resourceId** (e.g. **MYBMPS:#4**). For a bitmap in a **.BMP** file, simply type in the complete file name (e.g. **C:\MYBMPS\MYCAT.BMP**). Note that the file extension *must* be **.BMP**.

Once you are satisfied with the background pattern, you may close the pattern pop-up window using its system menu.

The size, position and pattern for the background window is saved as part of the user preferences for DrDialog.

Managing your DrDialog workspace

Because of the many tools provided by DrDialog, your desktop (workshop?) can easily become cluttered with windows. To help control this clutter, DrDialog provides several ways for you to organize and manage your tool windows:

- o Use the **pop-up** menu or DrDialog menu bar whenever possible. The pop-up menu especially provides quick access to most editor functions, and any dialogs it displays are popped up near the pointer and are removed automatically as soon as you are done using them.

- o DrDialog automatically remembers the position and size of each tool window . Each time a tool is invoked, it will appear at the same location and with the same size it had the last time you used it.

- o Tool windows can be opened automatically at the beginning of each editor session if desired. The system menu of each tool window contains an **Auto open** option. If this option is checked, the associated tool window will automatically be opened at the start of each edit session if the **Auto hide** option for the window is **not** checked. If the option is not checked, or the **Auto hide** option is checked, the window will not be opened until requested. To change the current setting of the **Auto open** option, simply select it from the tool's system menu. The setting is automatically remembered across editing sessions and so only needs to be set once.

- o Tools can automatically be hidden when they are not being used. The system menu of each tool window contains an **Auto hide** option. If this option is checked, the associated window is hidden whenever another tool window or the background window is selected, or the pop-up menu is displayed. If this option is not checked , the tool window is left visible until explicitly closed. To change the current setting of the **Auto hide** option, simply select it from the tool's system menu. The setting is automatically remembered across editing sessions and so only needs to be set once.

Note: Selecting an edit dialog will not cause a tool window to be hidden, even if **Auto hide** is active for the tool. The tool window will only be hidden if another tool window or the background window is selected, or the pop-up menu is displayed.

A tool window that has been automatically hidden can be redisplayed simply by clicking its corresponding button either in the tools window or the **Tools** submenu of the DrDialog menu bar or pop-up menu.

- o Tools can either **float** above the dialog being edited or drift to the bottom of the window stack. If the **Float** option of a tool's system menu is checked, the tool window will always **float** above the dialog currently being edited, even if the edit dialog is given the focus. If the **Float** option is not checked, the tool window will drop behind the current edit dialog whenever the edit dialog receives the input focus. To change the current setting of the **Float** option, simply select it from the tool's system menu. The setting is automatically remembered across editing sessions and so only needs to be set once.

Invoking DrDialog

The syntax for invoking **DrDialog** is:

```
DrDialog [ -p[iniPath] ] [ resourceFile ]
```

where:

iniPath Optional path name used to locate the **DRDIALOG.INI** user preference profile

resourceFile Optional name of a .RES resource file to be loaded into the editor at startup. If the file name contains special characters (e.g. blanks) it should be enclosed in double quotes (e.g. 'My Application Dialogs').

If the **-p** option is not specified, the editor will check to see if the system profile contains the path name to use for locating your user preference profile. If it does, the specified path name is used. If it does not, you will be prompted to enter the path name (i.e. directory) where **DrDialog** should keep its user preference profile (**DRDIALOG.INI**). Once you enter the path name, it will be stored in the system profile so that future invocations of **DrDialog** will not prompt you for the path name.

Note: If **-p** is specified with no path following it, the current path stored in the system profile will be deleted. You will then be prompted to enter the new path name to store in the system profile. This case is useful if you re-install or move the **DrDialog** files to a different drive or directory and want to change the default location of the **DrDialog** user preference profile.

If **resourceFile** is specified, the editor will attempt to load the specified dialog at startup. If **resourceFile** is not specified, the editor will initially display an empty dialog with the same size and position as the last dialog edited.

The first time **DrDialog** is run, the **DrDialog** logo screen will also be displayed. From this screen you can either continue into the editor or display the on-line help. The logo will only be displayed automatically the first time **DrDialog** is run. It can be also be displayed any time the editor is running by clicking on the about button either in the tools window. or the **Tools** submenu of the DrDialog menu bar or pop-up menu.

DrDialog and the Workplace Shell

DrDialog supports the OS/2 Workplace Shell as follows:

oResource files that have been edited with DrDialog but which have no DrRexx code associated with them will

appear in a folder with the DrDialog data file icon :



Double clicking the icon will invoke DrDialog on the specified resource file . Dragging and dropping the icon on the DrDialog icon will have same effect.

oResource files that have been edited with DrDialog and which have DrRexx code associated with them will

appear in a folder with the DrRexx data file icon :



Double clicking the icon will invoke DrRexx on the specified resource file (i .e. will **run** the application). Dragging and dropping the icon on the DrRexx icon will have same effect. Dragging and dropping the icon on the DrDialog icon will invoke the editor on the specified resource file.

oNew, empty DrDialog or DrRexx resource files may be created by dragging and dropping the DrDialog or DrRexx template icon from the **DrDialog** folder into any other folder.

DrDialog also includes two simple DrRexx applications: [REStoPgm](#) and [REStoEXE](#) which can help turn your finished DrRexx application into a drag-and-drop Workplace Shell application or a standard OS/2 executable (i.e. .EXE) file .

REStoPgm

The DrDialog package includes a simple DrRexx application called **REStoPgm** that turns your finished DrRexx applications into drag-and-drop Workplace Shell applications.

To use **REStoPgm** just drag a DrRexx **.RES** file that you want to convert and drop it on the **REStoPgm** icon in the **DrDialog** folder. **REStoPgm** will create a new program object in your desktop folder with the same name as your original **.RES** file minus the **.RES** extension. Now, drag any file or other object that your DrRexx application can process and drop it on the new program object. Your DrRexx application will be invoked with the file or other object passed as the command line argument (i .e. your program can use **PARSE ARG** to retrieve it).

Of course, once the program object has been created, you are free to move it to a new location or rename it.

To demonstrate **REStoPgm**, try dropping the **BmpList.RES** icon in the DrDialog **SAMPLE** folder onto the **REStoPgm** icon. A program object called **BmpList** should appear on your desktop. Now try dropping a folder containing some **.BMP** files onto the new **BmpList** icon. The **BmpList** program should appear with all the **.BMP** files in the folder you dropped on it listed.

REStoEXE

The DrDialog package includes a DrRexx application called **REStoEXE** that turns your finished DrRexx applications into standard OS/2 executable (i.e. **.EXE**) files.

To use **REStoEXE**, just drag a DrRexx **.RES** file that you want to convert and drop it on the **REStoEXE** icon in the **DrDialog** folder. **REStoEXE** will create an executable version of your DrRexx application in the same directory as your **.RES** file and with the same name except for a **.EXE** file extension. **REStoEXE** will also create a new program object in your desktop folder with the same name as your **.RES** file minus the **.RES** extension.

If an icon file exists with the same name as your DrRexx **.RES** file, but with a **.ICO** extension, it will be used as the icon for both the **.EXE** and program objects created by **REStoEXE**. If no icon file is found, the standard DrRexx icon will be used instead.

Once **REStoEXE** has created the executable version of your program, you can invoke it simply by double-clicking its program icon. If your application accepts a file as its first command line argument, you can also drag any file or other object that your DrRexx application can process and drop it on the new program object. Your DrRexx application will be invoked with the file or other object passed as the command line argument (i.e. your program can use **PARSE ARG** to retrieve it).

Of course, once the program object has been created, you are free to move it to a new location or rename it.

To demonstrate **REStoEXE**, try dropping the **BmpList.RES** icon in the DrDialog **SAMPLE** folder onto the **REStoEXE** icon. A program object called **BmpList** should appear on your desktop. Now try dropping a folder containing some **.BMP** files onto the new **BmpList** icon. The **BmpList** program should appear with all the **.BMP** files in the folder you dropped on it listed.

Note: The **.EXE** file created by **REStoEXE** is completely self-contained and does not require the **DrRexx.EXE** program in order to run.

Note: **REStoEXE** requires that **RC** (i.e. Resource Compiler) be in your **PATH**.

DrRexx

DrRexx is a powerful feature of DrDialog which allows REXX code to be attached to a resource file created by DrDialog. The resulting extended resource file is referred to as a DrRexx application.

The REXX code attached to a resource file defines actions associated with events that occur while the DrRexx application is running (e.g. a pushbutton being clicked, or a drop-down menu item being selected). A DrRexx application can either be run stand-alone, using the DrRexx run-time environment (DRREXX.EXE), or under control of the DrDialog editor. The latter is an especially powerful tool since it allows the DrRexx application developer to rapidly and iteratively develop an application by switching back and forth between *edit* and *run* mode.

The DrRexx notebook

All DrRexx editing functions are accessed using the DrRexx tool window. The DrRexx window is displayed by

clicking on the  button in the tools window. The DrRexx window is organized as a notebook with three major sections:

 Name

The events section allows editing the REXX event handling code for the currently active control.

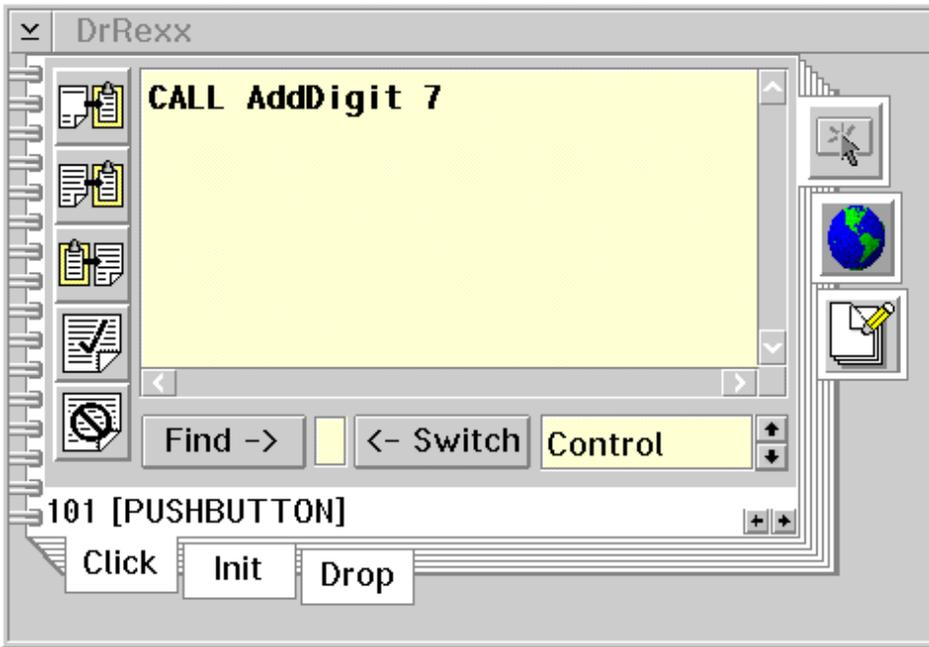


The global procedures section allows editing REXX procedures which are global to the entire application (i.e. resource file).



The notepad section allows editing common REXX code fragments or other useful pieces of information

Events section



Events section

The events section of the DrRexx notebook allows you to define REXX routines to handle the various events associated with each control in your application.

The *cover page* of the events section is also called the name tool, and displays a list of all currently defined controls for the dialog being edited. Double-clicking any entry in the list will *turn* to the subsection of the notebook defining the event handlers for the selected control. The name and numeric ID of the currently selected control can also be changed by editing the corresponding entry fields at the top of the page and pressing **Enter**.

The control events subsection of the DrRexx notebook has a page for each event defined for the currently active control. Selecting the tab for an event will display the REXX code associated with the event in the edit control and allow it to be edited.

Note: From the edit dialog you can also get to a specified page quickly by selecting the matching event name from the **Events** submenu of the pop-up menu for the control.

There are two types of handlers that can be defined for each event: control specific event handlers or class event handlers.

A control specific event handler is a handler for an event that is specific to a particular control (e.g. do this action whenever *this specific* pushbutton is clicked). This is the normal type of event handler.

A class event handler is a handler for an event that is generic to the class of the control (e.g. do this action whenever *any* pushbutton in this dialog is clicked).

When an event occurs for a control, the DrRexx run-time environment does the following:

- 1.If there is a control specific event handler for the event, it invokes that handler and goes to step 4.
- 2.If there is a class handler for that event for that kind of control, it invokes that handler and goes to step 4.

3.If there is a handler for the **Any** event for the dialog containing the control, it invokes that handler and goes to step 4.

4.It waits for the next event.

The type of handler currently being edited is determined by the state of the spin button located in the lower right hand corner of the event page. If the spin button has **Control** selected, the control specific handler is being edited. If the spin button has **Class** selected, the class handler is being edited.

Note: The spin button also has two additional states (**Events** and **Functions**) which display different types of help information about the currently selected control. When either of these two items is selected, the information in the edit control is read-only.

When **Events** is selected, the edit control contains a list of the events that are defined for the currently selected control with a brief description of each .

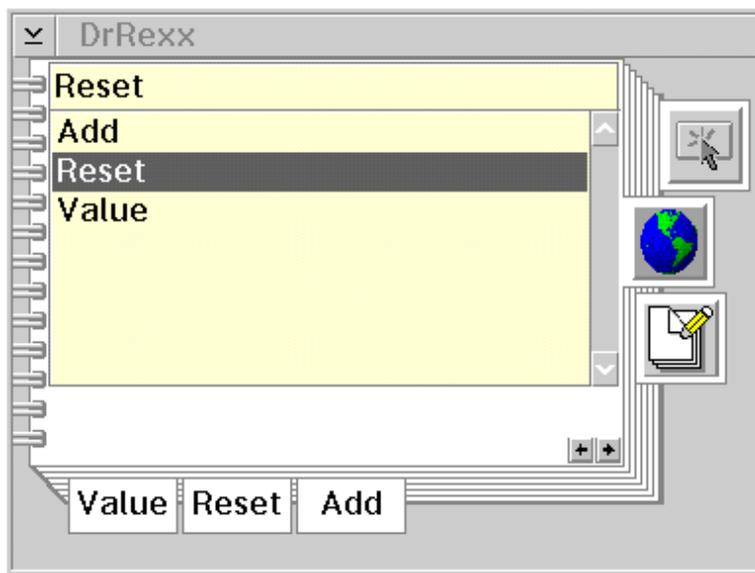
When **Functions** is selected, the edit control contains a list of the window functions that are defined for the currently selected control.

The editor indicates which type of handler will handle an event for the currently active control by the trailing character in the page tab for the event. If the trailing character is a '*', then the event will be handled by a control specific event handler. If the trailing character is a '-', then the event will be handled by a class event handler. If neither '*' nor '-' appears as the trailing character in the page tab, neither a control specific nor a class event handler is defined for the event (although the event may still be handled by the **Any** event handler for the dialog).

Note: There is only one class handler per event for all controls of the same type within a dialog. Editing the class handler for an event in one control changes the class handler for that event in all controls of the same type for that dialog.

There is an exception to the above rule in the case of the **Init** event, which has *no* class handler. All **Init** event handlers *must* be control specific. Any class handler defined for an **Init** event will be ignored.

Global procedures section



Global procedures section

The global procedures section of the DrRexx notebook allows you to define REXX procedures which can be called from other parts of your DrRexx application. The procedures are global because they can be called from any dialog within the current resource file, not just the one currently being edited.

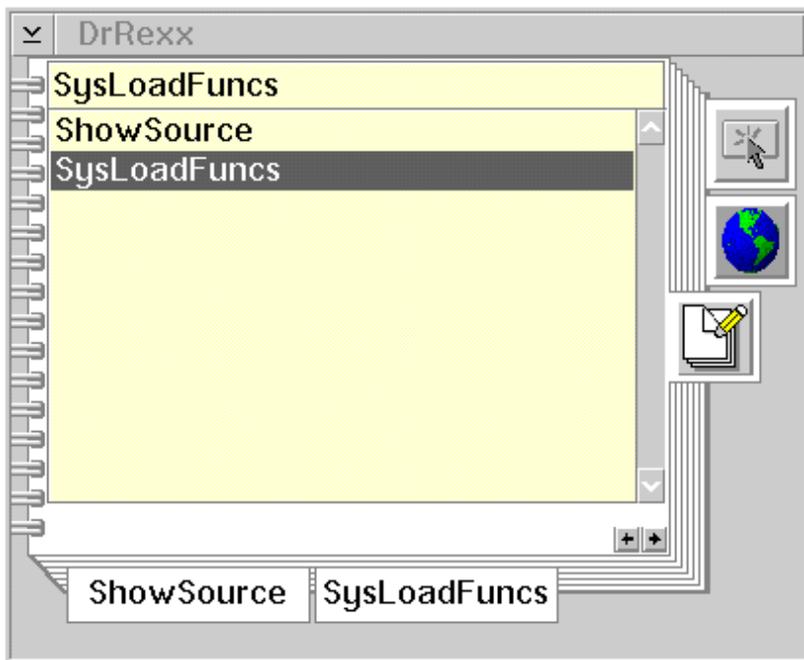
The *cover page* of the global procedures section displays a list of all currently defined procedures. Double-clicking any entry in the list will *turn* to the notebook page defining the procedure and allow it to be edited.

A new procedure can be added by typing its name into the entry field at the top of the page and pressing **Enter**. A new, empty page with the name just entered will be added to the section and automatically selected so that you can enter the code for the new procedure.

The global procedures section has a page for each global procedure currently defined. Selecting the tab for a procedure will display the REXX code associated with the procedure in the edit control and allow it to be edited.

A procedure can be deleted by deleting all its code. The page defining the procedure will automatically be deleted from the notebook when any other page is selected.

Notepad section



Notepad section

The notepad section of the DrRexx notebook allows you to keep track of handy fragments of REXX code or other useful information. For example, many REXX programs make use of the standard **RexxUtil** functions, and so must include the following lines of REXX code somewhere in the program:

```
CALL RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'  
CALL SysLoadFuncs
```

Using the DrRexx notepad, this fragment of code can be assigned a name and always be available for quick inclusion in any DrRexx application.

Note: Information in the DrRexx notepad section is associated with the DrDialog editor, and not with the current resource file being edited. As such, its contents are available in every editing session. As a corollary, nothing in the notepad section is stored as part of a DrRexx application resource file. To include something from the notepad in the application, it must first be copied from the notepad into one of the event or global procedure pages of the DrRexx notebook.

The **cover page** of the notepad section displays a list of the names of all current notes. Double-clicking any entry in the list will **turn** to the notebook page defining the note and allow it to be edited.

A new note can be added by typing its name into the entry field at the top of the page and pressing **Enter**. A new, empty page with the name just entered will be added to the section and automatically selected so that you can enter the text of the note (i.e. REXX code or other information you wish to keep track of).

The notepad section has a page for each note currently defined. Selecting the tab for a note will display the text associated with the note in the edit control and allow it to be edited.

A note can be deleted by deleting all its text. The page defining the note will automatically be deleted from the notebook when any other page is selected .

Using the DrRexx editor

All DrRexx notebook pages that allow text to be edited make use of the same set of editing controls.

The text to be edited appears in a standard Presentation Manager multi-line edit control. You may use a Workplace Shell **Font Palette** object to drag and drop the font to use for editing onto the multi-line edit control. You can do the same thing with a Workplace Shell **Color Palette** object to set the color of the multi-line edit control. Whatever font and color you select will be saved as a user preference item for use in future editing sessions. The default font is the same font currently selected for use by the **System Editor**.

Beneath the edit control is an entry field with a button labeled **Find ->**. You can enter a string in the entry field and click the find button to search for the next occurrence of the specified string in the text being edited. Searching always starts from the current cursor location.

The button labeled **<- Switch** on the other side of the entry field allows you to switch to any application currently listed on the OS/2 Presentation Manager Window List. Enter the first few characters of the application name as it

appears on the Window List into the entry field and click the <- **Switch** button. If the application is running, it will be brought to the foreground automatically. If the application cannot be found, the system will beep to indicate that it could not switch to the requested application. The text in the entry field at the time the <- **Switch** button is pressed is saved as a user preference item when DrDialog exits. Its previous value can be retrieved by clicking the <- **Switch** button with an empty entry field.

To the left of the edit control are a column of buttons with the following functions:



Delete the currently selected text and copy it to the clipboard.



Copy the currently selected text to the clipboard. If no text is selected, the *entire* contents of the edit control are selected and then copied to the clipboard.



Replace the currently selected text by the contents of the clipboard.



Save all of the text being edited.



Replace all of the text by the last saved copy of the text (or the original contents if the save button has not been pressed).

Using your own editor

Not everyone is comfortable using a new or different text editor. Recognizing this, DrDialog allows you to edit the REXX code for your application using your own editor and have the resulting REXX code automatically imported into your DrRexx application.

In order to use this feature of DrDialog, you must adhere to the following naming convention:

If the name of your DrRexx application is **drive:\path\filename .RES**, then the name of the file containing the associated REXX code must be **drive :\path\filename.REX**.

Each time DrDialog *saves* or *runs* your **.RES** file, it checks for the existence of a corresponding **.REX** file. If the file exists, its contents are automatically included in your **.RES** file as a special resource type. When your application runs, this special resource is added to the end of the REXX program that DrRexx creates for you.

Note: When you distribute your DrRexx application, you need not include your **.REX** file, since its contents have already been included in your **.RES** (or **.EXE**) file.

While the bulk of your DrRexx application can be stored in a separate **.REX** file, you must still use the DrRexx editor to provide pointers to the various event handlers in your **.REX** code. DrDialog assists you with this *linking* operation as follows:



If the multi-line edit control in the events section of the DrRexx notebook is empty, and you click the button, DrDialog will insert text of the form: **CALL dialog.control.event** into the multi-line edit control. In addition, it will copy text of the form: **dialog.control.event:** into the system clipboard.

The **CALL** statement inserted into the edit control provides the necessary *link* to your **.REX** code. You can then switch to your editor (e.g. using the Switch button) and paste the label created by DrDialog into your **.REX** file. Finish the process by adding the REXX code needed to handle the event and you're done!

Note: You do not need to use this feature of DrDialog in order to call your **.REX** code. You are free to choose whatever names for your routines that you like. If you do, you are responsible for ensuring that there are no duplicate or invalid labels in your application.

The following summarizes the important points to remember when storing your REXX code in an external file:

- oThe REXX code must be stored in a file having the same name as your **.RES** file, but with a **.REX** extension.

- oThe **.REX** code is automatically included whenever DrDialog *saves* or *runs* your **.RES** file. It only includes the version stored on disk, so make sure you save the **.REX** file in your editor before saving or running the **.RES** file in DrDialog.

- oTerminate your **.REX** file event handler routines with a **RETURN** or **SIGNAL RETURN** statement, depending on whether you **CALL**ed or **SIGNAL**ed the event handler from your DrRexx code.

- oYour **.REX** code becomes part of the **.RES** file when you save the **.RES** file in DrDialog, so you need not include it when you distribute your application to other users.

Writing REXX code for DrRexx

For the most part, writing REXX code for use with DrRexx is no different than writing any other kind of REXX code. The full power of the REXX interpreter and language are available for your use, including the use of external function libraries such as **RexxUtil**.

However, there are a number of details about the DrRexx environment that should be kept in mind when writing an application:

- oEvent handlers are dispatched using the REXX **SIGNAL** statement. You do not need to specify a label in your code though. DrRexx will generate the appropriate label for you. In addition, DrRexx will automatically generate a **SIGNAL** statement at the end of your event handler to return control to the DrRexx event dispatcher. If you want to exit a signal handler other than by falling out the bottom of your code, you can do so by issuing a **SIGNAL RETURN**. **RETURN** is the label of the DrRexx statement that waits for and dispatches the next event.

- oGlobal procedures that you write can either be **CALL**ed or **SIGNAL**ed. DrRexx automatically generates the name you supplied as the label for the global procedure, and also generates a **RETURN** statement after the last statement of the procedure.

- oAll DrRexx applications are by their nature OS/2 Presentation Manager programs. Only operations that make sense within the context of a PM program should be performed in a DrRexx application. For example, you should not use the **RexxUtil** function **SysTextScreenRead**, since this function is defined to work only in a windowed or full-screen application.

- oAll SAY or trace output is directed to the DrRexx run-time control window (unless redirected to a different window using the IsDefault window function).

This window is always available when running a DrRexx application under control of DrDialog. When running under the stand-alone DrRexx environment, SAY or trace output causes the DrRexx run-time control window to appear automatically.

Each line of SAY or trace output appears as a separate line inserted at the bottom of a scrolling list in the run-time control window. Only the last 100 lines of output are kept.

o**PARSE SOURCE** returns **OS/2 SUBROUTINE resFile**, where **resFile** is the name of the **.RES** file being run. Note that when running the application under **DrDialog**, **resFile** is the name of the file most recently saved or loaded, even if it was from a previous session.

o**PULL** and other related REXX instructions that read from the data queue will cause a pop-up entry field window to appear. Data entered into this window is returned as the value **PULL'ed**.

The DrRexx execution model

In writing a DrRexx application, the following concept should always be kept in mind: all REXX code is executed in response to some event (of course, there is one exception to this rule, but we'll get to that later)

If you are already familiar with Presentation Manager programming, the preceding statement comes as no surprise. But if you are mostly familiar with writing batch mode REXX programs, this notion might take some getting used to.

Basically, a DrRexx application works as follows:

- 1.The DrRexx run-time monitor waits for an event to occur.
- 2.Once an event occurs, it determines which piece of REXX code should be executed to handle it (see the [Events](#) section for more information on how this is done) .
- 3.If no handler is found, the event is ignored, and control returns to step 1.
- 4.If a handler is found, it is executed, and then control returns to step 1 .

It really is that simple. Of course, there are a few other questions that need to be asked (and answered):

oQ. What happens before the first event occurs (i.e. how does a DrRexx application get started)?

oA. After the DrRexx run-time environment finishes loading your resource file, it starts the ball rolling by automatically invoking the REXX procedure called **Init**. If you have written such a procedure in the [global procedures](#) section of the DrRexx notebook, it will be called at this time. If you have not written such a procedure, the run-time environment will automatically supply an empty procedure for you. Note that the **Init** procedure is the one exception to the rule for executing REXX code that we mentioned above (i.e. it is not executed in response to any event).

After running the **Init** procedure, the run-time environment checks to see if any dialogs have been created yet (you'll see how to do this in the section on DrRexx [window functions](#)). If one or more dialogs have already been created, the run-time environment enters the loop we described previously.

If no dialogs are created by the **Init** procedure, the run-time environment searches through all the dialogs in the resource file for the one with the smallest ID number and automatically creates it for you. It then enters the loop described above to wait for the first event.

If your resource file does not contain any dialogs at all, the run-time environment pops up the run-time control



window and displays the error message: 'No dialog to load'. You can then click on the  button to terminate the DrRexx run-time environment (this situation might happen if you were to drag and drop the DrRexx template icon to create a new DrRexx application, and then double-clicked the new icon thinking you would invoke the DrDialog editor, but instead end up invoking the DrRexx run-time environment).

oQ. How do I terminate a DrRexx application?

oA. You can terminate a DrRexx application in one of two ways:

-Execute the REXX EXIT statement.

-Close all of your dialogs (refer to the DrRexx [window functions](#) section for more information on how to do this).

DrRexx subcommand environments

In addition to the standard REXX **CMD** subcommand environment (which is the initial environment for all DrRexx applications), DrRexx also defines two additional subcommand environments:

SAY Treats each subcommand string as if a REXX SAY statement had been performed on it.

NULL Discards each subcommand string without taking any action.

For example, if **ADDRESS 'NULL'** appears in the DrRexx **Init** global procedure, all subsequent subcommands will be discarded for the duration of the DrRexx application (or until another **ADDRESS** statement is executed).

Using the **NULL** subcommand environment allows DrRexx code to be written in a somewhat simpler style, as the following example illustrates:

```
/* Load a listbox with data: */
list.Delete()
DO i = 1 TO file.0
list.Add( file.i )
END
```

If **NULL** were not the current subcommand environment, the preceding example might have been written as:

```
/* Load a listbox with data: */
CALL list.Delete
DO i = 1 TO file.0
CALL list.Add file.i
END
```

Note: It is a common error to use the coding style of the first example even though **NULL** is not the current subcommand environment. This can slow your application down significantly since the subcommand strings are normally passed to the **CMD** subcommand environment, which tries to interpret them as OS/2 commands.

Error handling in DrRexx

If a REXX error occurs in the process of executing a DrRexx application, the DrRexx run-time environment handles the error by:

- oAborting the currently executing REXX routine
- oDisplaying an error message indicating the cause and source of the error in the DrRexx run-time control window
- oReturning to the event loop to wait for the next event

A REXX error is thus local to the event handler it occurs in and does not abort execution of the entire DrRexx application.

Note: If the error occurs while running under the stand-alone DrRexx run-time environment, and the run-time control window is not visible at the time of the error, the window will be made visible before displaying the error message.

Note: In order for the DrRexx error handling code just described to work correctly, you should not handle REXX **syntax** or **halt** errors directly in your program.

Invoking DrRexx

The syntax for invoking **DrRexx** is as follows:

```
DrRexx resourceFile[.RES] [arguments]
```

where:

resourceFile Name of the resource file containing the DrRexx application to be run. The file extension must be **.RES** and may be omitted. If the file name contains special characters (e.g. blanks), it should be enclosed in double quotes (e.g. 'My DrRexx Application').

arguments Optional command line arguments to be passed to the DrRexx application.

Getting started: Your first DrRexx application

To help overcome any confusion you might have about how to construct a DrRexx application, we'll walk you through the steps for creating a very simple DrRexx application.

First, the application...

Allow the user to type in and execute OS/2 commands and view their output in an editable window.

Now, the DrRexx solution...

1. Open the **DrDialog** folder and drag the **DrRexx.RES** icon into whatever folder you want to store the DrRexx application. If you like, rename the resulting DrRexx file (e.g. **CmdEdit.RES**).
2. Drag and drop the new DrRexx file you just created onto the DrDialog icon in the **DrDialog** folder. This will invoke the DrDialog editor and bring up an empty dialog for editing.
3. For this application we will use:

- o A single-line edit control for entering commands
- o A pushbutton to execute the current command
- o A multi-line edit control to hold the output generated by a command.

Click button 2 to display the pop-up menu and then select from the **Controls** submenu each of the controls we

need (i.e.  ,  and ). You will have to display the pop-up menu three times to do this, once for each control. Then arrange and size the controls in whatever way you like using button 2 of the mouse.

4. Next, use the **Text** option of the pop-up menu to change the text of the controls to match our intended application (e.g. position the pointer over the pushbutton, click button 2 to display the pop-up menu, then select the **Text** option. When the pop-up dialog appears, type **Execute** and press **Enter**).
5. In a similar manner, use the **Name** option of the pop-up menu to give the single-line entry field the name **command**, and the multi-line edit control the name **output**.
6. Next, position the pointer over the pushbutton, display the pop-up menu using button 2, and select the **Click** option from the **Events** submenu.
7. The DrRexx notebook should appear with the **Click** event page of the events section already selected. Type the following code into the multi-line edit control on the notebook page:

```
command.Text() '| RXQUEUE'  
file = ''  
DO i = 1 TO queued()  
  PULL line  
  file = file || line || '0D0A'  
END  
CALL output.Text file
```

This is the REXX code we want to execute whenever we click on the pushbutton . It passes the current contents of the single-line entry field **command** to the **CMD** subcommand environment and queues its output using **RXQUEUE**. It then concatenates all lines queued by the command into a single variable, **file**,

and makes that the new content of the multi-line edit control: **output**.

8. Now display the run-time window (e.g. by selecting the  icon in the **Tools** submenu of the button 2 pop-up menu). When the window appears, click

on the  button to run the DrRexx application.

9. All of the DrDialog tool windows (except for the DrRexx notebook) should disappear and be replaced by your application dialog. Type an OS/2 command (e.g. **DIR**) into the single-line edit control, then click on the pushbutton to execute the command. After a (hopefully brief) delay, the command output should appear in the multi-line edit control. Voila!

10. Once you have satisfied yourself that the application does indeed work,

either close the dialog window or click on the  button in the run-time window to stop the application. The application dialog should disappear, and the edit dialog should reappear.

11. Now save the application by selecting the **Save** option from the **File** submenu of the DrDialog menu bar.

12. Exit DrDialog by double-clicking the system menu icon in the top-left hand corner of the DrDialog background window.

13. If you want, you can verify that you now have a working DrRexx application by double-clicking on the icon for the DrRexx file in the folder you originally placed it in. The application dialog should appear and behave exactly as it did when you ran it under control of the DrDialog editor.

That's it... you've created your first DrRexx application!

There's plenty of room left for improvement in our little example, so please feel free to go back, experiment, and try out new ideas. The whole point of DrRexx is to make the process of developing PM applications faster, simpler, and maybe even a little bit fun.

DrRexx programming techniques

This section presents some answers to commonly asked questions about programming using DrRexx. The particular questions answered are:

oHow can I create a modal dialog?

oHow can I associate data with a particular control or dialog?

oHow do I adjust the size and position of controls when a user changes the size of a dialog?

oHow do I create and display a pop-up menu?

oHow can I get the Enter key to signal data entry is complete for a dialog?

oWhat's the easiest way to work with controls whose identity is not known until run-time?

oCan my DrRexx application display the same kind of user hints that the DrDialog editor displays at the bottom of the background window?

oHow do I stop my dialogs from flashing when they are first displayed?

Creating a modal dialog

A modal dialog is a dialog which the user must finish interacting with before proceeding with other parts of an application. While the modal dialog is active, no other dialog or window in the application can be used (i.e. they are disabled).

A common example of a modal dialog is a password dialog. Before letting a user proceed with some application function, a valid password must be entered. Until the password has been entered and validated, no other application function is to be allowed (even though the dialog windows may be visible).

Ideally, creating a modal dialog should be as simple as possible. For example, in our password example, we would like to be able to write something like the following:

```
IF PasswordDialog() = 'TOPSECRET' THEN
/* proceed with password protected function */
ELSE /* indicate user has not entered a valid password */
```

Normally, because DrRexx applications are event-driven, creating a modal dialog would be difficult, because we need to suspend one event handler until some future event handler (e.g. the password dialog's 'OK' button **Click** event handler) has executed.

However, there is a DrRexx function that allows us to easily create modal dialogs. This function, **ModalFor**, is similar to the **Open** (or **OpenFor**) function, but differs in that control does not return until the user has finished interacting with the dialog being displayed.

After **ModalFor** displays a specified dialog, it does not return to the caller until some future event handler executes a REXX **RETURN** instruction. If you recall , event handlers normally end with a **SIGNAL RETURN** statement. However, for modal dialogs, event handlers which terminate the modal dialog need to exit using a **RETURN** statement. Assuming the caller has properly invoked the modal dialog using the **ModalFor** function, this accomplishes two things:

- 1.It allows the caller of the modal dialog to resume execution.
- 2.It allows the modal dialog to return a result to the caller (i.e. the expression specified on the **RETURN** statement).

Note: It is important to remember when writing the event handlers for a modal dialog that **every** event handler that terminates the dialog **must** exit with a **RETURN** statement. Equally important, no other event handlers for the modal dialog can exit using a **RETURN** statement.

To finish illustrating this technique using our password example, study the following code fragments:

```
PasswordDialog: RETURN ModalFor( 'passwordDlg' )

/* Event handler for 'passwordDlg' OK button: */
password = entryField.Text()
CALL passwordDlg.Close
RETURN password
```

Associating data with dialogs and controls

Quite often it is useful to be able to associate one or more pieces of data with particular dialogs or controls. For example, imagine you are writing a telephone dialing application with 10 *speed dialing* buttons. When the user clicks on a speed dialing button, you want to dial the number associated with that button. Because you probably don't know the numbers when you implement the program, you can't explicitly write them in the event handler for each button. Instead you would like to be able to save the names and numbers externally (for example, in a .INI file), and then attach the information dynamically to the buttons each time your program is run.

While DrRexx provides no function to perform this type of association of data with controls, the REXX language already has built into it a facility for doing just this sort of thing. The REXX language feature referred to is a *compound symbol*.

A **compound symbol** is simply a REXX variable whose name is divided into several segments, each separated by a period (e.g. **number.dialer.C101**). When the REXX interpreter evaluates a compound symbol, each section of the name, except the first, is evaluated as a variable and the result is substituted for the original in determining the final variable name. To illustrate how this language feature can be used to implement our speed dialing example, consider the following function:

```
/* Usage is: oldValue = NumberFor( dialog, control [, newValue] ) */
NumberFor: PROCEDURE EXPOSE number.
PARSE ARG dialog, control, newValue
oldValue = number.dialog.control
IF arg(3,'E') THEN
number.dialog.control = newValue
RETURN oldValue
```

Assuming the dialing pushbuttons are numbered **101** through **110**, this function can be used to implement the speed dialing example as follows:

```
/* In the dialog 'Init' event handler: */
/* Assume 'data' contains the list of phone numbers */
DO i = 1 TO 10
CALL NumberFor Dialog(), 'C'|(100+i), word( data, i )
END

/* The 'Click' CLASS event handler for each pushbutton: */
```

```
/* Assume 'PhoneDial' actually dials the telephone */  
CALL PhoneDial NumberFor( Dialog(), Control() )
```

Note that a **class** event handler is used in this example since it avoids having to write a unique event handler for each pushbutton.

Adjusting controls when a dialog is resized

Many Presentation Manager **windows** have **size** borders that allow the user to change the shape of the window. After the size border is dragged into the desired shape, the content of the window automatically adapts itself to the new dimensions. Since a **window** normally only contains a few controls (e.g. a client window, horizontal and/or vertical scroll bar, and a drop-down menu), re-arranging the contents is a fairly straightforward matter, and is in fact performed automatically by the window's frame control.

The situation for a **dialog** is somewhat different however. Dialogs can contain many different types of controls, arranged in a variety of configurations. Rearranging the size and position of each control after the user changes the shape of the dialog can be a much more challenging task. For that reason, many Presentation Manager dialogs do not have sizing borders. This eliminates the application developer's problem, but often comes at the expense of the application user's convenience. A good example of this is the standard file dialog prompt. How many times have you wanted to make the file dialog taller so that you could see more file names in the selection list box?

DrRexx addresses this common problem by **automatically** rearranging the size and position of each control contained within a dialog after the user changes its size. This rearrangement occurs without the need for you to write any code whatsoever. The technique used by DrRexx in performing the rearrangement employs a **heuristic** based on common dialog layout styles and patterns. Because it is a heuristic, it may or may not rearrange a particular dialog in an **optimal** manner. In a case where the algorithm does not perform satisfactorily, you have one of two choices:

- oRearrange the layout of the controls within the dialog until the automatic resizing algorithm performs in a desirable manner.

- oWrite a **Size** event handler for the dialog. The existence of a **Size** event handler automatically disables the automatic resizing logic and allows your event handler to perform whatever rearrangement you wish.

Some general characteristics of the automatic resizing algorithm that can be used in designing a dialog are as follows:

- oThe horizontal and vertical dimensions of the dialog are handled independently. This should be kept in mind when applying the other guidelines.

- oThe largest controls are used to determine how to position and/or size the smaller controls.

- oThe alignment of the smaller controls relative to the larger controls is important. Smaller controls positioned between or **spanning** the boundaries of the largest controls are handled differently than controls completely contained between the edges of one of the largest controls.

- oThe horizontal and vertical dimensions are handled somewhat differently. Smaller controls tend to grow or shrink horizontally, while smaller controls tend to be displaced vertically.

Note: The automatic resizing algorithm is invoked any time the user changes a dialog's size using the size border and no **Size** event handler is defined for the dialog. It will also rearrange the contents of a dialog when the dialog's size is changed under program control as long as the dialog has a size border and no **Size** event handler.

Creating and displaying pop-up menus

Pop-up menus have many applications, especially in the area of context-sensitive action selection. DrDialog allows you to easily create and display them using the drop-down menu window and the MenuPopUp menu function.

To define a pop-up menu, simply follow these steps:

1. Create a new, empty dialog by double-clicking on the **New** list box entry in the dialog select tool.
2. Invoke the drop-down menu window for the newly created dialog.
3. Create a new submenu entry in the menu window. The submenu will become the pop-up menu when invoked using the **MenuPopUp** function. Be sure to give the submenu entry a label (the label will be used to identify the submenu to the **MenuPopUp** function).
4. Add the pop-up menu items to the submenu as you would for a drop-down menu .

That's basically all there is to defining a pop-up menu. To display the menu at run-time, apply the **MenuPopUp** function to the dialog and submenu created in steps 1 and 3. The submenu will be displayed as a pop-up menu at the current pointer location.

If your application has several pop-up menus, you can add each additional pop-up menu as a new submenu by repeating steps 3 and 4 above for each new pop-up . **Enter** key. Many application writers are therefore surprised when they search the list of edit control events and cannot find an event to signal that data entry is complete.

In fact, there are several ways to detect that the user has completed data entry into one or more single line edit controls:

- o Define a pushbutton (possibly hidden) with the **Default** style. When the user presses **Enter** it will generate a **Click** event for the pushbutton.
- o Define a **Key** event handler for the dialog, and check the information returned by the EventData function to see if the key pressed was the **Enter** key.
- o Define a **LoseFocus** event handler for each single line edit control. This handler will be invoked any time the edit control loses focus (e.g. when the user presses the **Tab** key or clicks the mouse pointer on another edit control).

Signaling that data entry is complete

A single line edit control does not process the keyboard **Enter** key. Many application writers are therefore surprised when they search the list of edit control events and cannot find an event to signal that data entry is complete.

In fact, there are several ways to detect that the user has completed data entry into one or more single line edit controls:

- o Define a pushbutton (possibly hidden) with the **Default** style. When the user presses **Enter** it will generate a **Click** event for the pushbutton.
- o Define a **Key** event handler for the dialog, and check the information returned by the EventData function to see if the key pressed was the **Enter** key.

oDefine a **LoseFocus** event handler for each single line edit control. This handler will be invoked any time the edit control loses focus (e.g. when the user presses the **Tab** key or clicks the mouse pointer on another edit control).

Working with dynamic controls

By *dynamic* control we mean a control whose identity is not known until run-time. DrRexx provides a means of dealing with such controls through use of the **...For** functions. That is, every DrRexx window function can be written in one of two ways:

```
result = dialog.control.function( ... )
or
result = functionFOR( 'dialog', 'control', ... )
```

The latter is especially well suited for the case where the identity of the control to operate on is not known until run-time. For example, the following routine is a modified version of the **Drop** class handler taken from the **Softball.RES** sample program:

```
CALL EventData
dlg = 'Softball'
src = 'C' || EventData.6
dst = 'C' || EventData.7
player = TextFor( dlg, src )
playerHint = HintFor( dlg, src )
position = TextFor( dlg, dst )
positionHint = HintFor( dlg, dst )
IF position = '' THEN DO
CALL TextFor dlg, src, ''
CALL DragFor dlg, src, ''
CALL HintFor dlg, src, ''
END
ELSE DO
CALL TextFor dlg, src, position
CALL HintFor dlg, src, positionHint
END
CALL TextFor dlg, dst, player
CALL DragFor dlg, dst, 'Player'
CALL HintFor dlg, dst, playerHint
```

In this case, the controls to operate on are determined from the **EventData** associated with the **Drop** event, and so are not known until run-time. Notice the heavy use of **...For** functions to operate on the affected controls.

A variation on this technique which oftens leads to simpler, more readable, code is based on the Isdefault function. **IsDefault** can be used to establish a new default context for DrRexx window function references. Study the

following version of the previous routine written using **IsDefault**:

```
CALL EventData
dlg = 'Softball'
src = 'C' ||EventData.6
dst = 'C' ||EventData.7
position = TextFor( dlg, dst )
positionHint = HintFor( dlg, dst )
CALL IsDefaultFor dlg, src
player = Text()
playerHint = Hint()
IF position = '' THEN DO
CALL Text ''
CALL Drag ''
CALL Hint ''
END
ELSE DO
CALL Text position
CALL Hint positionHint
END
CALL IsdefaultFor dlg, dst
CALL Text player
CALL Drag 'Player'
CALL Hint playerHint
```

In this case, notice how the use of **IsDefault** simplified the resulting code . Also, given the way DrRexx resolves window function references, using the default context for window functions wherever possible may actually improve performance .

Putting user hints into your DrRexx application

Many DrDialog users like the helpful *hints* that appear at the bottom of the DrDialog background window describing the name or function of the control being pointed at. And they wonder if it is possible to incorporate such hints into their own DrRexx applications. The answer is yes...simply follow these steps :

oIn the editor, point at a control to be given a hint. Then either press the '?' key or select the **Hint...** option from the pop-up menu . Type the hint text into the prompt dialog that appears and press **Enter**. The hint text associated with the control will appear in quotes in the hint control at the bottom of the DrDialog background window.

Repeat this step for each control which is to have a hint, including the dialog if desired.

oIn the REXX code for your application, use the IsDefault function to specify which control to display hint text in. If desired, two controls can be used : one for control hints and the other for dialog hints (much like the DrDialog hint controls). Any control that accepts text can be use as a hint control.

oIf desired, the hint for a control can be changed dynamically at run-time using the Hint function.

Preventing dialogs from initially flashing

If your dialog has fields that need to be initialized, or the entire dialog needs to be repositioned after it is opened, you may notice an annoying *flash* effect as the fields are initialized or the dialog is moved. Happily, there is a simple way to prevent this from happening:

1. In the editor, turn off the **Visible** attribute in the dialog's pop-up style dialog. This will prevent the dialog from automatically displaying when it is opened.
2. At the end of the dialog's **Init** event handler, add the following statement:

```
CALL Show
```

This will display the dialog after all initialization has occurred, thus eliminating the *flash* effect.

Step 1 also applies to dialogs used as notebook pages and will prevent the dialog from displaying before it is added to the notebook. Step 2 is not necessary because the notebook will automatically make the dialog visible at the appropriate time once it has been added as a notebook page.

DrRexx sample programs

The DrDialog package includes a number of sample DrRexx applications to illustrate programming using DrRexx:

File Description

Events.RES A program that demonstrates the various events that can be processed by a DrRexx application

Calc.RES A simple four function calculator

Clock.RES A simple clock

Clock2.RES A somewhat fancier clock

Swapper.RES Displays the size of SWAPPER.DAT

BMPList.RES Displays .BMP and .GIF files

Drives.RES Displays space utilization for a selected drive

Fac.RES A simple multi-threaded program to calculate factorials. One thread computes factorials while the other thread displays the results.

TestKey.RES Displays the information obtained from the EventData function for a dialog 'Key' event (useful for finding the names of particular keys when writing a keyboard driven application)

TestDrop.RES Displays the information obtained from a 'Drop' event (useful for debugging a drag and drop application)

DragDrop.RES Illustrates how drag and drop support works with various controls

Softball.RES Allows a softball team captain to assign players to positions . Illustrates drag and drop, using metafiles with **turtle** controls, using the system clipboard, and static and dynamic control hint text. Also illustrates working with controls whose identity is not known until run-time.

You might wish to try running and editing a few of these applications to get a feel for what DrRexx programming is like.

If you have opened the DrDialog subdirectory folder, you can run any of the sample DRREXX applications by double-clicking on its icon. You can also edit any of the sample applications by dragging and dropping its icon on the DrDialog icon .

DrRexx example programs

In addition to the sample programs provided with DrDialog, there are also a number of example programs contributed by users of DrDialog. The examples are provided as part of the DrDialog package in the file: **DREXAM.RAM**. Each example typically illustrates one or more techniques for using DrRexx and DrDialog that may be helpful when trying to become productive with DrDialog as quickly as possible.

To install the examples, simply double-click on the **DrExam** icon in the **DrDialog** folder and fill in the appropriate information in the installation dialog that appears. A new **Example** folder containing the examples will be added to your **DrDialog** folder when installation is complete.

Note: The **DrExam** icon will only be present in your **DrDialog** folder if the **DREXAM.RAM** file was in your **DrDialog** directory at the time you installed DrDialog. If no **DrExam** icon is present, you should copy **DREXAM.RAM** into the **DrDialog** directory and re-run the DrDialog **INSTALL** program.

Once you have successfully installed the examples, a good place to start is to double-click on the **housecal.res** icon in the **Examples** folder. This program lets you:

- oRead a brief description of each example
- oRun each example to see what it does
- oEdit each example to see what makes it tick

The author would like to thank Bill Gillquist, who has coordinated and collected each of the examples found in **DREXAM.RAM**, as well as the following people who have contributed one or more examples to the collection:

- oRalph Baeumer
- oBuzzy Brown
- oGuy De Ceulaer
- oMark Fiechtner
- oBill Gillquist
- oJacques Gourdon
- oThomas Lindqvist
- oPierre Lotigie-Laurent
- oGregor Neaga
- oMark Radford
- oHelmut Saueregger

DrRexx window functions

In addition to the normal suite of REXX built-in functions, the DrRexx run-time environment also defines an extensive set of functions for interacting with Presentation Manager dialogs and controls (i.e. windows). Each of these window functions is invoked using the following *object oriented* syntax style:

```
[dialog.][control.]function ( arguments )
```

where **dialog** refers to the name assigned to a dialog using the DrDialog Name window, and **control** refers to the name assigned to a control using the same tool. For example, the statement:

```
value = myDialog.entry.Text()
```

might be used to retrieve the contents of the **entry** edit field of the **myDialog** dialog and assign it to the variable **value**.

Note that the use of dialog and control names are optional. If either or both are omitted, the following rules apply:

- oIf only **dialog** is omitted, the function applies to the named control within the dialog in which the event occurred.

- oIf both **dialog** and **control** are omitted, the function applies to the control generating the event.

- oIf only **control** is omitted, the function applies to the frame of the named dialog.

Note: In the case of the **Init** procedure used to start a DrRexx application, there is no event. Therefore, any window function calls it contains must use a fully qualified name. This is the only exception to the above set of rules.

If no name was assigned to a dialog using the DrDialog Name window, **Dnnn**, where **nnn** is the ID number of the dialog, may be used in place of the dialog name (e.g. **D100.entry.Text()**).

Cnnn, where **nnn** is the ID number of a control, may also be used in place of the name of a control, whether the control has a name or not (e.g. **D100.C101.Text()**).

In addition, DrRexx also supports the following variation for invoking window functions:

```
functionFOR( dialog [, control [, arguments]] )
```

where **dialog** and **control** are as described above. For example:

```
value = TextFor( 'myDialog', 'entry' )
```

would be the equivalent way of writing the previous example using the alternate syntax style.

The main use of this alternate style is in cases where the dialog and control name are determined dynamically at run-time, rather than statically at edit time .

The defined window functions are as follows:

Name	Description
<u>Open</u>	Open (i.e. create) a dialog window
<u>Close</u>	Close (i.e. destroy) a dialog window
<u>Owner</u>	Get/Set the owner of a dialog window
<u>Frame</u>	Get the size of a dialog window's frame
<u>Hide</u>	Hide a window
<u>Show</u>	Show a window
<u>Visible</u>	Get/Set the visibility state of a window
<u>Top</u>	Make a window topmost
<u>Bottom</u>	Make a window bottommost
<u>Enable</u>	Enable a window
<u>Disable</u>	Disable a window
<u>Enabled</u>	Get/Set the enable state of a window
<u>Focus</u>	Give the focus to a window
<u>Position</u>	Get/Set a window's position and/or size
<u>Text</u>	Get/Set a window's text
<u>Hint</u>	Get/Set a window's hint text
<u>H</u>	Add an item to a window
<u>Delete</u>	Delete an item from a window
<u>Item</u>	Retrieve an item from a window
<u>Select</u>	Query/Select an item in a window
<u>Range</u>	Set a window's range
<u>Style</u>	Get/Set a window's style bits
<u>Font</u>	Get/Set a window's font
<u>Color</u>	Get/Set a window's color
<u>ID</u>	Get a window's ID number
<u>Drag</u>	Enable/Disable dragging a control
<u>Drop</u>	Enable/Disable a control as a drop target
<u>IsDefault</u>	Make a window the current default window
<u>Timer</u>	Start/Stop a window timer
<u>View</u>	Get/Set a window's viewing mode
<u>SetStem</u>	Set a list of window values
<u>GetStem</u>	Get a list of window values
<u>Controls</u>	Get the names of all controls in a dialog
<u>Classes</u>	Get the classes of all controls in a dialog

Open

```
rc = [dialog.]Open(  
[alias] [, registeredName]  
[, 'Modal' | 'Nonmodal' ] )
```

Opens (i.e. creates) the dialog for the specified window, which must be a dialog.

If no arguments are specified, it creates the dialog corresponding to **dialog**.

Note: There can be only one dialog with a specified name in existence at a time.

If **alias** is specified, the dialog is created with the name **alias**. Again, **alias** must not be the name of an already existing dialog.

If **registeredName** is specified, the dialog is registered with the Presentation Manager so that it will appear on the Window List with the specified name, and the dialog is owned by the desktop. If **registeredName** is not specified, the dialog is not registered with PM, and in addition the current dialog is made the owner of the newly created dialog. The **current** dialog is the dialog to which the current event belongs.

If the third argument is omitted or **Nonmodal**, then the dialog is displayed normally. If the third argument is **Modal**, then the dialog is displayed modally. A modal dialog disables all other dialogs currently open and prevents the user from interacting with them until the modal dialog is closed. When the modal dialog is closed, all previously disabled dialogs are re-enabled. Note that only the first letter (i.e. **M** or **N**) of the third argument need be specified.

A result of 1 indicates the window was successfully created, and a result of 0 indicates that the window could not be created (e.g. a dialog with the same name already exists).

For example:

```
/* Create two calculator windows */  
/* then hide the second one: */  
CALL calc.Open 'calc1'  
CALL calc.Open 'calc2'  
CALL calc2.Hide
```

Close

```
rc = [dialog.]Close()
```

Closes (i.e. destroys) the specified window, which must be a dialog .

A result of 1 indicates the window was successfully destroyed, and a result of 0 indicates that the window could not be destroyed (e.g. it did not exist to start with).

For example:

```
/* Close a prompt window: */  
CALL prompt.Close
```

Owner

```
oldOwner = [dialog.]Owner( [newOwner] )
```

Gets or sets the owner window for the specified window, which must be a dialog.

Returns the current owner of the specified window. If the desktop is the current owner of the window, the null string is returned.

NewOwner specifies the name of the window which is to be the new owner of the specified window. If **newOwner** is the null string, the desktop window is made the new owner of the window; otherwise **newOwner** must be the name or alias of an existing dialog window.

Making one window the **owner** of another affects the windows in the following manner:

- oIf the **owner** window is moved, the **owned** window moves with it.
- oIf the **owner** window is destroyed, any windows it **owns** are also destroyed.
- oAn **owner** window always appears **behind** all windows it owns.

For example:

```
/* Make the window 'owned' by the desktop: */  
CALL prompt.Owner ''
```

Frame

```
result = [dialog.]Frame()
```

Gets the size of the frame for the specified window, which must be a dialog . The result is a string of the form: **left bottom right top**, where:

```
left Width of left side of dialog frame
bottom Height of bottom side of dialog frame
right Width of right side of dialog frame
top Height of top side of dialog frame
```

For example:

```
/* Make the dialog big enough to hold */
/* 2 40 x 40 icon buttons: */
PARSE VALUE dialog.Position() WITH x y . .
PARSE VALUE dialog.Frame() WITH left bottom right top
CALL button1.Position left, bottom
CALL button2.Position left + 40, bottom
CALL dialog.Position x, y, left + 80 + right,,
bottom + 40 + top
```

Hide

```
[dialog.][control.]Hide(
[ 'Update' | 'Noupdate' ] )
```

Hides the specified window, which can be any dialog or control.

If no argument or **Update** is specified, the window is hidden immediately. If **Noupdate** is specified, the window is not hidden, but all future updates to the window are inhibited until the window is made visible again using the Show function . Only the first letter (i.e. **U** or **N**) need be specified.

For example:

```
/* Hide a list while it is being updated: */
CALL list.Hide 'N'
CALL list.Delete
DO i = 1 TO data.0
CALL list.Add data.i
```

```
END
CALL list.Show
```

Show

```
[dialog.][control.]Show()
```

Shows the specified window, which can be any dialog or control.

This function makes visible a window that was initially hidden or made hidden using the Hide function.

For example:

```
/* Show a pushbutton: */
CALL button.Show
```

Visible

```
oldState = [dialog.][control.]Visible(
[newState] )
```

Gets or sets the visibility state of the specified window, which can be any dialog or control.

Returns **1** if the window is currently visible and **0** if it is hidden.

If **newState** is specified, it makes the specified window visible if **newState** is not **0**, and hides the window if **newState** is **0**.

For example:

```
/* Hide the 'OK' pushbutton if the entry field is hidden: */
IF entry.Visible() = 0 THEN CALL okButton.Visible 0
```

Top

```
[dialog.][control.]Top()
```

Makes the specified window, which can be any dialog or control, the topmost in its group. If the window is a control, the control is moved in front of all other controls in the same dialog. If the window is a dialog, the dialog is moved in front of all other dialogs on the screen.

For example:

```
/* Move the playing card to the top of the deck: */  
CALL TopFor 'Tableau', curCard
```

Bottom

```
[dialog.][control.]Bottom()
```

Makes the specified window, which can be any dialog or control, the bottommost in its group. If the window is a control, the control is moved behind all other controls in the same dialog. If the window is a dialog, the dialog is moved behind all other dialogs on the screen.

For example:

```
/* Move the playing card to the bottom of the deck: */  
CALL BottomFor 'Tableau', curCard
```

Enable

```
[dialog.][control.]Enable()
```

Enables the specified window, which can be any dialog or control.

Enabling a window allows a user to interact with it. This function can be used to re-enable a window that was previously disabled using the Disable function .

For example:

```
/* Enable a pushbutton: */  
CALL button.Enable
```

Disable

```
[dialog.][control.]Disable()
```

Disables the specified window, which can be any dialog or control.

Disabling a window prevents a user from interacting with it. The window can later be re-enabled using the Enable function.

For example:

```
/* Disable a pushbutton: */  
CALL button.Disable
```

Enabled

```
oldState = [dialog.][control.]Enabled(  
[newState] )
```

Gets or sets the enabled state of the specified window, which can be any dialog or control.

Returns **1** if the window is currently enabled, and **0** if it is disabled.

If **newState** is specified, it enables the specified window if **newState** is not **0**, and disables the window if **newState** is **0**.

For example:

```
/* Disable the entry field if the check box is not selected: */  
IF check.Select() = 0 THEN CALL entry.Enabled 0
```

Focus

```
[dialog.][control.]Focus()
```

Gives the input focus to the specified window, which can be any dialog or control.

For example:

```
/* Give the focus to an entry field: */  
CALL entry.Focus  
  
/* Give the focus to a different dialog: */  
CALL promptDialog.Focus
```

Position

```
result = [dialog.][control.]Position(  
[x [y [dx [dy] ] ] ] )  
[x [, y [, dx [, dy] ] ] ] )
```

Gets or sets the position of the specified window, which can be any dialog or control.

If no arguments are specified, it returns the current size and position of the window as a string of the form: **x y dx dy**.

The size and position of the window can be set by specifying one or more arguments as shown above. omitted trailing arguments leave their current value unchanged.

For example:

```
/* Move a pushbutton to a new location: */  
CALL button.Position 20, 30
```

Text

```
oldText = [dialog.][control.]Text(  
[newText] )
```

Returns the current text associated with the specified window, which can be any dialog or control, and optionally sets a new text value.

The text associated with a window varies from window to window. For a pushbutton, it is its label. For a dialog, it is its window bar title. For a multi-line edit control, it is the complete text contained within the control. Other controls may have not any text associated with them (e.g. a **Rectangle** control). In that case, the result returned is the null string, and any value set is ignored.

Note: If the specified window is a dialog, setting a new value for the title bar using **Text** will also set the dialog's **Window List** entry if the dialog was originally registered with Presentation Manager in the Open function used to create the dialog.

For example:

```
/* Clear the contents of an edit field: */  
CALL myDialog.edit.Text ''
```

Hint

```
oldHint = [dialog.][control.]Hint(  
[newHint] )
```

Returns the current hint text associated with the specified window, which can be any dialog or control, and optionally sets a new hint text value.

If **newHint** is specified, it replaces the current hint text associated with the control. The exception occurs when **newHint** is the null string, in which case the hint text will be restored to whatever value was set in the editor using the hint pop-up dialog.

The control in which hint text is displayed can be specified using the IsDefault function. Hint text associated with a dialog can optionally be displayed in a different control than hint text associated with controls contained within the dialog. The hint text associated with a control or dialog is displayed in the current hint control whenever the pointer passes over the control or dialog.

For example:

```
/* Set the label and hint text for a button: */  
CALL myDialog.button.Text '911'  
CALL myDialog.button.Hint 'Dial 911 to report an emergency'
```

Add

Adds an item to the specified window, which must be a list box, combo box, notebook or container.

Add (for a list box or combo box)

```
result = [dialog.][control.].Add(  
item  
[, 'Ascending' | 'Descending' | 'Last' | n]  
[, data] )
```

Adds an item to the specified list box or combo box. The text of the item to be added is specified by **item**. The second argument specifies where the item is to be added into the list:

- oIn **Ascending** order
- oIn **Descending** order
- o**Last**
- oAs item **n**

Only the first letter (i.e. **A**, **D**, or **L**) need be specified. If the second argument is omitted, the item will be added at the end of the list (i.e. last).

The third argument specifies an optional data value to be associated with the item. **Data** will not appear in the list, but can later be retrieved using the Item function.

The index of the inserted item in the list is returned as the result.

For example:

```
/* Put all words in a string into a */  
/* list in alphabetical order: */  
DO i = 1 TO words( string )  
CALL myDialog.list.Add word( string, i ), 'A'  
END
```

Add (for a notebook)

```
result = [dialog.][control.].Add(  
[page]
```

```
[, 'First' | 'Last' | 'Next' | 'Previous']  
[, dialog] [, tab] [, status] [, data] )
```

Inserts a new page into a notebook. **Page** specifies the page to insert relative to, and should either be 0 or a page ID returned by a previous Add call for the same notebook. The second argument specifies where the new page is to be inserted into the notebook:

- oAs the **First** page (**page** may be 0)
- oAs the **Last** page (**page** may be 0)
- oAs the **Next** page after **page**
- oAs the **Previous** page before **page**

Only the first letter (i.e. *F*, *L*, *N* or *P*) need be specified. Note also that the case of the first character of the second argument is important. If upper case, a page with a major tab is inserted. If lower case, a page with a minor tab is inserted. If the second argument is omitted, the new page will be added with a major tab as the last page of the notebook.

Dialog specifies the name of the dialog to be used as the contents of the new page. If omitted, a blank page is inserted. The dialog name may either be the name of a dialog in the resource file or the alias of a dialog created using the Open function. If a dialog resource name is specified, it need not have already been opened.

Note: It is good practice to define all dialogs used as notebook pages with the **Visible** attribute **off**. This prevents the dialog from momentarily appearing outside of the notebook when it is added. The notebook control will automatically make the dialog visible whenever its page is selected. Also note that a single dialog can be used as the contents for more than one page of the same notebook .

Tab, **status**, and **data** specify the initial contents of the new page's tab, status line, and data fields respectively. See the Item function for more information about getting and setting the value of these fields. Note that if **status** is specified and is not empty, the new page will have a status line; otherwise it will not .

The ID of the newly created page is returned as the result. If the result is 0, the page was not created successfully.

For example:

```
/* Insert a major and minor page at the */  
/* beginning of a notebook: */  
page1 = notebook.Add( 0, 'F', 'CoverPage', '=BITMAP:#50' )  
page2 = notebook.Add( page1, 'n', 'DataPage',  
'Data', 'Customer data' )
```

Add (for a container)

```
result = [dialog.][control.].Add(  
label [, bitmap]  
[, 'First' | 'Last' | 'Next'] [, item]  
[, data] )
```

Inserts a new item into a container. **Label** specifies the text label to use for the new item. **Bitmap** specifies the name of the optional bitmap to display with the new item. If specified, it should be a string of the form: **dll:#resource**, where **resource** is the resource number of the bitmap within the DLL specified by **dll** (e.g. **BITMAP:#50**). Alternatively, the form : **filename.BMP** can also be used, where **filename.BMP** is the name of a file containing a bitmap in **.BMP** format. If **bitmap** is not specified, then no bitmap is displayed for the new item.

The third argument specifies where the new item should be inserted in the container:

- oAs the **First** item
- oAs the **Last** item (the default)
- oAs the **Next** item after **item**

If **Next** is specified, the new item is always inserted after the entry specified by **item**. If the second argument is **First** or **Last**, the new item is inserted at the beginning or end of the container, unless **item** is specified, in which case the new item is inserted as the first or last item whose parent is **item**. If specified, **item** must be a value returned by a previous **Add** request for the same container . Specifying **item** allows the items in the container to be organized and displayed hierarchically using the **Hierarchy** or **Outline** argument to the View function.

Only the first letter (i.e. **F**, **L** or **N**) of the third argument need be specified. The case of the first letter also determines the placement of the new item in the container's z-order. An upper case letter (e.g. **L**) will place the item at the top of the z-order (i.e. **on top of** all other container items); while a lower case letter (e.g. **l**) will place the new item at the bottom of the z-order (i.e. **beneath** all other container items).

Data specifies an optional data value to associate with the new item. Its value can later be retrieved or modified using the Item function.

The ID of the newly created item is returned as the result.

For example:

```
/* Insert a folder container two files into */  
/* a container:
```

```
folder = container.Add( 'Sample', 'BITMAP:#68' )
CALL container.Add 'TEST.RES', 'BITMAP:#1',, folder
CALL container.Add 'TEST.REX', 'BITMAP:#1',, folder
```

Delete

Deletes one or all items from the specified window, which must either be a list box, combo box, notebook or a container.

Delete (for a list box or combo box)

```
rc = [dialog.][control.]Delete(
[index] )
```

Deletes one or all items from the specified list box or combo box. If no arguments are specified, all list items are deleted. If **index** is specified, the **index** th item in the list is deleted (the index of the first list item is 1).

The function returns the number of items remaining in the list box or combo box after the delete.

For example:

```
/* Delete all list items: */
CALL myDialog.list.Delete
```

Delete (for a notebook)

```
rc = [dialog.][control.]Delete(
[page] )
```

Deletes one or all pages from the specified notebook. If no arguments are specified, all notebook pages are deleted. If **page** is specified, the corresponding notebook page is deleted. **Page** should be a page ID returned by a previous Add call for the same notebook.

The function returns 1 if the delete was successful and 0 otherwise.

For example:

```
/* Delete a particular notebook page: */  
CALL notebook.Delete page1
```

Delete (for a container)

```
rc = [dialog.][control.].Delete(  
[item] )
```

Deletes one or all items from the specified container. If no arguments are specified, all items are deleted; otherwise only the item specified by **item** is deleted.

The function returns the number of items successfully deleted from the container.

For example:

```
/* Delete all container items: */  
CALL container.Delete
```

Item

Gets or sets an item from the specified window, which must be a list box, combo box, notebook, value set, slider or container.

Item (for a list box or combo box)

```
result = [dialog.][control.].Item(  
[index [, 'Value' | 'Data' ] [, value] ] )
```

Gets or sets an item from the specified list box or combo box.

If no arguments are specified, it returns the number of items currently in the list.

If **index** is specified, it returns information about the **index**th item in the list (the index of the first list item is 1). If the second argument is

Value or omitted, the list item itself is returned. If the second argument is **Data**, the data originally associated with the item using the Add function is returned. Only the first letter of the second argument (i.e. **V** or **D**) need be specified.

If **value** is specified, it replaces the current value (or data) of the **index**th item.

For example:

```
/* Move a list item from one list to another: */  
CALL list2.Add list1.Item( index )  
CALL list1.Delete index
```

Item (for a notebook)

```
result = [dialog.][control.]Item(  
[page [, 'Tab' | 'Status' | 'Data'] [, value] ] )
```

Gets or sets a page item from the specified notebook.

If no arguments are specified, it returns the number of pages in the notebook .

If **page** is specified, it gets or sets information about the specified notebook page, which should be an ID returned by a previous Add call for the same notebook. The second argument specifies what kind of information about the page is to be set or retrieved:

- oThe page **Tab**
- oThe page **Status** line
- oThe page user defined **Data**

Only the first letter (i.e. **T**, **S**, or **D**) need be specified. If the second argument is omitted, it defaults to getting or setting the notebook page tab .

Value specifies the data used to set new page information. If **value** is omitted, the current value is not changed.

The function returns the current value for the specified page prior to setting any new value.

Note: A bitmap can be used as a page tab by specifying **value** as: **=dll:#resource**, where **resource** is the resource number of the bitmap within the DLL specified by **dll** (e.g. **=BITMAP:#50**). Alternatively, the form: **=filename.BMP** can also be used, where **filename.BMP** is the name of a file

containing a bitmap in **.BMP** format.

For example:

```
/* Set a notebook page's tab and status line: */  
CALL notebook.item page1, 'T', '=BITMAP:#50'  
CALL notebook.item page1, 'S', 'Order:' orderNum
```

Item (for a value set)

```
result = [dialog.][control.]Item(  
[row, column [, value] ] )
```

Gets or sets an item from the specified value set.

If no arguments are specified, it returns the number of rows and columns in the value set in the form: **rows columns**.

If **row** and **column** are specified, it returns the current value of the item at the specified row and column.

Value specifies the new value for the item at the specified row and column . If omitted, the current value is not changed. **Value** may be any of the following forms:

[@]text A text label (e.g. 'Next'). Note that the leading '@' character is optional, and can be used to distinguish a label starting with '=' or '#'.

=dll:#resource A bitmap with resource number **resource** in the **.DLL** specified by **dll**.

=filename.BMP A bitmap stored in the **.BMP** file specified by **filename .BMP**. The **.BMP** file extension **must** be specified.

#color A color index (e.g. '#1' equals the color **blue**).

#r g b A triplet of numbers specifying a color as its component **red**, **green** and **blue** values (e.g. '#255 255 0' is a bright yellow).

For example:

```
/* Set up a color select value set table: */  
DO i = 1 TO 16  
CALL valueset.item 1, i, '#'||(i-1)  
END
```

Item (for a slider)

```
result = [dialog.][control.]Item(  
[tick [, 'Value' | 'Size' ] [, value]] )
```

Gets or sets a tick value for the specified slider.

If no arguments are specified, it returns the number of tick marks in scale 1 and scale 2 of the slider in the form: **ticks1 ticks2**.

If **tick** is specified, it returns information about the tick mark specified by **tick**. If **tick** is greater than 0, then scale 1 of the slider is used and scale 1 becomes the current **primary scale** for the slider. If **tick** is less than 0, then scale 2 of the slider is used and scale 2 becomes the current **primary scale** for the slider. In this case, **abs(tick)** is used to determine the number of the tick to get or set information about.

If the second argument is **Value** or omitted, then the text associated with the specified tick mark is set or returned. If the second argument is **Size**, then the width (or height) of the specified tick mark is set or returned. Note that only the first character of the second argument (i.e. **V** or **S**) need be specified .

The third argument, **value**, specifies the new value for the specified tick mark's text or size. If omitted, the current value is not changed.

For example:

```
/* Label a slider with Fahrenheit and Centigrade */  
/* scales with tick marks every two degrees, and */  
/* text labels every 10 degrees: */  
DO i = 32 TO 212 BY 2  
CALL slider.Item i - 31, 'S', 5  
END  
DO i = 40 TO 210 BY 10  
CALL slider.Item i - 31,, i  
END  
DO i = 0 TO 100 BY 2  
CALL slider.Item -(i + 1), 'S', 5  
END  
DO i = 0 TO 100 BY 10  
CALL slider.Item -(i + 1),, i  
END
```

Item (for a container)

```
result = [dialog.][control.]Item(  
[item [, 'Value' | 'Bitmap' | 'Data' | column ]  
[, value] ] )
```

Gets or sets an item value from the specified container.

If no arguments are specified, it returns the number of items currently in the container.

If **item** is specified, it gets or sets information about the specified container item, which can either be 0 or an ID returned by a previous Add call for the same container. An **item** of 0 refers to the container title.

The second argument specifies what kind of information about the container item is to be set or retrieved:

- oThe item **Value** (i.e. label)

- oThe item **Bitmap**

- oThe item **Data**

- oA specified item **column**: 1..n, where **n** is the number of columns of data previously defined for each container item using the SetStem function.

For the first three cases, only the first letter (i.e. **V**, **B**, or **D**) need be specified. If the second argument is omitted, it defaults to getting or setting the item's value (i.e. label).

Value specifies the data used to set new item information. If **value** is omitted, the current value is not changed.

The function returns the current value for the specified item field prior to setting any new value.

For example:

```
/* If the data field for a container item = 'FILE', */  
/* change its icon to be a 'sticky pad': */  
IF container.Item( item, 'D' ) = 'FILE' THEN DO  
CALL container.Item item, 'I', 'BITMAP:#60'  
END
```

Select

Gets or selects an item in the specified window, which must be a list box, combo box, single-line edit control, horizontal or vertical scroll bar, spinbutton, push button, radio button, check box, bagbutton, notebook, value set, slider or container.

Select (for a list box or combo box)

```
result = [dialog.][control.].Select(  
[index [, 'Select' | 'Unselect' | 'Next' | 'Top' ] ] )
```

Gets or selects an item in the specified list box or combo box.

If no arguments are specified, the index of the first selected item in the window is returned. If no item is selected, 0 is returned.

If **index** is specified and the second argument is **Select** or omitted, the **index**th item in the list is selected. The index of the first previously selected item, if any, is returned as the result.

If the second argument is **Unselect**, the **index**th item in the list is unselected. The index of the first previously selected item, if any, is returned as the result.

If the second argument is **Next**, the index of the next selected item in the list following the **index**th item is returned, if any. If there is no selected item following the specified index, 0 is returned.

If the second argument is **Top**, the list is scrolled so that the **index**th item in the list is the topmost. The index of the previous topmost item is returned as the result.

Only the first letter of the second argument (i.e. **S**, **U**, **N**, or **T**) need be specified.

For example:

```
/* Copy the selected item to an edit field: */  
CALL entry.Text list.Item( list.Select() )
```

Select (for a single line edit control)

```
result = [dialog.][control.].Select(  
[left [, length] ] )  
[left length] )
```

Gets or selects the range of selected text for the specified single line edit control.

It returns the current selection range in the form: **start length**, where **start** is the one-based index of the first character selected, and **length** is the number of characters selected (0 indicates no character is selected).

The **Start** argument specifies the first character to be selected. The **length** argument specifies the number of characters to include in the selection. **Length** defaults to 0, which means that no characters are selected (i.e. the cursor is simply moved to the location specified by **start**).

For example:

```
/* Select the entire edit control's contents: */  
CALL entry.Select 0, 9999
```

Select (for a horizontal or vertical scroll bar)

```
result = [dialog.][control.]Select(  
[position] )
```

Gets or selects the position of the horizontal or vertical scroll bar's slider.

It returns the current position of the scroll bar slider.

Position specifies the new position of the slider.

For example:

```
/* Scroll to the beginning of the file: */  
CALL scroll.Select 1
```

Select (for a spinbutton)

```
result = [dialog.][control.]Select(  
[position] )
```

Gets or selects a spinbutton item.

It returns the current value or index selected by the spinbutton.

Position specifies the new value or index to be selected by the spinbutton . If the spinbutton is numeric, it specifies the new number selected. If text , **position** specifies the new index of the value to be selected.

For example:

```
/* Copy the current spinbutton item to the edit field: */  
item = spin.Select()  
CALL entry.Text value.item
```

Select (for a push button, check box, radio button or bagbutton)

```
result = [dialog.][control.].Select(  
[state] )
```

Gets or sets the state of the specified push button, check box, radio button or bagbutton.

It returns the current state of the button or check box. A **0** means it is in the **unchecked** state; a **1** means it is in the **checked** state; and a **2** means it is in the **indeterminate** state.

State specifies the new state of the button or check box, and should have the value **0**, **1**, or **2** as described above.

For example:

```
/* Set a check box to the 'checked' state: */  
CALL check.Select 1
```

Select (for a notebook)

```
result = [dialog.][control.].Select(  
[page] )
```

Gets or selects a page in the specified notebook.

It returns the ID of the currently selected page of the notebook.

Page specifies the ID of the new notebook page to be selected (i.e. brought to the front of the notebook).

For example:

```
/* Set the status line of the selected notebook page: */  
CALL notebook.Item notebook.Select(), 'S', num 'files found'
```

Select (for a value set)

```
result = [dialog.][control.]Select(  
[row, column] )
```

Gets or selects an item in the specified value set.

It returns the currently selected item of the value set in the form: **row column**.

Row and **column** specify the position of the new value set item to be selected . The rows and columns of the value set are numbered starting from 1.

For example:

```
/* Select the top-left value set item: */  
CALL valueset.Select 1, 1
```

Select (for a slider)

```
result = [dialog.][control.]Select(  
[tick] )
```

Gets or selects a tick mark in the specified slider.

It returns the number of the currently selected tick on the current **primary scale** of the slider.

Tick specifies the new position of the slider arm in terms of tick marks (the first tick mark is 1). If **tick** is greater than 0, then scale 1 of the slider is used and scale 1 becomes the current **primary scale** for the slider.

If **tick** is less than 0, then scale 2 of the slider is used and scale 2 becomes the current **primary scale** for the slider. In this case, **abs(tick)** is used to determine the number of the tick mark to position the slider arm at.

For example:

```
/* Get the current temperature setting of the slider: */  
temp = slider.Select() + 31
```

Select (for a container)

```
result = [dialog.][control.].Select(  
[item  
[, '[+/-]Select' | '[+/-]Mark' |  
'Cursor' | 'Next' ] ] )
```

Gets or selects an item in the specified container.

If no arguments are specified, the first selected item in the container is returned. If no item is selected, 0 is returned.

If **item** is specified, then if the second argument is:

Omitted The item specified by **item** is selected. The current first selected item is returned as the result.

[+]Select The item specified by **item** is selected. The current first selected item is returned as the result.

-Select The item specified by **item** is unselected. The current first selected item is returned as the result.

[+]Mark The item specified by **item** is marked. The current first marked item is returned as the result.

-Mark The item specified by **item** is unmarked. The current first marked item is returned as the result.

Cursor The cursor is moved to the item specified by **item** The current item containing the cursor is returned as the result.

Next The next selected item, if any, after the item specified by **item** is returned. If no selected items follow **item**, then 0 is returned.

Only the first **letter** of the second argument (i.e. **S**, **M**, **C**, or **N**) need be specified (e.g. '-M' is equivalent to '-Mark').

For example:

```
/* Mark the currently selected container item: */  
CALL container.Select container.Select(), 'Mark'
```

Range

Sets the range for the specified window, which must be a dialog, single-line edit control, horizontal or vertical scroll bar, spinbutton, value set or slider .

Range (for a dialog)

```
result = [dialog.].Range(  
[ minDx, minDy [, maxDx, maxDy]] )
```

Returns the current minimum and maximum size for a dialog that can be set using the dialog's sizing border. The result is returned as a string of the form: **minDx minDy maxDx maxDy**. If **maxDx** and **maxDy** are 0, then there is no upper bound for the size of the dialog.

If **minDx** and **minDy** are specified, they become the new current minimum width and height that can be set for the dialog using the sizing border. By default, the minimum size is the original size of the dialog at the time it is opened.

If **maxDx** and **maxDy** are specified, they become the new current maximum width and height that can be set for the dialog using the sizing border. If **maxDx** is less than **minDx**, or **maxDy** is less than **minDy**, then the upper bound for the dialog is removed. By default, there is no upper bound for the size of a dialog.

Note: The automatic resizing feature of DrRexx often does not work well when the dialog is made smaller than its original size. This is why the default minimum size is the original size of the dialog. The **Range** function can be used to establish a different minimum size in cases where the application manages the resizing of the dialog, or the automatic resizing algorithm produces acceptable results at sizes other than the default.

For example:

```
/* Allow the dialog to be sized down to  
half its original dimensions: */  
PARSE VALUE myDialog.Range() WITH mindx mindy . .  
CALL myDialog.Range mindx % 2, mindy % 2
```

Range (for a single-line edit control)

```
result = [dialog.][control.]Range(  
length )
```

Length specifies the maximum number of characters that can be entered into the edit control.

A result of 1 indicates the range was set successfully, and a result of 0 indicates that an error occurred.

For example:

```
/* Allow the user to enter up to 100 characters: */  
CALL entry.Range 100
```

Range (for a horizontal or vertical scroll bar)

```
result = [dialog.][control.]Range(  
first, last [, visible] )
```

First and **last** specify the range of values that can be scrolled over. **Visible** specifies how many items within the range are visible (defaults to 1 if not specified).

A result of 1 indicates the range was set successfully, and a result of 0 indicates that an error occurred.

For example:

```
/* Set up to scroll over a file of 100 lines, */  
/* with 25 lines visible at a time: */  
CALL scroll.Range 1, 100, 25
```

Range (for a spinbutton)

```
result = [dialog.][control.]Range(  
low, high )  
stem )
```

For a numeric spin button, **low** and **high** specify the range of values that can be selected. For a text spin button, **stem** specifies the name of a stem variable containing the values that can be selected. **Stem.0** contains the number of values, and **stem.1** through **stem.n** contain the actual values.

A result of 1 indicates the range was set successfully, and a result of 0 indicates that an error occurred.

For example:

```
/* Set up to spin over the range of values: */
/* 'Mr', 'Ms', and 'Mrs' */
value.0 = 3
value.1 = 'Mr'
value.2 = 'Ms'
value.3 = 'Mrs'
CALL spin.Range 'VALUE'
```

Range (for a value set)

```
result = [dialog.][control.]Range(
rows, columns )
```

Rows and **columns** specify the number of rows and columns in the value set control.

A result of 1 indicates the range was set successfully, and a result of 0 indicates that an error occurred.

For example:

```
/* Set up a 2 x 8 color value set table: */
CALL value.Range 2, 8
DO row = 1 TO 2
DO col = 1 TO 8
CALL value.Item row, column, 8*(row-1)+col-1
END
END
```

Range (for a slider)

```
result = [dialog.][control.].Range(  
ticks1 [, spacing1 [, ticks2 [, spacing2]]) )
```

Ticks1 specifies the number of tick marks on scale 1 of the slider. **Spacing1** specifies the spacing of the tick marks for scale 1 in pels. If 0 or omitted, the spacing is calculated automatically based on the size of the slider control and the number of ticks.

Ticks2 specifies the number of tick marks on scale 2 of the slider. If omitted, scale 2 of the slider is not defined. **Spacing2** specifies the spacing of the tick marks for scale 2 in pels. If 0 or omitted, the spacing is calculated automatically based on the size of the slider control and the number of ticks.

A result of 1 indicates the range was set successfully, and a result of 0 indicates that an error occurred.

For example:

```
/* Set up a slider with Fahrenheit and Centigrade */  
/* scales between the freezing and boiling point */  
/* of water: */  
CALL slider.Range 181, 0, 101, 0
```

Style

```
oldStyle = [dialog.][control.].Style(  
[newStyle] )
```

Returns the style bits for the specified window, which can be any dialog or control. The result is always a four byte long string containing the 32 style bits for the specified control, with the low order bits in the first byte of the string and the high order bits in the last byte of the string.

If **newStyle** is specified, the current style bits are replaced by **newStyle**, which must also be a four byte long string containing the new style bits in the same format.

For example:

```
/* Turn a style bit on: */  
CALL control.Style BITOR( control.Style(), '01000000'X )
```

Font

```
oldFont = [dialog.][control.]Font(  
[newFont] )
```

Returns the font for the specified window, which can be any dialog or control .

If **newFont** is specified, the current font is replaced by **newFont**.

Note: A font is specified as a string of the form: **size.name**, where **size** is the point size, and **name** is the family name of the font (e.g. **10.Courier**).

For example:

```
/* Use a Courier font: */  
CALL myDialog.list.Font '10.Courier'
```

Color

```
oldColor = [dialog.][control.]Color(  
[attribute] [, newColor] )
```

Returns the specified window color attribute for any dialog or control.

Attribute specifies which window color attribute the function applies to. The attribute consists of a string of characters, each of which specifies a color attribute modifier. The defined attribute modifiers are as follows:

- + Foreground (group 1)
- Background (group 1)
- A Active (group 2)
- I Inactive (group 2)
- H Highlight (group 2)
- D Disabled (group 2)
- T Text (group 3)
- M Menu (group 3)
- B Border (group 3)

The three groups represent more or less disjoint sets of attributes. In forming an attribute name, no more than one character from each group should be used. However, not all combinations of characters specify a valid color attribute. The list of valid color attribute character combinations is as follows:

- + Foreground color
- Background color
- A Active color
- I Inactive color
- AT+ Active text foreground color
- AT- Active text background color
- IT+ Inactive text foreground color
- IT- Inactive text background color
- H+ Highlight foreground color
- H- Highlight background color
- D+ Disabled foreground color
- D- Disabled background color
- M+ Menu foreground color
- M- Menu background color
- MH+ Menu highlight foreground color
- MH- Menu highlight background color
- MD+ Menu disabled foreground color
- MD- Menu disabled background color
- B Border color

Note: The order of the characters in **attribute** does not matter.

Not all windows support all color attributes. The most commonly supported attributes are foreground and background color. If **attribute** is omitted, it defaults to background color.

If **newColor** is specified, the specified window color attribute is replaced by **newColor**.

Note: A color is specified as a string of the form:

```
#index  
or #red green blue
```

where **index** is a color index, and **red**, **green** and **blue** are the color components of an RGB triplet (each component should be in the range 0 to 255).

The result of the function is also one of these two forms, depending on which form was originally used to set the corresponding color attribute.

Note: If no color attribute has been specified for a control, the null string is returned as the result.

For example:

```
/* Set an entry field to a black background with white text: */  
CALL myDialog.entry.Color , '#0 0 0'  
CALL myDialog.entry.Color '+', '#0'
```

ID

```
id = [dialog.][control.]ID()
```

Returns the ID number of the specified window, which can be any dialog or control.

Note: The ID number is the same number assigned to the control or dialog in the editor using the Name window.

For example:

```
/* Use the ID number as a digit: */  
num = 10 * num + ID()
```

Drag

```
oldDragData = [dialog.][control.]Drag(  
[newDragData] )
```

Returns the drag information associated with the specified window, which can be any control.

Note: The format of the function for container controls is described in the section on Drag (for a container).

If **newDragData** is specified, the current drag information is replaced by **newDragData**.

If **newDragData** is not the null string, then the specified window is enabled for **dragging**. That is, the user will be able to drag the control and drop it on other controls that have been enabled for **dropping**.

If **newDragData** is the null string, the specified window can not be dragged . This is the initial default for all controls.

The format of the drag string is: **type[,type,...,type][:format][=data]**, where:

type is a string denoting the type of data being dragged (e.g. **Plain text**). More than one type can be specified, if desired, with each succeeding type separated by a comma. When the control is being dragged, it can only be **dropped** on controls which accept at least one of the specified types. The first type is called the **true type**, and some non-DrRexx controls may only allow dropping controls whose **true type** they accept.

format is a string denoting the format of the data being dragged. While OS/2 allows any number of formats, DrRexx only supports two: **STRING** and **FILE** . Only the first character of the format (i.e. **S** or **F**) need be specified . If omitted, it defaults to **STRING**.

The **STRING** format denotes data passed as a REXX string (i.e. its format is simply a REXX string). It is intended mainly for exchanging information between DrRexx applications, although it can be supported by non-DrRexx applications .

The **FILE** format denotes data passed in an OS/2 file (actually, only the name of the file is passed as part of the drag and drop operation). This format is compatible with files represented by the Workplace Shell, and so may allow drag and drop operations between DrRexx applications and some Workplace Shell objects.

data is a string containing the information to be passed to the target of a drag and drop operation. If the format is **STRING**, it can be any REXX string; if the format is **FILE** it should be the name of an OS/2 file. If omitted, its default depends upon the type of control being dragged. For most controls, the default is the value returned by the Text function at the start of the drag operation . The exceptions to this rule are as follows:

Value set Defaults to the value of the item under the pointer at the start of the drag operation (i.e. the value that would be returned by the Item function for the value set item being pointed at).

List box Defaults to the text of the currently selected item (i.e. the value that would be returned by the Item function for the selected item). If more than one item is selected, each item is dragged separately. That is, the target will receive a Drop event for each separate item. If no items are selected, dragging is disabled.

Single-line edit control, multi-line edit control Defaults to the current contents of the edit control unless some text is selected, in which case it defaults to the selected text.

Notebook, slider, horizontal scroll bar, vertical scroll bar Defaults to the value that would be returned by the Select function at the start of the drag operation.

When a drag operation begins, DrRexx automatically determines the bitmap

used to represent the data based on the type of control being dragged. For most controls, the bitmap is the same one used by DrDialog to represent the



type of the control (e.g.  is used to represent a pushbutton control). However, there are a few cases where that rule does not apply:

Value set If the value set item being dragged is a bitmap (i.e. not a color



or text string), the item's bitmap is used. Otherwise the  bitmap is used.

Icon button The bitmap that appears on the button is used. If the name of the bitmap begins with '=' then the bitmap is left at its actual size, and is not scaled to the size of an standard icon. In addition, the icon button control is hidden at the beginning of a drag operation. If the drag operation fails, the button is automatically made visible again; otherwise it is the application's responsibility to make the button visible again (this allows the application to easily simulate the button **actually** being dragged to a new location).

Billboard The bitmap that appears on the billboard is used. If the name of the bitmap begins with '=' then the bitmap is left at its actual size, and is not scaled to the size of an standard icon. In addition, the billboard control is hidden at the beginning of a drag operation. If the drag operation fails, the billboard is automatically made visible again; otherwise it is the application's responsibility to make the billboard visible again (this allows the application to easily simulate the billboard **actually** being dragged to a new location).

Specifying a non-null drag string allows a DrRexx control to be dragged. In order for it to be successfully **dropped**, the target control must be enabled for dropping. For a DrRexx application this means that the Drop function has been issued for the target control with a non-null drop string. Dropping a compatible object on it will then generate a Drop event for the target control.

Note: It is possible for non-DrRexx applications to allow DrRexx controls to be dropped on them.

For example:

```
/* Allow a pushbutton to be dragged: */  
CALL button.Drag 'Command'
```

```
/* Prevent a pushbutton from being dragged: */  
CALL button.Drag ''
```

```
/* Allow a list box of file names to be dragged: */  
CALL list.Drag 'Plain text:FILE'
```

```
/* Define a draggable 'debug' control: */  
CALL text.Drag 'REXX code:STRING=SAY Control()'
```

Drag (for a container)

```
oldDragData = [dialog.][control.].Drag(  
item, [newDragData] )
```

Returns the drag information associated with the specified container control **item**.

If **newDragData** is specified, the current drag information for **item** is replaced by **newDragData**.

For more information about the format and meaning of the drag string, refer to the description of the [Drag](#) function that applies to other DrRexx control types .

Note that, for a container control, if no **data** is specified in the drag string, the associated item **label** is passed as the data when the item is dragged.

The major difference between a container control and other DrRexx controls that can be dragged is that a container control can specify drag information independently for each item it contains, not just globally for the container as a whole.

Also, because a container may contain several draggable items, it is possible to drag more than one container item at a time. When the user begins a drag operation, DrRexx checks to see if the item to be dragged is selected. If not, only the item pointed at is dragged. If it is selected, then both the item pointed at and all other selected items in the container are dragged.

Note: If more than four items are selected, they will all be dragged, but only four bitmaps will appear in the drag image. The bitmaps used are the ones associated with the items being dragged.

Refer also to the section on the [Drop event](#) for information about the data the target DrRexx control receives when the container items are dropped on it.

For example:

```
/* Add a file item and make it draggable: */  
item = cnr.Add fileName, 'BITMAP:#61'  
CALL cnr.Drag item, 'Plain text:FILE'  
  
/* Disable the item from being dragged: */  
CALL cnr.Drag item, ''
```

```
/* Create a command item and make it draggable: */
item = cnr.Add 'Merge', 'BITMAP:#45'
CALL cnr.Drag item, 'Command'
```

Drop

```
oldDropData = [dialog.][control.]Drop(
[newDropData] )
```

Returns the drop information associated with the specified window, which can be any control.

Note: The format of the function for container controls is described in the section on Drop (for a container).

If **newDropData** is specified, the current drop information is replaced by **newDropData**.

If **newDropData** is not the null string, then the specified window is enabled for **dropping**. That is, the user will be able to drop controls that are compatible with the drop criteria onto the specified control.

If **newDropData** is the null string, the specified window can not be dropped on . This is the initial default for all controls.

The format of the drop string is: **type[,type,...,type][:format][=operation]**, where:

type is a string denoting the type of data that can be dropped (e.g. **Plain text**). More than one type can be specified, if desired, with each succeeding type separated by a comma. Only dragged objects which have at least one matching type can be dropped on the control. The special type **ANY** can also be used to indicate that any type of data can be dropped on the control. If specified, it must be the only type specified.

format is a string denoting the format of the data that can be dropped. While OS/2 allows any number of formats, DrRexx only supports two: **STRING** and **FILE**. In addition, **ANY** can be specified to indicate that either format is acceptable. Only the first character of the format (i.e. **S**, **F** or **A**) need be specified. If omitted, it defaults to **STRING**. Only dragged objects which match the specified format can be dropped on the control.

The **STRING** format denotes data passed as a REXX string (i.e. its format is simply a REXX string). It is intended mainly for exchanging information between DrRexx applications, although it can be supported by non-DrRexx applications .

The **FILE** format denotes data passed in an OS/2 file (actually, only the name of the file is passed as part of the drag and drop operation). This format is compatible with files represented by the Workplace Shell, and so may allow drag and drop operations between DrRexx applications and some Workplace Shell objects.

operation is a string describing the type of operation that will be performed if an object is dropped on the control. The possible values are: **MOVE**, **COPY**, **LINK** or **ANY**. Only the first letter of the operation need be specified (i .e. **M**, **C**, **L** or **A**). If omitted, the default is **MOVE**. Only dragged objects which will allow the specified operation can be dropped on the control.

Note: No check is made to ensure that the specified operation actually occurs when an object is dropped.

Specifying a non-null drop string allows a DrRexx control to be dropped on . In order for it to be successfully dropped on, there must be a source object enabled for **dragging**. For a DrRexx application this means that the Drag function has been issued for the source control with a non-null drag string. Dropping a compatible object on it will then generate a Drop event for the target control.

Note: It is possible for non-DrRexx applications to enable controls that can be dropped on DrRexx controls. In particular, Workplace Shell file objects can typically be dropped on DrRexx controls which accept the **FILE** format and an appropriate type, or types (e.g. **ANY**, **Plain text**, **DrRexx.RES**).

For example:

```
/* Allow user to drop customers or */
/* commands onto a listbox: */
CALL list.Drop 'Customer,Command'

/* Disallow user from dropping onto a listbox: */
CALL list.Drop ''

/* Allow user to drop .BMP files onto a */
/* billboard controls: */
CALL bmp.Drop 'Bitmap:FILE'

/* Allow user to drop any kind of object */
/* onto a list box: */
CALL list.Drop 'ANY:ANY=ANY'
```

Drop (for a container)

```
oldDropData = [dialog.][control.]Drop(
item, [newDropData] )
```

Returns the drop information associated with the specified container control **item**. If **item** is **0**, it refers to the container itself, not a particular item in the container. That is, the empty space in a container not occupied by a container item can also be the target for a drop operation if desired.

If **newDropData** is specified, the current drop information for **item** is replaced by **newDropData**.

For more information about the format and meaning of the drop string, refer to the description of the Drop function that applies to other DrRexx control types .

The major difference between a container control and other DrRexx controls that can be dropped on is that a container control can specify drop information independently for each item it contains, not just globally for the container as a whole.

Refer also to the section on the Drop event for information about the data the container control receives when an object is dropped on it.

Note: An object is dropped on a particular container item, or on the container itself. Target emphasis (i.e. a box drawn around the item or container) is drawn to indicate the current drop target. When an object is dropped on the container, one or more **Drop** events are generated for the container. Information in the associated EventData for each Drop event describes the particular item within the container that was dropped on.

For example:

```
/* Create a command item to delete */
/* customer entries: */
item = cnr.Add 'Delete', 'BITMAP:#54'
CALL cnr.Drop item, 'Customer'

/* Disable the command item: */
CALL cnr.Drop item, ''

/* Create an item which allows DrDialog or */
/* DrRexx .RES files to be dropped on it: */
item = cnr.Add 'REview', 'BITMAP:#23'
CALL cnr.Drop item, 'DrDialog.RES,DrRexx.RES:FILE'
```

IsDefault

```
[dialog.][control.]IsDefault(
[ 'Object' | 'Say' | 'Dialoghint' | 'Controlhint' ] )
```

Makes the specified window the default window.

If no argument is given, or the argument is **Object**, then the specified window becomes the default window for all subsequent window function calls with omitted references until the next event or call to **IsDefault**.

If **Say** is specified as the argument, then all subsequent REXX SAY statements will direct their output to the specified window. This include implicit SAY output via the DrRexx SAY subcommand environment.

If **Dialoghint** is specified as the argument, then the hint text associated with dialogs will be displayed in the specified window whenever the pointer passes over a dialog. The window specified should be capable of displaying text. By default, dialog hints are not displayed.

If **Controlhint** is specified as the argument, then the hint text associated with controls will be displayed in the specified window whenever the pointer passes over a dialog control. If no window has been specified to display dialog hints, then dialog hints will also be displayed in the same window. The window specified should be capable of displaying text. By default, control hints are not displayed.

Note: Only the first letter of the argument need be specified (i.e . **O**, **S**, **D** or **C**).

For example:

```
CALL myDialog.list.IsDefault
DO i = 1 TO 10
CALL Add i
END
CALL Select 0
```

is equivalent to:

```
DO i = 1 TO 10
CALL myDialog.list.Add i
END
CALL myDialog.list.Select 0
```

Timer

```
rc = [dialog.][control.]Timer(
[interval] )
```

Starts or stops a timer for the specified window, which must be a dialog.

If no argument is specified, the current timer, if any, is canceled (i.e. stopped). If an argument is specified, a timer is started which will generate a **Timer** event every **interval** milliseconds. An **interval** of 0 specifies that the timer events should be generated as fast as possible.

A result of 1 indicates the request was successful and a result of 0 indicates the request failed.

For example:

```
/* Generate a timer event once a second: */  
CALL myDialog.Timer 1000  
/* Cancel any previous timer: */  
CALL D100.Timer
```

View

```
oldView = [dialog.][control.]View(  
['Bitmap' | 'Name' | 'Flowedname' |  
'Text' | 'Column' | 'Detail' |  
'Outline' | 'Hierarchy']  
[, ['<|>_;']title],  
[, cxBitmap, cyBitmap]  
[, expandBitmap, collapseBitmap] )
```

Returns the current view for the specified window, which must be a container control.

The first argument specifies the view to use for the container:

Omitted No change is made to the current view.

Bitmap Display items as bitmaps with labels centered below them.

Name Display items as a single column of bitmaps with labels on the right.

Flowedname Displays items as one or more columns of bitmaps with labels on the right. The columns are filled top-to-bottom, left-to-right.

Text Displays items as a single column of labels, with no bitmaps.

Column Displays items as one or more columns of labels, with no bitmaps. The columns are filled top-to-bottom, left-to-right.

Detail Displays items one per line, with each item consisting of one or more columns of text or bitmaps. The container may also be split into two independent groups of columns, each with its own scroll bar and a movable separator between the two groups.

Outline Displays the items as an outline, with child items indented to the

right of parent items. Levels in the outline may be expanded or collapsed independently. Unlike the **Hierarchy** view, no lines are drawn to indicate the levels in the structure. Individual items in the outline are displayed as bitmaps with their labels on the right.

Hierarchy Displays the items as a hierarchy, with child items indented to the right of parent items. Levels in the outline may be expanded or collapsed independently. Lines are drawn to the left of the items to indicate the levels in the structure. Individual items are displayed as bitmaps with their labels on the right .

Only the first letter (i.e. **B**, **F**, **N**, **T**, **C**, **D**, **O** or **H**) of the first argument need be specified.

Title specifies the title to display at the top of the container. If not specified, the current title is not changed. The title may optionally begin with one or more of the following special formatting characters:

None The title is centered with no separator line.

< The title is left justified.

| The title is centered.

> The title is right justified.

_ A separator line is drawn between the title and the rest of the container.
; End of the formatting characters; the title begins with the next character . This character is optional. If not specified, the title begins with the first character **not** in this list.

If **title** is specified, but consists **only** of formatting characters (e.g . ';'), the current title, if any, is removed from the container.

CxBitmap and **cyBitmap** specify the size to which all bitmaps displayed in the container are scaled. If not specified, no change is made to the current bitmap size . If both are specified as 0, the default system bitmap size is used.

ExpandBitmap and **collapseBitmap** specify the bitmaps to use in the **Outline** or **Hierarchy** views to indicate that an item may be expanded or collapsed, respectively. If not specified, no change is made to the current bitmaps used for this purpose . If specified, both should be strings of the form: **dll:#resource**, where **resource** is the resource number of the bitmap within the DLL specified by **dll** (e.g. **BITMAP:#50**). Alternatively, the form: **filename.BMP** can also be used, where **filename.BMP** is the name of a file containing a bitmap in **.BMP** format.

For example:

```
/* Display the hierarchy view of a container: */  
CALL container.View 'H', '<_;Hierarchy view',,,  
'MYDLL:#1', 'MYDLL:#2'
```

SetStem

```
[dialog.][control.]SetStem(  
[stem]  
[, '[+/-]Select' | '[+/-]Mark' |  
'Format' | 0 | item] )
```

Sets a list of values for the specified window, which must be a container control.

The list of values to be set are in the stem variable specified by **stem**. If **stem** is omitted, **STEM** is used as the name of the stem variable.

Stem.0 contains the number of values in the list to be set. **Stem.1** through **stem.n** contain the individual values to set. The type of values set depends on the second argument as follows:

```
[+]Select All items in stem are selected.  
-Select All items in stem are unselected.  
[+]Mark All items in stem are marked.  
-Mark All items in stem are unmarked.
```

Format The items in **stem** are used to determine the number, format and content of each column in the detail view of the container. The number of columns of data is specified by **stem.0**. **Stem.1** through **stem.n** specify the format and content of columns 1 through n. Each item is a string of 0 or more of the following characters:

```
= Column will display a bitmap. If specified, column values should be  
strings of the form: dll:#resource, where resource is the resource number of  
the bitmap within the DLL specified by dll (e.g. BITMAP:#50). Alternatively,  
the form: filename.BMP can also be used, where filename.BMP is the name of a  
file containing a bitmap in .BMP format. If not specified, the column data  
is assumed to be text.  
~ Column is invisible (i.e. not shown).  
X Column data is read-only. If not specified, the user can edit the data in  
this column if it is not a bitmap.  
. This column is the last column in the left part of the split view. If more  
than one column format string contains a period, the highest numbered column  
is the one used as the split point. If no column format string contains a  
period, the container will not display a split view.  
^ Column data is top aligned.  
V Column data is bottom aligned.  
- (minus sign) Column data is vertically centered (the default).  
< Column data is left aligned.  
> Column data is right aligned.  
| Column data is horizontally centered (the default).  
_ (underscore) Column title will have a separator drawn below it.  
! Column data will have a vertical separator drawn on the right.
```

Note: If the detail view is used in an application, the format of the column data must be specified prior to adding any items to the container. Once the format has been specified, subsequent attempts to change the following aspects of the format will be ignored:

- oThe number of columns
- oThe format of the column data (i.e. bitmap or text)

All other aspects of the column format can be changed as needed.

0 The values in **stem** are used to set the column titles. Note that the column titles need not have the same format as the data they describe (e.g. the column data could be text, while the column title is a bitmap). If a title value begins with a '=', then the title is assumed to be a bitmap specified by a string of the form: **=dll:#resource**, where **resource** is the resource number of the bitmap within the DLL specified by **dll** (e.g. **=BITMAP:#50**) . Alternatively, the form: **=filename.BMP** can also be used, where **filename.BMP** is the name of a file containing a bitmap in **.BMP** format. If the title value does not begin with a '=', the title is assumed to be text .

item The values in **stem** are used to set new values for the column data of the item specified by **item**.

If the second argument is omitted, it defaults to **Select**.

For the first three cases, only the first letter of the second argument need be specified (i.e. **S**, **M** or **F**).

For example:

```
/* Set up the detail view format: */
format.0 = 3 /* 3 columns of data */
format.1 = '= _!' /* Bitmap, horizontal/vertical separators */
format.2 = '. _!' /* Last column in split view, H/V seps */
format.3 = '> _!' /* Right aligned, H/V seps */
CALL container.SetStem 'FORMAT', 'F'
```

```
/* Set up the detail view titles: */
title.0 = 3
title.1 = 'Type'
title.2 = 'Part #'
title.3 = 'Quantity'
CALL container.SetStem 'TITLE', 0
```

```
/* Add an item: */
item = container.Add( 'Widget' )
data.0 = 3
data.1 = 'BITMAP:#27'
data.2 = '12-456AQ'
data.3 = '52'
CALL container.SetStem 'DATA', item
```

```
/* Mark all currently selected items: */
CALL container.GetStem 'ITEM', 'S'
```

```
CALL container.SetStem 'ITEM', 'M'
```

GetStem

```
[dialog.][control.]GetStem(  
[stem]  
[, 'Select' | 'Mark' | 'Cursor' | 0 | item] )
```

Returns all requested values for the specified window, which must be a container control.

The values are returned in the stem variable specified by **stem**. If **stem** is omitted, **STEM** is used as the name of the stem variable.

On return, **Stem.0** contains the number of values returned. **Stem.1** through **stem.n** contain the individual values returned. The type of values returned depends on the second argument as follows:

Select All currently *selected* items are returned in **stem**.

Mark All currently *marked* items are returned in **stem**.

Cursor The item containing the *cursor* is returned in **stem**.

0 The column titles for the detail view are returned in **stem**.

item The values for each of the columns displayed in the detail view of **item** are returned in **stem**.

If the second argument is omitted, it defaults to **Select**.

For the first three cases, only the first letter of the second argument need be specified (i.e. **S**, **M** or **C**).

For example:

```
/* Delete all currently selected items: */  
CALL container.GetStem  
DO i = 1 TO stem.0  
CALL container.Delete stem.i  
END
```

```
/* Mark all currently selected items: */  
CALL container.GetStem 'ITEM', 'S'  
CALL container.SetStem 'ITEM', 'M'
```

Controls

```
[dialog.][control.]Controls(  
[stem] )
```

Returns the names of all controls for the specified window, which can be any dialog or control.

The names are returned in the stem variable specified by **stem**. If **stem** is omitted, **CONTROLS** is used as the name of the stem variable.

Stem.0 contains the number of names returned. **Stem.1** through **stem .n** contain the names of each control for the specified window. If a control was not assigned a name using the DrDialog Name window, a name of the form **Dnnn** (for dialogs) or **Cnnn** (for controls), where **nnn** is the numeric ID of the dialog or control, is returned instead. **Stem.1** always contains the name of the dialog .

The function may be applied to a dialog or to any control contained within the dialog. In either case, the result returned is always the list of names for the entire dialog.

For example:

```
/* Put all control names into a list box: */  
CALL myDialog.Controls  
DO i = 1 TO controls.0  
CALL myDialog.list.Add controls.i  
END
```

Classes

```
[dialog.][control.]Classes(  
[stem] )
```

Returns the class names of all controls for the specified window, which can be any dialog or control.

The class names are returned in the stem variable specified by **stem**. If **stem** is omitted, **CLASSES** is used as the name of the stem variable.

Stem.0 contains the number of class names returned. **Stem.1** through **stem.n**

contain the class names of each control for the specified window . **Stem.1** always contains the class name of the dialog (i.e. **DIALOG**) . The order of the class names returned corresponds to the order of the names returned by the Controls function.

The function may be applied to a dialog or to any control contained within the dialog. In either case, the result returned is always the list of class names for the entire dialog.

For example:

```
/* Put all control names and classes into a list box: */
CALL myDialog.Controls
CALL myDialog.Classes
DO i = 1 TO controls.0
CALL myDialog.list.Add controls.i 'is a' classes.i
END
```

DrRexx menu functions

Besides the window functions for interacting with Presentation Manager dialogs and controls, DrRexx also defines a number of functions for interacting with drop-down menus. Each of these menu functions is invoked using an *object oriented* syntax style similar to the window functions:

```
[dialog.][label.]function ( arguments )
```

where **dialog** refers to the name assigned to a dialog using the DrDialog Name window, and **label** refers to the label assigned to one or more menu items using the drop-down menu tool. For example, the statement:

```
CALL myDialog.Save.Disabled 1
```

might be used to disable all menu items labeled **Save** in the drop-down menu for dialog **myDialog**.

Note: Unlike window functions which apply to a single dialog or control , menu functions apply to *all* menu items with a specified label within a drop-down menu.

The use of dialog names and labels is optional. If either or both are omitted, the following rules apply:

oIf only **dialog** is omitted, the function applies to all menu items with the specified label within the dialog in which the event occurred.

oIf both **dialog** and **label** are omitted, the function applies to the menu item generating the event.

Note: In the case of the **Init** procedure used to start a DrRexx application, there is no event. Therefore, any menu function calls it contains must use a fully qualified name. This is the only exception to the above set of rules .

If no name was assigned to a dialog using the DrDialog Name window, **Dnnn**, where **nnn** is the ID number of the dialog, may be used in place of the dialog name (e .g. **D100.Save.Disabled(1)**).

As with window functions, DrRexx also supports the following variation for invoking menu functions:

```
functionFOR( dialog, label [, arguments] )
```

where **dialog** and **label** are as described above. For example:

```
CALL DisabledFor 'myDialog', 'Save', 1
```

would be the equivalent way of writing the previous example using the alternate syntax style.

The main use of this alternate style is in cases where the dialog and label name are determined dynamically at run-time, rather than statically at edit time .

The defined menu functions are as follows:

Name	Description
<u>MenuPopUp</u>	Display a pop-up menu
<u>MenuChecked</u>	Get/Set a menu item's checked state
<u>MenuDisabled</u>	Get/Set a menu item's disabled state
<u>MenuText</u>	Get/Set a menu item's text

MenuPopUp

```
rc = [dialog.][label.]MenuPopUp(  
[initial] )
```

Displays the specified menu item as a pop-up menu at the current pointer position. Returns **1** if the pop-up menu is displayed successfully; and **0** otherwise . The specified menu item must be a sub-menu within the specified dialog's drop-down menu.

If **Initial** is specified, it should be the label of a menu item within the specified sub-menu. The pop-up menu will be displayed with the **initial** menu item centered under the pointer and already selected. Note that the specified menu item must not be a static menu item (i.e. it must have some REXX code associated with it).

If **initial** is not specified, the pop-menu will be displayed with the pointer in the lower-left hand corner of the menu and the first non-static menu item already selected.

For example:

```
/* Display a pop-up menu: */  
CALL menuDialog.options.MenuPopUp
```

```
/* Display a pop-up menu with the 'default' item selected: */  
CALL menuDialog.options.MenuPopUp 'default'
```

MenuChecked

```
oldState = [dialog.][label.]MenuChecked(  
[newState] )
```

Returns the current checked state of the specified menu items. A **0** means the menu items are not checked, and a **1** means they are checked.

newState specifies the new checked state for all specified menu items, and should be one of the two values described above.

For example:

```
/* Toggle the current menu item's checked state: */  
CALL MenuChecked 1 - MenuChecked()
```

```
/* 'Check' a menu item: */  
CALL myDialog.Attached.MenuChecked 1
```

MenuDisabled

```
oldState = [dialog.][label.]MenuDisabled(  
[newState] )
```

Returns the current disabled state of the specified menu items. A **0** means the menu items are not disabled, and a **1** means they are disabled.

newState specifies the new disabled state for all specified menu items, and should be one of the two values described above.

For example:

```
/* Disable all 'save' menu options: */  
CALL Save.MenuDisabled 1
```

MenuText

```
oldText = [dialog.][label.]MenuText(  
[newText] )
```

Returns the current text of the specified menu items.

newText specifies the new text for all specified menu items. A '~' character in **newText** can be used to specify a keyboard accelerator (e.g. '~Save' specifies that **alt-S** is the keyboard accelerator character for this menu item).

For example:

```
/* Include the file name in the save option: */  
CALL Save.MenuText '~Save' filename
```

DrRexx concurrency functions

A DrRexx application can perform *concurrent* execution of more than one task at a time using the following functions:

Name Description

Start Start a new concurrent thread of execution

Stop Stop an existing concurrent thread of execution

Result Wait for a concurrent thread to complete

Notify Notify the main thread of an event

Use Get/Release exclusive use of a shared resource

Val Get/Set the value of a shared variable

Sleep Suspend execution for a specified time

The following sections describe each of these functions in detail. In addition, the [Concurrent programming example](#) section gives a complete example that illustrates how these functions can be used together in writing a program.

Start

```
tid = Start( label [, argument] )
```

Starts a new concurrent thread of execution and returns the ID of the new thread.

The new thread begins execution at **label**, which should either be the label of a global procedure or statement in the external REXX code for the program. If **argument** is specified, it will be passed to the new thread as its first argument. If **argument** is not specified, then the new thread's first argument is the null string . The second argument to the new thread is always its own thread ID.

Threads started with the **Start** function have a slightly different set of capabilities than the main thread used to start a DrRexx application. In particular, **started** threads only have access to the following DrRexx functions:

- oStart
- oStop
- oResult
- oNotify
- oUse
- oVal
- oSleep
- oClipboard
- oDrRexxVersion

Any **started** thread attempting to use a DrRexx function not in this list will generate a **Function not found** error.

Trace and **SAY** output for a **started** thread is handled in exactly the same way as for the main DrRexx thread (i.e. it will appear in the [Run-time](#) window unless directed to another control using the [IsDefault](#) function).

If a **started** thread attempts to read from an empty data queue, a **HALT** condition will be raised (unlike the main thread, which will display a pop-up dialog for user input).

Other than these restrictions, **started** threads have full access to all features and facilities of the REXX language.

For example:

```

/* Start a new thread to make a list
of all files on the C: drive: */
CALL Start 'FileList', 'C:\*.*'
...

/* List all files in a specified subtree
and put them in the shared variable 'files': */
FileList:
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'list', 'FOS'
DO i = 1 TO list.0
CALL Val 'files.'i, list.i
END
CALL Notify 'MyDlg', 'ListDone', 'files'
RETURN

```

Stop

```
rc = Stop( tid )
```

Stops execution of the thread whose ID is **tid**. Returns 1 if successful, and 0 otherwise.

Note: This function should be used with caution since it immediately aborts the specified thread without allowing it to perform any clean-up processing

.

For example:

```

/* User is exiting application,
cancel the background thread: */
CALL Stop backgndTid

```

Result

```
value = Result( [tid] )
```

Waits for the thread whose ID is specified by **tid** to complete processing and returns its result. If **tid** is not specified, it waits for the next thread to complete processing, and returns its result.

In all cases, the variable **tid** is set equal to the ID of the thread that completed processing. This is most useful in the case where **tid** is not specified as an argument.

For example:

```
/* Begin counting files matching a
specified pattern: */
agent = Start( 'CountFiles', fileSpec )
...
/* Do additional processing here */
...
/* Now wait for the result */
files = Result( agent )

/* Count number of files matching a
file spec in a specified subtree: */
CountFiles:
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'files', 'FOS'
RETURN files.0
```

Notify

```
Notify( dialog [, event] [, data] )
```

Generates a **Notify** event for **dialog**, which must be the name of a currently open dialog.

If specified, **event** and **data** are passed as data for the **Notify** event and can be retrieved within the **Notify** event handler using the [EventData](#) function. Each defaults to the null string if not specified.

For example:

```
/* Start a new thread to delete a list of files: */
CALL Start 'DeleteList', 'C:\TEMP\*.*'
...

/* Delete all specified files in a directory
and notify after each delete, and when
completed: */
DeleteList:
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'files', 'FO'
DO i = 1 TO files.0
'DEL' files.i
```

```
CALL Notify 'MyDlg', 'Delete', files.i
END
CALL Notify 'MyDlg', 'Done'
RETURN
```

Use

```
rc = Use( name [, request] )
```

Gets or releases exclusive access to the resource specified by **name**, which can be any arbitrary string designating a logical resource which multiple concurrent threads of execution may be competing to use.

If **request** is not 0, the requesting thread is suspended until the resource specified by **name** is available. When control returns, the current thread is marked as the **owner** of the resource until it releases control by invoking **Use** with a 0 **request** argument. All other threads requesting use of the same resource will be suspended until the owning thread releases it.

If **request** is 0, the requesting thread releases control of the specified resource. If any other threads are waiting on the same resource, one of them will be allowed to resume execution and become the new owner of the resource. For this operation to succeed, the requesting thread must already own the resource.

If **request** is not specified, the current status of the resource is immediately returned as the result. A result of 1 indicates that the resource is currently owned, and a result of 0 indicates that the resource is not currently owned .

For example:

```
/* Read the next element from the
REXX data queue (where multiple
threads are reading simultaneously): */
CALL Use 'Q', 1
DO WHILE queued() = 0
CALL Sleep 100
END
PULL data
CALL Use 'Q', 0
...
```

Val

```
value = Val( name [, value] )
```

Gets and optionally sets the **value** associated with a specified **name** that is shared across all concurrent threads of execution.

Each concurrent thread has a completely separate set of REXX variables from every other thread. While this is useful in general, there may be instances when concurrent threads need to share data. The **Val** function can be used to accomplish this .

Val allows arbitrary value strings to be associated with arbitrary name strings. Each concurrent thread has access to the same set of name/value pairs using the **Val** function. Access to the names and values are serialized so that no two threads can access a given name at the same time. The Use function can be used to further restrict access to a variable or group of variables over longer periods of time than a single access.

If **value** is specified, it becomes the new value associated with **name**, and is returned as the result.

If **value** is omitted, the current value associated with **name** is returned as the result. If no value has yet been associated with **name**, the null string is returned.

For example:

```
/* Start a new thread to make a list
of all files on the C: drive: */
CALL Start 'FileList', 'C:\*.*'
...

/* List all files in a specified subtree
and put them in the shared variable 'files.i': */
FileList:
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'list', 'FOS'
DO i = 1 TO list.0
CALL Val 'files.'i, list.i
END
CALL Notify 'MyDlg', 'ListDone', 'files'
RETURN
```

Sleep

Sleep([time])

Causes the requesting thread to suspend execution for **time** milliseconds.

If **time** is not specified, it allows the next ready thread to execute. The operating system will return control to the suspended thread as soon as its turn comes up again.

For example:

```
/* Wait for some work to process
in the REXX data queue: */
DO FOREVER
DO WHILE queued() = 0
CALL Sleep 100
END
PULL workItem
/* Process the work item read from the queue... */
END
```

Concurrent programming example

To illustrate how the DrRexx concurrency functions can be used in writing an application, consider the following example:

Write a program that counts lines in a group of files. Assume there is a command, **COUNT**, which accepts a single file name and returns as its return code the number of text lines in the file.

First, look at the following non-concurrent solution to this problem:

```
/* Usage: total = CountLines( fileSpec ) */
/* where 'fileSpec' may contain wildcard characters */
/* Returns: Total number of lines in all files */
/* matching 'fileSpec' */
CountLines: PROCEDURE
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'files', 'F0'
total = 0
DO i = 1 TO files.0
'COUNT' files.i
total = total + rc
END
```

RETURN total

While the above procedure is simple and straightforward, it has two potential problems. First, if there are a large number of files, it may take a long time to count all the lines. While the DrRexx application is busy performing this routine, it cannot be processing other requests from the user (e.g. aborting the loop if the user gets tired of waiting for the result). Secondly, it may not be fully utilizing the capabilities of the user's machine. Since each file is counted sequentially, there is no overlapping of file I/O and processing.

The following code illustrates how both of these problems can be overcome using the DrRexx concurrency functions:

```
/* Usage: CountLines( fileSpec ) */
/* where 'fileSpec' may contain wildcard characters */
/* Returns immediately. The application is notified */
/* of the total later via a 'Notify' event */
CountLines: PROCEDURE
PARSE ARG fileSpec
CALL Start 'Manager', fileSpec
RETURN
```

```
Manager:
/* Number of concurrent 'counter' threads: */
threads = 5
PARSE ARG fileSpec
CALL SysFileTree fileSpec, 'files', 'F0'
total = 0
DO i = 1 TO files.0
QUEUE files.i
END
DO i = 1 TO threads
thread.i = Start( 'Counter' )
END
total = 0
DO i = 1 TO threads
total = total + Result( thread.i )
END
CALL Notify 'MyApp', 'Total', total
RETURN
```

```
Counter:
total = 0
DO FOREVER
CALL Use 'QUEUE', 1
IF queued() THEN PULL file
ELSE file = ''
CALL Use 'QUEUE', 0
IF file = '' THEN RETURN total
```

```
'COUNT' file
total = total + rc
END
```

To help better understand this example, a brief explanation of each of the above procedures follows:

CountLines Starts another thread of execution to manage the counting process and passes it the **fileSpec** to be processed. It then returns immediately to the caller. The final total will later be sent to the main application thread via a **Notify** event.

Manager This is the thread that manages the counting process. It consists of four major sections:

- 1.Enumerate the files to be counted and queue them to the REXX data queue for processing by the counting threads.
- 2.Start a fixed number of counting threads and save their ID's.
- 3.Total the results returned by each of the counting threads.
- 4.Notify the main application thread of the final total.

Counter An arbitrary number of threads are started to count lines. Each thread is a simple loop that:

- oReads the name of a file to count from the REXX data queue
- oIf the queue is empty, returns the total number of lines it has counted
- oOtherwise, it adds the number of lines in the current file to its total and returns to step 1.

There are a number of interesting things to note about this solution compared to the first:

- oIt is significantly longer. This is probably true of most concurrent programs when compared to their non-concurrent counterparts. Concurrency often comes at the price of additional complexity and coding and debugging time.

- oThe use of the REXX variable **total** in both **Manager** and **Counter** does not create a problem. Each routine executes on its own thread and has a separate name space from the other (this is also true for the multiple instances of **Counter** running simultaneously).

- oNote that each instance of **Counter** must obtain exclusive access to the REXX data queue (via the **Use** function) prior to examining it. Failure to do so might result in **race** conditions and lead to erratic program behavior. In particular, if between the time thread 1 saw that there was data on the queue and read it thread 2 got control and read the last queue element, it would cause thread 1 to halt abnormally by reading an empty queue. As a result, thread 1 would not correctly return its total, creating an error in the final tally. Note also that it would be a serious error to have written **Counter** as follows:

Counter:

```

total = 0
DO FOREVER
CALL Use 'QUEUE', 1
IF queued() = 0 THEN RETURN total
PULL file
CALL Use 'QUEUE', 0
'COUNT' file
total = total + rc
END

```

The error is that when the queue becomes empty, **Counter** returns without releasing its use of the REXX data queue. As a result, all other instances of **Counter** still running will go into an infinite wait the next time they attempt to gain access to the REXX data queue. Such errors are easy to make and sometimes difficult to find when writing concurrent applications.

Also note that while the following version of **Counter** is valid, it too contains a serious problem:

```

Counter:
total = 0
DO FOREVER
CALL Use 'QUEUE', 1
IF queued() THEN DO
PULL file
'COUNT' file
total = total + rc
END
ELSE file = ''
CALL Use 'QUEUE', 0
IF file = '' THEN RETURN total
END

```

The problem in this case is that the REXX data queue is not released until **after** the current file has been processed. As a result, this eliminates nearly all overlapping of file I/O and processing time, and ends up being just a more complicated version of the original non-concurrent solution.

DrRexx miscellaneous functions

In addition to the window functions, the DrRexx run-time environment also defines the following additional REXX functions:

Name	Description
<u>ModalFor</u>	Return the result of a modal dialog
<u>EventData</u>	Return the data associated with the current event
<u>Event</u>	Return the name of the current event
<u>Control</u>	Return the name of the current event's control

Class Return the class of the current event's control

Dialog Return the name of the current event's dialog

Dialogs Return the names of all defined dialogs

FilePrompt Return the name of a file selected using the system file dialog

Clipboard Get/Set the contents of the system clipboard

ScreenSize Returns the size of the display screen

DrRexxVersion Get the DrRexx version

ModalFor

```
result = ModalFor( dialog [, alias] [, registeredName] )
```

Returns the result returned after displaying the modal dialog specified by **dialog**.

If **alias** is specified, the dialog is created with the name **alias**. Because each open dialog must have a unique name, **alias** can be used in the case where **dialog** is already open to give the modal dialog a unique name. **Alias** must not be the name of a currently open dialog.

If **registeredName** is specified, the dialog is registered with the Presentation Manager so that it will appear on the Window List with the specified name, and the dialog is owned by the desktop. If **registeredName** is not specified, the dialog is not registered with PM, and in addition the current dialog is made the owner of the newly created dialog. The **current** dialog is the dialog to which the current event belongs.

Invoking **ModalFor** disables all other dialogs currently open and prevents the user from interacting with them until the modal dialog is closed. When the modal dialog is closed, all previously disabled dialogs are re-enabled.

The event handler invoking **ModalFor** is **suspended** until an event handler for the modal dialog executes a RETURN statement. The value, if any, specified on the RETURN statement is then returned as the result of the **ModalFor** function.

Note: Normally, event handlers exit by either **falling out** the bottom of the event handler code or executing a SIGNAL RETURN statement. However, in the case of a modal dialog it is important to remember that any event handler that terminates display of the modal dialog must exit using a RETURN statement (either with or without a result). Failure to do so will prevent control from ever returning to the event handler that originally displayed the modal dialog.

For example:

```
/* Get the user password and */  
/* exit if it is invalid: */
```

```
IF ModalFor( 'password' ) <> 'SECRET' THEN EXIT
```

EventData

```
EventData( [stem] )
```

Returns the list of data items associated with the current event in the stem variable specified by **stem**. If **stem** is omitted, it defaults to **EVENTDATA**.

The number of data items returned is specified by **stem.0**. **Stem.1** through **stem.n** contain the individual data items associated with the current event . For a description of the data returned by a specific event for a particular control type, refer to the Controls section.

For example:

```
/* Mark each container item when the cursor */  
/* touches it: */  
CALL EventData 'DATA'  
IF data.2 = '+CURSOR' THEN CALL Select data.1, 'MARK'
```

Event

```
event = Event()
```

Returns the name of the current event (e.g. 'Click').

If no event has occurred yet (i.e. the **Init** function is being processed), the null string is returned.

For example:

```
/* Display the current event: */  
SAY Event() 'occurred for' Control() '('Class()')'
```

Control

```
name = Control()
```

Returns the name of the control generating the current event (e.g. 'list').

If no event has occurred yet (i.e. the **Init** function is being processed), the null string is returned.

For example:

```
/* Display the current event: */  
SAY Event() 'occurred for' Control() '('Class()')
```

Class

```
class = Class()
```

Returns the class of the control generating the current event (e.g. 'LISTBOX').

If no event has occurred yet (i.e. the **Init** function is being processed), the null string is returned.

For example:

```
/* Display the current event: */  
SAY Event() 'occurred for' Control() '('Class()')
```

Dialog

```
name = Dialog()
```

Returns the name of the dialog containing the control which generated the current event (e.g. 'password').

If no event has occurred yet (i.e. the **Init** function is being processed), the null string is returned.

For example:

```
/* Display the current event: */  
SAY Event() 'occurred for' Control() 'in' Dialog()
```

Dialogs

```
Dialogs( [stem] )
```

Returns the names of all defined dialogs.

The names are returned in the stem variable specified by **stem**. If **stem** is omitted, **DIALOGS** is used as the name of the stem variable.

Stem.0 contains the number of names returned. **Stem.1** through **stem .n** contain the names of each defined dialog. Note that the list includes dialogs currently open as well as those defined but not currently open.

For example:

```
/* Put all dialog names into a list box: */  
CALL Dialogs  
DO i = 1 TO dialogs.0  
CALL myDialog.list.Add dialogs.i  
END
```

FilePrompt

```
file = FilePrompt( [pattern] [, title] [, ok]  
[, 'Open' | 'Save'] )
```

Prompts the user for the name of a file using the standard system file dialog , and returns the name of the file selected as the result.

Pattern specifies the type of file the user should select, and may contain a path or wildcard characters (i.e. '*' or '?'). If **pattern** is not specified, it defaults to '*.*'.

Title specifies the text to display on the file dialog title bar. If not specified, it defaults to 'Open file...' for a file open dialog, and 'Save file as...' for a file save dialog.

Ok specifies the label for the button used to accept the current file as the file to return as the result. If not specified, it defaults to 'Open' for a file open dialog, and 'Save' for a file save dialog.

The fourth argument specifies whether the file dialog is for opening a file (i.e. 'Open'), or saving a file (i.e. 'Save'). Only the first character (i.e. 'O' or 'S') need be specified . If omitted, it defaults to being a file open dialog.

If the user selects a file using the file dialog, the name of the file is returned as the result. If the user cancels the file dialog without selecting a file , the null string is returned as the result.

For example:

```
/* Prompt the user for the name of a .BMP file: */  
file = FilePrompt( '*.BMP', 'Enter name of bitmap file to open' )
```

Clipboard

```
result = Clipboard( [text] )
```

Returns the current contents of the system clipboard.

If **text** is specified, it replaces the current contents of the system clipboard.

For example:

```
/* Put current list selection into the clipboard: */  
CALL Clipboard list.Item( list.Select() )
```

ScreenSize

```
result = ScreenSize()
```

Returns the size of the display screen in the form: **width height** (e .g. '1024 768').

For example:

```
/* Center a window on the screen: */  
PARSE VALUE dialog.Position() WITH x y cx cy  
PARSE VALUE ScreenSize() WITH dx dy  
CALL dialog.Position (dx - cx)%2, (dy - cy)%2, cx, cy
```

DrRexxVersion

```
result = DrRexxVersion()
```

Returns the version of DrRexx in the form: **v.vv mm/dd/yy exe file** (e .g.
'2.10 09/18/93 D:\DRDIALOG\DRREXX.EXE') .

For example:

```
/* Display the version of DrRexx: */  
CALL display.Text DrRexxVersion()
```


DrDialog controls

DrDialog supports the following types of controls:

- o Dialog control
- o Push button control
- o Check box control
- o Radio button control
- o Text box control
- o Notebook control
- o Container control
- o List box control
- o Single-line edit control
- o Multi-line edit control
- o Combo box control
- o Spin button control
- o Value set control
- o Vertical scroll bar control
- o Horizontal scroll bar control
- o Slider control
- o Group box control
- o Frame control
- o Rectangle control
- o Billboard control
- o Canvas control
- o Paint control
- o Bitmap button control
- o Bag button control
- o Turtle control
- o Bitmap control
- o User defined control
- o Marquee control

Dialog control

A dialog is not actually a *control*, but nevertheless has a set of defined events and functions similar to a control. For that reason, it is described here as if it were a control.

The events defined for a dialog are:

Event Description

Init The dialog is being initialized when the dialog is opened. This is always the first 'Init' event handler run when a dialog is opened (i.e . it is run before the 'Init' event handlers defined for any controls within the dialog).

Open The dialog is completing initialization when the dialog is opened. This routine is run after all 'Init' event handlers for the dialog have completed .

Exit The dialog has been closed

Timer A time interval set by the 'Timer' function has expired

Move The dialog has moved
Size The dialog has changed size
Key The user has pressed a key on the keyboard.

The **EventData** function can retrieve the following information about the event :

EventData.1 The name of the key (e.g. 'a', 'F2 ', 'NEWLINE').
EventData.2 The class of the key (i.e. 'CHAR', 'FUNCTION' or 'CONTROL').
EventData.3 A string containing which *shift* keys (i.e. **ALT**, **CTRL** or **SHIFT**) were pressed (e.g. ", 'ALT', 'CTRLSHIFT ', 'ALTCTRLSHIFT').
EventData.4 **1** if the **Alt** key was pressed, and **0** otherwise.
EventData.5 **1** if the **Ctrl** key was pressed, and **0** otherwise.
EventData.6 **1** if the **Shift** key was pressed, and **0** otherwise.

For more details about the information received for a particular key, run the **TestKey.RES** program in the **SAMPLE** folder. This program displays all **EventData** fields received for each key pressed.

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)
Notify Another program is notifying the dialog of an event.

The **EventData** function can retrieve the following information about the event :

EventData.1 The name of the event
EventData.2 Data associated with the event

Note: A **Notify** event is generated using the Notify function. The actual data retrieved using **EventData** depends on the information passed as arguments to the **Notify** function.

GetFocus The dialog has been given the input focus
LoseFocus The dialog has lost the input focus
Any An event *not* handled by a control specific or class handler for the dialog has occurred

The DrRexx window functions that can be applied to a dialog are:

Open Open (i.e. create) a new dialog
Close Close (i.e. destroy) a dialog
Owner Get/Set a dialog's owner
Frame Get the size of a dialog's frame
Timer Start/stop a dialog timer
Text Get/Set a dialog's title bar text
Style Get/Set a dialog's style mask
Font Get/Set a dialog's title bar font
Color Get/Set a dialog attribute's color
ID Get a dialog's window ID
Position Get/Set a dialog's position and size
Hide Hide a dialog
Show Show a dialog
Visible Get/Set a dialog's visibility state
Top Make a dialog the topmost window
Bottom Make a dialog the bottommost window
Enable Enable a dialog
Disable Disable a dialog
Enabled Get/Set a dialog's enabled state
Focus Give a dialog the input focus

IsDefault Make a dialog the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Push button control



Creates a pushbutton control.

The events defined for a pushbutton control are:

Event Description

Click The user has clicked the pushbutton

Init The pushbutton is being initialized when the dialog is opened

Drop An object has been dropped on the pushbutton. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a push button control are :

Select Get/Set a push button control's select state

Text Get/Set a push button control's text

Style Get/Set a push button control's style mask

Font Get/Set a push button control's text font

Color Get/Set a push button control attribute's color

ID Get a push button control's window ID

Position Get/Set a push button control's position and size

Hide Hide a push button control

Show Show a push button control

Visible Get/Set a push button control's visibility state

Top Make a push button the topmost control

Bottom Make a push button the bottommost control

Enable Enable a push button control

Disable Disable a push button control

Enabled Get/Set a push button control's enabled state

Focus Give a push button control the input focus

Drag Enable/Disable dragging a push button control

Drop Enable/Disable dropping on a push button control

IsDefault Make a push button control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Check box control



Creates a check box control.

The events defined for a check box control are:

Event Description

Click The user has clicked the check box

Init The check box is being initialized when the dialog is opened

Drop An object has been dropped on the check box. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a check box control are :

- [Select](#) Get/Set a check box control's select state
- [Text](#) Get/Set a check box control's text
- [Style](#) Get/Set a check box control's style mask
- [Font](#) Get/Set a check box control's text font
- [Color](#) Get/Set a check box control attribute's color
- [ID](#) Get a check box control's window ID
- [Position](#) Get/Set a check box control's position and size
- [Hide](#) Hide a check box control
- [Show](#) Show a check box control
- [Visible](#) Get/Set a check box control's visibility state
- [Top](#) Make a check box the topmost control
- [Bottom](#) Make a check box the bottommost control
- [Enable](#) Enable a check box control
- [Disable](#) Disable a check box control
- [Enabled](#) Get/Set a check box control's enabled state
- [Focus](#) Give a check box control the input focus
- [Drag](#) Enable/Disable dragging a check box control
- [Drop](#) Enable/Disable dropping on a check box control
- [IsDefault](#) Make a check box control the current default control
- [Controls](#) Get a list of all dialog controls
- [Classes](#) Get a list of all dialog control classes

Radio button control



Creates a radio button control.

The events defined for a radio button control are:

Event Description

Click The user has clicked the radio button

Init The radio button is being initialized when the dialog is opened

Drop An object has been dropped on the radio button. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a radio button control are :

- [Select](#) Get/Set a radio button controls's select state
- [Text](#) Get/Set a radio button control's text
- [Style](#) Get/Set a radio button control's style mask
- [Font](#) Get/Set a radio button control's text font
- [Color](#) Get/Set a radio button control attribute's color
- [ID](#) Get a radio button control's window ID
- [Position](#) Get/Set a radio button control's position and size
- [Hide](#) Hide a radio button control

Show Show a radio button control
Visible Get/Set a radio button control's visibility state
Top Make a radio button the topmost control
Bottom Make a radio button the bottommost control
Enable Enable a radio button control
Disable Disable a radio button control
Enabled Get/Set a radio button control's enabled state
Focus Give a radio button control the input focus
Drag Enable/Disable dragging a radio button control
Drop Enable/Disable dropping on a radio button control
IsDefault Make a radio button control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Text control



Creates a static text control.

The events defined for a text control are:

Event Description

Init The text control is being initialized when the dialog is opened

Drop An object has been dropped on the text control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a text control are:

Text Get/Set a text control's text
Style Get/Set a text control's style mask
Font Get/Set a text control's text font
Color Get/Set a text control attribute's color
ID Get a text control's window ID
Position Get/Set a text control's position and size
Hide Hide a text control
Show Show a text control
Visible Get/Set a text control's visibility state
Top Make a text control the topmost control
Bottom Make a text control the bottommost control
Enable Enable a text control
Disable Disable a text control
Enabled Get/Set a text control's enabled state
Focus Give a text control the input focus
Drag Enable/Disable dragging a text control
Drop Enable/Disable dropping on a text control
IsDefault Make a text control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Notebook control



Creates a notebook control.

The events defined for a notebook control are:

Event Description

Select A notebook page has been selected

Size The notebook control has been resized

Delete A notebook page has been deleted

Init The notebook control is being initialized when the dialog is opened

Help The notebook control has been requested to display help

Drop An object has been dropped on the notebook. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a notebook control are :

[Add](#) Add a new notebook page

[Delete](#) Delete one or all notebook pages

[Select](#) Get/Set the current notebook page

[Item](#) Get/Set a notebook page item's value

[Style](#) Get/Set a notebook control's style mask

[Font](#) Get/Set a notebook control's text font

[Color](#) Get/Set a notebook control attribute's color

[ID](#) Get a notebook control's window ID

[Position](#) Get/Set a notebook control's position and size

[Hide](#) Hide a notebook control

[Show](#) Show a notebook control

[Visible](#) Get/Set a notebook control's visibility state

[Top](#) Make a notebook the topmost control

[Bottom](#) Make a notebook the bottommost control

[Enable](#) Enable a notebook control

[Disable](#) Disable a notebook control

[Enabled](#) Get/Set a notebook control's enabled state

[Focus](#) Give a notebook control the input focus

[Drag](#) Enable/Disable dragging a notebook control

[Drop](#) Enable/Disable dropping on a notebook control

[IsDefault](#) Make a notebook control the current default control

[Controls](#) Get a list of all dialog controls

[Classes](#) Get a list of all dialog control classes

Container control



Creates a container control.

The events defined for a container control are:

Event Description

Changed The user has changed the value of a container item field.

The **EventData** function can retrieve the following information about the event :

EventData.1 **Item** that has changed.

EventData.2 Field within **item** that has changed. Possible values are **VALUE** for the item value (i.e. label), or **1** through **n** for detail fields 1 through n.

Enter The user has pressed **Enter** or double-clicked in the container.

The **EventData** function can retrieve the following information about the event :

EventData.1 **Item** that was double-clicked on (or **0** if the pointer was not over an item in the container).

Select The emphasis (*select*, *mark* or *cursor*) of a container item has changed .

The **EventData** function can retrieve the following information about the event :

EventData.1 **Item** whose emphasis has changed.

EventData.2 String indicating how the emphasis has changed:

+SELECT **Item** was *selected*.

-SELECT **Item** was *unselected*.

+MARK **Item** was *marked*.

-MARK **Item** was *unmarked*.

+CURSOR **Item** has the *cursor*.

-CURSOR **Item** has lost the *cursor*.

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button) for the container.

The **EventData** function can retrieve the following information about the event :

EventData.1 **Item** that was clicked on (or **0** if the pointer was not over an item in the container).

Init The container control is being initialized when the dialog is opened

Scroll The container control has scrolled

GetFocus The container control has been given the input focus

LoseFocus The container control has lost the input focus

Drop An object has been dropped on the container control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a container control are :

[Add](#) Add a new container item

[Delete](#) Delete one or all container items

[Select](#) Get/Set the select state of a container item

[Item](#) Get/Set a container item field's value

[View](#) Get/Set the view of a container

[SetStem](#) Set a list of values for a container

[GetStem](#) Get a list of values for a container

[Style](#) Get/Set a container control's style mask

[Font](#) Get/Set a container control's text font

[Color](#) Get/Set a container control attribute's color

[ID](#) Get a container control's window ID

[Position](#) Get/Set a container control's position and size

[Hide](#) Hide a container control

[Show](#) Show a container control

Visible Get/Set a container control's visibility state
Top Make a container the topmost control
Bottom Make a container the bottommost control
Enable Enable a container control
Disable Disable a container control
Enabled Get/Set a container control's enabled state
Focus Give a container control the input focus
Drag Enable/Disable dragging a container control
Drop Enable/Disable dropping on a container control
IsDefault Make a container control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

List box control



Creates a list box control.

The events defined for a list box control are:

Event Description

Enter The user has pressed **Enter** or double-clicked on a list box entry
Select The user has selected a list box entry
Init The list box control is being initialized when the dialog is opened
Scroll The list box control has scrolled horizontally
GetFocus The list box control has been given the input focus
LoseFocus The list box control has lost the input focus
Drop An object has been dropped on the list box control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a list box control are :

Add Add a new list box item
Delete Delete one or all list box items
Select Get/Set the select state of a list box item
Item Get/Set a list box item's value
Style Get/Set a list box control's style mask
Font Get/Set a list box control's text font
Color Get/Set a list box control attribute's color
ID Get a list box control's window ID
Position Get/Set a list box control's position and size
Hide Hide a list box control
Show Show a list box control
Visible Get/Set a list box control's visibility state
Top Make a list box the topmost control
Bottom Make a list box the bottommost control
Enable Enable a list box control
Disable Disable a list box control
Enabled Get/Set a list box control's enabled state
Focus Give a list box control the input focus
Drag Enable/Disable dragging a list box control

Drop Enable/Disable dropping on a list box control
IsDefault Make a list box control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Single line edit control



Creates a single-line entry field control.

The events defined for a single line edit control are:

Event Description

Changed The edit control contents have changed
Init The edit control is being initialized when the dialog is opened
Scroll The edit control has scrolled horizontally
GetFocus The edit control has been given the input focus
LoseFocus The edit control has lost the input focus
Drop An object has been dropped on the edit control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.
Overflow The edit control has overflowed

The DrRexx window functions that can be applied to a single line edit control are:

Select Get/Set the selection bounds of a single line edit control
Range Set the maximum length of a single line edit control
Text Get/Set a single line edit control's text content
Style Get/Set a single line edit control's style mask
Font Get/Set a single line edit control's text font
Color Get/Set a single line edit control attribute's color
ID Get a single line edit control's window ID
Position Get/Set a single line edit control's position and size
Hide Hide a single line edit control
Show Show a single line edit control
Visible Get/Set a single line edit control's visibility state
Top Make a single line edit control the topmost control
Bottom Make a single line edit control the bottommost control
Enable Enable a single line edit control
Disable Disable a single line edit control
Enabled Get/Set a single line edit control's enabled state
Focus Give a single line edit control the input focus
Drag Enable/Disable dragging a single line edit control
Drop Enable/Disable dropping on a single line edit control
IsDefault Make a single line edit control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Multi-line edit control



Creates a multi-line edit control.

The events defined for a multi-line edit control are:

Event Description

Changed The edit control contents have changed

Init The edit control is being initialized when the dialog is opened

GetFocus The edit control has been given the input focus

LoseFocus The edit control has lost the input focus

Drop An object has been dropped on the edit control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a multi-line edit control are:

[Text](#) Get/Set a multi-line edit control's text content

[Style](#) Get/Set a multi-line edit control's style mask

[Font](#) Get/Set a multi-line edit control's text font

[Color](#) Get/Set a multi-line edit control attribute's color

[ID](#) Get a multi-line edit control's window ID

[Position](#) Get/Set a multi-line edit control's position and size

[Hide](#) Hide a multi-line edit control

[Show](#) Show a multi-line edit control

[Visible](#) Get/Set a multi-line edit control's visibility state

[Top](#) Make a multi-line edit control the topmost control

[Bottom](#) Make a multi-line edit control the bottommost control

[Enable](#) Enable a multi-line edit control

[Disable](#) Disable a multi-line edit control

[Enabled](#) Get/Set a multi-line edit control's enabled state

[Focus](#) Give a multi-line edit control the input focus

[Drag](#) Enable/Disable dragging a multi-line edit control

[Drop](#) Enable/Disable dropping on a multi-line edit control

[IsDefault](#) Make a multi-line edit control the current default control

[Controls](#) Get a list of all dialog controls

[Classes](#) Get a list of all dialog control classes

Combo box control



Creates a prompted entry field control (i.e. *combo box*).

The events defined for a combo box control are:

Event Description

Enter The user has pressed **Enter** or double-clicked on a combo box entry

Select The user has selected a combo box entry

Init The combo box control is being initialized when the dialog is opened

Changed The entry field component contents have been changed

ScrollEntry The entry field component has scrolled horizontally

ScrollList The list box component has been scrolled

ShowList The list box component has been displayed (i.e. has *dropped down*)

Drop An object has been dropped on the combo box. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a combo box control are :

[Add](#) Add a new combo box item

[Delete](#) Delete one or all combo box items

[Select](#) Get/Set the select state of a combo box item

[Item](#) Get/Set a combo box item's value

[Style](#) Get/Set a combo box control's style mask

[Font](#) Get/Set a combo box control's text font

[Color](#) Get/Set a combo box control attribute's color

[ID](#) Get a combo box control's window ID

[Position](#) Get/Set a combo box control's position and size

[Hide](#) Hide a combo box control

[Show](#) Show a combo box control

[Visible](#) Get/Set a combo box control's visibility state

[Top](#) Make a combo box the topmost control

[Bottom](#) Make a combo box the bottommost control

[Enable](#) Enable a combo box control

[Disable](#) Disable a combo box control

[Enabled](#) Get/Set a combo box control's enabled state

[Focus](#) Give a combo box control the input focus

[Drag](#) Enable/Disable dragging a combo box control

[Drop](#) Enable/Disable dropping on a combo box control

[IsDefault](#) Make a combo box control the current default control

[Controls](#) Get a list of all dialog controls

[Classes](#) Get a list of all dialog control classes

Spin button control



Creates a spin button control.

The events defined for a spin button control are:

Event Description

Changing The spin field contents have changed

LineUp The up arrow has been pressed or clicked

LineDown The down arrow has been pressed or clicked

GetFocus The spin button control has been given the input focus

LoseFocus The spin button control has lost the input focus

Drop An object has been dropped on the spin button. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

Done The user has released the select button or one of the arrow buttons

Init The spin button control is being initialized when the dialog is opened

The DrRexx window functions that can be applied to a spin button control are :

Select Get/Set the current spin button item
Range Set the range of spin button items
Style Get/Set a spin button control's style mask
Font Get/Set a spin button control's text font
Color Get/Set a spin button control attribute's color
ID Get a spin button control's window ID
Position Get/Set a spin button control's position and size
Hide Hide a spin button control
Show Show a spin button control
Visible Get/Set a spin button control's visibility state
Top Make a spin button the topmost control
Bottom Make a spin button the bottommost control
Enable Enable a spin button control
Disable Disable a spin button control
Enabled Get/Set a spin button control's enabled state
Focus Give a spin button control the input focus
Drag Enable/Disable dragging a spin button control
Drop Enable/Disable dropping on a spin button control
IsDefault Make a spin button control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Value set control



Creates a value set control.

The events defined for a value set control are:

Event Description

Enter The user has pressed **Enter** or double-clicked on a value set item

Select The user has selected a value set item

Init The value set control is being initialized when the dialog is opened

GetFocus The value set control has been given the input focus

LoseFocus The value set control has lost the input focus

Drop An object has been dropped on the value set control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a value set control are :

Select Get/Set the currently selected value set item

Item Get/Set the value of a value set item

Style Get/Set a value set control's style mask

Font Get/Set a value set control's text font

Color Get/Set a value set control attribute's color

ID Get a value set control's window ID

Position Get/Set a value set control's position and size

Hide Hide a value set control

Show Show a value set control

Visible Get/Set a value set control's visibility state

Top Make a value set the topmost control

Bottom Make a value set the bottommost control
Enable Enable a value set control
Disable Disable a value set control
Enabled Get/Set a value set control's enabled state
Focus Give a value set control the input focus
Drag Enable/Disable dragging a value set control
Drop Enable/Disable dropping on a value set control
IsDefault Make a value set control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Vertical scroll bar control



Creates a vertical scroll bar.

The events defined for a vertical scroll bar control are:

Event Description

Init The scroll bar control is being initialized when the dialog is opened

Changed The user has released the scroll bar slider

Changing The user has moved the scroll bar slider

LineUp The user has clicked on the scroll bar up arrow

LineDown The user has clicked on the scroll bar down arrow

PageUp The user has clicked on the area above the scroll bar slider

PageDown The user has clicked on the area below the scroll bar slider

Done The user has finished scrolling (but not using the scroll bar slider)

Drop An object has been dropped on the scroll bar control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a vertical scroll bar control are:

Range Set the range for a vertical scroll bar control

Select Get/Set the current vertical scroll bar control position

Style Get/Set a vertical scroll bar control's style mask

Color Get/Set a vertical scroll bar control attribute's color

ID Get a vertical scroll bar control's window ID

Position Get/Set a vertical scroll bar control's position and size

Hide Hide a vertical scroll bar control

Show Show a vertical scroll bar control

Visible Get/Set a vertical scroll bar control's visibility state

Top Make a vertical scroll bar the topmost control

Bottom Make a vertical scroll bar the bottommost control

Enable Enable a vertical scroll bar control

Disable Disable a vertical scroll bar control

Enabled Get/Set a vertical scroll bar control's enabled state

Focus Give a vertical scroll bar control the input focus

Drag Enable/Disable dragging a vertical scroll bar control

Drop Enable/Disable dropping on a vertical scroll bar control

IsDefault Make a vertical scroll bar control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Horizontal scroll bar control



Creates a horizontal scroll bar.

The events defined for a horizontal scroll bar control are:

Event Description

Init The scroll bar control is being initialized when the dialog is opened

Changed The user has released the scroll bar slider

Changing The user has moved the scroll bar slider

LineLeft The user has clicked on the scroll bar left arrow

LineRight The user has clicked on the scroll bar right arrow

PageLeft The user has clicked on the area to the left of the scroll bar slider

PageRight The user has clicked on the area to the right of the scroll bar slider

Done The user has finished scrolling (but not using the scroll bar slider)

Drop An object has been dropped on the scroll bar control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a horizontal scroll bar control are:

Range Set the range for a horizontal scroll bar control

Select Get/Set the current horizontal scroll bar control position

Style Get/Set a horizontal scroll bar control's style mask

Color Get/Set a horizontal scroll bar control attribute's color

ID Get a horizontal scroll bar control's window ID

Position Get/Set a horizontal scroll bar control's position and size

Hide Hide a horizontal scroll bar control

Show Show a horizontal scroll bar control

Visible Get/Set a horizontal scroll bar control's visibility state

Top Make a horizontal scroll bar the topmost control

Bottom Make a horizontal scroll bar the bottommost control

Enable Enable a horizontal scroll bar control

Disable Disable a horizontal scroll bar control

Enabled Get/Set a horizontal scroll bar control's enabled state

Focus Give a horizontal scroll bar control the input focus

Drag Enable/Disable dragging a horizontal scroll bar control

Drop Enable/Disable dropping on a horizontal scroll bar control

IsDefault Make a horizontal scroll bar control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Slider control



Creates a slider control.

The events defined for a slider control are:

Event Description

Init The slider control is being initialized when the dialog is opened

Changed The slider arm has been released

Changing The slider arm position has changed

GetFocus The slider control has been given the input focus

LoseFocus The slider control has lost the input focus

Drop An object has been dropped on the slider control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a slider control are :

Item Get/Set a slider control tick label or size

Select Get/Set a slider control's slider position

Range Sets the slider control's range of tick values

Style Get/Set a slider control style mask

Font Get/Set a slider control label font

Color Get/Set a slider control attribute's color

ID Get a slider control window ID

Position Get/Set a slider control's position and size

Hide Hide a slider control

Show Show a slider control

Visible Get/Set a slider control's visibility state

Top Make a slider the topmost control

Bottom Make a slider the bottommost control

Enable Enable a slider control

Disable Disable a slider control

Enabled Get/Set a slider control's enabled state

Focus Give a slider control the input focus

Drag Enable/Disable dragging a slider control

Drop Enable/Disable dropping on a slider control

IsDefault Make a slider control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Group box control



Creates a group box control.

Note: This a **DrDialog** [container](#) control.

The events defined for a group box control are:

Event Description

Init The group box is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the group box. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a group box control are :

Text Get/Set a group box control title text
Style Get/Set a group box control style mask
Font Get/Set a group box control label font
Color Get/Set a group box control attribute's color
ID Get a group box control window ID
Position Get/Set a group box control's position and size
Hide Hide a group box control
Show Show a group box control
Visible Get/Set a group box control's visibility state
Top Make a group box the topmost control
Bottom Make a group box the bottommost control
Enable Enable a group box control
Disable Disable a group box control
Enabled Get/Set a group box control's enabled state
Focus Give a group box control the input focus
Drag Enable/Disable dragging a group box control
Drop Enable/Disable dropping on a group box control
IsDefault Make a group box control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Frame control



Creates a frame control.

Note: This is a **DrDialog** container control.

The events defined for a frame control are:

Event Description

Init The frame control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the frame control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a frame control are:

Style Get/Set a frame control style mask
Color Get/Set a frame control attribute's color
ID Get a frame control window ID
Position Get/Set a frame control's position and size
Hide Hide a frame control
Show Show a frame control
Visible Get/Set a frame control's visibility state
Top Make a frame control the topmost control
Bottom Make a frame control the bottommost control
Drag Enable/Disable dragging a frame control
Drop Enable/Disable dropping on a frame control
IsDefault Make a frame control the current default control
Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Rectangle control



Creates a rectangle control.

Note: This is a **DrDialog container** control.

The events defined for a rectangle control are:

Event Description

Init The rectangle control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the rectangle control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a rectangle control are :

Style Get/Set a rectangle control style mask

Color Get/Set a rectangle control attribute's color

ID Get a rectangle control window ID

Position Get/Set a rectangle control's position and size

Hide Hide a rectangle control

Show Show a rectangle control

Visible Get/Set a rectangle control's visibility state

Top Make a rectangle control the topmost control

Bottom Make a rectangle control the bottommost control

Drag Enable/Disable dragging a rectangle control

Drop Enable/Disable dropping on a rectangle control

IsDefault Make a rectangle control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Billboard control



Creates a billboard control.

Note: This control is not a standard OS/2 control.

The events defined for a billboard control are:

Event Description

Init The billboard control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the billboard control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a billboard control are :

Style Get/Set a billboard control style mask
ID Get a billboard control window ID
Position Get/Set a billboard control's position and size
Hide Hide a billboard control
Show Show a billboard control
Visible Get/Set a billboard control's visibility state
Top Make a billboard the topmost control
Bottom Make a billboard the bottommost control
Drag Enable/Disable dragging a billboard control
Drop Enable/Disable dropping on a billboard control
IsDefault Make a billboard control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Canvas control



Creates a canvas control.

Note: This a **DrDialog** container control.

Note: This control is not a standard OS/2 control.

The events defined for a canvas control are:

Event Description

Init The canvas control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the canvas control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a canvas control are :

Style Get/Set a canvas control style mask
Font Get/Set a canvas control label font
ID Get a canvas control window ID
Position Get/Set a canvas control's position and size
Hide Hide a canvas control
Show Show a canvas control
Visible Get/Set a canvas control's visibility state
Top Make a canvas control the topmost control
Bottom Make a canvas control the bottommost control
Drag Enable/Disable dragging a canvas control
Drop Enable/Disable dropping on a canvas control
IsDefault Make a canvas control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Paint control



Creates a paint control.

Note: This is a **DrDialog** container control.

Note: This control is not a standard OS/2 control.

The events defined for a paint control are:

Event Description

Init The paint control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the paint control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a paint control are:

Style Get/Set a paint control style mask

ID Get a paint control window ID

Position Get/Set a paint control's position and size

Hide Hide a paint control

Show Show a paint control

Visible Get/Set a paint control's visibility state

Top Make a paint control the topmost control

Bottom Make a paint control the bottommost control

Drag Enable/Disable dragging a paint control

Drop Enable/Disable dropping on a paint control

IsDefault Make a paint control the current default control

Controls Get a list of all dialog controls

Classes Get a list of all dialog control classes

Bitmap button control



Creates a bitmap button control.

Note: This control is not a standard OS/2 control.

The events defined for a bitmap button control are:

Event Description

Click The user has clicked the bitmap button

Init The bitmap button is being initialized when the dialog is opened

Drop An object has been dropped on the bitmap button. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a bitmap button control are:

Text Get/Set a bitmap button control icon
Style Get/Set a bitmap button control style mask
ID Get a bitmap button control window ID
Position Get/Set a bitmap button control's position and size
Hide Hide a bitmap button control
Show Show a bitmap button control
Visible Get/Set a bitmap button control's visibility state
Top Make a bitmap button the topmost control
Bottom Make a bitmap button the bottommost control
Enable Enable a bitmap button control
Disable Disable a bitmap button control
Enabled Get/Set a bitmap button control's enabled state
Focus Give a bitmap button control the input focus
Drag Enable/Disable dragging a bitmap button control
Drop Enable/Disable dropping on a bitmap button control
IsDefault Make a bitmap button control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Bagbutton control



Creates a bagbutton control.

Note: This control is not a standard OS/2 control.

The events defined for a bagbutton control are:

Event Description

Click The user has clicked the bagbutton

Init The bagbutton is being initialized when the dialog is opened

Drop An object has been dropped on the bagbutton. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a bagbutton control are :

Select Get/Set a state of the bagbutton control

Text Get/Set a bagbutton control label text

Style Get/Set a bagbutton control style mask

Font Get/Set a bagbutton control label font

ID Get a bagbutton control window ID

Position Get/Set a bagbutton control's position and size

Hide Hide a bagbutton control

Show Show a bagbutton control

Visible Get/Set a bagbutton control's visibility state

Top Make a bagbutton the topmost control

Bottom Make a bagbutton the bottommost control

Enable Enable a bagbutton control

Disable Disable a bagbutton control

Enabled Get/Set a bagbutton control's enabled state

Focus Give a bagbutton control the input focus

Drag Enable/Disable dragging a bagbutton control
Drop Enable/Disable dropping on a bagbutton control
IsDefault Make a bagbutton control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Turtle control



Creates a turtle control.

Note: This control is not a standard OS/2 control.

The events defined for a turtle control are:

Event Description

Init The turtle control is being initialized when the dialog is opened
ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)
MouseMove The mouse has moved over the turtle control (see note below)
Button1Down The user has pressed button 1 over the turtle control (see note below)
Button1Up The user has released button 1 (see note below)
Button1DbIClk The user has double-clicked button 1 over the turtle control (see note below)
Button2Down The user has pressed button 2 over the turtle control (see note below)
Button2Up The user has released button 2 (see note below)
Button2DbIClk The user has double-clicked button 2 over the turtle control (see note below)
Drop An object has been dropped on the turtle control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

Note: The mouse events will *only* be generated if the **Container** style is *not* checked in the style dialog for the turtle control. In addition, the **EventData** function can retrieve the following information about the event:

EventData.1 x position of the pointer within the turtle control
EventData.2 y position of the pointer within the turtle control

The DrRexx window functions that can be applied to a turtle control are :

Text Get/Set a turtle control's command text
Style Get/Set a turtle control's style mask
ID Get a turtle control's window ID
Position Get/Set a turtle control's position and size
Hide Hide a turtle control
Show Show a turtle control
Visible Get/Set a turtle control's visibility state
Top Make a turtle control the topmost control
Bottom Make a turtle control the bottommost control
Drag Enable/Disable dragging a turtle control
Drop Enable/Disable dropping on a turtle control
IsDefault Make a turtle control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Bitmap control



Creates a bitmap control.

The events defined for a bitmap control are:

Event Description

Init The bitmap control is being initialized when the dialog is opened

Drop An object has been dropped on the bitmap control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a bitmap control are :

[Style](#) Get/Set a bitmap control style mask

[Color](#) Get/Set a bitmap control attribute's color

[ID](#) Get a bitmap control window ID

[Position](#) Get/Set a bitmap control's position and size

[Hide](#) Hide a bitmap control

[Show](#) Show a bitmap control

[Visible](#) Get/Set a bitmap control's visibility state

[Top](#) Make a bitmap control the topmost control

[Bottom](#) Make a bitmap control the bottommost control

[Enable](#) Enable a bitmap control

[Disable](#) Disable a bitmap control

[Enabled](#) Get/Set a bitmap control's enabled state

[Focus](#) Give a bitmap control the input focus

[Drag](#) Enable/Disable dragging a bitmap control

[Drop](#) Enable/Disable dropping on a bitmap control

[IsDefault](#) Make a bitmap control the current default control

[Controls](#) Get a list of all dialog controls

[Classes](#) Get a list of all dialog control classes

User defined control



Creates a user defined control.

Note: The actual window class of the control can be specified by editing the control's [style](#).

The events defined for a user defined control are:

Event Description

Init The user defined control is being initialized when the dialog is opened

Drop An object has been dropped on the user defined control. Refer to the [Drop event](#) section for details about the information that can be retrieved for the event using the [EventData](#) function.

The DrRexx window functions that can be applied to a user defined control are :

[Text](#) Get/Set a user defined control's text

[Style](#) Get/Set a user defined control style mask

Color Get/Set a user defined control attribute's color
ID Get a user defined control window ID
Position Get/Set a user defined control's position and size
Hide Hide a user defined control
Show Show a user defined control
Visible Get/Set a user defined control's visibility state
Top Make a user defined control the topmost control
Bottom Make a user defined control the bottommost control
Enable Enable a user defined control
Disable Disable a user defined control
Enabled Get/Set a user defined control's enabled state
Focus Give a user defined control the input focus
Drag Enable/Disable dragging a user defined control
Drop Enable/Disable dropping on a user defined control
IsDefault Make a user defined control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Marquee control



Creates a marquee control.

Note: This control is not a standard OS/2 control.

The events defined for a marquee control are:

Event Description

Init The marquee control is being initialized when the dialog is opened

ShowMenu The user has requested a context-sensitive menu (by clicking the right mouse button)

Drop An object has been dropped on the marquee control. Refer to the Drop event section for details about the information that can be retrieved for the event using the EventData function.

The DrRexx window functions that can be applied to a marquee control are :

Text Get/Set a marquee control's text
Style Get/Set a marquee control style mask
ID Get a marquee control window ID
Position Get/Set a marquee control's position and size
Hide Hide a marquee control
Show Show a marquee control
Visible Get/Set a marquee control's visibility state
Top Make a marquee control the topmost control
Bottom Make a marquee control the bottommost control
Drag Enable/Disable dragging a marquee control
Drop Enable/Disable dropping on a marquee control
IsDefault Make a marquee control the current default control
Controls Get a list of all dialog controls
Classes Get a list of all dialog control classes

Drop event

When a **source** object is dropped on a drop enabled DrRexx control, it generates one or more **Drop** events for the dropped on, or **target**, control. A Rexx event or class handler can be written to handle these events and take appropriate action. In order to determine what the appropriate action is, the EventData function can be used to retrieve the following information about the source object :

EventData.1 The data supplied by the source object.

EventData.2 The source object container.

EventData.3 The type of the source object data.

EventData.4 The format of the source object data (i.e. **STRING** or **FILE**).

EventData.5 The operation to be performed on the source object data (i.e. **MOVE**, **COPY** or **LINK**).

EventData.6 The source object ID.

EventData.7 The target object ID.

EventData.8 The location where the source object was dropped.

For some of these items, the form the value takes depends on the type of the source object as follows:

EventData.1 If the format of the source object is **STRING**, the data can be any valid REXX string. If the format is **FILE**, the data should be the fully qualified name of an OS/2 file (e.g. **C:\OS2\BITMAP\OS2LOGO.BMP**).

EventData.2 The source object container depends both on the format and source of the dropped object. If the format is **FILE**, then the container is the path name of the OS/2 file being dropped (e.g. **C:\OS2\BITMAP**). If the format is **STRING** and the originator is another DrRexx control (not necessarily in the same application), the container is a string of the form: **dialog .control=class**, where **dialog** is the name of the DrRexx dialog containing the control that was dragged, **control** is the name of the control, and **class** is the class of the control (e.g. **orders.customers=CONTAINER**). In the case of a non -DrRexx source object of format **STRING**, the form of the container field is not specified.

EventData.3 The type of the source object data is the intersection of the source and target types (i.e. source types not understood by the target are not included in the value). If the target accepts **ANY** type, then all of the source object types are included in the value.

EventData.4 The format of the source object data will either be **STRING** or **FILE**.

EventData.5 The operation to be performed on the source object data will either be **MOVE**, **COPY** or **LINK**.

EventData.6 The source object ID gives more information about which part of the source object was dragged. Since most DrRexx controls only have a single part, this value is normally just the source control ID (e.g. **101**). The exceptions are:

Container The source ID is the container item that was dragged.

Value set The source ID is the value set item that was dragged. This is a single number of the form: **row + 65536 * column**.

List box The source ID is the number of the list box item that was dragged .

Note: In the case where the source object is not a DrRexx control, the meaning of the source ID is not defined, other than that it is a number.

EventData.7 The target object ID gives more information about which part of the target control the source object was dropped on. Since most controls only have a single part, this value is normally just the target control ID (e.g . **101**). The exceptions are:

Container The target ID is the container item the source object was dropped on (**0** if the object was dropped on empty space within the container).

Value set The target ID is the value set item the source object was dropped on. This is a single number of the form: **row + 65536 * column**.

EventData.8 The location where the source object was dropped is returned as a string of the form: **x1 y1 x2 y2**. **X1** is the horizontal, and **y1** is the vertical distance of the pointer from the lower left hand corner of the dialog the target control is contained in at the time the item was dropped. **X2** is the horizontal, and **y2** is the vertical distance of the lower left hand corner of the drag image (i.e. icon) from the lower left hand corner of the dialog the target control is contained in.

Note: When a source object is dropped on a DrRexx control, it may generate more than one **Drop** event. For example, selecting several items in a drag enabled DrRexx list box and then dropping it onto a drop enabled control will generate one **Drop** event for each selected item in the list box. This is indicated visually during the drag operation by the appearance of several bitmaps (up to a maximum of four) staggered behind the pointer, instead of just a single bitmap.

DrDialog specific controls

Most of the controls available in the controls window palette are standard OS /2 PM controls (e.g. pushbuttons, list boxes, containers, etc.). You should refer to the *OS/2 Presentation Manager Programming Guide* and the *OS/2 Presentation Manager Programming Reference Volume III* for detailed information about the purpose and programming interface for these controls.

However, there are also a number of controls that are unique to **DrDialog** :

- o Billboard controls
- o Canvas controls
- o Paint controls
- o Bitmap button controls
- o Bagbutton controls
- o Turtle controls
- o Marquee controls

If you use any of these controls in your dialogs, you will also need to have **DRDIALOG.DLL** in the **LIBPATH** when your application runs.

In addition, prior to loading any dialogs containing **DrDialog** specific controls, your program must call **UseDrDialog** to initialize the DrDialog control window classes. This function has no arguments, and returns 1 if the DrDialog control window classes were successfully initialized, and 0 otherwise. You must link with the **DRDIALOG.LIB** file supplied with DrDialog in order to include **UseDrDialog** in your program.

The **DRDIALOG.H** and **DRDIALOG.OH** files contain the function prototype for **UseDrDialog** for the **C (C++)** and **Oberon** languages respectively.

These files also contain definitions for the various style bits supported by the DrDialog specific controls. You may need to use these if your application changes the style of any DrDialog specific controls at run-time.

Billboard controls

A billboard control is a bitmap used to dress up an otherwise boring dialog . It has a simple interface that requires no application programming, and a selection of styles that supports a wide variety of presentation techniques.

The bitmap displayed in a **billboard** control is specified using the control's *window text*. The format of the text is either: [**DLLname**:]# **resourceId** (e.g. 'BITMAP:#23'), or **filename.BMP** (e.g. **C:\OS2\BITMAP\OS2LOGO.BMP**).

In the first form, the optional **DLLname** specifies the name of the DLL the bitmap can be found in. If omitted, the bitmap is assumed to be part of the application's EXE file. The **resourceId** specifies the resource number of the bitmap within the DLL or EXE file.

In the second form, the bitmap is assumed to be stored in a standard **.BMP** file (e.g. a **.BMP** file created by **IconEdit**, the OS/2 icon editor). Note that the last four characters of the file name must be **.BMP**.

The window text string can be specified using the text window or, for bitmaps stored in a DLL, using the billboard control's style dialog, displayed whenever button 2 is pressed while the pointer is over a billboard control.

The billboard style dialog has a text entry field for specifying the name of the DLL to use (the default DLL is the **BITMAP** DLL supplied with **DrDialog**). Clicking the **Load** button opens the DLL and displays all of its bitmap resources in the array of billboard controls located below it. The scroll bar located next to the billboard controls can be used to scroll through the DLL's bitmap resources. Clicking any of the billboard controls in the style dialog will cause the associated billboard control in the edit dialog to display the same bitmap. The billboard control's window text will also be updated accordingly.

The style dialog also allows various display characteristics of a billboard control to be set:

Container: If selected, the billboard is a **DrDialog container**. It will appear below all non-container controls and can *contain* other controls (including other container controls). If not selected, the billboard control is a normal, non-container control.

Mode:

Scale The bitmap is scaled to completely fill the control.

Replicate The bitmap is displayed at its normal size, but is replicated (starting from the lower left corner) until the control is completely filled.

Center A single, normal sized copy of the bitmap is displayed in the center of the control.

Speed:

Stopped The bitmap is displayed in a stationary position.

Slow The bitmap (or replicated copies of the bitmap) are scrolled slowly.

Medium The bitmap is scrolled twice as fast as the *slow* speed.

Fast The bitmap is scrolled at its fastest speed.

Direction:

Left The bitmap scrolls toward the left of the control.

Right The bitmap scrolls toward the right of the control.

Top The bitmap scrolls toward the top of the control.

Bottom The bitmap scrolls toward the bottom of the control.

In addition, the standard PM control styles (**Visible**, **Disabled**, **Tab stop**, and **Group**) can also be specified in the billboard style dialog.

Canvas controls

A **canvas** control is similar to a group box. It is a **DrDialog container** control used to visually group other controls. It has a wide variety of visual styles which can be specified via its style bits. It also has an optional text label which can be modified using the text window.

The visual appearance of a canvas control is specified using the canvas style dialog, displayed when button 2 is clicked while the pointer is in a canvas control . With the canvas style dialog you can select:

Canvas styles/border width:

Indented The edge of the control has an *indented* look.

Raised The edge of the control has a *raised* look.

Ridged The edge of the control has either a *ridged* or *grooved* appearance, depending on whether **Raised** or **Indented** is selected. This effect may be turned on or off.

Thickness The thickness of the control edge can be selected using the spin button. You can turn off the edge entirely by selecting a thickness of 0.

Horizontal text alignment:

Left The text label is left aligned in the control.

Center The text label is horizontally centered in the control.

Right The text label is right aligned in the control.

Vertical text alignment:

Top The text label is top aligned in the control.

Center The text label is vertically centered in the control.

Bottom The text label is bottom aligned in the control.

Text styles:

Separator A separator will be drawn to separate the text label from the rest of the control. This effect can be turned on or off. If the text label is not specified, or is both horizontally and vertically centered, the separator is not drawn, even if requested.

Canvas patterns: Any of 16 different background patterns for the control can be specified by clicking on the desired pattern.

In addition, the standard PM control styles (**Visible**, **Disabled**, **Tab stop**, and **Group**) can also be specified in the canvas style dialog.

Paint controls

A **paint** control provides background color for a dialog. It is a **DrDialog** container control that mixes two colors, a *paint color* and a *mix color*, in various proportions to achieve a desired shade.

The paint control's color is specified using the paint control's style dialog, displayed when button 2 of the pointer is clicked in a paint control. Click on the desired paint and mix colors, and use the *mix value* spin button to adjust their relative proportions.

Bitmap button controls

A **bitmap button** control is a push button that uses a bitmap instead of text to describe its function. The bitmap displayed is specified by the control's *window text*. The format of the text is either: **[DLLname:]# resourceId** (e.g. 'BITMAP:#23'), or **filename.BMP** (e.g. **C :\OS2\BITMAP\OS2LOGO.BMP**).

In the first form, the optional **DLLname** specifies the name of the DLL the bitmap can be found in. If omitted, the bitmap is assumed to be part of the application's EXE file. The **resourceId** specifies the resource number of the bitmap within the DLL or EXE file.

In the second form, the bitmap is assumed to be stored in a standard **.BMP** file (e.g. a **.BMP** file created by **IconEdit**, the OS/2 icon editor). Note that the last four characters of the file name must be **.BMP**.

The window text string can be specified using the text window or, for bitmaps stored in a DLL, using the bitmap button control's style dialog, displayed whenever button 2 is pressed while the pointer is over a bitmap button control.

The bitmap button style dialog has a text entry field for specifying the name of the DLL to use (the default DLL is the **BITMAP** DLL supplied with **DrDialog**). Clicking the **Load** button opens the DLL and displays all of its bitmap resources in the array of bitmap button controls located below it. The scroll bar located next to the bitmap buttons can be used to scroll through the DLL's bitmap resources. Clicking any of the bitmap button in the style dialog will cause the associated bitmap button control in the edit dialog to display the same bitmap. The bitmap button control's window text will also be updated accordingly.

In addition, the standard PM control styles (**Visible**, **Disabled**, **Tab stop**, and **Group**) can also be specified in the bitmap button style dialog.

Bagbutton controls

A **bagbutton button** control is a push button that allows other controls to be stacked on top of it as if they were held in a *bag*. The stacked controls behave visually as if they were part of the bagbutton control. That is, when the button is clicked, the controls *held* in the bagbutton will shift as part of the 3-D effect of the button press. The result is a composite button that behaves as if it were a single, seamless control.

The bagbutton is also a DrDialog container control. Any controls placed on top of it will move with it when the bagbutton is moved.

The normal appearance of a bagbutton includes:

- oOptional text
- oOptional **LED**. The LED will *light* whenever the bagbutton is in the checked/ pressed state.

The text string can be specified using the text window.

The visual appearance and behavior of a bagbutton control is specified using the bagbutton style dialog, displayed when button 2 is clicked while the pointer is in a bagbutton control. With the bagbutton style dialog you can select:

Button style:

Pushbutton The bagbutton behaves like a pushbutton.

Clicker The bagbutton behaves like a *clicker*. A clicker is a button that sends its owner window a continuous stream of **WM_COMMAND** messages while the bagbutton is pressed.

Check box The bagbutton behaves like a check box.

Radio button The bag button behaves like a radio button.

Clicker speed (i.e. rate at which WM_COMMAND messages are generated if the bagbutton has the **clicker** style):

- oSlow
- oMedium
- oFast
- oFastest

LED style:

- oNone (i.e. no LED)
- oRound
- oRectangular

LED color:

- oRed
- oGreen
- oYellow
- oCyan

Horizontal text/LED alignment:

- Left The text label and/or LED is left aligned in the control.
- Center The text label and/or LED is horizontally centered in the control.
- Right The text label and/or LED is right aligned in the control.

Vertical text/LED alignment:

- Top The text label and/or LED is top aligned in the control.
- Center The text label and/or LED is vertically centered in the control.
- Bottom The text label and/or LED is bottom aligned in the control.

Sizes:

- Border width Thickness of the 3-D border around the bagbutton (1 to 4).
- Window shift Amount by which other controls within the bagbutton are shifted when the button is pressed and released (0 to 3).

Note: Only controls which are *transparent* to the mouse should be placed on top of a bagbutton control. If the control is not pointer transparent, it will interfere with the correct operation of the bagbutton. Valid transparent controls are:

- oBillboard controls
- oTurtle controls
- oMarquee controls
- oCanvas controls
- oPaint controls

Below are several examples of composite bagbutton controls along with a description of each button:



1. Bagbutton control and Marquee control
2. Bagbutton control with text and Billboard control
3. Bagbutton control with text and Turtle control
4. Bagbutton control with LED and Billboard control

Turtle controls

A **turtle** control is named after the turtle made famous by the **Logo** language . In Logo, a robotic *turtle* with a pen was controlled by means of simple commands to draw figures on a piece of paper.

DrDialog turtle controls behave in much the same manner. The simple commands understood by a turtle control are specified using its window text.

The window text can be specified using the text window or, more conveniently, using the turtle control's style dialog displayed when button 2 is pressed while the pointer is over a turtle control.

The turtle style dialog has a text entry field for specifying the name of a text file containing one or more turtle control strings, one per line (the default file name is for the **DRDIALOG.TUR** file supplied with **DrDialog**). Clicking the **Load** button opens the file and displays each control string as it would appear in a turtle control in the array of turtle controls located below it. The scroll bar located next to the turtle controls can be used to scroll through the file's turtle control strings. Clicking any of the turtle controls in the style dialog will cause the associated turtle control in the edit dialog to display the same turtle control string. The turtle control's window text will also be updated accordingly.

The turtle style dialog also allows various display characteristics of a turtle control to be set:

Container: If selected, the turtle control is a **DrDialog container**. It will appear below all non-container controls and can *contain* other controls (including other container controls). If not selected, the turtle control is a normal, non-container control.

Drawing color: The color the turtle control draws with can be specified by clicking on the desired color.

Background color: The background color for the turtle control can be specified by clicking on the desired color.

In addition, the standard PM control styles (**Visible**, **Disabled**, **Tab stop**, and **Group**) can also be specified in the turtle style dialog.

Turtle control commands

A turtle control interprets a simple command language. Each command consists of a single letter followed (in most cases) by an optional integer modifier . For example, the string '<urdl>' draws a filled box by giving the turtle the following commands:

< Put the drawing pen down on the paper (i.e. screen)
u Move up
r Move right
d Move down
l Move left
> Pick the drawing pen up

The following commands are understood by a turtle control:

f[n] Set the fractional movement divisor to **n**. The initial value and default is 100 (i.e. fractions are *percentages*).
z[n] Set the default fractional movement size to **n**. The initial value is 10, and the default value is no change. A fractional movement is a movement expressed as a fraction of the total control width or height (e.g. movement = $(z * width) / f$).
Z[n] Set the default absolute movement size to **n**. The initial value is 1 and the default value is no change. An absolute movement is a movement expressed in pels.
u[n] Move the pen up **n** fractional units. The default value is the current value of **z**.
d[n] Move the pen down **n** fractional units. The default value is the current value of **z**.
l Move the pen left **n** fractional units. The default value is the current value of **z**.
r Move the pen right **n** fractional units. The default value is the current value of **z**.
U Move the pen up **n** absolute units. The default value is the current value of **Z**.
D Move the pen down **n** absolute units. The default value is the current value of **Z**.
L Move the pen left **n** absolute units. The default value is the current value of **Z**.
R Move the pen right **n** absolute units. The default value is the current value of **Z**.
x[m[,n]] Move the pen to the position whose fractional x coordinate is given by **m**, and whose fractional y coordinate is given by **n**. The default value for **m** is 0. If **n** is not specified, the current y coordinate of the pen is not changed.
y[m[,n]] Move the pen to the position whose fractional y coordinate is given by **m**, and whose fractional x coordinate is given by **n**. The default value for **m** is 0. If **n** is not specified, the current x coordinate of the pen is not changed.
X[m[,n]] Move the pen to the position whose absolute x coordinate is given by **m**, and whose absolute y coordinate is given by **n**. The default value for **m** is 0. If **n** is not specified, the current y coordinate of the pen is not changed.
Y[m[,n]] Move the pen to the position whose absolute y coordinate is given by **m**, and whose absolute x coordinate is given by **n**. The default value for **m** is 0. If **n** is not specified, the current x coordinate of the pen is not changed.
c[n] Set the drawing color of the turtle to **n**. This should be a number in the range from 0 to 15. The initial and default values are 7 (black).
b[n] Set the background color of the turtle to **n**. This should be a number in the range from 0 to 15. The initial and default values are 15 (gray).
p[n] Set the fill pattern for the turtle to **n**. This should be a number in the range from 0 to 19. The initial and default values are 0 (solid fill).
< Put the drawing *pen* down (i.e. on the screen). Figures can only be drawn when the pen is down. The pen is initially up.
> Pick the drawing *pen* up (i.e. off the screen). No figures are drawn when the pen is up. The pen is initially up.
h[n] Set the current heading to **n** degrees. This should be a number in the range from 0 to 360. The initial and default values are 0 (i.e. up).
a[n] Set the default heading increment to **n** degrees. The initial value is 90, and the default is no change.
t[n] Increase the current heading by **n** degrees counter-clockwise. The default value is the current value of **a**.

T[n] Increase the current heading by **n** degrees clockwise. The default value is the current value of **a**.
m[n] Move the pen **n** fractional units in the current heading. The default is the current value of **z**.
M[n] Move the pen **n** absolute units in the current heading. The default is the current value of **Z**.
'..' Display the text between the two quote marks at the current pen position using the current drawing color. The starting pen position is at the lower left hand corner of the text displayed.
'..' Display the text between the two quote marks at the current pen position using the current drawing color. The starting pen position is at the lower left hand corner of the text displayed.
[...]**n** Do the commands in square brackets **n** times. The default value of **n** is 1.
{...}**c** Assign the commands in curly braces the name **c** (a single character, which must be specified).
=**c** Interpret the series of commands previously given the name **c** (a single character, which must be specified).
(...) Evaluate the series of pen movement commands enclosed in parentheses without actually moving the pen. When the closing parenthesis is encountered, move the pen to the final position computed.
|..| The characters between the two vertical bars are the fully qualified name of an OS/2 PM metafile. Alternatively, if no characters occur between the vertical bars, the metafile is assumed to be stored in the system clipboard .

If this command is used it must be the very first (or only) command in the turtle control string. If no other commands follow it, then the metafile specified between the vertical bars is displayed in the turtle control. If other turtle commands do follow the second vertical bar, then they are used to draw the contents of the turtle control and the result is stored in the specified metafile (i.e. an OS/2 file or the system clipboard).

Note: Characters, including blanks, which are not recognized as valid turtle commands are ignored.

For some examples of turtle command strings, refer to the contents of the **DRDIALOG.TUR** file that comes with **DrDialog**.

Marquee controls

A **marquee** control is similar to a text control, but uses vector rather than bitmap fonts. As a result, any size text can be displayed simply by sizing the marquee control appropriately. The text for a marquee control can be entered using the text window.

The visual appearance of a marquee control is controlled by its style bits and can be specified using the marquee style dialog displayed when button 2 is clicked while the pointer is in a marquee control. With the marquee style dialog you can select:

Font:

- oHelvetica
- oTimes Roman
- oCourier
- oSymbol Set

Style:

Bold Displays the text using a **bold** version of the selected font.

Italic Displays the text using an *italic* version of the selected font.

Embossed Displays the text so that it appears to be **embossed**. If not specified, the text will have an **engraved** appearance. The embossed/engraved effect only applies if the text and background color are the same.

Speed:

Stopped The text is displayed stationary. The height of the control determines the scaling of the text in both the horizontal and vertical directions.

Slow The text is slowly scrolled from one side of the control to the other.

Medium The text is scrolled somewhat faster than the **slow** rate.

Fast The text is scrolled at its fastest rate.

Direction:

Right->Left The text is scrolled from right to left. This has no effect if **stopped** is specified.

Left->Right The text is scrolled from left to right. This has no effect if **stopped** is specified.

Text color: The text color can be specified by clicking on the desired color.

Background color: The background color for the control can be specified by clicking on the desired color.

In addition, the standard PM control styles (**Visible**, **Disabled**, **Tab stop**, and **Group**) can also be specified in the marquee style dialog.

Some examples of marquee controls are as follows:



DrsAide

DrsAide is an extension to **DrDialog** that allows new tools to be written using **DrRexx** and *seamlessly* integrated into the DrDialog programming environment.

This section describes:

- o the extension mechanism to DrDialog provided by DrsAide
- o the default DrsAide tool provided with DrDialog
- o a suite of tools provided with DrDialog and written in DrRexx using the DrsAide interface
- o how to go about writing your own DrsAide tools

The DrsAide extension mechanism

There are two components to the DrsAide extension mechanism:

- o The DrsAide tool
- o The DrDialog function

The DrsAide tool

The DrDialog **Tools** window and the **Tools** submenu of the DrDialog menu bar and pop-up menu contain the



button. Clicking this button causes one of the following two actions to occur:

- o If the DrsAide tool is not running, DrDialog starts it.
- o If the DrsAide tool is running, DrDialog brings all of its associated windows to the foreground.

DrDialog starts the DrsAide tool running by executing the following command :

```
drdialogPath\DRREXX drdialogPath\DRSAIDE.RES -HdrdialogHandle
```

where **drdialogPath** is the fully qualified path from which the DrDialog . EXE file was invoked, and **-HdrdialogHandle** is the handle a DrRexx application needs to start a conversation with DrDialog using the **Init** subcommand of the DrDialog function.

For example, the following command might be used to invoke the DrsAide tool :

```
D:\DRDIALOG\DRREXX D:\DRDIALOG\DRSAIDE.RES -H1482695048
```

Note: The **DrsAide.RES** file *must* be in the same directory as the **DrDialog.EXE** file.

From the above, it can be seen that the DrsAide tool must be a DrRexx application, and must be available in **.RES** file format (i.e. *not* **.EXE** file format). Other than that, the **DrsAide.RES** file can be any valid DrRexx application. If desired, it can use the **-H** command line argument passed to it to establish a conversation with DrDialog using the DrDialog function. However, it is not required to do so.

DrDialog tracks whether DrsAide.RES is running or not. If DrsAide .RES terminates, DrDialog takes note of the fact and will automatically start a

new copy the next time the  button is clicked.

DrDialog also treats the execution status of DrsAide.RES as a user preference item, and will automatically start DrsAide.RES when DrDialog is invoked if DrsAide.RES was running when DrDialog last terminated.

If DrsAide.RES is already running when the  button is clicked, DrDialog will simply bring all dialogs registered with DrDialog with the application name '**DrsAide**' to the foreground (a dialog is registered with DrDialog using the **Owner** subcommand of the DrDialog function).

Note: It is normally not necessary to write the DrsAide.RES tool yourself. DrDialog is distributed with an extensible DrsAide.RES tool which will satisfy most requirements. This section simply documents the mechanism used in case it ever becomes necessary for you to write or modify the DrsAide.RES tool.

The default DrsAide tool

DrDialog is distributed with a default DrsAide.RES tool already defined . This tool is written in such a way as to allow new tools to be easily added to the system.

The default DrsAide tool can actually be invoked in one of three ways:

- oFrom DrDialog, by clicking on the  button.
- oFrom the Workplace Shell, by dropping a new or existing tool's .RES file icon onto the DrsAide icon.
- oFrom the Workplace Shell, by double-clicking the DrsAide icon.

Invoking the default DrsAide tool from DrDialog

When invoked from DrDialog, DrsAide.RES displays a tool bar where each icon button represents a tool that can be invoked via DrsAide. The tool bar window can be moved to any desired screen position, and will remember its location each time the tool bar window is closed. This saved location will be used to position the icon bar the next time DrsAide is invoked from DrDialog.

The position of the tool bar icon buttons can also be changed by dragging an icon button from its current location and dropping it on its new location. The new arrangement will be saved when the tool bar window is closed, and will be used to define the order of the icon buttons the next time DrsAide is invoked from DrDialog.

Clicking an icon button in the DrsAide tool bar will do one of two things, depending upon the status of the corresponding tool:

oIf the tool is not running, DrsAide starts it.

oIf the tool is running, DrsAide brings all of its associated windows to the foreground.

Note that this is very similar to the actions taken by DrDialog with regard to the DrsAide tool.

If the tool is not running, DrsAide starts it by executing a command of the form:

```
'START drdialogPath\DRREXX tool.RES -HdrdialogHandle iniFile'
```

where:

odrdialogPath is the fully qualified path from which the DrDialog .EXE file was invoked

otool.RES is the name of the DrRexx tool to be executed

o-HdrdialogHandle is the handle a DrRexx application needs to start a conversation with DrDialog using the **Init** subcommand of the DrDialog function.

oiniFile is the fully qualified name of the DrsAide.INI file that the tool can use to save or restore information.

For example, the following command might be used to invoke the **CLOCK.RES** sample program from DrsAide:

```
START D:\DRDIALOG\DRREXX D:\DRDIALOG\SAMPLE\CLOCK.RES  
-H1482695048 D:\DRDIALOG\DRSAIDE.INI
```

Note: The **tool.RES** file can be in any directory (unlike the **DrsAide .RES** file, which must be in the same directory as the **DrDialog.EXE** file).

From the above, it can be seen that, like DrsAide, any tool invoked from DrsAide must be a DrRexx application, and must be available in **.RES** file format (i .e. **not .EXE** file format). Other than that, a tool.RES file can be any valid DrRexx application. If desired, it can use the **-H** command line argument passed to it to establish a conversation with DrDialog using the DrDialog function. However, it is not required to do so. It can also use the **iniFile** passed to it to store long term tool specific information (e.g. the saved position of the tool window). More information on this is available in

the section on writing a DrsAide tool.RES file.

If the tool is already running when the tool's icon button is clicked, DrsAide will simply bring all dialogs registered with DrDialog using the tool's fully qualified **.RES** file name to the foreground (a dialog is registered with DrDialog using the **Owner** subcommand of the DrDialog function).

Note: Unlike DrDialog, DrsAide does not actually track the execution state of each tool it starts. DrsAide only tracks the dialogs registered with DrDialog. If a tool registers one or more dialogs with DrDialog, DrsAide will bring those dialogs to the foreground when the tool's icon button is clicked. If it has not registered any dialogs, DrsAide will simply launch another copy of the tool, even if the previous copy is still running. It is the tool writer's responsibility to register any necessary dialogs with DrDialog if they do not want multiple copies of the tool to be running simultaneously.

Invoking the default DrsAide tool from the Workplace Shell

Invoking DrsAide from the Workplace Shell allows you to add or delete tools from the tool bar that DrsAide displays when invoked from within DrDialog.

To add or delete a single tool, simply drag its **.RES** file icon and drop it on the **DrsAide** icon in the **DrDialog** folder. If the tool is not already known to DrsAide, it will add the tool to the tool bar. If the tool is already in the tool bar, DrsAide will prompt to see if you wish to delete the tool or cancel the request. If you specify delete, the tool will be removed from the DrsAide tool bar. If you specify cancel, no action will be taken.

When either adding or deleting a tool, no change to the DrsAide tool bar will occur until the next time DrsAide is invoked from within DrDialog.

In order to display an appropriate bitmap button in the DrsAide tool bar, each tool added to DrsAide must have a corresponding **.BMP** file. The **.BMP** file must reside in the same directory and have the same name (with a **.BMP** extension) as the **.RES** file it corresponds to. If no **.BMP** file with this name is found, DrsAide will not add the specified **.RES** file to its tool bar. The size of the bitmap contained in the **.BMP** file should also be 40 x 40 in order to be consistent with other DrsAide tools.

You can also directly edit the set of installed DrsAide tools by double-clicking the DrsAide icon in the **DrDialog** folder. A dialog containing a list of tools currently installed will appear. You can then:

- oDelete a tool by selecting its entry and clicking the **Delete** button.
- oAdd a tool by typing its **.RES** file name into the appropriate entry field and clicking the **Add** button. If the specified **.RES** file exists and is not already installed, it will be added as a new DrsAide tool immediately after the currently selected entry in the list of installed tools. If the **.BMP** file entry field is not empty, the specified file will be used as the name of the **.BMP** file to use for the tool's icon button. If it is empty, the **.RES** file's name with a **.BMP** extension will be used as the name of the **.BMP** file.
- oChange the name of the **.BMP** file to use for a tool's icon button by clicking on its **.BMP** file name while holding the **Alt** key. This will allow you to edit the name of the **.BMP** file. When you are done editing, click on any other field to indicate you are done.

You can also copy a currently installed tool's information into the entry fields by double-clicking its entry in the list. This is useful when you wish to move a tool from one location in the DrsAide tool bar to another. Simply double-click the entry to move (thus copying its information into the entry fields), then click the **Delete** button. Then select the list entry preceding the new location you wish to use and click the **Add** button.

When you are finished making changes, click the **Save** button to permanently update the installed tool's information. No permanent changes are made until the **Save** button is clicked.

DrDialog function

```
result = DrDialog( subcommand [, arguments] )
```

The **DrDialog** function is the mechanism by which DrDialog tools written using DrRexx communicate with DrDialog. The **subcommand** argument specifies which of 27 different subcommands are to be executed by DrDialog in response to the **DrDialog** function request. The additional arguments, if any, following the subcommand depend on the particular subcommand specified.

The available subcommands are as follows:

- Init Initialize conversation with DrDialog
- Owner Set window owner
- FOCUS Give focus to .RES files windows
- GETres Get current .RES contents
- SETres Set current .RES contents
- Filename Get current .RES file name
- Modified Get/Set modified flag
- DIALOGS Get all dialogs
- CONTROLS Get all controls for current dialog
- EVENTS Get events for a control type
- GLOBALS Get global procedure names
- GLOBAL Get/Set global procedure code
- NEWDialog Create new dialog
- NEWControl Create new control
- DROPDialog Delete dialog
- DROPControl Delete control
- DIALOG Get/Select current dialog
- CONTROL Get/Select current control
- SELEct Get/Select current selected controls
- NAME Get/Set control name
- Text Get/Set control text
- Hint Get/Set control hint text
- Position Get/Set control size/position
- STyle Get/Set control style
- Font Get/Set control font
- COLor Get/Set control color
- EVENT Get/Set control event handler
- CLass Get/Set control class handler

Note: In the above list of subcommands, capital letters indicate characters required to identify the subcommand, while lower case letters indicate optional characters that can be specified if desired.

DrDialog 'Init' subcommand

CALL DrDialog 'Init', handle

Initializes a conversation with the DrDialog session identified by **handle** . **Handle** must be the string of characters starting with '-H' that is passed as the first command line argument when a DrsAide tool is invoked.

This **DrDialog** subcommand must be the first one issued by a tool wishing to interact with DrDialog. Failure to do so, or specifying an invalid **handle**, will result in all subsequent **DrDialog** function calls generating an error.

This function need only be issued once. All subsequent **DrDialog** functions will automatically direct their requests to the DrDialog session specified by **handle** . If a single tool needs to talk to more than one DrDialog session, an **Init** subcommand must be issued every time a different DrDialog session is to be addressed.

DrDialog 'Owner' subcommand

CALL DrDialog 'Owner', dialog [, toolName]

Specifies that DrDialog is to **own** the tool dialog specified by **dialog**, and that the dialog is to be registered as belonging to the tool specified by **toolName** . If **toolName** is not specified, it defaults to the fully qualified name of the tool's **.RES** file.

Owning a dialog allows DrDialog to display the dialog as part of its collection of tool windows and prevents the dialog from disappearing behind the DrDialog background window.

DrDialog also divides the windows it owns into one or more collections organized by tool name. The **DrDialog Focus** subcommand can be used to bring all dialogs belonging to a particular tool back to the foreground.

Note: The standard DrRexx **Owner** function can be used to take ownership of the dialog away from DrDialog at some later point if desired.

DrDialog 'Focus' subcommand

```
rc = DrDialog( 'FOCUS', toolName )
```

Requests that DrDialog bring all dialogs registered as belonging to **toolName** to the foreground. It returns 0 if no dialogs belonging to **toolName** are found; and 1 otherwise.

Dialogs are registered with DrDialog using the **DrDialog Owner** subcommand.

DrDialog 'GetRES' subcommand

```
resData = DrDialog( 'GETres' )
```

Requests that DrDialog capture and return the contents of the current set of dialogs being edited in **.RES** file format. That is, it behaves as if the user had requested that the current edit session be saved to a file, and returns what DrDialog would have written to the **.RES** file as the result.

The format of the **resData** returned is that of a standard OS/2 resource file with the addition of a few new resource types unique to DrDialog/DrRexx.

DrDialog 'SetRES' subcommand

```
rc = DrDialog( 'SETres', resData )
```

Requests that DrDialog discard all current dialogs being edited and load the set of dialogs specified by **resData**. Returns 1 if successful, and 0 otherwise .

ResData must be structurally equivalent to the contents of a valid resource (i.e. **.RES**) file created by DrDialog.

Note: When DrDialog receives this request, it will discard the current set of dialogs being edited, even if changes have been made. It is the tool writer's responsibility to prevent the user from losing data. The **DrDialog Modified** subcommand can be used to check if changes have been made in the current DrDialog edit session.

DrDialog 'Filename' subcommand

```
oldFilename = DrDialog( 'Filename' [, newFilename] )
```

Returns the name of the current **.RES** file being edited by DrDialog, or the null string if no file name has been specified. If **newFilename** is specified, DrDialog saves it as the new name of the file being edited.

DrDialog 'Modified' subcommand

```
oldModified = DrDialog( 'Modified', newModified )
```

Returns 1 if changes have been made to the set of dialogs currently being edited by DrDialog, and 0 if no changes have been made. If **newModified** is specified, it sets the modified state of DrDialog to *modified* if **newModified** is not zero, and to *unmodified* if **newModified** is 0.

DrDialog 'Dialogs' subcommand

```
dialogs = DrDialog( 'DIALOGS' )
```

Returns as a blank delimited string the ID numbers of all dialogs currently being edited by DrDialog (e.g. '**100 200 300**').

DrDialog 'Controls' subcommand

```
controls = DrDialog( 'CONTROLS' )
```

Returns as a blank delimited string the ID numbers of all controls in the current dialog being edited by DrDialog (e.g. '**100 101 102 111**'). The ID number of the dialog frame is always the first ID returned.

DrDialog 'Events' subcommand

```
events = DrDialog( 'EVENTS', class )
```

Returns as a blank delimited string the names of all events defined for the control type specified by **class**. For example, if **class** is **PUSHBUTTON**, the result is **'Click Init'**.

DrDialog 'Globals' subcommand

```
dialogs = DrDialog( 'DIALOGS' )
```

Returns as a blank delimited string the names of all global procedures currently defined by the DrDialog session (e.g. **'Init AddDigit FormatData'**).

DrDialog 'Global' subcommand

```
oldCode = DrDialog( 'GLOBAL', name [, newCode] )
```

Returns the definition of the global procedure specified by **name** in the current DrDialog session. If **newCode** is specified, it replaces the current definition of **name**, unless **newCode** is the null string, in which case **name** is deleted.

If **name** is not defined, the null string is returned. If **name** is not defined and **newCode** is specified and not the null string, **name** is added to the list of global procedures.

DrDialog 'NewDialog' subcommand

```
actualId = DrDialog( 'NEWDialog' [, newId] )
```

Requests DrDialog to create a new dialog and select it for editing. If **newId** is specified, DrDialog attempts to assign **newId** as the ID of the new dialog . It returns the actual ID assigned to the new dialog (which may be different from **newId** if **newId** is already in use).

DrDialog 'NewControl' subcommand

```
actualId = DrDialog( 'NEWControl', id/class [, newId] )
```

Requests DrDialog to create a new control, either cloning an existing control with a specified **id**, or having a specified **class**. If **newId** is specified, DrDialog attempts to assign **newId** as the ID of the new control. It returns the actual ID assigned to the new control (which may be different from **newId** if **newId** is already in use).

If **id** is specified, it must be the ID of an existing control which is not of type **DIALOG**. The newly created control will be identical to **id**, including its size and position.

If **class** is specified, it must be a valid DrDialog control type, but not **DIALOG** (e.g. **PUSHBUTTON**). A new control of the specified type will be created and positioned in the center of the current dialog being edited.

DrDialog 'DropDialog' subcommand

```
CALL DrDialog 'DROPDialog' [, id]
```

Requests DrDialog to discard the dialog specified by **id**. If **id** is omitted , it defaults to the current dialog being edited.

If the dialog being discarded is the current dialog, DrDialog will automatically select another dialog as the new current dialog. If there are no other dialogs, DrDialog will automatically create a new empty dialog.

DrDialog 'DropControl' subcommand

```
CALL DrDialog 'DROPControl' [, id]
```

Requests DrDialog to discard the control specified by **id**. If **id** is omitted, it defaults to the current selected control, if any.

DrDialog 'Dialog' subcommand

```
oldId = DrDialog( 'DIALOG' [, newId] )
```

Returns the ID of the current dialog being edited by DrDialog. If **newId** is specified, it requests that DrDialog select the specified dialog as the new current dialog being edited.

DrDialog 'Control' subcommand

```
oldId = DrDialog( 'CONTROL' [, newId] )  
actualId = DrDialog( 'CONTROL', id, newId )
```

In the first form, it returns the ID of the current active DrDialog control, if any. If no control is currently active, it returns the null string. If **newId** is specified, it requests that DrDialog deselect all current controls and select **newId** as the new active control.

If the second form is used, it requests that DrDialog assign **newId** as the new ID of the control whose current ID is **id**. The actual new ID assigned is returned as the result (it may be different from **newId** if **newId** is already in use).

DrDialog 'Select' subcommand

```
oldIdList = DrDialog( 'SElect' [, newIdList] )
```

Returns as a blank delimited string the IDs of all currently selected DrDialog controls (e.g. '**105 101 118**'). The first ID in the list is always the current active control. If no controls are currently selected, the null string is returned.

If **newIdList** is specified, DrDialog unselects all currently selected controls and selects all the controls specified in **newIdList**, which should be a blank delimited list of control IDs, with the first ID in the list being the ID of the new active control.

Note: Not all controls in **newIdList** may end up being selected. DrDialog does not allow a container control and any of its contained controls to be selected at the same time.

DrDialog 'Name' subcommand

```
oldName = DrDialog( 'Name', id [, newName] )
```

Returns the name currently assigned to the DrDialog control or dialog whose ID is specified by **id**. If no name is currently assigned to **id**, the null string is returned.

If **newName** is specified, DrDialog attempts to assign it as the new name for the control or dialog specified by **id**.

Note: If **newName** has already been assigned to another control or dialog , DrDialog will attach a numeric suffix of the form **_n** to **newName** and use that as the new assigned name.

DrDialog 'Text' subcommand

```
oldText = DrDialog( 'Text', id [, newText] )
```

Returns the text currently associated with the DrDialog control whose ID is specified by **id**.

If **newText** is specified, DrDialog replaces the current text with **newText**.

The text associated with a control varies from control to control. For a pushbutton, it is its label. For a dialog, it is its window bar title. For a multi -line edit control, it is the complete text contained within the control. Other controls may have not any text associated with them (e.g. a **Rectangle** control). In that case, the result returned is the null string, and any value set is ignored.

DrDialog 'Hint' subcommand

```
oldHint = DrDialog( 'Hint', id [, newHint] )
```

Returns the hint text currently associated with the DrDialog control whose ID is specified by **id**.

If **newHint** is specified, DrDialog replaces the current hint text with **newHint** .

The hint text associated with a control is displayed at run-time whenever the pointer passes over the control.

DrDialog 'Position' subcommand

```
oldPos = DrDialog( 'Position', id [, newPos] )
oldPos = DrDialog( 'Position', id [, newX [, newY
[, newDX [, newDY]]]] )
```

Returns the current position and size of the DrDialog control whose ID is specified by **id** as a string of the form: **x y dx dy**, where **x y** is the coordinate of the lower left hand corner of the control, and **dx dy** is the width and height of the control in pels.

If **newPos** is specified, DrDialog sets the new size and position of the control using the values in **newPos**, which should be a string of the form: **x y dx dy**. If any trailing values are omitted, their current values remain unchanged .

If **newX**, **newY**, **newDX** and **newDY** are specified, DrDialog uses these values to set the new position and size of the control. If any trailing arguments are omitted, their current values remains unchanged.

In either case, DrDialog will adjust the values specified if the resulting size or position would place the control outside the boundaries of the current dialog (or the screen if the control specified is the dialog frame).

DrDialog 'Style' subcommand

```
oldStyle = DrDialog( 'Style', id [, newStyle] )
```

Returns the style currently associated with the DrDialog control whose ID is specified by **id**. The style returned is always a four byte long string whose bits correspond to the style bit mask associated with the control.

If **newStyle** is specified, DrDialog replaces the current style mask with **newStyle**, which must also be a four byte long string encoding the new style

bit mask for the control.

DrDialog 'Font' subcommand

```
oldFont = DrDialog( 'FONt', id [, newFont] )
```

Returns the font currently associated with the DrDialog control whose ID is specified by **id**. The font is returned as a string of the form: **size.name**, where **size** is the point size, and **name** is the family name of the font (e.g. **10.Courier**). If the current font is the default font for the control, the null string is returned.

If **newFont** is specified, DrDialog replaces the current font with **newFont**, which must also be a string of the form: **size.name** or the null string. If the null string is specified, DrDialog resets the font for the control back to the default font.

DrDialog 'Color' subcommand

```
oldColor = DrDialog( 'COLor', id , attribute [, newColor] )
```

Returns the specified color attribute for the DrDialog control whose ID is specified by **id**.

Attribute specifies which color attribute the function applies to. The attribute consists of a string of characters, each of which specifies a color attribute modifier. The defined attribute modifiers are as follows:

- + Foreground (group 1)
- Background (group 1)
- A Active (group 2)
- I Inactive (group 2)
- H Highlight (group 2)
- D Disabled (group 2)
- T Text (group 3)
- M Menu (group 3)
- B Border (group 3)

The three groups represent more or less disjoint sets of attributes. In forming an attribute name, no more than one character from each group should be used . However, not all combinations of characters specify a valid color attribute . The list of valid color attribute character combinations is as follows:

+ Foreground color
- Background color
A Active color
I Inactive color
AT+ Active text foreground color
AT- Active text background color
IT+ Inactive text foreground color
IT- Inactive text background color
H+ Highlight foreground color
H- Highlight background color
D+ Disabled foreground color
D- Disabled background color
M+ Menu foreground color
M- Menu background color
MH+ Menu highlight foreground color
MH- Menu highlight background color
MD+ Menu disabled foreground color
MD- Menu disabled background color
B Border color

Note: The order of the characters in **attribute** does not matter.

Not all controls support all color attributes. The most commonly supported attributes are foreground and background color.

If **newColor** is specified, the specified control color attribute is replaced by **newColor**.

Note: A color is specified as a string of the form:

#index
or #red green blue

where **index** is a color index, and **red**, **green** and **blue** are the color components of an RGB triplet (each component should be in the range 0 to 255).

The result of the function is also one of these two forms, depending on which form was originally used to set the corresponding color attribute.

Note: If no color attribute has been specified for a control, the null string is returned as the result.

DrDialog 'Event' subcommand

```
oldCode = DrDialog( 'EVENT', id, event [, newCode] )
```

Returns the REXX event handler code currently associated with the event specified by **event** for the DrDialog control whose ID is specified by **id**.

Event must be a valid event for the class of control specified by **id** (e.g. the valid events for a control of class **PUSHBUTTON** are **Init** and **Click**). A list of valid event names for a particular class of control can be obtained using the Event subcommand of the **DrDialog** function.

If **newCode** is specified, it replaces the current event handler for the specified **event** and **id**. If **newCode** is specified, but is the null string, the current event handler for **event** and **id** is deleted.

DrDialog 'Class' subcommand

```
class = DrDialog( 'Class', id )
oldCode = DrDialog( 'Class', class, event [, newCode] )
```

In the first case, it returns the class of the DrDialog control whose ID is specified by **id**.

In the second case, it returns the REXX class handler code currently associated with the event specified by **event** for the DrDialog control class specified by **class**.

Event must be a valid event for the class of control specified by **class** (e.g. if **class** is **PUSHBUTTON** the valid events are **Init** and **Click**). A list of valid event names for a particular class of control can be obtained using the Event subcommand of the **DrDialog** function.

If **newCode** is specified, it replaces the current class handler for the specified **event** and **class**. If **newCode** is specified, but is the null string, the current class handler for **event** and **class** is deleted.

DrsAide tools

The DrDialog package includes a number of tools written in DrRexx using the DrsAide extension mechanism. These tools are intended to both increase the power and usefulness of the DrDialog programming environment as well as to illustrate how to write additional DrsAide tools.

Some of the tools are pre-installed in the DrsAide tool, while others can be installed at the user's discretion using the methods discussed in the section on using the default DrsAide tool from the Workplace Shell. The available tools are :

Array Generates and lays out a rectangular grid of controls (pre-installed)

REView Displays the current set of dialogs in outline form (pre-installed)

REStoRXX Writes all REXX source code for the current set of dialogs into an annotated listing file (pre-installed)

RexxLib Browses all code currently stored in the REXX library (pre-installed)

RexxUse Includes as global procedures all REXX library routines referenced by the current set of dialogs (pre-installed)

BMPList.RES Displays **.BMP** and **.GIF** files (optional tool available in the **SAMPLE** subdirectory)

Clock.RES Simple clock (optional tool available in the **SAMPLE** subdirectory)

Clock2.RES Fancier clock (optional tool available in the **SAMPLE** subdirectory)

Calc.RES Simple calculator (optional tool available in the **SAMPLE** subdirectory)

Array tool



The array tool provides a simple interface for creating and laying out rectangular grids of the same kind of control (e.g. a grid of **ICONBUTTON**s for a tool bar).

To use the array tool:

1. Make an instance of the control you wish to create an array of.
2. Initialize and set its attributes (i.e. size, text, font, colors, style).
3. Position it at the top-left corner of the rectangular array you wish to create.



4. Invoke the array tool from the DrsAide tool by clicking the  button.
5. Use the appropriately labeled spin buttons in the array tool to specify the number of rows and columns the array of controls is to have.
6. Use the appropriately labeled spin buttons to specify the horizontal and vertical spacing between controls in the array.
7. Click the **Create** button to create the array of controls and lay them out .

If the spacing is not quite right, simply change the values displayed in the spacing spin buttons and click the **Space** button to lay out the controls in the array again. Repeat this step until you are satisfied with the layout.

If you make a mistake or change your mind, you can delete all the controls in the array (except the original) by clicking the **Delete** button.

Once you are finished you can either close the array tool, or click the **Done** button to signal that you are ready to create a new array of controls.

Note: The **Create** button creates an array of controls using the currently selected control as its template. It copies the class, size, text, font, colors and style of the selected control to each new control in the array. Once an array of controls has been created, clicking the **Done** button re-enables the **Create** button and allows a new array of controls to be created from the currently selected control.

REView tool



The review tool displays the current set of dialogs being edited in outline form. The outline consists of labeled icons, each representing a particular dialog or dialog component.

Initially, the outline is in its *collapsed* form, with only the dialog and global procedure icons displayed. Clicking the **plus sign** to the left of an icon expands the outline to include the components of the icon.

In the case of a dialog icon, expanding it displays an icon for each control within the dialog, including the frame and drop-down menu if any. Further expanding a control icon displays icons for each event or class handler defined for the control. Expanding a drop-down menu icon displays icons for each submenu or menu item.

Expanding the global procedures icon displays an icon for each defined global procedure.

Double-clicking an icon that represents code, such as a global procedure or control event handler, displays its associated REXX code in the tool's edit control. Alternatively, you can also drag the icon and drop it on the edit control to display its associated REXX code. If desired, the *Copy to clipboard* menu option can be used to copy the code to the system clipboard.

Once an icon has been expanded, it can be collapsed again by clicking the **minus sign** to the left of the icon.

RexxUse tool



The RexxUse tool defines as global procedures any REXX library routines referenced by the current set of dialogs being edited. It searches through your application's REXX code looking for procedure references of the form: **CALL _procedure_ or _function_ (...)**, and if **_procedure_ or _function_** is defined in the REXX library, it copies the definition from the library into the application as a global procedure.

REXX library references within already included REXX library routines are also resolved automatically. In addition, if a previously included REXX library routine is no longer referenced by the application, its global procedure will be deleted from the application.

Note: Procedures may be entered into the REXX library using the [RexxLib](#) tool.

RexxLib tool



The RexxLib tool allows you to update and browse the contents of the REXX library. The REXX library is a collection of generally useful REXX functions and procedures that can easily be included into any application using the [RexxUse](#) tool.

The RexxLib tool can be invoked in one of two ways:



oFrom the DrsAide tool by clicking the  button. When invoked this way, RexxLib allows you to browse through the names and definitions of all current REXX library entries. A list of all current library entries appears on the left side of the dialog. Double-clicking an entry displays its definition on the right. The bottom of the dialog also displays information about the origins of the routine . Clicking the **Copy to clipboard** button copies the current definition to the system clipboard.

oFrom the Workplace Shell. Dragging and dropping a file containing REXX procedures onto the **RexxLib** icon in the **DrDialog** folder adds or updates the REXX procedures contained in the file to the REXX library. For each procedure in the file, one of the following actions occurs:

- If the procedure is not already in the REXX library, it is added to the library.
- If the procedure is already in the library, and was originally from the same file, its definition is updated.
- If the procedure is already in the library, and was originally from a different file, its definition is *not* updated. A warning message is displayed indicating that the procedure is already defined by another file. Double-clicking on the warning message will override the warning and replace the previous definition of the procedure with the new one.

In addition, if the REXX library contains procedures previously defined by the file, but which are no longer contained in the file, those definitions are deleted from the library.

The file containing REXX procedures should have the following format:

```
[discarded header information]
_label1_ : [REXX statements]
[more REXX statements]
_label2_ : [ REXX statements]
[more REXX statements]
...
_labeln_ : [ REXX statements]
[more REXX statements]
```

In order to be recognized as a REXX library procedure, the label defining the start of a procedure must begin and end with underscores (i.e. '_') . The label is used as the name of the entry in the REXX library. The definition of the entry consists of all following REXX statements until the next label beginning and ending with underscores, or the end of the file, is encountered. Other labels not meeting the above criteria are simply included as part of the definition of the last label that does.

BMPList tool



The BMPList tool allows you to browse collections of .BMP and .GIF files. A radio button allows you to select which type of file you are currently interested in. To select a group of files for browsing, enter the path name of



their directory into the entry field, then press **Enter** or click the  button. The list box will display the names of all selected files in the specified directory. To view a particular file, simply select its name from the list box. If

the currently selected control is an **ICONBUTTON** or **BILLBOARD**, the image selected will also appear in the control.



To temporarily save a copy of a particular image, click the  button and a new dialog containing a copy of the image will be created.

Each image dialog also has controls to allow the current image to be viewed :

- oCentered within the display area
- oScaled to fill the entire display area
- oReplicated to fill the entire display area

Writing your own DrsAide tool

There are basically three parts to writing a DrsAide tool:

- oCreating the tool
- oIntegrating the tool into DrDialog
- oCreating a bitmap for the tool's DrsAide icon button

The first part, creating the tool, is usually accomplished using various subcommands of the **DrDialog** function to examine and modify information within the DrDialog programming environment. Feel free to use any of the DrsAide tools distributed with DrDialog as examples of how to go about doing this.

The second part, integrating the tool into DrDialog, can easily be accomplished using several REXX library functions provided with DrDialog:

DrsAideInit This function should be called from the **Init** global procedure of your tool. Its purpose is to initialize the link to DrDialog via the DrDialog function and to open the first dialog of your tool if it was invoked via DrsAide. Its usage is:

```
rc = _DrsAideInit_( hwnd, iniFile [, dialog] [, bitmap] )
```

where **hwnd** and **iniFile** are the first two command line arguments passed to the tool by **DrsAide**, **dialog** is the optional name of the dialog to open (it defaults to the first dialog in the application), and **bitmap** is the name of the bitmap to display on the DrsAide icon button for your tool (specified only if your tool does not have a **.BMP** file with the same name as the tool's **.RES** file).

The function returns 1 if the tool was invoked from DrsAide, and 0 otherwise . If 0 is returned, you may wish to initialize the tool differently, or display an error message.

For example:

```
PARSE ARG hwnd iniFile rest
IF _DrsAideInit( hwnd ) = 0 THEN EXIT
```

`_DrsAideDialogInit_` This routine should be called from the **Init** event handler of each dialog opened by your tool (unless the dialog is owned by another dialog in your tool). It registers the dialog with `DrDialog` and attempts to restore the last saved size and position of the dialog.

Note: This procedure also *shows* the dialog. It is a good idea to create the dialog with the **Visible** attribute of the dialog off in order to allow the `_DrsAideDialogInit` procedure to size and position the dialog correctly before displaying it.

Its usage is:

```
CALL _DrsAideDialogInit_ hwnd, iniFile
```

where **hwnd** and **iniFile** are the first two command line arguments passed to the tool by **DrsAide**.

For example:

```
...in Init global procedure ...
PARSE ARG hwnd iniFile rest
...
... in dialog Init event handler ...
CALL _DrsAideDialogInit_ hwnd, iniFile
```

`_DrsAideDialogExit_` This routine should be called from the **Exit** event handler of each dialog opened by your tool whose **Init** handler calls `_DrsAideDialogInit_`. It saves the size and position of the dialog in the `DrsAide .INI` file so that its location can be restored by the `_DrsAideDialogInit_` procedure the next time the tool is invoked. Its usage is:

```
CALL _DrsAideDialogExit_ hwnd, iniFile
```

where **hwnd** and **iniFile** are the first two command line arguments passed to your tool by **DrsAide**.

For example:

```
...in Init global procedure ...
PARSE ARG hwnd iniFile rest
...
... in dialog Exit event handler ...
CALL _DrsAideDialogExit_ hwnd, iniFile
```

The source for each of the above routines is contained in the **DrsAide.RXL** file in the **DrDialog** folder. To add these routines to your REXX library, drag and drop the **DrsAide.RXL** file icon onto the **RexxLib** icon, also in the **DrDialog** folder. Once you have done this, you may obtain more detailed information by browsing their source using the RexxLib tool.

Also, once installed into your REXX library, you may incorporate these routines into a tool by first writing the code invoking them and then using the RexxUse tool to automatically include the correct procedures into your application.

Alternatively, you can use the **DrsAideT.RES** template in the **DrDialog** directory to quickly create a template for a new DrsAide tool. Drag and drop the **DrsAideT.RES** icon into a folder to create a new **.RES** file which has the appropriate calls to the routines just described already installed. Once the new **.RES** file is created, you can:

- oRename the tool by clicking on its icon label with the **Alt** key pressed and then typing in the name of the tool you are creating.
- oInstall the tool into **DrsAide** by dragging and dropping the new **.RES** file's icon onto the **DrsAide** icon in the **DrDialog** folder. By default, the new tool will use a standard icon in the **DrsAide** tool bar. If you wish, you can create a new **.BMP** file with the same name prior to installing the **.RES** file into **DrsAide**, or you can change the default icon later by double-clicking the **DrsAide** icon to edit the tool's bitmap name directly.
- oDefine the tool by dragging and dropping the new **.RES** file's icon onto the **DrDialog** icon in the **DrDialog** folder. Once the editor is invoked, you may add the necessary controls and REXX code to complete your application. Once the tool has been saved, you can test it simply by clicking on its icon button in the **DrsAide** tool bar.

The final part, creating a bitmap for the tool's DrsAide icon button, can be done using any bitmap editor (e.g. ICONEDIT.EXE). The resulting bitmap should be given the same name as your tool's **.RES** file. This will allow **DrsAide** to locate the bitmap when a user installs your tool.

Alternatively, if you wish to use a bitmap stored in a **.DLL** (e.g. **BITMAP.DLL**), you can supply the name of the bitmap in the call to the **_DrsAideInit_** function. When the user installs your tool and **DrsAide** is not able to find a corresponding **.BMP** file, it will invoke your tool with a special command line argument which indicates that your tool must install itself. If you call **_DrsAideInit_** in your global **Init** procedure and pass it the name of the bitmap to use, it will automatically handle this special case and install your application correctly .

And finally, each tool invoked by **DrsAide** is passed as its second command line argument the name of an **.INI** file the tool can use to save information in . Information can saved and restored in the **.INI** file using the REXX **SysIni** function. The REXX library **_DrsAideIniApp_** function can also be used to generate application keys of the form: **resFile:dialog** (e.g. **Calc:calc**) .

These can be used with the **SysIni** function to ensure that each DrsAide tool uses unique application keys for its data. The usage for **_DrsAideIniApp_** is :

```
key = _DrsAideIniApp_()
```

Note that this function will generate a unique application key for each tool dialog it is used from.

For example, a DrsAide tool that does special initialization the first time it is invoked might use the following code:

```
PARSE hwnd iniFile rest
...
/* Dialog 'Init' event handler: */
IF SysIni( iniFile, _DrsAideIniApp_(), 'Inited' ) = 'ERROR:' THEN DO
CALL SysIni iniFile, _DrsAideIniApp_(), 'Inited', 1
/* One time initialization code goes here... */
...
END
...
```

Adding hints to your DrsAide tool

Since a DrsAide tool is really just a DrRexx application, it can have user hints like any other DrRexx program. See the section on adding user [hints](#) to a DrRexx application for more information on how to do this.

Ideally though, it would be useful if a DrsAide tool's hints appeared in the same hint controls as the DrDialog editor, providing a seamless integration of DrsAide tools with other DrDialog tools. And this is just what happens!

When your DrsAide tool registers itself with DrDialog using the **Owner** subcommand of the [DrDialog](#) function, it is automatically set up to display its hint text in the same controls that DrDialog uses. Note however that this will only work correctly if your DrsAide tool does not explicitly set a hint control using the [IsDefault](#) function, either before or after issuing the **Owner** subcommand. By default it will just do the right thing!

Utilities

The DrDialog package includes four utilities that help create and maintain DrDialog and DrRexx applications:

BMPTODLL Create a **.DLL** from one or more **.BMP**, **.ICO** or **.PTR** files.

REStoRXX Create a source listing of the REXX code associated with a DrRexx **.RES** file.

REView Display a DrRexx **.RES** file in the form of an outline

REVis Allows sections of a DrRexx **.RES** file to be cut and pasted using drag and drop operations.

BMPtoDLL

The **BMPtoDLL** utility creates a **.DLL** from one or more **.BMP**, **.ICO** or **.PTR** files. This utility is useful in conjunction with DrDialog **ICONBUTTON** and **BILLBOARD** controls, which can display bitmaps stored in a **.DLL**.

The syntax for invoking **BMPtoDLL** is:

```
BMPtoDLL dllName [file1 file2 ... fileN]
```

where **dllName** is the name of the **.DLL** to create (the **.DLL** extension is optional). **file1** through **fileN** are the names of the **.BMP**, **.ICO** or **.PTR** files to include in the **.DLL**. Wildcard characters (i.e. '*' or '?') may be used in the file names. If not specified, **file1** defaults to '*.BMP'.

Note: The **BMPtoDLL** command uses a data file called **BMPtoDLL.DAT**, which must be present either in your **PATH** or **DPATH**. If it cannot be found, an error message will be displayed and no **.DLL** file will be created.

REStoRXX

The **REStoRXX** utility creates a source file listing of the REXX code associated with a DrRexx **.RES** file. To use **REStoRXX**, drop any DrRexx **.RES** file onto the **REStoRXX** icon in the **DrDialog** folder. **REStoRXX** will create a file with the same name, but with a **.RXX** extension, containing a listing of the REXX source code associated with the **.RES** file.

Note: The **.RXX** source file created by **REStoRXX** is very similar to the REXX program created by DrDialog when your DrRexx application runs, but it is not identical. The source listing is intended mainly as documentation for your DrRexx application. At present, there is no way for the **.RXX** file to be directly imported back into DrDialog.

REView

The **REView** utility displays a DrRexx **.RES** file in the form of an outline . At the top level of the outline are dialogs and global procedures. At the level below a dialog are the dialog's controls and drop down menu. At the level below a control are its event handlers, and so on. The outline may be expanded or collapsed to display more or less detail about the structure of the application.

To invoke **REView** from the Workplace Shell, drop any DrRexx **.RES** file onto the **REView** icon in the **DrDialog** folder.

For more information about how to use **REView** once it has been invoked, refer to the [REView tool](#) section in the DrsAide portion of this document.

REVise

The REVise utility provides a simple, graphical means of copying dialogs and REXX code from one DrRexx application to another.

REVise works by displaying a DrRexx **.RES** file in the form of an outline . At the top level of the outline is the **.RES** file itself. At the level below that are sections for dialogs, global procedures and any external code contained in the file. And finally, at the level below that are the individual dialogs and global procedures contained within the file. Levels within the hierarchy can be expanded or collapsed as desired. Each icon within the hierarchy can also be dragged and dropped on either the **Add** or **Delete** icons located at the bottom of the dialog, either in the same dialog, or in the case of the **Add** icon, in any other instance of the REVise utility currently running.

Dropping an object on the **Add** icon adds an identical copy of the specified object to the corresponding **.RES** file. If any names or IDs within the object being copied conflict with names or IDs already in the target **.RES** file, the REVise utility will automatically rename or renumber the objects as needed to avoid conflicts. Messages indicating the changes made will be displayed in a pop-up message log window. Note that copying a dialog also copies any REXX event handlers associated with the dialog.

Dropping an object on the **Delete** icon removes the specified object from the **.RES** file. Note that objects can only be deleted from the **.RES** file they are contained in.

No changes are actually made to the **.RES** file until you close the dialog . At that time, if any changes have been made, you will be asked whether you wish to save or discard the changes.

To invoke **REVise** from the Workplace Shell, drop any DrRexx **.RES** file onto the **REVise** icon in the **DrDialog** folder.

User preferences

DrDialog keeps track of a number of user preference items in the **DRDIALOG .INI** file.

The user preference items are:

- oThe last position, size and pattern for the background window
- oThe last position and size of each tool window
- oThe **Auto open**, **Auto hide**, and **Float settings** of each tool window
- oThe last set of ID tool options used
- oThe last selected size tool unit (**pels** or **dialog** units)
- oThe name of the last dialog file saved or loaded
- oThe dialog file save options last used
- oThe last application switched to using the <- **Switch** button
- oThe last font and color set for both the DrRexx event and drop-down menu action multi-line edit controls. Note that the font can be set by dragging and dropping a Workplace Shell **Font Palette** object onto either multi-line edit control. The color can be set by dragging and dropping a Workplace Shell **Color Palette** object onto either multi-line edit control.

These preference items are used the next time the editor is run to restore the editing environment to its previous state as much as possible.

Related packages

The **ZBMFUNCS** package on OS2TOOLS by Dario de Judicibus contains several **.DLL** files containing bitmaps that are well suited for use with DrDialog bitmap buttons. The **.DLL's** are organized by category, and include alphabetical and typographical symbols, as well as symbols useful with paint and publishing programs. All of the bitmaps in the collection are well executed graphically and visually appealing.

Acknowledgements



David C. Morrill

DrDialog was written by:

David C. Morrill
IBM T. J. Watson Research Center

TOPVIEW at YKTVMH
topview@watson.ibm.com

DrDialog is an object-oriented program completely written in Oberon-2 using the HOPE Oberon programming environment, also written by David C. Morrill. HOPE is available separately as the HOPE PACKAGE on OS2TOOLS.

Screen capture for this document was done using BMP, another Oberon program developed by David C. Morrill using HOPE.

Footnote

Clicking the **Replace** button updates the selected menu item with the current values of the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields.

Footnote

Clicking the **Cancel** button restores the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields to their last saved state.

Footnote

Clicking the **Delete** button deletes the currently selected menu item.

Footnote

Clicking the **Menu Item** button inserts a new menu item after the currently selected menu item. The contents of the **Text**, **Label**, **Action**, **Checked**, and **Disabled** fields are used to define the new menu item.

If the **action** field is empty, the menu item is a static menu item.

Footnote

Clicking the **Submenu** button inserts a new submenu after the currently selected menu item. The new submenu initially has no menu items. The contents of the **Text**, **Label**, **Checked** and **Disabled** fields are used to defined the new submenu item.

Footnote

Clicking the **Separator** button inserts a new separator after the currently selected menu item. The contents of the **Text**, **Label**, **Action**, **Checked** and **Disabled** fields are ignored.

Footnote

The state of the **Checked** field is used to set the check status of a menu item when the **Replace**, **Menu Item**, or **Submenu** buttons are clicked.

Footnote

The state of the **Disabled** field is used to set the disabled status of a menu item when the **Replace**, **Menu Item**, or **Submenu** buttons are clicked.

Footnote

The contents of the **Text** field are used to set the text of a menu item when the **Replace, Menu Item,** or **Submenu** buttons are clicked.

Footnote

The contents of the **Label** field are used to set the label of a menu item when the **Replace**, **Menu Item**, or **Submenu** buttons are clicked.

Footnote

The contents of the **Action** field are used to define the REXX code associated with a menu item when the **Replace** or **Menu Item** buttons are clicked.

Footnote

The menu bar shows what the drop-down menu being edited will look like when the application is running. Selecting an item from the menu will also select the corresponding menu item for editing.

Footnote

The left-most listbox shows the menu items currently defined for the menu bar . Selecting an entry from the list selects the corresponding menu item for editing or insertion.

Footnote

The middle listbox shows the menu items currently defined for the submenu selected in the left-most listbox. Selecting an entry from the list selects the corresponding menu item for editing or insertion.

Footnote

The right-most listbox shows the menu items currently defined for the submenu selected in the middle listbox. Selecting an entry from the list selects the corresponding menu item for editing or insertion.

Footnote

Clicking the **stop** button terminates execution of the DrRexx application.

If running in the stand-alone DrRexx environment, the entire application is terminated.

If running under control of the DrDialog editor, all application dialogs are closed and control returns to the editor.

Run mode is ended and **edit** mode resumes. The  button replaces the  button. You may click the **run** button to start the DrRexx application again.

Note: While in **edit** mode, all DrRexx run-time controls (other than the **run** button) are disabled.

Footnote

Clicking the **break** button interrupts the currently executing REXX code (if any) and forces the run-time environment to wait for the next input event.

If REXX code was being executed, an error message indicating what code was executing will be displayed in the output list control.

Footnote

Clicking the **clear** button clears the contents of the output list control.

Footnote

Clicking the **evaluate** button causes the contents of the entry field to its right to be interpreted as a REXX statement. This can be useful for displaying or modifying information about the state of the currently running DrRexx application.

Note: If interactive debug mode is active, the  button may be replaced by the  button. This indicates that the contents of the entry field to the right of the button will be used to control the REXX interactive trace. It also indicates that the REXX code is suspended in trace mode, and is not ready to process the next input event. The reappearance of the **evaluate** button indicates that the DrRexx application is waiting for the next event.

Footnote

Clicking the **trace** button toggles between REXX *trace* and *interactive debug* modes. The current mode is

indicated by the appearance of the **trace** button. The  button indicates trace mode, while the  button indicates interactive debug mode .

In trace mode, REXX trace messages appear in the output list control, but the program does not stop.

In interactive debug mode, REXX trace messages appear in the output list control, and execution stops after every

trace event. Clicking the  button allows the contents of the entry field to its right to control REXX interactive debug mode .

Note: The **trace** button does not turn REXX trace mode on or off, but only indicates whether the trace should be interactive or not. The drop down list to its right can be used to set the current trace mode.

Footnote

Selecting an item from the **trace mode** list determines the current REXX trace mode in effect. See the **Procedures Language/2 REXX Reference** for a detailed explanation of the various REXX trace modes.



The  button can be used in conjunction with the setting of this control to select whether tracing should be interactive or not.

Footnote

Clicking the **variables** control displays the current value of all REXX variables for the DrRexx application in the control's drop down list.

Selecting an item in the list will copy a statement of the form **SAY 'var =' var** to the entry field to allow for easy monitoring of the selected variable using the  button.

Footnote

The current contents of the **interpret** field can be evaluated as a REXX statement by clicking on the  button to its left or by pressing the **Enter** key.

Note: In interactive debug mode, the **evaluate** button is replaced by the  button. The contents of the **interpret** field are then used to control interactive debug mode when the **debug** button is clicked.

Footnote

The contents of the **output** list shows the results of REXX SAY statements or program trace messages. Only the last 100 messages issued are displayed.

Selecting an entry in the list copies it to the **interpret** entry field above it (unless the selected item is of the form **var = value**, in which case **SAY ' var = ' var** is copied to the **interpret** field.

Footnote

Clicking the **undo** button restores the edit control to the last *saved* value of the text (or the original value if it has not yet been saved).



The text can be *saved* by clicking on the button located just above the **undo** button.

Footnote

Clicking the **save** button *saves* the current contents of the edit control. If you later make some changes to the text that you wish to discard, you can restore the text back to the last saved value by clicking the  button located just below the **save** button..

Footnote

Clicking the **paste** button replaces the currently selected text with the contents of the system clipboard.

Footnote

Clicking the **copy** button copies the currently selected text into the system clipboard.

Footnote

Clicking the **cut** button deletes the currently selected text and copies it into the system clipboard.

Footnote

Clicking the **find** button searches the edit control for the next occurrence of the text string currently in the search field next to the find button. The search always begins at the current cursor position.

Footnote

Clicking the **switch** button causes the application whose OS/2 Presentation Manager Window List entry starts with the contents of the search field to be brought to the foreground.

Footnote

Enter the text to search for into the **search** field, then click the **Find** button to search for the text in the edit control. The search always begins at the current cursor location.

Footnote

Selecting an item in the spin button control selects the type of text to be displayed in the edit control:

Control The control specific REXX code

Class The generic class REXX code

Events The set of events defined for the control

Functions The set of window functions defined for the control

Footnote

Clicking the **notepad** page tab will select the DrRexx notepad section, which allows you to enter and edit REXX code fragments or other useful pieces of information that will be available in every DrDialog editing session.

Footnote

Clicking the **global procedures** page tab will select the global procedures section, which allows you to enter and edit REXX procedures callable from other parts of your DrRexx application.

Footnote

Clicking the **events** page tab will select the events section, which allows you to enter and edit REXX event handling code associated with the currently active control.

Footnote

The **edit** control allows you to enter and edit REXX code. It is a standard Presentation Manager multi-line edit control, and it uses the same font as the **System Editor** .

Footnote

The **status line** displays the name, ID, and type of the currently active control.

Footnote

Clicking an **event** page tab selects the code associated with the event whose name appears on the tab into the edit control for editing.

Footnote

Clicking a **global procedure** page tab selects the code associated with the procedure whose name appears on the tab into the edit control for editing.

Footnote

All currently defined global procedures are displayed alphabetically in this list. Double-clicking an entry in the list will select the page containing the code associated with the selected procedure.

Footnote

Enter the name of a global procedure into this entry field and press **Enter** to edit it. If the name is new, a blank notebook page will be created with the name of the procedure on its page tab.

Footnote

Clicking a **notepage** page tab selects the text associated with the note whose name appears on the tab into the edit control for editing.

Footnote

All currently defined notes are displayed alphabetically in this list. Double-clicking an entry in the list will select the page containing the text associated with the selected note.

Footnote

Enter the name of a note into this entry field and press **Enter** to edit it . If the name is new, a blank notebook page will be created with the name of the note on its page tab.

