Using Borland Windows custom controls

The custom dialog class, BORDLG, works on both a visual and functional level:

▪ It changes the appearance of your dialog window by painting the background with a brush that varies according to the target display device. For screens of VGA and higher resolution, the background is a fine grid of perpendicular white lines, giving the effect of "chiseled steel." For EGA and monochrome screens, the background is white.

▪ It optimizes the drawing of dialog boxes by calling the custom control drawing routines directly instead of waiting for Windows to paint the controls. This eliminates the typical sluggish drawing of dialog boxes.

To use the Borland custom dialog class,

1. Open the dialog resource you want to convert.

2. Double-click in an empty area of the dialog box to display the Property Inspector.

3. Select the Window page.

4. Enter "bordlg" as the Class and click OK.

Customizing existing applications for Borland custom controls

Note: This procedure applies to 16-bit Windows applications only.

Resource Workshop allows you to customize existing Windows applications with Borland-style custom controls. There are two steps to this process:

1. Modify your WIN.INI file to load the Borland Windows Custom Control (BWCC) library each time you start Windows.
2. Edit the application to change user interface features such as dialog boxes, menus, icons, and so on.

Loading BWCC to enable Borland custom controls

The BWCC library, which provides support for Borland-style custom controls, must be loaded before an application can use BWCC's features.

Edit the WIN.INI file (located in the Windows main directory) so that Windows loads the file LOADBWCC.EXE into memory at start up. (The installation program should have put LOADBWCC.EXE in the language compiler directory and added this directory to your PATH.)

Add LOADBWCC.EXE to the beginning of the list of files that appear after the "LOAD=" statement. LOADBWCC.EXE must appear first in the statement to ensure that BWCC is loaded into memory before any modified applications are executed.

Borland custom controls

These Borland custom control styles available are:

Control Name	Control	Description
Tri-State Checkbox		A Borland-style check box that has three states - on, off, and "indeterminate," which is displayed as a checkerboard pattern. The application determines what is meant by "indeterminate." The application must call the **CheckDlgButton** function to send a BM_SETCHECK message to check the selected box.
Auto Tri-State Checkbox		A check box that is identical to a Borland-style 3-state check box, except that BWCC and Windows combine to handle checking the selection box.
Auto Checkbox		A check box that is identical to a Borland-style check box, except that BWCC and Windows combine to handle checking the selection box.
Auto Radiobutton		A radio button that is identical to a Borland-style radio button, except that BWCC and Windows combine to handle highlighting the selected button and deselect the other buttons.
Bitmap		A bitmap.
Checkbox		A Borland-style check box. The application must call the **CheckDlgButton** function to send a BM_SETCHECK message to check the selected box.
Default Pushbutton		A push button that is identical to a Borland-style push button, but includes a bold border indicating that it is the default response if the user presses Enter.
Horizontal Bump		A convex horizontal line.
Horizontal Dip		A concave horizontal line.
Pushbutton		A Borland-style push button. When the user clicks the button, a BN_CLICKED message is sent to the parent window.

Radiobutton		A Borland-style radio button. The application must call the **CheckRadioButton** function to send a BM_SETCHECK message to highlight the selected button and deselect other buttons.
Raised Gray Group		A gray box that appears raised above the surface of the dialog box.
Recessed Gray Group		A gray box that appears recessed below the surface of the dialog box.
Static Text		A fixed text string used for labeling parts of the dialog box.
Vertical Bump		A convex vertical line.
Vertical Dip		A concave vertical line.

Borland button and check box enhancements

See also

Borland push buttons, radio buttons, and check boxes have the following functional enhancements over standard Windows controls:

▪ An additional level of parent window notification and control over keyboard focus and tab movement. If you choose the Notify Parent option in the control's property page, the control sends the appropriate message at run time:

▪ BBN_SETFOCUS indicates to the parent window that the push button, radio button, or check box has gained keyboard focus through an action other than a mouse click.

▪ BBN_SETFOCUSMOUSE indicates to the parent window that the push button, radio button, or check box has gained keyboard focus through a mouse click.

▪ BBN_GOTATAB indicates to the parent window that the user has pressed the Tab key while the push button, radio button, or check box has keyboard focus. The parent can intervene in the processing of the keystroke by returning a nonzero value.

▪ BBN_GOTABTAB indicates to the parent window that the user has pressed Shift-Tab (back-tab) while the push button, radio button, or check box has keyboard focus. The parent can intervene in the processing of the keystroke by returning a nonzero value.

▪ An owner-draw option that allows the parent window to draw the push button, radio button, or check box. Because your application handles drawing the control, it will not necessarily look like a Borland control, but it will have the standard behavior of that class of control.

Borland Windows custom control API

The following topics describe technical aspects of the Borland Windows Custom Controls (BWCC) and contains information that might be useful or of interest to the advanced resource designer:

- BWCC functions
- Defining a derivative dialog dlass
- Technical description of Borland Windows custom controls
- Using BWCC controls in non-dialog windows

Borland Windows custom control functions

BWCC.DLL exports these additional functions:

- BWCCGetPattern
- BWCCGetVersion
- BWCCIntlInit
- BWCCIntlTerm
- BWCCMessageBox
- BWCCRegister

BWCCRegister

Syntax
```
BWCCRegister (HINSTANCE hInst)
```

Description
This function is used to register an instance of the application with BWCC.DLL. It is required for 32-bit applications using BWCC controls and should be called when the application is initialized.

BWCCIntlInit

See also

Syntax

`BWCCIntlInit (UINT language)`

Description

This function (call on startup) selects a language (see BWCC.H for language choices) for text and bitmaps (it returns TRUE for success or FALSE for failure). If you do not use this function. BWCC uses its default resource language, which depends on which translated version you have. Each BWCC client can use a different language.

BWCCIntlTerm

Syntax
```
BWCCIntlTerm (VOID)
```

Description
This function frees memory (use on exit) after you use BWCCIntlInit. This function returns TRUE (success) or FALSE (failure).

BWCCGetVersion

Syntax

```
BWCCGetVersion()
```

Description

This function, which takes no parameters, returns the current version of BWCC.DLL. The value it returns is defined in BWCC.H as BWCCVERSION:

Value	Platform
0x0200	Windows 3.*x*
0x10200	Win32, Win32s, Windows NT, Windows 95

BWCCGetPattern

Syntax
```
BWCCGetPattern()
```

Description
This function, which takes no parameters, returns a handle to the brush used to paint the background of BorDlg class dialogs. Since this brush could be a patterned brush, you must align it by calling **UnrealizeObject** and **SetBrushOrg** before selecting it into a device context. Do not delete this brush by calling **DeleteObject**!

BWCCMessageBox

Syntax

```
BWCCMessageBox()
```

Description

This function, which is call-compatible with the Windows standard function **MessageBox**, displays a message box that is consistent with the Borland dialog box style.

Technical description of Borland custom control classes

These topics contain descriptions of each of these Borland Windows Custom Control classes:

- BORBTN control
- BORRADIO control
- BORCHECK control
- BORSHADE control
- BORSTATIC control
- BORDLG dialog class

Most of the subsection headings are self-explanatory, with the possible exception of the following:

- **Class name** gives the Resource Workshop name in quotation marks, followed by the identifier name.
- **Window styles** include **Types** and **Options**. Within each class there may be several *types* of controls. Types dictate the overall appearance and functionality of the control. Options are those available to each control type.
- **Messages** include **Commands** and **Notifications**. Commands are messages to a control. Notifications are a special type of WM_COMMAND message used by controls. The control ID of the control is passed in the *wParam* of the message, while the *lParam* contains both the notification type and the window handle of the control. The notification type is contained in the high-order word of *lParam* and can be extracted using the HIWORD macro; the window handle is contained in the low-order word of *lParam* and can be extracted using the LOWORD macro.

BORBTN control

Function
Bitmapped push buttons and "splash panels"

Class name
"borbtn" (BUTTON_CLASS)

Types inherited from standard Windows controls
BS_DEFPUSHBUTTON Defines the two standard Windows push button types:
- BS_DEFPUSHBUTTON
- BS_PUSHBUTTON

> The BS_DEFPUSHBUTTON type identifies the "default" push button. When the user presses the Enter key in a dialog box, the ID of the default button is in the *wParam* of the WM_COMMAND message sent to the parent window of the button. The Windows dialog manager sends a BN_CLICKED notification from that button to the dialog window.

> There are two exceptions:

- If another button gains keyboard focus through a Tab keystroke, that key temporarily becomes the default button and is referenced in the BN_CLICKED notification.
- If keyboard focus is in an edit control for which the ES_WANTRETURN flag is set, the Enter key inserts a carriage return into the text in the edit control.

Types unique to BWCC
BBS_BITMAP   This type is used to display "splash panels," which are bitmaps the user does not interact with.

Options unique to BWCC
BBS_PARENTNOTIFY This option causes the control to generate the following notification messages at run time:
- BBN_SETFOCUS
- BBN_SETFOCUSMOUSE
- BBN_GOTATAB
- BBN_GOTABTAB
BBS_OWNERDRAW This option causes the control to send WM_DRAWITEM to its parent at run time, for specialized drawing.

Commands inherited from standard Windows controls
BM_SETSTYLE   The Windows dialog manager uses this message to toggle between the BS_DEFPUSHBUTTON and BS_PUSHBUTTON types.

BM_SETSTATE   This message changes the "highlight" state of a button. If the wParam of the message is nonzero, the button is highlighted (drawn as if it were pressed).

BM_GETSTATE   This message determines whether a button is highlighted, has focus, and whether it is "checked" (checking does not, however, apply to buttons). The 0x0004 bit of the return value indicates that the button is highlighted (drawn with a heavy outline around the button); the 0x0008 bit indicates that the button has the focus (a dotted line surrounds the text caption).

Commands unique to BWCC
BBM_SETBITS   The application uses this message to pass a set of bitmap handles to the button. Normally, the buttons use the button control ID to automatically load bitmaps from the   resources of the user. If the bitmaps do not exist, the button caption is drawn into a default bitmap by using a lighter-weight version of the dialog font. To

use this message, you must first create three bitmap images of a single button:

- the button without keyboard focus
- the button with keyboard focus, but not pressed
- the button when it is "pressed" (or highlighted)

After creating the bitmaps, you must put the handles to these bitmaps into an array and pass a far pointer to this array in the IParam of the BM_SETBITS message.

Notifications inherited from standard Windows controls

BN_CLICKED — The button sends this message when it has been "pressed" by the user, either by clicking while the mouse pointer is within the button window or by either of the following keyboard actions:

- The user presses the Spacebar or the Enter key when the button has keyboard focus.
- The user presses the accelerator key for the the button when keyboard focus is in another control.

To associate an accelerator key with a button, place an ampersand before the ASCII value of the key in the text of the button (for example, "&Yes"). Note that case is not significant for button accelerators.

BN_DOUBLECLICKED — The button sends this message when it has been double-clicked by the user. The notification is sent at the time of the second mouse button-down message.

Notifications unique to BWCC

The following notifications are available if you have specified the BBS_PARENTNOTIFY style:

BBN_SETFOCUS — The button sends this notification to its parent window when it gains keyboard focus through an action other than a mouse click.

BBN_SETFOCUSMOUSE — The button sends this notification to its parent window when it gains keyboard focus through a mouse click.

BBN_GOTATAB — The button sends this notification to its parent window when the user presses the <Tab> key while keyboard focus is in the button. The parent can then intervene in the processing of the keystroke by returning a nonzero value.

BBN_GOTABTAB — The button sends this notification to its parent window when the user presses Shift-Tab (back-tab) while keyboard focus is in the button. The parent can then intervene in the processing of the keystroke by returning a nonzero value.

WM_DRAWITEM — If you specify the BBS_OWNERDRAW style for the button, it sends a WM_DRAWITEM message to its parent window. The IParam of the message contains a far pointer to a DRAWITEMSTRUCT structure. The fields of that structure are described in the Windows SDK documentation for this message, but with the following enhancement:

For Windows owner-draw buttons, the itemID field of the DRAWITEMSTRUCT structure is unused. Borland buttons use this field to pass their type. If the button is a default push button, this field contains the value BS_DEFPUSHBUTTON. Otherwise, it contains the value BS_PUSHBUTTON.

The other fields and the values passed in them are:

Field	Value
CtlType	ODT_BUTTON
CtlID	The control ID of the button (GetWindowWord(hWnd, WW_ID))

itemAction	ODA_DRAWENTIRE, unless the repaint is being caused by a focus change, in which case this field contains ODA_FOCUS	
itemState	The combination of the following values, depending on the current state of the button:	
ODS_FOCUS	if the button has keyboard focus	
ODS_DISABLED	if the button is disabled	
ODS_SELECTED	if the button is highlighted	
hwndItem	The window handle of the control	
hDC	A device context for the window, with all values in the default state returned by GetDC	
rcItem	The client rectangle of the control	

Button resource ID numbering scheme

The Microsoft resource compiler does not provide user-specified control initialization data when it parses the Windows dialog template data structure. Because of this, the control ID field is used as a base from which to derive the resource IDs of the bitmaps required by a button. For each bitmap button, there are six images: three for EGA and monochrome devices, and three for VGA and higher-resolution devices.

The bitmap resource IDs are derived from the button control using the following formulas:

Control ID + 1000	Normal VGA-resolution image
Control ID + 3000	Pressed VGA-resolution image
Control ID + 5000	Focused VGA-resolution image
Control ID + 2000	Normal EGA-resolution image
Control ID + 4000	Pressed EGA-resolution image
Control ID + 6000	Focused EGA-resolution image

BORBTN example

```
HBITMAP hBits[3];
HWND  hWndButton = GetDlgItem( hWnd, ID_FOO);

hBits[0] = MakeNormalBitmap(...);
hBits[1] = MakeHighlightBitmap(...);
hBits[2] = MakeFocusBitmap(...);

SendMessage( hWndButton, BBM_SETBITS, 0, (LONG) (LPSTR) hBits);
```

BORRADIO control

Function
Alternate radio button style

Class name
"borradio" (RADIO_CLASS)

Types inherited from standard Windows controls

BS_RADIOBUTTON	A nonautomatic radio button. The button merely informs the application program that it has been "checked" (pressed) via the BN_CLICKED notification. The application is responsible for calling the CheckRadioButton function to change the state of a button and the state of the other buttons it is grouped with.
BS_AUTORADIOBUTTON	An "automatic" radio button. When the user selects one of these buttons, it is automatically marked (with a circle or diamond), and the previously selected button within the group is deselected, without the intervention of the application program.

Options inherited from standard Windows controls

BS_LEFTTEXT	This option causes the text associated with the button to be displayed to the left of the button, rather than to the right of the button.

Options unique to BWCC

BBS_PARENTNOTIFY	This option causes the control to generate the following notification messages at run time:

- BBN_SETFOCUS
- BBN_SETFOCUSMOUSE
- BBN_GOTATAB
- BBN_GOTABTAB

BBS_OWNERDRAW	This option causes the control to send WM_DRAWITEM to its parent at run time, for specialized drawing.

Commands inherited from standard Windows controls

BM_GETCHECK	This message causes the button to return its current "check" state (the message names and descriptions all use check box imagery). If it is checked (pressed), it returns a nonzero value. Otherwise, it returns zero.
BM_SETCHECK	This message changes the check state of a button. If the wParam of the message is nonzero, the button is checked (filled with a circle or a diamond).
BM_GETSTATE	This message determines whether a button is highlighted, has focus, and whether it is checked. The low-order two bits (0x0003) of the return value contain the check state: 0 indicates unchecked and 1 indicates checked. The 0x0004 bit of the return value indicates that the button is highlighted (drawn with a heavy outline around the circle or diamond); the 0x0008 bit indicates that the button has the focus (a dotted line surrounds the text caption).
BM_SETSTATE	This message changes the highlight state of a button. If the wParam of the message is nonzero, the button is highlighted.

Notifications inherited from standard Windows controls

BN_CLICKED	The button sends this message when it has been "pressed" by the user, either by clicking while the mouse pointer is within the button window or by either of the following keyboard actions:

- The user presses the Spacebar or the Enter key when the button has keyboard focus.

▪ The user presses the accelerator key for the the button when keyboard focus is in another control.

> To associate an accelerator key with a button, place an ampersand before the ASCII value of the key in the text of the button (for example, "&Yes"). Note that case is not significant for button accelerators.

BN_DOUBLECLICKED The button sends this message when it has been double-clicked by the user. The notification is sent at the time of the second mouse button-down message.

Notifications unique to BWCC

The following notifications are sent to the parent window only if the programmer has specified the BBS_PARENTNOTIFY style.

- ▪ BBN_SETFOCUS
- ▪ BBN_SETFOCUSMOUSE
- ▪ BBN_GOTATAB
- ▪ BBN_GOTABTAB

WM_DRAWITEM This notification is identical to **BORBTN's,** with the following exception: For automatic radio buttons, the itemID field of the DRAWITEMSTRUCT structure contains the value BS_AUTORADIOBUTTON. Otherwise, it contains the value BS_RADIOBUTTON.

BORCHECK control

Function
Alternate check box style

Class name
"borcheck" (CHECK_CLASS)

Types inherited from standard Windows controls

BS_CHECKBOX	A nonautomatic check box. Application program intervention is required to change its visual state after it has been "clicked."
BS_AUTOCHECKBOX	A check box that automatically changes state when clicked.
BS_3STATE	A nonautomatic check box that switches between three states: checked, unchecked, and indeterminate.
BS_AUTO3STATE	An automatic version of BS_3STATE.

Options inherited from standard Windows controls

BS_LEFTTEXT	This option causes the text associated with the button to be displayed to the left of the button, rather than to the right of the button.

Options unique to BWCC

BBS_PARENTNOTIFY	This option causes the control to generate the following notification messages at run time:

- BBN_SETFOCUS
- BBN_SETFOCUSMOUSE
- BBN_GOTATAB
- BBN_GOTABTAB

BBS_OWNERDRAW	This option causes the control to send WM_DRAWITEM to its parent at run time, for specialized drawing.

Commands inherited from standard Windows controls

BM_GETCHECK	This message causes the control to return its current "check" state. The return value is 0 if the control is unchecked; 1 if checked; and 2 if indeterminate (applies only for 3-state check boxes).
BM_SETCHECK	This message changes the state of a check box. If the wParam of the message is 0, the check box is drawn empty; if 1, the check box is checked; and if 2, it is drawn with with a pattern indicating the indeterminate state.
BM_GETSTATE	This message determines whether a check box is highlighted, has focus, and whether it is checked. The low-order two bits (0x0003) of the return value contain the check state: 0 indicates unchecked; 1 indicates checked; and 2 indicates the indeterminate state for 3-state check boxes. The 0x0004 bit of the return value indicates that the check box is highlighted (drawn with a heavy outline); the 0x0008 bit indicates that the button has the focus (a dotted line surrounds the text caption).
BM_SETSTATE	This message changes the highlight state of a check box. If the wParam of the message is a nonzero value, the check box is highlighted.

Notifications inherited from standard Windows controls

BN_CLICKED	The button sends this message when it has been "pressed" by the user, either by clicking while the mouse pointer is within the button window or by either of the following keyboard actions:

- The user presses the Spacebar or the Enter key when the button has keyboard focus.

▪   The user presses the accelerator key for the the button when keyboard focus is in another control.

> To associate an accelerator key with a button, place an ampersand before the ASCII value of the key in the text of the button (for example, "&Yes"). Note that case is not significant for button accelerators.

BN_DOUBLECLICKED The button sends this message when it has been double-clicked by the user. The notification is sent at the time of the second mouse button-down message.

Notifications unique to BWCC

The following notifications are sent to the parent window only if the programmer has specified the BBS_PARENTNOTIFY style:

BBN_SETFOCUS

BBN_SETFOCUSMOUSE

BBN_GOTATAB

BBN_GOTABTAB

WM_DRAWITEM   This notification is identical to BORBTN's, with the following exception: For automatic check boxes, the itemID field of the DRAWITEMSTRUCT structure contains the value BS_AUTOCHECKBOX or BS_AUTO3STATE. Otherwise, it contains the value BS_CHECKBOX or BS_3STATE.

BORSHADE control

Function
Alternate panel and divider styles

Class name
"borshade" (SHADE_CLASS)

Types unique to BWCC

BSS_GROUP	This style draws a "chiseled" gray box with a recessed appearance.
BSS_RGROUP	This style draws a "chiseled" gray box with a raised appearance.
BSS_HDIP	This style draws a horizontal dividing line that can be used to separate sections of a dialog box.
BSS_VDIP	This style draws a vertical dividing line that can be used to separate sections of a dialog box.
BSS_HBUMP	This style draws a horizontal dividing line that can be used to separate sections of a gray group shade (BSS_GROUP or BSS_RGROUP).
BSS_VBUMP	This style draws a vertical dividing line that can be used to separate sections of a gray group shade (BSS_GROUP or BSS_RGROUP).

Options unique to BWCC

BSS_CAPTION   This option applies only to the BSS_GROUP and BSS_RGROUP types. It causes the caption of the group shade box (if any) to be appear above the recessed (or raised) portion of the box. The dimensions of the box include the caption as well as the box.

BSS_CTLCOLOR   This option applies only to the BSS_GROUP and BSS_RGROUP types. It causes the control to send registered messages to its parent prior to erasing. The parent can then provide a different brush for painting the group box background, and make other changes to the HDC as needed. To use this mechanism, you must first register a special message using the Windows RegisterWindowMessage() API. In the file BWCC.H you will find the following definition:

```
#define BWCC_CtlColor_Shade "BWCC_CtlColor_Shade"
```

Include the following static declaration in your program:

```
WORD hCtlColor_Shade;
```

Then, in your application initialization function, register the message:

```
hCtlColor_Shade=RegisterWindowMessage(BWCC_CtlColor_Shade);
```

In your window procedure, dialog box window procedure, or most commonly your dialog procedure, test for the message:

```
if (msg == hCtlColor_Shade)
{
    ...
}
```

The parameters for the message are the same as for WM_CTLCOLOR, and the message is handled in the same manner. For example, the text foreground and background colors and the background mode in the HDC may be modified, in order to change the appearance of the caption. A background brush may be also returned. (As with normal WM_CTLCOLOR handling, be sure not to create a new brush every time the message is processed.)

In order to return a brush from a dialog procedure (as opposed to from a dialog box window procedure or a window procedure), you must place the value of the brush into offset DWL_MSGRESULT in the window structure with SetWindowLong() and then return TRUE. Here is an example:

```
if (msg == hCtlColor_Shade)
{
  SetTextColor( (HDC) wParam, RGB(255,0,0) ); // red text
  SetBkColor( (HDC) wParam, RGB(128,128,128) ); // gray
  SetBkMode ( (HDC) wParam, OPAQUE);
  SetWindowLong( hwndDlg, DWL_MSGRESULT,
  GetStockObject(WHITE_BRUSH) );
  return TRUE;
}
```

The Windows include files provide a macro that combines the last two steps: `SetDlgMsgResult(hwnd, msg, result)`, which you would use with `hCtlColor_Shade` as the second parameter.

BSS_NOPREFIX   This option applies only to the BSS_GROUP and BSS_RGROUP types, and is the equivalent of the SS_NOPREFIX option for static text: it causes any ampersands (&) within the caption to be treated as normal characters, rather than causing the next character to be underlined.

BSS_LEFT, BSS_CENTER, BSS_RIGHT

These options apply only to the BSS_GROUP and BSS_RGROUP types, and control the horizontal placement of the caption.

Commands unique to BWCC
`RegisterWindowMessage(BWCC_CtlColor_Shade)`

BORSTATIC control

Function
Static text with a gray background

Class name
"borstatic" (STATIC_CLASS)

Types inherited from standard Windows controls

SS_LEFT	The text is left-justified in the control.
SS_RIGHT	The text is right-justified in the control.
SS_CENTER	The text is center-justified in the control.
SS_SIMPLE	The text is left-justified in a single line within the control and does not word wrap.
SS_LEFTNOWORDWRAP	The text is left-justified within the control and does not word wrap.

Options inherited from standard Windows controls

SS_NOPREFIX	Ampersands (&) within the text do not cause the following character to be underlined.

BORDLG dialog class

Function
"Turbo" fast dialog box drawing

Class name
"bordlg" (BORDLGCLASS)

Description
This custom dialog window class implements the "turbo painting" of Borland custom controls by keeping its own private list of controls within a dialog box and painting those controls itself. It also automatically provides a patterned background on VGA and higher-resolution displays. If you want your dialogs to have the "Borland look," specify this dialog class in your dialog box template. (As an alternative to specifying "bordlg" as the class, you may also call `BWCCDefDlgProc()`.)

Types inherited from standard Windows controls
All valid styles for a standard Windows dialog box.

Commands inherited from standard Windows controls

WM_CTLCOLOR	If the user has provided a dialog procedure, it is called with the WM_CTLCOLOR message. If it returns a non-zero value, then no further processing takes place, and that value is returned. Otherwise, the processing depends on which CTCOLOR value is specified. For list boxes, the background is set to a gray brush. For static and button controls, the background mode is set to transparent; the text color to COLOR_WINDOWTEXT; for non-monochrome monitors, the background color is set to COLOR_GRAYTEXT; and a gray background brush is returned.
	For CTLCOLOR_DLG, the steel-gray dialog background brush is returned, but it is first unrealized and the origin of the HDC is reset to match the dialog box.
	For other CTLCOLOR values, DefWindowProc() is called and its value returned.
WM_NCCREATE	This message sets up a structure, which is attached as a property to the dialog window. As Borland controls are then created, they will register themselves with the dialog window, and information about each control will be added to this structure. This is the mechanism used to provide turbo-painting.
	After attaching the structure, WM_NCCREATE calls DefDlgProc() and returns its value.
WM_ERASEBKGND	This message first sends a WM_CTLCOLOR message with CTLCOLOR_DLG to the dialog procedure of the user (if any) to get a background brush for the dialog. If zero is returned, the chiseled-steel brush is used. But before painting the background, the control structure is iterated and any Borland group shades and Borland static text controls are painted with a gray background (for speed). (Note, however, that the brush used for group shades may be modified by an additional CTLCOLOR-like message, as described in the BORSHADE section.)
	The background brush is realigned with the top left corner of the dialog window and the dialog background is painted with it, excluding any rectangles that were painted for group shades and static text controls. Finally, WM_ERASEBKGND returns TRUE, to indicate to Windows that no further erasing is necessary.
WM_PAINT	This message iterates through the control structure described above and paints each of the Borland controls. For each control that is painted, its window is validated, so that it will not itself get WM_PAINT or WM_ERASE messages.
	After all Borland controls are painted, a thin frame is drawn around the dialog to provide a sense of depth, and zero is returned.

WM_DESTROY   This message simply frees the control list attached to the dialog window and then calls `DefDlgProc()`, returning its value.

Using BWCC controls in non-dialog windows

If you want your non-dialog windows to look like the BorDlg windows (with the steel-gray background and light gray background for static controls), BWCC.DLL provides two functions that replace the Windows standard "Def" window functions and that should be called in place of them:

▪ For MDI child windows, call `BWCCDefMDIChildProc` instead of the Windows standard function `DefMDIChildProc`.

▪ For all other windows, call `BWCCDefWindowProc` instead of the Windows standard function `DefWindowProc`.

As described earlier for `BWCCDefDlgProc`, your window proc must call either `BWCCDefMDIChildProc` or `BWCCDefWindowProc` for the following messages:

▪ WM_CTLCOLOR
▪ WM_NCCREATE
▪ WM_NCDESTROY
▪ WM_PAINT
▪ WM_ERASEBKGND

Note: BWCC does not provide a replacement function for `DefFrameProc`.

Defining a derivative dialog class

To create your own dialog window class (for example, if you want the dialog box to have its own icon), you must "derive" your class from the BORDLG class.

To derive a class from BORDLG,

1. Your dialog window function should call `BWCCDefDlgProc`, note the Windows standard `DefDlgProc` for messages that it does not process.

2. The window proc must call `BWCCDefDlgProc` for the following messages:

- WM_CTLCOLOR
- WM_NCCREATE
- WM_PAINT
- WM_ERASEBKGND
- WM_NCDESTROY

Designing Borland custom control dialog boxes

The following topics present style considerations you can follow when designing Borland Windows Custom Control (BWCC) dialog boxes for your Windows-based software.

Panels

Fonts

Group boxes

Push buttons

Examining your dialog box

Panels

Each dialog box has two panels: a Main panel and an Action panel. The Main panel should contain all the required controls. The Action panel should contain the push buttons.

Your finished dialog box should be relatively square. If the Main panel is wider than it is tall, put the Action panel along the bottom of your dialog box. If the Main panel is taller than it is wide, put the Action panel on the right side.

Main panel

You can arrange the group boxes on the Main panel in either a single column or row, or in an array. Here are some guidelines for arranging group boxes on the Main panel. You should treat group titles as part of the group boxes.

- Space group boxes 8 <u>dialog units</u> apart, both vertically and horizontally.
- Leave a margin of 8 dialog units from all edges of the dialog to the nearest group box.
- In a column of group boxes, make all group boxes the same width. The width should accommodate the widest item or title. Widen the other group boxes to match.
- In a row of group boxes, vary the group box heights. Align the tops of the group boxes and let the bottoms of the group boxes vary.
- If some of the group boxes in a row have titles and some do not, align the top of the recessed group boxes with each other, not with the title rectangles. For these "mixed" groups of boxes, the margin above group boxes without titles should include the space for a title.
- If some of the group boxes you want to align in a row are taller than others, compute the bottom margin using the tallest group box.

Action panel

An Action panel can appear at the bottom or the right side of a dialog box. Here are the guidelines for Action panels:

▪   Make the Action panel tall or wide enough to contain the push buttons while leaving a margin of 8 dialog units above and below or to the sides of the push buttons.

▪   Distribute the push buttons evenly along the Action panel, leaving a minimum of 8 dialog units between the buttons and between the buttons and the edges of the dialog box. Try to use the same number of dialog units between each button and between the buttons and the edges of the dialog box. You can put more space between the buttons than between the buttons and the edges of the dialog box, if necessary, but the two margin spaces should be equal and the spaces between the buttons should be equal.

Fonts

Borland dialog boxes use 8-point Helvetica Bold. The Borland Windows custom dialog controls look best when you use this font. An 8-point font is small; using it prevents your dialog boxes from growing too big. Of course, you can use other fonts for other custom controls.

Group boxes

Collect all options that appear in the Main panel into Borland Windows custom group boxes. For example, place a group of related check boxes in a group box. You should place each single control, such as a file name text box or combo box, in a group box also. You will not have to do this with a Borland list box because a list box draws its own group box.

Group box titles

A group box title identifies what a group box contains. By default, a group box title in a Borland dialog box has a gray background. Here are guidelines for using group box titles:

- If a group box contains multiple controls, place the group box title above and touching the top edge of the group box.
- If a group box contains a single check box, place the group box title above and touching the top edge of the group box.
- If a group box contains a single text box or combo box control, you can either put the title to the left of the control and 4 dialog units from the edge of the group box or you can put it above the control.
- If a group box contains two or more editable text fields or combo boxes or both, precede each with a short label.
- Align group box titles above the recessed group boxes.
- Make all group box titles 9 dialog units high.
- Make the titles the same width as the group boxes, including the beveled sides.

Group box elements

These suggestions help you arrange elements within a group box:

- Distribute controls within a group box vertically every 13 units from the bottom of one line of text to the bottom of the next.
- Left-justify the controls.
- The left and right margins between the edges of the group box and the widest control within it should be 4 dialog units wide.
- Make the margin between the top of the group box and the first control in the group 4 dialog units.
- Make the margin between the bottom of the group box and the last control in the group 4 dialog units.
- If a group box contains two or more editable text fields or combo boxes or both, make them the same width. Space them so that the bottom of one is 13 units from the bottom of the next one. Right-justify these controls in the group box 4 units from the right edge. Left-justify the titles, leaving a 4 unit margin. Make the group box wide enough to leave 4 units between the longest title and its control.

Push buttons

The following are style considerations for push buttons:

- The Borland custom push buttons use glyphs (small bitmapped images). For example, a question-mark glyph is used on the Help push button. Place the glyph inside the button on the left side.
- Use Helvetica (normal, not bold) for the text of a button text and right-justify it.

Action panel push buttons

Action panel push buttons usually indicate the end of the user's work with a dialog box, but can also serve as a major departure from the function of the dialog box, such as bringing up Help with the Help button. The guidelines for these buttons are:

- Put the buttons on the Main panel rather than the Action panel.
- Do not put Action panel push buttons in a group box. Place them directly on the surface of the Main panel.
- Make all push buttons in a group the same width. They should be just wide enough to accommodate the widest text string.
- Make the buttons 14 dialog units in height.
- Try to restrict text to 20 characters or less.
- Place the buttons in either a row or a column, depending on what looks best in your dialog box.
- Leave 8 dialog units to the left and right of a column of push buttons. The vertical space between the buttons and any other controls or borders above or below the buttons should be equal.
- Leave 8 dialog units above and below a row of push buttons. The horizontal space between the buttons and any other controls or borders to the left or right of them should be equal.

Examining your dialog box

When Windows calculates dialog units, it rounds the computation. Rounding errors can affect the appearance of your dialog box. Examine your dialog box carefully and look for these problems:

- A crack between the title text and the top of a gray group box
- Obvious uneven spacing in a vertical group of radio buttons or check boxes
- An inconsistent border width in exposed panel areas

Usually, making an adjustment of 1 dialog unit will fix these problems. Occasionally in a large group of repeating controls, two or more rounding errors can occur. You cannot tell how text in controls will appear when you are designing your dialog box. Editable text, large static text fields, and combo boxes fall into this category. You may have to modify your original design to be sure text appears correctly without being clipped at run time.

Creating custom control classes

Windows provides standard control classes, such as list boxes and radio buttons, that you can add to your dialog box resources. In addition to these standard classes, you can create and use custom control classes, which must be in a DLL (dynamic-link library). The following topics describe the functions you will need to use to make your custom controls accessible.

- ListClasses function
- Info function
- Style function
- Flags function

The DLL file of custom controls must contain functions that let Resource Workshop work with the custom controls just as it works with the standard Windows controls. In particular, you must implement the `ListClasses` function and export it by name. This function provides information about the custom control classes in the DLL.

You must also provide the following functions for each custom control window class:

- Info
- Style
- Flags

These functions can have any name. They must, however, be exported by the DLL, and pointers to them must be supplied in the `ListClasses` function.

ListClasses function

ListClasses is a programmer-implemented function that passes information about the custom control classes back to Resource Workshop. Exporting ListClasses marks your DLL as supporting this custom control specification.

If ListClasses is present in the DLL, the function is called, passing information about itself along with two utility function variables used in editing the custom control.

ListClasses should return a handle to global memory allocated by calling GlobalAlloc. The memory referenced by this handle holds a structure of type CTLCLASSLIST, which describes the controls in the library. CTLCLASSLIST is described later in this section. The handle is freed by Resource Workshop and should not be freed by the DLL.

Syntax

```
HGLOBAL CALLBACK ListClasses( LPSTR szAppClass, UINT wVersion, LPFNLOADRES
  fnLoad, LPFNEDITRES fnEdit);
```

Return value

Returns a global handle to the data structure.

Parameters

szAppClass   The class name of the application's main window. The class name can be used by the custom control to determine if it is running under a resource editor. If szAppClass is "rwswnd", the calling application is Resource Workshop.

wVersion   The version number of the calling application. The major version is in the high-order byte and the minor version in the low-order byte. For example, version 1.02 is 0x0102.

fnLoad   A pointer to a the function that a custom control can use to get a binary version of any resource in the project being edited by the calling application--the equivalent of the Windows API function LoadResource(). The function takes two parameters: a resource type name and a resource name. The custom control must free the global handle (if any) returned by the function.

fnEdit   A pointer to a function that a custom control can use to start a resource editor for any resource in the project. It takes two parameters: a resource type name and a resource name.

Data structures

```
typedef struct
{
  LPFNINFO  fnRWInfo;        // Info function
  LPFNSTYLE fnRWStyle;       // Style function
  LPFNFLAGS fnFlags;         // Flags function
  char  szClass[ CTLCLASS];  // Class name

} RWCTLCLASS, FAR *LPRWCTLCLASS;

typedef struct {
  short  nClasses;           // Number of classes in list
  RWCTLCLASS Classes[];      // Class list
} CTLCLASSLIST, FAR *LPCTLCLASSLIST;
```

The CTLCLASSLIST structure contains a variable number of RWCTLCLASS strucures, the number of which is determined by the nClasses field.

Each control class in the DLL must have a corresponding RWCTLCLASS structure in the

CTLCLASSLIST. The szClass field contains the name with which the class was registered. For example, if you called RegisterClass giving the class name "MYBUTTON", szClass must be "MYBUTTON".

The function variables Info, Style, and Flags--which correspond to the pointers fnRWInfo, fnRWStyle, and fnFlags--are described in the following sections.

Info function

The Info function is called to retrieve information about the control class, including the string to add to the control menu and the bitmap to add to the tool palette. The function returns a memory handle that can be allocated by GlobalAlloc. This handle must refer to memory that contains a RWCTLINFO structure. Like ListClasses, the handle returned by Info is freed by Resource Workshop and should not be freed by the DLL. Resource Workshop calls this function once when it loads the DLL.

Syntax
```
HGLOBAL CALLBACK Info( void);
```

Parameters
None.

Data structures

The RWCTLINFO structure, defined by a typedef in the file CUSTCNTL.H, has two basic parts:

▪ The first part has a fixed length and provides information about the whole control class.
▪ The second part is a variable-length array of fixed-length records. Each record provides information about a particular type or subclass of the control.

```
/* general size definitions */
#define  CTLTYPES12  /* number of control types*/
#define  CTLDESCR22  /* size of control menu name */
#define  CTLCLASS20  /* max size of class name */
#define  CTLTITLE94  /* max size of control text  */

typedef struct {
  UINTwVersion;             // control version
  UINTwCtlTypes;            // control types
  charszClass[CTLCLASS];    // control class name
  charszTitle[CTLTITLE];    // control title
  charszReserved[10];       // reserved for future
  RWCTLTYPE Type[CTLTYPES]; // control type list
} RWCTLINFO;

 typedef RWCTLINFO *RWPCTLINFO;
 typedef RWCTLINFO FAR *LPRWCTLINFO;
```

wVersion	The version number of the custom control library. The major version is in the high- order byte and the minor version is in the low-order byte. For example, version 1.02 is 0x0102. Resource Workshop does not use this.
wCtlTypes	The number of control sub-types defined in the Type array.
szClass	The name of the class as registered with Windows. This is duplicated from the CTLCLASSLIST structure to retain upward compatiblity with the Windows custom control specificiation.
szReserved	Space reserved for future expansion. Must be cleared to null characters (0).
Type	An array of sub-type description structures of type RWCTLTYPE.

```
/*
 * RWCTLTYPE DATA STRUCTURE
 *
 * This data structure is returned by the control options
 * function while inquiring about the capabilities of a
 * particular control. Each control may contain various types
 * (with predefined style bits) under one general class.
 *
```

```
                 * The width and height fields provide the application with
                 * a suggested size. Use pixels or dialog units for the
                 * values in these fields. If you use pixels, turn on the
                 * most significant bit (MSB). If you use dialog units, turn
                 * off the MSB.
                 *
                 */

            typedef struct {
              UINT wType;                  // type style
              UINT wWidth;                 // suggested width
              UINT wHeight;                // suggested height
              DWORDdwStyle;                // default style
              char szDescr[CTLDESCR];      // menu name
              HBITMAP  hToolBit;           // Toolbox bitmap
              HCURSOR  hDropCurs;          // Drag and drop cursor
            } RWCTLTYPE, FAR * LPRWCTLTYPE;
```

wType	A user-defined value used to indicate the sub-type of the control. This value is not used by Resource Workshop.
wWidth	The default width for the control. Resource Workshop uses this value if, for example, the control is created using Control palette tool. wWidth is in dialog coordinates unless the most significant bit is set, in which case the value is in pixels. For example, a value of "32" is 32 in dialog coordinates, but the value "32 | 0x8000" is in pixels.
wHeight	The default height for the control. Resource Workshop uses this value if, for example, the control is created using the Control palette tool. wHeight is in dialog coordinates unless the most significant bit is set, in which case the value is in pixels. For example, a value of "32" is 32 in dialog coordinates, but the value "32 | 0x8000" is in pixels.
wStyle	The default style Resource Workshop uses to create the Window. This is the key field that you use to distinguish one subtype from another.
szDescr	The description of the control subtype. Resource Workshop uses the text to construct a menu item that the user can use to create an instance of your custom control.
hToolBit	A handle to a bitmap which will be placed on the tool palette. Resource Workshop requires the bitmap be a 22x22 black and gray bitmap containing a 2-pixel border that is white on the top and left and black on the bottom and right. You can use the bitmaps contained in BITBTN.RES as templates.
hDropCurs	A cursor to be used while dragging the control from the tool palette.

Style function

The Style function makes it possible for you to edit your custom control. You must first create an appropriate dialog box in the Dialog editor and then implement a Boolean function that displays that dialog box. This function is called whenever you initiate a request to edit the custom control. The function passes a handle to the window that is the parent of the dialog, a handle to memory containing the RWCTLSTYLE structure, and two function variables for string conversion.

Syntax

```
BOOL CALLBACK Style( HWND hWnd, HGLOBAL hCtlStyle, LPFNSTRTOID lpfnStrToId,
  LPFNIDTOSTR lpfnIdToStr);
```

Return value

If the user changes any options for the control, this function's return value is TRUE. If the user does not make changes or if an error prevents changes, the return value is FALSE.

Parameters

hWnd	A handle to the parent window of the dialog box displayed by this function.
hCtlStyle	A handle to global memory containing the RWCTLSTYLE structure to be edited.
lpfnStrToId	A function variable that converts a string into a control ID for the wId field of RWCTLSTYLE. This lets the user enter the control ID using a constant identifier. This routine evaluates the string as an expression, returning the result. The ID can convert back into a string by calling lpfnIdToStr.
lpfnIdToStr	A function variable that converts the control ID in the wId field of RWCTLSTYLE to a string for editing. The ID can be converted back into a word by calling lpfnStrToId. This function variable lets the user see the symbolic constant that represents the control ID instead of the word value.

Data structures

```
/*
 * CONTROL-STYLE DATA STRUCTURE
 *
 * The class style function uses this data structure
 * to set or reset various control attributes.
 *
 */

typedef struct {
  UINT wX;                      // x origin of control
  UINT wY;                      // y origin of control
  UINT wCx;                     // width of control
  UINT wCy;                     // height of control
  UINT wId;                     // control child id
  DWORDdwStyle;                 // control style
  char szClass[CTLCLASS];       // control class name
  char szTitle[CTLTITLE];       // control text
  BYTE CtlDataSize;             // control data size
  BYTE CtlData[ CTLDATALENGTH]; // control data
} RWCTLSTYLE;

typedef RWCTLSTYLE *    PRWCTLSTYLE;
typedef RWCTLSTYLE FAR *   LPRWCTLSTYLE;
```

wX	The horizontal (X) location of the control in dialog coordinates.

wY	The vertical (Y) location of the control in dialog coordinates.
wCx	The width of the control (dialog coordinates).
wCy	The height of the control (dialog coordinates).
wId	The control's ID value. This value must be converted to a string by calling lpfnIdToStr before being displayed for editing. It must be converted back into a word for storage by calling lpfnStrToId after editing.
dwStyle	The style flags of the control.
szClass	The class name of the control.
szTitle	The title of the control.
CtlDataSize	Windows lets controls in a resource file have up to 255 bytes of control-defined data. This field indicates how much of that space is being used by the control. The data is stored in CtlData.
CtlData	This field holds up to 255 bytes of control-specific data. The amount used must be recorded in the CtlDataSize field. The use of this data area is user-defined.

When you save your project, the CtlData array is saved into the .RC or .RES file.

To enable a custom control to access this array from within your program at run time, lParam of the WM_CREATE message points to a CREATESTRUCT data structure. The CREATESTRUCT structure contains a field, lpCreateParams, that is a pointer to the extra data you stored in the CtlData array. If the pointer is NULL, there is no CtlData.

The CtlDataSize variable is not available to your program. To make the size data accessible to your program, the CtlData array should either contain a fixed amount of data, or its first byte should contain the length of the data.

The Style function first converts the ID to a string by passing the numerical ID value to LPFNIDTOSTR. The Style function then displays the string in the dialog box.

If the user changes the string returned by LPFNIDTOSTR, the Style function verifies the string by passing it to LPFNSTRTOID, which determines if the string is a valid constant expression. If LPFNSTRTOID returns a zero in the LOWORD, the ID is illegal and is displayed in the dialog box, so the user can change it to a valid ID. If LPFNSTRTOID is successful, it returns a nonzero value in the LOWORD and the ID in the HIWORD.

Flags function

The Flags function is used to translate the style of a control into text. Resource Workshop inserts the text into the .RC file being edited. The function must only convert the values unique to the control. For example, if you were creating a Flags function for the Windows button class, you would only examine the lower sixteen bits of Flags and translate them into one of the bs_XXXX constants.

Syntax

```
UINT CALLBACK Flags(DWORD dwFlags, LPSTR lpStyle, UINT wMaxString);
```

Return value

Returns the number of bytes copied into the destination string. Returns 0 if the Flags word is not valid or the string exceeds MaxString in length.

Parameters

dwFlags	The control style to be translated into text. This field is derived from the dwStyle field of the RWCTLSTYLE structure passed to the Style function variable.
lpStyle	The location to write the translated text.
wMaxString	The maximum number of bytes the Flags function can write into Style.