A Truly Usable Message Box

How to overcome problems with the built-in message boxes and learn lots of useful techniques along the way...

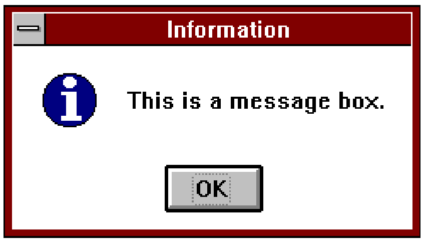
by Steven J Colagiovanni

Just about All Windows programs use message boxes and Borland has provided us with essentially two function calls for displaying them: Application.MessageBox and MessageDlg. Both are single line calls specifying the message you want to display, a glyph and the buttons you want available to the user. With these function calls, Borland has provided us with the ability to create message boxes that promote code reusability and customizability.

Application.MessageBox is actually a wrapper for the Windows MessageBox API call which makes life easier.

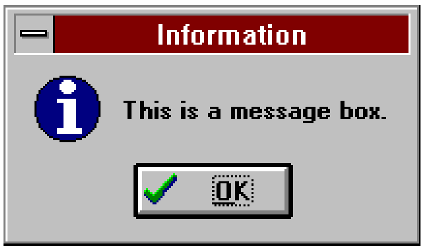
The MessageDlg function creates an attractive message box with a three dimensional look. This and three related functions (MessageDlgPos, ShowMessage and ShowMessagePos) are contained in the Dialogs unit and dynamically create and destroy a TForm object at runtime, placing the appropriate glyph, buttons and message on the form. MessageDlg provides several features not available in Application.MessageBox. You can specify any combination of buttons and allow your user to access a help file. Using the MessageDlgPos function, you can also specify the screen position of the message box. It's interesting that Borland made MessageDlg a function, not an object or a component. This makes it more versatile and allows it to be used within DLLs.

With all of its benefits, MessageDlg has many shortcomings. It doesn't allow us to specify a caption for the message box and the default captions (Warning, Error, Information, Confirm and the name of your executable file) are, to be nice, poor. The captions are determined by the type of message box created. Also, the system menu on the dialog contains the Close item (the dialog for Application.MessageBox does not). Close on the system menu passes mrCancel as the result. If your MessageDlg displays Yes and No buttons and you are only checking for a result of mrYes and mrNo this could cause problems.



This is a message box.

➤ *Figure 1: Application.MessageBox*



This is a message box.

➤ *Figure 2: MessageDlg*

Unfortunately, Application.MessageBox has some major shortcomings also. When our application is used under Windows 3.1x, the 3D look that we worked so hard on displays two dimensional message boxes with a white background. This hurts the continuity of our application and looks poor.

Figures 1 and 2 show the message boxes created by Application.MessageBox and MessageDlg respectively, as they appear in Windows 3.1x. Windows 95 does not produce this problem because all windows, including message boxes, are created with a 3D appearance.

Having to pass the message to Application.MessageBox as a PChar produces extra work. Simple messages like 'Are you certain you want to close the application?' can be passed as a literal value (Delphi knows how to automatically translate strings that appear in quotes into PChars). However, if you want to pass a message like

```
'Are you certain you want to '+
'delete file: '+varFileName+'?'
```

then you would have to declare a PChar variable, allocate memory for it, copy the text into the PChar, and destroy the memory allocated after the PChar variable is passed to Application.MessageBox. This is a lot of hassle just to be able to insert the name of the file about to be deleted into a message box. Also, Application.MessageBox has only six possible combinations of buttons.

There are some advantages to Application.MessageBox. Because the message is a PChar we can have messages that are longer than 255 characters. However, large message boxes like this are usually not desirable. If you need a dialog that displays only a Retry and a Cancel button, Application.MessageBox puts the Retry button first, but MessageDlg puts the Cancel button first. A program using MessageDlg would not be consistent with other Windows applications. Application.MessageBox allows you to designate which button will have the focus and be the default. Windows File Manager uses a message box that asks the user to verify that they want to delete a file, displaying Yes and No buttons, and the No button is the default. Using MessageDlg, the Yes button would be the default. This doesn't protect the user if they accidentally press the Enter key.

What we need is a message box function that has all the features of both Application.MessageBox and MessageDlg without any of their flaws. Unfortunately, the only way to do this is to create an entirely new set of functions. If you have the Delphi source code, you could re-write or modify the Dialogs unit source code, but this is not recommended. If your modifications create an error that causes a bug, or worse, a GPF, you're in trouble. It is better to create a new set of functions and use slightly different names to avoid possible conflicts. You may still need to include the Dialogs unit in your application, because it also contains the wrapper functions for the Windows common dialogs (Open File, Save File, Printer, etc).

The CreateMsgBox Function

Our new set of functions will call a CreateMsgBox function, which will contain the majority of the code which will dynamically create a TForm object and place the appropriate glyph, buttons and message (contained in a TLabel component) on the form. CreateMsgBox and its calling functions will use several enumerated types that need to be declared in the interface section. To make using these functions easier, we will also declare constants for the button combinations that are used most often. These custom types and constants, some of which are similar to those used by MessageDlg, are defined in Listing 1.

If you have the VCL source code, you can compare the TMsgBoxType in Listing 1 with the TMsgDlgType used for the MessageDlg function (in DIALOGS.PAS). I have placed the mbCancel button between mbAll and mbHelp. This will allow our functions to display the Cancel button last when our dialog box displays Retry and Cancel buttons. You may have noticed that I added constant declarations for mbYesNo and mbRetryCancel, which do not exist for Borland's MessageDlg function.

In the implementation section are several arrays used when placing the glyph, message and buttons on the form. These arrays are defined in Listing 2. The CreateMsgBox function also uses several constants when constructing the form and all the components we put on it. The complete listing for CreateMsgBox is shown in Listing 3. Please refer to this as you follow the discussion below.

The first thing CreateMsgBox does is dynamically create a TForm and set some of the properties. Setting Font.Height := -13 will allow precise sizing of the font, and will be approximately equal to 10 point. The Windows MessageBox API function uses 10 point, MS Sans Serif font so these settings should allow our message box to look similar.

Creating The Message

Message boxes created with Application.MessageBox or MessageDlg vary in size depending on how much text is displayed in the message. The Windows API call DrawText is used to determine how large an area is required to display the text, then we size the message box around this. The DT_WORDBREAK value allows DrawText to automatically wrap the text (if necessary), while the DT_CALCRECT value will expand the bottom of the rectangle as needed to display all the text. DT_NOPREFIX is included so that we can use an ampersand (&) sign in our message. It serves the same purpose as TLabel's ShowAccelChar property. One of the parameters passed in DrawText is a TRect, the values of which are the boundaries of a rectangle the text will be placed in. A rectangle half the width of the screen should be sufficient, while keeping the message box from filling up the entire screen. We can set the height to 0, because when DrawText is done the size of TextRect will reflect the actual height and width required to display all the text.

The function then creates a TLabel component that will be used to display the message. If the text was drawn on the form's canvas using DrawText, we would have to redraw the text every time the form requires repainting. Assigning the message to the label's Caption property will handle the repainting of the text for us. The label's parent is set to Result (the form) so the label is placed on the form. The label's BoundsRect property is used to set the width and height equal to TextRect. The label's ShowAccelChar property is set to False so that we can use an ampersand.

Creating The Glyph

The next step is to create and place the glyph or icon. The Windows

➤ *Listing 1*

```
type
  TMsgBoxType = (mtWarning, mtError, mtInformation,
                 mtConfirmation, mtCustom);
  TMsgBoxBtn = (mbYes, mbNo, mbOK, mbAbort, mbRetry,
                mbIgnore, mbAll, mbCancel, mbHelp);
  TMsgBoxButtons = set of TMsgBoxBtn;
  TDefaultBtn = (dfFirst, dfSecond, dfThird);     { Default Button }
const
  mbYesNo = [mbYes, mbNo];
  mbYesNoCancel = [mbYes, mbNo, mbCancel];
  mbOKCancel = [mbOK, mbCancel];
  mbRetryCancel = [mbRetry, mbCancel];
  mbAbortRetryIgnore = [mbAbort, mbRetry, mbIgnore];
```

➤ *Listing 2*

```
const
  { Identifiers for MessageBox Icons - stored in video driver }
  ResIDs: array[TMsgBoxType] of PChar =
    (IDI_EXCLAMATION, IDI_HAND, IDI_ASTERISK, IDI_QUESTION, nil);
  { Button.Caption - strings stored in Windows, load on call }
  BtnCaptions: array[TMsgBoxBtn] of Word =
    (SMsgdlgYes, SMsgdlgNo, SMsgdlgOK, SMsgDlgAbort, SMsgDlgRetry,
     SMsgDlgIgnore, SMsgDlgAll, SMsgdlgCancel, SMsgdlgHelp);
  { Button.Name }
  BtnNames: array[TMsgBoxBtn] of PChar = ('btnYes', 'btnNo', 'btnOK',
   'btnAbort', 'btnRetry', 'btnIgnore', 'btnAll', 'btnCancel', btnHelp');
  { Button.Result }
  BtnResult: array[TMsgBoxBtn] of TModalResult = (mrYes, mrNo, mrOK,
    mrAbort, mrRetry, mrIgnore, mrAll, mrCancel, mrNone);
```

video driver (eg VGA.DRV) contains the icons displayed in the Windows message boxes. The glyphs are stored in the video driver because the size of the glyphs can vary depending on the resolution and design of the driver. This is why the glyphs change when your program is run under Windows 95. By loading one of these icons into memory and placing it on the form, we will always display a properly sized and colored glyph. The constant definitions for these icons are stored in the `ResIDs` array. If `TMsgBoxType` is `mtCustom` then `ResIDs` equals `nil` and no glyph will be displayed. We will use a `TImage` component to place the glyph and, like the label, `TImage` will handle repainting. To place the glyph in the `TImage`

➤ *Listing 3*

```
function CreateMsgBox(const AMsg: string; const ACaption:
  string; AType: TMsgBoxType; AButtons: TMsgBoxButtons;
  ADefaultButton: TDefaultBtn): TForm;
const
  MsgDlgMinWidth = 150;
  MsgDlgMinHeight = 55;
  MsgDlgBtnSize: TPoint = (X: 77; Y: 28);
  { mgTextMargin: Top, left, right margins around text & glyph }
  mgTextMargin = 10;
  mgGlyphSpacing = 15;   { Spacing between glyph and text }
  mgBtnLRMargin = 15;
  mgBtnTopMargin = 20;
  mgBtnBottomMargin = 8;
  mgButtonSpacing = 8;
var
  MsgLabel: TLabel;
  Glyph: TImage;
  FIcon: TIcon;
  Buttons: array[TMsgBoxBtn] of TButton;
  Btn: TMsgBoxBtn;
  ButtonCount: Integer;
  ButtonSize: TPoint;
  InfoSize: TPoint;
  TextRect: TRect;
  C: array[0..255] of Char;
  ButtonX: Integer;
  ButtonTop: Integer;
  function Max(v1, v2: Integer): Integer;
  begin
    if v2 > v1 then Result := v2
    else Result := v1;
  end;
begin
  Result := TForm.CreateNew(Application);
  with Result do begin
    PixelsPerInch := 96;
    BorderStyle := bsDialog;
    BorderIcons := [biSystemMenu];
    Ctl3D := True;
    Font.Name := 'MS Sans Serif';
    Font.Height := -13;
    Font.Style := [fsBold];
    TextRect := Rect(0, 0, Screen.Width div 2, 0);
    DrawText(Canvas.Handle, StrPCopy(C, AMsg), -1, TextRect,
      DT_CALCRECT or DT_WORDBREAK or DT_NOPREFIX);
    { create the text }
    MsgLabel := TLabel.Create(Result);
    MsgLabel.Name := 'Message';
    MsgLabel.Parent := Result;
    MsgLabel.WordWrap := True;
    MsgLabel.ShowAccelChar := False;
    MsgLabel.Caption := AMsg;
    MsgLabel.BoundsRect := TextRect;
    if ResIDs[AType] <> nil then begin
      Glyph := TImage.Create(Result);
      Glyph.Name := 'Image';
      Glyph.Parent := Result;
      FIcon := TIcon.Create;
      try
        FIcon.Handle := LoadIcon(0, ResIDs[AType]);
        Glyph.Picture.Graphic := FIcon;
        Glyph.BoundsRect := Bounds(mgTextMargin,
          mgTextMargin, FIcon.Width, FIcon.Height);
      finally
        FIcon.Free;
      end;
    end else
      Glyph := nil;
    { sum up the size of the informational items }
    InfoSize.X := (TextRect.Right - TextRect.Left) +
                  (mgTextMargin * 2);
    if Glyph <> nil then
      Inc(InfoSize.X, Glyph.Picture.Graphic.Width +
      mgGlyphSpacing);
    if Glyph <> nil then
      InfoSize.Y := Max(Glyph.Picture.Graphic.Height,
        TextRect.Bottom - TextRect.Top) + mgTextMargin
    else
      InfoSize.Y := (TextRect.Bottom - TextRect.Top) +
                     mgTextMargin;
    { create the buttons }
    ButtonCount := 0;
    for Btn := Low(TMsgBoxBtn) to High(TMsgBoxBtn) do begin
      if Btn in AButtons then begin
        Inc(ButtonCount);
        Buttons[Btn] := TButton.Create(Result);
        with Buttons[Btn] do begin
          Parent := Result;
          SetBounds(0, 0, MsgDlgBtnSize.X, MsgDlgBtnSize.Y);
          Caption := LoadStr(BtnCaptions[Btn]);
          Name := StrPas(BtnNames[Btn]);
          ModalResult := Btnresult[Btn];
          { Need to set Default and Cancel properties
            for button }
          If (Btn = mbNo) or (Btn = mbCancel) then
            Cancel := True;
          If (Btn = mbYes) or (Btn = mbOK) then
            Default := True;
          { Determine if button is to be the default }
          If ButtonCount = Ord(ADefaultButton) + 1 then
            ActiveControl := TButton(Buttons[Btn]);
        end;
      end else
        Buttons[Btn] := nil;
    end;
   { If both a No and a Cancel button exist, then turn off
     the Cancel style of the NO button }
   if (mbNo in AButtons) and (mbCancel in AButtons) then
     Buttons[mbNo].Cancel := False;
   { If both a Yes and an OK button exist, then turn off
     the Default style of the Yes button }
   if (mbYes in AButtons) and (mbOK in AButtons) then
     Buttons[mbYes].Default := False;
   { if only an OK button exists, mark it as Cancel also }
   if (mbOK in AButtons) and (ButtonCount = 1) then
     Buttons[mbOK].Cancel := True;
   ButtonSize.X := (ButtonCount * MsgDlgBtnSize.X) +
     (mgButtonSpacing * (ButtonCount - 1)) +
     (mgBtnLRMargin * 2);
   ButtonSize.Y := MsgDlgBtnSize.Y + mgBtnTopMargin +
                   mgBtnBottomMargin;
   { set the caption }
   if ACaption <> EmptyStr then
     Caption := ACaption
   else
     Caption := Application.Title;
   { Set minimum width and height for TForm necessary to
     display complete caption, icon, message and buttons }
   ClientWidth := Max(Max(Canvas.TextWidth(Caption) + 50,
     MsgDlgMinWidth), Max(InfoSize.X, ButtonSize.X));
   ClientHeight := Max(MsgDlgMinHeight, InfoSize.Y +
     ButtonSize.Y);
   { layout the text and glyph }
   if (Glyph <> nil) and (Glyph.Height >
     (TextRect.Bottom - TextRect.Top)) then begin
     Glyph.Top := mgTextMargin;
     MsgLabel.Top := Glyph.Top +
       (Glyph.Picture.Graphic.Height div 2) -
       ((TextRect.Bottom - TextRect.Top) div 2);
     ButtonTop := Glyph.Top + Glyph.Height;
   end else begin
     MsgLabel.Top := mgTextMargin;
     if Glyph <> nil then
       Glyph.Top := MsgLabel.Top + (((TextRect.Bottom -
         TextRect.Top) div 2) - (Glyph.Height div 2));
     ButtonTop := MsgLabel.Top + MsgLabel.Height;
   end;
   if Glyph <> nil then
     MsgLabel.Left :=
       Glyph.Left + Glyph.Width + mgGlyphSpacing
   else
     MsgLabel.Left := mgTextMargin;
   { layout the buttons }
   ButtonX := (Result.ClientWidth div 2) -
     (ButtonSize.X div 2) + mgBtnLRMargin;
   for Btn := Low(TMsgBoxBtn) to High(TMsgBoxBtn) do
     if Buttons[Btn] <> nil then begin
       Buttons[Btn].Left := ButtonX;
       Buttons[Btn].Top := mgBtnTopMargin + ButtonTop;
       Inc(ButtonX, Buttons[Btn].Width + mgButtonSpacing);
     end;
    { Removes CLOSE option from system menu.
      GetSystemMenu(Result.Handle, False) Obtains Handle
      to System Menu }
    RemoveMenu(GetSystemMenu(Result.Handle, False), 1,
      MF_BYPOSITION);
  end;
  { Center form on the screen }
  Result.Left :=
    (Screen.Width div 2) - (Result.Width div 2);
  Result.Top :=
    (Screen.Height div 2) - (Result.Height div 2);
end;
```

component, we first have to create a TIcon and load the glyph into it using LoadIcon. We can then set the TImage.Picture.Graphic property to TIcon. Lastly, we will need to free the TIcon.

At this point we can calculate how much space is required to display the message and the glyph (the information area). Since we need to calculate both the height and width of this area, we need two integer variables. A TPoint variable stores two integers, usually a set of X and Y coordinates. We can also use it here to store the height and width of a component, or the image area. The width will be InfoSize.X and the height InfoSize.Y.

Later, the ClientWidth of the form will be set at least equal to InfoSize.X, which will be the width of the message, plus the width of the glyph, a margin between these two items and the margins between these items and the edges of the form. If TMsgBoxType is mtCustom then there is no glyph and InfoSize.X will be the width of the message and margins at the left and right of the message. TextRect contains the X and Y coordinates of its boundaries, so to determine the width of the text we subtract the TextRect.Left boundary from the TextRect.Right boundary.

We will also need to calculate the height of the information area. If we have a one line caption, the height of the glyph will be greater than the height of the message. With a large multi-line message, the opposite will be true. We use the Max function to determine which is greater, the height of the glyph or the calculated height of TextRect (the message), and return the maximum value. A margin between the top of the message or glyph and the top of the form's client area will be added to the result.

Now we calculate the height of the information area.

Creating The Buttons

Next, we need to create the buttons. By looping through our list of possible buttons (TMsgBoxBtn) and comparing each one to the list of buttons we want to display (AButtons), we can create only the buttons we need. We will keep track of how many buttons we have created with the ButtonCount variable. The ModalResult, Default and Cancel properties will be assigned to each button as appropriate. If we compare the ButtonCount value to the ADefaultButton parameter, we can determine which button needs to be the default and set the form's ActiveControl property to this button.

I don't use the Borland BitButtons in my applications, I prefer to use the standard Windows buttons (TButton component). If you prefer BitButtons then feel free to substitute TBitBtn for TButton. Setting the TBitBtn Kind property will set the appropriate glyph, as well as the ModalResult, Default and Cancel properties.

We should check that we haven't created any conflicts by producing two buttons with the Cancel or Default properties set to True. If we have, we need to set the respective property to False for one of the buttons.

If we only have an OK button, we should set the button's Cancel property to True.

We now need to calculate the height and width of the space needed for displaying the buttons. This includes the width of each button, the margins between the buttons and the edges of the form, and a space between each button. The height is simply the height of a button, plus a margin at the bottom and top of the buttons.

Titling Our Message Box

Often, I use the title of my application as the caption for message boxes. This way, if the user is running my application in the background, while using another program, they can identify what program created the displayed message box. When I don't use the application title, I try to make the caption descriptive of both the reason for the message box, and the application that created it. Usually something like MyApp Error. The title of the application is assigned on the Application page of the Project Options dialog and stored in the Application.Title property.

Rather than constantly passing Application.Title as the Caption parameter, we'll have CreateMsgBox check ACaption to see if it is an empty string. If it is, we'll set the form's Caption property to Application.Title, otherwise we'll set it to the ACaption parameter of the function. This way we can simply pass an empty string as the ACaption parameter when we want to use the application's title as the message box caption.

Size And Position

Now it's time to resize the form so it will display all the buttons, the message, the glyph and the form's caption. This is done by setting the form's ClientWidth and ClientHeight properties. The form's ClientWidth property will be the greatest of the width of the area required by the buttons (ButtonSize.X), the area required by the message and glyph (InfoSize.X), and the width of the Caption. The width of the caption can be determined by using the form's Canvas.TextWidth method to calculate the width in pixels of the caption. A margin will be added to this so that the caption is not pinched between the system menu button and the right edge of the form. To ensure that the message box is not too small, we'll also compare these values to a MsgDlgMinWidth constant. The form's ClientHeight will be the total height of the message and glyph (InfoSize.Y) and the area required for the buttons (ButtonSize.Y). We will also compare this to a MsgDlgMinHeight constant. All the necessary margins were included when we calculated the InfoSize and ButtonSize variables earlier.

Now we'll adjust the positions of the image control containing the glyph and the label containing our message. Earlier the Top and Left of the image control were set equal to mgTextMargin and the Top and Left of the message label were set to 0. If the height of the message text is less than the height of the glyph, the text will be centered vertically within the height of the glyph. If the height of the message text is greater, the glyph will be centered vertically within the height of the message text. The bottom edge of the message text and glyph (information area) are then calculated and assigned to the ButtonTop variable so that we can use this to position the tops of the buttons.

The Left property for the text label is then set. The Left property of the glyph was set earlier.

Next we'll loop through all the buttons, setting the Left and Top properties for each one. So that the buttons are centered horizontally, we'll calculate the left position for the leftmost button by subtracting the center point for the ButtonSize.X variable from the center point of the form's ClientWidth. Since the ButtonSize.X variable includes the left and right margins between the buttons and the edges of the form, we will add the margin value to this calculation. After placing each button we'll calculate the Left property for the next, with space between the buttons.

Removing Close From The System Menu

We need to remove Close from the system menu (one of the shortcomings discussed earlier). We can do this with the RemoveMenu API. Earlier, the form's BorderIcons property was set to [biSystemMenu]. This left only the Move and Close items on the menu. The system menu, like all menus, is zero based. This means that the Move or topmost option is in position 0 and the Close option is in position 1. The RemoveMenu API function allows us to remove each item based on its position. After each menu item is removed, the remaining items are renumbered. Because of this, it is recommended that menu items always be removed starting at the bottom. The GetSystemMenu API function returns the handle of the system menu, which is required by the RemoveMenu API function.

Finally, the default position of the message box is set to the center of the screen.

Calling CreateMsgBox

Now that we have created our message box, we need a function that will be responsible for calling the CreateMsgBox function, returning the form's ModalResult value and finally freeing the memory used by our message box. Since the form itself is returned by CreateMsgBox, we will still be able to assign the Help Context value to the form and position it somewhere other than the center of the screen. This way we won't have to pass these parameters to CreateMsgBox, simplifying the code. This new function will be called MsgDlgPos (see Listing 4).

The result variable for MsgDlgPos will be initialized to zero, so that if there is a problem creating the message box the function will return zero. The message box is created by calling CreateMsgBox, passing the AMsg, ACaption, AType, AButtons, and ADefaultButton parameters, and assigning the result to the W variable, of type TForm.

To ensure that our application does not fail to free up the memory used by the form, we use a try...finally block, placing W.Free in the finally portion of the block. In the try portion, we can assign the HelpCtx parameter to the form and change the form position if we desire. To leave the form in the default position, at the center of the screen, we pass -1 for the X and Y parameters of MsgDlgPos. If these parameters are anything other than -1, we assign them to the Top and Left properties for the form.

To maintain a consistent size for our form, regardless of the screen resolution used by the user, the form's ScaleBy method is used, passing the Screen.PixelsPerInch property and the form's PixelsPerInch property (which was set to 96 in CreateMsgBox). The message box is then displayed using the form's ShowModal method. The result of ShowModal (which is the ModalResult property of the button selected by the user) is assigned to the Result property of the MsgDlgPos function. This will allow us to use the result property of the MsgDlgPos function in our program to determine which button was selected.

Taming The Mouse Cursor

When I need to start a long operation, I set the mouse cursor to an hourglass, to let the user know this will take a while. This has become common practice and has been standardized by Microsoft (see Section 3.6.1.1 on *Graphical Feedback* in *The Windows Interface: An Application Design Guide*, Microsoft Press, 1992).

Changing the mouse cursor is done by setting the Screen.Cursor property to crHourglass. Contrary to its naming, the Screen.Cursor property does not change the cursor for the entire screen, just for the application and all forms owned by the application.

Occasionally, I may use a message box to inform the user of a problem within a long operation. When the Screen.Cursor property

➤ *Listing 4*

```
function MsgDlgPos(const AMsg: string; const ACaption: string;
  AType: TMsgBoxType; AButtons: TMsgBoxButtons;
  ADefaultButton: TDefaultBtn; HelpCtx: Longint; X, Y: Integer): Word;
var
  W: TForm;
  TempCursor: TCursor ;
begin
  Result := 0;
  TempCursor := Screen.Cursor;  { Store the current Screen.Cursor }
  { Use the following line if the message box is to be System Modal }
  { If SysModal Then AType := mtError; }
  W := CreateMsgBox(AMsg, ACaption, AType, AButtons, ADefaultButton);
  try
    W.HelpContext := HelpCtx;
    if X > -1 then W.Left := X;
    if Y > -1 then W.Top := Y;
    W.ScaleBy(Screen.PixelsPerInch, 96);
    Screen.Cursor := crDefault;  { Set the Screen.Cursor to Default }
    { Make some noise }
    MessageBeep(MsgBeep[AType]);
    { Use the following line if the message box is to be System Modal }
    { If SysModal Then SetSysModalWindow(W.Handle); }
    Result := W.ShowModal;
  finally
    Screen.Cursor := TempCursor;  { Restore the original Screen.Cursor }
    W.Free;
  end;
end;
```

has been set to crHourglass and the cursor is positioned over our message box, it will remain an hourglass. To change this, we'll store the current cursor before calling CreateMsgBox, then change the Screen.Cursor property to the default before displaying our message box. Changing the Screen.Cursor property just before displaying the message box, keeps the cursor an hourglass while the message box is being created, just in case it takes a little while. After the form is closed, we can then set the Screen.Cursor property back to the value we stored. If the current cursor is already the default, these procedures will have very little impact on the speed or memory required by our application.

Let's Make Some Noise

Many application users, especially database users and heads-down data entry clerks, probably won't see our message box appear on the screen. It has become common to get their attention with sound. Microsoft included the MessageBeep function in the Windows API for just this purpose. Four of the six possible values that can be passed to MessageBeep are designed to accompany message boxes: MB_ICONEXCLAMATION, MB_ICONSTOP, MB_ICONINFORMATION (*Asterisk* in the Sound applet) and MB_ICONQUESTION. These are the same values that are passed to the Windows MessageBox API function to select the icon or glyph that is displayed. All of these sounds, plus others, are assigned using the Sound applet in Windows Control Panel. The fifth value that can be passed to MessageBeep is MB_OK. This sound is assigned with the *Default Beep* entry in the Sound applet. If the PC does not have a sound card, or the user has disabled *System Sounds* in the Sound applet, Windows will use a PC speaker beep.

There is one more value that can be passed, but is not documented very well. A value of 65535 will always produce a quick beep or chirp from the PC speaker. Since the user could assign a long wave file to any of the entries in the Sound applet, a value of 65535 allows us to produce a brief sound. I have an application that places values in a grid component. As each value is placed in the grid, I use MessageBeep(65535) to inform the user the operation has been successful. If the user is expecting three values and only hears one beep, they know there is a problem and can fix it immediately.

By declaring an array of values matching the possible message box types, we can assign each sound based on its ordinate position in the array. The *Default Beep* (MB_OK) will be used if our message box is of type mtCustom. MessageBeep is called just before the message box is shown.

System Modal Message Box

There is one feature of the Windows MessageBox API function that we haven't included: the ability to create a system modal message box, rather than application modal. An application modal message box prevents the user from using the application until the message box is closed and its memory is freed. When an application modal message box is displayed, all the other running programs are accessible and can be used. A system modal message box prevents the user from using any application until the message box is closed and its memory is freed. This essentially freezes the operating system. I purposely commented this out in the code, because a system modal message box is seldom required and can be dangerous. System modal message boxes and forms have the potential to lock up the entire system, forcing the user to reboot and possibly lose data. System modal message boxes should be used only for serious errors where system damage could result if the message box is ignored. Remember, a system modal message box or window will not release its lock on the user's system until it is destroyed and its memory is freed.

Our message box is application modal. If you have a need to create a system modal message box, you can pass a Boolean parameter (SysModal: Boolean) as part of the MsgDlgPos function (see the commented out lines in Listing 4).

In keeping with the Windows standards that Microsoft created (in the MessageBox function), a system modal message box should be of type mtError.

Simpler Calls For MsgDlgPos

This ends our MsgDlgPos function. To use it, simply include the MsgDlgs unit in the Uses clause of the relevant unit and place calls to MsgDlgPos in your code. To make coding easier, I created a couple of extra functions that call MsgDlgPos and pass it a subset of our parameters (the remaining parameters are set to defaults). See Listing 5.

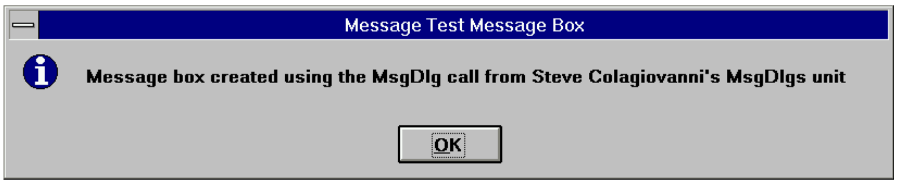
Centering Over Application

I have an application that only covers part of the screen, about 300x200 pixels. Most users run it in the corner of the screen, while using other applications. Since the height and width of the message box are unknown at design time, we

➤ *Listing 5*

```
{ The majority of message boxes are placed in the center of the screen and do not
  require a help context value. The following MsgDlg function passes defaults for
  these two parameters. }
function MsgDlg(const AMsg: string; const ACaption: string; AType:
  TMsgBoxType; AButtons: TMsgBoxButtons; ADefaultButton: TDefaultBtn): Word;
begin
  Result :=
    MsgDlgPos(AMsg, ACaption, AType, AButtons, ADefaultButton, 0, -1, -1);
end;

{ Sometimes you may want to display a simple message, without a glyph, and only
  using a simple OK button. The following two functions will do this. The
  ShowMsg function only has one parameter, the message to display. Default values
  are used for the other parameters. }
procedure ShowMsgPos(const AMsg: string; X, Y: Integer);
begin
  MsgDlgPos(AMsg, '', mtCustom, [mbOK], dfFirst, 0, X, Y);
end;
procedure ShowMsg(const AMsg: string);
begin
  MsgDlgPos(AMsg, '', mtCustom, [mbOK], dfFirst, 0, -1, -1);
end;
```


Message Test Message Box

Message box created using the MsgDlg call from Steve Colagiovanni's MsgDlgs unit

➤ *Figure 3: Our new, better, message box in action!*

can't calculate the proper top and left coordinates to center the message box over the application. Our final function, `FormCenteredMsgDlg` does this. Instead of passing the X and Y coordinates for the message box, you pass the form that you want to center the message box on. For an MDI application, to center the message box on the application, rather than the current active window, pass the parent form as the last parameter, rather than the form making the call. The code to calculate the position is as follows:

```
W.Left := (AForm.Width div 2)-
  (W.Width div 2)+AForm.Left;
W.Top := (AForm.Height div 2)-
  (W.Height div 2)+AForm.Top;
```

Conclusion

The ability to customize and extend with Delphi's is one of its greatest advantages.

As I've demonstrated, this isn't just limited to extending and subclassing components: we can extend or replace runtime functions and procedures to improve functionality too.

Steven J. Colagiovanni is a Photographic Technician with a major photographic manufacturer and is currently living in Los Angeles, California. He is a member of the Los Angeles Delphi User Group and programs in Delphi as a hobby. He can be reached at 76063.2220@compuserve.com