Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 15: IDE Main Menus

By David | March 28, 2012 0 Comment

Well its taken me a long time to produce this chapter because of what I believe are bugs in the various IDEs which I thought at the time was just me not doing it right.

The topic today is all about a more correct way of adding menus to the IDEs. The previous way I showed is a hang over from my very early days with the Open Tools API in Delphi 3 and does not take into account the changes that were introduced in later IDEs.

In producing this chapter I came across various problem but still wanted to produce a consistent way of adding menus to the IDE across all IDEs.

So lets start having a look at adding menus to the IDE.

Below is a definition of a class to manage the installation, lifetime and destruction of the menus.

```
TApplicationMainMenu = Class
{$IFDEF D2005} Strict {$ENDIF} Private
  FOTAMainMenu  : TMenuItem;
  {$IFNDEF D2005}
  FPatchTimer   : TTimer;
  {$ENDIF}
{$IFDEF D2005} Strict {$ENDIF} Protected
  Procedure InstallMainMenu;
  Procedure AutoSaveOptionsExecute(Sender : TObject);
  Procedure AboutExecute(Sender : TObject);
  Procedure ProjCreateWizardExecute(Sender : TObject);
  Procedure ShowCompilerMessagesClick(Sender : TObject);
  Procedure ShowCompilerMessagesUpdate(Sender : TObject);
  Procedure ShowEditorMessagesClick(Sender : TObject);
  Procedure ShowEditorMessagesUpdate(Sender : TObject);
  Procedure ShowIDEMessagesClick(Sender : TObject);
  Procedure ShowIDEMessagesUpdate(Sender : TObject);
  {$IFNDEF D2005}
  Procedure PatchShortcuts(Sender : TObject);
  {$ENDIF}
```

```
    Public
      Constructor Create;
      Destructor  Destroy; Override;
    End;
```

The above class contains a number of TNotifyEvents for menu clicks and update event handlers for the actions however these are not the important items. The class contains an internal variable `FOTAMainMenu` to hold a reference to the main menu you create, such that freeing this menu will free all child menus and thus you don't need to hold reference to all the menus you add. Additionally, and for Delphi 7 and below, there is a timer that will patch the shortcut menus as the IDEs seem to loose this information. There is a method to install the menu, `InstallMainMenu` and a `PatchShortcuts` method for the Delphi 7 and below patching of shortcuts.

But first we need to understand how to create this class and subsequently your expert's main menu. To do this I've made an interval private variable for the class and created it in the `Initialization` section of the unit and freed it in the `Finalization` section of the unit. This way the menu does not need to be invoked by the main initiatiation code where all your other experts are created but this does pose a problem. For those other elements to be able to be invoked by a menu they must expose a class method that invokes the functionality.

```
  Var
    ApplicationMainMenu : TApplicationMainMenu;

  Initialization
    ApplicationMainMenu := TApplicationMainMenu.Create;
  Finalization
    ApplicationMainMenu.Free;
  End.
```

The constructor below is fairly simple in that it initialises the menu reference to `nil` and runs the method `InstallMainMenu`. For Delphi 7 and below it also creates a TTimer control and assigns it to an event handler to patch the shortcuts.

```
  constructor TApplicationMainMenu.Create;
  begin
    FOTAMainMenu := Nil;
    InstallMainMenu;
    {$IFNDEF D2005} // Fixes a bug in D7 and below where shortcuts are lost
    FPatchTimer := TTimer.Create(Nil);
    FPatchTimer.Interval := 1000;
    FPatchTimer.OnTimer := PatchShortcuts;
    {$ENDIF}
  end;
```

The destructor simply frees the menu reference (must be assigned in the `InstallMainMenu` method) and frees the timer in Delphi 7 and below.

```
destructor TApplicationMainMenu.Destroy;

begin
  {$IFNDEF D2005}
  FPatchTimer.Free;
  {$ENDIF}
  FOTAMainMenu.Free; // Frees all child menus
  Inherited Destroy;
end;
```

The `InstallMainMenu` method is where most of the work is done for creating the menus in the IDE. This relies on a number of utility methods which we will go through in a while but its been designed to provide a simple interface for creating menus.

The below code checks that there is a main meun provided by the IDE and then creates a top level menu item assigning it to the `FOTAMainMenu` variable (so it can be freed later) and then creates the menu structure underneath that item.

You could use this technique to create a new menu structure underneath an existing IDE menu item but you will need to workout the menu item^#39;s `name` to do this.

I will describe the parameters of the `CreateMenuItem` method in a while.

```
procedure TApplicationMainMenu.InstallMainMenu;

Var
  NTAS : INTAServices;

begin
  NTAS := (BorlandIDEServices As INTAServices);
  If (NTAS <> Nil) And (NTAS.MainMenu <> Nil) Then
    Begin
      FOTAMainMenu := CreateMenuItem('OTATemplate', '&OTA Template', 'Tools',
        Nil, Nil, True, False, '');
      CreateMenuItem('OTAAutoSaveOptions', 'Auto Save &Option...', 'OTATemplate',
        AutoSaveOptionsExecute, Nil, False, True, 'Ctrl+Shift+O');
      CreateMenuItem('OTAProjectCreatorWizard', '&Project Creator Wizard...',
        'OTATemplate', ProjCreateWizardExecute, Nil, False, True, 'Ctrl+Shift+P');
      CreateMenuItem('OTANotifiers', 'Notifer Messages', 'OTATemplate', Nil, Nil,
        False, True, '');
      CreateMenuItem('OTAShowCompilerMsgs', 'Show &Compiler Messages',
        'OTANotifiers', ShowCompilerMessagesClick, ShowCompilerMessagesUpdate,
        False, True, '');
      CreateMenuItem('OTAShowEditorrMsgs', 'Show &Editor Messages',
        'OTANotifiers', ShowEditorMessagesClick, ShowEditorMessagesUpdate,
        False, True, '');
      CreateMenuItem('OTAShowIDEMsgs', 'Show &IDE Messages',
        'OTANotifiers', ShowIDEMessagesClick, ShowIDEMessagesUpdate,
        False, True, '');
      CreateMenuItem('OTASeparator0001', '', 'OTATemplate', Nil, Nil, False, True, '');
      CreateMenuitem('OTAAbout', '&About...', 'OTATemplate', AboutExecute, Nil,
        False, True, 'Ctrl+Shift+Q');
    End;
end;
```

Below are examples of `OnClick` and `OnUpdate` event handlers for the actions associated with the menus. Here I've used an enumerate amd set to handle some options in the application and update the checked property of the action based on the inclusion or exclusion of the enumerate in the set. The click action simply adds or removes the enumerate from the set. You will probably ask why I don't use `include` or `exclude` for the sets and enumerates. Since the set is a property of a class, you can not use the `include` or `exclude` methods on a property of a class.

```
Procedure UpdateModuleOps(Op : TModuleOption);

Var
  AppOps : TApplicationOptions;

Begin
  AppOps := ApplicationOps;
  If Op In AppOps.ModuleOps Then
    AppOps.ModuleOps := AppOps.ModuleOps - [Op]
  Else
    AppOps.ModuleOps := AppOps.ModuleOps + [Op];
End;

procedure TApplicationMainMenu.ShowCompilerMessagesClick(Sender: TObject);
begin
  UpdateModuleOps(moShowCompilerMessages);
end;

procedure TApplicationMainMenu.ShowCompilerMessagesUpdate(Sender: TObject);
begin
  If Sender Is TAction Then
    With Sender As TAction Do
      Checked := moShowCompilerMessages In ApplicationOps.ModuleOps;
end;
```

Finally for this module we have the Delphi 7 `OnTimer` event handler for the patching of the shortcuts. This is handled by a utility function which we will look at in a while but the event waits for a visible IDE before invoking the utility function and then switches off the timer.

```
{$IFNDEF D2005}
Procedure TApplicationMainMenu.PatchShortcuts(Sender : TObject);

Begin
  If Application.MainForm.Visible Then
    Begin
      PatchActionShortcuts(Sender);
      FPatchTimer.Enabled := False;
    End;
End;
{$ENDIF}
```

Now for the utility functions.

The `AddImageToIDE` function is called internally by `CreateMenuItem` and adds an image from the projects resource file to the IDEs image list and returns the index of that image in the IDEs image list so that it can be referenced in the action. You will note that there is a commented out section of this method, this is because it continually caused an exception, so an older method is used. If the resource is not found then no image is added to the IDE and -1 is returned as the image index.

```
Function AddImageToIDE(strImageName : String) : Integer;

Var
  NTAS : INTAServices;
  ilImages : TImageList;
  BM : TBitMap;

begin
  Result := -1;
  If FindResource(hInstance, PChar(strImageName + 'Image'), RT_BITMAP) > 0 Then
    Begin
      NTAS := (BorlandIDEServices As INTAServices);
      // Create image in IDE image list
      ilImages := TImageList.Create(Nil);
      Try
        BM := TBitMap.Create;
        Try
          BM.LoadFromResourceName(hInstance, strImageName + 'Image');
          {$IFDEF D2005}
          ilImages.AddMasked(BM, clLime);
          // EXCEPTION: Operation not allowed on sorted list
          // Result := NTAS.AddImages(ilImages, 'OTATemplateImages');
          Result := NTAS.AddImages(ilImages);
          {$ELSE}
          Result := NTAS.AddMasked(BM, clLime);;
          {$ENDIF}
        Finally
          BM.Free;
        End;
      Finally
        ilImages.Free;
      End;
    End;
end;
```

The **FindMenuItem** function is called internally by **CreateMenuItem** and is used to find a named menu item (i.e. the name assigned to the **name** property of an existing menu item. The named menu item is returned if found else **nil** is returned. This function recursively searches the main menu system.

```
function FindMenuItem(strParentMenu : String): TMenuItem;

  Function IterateSubMenus(Menu : TMenuItem) : TMenuItem;

  Var
    iSubMenu : Integer;

  Begin
    Result := Nil;
    For iSubMenu := 0 To Menu.Count - 1 Do
      Begin
        If CompareText(strParentMenu, Menu[iSubMenu].Name) = 0 Then
          Result := Menu[iSubMenu]
        Else
          Result := IterateSubMenus(Menu[iSubMenu]);
        If Result <> Nil Then
          Break;
      End;
  End;

Var
  iMenu : Integer;
  NTAS : INTAServices;
  Items : TMenuItem;

begin
  Result := Nil;
  NTAS := (BorlandIDEServices As INTAServices);
  For iMenu := 0 To NTAS.MainMenu.Items.Count - 1 Do
    Begin
      Items := NTAS.MainMenu.Items;
      If CompareText(strParentMenu, Items[iMenu].Name) = 0 Then
        Result := Items[iMenu]
      Else
        Result := IterateSubMenus(Items);
      If Result <> Nil Then
        Break;
    End;
end;
```

This next method is the heart of the experts ability to create a menu item in the IDEs main menu system and I will explain how it works.

- Firstly an image is added to the IDEs image list (if the resource exists in the expert);
- Next the menu item is created with the main menu as its owner;
- Next, if there is an `OnClick` event handler, then an Action is created and assigned various attributes like caption, etc;
- Next a catch is made for menu items that have no event handler (heads of sub-menus or separators);
- Then the action is assigned to the menu;
- This position of the parent menu is located;
- Adds the menu to the IDE relative to the parent menu.

You will probably note that there is more commented out code, this is because the “new” way to create menus in the IDE does not create icons next to the menus. It could be something that I'm not doing right but I spent an inordinate amount of time trying to get it to work.

Some explanation of the parameter is also needed as follows:

- **strName** – This is the name of the action / menu (which will be appropriately appended with text);
- **strCaption** – This is the name (with accelerator) of the action / menu;
- **strParentMenu** – This is the name of the parent menu. This is either the menu under which you want child menus or is the menu item which comes before or after your new menu depending on the below options;
- **ClickProc** – This is the `OnClick` event handler for the action / menu that does something when the menu or action is clicked or invoked. If you do not want to implement this, say for a top level menu, the pass `nil`;
- **UpdateProc** – This is an optional `OnUpdate` event handler for the action /menu. If you do not want to implment this simply pass `nil`;
- **boolBefore** – If true this will make the new menu appear before the Parent menu item;
- **boolChildMenu** – If true this will add the new menu as a child of the Parent menu;
- **strShortcut** – This is a shortcut string to be assigned to the action / menu. Just pass an empty string if you do not want to implement a shortcut.

```
Function CreateMenuItem(strName, strCaption, strParentMenu : String;
  ClickProc, UpdateProc : TNotifyEvent; boolBefore, boolChildMenu : Boolean;
  strShortCut : String) : TMenuItem;

Var
  NTAS : INTAServices;
  CA : TAction;
  //{$IFNDEF D2005}
  miMenuItem : TMenuItem;
  //{$ENDIF}
  iImageIndex : Integer;
```

```
begin
  NTAS := (BorlandIDEServices As INTAServices);
  // Add Image to IDE
  iImageIndex := AddImageToIDE(strName);
  // Create the IDE action (cached for removal later)
  CA := Nil;
  Result := TMenuItem.Create(NTAS.MainMenu);
  If Assigned(ClickProc) Then
    Begin
      CA := TAction.Create(NTAS.ActionList);
      CA.ActionList := NTAS.ActionList;
      CA.Name := strName + 'Action';
      CA.Caption := strCaption;
      CA.OnExecute := ClickProc;
      CA.OnUpdate := UpdateProc;
      CA.ShortCut := TextToShortCut(strShortCut);
      CA.Tag := TextToShortCut(strShortCut);
      CA.ImageIndex := iImageIndex;
      CA.Category := 'OTATemplateMenus';
      FOTAActions.Add(CA);
    End Else
  If strCaption <> '' Then
    Begin
      Result.Caption := strCaption;
      Result.ShortCut := TextToShortCut(strShortCut);
      Result.ImageIndex := iImageIndex;
    End Else
      Result.Caption := '-';
  // Create menu (removed through parent menu)
  Result.Action := CA;
  Result.Name := strName + 'Menu';
  // Create Action and Menu.
  //{$IFDEF D2005}
  // This is the new way to do it BUT doesnt create icons for the menu.
  //NTAS.AddActionMenu(strParentMenu + 'Menu', CA, Result, boolBefore, boolChildMenu);
  //{$ELSE}
  miMenuItem := FindMenuItem(strParentMenu + 'Menu');
  If miMenuItem <> Nil Then
    Begin
      If Not boolChildMenu Then
        Begin
          If boolBefore Then
            miMenuItem.Parent.Insert(miMenuItem.MenuIndex, Result)
          Else
            miMenuItem.Parent.Insert(miMenuItem.MenuIndex + 1, Result);
        End Else
          miMenuItem.Add(Result);
    End;
  //{$ENDIF}
end;
```

This next utility function is used to patch the IDE shortcuts which are lost by Delphi 7 and below. This method is called by the on timer event handler in the `MainMenuModule`. It uses the fact that we stored the menu shortcut in the **tag** property to re-apply the shortcut after the IDE is loaded.

```
Procedure PatchActionShortcuts(Sender : TObject);

Var
  iAction : Integer;
  A : TAction;

Begin
  For iAction := 0 To FOTAActions.Count - 1 Do
    Begin
      A := FOTAActions[iAction] As TAction;
      A.ShortCut := A.Tag;
    End;
End;
```

Finally, this last utility function is provided to remove any of your custom actions from the toolbars> If you unloaded your BPL file and then tried to use your custom action then you would have an access violation in the IDE.

```
Procedure RemoveToolbarButtonsAssociatedWithActions;

  Function IsCustomAction(Action : TBasicAction) : Boolean;

  Var
    i: Integer;

  Begin
    Result := False;
    For i := 0 To FOTAActions.Count - 1 Do
      If Action = FOTAActions[i] Then
        Begin
          Result := True;
          Break;
        End;
  End;

  Procedure RemoveAction(TB : TToolbar);

  Var
    i: Integer;

  Begin
    If TB <> Nil Then
      For i := TB.ButtonCount - 1 DownTo 0 Do
        Begin
          If IsCustomAction(TB.Buttons[i].Action) Then
            TB.RemoveControl(TB.Buttons[i]);
        End;
  End;

Var
  NTAS : INTAServices;

Begin
  NTAS := (BorlandIDEServices As INTAServices);
  RemoveAction(NTAS.ToolBar[sCustomToolBar]);
  RemoveAction(NTAS.Toolbar[sStandardToolBar]);
  RemoveAction(NTAS.Toolbar[sDebugToolBar]);
  RemoveAction(NTAS.Toolbar[sViewToolBar]);
  RemoveAction(NTAS.Toolbar[sDesktopToolBar]);
  {$IFDEF D0006}
  RemoveAction(NTAS.Toolbar[sInternetToolBar]);
  RemoveAction(NTAS.Toolbar[sCORBAToolBar]);
  {$IFDEF D2009}
  RemoveAction(NTAS.Toolbar[sAlignToolbar]);
  RemoveAction(NTAS.Toolbar[sBrowserToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLDesignToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLFormatToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLTableToolbar]);
  RemoveAction(NTAS.Toolbar[sPersonalityToolBar]);
  RemoveAction(NTAS.Toolbar[sPositionToolbar]);
  RemoveAction(NTAS.Toolbar[sSpacingToolbar]);
  {$ENDIF}
  {$ENDIF}
End;
```

The last part of the utility unit creates and frees the memory for a collection holding the actions that we've added to the IDE. Since the collection owns the action, freeing the collection removes the action from the IDE.

```
    Initialization
      FOTAActions := TObjectList.Create(True);
    Finalization
      RemoveToolbarButtonsAssociatedWithActions;
      FOTAActions.Free;
    End.
```

Well I hope that's useful. The code can be downloaded here (OTAChapter15.zip). In the next couple of chapters I'm going to look at creating forms and inherited forms.

regards
Dave.

Category: Open Tools API Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, Experts, INTAServices, OTA, RAD Studio