Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 5: Useful Open Tools Utility Functions

By David | April 14, 2011 0 Comment

I thought that before tackling some of the more interesting Open Tools API topics previously mentioned that I would describe some useful utility function that can make using the OTA easier.

Messages

First I thought we would look at messages (but not custom ones, they will require their own chapter). These are useful for outputting information from your expert and also for helping debug your application without having to run a debugging session of the IDE from within the IDE.

```
Procedure OutputMessage(strText : String);

Begin
  (BorlandIDEServices As IOTAMessageServices).AddTitleMessage(strText);
End;
```

The above code outputs a **Title Message** to the message window of the IDE. Its only parameter is the message you want displayed in the message window. The message doesn’t allow for any interaction, i.e. click on it and going to a line number. For that we need a **Tool Message** as below:

```
Procedure OutputMessage(strFileName, strText, strPrefix : String;
  iLine, iCol : Integer);

Begin
  (BorlandIDEServices As IOTAMessageServices).AddToolMessage(strFileName,
    strText, strPrefix, iLine, iCol);
End;
```

This is an overloaded version of the first function, so you will require the reserved word **overload** in the function interface declaration. This procedure has a number of parameters as follows:

- **strFileName** – This is the name of the file to which the message should be associated, i.e. error message from some code where the file name is the file name of the code file (D:\Path \MyModule.pas);
- **strText** – This is the message to be displayed;
- **strPrefix** – This is a prefix text that is displayed in front of the message in the message window to define which tool produced the message;
- **iLine** – This is the line number in the file name above at which the message should be associated;
- **iCol** – This is the column number in the file name above at which the message should be associated.

The supplying of the file name, line and column allow the IDE to go to that file, line and column when you double click the message in the message window.

Next we need to be able to clear the messages from the message window. The OTA defines 3 methods for this. I've wrapped them up into a single method which requires an enumerate and set to define which messages are cleared as follows:

```
Type
  (** This is an enumerate for the types of messages that can be cleared. **)
  TClearMessage = (cmCompiler, cmSearch, cmTool);
  (** This is a set of messages that can be cleared. **)
  TClearMessages = Set of TClearMessage;
```

Thus the method can be called with one or more enumerates in the set to define which messages are cleared from the message windows.

```
Procedure ClearMessages(Msg : TClearMessages);

Begin
  If cmCompiler In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearCompilerMessages;
  If cmSearch In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearSearchMessages;
  If cmTool In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearToolMessages;
End;
```

When working with messages you may wish at a point in the processing of your information to force the message window to be displayed / brought to the foreground. You can do this with the following function:

```
Procedure ShowHelperMessages;
```

```
Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    ShowMessageView(Nil);
End;
```

This displays the main message window of the IDE. In later versions of the IDEs you can output messages to tabs within the message window. To display those messages the above code needs to be modified as follows:

```
Procedure ShowHelperMessages;

Var
  G : IOTAMessageGroup;

Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      G := GetGroup('My Message');
      ShowMessageView(G);
    End;
End;
```

Obviously you can parameterize this method to allow you to pass the name of the message tab to the method to make it more flexible.

So the question arises as to how we added messages to these tabbed message windows?

First you need to create a message group as follows:

```
Var
  MyGroup : IOTAMessageGroup;

Begin
  ...
  MyGroup := AddMessageGroup('My Message Group');
  ...
End;
```

You can also get an existing Message Group as follows:

```
Var
  MyGroup : IOTAMessageGroup;
```

```
Begin
  ...
  MyGroup := function GetGroup('My Message Group');
  ...
End;
```

The later IDEs have similar overloaded methods to those described above which take an extra parameter which is the Message Group as follows:

```
    procedure AddTitleMessage(const MessageStr: string; const MessageGroupIntf:
IOTAMessageGroup); overload;
    procedure AddToolMessage(const FileName, MessageStr, PrefixStr: string;
      LineNumber, ColumnNumber: Integer; Parent: Pointer; out LineRef: Pointer;
      const MessageGroupIntf: IOTAMessageGroup); overload;
```

So for instance the first method I described could be re-coded as follows:

```
Procedure OutputMessage(strText : String; strGroupName : String);

Var
  Group : IOTAMessageGroup

Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      Group := GetGroup(strGroupName);
      AddTitleMessage(strText, Group);
    End;
End;
```

It should be noted that all the above function do not check to see if the **BorlandIDEServices** interface is available. For most OTA code this shouldn't be a problem as if this services isn't available, you've got bigger problems with the IDE than not being able to use the interface. The only situation I can think of off the top of my head where this could be a problem is if you call this when creating a splash screen for BDS/RAD Studio 2005 and above as this code get called before this service is available. But since the message window isn't available you wouldn' be able to log messages.

Projects and Project Groups

The following code samples provide mean by which you can get access to project groups, projects, modules and editor code. This is not a comprehensive list, but other code will appear in the other chapters that should fill in the missing gaps.

```
Function ProjectGroup: IOTAProjectGroup;

Var
  AModuleServices: IOTAModuleServices;
  AModule: IOTAModule;
  i: integer;
  AProjectGroup: IOTAProjectGroup;

Begin
  Result := Nil;
  AModuleServices := (BorlandIDEServices as IOTAModuleServices);
  For i := 0 To AModuleServices.ModuleCount - 1 Do
    Begin
      AModule := AModuleServices.Modules[i];
      If (AModule.QueryInterface(IOTAProjectGroup, AProjectGroup) = S_OK) Then
       Break;
    End;
  Result := AProjectGroup;
end;
```

The above code returns a reference to the project group (there is only 1 in the IDE at a time). If there is no group open (i.e. nothing in the Project Manager) then this will return **nil**.

```
Function ActiveProject : IOTAProject;

var
  G : IOTAProjectGroup;

Begin
  Result := Nil;
  G := ProjectGroup;
  If G <> Nil Then
    Result := G.ActiveProject;
End;
```

This above code returns a reference to the active project in the Project Manager (the project highlighted in bold in the tree view). If there is no active project then this function will return **nil**.

```
Function ProjectModule(Project : IOTAProject) : IOTAModule;

Var
  AModuleServices: IOTAModuleServices;
  AModule: IOTAModule;
  i: integer;
  AProject: IOTAProject;
```

```
Begin
  Result := Nil;
  AModuleServices := (BorlandIDEServices as IOTAModuleServices);
  For i := 0 To AModuleServices.ModuleCount - 1 Do
    Begin
      AModule := AModuleServices.Modules[i];
      If (AModule.QueryInterface(IOTAProject, AProject) = S_OK) And
        (Project = AProject) Then
        Break;
    End;
  Result := AProject;
End;
```

The above code returns a reference to the projects source modules (DPR, DPK, etc) for the given project.

```
Function ActiveSourceEditor : IOTASourceEditor;

Var
  CM : IOTAModule;

Begin
  Result := Nil;
  If BorlandIDEServices = Nil Then
    Exit;
  CM := (BorlandIDEServices as IOTAModuleServices).CurrentModule;
  Result := SourceEditor(CM);
End;
```

The above code returns a reference to the active IDE source editor interface. If there is no active editor then this method returns **nil**.

```
Function SourceEditor(Module : IOTAMOdule) : IOTASourceEditor;

Var
  iFileCount : Integer;
  i : Integer;

Begin
  Result := Nil;
  If Module = Nil Then Exit;
  With Module Do
    Begin
      iFileCount := GetModuleFileCount;
      For i := 0 To iFileCount - 1 Do
        If GetModuleFileEditor(i).QueryInterface(IOTASourceEditor,
          Result) = S_OK Then
          Break;
    End;
  End;
```

The above code provides a reference to the given modules source editor interface.

```
Function EditorAsString(SourceEditor : IOTASourceEditor) : String;

Var
  Reader : IOTAEditReader;
  iRead : Integer;
  iPosition : Integer;
  strBuffer : AnsiString;

Begin
  Result := '';
  Reader := SourceEditor.CreateReader;
  Try
    iPosition := 0;
    Repeat
      SetLength(strBuffer, iBufferSize);
      iRead := Reader.GetText(iPosition, PAnsiChar(strBuffer), iBufferSize);
      SetLength(strBuffer, iRead);
      Result := Result + String(strBuffer);
      Inc(iPosition, iRead);
    Until iRead < iBufferSize;
  Finally
    Reader := Nil;
  End;
End;
```

Lasty, the above code returns the given editor source's code as a string.

All of the above is not meant to be a complete set of utilities, just those that I've coded for my Browse And Doc It and Integrated Testing Helper IDE experts. It does make me think now that I should combine these into a single utility module across all my IDE experts (there currently in different project specific modules with only a couple of duplications).

Category: Browse and Doc It Integrated Testing helper Open Tools API RAD Studio Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, Experts, IOTAEditReader, IOTAMessageServices, IOTAModule, IOTAModuleServices, IOTAProject, IOTAProjectGroup, IOTASourceEditor, OTA, RAD Studio