Dave's Development Blog
Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 10: Reading editor code

By David | August 29, 2011 0 Comment

In this chapter I'll go through how to get the code from the editor and do something with it. For this example I'm simply going to allow you to select a method from the current module and move to its position. This will use some of the utility functions I've described before in (see Chapter 5: Useful Open Tools Utility Functions) but I'll describe how they work.

All the code for this is attached in a zip file at the end of this article.

To invokes this code I'm going to extend the key bindings we introduced in Chapter 4: Key Bindings and Debugging Tools.

Below I've added a new key binding attached to a new handler.

```
Procedure TKeybindingTemplate.BindKeyboard(Const BindingServices: IOTAKeyBindingServices);

Begin
  ...
  BindingServices.AddKeyBinding([TextToShortcut('Ctrl+Shift+Alt+F9')],
SelectMethodExecute, Nil);
End;
```

The handler is implemented to call a method to select a method as below.

```
Procedure TKeybindingTemplate.SelectMethodExecute(Const Context: IOTAKeyContext;
  KeyCode: TShortcut; Var BindingResult: TKeyBindingResult);

Begin
  SelectMethod;
  BindingResult := krHandled;
End;
```

Below is the implementation of the `SelectMethod` method.

```
Procedure SelectMethod;

Var
  slItems: TStringList;
  SE: IOTASourceEditor;
  CP: TOTAEditPos;
  recItemPos : TItemPosition;
  iIndex: Integer;

Begin
  slItems := TStringList.Create;
  Try
    GetMethods(slItems);
    iIndex := TfrmItemSelectionForm.Execute(slItems, 'Select Method');
    If iIndex > -1 Then
      Begin
        SE := ActiveSourceEditor;
        If SE <> Nil Then
          Begin
            recItemPos.Data := slItems.Objects[iIndex];
            CP.Line := recItemPos.Line;
            CP.Col := 1;
            SE.EditViews[0].CursorPos := CP;
            SE.EditViews[0].Center(CP.Line, 1);
          End;
      End;
  Finally
    slItems.Free;
  End;
End;
```

There's a lot in here so I'll try and break it down into stages. This method creates a string list in which we will added the methods to be selected from. This string list is passed to the method `GetMethods` to extract the methods from the source code. We will look at this in more detail later. This string list is then passed to a form which I'm not going to describe here but is included in the code at the bottom of the article. All you need to know is the forms method takes the string list and returns the index of the selected it in the string list. The rest of the above method moves the cursor position to the line number of the selected method (the line number is stored in the `Object` member of the string list. I'll explain how this is done below but you can look up the technique in Storing numbers, enumerates and sets in a TStringList all at the same time.

```
Type
  TSubItem = (siData, siPosition);

  TItemPosition = Record
    Case TSubItem Of
```

```
      siData: (Data : TObject);
      siPosition: (
        Line   : SmallInt;
        Column : SmallInt
      );
  End;
```

The idea about the above record its that you define a variant record where by the Line and Column information decsribed as **SmallInt**s (16 bits each) are described in the same memory location as the 32 bit **TObject** pointer. This way the Line and Column information is stored in the lower and upper portions of the Data reference. This also means we do not need casting and shifting commands as we just assign the **TStringList TObject** data to the record's Data memory and then we can access the Line and Column information directly from the record and visa versa.

The first method called in the **SelectMethod** method above is **GetMethods** as shown below:

```
Procedure GetMethods(slItems : TStringList);

Var
  SE: IOTASourceEditor;
  slSource: TStringList;
  i: Integer;
  recPos : TItemPosition;
  boolImplementation : Boolean;
  iLine: Integer;

Begin
  SE := ActiveSourceEditor;
  If SE <> Nil Then
    Begin
      slSource := TStringList.Create;
      Try
        slSource.Text := EditorAsString(SE);
        boolImplementation := False;
        iLine := 1;
        For i := 0 To slSource.Count - 1 Do
          Begin
            If Not boolImplementation Then
              Begin
                If Pos('implementation', LowerCase(slSource[i])) > 0 Then
                  boolImplementation := True;
              End Else
            If IsMethod(slSource[i]) Then
              Begin
                recPos.Line := iLine;
                recPos.Column := 1;
                slItems.AddObject(slSource[i], recPos.Data);
              End;
            Inc(iLine);
```

```
          End;
        slItems.Sort;
      Finally
        slSource.Free;
      End;
    End;
End;
```

The first things this method does is get a reference to the current Source Editor in the IDE using the utility function **ActiveSourceEditor** below:

```
Function ActiveSourceEditor : IOTASourceEditor;

Var
  CM : IOTAModule;

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

This method uses the **IOTAModuleServices** interface to get the IDE's current module being edited. It then uses another utility function **SourceEditor** to return the **IOTASourceEditor** interface from this current module provided as below:

```
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 cycles through the module files associated with the given module looking for a **IOTASourceEditor** interface. When found this interface is returned. This function uses the **QueryInterface** method to test for the interface we want. There are a number of occasion in the Open Tools API where the required interface is not directly exposed and the use of **QueryInterface** is required to test for the interface we require.

Returning to the **GetMethods** method, if we have a valid source editor interface we then need the source code associated with that editor using the **SourceEditor** function as follows:

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

Const
  iBufferSize : Integer = 1024;

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;
```

This function gets a Reader interface from the source editor by calling the **CreateReader** method. To get the text from the Reader interface we need to get the text in chunks using the **GetText** method of the **IOTAEditReader** interface. The buffer must be a AnsiChar buffer as the editor only returns Unicode UTF8 code or ANSI code not double bit Unicode. We loop the **GetText** method until it returns a number (the number of characters read) less than the buffer size and we add the buffer contents to the end of the resultant string for the function. We must maintain through out this the position in the text (**iPosition**) as the **GetText** method returns the chunk of text starting at a position.

Returning again now to **GetMethods**, we can assign the source editor text to the **Text** property of a string list and cycle through this strings searching for method headings in the implementation section. To help with

that I created the below function to check a line of text for method heading.

```
Function IsMethod(strLine : String) : Boolean;

Const
  strMethods : Array[1..4] Of String = ('procedure', 'function', 'constuctor',
'destructor');

Var
  i : Integer;

Begin
  Result := False;
  For i := Low(strMethods) To High(strMethods) Do
    If Pos(strMethods[i], LowerCase(strLine)) > 0 Then
      Begin
        Result := True;
        Break;
      End;
End;
```

If we find a line that contains a method heading, then this line is added to the string list passed to `GetMethods`.

Back to `SelectMethod`, we then pass the string list to our form code for selection by the user. Once the user has selected the
method they want we get the line number of that method from the string lists `Objects` property and using the active source editors `EditViews` property move the cursor position and centre the editor on that line.

The code for this chapter and all the previous code can be downloaded here (OTA Chapter 10).

Dave.

Category: Open Tools API Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, Experts, IOTAEditReader, IOTAKeyBindingServices, IOTAModule, IOTAModuleServices, IOTASourceEditor, OTA, RAD Studio