



Essential Studio 2013 Volume 4 - v.11.4.0.26

Essential Edit for Windows Forms



Contents

1	Overview	9
1.1	Introduction To Essential Edit.....	9
1.2	Prerequisites and Compatibility	11
1.3	Documentation	12
2	Installation and Deployment	14
2.1	Installation.....	14
2.2	Sample and Location.....	14
2.3	Deployment Requirements.....	17
2.3.1	Toolbox Entries.....	17
2.3.2	DLLs	17
3	Getting Started	18
3.1	Control Structure	18
3.2	Creating an Edit Control	18
3.2.1	Through Designer.....	19
3.2.2	Through Code.....	20
4	Concepts And Features	23
4.1	Configuration Settings	23
4.1.1	Creating a Custom Language Configuration File	23
4.1.2	Creating Configuration Settings Programmatically	27
4.2	Editing Features	30
4.2.1	Undo / Redo Actions.....	30
4.2.2	New Line Styles.....	34
4.2.3	Clipboard Operations.....	34
4.2.3.1	EnableMD5.....	36
4.2.4	Keystroke - Action Combinations Binding	37
4.2.5	Regular Expressions	40
4.2.5.1	Language Elements.....	41
4.2.5.2	Lexical Macros.....	45
4.2.6	Block Indent and Outdent	47
4.2.7	Right-To-Left (RTL) Support.....	49

4.3	Code Completion.....	51
4.3.1	AutoComplete Support	51
4.3.2	AutoReplace Triggers.....	53
4.4	Text Visualization	55
4.4.1	Text Navigation.....	55
4.4.1.1	Positions and Offsets.....	59
4.4.2	Column Guides.....	63
4.4.3	Content Dividers	65
4.4.4	Underlines, Wavelines and StrikeThrough.....	67
4.4.5	Text Handling	70
4.4.5.1	Appending, Deleting and Inserting Multiple Lines of Text	71
4.4.6	Spaces and Tabs.....	75
4.4.6.1	WhiteSpace Indicators	78
4.4.7	Line Numbers and Current Line Highlighting	81
4.4.8	Bookmarks and Custom Indicators	84
4.4.9	Comments	89
4.4.10	Break Points	90
4.4.11	Text Formatting	92
4.4.11.1	Bracket Highlighting and Indentation Guidelines	92
4.4.11.2	Auto Indentation	97
4.4.11.2.1	Lexem Support for AutoIndent Block Mode	99
4.4.11.3	AutoFormatting.....	100
4.4.11.4	Unicode	102
4.4.11.5	Automatic Outlining	103
4.4.11.5.1	Outlining Tooltip	106
4.4.11.6	Wordwrap	108
4.4.11.6.1	Wordwrap Margin Customization and Wrapping Images.....	112
4.4.11.7	Read-Only Text	116
4.4.12	Customizing Text.....	118
4.4.12.1	Text Color	118
4.4.12.2	Text Border.....	118
4.4.12.3	Encoding Text.....	120
4.4.12.4	Text Selection.....	122
4.5	Syntax Highlighting and Code Coloring.....	126

4.5.1	XML Based Configuration Files.....	132
4.5.2	Multiple Language Syntax Highlighting	134
4.6	Runtime Features	135
4.6.1	Insert Mode.....	135
4.6.2	Keyboard Shortcuts	136
4.6.3	Bitmap Generation.....	138
4.6.4	Find, Replace and Goto.....	139
4.6.5	Enhanced Find Dialog	145
4.6.6	Scrolling Support	146
4.6.6.1	Office 2007 Visual Style	150
4.6.6.1.1	ToolTip.....	151
4.6.6.2	Interactive Features.....	152
4.6.6.2.1	Customizable Context Menu	152
4.6.6.2.2	IntelliPrompt Features	156
4.6.6.2.3	Custom Cursor	183
4.6.6.2.4	Intellimouse Scrolling	184
4.6.6.2.5	Drag-and-drop	185
4.7	Text Export	186
4.7.1	XML, RTF and HTML Export.....	186
4.7.2	Schema Definition File for XML Syntax Coloring Configuration File	188
4.8	File Sharing and Stream Handling	188
4.8.1	Creating, Loading, Saving And Dropping Files	189
4.8.2	Loading And Saving Contents.....	192
4.8.3	Saving And Cancelling Changes	194
4.8.4	File Sharing	198
4.8.5	Lexical Analysis And Semantic Parsing	198
4.8.6	Clearing/Flushing Saved Changes	200
4.9	Appearance	201
4.9.1	Visual Settings	201
4.9.1.1	Size 201	
4.9.1.2	Split Views	202
4.9.1.3	Applying Themes.....	205
4.9.1.4	Border Style.....	206
4.9.1.5	Graphics Customization Settings	207
4.9.2	Margins	208

4.9.2.1	Selection Margin.....	208
4.9.2.2	User Margin	211
4.9.3	Background Settings	214
4.9.4	Font Customization.....	220
4.9.5	Single Line Mode.....	222
4.9.6	Customizable Find Dialog	223
4.10	Status Bar	225
4.11	Printing.....	228
4.12	Performance	234
4.13	Localization and Globalization.....	235
4.14	Edit Control Events.....	238
4.14.1	CanUndoRedoChanged Event.....	238
4.14.2	Closing Event	239
4.14.3	Code Snippet Events.....	239
4.14.3.1	CodeSnippetActivating Event.....	239
4.14.3.2	CodeSnippetDeactivating Event.....	240
4.14.3.3	CodeSnippetTemplateTextChanging Event.....	241
4.14.3.4	NewSnippetMemberHighlighting Event.....	242
4.14.4	ConfigurationChanged Event	243
4.14.5	Collapse Events.....	243
4.14.5.1	CollapsedAll Event	243
4.14.5.2	CollapsingAll Event	244
4.14.6	ContextChoice Events	245
4.14.6.1	ContextChoiceBeforeOpen Event	245
4.14.6.2	ContextChoiceSelectedTextInsert Event	246
4.14.6.3	ContextChoiceClose Event	246
4.14.6.4	ContextChoiceItemSelected Event	246
4.14.6.5	ContextChoiceUpdate Event.....	246
4.14.6.6	ContextChoiceOpen Event.....	248
4.14.6.7	ContextChoiceRightClick Event	248
4.14.7	ContextPrompt Events.....	249
4.14.7.1	ContextPromptBeforeOpen Event.....	249
4.14.7.2	ContextPromptClose Event	249
4.14.7.3	ContextPromptOpen Event	249
4.14.7.4	ContextPromptSelectionChanged Event.....	249

4.14.7.5	ContextPromptUpdate Event.....	249
4.14.8	CursorPositionChanged Event.....	250
4.14.9	Expand Events	250
4.14.9.1	ExpandedAll Event	250
4.14.9.2	ExpandingAll Event	251
4.14.10	Indicator Margin Events.....	252
4.14.10.1	IndicatorMarginClick Event.....	252
4.14.10.2	IndicatorMarginDoubleClick Event	253
4.14.10.3	DrawLineMark Event	254
4.14.11	InsertModeChanged Event.....	255
4.14.12	LanguageChanged Event.....	256
4.14.13	MenuFill Event.....	257
4.14.14	Operation Events.....	257
4.14.14.1	OperationStarted Event.....	257
4.14.14.2	OperationStopped Event.....	258
4.14.15	Outlining Events	258
4.14.15.1	OutliningBeforeCollapse Event	259
4.14.15.2	OutliningBeforeExpand Event	259
4.14.15.3	OutliningCollapse Event	260
4.14.15.4	OutliningExpand Event.....	261
4.14.15.5	OutliningTooltipBeforePopup Event	262
4.14.15.6	OutliningTooltipClose Event	262
4.14.15.7	OutliningTooltipPopup Event.....	262
4.14.16	Print Events	263
4.14.16.1	PrintHeader Event	263
4.14.16.2	PrintFooter Event.....	263
4.14.17	ReadOnlyChanged Event.....	264
4.14.18	RegisteringDefaultKeyBindings Event.....	264
4.14.19	RegisteringKeyCommands Event	265
4.14.20	Save Events	265
4.14.20.1	SaveFileWithDataLoss Event.....	265
4.14.20.2	SaveStreamWithDataLoss Event.....	266
4.14.21	Scroll Events.....	266
4.14.21.1	HorizontalScroll Event	267
4.14.21.2	VerticalScroll Event	267

4.14.22 SelectionChanged Event	268
4.14.23 SingleLineChanged Event	269
4.14.24 Text Events	269
4.14.24.1 TextChanged Event	270
4.14.24.2 TextChanging Event	270
4.14.24.3 Line Modification Events	271
4.14.24.3.1 Line Changed	272
4.14.24.3.2 Line Inserted	273
4.14.24.3.3 Line Deleted	274
4.14.25 UnreachableTextFound Event	274
4.14.26 UpdateBookmarkToolTip Event	275
4.14.27 UpdateContextToolTip Event	277
4.14.28 User Margin Events	277
4.14.28.1 DrawUserMarginText Event	277
4.14.28.2 PaintUserMargin Event	277
4.14.29 WordWrapChanged Event	278
5 Frequently Asked Questions	280
5.1 How To Access the Text Associated With Individual Lines In the Selected Text Region Of the Edit Control	280
5.2 How To Change the Lexems Dynamically	281
5.3 How To Clear the Undo Buffer In Essential Edit	282
5.4 How To Convert Offset Values Into Text Range In the Edit Control	283
5.5 How To Data Bind an Edit Control To a Datasource	284
5.6 How To Disable Keyboard Shortcuts For the Edit Control	285
5.7 How To Dynamically Add Configuration Settings At Runtime	286
5.8 How To Dynamically Validate Text Using the TextChanged Event	287
5.9 How To Format Keywords In the Contents Of the Edit Control Using Configuration Settings	289
5.10 How To Get a Count Of the Number Of Occurrences Of a String In the Edit Control	290
5.11 How To Get All the ConfigLexems In the Contents Of the Edit Control	290
5.12 How To Get the Tokens In Each Line Of the Edit Control	291
5.13 How To Implement VS.NET-like XML Tag Insertion Feature Using Edit Control	292
5.14 How To Perform VS.NET-like Underlining For Offending Code In the Edit Control	293
5.15 How To Plug-in an External Configuration Dile Into the Edit Control	295

5.16	How To Programmatically Display the Code Snippets.....	295
5.17	How To Set Different Background Colors For the Lines In the Edit Control	296
5.18	How To Suspend And Resume Painting Of the Edit Control	299

1 Overview

This section covers information on Essential Edit control, its key features, prerequisites to use the control, its compatibility with various OS and browsers and finally the documentation details complimentary with the product. It comprises the following sub sections:

1.1 Introduction To Essential Edit

Essential Edit is a 100% Native .NET UI library that provides a powerful syntax coloring UI for building modern Windows applications using the Microsoft .NET framework. Our control and framework packages can be used in any .NET environment including C#, VB.NET and managed C++.

Essential Edit will greatly benefit those end users who wants to build a .NET application with syntax highlighting and code coloring for general text editing purposes.

Our Edit control can be used in the Visual Studio Editor as it is featured with syntax coloring, code grouping, outlining tooltips etc., similar to Visual Studio.

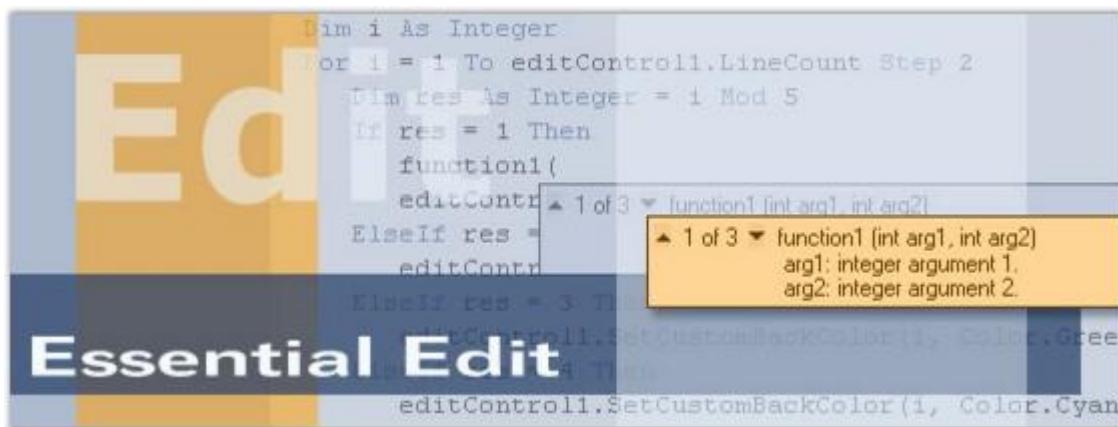


Figure 1: Essential Edit

Key Features

Some of the key features are listed below.

- Essential Edit offers fully configurable syntax highlighting and code coloring for general text editing purposes which greatly improves the readability of any text.
- Essential Edit offers advanced text indentation support which can be customized to suit the requirements of the user.

- Essential Edit supports multiple levels of Undo / Redo, whereas the default Edit control in Windows Forms supports just one level of Undo / Redo.
- Autocomplete feature auto-completes the rest of the member name once the user has entered enough characters to distinguish it, and AutoReplace Trigger feature automatically corrects some of the known predefined typing errors.
- Edit supports clipboard operations for editing the text.
- Essential Edit enables users to locate a section or a line of a document, using the Bookmarks and Custom Indicators features like in Visual Studio.
- Edit provides support for Word Wrap. You can also set images for Line and Point Wrapping.
- Essential Edit provides Visual Studio like support for collapsing and expanding blocks of code through the use of collapser (plus-minus buttons).
- Sections of code which form the outlining blocks can be specified using the Configuration Settings. Edit control defines different brackets for highlighting different languages.
- Essential Edit provides Printing support. It includes printing options to print a page, print a selection, print an entire document, print a current page and print only a selected set of pages.
- In the age of globalization the markets for all goods become more and more internationalized, enforcing the need to provide information in a variety of languages. The edit control supports complete localization to any desired language of all the dialogs and strings associated with it.
- Edit control offers support for text navigation at character, word, line, page or entire document levels. It also offers support for text manipulation operations and multiline insertion operations.
- Edit supports interactive features like outlining ToolTip, which is built-in and appears automatically when the mouse pointer is placed over the collapsed block of text.

User Guide Organization

The product comes with numerous samples as well as an extensive documentation to guide you. This User Guide provides detailed information on the features and functionalities of the Edit control. It is organized into the following sections:

- **Overview**-This section gives a brief introduction to our product and its key features.
- **Installation and Deployment**-This section elaborates on the install location of the samples, license etc.
- **What's New**-This section lists the new features implemented for every release.
- **Getting Started**-This section guides you on getting started with Windows application, controls etc.
- **Concepts and Features**-The features of the Edit control is illustrated with use case scenarios, code examples and screen shots under this section.
- **Frequently Asked Questions**-This section illustrates the solutions for various task-based queries about Essential Edit.

Document Conventions

The conventions listed below will help you to quickly identify the important sections of information, while using the content:

Convention	Icon	Description
Note	Note:	Represents important information
Example	Example	Represents an example
Tip		Represents useful hints that will help you in using the controls/features
Additional Information		Represents additional information on the topic

1.2 Prerequisites and Compatibility

This section covers the requirements mandatory for using Essential Edit. It also lists operating systems and browsers compatible with the product.

Prerequisites

The prerequisites details are listed below:

Development Environments	<ul style="list-style-type: none">• Visual Studio 2013• Visual Studio 2012• Visual Studio 2010 (Ultimate and Express)• Visual Studio 2008 (Team, Professional, Standard and Express)• Visual Studio 2005 (Team, Professional, Standard and Express)• Borland Delphi for .NET• SharpCode
.NET Framework versions	<ul style="list-style-type: none">• .NET 4.5• .NET 4.0• .NET 3.5• .NET 2.0

Compatibility

The compatibility details are listed below:

Operating Systems	<ul style="list-style-type: none">Windows 8.1 (32 bit and 64 bit)Windows Server 2008 (32 bit and 64 bit)Windows 7 (32 bit and 64 bit)Windows Vista (32 bit and 64 bit)Windows XPWindows 2003
-------------------	---

1.3 Documentation

Syncfusion provides the following documentation segments to provide all necessary information for using Essential Edit control for Windows application in an efficient manner.

Type of documentation	Location
Readme	[drive:]Program Files\Syncfusion\Essential Studio\x.x.x\Infrastructure\Data\Release Notes\readme.htm
Release Notes	[drive:]Program Files\Syncfusion\Essential Studio\x.x.x\Infrastructure\Data\Release Notes\Release Notes.htm
User Guide (this document)	Online http://help.syncfusion.com/resources (Navigate to the Edit for Windows Forms User Guide.)  Note: Click Download as PDF to access a PDF version. Installed Documentation Dashboard -> Documentation -> Installed Documentation.
Class Reference	Online http://help.syncfusion.com/resources (Navigate to the Windows Forms User Guide. Select <i>Edit</i> in the second text box, and then click the Class Reference link found in the upper right section of the page.) Installed Documentation

	Dashboard -> Documentation -> Installed Documentation.
--	--

2 Installation and Deployment

This section covers information on the install location, samples, licensing, patches update and updation of the recent version of Essential Studio. It comprises the following sub-sections:

2.1 Installation

For step-by-step installation procedure for the installation of Essential Studio, refer to the Installation topic under Installation and Deployment in the Common UG.

See Also

For licensing, patches and information on adding or removing selective components, refer the following topics in Common UG under Installation and Deployment.

- Licensing
- Patches
- Add / Remove Components

2.2 Sample and Location

This section covers the location of the installed samples and describes the procedure to run the samples through the sample browser. It also lists the location of source code.

Sample Installation Location

The Essential Edit Windows Forms samples are installed in the following location.

***...\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0***

Viewing Samples

To view the samples, follow the steps below:

1. Click **Start-->All Programs-->Syncfusion-->Essential Studio <version number> -->Dashboard.**

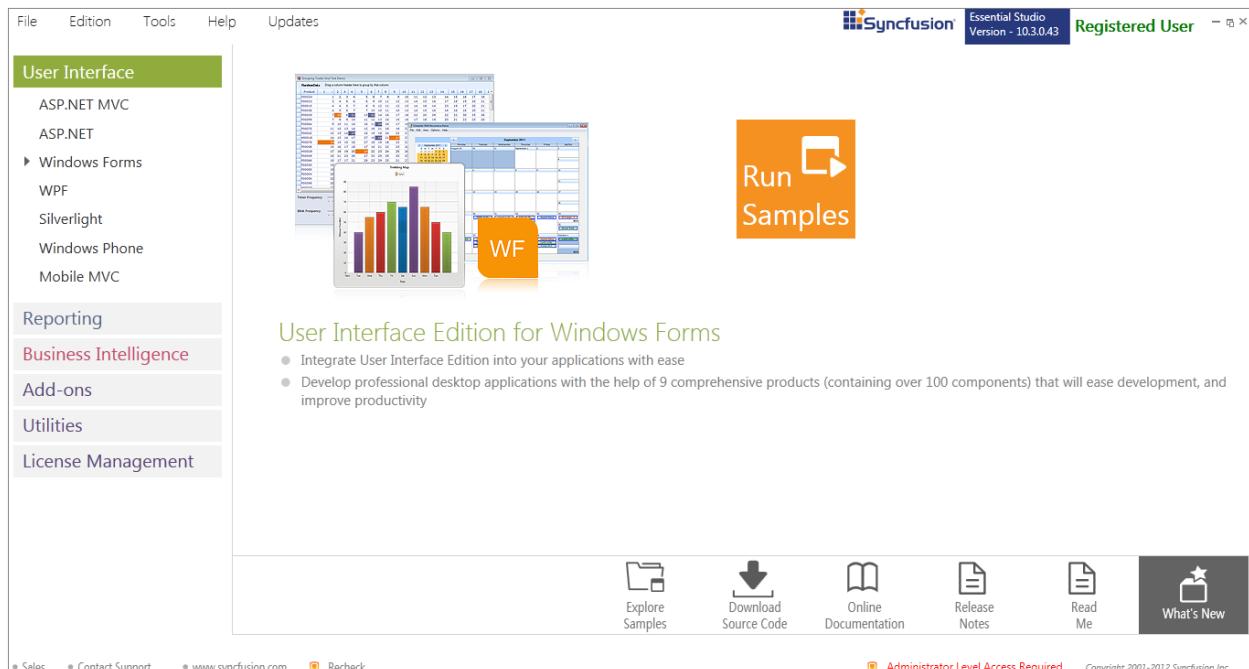


Figure 2: Syncfusion Essential Studio Dashboard

2. In the Dashboard window, click **Run Samples** for Windows Forms under UI Edition. The UI Windows Form Sample Browser window is displayed.



Note: You can view the samples in any of the following three ways:

- **Run Samples**-Click to view the locally installed samples
- **Online Samples**-Click to view online samples
- **Explore Samples**-Explore BI Web samples on disk

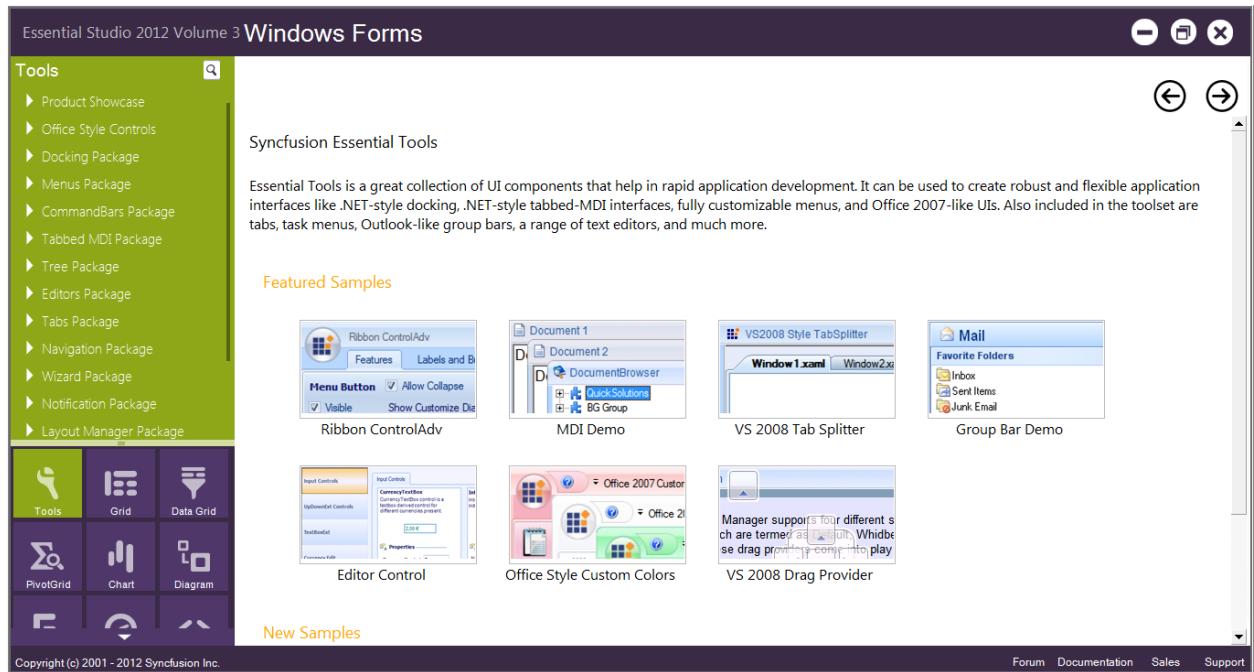


Figure 3: User Interface Edition Windows Forms Sample Browser

- To view the samples of Edit control, click Edit from the bottom-left pane.

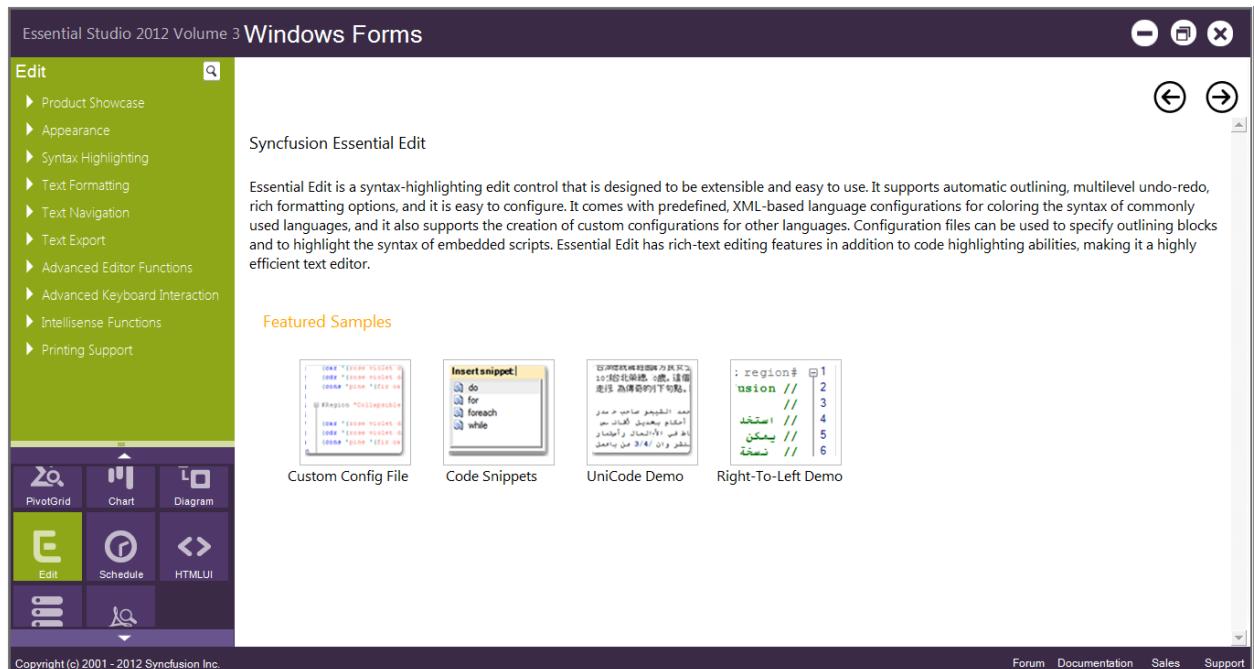


Figure 4: Essential Edit Samples for Windows

- Select any sample and browse through the features.

Source Code Location

The source code for Edit Windows is available under the following default location:

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\Windows>Edit.Windows\Src

2.3 Deployment Requirements

This section gives the deployment requirements for using Essential Edit in a Windows application. It comprises the below sections:

2.3.1 Toolbox Entries

Essential Edit places the following control into your VisualStudio .NET toolbox from where you can drag onto the form and start working with it.

- EditControl

2.3.2 DLLs

While deploying an application that references a Syncfusion Essential Edit assembly, the following dependencies must be included in the distribution.

- Syncfusion.Core.dll
- Syncfusion.Shared.Base.dll
- Syncfusion.Shared.Windows.dll
- Syncfusion.Edit.Windows.dll

3 Getting Started

This section guides you on getting started with Windows application, controls etc. It comprises the following topics:

3.1 Control Structure

The following screen shot illustrates the structure of the Edit Control.

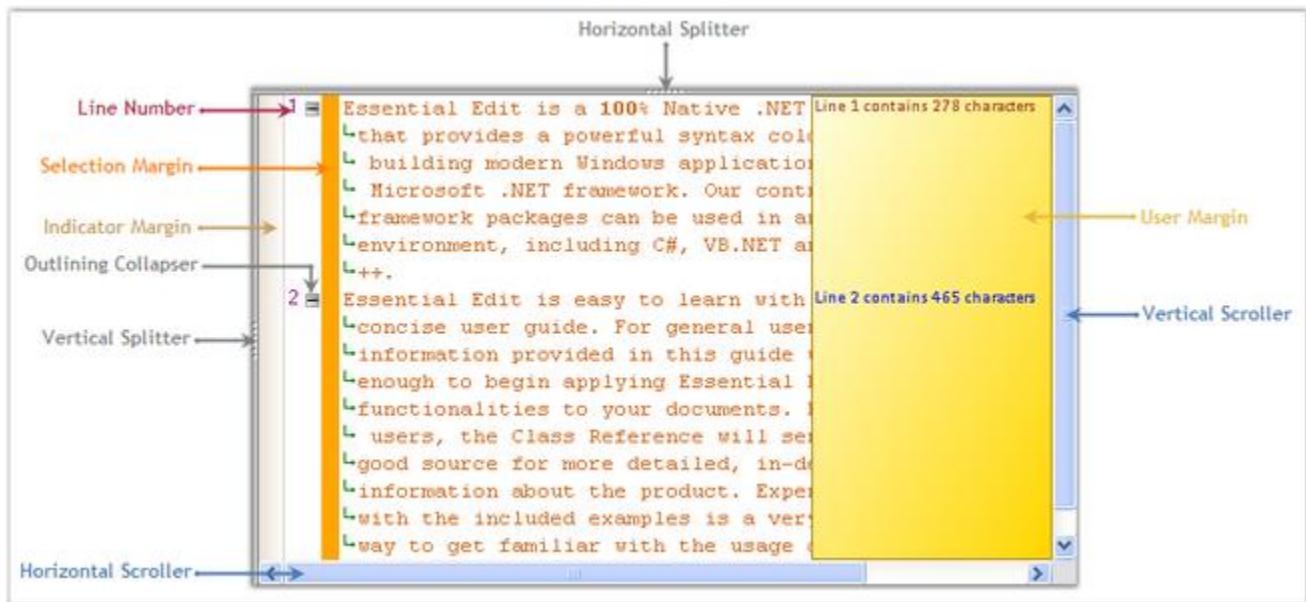


Figure 5: Structure of EditControl

3.2 Creating an Edit Control

Essential Edit provides users with a powerful editing control, modeled after the VS.NET code editor. When complete, Edit will have almost all the code editing features available in VS.NET.

In the following sections, you will learn how to create an Edit Control and use it in windows application.

3.2.1 Through Designer

The following steps illustrate how to create an Edit Control through designer.

1. Open the host form (or user control) within the Visual Studio.NET designer, and drag the Edit Control from the VS.NET toolbox onto the form. This will create an instance of the Edit Control and add it to the form at the desired location.

The following image shows Edit Control in the toolbox.



Figure 6: Edit Control in Toolbox

2. Customize the Edit Control as per your requirements by setting the properties of the control through the Properties grid.
3. Run the application.

The following illustration shows Edit Control created through designer.



Figure 7: Edit Control created Through Designer

See Also

[Through Code](#)

3.2.2 Through Code

The following steps illustrate how to create an Edit Control programmatically.

1. Import the Edit Control package in your application for easier coding experience.

[C#]

```
using Syncfusion.Windows.Forms.Edit;
```

[VB .NET]

```
Imports Syncfusion.Windows.Forms.Edit
```

2. Create an instance of the Edit Control.

[C#]

```
private Syncfusion.Windows.Forms.Edit>EditControl editControl1;  
editControl1 = new Syncfusion.Windows.Forms.Edit>EditControl();
```

[VB .NET]

```
Private editControl1 As Syncfusion.Windows.Forms.Edit>EditControl  
editControl1 = New Syncfusion.Windows.Forms.Edit>EditControl()
```

3. Set an appropriate size for the Edit Control.

[C#]

```
editControl1.Size = new Size(50, 50);
```

[VB.NET]

```
editControl1.Size = New Size(50, 50)
```

4. Set the **Dock** property to the appropriate DockStyle enumeration value if desired.

[C#]

```
editControl1.Dock = DockStyle.Fill;
```

[VB.NET]

```
editControl1.Dock = DockStyle.Fill
```

5. Set an appropriate BorderStyle to the Edit Control instance.

[C#]

```
editControl1.BorderStyle = BorderStyle.Fixed3D;
```

[VB.NET]

```
editControl1.BorderStyle = BorderStyle.Fixed3D
```

6. Add this instance of the Edit Control to the Host Form or an UserControl.

[C#]

```
// Adding instance of the EditControl to the Host Form.  
this.Controls.Add(editControl1);
```

[VB.NET]

```
' Adding instance of the EditControl to the Host Form.  
Me.Controls.Add(editControl1)
```

7. Run the application.

The following illustration shows Edit Control created through code.



Figure 8: Edit Control created Through Code

See Also

[Through Designer](#)

4 Concepts And Features

Essential Edit is designed to be an extensible and easy to use syntax highlighting Edit Control, equipped with a powerful set of features such as, syntax highlighting, undo/redo actions, Visual Studio.NET style collapsible editing, easy configuration, and more features. Essential Edit is closely modeled on the Visual Studio.NET editor and it incorporates almost all its features.

Essential Edit supports custom configuration files, and hence you can syntax highlight any piece of code or text according to any desired highlighting settings.

This section covers the following:

4.1 Configuration Settings

With careful design and implementation, Essential Edit is built to be flexible and easy-to-use. To configure Essential Edit for an application, only a configuration file is needed. The configuration settings contain sections that control various customizations such as rendering colors for keywords, token-based segments for comments, strings and more.

It comprises the following topics:

4.1.1 Creating a Custom Language Configuration File

The following code snippet illustrates a sample XML-based configuration file.

```
[XML]  
  
<?xml version="1.0" encoding="utf-8" ?>  
<ArrayOfConfigLanguage>  
  <ConfigLanguage name="default_language">  
    <formats>  
      <format name="Text" Font="Courier New, 10pt" FontColor="Black" />  
      <format name="SelectedText" Font="Courier New, 10pt"  
BackColor="Highlight" FontColor="HighlightText" />  
    </formats>
```

```

<extensions />
<lexems />
<splits />
</ConfigLanguage>
<ConfigLanguage name="C#">
  <formats>
    <format name="Text" Font="Courier New, 10pt" FontColor="Black" />
    <format name="SelectedText" Font="Courier New, 10pt"
BackColor="Highlight" FontColor="HighlightText" />
    <format name="Whitespace" Font="Courier New, 10pt" FontColor="Black" />
    <format name="KeyWord" Font="Courier New, 10pt" FontColor="Blue" />
  </formats>
  <extensions>
    <extension>cs</extension>
  </extensions>
  <lexems>
    <lexem BeginBlock="public" Type="KeyWord" />
  </lexems>
  <splits>
    <split>+=</split>
  </splits>
  </ConfigLanguage>
</ArrayOfConfigLanguage>

```

From the code given above, the configuration file contains a set of language configurations. Every configuration file must have configuration for the language named **default_language**, which is used as a default configuration.

Language Configuration (ConfigLanguage) Definition

Name of the language must be set using the **name** attribute of the ConfigLanguage tag. If language is case insensitive, you should set the **CaseInsensitive** attribute to 'True'.

Language configuration is divided into the following four sections:

- Extensions
- Splits
- Formats
- Lexems
- **Extensions**-Contains a list of extensions that are associated with this language. Every extension can be specified like the following:

```
<extension>cs</extension>
```

- **Splits**-Contains a list of expressions that must be treated as one word. By default, "=" and "+" are splitters; So each of them will be returned by the tokenizer as a single char. But if you want to specify some configuration for "+=", you should specify "+=" in the Splits section. To do this, just add the below string to the Splits section:

```
<split>+=</split>
```

- **Formats**-Contains a list of definitions of the formats that can be used later in lexem configuration. Every format is specified by a tag `<format>`. Every format contains the attributes such as name, font, fore color, font color, back color, style, weight, underline and line color.
- **Name**-Specifies the name of the format. **SelectedText** is always used for selection (if fontcolor is not specified, selected text is drawn with its own color; only the background is changed).
- **Font**-String with XML representation of the font. Refer to the default configuration file for examples.
- **ForeColor**-Specifies the color of the rectangle that is drawn around the text. It is not drawn if fore color is not specified.
- **FontColor**-Specifies the color of the text.
- **BackColor**-Specifies the background color of the text.
- **Style**-Specifies the fill style of the background. Look at the HatchStyle enumeration members for the list of possible values.
- **Weight**-Weight of the underlining. Possible values: Thick, Bold, Double, and Double Bold.
- **Underline**-Underline style. Possible values: None, Solid, DashDot, and Wave.
- **LineColor**-Color of the underlining.
- **Lexems**-Contains rules for parsing text. In other words, rules for setting lexem format. There are two attributes to specify the format of the lexem: **Type** and **FormatName**. FormatName is used only if Type is 'Custom'. Type is used for standard predefined types of lexems, some of them have special meaning for controls (such as SelectedText). For a list of possible values. Refer to the definition of the FormatType enumeration.

The simplest case of lexem definition looks like the following:

[XML]

```
<lexem BeginBlock="public" Type="KeyWord" />
```

It means that the word public will be drawn using the **KeyWord** format setting. For non-complex lexems, you can specify **ContinueBlock** and **EndBlock** attributes.

- If you specify ContinueBlock, the parser will read words (tokens) and set the specified formatting for them until it encounters a ContinueBlock.

- If you specify EndBlock, specified formatting will be set only if first token matches ContinueBlock and is followed by EndBlock.

All matched text will be treated later as one word, and won't be broken into parts in WordWrap mode.

If you want to use regular expressions in [Begin / Continue / EndBlock], you should set IsBeginRegex/IsContinueRegex/IsEndRegex to 'True'.

Example

[XML]

```
<lexem BeginBlock="$" EndBlock="^ [0-9a-fA-F]+$" IsEndRegex="true"
Type="Number" />
```

In Delphi file parsing, numbers in hexadecimal format like \$54df54af will be treated as one word.

If the **IsComplex** attribute is set to 'True', and the token matches the BeginBlock of the lexem, then the lexem found is inserted into the stack. At the start, the stack contains only language, so the parser checks only for children of the `<lexems>` tag. Configuration for the token is always searched among sub-lexems of the last lexem in the stack. If the configuration is not found, a search is done among the sub-lexems of the second lexem in the stack, and so on. This feature can be disabled by setting the **OnlyLocalSublexems** attribute to 'True', and the token will be colored like the last lexem from the stack. If the configuration is still not found, the parser checks if it is the EndBlock of the last lexem in the stack, and if it matches, the token is formatted accordingly and the lexem is removed from the stack. If the token is the **EndBlock**, and the **IsPseudoEnd** attribute is set to 'True', the lexem is removed from the stack, but the search process for that token does not stop. Refer to the sample code below.

To parse a C# string, a typical lexem would be as shown below:

[XML]

```
<lexem BeginBlock="\"" EndBlock="\"" Type="String" IsComplex="true"
OnlyLocalSublexems="true">
<SubLexems>
    <lexem BeginBlock="\\" EndBlock="\"" Type="String" />
</SubLexems>
</lexem>
```

To collapse complex lexems, set **IsCollapsible** to 'True'. **CollapseName** property specifies the text to be set instead of the collapsed construction. To make the C# string collapsible, you should use the following code:

[XML]

```
<lexem BeginBlock=""" EndBlock=""" Type="String" IsComplex="true"
OnlyLocalSublexems="true" IsCollapsable="true" CollapseName="String">
<SubLexems>
<lexem BeginBlock="\\" EndBlock=""" Type="String" />
</SubLexems>
</lexem>
```

Loading a Config File

To load a Config file to the Edit Control, use the following code snippet.

[C#]

```
this.editControl1.Configurator.Open(string fileName);
```

[C#]

```
Me.editControl1.Configurator.Open(String fileName)
```

See Also

[Creating Configuration Settings Programmatically](#)

4.1.2 Creating Configuration Settings Programmatically

Edit Control offers rich set of APIs to create configuration settings in code. This provides greater flexibility so that users can dynamically modify configuration settings of the currently loaded configuration as per their requirements. The following procedure will walk you through the entire process of creating configuration settings programmatically.

1. A new configuration language can be added to the Edit Control by using the **CreateLanguageConfiguration** method. Once the new configuration language is created, apply it to the contents of the Edit Control by using the **ApplyConfiguration** method.

[C#]

```
// Create a new configuration language and apply the same to the contents of  
the Edit Control.  
IConfigLanguage currentConfigLanguage =  
this.editControl1.Configurator.CreateLanguageConfiguration(newConfigLanguage)  
;  
this.editControl1.ApplyConfiguration(currentConfigLanguage);
```

[VB.NET]

```
' Create a new configuration language and apply the same to the contents of  
the Edit Control.  
Dim currentConfigLanguage As IConfigLanguage =  
Me.editControl1.Configurator.CreateLanguageConfiguration(NewConfigLanguage)  
Me.editControl1.ApplyConfiguration(currentConfigLanguage)
```

2. Create a custom format object by using the **Language.Add** method of the Edit Control and define its attributes.

[C#]

```
// Creating a custom format object.  
ISnippetFormat formatMethod = this.editControl1.Language.Add("CodeBehind");  
  
// Defining its attributes.  
formatMethod.FontColor = Color.IndianRed;  
formatMethod.Font = new Font("Garamond", 12);  
formatMethod.BackColor = Color.Yellow;
```

[VB.NET]

```
' Creating a custom format object.  
Dim formatMethod As ISnippetFormat =  
Me.EditControl1.Language.Add("CodeBehind")  
  
' Defining its attributes.  
formatMethod.FontColor = Color.IndianRed  
formatMethod.Font = New Font("Garamond", 12)  
formatMethod.BackColor = Color.Yellow
```

3. Create a **ConfigLexem** object that belongs to the above defined format and define its attributes.

[C#]

```
// Creating a ConfigLexem object that belongs to the above defined format.
ConfigLexem configLex = new ConfigLexem("<%@", "%>", FormatType.Custom,
false);

// Defining its attributes.
configLex.IsBeginRegex = false;
configLex.IsEndRegex = false;
configLex.ContinueBlock = ".+";
configLex.IsContinueRegex = true;
configLex.FormatName = "CodeBehind";
```

[VB.NET]

```
// Creating a ConfigLexem object that belongs to the above defined format.
Dim configLex As ConfigLexem = New ConfigLexem("<%", "%>", FormatType.Custom,
False)

' Defining its attributes.
configLex.IsBeginRegex = False
configLex.IsEndRegex = False
configLex.ContinueBlock = ".+"
configLex.IsContinueRegex = True
configLex.FormatName = "CodeBehind"
```

4. Add the **ConfigLexem** object to the **Lexems** collection of the current language.

[C#]

```
this.editControl1.Language.Lexems.Add(configLex);
```

[VB.NET]

```
Me.editControl1.Language.Lexems.Add(configLex)
```

5. Add the appropriate splits and extensions to the **Language.Splits** and **Language.Extensions** collections.

[C#]

```
// Adding the necessary split definitions to the current language's Splits
collection.
this.editControl1.Language.Splits.Add("<%@");
this.editControl1.Language.Splits.Add("%>");

// Adding the necessary extension definitions to the current language's
Extensions collection.
```

```
this.editControl1.Language.Extensions.Add("aspx");
```

[VB .NET]

```
' Adding the necessary split definitions to the current language's Splits
collection.
Me.EditControl1.Language.Splits.Add("<%")
Me.EditControl1.Language.Splits.Add("%>")

' Adding the necessary extension definitions to the current language's
Extensions collection.
Me.EditControl1.Language.Extensions.Add("aspx")
```

6. Invoke the **ResetCaches** method to apply these newly added configuration settings.

[C#]

```
// Reset the current configuration language cache to reflect these changes.
this.editControl1.Language.ResetCaches();
```

[VB .NET]

```
' Reset the current configuration language cache to reflect these changes.
Me.editControl1.Language.ResetCaches()
```

See Also

[Creating a Custom Language Configuration File](#)

4.2 Editing Features

Essential Edit comes with advanced editing features. Some of the important features discussed in this section are given below.

4.2.1 Undo / Redo Actions

Action Grouping allows you to specify a set of actions as groups for **Undo** / **Redo** purposes. When an action group is created, and a set of actions is added to it, the entire set is considered as one entity. This implies that the set of actions can be performed or undone using the **Redo** or **Undo** method call. You can use the **UndoGroupOpen**, **UndoGroupClose** and **UndoGroupCancel** methods to programmatically manipulate the undo / redo action grouping.

Grouping Actions

To undo/redo an action group, do the following steps:

1. Invoke the **UndoGroupOpen** method to begin a new action group.
2. Perform any desired set of actions, and invoke the **UndoGroupClose** method to close the action group. All the actions performed between the **UndoGroupOpen()** and **UndoGroupClose()** method calls get grouped as one entity.
3. Now, when the **Undo** / **Redo** methods are invoked, the newly created group (or set of actions) gets undone / redone appropriately.
4. To cancel an already open action group, you have to invoke the **UndoGroupCancel** method.
5. The **CanUndo** property gets a flag that determines whether the undo operation can be performed in the Edit Control.
6. The **CanRedo** property gets a flag that determines whether the redo operation can be performed in the Edit Control.

Unlimited Undo and Redo

Essential Edit supports multiple levels of undo / redo, whereas the default Edit Control in Windows Forms supports just one level of undo / redo. This makes Essential Edit a better choice for all your editing needs. The ability to undo and redo changes in Essential Edit improves the usability of any application that has any form of text editing.

Essential Edit allows the following methods to be invoked any number of times.

Edit Control Method	Description
Undo	Performs an undo operation. (CTRL+Z)
Redo	Performs a redo operation. (CTRL+Y)
CanUndo	Indicates whether it is possible to undo the actions in the Edit Control.
CanRedo	Indicates whether it is possible to redo the actions in the Edit Control.

ResetUndoInfo	Clear the undo buffer. Hence undo operation is not allowed on contents/actions previously added/Performed up to that point.
---------------	---



Note: The undo/redo buffer is cleared after the 'Save' operation.

Enabling Grouping

Grouping is enabled using the below given property.

Edit Control Property	Description
GroupUndo	Specifies whether grouping should be enabled for undo/redo actions.

[C#]

```
// Accomplish Undo operation.
this.editControl1.Undo();

// Accomplish Redo operation.
this.editControl1.Redo();

// Indicates whether it is possible to Undo in the Edit Control.
bool canUndo = this.editControl1.CanUndo;

// Indicates whether it is possible to Redo in the Edit Control.
bool canRedo = this.editControl1.CanRedo;

// Clears the Undo buffer.
this.editControl1.ResetUndoInfo();

// Enable grouping for Undo / Redo actions.
this.editControl1.GroupUndo = true;
```

[VB .NET]

```
' Accomplish Undo operation.
Me.editControl1.Undo()

' Accomplish Redo operation.
Me.editControl1.Redo()

' Indicates whether it is possible to Undo in the Edit Control.
```

```
Dim canUndo as bool = Me.editControl1.CanUndo  
  
' Indicates whether it is possible to Redo in the Edit Control.  
Dim canRedo as bool = Me.editControl1.CanRedo  
  
' Clears the Undo buffer.  
Me.editControl1.ResetUndoInfo()  
  
' Enable grouping for Undo / Redo actions.  
Me.editControl1.GroupUndo = True
```

The following screen shot shows action grouping in Edit Control.

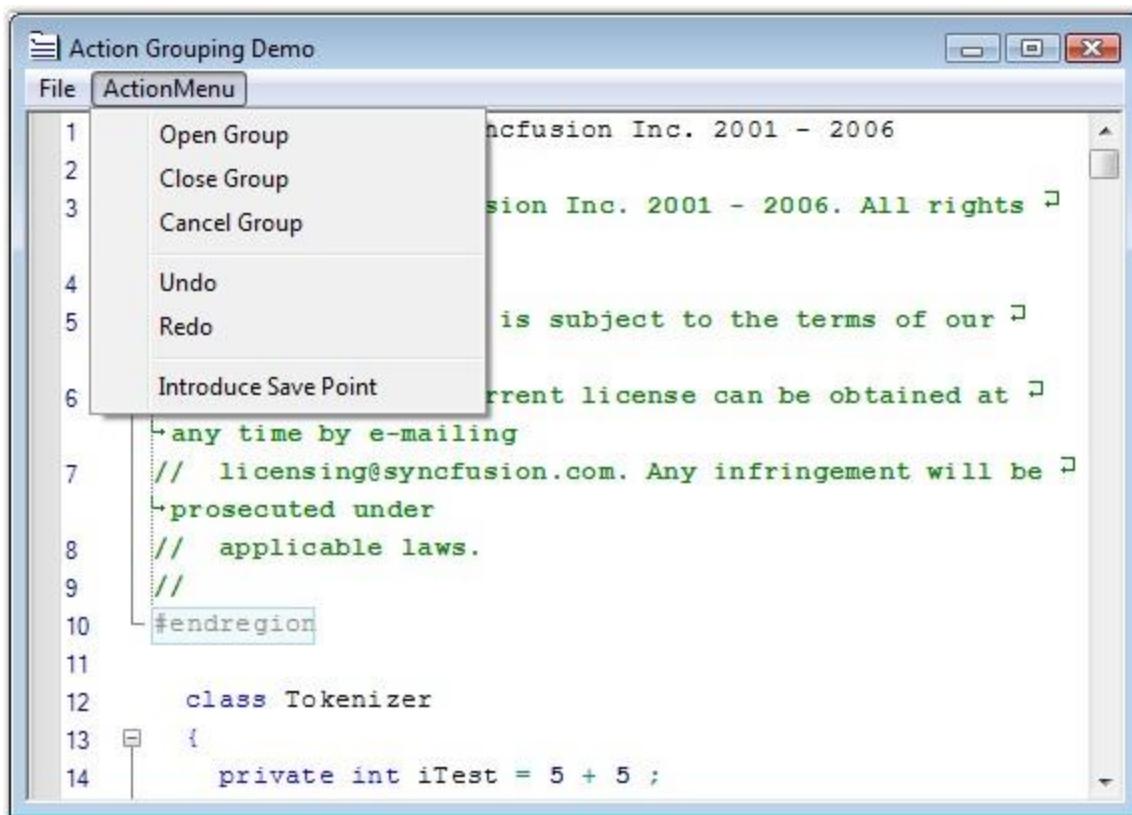


Figure 9: Grouping Actions in Edit Control

A sample which demonstrates Action Grouping is available in the following sample installation location.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\ActionGroupingDemo**

4.2.2 New Line Styles

Edit Control allows you to specify a new line style, or get the currently used new line style in the text.

SetNewLineStyle method sets the current new line style in the Edit Control. SetNewLineStyle method accepts values from the **NewLineStyle** enumerator which has values like Windows, Mac, Unix and Control, which correspond to new line styles "\\r\\n", "\\r", "\\n\\r" and "\\n\\r" respectively.

Similarly, the **GetNewLineStyle** method returns a NewLineStyle enumerator value which indicates the currently used new line style in the Edit Control.



Note: The default new line style value is set to 'Control'. This value can be changed according to the needs of the user using the DefaultNewLineStyle property.

[C#]

```
// Change the current new line style in the Edit Control.  
this.editControl1.SetNewLineStyle(Syncfusion.IO.NewLineStyle.Control);  
this.editControl1.GetNewLineStyle();  
  
// Specify the default new line style.  
this.editControl1.DefaultNewLineStyle = Syncfusion.IO.NewLineStyle.Windows;
```

[VB .NET]

```
' Change the current new line style in the Edit Control.  
Me.editControl1.SetNewLineStyle(Syncfusion.IO.NewLineStyle.Control)  
Me.editControl1.GetNewLineStyle()  
  
' Specify the default new line style.  
Me.editControl1.DefaultNewLineStyle = Syncfusion.IO.NewLineStyle.Windows
```

4.2.3 Clipboard Operations

Edit Control uses the clipboard to cut, copy or paste the text data. It stores the data in the clipboard for cut and copy operations and retrieves data from the clipboard for paste operations. The following APIs in the Edit Control facilitates these clipboard operations.

Edit Control Method	Description
Copy	Copies the selected text contents into the clipboard.
Cut	Cuts the selected text contents from Edit Control and places it into the clipboard.
Paste	Retrieves copied contents from the clipboard and pastes it into Edit Control.
CanCopy	Indicates whether it is possible to perform copy operations in Edit Control.
CanCut	Indicates whether it is possible to perform cut operations in Edit Control.
CanPaste	Indicates whether it is possible to perform copy, cut and paste operations in Edit Control.
ClearClipboard	Clears all contents in the clipboard associated with Essential Edit. This is generally used immediately after the application loads, to clear any junk from previous clipboard operations.

[C#]

```
// Copies the selected text into the clipboard.
this.editControl1.Copy();

// Cuts the selected text contents from Edit Control and places it into the
// clipboard.
this.editControl1.Cut();

// Retrieves copied contents from the clipboard and pastes it into Edit
// Control.
this.editControl1.Paste();

// Indicates whether it is possible to perform copy operation in Edit
// Control.
bool canCopy = this.editControl1.CanCopy;

// Indicates whether it is possible to perform cut operation in Edit Control.
bool canCut = this.editControl1.CanCut;
```

```
// Indicates whether it is possible to perform paste operation in Edit Control.  
bool canPaste = this.editControl1.CanPaste;  
  
// Clears all contents in the clipboard associated with Essential Edit.  
this.editControl1.ClearClipboard();
```

[VB.NET]

```
' Copies the selected text into the clipboard.  
Me.editControl1.Copy()  
  
' Cuts the selected text contents from Edit Control and places it into the clipboard.  
Me.editControl1.Cut()  
  
' Retrieves copied contents from the clipboard and pastes it into Edit Control.  
Me.editControl1.Paste()  
  
' Indicates whether it is possible to perform copy operation in Edit Control.  
Dim canCopy as bool = Me.editControl1.CanCopy  
  
' Indicates whether it is possible to perform cut operation in Edit Control.  
Dim canCut as bool = Me.editControl1.CanCut  
  
' Indicates whether it is possible to perform paste operation in Edit Control.  
Dim canPaste as bool = Me.editControl1.CanPaste  
  
' Clears all contents in the clipboard associated with Essential Edit.  
Me.editControl1.ClearClipboard()
```

4.2.3.1 EnableMD5

The EditControl is mainly based on the MD5 algorithm. By default, **EnableMD5** property is enabled in EditControl.

FIPS

The system's cryptography is based on the FIPS compliant algorithms for encryption, hashing and security.

When FIPS is enabled, the Clipboard Operations of EditControl are affected as EditControl uses the MD5 algorithm. To avoid this, before enabling FIPS, you must disable the EditControl's MD5 algorithm by setting the **EnableMD5** property to *false*.

Property	Description
EnableMD5	Specifies whether to enable or disable MD5 algorithm.

[C#]

```
this.editControl1.EnableMD5 = true;
```

[VB .NET]

```
Me.editControl1.EnableMD5 = True
```

**Note:**

To enable FIPS:

1. Click **Start**, click **Control Panel**, and the click on **Administrative Tools**.
2. Double-click **Local Security Policy**.
3. Double-click **Local Policies**.
4. Click **Security Options**. Under **Policies** listed in the right pane, double-click **System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing**.
5. Select **Enabled** to enable FIPS on your machine.

4.2.4 Keystroke - Action Combinations Binding

Edit Control offers support for the action-keystroke binding functionality, providing you the ability to perform advanced customization of action-keystroke bindings to suit your preferences. You can bind any desired keystroke combination to a standard (or custom) command like Copy, Cut, Paste or Find in the designer using the **Keys Binding** dialog as illustrated in the following procedure:

1. In the Editor Keys Binding dialog box, select the desired standard command. The default shortcuts assigned for a particular command are listed in the combobox under the **Shortcut(s) for selected command:** label.
2. Set the focus to the Edit Box Press TAB to navigate to the shortcuts drop-down list.
3. Press the desired key or key combination.
4. Now, click the Assign button, to assign this keystroke combination as the shortcut for that particular standard command. Click OK.

The **KeyBinder** property is used to get the key binder, and the **KeyBindingProcessor** property is used to get / set the key binding processor.

The Editor Keys Binding dialog is invoked using the **ShowKeysBindingEditor** method of the Edit Control.

The following illustration shows the Keys Binding dialog box.

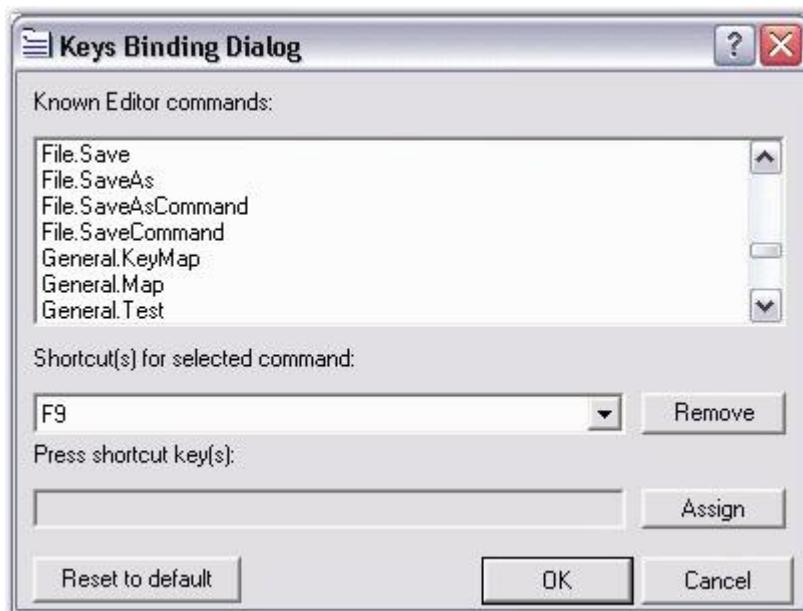


Figure 10: Preview of Keys Binding Dialog Box

You can also make use of the **RegisteringKeyCommands** and **RegisteringDefaultKeyBindings** events to specify user-defined commands and bind the desired custom keystroke combinations to them.

This following code snippet registers the "File.Open" command and binds a Ctrl+O keystroke combination to it.

```
[C#]

// Invoke the Editor Keys Binding dialog.
this.editControl1.ShowKeysBindingEditor();

// Bind the action name to the action using the RegisteringKeyCommands and
// ProcessCommandEventHanlder events.
private void this.editControl1_RegisteringKeyCommands(object sender,
EventArgs e)
{
    this.editControl1.Commands.Add( "File.Open" ).ProcessCommand += new
    ProcessCommandEventHanlder( Command_Open );
```

```
}

// Bind key combinations to the action name using the
RegisteringDefaultKeyBindings event.
private void this.editControl1_RegisteringDefaultKeyBindings(object sender,
EventArgs e)
{
this.editControl1.KeyBinder.BindToCommand( Keys.Control | Keys.O, "File.Open"
);
}

// Define the action that needs to be performed.
private void Command_Open()
{
/* Do the desired task. */
}
```

[VB.NET]

```
' Invoke the Editor Keys Binding dialog.
Me.editControl1.ShowKeysBindingEditor()

' Bind the action name to the action using the RegisteringKeyCommands and
ProcessCommandEventHanlder events.
Private Sub Me.editControl1_RegisteringKeyCommands(ByVal sender As Object,
 ByVal e As EventArgs)
    Me.editControl1.Commands.Add("File.Open").ProcessCommand += New
    ProcessCommandEventHanlder(Command_Open)
End Sub

' Bind key combinations to the action name using the
RegisteringDefaultKeyBindings event.
Private Sub Me.editControl1_RegisteringDefaultKeyBindings(ByVal sender As
Object, ByVal e As EventArgs)
    Me.editControl1.KeyBinder.BindToCommand(Keys.Control | Keys.O,
    "File.Open")
End Sub

' Define the action that needs to be performed.
Private Sub Command_Open()
    ' Do the desired task.
End Sub
```

A sample which demonstrates Keys Binding is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Keyboard
Interaction\\KeysBindingDemo**

4.2.5 Regular Expressions

Non-Deterministic Finite Automation (NFA) regular expressions are a powerful way of parsing text and are used in a wide range of products like the Microsoft .NET platform, Perl, Python, Grep (Global Regular Expression Print), VI Editor, Tcl, Awk, and various shells. Regular expressions figure into all kinds of text-manipulation tasks like searching, search-replace and can also be used to test for certain conditions in a text file or data stream.

Edit Control implements a customized regular expression engine which is capable of parsing extremely complicated languages including embedded scripts. The search and search-replace functionalities also use the regular expressions internally.

Language Elements

Edit Control offers complete support to a variety of common constructs for regular expression patterns. Refer to the [Language Elements](#) topic for more information on the regular expression pattern syntax.

Lexical Macros

Lexical macros definitions create named regular expressions that can be used to replace certain sections of the regular expression patterns. This improves the reusability of common patterns and simplifies the task of creating lexems in configuration files. Refer to the [Lexical Macros](#) topic for more information in this regard.

Regular Expressions in XML based Configuration File

There are certain regular expression command characters that must be translated to an XML compatible format while being used in an XML-based configuration file. The following is an example of a lexem tag block that has been used for outlining.

[XML]

```
<lexem BeginBlock="#region" EndBlock="#end region" Type="PreprocessorKeyword"  
IsEndRegex="true" IsComplex="true" IsCollapsible="true"  
AutoNameExpression="\s*(?<text>.*).*\n" AutoNameTemplate="${text} "
```

```

IsCollapseAutoNamed="true" CollapseName="#region">
    <SubLexems>
        <lexem BeginBlock="\n" IsBeginRegex="true" />
    </SubLexems>
</lexem>
```

Ampersands (&) can be escaped using &, less-than symbols (<) can be escaped using <, and greater-than symbols (>) can be escaped using >.

Invalid Regular Expressions

If you enter an invalid regular expression pattern in a language definition that the Edit Control cannot parse, an exception is raised with a diagnostic message describing the problem.

4.2.5.1 Language Elements

The Edit Control regular expression engine accepts an extensive set of regular expression elements that enable you to efficiently search for text patterns. This section details the set of characters, operators and constructs that you can use to define regular expressions.

Character Escapes

Most of the important regular expression language operators are unescaped single characters. The escape character \ (a single backslash) signals to the regular expression parser that the character following the backslash is not an operator. For example, the parser treats an asterisk (*) as a repeating quantifier, and a backslash followed by an asterisk (*) as the Unicode character \u002A.

Escaped Character	Description
(Ordinary characters)	Characters other than . \$ ^ { [() * + ? \ match themselves.
\a	Matches a bell (alarm) \u0007.
\t	Matches a tab \u0009.
\r	Matches a carriage return \u000D.
\v	Matches a vertical tab \u000B.
\f	Matches a form feed \u000C.

\n	Matches a new line \u000A.
\e	Matches an escape \u001B.
\040	Matches an ASCII character as octal (exactly three digits). The character \040 represents a space.
\x20	Matches an ASCII character using hexadecimal representation (exactly two digits).
\u0020	Matches a Unicode character using hexadecimal representation (exactly four digits).
\	Matches a character when followed by a character that is not recognized as an escaped character. For example, * is the same as \xA.

Character Classes

A character class is a set of characters that will find a match if any one of the characters included in the set matches. The following table summarizes the character matching syntax.

Character Class	Description
.	Matches any character except \n. When within a character class, the . will be treated as a period character.
[aeiou]	Matches any single character included in the specified set of characters.
[^aeiou]	Matches any single character not in the specified set of characters.
[0-9a-fA-F]	Use of a hyphen (-) allows specification of contiguous character ranges.
\w	Matches any word character.
\W	Matches any non-word character.
\s	Matches any whitespace character.
\S	Matches any non-whitespace character.
\d	Matches any decimal digit.

\D	Matches any non-digit.
[.\w\s]	Escaped built-in character classes such as \w and \s may be used in a character class. This example matches any period, word or whitespace character.

Quantifiers

Quantifiers add optional quantity data to a regular expression. A quantifier expression applies to the character, group, or character class that immediately precedes it. The .NET Framework regular expressions support minimal matching ("lazy") quantifiers.

The following table describes the metacharacters that affect the matching quantity.

Quantifier	Description
*	Specifies zero or more matches; for example, \w* or (abc)*. Same as {0,}.
+	Specifies one or more matches; for example, \w+ or (abc)+. Same as {1,}
?	Specifies zero or one matches; for example, \w? or (abc)? . Same as {0,1}.
{n}	Specifies exactly n matches; for example, (pizza){2}.
{n,}	Specifies at least n matches; for example, (abc){2,}.
{n,m}	Specifies at least n, but no more than m, matches.

Atomic Zero-Width Assertions

The metacharacters described in the following table do not cause the engine to advance through the string or consume characters. They simply cause a match to succeed or fail depending on the current position in the string. For instance, ^ specifies that the current position is at the beginning of a line or string. Thus, the regular expression ^#region, returns only those occurrences of the character string #region that occur at the beginning of a line.

The following table lists other regular expression constructs.

Assertion	Description
^	Specifies that the match must occur at the beginning of the string or the beginning of the line.
\$	Specifies that the match must occur at the end of the string, before \n at the end of the string, or at the end of the line.
\A	Specifies that the match must occur at the beginning of the document.
\z	Specifies that the match must occur at the end of the document.
\b	Specifies that the match must occur on a boundary between \w (alphanumeric) and \W (nonalphanumeric) characters.
\B	Specifies that the match must not occur on a \b boundary.
(?=)	Zero-width positive look ahead assertion. Continues match only if the subexpression matches at this position on the right. For example, _(?=\w) matches an underscore followed by a word character, without matching the word character.
(?!)	Zero-width negative look ahead assertion. Continues match only if the subexpression does not match at this position on the right. For example, \b(?!un)\w+\b matches words that do not begin with un.

Miscellaneous Constructs

The following table lists other regular expression constructs.

Construct	Description
" "	Encapsulates a fixed string of characters.
{ }	Provides a call to a lexical macro. The use of a WordMacro (which is similar to \w) would appear as {WordMacro}.
()	Provides a grouping construct that groups the contained regular expression elements and changes their precedence.
(?#)	Inline comment inserted within a regular expression. The comment terminates at the first closing parenthesis character.
	Provides an alternation construct that matches any one of the terms

	separated by the (vertical bar) character. For example, cat dog tiger. The leftmost successful match wins.
--	--

See Also

[Lexical Macros](#)

4.2.5.2 Lexical Macros

Edit Control allows you to define macros that represent regular expression elements. These macros are valid for use in any regular expression.

Usage

Using defined macros is easy. To reference a macro, simply type its name within curly braces ({{ ... }}). The following examples illustrate this feature better:

- This regular expression uses a macro that represents the character class [0-9] to build a decimal number regular expression.

{DigitMacro}+ (\. {DigitMacro}+)?

- This regular expression builds a C# identifier using two macros.

(_ | {AlphaMacro})({WordMacro})*

Built-In Macros

Edit Control recognizes a number of built-in macros. If a language definition defines a lexical macro of the same name as a built-in lexical macro, the user's definition will override the system definition. The following table summarizes the built-in macros of Edit Control.

Macro	Description
AllMacro	Contains all Unicode characters. This is the same as [\u0000-\uFFFF]
AlphaMacro	Contains all Unicode alphanumeric digits. This is the same as: [a-zA-Z]

DigitMacro	Contains all Unicode decimal digits. This is the same as: [0-9]
HexDigitMacro	Contains all Unicode hexadecimal digits. This is same as:[0-9a-fA-F]
LineTerminatorMacro	Contains all Unicode line terminators. This is the same as: [\r\n\u2028\u2029]
LineTerminatorWhitespaceMacro	Contains all Unicode line terminators and whitespace characters. This is the same as: [\r\n\u2028\u2029\f\t\v\x85]
NonAlphaMacro	Contains the inverse of AlphaMacro.
NonDigitMacro	Contains the inverse of DigitMacro.
NoneMacro	Contains no characters.
NonHexDigitMacro	Contains the inverse of HexDigitMacro.
NonLineTerminatorMacro	Contains the inverse of LineTerminatorMacro.
NonLineTerminatorWhitespaceMacro	Contains the inverse of LineTerminatorWhitespaceMacro.
NonWhitespaceMacro	Contains the inverse of WhitespaceMacro.
NonWordMacro	Contains the inverse of WordMacro.
WhitespaceMacro	Contains all Unicode whitespace characters. This is the same as: [\f\t\v\x85]
WordMacro	Contains all Unicode word characters. This is the same as: [0-9a-zA-Z]

The lexical macros are used to specify configuration settings, and can be added to the current configuration language settings, as shown below.

```
[C#]

// Create and add a lexical macro to the Edit Control.LexicalMacrosManager's
// collection.
// The Add method also returns the IMacro object associated with the lexical
// macro.
IMacro macro = this.EditControl1.LexicalMacrosManager.Add("testMacro",
".+");
```

```
// Consider a scenario where configuration settings are being created  
dynamically in code.  
// Create a config lexem that belongs to a custom format.  
ConfigLexem configLex = new ConfigLexem("<%@", "%>", FormatType.Custom,  
false);  
  
// The actual regex can then be substituted with the lexical macro while  
defining the config lexem.  
// NameInConfig returns the name of the macro rounded with braces, like  
"{testmacro}."  
configLex.ContinueBlock = macro.NameInConfig;  
configLex.IsContinueRegex = true;
```

[VB.NET]

```
' Create and add a lexical macro to the Edit Control.LexicalMacrosManager's  
collection.  
' The Add method also returns the IMacro object associated with the lexical  
macro.  
Dim macro As IMacro = Me.Edit Control1.LexicalMacrosManager.Add("testMacro",  
".+")  
  
' Consider a scenario where configuration settings are being created  
dynamically in code.  
' Create a config lexem that belongs to a custom format.  
Dim configLex As ConfigLexem = New ConfigLexem("<%@", "%>",  
FormatType.Custom, False)  
  
' The actual regex can then be substituted with the lexical macro while  
defining the config lexem.  
' NameInConfig returns name of the macro rounded with braces, like  
"{testmacro}."  
configLex.ContinueBlock = macro.NameInConfig  
configLex.IsContinueRegex = True
```

See Also

[Language Elements](#)

4.2.6 Block Indent and Outdent

Edit Control supports VS.NET-like Block Indent and Outdent. In other words, when a block of text is selected, and the TAB or SPACE keys are pressed, appropriate number of tabs or spaces are added to the beginning of each line in the selected block. This will move the selected section of the code by the appropriate number of tabs or spaces to the right. Similarly, when the SHIFT+TAB keys combination is pressed, the tabs or spaces added gets removed, i.e., the previous action performed by the TAB or SPACE keys gets undone. Hence, pressing the SHIFT+TAB keys combination, moves the selected text by the appropriate number of tabs or spaces, to the left.

You can also set the tab size to the desired number of spaces using the **TabSize** property of the Edit Control as shown below. By default, the TabSize property value is set to **2**.

[C#]

```
// "n" is the integer value specifying the number of spaces.  
this.editControl1.TabSize = n;
```

[VB .NET]

```
' "n" is the integer value specifying the number of spaces.  
Me.editControl1.TabSize = n
```

Indent and Outdent Text Programmatically

The following methods are used to indent and outdent text in the Edit Control.

Edit Control Method	Description
IndentText	Indents text in the specified range.
IndentSelection	Indents selected text.
OutdentText	Outdents text in the specified range.
OutdentSelection	Outdents selected text.

[C#]

```
// Indents text in the specified range.  
this.editControl1.IndentText(new Point(5, 5), new Point(10, 10));  
// Indents selected text.  
this.editControl1.IndentSelection();
```

```
// Outdents text in the specified range.  
this.editControl1.OutdentText(new Point(5, 5), new Point(10, 10));  
// Outdents selected text.  
this.editControl1.OutdentSelection();
```

[VB.NET]

```
' Indents text in the specified range.  
Me.editControl1.IndentText(New Point(5, 5), New Point(10, 10))  
' Indents selected text.  
Me.editControl1.IndentSelection()  
  
' Outdents text in the specified range.  
Me.editControl1.OutdentText(New Point(5, 5), New Point(10, 10))  
' Outdents selected text.  
Me.editControl1.OutdentSelection()
```

4.2.7 Right-To-Left (RTL) Support

Right-To-Left Support for EditControl

EditControl supports rendering content in Right-To-Left (RTL) layout.

The following features that are present in Left-To-Right layout are also supported in Right-To-Left layout:

- Line numbers, Book Marks and Selection margins
- Context Menus, ToolTips and Dialogs
- Printing and print preview
- Line borders, Underline and Text Range customization

Use Case Scenarios

With RTL support, you can use EditControl, to render content in Right-To-left layout for languages such as Arabic. This is depicted in the screenshot below:



Figure 11: Right-To-Left Layout of Arabic

Properties

Table 1: Property Table

Property	Description	Type	Data Type	Reference links
RenderRightToLeft	Gets or sets a value indicating whether to render the content of the control in RightToLeft layout.	Boolean	Boolean	

Enabling Right-To-Left in EditControl

RTL can be enabled in EditControl with the Application Programming Interface (API) **RenderRightToLeft** as given in the following codes:

[C#]

```
this.editControl1.RenderRightToLeft = true;
```

[VB .NET]

```
Me.editControl1.RenderRightToLeft = True
```

Sample Link

To view a sample:

1. Open the WPF sample browser from the dashboard.
2. Navigate to **WPF Edit -> Advanced Editor Functions -> Right-To-Left Demo.**

4.3 Code Completion

The following topics are covered under this section:

4.3.1 AutoComplete Support

Complete Word feature is a user-friendly functionality that can be used in conjunction with the Context Choice, and is analogous to the Complete Word feature in Visual Studio. This feature autocompletes the rest of the member name once you have entered enough characters to distinguish it. Type the first few letters of the member name, and then press ALT+RIGHT ARROW or CTRL+SPACEBAR keys to see this functionality.

Example

When the following text is typed - "this.editControl1.", it displays a Context Choice list with members in the following order

- New
- Word
- WordLeft
- WordRight

Case 1

If you type "w" after "this.editControl1.", such that it looks like - "this.editControl1.w", and press the ALT+RIGHT ARROW (or CTRL+SPACEBAR) keys, it will autocomplete it with the first matching member name. In this case, it will be autocompleted as "this.editControl1.Word".

Case 2

If you type "wordr" after "this.editControl1.", such that it looks like - "this.editControl1.wordr", and press the ALT+RIGHT ARROW (or CTRL+SPACEBAR) keys, it will autocomplete it with the first matching member name. In this case, it will be autocompleted as "this.editControl1.WordRight".

Case 3

If you type "move" after "this.editControl1.", such that it looks like - "this.editControl1.move", and press the ALT+RIGHT ARROW (or CTRL+SPACEBAR) keys, it will autocomplete it with the first matching member name. In this case, there is no matching member name to autocomplete, and hence nothing will happen.

Case 4

If you type nothing after "this.editControl1.", and press the ALT+RIGHT ARROW (or CTRL+SPACEBAR) keys, it will autocomplete it with the first member name in the Context Choice list. In this case, it should be autocompleted as "this.editControl1.New".

Note that the searching process for the first matching member is not case sensitive. For example, "wordr" and "WordR" will be treated in the same way.

Set the **UseAutocomplete** property associated with the **IContextChoiceController** to **True**, to enable this functionality while using Context Choice.

[C#]

```
private void  
editControl1_ContextChoiceOpen(Syncfusion.Windows.Forms.Edit.Interfaces.ICont  
extChoiceController controller)  
{  
controller.UseAutocomplete = true;  
}
```

[VB .NET]

```
Private Sub editControl1_ContextChoiceOpen(ByVal controller As  
Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController) Handles  
editControl1.ContextChoiceOpen  
    controller.UseAutocomplete = True  
End Sub
```

See Also

[AutoReplace Triggers](#)

4.3.2 AutoReplace Triggers

The Edit Control comes with the AutoReplace Triggers feature which allows the control to automatically correct some of the known predefined typing errors. AutoReplace Triggers are fired when certain keys are pressed. These keys are defined within the language definition. When the AutoReplace Trigger key is pressed, the editor checks the word before the AutoReplace Trigger, to see if it is in the AutoReplace table. If it is present, then the word is automatically replaced with its replacement word.

The AutoReplace Trigger keys are defined within the language definitions. This means that different keys can be defined as triggers for different languages.

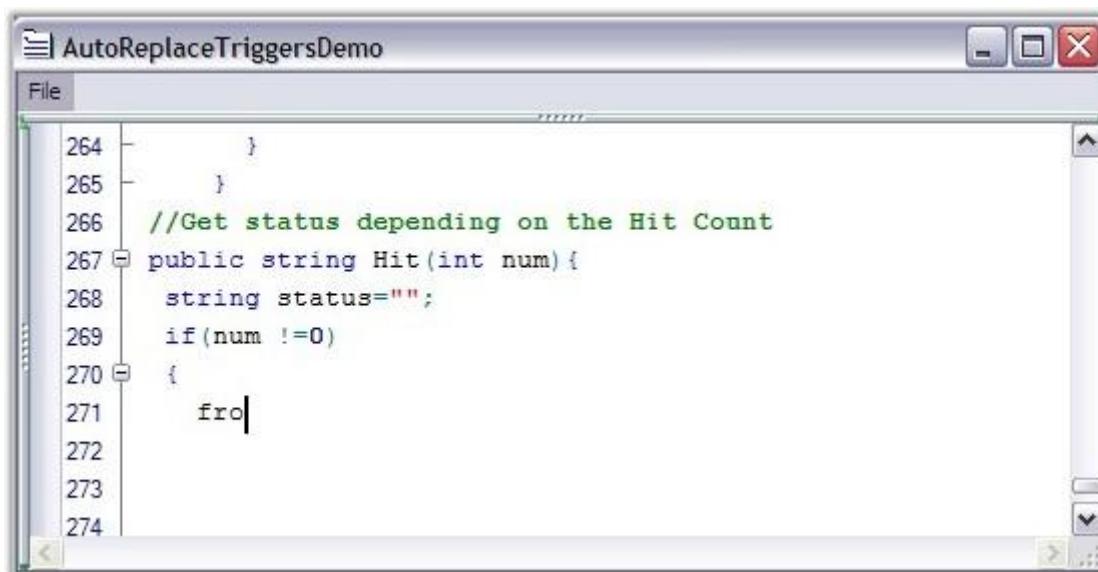


Figure 12: "for" has been incorrectly typed as "fro"

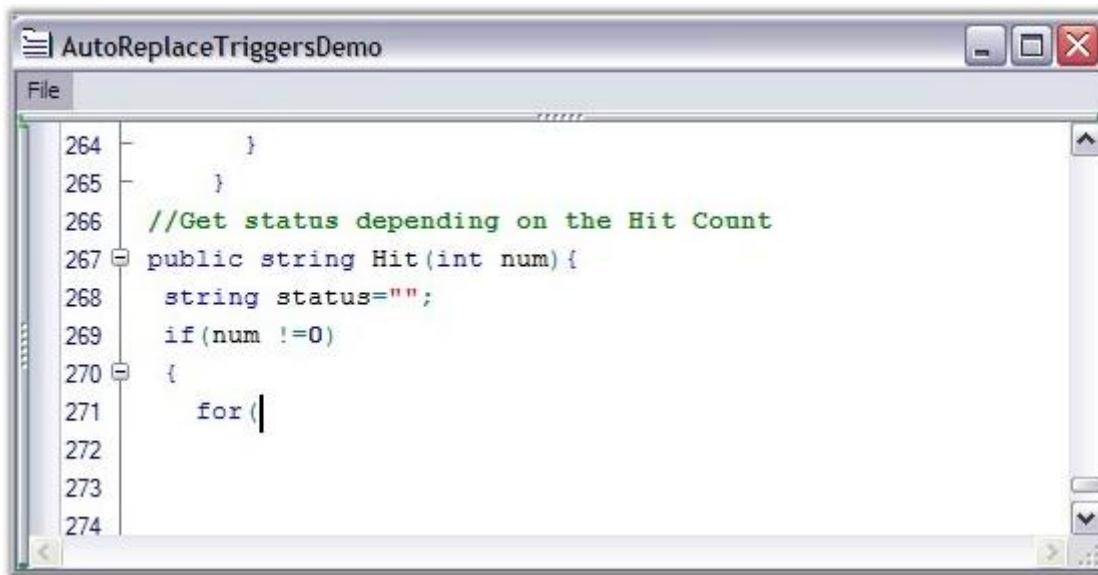


Figure 13: After typing '(' the incorrect token "fro" is replaced with the correct token "for"

AutoReplace Triggers can be enabled by using the **UseAutoreplaceTriggers** property as shown below.

[C#]

```
// Enables AutoReplace Triggers.
this.editControl1.UseAutoreplaceTriggers = true;
```

[VB.NET]

```
' Enables AutoReplace Triggers.
Me.editControl1.UseAutoreplaceTriggers = True
```

The keys used as AutoReplace Triggers are defined by using the **TriggersActivators** attribute of the language in the configuration file, as shown below.

```
<ConfigLanguage name ="C#" Known ="Csharp" StartComment ="//"
TriggersActivators =" ;.=()">
```

Triggers can be flagged as valid only within the specific lexical states. For example, you can set a trigger not to fire, if it is in a comment within a language, by using the **AllowTriggers** attribute, as shown below.

```
<lexem BeginBlock="/* EndBlock= */" Type="Comment" OnlyLocalSublexems="true"
IsComplex="true" IsCollapsible="true" CollapseName="/*...*/"
AllowTriggers="false">
```

The words to be replaced when the AutoReplace Triggers key is pressed can be defined by using the code given below.

[C#]

```
this.editControl1.Language.AutoReplaceTriggers.AddRange(new
AutoReplaceTrigger[] {new AutoReplaceTrigger("tis", "this"), new
AutoReplaceTrigger("fro", "for")});
```

[VB .NET]

```
Me.editControl1.Language.AutoReplaceTriggers.AddRange(New
AutoReplaceTrigger() {New AutoReplaceTrigger("tis", "this"), New
AutoReplaceTrigger("fro", "for")})
```

The words to be replaced can also be defined within the language definition in the configuration file, as shown below.

```
<AutoReplaceTriggers>
  <AutoReplaceTrigger From ="tis" To ="this" />
  <AutoReplaceTrigger From ="itn" To ="int" />
</AutoReplaceTriggers>
```

See Also

[AutoComplete Support](#)

4.4 Text Visualization

The various text visualization features of Edit control is elaborated under the following topics:

4.4.1 Text Navigation

Edit Control offers extensive support for text navigation. You can perform navigation at character, word, line, page or entire document levels. Here is a brief summary of the APIs available at each level.

Character Level Navigation

The following APIs enable text navigation in the Edit Control, in terms of characters or columns.

Edit Control Method	Description
MoveUp	Moves cursor up, if possible.
MoveDown	Moves cursor down, if possible.
MoveLeft	Moves cursor left, if possible.
MoveRight	Moves cursor right, if possible.

[C#]

```
this.editControl1.MoveUp();
this.editControl1.MoveDown();
this.editControl1.MoveLeft();
this.editControl1.MoveRight();
```

[VB .NET]

```
Me.editControl1.MoveUp()
Me.editControl1.MoveDown()
Me.editControl1.MoveLeft()
Me.editControl1.MoveRight()
```

Word Level Navigation

The following APIs enable text navigation in the Edit Control, in terms of words.

Edit Control Method	Description
MoveLeftWord	Moves caret to the left by one word.
MoveRightWord	Moves caret to the right by one word.

[C#]

```
this.editControl1.MoveLeftWord();
this.editControl1.MoveRightWord();
```

[VB .NET]

```
Me.editControl1.MoveLeftWord();
Me.editControl1.MoveRightWord();
```

Line Level Navigation

The following APIs enable text navigation in the Edit Control, in terms of lines.

Edit Control Method	Description
MoveToLineStart	Moves caret to the beginning of the line. First whitespaces will be skipped.
MoveToLineEnd	Moves caret to the end of the line.

[C#]

```
this.editControl1.MoveToLineStart();
this.editControl1.MoveToLineEnd();
```

[VB .NET]

```
Me.editControl1.MoveToLineStart();
Me.editControl1.MoveToLineEnd();
```

Page Level Navigation

The following APIs enable text navigation in the Edit Control, in terms of pages.

Edit Control Method	Description
MovePageUp	Moves caret one page up.
MovePageDown	Moves caret one page down.

[C#]

```
this.editControl1.MovePageUp();  
this.editControl1.MovePageDown();
```

[VB .NET]

```
Me.editControl1.MovePageUp();  
Me.editControl1.MovePageDown();
```

Document Level Navigation

The following APIs enable text navigation in the Edit Control, in terms of documents.

Edit Control Method	Description
MoveToBeginning	Moves caret to the beginning of the file.
MoveToEnd	Moves caret to the end of the file.

[C#]

```
this.editControl1.MoveToBeginning();  
this.editControl1.MoveToEnd();
```

[VB .NET]

```
Me.editControl1.MoveToBeginning();  
Me.editControl1.MoveToEnd();
```

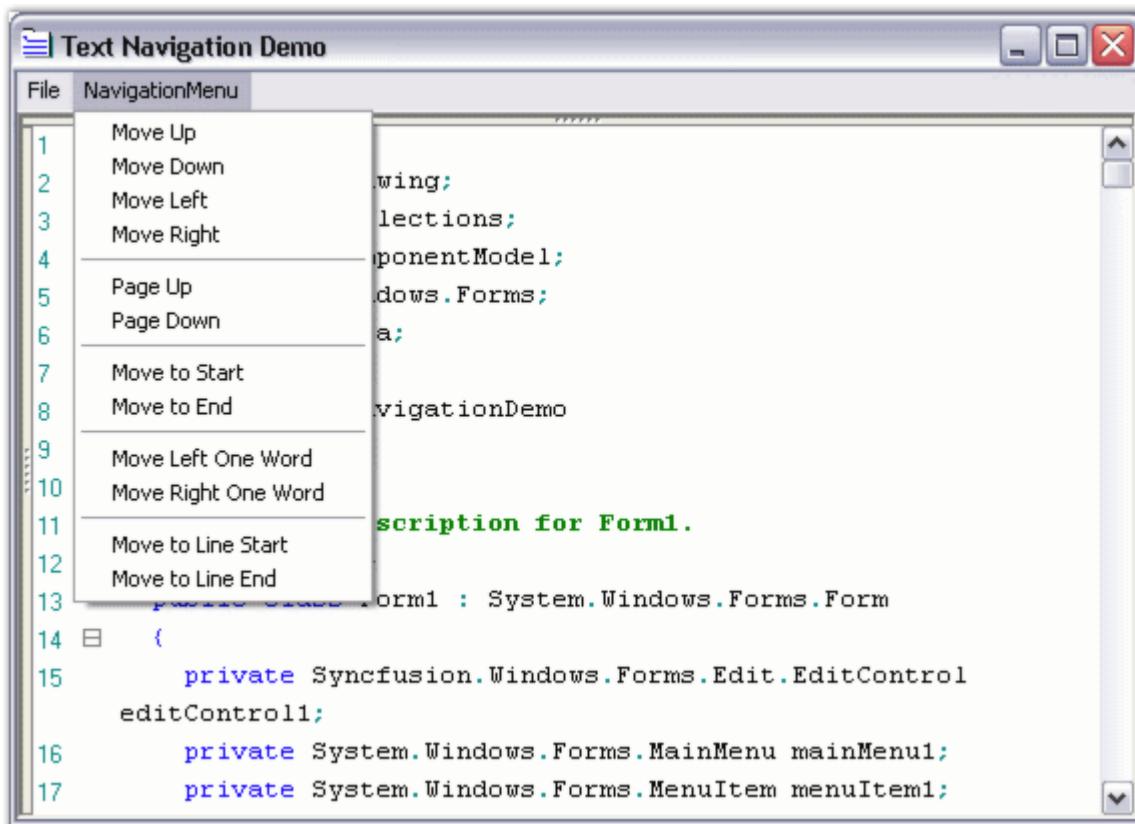


Figure 14: Text Navigation Options in Edit Control

A sample which demonstrates Text Navigation is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Navigation\\TextNavigationDemo**

4.4.1.1 Positions and Offsets

Edit Control has a wide array of APIs for handling text operations by using Positions and Offsets. The **PhysicalLineCount** property is an useful API that returns the actual number of lines in the Edit Control. The following APIs can be used to set the position of the cursor using the keyboard.

Edit Control Property	Description
CurrentColumn	Gets / sets the current column.

CurrentLine	Gets / sets the current line.
CurrentLineInstance	Gets instance of the current line.
CurrentLineText	Gets text of the current line.
CursorPosition	Gets / sets current position of the cursor in virtual coordinates.
PhysicalLineCount	Gets the count of the lines in the file.

You can use the **GoTo** method to navigate to any desired position in a file.

Edit Control Method	Description
GoTo	Navigates to the specified position in the opened file.

[C#]

```
// Gets or sets the current column of the cursor.
this.editControl1.CurrentColumn = 10;

// Gets or sets the current line of the cursor.
this.editControl1.CurrentLine = 7;

// Gets or sets current cursor position.
this.editControl1CursorPosition = new Point(10, 2);

this.editControl1.GoTo(7);
```

[VB .NET]

```
' Gets or sets the current column of the cursor.
Me.editControl1.CurrentColumn = 10

' Gets or sets the current line of the cursor.
Me.editControl1.CurrentLine = 7

' Gets or sets current cursor position.
Me.editControl1CursorPosition = New Point (10, 2)

Me.editControl1.GoTo(7)
```

The coordinates associated with the above properties are referred to as **Virtual** (or **Visible**), because their values vary depending on factors that affect the state of the collapsible blocks, font size of the text, and so on.



Note: The Virtual coordinates of the top-left corner in the Edit Control is (1,1), and it is not a zero-based coordinates system.

The following APIs are used for inter-conversion between virtual / actual positions and offsets.

Edit Control Method	Description
PointToVirtualPosition	Converts point in client coordinates to the virtual position in text.
PointToPhysicalPosition	Converts point in client coordinates to the physical position in text.
ConvertVirtualPositionToPhysical	Converts virtual coordinates to physical coordinates.
ConvertVirtualPositionToOffset	Converts virtual position in text to the offset in stream.
ConvertOffsetToVirtualPosition	Converts in-stream offset to virtual coordinates.
ConvertVirtualPointToCoordinatePoint	Converts point in virtual coordinates to coordinate point.

[C#]

```
// Convert coordinates associated with mouse position to virtual coordinates.  
Point virtualPosition =  
this.editControl1.PointToVirtualPosition(Control.MousePosition);  
  
// Converts coordinates associated with mouse position to physical  
// coordinates.  
Point physicalPosition =  
this.editControl1.PointToPhysicalPosition(Control.MousePosition);  
  
// Converts virtual coordinates to physical coordinates.  
Point physicalPosition =  
this.editControl1.ConvertVirtualPositionToPhysical(virtualPosition);  
  
// Converts virtual coordinates to offset value.  
long offset =  
this.editControl1.ConvertVirtualPositionToOffset(virtualPosition);  
  
// Converts the offset value to virtual coordinates.
```

```
Point virtualPosition =
this.editControl1.ConvertOffsetToVirtualPosition(offset);

// Converts point in virtual coordinates to coordinate point.
this.editControl1.ConvertVirtualPointToCoordinatePoint(int Column, int line);
```

[VB.NET]

```
' Converts coordinates associated with mouse position to virtual coordinates.
Dim virtualPosition As Point =
Me.editControl1.PointToVirtualPosition(Control.MousePosition)

' Converts coordinates associated with mouse position to physical
coordinates.
Dim physicalPosition As Point =
Me.editControl1.PointToPhysicalPosition(Control.MousePosition)

' Converts virtual coordinates to physical coordinates.
Dim physicalPosition As Point =
Me.editControl1.ConvertVirtualPositionToPhysical(virtualPosition)

' Converts virtual coordinates to offset value.
Dim offset As Long =
Me.editControl1.ConvertVirtualPositionToOffset(virtualPosition)

' Converts the offset value to virtual coordinates.
Dim virtualPosition As Point =
Me.editControl1.ConvertOffsetToVirtualPosition(offset)

' Converts point in virtual coordinates to coordinate point.
Me.editControl1.ConvertVirtualPointToCoordinatePoint(Integer Column, Integer
line)
```

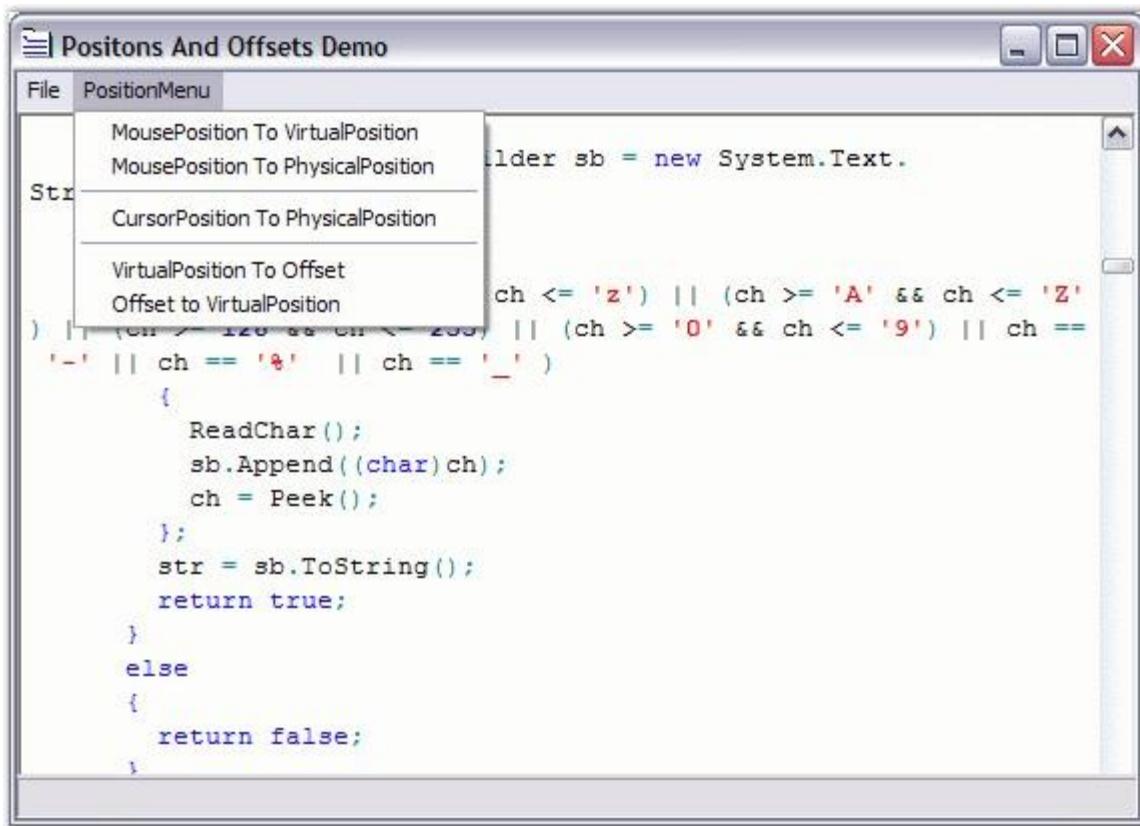


Figure 15: Positions and Offsets Conversion Options in Edit Control



Note: The Offset value is always calculated from the top-left corner of the Edit Control from the Virtual coordinates (1,1).

A sample which demonstrates the above features is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Navigation\\PositionsAndOffsetsDemo**

See Also

[Line Numbers and Current Line Highlighting](#)

4.4.2 Column Guides

Column Guides are used to highlight columns with special meaning. Essential Edit supports unlimited number of column guides.

Each column guide can be provided with a custom color and location. This can be done by setting the **ShowColumnGuides** property of the Edit Control to **True**, and then specifying the color and the location of the Column Guides using **ColumnGuideItem Collection Editor**. The font used to calculate the column location is customized by using **ColumnGuidesMeasuringFont** property.

Edit Control Property	Description
ShowColumnGuides	Gets / sets value that indicates whether column guides should be drawn.
ColumnGuideItems	Gets / sets array of ColumnGuideItem objects.
ColumnGuidesMeasuringFont	Gets / sets font that is used while measuring the position of the column guides.

[C#]

```
// Enable Column Guides.  
this.editControl1.ShowColumnGuides = true;  
  
// Specify the color and the location of the Column Guides.  
ColumnGuideItem[] columnGuideItem = new ColumnGuideItem[2];  
columnGuideItem[0] = new ColumnGuideItem(20, Color.Yellow);  
columnGuideItem[1] = new ColumnGuideItem(40, Color.IndianRed);  
this.editControl1.ColumnGuideItems = columnGuideItem;  
  
// Font used to calculate the column location.  
this.editControl1.ColumnGuidesMeasuringFont = new Font("Microsoft Sans  
Serif", 12);
```

[VB.NET]

```
' Enable Column Guides.  
Me.editControl1.ShowColumnGuides = True  
  
' Specify the color and the location of the Column Guides.  
Dim columnGuideItem() As ColumnGuideItem = New ColumnGuideItem(2)  
columnGuideItem(0) = New ColumnGuideItem(20, Color.Yellow)  
columnGuideItem(1) = New ColumnGuideItem(40, Color.IndianRed)  
Me.editControl1.ColumnGuideItems = columnGuideItem  
  
' Font used to calculate the column location.
```

```
Me.editControl1.ColumnGuidesMeasuringFont = New Font("Microsoft Sans  
Serif",12)
```

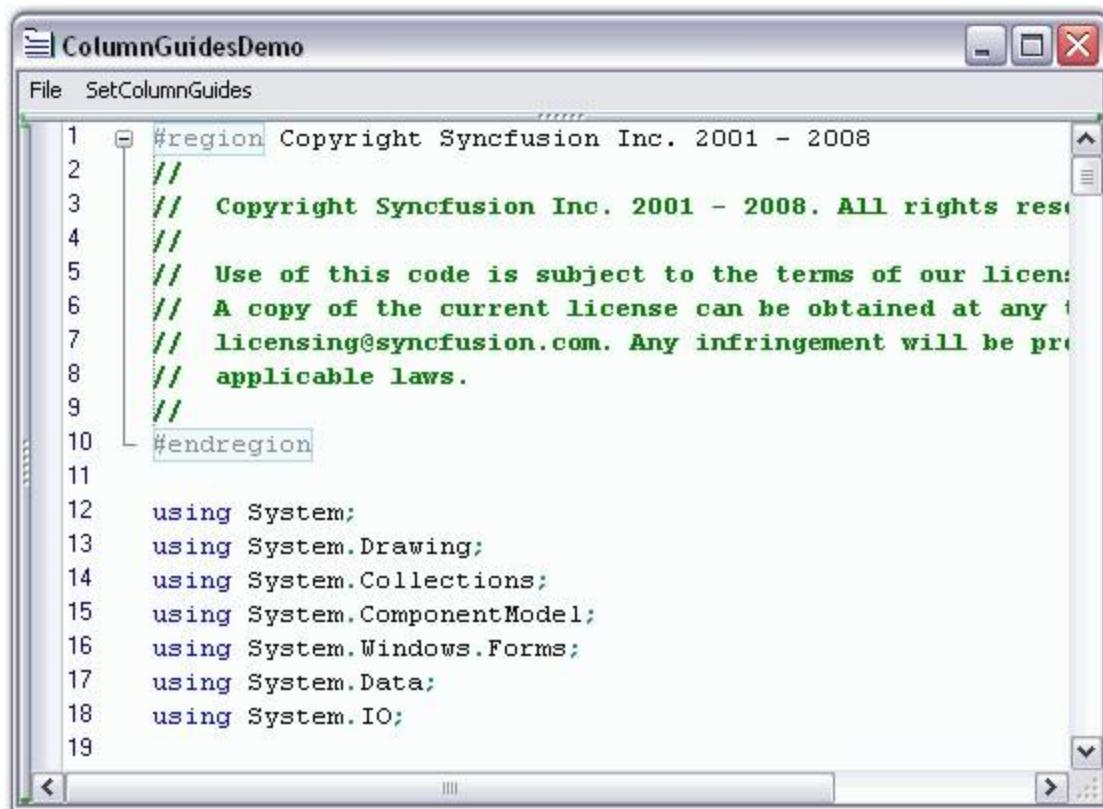


Figure 16: Customized Column Guide Items positioned at Equal Intervals

A sample which illustrates the above feature is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\ColumnGuidesDemo**

4.4.3 Content Dividers

Edit Control supports content dividers just like VB.NET code in Visual Studio.NET code editor. This helps in logical divisioning and better organization of the contents of the Edit Control, thereby improving the readability of the code.

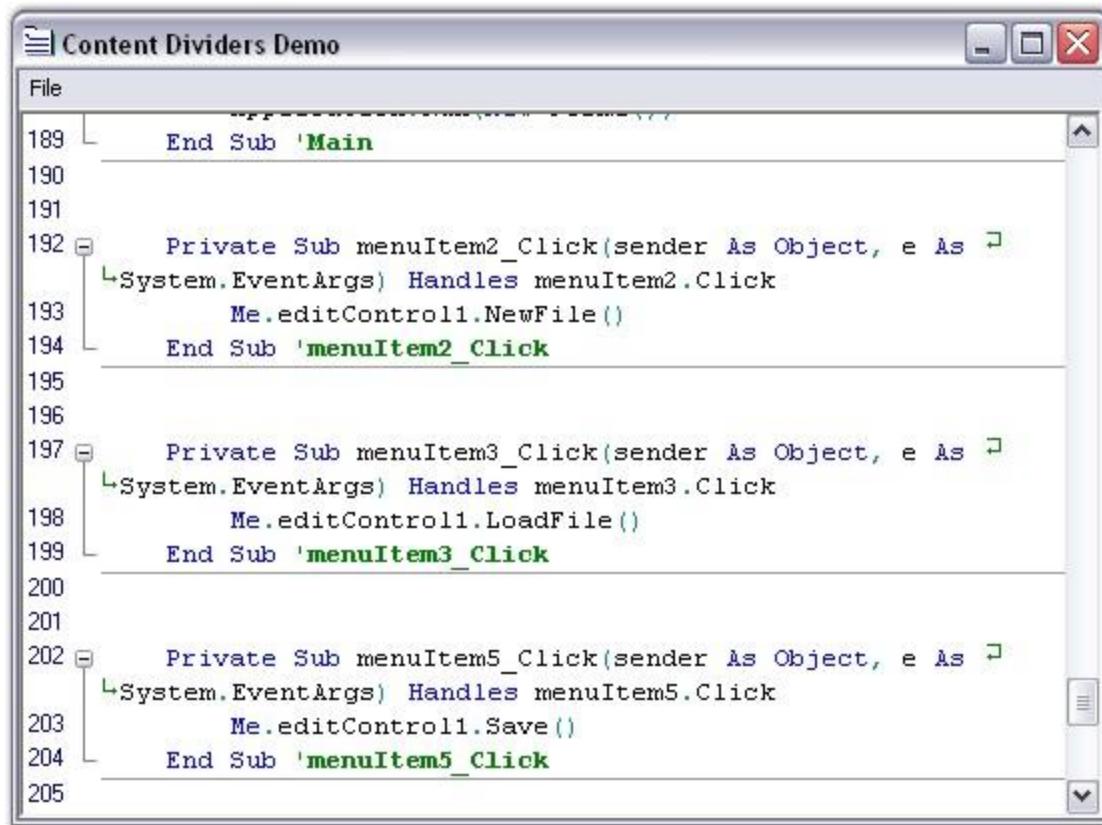


Figure 17: Content Dividers separating the Event Contents into Sections

This feature can be enabled for sections of the Edit Control contents, by setting the **ContentDivider** field to **True**, within its lexem definition in the configuration file.

[XML]

```

// Enable content dividers within its lexem definition in the configuration
file.

<lexem BeginBlock="Function" EndBlock="End Function" Type="KeyWord"
IsComplex="true" IsCollapsible="true" Indent="true"
CollapseName="{Function...End Function}"
AutoNameExpression='.*Function.*\s+(?<text>\w+)\s*\('
AutoNameTemplate="Function [{text}]"
    IsCollapseAutoNamed="true" ContentDivider="true" >
    <References>
        <reference RefID="777"/>
    </References>
    <SubLexems>
        <lexem BeginBlock="\n" IsBeginRegex="true" />
    </SubLexems>
</lexem>

```

A sample which demonstrates Content Dividers is available in the following sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Formatting\\ContentDividersDemo

4.4.4 Underlines, Wavelines and StrikeThrough

Underlines and Wavelines are mainly used to highlight certain sections of text, possibly to notify the user about errors or important sections of the document. Edit Control allows you to underline any desired text in its contents. The underlines can be of different styles, colors and weights, with each of them being used to convey a different meaning. Edit Control supports underlines of the following styles: **Solid**, **Dot**, **Dash**, **Wave** and **DashDot** styles. You can also specify the weight of the underlines to be **Single** or **Double**.

Before the underlining can be applied to the selected text, a custom underlining format has to be defined. The **RegisterUnderlineFormat** method of **ISnippetFormat**, registers the custom underline format to be used while underlining a region. You can create a custom underlining format, as shown in the code below.

[C#]

```
// Registers the custom underline format.
ISnippetFormat format = editControl1.RegisterUnderlineFormat (SelectedColor,
SelectedStyle, SelectedWeight);
```

[VB .NET]

```
' Registers the custom underline format.
Dim format As ISnippetFormat =
editControl1.RegisterUnderlineFormat(SelectedColor, SelectedStyle,
SelectedWeight)
```

The **SelectedColor** value can be set to any desired color. The **SelectedStyle** value is specified by using the **UnderlineStyle** enumerator. The **SelectedWeight** value is specified by using the **UnderlineWeight** enumerator.

Edit Control Underline Enumerator	Description
UnderlineStyle	UnderlineStyle.Solid(default),

	UnderlineStyle.Dot, UnderlineStyle.Dash, UnderlineStyle.Wave, and UnderlineStyle.DashDot.
UnderlineWeight	UnderlineWeight.Thick(default) and UnderlineWeight.Double.

Underlining Selected Text

Underlining can be set and removed for selected text by using the below given methods.

Edit Control Method	Description
SetUnderline	Sets underlining of the specified text region.
RemoveUnderLine	Removes underlining in the specified region.

[C#]

```
this.editControl1.SetUnderline(this.editControl1.Selection.Top,
this.editControl1.Selection.Bottom, format);
this.editControl1.RemoveUnderline(this.editControl1.Selection.Top,
this.editControl1.Selection.Bottom);
```

[VB .NET]

```
Me.editControl1.SetUnderline(Me.editControl1.Selection.Top,
Me.editControl1.Selection.Bottom, format)
Me.editControl1.RemoveUnderline(Me.editControl1.Selection.Top,
Me.editControl1.Selection.Bottom)
```

Underlining using Configuration File

You can also set the underlining from the configuration file, as shown in the below example.

[XML]

```
<format name="Comment" Font="Courier New, 10pt, style=Bold" FontColor="Green"
LineColor="Red" Weight="Thick" Underline="DashDot" />
```

LineColor, **Weight** and **Underline** parameters are used to specify the type of underlining to be used.

```
private System.Windows.Forms.RadioButton radioButton1;
private System.Windows.Forms.RadioButton radioButton2;
private System.Windows.Forms.RadioButton radioButton3;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.GroupBox groupBox2;
```

Figure 18: Text with Double Solid Style, Double Dot Style, Wave Style Underlines

A sample which demonstrates this feature is available in the below location.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\UnderlinesDemo

Striking Through Text

The **StrikeThrough** method allows you to perform strikethrough operation on the text contained in the Edit Control. This is a very useful feature in denoting text that was deleted from the original document or highlighting offending code. You can also specify any custom color for the strikethrough line.

[C#]

```
// Strikeout the current line.
this.editControl1.StrikeThrough(this.editControl1.CurrentLine,
Color.IndianRed);

// Strikeout the selected text.
this.editControl1.StrikeThrough(this.editControl1.Selection.Top,
this.editControl1.Selection.Bottom, Color.Navy);

// Strikeout the text in the specified text range.
this.editControl1.StrikeThrough(startCoordinatePoint, endCoordinatePoint,
Color.Aqua);
```

[VB.NET]

```
' Strikeout the current line.
```

```
Me.editControl1.StrikeThrough(Me.editControl1.CurrentLine, Color.IndianRed)

' Strikeout the selected text.
Me.editControl1.StrikeThrough(Me.editControl1.Selection.Top,
Me.editControl1.Selection.Bottom, Color.Navy)

' Strikeout the text in the specified text range.
Me.editControl1.StrikeThrough(startCoordinatePoint, endCoordinatePoint,
Color.Aqua)
```

To remove the strikethrough line, just call one of the above mentioned methods and specify the **Color** parameter as **Color.Empty**.

```
using Syncfusion.Windows.Forms;
using Syncfusion.Windows.Forms.Edit;
using Syncfusion.Windows.Forms.Edit.Implementation;
using Syncfusion.Windows.Forms.Edit.Implementation.IO;
using Syncfusion.Windows.Forms.Edit.Implementation.Parser;
```

Figure 19: Striking Through Range of Text

A sample which demonstrates the StrikeThrough feature is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\StrikeThroughDemo**

See Also

[Text Border](#), [Text Selection](#)

4.4.5 Text Handling

Edit control offers support for text manipulation operations like append, delete and insertion of multiple lines of text, which is elaborated in the below topic:

4.4.5.1 Appending, Deleting and Inserting Multiple Lines of Text

Edit Control offers support for text manipulation operations like append, delete and insertion of multiple lines of text, through the use of the following APIs.

Appending Text

Text can be appended to the Edit Control by using the below given method.

Edit Control Method	Description
AppendText	Appends the specified text to the end of the existing contents of the Edit Control.

[C#]

```
// Appends the given string to the end of the text in Edit Control.
this.editControl1.AppendText(" text to be appended ");
```

[VB .NET]

```
' Appends the given string to the end of the text in the Edit Control.
Me.editControl1.AppendText(" text to be appended ")
```

Inserting Text

The Insert mode can be enabled in the Edit Control by setting the **EditMode** property to True.

Text can be inserted anywhere inside the Edit Control by using the **InsertText** method given below.

Edit Control Method	Description
InsertText	Inserts a piece of text at any desired position in the Edit Control.

Inserting Multiple Lines

Collection of text lines can be inserted by using the property given below.

Edit Control Property	Description

Lines	Lets you specify multiple lines of text to the Edit Control in the form of a string array. This feature is similar to the one in .NET RichTextBox control.
-------	--

Inserting Text based on Conditions

The below given properties can be used to insert text based on conditions which have been described below.

Edit Control Property	Description
AllowInsertBeforeReadOnly.NewLine	Specifies whether inserting text should be allowed at the beginning of readonly region at the start of new line.
InsertDroppedFileIntoText	Specifies whether the outer file dragged and dropped onto the Edit Control should be inserted into the current content. When this property is set to 'False', the current file is closed, and the dropped outer file is opened.
RespectTabStopsOnInsertingText	Specifies whether tab stops should be respected on inserting blocks of text.

[C#]

```
// Set the Insert mode.
this.editControl1.InsertMode = true;

// Inserts a string at the given line and column.
this.editControl1.InsertText(1, 1, " text to be inserted ");

// Specifies multiple lines of text to the EditControl in the form of a
// string array.
this.editControl1.Lines = new string[] { " first line ", " second line ", "
third line "};

// Allows text insertion only at the beginning of the readonly region at the
// start of a new line.
this.editControl1.AllowInsertBeforeReadOnlyNewLine = true;

// Specifies whether the outer file dragged and dropped onto the editcontrol
// should be inserted into the current content.
this.editControl1.InsertDroppedFileIntoText = true;
```

[VB.NET]

```
' Set the Insert mode.  
Me.editControl1.InsertMode = True  
  
' Inserts a string at the given line and column.  
Me.editControl1.InsertText(1, 1, "text to be inserted")  
  
' Specifies multiple lines of text to the EditControl in the form of a string array.  
Me.editControl1.Lines = New String() {"first line", "second line", "third line"}  
  
' Allows text insertion only at the beginning of the readonly region at the start of a new line.  
Me.editControl1.AllowInsertBeforeReadOnlyNewLine = True  
  
' Specifies whether the outer file dragged and dropped onto the editcontrol should be inserted into the current content.  
Me.editControl1.InsertDroppedFileIntoText = True
```

Deleting Text

Text can be deleted in the Edit Control by using the below given methods.

Edit Control Method	Description
DeleteChar	Deletes a character to the right of the current cursor position.
DeleteCharLeft	Deletes a character to the left of the current cursor position.
DeleteWord	Deletes a word to the right of the current cursor position.
DeleteWordLeft	Deletes a word to the left of the current cursor position.
DeleteAll	Deletes all text in the document.
DeleteText	Deletes the specified text.

[C#]

```
// Deletes the character to the right of the cursor.  
this.editControl1.DeleteChar();  
  
// Deletes the character to the left of the cursor.
```

```
this.editControl1.DeleteCharLeft();

// Deletes a word to the right of the current cursor position.
this.editControl1.DeleteWord();

// Deletes a word to the left of the current cursor position.
this.editControl1.DeleteWordLeft();

// To delete all the text.
this.editControl1.DeleteAll();

// To delete a selection.
this.editControl1.DeleteText(this.editControl1.Selection.Top,
this.editControl1.Selection.Bottom);
```

[VB.NET]

```
' Deletes the character to the right of the cursor.
Me.editControl1.DeleteChar()

' Deletes the character to the left of the cursor.
Me.editControl1.DeleteCharLeft()

' Deletes a word to the right of the current cursor position.
Me.editControl1.DeleteWord()

' Deletes a word to the left of the current cursor position.
Me.editControl1.DeleteWordLeft()

' Deletes all the text.
Me.editControl1.DeleteAll()

' Deletes a selection.
Me.editControl1.DeleteText(Me.editControl1.Selection.Top,
Me.editControl1.Selection.Bottom)
```

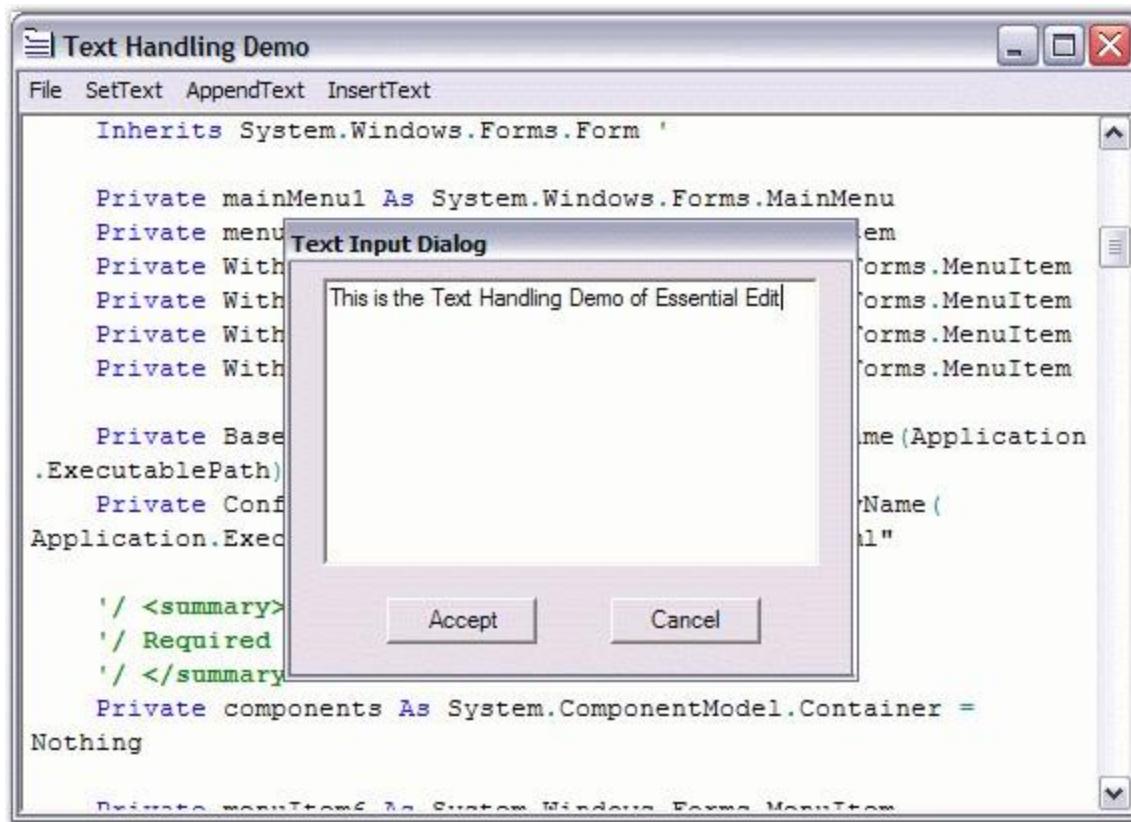


Figure 20: Input entered for Handling Text

A sample which demonstrates the above features is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\TextHandlingDemo

4.4.6 Spaces and Tabs

Edit Control supports text operations with tabs and spaces by using the APIs discussed in this section.

Essential Edit controls the insertion of tabs using the **UseTabs** property, which lets you specify whether a tab (or an equivalent number of spaces) needs to be inserted, when the TAB key is pressed in the Edit Control. Similarly, tab stops can also be inserted.

Edit Control Property	Description
UseTabs	Specifies whether tab symbol is allowed or spaces should be used instead. Setting this property to True, allows you to insert tabs, whereas setting it to False, allows you to insert spaces.
UseTabStops	Gets / sets value that indicates whether tab stops should be used.
TabStopsArray	Gets / sets an array of tab stops.

[C#]

```
this.editControl1.UseTabs = true;
this.editControl1.UseTabStops = true;
this.editControl1.TabStopsArray = new int[] { 8, 16, 24, 32, 40};
```

[VB .NET]

```
Me.editControl1.UseTabs = True
Me.editControl1.UseTabStops = True;
Me.EditControl1.TabStopsArray = New Integer() {8, 16, 24, 32, 40}
```

Specifying Tab Size

The size of the tab can be specified by using the below given property.

Edit Control Property	Description
TabSize	Specifies tab size in spaces.

[C#]

```
// Size of the tab in terms of space.
this.editControl1.TabSize = 8;
```

[VB .NET]

```
' Size of the tab in terms of space.
Me.editControl1.TabSize = 8
```

TAB key Functionality

The **TransferFocusOnTab** property allows you to specify, if the Edit Control should process the TAB key as a text input, or transfer the focus to the next control (by the order of TabIndex property value) on the Form or the User Control hosting the Edit Control.

[C#]

```
// Insert tabs into the EditControl as text input.  
this.editControl1.TransferFocusOnTab = false;  
  
// Transfer focus to the next control.  
this.editControl1.TransferFocusOnTab = true;
```

[VB.NET]

```
' Insert tabs into the EditControl as text input.  
this.editControl1.TransferFocusOnTab = False  
  
' Transfer focus to the next control.  
this.editControl1.TransferFocusOnTab = True
```

TAB key Functionality on Selected Text

The below given methods can be used convert the spaces in a selected region into tabs and vice versa. Tab symbols can also be added, inserted or removed from selected text.

Edit Control Method	Description
TabifySelection	Lets you convert the spaces in the selected region into equivalent number of tabs.
UntabifySelection	Lets you convert the tabs in the selected region into equivalent number of spaces.
AddTabsToSelection	Adds leading tab symbol to the selected lines, or just inserts the tab symbol.
RemoveTabsFromSelection	Removes leading tab symbol (or its spaces equivalent) from selected lines.

[C#]

```
// Covert spaces to tabs.  
this.editControl1.TabifySelection();  
  
// Converts tabs to spaces.  
this.editControl1.UntabifySelection();  
  
// Add or insert leading tab symbol to selected lines.  
this.editControl1.AddTabsToSelection();  
  
// Remove leading tab symbol from selected lines.  
this.editControl1.RemoveTabsFromSelection();
```

[VB .NET]

```
' Covert spaces to tabs.  
Me.editControl1.TabifySelection()  
  
' Converts tabs to spaces.  
Me.editControl1.UntabifySelection()  
  
' Add or insert leading tab symbol to selected lines.  
Me.editControl1.AddTabsToSelection()  
  
' Remove leading tab symbol from selected lines.  
Me.editControl1.RemoveTabsFromSelection()
```

4.4.6.1WhiteSpace Indicators

Edit Control has the ability to indicate whitespaces in its contents with default indicators, explained as follows.

1. Single Spaces are indicated by using Dots.
2. Tabs are indicated by using Right Arrows.
3. Line Feeds are indicated by using a special Line Feed Symbol.

```

43  ¶
44  ¶
45  ¶
46 □ BEGIN(*•main•*) ¶
47 → → → Animation; ¶
48 .....readln; ¶
49 □ END . . . (*•main•*) ¶
50  ¶
51  ¶
52  ¶
53

```

Figure 21: Indicators for Single Spaces, Tabs and a Line Feed

You can enable whitespace indicators by setting the **ShowWhiteSpaces** property to **True**. By default, this property is set to **False**.

Edit Control Property	Description
ShowWhiteSpaces	Gets / sets value indicating whether whitespaces should be shown as special symbols.

You can also toggle the visibility of the whitespace indicators by using the **ToggleShowingWhiteSpaces** method, or by setting the **ShowWhiteSpaces** property to False.

Edit Control Method	Description
ToggleShowingWhiteSpaces	Toggles showing of whitespaces.

[C#]

```

// Enabling white space indicators.
this.editControl1.ShowWhitespaces = true;

// Toggle the visibility of the white space indicators.
this.editControl1.ToggleShowingWhiteSpaces();

```

[VB.NET]

```

' Enabling white space indicators.
Me.editControl1.ShowWhitespaces = True

' Toggle the visibility of the white space indicators.

```

```
Me.editControl1.ToggleShowingWhiteSpaces()
```

Showing / Hiding Indicators

You can selectively show / hide the whitespace indicators by using the following subproperties of the **WhiteSpaceIndicators** property - **ShowSpaces**, **ShowTabs** and **ShowNewLines**.

Edit Control Property	Description
ShowSpaces	Indicates whether spaces should be replaced with symbols.
ShowTabs	Indicates whether tabs should be replaced with symbols.
ShowNewLines	Indicates whether new lines should be replaced with symbols.

[C#]

```
// Custom indicator for Line Feed.  
this.editControl1.WhiteSpaceIndicators.ShowSpaces = true;  
  
// Custom indicator for Tab.  
this.editControl1.WhiteSpaceIndicators.ShowTabs = true;  
  
// Custom indicator for Space Character.  
this.editControl1.WhiteSpaceIndicators.SpaceNewLines = true;
```

[VB .NET]

```
' Custom indicator for Line Feed.  
Me.editControl1.WhiteSpaceIndicators.ShowSpaces = True  
  
' Custom indicator for Tab.  
Me.editControl1.WhiteSpaceIndicators.ShowTabs = True  
  
' Custom indicator for Space Character.  
Me.editControl1.WhiteSpaceIndicators.SpaceNewLines = True
```

You can also set the indicators to indicate single spaces, tabs and line feeds by using the **NewLineString**, **TabString** and **SpaceChar** subproperties of the **WhiteSpaceIndicators** property, as shown below.

Edit Control Property	Description
NewLineString	Gets / sets string that represents line feed in WhiteSpace mode.

TabString	Gets / sets string that represents Tab in WhiteSpace mode.
SpaceChar	Gets / sets character that represents line feed in WhiteSpace mode.

[C#]

```
// Custom indicator for Line Feed.  
this.editControl1.WhiteSpaceIndicators.NewLineString = "LF";  
  
// Custom indicator for Tab.  
this.editControl1.WhiteSpaceIndicators.TabStop = "TAB";  
  
// Custom indicator for Space Character.  
this.editControl1.WhiteSpaceIndicators.SpaceChar = "s";
```

[VB .NET]

```
' Custom indicator for Line Feed.  
Me.editControl1.WhiteSpaceIndicators.NewLineString = "LF"  
  
' Custom indicator for Tab.  
Me.editControl1.WhiteSpaceIndicators.TabStop = "TAB"  
  
' Custom indicator for Space Character.  
Me.editControl1.WhiteSpaceIndicators.SpaceChar = "s"
```

See Also

[Spaces and Tabs](#)

4.4.7 Line Numbers and Current Line Highlighting

Line Numbers can be automatically assigned to the contents of the Edit Control by enabling its **ShowLineNumbers** property.

The number of lines in the Edit Control can be obtained by using the **PhysicalLineCount** property. This property returns the actual number of lines in the Edit Control, without considering the lines that maybe hidden because of a collapsed outlining block or new lines that maybe added because of wordwrap.

Edit Control Property	Description
ShowLineNumbers	Gets / sets value indicating whether line numbers should be shown.
PhysicalLineCount	Gets the count of lines in the files.

[C#]

```
// Assigning Line Numbers to the contents of the Edit Control.
this.editControl1.ShowLineNumbers = true;

// Gets the number of lines in the Edit Control.
int actualLineCount = this.editControl1.PhysicalLineCount;
```

[VB.NET]

```
' Assigning Line Numbers to the contents of the Edit Control.
Me.editControl1.ShowLineNumbers = True

' Gets the number of lines in the Edit Control.
Dim actualLineCount As Integer = Me.editControl1.PhysicalLineCount
```

Line numbers can be customized by using the below given Edit Control properties.

Edit Control Property	Description
LineNumbersAlignment	Specifies the alignment of line numbers. The options provided are <i>Left</i> <i>Right</i>
LineNumbersColor	Specifies the color of line numbers.
LineNumbersFont	Specifies the font of line numbers.
SelectOnLineNumberClick	Gets / sets value indicating whether click on line numbers performs selection.

[C#]

```
// Specify the alignment of line numbers.  
this.editControl1.LineNumbersAlignment =  
Syncfusion.Windows.Forms.Edit.Enums.LineNumberAlignment.Right;  
  
// Assign any color to the line numbers.  
this.editControl1.LineNumbersColor = Color.IndianRed;  
  
// Assign any font to the line numbers.  
this.editControl1.LineNumbersFont = new Font("Verdana", 9);  
  
// Enabling SelectOnLineNumberClick property to perform selection on  
clicking the line numbers.  
this.editControl1.SelectOnLineNumberClick = true;
```

[VB.NET]

```
' Specify the alignment of line numbers.  
Me.editControl1.LineNumbersAlignment =  
Syncfusion.Windows.Forms.Edit.Enums.LineNumberAlignment.Right  
  
' Assign any color to the line numbers.  
Me.editControl1.LineNumbersColor = Color.IndianRed  
  
' Assign any font to the line numbers.  
Me.editControl1.LineNumbersFont = new Font("Verdana", 9)  
  
' Enabling SelectOnLineNumberClick property to perform selection on  
clicking the line numbers.  
Me.editControl1.SelectOnLineNumberClick = True
```

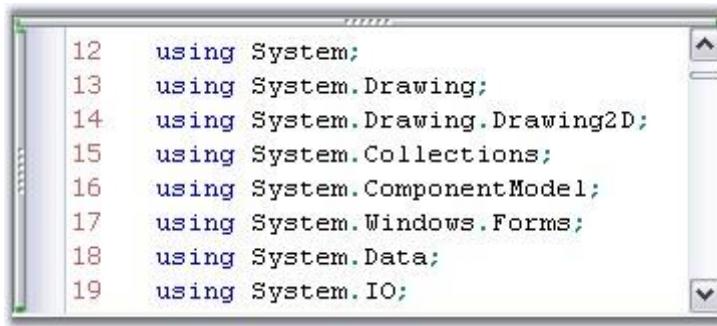


Figure 22: IndianRed Color Line Numbers with *FontSize* = "9", *FontStyle* = "Verdana"

Highlighting Current Line at Run Time

You can highlight the current line where the mouse pointer is present by setting the **HighlightCurrentLine** property of the Edit Control to **True**. Set the color for the highlighted line by using the **CurrentLineHighlightColor** property.

Edit Control Property	Description
HighlightCurrentLine	Gets / sets value indicating whether current line should be highlighted.
CurrentLineHighlightColor	Gets / sets color of current line highlight.

[C#]

```
this.editControl1.HighlightCurrentLine = true;
this.editControl1.CurrentLineHighlightColor = Color.Orange;
```

[VB .NET]

```
Me.editControl1.HighlightCurrentLine = true
Me.editControl1.CurrentLineHighlightColor = Color.Orange
```

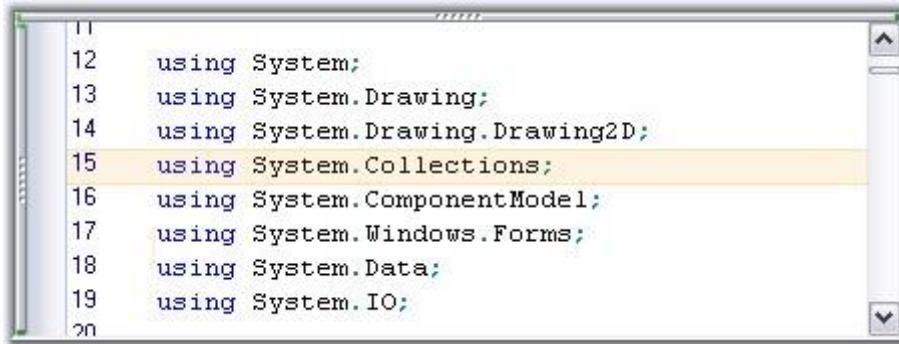


Figure 23: CurrentLineHighlightColor = "Orange"

You can also highlight the selected text by using the Text Highlighting feature discussed in Background Settings.

4.4.8 Bookmarks and Custom Indicators

Essential Edit enables users to locate a section or a line of a document by using the Bookmarks and Custom Indicators feature like in Visual Studio. This provides quick access to any part of the contents of the Edit Control.

The Edit Control allows any number of custom images or bookmarks to be added to a document.



Note: At any given point of time, each line can have only one indicator or bookmark associated with it.

Displaying Bookmarks

The Edit Control provides an indicator margin for the purpose of displaying the custom indicators or bookmarks. This can be enabled by using the **ShowIndicatorMargin** property, as shown below.

Edit Control Property	Description
ShowIndicatorMargin	Gets / sets value indicating whether bookmarks and indicator margins should be visible.
MarkerAreaWidth	Gets / sets width of marker area.

[C#]

```
// Displays the Indicator margin.  
this.editControl1.ShowIndicatorMargin = true;  
  
// Sets the width of the Indicator margin.  
this.editControl1.MarkerAreaWidth = 20;
```

[VB .NET]

```
' Displays the Indicator margin.  
Me.editControl1.ShowIndicatorMargin = True  
  
' Sets the width of the Indicator margin.  
Me.editControl1.MarkerAreaWidth = 20
```

Customizing Bookmarks

You can either display the default bookmark image (like in Visual Studio.NET) or display custom images as indicators. This can be done by making use of the following methods of the Edit Control.

Edit Control Method	Description
BookmarkToggle	Sets bookmark to the current line.

BookmarkAdd	Sets bookmark at the specified line.
BookmarkRemove	Removes bookmark at the specified line.
BookmarkGet	Gets bookmark at the specified line.
BookmarkNext	Goes to the next bookmark.
BookmarkPrevious	Goes to the previous bookmark.
BookmarkClear	Clears all the bookmarks.

[C#]

```
// Sets bookmark at the specified line.  
this.editControl1.BookmarkAdd(this.editControl1.CurrentLine);  
  
// Removes bookmark at the specified line.  
this.editControl1.BookmarkRemove(this.editControl1.CurrentLine);  
  
this.editControl1.BookmarkRemove(this.editControl1.CurrentLine);  
  
// Draw the bookmark with custom look and feel specified in the  
BrushInfo object.  
BrushInfo brushInfo = new BrushInfo(GradientStyle.ForwardDiagonal,  
Color.IndianRed, Color.Ivory);  
this.editControl1.BookmarkAdd(this.editControl1.CurrentLine,  
brushInfo);  
  
// Get the Bookmark object of the current line.  
IBookmark bookmark =  
this.editControl1.BookmarkGet(this.editControl1.CurrentLine);
```

[VB.NET]

```
' Sets bookmark at the specified line.  
Me.editControl1.BookmarkAdd(Me.editControl1.CurrentLine)  
  
' Removes bookmark at the specified line.  
Me.editControl1.BookmarkRemove(Me.editControl1.CurrentLine)  
  
' Draw the bookmark with custom look and feel specified in the  
BrushInfo object.  
Dim brushInfo As BrushInfo = new  
BrushInfo(GradientStyle.ForwardDiagonal, Color.IndianRed, Color.Ivory)  
Me.editControl1.BookmarkAdd(Me.editControl1.CurrentLine, brushInfo)
```

```
' Get the Bookmark object of the current line.
Dim bookmark As IBookmark =
Me.EditControl1.BookmarkGet(Me.EditControl1.CurrentLine)
```

Setting Bookmarks

Bookmarks can be set and removed by using the below given methods.

Edit Control Method	Description
SetCustomBookmark	Sets custom bookmark for the desired line.
RemoveCustomBookmark	Removes the custom bookmark from the desired line.



Note: To clear the bookmarks set by using the SetCustomBookmark method, you must use the **BookmarkClear** method with its **bool** argument set as **True**.

The bookmarks set by using the SetCustomBookmark method, do not respond to the **BookmarkNext** and **BookmarkPrevious** methods automatically. In order to enable this, you have to set the **UseInBookmarkSearch** property of the custom bookmark to **True**.

[C#]

```
// Sets custom bookmarks and enables it to respond to BookmarkNext and
BookmarkPrevious methods.
ICustomBookmark customBookmark =
this.editControl1.SetCustomBookmark(this.editControl1.CurrentLine, new
BookmarkPaintEventHandler(CustomBookmarkPainter));
customBookmark.UseInBookmarkSearch = true;

// Removes the bookmark of the current line.
ICustomBookmark customBookmark =
this.editControl1.RemoveCustomBookmark(this.editControl1.CurrentLine,
BookmarkPaintEventHandler(CustomBookmarkPainter));
```

[VB .NET]

```
' Sets custom bookmarks and enables it to respond to BookmarkNext and
BookmarkPrevious methods.
Dim customBookmark As ICustomBookmark =
Me.editControl1.SetCustomBookmark(Me.editControl1.CurrentLine, New
BookmarkPaintEventHandler(CustomBookmarkPainter))
```

```
customBookmark.UseInBookmarkSearch = True

' Removes the bookmark of the current line.
Dim customBookmark As ICustomBookmark =
Me.editControl1.RemoveCustomBookmark(Me.editControl1.CurrentLine,
BookmarkPaintEventHandler(CustomBookmarkPainter))
```

Setting Tooltips for Bookmarks

Tooltips can be set for bookmarks and customized by using the below given properties.

Edit Control Property	Description
ShowBookmarkTooltip	Specifies whether the tooltip of the bookmark is shown.
BookmarkTooltipBackgroundBrush	Gets / sets brush for bookmark tooltip background.
BookmarkTooltipBorderColor	Specifies the color of the bookmark tooltip form border.

[C#]

```
// Shows the tooltip of the bookmark.
this.editControl1.ShowBookmarkTooltip = true;

// Gets or sets brush for bookmark tooltip background.
this.editControl1.BookmarkTooltipBackgroundBrush = new
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.PatternStyle.Percent05,
System.Drawing.SystemColors.WindowText, System.Drawing.Color.Gold);

// Specify the color of the bookmark tooltip form border.
this.editControl1.BookmarkTooltipBorderColor =
System.Drawing.Color.Crimson;
```

[VB .NET]

```
' Shows the tooltip of the bookmark.
Me.editControl1.ShowBookmarkTooltip = True

' Gets or sets brush for bookmark tooltip background.
Me.editControl1.BookmarkTooltipBackgroundBrush = New
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.PatternStyle.Percent05,
System.Drawing.SystemColors.WindowText, System.Drawing.Color.Gold)

' Specify the color of the bookmark tooltip form border.
Me.editControl1.BookmarkTooltipBorderColor =
```

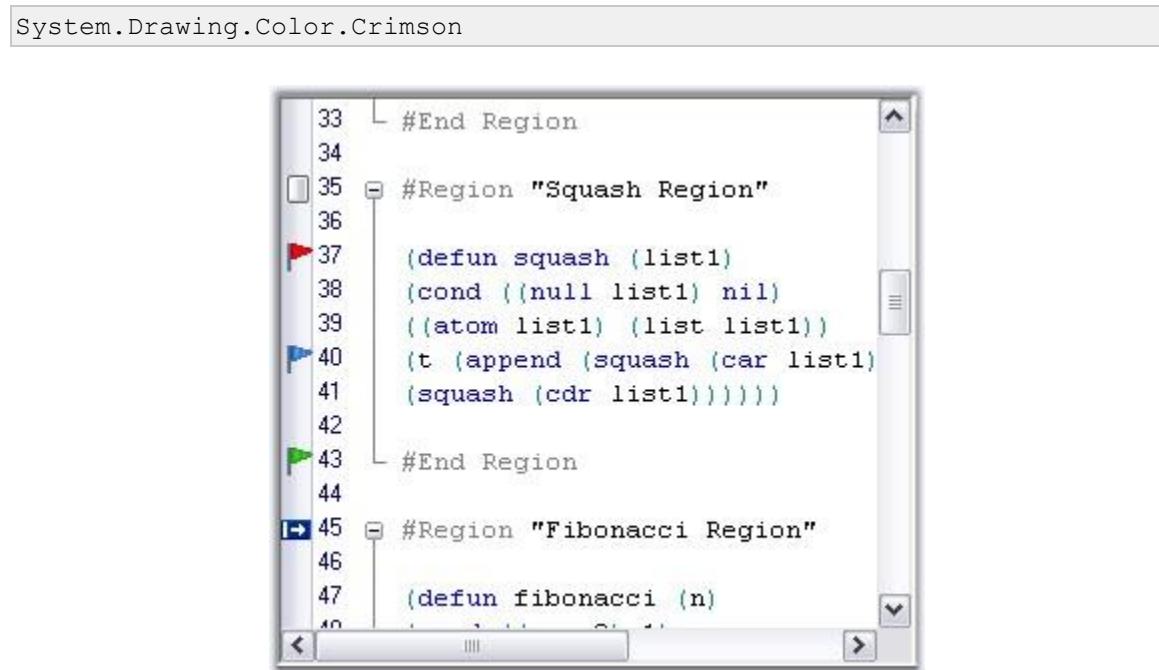


Figure 24: Edit Control with Custom Bookmarks

A sample which illustrates the above features is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Text Navigation\\CustomBookmarksDemo

4.4.9 Comments

This section discusses how comments can be set for the text in Edit Control.

Comments can be set for a single line, selected text, and for text within a specified range by using the below given methods.

Edit Control Method	Description
CommentLine	Comments single line.
CommentSelection	Comments selected text.
CommentText	Comments text in the specified range.

[C#]

```
this.editControl1.CommentLine(1);
this.editControl1.CommentSelection();
this.editControl1.CommentText(new Point(1, 1), new Point(7, 7));
```

[VB.NET]

```
Me.editControl1.CommentLine(1)
Me.editControl1.CommentSelection()
Me.editControl1.CommentText(New Point(1, 1), New Point(7, 7))
```

Removing Comments

Comments can be removed by using the below given methods.

Edit Control Method	Description
UnCommentLine	UnComments single line.
UnCommentSelection	UnComments selected text.
UnCommentText	UnComments text in the specified range.

[C#]

```
this.editControl1.UnCommentLine();
this.editControl1.UncommentSelection();
this.editControl1.UncommentText(new Point(1, 1), new Point(7, 7));
```

[VB.NET]

```
Me.editControl1.UnCommentLine()
Me.editControl1.UncommentSelection()
Me.editControl1.UncommentText(New Point(1, 1), New Point(7, 7)))
```

4.4.10 Break Points

Essential Edit allows you to set a pause at some specified location in the Edit Control by using the **Break Points** feature. This is done by combining the [Line Background](#) and Custom Indicator features. **IndicatorMarginClick** event can be handled to insert a break point.

[C#]

```
private void editControl1_IndicatorMarginClick(object sender,
Syncfusion.Windows.Forms.Edit.IndicatorEventArgs e)
{
    // Set breakpoint indicator.
    this.editControl1.SetCustomBookmark(e.LineIndex, new
BookmarkPaintEventHandler(CustomBookmarkPainter));

    // Highlight the relevant line.
    IBackgroundFormat format =
    this.editControl1.RegisterBackColorFormat(color, Color.Transparent);
    this.editControl1.SetLineBackColor(e.LineIndex, true, format);
}
```

[VB.NET]

```
Private Sub editControl1_IndicatorMarginClick(sender As Object, e As
Syncfusion.Windows.Forms.Edit.IndicatorEventArgs) Handles
editControl1.IndicatorMarginClick
    ' Set breakpoint indicator.
    Me.editControl1.SetCustomBookmark(e.LineIndex, New
BookmarkPaintEventHandler(AddressOf CustomBookmarkPainter))

    ' Highlight the relevant line.
    Dim format As IBackgroundFormat =
    Me.editControl1.RegisterBackColorFormat(color, Color.Transparent)
    Me.editControl1.SetLineBackColor(e.LineIndex, True, format)
End Sub
```

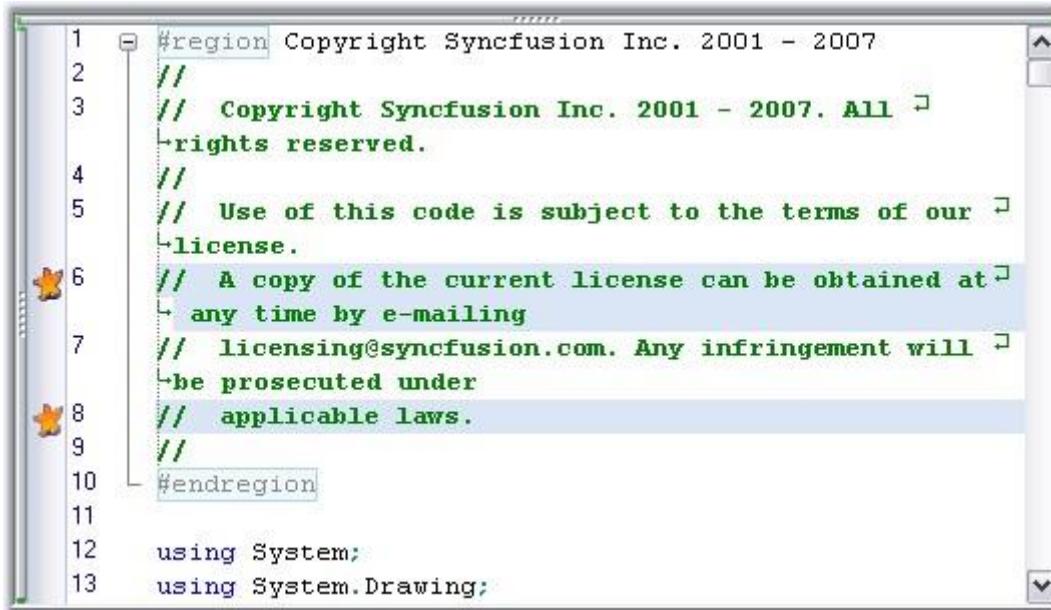


Figure 25: Inserting Break Points in Edit Control

A sample which demonstrates setting custom indicators is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Text Navigation\\BreakPointDemo

4.4.11 Text Formatting

Edit control has some text formatting features which are discussed under the following topics:

4.4.11.1 Bracket Highlighting and Indentation Guidelines

Edit Control has one of the most powerful and intelligent Bracket Highlighting and Indentation Guideline features. Edit Control is also capable of supporting language domains that have multiple languages, such as HTML or XML. Moreover, for each language, different brackets can be defined for highlighting. In C#, curly braces can be highlighted, while in HTML or XML, angled braces (for tags) can be highlighted.

Consider the following example.

[C#]

```
public void Test()
{
    string str;
    str = "{}";
}
```

If the cursor is positioned on the end curly brace, most editors will match to the open curly brace in the string. On the contrary, Edit Control matches to the open curly brace for the method.

The Bracket Highlighting and Indentation Guidelines functionalities are supported using the following APIs in the Edit Control.

- ShowIndentationGuidelines
- HideIndentationGuidelines
- ShowIndentGuideline
- IndentLineColor
- IndentBlockHighlightingColor
- IndentationBlockBackgroundBrush
- IndentationBlockBorderColor
- IndentationBlockBorderStyle
- JumpToIndentBlockStart
- JumpToIndentBlockEnd
- OnlyHighlightMatchingBraces

The preceding APIs are explained below in detail.

The indentation guidelines are vertical lines that connect the matching brackets. This feature enhances the readability of code.

Edit Control Property	Description
ShowIndentationGuidelines	Gets / sets value indicating whether indentation guidelines should be shown.

The indentation guidelines can be turned on by setting the **ShowIndentationGuidelines** property to **True**. It can be turned off either by setting this property to **False**, or by invoking the **HideIndentGuideline** method.

Also, the indent guideline for the current region can be set by using the **ShowIndentGuideline** method.

Edit Control Method	Description
HideIndentGuideline	Hides indentation guideline.
ShowIndentGuideline	If possible, shows indent guideline of the current region.

[C#]

```
// Indentation Guidelines are displayed.  
this.editControl1.ShowIndentGuideline();  
  
// Hide Indentation Guideline.  
this.editControl1.HideIndentGuideline();  
  
// Show Indentation Guideline.  
this.editControl1.ShowIndentGuideline();
```

[VB .NET]

```
' Indentation Guidelines are displayed.  
Me.editControl1.ShowIndentGuideline = True  
  
' Hide Indentation Guideline.  
Me.editControl1.HideIndentGuideline()  
  
' Show Indentation Guideline.  
Me.editControl1.ShowIndentGuideline()
```

Bracket Highlighting

The bracket highlighting feature can be turned on by enabling the **ShowIndentGuidelines** and **OnlyHighlightMatchingBraces** properties. Setting the **OnlyHighlightMatchingBraces** property to **True**, enables bracket highlighting whereas the indentation guidelines are not displayed.

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    this.editControl1.LoadFile("../..\..\Form1.cs");

    this.editControl1.IndentLineColor = Color.Khaki;
}
```

Figure 26: Bracket Highlighting with Indentation Guidelines

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    this.editControl1.LoadFile("../..\..\Form1.cs");

    this.editControl1.IndentLineColor = Color.Khaki;
}
```

Figure 27: Bracket Highlighting without Indentation Guidelines

Customizing the Appearance

It is possible to specify custom colors for the indentation guidelines and bracket highlighting blocks by using the below given properties.

Edit Control Property	Description
IndentLineColor	Specifies color of the indent line.
IndentBlockHighlightingColor	Specifies color of the indent block start and end.
IndentationBlockBackgroundBrush	Gets / sets brush for indentation block background.
IndentationBlockBorderColor	Specifies color of indentation block border line.

IndentationBlockBorderStyle	Specifies style of indentation block border line.
ShowIndentationBlockBorders	Specifies whether indentation block borders should be drawn.

[C#]

```
this.editControl1.IndentLineColor = Color.OrangeRed;
this.editControl1.IndentBlockHighlightingColor = Color.IndianRed;
this.editControl1.IndentationBlockBackgroundBrush = new
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardD
iagonal, System.Drawing.SystemColors.Info, System.Drawing.Color.Khaki);
this.editControl1.IndentationBlockBorderColor =
System.Drawing.Color.Crimson;
this.editControl1.IndentationBlockBorderStyle =
Syncfusion.Windows.Forms.Edit.Enums.FrameBorderStyle.DashDot;
this.editControl1.ShowIndentationBlockBorders = true;
```

[VB.NET]

```
Me.editControl1.IndentLineColor = Color.OrangeRed
Me.editControl1.IndentBlockHighlightingColor = Color.IndianRed
Me.editControl1.IndentationBlockBackgroundBrush = New
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardD
iagonal, System.Drawing.SystemColors.Info, System.Drawing.Color.Khaki)
Me.editControl1.IndentationBlockBorderColor =
System.Drawing.Color.Crimson
Me.editControl1.IndentationBlockBorderStyle =
Syncfusion.Windows.Forms.Edit.Enums.FrameBorderStyle.DashDot
Me.editControl1.ShowIndentationBlockBorders = True
```

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    this.editControl1.LoadFile("../..\..\Form1.cs");

    this.editControl1.IndentLineColor = Color.Khaki;
}
```

Figure 28: *IndentLineColor = "OrangeRed"; IndentBlockHighlightingColor = "IndianRed"*

Positioning

It is also possible to position the caret at the beginning or end of the indentation block by using the **JumpToIndentBlockStart** and **JumpToIndentBlockEnd** methods respectively.

Edit Control Method	Description
JumpToIndentBlockStart	Jumps to the start of the block.
JumpToIndentBlockEnd	Jumps to the end of the block.

Refer to the Indentation Guidelines Demo sample for more information in this regard.

..\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Navigation\\IndentationGuidelinesDemo

4.4.11.2 Auto Indentation

The Edit control offers advanced text indentation support to suit the requirements of the user.

The properties given in the following table can be used to customize the auto indentation settings of the Edit control.

Property	Description
AutoIndentMode	Specifies mode of auto indentation. The options provided are <ul style="list-style-type: none"> • None • Block • Smart
AutoIndentGuideline	Gets / sets the value that specifies whether indent guideline should be shown automatically after cursor repositioning.

[C#]

```
// Sets the AutoIntentMode.
this.editControl1.AutoIndentMode =
Syncfusion.Windows.Forms.Edit.Enums.AutoIndentMode.None;
```

[VB .NET]

```
' Sets the AutoIntentMode.
Me.editControl1.AutoIndentMode =
Syncfusion.Windows.Forms.Edit.Enums.AutoIndentMode.None
```

If Enter is pressed when the AutoIndentMode is set to None, the text is not indented.

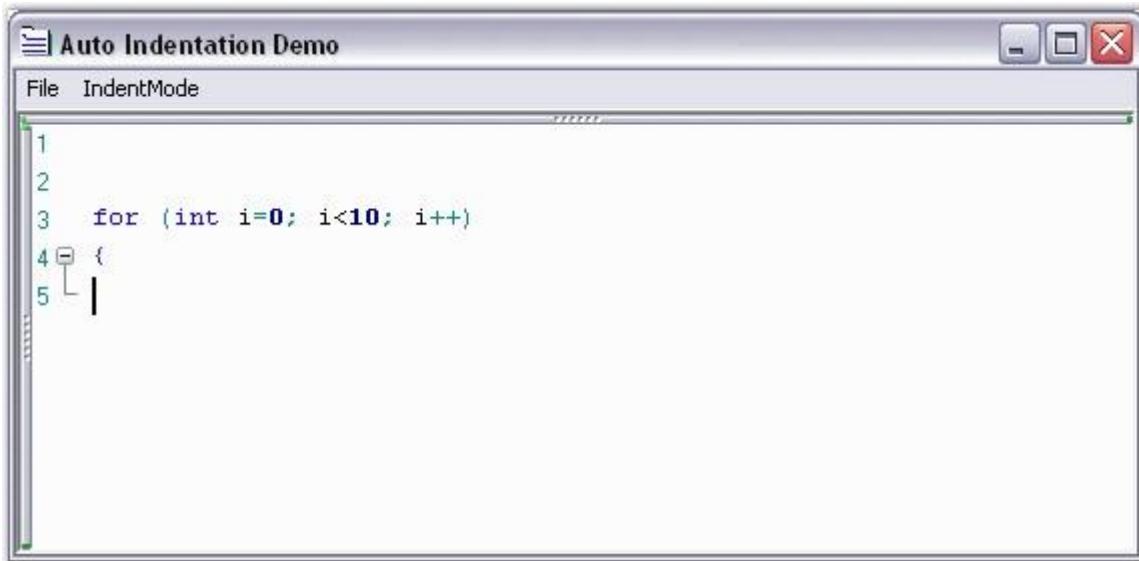


Figure 29: AutoIndentMode = "None"

When the AutoIndentMode is set to **Smart**, the next line is indented by one TabSize from the first column of the previous line on pressing Enter.

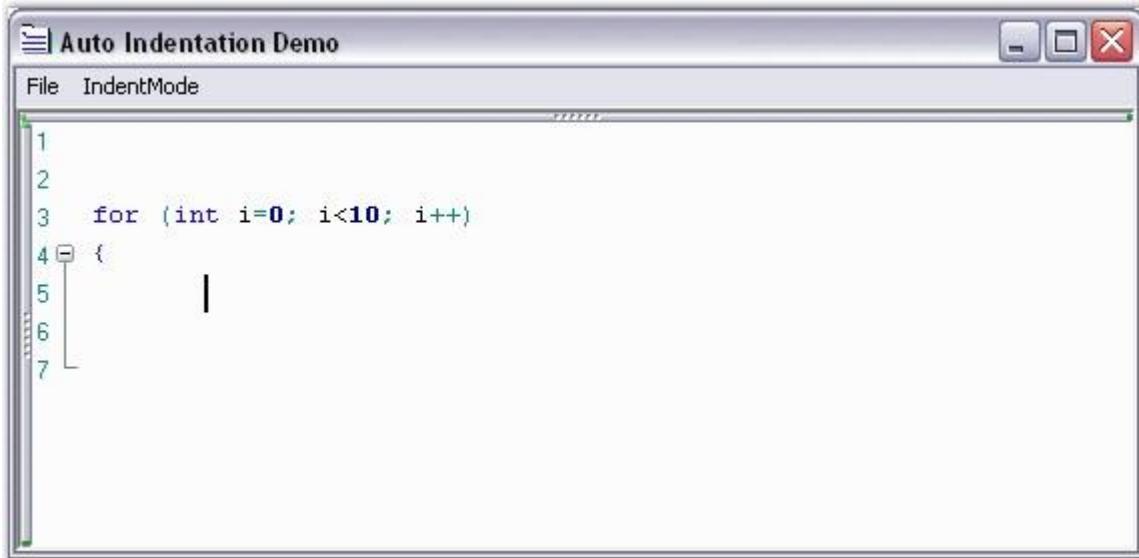


Figure 30: AutoIndentMode = "Smart"

When the AutoIndentMode is set to **Block**, the next line begins at the same column as the previous line on pressing the ENTER key.

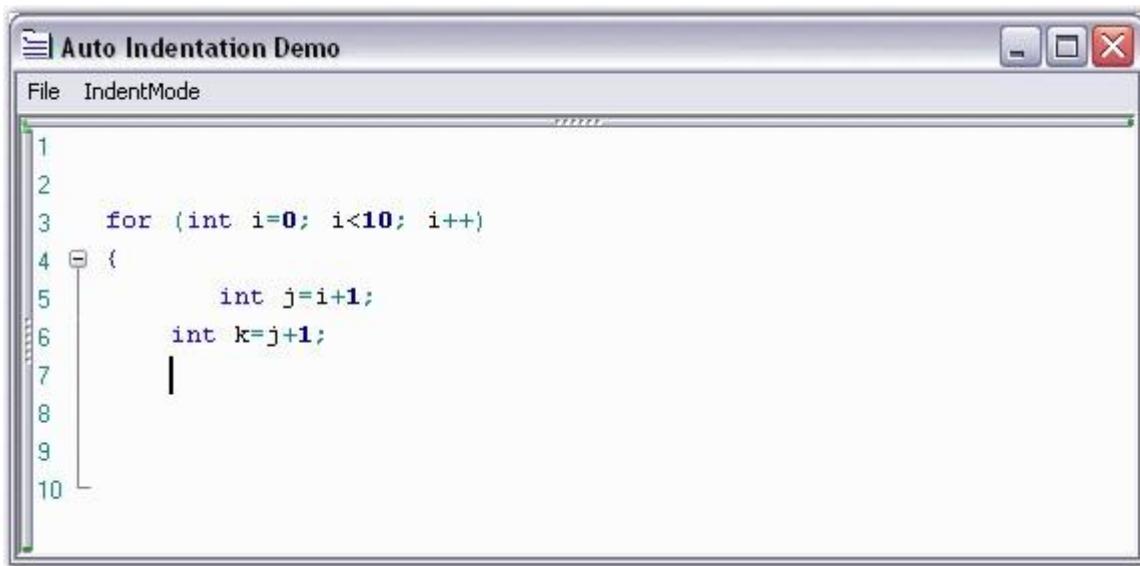


Figure 31: AutoIndentMode = "Block"

The Auto Indentation characters can be specified by setting the **Indent** field to **True** in the lexem definition of the configuration file, as shown below.

[XML]

```
<lexem BeginBlock="{" EndBlock="}" Type="Operator" IsComplex="true"
IsCollapsible="true" Indent="true" CollapseName="..." 
IndentationGuideline="true">
```

A sample which demonstrates Auto Indentation is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Formatting\\AutoIndentationDemo**

4.4.11.2.1 Lexem Support for AutoIndent Block Mode

In the Edit control, the **EnableSmartInBlockIndent** property ensures the AutoIndent Block mode with respect to the lexem's config.indent. With this property, the Block mode will work like Smart mode for conditional statements.

When this property is enabled, the lines will be aligned to the position of the previous indented line. The lines will begin at the original start position if disabled.

Property	Description
EnableSmartInBlockIndent	Gets or sets a value to make the Block mode work like Smart mode for conditional statements.

[C#]

```
// Gets or sets a value to make the Block mode work like Smart mode for  
conditional statements.
```

```
this.editcontrol1.EnableSmartInBlockIndent = true;
```

[VB .NET]

```
// Gets or sets a value to make the Block mode work like Smart mode for  
conditional statements.
```

```
Me.editcontrol1.EnableSmartInBlockIndent = True
```

4.4.11.3 AutoFormatting

The Edit Control offers autoformatting and smart indentation support for code as in Visual Studio. Currently, only C# has built-in support for this feature.

AutoFormatting can be enabled by using the below given method.

Edit Control Method	Description
AutoFormatText	AutoFormats given range of text.

For example, the closing brace gets automatically aligned with the opening brace. Consider some C# code as shown in the below screenshot.

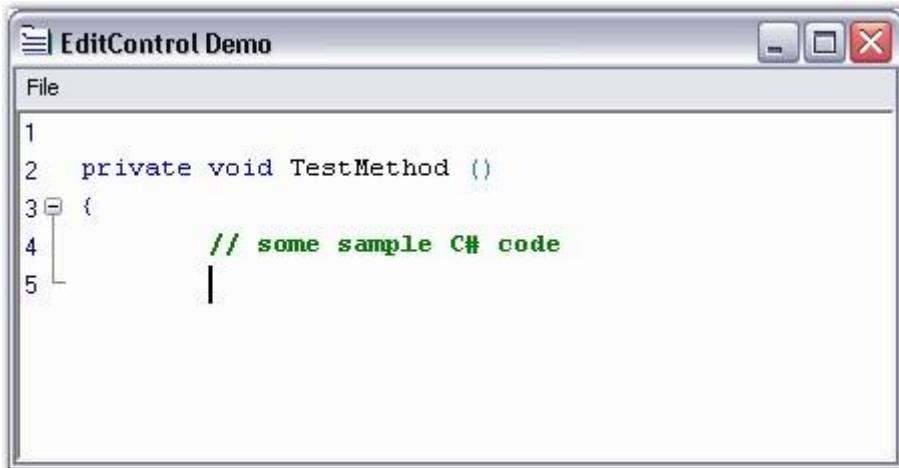


Figure 32: Code is entered into the Edit Control

Now, when the closing brace '}' is typed, it gets automatically aligned with the opening brace, as shown in the screenshot below.

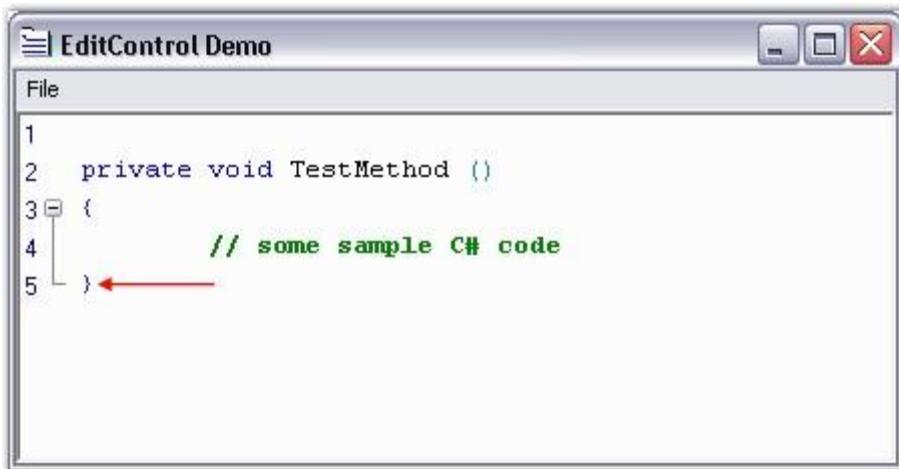


Figure 33: AutoFormatting support for code in Edit Control

 **Note:** The [AutoIndentMode](#) property for the Edit Control should be set to Smart for this purpose.

Essential Edit provides an extensible interface, **IAutoFormatter**, which can be implemented to provide any kind of formatter for any desired language. This can be used to take care of some of the special scenarios explained below.

- XML or HTML text of the following format - <abc> <xyz> </xyz> </abc> should be autoformatted as follows.

[HTML or XML]

```
<abc>
  <xyz>
  ...
  </xyz>
</abc>
```

- Similarly, when the Edit Control is using C# configuration settings, any text enclosed within '{' and '}' should get automatically indented, just as in the VS.NET editor. Also, the closing brace should be automatically indented with its matching opening brace.
- For languages like VB.NET, the End statement should get automatically indented on pressing the ENTER key, after entering the method header for the VB.NET samples.

[VB .NET]

```
Private sub TestMethod()  '----> Method header

  '----> Press Enter key

End sub '----> End statement should be automatically aligned with the
function header
```

4.4.11.4 Unicode

Unicode is a standard used to encode all the languages of the world in computers. It is an international standard used with the goal to resolve ambiguities that traditionally arise with complex scripts like Japanese, Arabian or Chinese, on computer systems. Beside solving many Internationalization issues, Unicode-enabled programs also run faster under Windows NT, 2000 and XP.

Edit Control fully supports serializing and displaying Unicode characters. All Unicode text is saved in UTF-8 format, by default. Moving Unicode text between Edit Control and other Word Processing software programs is also straightforward through Copy / Paste clipboard functions.

Essential Edit also supports handling of all other text encoding formats specified in the **System.Text.Encoding** class like ASCII, UTF7, UTF8 and BigEndianUnicode.

The following screenshot illustrates the use of Chinese, Arabic, Hindi, Russian and Greek text in the Edit Control.

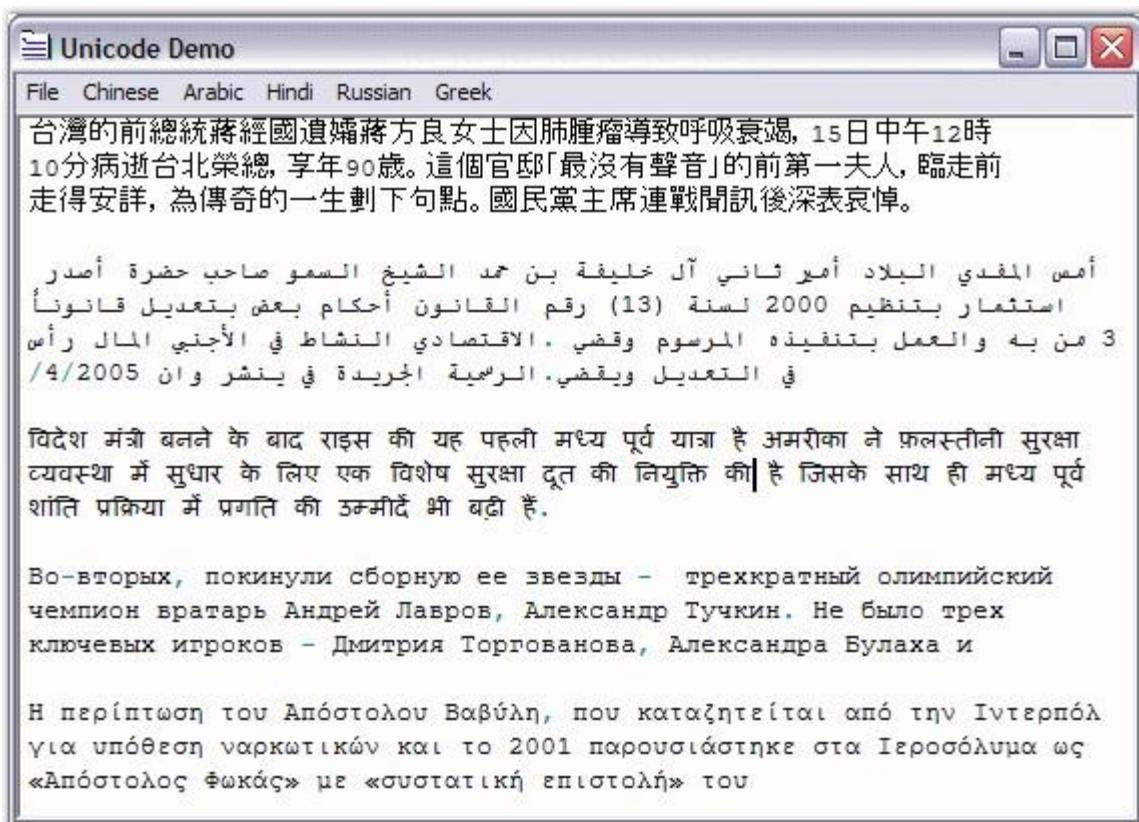


Figure 34: Unicode support in Edit Control

A sample which demonstrates Unicode is available in the following sample installation path.

`..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Formatting\\UnicodeDemo`

4.4.11.5 Automatic Outlining

Outlining can be performed by having appropriate "lexem", "split", and "extension" tag entries in the configuration file. Refer to the [Configuration Settings](#) topic for more information regarding the configuration file.

Essential Edit provides Visual Studio-like support for collapsing and expanding blocks of code through the use of Collapsers (plus-minus buttons). Sections of code which form the outlining blocks can be specified by using the configuration settings. The outlining blocks can be specified for code as well as for plain text.

Setting the **ShowOutliningCollapsers** property to **True**, will enable Automatic Outlining. Edit provides the following APIs to support Outlining.

Edit Control Method	Description
Collapse	Collapses all regions in currently selected area or in the current line.
Expand	Expands all collapsed regions in currently selected area or in the current line.
SwitchCollapsingOn	Turns on collapse and collapse all option.
SwitchCollapsingOff	Turns off collapse option.
CollapseAll	Collapses all regions.
ExpandAll	Expands all collapsed regions.
ToggleLineCollapsing	Toggles collapse option for current line.

[C#]

```
// Enabling Automatic Outlining.
this.editControl1.ShowOutliningCollapsers = true;

// Collapses all regions in currently selected area or in the current line.
this.editControl1.Collapse();

// Expands all collapsed regions in currently selected area or in the current line.
this.editControl1.Expand();

// Turns on collapse and collapse all option.
this.editControl1.SwitchCollapsingOff();

// Turns off collapse option.
this.editControl1.SwitchCollapsingOn();

// Collapses all regions.
this.editControl1.CollapseAll();

// Expands all collapsed regions.
this.editControl1.ExpandAll();

// Toggles collapse option for current line.
this.editControl1.ToggleLineCollapsing();
```

[VB.NET]

```
' Enabling Automatic Outlining.  
Me.editControl1.ShowOutliningCollapsers = True  
  
' Collapses all regions in currently selected area or in the current line.  
Me.editControl1.Collapse()  
  
' Expands all collapsed regions in currently selected area or in the current line.  
Me.editControl1.Expand()  
  
' Turns on collapse and collapse all option.  
Me.editControl1.SwitchCollapsingOff()  
  
' Turns off collapse option.  
Me.editControl1.SwitchCollapsingOn()  
  
' Collapses all regions.  
Me.editControl1.CollapseAll()  
  
' Expands all collapsed regions.  
Me.editControl1.ExpandAll()  
  
' Toggles collapse option for current line.  
Me.editControl1.ToggleLineCollapsing()
```

Outlining Operations

The Edit Control supports the following events to handle the various Outlining operations.

Edit Control Event	Description
OutliningBeforeCollapse	Occurs before the region is about to collapse.
OutliningBeforeExpand	Occurs before the region is about to expand.
OutliningCollapse	Occurs when the region collapses.
OutliningExpand	Occurs when the region expands.
CollapsedAll	Occurs when CollapseAll method was called.
ExpandedAll	Occurs when ExpandedAll method was called.

CollapsingAll	Occurs when CollapseAll method is called.
ExpandingAll	Occurs when ExpandAll method is called.

The above events can be canceled, and can be used to optionally cancel the Outlining Collapse and Expand operations respectively. They are discussed in detail in the Edit Control Events section.

The Custom Outlining Demo sample demonstrates how the outlining feature can be used on any custom file or plain text, and not necessarily on programming language code samples. This sample is available in the following location.

..\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Formatting\\CustomOutliningDemo

4.4.11.5.1 Outlining Tooltip

Outlining Tooltip is displayed for each collapsed outlining block, and it shows the contents of the collapsed block. This feature is similar to the one available in Visual Studio.NET editor.

The Outlining Tooltip can be optionally shown / hidden by using the **ShowOutliningTooltip** property in the Edit Control.

[C#]

```
this.editControl1.ShowOutliningTooltip = true;
```

[VB .NET]

```
Me.editControl1.ShowOutliningTooltip = True
```



Figure 35: Outlining Tooltip displaying the Collapsed Block of Text

Using Events

Edit Control supports the following Outlining Tooltip events.

Edit Control Event	Description
OutliningTooltipBeforePopup	Occurs when outlining tooltip is about to be shown.
OutliningTooltipPopup	Occurs when outlining tooltip is shown.
OutliningTooltipClose	Occurs when outlining tooltip is closed.

The **OutliningTooltipBeforePopup** event is used to control the visibility of the outlining tooltip. The **ShowMode** property of the **OutliningTooltipBeforePopupEventArgs** is used for this purpose. By default, the ShowMode property is set to **On**.

```
[C#]

private void editControl1_OutliningTooltipBeforePopup(object sender,
Syncfusion.Windows.Forms.Edit.OutliningTooltipBeforePopupEventArgs e)
{
// To display the outlining tooltip
e.ShowMode = OutliningTooltipShowMode.On;
```

```
// To hide the outlining tooltip  
e.ShowMode = OutliningTooltipShowMode.Off;  
}
```

[VB .NET]

```
Private Sub editControl1_OutliningTooltipBeforePopup(sender As Object, e As Syncfusion.Windows.Forms.Edit.OutliningTooltipBeforePopupEventArgs) Handles editControl1.OutliningTooltipBeforePopup  
  
' To display the outlining tooltip  
e.ShowMode = OutliningTooltipShowMode.On  
  
' To hide the outlining tooltip  
e.ShowMode = OutliningTooltipShowMode.Off  
  
End Sub
```

See Also

[Automatic Outlining](#)

4.4.11.6 Wordwrap

Wordwrap allows users to view the entire contents of a line, by wrapping text at the edge of the control (or text area) into one or more lines, that normally would have been outside the view in the Edit Control.

Edit Control allows advanced customization by using the Wordwrap functionality.

Type of Wordwrap

Wordwrap is enabled by setting the WordWrap property of the Edit Control to True. The two types of Wordwrap in Edit Control have been explained below.

Edit Control Property	Description
WordWrap	Gets / sets state of the word wrapping mode.
WordWrapType	Gets / sets type of word wrapping. The options provided are

- | | |
|--|---|
| | <ul style="list-style-type: none"> • <i>WrapByChar</i> - wraps the text by individual characters • <i>WrapByWord</i> - wraps the text by individual words |
|--|---|

The default value is *WrapByWord*.

[C#]

```
// WordWrap property set.

this.editControl1.WordWrap = true;

// WordWrapType property set.

this.editControl1.WordWrapType =
Syncfusion.Windows.Forms.Edit.Enums.WordWrapType.WrapByChar;
```

[VB .NET]

```
' WordWrap property set.

Me.editControl1.WordWrap = True

' WordWrapType property set.

Me.editControl1.WordWrapType =
Syncfusion.Windows.Forms.Edit.Enums.WordWrapType.WrapByChar
```

Wordwrap Mode

The following properties are associated with setting the mode of Word Wrapping.

Edit Control Property	Description
WordWrapMode	<p>Gets / sets state of the word wrapping mode. The options provided are</p> <p><i>WordWrapMargin</i> - wraps text at the boundary between text area and wordwrap margin of the Edit Control</p> <p>The area beyond the text area in the Edit Control is referred to as</p>

	<p>the wordwrap margin. Hence, the width of the wordwrap margin is the difference between Edit Control's width and the TextArea width.</p> <p><i>Control</i> - wraps the text at the edge of the Edit Control</p> <p><i>SpecifiedColumn</i> - wraps the text at the specified column that is specified in WordWrapColumn property</p> <p>The default value is set to Control.</p>
WordWrapColumnMeasuringFont	Gets / sets the font used while calculating the position of WordWrapColumn.
WordWrapColumn	<p>Specifies column for wrapping text. Used when WordWrapMode is set to SpecifiedColumn.</p> <p>The default value is 100.</p>
TextAreaWidth	<p>Gets / sets the width of the text area of the Edit Control.</p> <p>The default value is 600.</p>
WrappedLinesOffset	Specifies offset of wrapped lines.

[C#]

```
// Sets the WordWrap mode.
this.editControl1.WordWrapMode =
Syncfusion.Windows.Forms.Edit.Enums.WordWrapMode.WordWrapMargin;

// Sets font that is used while calculating the position of the WordWrap
// column.
this.editControl1.WordWrapColumnMeasuringFont = new
System.Drawing.Font("Arial", 9.75F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));

// Specifies column for wrapping text.
this.editControl1.WordWrapColumn = 125;
```

```
// Set the width of the EditControl's text area.  
this.editControl1.TextAreaWidth = 300;  
  
// Specifies offset for the wrapped lines.  
this.editControl1.WrappedLinesOffset = 10;
```

[VB.NET]

```
' Sets the WordWrap mode.  
Me.editControl1.WordWrapMode =  
Syncfusion.Windows.Forms.Edit.Enums.WordWrapMode.WordWrapMargin  
  
' Sets font that is used while calculating the position of the WordWrap  
column.  
Me.editControl1.WordWrapColumnMeasuringFont = New  
System.Drawing.Font("Arial", 9.75F, System.Drawing.FontStyle.Regular,  
System.Drawing.GraphicsUnit.Point, (CType((0), Byte)))  
  
' Specifies column for wrapping text.  
Me.editControl1.WordWrapColumn = 125  
  
' Set the width of the EditControl's text area.  
Me.editControl1.TextAreaWidth = 300  
  
' Specifies offset for the wrapped lines.  
Me.editControl1.WrappedLinesOffset = 10
```

The following illustration shows the Edit Control with the WordWrappingMode and WordWrapType properties set.

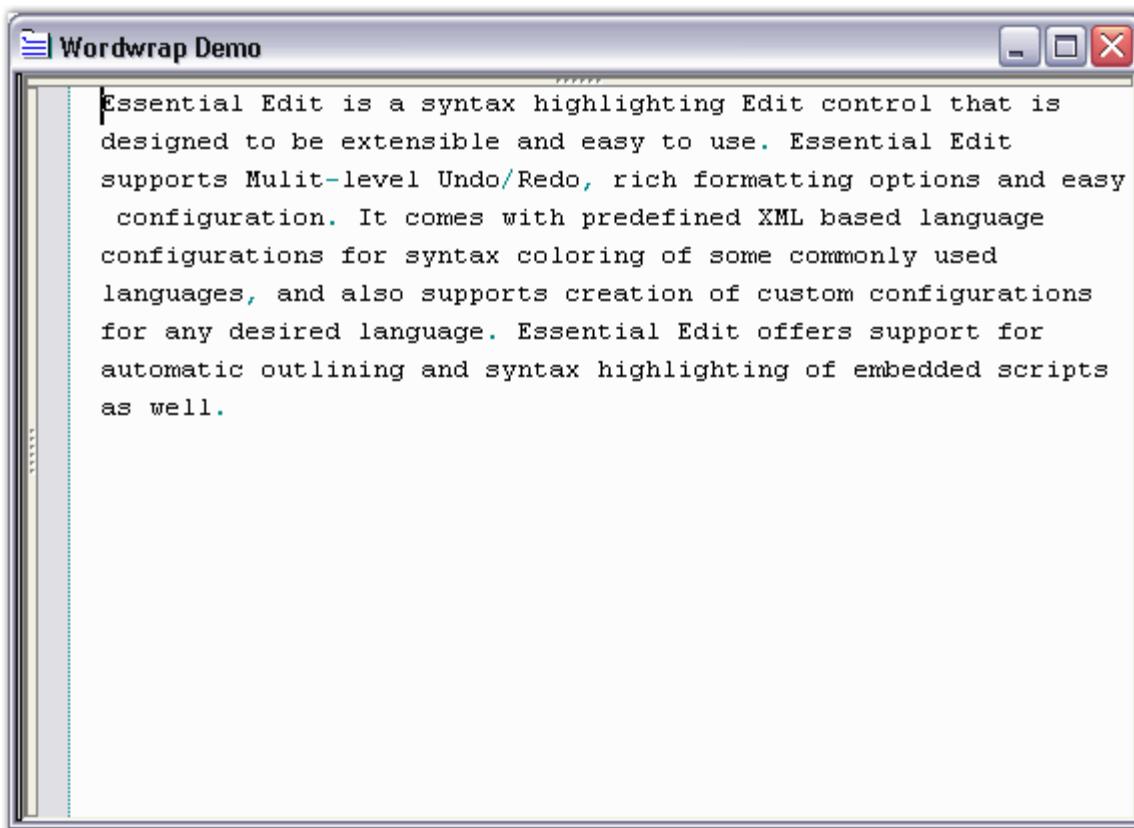


Figure 36: *WordWrappingMode = "Control"; WordWrapType= "WrapByWord"*

Refer to the WordWrap Demo sample in the following sample installation location.

```
..\My Documents\Syncfusion\EssentialStudio\Version  
Number\Windows>Edit.Windows\Samples\2.0\Text Formatting\WordwrapDemo
```

4.4.11.6.1 Wordwrap Margin Customization and Wrapping Images

This section discusses the wordwrap margin customization settings. Also, it discusses how images can be set for the wrapped and wrapping lines of the Edit Control.

Margin Line Style and Line Color Settings

Wordwrap margin of the Edit Control can be set and customized by using the below given properties.

Edit Control Property	Description
WordWrapMarginVisible	Gets / sets value indicating whether the wordwrap margin should be visible.
WordWrapMarginLineStyle	Specifies style of line that is drawn at the border of the wordwrap margin. The options provided are <ul style="list-style-type: none"> • Solid • Dash • Dot • DashDot • DashDotDot • Custom The default value is Solid .
WordWrapMarginLineColor	Sets custom color for the line that is drawn at the border of the wordwrap margin.
WordWrapMarginBrush	Gets / sets BrushInfo object that is used when the area situated after the text area is drawn.

[C#]

```
// Specifies whether the wordwrap margin should be visible.
this.editControl1.WordWrapMarginVisible = true;

// Specifies the line style of the wordwrap margin.
this.editControl1.WordWrapMarginLineStyle = DashStyle.Dash;

// Specifies the line color of the wordwrap margin.
this.editControl1.WordWrapMarginLineColor = Color.Green;

// Specifies the BrushInfo object that is used when the area situated after
// the text area is drawn.
this.editControl1.WordWrapMarginBrush = new
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.Horizontal,
System.Drawing.Color.White, System.Drawing.Color.LightSalmon);
```

[VB .NET]

```
' Specifies whether the wordwrap margin should be visible.
Me.editControl1.WordWrapMarginVisible = True

// Specifies the line style of the wordwrap margin.
```

```
Me.editControl1.WordWrapMarginLineStyle =
System.Drawing.Drawing2D.DashStyle.Dash

// Specifies the line color of the wordwrap margin.
Me.editControl1.WordWrapMarginLineColor = System.Drawing.Color.Green

// Specifies the BrushInfo object that is used when the area situated after
the text area is drawn.
Me.editControl1.WordWrapMarginBrush = New
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.Horizontal,
System.Drawing.Color.White, System.Drawing.Color.LightSalmon)
```

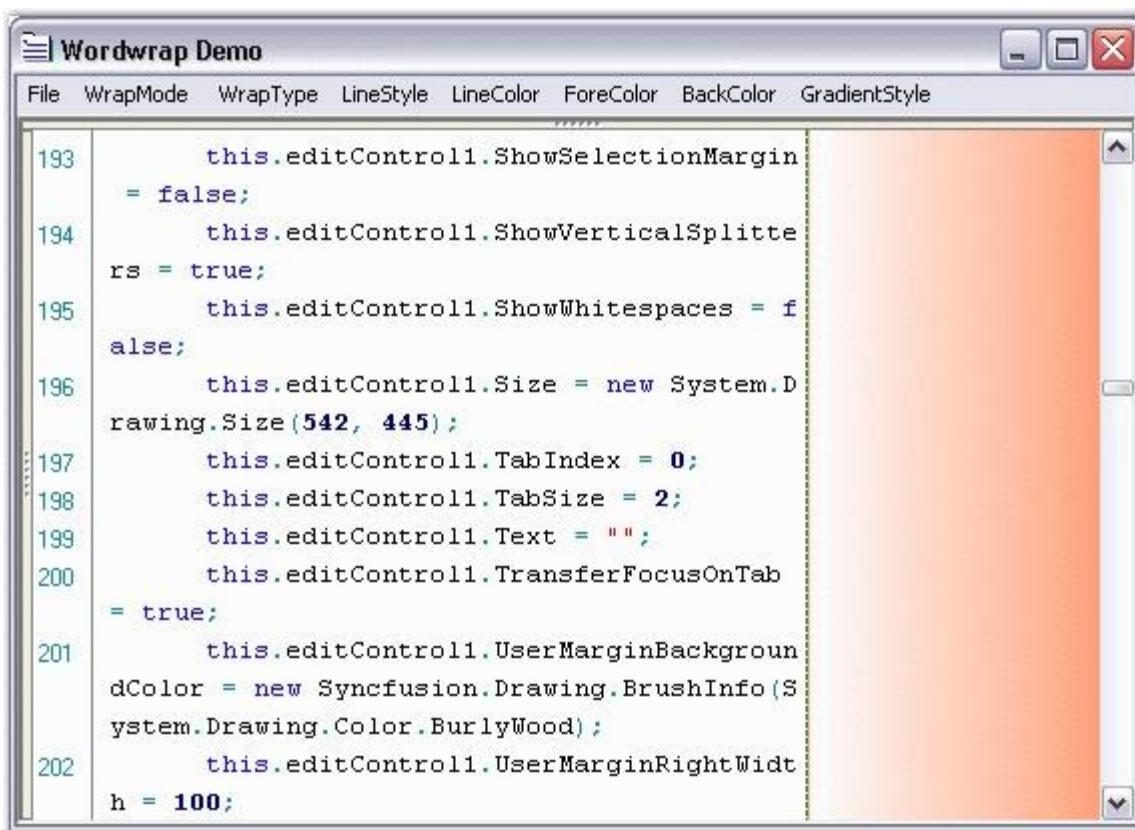


Figure 37: Edit Control with Character Wrapping and Custom Painted Wordwrap Margin

Line Wrapping Images

It is also possible to associate images to indicate line wrapping. This feature can be turned on by setting the **MarkLineWrapping** property to **True**. There can be two types of image indicators:

1. Images that indicate the line that is being wrapped. These are displayed at the beginning of the line being wrapped. This can be set by using the **CustomWrappedLinesMarkingImage** property.

2. Images that indicate the point at which the line is being wrapped. This can be set by using the **CustomLineWrappingMarkingImage** property.

Also, to indicate whether wrapped lines should be marked, the **MarkWrappedLines** property can be used.

Edit Control Property	Description
MarkLineWrapping	Specifies whether line wrapping should be marked.
MarkWrappedLines	Specifies whether wrapped lines should be marked.
CustomWrappedLinesMarkingImage	Gets / sets custom image that marks wrapped lines.
CustomLineWrappingMarkingImage	Gets / sets custom image that marks wrapping lines.

[C#]

```
// Enable images to indicate line wrapping.
this.editControl1.MarkLineWrapping = true;

// Images that indicate the line that is being wrapped.
this.editControl1.CustomWrappedLinesMarkingImage =
((System.Drawing.Image)(resources.GetObject("$this.Sunset")));

// Images that indicate the point at which the line is being wrapped.
this.editControl1.CustomLineWrappingMarkingImage =
((System.Drawing.Image)(resources.GetObject("$this.Blue_hills")));

// Indicate wrapped lines.
this.editControl1.MarkWrappedLines = true;
```

[VB .NET]

```
' Enable images to indicate line wrapping.
Me.editControl1.MarkLineWrapping = True

' Images that indicate the line that is being wrapped.
Me.editControl1.CustomWrappedLinesMarkingImage =
(CType(resources.GetObject("$this.Sunset"), System.Drawing.Image))
```

```
' Images that indicate the point at which the line is being wrapped.
Me.editControl1.CustomLineWrappingMarkingImage =
(CType(resources.GetObject("$this.Blue_hills")), System.Drawing.Image)

' Indicate wrapped lines.
Me.editControl1.MarkWrappedLines = True
```

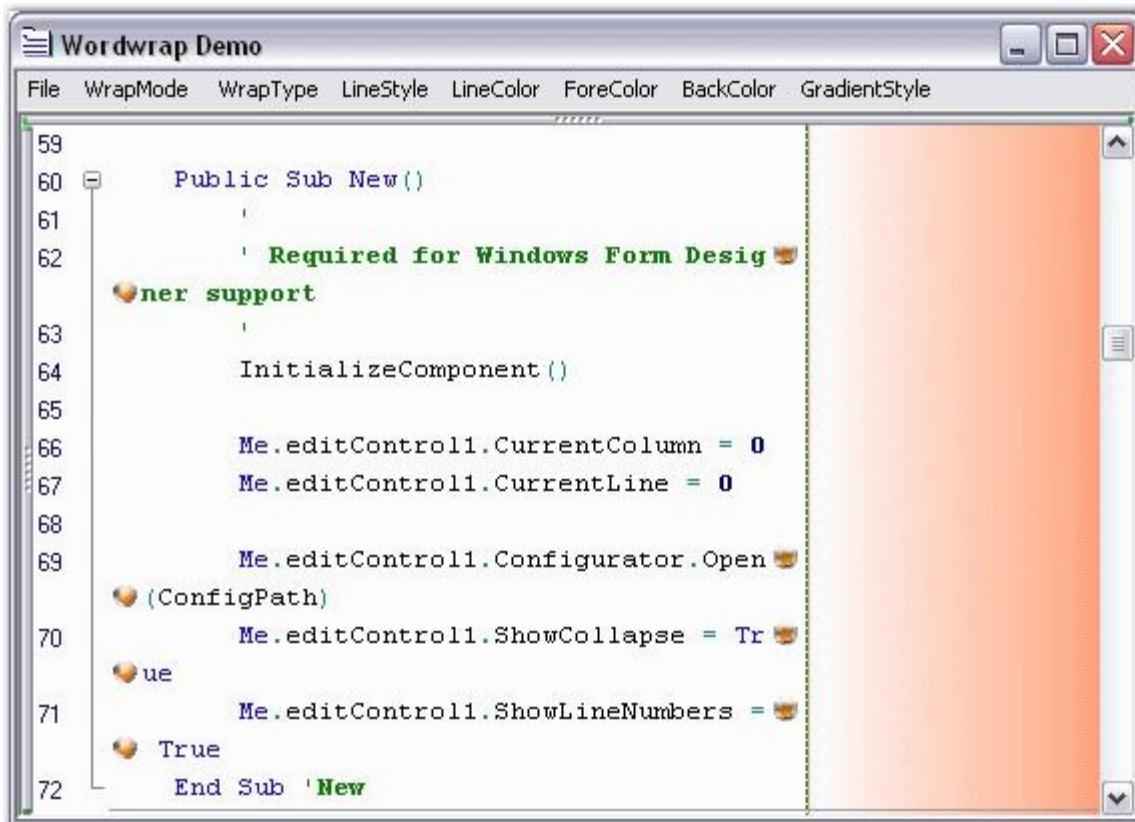


Figure 38: Wrapping Images indicating Wrapped Lines and Point of Wrapping

4.4.11.7 Read-Only Text

Edit Control allows you to specify read-only regions in the code, i.e., regions that are uneditable. This can be achieved through the following methods.

Edit Control Method	Description
MarkAsReadOnly	Sets text as read-only.
RemoveReadOnly	Removes read-only status of specified region.

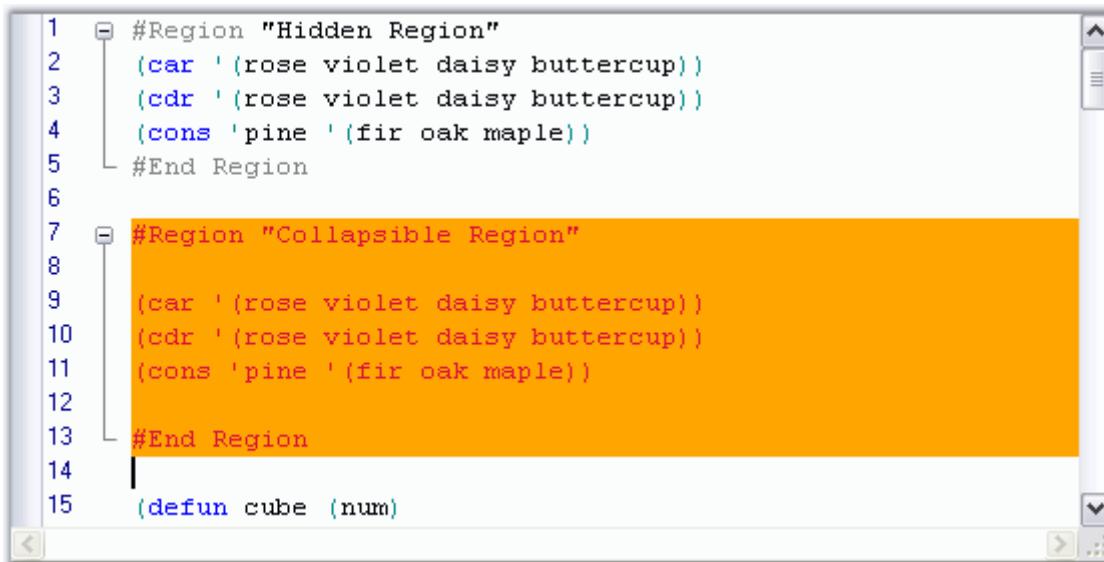
[C#]

```
// Specify a read-only region.  
this.editControl1.MarkAsReadOnly(this.editControl1.Selection.Start,  
this.editControl1.Selection.End, Color.Orange, Color.Crimson);  
  
// Reset a read-only region.  
this.editControl1.RemoveReadOnly(this.editControl1.Selection.Start,  
this.editControl1.Selection.End);
```

[VB.NET]

```
' Specify a read-only region.  
Me.editControl1.MarkAsReadOnly(Me.editControl1.Selection.Start,  
Me.editControl1.Selection.End, Color.Orange, Color.Crimson)  
  
' Reset a read-only region.  
Me.editControl1.RemoveReadOnly(Me.editControl1.Selection.Start,  
Me.editControl1.Selection.End)
```

The following screen shot shows a read-only region in the code section of the Edit Control.



The screenshot shows a code editor window with the following content:

```
1 #Region "Hidden Region"  
2   (car '(rose violet daisy buttercup))  
3   (cdr '(rose violet daisy buttercup))  
4   (cons 'pine '(fir oak maple))  
5 #End Region  
6  
7 #Region "Collapsible Region"  
8  
9   (car '(rose violet daisy buttercup))  
10  (cdr '(rose violet daisy buttercup))  
11  (cons 'pine '(fir oak maple))  
12  
13 #End Region  
14  
15 (defun cube (num))
```

The region from line 7 to line 13 is highlighted with an orange background and crimson text color, indicating it is a read-only region.

Figure 39: Read-Only Region with Orange Background and Crimson Text Color

A sample which demonstrates this feature is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\
TextRangeCustomizationDemo**

4.4.12 Customizing Text

The following text customization features are discussed in this section:

4.4.12.1 Text Color

This section discusses how the text color of the Edit Control can be changed.

The text color of the Edit Control is set by using the **SetTextColor** method.

[C#]

```
// Set the color of the text for the Edit Control.  
this.editControl1.SetTextColor(new Point(1, 1), new Point(5, 5),  
Color.Orange);
```

[VB.NET]

```
' Set the color of the text for the Edit Control.  
Me.editControl1.SetTextColor(New Point(1, 1), New Point(5, 5), Color.Orange)
```

4.4.12.2 Text Border

This section discusses how borders can be set for the text in the Edit Control.

Edit Control supports borders for its text by using the methods given below.

Edit Control Method	Description
SetTextBorder	Sets border around text.
RemoveTextBorder	Removes border around text with given coordinates.

Edit Control Border Enumerator	Description
FrameBorderStyle	Specifies the style of border line. The options provided are <ul style="list-style-type: none"> • Dash • DashDot • Dot • None • Solid • Wave
BorderWeight	Specifies the weight of the border line. The options provided are <ul style="list-style-type: none"> • Bold • Double • Thin

[C#]

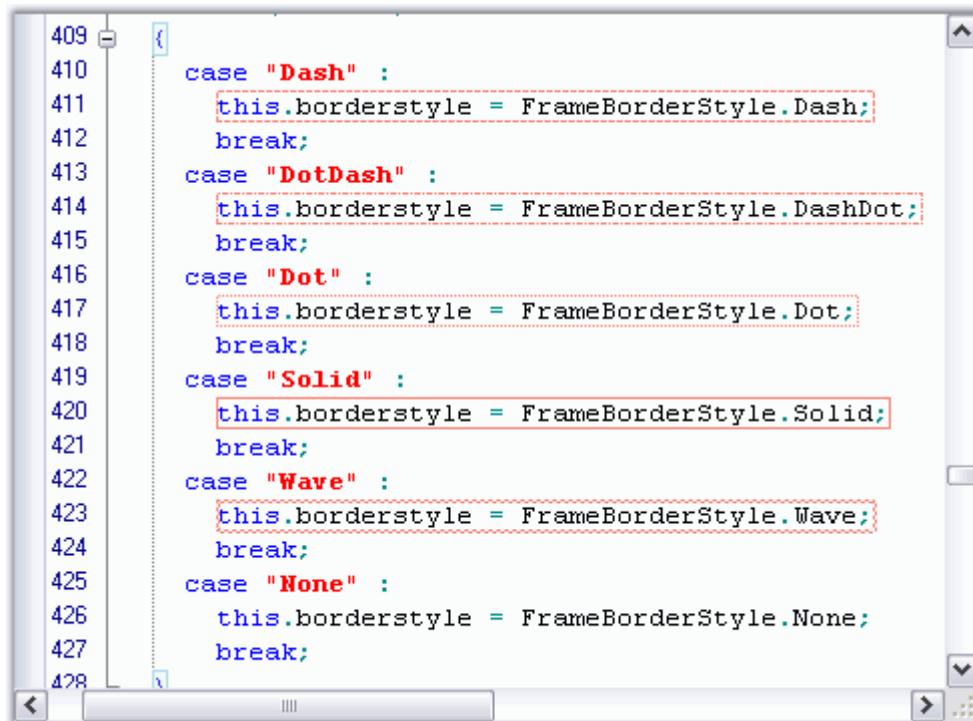
```
// Set borders for the specified text range.
this.editControl1.SetTextBorder(new Point(1, 13), new Point(15, 13),
Color.Red, FrameBorderStyle.Wave, BorderWeight.Double);

// Remove borders from the specified text range.
this.editControl1.RemoveTextBorder(new Point(1, 13), new Point(15, 13);
```

[VB .NET]

```
' Set borders for the specified text range.
Me.editControl1.SetTextBorder(New Point(1, 13), New Point(15, 13), Color.Red,
FrameBorderStyle.Wave, BorderWeight.Double)

' Remove borders from the specified text range.
Me.editControl1.RemoveTextBorder(New Point(1, 13), New Point(15, 13))
```



```

409     case "Dash" :
410         this.borderstyle = FrameBorderStyle.Dash;
411         break;
412     case "DotDash" :
413         this.borderstyle = FrameBorderStyle.DashDot;
414         break;
415     case "Dot" :
416         this.borderstyle = FrameBorderStyle.Dot;
417         break;
418     case "Solid" :
419         this.borderstyle = FrameBorderStyle.Solid;
420         break;
421     case "Wave" :
422         this.borderstyle = FrameBorderStyle.Wave;
423         break;
424     case "None" :
425         this.borderstyle = FrameBorderStyle.None;
426         break;
427
428

```

Figure 40: Text Borders in Edit Control

A sample which demonstrates the above feature is available in the following sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\BordersDemo

See Also

[Underlines, Wavelines and StrikeThrough](#)

4.4.12.3 Encoding Text

Edit Control facilitates saving the contents of a file in any desired encoding and new line style. This can be accomplished by using the below given method.

Edit Control Method	Description
SaveFile	Saves content to the specified file.

[C#]

```
this.editControl1.SaveFile("EditControl", Encoding.Unicode,  
Syncfusion.IO.NewLineStyle.Mac);
```

[VB.NET]

```
Me.editControl1.SaveFile("EditControl", Encoding.Unicode,  
Syncfusion.IO.NewLineStyle.Mac)
```

Edit Control supports all the encoding styles supported by the **System.Text.Encoding** enumerator. The below given methods can be used to get / set the encoding style for the text in the Edit Control.

Edit Control Method	Description
GetEncoding	Gets the current text encoding.
SetEncoding	Sets the current text encoding. The options provided are <ul style="list-style-type: none">• ASCII• BigEndianUnicode• Default• UTF32• UTF7• UTF8• Unicode

[C#]

```
// Gets the current text encoding.  
this.editControl1.GetEncoding();  
  
// Sets the current text encoding.  
this.editControl1.SetEncoding(Encoding.ASCII);
```

[VB.NET]

```
' Gets the current text encoding.  
Me.editControl1.GetEncoding()  
  
// Sets the current text encoding.
```

```
Me.editControl1.SetEncoding(Encoding.ASCII)
```

It also supports all the new line styles supported by the **Syncfusion.IO.NewLineStyle** enumerator - **Windows, Mac, Unix** and **Control**.

New Line Styles	Description
Windows	\r\n
Mac	\r
Unix	\n\r
Control	\n

The **SaveFilewithDataLoss** and **SaveStreamWithDataLoss** events are fired whenever there is a data loss while saving the file by using the specified encoding format. Files or streams can be corrupted if you have some Unicode characters that cannot be saved using the specified encoding format. For example, if you have a file or stream that contains some specific characters of German language, and if you try to save it using ASCII encoding, then data loss will occur. If the save operation is not canceled here, characters will be saved incorrectly.

4.4.12.4 Text Selection

The Edit Control supports text selection operations through the use of the APIs discussed in this section.

Selecting Text

Edit Control provides support to select text programmatically. The **StartSelection** and **StopSelection** methods are used to programmatically specify the starting and ending bounds for the text to be selected.

Edit Control Method	Description
StartSelection	Sets selection start at the specified position in text.
StopSelection	Sets selection end at the specified position in text.
SetSelection	Sets selected area of the text.

SelectLine	Selects line with specified index.
SelectAll	Selects all text.

Line selection in Edit Control is extended by using the **ExtendSelectionToFarRight** property.

Edit Control Property	Description
ExtendSelectionToFarRight	Gets / sets value indicating whether line selection should be extended to the far right.

[C#]

```
// Specifies start position for selecting text.
this.editControl1.StartSelection(1, 1);

// Specifies end position for selecting text.
this.editControl1.StopSelection(10, 1);

// Selects line with specified index.
this.editControl1.SelectLine(5);

// Extend line selection to far right.
this.editControl1.ExtendSelectionToFarRight = true;
```

[VB.NET]

```
' Specifies start position for selecting text.
Me.editControl1.StartSelection(1, 1)

' Specifies end position for selecting text.
Me.editControl1.StopSelection(5, 1)

' Selects line with specified index.
Me.editControl1.SelectLine(5)

' Extend line selection to far right.
Me.editControl1.ExtendSelectionToFarRight = True
```

Text can also be selected after drag / drop operations by using the below given property.

Edit Control Property	Description
-----------------------	-------------

SelectTextAfterDragDrop	Specifies whether text should be selected after drag / drop operations.
-------------------------	---

[C#]

```
this.editControl1.SelectTextAfterDragDrop = true;
```

[VB .NET]

```
Me.editControl1.SelectTextAfterDragDrop = True
```

Selected Text

The following properties can be used to get / set selected text.

Edit Control Property	Description
SelectedText	Gets / sets selected text.
Selection	Gets selected text range.

[C#]

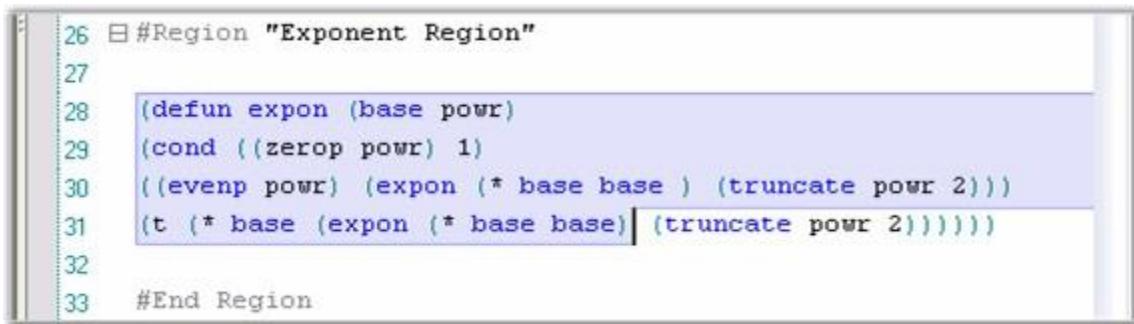
```
// Returns the currently selected text in the Edit Control.  
string editText = this.editControl1.SelectedText;
```

[VB .NET]

```
' Returns the currently selected text in the Edit Control.  
Dim editText as String = Me.editControl1.SelectedText
```

Transparent Selection

Setting the **TransparentSelection** property to **True**, will highlight the selected text range with a transparent blue background (which will let you view the syntax highlighting in the text within the selected region), as shown in the following screenshot.



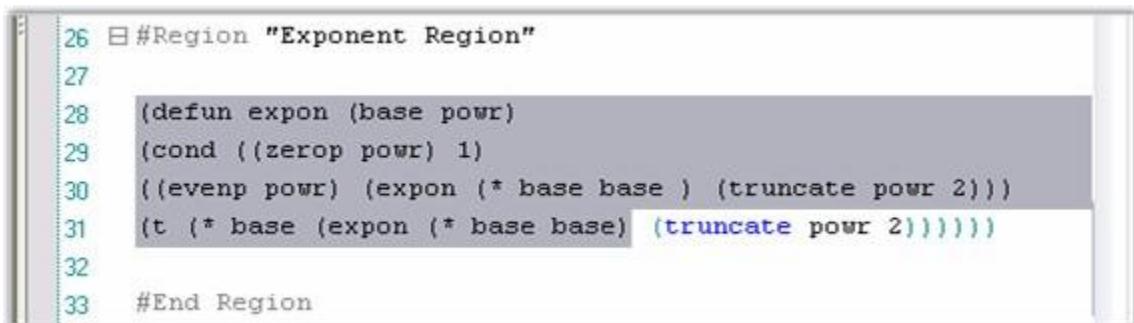
```

26 #Region "Exponent Region"
27
28 (defun expon (base powr)
29 (cond ((zerop powr) 1)
30 ((evenp powr) (expon (* base base ) (truncate powr 2)))
31 (t (* base (expon (* base base) (truncate powr 2))))))
32
33 #End Region

```

Figure 41: Transparent Selection Enabled

Setting the **TransparentSelection** property to **False**, will highlight the selected text range with a dark background (which will not let you view the syntax highlighting in the text within the selected region), as shown in the following screenshot.



```

26 #Region "Exponent Region"
27
28 (defun expon (base powr)
29 (cond ((zerop powr) 1)
30 ((evenp powr) (expon (* base base ) (truncate powr 2)))
31 (t (* base (expon (* base base) (truncate powr 2))))))
32
33 #End Region

```

Figure 42: Transparent Selection Disabled

Cancelling / Resetting Selection

Text selection can be either cancelled or reset by using the below given methods.

Edit Control Method	Description
SelectionCancel	Removes selection and causes invalidation of the area that was selected.
ResetSelection	Resets selection.

[C#]

```
// Removes selection from text.
this.editControl1.SelectionCancel();
```

```
// Resets selection.  
this.editControl1.ResetSelection();
```

[VB.NET]

```
' Removes selection from text.  
Me.editControl1.SelectionCancel()  
  
' Resets selection.  
Me.editControl1.ResetSelection()
```

4.5 Syntax Highlighting and Code Coloring

Essential Edit supports Syntax Highlighting and Code Coloring of some of the commonly used languages with the help of configuration files. It provides pre-defined configuration files for languages like SQL, Delphi or Pascal, HTML, VB.NET, XML, Java, VBScript, JScript and C#.

These configuration settings are made available in the **EditControl.Configurator.KnownLanguages** collection. The order of the languages in this collection is as follows: C#, Delphi, HTML, Java, JScript, Default Text, SQL, VB.NET, VBScript and XML.

Pre-defined Configuration Files

You can set the Edit Control to use any of the pre-defined configuration settings by using the **ApplyConfiguration** method, as shown below.

[C#]

```
// Considering configuration settings for SQL as an example.  
// Using the KnownLanguages enumerator.  
this.editControl1.ApplyConfiguration(KnownLanguages.SQL);  
  
// Using the file extension of the associated language.  
this.editControl1.ApplyConfiguration(this.editControl1.Configurator.GetLanguage("sql") as IConfigLanguage);  
  
// Using the associated index in the KnownLanguages collection.
```

```
this.editControl1.ApplyConfiguration(this.editControl1.Configurator.KnownLanguageNames[1] as IConfigLanguage);

// Using the name of the language in the associated configuration file.
IConfigLanguage config = this.editControl1.Configurator.GetLanguage("sql") as IConfigLanguage;
this.editControl1.ApplyConfiguration(config.Language);
```

[VB.NET]

```
' Considering configuration settings for SQL as an example.
' Using the KnownLanguages enumerator.
Me.editControl1.ApplyConfiguration(KnownLanguages.SQL)

' Using the file extension of the associated language.
Me.editControl1.ApplyConfiguration(Me.editControl1.Configurator.GetLanguage("sql"))

' Using the associated index in the KnownLanguages collection.
Me.editControl1.ApplyConfiguration(Me.editControl1.Configurator.KnownLanguageNames(1))

' Using the name of the language in the associated configuration file.
Dim config As IConfigLanguage =
Me.EditControl1.Configurator.GetLanguage("sql")
Me.editControl1.ApplyConfiguration(config.Language)
```

You can also load any of the configuration settings by using the **ResetColoring** method, as shown in the code below.

[C#]

```
// Set the Edit Control to use the configuration settings for the default language.
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages[0] as IConfigLanguage);

// Set the Edit Control to use the configuration settings for SQL.
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages[1] as IConfigLanguage);

// Set the Edit Control to use the configuration settings for SQL using the file extension.
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("sql") as IConfigLanguage);
```

```
// Set the Edit Control to use the configuration settings for Pascal.  
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages  
[2] as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for Pascal using  
the file extension.  
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("p  
as") as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for HTML (light).  
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages  
[3] as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for HTML (light)  
using the file extension.  
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("h  
tml") as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for VB.NET.  
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages  
[4] as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for VB.NET using  
the file extension.  
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("v  
b") as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for XML.  
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages  
[5] as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for XML using the  
file extension.  
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("x  
ml") as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for C#.  
this.editControl1.ResetColoring(this.editControl1.Configurator.KnownLanguages  
[6] as IConfigLanguage);  
  
// Set the Edit Control to use the configuration settings for C# using the  
file extension.  
this.editControl1.ResetColoring(this.editControl1.Configurator.GetLanguage("c  
s") as IConfigLanguage);
```

[VB.NET]

```
' Set the Edit Control to use the configuration settings for the default
```

```
language.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(0))  
  
' Set the Edit Control to use the configuration settings for SQL.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(1))  
  
' Set the Edit Control to use the configuration settings for SQL using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("sql"))  
  
' Set the Edit Control to use the configuration settings for Pascal.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(2))  
  
' Set the Edit Control to use the configuration settings for Pascal using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("pas"))  
  
' Set the Edit Control to use the configuration settings for HTML (light).  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(3))  
  
' Set the Edit Control to use the configuration settings for HTML using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("html"))  
  
' Set the Edit Control to use the configuration settings for VB.NET.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(4))  
  
' Set the Edit Control to use the configuration settings for VB.NET using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("vb"))  
  
' Set the Edit Control to use the configuration settings for XML.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(5))  
  
' Set the Edit Control to use the configuration settings for XML using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("xml"))  
  
' Set the Edit Control to use the configuration settings for C#.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.KnownLanguages(6))  
  
' Set the Edit Control to use the configuration settings for C# using the  
file extension.  
Me.editControl1.ResetColoring(Me.editControl1.Configurator.GetLanguage("cs"))
```

External Configuration File

You can plug-in an external configuration file that defines a custom language to the Edit Control by using the **Configurator.Open** and **ApplyConfiguration** methods in conjunction, as shown in the below code snippet.

[C#]

```
// Plug-in an external configuration file.  
this.editControl1.Configurator.Open(configFile);  
  
// Apply the configuration defined in the configuration file.  
this.editControl1.ApplyConfiguration("CustomLanguage");
```

[VB.NET]

```
' Plug-in an external configuration file.  
Me.editControl1.Configurator.Open(configFile)  
  
' Apply the configuration defined in the configuration file.  
Me.editControl1.ApplyConfiguration("CustomLanguage")
```

Run Time Configuration Settings

Syntax Highlighting and Code Coloring can be implemented at run time by using the **Language Coloring Configuration Editor**.

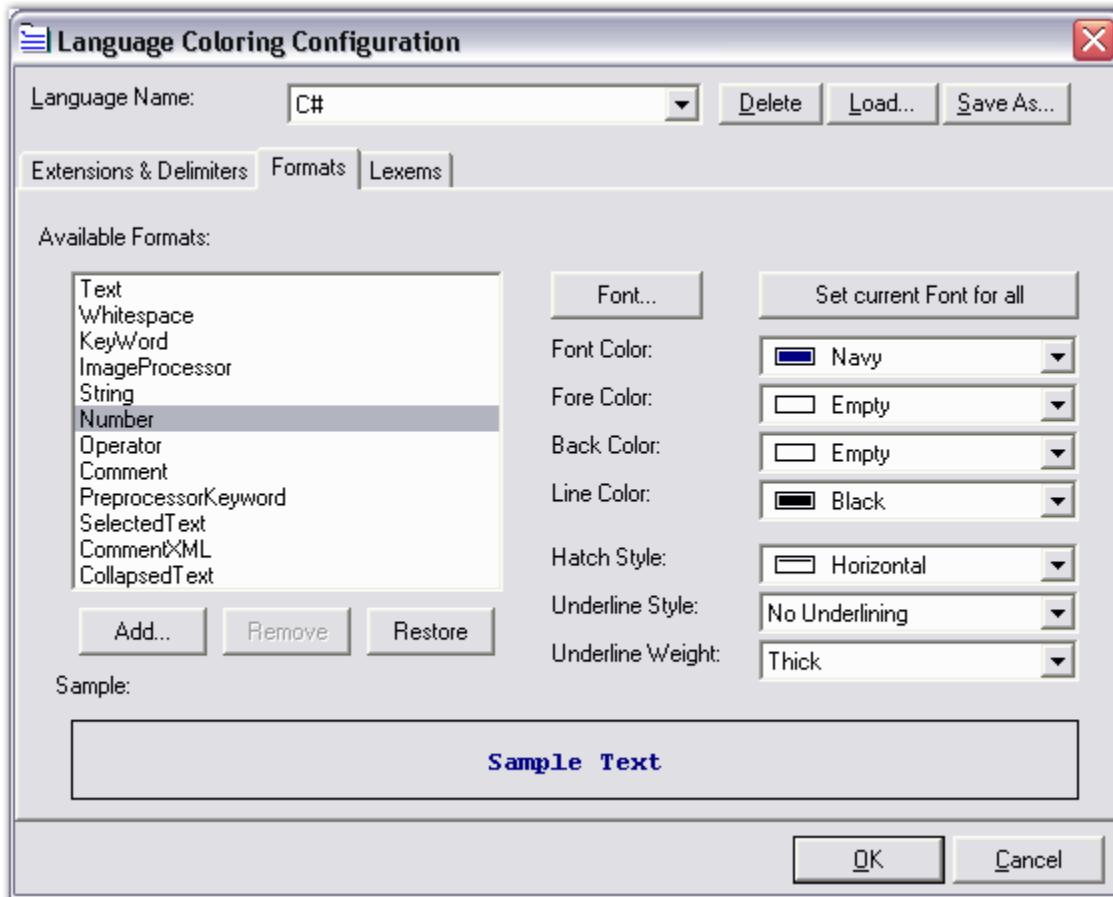


Figure 43: Configuration Customization Dialog Box

The Language Coloring Configuration Editor can be invoked programmatically as follows.

```
[C#]

IConfigLanguage activeLang = this.editControl1.Parser.Formats as
IConfigLanguage;

// Create an instance of ConfigurationDialog.
ConfigurationDialog editConfig = new
ConfigurationDialog(this.editControl1.Configurator, activeLang);
if(editConfig.ShowDialog(this) == DialogResult.OK && activeLang != null)
{
    IConfigLanguage newLang =
editConfig.Configurator.KnownLanguageNames.Contains(activeLang.Language) ?
editConfig.Configurator[activeLang.Language] :
editConfig.Configurator.DefaultLanguage;
    if(newLang != null)
    {
```

```
// Set language configuration instance object.  
this.editControl1.Configurator = editConfig.Configurator;  
  
// Applies coloring of the specified language to the text.  
this.editControl1.ApplyConfiguration(newLang);  
}  
}
```

[VB.NET]

```
Dim activeLang As IConfigLanguage = Me.EditControl1.Parser.Formats  
  
' Create an instance of ConfigurationDialog.  
Dim editConfig As New frmConfigDialog(Me.editControl1.Configurator,  
activeLang)  
  
If editConfig.ShowDialog(Me) = DialogResult.OK AndAlso Not (activeLang Is  
Nothing) Then  
    Dim newLang As IConfigLanguage =  
    If(editConfig.Configurator.KnownLanguageNames.Contains(activeLang.Language),  
editConfig.Configurator(activeLang.Language),  
editConfig.Configurator.DefaultLanguage)  
        If Not (newLang Is Nothing) Then  
            ' Set language configuration instance object.  
            Me.editControl1.Configurator = editConfig.Configurator  
  
            ' Applies coloring of the specified language to the text.  
            Me.editControl1.ApplyConfiguration(newLang)  
        End If  
    End If  
End If
```

A sample which demonstrates the above features is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Syntax Highlighting\\CustomConfigFileDemo**

See Also

[Creating a Custom Language Configuration File](#),

4.5.1 XML Based Configuration Files

Syntax Highlighting is accomplished in Essential Edit through the use of XML-based configuration files.

The language-specific configuration is stored in XML files. The below given code snippet illustrates a sample configuration file that can be used for syntax highlighting a LISP-like code.

```
[XML]

<ConfigLanguage name="Custom LISP">
    <formats>
        <format name="Text" Font="Courier New, 10pt"
FontColor="Black" />
        <format name="Whitespace" Font="Courier New, 10pt"
FontColor="Black" />
        <format name="KeyWord" Font="Courier New, 10pt"
FontColor="Blue" />
        <format name="String" Font="Courier New, 10pt, style=Bold"
FontColor="Red" />
        <format name="Number" Font="Courier New, 10pt, style=Bold"
FontColor="Navy" />
        <format name="Operator" Font="Courier New, 10pt"
FontColor="DarkCyan" />
        <format name="Comment" Font="Courier New, 10pt, style=Bold"
FontColor="Green" />
        <format name="SelectedText" Font="Courier New, 10pt"
BackColor="Highlight" FontColor="HighlightText" />
        <format name="CollapsedText" Font="Courier New, 10pt"
FontColor="Black" BackColor="White"/>
    </formats>
    <extensions>
        <extension>lsp</extension>
    </extensions>
    <lexems>
        <lexem BeginBlock="(" Type="Operator" />
        <lexem BeginBlock=")" Type="Operator" />
        <lexem BeginBlock="!" Type="Operator" />
        <lexem BeginBlock="car" Type="KeyWord" />
        <lexem BeginBlock="cdr" Type="KeyWord" />
        <lexem BeginBlock="cons" Type="KeyWord" />
        <lexem BeginBlock="#Region" EndBlock="#End Region"
Type="PreprocessorKeyword" IsEndRegex="true"
IsComplex="true" IsCollapsable="true"
AutoNameExpression="\s*(?<text>.*).*\n" AutoNameTemplate="${text} "
IsCollapseAutoNamed="true" CollapseName="#Region">
            <SubLexems>
                <lexem BeginBlock="\n" IsBeginRegex="true" />
            </SubLexems>
    
```

```
</lexem>
</lexems>
<splits>
    <split>#Region</split>
    <split>#End Region</split>
</splits>
</ConfigLanguage>
```

For additional information, Refer to the [Creating a Custom Language Configuration File](#) subsection under the [Configuration Settings](#) section.

See Also

[Syntax Highlighting and Code Coloring](#), [Multiple Language Syntax Highlighting](#)

4.5.2 Multiple Language Syntax Highlighting

Edit Control supports syntax highlighting in scenarios where more than one language is involved. For example, HTML files with embedded JScript.



The screenshot shows a code editor window with syntax highlighting applied to an HTML file. The file contains both standard HTML tags and embedded JScript. The code is numbered from 19 to 38. The syntax highlighting uses different colors for various elements: blue for keywords like ``, `var`, `if`, and `else`; red for attributes like `class="page"` and `language="jscript"`; green for strings like `"#ca"`, `"/"`, and `www.microsoft.com`; and purple for comments like `/*` and `*/`. The editor has scroll bars and a status bar at the bottom.

```
19  <a href="#ca" class="hide">
20  </a>
21  <div id="dPage" class="page">
22  <script language="jscript">
23  var hN = window.location.hostname.toLowerCase();
24  if(window.location.pathname == "/" && (hN == "www.microsoft.com"))
25  {
26      var rs='';
27      var r=window.document.referrer;
28      var TG='<layer visibility="hide"><div style="display:none;">';
29
30      if( "" != r )
31      {
32          TG += '&#amp;r'+escape(r)
33      }
34
35      TG+=" height="0" width="0" hspace="0" vspace="0" border="0">
36      TG+.'
37      document.writeln(TG);
38  }
```

Figure 44: Syntax Highlighting illustrated in HTML with Embedded JScript

A sample which demonstrates the above feature is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Syntax Highlighting\\SyntaxColoringDemo**

See Also

[Syntax Highlighting and Code Coloring, XML Based Configuration Files](#)

4.6 Runtime Features

The runtime features of the Edit control are as follows:

4.6.1 Insert Mode

The mode of the INSERT key in the Edit Control can be controlled programmatically by using the **InsertMode** property. Toggling the value of this property is equivalent to pressing the INSERT key on the keyboard. When InsertMode is set to **True**, the characters typed get inserted into the Edit Control, without overwriting the existing text. When set to **False**, the characters typed overwrite the existing text of the Edit Control. By default, this property is set to **True**.

The mode of the INSERT key can also be toggled by using the **ToggleInsertMode** method of the Edit Control.

[C#]

```
// Enable the insert key mode in Edit Control.  
this.editControl1.InsertMode = true;  
  
// Toggle the insert mode.  
this.editControl1.ToggleInsertMode();
```

[VB .NET]

```
' Enable the insert key mode in Edit Control.  
Me.editControl1.InsertMode = True  
  
' Toggle the insert mode.  
Me.editControl1.ToggleInsertMode()
```

4.6.2 Keyboard Shortcuts

The keyboard shortcuts for the commands in the Edit Control are listed below.

Command	Shortcut
Clipboard	
Copy	CTRL+C, CTRL+INSERT
Paste	CTRL+V, SHIFT+INSERT
Cut	CTRL+X, SHIFT+DEL
SelectAll	CTRL+A
File Operation	

Save	CTRL+S
SaveAs	CTRL+SHIFT+S
New	CTRL+N
Open	CTRL+O
Printing	
Print	CTRL+P
Positioning	
Go to line	CTRL+G
Go to start	CTRL+HOME
Go to end	CTRL+END
Search and Replace	
Find	CTRL+F
FindNext	F3
FindSelected	CTRL+F3
Replace	CTRL+H
Undo and Redo	
Undo	CTRL+Z
Redo	CTRL+Y
Bookmark	
Toggle unnamed bookmark	CTRL+F2, CTRL+K->CTRL+K
Go to next bookmark	F2, CTRL+K->CTRL+N
Go to previous bookmark	F3, CTRL+K->CTRL+P
Toggle named bookmark	CTRL+[index of bookmark]
Go to named bookmark	CTRL+SHIFT+[index of bookmark]

Tab	
Add leading tab	TAB with multiple line selection
Remove leading tab	SHIFT+TAB
Outlining	
Switch on outlining and collapse all	CTRL+M->CTRL+O
Switch off outlining	CTRL+M->CTRL+P
Toggle outlining	CTRL+M->CTRL+M
WhiteSpace	
Show white space	CTRL+SHIFT+W
Intellisense	
Show context prompt	CTRL+SHIFT+SPACEBAR
Show context choice	CTRL+SPACEBAR

4.6.3 Bitmap Generation

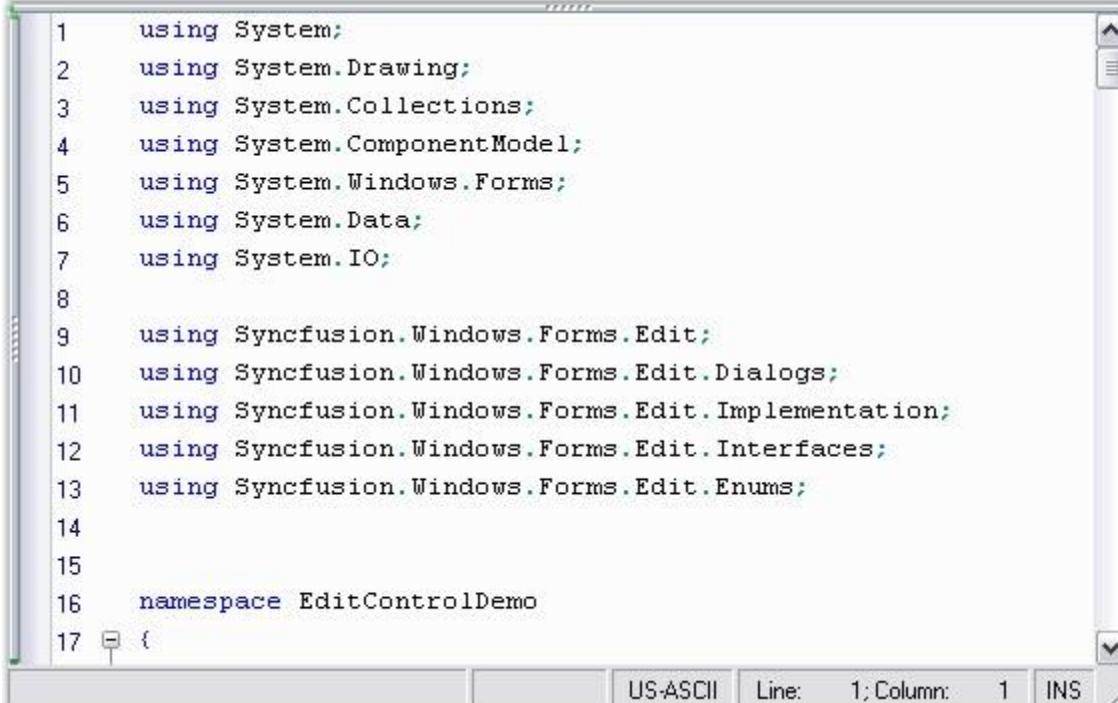
The Edit Control has the ability to generate a bitmap image of itself. The bitmap image looks exactly like an actual snapshot of a live instance of Edit Control. This is achieved through the use of the **CreateBitmap** method.

[C#]

```
// Creates bitmap of the Edit Control.
Bitmap bmp = this.editControl1.CreateBitmap();
```

[VB.NET]

```
' Creates bitmap of the Edit Control.
Dim bmp as Bitmap = Me.editControl1.CreateBitmap()
```



```

1  using System;
2  using System.Drawing;
3  using System.Collections;
4  using System.ComponentModel;
5  using System.Windows.Forms;
6  using System.Data;
7  using System.IO;
8
9  using Syncfusion.Windows.Forms.Edit;
10 using Syncfusion.Windows.Forms.Edit.Dialogs;
11 using Syncfusion.Windows.Forms.Edit.Implementation;
12 using Syncfusion.Windows.Forms.Edit.Interfaces;
13 using Syncfusion.Windows.Forms.Edit.Enums;
14
15
16 namespace EditControlDemo
17 {

```

The screenshot shows a code editor window with the following C# code:

```

1  using System;
2  using System.Drawing;
3  using System.Collections;
4  using System.ComponentModel;
5  using System.Windows.Forms;
6  using System.Data;
7  using System.IO;
8
9  using Syncfusion.Windows.Forms.Edit;
10 using Syncfusion.Windows.Forms.Edit.Dialogs;
11 using Syncfusion.Windows.Forms.Edit.Implementation;
12 using Syncfusion.Windows.Forms.Edit.Interfaces;
13 using Syncfusion.Windows.Forms.Edit.Enums;
14
15
16 namespace EditControlDemo
17 {

```

The status bar at the bottom of the editor shows "US-ASCII", "Line: 1; Column: 1", and "INS".

Figure 45: Bitmap of a Live Instance of Edit Control

4.6.4 Find, Replace and Goto

The Edit Control supports text search and replace functionalities through the use of the **FindText** and **ReplaceText** methods. There are also other useful methods like **FindCurrentText**, **FindNext** and **ReplaceAll** that assist in this purpose.

Edit Control Method	Description
FindText	Finds the first occurrence of the specified text as per the conditions specified like match case, match whole word, search hidden text and search up.
FindRange	Searches for given string in the text of control and returns text range of first found occurrence.
FindRegex	Looks for specified expression in text.
ReplaceText	Replaces the first occurrence of the specified text with the replacement text as per the conditions specified like match case, match whole word, search hidden text and search up.

FindCurrentText	Finds the next occurrence of the word on which the cursor is presently on.
FindNext	Finds the next occurrence of the current search text.
ReplaceAll	Replaces all occurrences of the search text with the replacement text as per the conditions specified like match case, match whole word, search hidden text and search up.

[C#]

```
// Finds the first occurrence of the specified text as per the conditions
// specified.
this.editControl1.FindText("Essential Edit", true, true, true, true, null);

// Searches for given string in the text of control and returns text range of
// first found occurrence.
this.editControl1.FindRange(searchString, startLocation, endLocation,
matchWholeWord, searchHiddenText, searchUp, useRegex);

// Looks for specified expression in text.
this.editControl1.FindRegex(startLine, startColumn, expression,
bSearchInCollapsed, searchUp);

// Replaces the first occurrence of the specified text with the replacement
// text as per the conditions specified.
this.editControl1.ReplaceText("ShowVerticalScrollbar",
"ShowVerticalScroller");

// Finds the next occurrence of the word on which the cursor is presently on.
this.editControl1.FindCurrentText();

// Finds the next occurrence of the current search text.
this.editControl1.FindNext();

// Replaces all occurrences of the search text with the replacement text as
// per the conditions specified.
this.editControl1.ReplaceAll(" Drag-and-drop", "Drag and drop");
```

[VB.NET]

```
' Finds the first occurrence of the specified text as per the conditions
// specified.
Me.editControl1.FindText("Essential Edit", True, True, True, True, Nothing)

' Searches for given string in the text of control and returns text range of
```

```
first found occurrence.  
Me.editControl1.FindRange(searchString, startLocation, endLocation,  
matchWholeWord, searchHiddenText, searchUp, useRegex)  
  
' Looks for specified expression in text.  
Me.editControl1.FindRegex(startLine, startColumn, expression,  
bSearchInCollapsed, searchUp)  
  
' Replaces the first occurrence of the specified text with the replacement  
text as per the conditions specified.  
Me.editControl1.ReplaceText("ShowVerticalScrollbar", "ShowVerticalScroller")  
  
' Finds the next occurrence of the word on which the cursor is presently on.  
Me.editControl1.FindCurrentText()  
  
' Finds the next occurrence of the current search text.  
Me.editControl1.FindNext()  
  
' Replaces all occurrences of the search text with the replacement text as  
per the conditions specified.  
Me.editControl1.ReplaceAll(" Drag-and-drop", "Drag and drop")
```

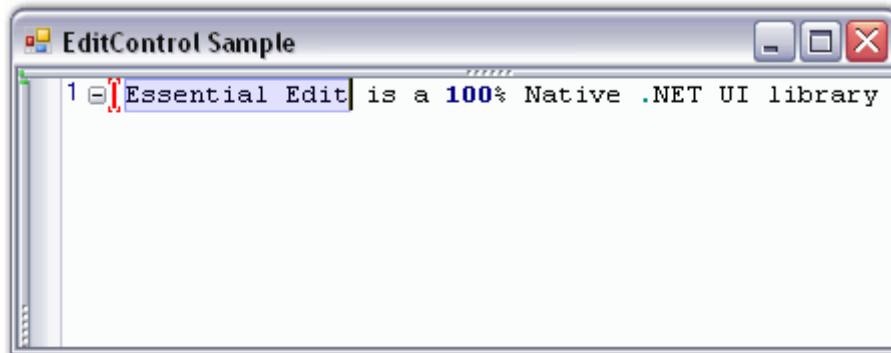


Figure 46: "FindText" method

Find and Replace Dialog Boxes

Edit Control also supports advanced and customizable Find and Replace dialog boxes. The Find dialog box is invoked by using the **ShowFindDialog** method. The keyboard shortcut to this dialog box is **Ctrl+F**.



Figure 47: Find Dialog Box

The Replace dialog box is invoked by using the **ShowReplaceDialog** method. The keyboard shortcut to this dialog box is **Ctrl+H**. The Replace dialog box also allows you to find and replace words within the selected text.

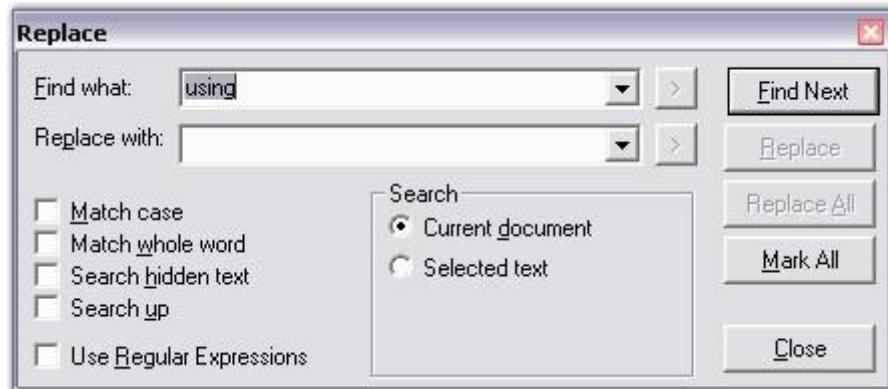


Figure 48: Replace Dialog Box

```
[C#]

// Invoke the Find Dialog.
this.editControl1.ShowFindDialog();

// Invoke the Replace Dialog.
this.editControl1.ShowReplaceDialog();
```

```
[VB .NET]

' Invoke the Find Dialog.
Me.editControl1.ShowFindDialog()

' Invoke the Replace Dialog.
Me.editControl1.ShowReplaceDialog()
```

Positioning Mouse Cursor on a Specified line

The Edit Control supports the "GoTo" functionality both through the use of a run time dialog box and through programmatic APIs. The **GoTo** method is used to position the mouse pointer on any specified line. The GoTo method not only positions the pointer on the appropriate line, but it also scrolls the concerned line into the view. The **linesAbove** argument can be used to specify the number of lines to be displayed above the pointer.

[C#]

```
// Places the cursor at the beginning of the given line number.
this.editControl1.GoTo(lineNumber);
this.editControl1.GoTo(lineNumber, linesAbove);
```

[VB .NET]

```
' Places the cursor at the beginning of the given line number.
Me.editControl1.GoTo(lineNumber)
Me.editControl1.GoTo(lineNumber, linesAbove);
```

The **CurrentLine** property explained in the [Positions and Offsets](#) section, also does the same task as the GoTo method. The Goto dialog box is invoked using the **ShowGoToDialog** method. The keyboard shortcut to this dialog box is **Ctrl+G**.

[C#]

```
// Invoke the GoTo Dialog.
this.editControl1.ShowGoToDialog();
```

[VB .NET]

```
' Invoke the GoTo Dialog.
Me.editControl1.ShowGoToDialog()
```



Figure 49: GoTo Dialog Box

Default key bindings to these dialogs can be changed as explained in the [Keystroke - Action Combinations Binding](#) topic.

History Properties

The **FindHistory** property is used to add/remove items from the find history in the Find dialog box. The **ReplaceHistory** property is used to add/remove items from the replace history in the Replace dialog box. Similarly, the **ReplaceSearchHistory** property is used to add / remove items from the find history in the Replace dialog box.

Edit Control Property	Description
FindHistory	Gets history of Find dialog.
ReplaceHistory	Gets history of Replace dialog.
ReplaceSearchHistory	Gets search history of Replace dialog.

The methods associated with the **FindHistory** property are used to perform the following operations.

FindHistory Method	Description
Insert	Inserts an element into the System.Collections.ArrayList at the specified index.
Remove	Removes an element or the first occurrence from the System.Collections.ArrayList of the specified index.
Sort	Sorts all the elements in the System.Collections.ArrayList.
Clear	Clears all the items in the FindHistory.

[C#]

```
this.editControl1.FindHistory.Insert(0, (object)ATH.addedItem);
this.editControl1.FindHistory.Remove(0);
this.editControl1.FindHistory.Sort();
this.editControl1.FindHistory.Clear();
```

[VB .NET]

```
Me.editControl1.FindHistory.Insert(0, CType(ATH.addItem, Object))
Me.editControl1.FindHistory.Remove(0)
Me.editControl1.FindHistory.Sort()
Me.editControl1.FindHistory.Clear()
```



Note: The above methods can also be set for the ReplaceHistory and ReplaceSearchHistory properties.

A sample which demonstrates the above features is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\FindReplaceDemo**

4.6.5 Enhanced Find Dialog

Essential Edit control **Find Dialog** is now enhanced with an alert message box. This displays the alert message box when find reaches the starting point of the search again.



Note: In search option Current Selection, click OK in alert message box, then the search area is selected again automatically as in VS editor.

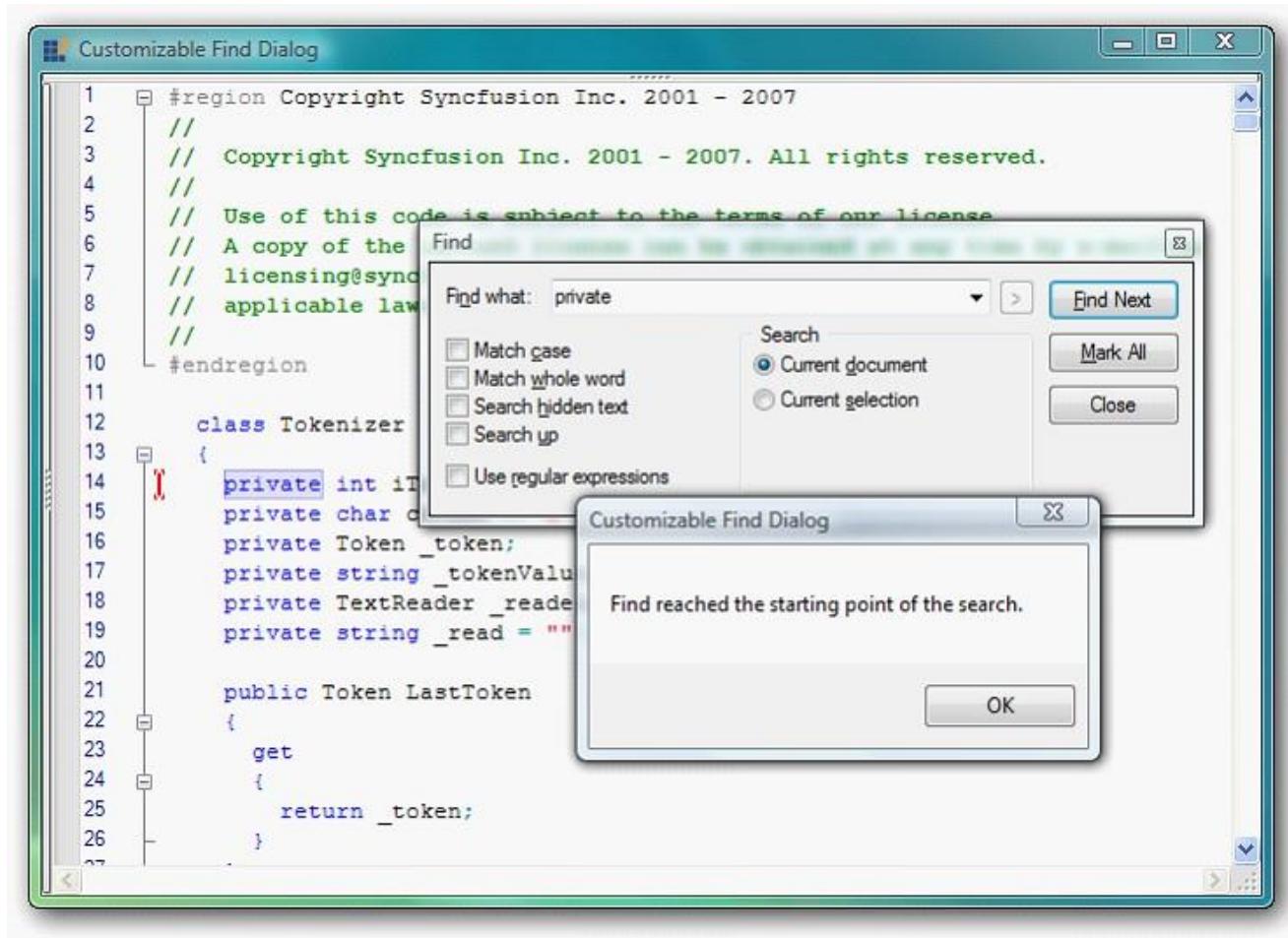


Figure 50: Alert Message Box

4.6.6 Scrolling Support

Edit Control offers extremely smooth scrolling behavior using idle-time processing and dynamic scroll area expansion techniques. The scrolling behavior is smooth even when large files are loaded, though the Edit Control scrolls by several hundred lines for a small movement of the scroller.

The scrollers in the Edit Control can be optionally shown / hidden by using the below given properties.

Edit Control Property	Description
ShowVerticalScroller	Gets / sets value indicating whether the vertical scroller can be shown.

ShowHorizontalScroller	Gets / sets value indicating whether the horizontal scroller can be shown.
AlwaysShowScrollers	Gets / sets value indicating whether scrollers should be always visible.

[C#]

```
// Display the Horizontal Scroller.  
this.editControl1.ShowHorizontalScroller = true;  
  
// Display the Vertical Scroller.  
this.editControl1.ShowVerticalScroller = true;  
  
this.editControl1.AlwaysShowScrollers = true;
```

[VB .NET]

```
// Display the Horizontal Scroller.  
Me.editControl1.ShowHorizontalScroller = True  
  
// Display the Vertical Scroller.  
Me.editControl1.ShowVerticalScroller = True  
  
Me.editControl1.AlwaysShowScrollers = True
```

The Edit Control supports scroller events that are raised when the scroll arrows are clicked. The scroller events are used to synchronize the scrolling of multiple Edit Controls.

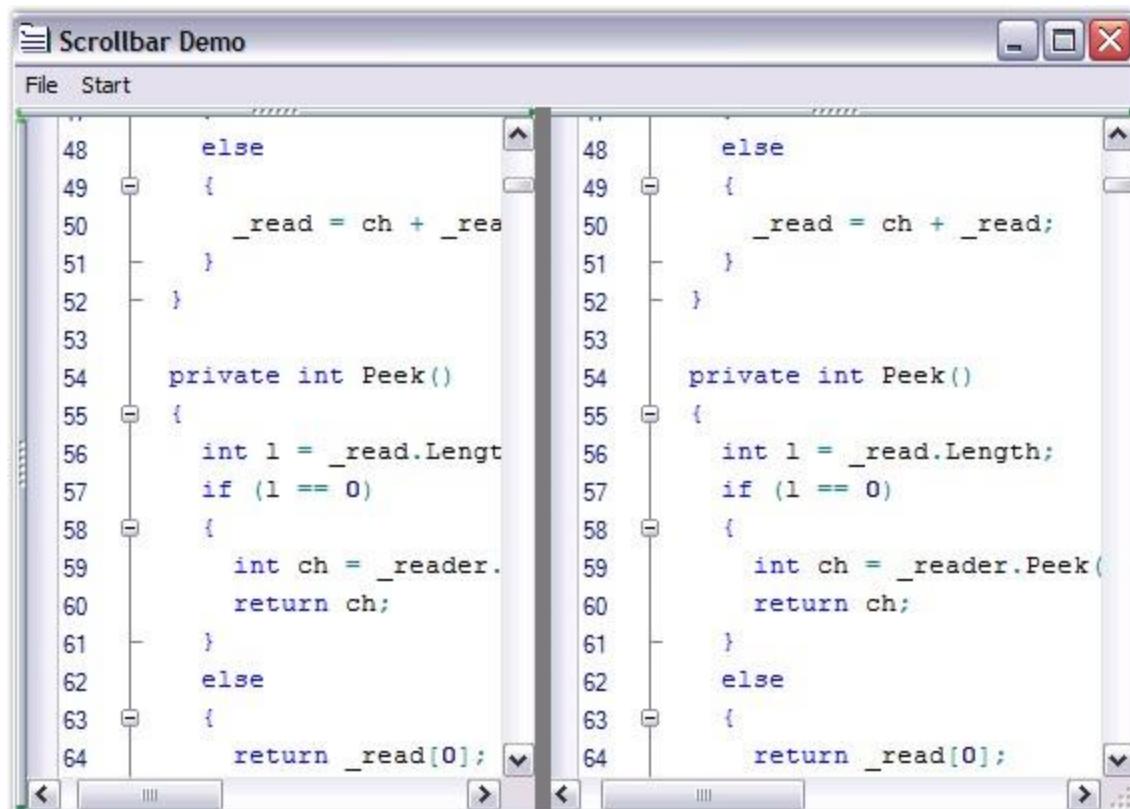


Figure 51: Scrolling support in Edit Control

Scroll Bar Buttons

Buttons can be displayed at the top, bottom, left or right of the scroll bars by using the below given properties.

Edit Control Property	Description
ScrollbarBottomButtons	Gets buttons at the bottom of vertical scrollbar.
ScrollbarLeftButtons	Gets buttons on the left of vertical scrollbar.
ScrollbarRightButtons	Gets buttons on the right of vertical scrollbar.
ScrollbarTopButtons	Gets buttons at the top of vertical scrollbar.

[C#]

```
this.editControl1.ScrollbarBottomButtons.AddRange(new
System.Windows.Forms.Control[] { this.scrollBarButton1 });
```

```
this.editControl1.ScrollbarLeftButtons.AddRange(new  
System.Windows.Forms.Control[] { this.scrollBarButton2 });  
this.editControl1.ScrollbarRightButtons.AddRange(new  
System.Windows.Forms.Control[] { this.scrollBarButton3 });  
this.editControl1.ScrollbarTopButtons.AddRange(new  
System.Windows.Forms.Control[] { this.scrollBarButton4 });
```

[VB.NET]

```
Me.editControl1.ScrollbarBottomButtons.AddRange(New  
System.Windows.Forms.Control() { Me.scrollBarButton1 })  
Me.editControl1.ScrollbarLeftButtons.AddRange(New  
System.Windows.Forms.Control() { Me.scrollBarButton2 })  
Me.editControl1.ScrollbarRightButtons.AddRange(New  
System.Windows.Forms.Control() { Me.scrollBarButton3 })  
Me.editControl1.ScrollbarTopButtons.AddRange(New  
System.Windows.Forms.Control() { Me.scrollBarButton4 })
```

Scroll Position and Offsets

The scroll position and offsets of the Edit Control are set by using the below given properties.

Edit Control Property	Description
ScrollPosition	Gets / sets scroll position of Edit Control.
ScrollOffsetBottom	Gets / sets the bottom scroll offset.
ScrollOffsetLeft	Gets / sets the left scroll offset.
ScrollOffsetRight	Gets / sets the right scroll offset.
ScrollOffsetTop	Gets / sets the top scroll offset.

[C#]

```
this.editControl1.ScrollPosition = new Point(1, 5);  
  
this.editControl1.ScrollOffsetBottom = 5;  
this.editControl1.ScrollOffsetLeft = 10;  
this.editControl1.ScrollOffsetTop = 5;  
this.editControl1.ScrollOffsetTop = 10;
```

[VB.NET]

```
Me.editControl1.ScrollPosition = New Point(1, 5)  
  
Me.editControl1.ScrollOffsetBottom = 5  
Me.editControl1.ScrollOffsetLeft = 10  
Me.editControl1.ScrollOffsetTop = 5  
Me.editControl1.ScrollOffsetTop = 10
```

4.6.6.1 Office 2007 Visual Style

Edit Control enables to provide Office 2007 appearance to scroll bars by setting the **ScrollVisualStyle** property to **Office2007**.

It supports all the three Office 2007 Color Schemes (Black, Blue and Silver), which can be set by using the **ScrollColorScheme** property. Also, custom colors can be applied to the scroll bars of the Edit Control. This can be done by setting the ScrollColorScheme property to **Managed**.

Edit Control Property	Description
ScrollVisualStyle	Specifies the visual style of the scroll bar.
ScrollColorScheme	Specifies the scroll bar color scheme when Office2007 or Office2007Generic Style is set. The options provided are <ul style="list-style-type: none">• Black• Blue• Silver• Managed

[C#]

```
this.editControl1.ScrollVisualStyle = ScrollBarCustomDrawStyles.Office2007;  
this.editControl1.ScrollColorScheme = Office2007ColorScheme.Blue;  
  
// Set custom color for the scroll bar.  
this.editControl1.ScrollColorScheme = Office2007ColorScheme.Managed;  
Syncfusion.Windows.Forms.Office2007Colors.ApplyManagedColors(this,  
Color.Green);
```

[VB .NET]

```

Me.editControl1.ScrollVisualStyle = ScrollBarCustomDrawStyles.Office2007
Me.editControl1.ScrollColorScheme = Office2007ColorScheme.Blue

' Set custom color for the scroll bar.
Me.editControl1.ScrollColorScheme = Office2007ColorScheme.Managed
Syncfusion.Windows.Forms.Offic2007Colors.ApplyManagedColors(Me, Color.Green)

```

The following illustration shows the Edit Control with custom color (green) set for the scroll bars.

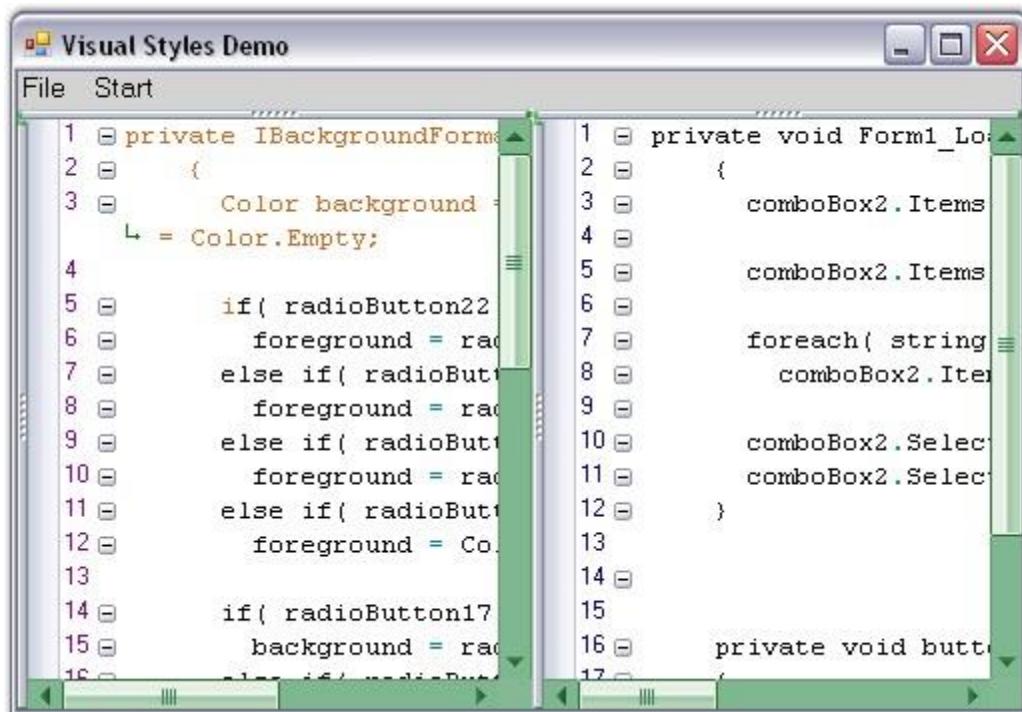


Figure 52: Edit Control with ScrollColorScheme property = "Managed"

4.6.6.1.1 ToolTip

Essential Edit supports ToolTip feature for various functionalities which are discussed in this section.

- ToolTip for lexems. This is discussed in the [Context Tooltip](#) topic.
- Three different tooltips for outlining is discussed in the [Outlining Tooltip](#) topic.

4.6.6.2 Interactive Features

The interactive features are discussed in the following topics:

4.6.6.2.1 Customizable Context Menu

Edit Control has a built-in context menu which is enabled, by default. This context menu allows you to edit the contents, and open or create a new file. It includes some advanced features like indent selection, comment selection, adding bookmarks, and much more. This is enabled by using the **EditControl1.ContextMenuManager.Enabled** property.

The context menu has the standard VS.NET-like appearance, and can optionally be provided with the Office 2003 appearance.

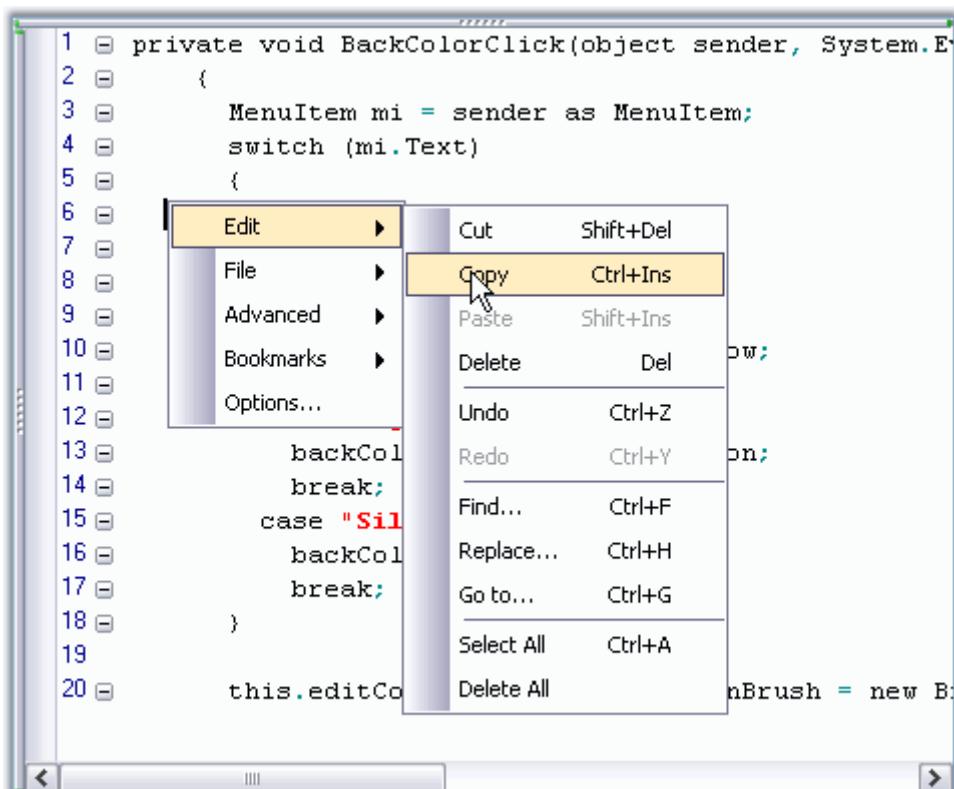


Figure 53: Edit Control's Context Menu in Office2003 Style

Set the appearance of the context menu by specifying the desired **ContextMenuProvider**.

[C#]

```
// Show Office2003 style context menu.  
this.editControl1.ContextMenuManager.ContextMenuProvider = new  
Syncfusion.Windows.Forms.Tools.XPMenus.XPMenusProvider();  
  
// Show Standard style context menu.  
this.editControl1.ContextMenuManager.ContextMenuProvider = new  
Syncfusion.Windows.Forms.StandardMenusProvider();
```

[VB.NET]

```
' Show Office2003 style context menu  
Me.editControl1.ContextMenuManager.ContextMenuProvider = New  
Syncfusion.Windows.Forms.Tools.XPMenus.XPMenusProvider()  
  
' Show Standard style context menu  
Me.editControl1.ContextMenuManager.ContextMenuProvider = New  
Syncfusion.Windows.Forms.StandardMenusProvider()
```

Adding Customized Menu Items

You can handle the **MenuFill** event to add Menu Items to the context menu. This is illustrated in the below code snippet.

[C#]

```
// Handle the MenuFill event which is called each time the context menu is  
displayed.  
this.editControl1.MenuFill += new EventHandler(cm_FillMenu);  
  
private void cm_FillMenu(object sender, EventArgs e)  
{  
    ContextMenuManager cm = (ContextMenuManager) sender;  
  
    // To clear default context menu items.  
    cm.ClearMenu();  
  
    // Add a separator.  
    cm.AddSeparator();  
  
    // Add custom context menu items and their Click eventhandlers.  
    cm.AddMenuItem("&Find", new EventHandler>ShowFindDialog());  
    cm.AddMenuItem("&Replace", new EventHandler>ShowReplaceDialog());  
    cm.AddMenuItem("&Goto", new EventHandler>ShowGoToDialog());
```

```
// If you need to get access to the underlying menu provider you can
access it using the below given code.
Syncfusion.Windows.Forms.IContextMenuProvider contextMenuProvider =
this.editControl1.ContextManager.ContextMenuProvider;
}

// Calling the in-built dialogs.

void ShowFindDialog(object sender, EventArgs e)
{
    this.editControl1.ShowFindDialog();
}

void ShowReplaceDialog(object sender, EventArgs e)
{
    this.editControl1.ShowReplaceDialog();
}

void ShowGoToDialog(object sender, EventArgs e)
{
    this.editControl1.ShowGoToDialog();
}
```

[VB.NET]

```
' Handle the MenuFill event which is called each time the context menu is
displayed.
AddHandler Me.editControl1.MenuFill, AddressOf cm_FillMenu

Private Sub cm_FillMenu(ByVal sender As Object, ByVal e As EventArgs)
    Dim cm As ContextManager = CType(sender, ContextManager)

    ' To clear default context menu items.
    cm.ClearMenu();

    ' Add a separator.
    cm.AddSeparator()

    ' Add custom context menu items and their Click eventhandlers.
    cm.AddMenuItem("&Find", New EventHandler(AddressOf ShowFindDialog))
    cm.AddMenuItem("&Replace", New EventHandler(AddressOf ShowReplaceDialog))
    cm.AddMenuItem("&Goto", New EventHandler(AddressOf ShowGoToDialog))

    ' If you need to get access to the underlying menu provider you can access
    it using the below given code.
    Dim contextMenuProvider As Syncfusion.Windows.Forms.IContextMenuProvider =
```

```
Me.editControl1.ContextMenuManager.ContextMenuProvider
End Sub 'cm_FillMenu

' Calling the in-built dialogs.
Sub ShowFindDialog(ByVal sender As Object, ByVal e As EventArgs)
    Me.editControl1.FindDialog()
End Sub

Sub ShowReplaceDialog(ByVal sender As Object, ByVal e As EventArgs)
    Me.editControl1.ReplaceDialog()
End Sub

Sub ShowGoToDialog(ByVal sender As Object, ByVal e As EventArgs)
    Me.editControl1.GoToDialog()
End Sub
```

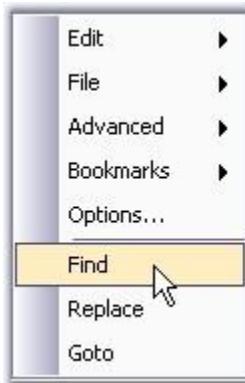


Figure 54: Customized Find, Replace and Goto Menu Items in Context Menu

Assembly Dependency

If the Syncfusion.Tools.Windows assembly is loaded before the instantiation of the context menu, then an XPMenus.PopupMenu is displayed as the context menu. Otherwise, a standard .NET context menu is shown.



Note: You must have reference to the **Syncfusion.Tools.Windows** assembly in your project.

A sample demonstrating the Context Menu feature is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\ContextMenuDemo**

4.6.6.2.2 IntelliPrompt Features

This section covers the following topics:

4.6.6.2.2.1 Code Snippets

Essential Edit supports an advanced feature of VS 2005 like Code Snippets. It is also used to load / save VS.NET 2005-compatible XML snippets.

Code Snippets are inserted into the Edit Control by following the procedure given below:

1. Type the snippet name. For example "do".
2. Pressing the TAB key, or CTRL + ' combination.
3. Select an item from the list as shown in the image below.

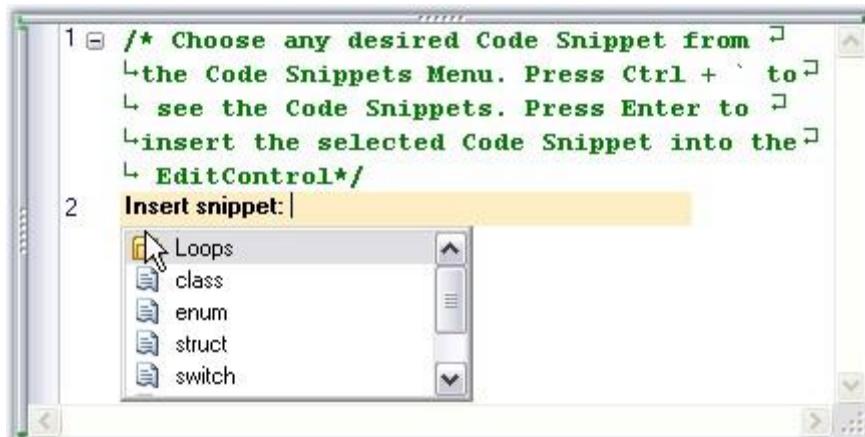


Figure 55: Inserting Code Snippets into the Edit Control

The code snippets allow you to input data to the highlighted fields.

Code Snippets can also be inserted into the Edit Control by using the static **Extract** method of the **CodeSnippetsExtractor** class. The Extract method takes the following two parameters:

1. Path of the folder containing the code snippets.
2. Instance of the Edit Control into which the extracted code snippet should be inserted.

This is illustrated in the code given below.

[C#]

```
CodeSnippetsExtractor.Extract(csharpsnippetsPath, editControl1);
```

[VB .NET]

```
CodeSnippetsExtractor.Extract(vbsnippetsPath, editControl1)
```

Code Snippets are added to the current language of the Edit Control by using the below given method.

Edit Control Method	Description
AddCodeSnippet	Adds new code snippet to current language.

[C#]

```
this.editControl1.AddCodeSnippet(string title, ArrayList literals, string code);
```

[VB .NET]

```
Me.editControl1.AddCodeSnippet(String title, ArrayList literals, String code)
```

The code snippets can also be contained in containers and displayed in the pop-up of the snippets. The static Extract method of the CodeSnippetsExtractor class is used to extract and fill the container object. The container object can be added to the SnippetsContainer of the Edit Control by using the **AddContainer** method. This is illustrated in the code given below.

[C#]

```
private CodeSnippetsContainer container = new  
Syncfusion.Windows.Forms.Edit.Utils.CodeSnippets.CodeSnippetsContainer();  
container = CodeSnippetsExtractor.Extract(csharpsnippetsPath@"\Loops");  
container.Name = "Loops";  
this.editControl1.Language.SnippetsContainer.AddContainer(container);
```

[VB .NET]

```
container As CodeSnippetsContainer = New  
Syncfusion.Windows.Forms.Edit.Utils.CodeSnippets.CodeSnippetsContainer()
```

```
container = CodeSnippetsExtractor.Extract(vbsnippetsPath "\Loops")
container.Name = "Loops"
Me.editControl1.Language.SnippetsContainer.AddContainer(container)
```

Code snippets can also be created by using the configuration file. For example, the code snippet for a structure in C# can be created as shown below.

```
<CodeSnippetsContainer Name ="Container 2">
    <CodeSnippet Format ="1.0.0">
        <Header>
            <Title>struct</Title>
            <Shortcut>struct</Shortcut>
        <Description>Code snippet for struct</Description>
        </Header>
        <Snippet>
            <Declarations>
                <Literal>
                    <ID>name</ID>
                    <ToolTip>Struct name</ToolTip>
                    <Default>MyStruct</Default>
                </Literal>
            </Declarations>
            <Code Language ="csharp"><![CDATA[struct $name$&
{
    ...
}]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippetsContainer>
```

The Literal element is used to identify a replacement for a piece of code that is entirely contained within the snippet, but one that will likely be customized after it is inserted into the code. For example, literal strings, numeric values, and some variable names should be declared as literals. The symbol \$ is placed at the beginning and end of the literal ID element value. For example, if a literal has an ID element that contains the value MyID, you must reference that literal in the code element as \$MyID\$. All code snippets must be placed between <![CDATA[and]]> brackets.

Showing Code Snippets

You can programmatically show the choice list of code snippets by calling ShowCodeSnippets method given below.

```
[C#]
```

```
// Shows the code snippets' choice list.  
this.editControl1.ShowCodeSnippets();
```

[VB .NET]

```
' Shows the code snippets' choice list.  
Me.editControl1.ShowCodeSnippets()
```

Border Settings

Border can be set for the active code snippets by using the **DrawCodeSnippetBorder** property of the Edit Control.

[C#]

```
this.editControl1.DrawCodeSnippetBorder = true;
```

[VB .NET]

```
Me.editControl1.DrawCodeSnippetBorder = True
```

A sample demonstrating the above feature is available in the following sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Intellisense Functions\\ContextSnippetsDemo**

4.6.6.2.2.2 Context Choice

The Context Choice support allows you to create pop-ups for displaying a list of options that are used to complete what the user is typing. This feature is modeled on the **List Members** intellisense feature of Visual Studio, and is very convenient when editing programming languages. For example, in C# or VB.NET, when the . (period) character is typed after a class instance, a pop-up containing all the members of the class gets displayed. As you type in the editor, the list automatically changes selection to synchronize with the text that has been entered. You can also autocomplete the word by using the UP/DOWN ARROW keys to choose the Context Choice item and pressing the TAB key. The Context Choice pop-up can be dismissed by pressing the ESC key.

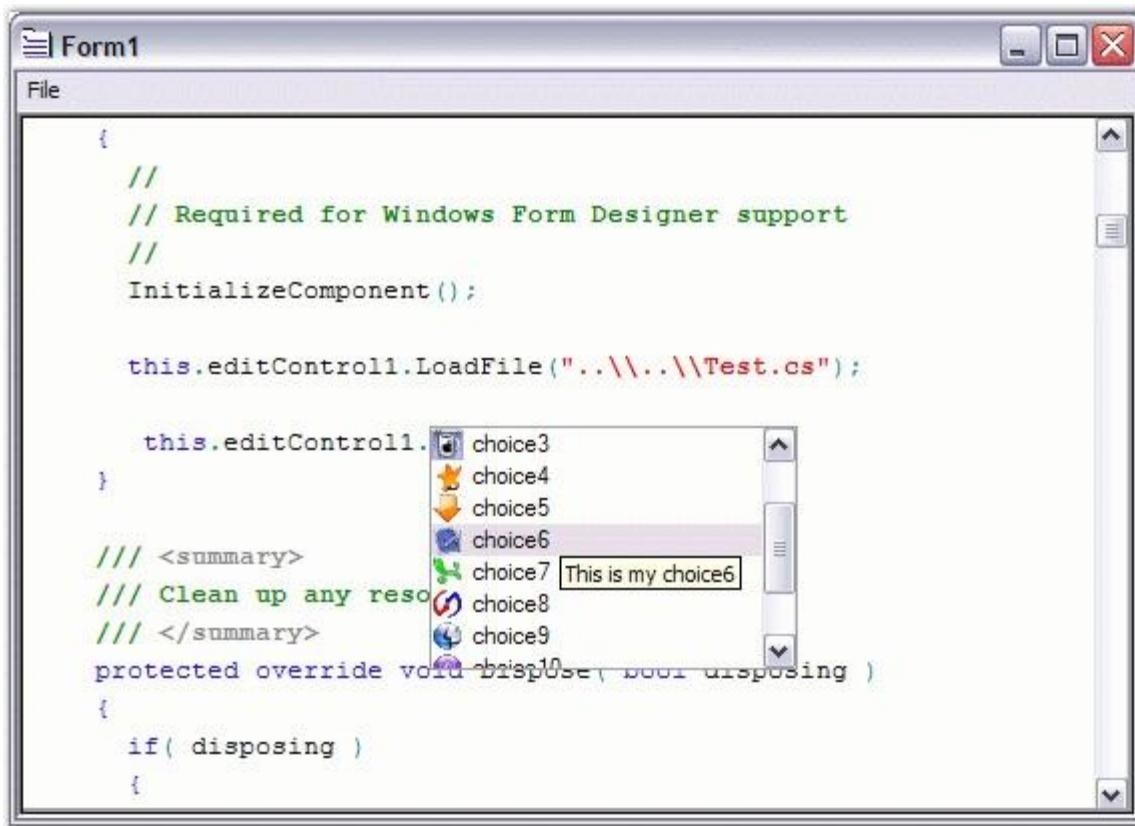


Figure 56: Context Choice List

The Context Choice displaying characters are specified in the configuration file by using the **DropContextChoiceList** field in the lexem for the corresponding character. If you wish to display the Context Choice dropdown in response to the period (".") or comma (",") being typed, use the following XML code.

[XML]

```

<lexem BeginBlock="." Type="Operator" DropContextChoiceList="true"/>
<lexem BeginBlock="," Type="Operator" DropContextChoiceList="true"/>
```

The preceding code has to be placed within the **<lexems>** section of the configuration file.

AutoCompleteSingleLexem

The **AutoCompleteSingleLexem** property indicates whether the Context Choice list gets autocompleted when a single lexem remains in the list.

[C#]

```
this.editControl1.AutoCompleteSingleLexem = true;
```

[VB .NET]

```
Me.editControl1.AutoCompleteSingleLexem = True
```

Populating the Context Choice List Items

The Context Choice list is populated by handling the **ContextChoiceOpen** event of the Edit Control, and adding items to the **Items** collection associated with the **IContextChoiceController** object.

Edit Control Event	Description
ContextChoiceOpen	This event occurs when the Context Choice window has been opened.

[C#]

```
private void
editControl1_ContextChoiceOpen(Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController controller)
{
    // Add items to the Items collection associated with
    // the IContextChoiceController object.
    controller.Items.Add("Method", "Method",
        this.editControl1.ContextChoiceController.Images["Image0"]);
    controller.Items.Add("FindText", "FindText",
        this.editControl1.ContextChoiceController.Images["Image1"]);
    controller.Items.Add("GetTextAsHTML", "GetTextAsHTML",
        this.editControl1.ContextChoiceController.Images["Image2"]);
    controller.Items.Add("LoadFile", "LoadFile",
        this.editControl1.ContextChoiceController.Images["Image3"]);
    controller.Items.Add("ToString", "ToString",
        this.editControl1.ContextChoiceController.Images["Image4"]);
    controller.Items.Add("Event", "Event",
        this.editControl1.ContextChoiceController.Images["Image5"]);
}
```

[VB .NET]

```
Private Sub editControl1_ContextChoiceOpen(ByVal controller As
Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController) Handles
EditControl1.ContextChoiceOpen
    ' Add items to the Items collection associated with
    ' the IContextChoiceController object.
    controller.Items.Add("Method", "Method",
```

```
Me.editControl1.ContextChoiceController.Images("Image0"))
controller.Items.Add("FindText", "FindText",
Me.editControl1.ContextChoiceController.Images("Image1"))
controller.Items.Add("GetTextAsHTML", "GetTextAsHTML",
Me.editControl1.ContextChoiceController.Images("Image2"))
controller.Items.Add("LoadFile", "LoadFile",
Me.editControl1.ContextChoiceController.Images("Image3"))
controller.Items.Add("ToString", "ToString",
Me.editControl1.ContextChoiceController.Images("Image4"))
controller.Items.Add("Event", "Event",
Me.editControl1.ContextChoiceController.Images("Image5"))
End Sub
```

Adding Custom Images to List Items

Custom images can also be added to the Context Choice list items by indexing them into the **Images** collection of the **IContextChoiceController** object associated with the Edit Control. The **Images** collection of the **IContextChoiceController** can be populated by using the code given below.

[C#]

```
int index = 0;
foreach (Image img in this.imageList1.Images)
{
    // Populating images using an external ImageList.
    this.editControl1.ContextChoiceController.AddImage("Image" +
index.ToString(), img);
    index++;
}
```

[VB .NET]

```
Dim index As Integer = 0
Dim img As Image
For Each img In Me.imageList1.Images
    ' Populating images using an external ImageList.
    Me.editControl1.ContextChoiceController.AddImage("Image" +
index.ToString(), img)
    index += 1
Next img
```

List Item ToolTip

ToolTip text is specified for each Context Choice list item while adding the items to the **IContextChoiceController**, as shown in the following code snippet.

[C#]

```
// Specify tooltip text for each Context Choice list item.  
controller.Items.Add("LoadFile", "Use this method to open a file in  
EditControl.", this.editControl1.ContextChoiceController.Images["Image3"]);
```

[VB .NET]

```
' Specify tooltip text for each Context Choice list item.  
controller.Items.Add("LoadFile", "Use this method to open a file in  
EditControl.", Me.editControl1.ContextChoiceController.Images["Image3"])
```

Customization

Border Settings

The border color of the Context Choice form is set by using the **ContextChoiceBorderColor** property.

Edit Control Property	Description
ContextChoiceBorderColor	Specifies the color of the Context Choice form border. Used when UseXPStyle property is set to 'False'. Otherwise 3D border is drawn.

[C#]

```
this.editControl1.ContextChoiceBorderColor = System.Drawing.Color.Red;
```

[VB .NET]

```
Me.editControl1.ContextChoiceBorderColor = System.Drawing.Color.Red
```

Size Settings

The size of the Context Choice form can be set by using the **ContextChoiceSize** property.

[C#]

```
this.editControl1.ContextChoiceSize = new System.Drawing.Size(100, 50);
```

[VB .NET]

```
Me.editControl1.ContextChoiceSize = New System.Drawing.Size(100, 50)
```

Context Choice Operations

The Edit Control provides the following set of events for performing Context Choice operations.

Edit Control Event	Description
ContextChoiceBeforeOpen	This event occurs when the Context Choice window is about to open.

[C#]

```
private void editControl1_ContextChoiceBeforeOpen(object sender,
System.ComponentModel.CancelEventArgs e)
{
    // Display Context Choice popup if the lexem used to invoke Context Choice
    // is "this" or "me" only
    int ind = GetContextChoiceCharIndex(lexemLine);
    ILexem lex = lexemLine.LineLexems[ind-1] as ILexem;
    if ((lex.Text == "this") || (lex.Text == "me"))
        e.Cancel = false;
    else
        // Cancels the display of the Context Choice list.
        e.Cancel = true;
}
```

[VB .NET]

```
Private Sub editControl1_ContextChoiceBeforeOpen(ByVal sender As Object,
ByVal e As System.ComponentModel.CancelEventArgs) Handles
EditControl1.ContextChoiceBeforeOpen
    ' Display Context Choice popup if the lexem used to invoke the Context
    ' Choice is "this" or "me" only
    Dim ind As Integer = GetContextChoiceCharIndex(lexemLine)
    Dim lex As ILexem = lexemLine.LineLexems(ind - 1)
    If lex.Text = "this" OrElse lex.Text = "me" Then
        e.Cancel = False
    Else
        ' Cancel the display of the Context Choice list.
        e.Cancel = True
    End If
End Sub
```

Edit Control Event	Description
ContextChoiceClose	This event occurs when the Context Choice window has been closed.

[C#]

```
private void
editControl1_ContextChoiceClose(Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController controller, System.Windows.Forms.DialogResult dialogresult)
{
    // Clear the Context Choice items.
    this.editControl1.ContextChoiceController.Items.Clear();
}
```

[VB.NET]

```
Private Sub editControl1_ContextChoiceClose(ByVal controller As
Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController, ByVal
dialogresult As System.Windows.Forms.DialogResult) Handles
EditControl1.ContextChoiceClose
    ' Clear the Context Choice items.
    Me.editControl1.ContextChoiceController.Items.Clear()
End Sub
```

Edit Control Event	Description
ContextChoicelItemSelected	This event is raised when a Context Choice list item is selected.
ContextChoiceSelectedTextInsert	This event is raised when the editor is about to insert selected Context Choice item to the text. Action can be cancelled.

[C#]

```
private void
editControl1_ContextChoiceItemSelected(Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController sender,
Syncfusion.Windows.Forms.Edit.ContextChoiceItemSelectedEventArgs e)
{
    // Gets the selected item.
    IContextChoiceController controller = sender as IContextChoiceController;
    string selectedItemText = e.SelectedItem.Text;
}
```

[VB.NET]

```
Private Sub editControl1_ContextChoiceItemSelected(ByVal sender As Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController, ByVal e As Syncfusion.Windows.Forms.Edit.ContextChoiceItemSelectedEventArgs) Handles EditControl1.ContextChoiceItemSelected
    ' Gets the selected item.
    Dim controller As IContextChoiceController = sender
    Dim selectedItemText As String = e.SelectedItem.Text
End Sub
```

[C#]

```
private void editControl1_ContextChoiceSelectedTextInsert(Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController sender,
Syncfusion.Windows.Forms.Edit.ContextChoiceTextInsertEventArgs e)
{
    IContextChoiceController controller = sender as IContextChoiceController;

    // Gets the displayed text.
    string displayText = e.DisplayText;

    // Gets the text to be inserted.
    string insertText = e.InsertText;

    // Gets the item selected.
    string selectedItemText = e.SelectedItem.Text;
}
```

[VB.NET]

```
Private Sub editControl1_ContextChoiceSelectedTextInsert(ByVal sender As Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController, ByVal e As Syncfusion.Windows.Forms.Edit.ContextChoiceTextInsertEventArgs) Handles EditControl1.ContextChoiceSelectedTextInsert
    Dim controller As IContextChoiceController = sender

    ' Gets the displayed text.
    Dim displayText As String = e.DisplayText

    ' Gets the text to be inserted.
    Dim insertText As String = e.InsertText

    ' Gets the item selected.
    Dim selectedItemText As String = e.SelectedItem.Text
End Sub 'editControl1_ContextChoiceSelectedTextInsert
```

Filtering AutoComplete Items

Edit Control provides options to filter items in AutoComplete. This can be done by using the **FilterAutoCompleteItems** property.

Edit Control Property	Description
FilterAutoCompleteItems	Gets / sets value indicating whether Context Choice items should be filtered while typing.

FilterAutoCompleteItems property when set to **True**, filters the item in the AutoComplete Context Choice, and the filtered item alone will be visible. When set to **False**, all the items will be visible, and the selection will be navigated to the item.



Figure 57: Filtering Items in AutoComplete Context Choice

Showing / Hiding Context Choice Pop-up

You can also programmatically show / hide the Context Choice pop-up by calling the **ShowContextChoice** and

CloseContextChoice methods.

```
[C#]

// Shows the Context Choice pop-up window.
this.editControl1.ShowContextChoice();

// Closes the ContextChoice pop-up window.
this.editControl1.CloseContextChoice();
```

```
[VB .NET]

' Shows the Context Choice pop-up window.
Me.editControl1.ShowContextChoice()

' Closes the ContextChoice pop-up window.
```

```
Me.editControl1.CloseContextChoice()
```

A sample demonstrating the Context Choice feature is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Intellisense
Functions\\ContextChoiceandPromptDemo**

See Also

[Context Prompt](#)

4.6.6.2.2.3 *Context Prompt*

The **Context Prompt** feature allows you to create pop-ups for displaying variations of syntax for the text input by using the [Context Choice](#). This feature is modeled on the **Parameter Info** intellisense feature of Visual Studio. Each of the context prompt items can have a syntax specifier string and text message providing additional information on each item. The user is able to scroll through the syntax variations either by using the UP/DOWN ARROW keys or clicking on the UP/DOWN buttons on the pop-up.

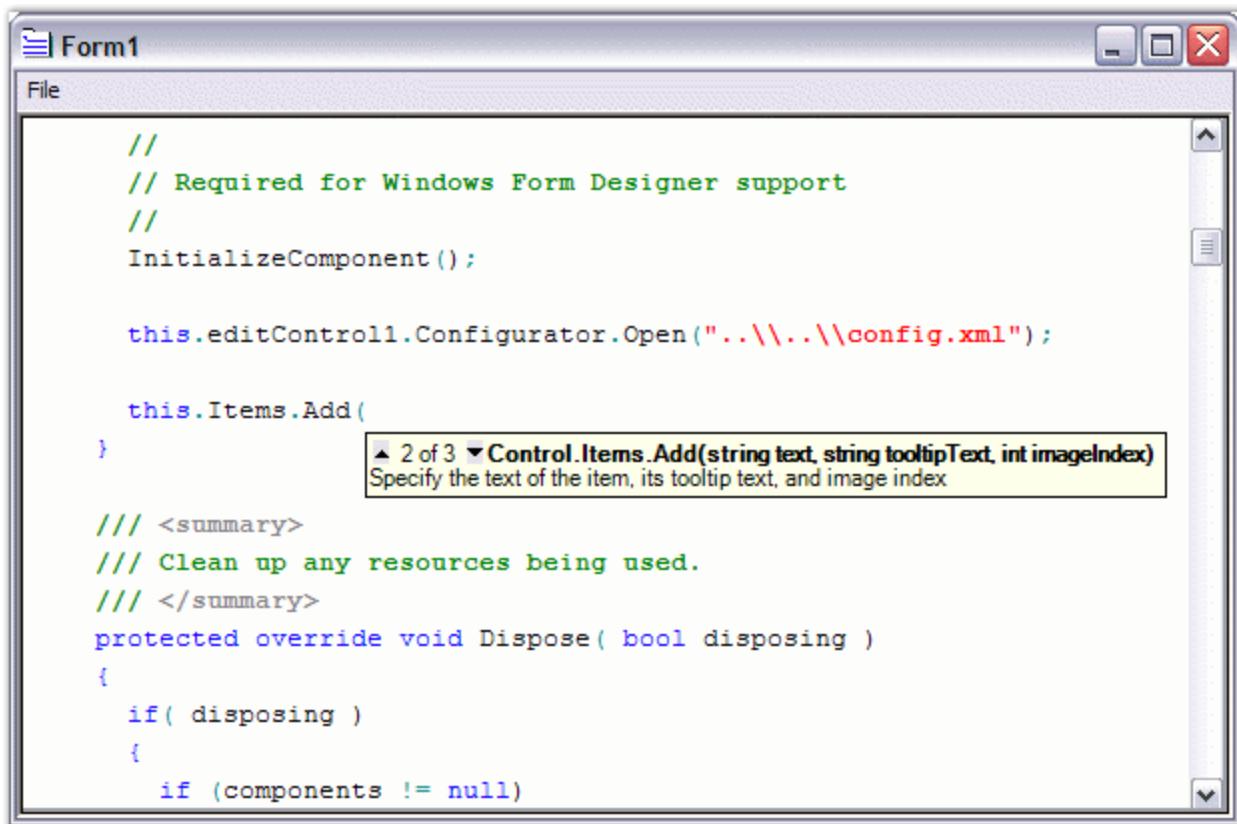


Figure 58: Context Prompt Pop-Up

The Context Prompt displaying characters are specified in the configuration file by using the **DropContextPrompt** field in the lexem for the corresponding character. If you wish to display the ContextPrompt pop-up in response to the opening brace - "(" or opening curly brace - "{" being typed, use the following XML code.

[XML]

```

<lexem BeginBlock="(" Type="Operator" DropContextPrompt="true"/>
<lexem BeginBlock="{{$" Type="Operator" DropContextPrompt="true"/>

```

The preceding code has to be placed within the `<lexems>` section of the configuration file.

Populating Context Prompt Popup

The Context Prompt is populated by handling the **ContextPromptOpen** event of Edit Control, and adding new prompts using the **AddPrompt** method.

Edit Control Event	Description
ContextPromptOpen	This event occurs when the Context Prompt has been opened.

[C#]

```
private void editControl1_ContextPromptOpen(object sender,
Syncfusion.Windows.Forms.Edit.ContextPromptUpdateEventArgs e)
{
    // Populate the Context Prompt.
    e.AddPrompt( "Control.Items.Add(string text, string tooltipText, int
imageIndex, int selectedImageIndex)", "Specify the text of the item, its
tooltip text, image index and selected image index" );
    e.AddPrompt( "Control.Items.Add(string text, string tooltipText, int
imageIndex)", "Specify the text of the item, its tooltip text, and image
index" );
    e.AddPrompt( "Control.Items.Add(string text, string tooltipText)", "
Specify the text of the item, and its tooltip text" );
}
```

[VB.NET]

```
Private Sub editControl1_ContextPromptOpen(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.ContextPromptUpdateEventArgs) Handles
EditControl1.ContextPromptOpen
    ' Populate the Context Prompt.
    e.AddPrompt("Control.Items.Add(string text, string tooltipText, int
imageIndex, int selectedImageIndex)", "Specify the text of the item, its
tooltip text, image index and selected image index")
    e.AddPrompt("Control.Items.Add(string text, string tooltipText, int
imageIndex)", "Specify the text of the item, its tooltip text, and image
index")
    e.AddPrompt("Control.Items.Add(string text, string tooltipText)", "Specify
the text of the item, and its tooltip text")
End Sub
```

Customization

Background Brush

The brush for the Context Prompt background is set by using the **ContextPromptBackgroundBrush** property of the Edit Control.

[C#]

```
this.editControl1.ContextPromptBackgroundBrush = new
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardDiagona
l, System.Drawing.Color.PapayaWhip, System.Drawing.Color.LemonChiffon);
```

[VB .NET]

```
Me.editControl1.ContextPromptBackgroundBrush = New  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardDiagona  
l, System.Drawing.Color.PapayaWhip, System.Drawing.Color.LemonChiffon)
```

Border Settings

The border color of the Context Prompt form is set by using the **ContextPromptBorderColor** property.

Edit Control Property	Description
ContextPromptBorderColor	Specifies the color of the Context Choice form border. Used when UseXPStyle property is set to False. Otherwise 3D border is drawn.

[C#]

```
this.editControl1.ContextPromptBorderColor = System.Drawing.Color.Pink;
```

[VB .NET]

```
Me.editControl1.ContextPromptBorderColor = System.Drawing.Color.Pink
```

Size Settings

The size of the Context Prompt form can be set by using the below given properties.

Edit Control Property	Description
ContextPromptSize	Gets / sets the size of the Context Prompt form.
UseCustomSizeContextPrompt	Gets / sets a value indicating whether custom Context Prompt size should be used.

[C#]

```
this.editControl1.ContextPromptSize = new System.Drawing.Size(125, 75);  
this.editControl1.UseCustomSizeContextPrompt = true;
```

[VB .NET]

```
Me.editControl1.ContextPromptSize = New System.Drawing.Size(125, 75)
Me.editControl1.UseCustomSizeContextPrompt = True
```

Context Prompt Operations

The Edit Control provides the following set of events for performing Context Prompt operations.

Edit Control Event	Description
ContextPromptBeforeOpen	This event occurs when the Context Prompt window is about to open. User can cancel it.
ContextPromptClose	This event occurs when the Context Prompt window has been closed.
ContextPromptSelectionChanged	This event occurs when a Context Prompt item has been selected.

[C#]

```
// Store the lexem name invoking the ContextPrompt popup.
string contextPromptLexem = "";

private void editControl1_ContextPromptBeforeOpen(object sender,
System.ComponentModel.CancelEventArgs e)
{
    ILexem lex;
    ILexemLine lexemLine =
        this.editControl1.GetLine(this.editControl1.CurrentLine);

    // Gets the index of the current word in that line.
    int ind = GetContextPromptCharIndex(lexemLine);

    if (ind<=0)
    {
        e.Cancel = true;
        return;
    }
    lex = lexemLine.LineLexems[ind-1] as ILexem;

    // If the count is less than '2', do not show the Context Prompt popup.
    if (lexemLine.LineLexems.Count<2)
        e.Cancel = true;
```

```
else
{
// Display Context Choice popup if the lexem used to invoke them is "this"
or "me" only.
if ((lex.Text == "Chat") || (lex.Text == "Database") || (lex.Text ==
"NewFile") || (lex.Text == "Find") || (lex.Text == "Home") || (lex.Text ==
"PieChart") || (lex.Text == "Tools"))
{
this.contextPromptLexem = lex.Text;
e.Cancel = false;
}
else
e.Cancel = true;
}
}
```

[VB.NET]

```
' Store the lexem name invoking the Context Prompt popup.
Dim contextPromptLexem As String = ""

Private Sub editControl1_ContextPromptBeforeOpen(ByVal sender As Object,
 ByVal e As System.ComponentModel.CancelEventArgs) Handles
editControl1.ContextPromptBeforeOpen
    Dim lex As ILexem
    Dim lexemLine As ILexemLine =
    Me.editControl1.GetLine(Me.editControl1.CurrentLine)

    ' Gets the index of the current word in that line.
    Dim ind As Integer = GetContextPromptCharIndex(lexemLine)

    If ind <= 0 Then
        e.Cancel = True
        Return
    End If
    lex = lexemLine.LineLexems(ind - 1)

    ' If the count is less than '2', do not show the Context Prompt popup.
    If lexemLine.LineLexems.Count < 2 Then
        e.Cancel = True
    Else
        ' Display Context Choice popup if the lexem used to invoke them is "this"
        or "me" only.
        If lex.Text = "Chat" OrElse lex.Text = "Database" OrElse lex.Text =
        "NewFile" OrElse lex.Text = "Find" OrElse lex.Text = "Home" OrElse
        lex.Text = "PieChart" OrElse lex.Text = "Tools" Then
            Me.contextPromptLexem = lex.Text
    End If
End Sub
```

```
e.Cancel = False  
Else  
e.Cancel = True  
End If  
End If  
End Sub
```

[C#]

```
// Clear the Context Prompt lexem name on close.  
private void editControl1_ContextPromptClose(object sender,  
Syncfusion.Windows.Forms.Edit.ContextPromptCloseEventArgs e)  
{  
    this.contextPromptLexem = "";  
}
```

[VB .NET]

```
' Clear the Context Prompt lexem name on close.  
Private Sub editControl1_ContextPromptClose(ByVal sender As Object, ByVal e  
As Syncfusion.Windows.Forms.Edit.ContextPromptCloseEventArgs)  
    Me.contextPromptLexem = ""  
End Sub
```

[C#]

```
// Display the selected Context Prompt item's index.  
private void  
editControl1_ContextPromptSelectionChanged(Syncfusion.Windows.Forms.Edit.Forms.Popup.ContextPrompt sender,  
Syncfusion.Windows.Forms.Edit.ContextPromptSelectionChangedEventArgs e)  
{  
    Console.WriteLine("SelectedIndex : " + e.SelectedIndex.ToString());  
    Console.WriteLine("ContextPromptSelectionChanged");  
}
```

[VB .NET]

```
' Display the selected Context Prompt item's index.  
Private Sub editControl1_ContextPromptSelectionChanged(ByVal sender As  
Syncfusion.Windows.Forms.Edit.Forms.Popup.ContextPrompt, ByVal e As  
Syncfusion.Windows.Forms.Edit.ContextPromptSelectionChangedEventArgs)  
    Console.WriteLine("SelectedIndex : " + e.SelectedIndex.ToString())  
    Console.WriteLine("ContextPromptSelectionChanged")  
End Sub
```

Advanced Customization

If you wish to do some advanced customization in the Context Prompt feature, like highlighting the current parameter to be input in bold, you can use the **ContextPromptOpen** and **ContextPromptUpdate** events.

For example, add the bolded items in the **ContextPromptOpen** event handler. The indices for the exact position of the text that needs to be bolded has to be manually calculated and specified along with some text information associated with that particular argument. The following code snippet illustrates this.

[C#]

```
// To display some text in bold within the prompt.
private void editControl1_ContextPromptOpen(object sender,
Syncfusion.Windows.Forms.Edit.ContextPromptEventArgs e)
{
    Console.WriteLine("ContextPromptOpen");

    // Bolded Items should be added in this handler.
    ContextPromptItem item = null;
    item = e.AddPrompt( "Control.Items.Add(string text, string tooltipText,
int imageIndex, int selectedImageIndex)", "Specify the text of the item,
its tooltip text, image index and selected image index" );

    // Specify the text to be displayed in bold in the Context Prompt.
    item.BoldedItems.Add( 18, 11, "Text to be added" );
    item.BoldedItems.Add( 31, 18, "Text of the tooltip" );
    item.BoldedItems.Add( 51, 14, "Zero-based index of the image or -1 if no
image should be used." );
    item.BoldedItems.Add( 67, 14, "Zero-based index of the image for selection
or -1 if no image should be used." );

    item = e.AddPrompt( "Control.Items.Add(string text, string tooltipText,
int imageIndex)", "Specify the text of the item, its tooltip text, and
image index" );
    item.BoldedItems.Add( 18, 11, "Text to be added" );
    item.BoldedItems.Add( 31, 18, "Text of the tooltip" );
    item.BoldedItems.Add( 51, 14, "Zero-based index of the image or -1 if no
image should be used." );

    item = e.AddPrompt( "Control.Items.Add(string text, string tooltipText)",
"Specify the text of the item, and its tooltip text" );
    item.BoldedItems.Add( 18, 11, "Text to be added" );
    item.BoldedItems.Add( 31, 18, "Text of the tooltip" );
}
```

[VB.NET]

```
' To display some text in bold within the prompt.  
Private Sub editControl1_ContextPromptOpen(ByVal sender As Object, ByVal e As Syncfusion.Windows.Forms.Edit.ContextPromptUpdateEventArgs) Handles EditControl1.ContextPromptOpen  
Console.WriteLine("ContextPromptOpen")  
    ' Bolded Items should be added in this handler.  
    Dim item As ContextPromptItem  
  
    item = e.AddPrompt("Control.Items.Add(string text, string tooltipText, int  
imageIndex, int selectedImageIndex)", "Specify the text of the item, its  
tooltip text, image index and selected image index")  
  
    ' Specify the text to be displayed in bold in the Context Prompt.  
    item.BoldedItems.Add(18, 11, "Text to be added")  
    item.BoldedItems.Add(31, 18, "Text of the tooltip")  
    item.BoldedItems.Add(51, 14, "Zero-based index of the image or -1 if no  
image should be used.")  
    item.BoldedItems.Add(67, 14, "Zero-based index of the image for selection  
or -1 if no image should be used.")  
  
    item = e.AddPrompt("Control.Items.Add(string text, string tooltipText, int  
imageIndex)", "Specify the text of the item, its tooltip text, and image  
index")  
    item.BoldedItems.Add(18, 11, "Text to be added")  
    item.BoldedItems.Add(31, 18, "Text of the tooltip")  
    item.BoldedItems.Add(51, 14, "Zero-based index of the image or -1 if no  
image should be used.")  
  
    item = e.AddPrompt("Control.Items.Add(string text, string tooltipText)",  
"Specify the•_
```

Select the items that should be bolded in the ContextPromptUpdate event handler. The following code snippet illustrates this.

[C#]

```
private void editControl1_ContextPromptUpdate(object sender,  
Syncfusion.Windows.Forms.Edit.ContextPromptUpdateEventArgs e)  
{  
    // Select the items that should be bolded.  
    if( e.List.SelectedItem != null )  
    {
```

```
// Get list of the lexems that are inside the current stack.  
IList list = editControl1.GetLexemsInsideCurrentStack( false );  
if( list == null ) return;  
  
int iBoldedIndex = 0;  
foreach( ILexem lexem in list )  
{  
    if( lexem.Text == "," )  
        iBoldedIndex++;  
}  
  
if( iBoldedIndex >= e.List.SelectedItem.BoldedItems.Count )  
    e.List.SelectedItem.BoldedItems.SelectedItem = null;  
else  
    // Gets or sets selected item.  
    e.List.SelectedItem.BoldedItems.SelectedItem =  
    e.List.SelectedItem.BoldedItems[iBoldedIndex];  
}  
}
```

[VB.NET]

```
Private Sub editControl1_ContextPromptUpdate(ByVal sender As Object, ByVal e  
As Syncfusion.Windows.Forms.Edit.ContextPromptUpdateEventArgs) Handles  
EditControl1.ContextPromptUpdate  
  
    ' Select the items that should be bolded.  
    If Not (e.List.SelectedItem Is Nothing) Then  
  
        ' Get list of the lexems that are inside the current stack.  
        Dim list As IList = editControl1.GetLexemsInsideCurrentStack(False)  
        If list Is Nothing Then  
            Return  
        End If  
  
        Dim iBoldedIndex As Integer = 0  
        Dim lexem As ILexem  
        For Each lexem In list  
            If lexem.Text = "," Then  
                iBoldedIndex += 1  
            End If  
        Next lexem  
  
        If iBoldedIndex >= e.List.SelectedItem.BoldedItems.Count Then  
            e.List.SelectedItem.BoldedItems.SelectedItem = Nothing  
        Else
```

```
' Gets or sets selected item.  
e.List.SelectedItem.BoldedItems.SelectedItem =  
e.List.SelectedItem.BoldedItems(iBoldedIndex)  
End If  
End If  
End Sub
```

Showing / Hiding Context Prompt Pop-up

You can also programmatically show / hide the Context Prompt pop-up using the **ShowContextPrompt** and **CloseContextPrompt** methods.

[C#]

```
// Shows the Context Prompt pop-up window.  
this.editControl1.ShowContextPrompt();  
  
// Closes the Context Prompt pop-up window.  
this.editControl1.CloseContextPrompt();
```

[VB .NET]

```
' Shows the Context Prompt pop-up window.  
Me.editControl1.ShowContextPrompt()  
  
' Closes the Context Prompt pop-up window.  
Me.editControl1.CloseContextPrompt();
```

A sample demonstrating the Context Prompt feature is available in the below sample installation path.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Intellisense
Functions\\ContextChoiceandPromptDemo*

4.6.6.2.2.4 Context ToolTip

The **Context ToolTip** displays helpful tooltips when the mouse is hovered over a lexem in the Edit Control. This feature is modeled on the **Quick Info** intellisense feature of Visual Studio. Whenever the mouse hovers over a token, the **UpdateContextTooltip** event is fired for quick information on the lexem. If some text information is provided, it is displayed in a tooltip.

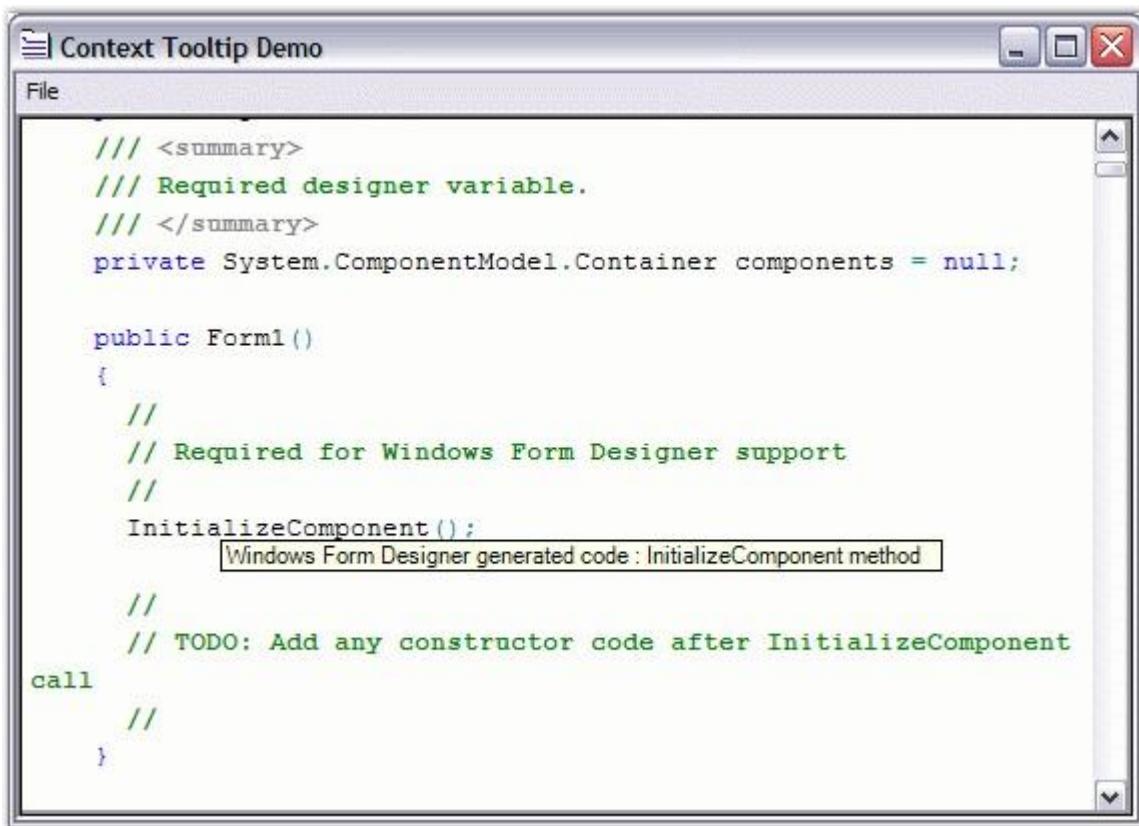


Figure 59: Context ToolTip

The Context ToolTip can be populated with additional information on the corresponding lexem by handling the **UpdateContextTooltip** event of Edit Control.

[C#]

```
private void editControl1_UpdateContextToolTip(object sender,
Syncfusion.Windows.Forms.Edit.Dialogs.UpdateTooltipEventArgs e)
{
    if( e.Text == string.Empty )
    {
        Point pointVirtual = editControl1.PointToVirtualPosition( new
Point( e.X, e.Y ) );
        if( pointVirtual.Y > 0 )
        {
            // Get the current line
            ILexemLine line = editControl1.GetLine(
pointVirtual.Y );
            if( line != null )
            {

```

```
// Get tokens from the current line
ILexem lexem = line.FindLexemByColumn(
pointVirtual.X );
if( lexem != null )
{
    // Set the desired information
tooltip
    e.Text = "This is additional
information on " + lexem.Text;
}
}
}
```

[VB.NET]

```
Private Sub editControl1_UpdateContextToolTip(ByVal sender As Object, ByVal e
As Syncfusion.Windows.Forms.Edit.Dialogs.UpdateTooltipEventArgs) Handles
EditControl1.UpdateContextToolTip
    If e.Text = String.Empty Then
        Dim pointVirtual As Point =
editControl1.PointToVirtualPosition(New Point(e.X, e.Y))

        If pointVirtual.Y > 0 Then
            ' Get the current line
            Dim line As ILine = editControl1.GetLine(pointVirtual.Y)

            If Not (line Is Nothing) Then
                ' Get tokens from the current line
                Dim lexem As ILexem =
line.FindLexemByColumn(pointVirtual.X)

                If Not (lexem Is Nothing) Then
                    ' Set the desired information tooltip
                    e.Text = "This is additional information on " +
lexem.Text;
                End If
            End If
        End If
    End If
End Sub
```

Customization

Background Brush

The brush for the Context ToolTip background can be set by using the **ContextTooltipBackgroundBrush** property.

[C#]

```
this.editControl1.ContextTooltipBackgroundBrush = new  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.PatternStyle.Percent05,  
System.Drawing.Color.LavenderBlush, System.Drawing.Color.Khaki);
```

[VB .NET]

```
Me.editControl1.ContextTooltipBackgroundBrush = New  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.PatternStyle.Percent05,  
System.Drawing.Color.LavenderBlush, System.Drawing.Color.Khaki)
```

Border Settings

The border color of the Context ToolTip form is set by using the **ContextTooltipBorderColor** property.

Edit Control Property	Description
ContextTooltipBorderColor	Specifies the color of the Context Tooltip form border. Used when UseXPStyle property is set to False. Otherwise 3D border is drawn.

[C#]

```
this.editControl1.ContextTooltipBorderColor = System.Drawing.Color.Orange;
```

[VB .NET]

```
Me.editControl1.ContextTooltipBorderColor = System.Drawing.Color.Orange
```

Showing the ToolTip

The Context ToolTip window can be shown by setting the ShowContextTooltip property to **True**.

[C#]

```
// Shows the Context ToolTip pop-up window.
```

```
this.editControl1.ShowContextTooltip = true;
```

[VB .NET]

```
' Shows the Context ToolTip pop-up window.  
Me.editControl1.ShowContextTooltip = True
```

ToolTip Delay

It is also possible to specify the time delay after which the tooltip should be displayed by using the **ToolTipDelay** property.

[C#]

```
// Displays the tooltip pop-up after 1000 milliseconds( 1 sec )  
this.editControl1.ToolTipDelay = 1000;
```

[VB]

```
' Displays the tooltip pop-up after 1000 milliseconds( 1 sec )  
Me.editControl1.ToolTipDelay = 1000
```

Closing the ToolTip

The Context ToolTip window is closed by using the **CloseContextTooltip** method.

[C#]

```
// Closes the Context ToolTip pop-up window.  
this.editControl1.CloseContextTooltip();
```

[VB .NET]

```
' Closes the Context ToolTip pop-up window.  
Me.editControl1.CloseContextTooltip();
```

A sample demonstrating the Context Tooltip feature is available in the below sample installation path.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Intellisense Functions\\ContextTooltipDemo**

4.6.6.2.3 Custom Cursor

This section discusses the cursor settings of the Edit Control.

Presently, Edit Control supports all the cursors contained in the **Windows Forms Cursors** enumerator. You can set any desired cursor to the Edit Control by using its **Cursor** property as shown below.

Edit Control Property	Description
Cursor	<p>Sets the cursor that is displayed when the mouse pointer is over the control. The options provided are</p> <ul style="list-style-type: none">• AppStarting• Arrow• Cross• Default• Hand• Help• HSplit• IBeam• No• NoMove2D• NoMoveHoriz• NoMoveVert• PanEast• PanNE• PanNorth• PanNW• PanSE• PanSouth• PanSW• PanWest• SizeAll• SizeNESW• SizeNS• SizeNWSE• SizeWE• UpArrow• VSplit• WaitCursor

[C#]

```
// Set any desired cursor to the Edit Control.  
this.editControl1.Cursor = System.Windows.Forms.Cursors.Hand;
```

[VB .NET]

```
' Set any desired cursor to the Edit Control.  
Me.editControl1.Cursor = System.Windows.Forms.Cursors.Hand
```

Showing / Hiding Cursor Caret

The **ShowCaret** and **HideCaret** methods are used to either show / hide the cursor caret.

[C#]

```
// Shows the cursor caret.  
this.editControl1.ShowCaret();  
  
// Hides the cursor caret.  
this.editControl1.HideCaret();
```

[VB .NET]

```
' Shows the cursor caret.  
Me.editControl1.ShowCaret()  
  
' Hides the cursor caret.  
Me.editControl1.HideCaret()
```

A sample demonstrating the Custom Cursor feature is available in the below sample installation path.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\CustomCursorDemo*

4.6.6.2.4 Intellimouse Scrolling

Essential Edit provides excellent support for viewport navigation including intellimouse scrolling. Commonly used keyboard navigation functions like PAGE UP/PAGE DOWN keys, ARROW keys, and CTRL+ARROW keys are fully supported by Essential Edit.

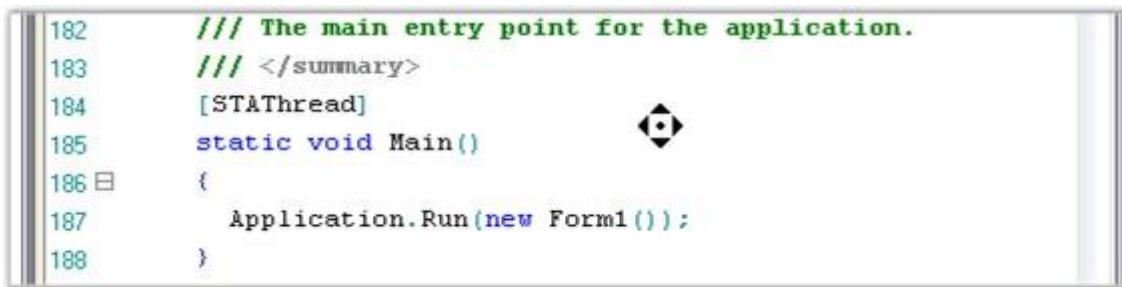


Figure 60: Preview of Intellimouse in Edit Control

See Also

Some of the intellisense features.

[Code Snippets](#), [Context Choice](#), [Context Prompt](#), [Context Tooltip](#)

4.6.6.2.5 Drag-and-drop

The Edit Control fully supports the file drop functionality. Any text file can be dragged onto the Edit Control, which then displays the contents of the file, as if the file had been opened with the Edit Control.

The Edit Control also supports the text drag-and-drop functionality. In other words, you can drag a piece of text from one region in the Edit Control to another. You can also drag text from other editor controls like the RichTextBox onto the Edit Control. These features are supported out of the box, and no explicit handling of drag-and-drop operations are required.

Make sure to set the **AllowDrop** property of the Edit Control to **True** for this purpose.

[C#]

```
// Enable drag and drop.
this.editControl1.AllowDrop = true;
```

[VB .NET]

```
' Enable drag and drop.
```

```
Me.editControl1.AllowDrop = True
```

4.7 Text Export

The following topics elaborates on the exporting feature in Essential Edit:

4.7.1 XML, RTF and HTML Export

Edit Control has the ability to export its contents and its associated [syntax highlighting](#) information into XML, RTF or HTML formats. This allows the user to share text associated with the Edit Control along with its attributes such as syntax highlighting, line numbers, underlines and many such useful features in universally accepted formats like XML, RTF and HTML.

The following methods can implemented for this purpose.

Edit Control Method	Description
SaveAsXML	Export the Edit Control's contents into XML format and save it into any desired XML file.
SaveAsRTF	Export the Edit Control's contents into RTF format and save it into any desired RTF file.
SaveAsHTML	Export the Edit Control's contents into HTML format and save it into any desired HTML file.

[C#]

```
// Export the Edit Control's contents into XML format and save it into a XML
file.
this.editControl1.SaveAsXML("testXML.xml");

// Export the Edit Control's contents into RTF format and save it into a RTF
file.
this.editControl1.SaveAsRTF("testRTF.rtf");

// Export the Edit Control's contents into HTML format and save it into a
HTML file.
this.editControl1.SaveAsHTML("testHTML.html");
```

[VB.NET]

```
' Export the Edit Control's contents into XML format and save it into a XML file.  
Me.editControl1.SaveAsXML("testXML.xml")  
  
' Export the Edit Control's contents into RTF format and save it into a RTF file.  
Me.editControl1.SaveAsRTF("testRTF.rtf")  
  
' Export the Edit Control's contents into HTML format and save it into a HTML file.  
Me.editControl1.SaveAsHTML("testHTML.html")
```

Edit Control is also capable of providing XML, RTF and HTML source code for generating documents in the corresponding formats by using the following methods.

Edit Control Method	Description
GetTextAsRTF	Gets the source code to generate XML document for the text in the Edit Control.
GetTextAsXML	Gets the source code to generate RTF document for the text in the Edit Control.
GetTextAsHTML	Gets the source code to generate HTML document for the text in the Edit Control.

[C#]

```
// Gets the source code to generate XML document.  
this.editControl1.GetTextAsXML();  
  
// Gets the source code to generate XML document for the text range specified.  
this.editControl1.GetTextAsXML(coordinatePoint1, coordinatePoint2);
```

[VB.NET]

```
' Gets the source code to generate XML document.  
Me.editControl1.GetTextAsXML()  
  
' Gets the source code to generate XML document for the text range specified.  
Me.editControl1.GetTextAsXML(coordinatePoint1, coordinatePoint2)
```

A sample demonstrating the above feature is available in the below sample installation path.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Export\\ExportDemo*

4.7.2 Schema Definition File for XML Syntax Coloring Configuration File

Essential Edit now comes with an XML Schema Definition (XSD) file that provides schema information for the XML language definition syntax. The main advantage of this is, just by including this XSD file along with the XML Syntax Coloring Configuration file in the Visual Studio .NET project, the member list drop-down will be displayed while the elements in the XML Syntax Coloring Configuration file are edited in the Visual Studio Code Editor.

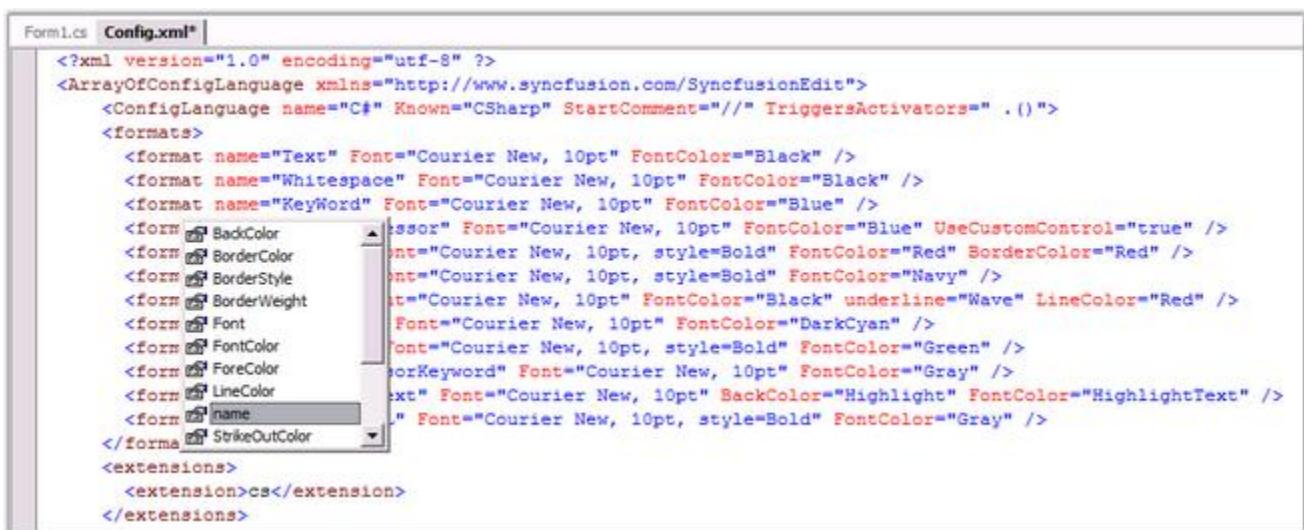


Figure 61: XML Schema Definition File

4.8 File Sharing and Stream Handling

Edit Control provides extensive support for File and Stream Handling operations through the APIs discussed in this section.

4.8.1 Creating, Loading, Saving And Dropping Files

This section discusses the file operations supported in Edit Control.

Creating Files

The **New** and **NewFile** methods are used to create a new stream or file, and optionally allow you to set the language to be used by specifying the appropriate configuration settings.

Edit Control Method	Description
New	Creates an empty stream and allows the editor to for editing.
NewFile	Creates new empty file with specified coloring.

[C#]

```
// Creates a new stream with default configuration settings.
this.editControl.New();

// Creates a new file with default configuration settings.
this.editControl.NewFile();

// Creates a new stream with specified configuration settings.
this.editControl.New(ConfigLanguage lang);

// Creates a new file with specified configuration settings.
this.editControl.NewFile(IConfigLanguage lang);
```

[VB .NET]

```
' Creates a new file.
Me.editControl1.NewFile()

Me.editControl1.[New]()

' "config" is Configuration Settings file of type IConfigLanguage.
Me.editControl1.NewFile(config)

Me.editControl1.[New](config)
```

Loading Files

The **LoadFile** method loads the content of any desired file into the Edit Control.

Edit Control Method	Description
LoadFile	Shows open file dialog to the user and opens the selected file.



Note: The character encoding for the text can also be specified while loading the file.

[C#]

```
// Displays the Open File dialog.
this.editControl1.LoadFile();

// Loads the content of the specified file.
this.editControl1.LoadFile("Temp.txt", Encoding.ASCII);
```

[VB .NET]

```
' Displays the Open File dialog.
Me.editControl1.LoadFile()

' Loads the content of the specified file.
Me.editControl1.LoadFile("Temp.txt", Encoding.ASCII)
```

Saving Files

The following methods are used to save a file in the Edit Control.

Edit Control Method	Description
SaveFile	Saves the contents of the Edit Control to a specified file.
Save	Invokes the save file dialog box and lets you save the contents of the Edit Control to the specified file.
SaveAs	Opens SaveAs dialog and prompts you to enter the name of the file.
SaveModified	Saves the file only if it was modified and prompts for filename if needed. This is especially useful when the application is about to be closed or a new file is being loaded into the Edit Control.

[C#]

```
// Saves the contents of the file.
```

```
this.editControl1.SaveFile("Temp.txt", Encoding.Unicode,  
Syncfusion.IO.NewLineStyle.Control);  
  
// Displays the Save File dialog.  
this.editControl1.Save();  
  
// Displays the SaveAs dialog.  
this.editControl1.SaveAs();  
  
// Saves the contents of the file after modification, when a new file is  
loaded, or when a file is closed.  
this.editControl1.SaveModified();
```

[VB.NET]

```
' Saves the contents of the file.  
Me.editControl1.SaveFile("Temp.txt", Encoding.Unicode,  
Syncfusion.IO.NewLineStyle.Control)  
  
' Displays the Save File dialog.  
Me.editControl1.Save()  
  
' Displays the SaveAs dialog.  
Me.editControl1.SaveAs()  
  
' Saves the contents of the file after modification, when a new file is  
loaded, or when a file is closed.  
Me.editControl1.SaveModified()
```

Dropping Files

Files can be dropped onto the Edit Control by using the properties given below.

Edit Control Property	Description
DropAllFiles	Gets / sets value indicating whether all files can be dropped onto Edit Control. If set to False, only files with extensions contained in FileExtensions can be dropped.
FileExtensions	Gets / sets extensions of files that can be dropped to Edit Control.

[C#]

```
// Drops all files onto Edit Control.  
this.editControl1.DropAllFiles = true;  
  
// Specifies the file extensions of files that can be dropped onto Edit  
Control.  
this.editControl1.FileExtensions = new string[] {".cs", ".sql", ".vb",  
".xml"};
```

[VB.NET]

```
' Drops all files onto Edit Control.  
Me.editControl1.DropAllFiles = True  
  
' Specifies the file extensions of files that can be dropped onto Edit  
Control.  
Me.editControl1.FileExtensions = New String() {".cs", ".sql", ".vb", ".xml"}
```

4.8.2 Loading And Saving Contents

The contents of the Edit Control can be loaded and saved to a particular stream. This can be achieved by using the methods given below.

Edit Control Method	Description
LoadStream	Loads the stream and the corresponding configuration.
FlushChanges	Flushes changes to the current stream.
SaveStream	Saves content to the specified stream using specified encoding and line end style.

[C#]

```
// Loads the content of the specified stream into the Edit Control.  
this.editControl1.LoadStream(streamName);  
  
// Loads the specified stream and configuration.  
this.editControl1.LoadStream(streamName, config);  
  
// Saves changes made to the contents of the Edit Control into the current  
stream.
```

```
this.editControl1.FlushChanges();

// Saves content to the specified stream using specified encoding and line
end style.
this.editControl1.SaveStream(System.IO.Stream.Null ,
Encoding.BigEndianUnicode, Syncfusion.IO.NewLineStyle.Mac);
```

[VB.NET]

```
' Loads the content of the specified stream into the edit control.
Me.editControl1.LoadStream(streamName)

' Loads the specified stream and configuration.
Me.editControl1.LoadStream(streamName, config)

' Saves changes made to the contents of the Edit Control into the current
stream.
Me.editControl1.FlushChanges()

' Saves content to the specified stream using specified encoding and line end
style.
Me.editControl1.SaveStream(System.IO.Stream.Null , Encoding.BigEndianUnicode,
Syncfusion.IO.NewLineStyle.Mac)
```

Getting Details of Currently Loaded File

The name of the file that is currently loaded can be set by using the **FileName** property.

Edit Control Property	Description
FileName	Gets / sets the name of the currently opened file.

[C#]

```
// Gets or sets the name of the file loaded in the Edit Control.
this.editControl1.FileName = "Temp.txt";
```

[VB.NET]

```
' Gets or sets the name of the file loaded in the Edit Control.
Me.editControl1.FileName = "Temp.txt"
```

Getting Details of Currently Loaded Stream

The name of the stream that is currently loaded in the Edit Control can be set by using the **FileOpened** property.

Edit Control Property	Description
FileOpened	Gets / sets the filestream that is used as an input.

[C#]

```
// Gets or sets the name of the stream that is currently loaded in the Edit
Control.
this.editControl1.FileOpened = new FileStream("Temp.txt", FileMode.Create);
```

[VB .NET]

```
' Gets or sets the name of the stream that is currently loaded in the Edit
Control.
Me.editControl1.FileOpened = New FileStream("Temp.txt", FileMode.Create)
```

See Also

[Creating, Loading and Saving a File](#), [Saving and Cancelling Changes](#)

4.8.3 Saving And Cancelling Changes

This section demonstrates how changes made to the contents of the Edit Control can be saved or discarded.

SaveOnClose Property

This property specifies whether the default Save Changes prompt should be displayed on closing the Edit Control.

[C#]

```
// Disables the default Save Changes prompt that appears when the form
hosting Edit Control containing unsaved contents is closed.
this.editControl1.SaveOnClose = false;
```

[VB.NET]

```
' Disables the default Save Changes prompt that appears when the form hosting
Edit Control containing unsaved contents is closed.
Me.editControl1.SaveOnClose = False
```

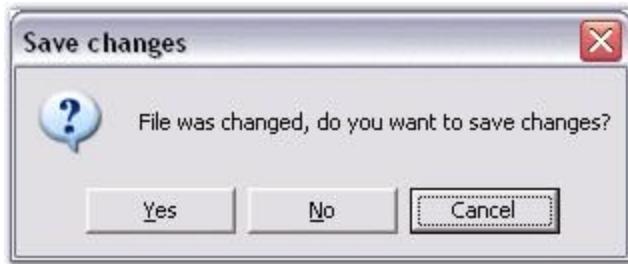


Figure 62: Default Save Changes Prompt Dialog Box

Saving Changes without displaying the Save Changes Prompt

When the **SaveOnClose** property is set to **False**, the default Save Changes prompt does not appear. The user should perform some custom Save routine in the **Closing** event handler of the host form, to save the unsaved contents in the Edit Control; If not they will be lost.

[C#]

```
private void Form1_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    if (this.editControl1.SaveOnClose == false)
    {
        if (this.editControl1.SaveModified() == true)
        // Perform custom Save routine or show custom Save Changes dialog or
        // set Cancel to False.
        e.Cancel = false;
        else
        e.Cancel = true;
    }
}
```

[VB.NET]

```
Private Sub Form1_Closing(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    If Me.editControl1.SaveOnClose = False Then
        If Me.editControl1.SaveModified() = True Then
            ' Perform custom Save routine or show custom Save Changes dialog or set Cancel to False.
            e.Cancel = False
        Else
            e.Cancel = True
        End If
    End If
End Sub 'Form1_Closing
```

Saving Changes using the Save Changes Prompt

When the **SaveOnClose** property is set to **True**, the default Save Changes prompt appears on closing the Edit Control without saving the contents. Click **Yes** to save the changes, **No** to discard the changes, or **Cancel** to close the Save Changes prompt.

The above task can be further customized by handling the **Closing** event of Edit Control. The Closing event is triggered just before a file or stream is closed in the Edit Control.

[C#]

```
private void editControl1_Closing(object sender,
Syncfusion.Windows.Forms.Edit.StreamCloseEventArgs e)
{
// Cancel the file or stream closing action.
e.Action = SaveChangesAction.Cancel;

// Close the file or stream without saving the unsaved contents, the changes will be lost forever.
e.Action = SaveChangesAction.Discard;

// Silently saves the unsaved contents to the currently open file or stream.
// If the contents have not been saved to a file or stream as yet, the Save Changes prompt is displayed.
e.Action = SaveChangesAction.Save;

// Displays the default Save Changes prompt if there are unsaved contents when the file or stream is closed.
e.Action = SaveChangesAction.ShowDialog;
}
```

[VB.NET]

```
Private Sub editControl1_Closing(ByVal sender As Object, ByVal e As Syncfusion.Windows.Forms.Edit.StreamCloseEventArgs) Handles EditControl1.StreamClose
    ' Cancel the file or stream closing action.
    e.Action = SaveChangesAction.Cancel

    ' Close the file or stream without saving the unsaved contents, the changes will be lost forever.
    e.Action = SaveChangesAction.Discard

    ' Silently saves the unsaved contents to the currently open file or stream.
    ' If the contents have not been saved to a file or stream as yet, the Save Changes prompt is displayed
    e.Action = SaveChangesAction.Save

    ' Displays the default Save Changes prompt if there are unsaved contents when the file or stream is closed.
    e.Action = SaveChangesAction.ShowDialog
End Sub
```



Note: The default value of e.Action is SaveChangesAction.ShowDialog.

Close Method

This method closes the currently open file or stream and displays the Edit Control in the read-only mode, until a new file or stream is opened.

Edit Control Method	Description
Close	Closes stream, makes control readonly.

[C#]

```
// Closes the currently open file or stream in the Edit Control.
this.editControl1.Close();
```

[VB.NET]

```
' Closes the currently open file or stream in the Edit Control.
Me.editControl1.Close()
```

See Also

[Creating, Loading and Saving a File](#), [Loading and Saving Contents](#)

4.8.4 File Sharing

By default, Edit Control locks the file currently loaded into it, and does not allow access to the same by any external application. To enable file sharing, set the **Shared FileMode** property of the Edit Control to **True**.

Edit Control Property	Description
Shared FileMode	Gets / sets value indicating whether file should be opened in shared mode.

[C#]

```
// Enable file sharing.
this.editControl1.Shared FileMode = true;
```

[VB .NET]

```
' Enable file sharing.
Me.editControl1.Shared FileMode = True
```

4.8.5 Lexical Analysis And Semantic Parsing

Text parsing occurs when a new document is loaded or when modifications occur in an already loaded document. In case of modifications, the Edit Control intelligently reparses only what is necessary to ensure that the text model is up to date with the contents of the editor. Ideally, parsing the Edit Control occurs in a two-phase approach. The first phase is lexical analysis and the second one is semantic parsing.

Lexical Analysis breaks up text into tokens, while semantic parsing goes a step further and assigns extra contextual meaning to the tokens. Semantic relations recognized by the semantic parser are based on how human beings represent knowledge of the world. Semantic parsing allows tokens to be accessed and processed in a more meaningful way than lexical analysis, moving the automation of understanding the tokens to a higher level. A semantic parser consumes the output of the lexical analyzer, and operates by analyzing the sequence of tokens returned. The parser matches these sequences to an end state which may be one of the possible end states. The end states define the goals of the parser. When an end state is reached, the program using the parser implements some action-specific code.

Additionally, parsers can detect the situation when no legal end state can be reached, from the sequence of tokens that have been processed.

Lexical Analysis

Lexical Analysis is the process of scanning text in a document and breaking it up into meaningful tokens. The purpose of lexical analyzers is to take a stream of input characters, and decode them into higher level tokens that a semantic parser can understand. In this stage, the text is split into tokens with the help of some special rules specified by the user. For instance, the user can specify "+=" or "end if" expressions as single tokens using the Split tag in the configuration file. Tokens are plain text, and have no additional information or meaning associated with them.

Semantic Parsing

In this stage, the syntax highlighting rules are applied. These rules can be as simple as identifying the format name of the token, and applying the appropriate font or color settings. But this simple two-phase procedure was not very flexible in complex scenarios involving embedded scripts. Hence the entire process has been enhanced from the very beginning, by merging the lexical analysis and semantic parsing.

The **Parser** property indicates the parser used for parsing the currently loaded document in the Edit Control. The parsing process could be performed for any (or all) of the following purposes - syntax highlighting, intellisense, outlining and so on. The rules for the parsing process are specified in the XML based configuration file used.

[C#]

```
// Indicates the parser used for parsing the currently loaded document in the
Edit Control.
RenderableLexemParser lexemParser = this.editControl1.Parser;
```

[VB .NET]

```
' Indicates the parser used for parsing the currently loaded document in the
Edit Control.
Dim lexemParser As RenderableLexemParser = Me.editControl1.Parser
```

Parsing Modes

Edit Control supports several modes of text parsing which can be specified to the **ParsingMode** property by using the **TextParsingMode** enumerator. The default value of the ParsingMode property is set to **PartialParsingNoFallback**.

Edit Control Property	Description
ParsingMode	Gets / sets text parsing mode. User can select between high parsing

	<p>speed or high syntax highlighting accuracy. The options provided are</p> <ul style="list-style-type: none"> • FullParsing • PartialParsingNoFallback • PartialParsingWithFallback
--	---

When ParsingMode is set to **FullParsing**, the text in the Edit Control is parsed completely and accurately, and then features like syntax highlighting, outlining, bracket highlighting, indentation guidelines, and so on are applied. FullParsing is time consuming, and can potentially cause performance issues as Edit Control stays frozen till this process is completed. Ideally, it should be undertaken for small files only.

When ParsingMode is set to **PartialParsingNoFallback**, text parsing is done on a need basis, i.e., only those regions of the text in the Edit Control that have to be displayed get parsed. The text parsing is not always accurate in such scenarios, and hence features like syntax highlighting, outlining, bracket highlighting, indentation guidelines, and so on, maybe incorrectly applied. This is the fastest ParsingMode in the Edit Control, and hence should be used in large file handling scenarios.

When ParsingMode is set to **PartialParsingWithFallback**, text parsing is once again done on a need basis like in PartialParsingNoFallback mode. The only difference is that if the text gets incorrectly parsed, the incorrectly parsed text is treated as of type regular "Text" format, and features like syntax highlighting, outlining, bracket highlighting, indentation guidelines, and so on, get applied as per Text format specifications in the associated configuration settings.

[C#]

```
// ParsingMode is set to FullParsing.
this.editControl1.ParsingMode =
Syncfusion.Windows.Forms.Edit.Enums.TextParsingMode.FullParsing;
```

[VB .NET]

```
' ParsingMode is set to FullParsing.
Me.editControl1.ParsingMode =
Syncfusion.Windows.Forms.Edit.Enums.TextParsingMode.FullParsing
```

4.8.6 Clearing/Flushing Saved Changes

When the **MarkChangedLines** property is used in the Edit control, the lines that are changed will be denoted by a yellow color indicator, and the saved lines will be indicated by a green color indicator.

When the **FlushChanges()** method is called, the changes performed in the editor after the last save will be removed from the editor, but the saved lines will remain.

If the **FlushSavedLines** property is enabled, the **FlushChanges()** method will remove the previously saved lines along with the recent changes.

Property	Description
FlushSavedLines	Gets \ Sets whether to reset the saved lines changes.

[C#]

```
//Gets or sets a value to reset the saved line changes.  
this.editControl1.FlushSavedLines = true;
```

[VB .NET]

```
//Gets or sets a value to reset the saved line changes.  
Me.editControl1.FlushSavedLines = True
```

4.9 Appearance

The appearance customization features of the Edit control are discussed under the following topics:

4.9.1 Visual Settings

This section covers the below topics:

4.9.1.1 Size

This section discusses the size settings of the Edit Control.

AutoSize

The Edit Control can be autoresized by setting the **AutoSize** property to **True**.

[C#]

```
this.editControl1.AutoSize = true;
```

[VB .NET]

```
Me.editControl1.AutoSize = True
```

Minimum Size

The **MinimumSize** property gets / sets the minimum size of the control in autosize mode.

[C#]

```
this.editControl1.MinimumSize = new System.Drawing.Size(10, 10);
```

[VB .NET]

```
Me.editControl1.MinimumSize = New System.Drawing.Size(10, 10)
```

4.9.1.2 Split Views

Edit Control provides in-built support for horizontal and vertical splitters, which facilitates the splitting of a single document in the Edit Control into several split views so that you can work with multiple different areas of a document at the same time. A maximum of four split views are supported. However, you can also limit the user to perform either a horizontal or vertical split, only if you wish to support two views instead of four.

The vertical and horizontal splitters are always visible, by default. They can be disabled by setting the below given properties to **False**.

Edit Control Property	Description
ShowHorizontalSplitters	Gets / sets value that indicates whether horizontal splitters are visible.
ShowVerticalSplitters	Gets / sets value that indicates whether vertical splitters are visible.

The following methods can be used to split the Edit Control into two equal horizontal or vertical halves.

Edit Control Method	Description
SplitHorizontally	Splits the Edit Control into two equal horizontal halves.
SplitVertically	Splits the Edit Control into two equal vertical halves.

[C#]

```
this.editControl1.ShowHorizontalSplitters = true;
this.editControl1.ShowVerticalSplitters = true;

this.editControl1.SplitHorizontally();
this.editControl1.SplitVertically();
```

[VB .NET]

```
Me.editControl1.ShowHorizontalSplitters = True
Me.editControl1.ShowVerticalSplitters = True

Me.editControl1.SplitHorizontally()
Me.editControl1.SplitVertically()
```

Positioning

The following properties can be used to position the horizontal and vertical splitters in the Edit Control.

Edit Control Property	Description
HorizontalSplitterPosition	Gets / sets position of the horizontal splitter.
TopVerticalSplitterPosition	Gets / sets position of the top vertical splitter.
BottomVerticalSplitterPosition	Gets / sets position of the bottom vertical splitter.

[C#]

```
this.editControl1.HorizontalSplitterPosition = 220;
```

```
this.editControl1.TopVerticalSplitterPosition = 260;
this.editControl1.BottomVerticalSplitterPosition = 260;
```

[VB .NET]

```
Me.editControl1.HorizontalSplitterPosition = 220
Me.editControl1.TopVerticalSplitterPosition = 260
Me.editControl1.BottomVerticalSplitterPosition = 260
```

SplitFourQuadrants Method

The **SplitFourQuadrants** method is used to split the Edit Control into four equal parts.

[C#]

```
this.editControl1.SplitFourQuadrants();
```

[VB .NET]

```
Me.editControl1.SplitFourQuadrants()
```

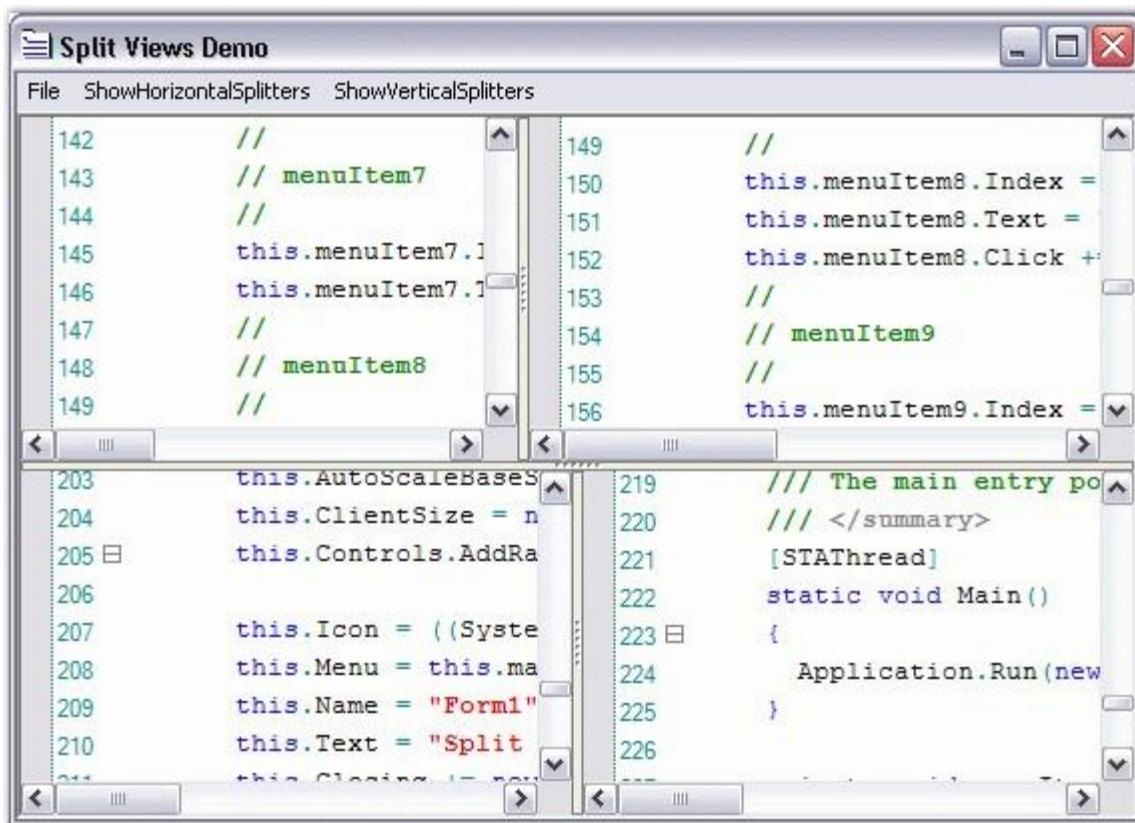


Figure 63: Edit Control Split into Four Quadrants

A sample which demonstrates Split Views is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Appearance\\SplitViewsDemo

See Also

[Scrolling Support](#)

4.9.1.3 Applying Themes

Edit Control has the ability to have Windows XP-like themed appearance. All its features like the scrollbars, splitters, control borders, outlining tooltip, intellisense popups - context tooltip, context choice, and context prompt appear themed. The XP themes support is independent of the underlying operating system, and hence the Edit Control appears themed even on Non-Windows XP systems.

The themed appearance can be provided for the Edit Control by setting the **UseXPStyle** property to **True**.

[C#]

```
this.editControl1.UseXPStyle = true;
```

[VB .NET]

```
Me.editControl1.UseXPStyle = True
```

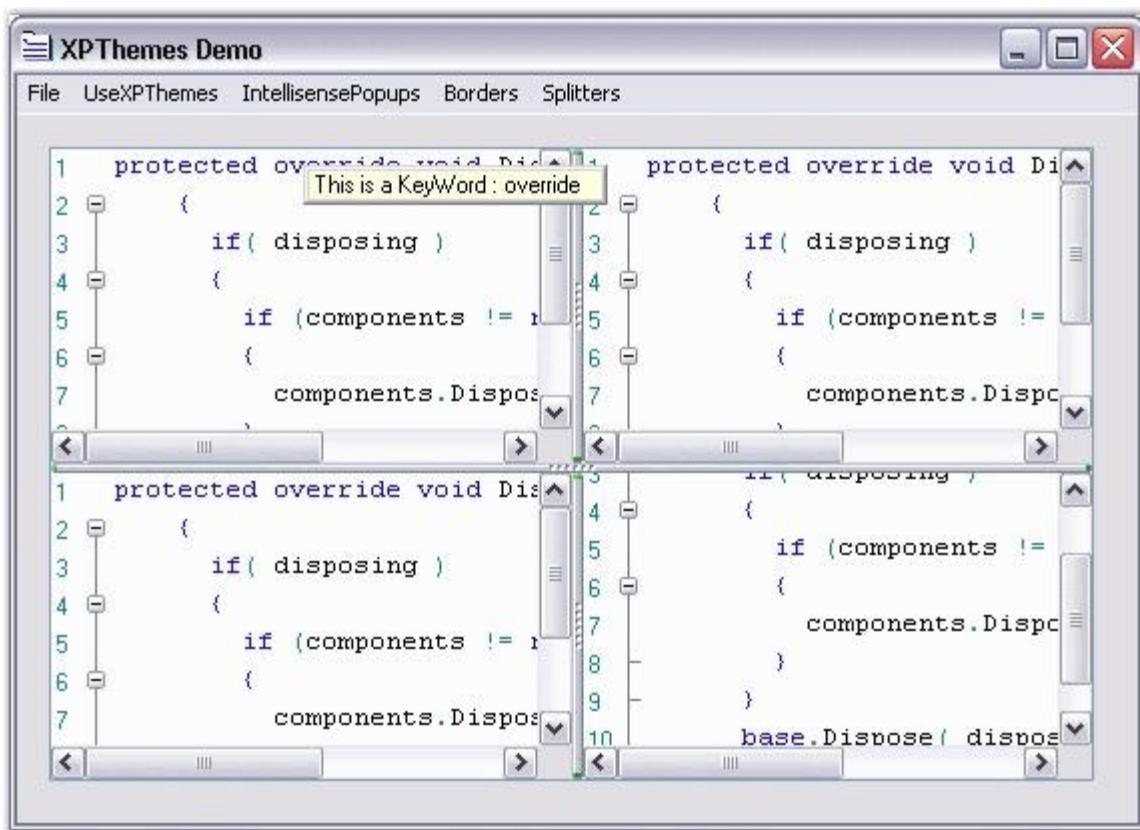


Figure 64: Edit Control using Windows XP Themes

A sample based on XP Themes is available in the below sample installation path.

`..\My Documents\Syncfusion\EssentialStudio\Version
Number\Windows>Edit.Windows\Samples\2.0\Appearance\XPThemesDemo`

4.9.1.4 Border Style

The border style for the Edit Control can be set by using the below given property.

Edit Control Property	Description
BorderStyle	<p>Gets/sets the border style of the control. The options provided are as follows:</p> <ul style="list-style-type: none"> • FixedSingle • Fixed3D • None

[C#]

```
this.editControl1.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
```

[VB .NET]

```
Me.editControl1.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle
```

4.9.1.5 Graphics Customization Settings

The following properties can be used to set the composition quality, interpolation mode and smoothing mode for images added to the Edit Control. The rendering hint can also be set for text added to the Edit Control.

Edit Control Property	Description
GraphicsCompositingQuality	Specifies image composition quality. The options provided are as follows: <ul style="list-style-type: none"> • Invalid • Default • HighSpeed • HighQuality • GammaCorrected • AssumeLinear
GraphicsInterpolationMode	Specifies the interpolation mode. The options provided are as follows: <ul style="list-style-type: none"> • Invalid • Default • Low • High • Bilinear • Bicubic • NearestNeighbor • HighQualityBilinear • HighQualityBicubic
GraphicsSmoothingMode	Specifies the smoothing mode. The options provided are as follows: <ul style="list-style-type: none"> • Invalid • Default

	<ul style="list-style-type: none">• HighSpeed• HighQuality• None• AntiAlias
GraphicsTextRenderingHint	Specifies the text hinting mode. The options provided are as follows: <ul style="list-style-type: none">• SystemDefault• SingleBitPerPixelGridFit• SingleBitPerPixel• AntiAliasGridFit• AntiAlias• ClearTypeGridFit

[C#]

```
this.editControl1.GraphicsCompositingQuality =
System.Drawing.Drawing2D.CompositingQuality.HighQuality;
this.editControl1.GraphicsInterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBilinear;
this.editControl1.GraphicsSmoothingMode =
System.Drawing.Drawing2D.SmoothingMode.HighSpeed;
this.editControl1.GraphicsTextRenderingHint =
System.Drawing.Text.TextRenderingHint.SingleBitPerPixelGridFit;
```

[VB .NET]

```
Me.editControl1.GraphicsCompositingQuality =
System.Drawing.Drawing2D.CompositingQuality.HighQuality
Me.editControl1.GraphicsInterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBilinear
Me.editControl1.GraphicsSmoothingMode =
System.Drawing.Drawing2D.SmoothingMode.HighSpeed
Me.editControl1.GraphicsTextRenderingHint =
System.Drawing.Text.TextRenderingHint.Integer.SingleBitPerPixelGridFit
```

4.9.2 Margins

This section covers the below topics:

4.9.2.1 Selection Margin

Selection Margin is a thin vertical strip along the left side of the Edit Control that enables you to select the contents of the entire line on the Edit Control, by simply clicking on the corresponding selection margin area of the line.

The **ShowSelectionMargin** property allows you to show / hide this selection margin. The following are the properties used to customize the margin.

Edit Control Property	Description
SelectionMarginForegroundColor	Gets / sets foreground color of the selection margin.
SelectionMarginBackgroundColor	Gets / sets background color of the selection margin.
SelectionMarginWidth	Sets the width of the selection margin.

[C#]

```
this.editControl1.SelectionMarginForegroundColor = Color.Gray;  
this.editControl1.SelectionMarginBackgroundColor = Color.IndianRed;  
this.editControl1.SelectionMarginWidth = 100;
```

[VB .NET]

```
Me.editControl1.SelectionMarginForegroundColor = Color.Gray  
Me.editControl1.SelectionMarginBackgroundColor = Color.IndianRed  
Me.editControl1.SelectionMarginWidth = 100
```

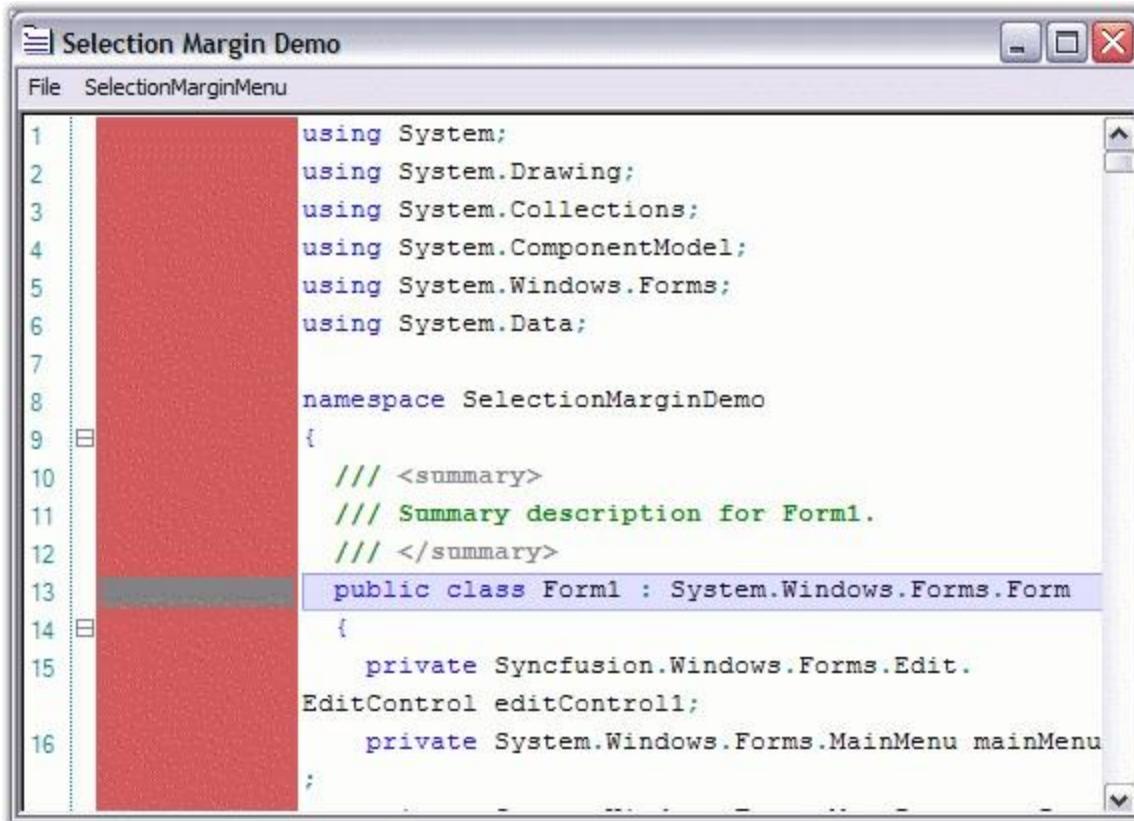


Figure 65: Selection Margin Set

Differentiating the Lines based on Actions

Edit Control supports marking the changed lines and the saved lines with different colors.

Lines that are modified after the file load or after the last file save operations are the changed lines. They are marked in yellow color, by default. Once they are saved, they will be changed to green, by default.

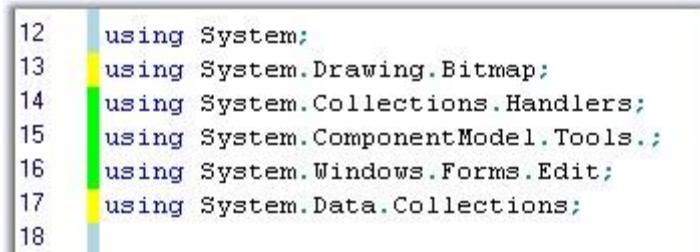
The changed lines marking feature can be enabled by setting the **MarkChangedLines** property to **True**. For this property to be visible in the Edit Control, the **SelectionMargin** property should also be enabled.

[C#]

```
this.editControl1.MarkChangedLines = true;
```

[VB .NET]

```
Me.editControl1.MarkChangedLines = True
```



A screenshot of a code editor window showing C# code. The code includes several using statements and some blank lines. A vertical color bar on the left side of the editor indicates the status of each line: green for saved changes and yellow for unsaved changes. Lines 12 through 18 are green, while lines 19 and 20 are yellow.

```
12 using System;
13 using System.Drawing.Bitmap;
14 using System.Collections.Handlers;
15 using System.ComponentModel.Tools. ;
16 using System.Windows.Forms.Edit;
17 using System.Data.Collections;
18
```

Figure 66: Saved Changes in Green and Unsaved Changes in Yellow

Refer to the Selection Margin Demo sample in the following sample installation location, for more information in this regard.

**..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\SelectionMarginDemo**

4.9.2.2 User Margin

Edit Control supports the User Margin feature, which can be used to display additional information regarding the contents in the Edit Control. Information can also be displayed on a line-by-line basis.

The User Margin feature can be turned on by setting the **ShowUserMargin** property to **True**. The user margin can be customized using the following properties.

Edit Control Property	Description
UserMarginWidth	Get / sets the width of the user margin.
UserMarginPlacement	Specifies placement of user margin.

[C#]

```
this.editControl1.UserMarginWidth = 100;

// Sets the User Margin to the Left.
this.editControl1.UserMarginPlacement =
Syncfusion.Windows.Forms.Edit.Enums.MarginPlacement.Left;
```

[VB.NET]

```
Me.editControl1.UserMarginWidth = 100  
  
// Sets the User Margin to the Left.  
Me.editControl1.UserMarginPlacement =  
Syncfusion.Windows.Forms.Edit.Enums.MarginPlacement.Left
```

Color Settings

The following properties can be used to set the background color, text color and border color of the user margin in the Edit Control.

Edit Control Property	Description
UserMarginBackgroundColor	Specifies BrushInfo object that is used when the user margin is being drawn.
UserMarginTextColor	Specifies default color of user margin text.
UserMarginBorderColor	Specifies color of the user margin border.

[C#]

```
this.editControl1.UserMarginBackgroundColor = new  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardDiagona  
l, System.Drawing.Color.Brown, System.Drawing.Color.MistyRose);  
this.editControl1.UserMarginBorderColor = Color.IndianRed;  
this.editControl1.UserMarginTextColor = Color.Green;
```

[VB.NET]

```
Me.editControl1.UserMarginBackgroundColor = New  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.BackwardDiagona  
l, System.Drawing.Color.Brown, System.Drawing.Color.MistyRose)  
Me.editControl1.UserMarginBorderColor = Color.IndianRed  
Me.editControl1.UserMarginTextColor = Color.Green
```

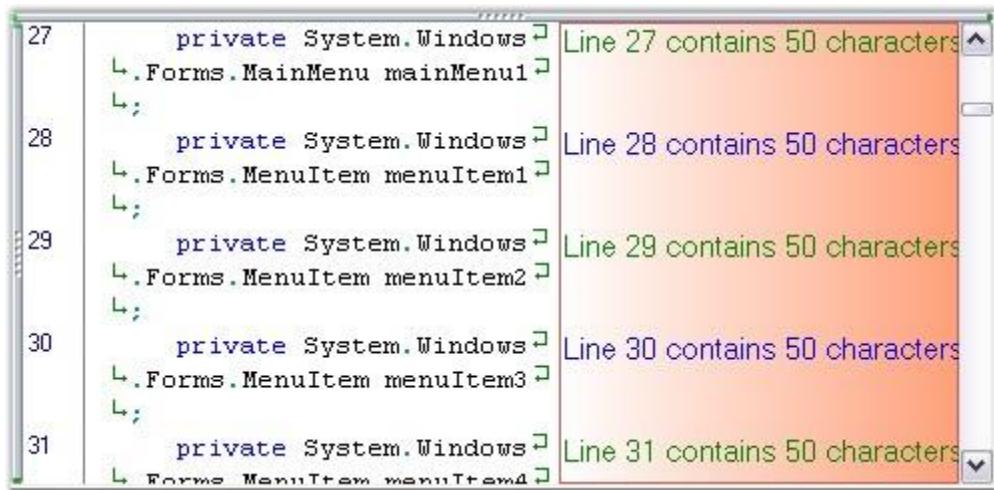


Figure 67: User Margin with Background Settings and Customized Text

It is possible to set custom text in the User Margin on a line-by-line basis by handling the **DrawUserMarginText** event of the Edit Control. Moreover, it is also possible to customize the font settings for the text of the User Margin.

[C#]

```
private void editControl1_DrawUserMarginText(object sender,
Syncfusion.Windows.Forms.Edit.DrawUserMarginTextEventArgs e)
{
    // Set text to be rendered at the user margin area.
    e.Text = "Line " + e.Line.LineIndex.ToString() + " contains " +
e.Line.LineLength.ToString() + " characters";
    // Set text font.
    e.Font = new Font("Garamond", 11);
    if(e.Line.LineIndex % 2 == 0)
    {
        // Set color of the text.
        e.Color = Color.Blue;
    }
}
```

[VB.NET]

```
Private Sub editControl1_DrawUserMarginText(ByVal sender As Object, ByVal e
As Syncfusion.Windows.Forms.Edit.DrawUserMarginTextEventArgs) Handles
EditControl1.DrawUserMarginText
    ' Set text to be rendered at the user margin area.
    e.Text = "Line " + e.Line.LineIndex.ToString() + " contains " +
e.Line.LineLength.ToString() + " characters"
```

```

' Set text font.
e.Font = New Font("Garamond", 11)
If e.Line.LineIndex Mod 2 = 0 Then
    ' Set color of the text.
    e.Color = Color.Blue
End If
End Sub

```

Refer to the User Margin Demo sample in the following sample installation location for more information in this regard.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\UserMarginDemo*

4.9.3 Background Settings

Edit Control can be displayed with a gradient background by setting the **BackgroundColor** property to the desired BrushInfo object. The following table lists some properties of the **EditControl** and their corresponding descriptions.

Edit Control Property	Description
BackgroundColor	Specifies background fill style and color.
Style	Specifies the brush style. The options provided are as follows: <i>Solid</i> <i>Pattern</i> <i>Gradient</i>
BackColor	Specifies the backcolor of the control.
ForeColor	Specifies the forecolor of the control.
PatternStyle	Specifies the pattern style. The options provided are as follows: <ul style="list-style-type: none"> • Horizontal • Vertical • ForwardDiagonal • BackwardDiagonal • Cross

	<ul style="list-style-type: none">• DiagonalCross• Percent05• Percent10• Percent20• Percent25• Percent30• Percent40• Percent50• Percent60• Percent70• Percent75• Percent80• Percent90• LightDownwardDiagonal• LightUpwardDiagonal• DarkDownwardDiagonal• DarkUpwardDiagonal• WideDownwardDiagonal• WideUpwardDiagonal• LightVertical• LightHorizontal• NarrowVertical• NarrowHorizontal• DarkVertical• DarkHorizontal• DashedDownwardDiagonal• DashedUpwardDiagonal• DashedHorizontal• DashedVertical• SmallConfetti• LargeConfetti• ZigZag• Wave• DiagonalBrick• HorizontalBrick• Weave• Plaid• Divot• DottedGrid• DottedDiamond• Shingle• Trellis• Sphere• SmallGrid
--	---

	<ul style="list-style-type: none">• SmallCheckerBoard• LargeCheckerBoard• OutlinedDiamond• SolidDiamond• None
GradientColors	<p>Specifies the gradient colors. The options provided are as follows:</p> <ul style="list-style-type: none">• ForwardDiagonal• BackwardDiagonal• Horizontal• Vertical• PathRectangle• PathEllipse <p>The first entry in this list will be the same as the backcolor property, and the last entry will be the same as the forecolor property.</p>

[C#]

```
this.editControl1.BackColor = new  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.ForwardDiagonal  
, new System.Drawing.Color[] {System.Drawing.Color.LavenderBlush,  
System.Drawing.Color.AliceBlue, System.Drawing.Color.BlanchedAlmond});
```

[VB .NET]

```
Me.editControl1.BackColor = New  
Syncfusion.Drawing.BrushInfo(Syncfusion.Drawing.GradientStyle.ForwardDiagonal  
, New System.Drawing.Color() {System.Drawing.Color.LavenderBlush,  
System.Drawing.Color.AliceBlue, System.Drawing.Color.BlanchedAlmond})
```

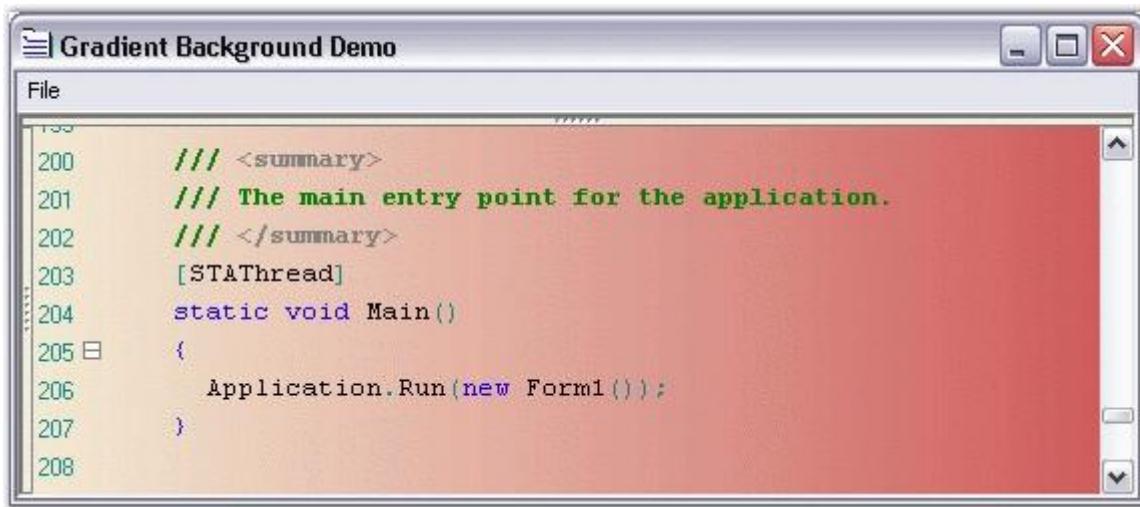


Figure 68: Edit Control with Gradient Background

A sample which demonstrates the Gradient Background feature is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio**Version Number**\\Windows\\Edit.Windows\\Samples\\2.0\\Appearance\\ GradientBackgroundDemo

Setting BackgroundColor for Specified Range of Text

The **SetBackgroundColor** method is used to set the background color for a specified range of text.

[C#]

```
this.editControl1.SetBackgroundColor(new Point(1, 1), new Point(9, 9),
Color.AliceBlue);
```

[VB .NET]

```
Me.editControl1.SetBackgroundColor(New Point(1, 1), New Point(9, 9),
Color.AliceBlue)
```

Setting Background Color for Individual Lines or Selected Blocks of Text

Edit Control allows setting custom background color for individual lines as well as for selected block of text.

You can set any desired background to a particular line or block of selection, as explained below.

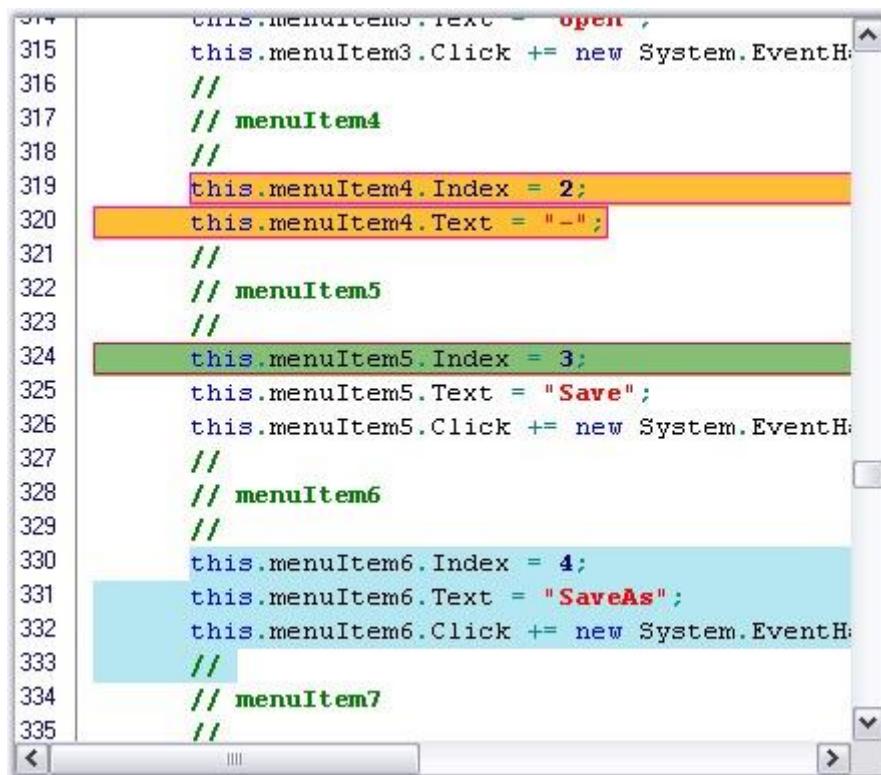
- Register a backcolor format with the Edit Control by using its **RegisterBackColorFormat** method, with appropriate values for BackgroundColor, ForegroundColor and HatchStyle parameters.
- Set the background color to the entire line or just the selected text by using the **SetLineBackColor** and **SetSelectionBackColor** methods respectively.

[C#]

```
// Register a backcolor format with EditControl.  
this.editControl1.RegisterBackColorFormat(Color.Aquamarine, Color.Beige,  
System.Drawing.Drawing2D.HatchStyle.Cross, true);  
  
// Set the background for the entire line of text.  
this.editControl1.SetLineBackColor(editControl1.CurrentLine, true, format);  
  
// Set the background for the selected block of text.  
this.editControl1.SetSelectionBackColor(format);
```

[VB .NET]

```
' Register a backcolor format with EditControl.  
Me.editControl1.RegisterBackColorFormat(Color.Aquamarine, Color.Beige,  
System.Drawing.Drawing2D.HatchStyle.Cross, True)  
  
' Set the background for the entire line of text.  
Me.editControl1.SetLineBackColor(editControl1.CurrentLine, true, format)  
  
' Set the background for the selected block of text.  
Me.editControl1.SetSelectionBackColor(format)
```



```

314     this.menuItem3.Text = "open";
315     this.menuItem3.Click += new System.EventHandler(this.MenuItem3_Click);
316     //
317     // menuItem4
318     //
319     this.menuItem4.Index = 2;
320     this.menuItem4.Text = "-";
321     //
322     // menuItem5
323     //
324     this.menuItem5.Index = 3;
325     this.menuItem5.Text = "Save";
326     this.menuItem5.Click += new System.EventHandler(this.MenuItem5_Click);
327     //
328     // menuItem6
329     //
330     this.menuItem6.Index = 4;
331     this.menuItem6.Text = "SaveAs";
332     this.menuItem6.Click += new System.EventHandler(this.MenuItem6_Click);
333     //
334     // menuItem7
335     //

```

Figure 69: Background Color and Border set for Text



Note: Refer the [Text Border](#) topic to know how to set the border for the text.

Removing Background Color for Individual Lines or Selected Blocks of Text

The following methods can be used to set the background color for individual lines or selected blocks of text.

Edit Control Method	Description
RemoveLineBackColor	Removes line back color.
RemoveSelectionBackColor	Removes background coloring from the selected text.

[C#]

```

// Removes line back color.
this.editControl1.RemoveLineBackColor(4);

```

```
// Removes background coloring from the selected text.  
this.editControl1.RemoveSelectionBackColor();
```

[VB .NET]

```
' Removes line back color.  
Me.editControl1.RemoveLineBackColor(4)  
  
' Removes background coloring from the selected text.  
Me.editControl1.RemoveSelectionBackColor()
```

A sample which demonstrates Text Highlighting is available in the following sample location.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\TextHighlightingDemo*

See Also

[Line Numbers and Current Line Highlighting](#)

4.9.4 Font Customization

The font customization in the Edit Control works slightly different from the regular text processing the control. The font customization is done only at the **Formats** level, and not at a word level or selected text level. Edit Control is more of a text parsing / syntax highlighting control, and less of a text editing control. Edit Control supports customization of fonts both through the configuration file and dynamically through a run-time **Formats Editor** dialog.

The Edit Control supports customization of fonts through the configuration file, as shown in the below code snippet.

[XML]

```
<format name="Text" Font="Courier New, 10pt" FontColor="Black" />  
<format name="SelectedText" Font="Courier New, 10pt" BackColor="Highlight"  
FontColor="HighlightText" />  
<format name="String" Font="Courier New, 10pt, style=Bold" FontColor="Red" />  
<format name="Whitespace" Font="Courier New, 10pt" FontColor="Black" />  
<format name="Operator" Font="Courier New, 10pt" FontColor="DarkCyan" />
```

```
<format name="Number" Font="Courier New, 10pt, style=Bold" FontColor="Navy"
/>
```

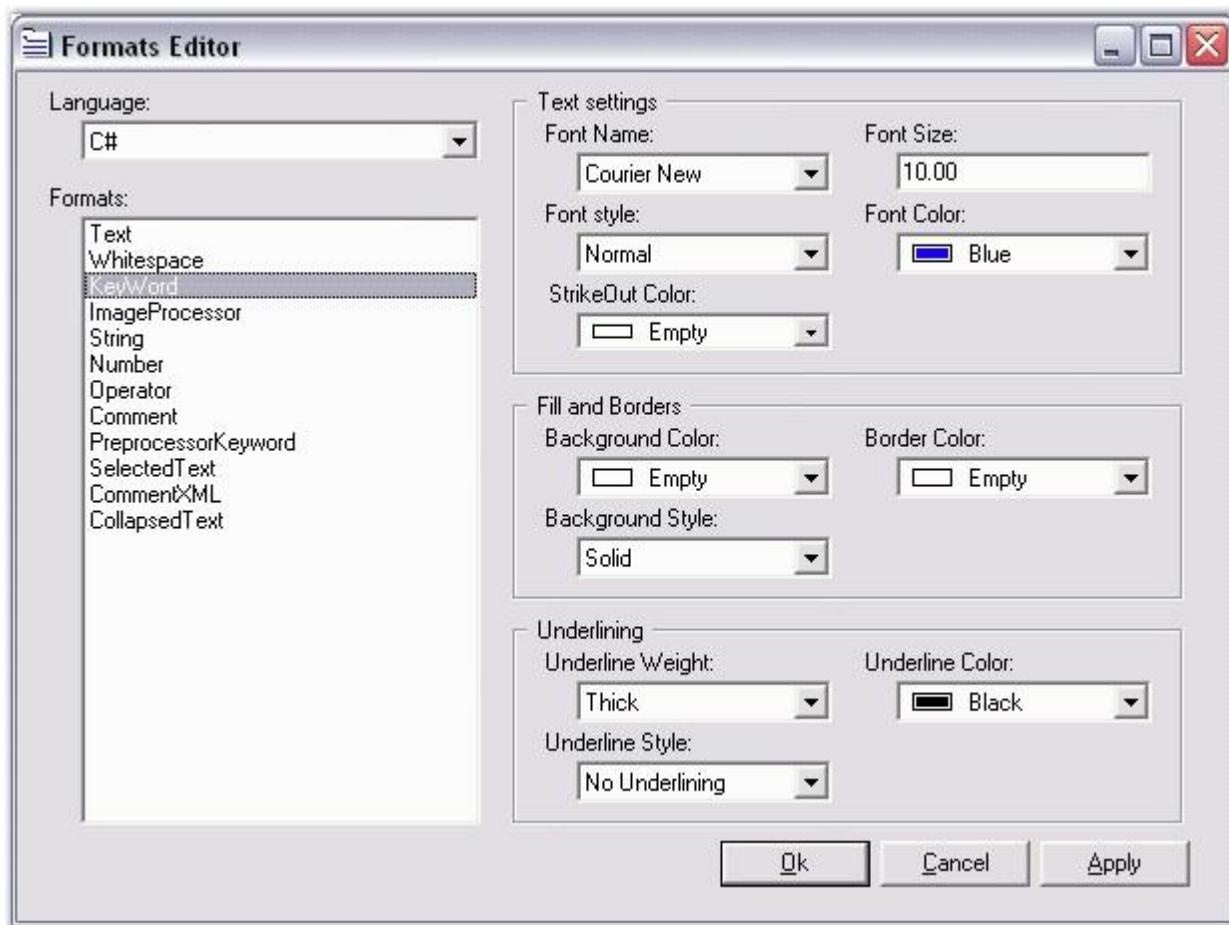


Figure 70: Formats Editor

The Edit Control supports font customization at run time through the use of the **frmFormatsConfig** dialog box which consists of three smaller controls like the **ControlFormatSettings**, **ControlFormatsList**, and the **ControlLanguageSelector**. The **ControlFormatSettings** dialog box contains the actual controls to customize all the rendering settings of the selected Format, including font settings. The **ControlFormatsList** dialog consists of the list of currently existing formats in the configuration file. Also, it provides support to create new formats or delete existing ones. The **ControlLanguageSelector** dialog has a Combo Box containing the list of configuration languages supported by the Edit Control. The list gets updated when a new configuration language is added or an existing one is removed. The following code illustrates how you can hook up these dialogs to the Edit Control.

```
[C#]

this.controlLanguageSelector1.EditControl = this.editControl1;
this.controlFormatsList1.EditControl = this.editControl1;
```

```
this.controlFormatsList1.LanguageSelector = this.controlLanguageSelector1;
this.controlFormatsSettings1.FormatsSelector = this.controlFormatsList1;

// Shows the font customization dialog.
this.editControl1.ShowFormatsCustomizationDialog();
```

[VB.NET]

```
Me.controlLanguageSelector1.EditControl = Me.editControl1
Me.controlFormatsList1.EditControl = Me.editControl1

Me.controlFormatsList1.LanguageSelector = Me.controlLanguageSelector1
Me.controlFormatsSettings1.FormatsSelector = Me.controlFormatsList1

// Shows the font customization dialog.
this.editControl1.ShowFormatsCustomizationDialog();
```

Refer to the Font Customization Demo sample for more information in this regard.

***..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor
Functions\\FontCustomizationDemo***

4.9.5 Single Line Mode

Edit Control can be operated in a single line mode much like a Windows Forms Text Box, by setting its **Multiline** property to **False**. This enables you to have a simple TextBox-like control, but with all the powerful features of Edit Control like syntax highlighting, selection, underlining, and so on.

You can turn on the single line mode of the Edit Control by setting the **SingleLineMode** property to **True**.



Note: The **SingleLineMode** is intended for use, only when the Edit Control contains small amounts of text data in it. Using it in a scenario where the Edit Control has a huge file loaded into it, may lead to poor performance.

```
static void Main(){ Application.Run(new Form1());}
```

Figure 71: Edit Control with SingleLineMode Turned On

Refer to the Single Line Mode Demo sample in the following sample installation location, for more information in this regard.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Appearance\\SingleLineModeDemo*

4.9.6 Customizable Find Dialog

Essential Edit now enables you to create a new find dialog by inheriting Essential Edit's find dialog. You can customize the **Find Dialog** by changing the properties and triggers the events of the buttons such as **Find**, **Mark All** and **Close**. You can also easily localize the captions of the controls in the **Find dialog**.

Creating a Class for Own Find Dialog

Create a class for own find dialog that inherits the **frmFindDialog** class.

The following code illustrates this.

[C#]

```
//Inherits the frmFindDialog
public class FindDialogExt :
Syncfusion.Windows.Forms.Edit.Dialogs.frmFindDialog
```

[VB .NET]

```
'Inherits the frmFindDialog.
Public Class FindDialogExt
    Inherits Syncfusion.Windows.Forms.Edit.Dialogs.frmFindDialog
End Class
```

FindComplete Event

This event occurs in **FindNext()** when search reaches the starting point of the search before the message box displays.

The event handler receives an argument of **FindCompleteEventArgs**. This argument class sets the text for message box. Users can localize this text.

[C#]

```
// Handle the FindComplete event.  
this.FindComplete += new  
EventHandler<FindCompleteEventArgs>(FindDialogExt_FindComplete);  
  
//Set the value for message box for when search reaches the  
starting point of search  
void FindDialogExt_FindComplete(object sender,  
frmFindDialog.FindCompleteEventArgs e)  
{  
    //Arabic text as message(localize)  
    if (messageString != string.Empty)  
        e.Message = "انتهى";  
    else  
        e.Message = "Find reached the starting point of the  
search.";  
}
```

[VB.NET]

```
' Handle the FindComplete event.  
  
Private Me.editControl1.FindComplete += New EventHandler(Of  
FindCompleteEventArgs)(AddressOf FindDialogExt_FindComplete)  
  
'Set the value for message box for when search reaches the starting point  
of search  
  
Private Sub FindDialogExt_FindComplete(ByVal sender As Object, ByVal e As  
frmFindDialog.FindCompleteEventArgs)  
    'Arabic text as message (localize)  
    If messageString <> String.Empty Then  
        e.Message = "انتهى"  
    Else  
        e.Message = "Find reached the starting point of the  
search."  
    End If  
End Sub
```

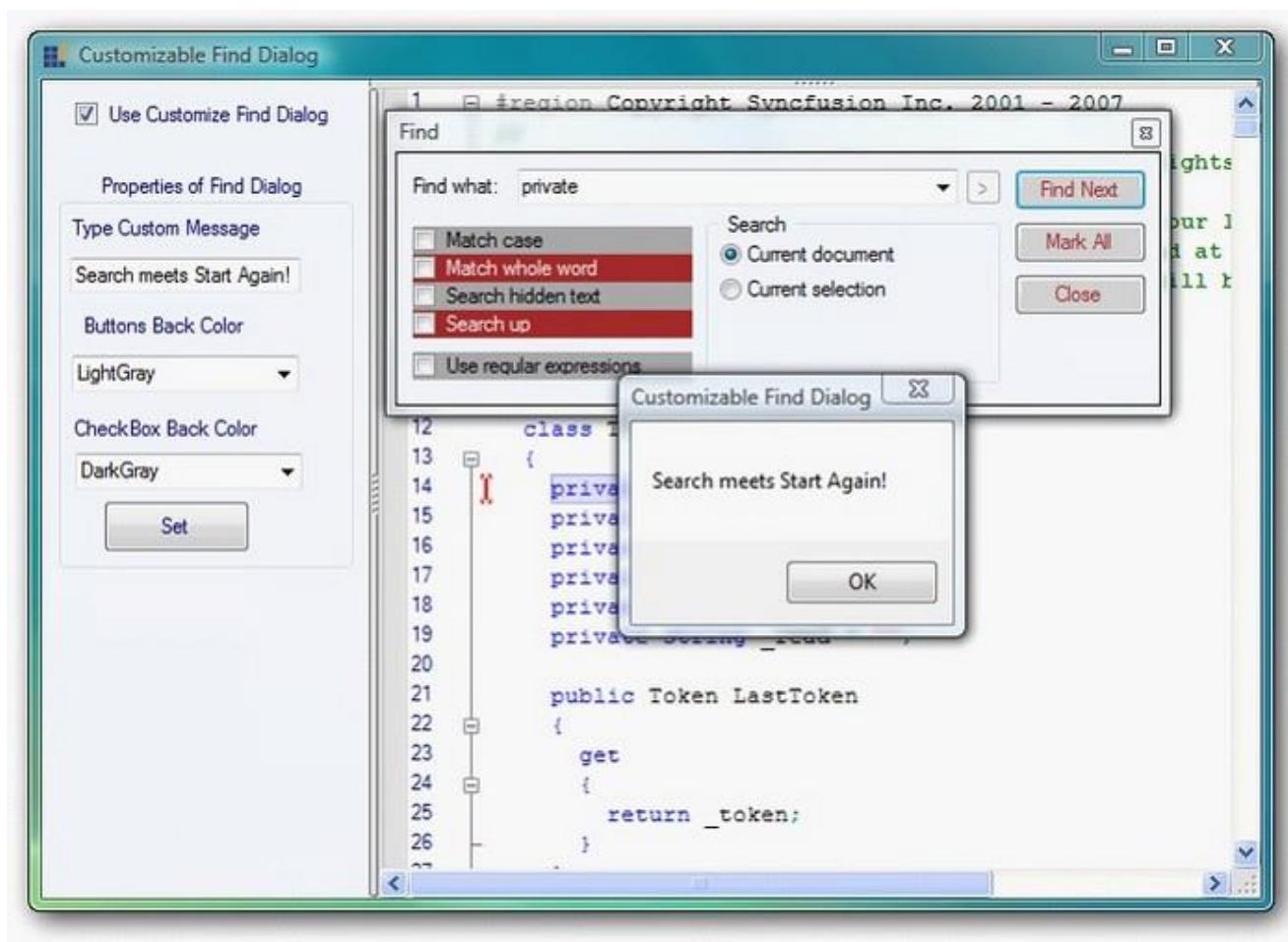
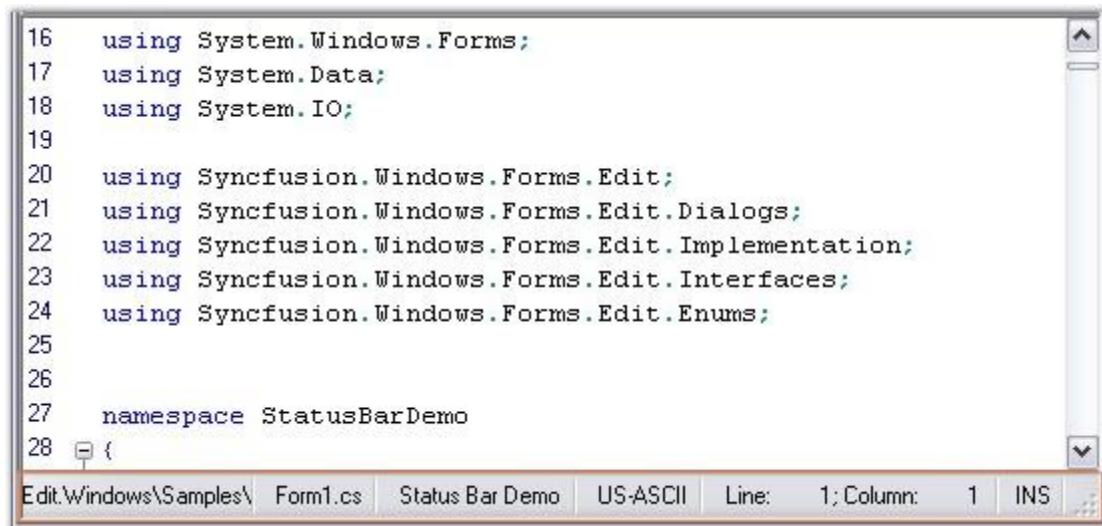


Figure 72: Customized Find Dialog

4.10 Status Bar

Essential Edit provides support to include a built-in Status Bar at the bottom of the control with different panels displaying different information. The built-in panels are as follows.

- TextPanel
- StatusPanel
- EncodingPanel
- FileNamePanel
- CoordsPanel
- InsertPanel



The screenshot shows a code editor window with the following code:

```

16  using System.Windows.Forms;
17  using System.Data;
18  using System.IO;
19
20  using Syncfusion.Windows.Forms.Edit;
21  using Syncfusion.Windows.Forms.Edit.Dialogs;
22  using Syncfusion.Windows.Forms.Edit.Implementation;
23  using Syncfusion.Windows.Forms.Edit.Interfaces;
24  using Syncfusion.Windows.Forms.Edit.Enums;
25
26
27  namespace StatusBarDemo
28  {

```

The status bar at the bottom of the editor window displays the following information:

- Edit\Windows\Samples\
- Form1.cs
- Status Bar Demo
- US-ASCII
- Line: 1; Column: 1
- INS

Figure 73: Status Bar and Status Bar Panels in Edit Control

In addition to the above information, any custom text can also be displayed in the Status Bar Panels.

Status Bar Settings

The **StatusBarSettings** property consists of the below given sub properties, which can be used to customize the appearance and visibility of the status bar and its panels.

StatusbarSettings Property	Description
TextPanel	Specifies StatusBaPanelSettings object for Text panel.
StatusPanel	Specifies StatusBaPanelSettings object for Status panel.
EncodingPanel	Specifies StatusBaPanelSettings object for Encoding panel.
FileNamePanel	Specifies StatusBaPanelSettings object for FileName panel.
CoordsPanel	Specifies StatusBaPanelSettings object for Coords panel.
InsertPanel	Specifies StatusBaPanelSettings object for Insert panel.
Panels	Gets the list of status bar panels settings.
StatusBar	Gets underlying status bar.
GripVisibility	Gets / sets visibility of status bar sizing grip. The options provided are as follows:

- | | |
|--|--|
| | <ul style="list-style-type: none">• Smart• Visible• Hidden |
|--|--|

[C#]

```
// Set the visibility of the statusbar sizing grip.  
this.editControl1.StatusBarSettings.GripVisibility =  
Syncfusion.Windows.Forms.Edit.Enums.SizingGripVisibility.Visible;
```

[VB .NET]

```
' Set the visibility of the statusbar sizing grip.  
Me.editControl1.StatusBarSettings.GripVisibility =  
Syncfusion.Windows.Forms.Edit.Enums.SizingGripVisibility.Visible
```



Figure 74: Sizing Gripper in the Status Bar

Visibility Settings

The StatusBar feature in Edit Control can be turned on by setting the **StatusBarSettings.Visible** property to **True**. By default, this property is set to **False**. The individual Status Bar Panels can be optionally shown / hidden by using the **Visible** property corresponding to the respective panel.

[C#]

```
this.editControl1.StatusBarSettings.GripVisibility =  
Syncfusion.Windows.Forms.Edit.Enums.SizingGripVisibility.Visible;  
  
// Shows the built-in statusbar.  
this.editControl1.StatusBarSettings.Visible = true;  
  
// Enable the TextPanel in the StatusBar.  
this.editControl1.StatusBarSettings.TextPanel.Visible = true;
```

[VB .NET]

```
Me.editControl1.StatusBarSettings.GripVisibility =  
Syncfusion.Windows.Forms.Edit.Enums.SizingGripVisibility.Visible
```

```
// Shows the built-in statusbar.  
Me.editControl1.StatusBarSettings.Visible = True  
  
' Enable the TextPanel in the StatusBar.  
Me.editControl1.StatusBarSettings.TextPanel.Visible = True
```

A sample which demonstrates the StatusBar feature is available in the below sample installation path.

..\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Windows\\Edit.Windows\\Samples\\2.0\\Advanced Editor Functions\\StatusBarDemo

4.11 Printing

The Edit Control provides complete support for printing its contents. You can either print the entire document, just the current page, specific pages, or selected text. The printing implementation is very similar to the one available in standard applications such as MS Office or Visual Studio.NET. A Print dialog box provides options to customize the printer settings, number of copies, the pages to be printed, and so on. Edit Control also includes a Print Preview feature that allows you to view the document before printing. Moreover, features like customizable header and footer are also available in Essential Edit.

In brief, the printing functionality of the Edit Control supports the following features.

- Print Preview
- Custom Header and Footer Text
- Document Name
- Page Numbers
- [Content Dividers](#)
- [Wordwrap](#)
- Color Printing to preserve Syntax Highlighting
- Selected Text Printing
- [Line Numbers](#)
- Printing a Specific Page or Set of Pages
- Printing Entire Document
- Creating a Printer Document
- Current Page Printing
- Printer Dialog
- Outlining Blocks

You can invoke the Print dialog box by using the **Print** method of the Edit Control, as shown in the below code snippet.

[C#]

```
// Invoke the print dialog.  
this.editControl1.Print();
```

[VB.NET]

```
' Invoke the print dialog.  
Me.editControl1.Print()
```

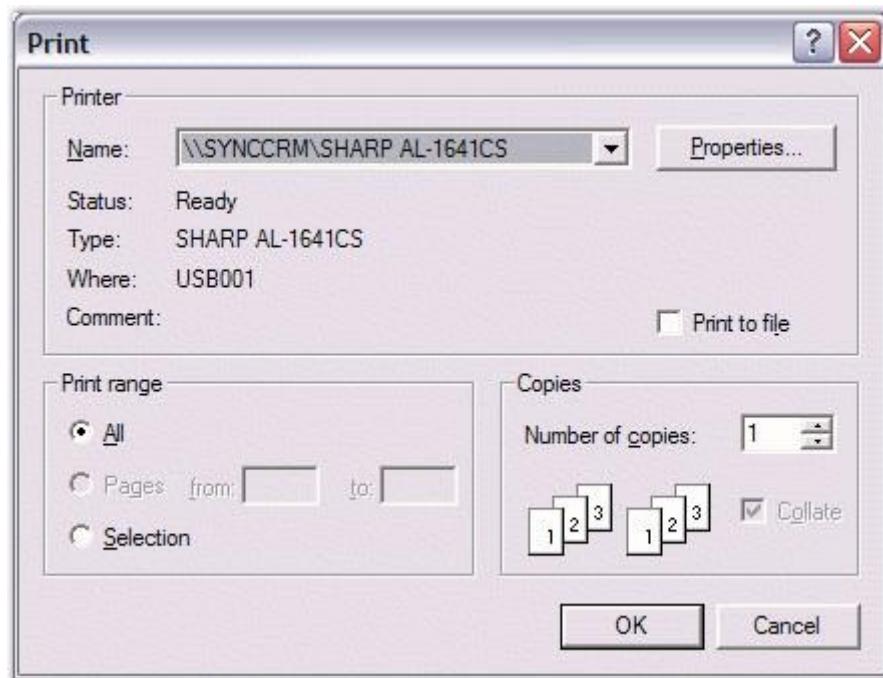


Figure 75: Print Dialog Box

Use the **PrintPreview** method to view the contents of the Edit Control before they are printed.

[C#]

```
// View the contents of the Edit Control before printing.  
this.editControl1.PrintPreview();
```

[VB.NET]

```
' View the contents of the Edit Control before printing.  
Me.editControl1.PrintPreview()
```

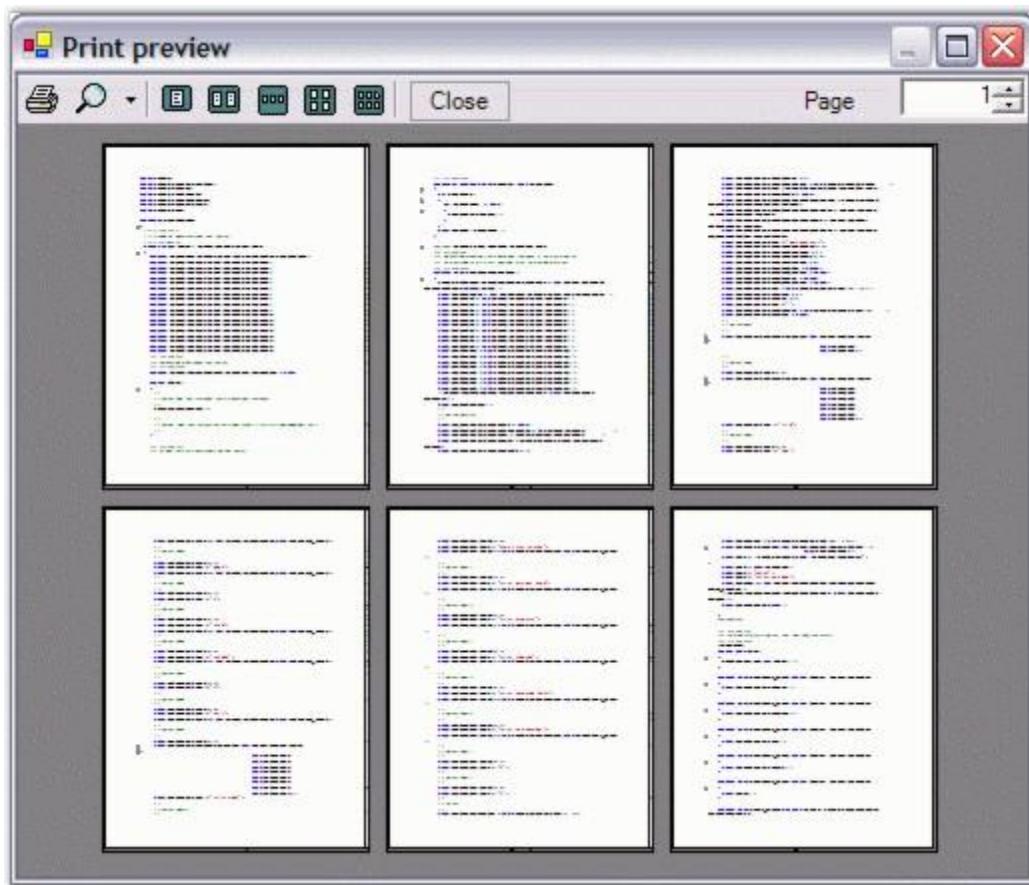


Figure 76: Print Preview

Specifying Printing Options

The following methods allow you to specify the options for printing.

Edit Control Method	Description
PrintCurrentPage	Prints current page on default printer.
PrintNoDialog	Prints entire document on default printer.
PrintSelection	Prints selected area on default printer.
PrintPages	Prints the pages in the specified range.

[C#]

```
// Print the current page.  
this.editControl1.PrintCurrentPage();  
  
// Print the entire document.  
this.editControl1.PrintNoDialog();  
  
// Print the selected area.  
this.editControl1.PrintPages(1, 10);  
  
// Print the pages in the specified range.  
this.editControl1.PrintSelection();
```

[VB.NET]

```
' Print the current page.  
Me.editControl1.PrintCurrentPage()  
  
' Print the entire document.  
Me.editControl1.PrintNoDialog()  
  
' Print the selected area.  
Me.editControl1.PrintPages(1, 10)  
  
' Print the pages in the specified range.  
Me.editControl1.PrintSelection()
```

Customized Header Footer

Header and Footer can be shown / hidden while printing the document by using the **PageHeaderAndFooterVisible** property.

The following properties are used to print the contents of the editor, the document name as the header, and the page number as footer.

Edit Control Property	Description
PrintDocument	Gets print document, that can be used to print the contents of the editor.
PrintDocumentName	Gets / sets value indicating whether the document name should be printed.
PrintPageNumber	Gets / sets value indicating whether the page number should be printed.

Users can also specify their desired text in the header and footer by handling the **PrintHeader** and **PrintFooter** events.

The default text in the header and footer is the fully qualified path of the file including the file name and page number respectively.

Edit Control Property	Description
PrintHeader	Occurs when page header is printed.
PrintFooter	Occurs when page footer is printed.

[C#]

```
private void editControl1_PrintHeader(object sender,
Syncfusion.Windows.Forms.Edit.PrintHeadlineEventArgs e)
{
    // Set the desired text in the header. The default text in the header is
    // the full path and the name of the file.
    e.Text = "This is the header";
}

private void editControl1_PrintFooter(object sender,
Syncfusion.Windows.Forms.Edit.PrintHeadlineEventArgs e)
{
    // Set desired text in the footer. The default text in the footer is the
    // page number.
    e.Text = "This is the footer";
}
```

[VB .NET]

```
Private Sub editControl1_PrintHeader(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.PrintHeadlineEventArgs) Handles
EditControl1.PrintHeader
    ' Set the desired text in the header. The default text in the header is
    ' the full path and the name of the file.
    e.Text = "This is the header"
End Sub 'editControl1_PrintHeader

Private Sub editControl1_PrintFooter(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.PrintHeadlineEventArgs) Handles
EditControl1.PrintFooter
    ' Set desired text in the footer. The default text in the footer is the
    ' page number.

```

```
e.Text = "This is the footer"  
End Sub
```

The following image shows a typical page with a header and footer in Print Preview mode.

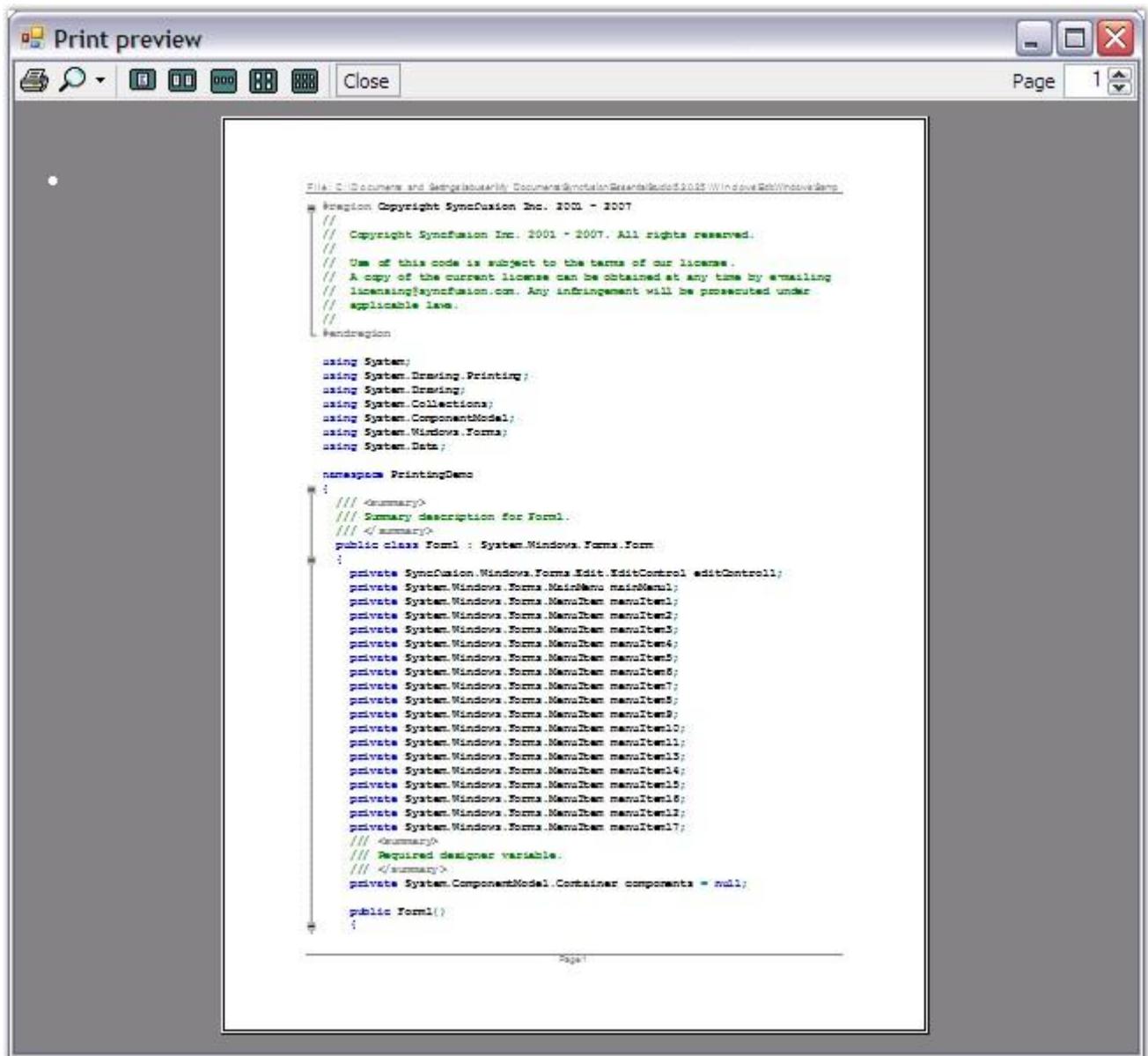


Figure 77: Preview of Header and Footer

Page Border Settings

Edit Control provides the following methods to display page borders for the Edit Control.

Edit Control Method	Description
SetPageBorder	Sets parameters of border that's drawn in page preview.
RemovePageBorder	Removes border drawing in page preview.

[C#]

```
// Set the page border.  
this.editControl1.SetPageBorder(Syncfusion.Windows.Forms.Edit.Enums.FrameBorderStyle.DashDot, Color.Red,  
Syncfusion.Windows.Forms.Edit.Enums.BorderWeight.Bold);  
  
// Remove the page border.  
this.editControl1.RemovePageBorder();
```

[VB .NET]

```
' Set the page border.  
Me.editControl1.SetPageBorder(Syncfusion.Windows.Forms.Edit.Enums.FrameBorderStyle.DashDot, Color.Red,  
Syncfusion.Windows.Forms.Edit.Enums.BorderWeight.Bold)  
  
' Remove the page border.  
Me.editControl1.RemovePageBorder()
```

Refer to the Printing Demo sample for more information in this regard.

***..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Printing Support\\PrintingDemo***

4.12 Performance

This section discusses the fast load mode of the Edit Control.

Quick File Loading

The Edit Control loads files extremely quick, and hence its performance is unmatched by any of our competitors or the Visual Studio.NET IDE.

The **ConvertOnLoad** property of the Edit Control should be set to **False** to enable the fast load mode. By default, this property is set to **True**.

[C#]

```
// Enable the fast load mode.  
this.editControl1.ConvertOnLoad = false;
```

[VB .NET]

```
' Enable the fast load mode.  
Me.editControl1.ConvertOnLoad = False
```

4.13 Localization and Globalization

In the age of globalization, the market for all goods become more and more internationalized, enforcing the need to provide information in a variety of languages. This is especially true for the software market, where the product itself consists of exclusive localizable information. Translation and customization of software involves a variety of specialists such as programmers, translators, localization engineers, quality assurance (QA) specialists and project managers. International users of computer software expect their software to talk to them in their own language. This is not only a matter of convenience or of national pride, but a matter of productivity. Users who understand a product fully will be more skilled in handling it, and will avoid making mistakes. So, users will prefer applications in their language and adapted to their cultural environment.

The following images shows how the Find dialog in Edit Control is localized to French and Spanish.

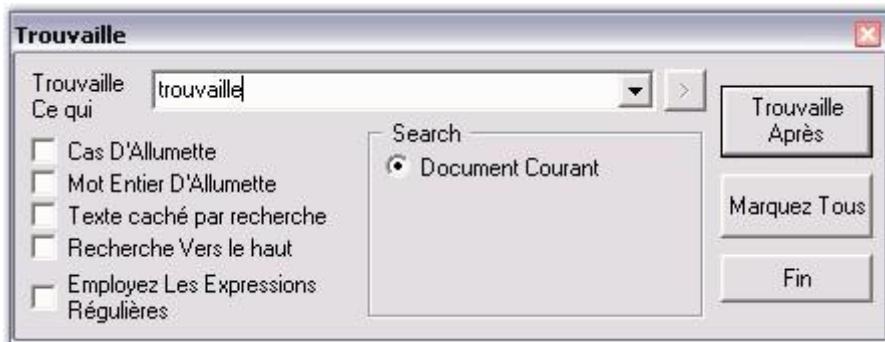


Figure 78: Find dialog localized to French

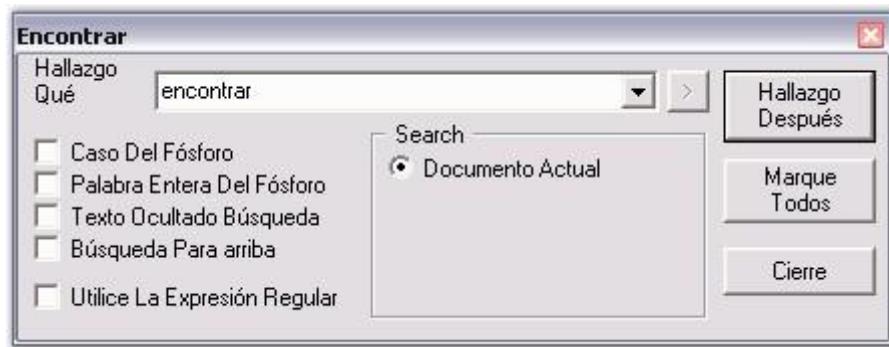


Figure 79: Find dialog localized to Spanish

Edit Control supports complete localization of all the dialog boxes associated with it to any desired language. The Neutral Resource files for the Edit Control is available in the directory - `\Edit.Windows\Src\LocalizationSet`. Follow the steps below to localize the Neutral Resources files.

1. The neutral resource of every Syncfusion Edit Control is present in the Localization folder of Edit.Windows source code. Assume that `C:\Program Files\` is the installation path for the Syncfusion components. Here is the path:

`C:\Program Files\Syncfusion\Essential Studio\x.x.x\Windows>Edit.Windows\Src\LocalizationSet v1.1`

2. Inside the LocalizationSet folder, there will be a number of resource files corresponding to the various dialog boxes of the Edit Control package. These resources will contain the form representation of English (Default & Neutral) culture.
3. Open WinRes from the following location:

`C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\bin\winRes.exe`

4. WinRes is used to work with the Windows Forms resources. The ResEdit tool cannot be used to edit Windows Forms resources because it can be used to work with image and string based resources only.
5. Open the resources using the WinRes utility, and replace the English strings with the culture equivalent.

For example, the following figure shows the `Syncfusion.Windows.Forms.Edit.Dialogs.frmFindDialog.resources` file that is opened in the WinRes tool, showing strings in German (strings are converted using some language converter).

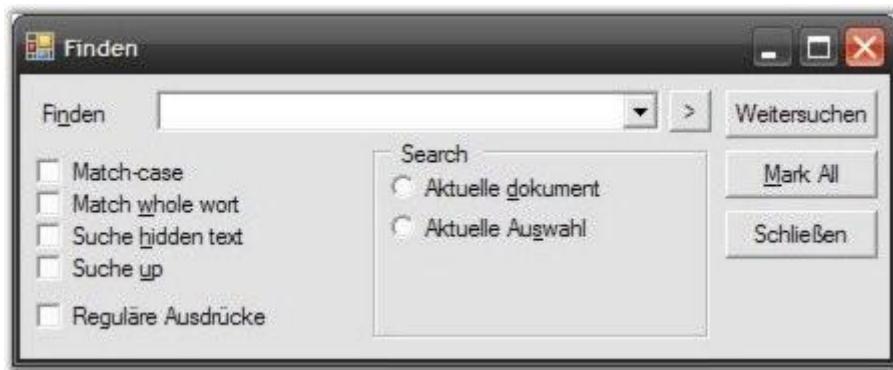


Figure 80: Replacing strings in English to German by using the WinRes Utility

6. Click File --> SaveAs, and select the Culture to be localized (in this case German - German). Now, a new resource file with the name Syncfusion.Windows.Forms.Edit.Dialogs.frmFindDialog.de-DE.resources will be added to the source path.
7. Similarly, repeat the process for other resources and save them.
8. Now, in the Visual Studio.NET 2005 Command Prompt, type the following command, and then press ENTER. Make sure that you have the sf.publicsnk file from the Localization folder.

```
al /t:lib /culture:de-DE /out:Syncfusion.Edit.Windows.resources.dll  
/v:1.0.0.0 /delay+ /keyf:sf.publicsnk /embed:  
Syncfusion.Windows.Forms.Edit.Dialogs.frmFindDialog.de-DE.resources
```

9. Press ENTER. Make sure that you have the sf.publicsnk file from the Localization folder.
10. The version (1.0.0.0) that you specify for these DLLs in the above al command, should be based on the SatelliteContractVersionAttribute setting in the product AssemblyInfo.cs file in Edit source. Note that the incorrect version won't localize the assembly properly.
11. On successful execution, an Assembly file named Syncfusion.Edit.Windows.resources.dll will be created.
12. Finally, mark this satellite DLL for verification skipping (since it is has not been signed with the same strong-name as the product assembly), as follows.

```
sn -Vr Syncfusion.Edit.Windows.resources.dll
```

13. Now, drop this DLL into an appropriate subdirectory under your EXE's directory (bin\Debug\), based on the naming conventions that are enforced in .NET. You should create a folder named "de-DE" under bin/Debug if this DLL contains resources from the German (Germany) culture.

14. Finally, you should specify your application to fetch the German resources during run time. To change the UI culture of the current thread, add the following code in the Forms constructor before the InitializeComponent().

```
System.Threading.Thread.CurrentThread.CurrentCulture = new  
System.Globalization.CultureInfo("de-DE");
```

15. Now, run your application that contains the Syncfusion Edit Control, and open the Find dialog box. You will see the dialog box in German.

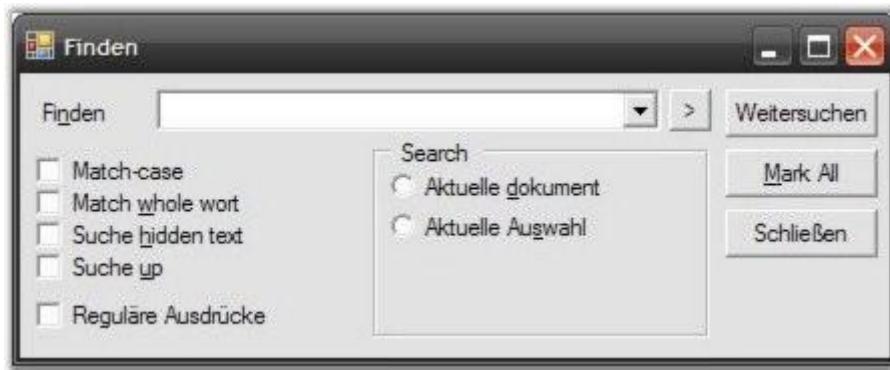


Figure 81: Find dialog box localized to German

4.14 Edit Control Events

This section discusses various events handled for the Edit control. The events are listed below:

4.14.1 CanUndoRedoChanged Event

This event occurs when the **CanUndoRedo** state is changed. The **CanUndo** and **CanRedo** properties indicate whether it is possible to undo and redo the actions in Edit Control respectively.

The event handler receives an argument of type **EventArgs**.

```
[C#]  
  
private void editControl1_CanUndoRedoChanged(object sender, EventArgs e)  
{  
    Console.WriteLine(" CanUndoRedoChanged event is raised ");  
}
```

[VB.NET]

```
Private Sub editControl1_CanUndoRedoChanged(ByVal sender As Object, ByVal e As EventArgs)
    Console.WriteLine(" CanUndoRedoChanged event is raised ")
End Sub
```

4.14.2 Closing Event

This event is discussed in the [Saving and Cancelling Changes](#) topic.

4.14.3 Code Snippet Events

This section discusses the below given code snippet events.

4.14.3.1 CodeSnippetActivating Event

This event occurs when the code snippet is to be activated.

The event handler receives an argument of type **CancellableCodeSnippetsEventArgs**. The following CancellableCodeSnippetsEventArgs members provide information, specific to this event.

Member	Description
Cancel	Indicates whether action has to be cancelled.
CodeSnippet	Code snippet that is currently activated.

[C#]

```
private void editControl1_CodeSnippetActivating(object sender,
Syncfusion.Windows.Forms.Edit.CancellableCodeSnippetsEventArgs e)
{
    // The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetActivating event is raised ");
```

```
}
```

[VB .NET]

```
Private Sub editControl1_CodeSnippetActivating(ByVal sender As Object, ByVal e As Syncfusion.Windows.Forms.Edit.CancellableCodeSnippetsEventArgs)
    ' The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetActivating event is raised ")
End Sub
```

4.14.3.2 **CodeSnippetDeactivating** Event

This event occurs when the code snippet is to be deactivated.

The event handler receives an argument of type **CodeSnippetsEventArgs**. The following **CodeSnippetsEventArgs** member provides information, specific to this event.

Member	Description
CodeSnippet	Code snippet that is currently activated.

[C#]

```
private void editControl1_CodeSnippetDeactivating(object sender,
Syncfusion.Windows.Forms.Edit.CodeSnippetsEventArgs e)
{
    // The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetDeactivating event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_CodeSnippetDeactivating(ByVal sender As Object,
ByVal e As Syncfusion.Windows.Forms.Edit.CodeSnippetsEventArgs)
    ' The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetDeactivating event is raised ")
End Sub
```

4.14.3.3 CodeSnippetTemplateTextChanging Event

This event is raised when the text of the code snippet template member is to be changed.

The event handler receives an argument of type **CodeSnippetTemplateTextChangingEventArgs**. The following **CodeSnippetTemplateTextChangingEventArgs** members provide information, specific to this event.

Member	Description
Cancel	Indicates whether action has to be canceled.
CodeSnippet	Code snippet that is currently activated.
NewText	New text.
TemplateMemberName	Name of template member that is to be changed.

[C#]

```
// Change the text of all template members with defined name of currently
activated code snippet.
this.editControl1.ChangeSnippetTemplateText(" old member name", " new text");

// Handle the CodeSnippetTemplateTextChanging event.
this.editControl1.CodeSnippetTemplateTextChanging+=new
Syncfusion.Windows.Forms.Edit.CodeSnippetTemplateTextChangingEventHandler(edit
Control1_CodeSnippetTemplateTextChanging);

private void editControl1_CodeSnippetTemplateTextChanging(object sender,
Syncfusion.Windows.Forms.Edit.CodeSnippetTemplateTextChangingEventArgs e)
{
    // The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetTemplateTextChanging event is raised ");
}
```

[VB .NET]

```
' Change the text of all template members with defined name of currently
activated code snippet.
Me.editControl1.ChangeSnippetTemplateText(" old member name", " new text")

' Handle the CodeSnippetTemplateTextChanging event.
Me.editControl1.CodeSnippetTemplateTextChanging+=New
```

```

Syncfusion.Windows.Forms.Edit.CodeSnippetTemplateTextChangingEventHandler(editControl1_CodeSnippetTemplateTextChanging)

Private Sub editControl1_CodeSnippetTemplateTextChanging(ByVal sender As Object, ByVal e As Syncfusion.Windows.Forms.Edit.CodeSnippetTemplateTextChangingEventArgs)
    ' The below line will be displayed in the output window at runtime.
    Console.WriteLine(" CodeSnippetTemplateTextChanging event is raised ")
End Sub

```

4.14.3.4 NewSnippetMemberHighlighting Event

This event is raised when a new code snippet member is highlighted.

The event handler receives an argument of type **NewSnippetMemberHighlightingEventArgs**. The following NewSnippetMemberHighlightingEventArgs members provide information, specific to this event.

Member	Description
Cancel	Indicates whether action has to be cancelled.
CodeSnippet	Code snippet that is currently activated.
NewSnippetMember	Snippet member that has to be highlighted.
OldSnippetMember	Previously highlighted snippet member.

[C#]

```

private void editControl1_NewSnippetMemberHighlighting(object sender,
Syncfusion.Windows.Forms.Edit.NewSnippetMemberHighlightingEventArgs e)
{
    // The below line will be displayed in the output window at runtime.
    Console.WriteLine(" NewSnippetMemberHighlighting event is raised ");
}

```

[VB .NET]

```

Private Sub editControl1_NewSnippetMemberHighlighting(ByVal sender As Object,
 ByVal e As Syncfusion.Windows.Forms.Edit.NewSnippetMemberHighlightingEventArgs)
    ' The below line will be displayed in the output window at runtime.
    Console.WriteLine(" NewSnippetMemberHighlighting event is raised ")

```

```
End Sub
```

4.14.4 ConfigurationChanged Event

This event is fired on changing the configuration of the Edit Control. Configuration can be set for the Edit Control using the **ApplyConfiguration** method.

The event handler receives an argument of type **EventArgs**.

[C#]

```
this.editControl1.ConfigurationChanged+=new  
EventHandler(editControl1_ConfigurationChanged);  
this.editControl1.ApplyConfiguration("XML");  
  
private void editControl1_ConfigurationChanged(object sender, EventArgs e)  
{  
    this.editControl1.ApplyConfiguration("XML");  
}
```

[VB .NET]

```
AddHandler Me.editControl1.ConfigurationChanged, AddressOf  
editControl1_ConfigurationChanged  
Me.editControl1.ApplyConfiguration("XML")  
  
Private Sub editControl1_ConfigurationChanged(ByVal sender As Object, ByVal e  
As EventArgs)  
    Console.WriteLine(" ConfigurationChanged event is raised ")  
End Sub
```

4.14.5 Collapse Events

This section discusses the below given collapse events.

4.14.5.1 CollapsedAll Event

This event is raised when the **CollapseAll** method is called.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the CollapsedAll event.  
this.editControl1.CollapsedAll+=new EventHandler(editControl1_CollapsedAll);  
// Call the CollapseAll method.  
this.editControl1.CollapseAll();  
  
private void editControl1_CollapsedAll(object sender, EventArgs e)  
{  
    // The below line will be displayed  
    Console.WriteLine(" CollapsedAll event is raised ");  
}
```

[VB .NET]

```
' Handle the CollapsedAll event.  
Me.editControl1.CollapsedAll+=New EventHandler(editControl1_CollapsedAll)  
' Call the CollapseAll method.  
Me.editControl1.CollapseAll()  
  
Private Sub editControl1_CollapsedAll(ByVal sender As Object, ByVal e As  
EventArgs)  
    Console.WriteLine(" CollapsedAll event is raised ")  
End Sub
```

4.14.5.2 CollapsingAll Event

This event is raised when the **CollapseAll** method is called.

The event handler receives an argument of type **CancelEventArgs**. The following CancellableEventArgs member provides information, specific to this event.

Member	Description
Cancel	Gets / sets a value indicating whether the event should be cancelled.

[C#]

```
// Handle the CollapsingAll event.  
this.editControl1.CollapsingAll+=new  
EventHandler(editControl1_CollapsingAll);  
  
// Call the CollapseAll method.  
this.editControl1.CollapseAll();  
  
private void editControl1_CollapsingAll(object sender, CancelEventArgs e)  
{  
    // The below given line will be displayed in the output window at runtime.  
    Console.WriteLine(" CollapsingAll event is raised ");  
    // Cancels the event.  
    e.Cancel = true;  
}
```

[VB .NET]

```
' Handle the CollapsingAll event.  
Me.editControl1.CollapsingAll+=New EventHandler(editControl1_CollapsingAll)  
  
' Call the CollapseAll method.  
Me.editControl1.CollapseAll()  
  
Private Sub editControl1_CollapsingAll(ByVal sender As Object, ByVal e As  
CancelEventArgs)  
    ' The below given line will be displayed in the output window at runtime.  
    Console.WriteLine(" CollapsingAll event is raised ")  
    ' Cancels the event.  
    e.Cancel = True  
End Sub
```

4.14.6 ContextChoice Events

This section discusses the below given context choice events.

4.14.6.1 ContextChoiceBeforeOpen Event

This event is discussed in the [Context Choice](#) topic.

4.14.6.2 ContextChoiceSelectedTextInsert Event

This event is discussed in the [Context Choice](#) topic.

4.14.6.3 ContextChoiceClose Event

This event is discussed in the [Context Choice](#) topic.

4.14.6.4 ContextChoiceItemSelected Event

This event is discussed in the [Context Choice](#) topic.

4.14.6.5 ContextChoiceUpdate Event

This event occurs when the context choice list is updated.

The event handler receives an argument of type **IContextChoiceController**. The following IContextChoiceController members provide information specific to this event.

Member	Description
Dropper	Gets / sets dropping lexem.
ExtendItemsFilteringString	Specifies whether autocomplete string should be extended.
FormSize	Gets / sets size of the context choice form.
Images	Gets collection of the INamedImage items.
IsVisible	Specifies whether context choice window associated with current controller is visible.
Items	Gets collection of the context choice items.
LexemBeforeDropper	Gets / sets lexem situated before dropper.

SelectedItem	Gets / sets currently selected item.
UseAutocomplete	Specifies whether autocomplete technique should be used with current context choice.

[C#]

```
// Create a new instance of the context choice item collection.
private ContextChoiceItemCollection c = new ContextChoiceItemCollection();

// Handle the ContextChoiceUpdate event.
this.editControl1.ContextChoiceUpdate+=new
Syncfusion.Windows.Forms.Edit.ContextChoiceEventHandler(editControl1_ContextC
hoiceUpdate);

// IContextChoiceController.LexemBeforeDropper property returns the lexem
before the dropper which displays the context choice. It is possible to
control the lexem being searched in the context choice list using the
ContextChoiceUpdate event.
private void
editControl1_ContextChoiceUpdate(Syncfusion.Windows.Forms.Edit.Interfaces.ICo
ntextChoiceController controller)
{
Console.WriteLine("LexemBeforeDropper:" +
controller.LexemBeforeDropper.Text);
controller.Items.Clear();
foreach (IContextChoiceItem item in c)
{
    if (item.Text.Equals(controller.LexemBeforeDropper.Text))
    {
        controller.Items.Add(item.Text);
    }
}
}
```

[VB .NET]

```
' Create a new instance of the context choice item collection.
Private c As ContextChoiceItemCollection = New ContextChoiceItemCollection()

' Handle the ContextChoiceUpdate event.
Me.editControl1.ContextChoiceUpdate+=New
Syncfusion.Windows.Forms.Edit.ContextChoiceEventHandler(editControl1_ContextC
hoiceUpdate)

' IContextChoiceController.LexemBeforeDropper property returns the lexem
```

```
before the dropper which displays the context choice. It is possible to
control the lexem being searched in the context choice list using the
ContextChoiceUpdate event.

Private Sub editControl1_ContextChoiceUpdate(ByVal controller As
Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController)
Console.WriteLine("LexemBeforeDropper:" + controller.LexemBeforeDropper.Text)
controller.Items.Clear()
Dim item As IContextChoiceItem
For Each item In c
    If item.Text.Equals(controller.LexemBeforeDropper.Text) Then
        controller.Items.Add(item.Text)
    End If
Next
End Sub
```

4.14.6.6 ContextChoiceOpen Event

This event is discussed in the [Context Choice](#) topic.

4.14.6.7 ContextChoiceRightClick Event

This event is raised when the context choice item is right-clicked.

The event handler receives an argument of type **ContextChoiceEventArgs**. The following CancellableCodeSnippetsEventArgs member provides information, specific to this event.

Member	Description
Item	Underlying ContextChoiceItem.

[C#]

```
private void
editControl1_ContextChoiceRightClick(Syncfusion.Windows.Forms.Edit.Interfaces
.IContextChoiceController sender,
Syncfusion.Windows.Forms.Edit.ContextChoiceEventArgs e)
{
    e.Item.ForeColor = System.Drawing.Color.Maroon;
    e.Item.BackColor = System.Drawing.Color.MistyRose;
    MessageBox.Show(" ContextChoiceRightClick event is raised ");
}
```

[VB.NET]

```
Private Sub editControl1_ContextChoiceRightClick(ByVal sender As Syncfusion.Windows.Forms.Edit.Interfaces.IContextChoiceController, ByVal e As Syncfusion.Windows.Forms.Edit.ContextChoiceItemEventArgs)
    e.Item.ForeColor = System.Drawing.Color.Maroon
    e.Item.BackColor = System.Drawing.Color.MistyRose
    MessageBox.Show(" ContextChoiceRightClick event is raised ")
End Sub
```

4.14.7 ContextPrompt Events

This section discusses the below given context prompt events.

4.14.7.1 ContextPromptBeforeOpen Event

This event is discussed in the [Context Prompt](#) topic.

4.14.7.2 ContextPromptClose Event

This event is discussed in the [Context Prompt](#) topic.

4.14.7.3 ContextPromptOpen Event

This event is discussed in the [Context Prompt](#) topic.

4.14.7.4 ContextPromptSelectionChanged Event

This event is discussed in the [Context Prompt](#) topic.

4.14.7.5 ContextPromptUpdate Event

This event is discussed in the [Context Prompt](#) topic.

4.14.8 CursorPositionChanged Event

This event is raised when the current cursor position is changed.

The event handler receives an argument of type **EventArgs**.

[C#]

```
private void editControl1_CursorPositionChanged(object sender, EventArgs e)
{
    MessageBox.Show(" CurrentCursorPosition event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_CursorPositionChanged(ByVal sender As Object, ByVal
e As EventArgs)
    MessageBox.Show(" CursorPositionChanged event is raised ")
End Sub
```

4.14.9 Expand Events

This section discusses the expand events given below.

4.14.9.1 ExpandedAll Event

This event is raised when the **ExpandAll** method is called.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the ExpandedAll event.  
this.editControl1.ExpandedAll+=new EventHandler(editControl1_ExpandedAll);  
  
// Call the ExpandAll method.  
this.editControl1.ExpandAll();  
  
private void editControl1_ExpandedAll(object sender, EventArgs e)  
{  
    // The below line will be displayed in the output window at runtime.  
    Console.WriteLine(" ExpandedAll event is raised ");  
}
```

[VB.NET]

```
' Handle the ExpandedAll event.  
Me.editControl1.ExpandedAll+=New EventHandler(editControl1_ExpandedAll)  
  
' Call the ExpandAll method.  
Me.editControl1.ExpandAll()  
  
Private Sub editControl1_ExpandedAll(ByVal sender As Object, ByVal e As  
EventArgs)  
    ' The below line will be displayed in the output window at runtime  
    Console.WriteLine(" ExpandedAll event is raised ")  
End Sub
```

4.14.9.2 ExpandingAll Event

This event is raised when the **ExpandAll** method is called.

The event handler receives an argument of type **CancellableEventArgs**. The following CancellableEventArgs member provides information, specific to this event.

Member	Description
Cancel	Gets / sets a value indicating whether the event should be cancelled.

[C#]

```
// Handle the ExpandingAll event.  
this.editControl1.ExpandingAll+=new EventHandler(editControl1_ExpandingAll);
```

```
// Call the ExpandAll method.  
this.editControl1.ExpandAll();  
  
private void editControl1_ExpandingAll(object sender, CancelEventArgs e)  
{  
    // The below given line will be displayed in the output window at runtime.  
    Console.WriteLine(" ExpandingAll event is raised ");  
  
    // Cancels the event.  
    e.Cancel = true;  
}
```

[VB .NET]

```
' Handle the ExpandingAll event.  
Me.editControl1.ExpandingAll+=New EventHandler(editControl1_ExpandingAll)  
  
' Call the ExpandAll method.  
Me.editControl1.ExpandAll()  
  
Private Sub editControl1_ExpandingAll(ByVal sender As Object, ByVal e As  
CancelEventArgs)  
    ' The below given line will be displayed in the output window at runtime.  
    Console.WriteLine(" CollapsingAll event is raised ")  
  
    ' Cancels the event.  
    e.Cancel = True  
End Sub
```

4.14.10 Indicator Margin Events

This section discusses the below given indicator margin events.

4.14.10.1 IndicatorMarginClick Event

This event is raised when the user clicks on the indicator margin area.

The event handler receives an argument of type **IndicatorClickEventArgs**. The following **IndicatorClickEventArgs** members provide information, specific to this event.

Member	Description
Bookmark	Gets clicked custom bookmark if available.
LineIndex	Gets clicked line index.

[C#]

```
private void editControl1_IndicatorMarginClick(object sender,
Syncfusion.Windows.Forms.Edit.IndicatorEventArgs e)
{
    Console.WriteLine(" IndicatorMarginClick event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_IndicatorMarginClick(ByVal sender As Object, ByVal e
As Syncfusion.Windows.Forms.Edit.IndicatorEventArgs)
    Console.WriteLine(" IndicatorMarginClick event is raised ")
End Sub
```

4.14.10.2 IndicatorMarginDoubleClick Event

This event is raised when the user double-clicks on the indicator margin area.

The event handler receives an argument of type **IndicatorEventArgs**. The following **IndicatorEventArgs** members provide information, specific to this event.

Member	Description
Bookmark	Gets clicked custom bookmark if available.
LineIndex	Gets clicked line index.

[C#]

```
private void editControl1_IndicatorMarginDoubleClick(object sender,
Syncfusion.Windows.Forms.Edit.IndicatorEventArgs e)
{
    Console.WriteLine(" IndicatorMarginDoubleClick event is raised ");
}
```

[VB.NET]

```
Private Sub editControl1_IndicatorMarginDoubleClick(ByVal sender As Object,
ByVal e Syncfusion.Windows.Forms.Edit.IndicatorClickEventArgs)
    Console.WriteLine(" IndicatorMarginDoubleClick event is raised ")
End Sub
```

4.14.10.3 DrawLineMark Event

This event occurs when a custom line mark should be drawn.

The event handler receives an argument of type **DrawLineMarkEventArgs**. The following **DrawLineMarkEventArgs** members provide information specific to this event.

Member	Description
CustomDraw	If set to True, user handles drawing of the bookmark.
Graphics	Graphics object.
MarkRect	Rectangle where line mark should be drawn.
PhysicalLine	Virtual line number.
VirtualLine	Physical line number.

[C#]

```
private void editControl1_DrawLineMark(object sender,
Syncfusion.Windows.Forms.Edit.DrawLineMarkEventArgs e)
{
    if( e.VirtualLine % 2 == 0 )
    {
        Brush brush = new LinearGradientBrush(e.MarkRect, Color.Red,
Color.Yellow, LinearGradientMode.Vertical);
        e.Graphics.FillRectangle(brush, e.MarkRect);
        e.Graphics.DrawRectangle(Pens.IndianRed, e.MarkRect);
    }
}
```

[VB.NET]

```

Private Sub editControl1_DrawLineMark(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.DrawLineMarkEventArgs)
    If e.VirtualLine Mod 2 = 0 Then
        Dim brush As Brush = New Drawing2D.LinearGradientBrush(e.MarkRect,
Color.Red, Color.Yellow, LinearGradientMode.Vertical)
        e.Graphics.FillRectangle(brush, e.MarkRect)
        e.Graphics.DrawRectangle(Pens.IndianRed, e.MarkRect)
    End If
End Sub

```

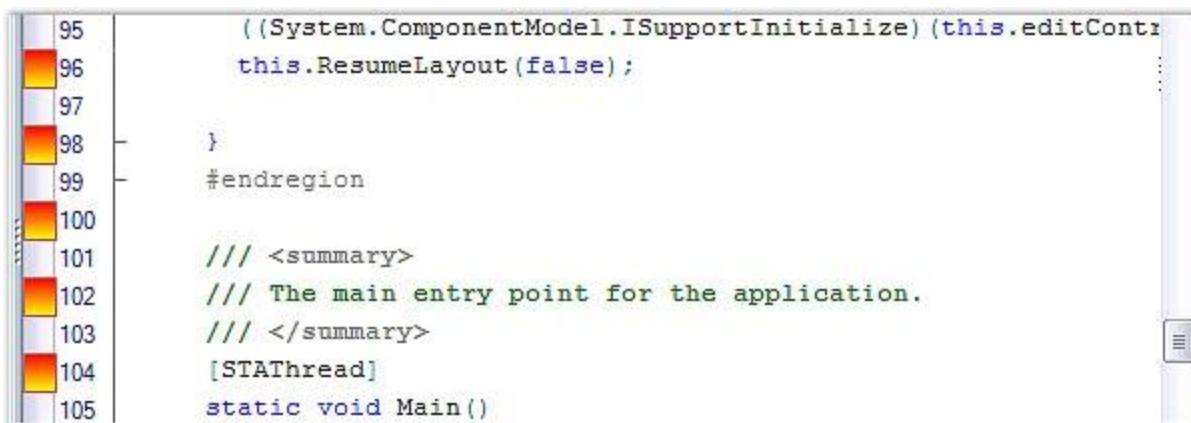


Figure 82: Custom Indicators in the Indicator Margin

4.14.11 InsertModeChanged Event

This event is fired when the value of the **InsertMode** property changes. The **InsertMode** property specifies the insert mode state.

The event handler receives an argument of type **EventArgs**.

```

[C#]

// Handle the InsertModeChanged event.
this.editControl1.InsertModeChanged+=new
EventHandler(editControl1_InsertModeChanged);

// Set the value of the InsertMode property.
this.editControl1.InsertMode = false;

private void editControl1_InsertModeChanged(object sender, EventArgs e)

```

```
{  
// The below statement can be seen in the output window at runtime.  
Console.WriteLine(" InsertModeChanged event is raised ");  
}
```

[VB.NET]

```
' Handle the InsertModeChanged event.  
AddHandler Me.editControl1.InsertModeChanged, AddressOf  
editControl1_InsertModeChanged  
  
' Set the value of the InsertMode property.  
Me.editControl1.InsertMode = False  
  
Private Sub editControl1_InsertModeChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    The below statement can be seen in the output window at runtime.  
    Console.WriteLine(" InsertModeChanged event is raised ")  
End Sub
```

4.14.12 LanguageChanged Event

This event occurs when the current parser language is changed.

The event handler receives an argument of type **EventArgs**.

[C#]

```
private void editControl1_LanguageChanged(object sender, EventArgs e)  
{  
    Console.WriteLine(" LanguageChanged event is raised ");  
}
```

[VB.NET]

```
Private Sub editControl1_LanguageChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    Console.WriteLine(" LanguageChanged event is raised ")  
End Sub
```

4.14.13 MenuFill Event

This event is discussed in the [Customizable Context Menu](#) topic.

4.14.14 Operation Events

This section discusses the below given operation events.

4.14.14.1 OperationStarted Event

This event occurs when an operation starts.

The event handler receives an argument of type **ILongOperation**. The following **ILongOperation** members provide information, specific to this event.

Member	Description
ID	Gets ID of the operation.
IsRunning	Gets value indicating whether operation is running now.
Name	Gets name of the operation.
RunningTime	Gets time of the operation activity.

[C#]

```
private void
editControl1_OperationStarted(Syncfusion.Windows.Forms.Edit.Interfaces.ILongO
peration operation)
{
    Console.WriteLine(" OperationStarted event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_OperationStarted(ByVal operation As
Syncfusion.Windows.Forms.Edit.Interfaces.ILongOperation)
    Console.WriteLine(" OperationStarted event is raised ")
```

```
End Sub
```

4.14.14.2 OperationStopped Event

This event occurs when an operation ends.

The event handler receives an argument of type **ILongOperation**. The following **ILongOperation** members provide information, specific to this event.

Member	Description
ID	Gets ID of the operation.
IsRunning	Gets value indicating whether operation is running now.
Name	Gets name of the operation.
RunningTime	Gets time of the operation activity.

[C#]

```
private void  
editControl1_OperationStopped(Syncfusion.Windows.Forms.Edit.Interfaces.ILongO  
peration operation)  
{  
    Console.WriteLine(" OperationStopped event is raised ");  
}
```

[VB .NET]

```
Private Sub editControl1_OperationStopped(ByVal operation As  
Syncfusion.Windows.Forms.Edit.Interfaces.ILongOperation)  
    Console.WriteLine(" OperationStopped event is raised ")  
End Sub
```

4.14.15 Outlining Events

This section discusses the below given outlining events.

4.14.15.1 OutliningBeforeCollapse Event

This event is raised before a region is about to collapse.

The event handler receives an argument of type **OutliningEventArgs**. The following **OutliningEventArgs** members provide information, specific to this event.

Member	Description
Cancel	Gets / sets value indicating whether the user cancels the underlying event.
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningBeforeCollapse(object sender,
Syncfusion.Windows.Forms.Edit.OutliningEventArgs e)
{
    Console.WriteLine(" OutliningBeforeCollapse event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_OutliningBeforeCollapse(ByVal sender As Object,
ByVal e As Syncfusion.Windows.Forms.Edit.OutliningEventArgs)
    Console.WriteLine(" OutliningBeforeCollapse event is raised ")
End Sub
```

4.14.15.2 OutliningBeforeExpand Event

This event is raised before a region is about to expand.

The event handler receives an argument of type **OutliningEventArgs**. The following **OutliningEventArgs** members provide information, specific to this event.

Member	Description
Cancel	Gets / sets value indicating whether the user cancels the underlying event.
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningBeforeExpand(object sender,
Syncfusion.Windows.Forms.Edit.OutliningEventArgs e)
{
    Console.WriteLine(" OutliningBeforeExpand event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_OutliningBeforeExpand(ByVal sender As Object, ByVal
e As Syncfusion.Windows.Forms.Edit.OutliningEventArgs)
    Console.WriteLine(" OutliningBeforeExpand event is raised ")
End Sub
```

4.14.15.3 OutliningCollapse Event

This event is raised when a region collapses.

The event handler receives an argument of type **CollapseEventArgs**. The following CollapseEventArgs members provide information, specific to this event.

Member	Description
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningCollapse(object sender,
```

```
Syncfusion.Windows.Forms.Edit.CollapseEventArgs e)
{
    Console.WriteLine(" OutliningCollapse event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_OutliningCollapse(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.CollapseEventArgs)
    Console.WriteLine(" OutliningBeforeCollapse event is raised ")
End Sub
```

4.14.15.4 OutliningExpand Event

This event is raised when a region expands.

The event handler receives an argument of type **CollapseEventArgs**. The following CollapseEventArgs members provide information, specific to this event.

Member	Description
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningExpand(object sender,
Syncfusion.Windows.Forms.Edit.CollapseEventArgs e)
{
    Console.WriteLine(" OutliningExpand event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_OutliningExpand(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.CollapseEventArgs)
    Console.WriteLine(" OutliningExpand event is raised ")
End Sub
```

4.14.15.5 OutliningTooltipBeforePopup Event

This event is discussed in the [Outlining ToolTip](#) topic.

4.14.15.6 OutliningTooltipClose Event

This event is raised when the outlining tooltip is closed.

The event handler receives an argument of type **CollapseEventArgs**. The following CollapseEventArgs members provide information, specific to this event.

Member	Description
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningTooltipClose(object sender,  
Syncfusion.Windows.Forms.Edit.CollapseEventArgs e)  
{  
    Console.WriteLine(" OutliningTooltipClose event is raised ");  
}
```

[VB .NET]

```
Private Sub editControl1_OutliningTooltipClose(ByVal sender As Object, ByVal  
e As Syncfusion.Windows.Forms.Edit.CollapseEventArgs)  
    Console.WriteLine(" OutliningTooltipClose event is raised ")  
End Sub
```

4.14.15.7 OutliningTooltipPopup Event

This event is raised when the outlining tooltip is shown.

The event handler receives an argument of type **CollapseEventArgs**. The following CollapseEventArgs members provide information, specific to this event.

Member	Description
CollapsedText	Gets / sets collapsed text.
CollapseName	Gets / sets collapse name.
Collapser	Gets / sets collapser.

[C#]

```
private void editControl1_OutliningTooltipPopup(object sender,  
Syncfusion.Windows.Forms.Edit.CollapseEventArgs e)  
{  
    Console.WriteLine(" OutliningTooltipPopup event is raised ");  
}
```

[VB .NET]

```
Private Sub editControl1_OutliningTooltipPopup(ByVal sender As Object, ByVal  
e As Syncfusion.Windows.Forms.Edit.CollapseEventArgs)  
    Console.WriteLine(" OutliningTooltipPopup event is raised ")  
End Sub
```

4.14.16 Print Events

It has the following events:

4.14.16.1 PrintHeader Event

This event is discussed in the [Printing](#) topic.

4.14.16.2 PrintFooter Event

This event is discussed in the [Printing](#) topic.

4.14.17 ReadOnlyChanged Event

This event occurs when the **ReadOnly** property is changed. The **ReadOnly** property specifies whether the **Edit Control** is in the read-only mode.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the ReadOnlyChanged event.  
this.editControl1.ReadOnlyChanged+=new  
EventHandler(editControl1_ReadOnlyChanged);  
  
// Set the ReadOnly property to True.  
this.editControl1.ReadOnly = true;  
  
private void editControl1_ReadOnlyChanged(object sender, EventArgs e)  
{  
    Console.WriteLine(" ReadOnlyChanged event is raised ");  
}
```

[VB .NET]

```
' Handle the ReadOnlyChanged event.  
Me.editControl1.ReadOnlyChanged+=New  
EventHandler(editControl1_ReadOnlyChanged)  
  
' Set the ReadOnly property to True.  
Me.editControl1.ReadOnly = True  
  
Private Sub editControl1_ReadOnlyChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    Console.WriteLine(" ReadOnlyChanged event is raised ")  
End Sub
```

4.14.18 RegisteringDefaultKeyBindings Event

This event is discussed in the [Keystroke - Action Combinations Binding](#) topic.

4.14.19 RegisteringKeyCommands Event

This event is discussed in the [Keystroke - Action Combinations Binding](#) topic.

4.14.20 Save Events

This section discusses the below given events that are generated when the user saves files and streams with data loss.

4.14.20.1 SaveFileWithDataLoss Event

This event is raised when user tries to save files with data loss.

The event handler receives an argument of type **SaveWithDataLosingEventArgs**. The following **SaveWithDataLosingEventArgs** members provide information, specific to this event.

Member	Description
SaveWithLoss	Gets / sets value that indicates whether data has to be saved with loss.
UserHandling	Gets / sets value that indicates whether user handled the event.

[C#]

```
private void editControl1_SaveFileWithDataLoss(object sender,  
Syncfusion.Windows.Forms.Edit.SaveWithDataLosingEventArgs e)  
{  
    e.SaveWithLoss = true;  
    e.UserHandling = true;  
}
```

[VB .NET]

```
Private Sub editControl1_SaveFileWithDataLoss(ByVal sender As Object, ByVal e  
As Syncfusion.Windows.Forms.Edit.SaveWithDataLosingEventArgs)  
    e.SaveWithLoss = True
```

```
e.UserHandling = True  
End Sub
```

4.14.20.2 SaveStreamWithDataLoss Event

This event is raised when user tries to save streams with data loss.

The event handler receives an argument of type **SaveWithDataLosingEventArgs**. The following **SaveWithDataLosingEventArgs** members provide information, specific to this event.

Member	Description
SaveWithLoss	Gets / sets value that indicates whether data has to be saved with loss.
UserHandling	Gets / sets value that indicates whether user handled the event.

[C#]

```
private void editControl1_SaveStreamWithDataLoss(object sender,  
Syncfusion.Windows.Forms.Edit.SaveWithDataLosingEventArgs e)  
{  
    e.SaveWithLoss = true;  
    e.UserHandling = true;  
}
```

[VB .NET]

```
Private Sub editControl1_SaveStreamWithDataLoss(ByVal sender As Object, ByVal  
e As Syncfusion.Windows.Forms.Edit.SaveWithDataLosingEventArgs)  
    e.SaveWithLoss = True  
    e.UserHandling = True  
End Sub
```

4.14.21 Scroll Events

This section discusses the below given events that are generated when horizontal and vertical scrolling takes place.

4.14.21.1 HorizontalScroll Event

This event is raised when user scrolls the window horizontally.

The event handler receives an argument of type **ScrollEventArgs**. The following ScrollEventArgs members provide information specific to this event.

Member	Description
NewValue	Gets / sets the new System.Windows.Forms.ScrollBar.Value for the scrollbar.
OldValue	Gets / sets the old System.Windows.Forms.ScrollBar.Value for the scrollbar.
ScrollOrientation	Gets the scrollbar orientation that raised the scroll event.
Type	Gets the type of scroll event that occurred.

[C#]

```
private void editControl1_HorizontalScroll(object sender, ScrollEventArgs e)
{
    Console.WriteLine(" HorizontalScroll event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_HorizontalScroll(ByVal sender As Object, ByVal e As
ScrollEventArgs)
    Console.WriteLine(" HorizontalScroll event is raised ")
End Sub
```

4.14.21.2 VerticalScroll Event

This event is raised when the user scrolls the window vertically.

The event handler receives an argument of type **ScrollEventArgs**. The following ScrollEventArgs members provide information specific to this event.

Member	Description
--------	-------------

NewValue	Gets / sets the new System.Windows.Forms.ScrollBar.Value for the scrollbar.
OldValue	Gets / sets the old System.Windows.Forms.ScrollBar.Value for the scrollbar.
ScrollOrientation	Gets the scrollbar orientation that raised the scroll event.
Type	Gets the type of scroll event that occurred.

[C#]

```
private void editControl1_VirtualScroll(object sender, ScrollEventArgs e)
{
    Console.WriteLine(" VirtualScroll event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_VirtualScroll(ByVal sender As Object, ByVal e As
EventArgs)
    Console.WriteLine(" VirtualScroll event is raised ")
End Sub
```

4.14.22 SelectionChanged Event

This event is raised when text selection has been changed.

The event handler receives an argument of type **EventArgs**.

[C#]

```
private void editControl1_SelectionChanged(object sender, EventArgs e)
{
    MessageBox.Show(" SelectionChanged event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_SelectionChanged(ByVal sender As Object, ByVal e As
EventArgs)
    MessageBox.Show(" SelectionChanged event is raised ")
End Sub
```

4.14.23 SingleLineChanged Event

This event is fired when the value of the **SingleLineMode** property is changed. The **SingleLineMode** property specifies whether the single line mode is enabled.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the SingleLineChanged event.  
this.editControl1.SingleLineChanged+=new  
EventHandler(editControl1_SingleLineChanged);  
  
// Set the SingleLineMode property to True.  
this.editControl1.SingleLineMode = true;  
  
private void editControl1_SingleLineChanged(object sender, EventArgs e)  
{  
    // The below statement can be seen in the output window at runtime.  
    Console.WriteLine(" SingleLineChanged event is raised ");  
}
```

[VB .NET]

```
' Handle the SingleLineChanged event.  
AddHandler Me.editControl1.SingleLineChanged, AddressOf  
editControl1_SingleLineChanged  
  
' Set the SingleLineMode property to True.  
Me.editControl1.SingleLineMode = True  
  
Private Sub editControl1_SingleLineChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    ' The below statement can be seen in the output window at runtime.  
    Console.WriteLine(" SingleLineChanged event is raised ")  
End Sub
```

4.14.24 Text Events

This section discusses the following text events:

- **TextChanged**

- TextChanging

4.14.24.1 TextChanged Event

This event is fired when the text in the Edit Control is changed.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the TextChanged event.  
this.editControl1.TextChanged += new EventHandler(editControl1_TextChanged);  
  
// Set the text of the EditControl.  
this.editControl1.Text = "Sample Text";  
  
private void editControl1_TextChanged(object sender, EventArgs e)  
{  
    // The below statement can be seen in the output window at runtime.  
    Console.WriteLine(" TextChanged event is raised ");  
}
```

[VB.NET]

```
' Handle the TextChanged event.  
AddHandler Me.editControl1.TextChanged, AddressOf editControl1_TextChanged  
  
' Set the text of the EditControl.  
Me.editControl1.Text = "Sample Text"  
  
Private Sub editControl1_TextChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    ' The below statement can be seen in the output window at runtime.  
    Console.WriteLine(" TextChanged event is raised ")  
End Sub
```

4.14.24.2 TextChanging Event

This event is raised when the text is to be changed.

The event handler receives an argument of type **TextChangingEventArgs**. The following **TextChangingEventArgs** members provide information, specific to this event.

Member	Description
Cancel	Gets/sets the value indicating whether text change has been canceled.
StartColumn	Gets/sets virtual column of Insert/Delete start.
StartLine	Gets/sets virtual line of Insert/Delete start.
Text	Gets/sets event's text.
Type	Gets/sets type of the event (Changed/Insert/Delete). It includes the below given options.

[C#]

```
private void editControl1_TextChanging(object sender,
Syncfusion.Windows.Forms.Edit.TextChangingEventArgs e)
{
    e.Type = Syncfusion.Windows.Forms.Edit.Enums.TextChange.Deleted;

    // The below statement can be seen in the output window at runtime when the
    // text of the Edit Control is deleted.
    Console.WriteLine(" TextChanging event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_TextChanging(ByVal sender As Object, ByVal e As
Syncfusion.Windows.Forms.Edit.TextChangingEventArgs)
    e.Type = Syncfusion.Windows.Forms.Edit.Enums.TextChange.Deleted

    ' The below statement can be seen in the output window at runtime when the
    ' text of the Edit Control is deleted.
    Console.WriteLine(" TextChanging event is raised ")
End Sub
```

4.14.24.3 Line Modification Events

The line modification events occur whenever a line in the Edit control is subjected to a change by modifying the text of an existing line, inserting a line, or removing a line in the editor.

The following events are triggered from the control when the editor is modified:

4.14.24.3.1 Line Changed

The **LineChanged** event will be fired when any line is modified in the Edit control.

[C#]

```
// Handle the LineChanged event.

this.editControl1.LineChanged += new
    Syncfusion.Windows.Forms.Edit.TextChangingEventHandler(editControl1_Lin
eChanged);

void
editControl1_LineChanged(object sender, Syncfusion.Windows.Forms.Edit.TextCh
angingEventArgs e)
{
    //The following statement can be seen in the output window at
    run time.

    Console.WriteLine("Line Changed");
}
```

[VB.NET]

```
'Handle the LineChanged event.

AddHandler Me.editControl1.LineChanged, AddressOf Me.editControl1_Lin
eChanged

Private Sub editControl1_LineChanged(ByVal , As object sender,
    ByVal e As Syncfusion.Windows.Forms.Edit.TextChangingEventArgs)

    'The following statement can be seen in the output window at ru
    ntime.

    Console.WriteLine("Line Changed")
```

```
End Sub
```

4.14.24.3.2 Line Inserted

The **LineInserted** event will be fired when a new line is inserted in the Edit control.

[C#]

```
// Handle the LineInserted event.

    this.editControl1.LineInserted += new
Syncfusion.Windows.Forms.Edit.LineInsertedEventHandler(editControl1_Lin
eInserted);

void
editControl1_LineInserted(object sender, Syncfusion.Windows.Forms.Edit.Li
nesEventArgs e)
{
    // The following statement can be seen in the output window at
run time.

    Console.WriteLine("Line Inserted");
}
```

[VB.NET]

```
'Handle the LineInserted event.

AddHandler Me.editControl1.LineInserted, AddressOf Me.editControl1_Line
Inserted

Private Sub editControl1_LineInserted(ByVal , As object sender,
ByVal e As Syncfusion.Windows.Forms.Edit.LinesEventArgs)

    'The following statement can be seen in the output window at ru
n time.
    Console.WriteLine("Line Inserted")

End Sub
```

4.14.24.3.3 Line Deleted

The **LineDeleted** event will be fired when any line is removed from the Edit control.

[C#]

```
// Handle the LineDeleted event.  
  
this.editControl1.LineDeleted += new  
Syncfusion.Windows.Forms.Edit.LineDeletedEventHandler(editControl1_Line  
Deleted);  
  
  
void editControl1_LineDeleted(object sender,  
Syncfusion.Windows.Forms.Edit.LinesEventArgs e)  
{  
    // The following statement can be seen in the output window  
    // at run time.  
    Console.WriteLine("Line Deleted");  
}
```

[VB.NET]

```
'Handle the LineDeleted event.  
AddHandler Me.editControl1.LineDeleted, AddressOf Me.editControl1_LineD  
eleted  
  
  
Private Sub editControl1_LineDeleted(ByVal sender As Object,  
  
    ByVal e As Syncfusion.Windows.Forms.Edit.LinesEventArgs)  
  
    'The following statement can be seen in the output window at run time.  
    Console.WriteLine("Line Deleted")  
  
End Sub
```

4.14.25 UnreachableTextFound Event

This event occurs when text in a hidden block is found and this block can't be expanded due to user's canceling.

The event handler receives an argument of type **UnreachableTextFoundEventArgs**. The following UnreachableTextFoundEventArgs members provide information, specific to this event.

Member	Description
ContinueSearch	Indicates whether search must be continued.
Point	Point of the location of unreachable text.
Text	Searched text.

[C#]

```
private void editControl1_UnreachableTextFound(object sender,  
Syncfusion.Windows.Forms.Edit.UnreachableTextFoundEventArgs e)  
{  
    Console.WriteLine(" UnreachableTextFound event is raised ");  
}
```

[VB .NET]

```
Private Sub editControl1_UnreachableTextFound(ByVal sender As Object, ByVal e  
As Syncfusion.Windows.Forms.Edit.UnreachableTextFoundEventArgs)  
    Console.WriteLine(" UnreachableTextFound event is raised ")  
End Sub
```

4.14.26 UpdateBookmarkToolTip Event

This event is fired when the bookmark tooltip text is updated.

The event handler receives an argument of type **UpdateBookmarkTooltipEventArgs**. The following UpdateBookmarkTooltipEventArgs members provide information specific to this event.

Member	Description
Bookmark	Bookmark.
HintedArea	Rectangle that represents an object which has this tooltip.

Image	Gets / sets image associated with the tooltip.
Line	Index of the bookmarked line.
Text	Text of the tooltip.
X	Mouse X coordinate in client coordinates.
Y	Mouse Y coordinate in client coordinates.

[C#]

```
// Handle the UpdateBookmarkToolTip event.  
this.editControl1.UpdateBookmarkToolTip+=new  
Syncfusion.Windows.Forms.Edit.UpdateBookmarkTooltipEventHandler(editControl1_  
UpdateBookmarkToolTip);  
// Set the bookmark at the specified line.  
this.editControl1.BookmarkAdd(this.editControl1.CurrentLine);  
// Specify whether bookmark tooltip should be shown.  
this.editControl1.ShowBookmarkTooltip = true;  
  
private void editControl1_UpdateBookmarkToolTip(object sender,  
Syncfusion.Windows.Forms.Edit.UpdateBookmarkTooltipEventArgs e)  
{  
    // Set the bookmark tooltip text.  
    e.Text = " Introduction to Essential Edit ";  
}
```

[VB.NET]

```
' Handle the UpdateBookmarkToolTip event.  
AddHandler Me.editControl1.UpdateBookmarkToolTip, AddressOf  
editControl1_UpdateBookmarkToolTip  
' Set the bookmark at the specified line.  
Me.editControl1.BookmarkAdd(Me.editControl1.CurrentLine)  
' Specify whether bookmark tooltip should be shown.  
Me.editControl1.ShowBookmarkTooltip = True  
  
Private Sub editControl1_UpdateBookmarkToolTip(ByVal sender As Object, ByVal  
e As Syncfusion.Windows.Forms.Edit.UpdateBookmarkTooltipEventArgs)  
    ' Set the bookmark tooltip text.  
    e.Text = " Introduction to Essential Edit "  
End Sub
```

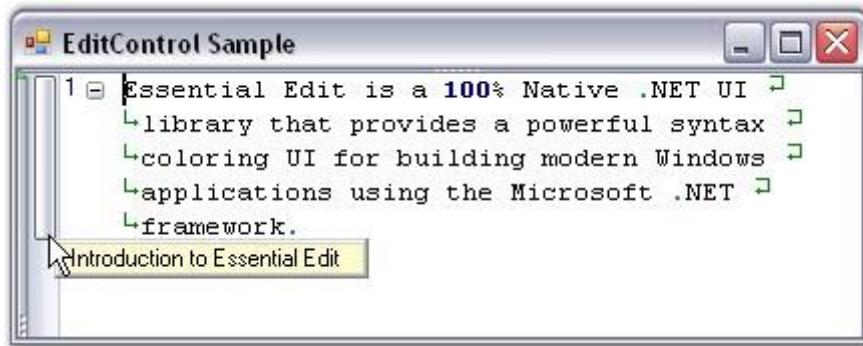


Figure 83: Bookmark ToolTip Text Displayed

4.14.27 UpdateContextToolTip Event

This event is discussed in the [Context Tooltip](#) topic.

4.14.28 User Margin Events

This section discusses the below given user margin events.

4.14.28.1 DrawUserMarginText Event

This event is discussed in the [User Margin](#) topic.

4.14.28.2 PaintUserMargin Event

This event occurs when the user margin has to be painted.

The event handler receives an argument of type **PaintEventArgs**. The following PaintEventArgs members provide information, specific to this event.

Member	Description
--------	-------------

ClipRectangle	Gets the rectangle area to paint.
Graphics	Gets the graphics that will be used to paint.

[C#]

```
private void editControl1_PaintUserMargin(object sender, PaintEventArgs e)
{
    Console.WriteLine(" PaintUserMargin event is raised ");
}
```

[VB .NET]

```
Private Sub editControl1_PaintUserMargin(ByVal sender As Object, ByVal e As
PaintEventArgs)
    Console.WriteLine(" PaintUserMargin event is raised ")
End Sub
```

4.14.29 WordWrapChanged Event

This event is fired when the value of the **WordWrapMode** property is changed. The WordWrapMode property specifies the mode of word wrapping.

The event handler receives an argument of type **EventArgs**.

[C#]

```
// Handle the WordWrapChanged event.
this.editControl1.WordWrapChanged+=new
EventHandler(editControl1_WordWrapChanged);

// Specify the mode of word wrapping.
this.editControl1.WordWrapMode =
Syncfusion.Windows.Forms.Edit.Enums.WordWrapMode.WordWrapMargin;

private void editControl1_WordWrapChanged(object sender, EventArgs e)
{
    // The below line will be displayed in the output window at runtime.
    Console.WriteLine(" WordWrapChanged event is raised ");
}
```

[VB.NET]

```
' Handle the WordWrapChanged event.  
AddHandler Me.editControl1.WordWrapChanged, AddressOf  
editControl1_WordWrapChanged  
  
' Specify the mode of word wrapping.  
Me.editControl1.WordWrapMode =  
Syncfusion.Windows.Forms.Edit.Enums.WordWrapMode.WordWrapMargin  
  
Private Sub editControl1_WordWrapChanged(ByVal sender As Object, ByVal e As  
EventArgs)  
    ' The below line will be displayed in the output window at runtime.  
    Console.WriteLine(" WordWrapChanged event is raised ")  
End Sub
```

5 Frequently Asked Questions

This section illustrates the solutions for various task-based queries about Essential Edit. Following are FAQs for the Edit Control:

5.1 How To Access the Text Associated With Individual Lines In the Selected Text Region Of the Edit Control

The below given code snippet illustrates how you can access the text associated with individual lines in the selected text region of the Edit Control.

[C#]

```
string cleanedSQL = "";
if (this.editControl1.SelectedText != "")
{
    // Get the start and end points of the selection
    CoordinatePoint start = this.editControl1.Selection.Top;
    CoordinatePoint end = this.editControl1.Selection.Bottom;
    string lineText = "";
    for (int i=start.VirtualLine; i<=end.VirtualLine; i++)
    {
        // Get the line object
        ILexemLine lexemLine = this.editControl1.GetLine(i);

        // Get the tokens in each line object and append them to get the line
        // text
        foreach (ILexem lexem in lexemLine.LineLexems)
        {
            lineText += lexem.Text;
        }

        // Store each of these line text
        cleanedSQL += lineText + "\n";
        lineText="";
    }
}
```

[VB .NET]

```
Dim cleanedSQL As String = ""
If Me.editControl1.SelectedText <> "" Then
    ' Get the start and end points of the selection
    Dim start As CoordinatePoint = Me.editControl1.Selection.Top
    Dim end As CoordinatePoint = Me.editControl1.Selection.Bottom
    Dim lineText As String = ""
    Dim i As Integer
    For i = start.VirtualLine To i<- 1 Step =end.VirtualLine
        ' Get the line object
        Dim lexemLine As ILexemLine = Me.editControl1.GetLine(i)

        ' Get the tokens in each line object and append them to get the line
        ' text
        Dim lexem As ILexem
        For Each lexem In lexemLine.LineLexems
            lineText += lexem.Text
        Next

        ' Store each of these line text
        cleanedSQL += lineText + "\n"
        lineText=""
    Next
End If
```

5.2 How To Change the Lexems Dynamically

You can change the lexems dynamically by adding / removing the lexems by using the **Lexem.Add** and **Lexem.Remove** methods.

[C#]

```
//Removing Lexems from the language
this.editControl1.Language.Lexems.Remove(objconfigLex);

//Changing the lexems
objconfigLex = new ConfigLexem(this.TextBox1.Text, "", FormatType.Custom,
false);
objconfigLex.IndentationGuideline = true;
objconfigLex.FormatName = "HighLight";

//Add it to the current language's Lexems collection
this.editControl1.Language.Lexems.Add(objconfigLex);
```

```
//Reset the current configuration language cache to reflect these changes.  
this.editControl1.Language.ResetCaches();
```

[VB .NET]

```
'Removing Lexemes from the language  
Me.editControl1.Language.Lexems.Remove(objconfigLex)  
  
'Changing the lexems  
objconfigLex = New ConfigLexem(Me.TextBox1.Text, "", FormatType.Custom,  
False)  
objconfigLex.IndentationGuideline = True  
objconfigLex.FormatName = "HighLight"  
  
' Add it to the current language's Lexemes collection  
Me.editControl1.Language.Lexems.Add(objconfigLex)  
  
' Reset the current configuration language cache to reflect these changes.  
Me.editControl1.Language.ResetCaches()
```

5.3 How To Clear the Undo Buffer In Essential Edit

You can use the **ResetUndoInfo** method to clear the undo buffer, and save the changes to the underlying stream. This is done to make sure that the changes on the contents/actions recently performed cannot be undone.

The Following code snippet illustrates this.

[C#]

```
// Code to clear the Undo buffer  
this.editcontrol1.ResetUndoInfo();  
  
// Code to discard all the Unsaved changes  
this.editControl1.DiscardChanges();
```

[VB .NET]

```
' Code to clear the Undo buffer  
Me.editcontrol1.ResetUndoInfo()  
  
' Code to discard all the Unsaved changes  
Me.editControl1.DiscardChanges()
```

5.4 How To Convert Offset Values Into Text Range In the Edit Control

This section explains how to get the associated `CoordinatePoint` values from text offset values. This can be done as follows.

You have to convert offset values into `VirtualPoints`, and then `VirtualPoints` to `ParsePoints` before converting them to `CoordinatePoints`.

The following code snippet illustrates this:

[C#]

```
// Starting offset converted to virtual point.
Point startVirtualPoint =
this.editControl1.ConvertOffsetToVirtualPosition(startOffsetValue);

// Ending offset converted to virtual point.
Point endVirtualPoint =
this.editControl1.ConvertOffsetToVirtualPosition(endOffsetValue);

// Converting the VirtualPoints to ParsePoints.
ParsePoint startParsePoint = new ParsePoint(startVirtualPoint.Y,
startVirtualPoint.X, 0);
ParsePoint endParsePoint = new ParsePoint(endVirtualPoint.Y,
endVirtualPoint.X, 0);

// Creating the associated CoordinatePoints that indicate the text range.
CoordinatePoint startCoordinatePoint = new
CoordinatePoint(this.editControl1.Parser as ILexemParser, startParsePoint,
startVirtualPoint.Y, startVirtualPoint.X, true);
CoordinatePoint endCoordinatePoint = new
CoordinatePoint(this.editControl1.Parser as ILexemParser, endParsePoint,
endVirtualPoint.Y, endVirtualPoint.X, true);
```

[VB.NET]

```
' Starting offset converted to virtual point.
Dim startVirtualPoint As Point =
Me.EditControl1.ConvertOffsetToVirtualPosition(startOffsetValue)

' Ending offset converted to virtual point.
Dim endVirtualPoint As Point =
Me.EditControl1.ConvertOffsetToVirtualPosition(endOffsetValue)
```

```
' Converting the VirtualPoints to ParsePoints.  
Dim startParsePoint As New ParsePoint(startVirtualPoint.Y,  
startVirtualPoint.X, 0)  
Dim endParsePoint As New ParsePoint(endVirtualPoint.Y, endVirtualPoint.X, 0)  
  
' Creating the associated CoordinatePoints that indicate the text range.  
Dim startCoordinatePoint As New  
CoordinatePoint(TryCast(Me.editControl1.Parser, ILexemParser),  
startParsePoint, startVirtualPoint.Y, startVirtualPoint.X, True)  
Dim endCoordinatePoint As New CoordinatePoint(TryCast(Me.editControl1.Parser,  
ILexemParser), endParsePoint, endVirtualPoint.Y, endVirtualPoint.X, True)
```

5.5 How To Data Bind an Edit Control To a Datasource

The following code snippet illustrates how an Edit Control can be data-bound to a table in a DataSet.

[C#]

```
// Create a new DataSet.  
this.dataset = new DataSet("MyDataSet");  
  
// Create a new DataTable.  
this.table = new DataTable("MyDataTable");  
  
// Create a new DataColumn and add it to the DataTable.  
this.datacolumn = new DataColumn("Code",  
System.Type.GetType("System.String"));  
this.table.Columns.Add(this.datacolumn);  
  
// Create a new DataRow, and assign it to the specific column.  
// Assign a string value 'program' to that DataRow-DataColumn field.  
this.datarow = this.table.NewRow();  
this.datarow[this.datacolumn] = program;  
  
// Add this DataRow to the DataTable.  
this.table.Rows.Add(this.datarow);  
  
// Add this DataTable to the DataSet.  
this.dataset.Tables.Add(this.table);  
  
// Databinding EditControl.Text to the DataColumn "Code",  
// where "Code" contains the program to be displayed in the EditControl.
```

```
this.editControl1.DataBindings.Add("Text", this.dataset.Tables[0], "Code");
```

[VB.NET]

```
' Create a new DataSet.  
Me.dataset = New DataSet("MyDataSet")  
  
' Create a new DataTable.  
Me.table = New DataTable("MyDataTable")  
  
' Create a new DataColumn and add it to the DataTable.  
Me.datacolumn = New DataColumn("Code", System.Type.GetType("System.String"))  
Me.table.Columns.Add(Me.datacolumn)  
  
' Create a new DataRow, and assign it to the specific column.  
' Assign a string value 'program' to that DataRow-DataColumn field.  
Me.datarow = Me.table.NewRow()  
Me.datarow(Me.datacolumn) = program  
  
' Add this DataRow to the DataTable.  
Me.table.Rows.Add(Me.datarow)  
  
' Add this DataTable to the DataSet.  
Me.dataset.Tables.Add(Me.table)  
  
' Databinding EditControl.Text to the DataColumn "Code",  
' where "Code" contains the program to be displayed in the EditControl.  
Me.editControl1.DataBindings.Add("Text", Me.dataset.Tables(0), "Code")
```

5.6 How To Disable Keyboard Shortcuts For the Edit Control

To disable keyboard shortcuts, first you must remove them from the context menu. Here is an example for removing the F5 shortcut.

[C#]

```
private void Form1_Load(object sender, System.EventArgs e)  
{  
    ContextMenu cm = this.editControl1.ContextMenu;  
    foreach(MenuItem mi in cm.MenuItems)  
    {  
        this.RemoveShortcutInEditControl(mi);  
    }  
}
```

```
        }

    }

private void RemoveShortcutInEditControl(MenuItem miParent)
{
    // Remove F5 shortcut.
    if(miParent.Shortcut == Shortcut.F5)
        miParent.Shortcut = Shortcut.None;

    // Parse through the children recursively.
    foreach(MenuItem mi in miParent.MenuItems)
    {
        this.RemoveShortcutInEditControl(mi);
    }
}
```

[VB.NET]

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim cm As ContextMenu = Me.editControl1.ContextMenu
    Dim mi As MenuItem
    For Each mi In cm.MenuItems
        Me.RemoveShortcutInEditControl(mi)
    Next
End Sub

Private Sub RemoveShortcutInEditControl(ByVal miParent As MenuItem)
    ' Remove F5 shortcut.
    If miParent.Shortcut = Shortcut.F5 Then
        miParent.Shortcut = Shortcut.None
    End If

    ' Parse through the children recursively.
    Dim mi As MenuItem
    For Each mi In miParent.MenuItems
        Me.RemoveShortcutInEditControl(mi)
    Next
End Sub
```

5.7 How To Dynamically Add Configuration Settings At Runtime

The following code snippet illustrates how to create Custom Formats and define ConfigLexems that belong to those Formats.

[C#]

```
// Create a format and set its attributes
ISnippetFormat formatMethod = this.editControl1.Language.Add("MethodName");
formatMethod.FontColor = Color.HotPink;
formatMethod.Font = new Font("Garamond",12);

// Create a lexem that belongs to this format
ConfigLexem configLex = new ConfigLexem("Method[0-9]*", "", 
FormatType.Custom, false);
configLex.IsBeginRegex = true;
configLex.FormatName = "MethodName";

// Add it to the current language's lexems collection
this.editControl1.Language.Lexems.Add(configLex);
this.editControl1.Language.ResetCaches();
```

[VB.NET]

```
' Create a format and set its properties
Dim formatMethod As ISnippetFormat =
Me.editControl1.Language.Add("MethodName")
formatMethod.FontColor = Color.HotPink
formatMethod.Font = New Font("Garamond", 12)

' Create a lexem that belongs to this format
Dim configLex As New ConfigLexem("Method[0-9]*", "", FormatType.Custom,
False)
configLex.IsBeginRegex = True
configLex.FormatName = "MethodName"

' Add it to the current language's lexems collection
Me.editControl1.Language.Lexems.Add(configLex)
Me.editControl1.Language.ResetCaches()
```

5.8 How To Dynamically Validate Text Using the TextChanged Event

Text can be validated dynamically by using the **TextChanged** event and a Timer. The validation routine is invoked in response to a brief pause by the user while typing. The following code snippet illustrates this.

[C#]

```
private void editControl1_TextChanged(object sender, System.EventArgs e)
{
    // Do not start the timer as long as characters are being typed.
    if (this.timer1.Enabled == true) {
        this.timer1.Stop();
    }
    this.timer1.Start();
}

private void timer1_Tick(object sender, System.EventArgs e)
{
    this.ValidateText();

    this.timer1.Stop();
}

private void ValidateText()
{
    // Perform your validation logic here.
    MessageBox.Show(" Text Validated ");
}
```

[VB.NET]

```
Private Sub editControl1_TextChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles EditControl1.TextChanged
    ' Do not start the timer as long as characters are being typed.
    If Me.timer1.Enabled = True Then
        Me.timer1.Stop()
    End If
    Me.timer1.Start()
End Sub

Private Sub timer1_Tick(ByVal sender As Object, ByVal e As System.EventArgs)
Handles timer1.Tick
    Me.ValidateText()

    Me.timer1.Stop()
End Sub

Private Sub ValidateText()
    ' Perform your validation logic here.
    MessageBox.Show(" Text validated ")
```

```
End Sub
```

5.9 How To Format Keywords In the Contents Of the Edit Control Using Configuration Settings

Handle the **ConfigurationChanged** event of the Edit Control and get all the tokens of the "Format" keyword. Then, handle the **OnCustomDraw** event of each of these tokens and perform string manipulation operations. The following code snippet illustrates this.

[C#]

```
private void editControl1_ConfigurationChanged(object sender,
System.EventArgs e)
{
    foreach( FormatManager lang in editControl1.Languages )
    {
        Format format = lang[FormatType.KeyWord] as Format;
        if( format != null )
        {
            format.OnCustomDraw += new
            CustomSnippetDrawEventHandler(format_OnCustomDraw);
        }
    }
}

private void format_OnCustomDraw(object sender, CustomSnippetEventArgs e)
{
    string text = e.Text;
    text = text.ToLower();
    text = text[0].ToString().ToUpper() + text.Substring( 1, text.Length - 1 );
    e.Text = text;
}
```

[VB .NET]

```
Private Sub editControl1_ConfigurationChanged(ByVal sender As Object, ByVal e
As System.EventArgs) Handles EditControl1.ConfigurationChanged
    Dim lang As FormatManager
    For Each lang In editControl1.Languages
        Dim format As Format = lang(FormatType.KeyWord) '
        If Not (format Is Nothing) Then
            AddHandler format.OnCustomDraw, AddressOf format_OnCustomDraw
        End If
    
```

```
    Next lang
End Sub

Private Sub format_OnCustomDraw(ByVal sender As Object, ByVal e As
CustomSnippetEventArgs)
    Dim [text] As String = e.Text
    [text] = [text].ToLower()
    [text] = [text](0).ToString().ToUpper() + [text].Substring(1,
[text].Length - 1)
    e.Text = [text]
End Sub
```

Refer to the Keyword Formatting Demo sample for more information in this regard.

*..\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\Edit.Windows\\Samples\\2.0\\Text Formatting\\KeywordFormattingDemo*

5.10 How To Get a Count Of the Number Of Occurrences Of a String In the Edit Control

You can get a count of the number of occurrences of a string in the Edit Control using the **Matches** method of the **Regex** class.

[C#]

```
Regex r = new Regex(searchText, RegexOptions.IgnoreCase);
MatchCollection ma = r.Matches(this.editControl1.Text);
return ma.Count;
```

[VB.NET]

```
Dim r As Regex = New Regex(pattern, RegexOptions.IgnoreCase)
Dim ma As MatchCollection = r.Matches(Me.editControl1.Text)
return ma.Count
```

5.11 How To Get All the ConfigLexems In the Contents Of the Edit Control

The following code snippet illustrates how to get all the ConfigLexems in the contents of the Edit Control.

[C#]

```
private ArrayList GetLexems()
{
    ArrayList configLexemList = new ArrayList();
    for (int i=1; i<=this.editControl1.PhysicalLineCount; i++)
    {
        ILexemLine line = this.editControl1.GetLine(i);
        foreach (ILexem lexem in line.LineLexems)
        {
            IConfigLexem configLexem = lexem.Config;
            configLexemList.Add(configLexem);
        }
    }
    return configLexemList;
}
```

[VB.NET]

```
Private Function GetLexems() As ArrayList
    Dim configLexemList As ArrayList = New ArrayList()
    Dim i As Integer
    For i = 1 To Me.editControl1.PhysicalLineCount Step i + 1
        Dim line As ILexemLine = Me.editControl1.GetLine(i)
        Dim lexem As ILexem
        For Each lexem In line.LineLexems
            Dim configLexem As IConfigLexem = lexem.Config
            configLexemList.Add(configLexem)
        Next
    Next
    Return configLexemList
End Function
```

5.12 How To Get the Tokens In Each Line Of the Edit Control

You can get the tokens present in a line of the Edit Control by getting hold of the **ILexemLine** object associated with that particular line, and then accessing its Lexems in the **LineLexems** collection. The following code snippet illustrates this.

[C#]

```
ILexemLine lexemLine =
this.editControl1.GetLine(this.editControl1.CurrentLine);
foreach (Lexem lexem in lexemLine.LineLexems)
{
    lexemArrayList.Add(lexem);
}
```

[VB .NET]

```
Dim lexemLine As ILexemLine =
Me.editControl1.GetLine(Me.editControl1.CurrentLine)
Dim lexem As Lexem
For Each lexem In lexemLine.LineLexems
    lexemArrayList.Add(lexem)
Next
```

5.13 How To Implement VS.NET-like XML Tag Insertion Feature Using Edit Control

The VS.NET-like XML tag insertion feature can be used while editing XML language tags in Essential Edit. The cursor can be placed at any position of the line, and the nodes will be inserted exactly at the beginning and end of the current line.

This feature saves time while editing your XML documents by using Essential Edit. The following code snippet illustrates this.

[C#]

```
private void menuItem_Click(object sender, System.EventArgs e)
{
    this.inputDialog.ShowDialog();
    if (this.accepted == true)
    {
        if(this.inputString.Equals(""))
        {
            MessageBox.Show("The node name cannot be empty");
        }
        else
        {
```

```
this.editControl1.MoveToLineStart();
this.editControl1.InsertText(this.editControl1.CurrentLine, (this.editControl1.CurrentColumn), " ");

this.editControl1.InsertText(this.editControl1.CurrentLine, this.editControl1.CurrentColumn, "<" + this.inputString + ">");
this.editControl1.InsertText(this.editControl1.CurrentLine, (this.editControl1.CurrentColumn), " ");
this.editControl1.AppendText("</" + this.inputString + ">");

}

}

}
```

[VB.NET]

```
Private Sub menuItem_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles menuItem2.Click
Me.inputDialog.ShowDialog()
If Me.accepted = True Then
    If Me.inputString.Equals("") Then
        MessageBox.Show("The node name cannot be empty")
    Else
        Me.editControl1.MoveToLineStart()
        Me.editControl1.InsertText(Me.editControl1.CurrentLine,
        (Me.editControl1.CurrentColumn), " ")
        Me.editControl1.InsertText(Me.editControl1.CurrentLine,
        Me.editControl1.CurrentColumn, "<" + Me.inputString + ">")
        Me.editControl1.InsertText(Me.editControl1.CurrentLine,
        (Me.editControl1.CurrentColumn), " ")
        Me.editControl1.AppendText("</" + Me.inputString + ">")
    End If
End If
End Sub
```

5.14 How To Perform VS.NET-like Underlining For Offending Code In the Edit Control

Edit Control supports VS.NET-like wavy underlining of text. The wavy underlines can also have custom color and double lines. The following code snippet illustrates how you can implement this feature in Edit Control.

[C#]

```
// Starting offset converted to virtual point
```

```
Point startVirtualPoint =
this.editControl1.ConvertOffsetToVirtualPosition(startOffsetValue);

// Ending offset converted to virtual point
Point endVirtualPoint =
this.editControl1.ConvertOffsetToVirtualPosition(endOffsetValue);

// Converting the VirtualPoints to ParsePoints
ParsePoint startParsePoint = new ParsePoint(startVirtualPoint.Y,
startVirtualPoint.X, 0);
ParsePoint endParsePoint = new ParsePoint(endVirtualPoint.Y,
endVirtualPoint.X, 0);

// Creating the associated CoordinatePoints that indicate the text range
CoordinatePoint startCoordinatePoint = new
CoordinatePoint((ILexemParser)this.editControl1.Parser, startParsePoint,
startVirtualPoint.Y, startVirtualPoint.X, true);

CoordinatePoint endCoordinatePoint = new
CoordinatePoint((ILexemParser)this.editControl1.Parser, endParsePoint,
endVirtualPoint.Y, endVirtualPoint.X, true);

ISnippetFormat format = editControl1.RegisterUnderlineFormat(Color.Red,
UnderlineStyle.Wave, UnderlineWeight.Thick);
this.editControl1.SetUnderline (startCoordinatePoint, endCoordinatePoint,
format);
```

[VB.NET]

```
' Starting offset converted to virtual point
Dim startVirtualPoint As Point =
Me.editControl1.ConvertOffsetToVirtualPosition(startOffsetValue)

' Ending offset converted to virtual point
Dim endVirtualPoint As Point =
Me.editControl1.ConvertOffsetToVirtualPosition(endOffsetValue)

' Converting the VirtualPoints to ParsePoints
Dim startParsePoint As New ParsePoint(startVirtualPoint.Y,
startVirtualPoint.X, 0)
Dim endParsePoint As New ParsePoint(endVirtualPoint.Y, endVirtualPoint.X, 0)

' Creating the associated CoordinatePoints that indicate the text range
Dim startCoordinatePoint As New CoordinatePoint(CType(Me.editControl1.Parser,
ILexemParser), startParsePoint, startVirtualPoint.Y, startVirtualPoint.X,
True)

Dim endCoordinatePoint As New CoordinatePoint(CType(Me.editControl1.Parser,
```

```
ILexemParser), endParsePoint, endVirtualPoint.Y, endVirtualPoint.X, True)

Dim format As ISnippetFormat =
editControl1.RegisterUnderlineFormat(Color.Red, UnderlineStyle.Wave,
UnderlineWeight.Thick)
Me.editControl1.SetUnderline(startCoordinatePoint, endCoordinatePoint,
format)
```

5.15 How To Plug-in an External Configuration Dile Into the Edit Control

The Edit Control supports the creation and plug-in of custom configuration files into the Edit Control for syntax coloring. The configuration file has to be in XML format, and as per the directions in the Configuration Settings section. The following code snippet illustrates how to plug-in an external configuration file.

[C#]

```
// Plug-In an external configuration file.
this.editControl1.Configurator.Open(" Configuration_File.xml ");
```

[VB.NET]

```
' Plug-In an external configuration file.
Me.editControl1.Configurator.Open(" Configuration_File.xml ")
```

5.16 How To Programmatically Display the Code Snippets

You can display the code snippets programmatically by using the **StreamEditControl** class of Edit Control. The following code snippet illustrates this.

[C#]

```
private void editControl1_ReadOnlyChanged(object sender, EventArgs e)
{
    edit = (StreamEditControl)sender;
}

private void menuItem15_Click(object sender, EventArgs e)
{
    edit.ShowCodeSnippets();
```

```
}
```

[VB .NET]

```
Private Sub editControl1_ReadOnlyChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles editControl1.ReadOnlyChanged
    edit = CType(sender, StreamEditControl)
End Sub

Private Sub MenuItem9_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MenuItem9.Click
    edit.ShowCodeSnippets()
End Sub
```

5.17 How To Set Different Background Colors For the Lines In the Edit Control

Edit Control provides support for custom coloring the background of the lines in the Edit Control. The following code snippet illustrates how to set the back color for the entire line and selected text.

[C#]

```
private IBackgroundFormat RegisterFormat()
{
    Color background = Color.Empty, foreground = Color.Empty;
    if (radioButton1.Checked)
        background = radioButton1.BackColor;
    else if (radioButton2.Checked)
        background = radioButton2.BackColor;
    else if (radioButton3.Checked)
        background = radioButton3.BackColor;
    if (radioButton6.Checked)
        foreground = radioButton6.BackColor;
    else if (radioButton5.Checked)
        foreground = radioButton5.BackColor;
    else if (radioButton4.Checked)
        foreground = radioButton4.BackColor;
    bool bUseHatchFille = comboBox1.SelectedIndex > 0;
    HatchStyle style = (bUseHatchFille) ?
        (HatchStyle)Enum.Parse(typeof(HatchStyle),
        comboBox1.SelectedItem.ToString()) : HatchStyle.Min;
    IBackgroundFormat format =
```

```
editControl1.RegisterBackColorFormat(background, foreground, style,
bUseHatchFille);
return format;
}

//code for setting line back color for the entire line
private void button1_Click(object sender, System.EventArgs e)
{
    IDynamicFormat[] temp =
editControl1.GetLineBackColorFormats(editControl1.CurrentLine);
IBackgroundFormat format = RegisterFormat();
editControl1.SetLineBackColor(editControl1.CurrentLine, true, format);
}

private void Form1_Load(object sender, System.EventArgs e)
{
    //comboBox1.Items.Clear();
comboBox1.Items.Add("Solid");
foreach (string name in Enum.GetNames(typeof(HatchStyle)))
comboBox1.Items.Add(name);
comboBox1.SelectedText = "Percent05";
comboBox1.SelectedIndex = 0;
editControl1.Text += "\n";
editControl1.CurrentLine = 1;
}

//code to set the back color for a selected text
private void button2_Click(object sender, System.EventArgs e)
{
    IBackgroundFormat format = RegisterFormat();
editControl1.SetSelectionBackColor(format);
}
```

[VB.NET]

```
Private Function RegisterFormat() As IBackgroundFormat
    Dim background As Color = Color.Empty
    Dim foreground As Color = Color.Empty
    If radioButton1.Checked Then
        background = radioButton1.BackColor
    ElseIf radioButton2.Checked Then
        background = radioButton2.BackColor
    ElseIf radioButton3.Checked Then
        background = radioButton3.BackColor
    End If
    If radioButton6.Checked Then
```

```
foreground = radioButton6.BackColor
ElseIf radioButton5.Checked Then
    foreground = radioButton5.BackColor
ElseIf radioButton4.Checked Then
    foreground = radioButton4.BackColor
End If
Dim bUseHatchFille As Boolean = comboBox1.SelectedIndex > 0
Dim style As HatchStyle
If bUseHatchFille = True Then
    style = CType([Enum].Parse(GetType(HatchStyle),
    comboBox1.SelectedItem.ToString()), HatchStyle)
Else
    style = HatchStyle.Min
End If
Dim format As IBackgroundFormat =
editControl1.RegisterBackColorFormat(background, foreground, style,
bUseHatchFille)
Return format
End Function 'RegisterFormat

' code to set back color for the entire line
Private Sub button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles button1.Click
    Dim temp As IDynamicFormat() =
editControl1.GetLineBackColorFormats(editControl1.CurrentLine)
    Dim format As IBackgroundFormat = RegisterFormat()
    editControl1.SetLineBackColor(editControl1.CurrentLine, True, format)
End Sub 'button1_Click

Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles MyBase.Load
    comboBox1.Items.Clear()
    comboBox1.Items.Add("Solid")
    Dim name As String
    For Each name In [Enum].GetNames(GetType(HatchStyle))
        comboBox1.Items.Add(name)
    Next name
    comboBox1.SelectedText = "Percent05"
    comboBox1.SelectedIndex = 0
    editControl1.Text += vbCrLf
    editControl1.CurrentLine = 1
End Sub 'Form1_Load

' Code to set back color for the selected text
Private Sub button2_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles button2.Click
    Dim format As IBackgroundFormat = RegisterFormat()
```

```
    editControl1.SetSelectionBackColor(format)
End Sub
```

5.18 How To Suspend And Resume Painting Of the Edit Control

The following code snippet illustrates how to suspend and resume painting of the EditControl.

[C#]

```
this.editControl1.SuspendPainting();
this.editControl1.ResumePainting();
```

[VB .NET]

```
Me.editControl1.SuspendPainting()
Me.editControl1.ResumePainting()
```

Index

1

1 Overview 9

 1.1 Introduction To Essential Edit 9

 1.2 Prerequisites and Compatibility 11

 1.3 Documentation 12

2

2 Installation and Deployment 14

 2.1 Installation 14

 2.2 Sample and Location 14

 2.3 Deployment Requirements 17

 2.3.1 Toolbox Entries 17

 2.3.2 DLLs 17

3

3 Getting Started 18

 3.1 Control Structure 18

 3.2 Creating an Edit Control 18

 3.2.1 Through Designer 19

 3.2.2 Through Code 20

4

4 Concepts And Features 23

 4.1 Configuration Settings 23

 4.1.1 Creating a Custom Language Configuration File 23

 4.1.2 Creating Configuration Settings Programmatically 27

 4.10 Status Bar 225

 4.11 Printing 228

 4.12 Performance 234

 4.13 Localization and Globalization 235

 4.14 Edit Control Events 238

 4.14.1 CanUndoRedoChanged Event 238

 4.14.10 Indicator Margin Events 252

- 4.14.10.1 IndicatorMarginClick Event 252
- 4.14.10.2 IndicatorMarginDoubleClick Event 253
- 4.14.10.3 DrawLineMark Event 254
- 4.14.11 InsertModeChanged Event 255
- 4.14.12 LanguageChanged Event 256
- 4.14.13 MenuFill Event 257
- 4.14.14 Operation Events 257
- 4.14.14.1 OperationStarted Event 257
- 4.14.14.2 OperationStopped Event 258
- 4.14.15 Outlining Events 258
- 4.14.15.1 OutliningBeforeCollapse Event 259
- 4.14.15.2 OutliningBeforeExpand Event 259
- 4.14.15.3 OutliningCollapse Event 260
- 4.14.15.4 OutliningExpand Event 261
- 4.14.15.5 OutliningTooltipBeforePopup Event 262
- 4.14.15.6 OutliningTooltipClose Event 262
- 4.14.15.7 OutliningTooltipPopup Event 262
- 4.14.16 Print Events 263
- 4.14.16.1 PrintHeader Event 263
- 4.14.16.2 PrintFooter Event 263
- 4.14.17 ReadOnlyChanged Event 264
- 4.14.18 RegisteringDefaultKeyBindings Event 264
- 4.14.19 RegisteringKeyCommands Event 265
- 4.14.2 Closing Event 239
- 4.14.20 Save Events 265
- 4.14.20.1 SaveFileWithDataLoss Event 265
- 4.14.20.2 SaveStreamWithDataLoss Event 266
- 4.14.21 Scroll Events 266
- 4.14.21.1 HorizontalScroll Event 267
- 4.14.21.2 VerticalScroll Event 267
- 4.14.22 SelectionChanged Event 268
- 4.14.23 SingleLineChanged Event 269
- 4.14.24 Text Events 269
- 4.14.24.1TextChanged Event 270
- 4.14.24.2 TextChanging Event 270

- 4.14.24.3 Line Modification Events 271
- 4.14.24.3.1 Line Changed 272
- 4.14.24.3.2 Line Inserted 273
- 4.14.24.3.3 Line Deleted 274
- 4.14.25 UnreachableTextFound Event 274
- 4.14.26 UpdateBookmarkToolTip Event 275
- 4.14.27 UpdateContextToolTip Event 277
- 4.14.28 User Margin Events 277
 - 4.14.28.1 DrawUserMarginText Event 277
 - 4.14.28.2 PaintUserMargin Event 277
 - 4.14.29 WordWrapChanged Event 278
- 4.14.3 Code Snippet Events 239
 - 4.14.3.1 CodeSnippetActivating Event 239
 - 4.14.3.2 CodeSnippetDeactivating Event 240
 - 4.14.3.3 CodeSnippetTemplateTextChanging Event 241
 - 4.14.3.4 NewSnippetMemberHighlighting Event 242
- 4.14.4 ConfigurationChanged Event 243
- 4.14.5 Collapse Events 243
 - 4.14.5.1 CollapsedAll Event 243
 - 4.14.5.2 CollapsingAll Event 244
- 4.14.6 ContextChoice Events 245
 - 4.14.6.1 ContextChoiceBeforeOpen Event 245
 - 4.14.6.2 ContextChoiceSelectedTextInsert Event 246
 - 4.14.6.3 ContextChoiceClose Event 246
 - 4.14.6.4 ContextChoiceItemSelected Event 246
 - 4.14.6.5 ContextChoiceUpdate Event 246
 - 4.14.6.6 ContextChoiceOpen Event 248
 - 4.14.6.7 ContextChoiceRightClick Event 248
- 4.14.7 ContextPrompt Events 249
 - 4.14.7.1 ContextPromptBeforeOpen Event 249
 - 4.14.7.2 ContextPromptClose Event 249
 - 4.14.7.3 ContextPromptOpen Event 249
- 4.14.7.4 ContextPromptSelectionChanged Event 249
- 4.14.7.5 ContextPromptUpdate Event 249
- 4.14.8 CursorPositionChanged Event 250
- 4.14.9 Expand Events 250
 - 4.14.9.1 ExpandedAll Event 250
 - 4.14.9.2 ExpandingAll Event 251
- 4.2 Editing Features 30
 - 4.2.1 Undo / Redo Actions 30
 - 4.2.2 New Line Styles 34
 - 4.2.3 Clipboard Operations 34
 - 4.2.3.1 EnableMD5 36
 - 4.2.4 Keystroke - Action Combinations Binding 37
 - 4.2.5 Regular Expressions 40
 - 4.2.5.1 Language Elements 41
 - 4.2.5.2 Lexical Macros 45
 - 4.2.6 Block Indent and Outdent 47
 - 4.2.7 Right-To-Left (RTL) Support 49
- 4.3 Code Completion 51
 - 4.3.1 AutoComplete Support 51
 - 4.3.2 AutoReplace Triggers 53
- 4.4 Text Visualization 55
 - 4.4.1 Text Navigation 55
 - 4.4.1.1 Positions and Offsets 59
 - 4.4.10 Break Points 90
 - 4.4.11 Text Formatting 92
 - 4.4.11.1 Bracket Highlighting and Indentation Guidelines 92
 - 4.4.11.2 Auto Indentation 97
 - 4.4.11.2.1 Lexem Support for AutoIndent Block Mode 99
 - 4.4.11.3 AutoFormatting 100
 - 4.4.11.4 Unicode 102
 - 4.4.11.5 Automatic Outlining 103
 - 4.4.11.5.1 Outlining Tooltip 106
 - 4.4.11.6 Wordwrap 108

4.4.11.6.1 Wordwrap Margin Customization and Wrapping Images 112	4.6.6.2.2 Context Choice 159
4.4.11.7 Read-Only Text 116	4.6.6.2.3 Context Prompt 168
4.4.12 Customizing Text 118	4.6.6.2.4 Context ToolTip 178
4.4.12.1 Text Color 118	4.6.6.2.3 Custom Cursor 183
4.4.12.2 Text Border 118	4.6.6.2.4 Intellimouse Scrolling 184
4.4.12.3 Encoding Text 120	4.6.6.2.5 Drag-and-drop 185
4.4.12.4 Text Selection 122	4.7 Text Export 186
4.4.2 Column Guides 63	4.7.1 XML, RTF and HTML Export 186
4.4.3 Content Dividers 65	4.7.2 Schema Definition File for XML Syntax Coloring Configuration File 188
4.4.4 Underlines, Wavelines and StrikeThrough 67	4.8 File Sharing and Stream Handling 188
4.4.5 Text Handling 70	4.8.1 Creating, Loading, Saving And Dropping Files 189
4.4.5.1 Appending, Deleting and Inserting Multiple Lines of Text 71	4.8.2 Loading And Saving Contents 192
4.4.6 Spaces and Tabs 75	4.8.3 Saving And Cancelling Changes 194
4.4.6.1 WhiteSpace Indicators 78	4.8.4 File Sharing 198
4.4.7 Line Numbers and Current Line Highlighting 81	4.8.5 Lexical Analysis And Semantic Parsing 198
4.4.8 Bookmarks and Custom Indicators 84	4.8.6 Clearing/Flushing Saved Changes 200
4.4.9 Comments 89	4.9 Appearance 201
4.5 Syntax Highlighting and Code Coloring 126	4.9.1 Visual Settings 201
4.5.1 XML Based Configuration Files 132	4.9.1.1 Size 201
4.5.2 Multiple Language Syntax Highlighting 134	4.9.1.2 Split Views 202
4.6 Runtime Features 135	4.9.1.3 Applying Themes 205
4.6.1 Insert Mode 135	4.9.1.4 Border Style 206
4.6.2 Keyboard Shortcuts 136	4.9.1.5 Graphics Customization Settings 207
4.6.3 Bitmap Generation 138	4.9.2 Margins 208
4.6.4 Find, Replace and Goto 139	4.9.2.1 Selection Margin 208
4.6.5 Enhanced Find Dialog 145	4.9.2.2 User Margin 211
4.6.6 Scrolling Support 146	4.9.3 Background Settings 214
4.6.6.1 Office 2007 Visual Style 150	4.9.4 Font Customization 220
4.6.6.1.1 ToolTip 151	4.9.5 Single Line Mode 222
4.6.6.2 Interactive Features 152	4.9.6 Customizable Find Dialog 223
4.6.6.2.1 Customizable Context Menu 152	5
4.6.6.2.2 IntelliPrompt Features 156	5 Frequently Asked Questions 280
4.6.6.2.2.1 Code Snippets 156	

- 5.1 How To Access the Text Associated With Individual Lines In the Selected Text Region Of the Edit Control 280
- 5.10 How To Get a Count Of the Number Of Occurrences Of a String In the Edit Control 290
- 5.11 How To Get All the ConfigLexems In the Contents Of the Edit Control 290
- 5.12 How To Get the Tokens In Each Line Of the Edit Control 291
- 5.13 How To Implement VS.NET-like XML Tag Insertion Feature Using Edit Control 292
- 5.14 How To Perform VS.NET-like Underlining For Offending Code In the Edit Control 293
- 5.15 How To Plug-in an External Configuration Dile Into the Edit Control 295
- 5.16 How To Programmatically Display the Code Snippets 295
- 5.17 How To Set Different Background Colors For the Lines In the Edit Control 296
- 5.18 How To Suspend And Resume Painting Of the Edit Control 299
- 5.2 How To Change the Lexems Dynamically 281
- 5.3 How To Clear the Undo Buffer In Essential Edit 282
- 5.4 How To Convert Offset Values Into Text Range In the Edit Control 283
- 5.5 How To Data Bind an Edit Control To a Datasource 284
- 5.6 How To Disable Keyboard Shortcuts For the Edit Control 285
- 5.7 How To Dynamically Add Configuration Settings At Runtime 286
- 5.8 How To Dynamically Validate Text Using the TextChanged Event 287
- 5.9 How To Format Keywords In the Contents Of the Edit Control Using Configuration Settings 289