Fast Colored TextBox for Syntax Highlighting

Pavel Torgashov

24 Oct 2014 LGPL3

Rate me: 4.97/5 (860 votes)

Custom text editor with syntax highlighting

Source and demo on GitHub

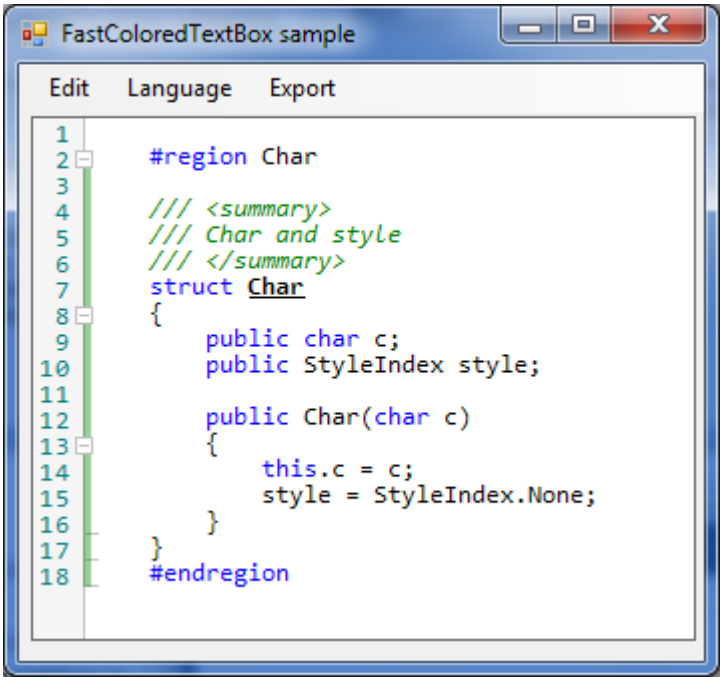
Download demo

Download help (CHM)

Nuget package

Download IronyFCTB (example of integration FCTB with parsing engine)

Also: Download source and demo for .NET Compact Framework 2.0



Introduction

For one of my projects, I have felt the need of a text editor with syntax highlighting. At first, I used a component inherited from `RichTextBox`, but while using it for a large amount of text I found out that `RichTextBox` highlights

very slowly a large number of colored fragments (from 200 and more). When such highlighting has to be made in a dynamic way, it causes a serious problem.

Therefore I created my own text component which uses neither Windows `TextBox` nor `RichTextBox`.

The rendering of a text was made completely only by the means of GDI+.

The component works fast enough with a large amount of text and also possesses tools to make comfortably dynamic syntax highlighting.

It has such settings as foreground color, font style, background color which can be adjusted for arbitrarily selected text symbols. One can easily gain access to a text with the use of regular expressions. WordWrap, Find/Replace, Code folding and multilevel Undo/Redo are supported as well.

Implementation

For storage of characters of text, structure `Char` is used:

C# Copy Code

```
public struct Char
{
    public char c;
    public StyleIndex style;
}
```

The structure keeps the symbol (`char`, 2 bytes) and style index mask (`StyleIndex`, 2 bytes). Thus, on each character of text consumes 4 bytes of memory. Symbols are grouped into lines, which are implemented using `List<Char>`.

`StyleIndex` is mask of styles indices, applied to this character. Each bit of `StyleIndex` means that this symbol will be drawn by appropriate style. Because `StyleIndex` has 16 bits, control supports no more than 16 different styles.

Styles are stored in a separate list:

C# Copy Code

```
public readonly Style[] Styles = new Style[sizeof(ushort)*8];
```

In fact, `Style` is renderer of chars, backgrounds, borders and other design elements of the text.
Below is a typical implementation of one of the styles for rendering text characters:

C# Shrink ▲ Copy Code

```
public class TextStyle : Style
{
    public Brush ForeBrush { get; set; }
    public Brush BackgroundBrush { get; set; }
    public FontStyle FontStyle { get; set; }

    public override void Draw(Graphics gr, Point position, Range range)
    {
        //draw background
        if (BackgroundBrush != null)
            gr.FillRectangle(BackgroundBrush, position.X, position.Y,
        (range.End.iChar - range.Start.iChar) *
        range.tb.CharWidth, range.tb.CharHeight);
        //draw chars
        Font f = new Font(range.tb.Font, FontStyle);
        Line line = range.tb[range.Start.iLine];
        float dx = range.tb.CharWidth;
```

```
        float y = position.Y - 2f;
        float x = position.X - 2f;

        Brush foreBrush = this.ForeBrush ?? new SolidBrush(range.tb.ForeColor);

        for (int i = range.Start.iChar; i < range.End.iChar; i++)
        {
            //draw char
            gr.DrawString(line[i].c.ToString(), f, foreBrush, x, y);
            x += dx;
        }
    }
}
```

`TextStyle` contains foreground color, background color and font style of the text. When creating a new style, component checks style on its list, and if there is no style, it creates a new style, with its index.

You can create custom styles, inherited from `Style` class.

To work with fragments of text, the class `Range` was used, representing a continuous block of text, given the initial and final positions:

C# Copy Code

```
public class Range
{
    Place start;
    Place end;
}
public struct Place
{
    int iLine;
    int iChar;
}
```

Using the Code

Syntax highlighting

Unlike `RichTextBox`, the component does not use RTF. The information about the color and type of symbols is kept only in the component. It means that the coloring of the component has to be redone every time when entering text. In this case, the event `TextChanged` is applied.

A `Range` object which contains the information about modified text range pass into the event `TextChanged`. It permits the highlighting of the altered text fragment only.

For the search of fragments of text which need to be colored, it is possible to employ overloaded method `Range.SetStyle()` which accepts search pattern (regular expression). For example, the following code can be used for the search and coloring of the comments of C# code (the part of the line starting from two forward slashes):

C# Copy Code

```
Style GreenStyle = new TextStyle(Brushes.Green, null, FontStyle.Italic);
...
private void fastColoredTextBox1_TextChanged(object sender, TextChangedEventArgs e)
{
    //clear style of changed range
    e.ChangedRange.ClearStyle(GreenStyle);
```

```
    //comment highlighting
    e.ChangedRange.SetStyle(GreenStyle, @"//.*$", RegexOptions.Multiline);
}
```

Before beginning the coloring call, the method `Range.ClearStyle()` is used to clean out and delete the previous style.

The method `SetStyle()` highlights the text fragment corresponding to a regular expression. However, if the expression includes the named group "`range`", the group with a name "`range`" is highlighted. The name of the class which comes after the key words "*`class`*", "*`struct`*" and "*`enum`*" was bolded in the following example:

C# Copy Code

```
e.ChangedRange.SetStyle(BoldStyle, @"\b(class|struct|enum)\s+(?<range>[\w_]+?)\b");
```

The event handler `TextChanged` utilized for coloring C#, VB, HTML and other languages syntax was implemented in demo application.

Apart from the event `TextChanged` the events `TextChanging`, `VisibleRangeChanged` and `SelectionChanged` may happen to be useful. The event `TextChanging` appears before the text starts to be modified. The event `SelectionChanged` occurs after the change of the cursor position in the component or while a selected fragment of text is being modified.

Code Folding

Control allows to hide blocks of text. To hide the selected text, use method `CollapseBlock()`:

C# Copy Code

```
fastColoredTextBox1.CollapseBlock(fastColoredTextBox1.Selection.Start.iLine,
                fastColoredTextBox1.Selection.End.iLine);
```

The result is shown in the picture:

The component supports automatic search for fragments of collapse (folding area). To set the pattern (Regex) to find the beginning and end of folding block, use method `Range.SetFoldingMarkers()` in `TextChanged` handler.

For example, to search of blocks *`{..}`* and *`#region .. #endregion`*, use next handler:

C# Copy Code

```
private void fastColoredTextBox1_TextChanged(object sender, TextChangedEventArgs e)
{
    //clear folding markers of changed range
    e.ChangedRange.ClearFoldingMarkers();
    //set folding markers
    e.ChangedRange.SetFoldingMarkers("{", "}");
    e.ChangedRange.SetFoldingMarkers(@"#region\b", @"#endregion\b");
}
```

The result is shown in the picture:

Folding blocks can be nested into each other.

Collapsed block can be opened by doubleclick on it, or click on marker '+'. Single click on folded area selects hidden block. Also, you can open hidden block programmatically by `ExpandBlock()` method.

Demo application contains sample for collapse all *`#region...#endregion`* blocks of the text.

In addition to hiding the text, folding blocks help visually define the boundaries of the block where the caret is located. For this purpose, the left side of the control draws a vertical line (folding indicator). It shows the beginning and end of the current folding block, in which the caret is located.

Custom Code Folding

Code folding using regex does not always bring the desired result.

For more specific cases - you can directly set the start and end markers blocks: `Line.FoldingStartMarker` and `Line.FoldingEndMarker`.

Following code example makes folding block for selected range:

C# Copy Code

```
const string marker = "myMarker";

var currentSelection = fctb.Selection.Clone();
currentSelection.Normalize();

if (currentSelection.Start.iLine != currentSelection.End.iLine)
{
    fctb[currentSelection.Start.iLine].FoldingStartMarker = marker;
    fctb[currentSelection.End.iLine].FoldingEndMarker = marker;
    fctb.Invalidate();
}
```

Note:

- FoldingMarker is identifier and has no relation to text content.
- `FoldingStartMarker` and `FoldingEndMarker` must be identical for folding block.
- The FCTB supports nested blocks with same FoldingMarker name.
- Also it supports two strategies of finding of blocks, use property `FindEndOfFoldingBlockStrategy`.

More complex example of custom code folding see in `CustomFoldingSample`.

Delayed Handlers

Many events (`TextChanged`, `SelectionChanged`, `VisibleRangeChanged`) have a pending version of the event. A deferred event is triggered after a certain time after the occurrence of major events.

What does this mean? If the user enters text quickly, then the `TextChanged` is triggered when you enter each character. And event `TextChangedDelayed` works only after the user has stopped typing. And only once.

It is useful for lazy highlighting of large text.

Control supports next delayed events: `TextChangedDelayed`, `SelectionChangedDelayed`, `VisibleRangeChangedDelayed`. Properties `DelayedEventsInterval` and `DelayedTextChangedInterval` contain time of pending.

Export to HTML

Control has property `Html`. It returns HTML version of colored text. Also you can use `ExportToHTML` class for more flexibility of export to HTML. You can use export to HTML for printing of the text, or for coloring of the code of your web-site.

Clipboard

Control copies the text in two formats - Plain text and HTML.
If the target application supports inserting HTML (e.g. Microsoft Word) then will be inserted colored text. Otherwise (such as Notepad) will be inserted plain text.

Hotkeys

The control supports following hotkeys:

- Left, Right, Up, Down, Home, End, PageUp, PageDown - moves caret
- Shift+(Left, Right, Up, Down, Home, End, PageUp, PageDown) - moves caret with selection
- Ctrl+F, Ctrl+H - shows Find and Replace dialogs
- F3 - find next
- Ctrl+G - shows GoTo dialog
- Ctrl+(C, V, X) - standard clipboard operations
- Ctrl+A - selects all text
- Ctrl+Z, Alt+Backspace, Ctrl+R - Undo/Redo opertions
- Tab, Shift+Tab - increase/decrease left indent of selected range
- Ctrl+Home, Ctrl+End - go to first/last char of the text
- Shift+Ctrl+Home, Shift+Ctrl+End - go to first/last char of the text with selection
- Ctrl+Left, Ctrl+Right - go word left/right
- Shift+Ctrl+Left, Shift+Ctrl+Right - go word left/right with selection
- Ctrl+-, Shift+Ctrl+- - backward/forward navigation
- Ctrl+U, Shift+Ctrl+U - converts selected text to upper/lower case
- Ctrl+Shift+C - inserts/removes comment prefix in selected lines
- Ins - switches between Insert Mode and Overwrite Mode

- Ctrl+Backspace, Ctrl+Del - remove word left/right
- Alt+Mouse, Alt+Shift+(Up, Down, Right, Left) - enables column selection mode
- Alt+Up, Alt+Down - moves selected lines up/down
- Shift+Del - removes current line
- Ctrl+B, Ctrl+Shift-B, Ctrl+N, Ctrl+Shift+N - add, removes and navigates to bookmark
- Esc - closes all opened tooltips, menus and hints
- Ctrl+Wheel - zooming
- Ctrl+M, Ctrl+E - start/stop macro recording, executing of macro
- Alt+F [char] - finds nearest [char]
- Ctrl+(Up, Down) - scrolls Up/Down
- Ctrl+(NumpadPlus, NumpadMinus, 0) - zoom in, zoom out, no zoom
- Ctrl+I - forced AutoIndentChars of current line

Note: You can change hotkey mapping. Use `Hotkeys` property in design mode and `HotkeysMapping` in runtime.

Brackets Highlighting

Control has built-in brackets highlighting. Simply set properties `LeftBracket` and `RightBracket`. If you want to disable brackets highlighting, set it to '`\x0`'. For adjust color of highlighting, use property `BracketsStyle`. For adjusting of time of pending of highlighting, change `DelayedEventsInterval`.

Interactive Styles

You can create own interactive(clickable) styles. To do this, derive your class from the `Style`, and call `AddVisualMarker()` from your overridden `Draw()` method. To handle a click on the marker, use events`FastColoredTextBox.VisualMarkerClick` or `Style.VisualMarkerClick` or override method `Style.OnVisualMarkerClick()`.
Also you can use built-in style `ShortcutStyle`. This class draws little clickable rectangle under last char of range.

Styles Priority

Each `char` can contain up to 16 different styles. Therefore, the matter in which order these styles will be drawn. To explicitly specify the order of drawing, use the method `FastColoredTextBox.AddStyle()`:

C# Copy Code

```
fastColoredTextBox1.AddStyle(MyUndermostStyle);
fastColoredTextBox1.AddStyle(MyUpperStyle);
fastColoredTextBox1.AddStyle(MyTopmostStyle);
```

This methods must be called before any calls of `Range.SetStyle()`. Otherwise, the draw order will be determined by the order of calls of methods `Range.SetStyle()`.

Note: By default, control draws only one `TextStyle`(or inherited) style - undermost from all. However, you can enable the drawing of the symbol in many `TextStyle`, using the property `FastColoredTextBox.AllowSeveralTextStyleDrawing`. This applies only to `TextStyle`(or inherited) styles, other styles(inherited from `Style`) are drawn in any case.

If `char` has not any `TextStyle`, it will drawing by `FastColoredTextBox.DefaultStyle`. `DefaultStyle` draws over all other styles.

Call method `ClearStyleBuffer()` if you need reset order of drawing.

Also, to adjust the look of text, you can apply a semitransparent color in your styles.

Built-in highlighter

`FastColoredTextBox` has built-in syntax highlighter for languages: C#, VB, HTML, SQL, PHP, JS, XML, Lua.
Note: You can create own syntax highlighter for any language.

Property `HighlightingRangeType` specifies which part of the text will be highlighted as you type (by built-in highlighter). Value `ChangedRange` provides better performance. Values `VisibleRange` and `AllTextRange` - provides a more accurate highlighting (including multiline comments), but with the loss of performance.

Multiline Comments Highlighting

If your custom language supports multiline comments or other multiline operators, you may encounter a problem highlighting such operators. Property `ChangedRange` from `TextChanged` event contains only changed range of the text. But multiline comments is larger than `ChangedRange`, from what follows incorrect highlighting. You can solve this problem in the following way: In `TextChanged` handler use `VisibleRange` or `Range` property of `FastColoredTextBox` instead of `ChangedRange`. For example:

C# Copy Code

```
private void fastColoredTextBox1_TextChanged(object sender, TextChangedEventArgs e)
{
    //old edition
    //Range range = e.ChangedRange;

    //new edition
    Range range = (sender as FastColoredTextBox).VisibleRange;//or (sender as
                        //FastColoredTextBox).Range

    //clear style of changed range
    range.ClearStyle(GreenStyle);
    //comment highlighting
    range.SetStyle(GreenStyle, @"//.*$", RegexOptions.Multiline);
    range.SetStyle(GreenStyle, @"(/\*.*?\*/)|(/\*.*)", RegexOptions.Singleline);
    range.SetStyle(GreenStyle, @"(/\*.*?\*/)|(.*\*/)", RegexOptions.Singleline |
                RegexOptions.RightToLeft);
}
```

Note: Using `VisibleRange` instead of `ChangedRange` decreases performance of the control.
Note: If you use built-in highlighter, you can adjust property `HighlightingRangeType`.

Using XML Syntax Descriptors

Component can use the XML file for syntax highlighting. File name specified in the property `DescriptionFile`.

For highlighting, you need to set the `Language` property to `Custom`.

The file may contain information about styles, rules of syntax highlighting, folding parameters and brackets.
Below is a list of valid tags and attributes to them:

- `<doc>...</doc>` - root XML node.
- `<brackets left="..." right="..." left2="..." right2="..." />` - sets the brackets for highlighting
- `<style name="..." color="..." backColor="..." fontStyle="..." />` - sets the style called name. Tag `<style>` creates only styles of type `TextStyle`. `color` and `backColor` determine foreground and background color. Allowed as a `string` color name or hex representation of the form #RGB or #ARGB. `fontStyle` - enumeration of `FontStyle` parameters.
  The sequence tags `<style>` determines the order of rendering these styles.
- `<rule style="..." options="...">regex pattern</rule>` - sets rule of highlighting. `style` - style name described in the tags `<style>`. `options` - enumeration of `RegexOptions` parameters. The contents of the tag - regex pattern for highlighting.

- `<folding start="..." finish="..." options="...">` - specifies the rules for folding. `start` and `finish` set regular expressions to the beginning and end of the block. `options` - enumeration of `RegexOptions` parameters.

Below is an example XML file for syntax highlighting HTML:

XML Copy Code

```
<doc>
    <brackets left="&lt;" right="&gt;" />
    <style name="Maroon" color="Maroon" fontStyle="Bold,Italic" />
    <style name="Blue" color="Blue"/>
    <style name="Red" color="Red" backColor="#f5f5e5" />
    <rule style="Blue">&lt;|/&gt;|&lt;/|&gt;</rule>
    <rule style="Maroon">&lt;(?&lt;range&gt;[!\w\d]+)</rule>
    <rule style="Maroon">&lt;/(?&lt;range&gt;[\w\d]+)&gt;</rule>
    <rule style="Red" options="Multiline">(?&lt;range&gt;\S+?)='[^']*
        '|(?&lt;range&gt;\S+)="[^"]*"|(?&lt;range&gt;\S+)=\S+</rule>
    <folding start="&lt;div" finish="&lt;/div&gt;" options="IgnoreCase"/>
</doc>
```

Backward/Forward Navigation

Control remembers lines in which the user has visited. You can return to the previous location by pressing [Ctrl + -] or call `NavigateBackward()`. Similarly, [Ctrl + Shift + -] and the method `NavigateForward()`.
You can also use the property `Line.LastVisit` for information on the last visit to this line. In the example `PowerfulCSharpEditor` this property is used for navigation in multitab mode, when you have multiple documents open.

Hieroglyphs and Wide Characters

Control supports input and display hieroglyphs and other wide characters (CJK languages, Arabic and other). Also Input Method Editor (IME) is supported.
To enable this feature to switch the property `ImeMode` to the state On.
Demo contains a sample of IME features usage.
Note: For a normal display wide characters may require a larger font size.
Note: Enabled IME mode can decrease performance of control.

Autocomplete

The library has class `AutocompleteMenu` to implement autocompleting functionality (like IntelliSense popup menu). `AutocompleteMenu` contains a list of `AutocompleteItem`. It supports displaying, filtering and inserting of items. You can use `AutocompleteMenu` for code snippets, keywords, methods and properties hints.

Notice to such properties and methods as `Fragment`, `SearchPattern`, `MinFragmentLength`, `Items.ImageList`, `Items.SetAutocompleteItems()`.

You can override class `AutocompleteItem` for more flexibility of functionality.

Also, you can use more advanced control AutocompleteMenu[^]. `AutocompleteMenu` is fully compatible with FastColoredTextBox.

See `AutocompleteSample` and `AutocompleteSample2` for more information.

AutoIndent

When the user is typing, and enabled `AutoIndent`, control automatically determines the left indent for the input string.

By default, the indentation made on the markers of Code Folding. But, if you select a specific `Language`, then padding makes integrated highlighter for appropriate language.

You can set their own rules for indentation.
Use the event `AutoIndentNeeded`.
The handler must return two values: `Shift` and `ShiftNextLines`.

`Shift` value indicates indention of this line relative to the previous line (in characters, can be negative).
`ShiftNextLines` value indicates a indent will be applied to the subsequent lines after this.

The pictures show some possible variants:

and:

Example of `AutoIndentNeeded` handler shown in **AutoIndentSample**.

Note: If you handle `AutoIndentNeeded` event, built-in highlighter's indention will disabled.

Also, you can manually make `AutoIndent` of selected text. Simply call `DoAutoIndent()` method. The below picture shows text before calling:

And this is picture after `DoAutoIndent()` calling:

AutoIndentChars

The control can align the position of the individual characters in adjacent lines. This functionality is called AutoIndentChars.

An example is shown below:

The property `AutoIndentCharsPatterns` contains a set of patterns (in each line - one pattern).

When the user types a character, line is checked against each pattern. If the string suitable for some of the patterns are checked near lines (the left indent these lines must match the indentation of the current line). And then these symbols of found lines are aligned by inserting whitespaces. Which symbols will be aligned is defined by named groups of regex, name of group must be "range". See example:

In this example will be aligned symbol "=" and text after its.

To deactivate AutoIndentChars set property `AutoIndentChars` to false (by default it is true). Built-in syntax highlighter already contains needed patterns for AutoIndentChars, so you shouldn't to define it.

You can make AutoIndentChars programmatically by method `DoAutoIndentChars()`. Also, user can call AutoIndentChars by hotkey Ctrl+I (It will worked even if AutoIndentChars = false).

Custom AutoIndentChars see in example `AutoIndentCharsSample`.

Printing

The component has built-in printing functionality. For print all text or selected range, call method `Print()`. You can specify which dialog boxes to show, by passing the object `PrintDialogSettings`. By default, the method makes printing of all text, without dialog boxes.

Lazy file loading

To work with files of extremally large size (100k lines and more) can be a useful "lazy" mode. In this mode, the control opens the file and reads its parts, as needed. Unused pieces of text are deleted from memory. This mode is supported by three methods:
`OpenBindingFile()` - Opens file for reading (in exclusive mode).
`CloseBindingFile()` - Closes opened file. After call the control returns to simple (non lazy) mode.
`SaveToFile()` - Saves text to the file. After this method, control will be binded to new file. This method can be used in simple (non lazy) mode too.

Split-screen mode

The control has property `SourceTextBox` of type `FastColoredTextBox`. If you set this property to another textbox, both the controls will display the same text. This mode can be used to split the screen, where you can edit the same text in different places.

Column selection mode

The component supports column selection mode. When its activated, user can work with vertical fragment of the text:

To enable column selection mode, press Alt key and select area by mouse. Another way - press Alt+Shift and select area by arrow keys. To switch off this mode simply click on the control or press any arrow key.

Column selection mode supports text inputting (include Del and Backspace, but exclude insert or delete lines), clipboard operations, clear selection, undo/redo operations.

Arrow keys(Right, Left, Up, Down, etc) switches off column mode.

Also, if you call methods `Text{set;}` or `AppendText`, column mode switches off. Methods `InsertText` and `SelectedText` do not change selection mode.

If you want prorgammatically enable/disable selection mode, use property `Selection.ColumnSelectionMode`.

Wordwrap mode does not support column selection.

Bookmarks

The control has built-in bookmarks. You can use property `Bookmarks` to get access to bookmarks. Also, user can use hot-keys `Ctrl-B`, `Ctrl-Shift-B`, `Ctrl-N` to add, remove and navigate to bookmark.

Note: Bookmarks are working even for virtual modes (lazy loading mode, custom `TextSource`, etc)

ToolTips

The component supports tooltips. For this you need to handle event `ToolTipNeeded` and pass parameters for tooltip. `ToolTipNeeded` is fired when user moves mouse over text. After the mouse is stopped, after `ToolTipDelay` ms `ToolTipNeeded` will be called and will be showed popup tooltip.

Hints

The component supports built-in hints. This is powerful feature allows you to insert hints into text.

The hint can contain simple text or arbitrary control. The hint is linked to range and move with it when user scrolls textbox. If user presses `Esc` or changes text all hints are removing.

Class `Hint` has several modes of displaying. Property `Dock` allows to place hint on whole line. If property `Inline` is True, hint will be built in the text, otherwise the hint will be located over text.

Also you can adjust colors of background, borders, fonts etc. If user clicks on the hint, event `HintClick` will be called. To create and show hint use collection `Hints`. Also the `FastColoredTextBox` has auxiliary methods to add and remove hints: `AddHint()` and `ClearHints()`.

Call method `Hint.DoVisible()` to scroll textbox to the hint.

Macro recording

The FCTB supports recording of macros. When recording is activated, FCTB remembers all pressed keys. Later you can execute recorded sequence.

Macros remember any entered symbols and control keys (include Ctrl-C, Ctrl-V, Home, End, etc). But macros do not support selection by mouse.

The user can start/stop recording pressed Ctrl-M. For executing macro - Ctrl-E.

For more efficiency of macros introduced the function 'char finding'. Press Alt-F and any char. Entered char will be found in the text, and caret will be placed immediately after found char.

You can get access to the macros infrastructure via property `MacrosManager`. There you can control macros programmatically.

WordWrap indenting

Property `WordWrapAutoIndent` automatically shifts secondary wordwrap lines on the shift amount of the first line:

Property `WordWrapIndent` specifies additional shift for secondary wordwrap lines (in chars):

Properties WordWrapAutoIndent and WordWrapIndent can operate simultaneously. Of course, they are working on WordWrap mode only.

Note: By default, WordWrapAutoIndent is on. If you need classical wordwrapping, switch it off.

Layout and colors

Below shown some layout and color properties:

Compact Framework Version

The control works under .NET Compact Framework 2.0. This means that you can use it for mobile platforms (PocketPC, Smartphones, PDA etc).

CF version supports all features except the following:

- IME mode and wide characters are not supported.
- `AutocompleteMenu` is not supported.
- Built-in printing is not supported.

The remaining features are presented, only slightly different from the full version.

Samples

Demo application has many samples. Below is a brief description:

- **Powerful sample**. Contains many features: syntax highlighting, code folding, export, same words highlighting and other.
- **PowerfulCSharpEditor.** Powerful multitab C# source files editor. Supports highlighting, multitab backward/forward navigation, dynamic methods list, autocomplete and other.

- **Simplest syntax highlighting sample**. Shows how to make simplest syntax highlighting.
- **Marker sample**. Shows how to make marker tool. Sample uses class `ShortcutStyle` for create clickable markers on text area:

- **Custom style sample**. This example shows how to create own custom style. Next custom style draws frame around of the words:

C# Copy Code

```
class EllipseStyle : Style
{
    public override void Draw(Graphics gr, Point position, Range range)
    {
        //get size of rectangle
        Size size = GetSizeOfRange(range);
        //create rectangle
        Rectangle rect = new Rectangle(position, size);
        //inflate it
        rect.Inflate(2, 2);
        //get rounded rectangle
        var path = GetRoundedRectangle(rect, 7);
        //draw rounded rectangle
        gr.DrawPath(Pens.Red, path);
    }
}
```

- **VisibleRangeChangedDelayed usage sample**. This example shows how to highlight syntax for extremely large text by `VisibleRangeChangedDelayed` event.
- **Simplest code folding sample**. This example shows how to make simplest code folding.
- **Custom code folding sample**. This example shows how to make costom code folding (on example of Python).
- **Autocomplete sample, Autocomplete sample 2**. Examples show how to create autocomplete functionality:

- **Tooltip sample.** It shows tooltips for words under mouse.
- **Dynamic highlighting sample**. Shows how to make dynamic syntax highlighting. This example finds the functions declared in the program and dynamically highlights all of their entry into the code of LISP.
- **Syntax highlighting by XML Description.** This example uses XML file for description syntax highlighting.
- **IMEmode sample.**This example supports IME entering mode and rendering of wide characters.

- **Image drawing sample.** Sample shows how to draw images instead of text:

This example supports animated GIF too.

- **AutoIndent sample.** Sample shows how to make custom autoindent functionality.
- **BookmarksSample.** Sample shows how to make bookmarks functionality.
- **Logger sample.** It shows how to add text with predefined style:

- **Split sample.** It shows how to make split-screen mode:

- **Lazy loading sample.** It shows how to work with files of extremally large size.
- **Console sample.** It shows how to create console emulator.
- **HintSample.** It shows how to use hints.
- **ReadOnlySample.** The example shows how to create readonly blocks of the text.
- **MacrosSample.** This sample shows how to use macros for hard formatting of the code.
- **PredefinedStylesSample.** Here we create large readonly text with predefined styles, hyperlinks and tooltips.
- **DocumentMapSample.** Shows document map feature.

- **DiffMergeSample.**

- **Joke sample.** It contains some additional features, implemented custom `TextStyle`:

Performance

For storing one megabyte of text requires approximately 6 MB of RAM (include undo/redo stack objects). The coloring does not consume significant resources.

The use of regular expressions and saving memory usage, allow to reach high performance component. I tested the file of 50,000 lines (about 1.6 MB) of C# code. The total time of insertion, and the syntax coloring was about 3 seconds. Further work with the text passed without significant delays.

Restrictions

The component does not support center or right alignment and uses only monospaced fonts. Also tabs are always replaced by spaces.

History

- 24 Feb 2011 - First release.
- 26 Feb 2011 - Added Find/Replace functionality, indent stuff, showing of line numbers. Also added VB syntax highlighting.
- 28 Feb 2011 - Added code folding functionality (include current block highlighting). Added some features (caret blinking, increase/decrease indent of selected text). Optimized `ReplaceAll` functionality. Added HTML syntax highlighting. Increased general performance of control.
- 2 Mar 2011 - Style logic was revised. Added HTML export. Added many samples. Added many features...
- 3 Mar 2011 - Increased performance. Fixed some bugs. Added SQL, PHP syntax highlighting examples. Brackets highlighting is built-in now. Some hotkeys was added.
- 4 Mar 2011 - `FastColoredTextBox` now supports built-in syntax highlighter for languages: C#, VB, HTML, SQL, PHP.
- 8 Mar 2011 - Added XML syntax descriptors. Fixed some bugs of font rendering.
- 14 Mar 2011 - Added Backward/Forward navigation. Added IME mode and CJK languages support. Added powerful C# multitab editor sample.
- 25 Mar 2011 - Added following features: colored text copy, auto indent for new line, insert/remove comment prefix for selected lines, uppercase/lowercase transformation of selected text. Fixed scroll flicker. Fixed some bugs. Added image drawing sample.
- 7 Apr 2011 - Added `AutocompleteMenu` class. `Autocomplete` sample was revised. Some other features was added.
- 15 Apr 2011 - `AutoIndent` was revised. Added printing functionality.
- 30 May 2011 - Control was downgraded from FW3.5 to FW2.0.
- 8 Jun 2011 - Added .NET Compact Framework 2.0 version. Added some features. Some bugs was fixed. Performance was increased.
- 18 Jun 2011 - License was changed from GPLv3 to LGPLv3. Now you can use control in proprietary software.
- 29 Jun 2011 - Changed behaviour of AutoIndent, Home key, word selection (Really, I made the PHP editor for myself, and discovered that these functions are inconvenient. Added some methods, events and properties (`LineInserted`, `LineRemoved`, `Line.UniqueId`). Added `BookmarksSample`. Built-in highlighter was improved. Fixed some flicker.
- 17 Aug 2011 - Improved support for multiline comments and operators (see property `HighlightingRangeType`). Fixed some bugs.
- 6 Jan 2012 - Some bugs was fixed. Autocomplete menu was improved.
- 4 Feb 2012 - Added split-screen mode, lazy file loading, overwrite mode. Performance was increased. Some bugs was fixed.
- 13 Mar 2012 - Added some properties and samples. Some bugs was fixed.
- 30 Apr 2012 - Bugfixing, small changes of keys behaviour, selecting and autoindent.
- 2 May 2012 - Column selection mode was added.
- 26 Sep 2012 - Some bugs was fixed. Changed some key's behaviour. Added printing of line numbers. Added bilingual sample. Added hotkeys for remove current line and move selected lines up/down.
- 03 Nov 2012 - Added properties: `AutoIndentExistingLines` (You can disable autoindenting for existing lines), `VirtualSpace` (In VirtualSpace mode user can set cursor into any position of textbox), `FindEndOfFoldingBlockStrategy` (You can adjust strategy of code folding). Added line selection

mode(When user drags mouse over left margin of textbox, corresponding lines will be selected). Fixed some bugs with scrolling. Added `HyperlinkSample`.

- 23 Jan 2013 - The control supports built-in Bookmarks, Tooltips and Hints. Event `OnPaste` was added. Fixed bugs with localization and binding to the `Text` property. `AutocompleteMenu` appearing was improved. Also behaviour of methods `IncreaseIndent` and `DecreaseIndent` was improved. Now you can change color scheme of built-in syntax highlighter (see properties `KeywordStyle`, `StringStyle`, etc.). Useful method `GetStylesOfChar()` was added. SQL highlighting was extended. Some samples was improved. Sample `HintSample` was added. Drag&drop is available now.
- 14 Feb 2013 - The control supports readonly blocks, zooming. Samples `ReadOnlyBlocksSample` and `PredefinedStylesSample` were added. Source codes of the component are available on GitHub now.
- 26 Feb 2013 - Macros are supported now, `MacrosSample` was added. Implemented features: border around text area, char finding. Indenting was improved. Scrolling by mouse was improved.
- 03 Mar 2013 - Added hotkey mapping. Now you can change hotkeys of FTCB. Also zooming and scrolling were improved.
- 07 Nov 2013 - Document map was added. Many features were improved.
- 13 Dec 2013 - DiffMerge sample was added. Intergation with Irony example was added.
- 23 Dec 2013 - Properties `WordWrapAutoIndent` and `WordWrapIndent` were added. Custom wordwrap mode was added (see sample `CustomWordWrappingSample`). Custom outside scrollbars are supported now (see `CustomScrollBarSample`).
- 29 Mar 2014 - Brackets autocomplete was added. Added method `OpenFile` with autodetecting of file encoding.
- 12 Jul 2014 - `AutoIndentChars` functionality was added (details see in article). `AutoIndentCharsSample` was added. Built-in highlighter supports `Lua` and `XML` now.
- 25 Oct 2014 - Nuget package was added.

License

Share

About the Author

Pavel Torgashov

Software Developer Freelancer

Ukraine

I am Pavel Torgashov, and I live in Kyiv, Ukraine.
I've been developing software since 1998.
Main activities: processing of large volumes of data, statistics, computer vision and graphics.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Copy line to string
jan binnendijk 4-Apr-22 12:49

Arrow cursor over highlighted text - Here's how
chrisbray 12-Mar-22 9:22

My vote of 5
TheGEZ 15-Feb-22 2:56

Use of tabs
Víctor Aranda 10-Feb-22 23:27

Re: Use of tabs
chrisbray 12-Mar-22 9:32

Without a doubt
warra warra 20-Dec-21 23:59

Custom Regex for highlightning
ABJ Bahadır Back 10-Dec-21 19:19

"Pling" sound after using a hotkey/shortcut on Windows 10
Member 15411159 28-Oct-21 5:27

changing the color of some words
Member 14872487 29-Aug-21 7:20

Brackets Highlighting Problems
Member 14597331 17-Aug-21 8:33

Coloured text slightly above other text.
Joseph Guenther 8-Jun-21 7:10

Slow to render the text
Skeletro 3M 24-Mar-21 5:28

KeyPress event
jdevries007 23-Oct-20 0:53

Re: KeyPress event
Member 14523550 11-Nov-20 6:07

Search on mulltiple FastColoredTextBox
Skeletro 3M 15-Oct-20 0:03

Message Removed
17-Aug-20 21:33

Problem on highlighting text
Xyzdesk 5-Jun-20 23:55

LGPLv3 question
Lyr 30-May-20 6:58

Problems on using Copy-Paste
AndyHo 6-May-20 9:23

XML DescriptionFile for XML
Skeletro 3M 5-May-20 2:44

Re: XML DescriptionFile for XML
Xyzdesk 5-Jun-20 23:56

My vote of 5
DavidLanham 27-Apr-20 13:28

Uppercase Text on Keypress
faaqui 4-Feb-20 5:21

Re: Uppercase Text on Keypress
The Magical Magikarp 11-Feb-20 21:21

Re: Uppercase Text on Keypress
Member 14523550 11-Nov-20 6:16

Refresh 1 2 3 4 5 6 7 8 9 10 11 Next ▷

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.