Lazarus

The SynEdit Bible

**DATA SHEET**

|  |  |
| --- | --- |
| **AUTHOR** | Tito Hinostroza – Lima Peru |
| **DATE** | Rev7 completed on 11/12/2015 |
| **APPLICABLE TO** | Lazarus SynEdit Package 1.0.12  The examples have been developed on Windows-32. |
| **DOCUMENT LEVEL** | Half. Knowledge of Free Pascal and Lazarus is assumed. |
| **PREVIOUS DOCUMENTS** | None |
| **BIBLIOGRAPHY** | SynEdit source code - Lazarus  SynEdit source code - SourceForge  <http://forum.lazarus.freepascal.org/>  <http://wiki.freepascal.org/SynEdit/es>  <http://wiki.freepascal.org/SynEdit> |

CHANGE CONTROL

|  |  |  |
| --- | --- | --- |
| **VERSION** | **DATE** | **DESCRIPTION MODIFICATIONS** |
| Rev 1 | 10/12/2013 | By Tito Hinostroza.  First complete revised version of the documentation.  It remains to be documented:   * The other controls in the SynEdit package. * More detailed operation in column mode. (smCurrent) * The use of plugins. * The autocomplete. |
| Rev2 | 10/12/2013 | By Tito Hinostroza  The syntax of some examples has been corrected.  Section 1.4.1 was expanded and the graph was corrected.  Added information to Section 1.4.4 |
| Rev3 | 10/19/2013 | By Tito Hinostroza  Added Appendix and included information about the “hash” algorithm used in the highlighter implementation in Lazarus.  Section 2.3.6 was modified.  Added section 1.7.2  Section 2.4 was modified |
| Rev4 | 10/27/2013 | By Tito Hinostroza  The properties table was passed to the end  Added section 1.4.9 about the “Options” and “Options2” properties  Information was added to section 1.4.2 and “Typography” section was added.  Reordered syntax highlighting section 2 and completed the introduction.  Added information about more SynEdit properties. |
| Rev5 | 01/26/2014 | By Tito Hinostroza  Corrects some words with errors in Section 1.3  Section 1.4.2 was modified  Several sections were modified and completed.  The “Modify content” section was created  Added information about creating attributes in 2.3.4  Section 2.4 is added.  Additional information related to code folding is added. |
| Rev6 | 04/05/2014 | Corrected typographical errors.  Added information about the TSynCustomFoldHighlighter and TSynCustomHighlighter classes.  More properties and methods are included in section 1.9.  Section 1.5.1 is created |
| Rev7 | 10/25/2017 | Information is added to section 2.3.4.  Added more information about editor coordinates and some additional SynEdit properties.  Information about “plugins” is stored. |

*“In the beginning, it was TECO and VI”*

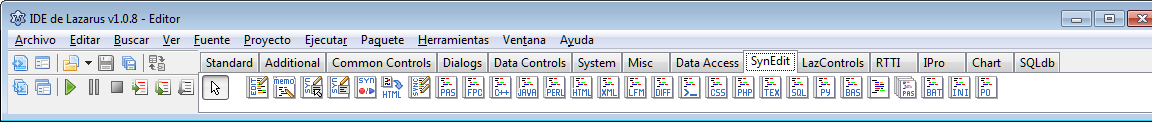
# Syntax-aware editor: SynEdit.

Much of this work is based on user experience, the limited documentation that exists on the web, reverse engineering, and the analysis of the source code of the SynEdit component.

## What is SynEdit?

It is a component or control that is integrated into the Lazarus environment. It's an editing control. It allows you to quickly implement text editors, with advanced features such as syntax highlighting.

To be exact, SynEdit is a whole package that is already integrated into Lazarus when it is installed (and that includes various components), but generally, when we say SynEdit, we are referring to the TSynEdit component, which is the editor with syntax highlighting capabilities .



It can be accessed through the “SynEdit” tab of the components palette. The TSynEdit editor is on the left. In this tab there are also various controls related to “TSynEdit”:

* TSynMemo.- Version of TSynEdit with some differences. It has fewer published methods and events. Derived from SynEdit. It can replace SynEdit, in many cases.
* TSynCompletion.- Non-visible control that allows implementing the “Code Completion” option.
* TSynAutoComplete.- Non-visible control that allows implementing the “Auto-Code Completion” option.
* TSynPasSyn.- Syntax component of the Pascal language.
* TSynFreePascalSyn.- Syntax component of the Free Pascal language.
* TSynCppSyn.- Syntax component of the C++ language.
* TSynJavaSyn.- Syntax component of the Java language.
* etc

The SynEdit control, which is included in Lazarus, is a modified version of the standalone SynEdit project. The version adapted for Lazarus has been developed from version 1.03, to which some additional features have been added, such as support for UTF-8 and Code Folding.

This component is well reviewed and tested, since it is the same one used by the Lazarus IDE for its Code Editor.

Unfortunately there is not enough technical documentation about the project, but what is known is that it is functional and performs very well.

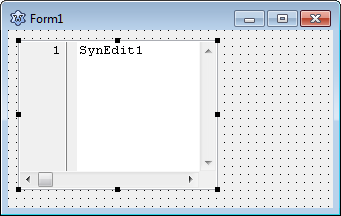
## SynEdit Features

The Lazarus SynEdit (TSynEdit) component has the following features:

* Component accessible from the Lazarus IDE.
* It does not require additional files (as is the case with Scintilla). Once integrated into the project, it is integrated into the code without any dependencies.
* Its code is completely accessible and modifiable.
* Works completely in UTF-8 encoding.
* Supports syntax coloring for several predefined languages or you can create a new syntax.
* Supports code completion and autocompletion options.
* Supports code folding. But it must be done by code.
* Includes “Undo” and “Redo” options, with extensive memory of changes.
* Contains methods for Search and Replace.
* Supports simple column selection.
* Allows you to number the lines.
* Supports highlighters and text markers.

## Appearance

By adding the TSynEdit component to the form, it is now operational. You can run the program and see that the editor responds like any TMemo type text box.



The main visual difference is in the vertical bar that appears on the left. This bar is used to show the line number, and other options. Another difference is that the horizontal font size is uniform. That is, the letter “m” has the same width as the letter “l”. This is the font that is loaded by default in a “SynEdit”.

Initially SynEdit does not include syntax highlighting options, because it does not have any associated syntax, yet. What it does include by default is the detection of “brackets”, that is, it highlights the parentheses that open and close, if the cursor is placed in one of the parentheses. The behavior is similar with square brackets, braces, and quotes. Apostrophes are not recognized.

This highlighting consists, by default, of putting the initial and final characters in bold.

texto **(**texto entre paréntesis (otro texto)**)** más texto.

To disable this feature, the “eoBracketHighlight” option must be removed from the “Options” property.

If you want to modify the delimiter highlighting attribute, you can use the following code:

SynEdit1.BracketMatchColor.Foreground **:=** clRed **;** **//** changes to red

Another feature that comes by default in SynEdit is the option to create bookmarks (See 1.8.5- Text markers). IF it is not going to be used, this option must be disabled because it could generate errors at run time.

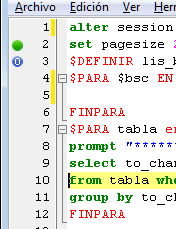
The Cut, Copy and Paste options are also enabled by default in the SynEdit control, without the need to implement them.

In general, all the shortcuts that are created by default in SynEdit correspond to actions that are predefined without the need to activate them.

There are several properties to change the appearance of the SynEdit control. We will describe some of them.

### Vertical Panel

The vertical panel that appears on the left side of the control is called “Gutter”, and is intended to display the line number, folding marks, change marks and markers.



Gutter

Marcas de “folding”

Marcadores

Número de línea

Marcas de cambio

The “Gutter” can be shown or hidden by code. To make it invisible you should do:

SynEdit1.Gutter.Visible **:=** **False ;**

In this case, our editor has the default name that is assigned when adding it to a form: SynEdit1.

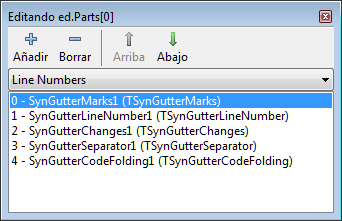
The “Gutter”, by default, has its width adjusted automatically, that is, it changes according to the number of rows in the editor. It can be set to a certain width by first setting the “Autosize” property to “false”:

ed.Gutter.AutoSize **:= false ;**

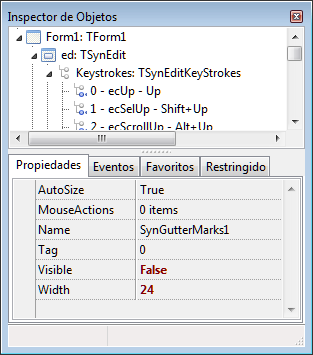
ed.Gutter.Width **:=** 30 **;**

It is not advisable to change the width in this way, because in this way, the elements it contains are not relocated, so part of the numbers or “folding” marks could be lost from sight.

It is preferable to leave the “Autosize” on “true” and disable individual elements of the “Gutter”, to vary its size. This can be easily done with the object inspector, by modifying the “Parts” property of the “Gutter” property:



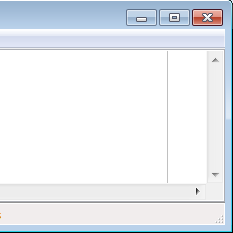
And then hiding the desired element:



This example hides the area intended for markers. As areas are hidden, the total size of the “Gutter” decreases.

### Right margin

By default in SynEdit, a vertical line appears on the right side of the text, usually in column 80. This line is helpful when you want to print the content and want to avoid exceeding the line size allowed by the printer.



To change the position you must modify the “RightEdge” property:

editor.RightEdge:= 100; //fixes position of the vertical line

You can also change its color using the “RightEdgeColor” property.

If you do not want this line to appear, you can set its position to a negative coordinate:

editor.RightEdge **:=** **-** 1 **;** **//** hide vertical line

Or it can be disabled using the “eoHideRightMargin” option, from the “Options” property:

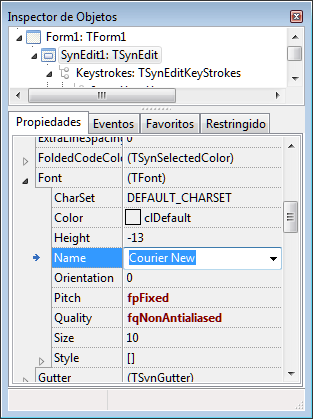
editor.Options **:=** editor.Options **+** [eoHideRightMargin] **;**

The editor scroll bars can be hidden or shown using the “ScrollBars” property.

### Typography

SynEdit allows you to configure various properties of the font to use. By default the text is displayed with the “Courirer New” font in size 10.

To change the font that will be used without SynEdit, the Font object must be configured. This task can be done by code or using the object inspector:

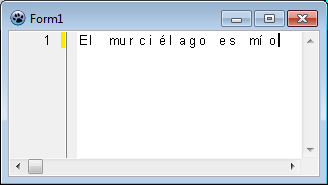


The Font object has various properties and methods, many of which can only be accessed by code.

Perhaps the most common property to change the appearance of text on the screen is the font. This can be changed using the “Name” property.

Consider that the characters to be displayed in SynEdit are always monospaced, meaning that all characters will have the same width on the screen. If a font with a different width will be used for each character, SynEdit will display it the same as a monospaced font, giving the impression that the characters are not equally spaced.

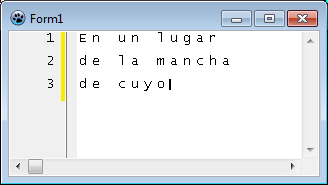
The following example shows an editor in which the “Arial” font, which is not monospaced, has been used:



The text has been correctly written and separated, but since the letters in “Arial” have different widths (the “m” is wider than the “i”), the text appears to be poorly spaced. Because of this effect, it is recommended to only use fonts that are uniform width, such as Courier, Fixed, or Lucida.

The font size is defined by modifying the “Size” property and the color with the “color” property. The “Style” property allows you to define the attributes: bold, underlined and italic.

It is also possible to change the spacing between lines and between characters, using the “ExtraCharSpacing” and “ExtraLineSpacing” properties:



The previous figure has been obtained using the following code:

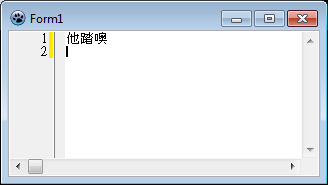
SynEdit1.ExtraCharSpacing **:=** 5 **;**

SynEdit1.ExtraLineSpacing **:=** 10 **;**

By default the spacing is zero. If you want to join, instead of separate, you can use negative values for these properties.

Another property that we can use to customize SynEdit is the character set (CharSet).

The character set allows you to change the language to be used in the editor. For example, using the CHINESEBIG5\_CHARSET character set, we could use Chinese characters:



Some character sets, like the one in the example, use twice as much space for one character in the editor as traditional Western characters. No problem, SynEdit can handle these types of characters and even combine single and double space characters (or two different sets of characters) in the same document.

## Functioning

### Editor Coordinates

For SynEdit, the screen is a grid of cells, where each cell represents a character [[1]](#footnote-1):

Celda

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| (1,1) | (2.1) | ... |  |  |  |  |  |
| (1,2) | (2,2) |  |  |  |  |  |  |
| ... |  |  |  |  |  |  | Cada celda está representada por su coordenada (fila, columna) |
|  |  |  |  |  |  |  |  |
|  |  |  | (x,y) |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

On the other hand, the information to be displayed on the screen is stored in a list of strings, where each string represents a line.

To handle both representations, two types of coordinates are used in a SynEdit:

* **Physical Coordinates.** Refers to the position in which a character appears on the screen, assuming that the screen is divided into cells of equal width.
* **Logical Coordinates.** Referring to the position of the byte (or bytes) that represents the character, in the string.

This difference is noticeable, especially due to the fact that SynEdit handles UTF-8 encoding, which complicates the management of coordinates on the screen.

Strings are stored as sequences of bytes, but what is displayed on the screen are sequences of characters located in cells.

The correspondence of bytes to characters is not 1 to 1 [[2]](#footnote-2).

* One byte in the string can represent more than one character on screen [[3]](#footnote-3). This is true when using tabs and they must be expanded into multiple spaces.
* A character on the screen can be represented by more than one byte in the string. This happens because SynEdit handles the universal UTF-8 encoding, which in some cases (such as stressed vowels), assigns more than one byte per character.

The following schematic shows how a typical text string is encoded in SynEdit:



The tab stop is represented as 4 spaces, but can vary depending on the editor settings. It should also be noted that in UTF-8, the character “ú” is represented by the bytes $C3 and $BA [[4]](#footnote-4). Another way of looking at this correspondence would be like this:



You can clearly see how the logical coordinates are different from the physical coordinates. Here we can find that the logical X coordinate of the letter “c” is 7, but its physical coordinate is 10.

Vertically, the logical and physical Y coordinate are always the same, so no transformations will have to be made.

The cursor always works in physical coordinates. The cursor coordinates are found in the CaretX, and CaretY properties.

CaretX goes from 1 to the end of the line. CaretY goes from 1 to the number of lines.

For example, to position the cursor on the fifth character of the second line, we would do:

SynEdit1.CaretX **:=** 5 **;**

SynEdit1.CaretY **:=** 2 **;**

You can also use the CaretXY property, which includes the two coordinates X and Y in a structure of type TPoint:

**var** Pos **:** TPoint **;**

...

Pos.x **:=** 5 **;**

Pos.y **:=** 2 **;**

SynEdit1.CaretXY **:=** Pos **;** **//** Equivalent to SynEdit1.CaretXY **:=** Point **(** 5,2 **);**

You can also access the logical coordinates of the cursor using the property: SynEdit1.LogicalCaretXY. So if what we have are logical coordinates, to correctly position the cursor in SynEdit, we can do this:

SynEdit1.LogicalCaretXY **:=** Point **(** 5, 2 **);**

To perform transformations between logical and physical coordinates, there is a group of transformation functions:

SynEdit1.LogicalToPhysicalCol **();**

SynEdit1.LogicalToPhysicalPos **();**

SynEdit1.PhysicalToLogicalCol **();**

SynEdit1.PhysicalToLogicalPos **();**

Usually we will not need these functions unless SynEdit contains UTF-8 characters of more than one byte, or tabs, because usually the logical and physical coordinates coincide.

Let's consider an example where we have content with checked vowels (they are encoded with 2 bytes in UTF-8). If in our editor we have the following text in the first line:

"icon"

And we want to obtain the third character (which must be the letter “o”). To get the actual position of the character in the string we must access the position:

SynEdit1.CaretX **:=** 3 **;**

SynEdit1.CaretY **:=** 1 **;**

xReal **:=** SynEdit1.PhysicalToLogicalPos **(** SynEdit1.CaretXY **)** .x

In xReal, we will get the actual position of the character within the string, which in our case is 4.

On certain occasions it may be useful to know the coordinates of the cursor in pixels. In this case the properties must be used:

SynEdit1.CaretXPix

SynEdit1.CaretYPix

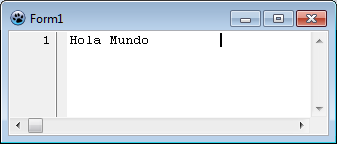
The CaretXPix coordinate is measured from the left edge of the control (including the width of the Vertical Panel or Gutter) and not from the editable area of the editor.

### Cursor handling

As we already saw, the position of the cursor is fixed with the CaretX and CaretY properties, but there are certain limits regarding the location of the cursor.

In practice it can be said that there is no practical limit to the size that a line can have in SynEdit, unless you want to exceed 2 billion. But the cursor has more restrictions.

A strange way SynEdit works is that it can place the cursor beyond the line boundaries. For example, if the line is only 10 characters wide, the cursor could be placed at position 20:



Cursor Flotante

This strange cursor placement can be achieved either by keyboard or by mouse.

I call this effect “Floating Cursor” and it is an unusual feature in text editors since the normal thing is that the cursor cannot be placed beyond the limits of the line.

This effect is only valid horizontally since vertically the cursor will always be limited by the number of lines that exist in the text, or in other words, the Y coordinate of the cursor cannot be greater than the number of lines.

To limit the horizontal position of the cursor, within the physical boundaries of the current line, this setting should be used:

SynEdit1.Options **:=** SynEdit1.Options **+** [eoKeepCaretX] **;**

However, the “floating cursor” mode can be limited through the “MaxLeftChar” property. If a maximum value is set for “MaxLeftChar”, horizontal placement beyond this preset value will not be allowed unless a line exists that exceeds this size. That is, “MaxLeftChar” does not limit the size of the current line, but it can limit the horizontal position of the cursor.

To force the cursor to be positioned without considering the size of the destination line, the MoveCaretIgnoreEOL() method can be used. This method will work even when “MaxLeftChar” is exceeded.

Another, somewhat strange, peculiarity of SynEdit is that it allows you to position the cursor in the middle of a tab stop as if they were simple spaces, giving the impression that there is no tab stop in that position.

To notice the existence of said tab stop, you can try to select a space within the tab stop area. If the selection is not possible, it will indicate that you are inside a tab stop.

If you want to disable this behavior and force tabs to be treated as a single character in the editor, you must activate the “eoCaretSkipTab” option, in the Options2 property:

SynEdit1.Options2 **:=** SynEdit1.Options2 **+** [eoCaretSkipTab] **;**

The position of the cursor can be changed at will, using the properties: CaretX, CaretY or CaretXY, as already mentioned, but you can also position the cursor by commands (See Section 1.5.1 - 1.5.1commands Run commands.

The following code shows how to position the cursor at the end of all SynEdit text:

SynEdit1.ExecuteCommand **(** ecEditorBottom, '' , **nil );**

To move the cursor as if using the arrow keys, the commands “ecLeft”, “ecRight”, “ecUp”, and “ecDown” must be used.

### Line delimiter

As is often the case in many aspects of computing, there is no consensus as to how a line break is defined in text. Currently there are 3 best-known ways to delimit a line:

1. Format TWO: Characters #13#10. Used on DOS/Windows systems.
2. Unix format: Character #10. Used on Linux/Unix systems.
3. MAC format: Character #13. Used in MAC systems.

When reading text from a file, using SynEdit1.Lines.LoadFromFile(), any of these delimiters will be recognized and the lines will be loaded correctly.

However, if you want to save the content with SynEdit1.Lines.SaveToFile(), the original encoding may be lost, depending on the operating system used or the system configuration.

To change the line feed (DOS, UNIX, MAC) used with SaveToFile(), you must use the SynEditLines unit and run the FileWriteLineEndType method of the TSynEditLines class:

**Uses...,** SynEditLines;

...

**If** type **=** 'TWO' **then** TSynEditLines **(** SynEdi1.Lines **)** .FileWriteLineEndType **:=** sfleCrLf **;**

**If** type **=** 'UNIX' **then** TSynEditLines **(** SynEdi1.Lines **)** .FileWriteLineEndType **:=** sfleLf **;**

**If** type **=** 'MAC' **then** TSynEditLines **(** SynEdi1.Lines **)** .FileWriteLineEndType **:=** sfleCr **;**

Another option is to change the “TextLineBreakStyle” property of the “SynEdit1.Lines” list. But this separator will only take effect when using the “Text” property of the list.

## Modify the content

The content of the SynEdit editor will usually be modified by the user, using the keyboard, but many times we will need to take control of the editor from within the program. Here we will describe the various ways to access the editor's content.

To clear all the content of the editor you must use the “ClearAll” method:

SynEdit1.ClearAll **;**

To write in SynEdit, the easiest way is to assign text to the Text property:

SynEdit1.Text **:=** 'Hello world' **;**

We can also assign line breaks:

SynEdit1.Text **:=** 'Hello' **+** #13#10 **+** 'world' **;**

Writing this way will cause all previous information that the control might contain to be lost (without an undo option), because we are assigning a new value. If we only wanted to add information we could do:

SynEdit1.Text **:=** SynEdit1.Text **+** 'Hello world' **;**

The Text property is a simple string and allows us to read or write to the content of the Editor. As it is a chain-type capo, the same operations that are done with strings can be performed with it (search, concatenation, etc.).

One detail to keep in mind is that when you read the content of SynEdit, through its “Text” property, you get an additional line break at the end of the text.

Another way to insert text in the editor is to use the InsertTextAtCaret() method:

SynEdit1.InsertTextAtCaret **(** 'inserted text' **);**

With InsertTextAtCaret(), the text is inserted directly at the position where the cursor is located. If there is a current selection, the selection is removed (the selected text is not removed) and the indicated text is placed at the cursor position.

The inserted text can be one or more lines.

Another useful function for modifying the content of SynEdit is TextBetweenPoints(). The following example shows a quick way to replace selected text with new one:

SynEdit1.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd] **:=** 'New text';

For information on BlockBegin and BlockEnd, go to section 1.6- Selection management.

Another way would be to use the TextBetweenPointsEx() method, which has more options for controlling the cursor, after replacement.

### Run commands

The editor can also be controlled, through the use of commands, with the “ExecuteCommand” method.

Virtually everything that can be done with the keyboard in the editor can also be done using commands. For example, to insert the character “x” at the current cursor position, you would use:

SynEdit1.ExecuteCommand **(** ecChar, 'x' , **nil );**

There is a huge group of commands that can be entered into SynEdit. All of them are declared in the “SynEditKeyCmds” unit. Some of them are shown:

ecLeft **=** 1 **;** **//** Move cursor left one char

ecRight **=** 2 **;** **//** Move cursor right one char

eqUp **=** 3 **;** **//** Move cursor up one line

ecDown **=** 4 **;** **//** Move cursor down one line

ecDeleteLastChar **=** 501 **;** **//** Delete last char **(** ie backspace key **)**

ecDeleteChar **=** 502 **;** **//** Delete char at cursor **(** ie delete key **)**

ecDeleteWord **=** 503 **;** **//** Delete from cursor **to end of** word

ecDeleteLastWord **=** 504 **;** **//** Delete from cursor **to** start **of** word

ecDeleteBOL **=** 505 **;** **//** Delete from cursor **to** beginning **of** line

ecDeleteEOL **=** 506 **;** **//** Delete from cursor **to end of** line

ecDeleteLine **=** 507 **;** **//** Delete current line

ecClearAll **=** 508 **;** **//** Delete everything

ecLineBreak **=** 509 **;** **//** **Break** line at current position, move caret **to** newline

ecInsertLine **=** 510 **;** **//** **Break** line at current position, leave caret

ecChar **=** 511 **;** **//** Insert a character at current position

ecSmartUnindent **=** 512 **;** **//** NOT regocnicated **as** command, used **for** group **-** undo, **set** by beautifier

ecImeStr **=** 550 **;** **//** Insert character **(** s **)** from IME

ecundo **=** 601 **;** **//** Perform undo **if** available

ecRedo **=** 602 **;** **//** Perform redo **if** available

ecCut **=** 603 **;** **//** Cut selection **to** clipboard

ecPaste **=** 604 **;** **//** Paste clipboard **to** current position

ecBlockIndent **=** 610 **;** **//** Indent selection

ecBlockUnindent **=** 611 **;** **//** Unindent selection

ecTab **=** 612 **;** **//** Tab key

ecShiftTab **=** 613 **;** **//** Shift **+** Tab key

...

As you can see, the commands can perform all kinds of actions such as pasting text, deleting a character or undoing changes.

Using the ExecuteCommand() commands, you can make all kinds of modifications in SynEdit, but it is a slow way of modifying, because it is done character by character.

The actions that can be performed also include managing bookmarks and folding blocks of text. The following commands serve this function:

ecGotoMarker0 **=** 301 **;** **//** **goto** marker

ecGotoMarker1 **=** 302 **;** **//** **goto** marker

ecGotoMarker2 **=** 303 **;** **//** **goto** marker

ecGotoMarker3 **=** 304 **;** **//** **goto** marker

ecGotoMarker4 **=** 305 **;** **//** **goto** marker

ecGotoMarker5 **=** 306 **;** **//** **goto** marker

ecGotoMarker6 **=** 307 **;** **//** **goto** marker

ecGotoMarker7 **=** 308 **;** **//** **goto** marker

ecGotoMarker8 **=** 309 **;** **//** **goto** marker

ecGotoMarker9 **=** 310 **;** **//** **goto** marker

ecSetMarker0 **=** 351 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker1 **=** 352 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker2 **=** 353 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker3 **=** 354 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker4 **=** 355 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker5 **=** 356 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker6 **=** 357 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker7 **=** 358 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker8 **=** 359 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecSetMarker9 **=** 360 **;** **//** **Set** marker, Data **=** PPoint **-** X, Y Pos

ecToggleMarker0 **=** 361 **;** **//** **If** marker **is in** the lie, remove marker, read **set** marker, Data **=** PPoint **-** X, Y Pos

ecToggleMarker1 **=** 362 **;**

ecToggleMarker2 **=** 363 **;**

ecToggleMarker3 **=** 364 **;**

ecToggleMarker4 **=** 365 **;**

ecToggleMarker5 **=** 366 **;**

ecToggleMarker6 **=** 367 **;**

ecToggleMarker7 **=** 368 **;**

ecToggleMarker8 **=** 369 **;**

ecToggleMarker9 **=** 370 **;**

EcFoldLevel1 **=** 371 **; //** fold all folds, greater **/** equal than nesting level 1

EcFoldLevel2 **=** EcFoldLevel1 **+** 1 **;**

EcFoldLevel3 **=** EcFoldLevel2 **+** 1 **;**

EcFoldLevel4 **=** EcFoldLevel3 **+** 1 **;**

EcFoldLevel5 **=** EcFoldLevel4 **+** 1 **;**

EcFoldLevel6 **=** EcFoldLevel5 **+** 1 **;**

EcFoldLevel7 **=** EcFoldLevel6 **+** 1 **;**

EcFoldLevel8 **=** EcFoldLevel7 **+** 1 **;**

EcFoldLevel9 **=** EcFoldLevel8 **+** 1 **;**

EcFoldLevel0 **=** EcFoldLevel9 **+** 1 **;**

EcFoldCurrent **=** 381 **;**

EcUnFoldCurrent **=** 382 **;**

EcToggleMarkupWord **=** 383 **;**

### Accessing Lines[]

The content of the editor (the lines of text) is stored in the “Lines” property, which is a list of strings similar to a “TStringList” object, so it can be accessed like any common list of strings. For example, to display the content of the first line, we would do:

showmessage **(** SynEdit1.Lines[0] **);**

Accessing Lines[] is a quick way to access SynEdit content. Accessing this way allows us direct string handling (because it is a list of strings), so we can use all the string functions for our purposes. However, the modifications made cannot be canceled with the editor's “Undo” method.

Therefore, it is not advisable to modify Lines[], if you intend to undo the changes later.

Since Lines[], contains all the text of SynEdit, if we wanted to access the first line of SynEdit, we must access Lines[0]. So to write a text in the first row of SynEdit, we must do:

SynEdit1.Lines[0] **:=** 'Hello world' **;**

This statement will always work, because SynEdit contains at least one line of text, but if we tried to access Lines[1], without a second line existing in the editor, an error will be generated at run time.

To know the number of lines in the editor we can use the Count method:

LineNumbers **:=** SynEdit1.Lines.Count;

Because Lines[] is a list, it shares many of the list methods we know. For example, to add one more line to the editor we can do:

SynEdit1.Lines.Add **(** 'New text line' **);**

Some of the properties of Lines[] are listed in the following table:

|  |  |
| --- | --- |
| **PROPERTY** | **DESCRIPTION** |
| Add | Add a line of text |
| AddStrings | Adds all content from another list. |
| Capacity | Defines the number of lines that are created when the list needs to be expanded. It saves time since several elements are created at once. |
| Clear | Clear the content of the list. |
| Count | Number of Line[] elements (editor lines) |
| Delete | Delete an item from the list. |
| Exchange | Swap positions two lines. |
| Insert | Inserts a line at a specific position. |
| LoadFromFile | Allows you to read the content from a file |
| SaveToFile | Allows you to save the content to a file |
| text | Returns the complete content of all lines in the editor |

To iterate over the entire contents of Lines[], the following construction can be used:

**for** i **:=** 0 **to** SynEdit1.Lines.Count **-** 1 **do**

ShowMessage **(** SynEdit1.Lines[i] **);**

To make modifications, it is preferable to do them using editor commands than by directly accessing “Lines[]”, to keep the “Undo” options active. However, complex command modifications can be much slower than Lines[] modifications.

### The Clipboard

Typical functions with the clipboard are activated with these methods:

SynEdit1.CopyToClipboard **;**

SynEdit1.CutToClipboard **;**

SynEdit1.PasteFromClipboard **;**

It is not necessary to identify what it does, because their names are quite well known, and we already know that they move data between the selected text and the clipboard.

The behavior is the same as if we executed the shortcuts Ctrl+C, Ctrl+V, and Ctrl+X. In fact this same key combination is active by default, when using a SynEdit.

It should be noted that these options will still work in column selection mode, so that rectangular blocks of text can be copied and pasted.

Clipboard options can also be accessed by using commands:

SynEdit1 . CommandProcessor **(** ecCopy , ' ' , **nil ) ;**

SynEdit1 . CommandProcessor **(** ecPaste , ' ' , **nil ) ;**

SynEdit1 . CommandProcessor **(** ecCut , ' ' , **nil ) ;**

Since the clipboard works with the selection, it is usual to work with the “BlockBegin” and “BlockEnd” properties.

It is also possible to place text directly on the clipboard, without having to make a “Copy”:

SynEdit1.DoCopyToClipboard **(** 'text to be clipped' ,'' **);** **//** put on clipboard

### Do and Undo.

SynEdit has very good control of changes made to the text. It allows you to undo and redo changes in almost all cases. These changes can even be modifications in column mode.

Every time a change is made, SynEdit saves the change made to internal memory

To control changes, the methods are used:

|  |  |
| --- | --- |
| **METHOD/PROPERTY** | **DESCRIPTION** |
| SynEdit1.Undo() | Undoes a change made on the screen |
| SynEdit1.Redo() | Redo a change that has been undone. |
| SynEdit1.ClearUndo() | Clears the list of changes (Undo) and will not allow undoing from that point. |
| SynEdit1.MaxUndo | Maximum number of actions (Undo) that will be recorded and can be undone. |
| SynEdit1.BeginUndoBlock() SynEdit1.EndUndoBlock() | They generate unique blocks of changes. |
| SynEdit1.CanUndo | Indicates if there are actions to undo |
| SynEdit1.CanRedo | Indicates if there are actions to be redone |

Almost all changes made manually from the keyboard in a SynEdit can be “undone”, however, when changing content from code, keep in mind that some actions cannot be undone.

The following table shows the change methods made in a SynEdit and whether they support “undo”.

|  |  |
| --- | --- |
| **ACTION** | **ALLOWS UNDO** |
| Methods:  SynEdit1.ClearAll;  SynEdit1.ExecuteCommand()  SynEdit1.InsertTextAtCaret()  SynEdit1.TextBetweenPoints()  SynEdit1.SearchReplace()  SynEdit1.SearchReplaceEx() | YEAH |
| Type changes:  SynEdit1.Text:='Hello';  SynEdit1.LineText:='Hello'; | NO |
| Direct modifications to SynEdit1.Lines[] | NO |
| Modifications using the clipboard:  SynEdit1.CopyToClipboard;  SynEdit1.CutToClipboard;  SynEdit1.PasteFromClipboard; | YEAH |

We must consider the methods we must use to make changes in a SynEdit, if we want to maintain the “undo” options.

Usually the changes they generate, each instruction that modifies the content of SynEdit and that supports “undo”, can be discarded with a simple call to “Undo”.

If you want to group several actions to be undone with a single “Undo”, you must use the BeginUndoBlock and EndUndoBlock methods:

SynEdit1.BeginUndoBlock **;**

**//** There may be several changes here that support undo.

...

SynEdit1.EndUndoBlock **;**

With this construction we will be able to undo all the changes made with a single call to “Undo”.

It may be practical to use the BeginUpdate and EndUpdate constructs to prevent the control from being refreshed until all changes to SynEdit have been made:

ActiveEditor . SynEditor . BeginUpdate **;** **//** SynEdit refresh is disabled

**try**

SynEdit1.BeginUndoBlock **;**

**//** There may be several changes here that support undo.

...

SynEdit1.EndUndoBlock **;**

**finally**

ActiveEditor . SynEditor . EndUpdate **;** **//** SynEdit refresh is reactivated

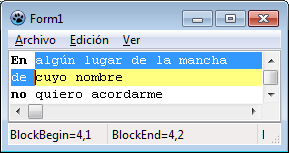
**end ;**

Working in this way allows us to improve the speed of changes, because the control does not have to be refreshed every time something is modified.

When SynEdit refresh is disabled, even requests of type Application.ProcessMessages() will be ignored.

## Selection management

Like most editors today, SynEdit handles only one selection block [[5]](#footnote-5). This block is the one used for cutting and copying text operations, but it is also used to modify text by overwriting:



The selection is defined, by code, using the “BlockBegin” and “BlockEnd” properties which are of type “Tpoint”. In the previous example, the values assigned to “BlockBegin” and “BlockEnd” are:

SynEdit1.BlockBegin.x **:=** 4 **;**

SynEdit1.BlockBegin.y **:=** 1 **;**

SynEdit1.BlockEnd.x **:=** 4 **;**

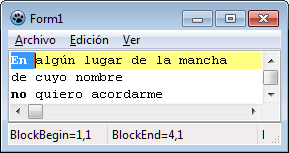
SynEdit1.BlockEnd.y **:=** 2 **;**

When there is no active selection, “BlockBegin” and “BlockEnd” indicate the same position.

Another way to define the selection block is to use the “SelStart” and “SelEnd” properties. These properties allow you to work in an equivalent way to “BlockBegin” and “BlockEnd”, but they are not of type “TPoint”, but are simple integers that map the text as if they were an uninterrupted series of characters.

Line breaks are considered two characters in Windows and can also be part of the selection.

The following example shows a selection defined with SelStart = 1 and SelEnd = 4:



The selection block is defined from the position of the character pointed to by “SelStart”, to the character before the character pointed to by “SelEnd”. Therefore, the number of selected characters will be equal to (SelEnd-SelStart).

When you want to select a region relative to the current cursor position, you can use these methods:

|  |  |
| --- | --- |
| SynEdit1.SelectWord; | Selects the current word where the cursor is located. |
| SynEdit1.SelectLine; | Select the current line where the cursor is located. |
| SynEdit1.SelectParagraph; | Select the current paragraph where the cursor is located. |
| SynEdit1. SelectToBrace; | Selects the block delimited by parentheses, braces, or square brackets. |

All of these selection methods are based on the current position of the cursor and work as if the selection were made manually, considering that SelectWord identifies a word using only the alphabetic characters, including the checked characters and the letter ñ.

“SelectLine“, has the following declaration:

**procedure** SelectLine **(** WithLeadSpaces **:** Boolean **=** **True );**

The optional parameter allows you to indicate whether you want to include the leading and trailing whitespace as part of the selection. If set to FALSE, the selection of the current line may not be complete if there are leading or trailing spaces in the line.

The SelectToBrace method allows you to select blocks of text that are delimited by parentheses, braces or brackets. It will only work when the following conditions are met:

* Let the current character where the cursor is located be; '(', '{' or '['or the character before the cursor is ')', '}', or ']'.
* That the corresponding delimiter exists on the same line or on any other line.

Additionally, it should be considered that selection with SelectToBrace allows the nesting of blocks of the same type.

To determine if there is an active selection (selected text), the “SelAvail” property must be used:

**if** editor.SelAvail **then** ...

Another way would be to compare the coordinates of “BlockBegin” and “BlockEnd”.

The selected text can be obtained using the “SelText” property. Only one selection block is supported. Usually the cursor is located at one of the boundaries of the selection block, but in “persistent block” mode, the cursor can be made independent of the position of the selection block.

The “SelText” property is also writable, so it allows us to modify the selected text. The following code removes the selected text:

SynEdit1.SelText **:=** '' **;**

However, to clear the selection there is the “ClearSelection” method, which is a shortened form.

To select all the text, we must use:

SynEdit1.SelectAll **;**

To remove the selected text you can use the “ClearSelection” method:

SynEdit1.ClearSelection **;**

You can also use sending the “ ecDeleteLastChar ” command with ExecuteCommand():

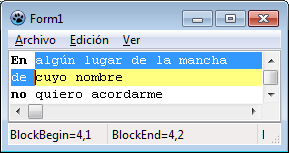
SynEdit1.ExecuteCommand **(** ecDeleteLastChar, '' , **nil );**

Another alternative way would be to replace the selected text with TextBetweenPoints():

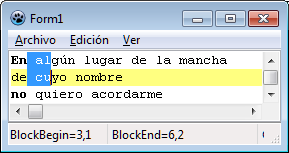
SynEdit1.TextBetweenPoints[editor.BlockBegin,editor.BlockEnd] **:=** '' **;**

### Selection in Column Mode

Selecting text in SynEdit can be done in various ways. Normally all rows between the start and end of the selection block are selected:



But you can also use the column selection mode:



In this mode the selection area forms a rectangle and only partially includes the lines in its path.

To select in column mode in SynEdit, you must use the <Alt>+<Shift>+directional key combination.

Once you have the text selected, you can execute the cut, copy or paste actions. You can also override the selection by pressing any key.

It should be noted that any key pressed other than the combination, <Alt>+<Shift>+directional, will cause the column mode selection mode to end.

To switch to column mode per program, the following code can be used:

**uses** ... , SynEditTypes **;**

**var** pos **:** Tpoint **;**

...

pos.x **:=** 3 **;**

pos.y **:=** 2 **;**

SynEdit1.BlockBegin **:=** pos **;** **//** define initial selection point

pos.x **:=** 8 **;**

pos.y **:=** 3 **;**

SynEdit1.BlockEnd **:=** pos **;** **//** define selection endpoint

SynEdit1.CaretXY **:=** pos **;**

SynEdit1.SelectionMode **:=** smColumn **;** **//** switch to column mode

...

Likewise, in this case, any key pressed other than the combination, <Alt>+<Shift>+directional, will cause the column mode selection mode to end.

There are other forms of selection, defined in the “SynEditTypes” unit:

* smNormal,
* smColumn
* smLine,

The smNormal mode is the mode that is active by default and is the normal selection mode.

The smColumn mode is the column selection mode.

The smLine mode is a selection mode that will cause all lines between BlockBegin and BlockEnd to be marked as part of the selection.

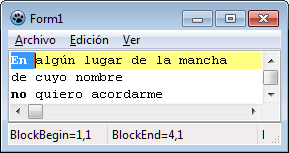
You can also select by column using keyboard commands:

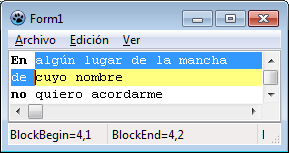
SynEdit1.ExecuteCommand **(** ecColSelUp, #0, **nil )**

The constants ecColSelUp, ecColSelDown, ecColSelLeft, ecColSelRight, and others, allow you to move the cursor while maintaining column mode, which is equivalent to holding down the <Alt>+<Shift> keys.

### BlockBegin and BlockEnd

They determine the coordinates of the selection block.





Selection blocks can be defined using BlockBegin and BlockEnd. For example, if you have an “ed” editor of type TSynEdit, you can select it with this code:

p **:** Tpoint **;**

...

ed.Text **:=** 'Don Quixote' **;**

**//** set block start

p **:=** ed.BlockBegin **;**

px **:=** 1 **;**

ed.BlockBegin **:=** p **;**

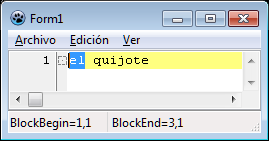
**//** set end of block

p **:=** ed.BlockEnd **;**

px **:=** 3 **;**

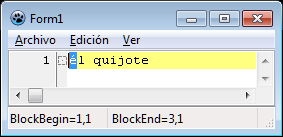
ed.BlockEnd **:=** p **;**

The code will generate the following output on the screen:



However, if SynEdit worked in UTF-8 (as it usually does), a special character can be two characters wide. Therefore, care must be taken with the logical coordinates (those handled by BlockBegin and BlockEnd) and the physical coordinates.

If the string in the editor had been “him quixote”, the selection would have another result:



To overcome this behavior, you must use the PhysicalToLogicalCol() function:

ed.Text **:=** 'him quixote' **;**

**//** set block start

p **:=** ed.BlockBegin **;**

px **:=** 1 **;**

ed.BlockBegin **:=** p **;**

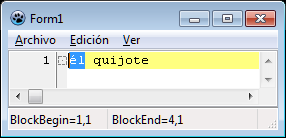
**//** set end of block

p **:=** ed.BlockEnd **;**

px **:=** ed.PhysicalToLogicalCol **(** ed.Lines[0],0, 3 **);**

ed.BlockEnd **:=** p **;**

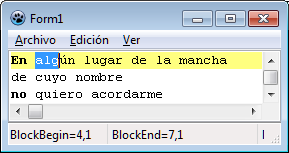
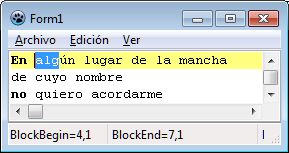
Now the behavior is as expected:



### BlockBegin and BlockEnd in Normal Selection

If the selection only has one row, the BlockBegin point will always point in the column to the left of the selection, no matter which direction the selection was made from.

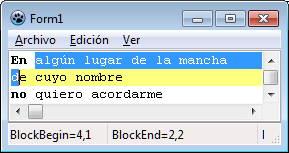
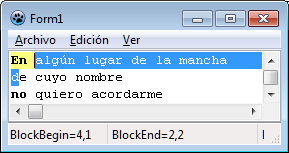
In the following examples, a selection has been made from left to right and from right to left respectively.

How it looks, there is no difference in the sense of selection.

If the selection has multiple rows, the BlockBegin point will always appear in the top row, no matter which direction the selection was made from.

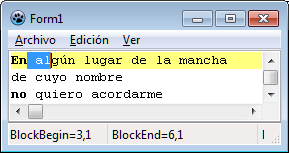
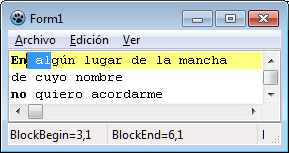
In the following examples, a selection has been made from left to right and from right to left respectively.

### BlockBegin and BlockEnd in Selection in Column Mode

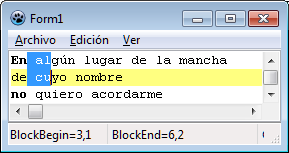
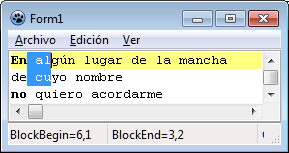
If the selection only has one row, the BlockBegin point will always point in the column to the left of the selection, no matter which direction the selection was made from.

In the following examples, a selection has been made from left to right and from right to left respectively.

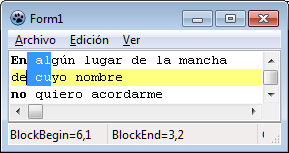
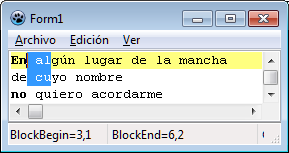
If the selection has multiple rows, the BlockBegin point will always appear in the top row, no matter which direction the selection was made from.

In the following examples, a selection has been made from left to right, down and up:

Note that in the last case, BlockBegin takes the location of the cursor.

In the following examples, a selection has been made from right to left, down and up:

The positions of BlockBegin and BlockEnd are the reverse of the previous case.

In all the cases seen, the cursor will always take the value of BlockBegin or BlockEnd.

## Search and Replacement

Almost every text editor has search and replace options included. So it's useful to know how to implement find and replace options in SynEdit.

After knowing some of its properties and methods, we could implement the search and replace functions ourselves. For the search we could explore, line by line, the content of SynEdit (accessing the Lines[] list) and select the searched text, using the BlockBegin and BlockEnd properties.

However, SynEdit already has two functions included for search and replacement:

**function** SearchReplace **( const** ASearch, AReplace **:** **string ;**   
AOptions **:** TSynSearchOptions **):** integer **;**

**function** SearchReplaceEx **( const** ASearch, AReplace **:** **string ;**   
AOptions **:** TSynSearchOptions **;** AStart **:** TPoint **):** integer **;**

SearchReplaceEx() is similar to SearchReplace(), with the difference that SearchReplaceEx() searches from the indicated position (AStart).

These functions provide us with search functionality up or down, with case sensitivity (upper/lower case), whole word, or the use of regular expressions.

The value returned by these functions is an integer:

|  |  |  |
| --- | --- | --- |
| MODE | WORTH | MEANING |
| Search | 0 | Item not found |
| Search | 1 | At least one item was found. |
| Replacement | 0 | Item not found |
| Replacement | n | “n” elements were replaced |

In Search mode, these functions stop when they find the first match, select it, and make the selected text visible in the editor.

The AOptions parameter is an array that can include the following elements:

TSynSearchOption **=**

**(** ssoMatchCase,

ssoWholeWord,

ssoBackwards,

ssoEntireScope,

ssoSelectedOnly,

ssoReplace,

ssoReplaceAll,

ssoPrompt,

ssoSearchInReplacement, **//continue** search **-** replace **in** replacement

// **(with** ssoReplaceAll **)** replace recursive

ssoRegExpr,

ssoRegExprMultiLine,

ssoFindContinue **//** Assume the current selection **is** the last match,

// **and** start search behind selection **(** before

// **if** ssoBackward **)**

**//** **Default is to** start at caret **(** Only SearchReplace **/**

**//** SearchReplaceEx has start **/end** param **)**

**);**

All of these constants are defined in the “SynEditTypes” unit.

When the ssoReplace or ssoReplaceAll elements are included, a replacement will be done, otherwise only a search will be done.

### Search

To search for a simple string, the following code can be used:

**var**

encon **:** integer **;**

searched **:** **string ;**

**begin**

searched **:=** 'text to search' **;**

encon **:=** editor.SearchReplace **(** searched, '' ,[] **);**

**if** encon **=** 0 **then**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

...

When no search options are specified, the following options are assumed;

* The search in SynEdit will always start from the cursor position.
* The search direction is forward from the cursor to the end of the file.
* There is no case sensitivity, upper and lower case letters are identified equally.

Calling SearchReplace() produces the following flow:

1. Start the search with the requested text and the indicated options.
2. If the search was unsuccessful, zero is returned and the function is exited.
3. If the search was successful, the first match is selected, and the selected text is made visible in the editor.

Selecting the found text causes the cursor to move to the end of the selected text. So the next call to SearchReplace() will search from this position (find next).

To perform a backward search:

encon **:=** editor.SearchReplace **(** searched, '' ,[ssoBackwards] **);**

To perform a search based on the box:

encon **:=** ed.SearchReplace **(** searched, '' ,[ssoMatchCase] **);**

Search options can be combined.

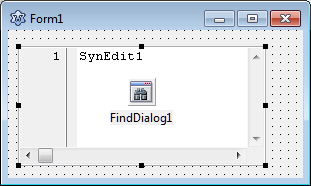
When the “ssoEntireScope” option is indicated, the search is always done from the beginning of the text (regardless of where the cursor is) and stops when the first element is found.

When the “ ssoSelectedOnly ” option is indicated, the search will be done within the selected text, stopping when the first element is found. If you try to perform a next search, the same result will always be given because the selected text, after calling SearchReplace(), will always contain the searched text.

### Search using TFindDialog

There exists among the Lazarus controls, a dialog created specifically for search operations. This component is FIndDialogo, and it is found in the components palette, in the “Dialogs” tab.

To use it, we must place this component in our form:



And then we must create a procedure to attend to the “OnFind” event.

The event procedure may take the following form:

**procedure** TForm1.FindDialog1Find **(** Sender **:** TObject **);**

**var**

encon **:** integer **;**

searched **:** **string ;**

options **:** TSynSearchOptions **;**

**begin**

searched **:=** FindDialog1.FindText **;**

options **:=** [] **;**

**if not (** frDown **in** FindDialog1.Options **)** **then** options **+=** [ssoBackwards] **;**

**if** frMatchCase **in** FindDialog1.Options **then** options **+=** [ssoMatchCase] **;**

**if** frWholeWord **in** FindDialog1.Options **then** options **+=** [ssoWholeWord] **;**

**if** frEntireScope **in** FindDialog1.Options **then** options **+=** [ssoEntireScope] **;**

encon **:=** editor.SearchReplace **(** searched, '' , options **);**

**if** encon **=** 0 **then**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

**end ;**

The “FindDialog1” control exposes the selected options through its “Options” property.

The idea is to pass the options selected in the dialog to the “options” variable before calling SearchReplace().

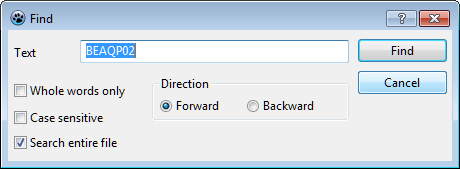
Now from some strategic part of our program (such as the response to the menu), we must include the code to open the dialog and start the search:

**procedure** TForm1.acSearchExecute **(** Sender **:** TObject **);** **//** Search

**begin**

FindDialog1.Execute **;** **//** open the search dialog

**end ;**



By clicking on “Find”, the OnFind event will be called, which must be associated with the “ FindDialog1Find ” method that we have created. The dialog, however, will remain visible until you click on “Cancel” or press the <escape> key.

The “ FindDialog1Find ” method also has another use. It is used to implement the “Find Next” functionality, because this method can be called, even when the “ FindDialog1 ” dialog is hidden.

If this window does not adapt to the required search needs, you can always create a special form for our personalized search.

### Replacement

The replacement process is similar to the search process. To search for a simple string, the following code can be used:

**var**

encon **:** integer **;**

searched **:** **string ;**

**begin**

searched **:=** FindDialog1.FindText **;**

encon **:=** editor.SearchReplace **(** searched, 'new string' , [ssoReplace] **);**

**if** encon **=** 0 **then**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

...

The difference is that the “ssoReplace” option must be indicated to the SearchReplace() method, in addition to indicating the replacement text to use.

The way SearchReplace() works, in replace mode is similar to search mode:

1. Start the search with the requested text and the indicated options.
2. If the search was unsuccessful, zero is returned and the function is exited.
3. If the search was successful, the first match is replaced, and the replaced text is made visible in the editor.
4. The cursor is left at the end of the replaced text, ready for another search sequence.

This mode of operation is useful in search mode, but in replace mode it can be strange, since no confirmation is asked to replace the text and the replacement occurs instantly, without any prior selection.

To improve this behavior, a confirmation window can be added before the replacement, so that it can also be used to see the text that is going to be replaced.

There is no predefined dialog to create a confirmation window. If we want to use one we will have to create it ourselves.

A simple dialog that could help us would be a MessageBox() with the Yes-No-Cancel buttons. So our replacement procedure could be written as follows:

**var**

encon, r **:** integer **;**

searched **:** **string ;**

options **:** TSynSearchOptions **;**

ed **:** TSynEdit **;**

**begin**

**...**

searched **:=** 'searched string' **;**

options **:=** [] **;** **//** search options

encon **:=** ed.SearchReplace **(** searched, '' , options **);** **//** search

**while** encon **<>** 0 **do begin**

**//** question

r **:=** Application.MessageBox **(** 'Replace this occurrence?' , 'Replacement', MB\_YESNOCANCEL **);**

**if** r **=** IDCANCEL **then exit ;**

**if** r **=** IDYES **then begin**

ed.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd] **:=** 'new string' **;**

**end ;**

**//** search next

encon **:=** ed.SearchReplace **(** searched, '' , options **);** **//** search next

**end ;**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

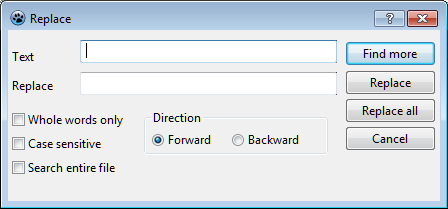
**end** .

The idea here is to ask before each replacement, giving you the option to skip an occurrence or cancel the entire process.

In this mode of work, we are not using the “ ssoReplace ” option, but rather we are using SearchReplace() , only in search mode. We do the replacement in the editor using the “ TextBetweenPoints() ” method.

### Replace using TReplaceDialog

Just as there is the TFindDialog dialog for searching, it is also possible to use the “TReplaceDialog” dialog from the components palette.



This dialog makes it easier for us to enter data to start a Search/Replace, but it does not provide us with a confirmation dialog before replacing.

The way to work with this dialog is simple. The “OnFind” and “OnReplace” events must be assigned. The first event will be executed when the “Find more” button is pressed and the second will be executed with the “Replace” button or the “Replace all” button.

The following code is an example of how to implement the replacement option using the “TReplaceDialog” dialog:

**procedure** TForm1.ReplaceDialog1Replace **(** Sender **:** TObject **);**

**var**

encon, r **:** integer **;**

searched **:** **string ;**

options **:** TSynSearchOptions **;**

**begin**

searched **:=** ReplaceDialog1.FindText **;**

options **:=** [ssoFindContinue] **;**

**if not (** frDown **in** ReplaceDialog1.Options **)** **then** options **+=** [ssoBackwards] **;**

**if** frMatchCase **in** ReplaceDialog1.Options **then** options **+=** [ssoMatchCase] **;**

**if** frWholeWord **in** ReplaceDialog1.Options **then** options **+=** [ssoWholeWord] **;**

**if** frEntireScope **in** ReplaceDialog1.Options **then** options **+=** [ssoEntireScope] **;**

**if** frReplaceAll **in** ReplaceDialog1.Options **then begin**

**//** been asked to replace everything

encon **:=** ed.SearchReplace **(** searched,ReplaceDialog1.ReplaceText,

options **+** [ssoReplaceAll] **);** **//** replace

ShowMessage **(** 'Replaced' **+** IntToStr **(** encon **)** **+** 'occurrences.' **);**

**exit ;**

**end ;**

**//** replace with confirmation

ReplaceDialog1.CloseDialog **;**

encon **:=** ed.SearchReplace **(** searched, '' , options **);** **//** search

**while** encon **<>** 0 **do begin**

**//** question

r **:=** Application.MessageBox **(** 'Replace this occurrence?' , 'Replacement', MB\_YESNOCANCEL **);**

**if** r **=** IDCANCEL **then exit ;**

**if** r **=** IDYES **then begin**

ed.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd] **:=** ReplaceDialog1.ReplaceText **;**

**end ;**

**//** search next

encon **:=** ed.SearchReplace **(** searched, '' , options **);** **//** search next

**end ;**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

**end ;**

This code corresponds to the one that must be associated with the “OnReplace” event. The code for the “OnFind” event can be a simple search:

**procedure** TForm1.ReplaceDialog1Find **(** Sender **:** TObject **);**

**var**

encon **:** integer **;**

searched **:** **string ;**

options **:** TSynSearchOptions **;**

**begin**

searched **:=** ReplaceDialog1.FindText **;**

options **:=** [] **;**

**if not (** frDown **in** ReplaceDialog1.Options **)** **then** options **+=** [ssoBackwards] **;**

**if** frMatchCase **in** ReplaceDialog1.Options **then** options **+=** [ssoMatchCase] **;**

**if** frWholeWord **in** ReplaceDialog1.Options **then** options **+=** [ssoWholeWord] **;**

**if** frEntireScope **in** ReplaceDialog1.Options **then** options **+=** [ssoEntireScope] **;**

encon **:=** ed.SearchReplace **(** searched, '' , options **);**

**if** encon **=** 0 **then**

ShowMessage **(** 'Not found **:** ' **+** searched **);**

**end ;**

In both examples, it is assumed that the editor “ed” and the dialog “ReplaceDialog1” have been included in the form.

This dialog has various options that can be customized from the “Options” property. These options allow you to hide or show certain buttons.

The “frPromptOnReplace” option allows you to pass control to a confirmation form, just as we did with MessageBox(), but this confirmation is requested through the “OnReplaceText” event of the text editor (not the dialog), which must have the form :

TReplaceTextEvent **=** **procedure (** Sender **:** TObject **;**

**const** ASearch, AReplace **:** **string ;**

Line, Column **:** integer **;**

**var** ReplaceAction **:** TSynReplaceAction **)** **of object ;**

## Redialing options

SynEdit is a fairly complete component. Among its various options, it includes content highlighting. These options are included in the editor code itself and are independent of the use of syntax highlighters, described in section 2.

### Highlighting a text.

To highlight any text, you can use the SetHighlightSearch() method, as follows:

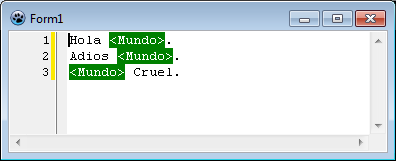
**uses** ..., SynEditTypes **;**

...

SynEdit1.HighlightAllColor.Background **:=** clGreen **;**

SynEdit1.SetHighlightSearch **(** '<World>' , [ssoSelectedOnly] **)** **;**

If this code is applied to a text containing the character sequence <World>, a result like the one shown will be obtained:



It should be noted that the text to be highlighted can be any combination of characters, not necessarily letters.

To specify that only whole words are marked, the “ssoWholeWord” option can be used:

SynEdit1.HighlightAllColor.Background **:=** clGreen **;**

SynEdit1.SetHighlightSearch **(** 'World' , [ssoSelectedOnly,ssoWholeWord] **);**

Search options are of type TSynSearchOption. These are:

TSynSearchOption **=**

**(** ssoMatchCase, ssoWholeWord,

ssoBackwards,

ssoEntireScope, ssoSelectedOnly,

ssoReplace, ssoReplaceAll,

ssoPrompt,

ssoSearchInReplacement,

ssoRegExpr, ssoRegExprMultiLine,

ssoFindContinue

**);**

As you can see, they are the same options used for Search/Replace. Only some of these options work with SetHighlightSearch().

The “ssoRegExpr” and “ssoRegExprMultiLine” options allow you to perform searches using regular expressions, such as “ab”. But you have to be careful with the expressions to use, because an expression of type “a\*” will match any character (even an empty character), causing the search to enter an infinite loop.

This type of highlighting uses the same options as when performing a text search (See Section 1.7- Search and Replacement), so it has the same considerations.

The text can be modified, but each time the searched sequence is found, the text will be marked again. It's like having a permanent text search engine implemented.

This text highlighting method works like a simple syntax highlighter, but it is limited because it only applies to a single text and because it only recognizes identifiers and symbols.

### Highlighting the current word

There is another form of text highlighting that is applicable only to words and at the current cursor position. A word can only contain alphanumeric characters, the dollar sign, and the underscore. Also included are stressed vowels and the letter ñ.

To make use of this feature, we need to include the “SynEditMarkupHighAll” unit, and create an object of the “TSynEditMarkupHighlightAllCaret” class.

**uses** ... , SynEditMarkupHighAll **;**

...

**var**

SynMarkup **:** TSynEditMarkupHighlightAllCaret **;**

**begin**

**//** Start highlighting of same words

SynMarkup **:=** TSynEditMarkupHighlightAllCaret **(**

SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret] **);**

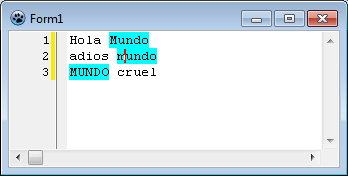
SynMarkup.MarkupInfo.Background **:=** clAqua **;**

SynMarkup.WaitTime **:=** 250 **;** **//** time in milliseconds

SynMarkup.Trim **:=** **True ;**

SynMarkup.FullWord **:=** **True ;** **//** will only match the entire word

This code will highlight words equal to the word the cursor is on, as shown in the following figure:

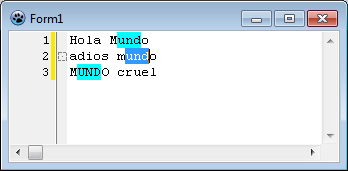


The editor automatically recognizes the word under the cursor and takes it to mark up the entire text. By default, the type of box (upper or lower case) is ignored.

The highlighting can include various attributes such as background color, text color, or border. These attributes are accessed through the “MarkupInfo” property.

The value set to “WaitTime” will indicate how many milliseconds it will wait to recognize the word where the cursor is located. IF not specified, it will wait 2 seconds. It is not recommended to use a very low value for “WaitTime”, because it could be forcing the editor to perform very frequent searches.

This highlighting option also works with selected text. So only the text that is within the selection will be searched:



It should be noted that the highlighting does not read the entire content of the entire editor. By default it will only read up to 100 lines in front and back of the current screen.

To disable highlighting of the current word, you can disable the marker assigned to SynEdit:

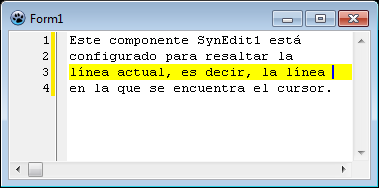
SynMarkup.Enabled **:=** **False ;** **//** disable highlighting

### Highlighting the current line

Many times it is convenient to mark the current line to easily identify where the cursor is. SynEdit allows you to easily change the background color of the line where the cursor is located:

SynEdit1.LineHighlightColor.Background **:=** clYellow **;**

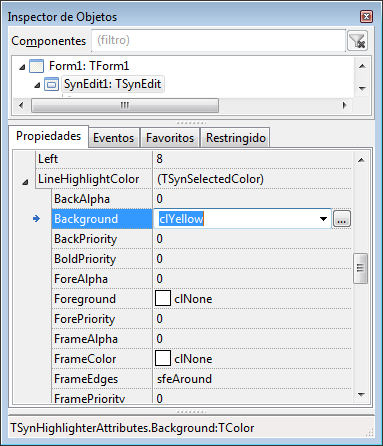
The previous instruction will paint the background of the current line yellow. The result will be similar to that shown in the following figure:



To disable highlighting of the current line, you can use the same editor background color, or you can use the clNone color:

SynEdit1.LineHighlightColor.Background **:=** clNone **;**

The highlighting of the current line can also be activated, using the object inspector, by setting the LineHighlightColor property:



Here you can notice that there are several additional properties to “Background” to change the appearance of the highlighted line. All of them can be configured by code, or from the object inspector.

### Highlighting any line

You can tell SynEdit to highlight one or more lines of content. To do this, we must first create a method in our form that identifies the line to be marked and assigns the attributes:

**procedure** TForm1.SynEdit1SpecialLineMarkup **(** Sender **:** TObject **;**

Line **:** integer **;**

**varSpecial :** boolean **;** \_

Markup **:** TSynSelectedColor **);**

**begin**

**if** Line **=** 2 **then begin**

Special **:=** **True ; //** mark as special line

Markup.Background **:=** clGreen **;** **//** background color

**end ;**

**end ;**

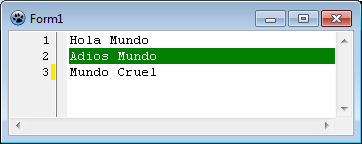
The type “TSynSelectedColor” is defined in the “SynEditMiscClasses” unit, so it will be necessary to include it first to work.

Then we must assign this method to the “OnSpecialLineMarkup” event of the editor:

SynEdit1.OnSpecialLineMarkup **:=** @SynEdit1SpecialLineMarkup **;**

Every time the editor scans a line, it calls the “OnSpecialLineMarkup” event, to see if it has any special attributes or is a common line. The implementation of this event must be quick to respond to avoid interfering with the operation of SynEdit.

The result of this code would be something like this:



In this case we have activated the highlighting of the second line, and it will remain marked until we cancel the event. The marked line will always be the second, even when lines are inserted or deleted. To mark a line that follows the contained text, additional processing must be added.

If the highlighted line is changed by code, the editor must be refreshed to update the new highlighted line. This can be done by calling the “SynEdit1.Refresh” method but it is recommended to use the “SynEdit1.Invalidate” method, which is more efficient.

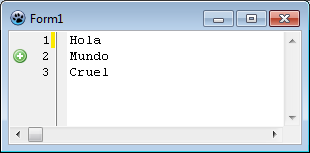
There is also the OnSpecialLineColors() event that allows you to perform a similar task, but it only allows you to change the background color and the text color. This event is defined as:

TSpecialLineColorsEvent **=** **procedure (** Sender **:** TObject **;** Line **:** integer **;**  
    **varSpecial :** boolean **;** \_ **var** FG, BG **:** TColor **)** **of object ;**

### Text markers

Text markers are special locations set in SynEdit content. You could say that they are the digital equivalent of the tape (separator) used to mark a page in a book.

However, SynEdit bookmarks save positions that include row and column, and there can be as many as you want. Text markers can be displayed graphically as an icon in the Vertical Panel:



Text markers, unlike those seen previously, do not highlight the editor's content, but rather store a location and have the option of displaying an icon in the editor's vertical panel.

These bookmarks are useful for saving the positions of certain lines of text. If lines are inserted or deleted, the marker will try to maintain its location, following its line.

To create a marker, you must include the SynEditMarks unit and have a list of icons, stored in a TImageList. The following code, used for the figure above, creates a marker and makes it visible on line 2:

**uses** ... , SynEditMarks **;**

**var**

mark **:** TSynEditMark **;**

...

mark **:=** TSynEditMark.Create **(** SynEdit1 **);** **//** create bookmark

mark.ImageList **:=** ImageList1 **;** **//** assign list of icons

mark.ImageIndex **:=** 1 **;** **//** choose your icon

mark.Line **:=** 1 **;** **//** defines the line where it will be located

flag.Visible **:= true ;** **//** makes it visible

SynEdit1.Marks.Add **(** mark **);** **//** add the marker to the editor

In this way, various markers can be added in different positions on the screen, with the same or different icon.

It is important that the line where the marker is located is valid, or else it will not appear at all.

The displayed markers are simply handled as special positions in the text, in no particular order.

There are, however, another type of bookmarks called “BookMark”, which also offer a number that can identify them.

“BookMarks” are usually created differently. A list of images is also required:

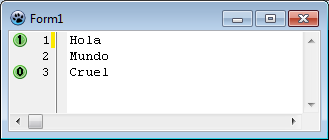
SynEdit1.BookMarkOptions.BookmarkImages **:=** ImageList1 **;**

SynEdit1.SetBookMark **(** 0, 1, 1 **);** **//** BookMark 0 on line 1

SynEdit1.SetBookMark **(** 1, 1, 3 **);** **//** BookMark 1 on line 3

The first parameter of SetBookMark() is the number of “BookMark” created. That number will identify you from then on. The other two parameters are the row and column they mark. Although the icon appears to select a row, BookMarks actually store the XY coordinates.

Although the icons used for the “BookMark” can be of any type, it is usual to use icons that are associated with the bookmark number:



To delete a “BookMark”, you must know its number:

SynEdit1.ClearBookMark **(** 5 **);**

This code removes the “BookMark” number 5. If the “BookMark” number does not exist, the command is ignored.

To access any bookmark (it may or may not be “BookMark”), it is done as if it were an element of an array:

mark **:=** SynEdit1.Marks[0] **;**

**if** brand.IsBookmark **then** ...

The “IsBookMark” property allows you to identify if a bookmark is a “BookMark”.

You can also create a “BookMark” like any bookmark:

**uses** ... , SynEditMarks **;**

**var**

mark **:** TSynEditMark **;**

...

SynEdit1.BookMarkOptions.BookmarkImages **:=** ImageList1 **;**

mark **:=** TSynEditMark.Create **(** SynEdit1 **)** **;**

mark.Line **:=** 1 **;**

mark.Column **:=** 5 **;**

mark.ImageIndex **:=** 0 **;**

bookmark.BookmarkNumber **:=** 1 **;**

mark.Visible **:=** **true ;**

mark.InternalImage **:=** **false ;**

SynEdit1.Marks.Add **(** mark **);**

The “InternalImage” property is not functional in the current version of Lazarus. It is reminiscent of a facility that allowed using internal icons for text markers.

Another way to create a “BookMark” is by adding it as a command:

**uses** ... , SynEditKeyCmds **;**

SynEdit1.BookMarkOptions.BookmarkImages **:=** ImageList1 **;**

SynEdit1.ExecuteCommand **(** ecSetMarker4,#0, **nil );**

There are 10 constants defined for the “BookMark”: ecSetMarker0 .. ecSetMarker9.

There are also 10 predefined constants to change the state of the “BookMark”: ecToggleMarker0 .. ecToggleMarker9 .

By default, shortcuts for adding bookmarks (BookMark) are created in SynEdit when you add the control. These shortcuts are of the type <Shift>+<Ctrl>+<marker number>.

An attempt to add a marker, without having an icon list assigned, will result in a runtime error.

To disable bookmarks, you can hide the part corresponding to bookmarks in the vertical panel (Gutter), or you can also eliminate keyboard shortcuts (“Keystrokes” Property).

### Access to bookmarks

The most basic markers in SynEdit are objects of the “TsynEditMarkup” class. All markers are derived directly or indirectly from “TsynEditMarkup”.

The “TsynEditMarkup” class is a more or less extensive class, and is defined in the “SynEditMarkup” unit. Its public fields are:

TSynEditMarkup **=** **class (** TObject **)**

**private**

...

**protected**

...

**public**

**constructor** Create **(** ASynEdit **:** TSynEditBase **);**

**destroyer** Destroy **;** **override ;**

**Procedure** PrepareMarkupForRow **(** aRow **:** Integer **);** **virtual ;**

**Procedure** FinishMarkupForRow **(** aRow **:** Integer **);** **virtual ;**

**Procedure** EndMarkup **;** **virtual ;**

**Function** GetMarkupAttributeAtRowCol **( const** aRow **:** Integer **;**

**const** aStartCol **:** TLazSynDisplayTokenBound **;**

**const** AnRtlInfo **:** TLazSynDisplayRtlInfo **)** **:** TSynSelectedColor **;** **virtual ;** **abstract ;**

**Procedure** GetNextMarkupColAfterRowCol **( const** aRow **:** Integer **;**

**const** aStartCol **:** TLazSynDisplayTokenBound **;**

**const** AnRtlInfo **:** TLazSynDisplayRtlInfo **;**

out ANextPhys, ANextLog **:** Integer **);** **virtual ;** **abstract ;**

**procedure** MergeMarkupAttributeAtRowCol **( const** aRow **:** Integer **;**

**const** aStartCol, AEndCol **:** TLazSynDisplayTokenBound **;**

**const** AnRtlInfo **:** TLazSynDisplayRtlInfo **;**

AMarkup **:** TSynSelectedColorMergeResult **);** **virtual ;**

**//** Notifications about Changes **to** the text

**Procedure** TextChanged **(** aFirstCodeLine, aLastCodeLine, ACountDiff **:** Integer **);** **virtual ;** **//** 1 based

**Procedure** TempDisable **;**

**Procedure** TempEnable **;**

**procedure** IncPaintLock **;** **virtual ;**

**procedure** DecPaintLock **;** **virtual ;**

**function** RealEnabled **:** Boolean **;** **virtual ;**

**property** MarkupInfo **:** TSynSelectedColor **read** fMarkupInfo **;**

**property** FGColor **:** TColor **read** GetFGColor **;**

**property** BGColor **:** TColor **read** GetBGColor **;**

**property** FrameColor **:** TColor **read** GetFrameColor **;**

**property** FrameStyle **:** TSynLineStyle **read** GetFrameStyle **;**

**property** Style **:** TFontStyles **read** GetStyle **;**

**property** Enabled **:** Boolean **read** GetEnabled write SetEnabled **;**

**property** Lines **:** TSynEditStrings **read** fLines write SetLines **;**

**property** Caret **:** TSynEditCaret **read** fCaret write SetCaret **;**

**property** TopLine **:** Integer **read** fTopLine write SetTopLine **;**

**property** LinesInWindow **:** Integer **read** fLinesInWindow write SetLinesInWindow **;**

**property** InvalidateLinesMethod **:** TInvalidateLines write SetInvalidateLinesMethod **;**

**end ;**

To create a new marker we must create an object of type “TsynEditMarkup” or one of its descendants.

We have already seen how to create a marker in section 1.8.2- Highlighting the current word, when we created an object of the class “TSynEditMarkupHighlightAllCaret”, because this class derives from “TsynEditMarkup”. Also the “TSynEditMarkupHighlightAll” class, used to mark any word, is a descendant of “TsynEditMarkup”.

The SynEdit component comes with several markers, defined by default. An exploration of them is achieved by iterating over the Markup[] table:

**//** iterate through markers

**for** i **:=** 0 **to** SynEdit1.MarkupCount **-** 1 **do begin**

tmp **:=** SynEdit1.Markup[i].MarkupInfo.StoredName **;** **//** read name

ShowMessage **(** tmp **);**

**end ;**

Most predefined bookmarks in SynEdit have their “StoredName” field empty, so the previous loop will show null strings, but if we had created a named bookmark, we could place it this way.

The following example shows how to disable a bookmark created with the name “CurrentWordRes”:

**var**

mark **:** TSynEditMarkup **;**

...

mark **:=** **nil ;**

**//** search bookmark by name

**for** i **:=** 0 **to** ed.MarkupCount **-** 1 **do begin**

tmp **:=** ed.Markup[i].MarkupInfo.StoredName **;**

**if** ed.Markup[i].MarkupInfo.StoredName **=** 'CurrentWordRes' **then**

marc **:=** ed.Markup[i] **;**

**end ;**

**if** marc **<> nil then begin //** there is a marker

marc.Enabled **:= false ;** **//** deactivate

**end ;**

Another way to locate a marker in SynEdit is to use the MarkupByClass[] method, which allows us to specify the name of the class of the marker we want to find:

marc **:=** SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret] **;**

If there are two markers of the same class, the first one will be returned.

The following code shows how to define the border color of the highlighted text, by the TSynEditMarkupHighlightAllCaret marker:

**var**

mark **:** TSynEditMarkup **;**

...

marc **:=** SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret] **;**

TSynEditMarkupHighlightAllCaret **(** marc **)** .MarkupInfo.FrameColor **:=** clGreen;

To access the visual properties of highlighted text, the “MarkupInfo” property of the highlighter is used.

The “MarkupInfo” property is a descendant of the “TSynHighlighterAttributes” class (See Section 2.1.1- Key concepts), therefore it behaves as a container for the appearance properties of a text.

When creating a marker, it is convenient to give it a name and save it in the “StoredName” property of “MarkupInfo” to be able to locate it later from the editor.

### More about bookmarks

The markers seen in the previous sections, which highlight the appearance of the displayed text, are relatively simple (highlighting a word or a line). However, SynEdit's bookmarks are quite elaborate, and allow you to implement cases of complex text marking.

An example of more complete markers are those used by the Lazarus editor, which allow highlighting the beginning and end of code blocks such as BEGIN-END or REPEAT-UNTIL.

The following example, assuming a form, with a SynEdit control on it, shows how to create a simple bookmark, which highlights a block of text, on the first line:

**unit** Unit1 **;**

{$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Forms, Controls, Graphics, SynEdit,

SynEditTypes, SynEditMarkupHighAll **;**

**type**

TMarkup **=** **class (** TSynEditMarkupHighlightMatches **);**

{ TForm1 }

TForm1 **=** **class (** TForm **)**

SynEdit1 **:** TSynEdit **;**

**procedure** FormCreate **(** Sender **:** TObject **);**

**private**

Markup **:** TMarkup **;** **//** marker 1

**end ;**

**var**

Form1 **:** TForm1 **;**

**implementation**

{$R **\*** .lfm}

{ TForm1 }

**procedure** TForm1.FormCreate **(** Sender **:** TObject **);**

**begin**

Markup **:=** TMarkup.Create **(** SynEdit1 **);**

Markup.MarkupInfo.FrameColor **:=** clRed **;**

Markup.MarkupInfo.FrameEdges **:=** sfeBottom **;**

SynEdit1.MarkupManager.AddMarkUp **(** Markup **);** **//** add marker

Markup.Matches.StartPoint[0] **:=** Point **(** 2,1 **);**

Markup.Matches.EndPoint[0] **:=** Point **(** 4,1 **);**

Markup.InvalidateSynLines **(** 1,1 **);**

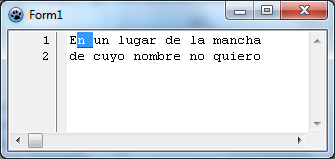
**end ;**

**end** .

In this case, the TSynEditMarkupHighlightMatches class is used as the basis for the marker, and since access to the protected property “Matches” of TSynEditMarkupHighlightMatches is necessary, the TMarkup class is defined. Otherwise TSynEditMarkupHighlightMatches would have been enough.

The block to be highlighted is defined with coordinates of type TPoint, in StartPoint[] and EndPoint[]. Note that by default, the bookmark is enabled.

The effect, in the text, is as indicated in the figure:



To create more elaborate bookmarks, you must access SynEdit protected fields, so you must create a class that derives from SynEdit and create our editor from that class. Once our editor is defined, we will have the possibility of creating any marker, whose logic must be defined by code.

The following code creates an editor from SynEdit, creating a descendant class of SynEdit (subclass), and defines two blocks of text to highlight them:

**unit** Unit1 **;** {$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Forms, Controls, Graphics,

SynEdit, SynEditMarkupSelection,

SynEditTypes, SynEditPointClasses, SynEditMarkup **;**

**type**

{ TMiEditor }

TMiEditor **=** **class (** TSynEdit **)** **//** defines my Editor class

**private**

Block1 **:** TSynEditSelection **;** **//** selection block 1

Markup1 **:** TSynEditMarkupSelection **;** **//** marker 1

Block2 **:** TSynEditSelection **;** **//** selection block 2

Markup2 **:** TSynEditMarkupSelection **;** **//** marker 2

**public**

**constructor** Create **(** AOwner **:** TComponent **);** **override ;**

**destroyer** Destroy **;** **override ;**

**end ;**

{ TForm1 }

TForm1 **=** **class (** TForm **)**

**procedure** FormCreate **(** Sender **:** TObject **);**

**procedure** FormDestroy **(** Sender **:** TObject **);**

**private**

MyEditor **:** TMiEditor **;**

**end ;**

**var**

Form1 **:** TForm1 **;**

**implementation**

{ TMiEditor }

**constructor** TMiEditor.Create **(** AOwner **:** TComponent **);**

**var**

MarkupManager **:** TSynEditMarkupManager **;**

**begin**

**inherited** Create **(** AOwner **);**

MarkupManager **:=** TSynEditMarkupManager **(** MarkupMgr **);** **//** MarkupMgr is "TObject"

**//** create a block for highlighting

Block1 **:=** TSynEditSelection.Create **(** ViewedTextBuffer, **false );**

Block1.InvalidateLinesMethod **:=** @InvalidateLines **;**

Markup1 **:=** TSynEditMarkupSelection.Create **( self** , Block1 **);**

MarkupManager.AddMarkUp **(** Markup1 **);** **//** add marker

**//** create a block for highlighting

Block2 **:=** TSynEditSelection.Create **(** ViewedTextBuffer, **false );**

Block2.InvalidateLinesMethod **:=** @InvalidateLines **;**

Markup2 **:=** TSynEditMarkupSelection.Create **( self** , Block2 **);**

MarkupManager.AddMarkUp **(** Markup2 **);** **//** add marker

**end ;**

**destructor** TMiEditor.Destroy **;**

**begin**

Block1.Free **;** **//** Markup1, destroyed with SynEdit

Block2.Free **;** **//** Markup2, destroyed with SynEdit

**inherited** Destroy **;**

**end ;**

{$R **\*** .lfm}

{ TForm1 }

**procedure** TForm1.FormCreate **(** Sender **:** TObject **);**

**begin**

MyEditor **:=** TMiEditor.Create **( self );** **//** create my Editor component

MyEditor.Parent **:=** **self ;** **//** places it in the form

MyEditor.Align **:=** alClient **;** **//** align it

**//** write a text

MyEditor.Lines.Add **(** 'In a spot' **);**

MyEditor.Lines.Add **(** 'whose name' **);**

MyEditor.Lines.Add **(** 'I don't want to remember' **);**

**//** define section 1 to highlight

MyEditor.Block1.StartLineBytePos **:=** Point **(** 5,1 **);**

MyEditor.Block1.EndLineBytePos **:=** Point **(** 8,1 **);**

MyEditor.Markup1.Enabled **:=** **True ;**

MyEditor.Markup1.MarkupInfo.FrameColor **:=** clRed **;**

**//** define section 2 to highlight

MyEditor.Block2.StartLineBytePos **:=** Point **(** 8,3 **);**

MyEditor.Block2.EndLineBytePos **:=** Point **(** 13,3 **);**

MyEditor.Markup2.Enabled **:=** **True ;**

MyEditor.Markup2.MarkupInfo.FrameColor **:=** clRed **;**

**end ;**

**procedure** TForm1.FormDestroy **(** Sender **:** TObject **);**

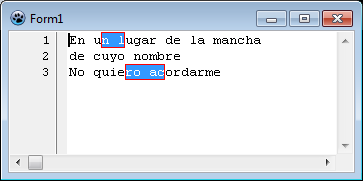
**begin**

MyEditor.Destroy **;** **//** release

**end ;**

**end** .

The effect on the text will be similar to that shown in the following figure:



For this highlighting, the “TSynEditSelection” class has been used, defined in the “SynEditPointClasses” unit, which allows specifying selection areas within the editor. By default it has the same background color as the selection, but it can be changed with the “MarkupInfo.Background” property.

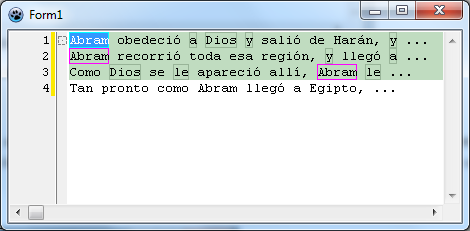
An observation in this highlighting is that the highlighted areas have fixed coordinates, even if the text is modified. If you want the highlighting to follow the text, the necessary logic must be implemented by code.

## Using plugins

Some SynEdit functionalities are offered in the form of “plugins”, which are nothing more than units that add additional features to SynEdit.

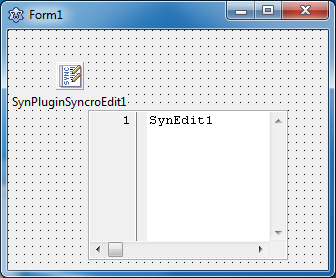
### Synchronous editing.

Synchronous editing is what allows you to edit the same identifier in different parts of a text.



To start it, you must select a block of text and press the Ctrl-J combination, which by default starts synchronous editing. At that moment the color of the selection will change and the identifiers that are the same as the one where the cursor is located will be marked. If in this state, the current identifier is edited, all other similar identifiers in the selection will change.

The simplest way to implement synchronous editing would be to add a SynPluginSyncroEdit element from the components palette and then associate it with a SynEdit element, setting its “Editor” property.



Another way would be to do it by code, including the SynPluginSyncroEdit unit and creating a TSynPluginSyncroEdit object. Also, in this case, you must set its “Editor” property to the SynEdit to which you want to add synchronous editing functionality.

### Multiple cursors.

SynEdit can handle editing across multiple cursors, just as if there were just one.

To enable this functionality, you must include the SynPluginMultiCaret unit, and create an object of the TSynPluginMultiCaret class, and configure its “Editor” field, to the SynEdit to which we want to add the multiple cursor functionality.

The following code configures a SynEdit, for editing with multiple, cursors, adding the combination Ctrl+Shift+Click, to position a new cursor:

fMultiCaret **:=** TSynPluginMultiCaret.Create **( self );**

**with** fMultiCaret **do begin**

Editor **:=** SynEdit1 **;**

**with** KeyStrokes **do begin**

Add.Command **:=** ecPluginMultiCaretSetCaret **;**

Add.Key **:=** VK\_INSERT **;**

Add.Shift **:=** [ssShift, ssCtrl] **;**

Add.ShiftMask **:=** [ssShift,ssCtrl,ssAlt] **;**

**end ;**

**end ;**

Then when you hold down the Ctrl+Shift keys and click the mouse anywhere in the text, a new cursor enabled for editing will be located there.

When you have multiple cursors, the action you take will be applied to all created cursors, including inserting and deleting characters.

To create a new cursor, by code you can use the method:

SynEdit1.CommandProcessor **(** ecPluginMultiCaretSetCaret, '' , **nil );**

## Summary of Properties and Methods

|  |  |
| --- | --- |
| **PROPERTY** | **DESCRIPTION** |
| BeginUpdate, EndUpdate | Disables and Enables (respectively) SynEdit screen refresh. See 1.5.4- Do and Undo. |
| BeginUndoBlock EndUndoBlock | They allow several change actions to be grouped together, to be undone with a single call to Undo. See 1.5.4- Do and Undo. |
| BlockBegin  BlockEnd | They indicate the coordinates of the selected text. See Section 1.6- Selection management |
| BlockIndent | Indicates the number of spaces used to indent a block of text, when block indentation actions are executed (In the Lazarus IDE they are mapped as Ctrl+U and Ctrl+I). |
| BookMarkOptions | It is a set of options that allow you to configure text markers. |
| BracketHighlightStyle | Configures the highlighting behavior of parentheses, braces, and square bracket delimiters. Indicates how highlighting is determined. They may be:  sbhsBoth  sbhsLeftOfCursor  sbhsRightOfCursor |
| BracketMatchColor | Attribute used to highlight parentheses, braces, and square bracket delimiters. |
| CanUndo | Indicates if there are actions to be undone in the editor |
| CanRedo | Indicates if there are actions to be redone in the editor |
| CanPaste | Indicates if there is content to paste into the editor |
| CaretX | They allow you to read or set the horizontal coordinate of the editor cursor. The first character has coordinate 1. |
| CaretY | They allow you to read or set the vertical coordinate of the editor cursor. The first row has coordinate 1. |
| CaretXY | They allow you to read or set the X, Y coordinates of the editor cursor. See Section 1.4.1- Editor |
| ClearAll | Delete all content from the editor. |
| ClearSelection | Delete the selected text. |
| ClearUndo | Clear and reset “undo” memory. Once executed, the changes cannot be undone. |
| Color | Editor background color. |
| CopyToClipboard | Copies the selected text to the clipboard. See Section 1.5.3- The Clipboard. |
| CutToClipboard | Cuts the selected text to the clipboard. See Section 1.5.3- The Clipboard. |
| ExecuteCommand | Send a command to the editor. See section 1.5.1- Run commands |
| ExtraCharSpacing | Indicates the spacing between letters. By default it is zero. See Section 1.3.3- Typography. |
| ExtraLineSpacing | Indicates the spacing between lines. By default it is zero. See Section 1.3.3- Typography. |
| FoldAll | Allows you to close all folding blocks that exist in an editor. You can also fold in levels. |
| Font | Object that defines the properties of the font to use in the editor. It has various properties such as font name, size, character set, etc. See Section 1.3.3- Typography |
| Gutter | Reference to the object that defines the side panel of the editor, where the line number usually appears. |
| GetHighlighterAttriAtRowCol | Reads the token and attribute for a specific position in the text. It is only valid when you have a highlighter associated with the editor. |
| Highlighter | Reference to the highlighter to be used to implement syntax highlighting (See Section 2.3- Syntax Coloring Using Code). |
| InsertMode | Allows you to go from normal mode to INSERT mode, where the entered characters are overwritten. When set to FALSE, you enter insert mode. |
| InsertTextAtCaret | Inserts text at the current cursor position. See Section 1.5- Modify the content |
| Keystrokes | Stores keyboard shortcuts, and the commands to which those keyboard shortcuts are associated. |
| LogicalCaretXY | Returns the position of the cursor in logical coordinates. Measured in bytes (not characters). |
| LineHighlightColor | It is the background color of the line the cursor is currently on. See section 1.8.3- Highlighting the current line |
| LinesInWindow  CharsInWindow | Indicates the number of Lines and columns visible on the screen. It depends only on the size of the screen and the spacing between characters and lines. |
| lines | List of all publisher content. It is a list of strings (similar to TStringList), each item represents a line. Start at element 0. |
| LineText | Always stores the content of the current line. |
| MaxLeftChar | Limits the horizontal position of the cursor when in “Floating cursor” mode - See section 1.4.2- Cursor handling |
| MaxUndo | Maximum number of operations to undo. |
| Modified | Indicates when the editor's content has been modified. It can also be written. |
| MoveCaretIgnoreEOL | Positions the cursor without considering the limits of the destination line. |
| Options | Various additional options to configure the editor, such as automatic indentation, the behavior of the cursor outside the line limits, or the conversion of tabs into spaces. See section 1.10.1- Options |
| PasteFromClipboard | Pastes the text from the clipboard at the cursor position. See Section 1.5.3- The Clipboard. |
| Redo | Perform an undo action again with “Undo”. See Section 1.5.4- Do and Undo. |
| RightEdge | Indicates the position of the vertical line (right margin) used to mark the boundary of the print area. See section 1.3- Appearance |
| RightEdgeColor | Color of the right margin. |
| SelectAll | Select all content from the editor |
| SelText | Property that allows you to read or modify the selected text. See Section 1.6- Selection management |
| SelectionMode | Cursor selection mode. Allows you to change the normal text selection mode. |
| SearchReplace | Allows search and replacement of text within the editor. See Section 1.7- Search and Replacement |
| SelectWord | Selects the current word where the cursor is located. See Section 1.6- Selection management. |
| SelectLine | Select the current line where the cursor is located. See Section 1.6- Selection management. |
| SelectParagraph | Select the current paragraph where the cursor is located. See Section 1.6- Selection management. |
| SelectToBrace | Selects the block delimited by parentheses, braces, or square brackets. See Section 1.6- Selection management. |
| ScrollBars | Controls the visibility of the horizontal and vertical scroll bars. |
| TabWidth | It is the number of spaces used to represent a tab stop. This option can be ignored in Smart Tab mode (Options.eoSmartTabs = TRUE) |
| TextBetweenPoints | Allows you to read or modify the content of a text block. |
| TextBetweenPointsEx | Allows you to read or modify the content of a text block, with cursor control. |
| TopLine | Indicates the number of the first line that is visible in the current editor window. |
| undo | Undoes the last action performed on the editor. See Section 1.5.4- Do and Undo. |
| UnfoldAll | Allows you to expand all folding blocks that have been closed. |

### Options and Options2 property

There are various parameters that can be configured using the “Options” property of SynEdit. The common syntax is:

SynEdit1.Options **:=** [eoKeepCaretX, eoTabIndent, ... ] **;**

This property is a set of type “TSynEditorOptions” that can include the following elements:

|  |  |
| --- | --- |
| **WORTH** | **DESCRIPTION** |
| eoAutoIndent | Positions the cursor on the new line with the same number of spaces as the previous line. |
| eoBracketHighlight | It will highlight the parenthetical delimiters, braces, and brackets. |
| eoEnhanceHomeKey | “Home” key will jump to the beginning of the line if it is closer (similar to Visual Studio) |
| eoGroupUndo | Groups all changes of the same type into a single Undo/Redo action, instead of handling each command separately. |
| eoHalfPageScroll | Scrolling page-up or page-down will only skip half a page at a time. |
| eoHideRightMargin | Hides the right margin line. |
| eoKeepCaretX | Limits the position of the cursor to move, only to the end of the line. Otherwise it can go beyond the line. See Section 1.4.2- Cursor handling |
| eoNoCaret | Makes the cursor invisible. |
| eoNoSelection | Disable text selection. |
| eoPersistentCaret | Does not hide the cursor when focus is lost. |
| eoScrollByOneLess | Page-up or page-down movements will be with one less line. |
| eoScrollPastEof | They allow the cursor to move past the end of file mark. |
| eoScrollPastEol | Allows the cursor to move past the last character of a line. It works even when eoKeepCaretX is present. |
| eoScrollHintFollows | Label scrolling (Hint) follows mouse scrolling when moving vertically. |
| eoShowScrollHint | Displays a label (Hint) with the number of the first visible line, when scrolling vertically. |
| eoShowSpecialChars | Shows special characters. |
| eoSmartTabs | When you tab the cursor will be located at the position of the next white space on the previous line. |
| eoTabIndent | The <Tab> and <Shift><Tab> keys will function to indent or unindent a selected block. |
| eoTabsToSpaces | Causes the <Tab> key to insert spaces (specified in the “TabWidth” property) instead of the TAB character. It does not convert existing tab characters, it simply prevents more from being added. |
| eoTrimTrailingSpaces | Spaces at the end of lines will be trimmed and not saved. |

In addition to “Options”, other values can be included in the “Options2” property:

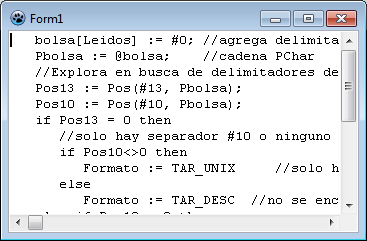
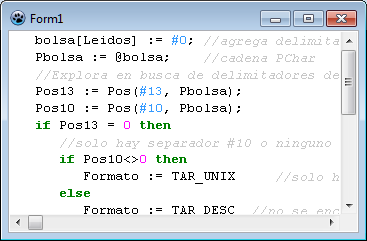
|  |  |
| --- | --- |
| **WORTH** | **DESCRIPTION** |
| eoCaretSkipsSelection | The cursor will jump over the selection when using the left and right arrow keys. |
| eoCaretSkipTab | The cursor will skip all the spaces that make up a tab stop. Without this option, the cursor will move through the tab stops as if they were spaces. |
| eoAlwaysVisibleCaret | Moves the cursor so that it is always visible when scrolling. |
| eoEnhanceEndKey | Pressing the <End> key moves to the end of the line, but without considering trailing spaces. |
| eoFoldedCopyPaste | Maintains folding properties in copy/paste operations. |
| eoPersistentBlock | Maintains selection blocks, even when the cursor is outside the block. |
| eoOverwriteBlock | Does not maintain persistent blocks. It overwrites them in Insert/Delete operations. |
| eoAutoHideCursor | Allows you to hide the cursor with keyboard operations. |

# Syntax Highlighting and Autocomplete with SynEdit.

## Introduction

Syntax highlighting is the ability to give different visual attributes to each element of a text, so that it makes it more readable. These attributes are only displayed in the editor and are not part of the text when saved to a file.

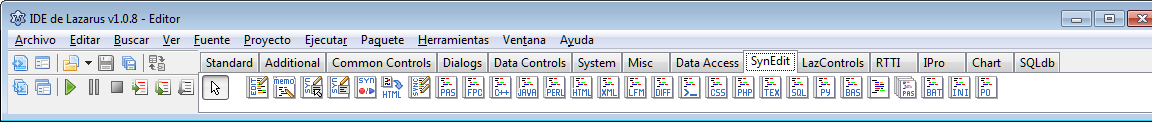
The following figure shows a piece of code in Pascal, displayed in a simple editor and in another with syntax highlighting:

The difference is evident. Code with syntax highlighting is easier to understand and read. This highlighting, however, must be done consistently and simply, otherwise it could confuse, rather than help.

Syntax highlighting is also called “Syntax coloring” because of the common use of color as a way to differentiate text elements, but color is only one of the attributes that can be used, in a more general context, that It can include, for example, the font. To generalize the use of diverse attributes, we use the term “Syntax Highlighting”, but we will also use “Syntax Coloring” as a synonym.

The TSynEdit component does not include built-in highlighters in the control itself, but must be associated with an external highlighter. For this, the SynEdit package includes various highlighter-type components (from various languages), which can be used with SynEdit.



We can also create our own highlighter, if we don't find one that suits our needs.

### Key concepts

In Section 1.4.1- Editor , we saw how SynEdit interprets the data bytes to get to the screen cell level. And that generally one character will occupy one cell.

To implement syntax highlighting, you need to raise the level of parsing. Characters are not appropriate for determining highlighting (unless you want character-by-character highlighting), it is better to use the concept of “token”.

Let's first review some important concepts, before moving forward with the topic:

* **Character.-** They are the minimum visual components that make up a text. They obey a defined encoding (UTF-8 in the case of SynEdit). They allow you to define larger elements such as tokens. They usually take up one cell on the screen, but could take up two.
* **Token.-** It is the unit of information composed of one or more characters, which are grouped by specific rules, and which make sense in a specific syntax. Tokens are the parts into which a programming language is divided, at a lexical level.
* **Attribute** - These are the highlighting properties that apply to a particular token. In SynEdit, an attribute is represented by an object of the “TSynHighlighterAttributes” class and allows you to define properties such as text color, background color, font, bold activation, and other properties.

Syntax highlighting starts at the token level and each token can have different attributes from the rest of the tokens. It can be said that the minimum unit of information for syntax highlighters in SynEdit is the token.

El token es la unidad mínima de información para el resaltado de sintaxis en el componente SynEdit

The following figure will help clarify the idea:



As you can see, tokens can be of various types, and attributes must be applied to each type of token (not to each token). Thus, all strings must have the same attributes in the same text.

Common types of tokens are:

* Identifier.- Some of them can be keywords
* Number.- They can have different representation.
* Space.- All characters that are not visible
* String.- Usually enclosed in single or double quotes.
* Comment.- They can be one or more lines.
* Symbol.- Various symbols as operators.
* Control.- May include line breaks or end marks.

The way characters are grouped into tokens (and their types) is not pre-defined by SynEdit. There are no established rules. When SynEdit performs syntax highlighting, it uses a highlighter, and it is this highlighter that defines the way tokens are identified in the text.

The predefined highlighters already include these definitions, but if we create a custom highlighter, we must define these rules ourselves.

Simplifying the syntax highlighting process, we could say that it consists of separating the text into tokens and giving a color (attribute) to each token. This simplistic description is, however, very true, but the process is more complicated than it sounds. The greatest difficulty lies in identifying each of the tokens accurately and fairly quickly.

Tokens are the basic elements that a lexical analyzer (lexer) would deal with. A syntax highlighter is, in a way, a lexical analyzer, but its objective is not to serve as a basis for subsequent syntactic or semantic analysis, but rather to identify the tokens and their types to facilitate highlighting [[6]](#footnote-6).

## Syntax Coloring Using Predefined Components

This type of syntax coloring involves the use of predefined syntax components. These components come prepared to work in a predefined language.

### Using a predefined language

These syntax components come already installed in the Lazarus environment. There are components for most common languages such as: Pascal, C++, Java, HTML, Python, etc.

The method is simple: We drag the control to the form, where the SynEdit control that we are going to use is located. Then we link it, using the “highlighter” property of SynEdit.

Later we can choose the colors and attributes of the keywords, comments, numbers, and other categories, accessing the properties of the “TSynXXXSyn” object (where XXX represents the chosen language). Each “TSynXXXSyn” control is represented by one unit.

The “TSynXXXSyn” object is optimized to work in its predefined language and responds well in speed. However, the keywords and their detection are hardcoded into the code of your unit. Trying to add one more reserved word is not so simple, due to the way the keyword search is optimized.

The use of predefined components saves us all the work of having to process a complete syntax of a known language.

### Using a custom language

In addition to the predefined language components, there is a component that can be customized for a different language. This is the “TSynAnySyn” component.

To use this component, just include it in the form, like any other syntax component. But before using it, you must be told which keywords make up its syntax.

Although this component supports the definition of several simple languages, it does not allow much flexibility when handling comments.

Furthermore, this component does not perform well in terms of speed, because its multi-language design does not allow it to be optimized properly. It is not recommended to use this component with language with many keywords or in large texts.

If you want to implement a new language with good performance and highly personalized, it is advisable to do it by code.

## Syntax Coloring Using Code

This type of coloring allows you to efficiently customize a specific syntax. It should be used when the language to be used does not already exist, in one of the predefined components, or does not comply with the required behavior.

Before designing our class to handle coloring, we need to understand a little about how syntax coloring works:

For syntax coloring, a descendant class of “TSynCustomHighlighter” must be created, we will call this class highlighter, or syntax class, from which we will instantiate a highlighter. This highlighter must then be associated with the SynEdit editor, which will implement syntax coloring.

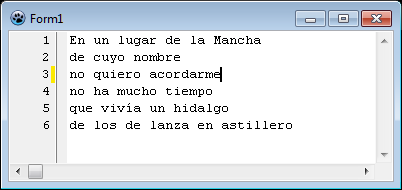
When a highlighter is associated with a “TSynEdit” editor, it will start calling the highlighter routines whenever it requires information about syntax coloring.

TSynEdit does not store text coloring information somewhere. Whenever you require coloring information, it will call the highlighter, to obtain, “on the fly”, the coloring information.

This highlighter must implement text analysis and identification of text elements to give text attribute information to SynEdit.

The exploration of the text to be colored is done by SynEdit, processing line by line. When a line is modified, the modified line and subsequent lines, starting from the modified line, are displayed in the visible window. This behavior is normal if you consider that a syntax element that affects the other lines of the text can be included in a line (such as the beginning of a multi-line comment).

Let's consider the case of an editor with the following text:



When the window is shown, after it has been hidden, the following events will be generated:

1. SetLine: In a place in La Mancha
2. SetLine: whose name
3. SetLine: I don't want to remember
4. SetLine: not long ago
5. SetLine: who lived a gentleman
6. SetLine: one of the lances in the shipyard

The “SetLine” label indicates that the displayed line is being scanned.

When modifying line number 3 of the same text (anywhere), the exploration sequence changes a little:

1. SetLine: I don't want to remember,
2. SetLine: not long ago
3. SetLine: who lived a gentleman
4. SetLine: whose name
5. SetLine: I don't want to remember,
6. SetLine: not long ago
7. SetLine: who lived a gentleman

We can see that the editor scans the next two lines, and then scans again, but starting a previous line.

### Syntax Highlighting Cases.

When performing syntax highlighting, we must consider various situations. 3 cases can be differentiated:

* Simple token highlighting. It is the common coloring in which certain words or key identifiers in the text are turned into a certain color. Various categories are normally defined such as reserved words, keywords, identifiers, variables, macros, etc. You can define various text attributes for each category. Tokens are identified by the characters they can contain.
* One-line comment highlighting. This coloring is typical of one-line comments in most languages. It involves putting a specific color on the text of a comment, from the beginning to the end of the line.
* Highlighting ranges of text or context. This coloring is also applied to comments, or chains. The coloring of the text affects a group of words, which can be on the same line or occupy several consecutive lines. Tokens are identified by their delimiters.

### Line Scan

Each line is assumed to be divided into elements called “tokens”. There is no part of a line that is not a token. A token can be an identifier, a symbol, a control character, a blank character, etc.

A token can be one or more characters long. Each token, or type of token, can have particular attributes.

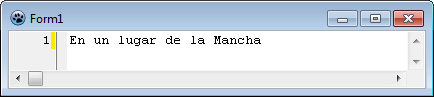
Every time SynEdit needs to process a line, it makes a sequence of calls to the highlighter (actually “TSynCustomHighlighter”, but it pipes them to the highlighter being used), to get the attributes of each element on the line:

* First, the “SetLine” method is called to indicate that the exploration of a new line is beginning. After this call expect the “GetTokenEx” or “GetTokenAttribute” method to return information about the current token.
* After the call to “SetLine”, multiple calls to the “Next” method will be generated, to access the following tokens.
* SynEdit expects that after each call to “Next”, the “GetTokenEx” and “GetTokenAttribute” methods return information about the currently pointed token.
* When SynEdit wants to check if the end of the work line has been reached, it will make a call to “GetEol”. This must be functional since “SetLine” is called.

Calls to these methods occur repeatedly and in large numbers for each line. Therefore, these methods must have a quick response and efficient implementation. Delay in processing any of these methods will affect the editor's performance.

To get an idea of the work that SynEdit does, in terms of syntax coloring, we present below the sequence of methods called when the editor window, which was hidden, is shown.

The example window contains this text:



The sequence of events called when displaying this window is:

1. SetLine: “In a place in La Mancha”
2. Next:0
3. GetEol
4. GetTokenEx
5. GetTokenAttribute
6. Next:2
7. GetEol
8. GetTokenEx
9. GetTokenAttribute
10. Next:3
11. GetEol
12. GetTokenEx
13. GetTokenAttribute
14. Next:5
15. GetEol
16. GetTokenEx
17. GetTokenAttribute
18. Next:6
19. GetEol
20. GetTokenEx
21. GetTokenAttribute
22. Next:11
23. GetEol
24. GetTokenEx
25. GetTokenAttribute
26. Next:12
27. GetEol
28. GetTokenEx
29. GetTokenAttribute
30. Next:14
31. GetEol
32. GetTokenEx
33. GetTokenAttribute
34. Next:15
35. GetEol
36. GetTokenEx
37. GetTokenAttribute
38. Next:17
39. GetEol
40. GetTokenEx
41. GetTokenAttribute
42. Next:18
43. GetEol
44. GetTokenEx
45. GetTokenAttribute
46. Next:24
47. GetEol

The value indicated after the Next Event corresponds to the initial character that is scanned. It could be observed that the editor always checks if the end has been reached, after each call to “Next”. If after a call to Next(), the editor returns TRUE in GetEol(), it will assume that it is at the final character of the line, and will no longer ask for attribute information.

The following sequence diagram will clarify the process:



The editor draws on the screen, always token by token. The GetTokenEx() method returns the token extension, and the GetTokenAttribute() method returns the attribute to apply to the text. That information is all SynEdit needs to draw a portion of the text with attributes on the screen.

This sequence, Next() – GetEol() - GetTokenEx() - GetTokenAttribute() , is repeated throughout the line until GetEol() returns TRUE, which will indicate that there are no more tokens left to explore on that line.

If we make a modification in our example text, such as inserting a comma at the end of the line, the following sequence is generated:

1. SetLine: “In a place in La Mancha”
2. Next:0
3. GetEol
4. Next:2
5. GetEol
6. Next:3
7. GetEol
8. Next:5
9. GetEol
10. Next:6
11. GetEol
12. Next:11
13. GetEol
14. Next:12
15. GetEol
16. Next:14
17. GetEol
18. Next:15
19. GetEol
20. Next:17
21. GetEol
22. Next:18
23. GetEol
24. Next:24
25. GetEol
26. Next:25
27. GetEol
28. SetLine: In a place in La Mancha,
29. Next:0
30. GetEol
31. GetTokenEx
32. GetTokenAttribute
33. Next:2
34. GetEol
35. GetTokenEx
36. GetTokenAttribute
37. Next:3
38. GetEol
39. GetTokenEx
40. GetTokenAttribute
41. Next:5
42. GetEol
43. GetTokenEx
44. GetTokenAttribute
45. Next:6
46. GetEol
47. GetTokenEx
48. GetTokenAttribute
49. Next:11
50. GetEol
51. GetTokenEx
52. GetTokenAttribute
53. Next:12
54. GetEol
55. GetTokenEx
56. GetTokenAttribute
57. Next:14
58. GetEol
59. GetTokenEx
60. GetTokenAttribute
61. Next:15
62. GetEol
63. GetTokenEx
64. GetTokenAttribute
65. Next:17
66. GetEol
67. GetTokenEx
68. GetTokenAttribute
69. Next:18
70. GetEol
71. GetTokenEx
72. GetTokenAttribute
73. Next:24
74. GetEol
75. GetTokenEx
76. GetTokenAttribute
77. Next:25
78. GetEol

It can be noted that the editor first performs a preliminary scan of the entire line before applying the attributes.

In addition to syntax coloring, SynEdit does its own independent scanning for detecting and highlighting “brackets” (parentheses, square brackets, curly braces, and quotes) when the cursor is pointing to any of these elements:

texto **(**texto entre paréntesis (otro texto)**)** más texto.

However, “SynEdit” allows the highlighter to collaborate with the identification of these delimiters. Why? Because the highlighter can provide additional information for highlighting the brackets, since it handles the attributes of the different parts of the text.

To serve the “brackets” functionality of SynEdit, the highlighter must correctly implement the methods: “GetToken”, and “GetTokenPos” and “GetTokenKind”.

How do they work? For an opening “bracket” to be associated with its corresponding closing “bracket”, it is verified that “tokenKind” returns the same value for both. If the scan finds a “bracket” with a different attribute, this will not be taken into account.

While these methods are not used for syntax coloring, they do determine the behavior of bracket highlighting.

These methods are called less frequently than the syntax coloring methods. They are only executed when the cursor points to a “bracket” or when one is added or removed.

If you have understood the syntax coloring process, we are now ready to take the first steps in implementing a code highlighter.

### First steps

First of all, it is advisable to create a special Unit to store the code of our new highlighter.

For this example we will create a unit called “uSyntax”, and we will include the units necessary for the creation of the objects to be used.

In this new unit we must necessarily create a class derived from “TSynCustomHighlighter” (defined in the “SynEditHighlighter” unit), for the creation of our highlighter:

{

Minimal unit that demonstrates the structure of a simple class that will be

used for syntax highlighting. It is not functional, it is only demonstrative.

Created by Tito Hinostroza: 08/04/2013

}

**unit** uSyntax **;** {$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Graphics, SynEditHighlighter **;**

**type**

{Class for creating a highlighter}

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

**protected**

posIni, posEnd **:** Integer **;**

linAct **:** **String ;**

**public**

**procedure** SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);** **override ;**

**procedure** Next **;** **override ;**

**function** GetEol **:** Boolean **;** **override ;**

**procedure** GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

**override ;**

**function** GetTokenAttribute **:** TSynHighlighterAttributes **;** **override ;**

**public**

GetToken **function :** **String ;** **override ;**

**function** GetTokenPos **:** Integer **;** **override ;**

**function** GetTokenKind **:** integer **;** **override ;**

**constructor** Create **(** AOwner **:** TComponent **);** **override ;**

**end ;**

**implementation**

**constructor** TSynMiColor.Create **(** AOwner **:** TComponent **);**

**//** Class constructor. Here you must create the attributes to use.

**begin**

**inherited** Create **(** AOwner **);**

**end ;**

**procedure** TSynMiColor.SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);**

{It is called by the editor, whenever you need to update the information of

colored on a line. After calling this function, it is expected that

GetTokenEx, return the current token. And also after every call to

"Next".}

**begin**

**inherited ;**

linAct **:=** NewValue **;** **//** copy the current line

endpos **:=** 1 **;** **//** points to the first character

Next **;**

**end ;**

**procedure** TSynMiColor.Next **;**

{It is called by SynEdit, to access the next Token. And it is executed by

each token of the current line. In this example it will always move one

character.}

**begin**

StartPos **:=** EndPos **;** **//** point to the next token

**If** posIni **>** length **(** linAct **)** **then //** End of line?

**exit //** exit

**else**

inc **(** posEnd **);** **//** move a character

**end ;**

**function** TSynMiColor.GetEol **:** Boolean **;**

{Indicates when the end of the line has been reached}

**begin**

Result **:=** posIni **>** length **(** linAct **);**

**end ;**

**procedure** TSynMiColor.GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

{Returns information about the current token}

**begin**

TokenStart **:=** @linAct[posIni] **;**

TokenLength **:=** posEnd **-** posIni **;**

**end ;**

**function** TSynMiColor.GetTokenAttribute **:** TSynHighlighterAttributes **;**

{Returns information about the current token}

**begin**

Result **:=** **nil ;**

**end ;**

{The following functions are used by SynEdit to manage the

braces, brackets, parentheses and quotes. They are not crucial for coloring

of tokens, but they must respond well.}

**function** TSynMiColor.GetToken **:** **String ;**

**begin**

Result **:=** '' **;**

**end ;**

**function** TSynMiColor.GetTokenPos **:** Integer **;**

**begin**

Result **:=** posIni **-** 1 **;**

**end ;**

**function** TSynMiColor.GetTokenKind **:** integer **;**

**begin**

Result **:=** 0 **;**

**end ;**

**end** .

This is probably the simplest class that can be implemented for syntax highlighting. However, this class will not highlight any text because it does not contain instructions for changing the text's attributes. It is limited to simply returning default values to “SynEdit” requests. It has no use, it is simply a minimalist demonstration example.

The methods that appear as “override” are the ones that need to be implemented to give our highlighter the coloring functionality.

With each call to “SetLine”, a copy of the string is saved in “linAct”, then this string is used to extract the tokens.

For each “Next” request, this unit only returns the next character found on the line and the attribute returned by “GetTokenAttribute” is always NIL, which means there are no attributes.

The methods “GetToken”, “GetTokenPos” and “GetTokenKind” also do not return significant values, but rather the corresponding null values.

The highlighter class we have created is called “TSynMiColor”. It is not possible to use the same “TSynCustomHighlighter” class as a highlighter, because this class is abstract, and is only used to properly channel the requirements of TsynEdit, when performing syntax coloring.

To use the new syntax, we must create an object and associate it with the TSynEdit component that we are going to use. If we have our main form in Unit1 and our TsynEdit object is called “editor”, then the code for using this syntax could be:

**unit** Unit1 **;**

{$mode objfpc}{$H **+** }

**interface**

**uses** ... uSyntax **;**

...

**var**

Syntax **:** TSynMiColor **;**

...

**procedure** TForm1.FormCreate **(** Sender **:** TObject **);**

...

Syntax **:=** TSynMiColor.Create **( Self );** **//** create highlighter

editor.Highlighter **:=** Syntax **;** **//** assign the syntax to the editor

**end ;**

**procedure** TForm1.FormDestroy **(** Sender **:** TObject **);**

**begin**

editor.Highlighter **:=** **nil ;** **//** remove highlighter

Syntax.Destroy **;** **//** release object

**end ;**

Understanding the basic operation of this working scheme is the first step in creating functional highlighters. If you don't understand how this simple example works, I recommend that you review the code before moving on to the following sections.

### Adding functionality to the syntax

The previous example was created for educational purposes only. The desired functionality is not met, but the structure that all kinds of syntax coloring should have is shown.

To begin, we must keep in mind that the methods to be implemented must be quick to execute. They should not be loaded with much processing, because they are called repeatedly for each modified line of the editor, so they do not allow delays, otherwise the editor will become heavy and slow.

The first modification that we must introduce is the string storage method. When you call “SetLine”, you must have information about the string. But the parent class “TSynCustomHighlighter” already saves a copy of the string before calling “SetLine”.

Therefore it is not efficient to create a new copy for us. It will be enough to save a reference, a pointer to this string, stored in “TSynCustomHighlighter”.

This involves modifying the “linAct” variable, so that it is a “PChar”, instead of a string. This is done in the class definition. The “SetLine”, “Next”, “GetEol”, “GetTokenEx” and “GetTokenAttribute” methods must also change:

Our unit skeleton would look like this:

{

Minimal unit that demonstrates the structure of a simple class that will be

used for syntax highlighting. It is not functional, it is only demonstrative.

Created by Tito Hinostroza: 08/04/2013

}

**unit** uSyntax **;** {$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Graphics, SynEditHighlighter **;**

**type**

{Class for creating a highlighter}

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

**protected**

posIni, posEnd **:** Integer **;**

linAct **:** PChar **;**

**public**

**procedure** SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);** **override ;**

**procedure** Next **;** **override ;**

**function** GetEol **:** Boolean **;** **override ;**

**procedure** GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

**override ;**

**function** GetTokenAttribute **:** TSynHighlighterAttributes **;** **override ;**

**public**

GetToken **function :** **String ;** **override ;**

**function** GetTokenPos **:** Integer **;** **override ;**

**function** GetTokenKind **:** integer **;** **override ;**

**constructor** Create **(** AOwner **:** TComponent **);** **override ;**

**end ;**

**implementation**

**constructor** TSynMiColor.Create **(** AOwner **:** TComponent **);**

**//** Class constructor. Here you must create the attributes to use.

**begin**

**inherited** Create **(** AOwner **);**

**end ;**

**procedure** TSynMiColor.SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);**

{It is called by the editor, whenever you need to update the information of

colored on a line. After calling this function, it is expected that

GetTokenEx, return the current token. And also after every call to

"Next".}

**begin**

**inherited ;**

linAct **:=** PChar **(** NewValue **);** **//** copy the current line

endPos **:=** 0 **;** **//** points to the first character

Next **;**

**end ;**

**procedure** TSynMiColor.Next **;**

{It is called by SynEdit, to access the next Token. And it is executed by

each token of the current line. In this example it will always move one

character.}

**begin**

StartPos **:=** EndPos **;** **//** point to the next token

**if** linAct[posIni] **=** #0 **then exit ;** **///** does it point to the end?

inc **(** posEnd **);** **//** move a character

**end ;**

**function** TSynMiColor.GetEol **:** Boolean **;**

{Indicates when the end of the line has been reached}

**begin**

Result **:=** linAct[posIni] **=** #0 **;**

**end ;**

**procedure** TSynMiColor.GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

{Returns information about the current token}

**begin**

TokenLength **:=** posEnd **-** posIni **;**

TokenStart **:=** linAct **+** posIni **;**

**end ;**

**function** TSynMiColor.GetTokenAttribute **:** TSynHighlighterAttributes **;**

{Returns information about the current token}

**begin**

Result **:=** **nil ;**

**end ;**

{The following functions are used by SynEdit to manage the

braces, brackets, parentheses and quotes. They are not crucial for coloring

of tokens, but they must respond well.}

**function** TSynMiColor.GetToken **:** **String ;**

**begin**

Result **:=** '' **;**

**end ;**

**function** TSynMiColor.GetTokenPos **:** Integer **;**

**begin**

Result **:=** posIni **-** 1 **;**

**end ;**

**function** TSynMiColor.GetTokenKind **:** integer **;**

**begin**

Result **:=** 0 **;**

**end ;**

**end** .

Now we see that we must start “posFin” at zero, which is where the chain now begins, in “linAct”.

But even this class is empty of attributes. The first thing we should do is create our attributes. These must be declared as properties of the “TSynMiColor” object:

fAtriComment **:** TSynHighlighterAttributes **;**

fAtriIdent **:** TSynHighlighterAttributes **;**

fAtriKey **:** TSynHighlighterAttributes **;**

fAtriNumber **:** TSynHighlighterAttributes **;**

fAtriEspac **:** TSynHighlighterAttributes **;**

fAtriString **:** TSynHighlighterAttributes **;**

All attributes are of type “TSynHighlighterAttributes”. This class contains the various attributes that can be associated with a token, such as text color, background color, border color, etc.

Then in the constructor, we must create and define the properties of these attributes:

**constructor** TSynMiColor.Create **(** AOwner **:** TComponent **);**

**//Constructor** of the class. Here you must create the attributes to use.

**begin**

**inherited** Create **(** AOwner **);**

**//** comments attribute

fAtriComment **:=** TSynHighlighterAttributes.Create **(** 'Comment' **);**

fAtriComment.Style **:=** [fsItalic] **;** **//** in italics

fAtriComment.Foreground **:=** clGray **;** **//** gray font color

AddAttribute **(** fAtriComment **);**

**//** keyword attribute

fAtriClave **:=** TSynHighlighterAttributes.Create **(** 'Key' **);**

fAtriClave.Style **:=** [fsBold] **;** **//** in bold

fAtriClave.Foreground **:=** clGreen **;** **//** green font color

AddAttribute **(** fAtriKey **);**

**//** numbers attribute

fAtriNumber **:=** TSynHighlighterAttributes.Create **(** 'Number' **);**

fAtriNumero.Foreground **:=** clFuchsia **;**

AddAttribute **(** fAtriNumber **);**

**//** spaces attribute. No attributes

fAtriEspac **:=** TSynHighlighterAttributes.Create **(** 'space' **);**

AddAttribute **(** fAtriEspac **);**

**//** strings attribute

fAtriString **:=** TSynHighlighterAttributes.Create **(** 'String' **);**

fAtriString.Foreground **:=** clBlue **;** **//** blue font color

AddAttribute **(** fAtriString **);**

**end ;**

Note that the constants fsBold, fsItalic, ... are defined in the “Graphics” unit.

Attributes of several categories of tokens have been defined. This is where you define what the token text will look like.

To create an attribute, it is recommended to use the pre-defined constants in the “SynEditStrConst” unit:

SYNS\_AttrASP **=** 'Asp' **;**

SYNS\_AttrAssembler **=** 'Assembler' **;**

SYNS\_AttrAttributeName **=** 'Attribute Name' **;**

SYNS\_AttrAttributeValue **=** 'AttributeValue' **;**

SYNS\_AttrBlock **=** 'Block' **;**

SYNS\_AttrBrackets **=** 'Brackets' **;**

SYNS\_AttrCDATASection **=** 'CDATA Section' **;**

SYNS\_AttrCharacter **=** 'Character' **;**

SYNS\_AttrClass **=** 'Class' **;**

SYNS\_AttrComment **=** 'Comment' **;**

SYNS\_AttrIDEDirective **=** 'IDE Directive' **;**

SYNS\_AttrCondition **=** 'Condition' **;**

SYNS\_AttrDataType **=** 'Data type' **;**

SYNS\_AttrDefaultPackage **=** 'Default packages' **;**

SYNS\_AttrDir **=** 'Direction' **;**

SYNS\_AttrDirective **=** 'Directive' **;**

SYNS\_AttrDOCTYPESection **=** 'DOCTYPE Section' **;**

...

The way to create an attribute, using these constants, would be to first identify the type of attribute that we are going to create and choose the constant that best describes it. For most syntax, these would be:

SYNS\_AttrComment

SYNS\_AttrReservedWord

SYNS\_AttrNumber

SYNS\_AttrSpace

SYNS\_AttrString

SYNS\_AttrSymbol

SYNS\_AttrDirective

SYNS\_AttrAssembler

Therefore, to create an attribute for keywords, we could do:

fAtriClave **:=** TSynHighlighterAttributes.Create **(** SYNS\_AttrReservedWord, SYNS\_XML\_AttrReservedWord **);**

Using predefined constants to create the attributes is not mandatory or necessary for the highlighter to function, but it is a good practice if we want our highlighters to be able to work correctly with other Lazarus programs. For more information on attributes, see section 2.4.3- Attributes.

We must remember that all elements of the line to be explored must necessarily be a token, including spaces and symbols.

The following example shows how you can split a string into various tokens:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | eleven | 12 | 13 | 14 | fifteen | 16 | 17 | 18 | 19 | twenty | twenty-one | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| x | p |  | : | = |  | x | p |  | + |  | 1 | ; |  |  |  | / | / | c | either | m | and | n | t | to | r | Yo | either |

In this example the first token is defined by characters 1 and 2, and is shown in yellow. The second token is a blank space and is indicated by the color green. Characters 4 and 5 can be considered as a single token or as two different tokens. The character 12 is a token that will almost certainly be in the number category. Characters 14, 15 and 16 must be grouped into a single token space 3 characters wide (it would be inefficient to treat it as 3 tokens). Starting at character 17, there is a token that extends to the end of the line.

The limits of the token are defined by the highlighter (which functions as a token extractor or “lexer”). The editor will submissively follow what is indicated by this object, coloring it according to the attributes provided.

As seen in the example, all characters on the line must belong to a token. The size of a token ranges from one character to the total number of characters on the line. The editor will process the line faster, the fewer tokens there are in it.

To easily identify attributes, it is convenient to create an enumeration for token attributes:

**//** ID to categorize tokens

TtkTokenKind **=** **(** tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString, tkUnknown **);**

The identifier “tkUnknown” will indicate that the current token has not been identified. In this case, it will be assumed that it has no attributes.

And we also need a field to identify the current token:

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

...

fTokenID **:** TtkTokenKind **;** **//** Id of the current token

...

**end ;**

Now when we want to assign an attribute to the current token, we must put in “fTokenID”, the identifier of the token.

Once the attributes are created, we must add functionality to the “Next” method, so that it can extract the tokens properly from the line of work. The implementation must be as efficient as possible, which is why we will use the function or method table method.

The idea is to read the character of a token, and according to its ASCCI value, we call an appropriate function to handle that character. To make the call efficient, we create a table and fill it with pointers to the appropriate functions.

**Type**

TProcTableProc **=** **procedure of object ;** **//** Procedure type to process the

**//** token for the initial character.

...

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

**protected**

...

fProcTable **:** **array** [#0..#255] **of** TProcTableProc **;** **//** function table

...

**end ;**

The “TProcTableProc” type is a simple method that defines procedures without parameters (so the call becomes faster). This type of procedure is what will be called when the initial character of a token is identified.

Now that the type of procedure to be used has been defined, these token treatment procedures must be created and the method table filled with their addresses. The following code is a simple example of filling the method table:

...

**procedure** TSynMyColor.CreateMethodTable **;**

{Build the table of the functions used for each initial character of the

token to process. Provides a quick way to process a token by the

initial character}

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

'\_' ,'A'..'Z','a'..'z' **:** fProcTable[I] **:=** @ProcIdent **;**

#0 **:** fProcTable[I] **:=** @ProcNull **;** **//** End of string mark character

#1..#9, #11, #12, #14..#32 **:** fProcTable[I] **:=** @ProcSpace **;**

**else** fProcTable[I] **:=** @ProcUnknown **;**

**end ;**

**end ;**

This method maps the address of a function to each of the 256 positions in the “fProcTable[]” table.

The “ProcIdent” procedure is called when an alphabetic character (or hyphen) is detected, because it corresponds to the beginning of an identifier. Its implementation is simple:

**procedure** TSynMiColor.ProcIdent **;**

**//** Processes an identifier or keyword

**begin**

**while** linAct[posFin] **in** [ '\_' ,'A'..'Z','a'..'z'] **do**

Inc **(** posEnd **);**

fTokenID **:=** tkKey **;**

**end ;**

The string “linAct” is scanned until a character is found that is not a valid character for an identifier. Note that the extended ASCII code characters (á,é,í, etc.) are not considered. In this simple example, the type of identifier is not distinguished, but rather the “tkKey” attribute is assigned to all of them. If you would like to choose just a few words to mark as “tkKey”, you should do so here.

The “ProcNull” procedure is called when the NUL character is detected, that is, the end of the string. So its processing just boils down to marking “fTokenID” as “tkNull”.

**procedure** TSynMiColor.ProcNull **;**

**//** Processes the occurrence of character #0

**begin**

fTokenID **:=** tkNull **;** **//** You only need this to indicate that the end of the line has been reached

**end ;**

Note that no further progress is made in the exploration of the chain. This procedure is important to detect the end of the chain and allows you to implement “GetEol” in a simple way:

**function** TSynMiColor.GetEol **:** Boolean **;**

**//** Indicates when the end of the line has been reached

**begin**

Result **:=** fTokenId **=** tkNull **;**

**end ;**

The “ProcSpace” procedure allows processing blocks of blank spaces. For syntax purposes, the first 32 characters of the ASCII code will be considered blank spaces, except for characters #10 and #13 which correspond to line breaks:

**procedure** TSynMiColor.ProcSpace **;**

**//** Processes character that is start of space

**begin**

fTokenID **:=** tkSpace **;**

**repeat**

Inc **(** posEnd **);**

**until (** linAct[posEnd] **>** #32 **)** **or (** linAct[posEnd] **in** [#0, #10, #13] **);**

**end ;**

The tab character #9 and space #32 are considered blank spaces. These blank characters are assigned the “tkSpace” attribute, which should normally be unhighlighted.

The other important procedure is “ProcUnknown”, which is intended to process all those tokens that are not considered within a special category. In our case it will be all the symbols and numbers:

**procedure** TSynMiColor.ProcUnknown **;**

**begin**

inc **(** posEnd **);**

**while (** linAct[posEnd] **in** [#128..#191] **)** **OR //** continued utf8 subcode

**((** linAct[posEnd] **<>** #0 **)** **and (** fProcTable[linAct[posFin]] **=** @ProcUnknown **))** **do** inc **(** posEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

It is important to always have a procedure of this type to consider all those tokens that are not categorized into predefined groups. Note that the UTF-8 characters of the extended ASCII code are also considered. This is normal since SynEdIt works only with UTF-8.

Once these basic procedures are defined, the call must be implemented in the “Next” method. The code would have the following form:

**procedure** TSynMiColor.Next **;**

//It is called by SynEdit, to access the next Token.

**begin**

StartPos **:=** EndPos **;** **//** point to the next token

fProcTable[linAct[posEnd]] **;** **//** The corresponding function is executed.

**end ;**

Although it is not obvious, you can see the call to the appropriate processing function for each character. Obviously “fProcTable” must have been filled first.

This processing mode is quite fast compared to a set of conditionals or even a “case .. of” statement.

The assigned processing function will be responsible for updating the “posFin” index, which must always point to the beginning of the next token or the end of the chain.

For the syntax to be recognized, all that remains is to modify “GetTokenAttribute” to tell the editor which attribute to use for each token:

**function** TSynMiColor.GetTokenAttribute **:** TSynHighlighterAttributes **;**

**//** Returns information about the current token

**begin**

**case** fTokenID **of**

tkComment **:** Result **:=** fAtriComment **;**

tkKey **:** Result **:=** fAtriClave **;**

tkNumber **:** Result **:=** fAtriNumber **;**

tkSpace **:** Result **:=** fAtriSpace **;**

tkString **:** Result **:=** fAtString **;**

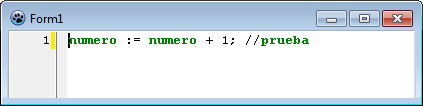
**else** Result **:=** **nil ;** **//** tkUnknown, tkNull

**end ;**

**end ;**

As we have defined our highlighter, all words will be recognized as keywords, and will be displayed in green. Symbols and other printable characters will be displayed without attributes, that is, they will take the default color of the text.

The following screen shows how a simple text would look, using this highlighter:



The complete code of the unit would look like this:

{

Minimal unit that demonstrates the structure of a simple class that will be used

for syntax highlighting. It is not functional, it is only demonstrative.

Created by Tito Hinostroza: 08/04/2013

}

**unit** uSyntax **;** {$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Graphics, SynEditHighlighter **;**

**type**

{Class for creating a highlighter}

**//** ID to categorize tokens

TtkTokenKind **=** **(** tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString, tkUnknown **);**

TProcTableProc **=** **procedure of object ;** **//** Procedure type to process the

**//** token for the initial character.

{TSynMiColor}

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

**protected**

posIni, posEnd **:** Integer **;**

linAct **:** PChar **;**

fProcTable **:** **array** [#0..#255] **of** TProcTableProc **;** **//** procedure table

fTokenID **:** TtkTokenKind **;** **//** Id of the current token

**//** define the categories of the "tokens"

fAtriComment **:** TSynHighlighterAttributes **;**

fAtriKey **:** TSynHighlighterAttributes **;**

fAtriNumber **:** TSynHighlighterAttributes **;**

fAtriEspac **:** TSynHighlighterAttributes **;**

fAtriString **:** TSynHighlighterAttributes **;**

**public**

**procedure** SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);** **override ;**

**procedure** Next **;** **override ;**

**function** GetEol **:** Boolean **;** **override ;**

**procedure** GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

**override ;**

**function** GetTokenAttribute **:** TSynHighlighterAttributes **;** **override ;**

**public**

GetToken **function :** **String ;** **override ;**

**function** GetTokenPos **:** Integer **;** **override ;**

**function** GetTokenKind **:** integer **;** **override ;**

**constructor** Create **(** AOwner **:** TComponent **);** **override ;**

**private**

**procedure** CreateMethodTable **;**

**procedure** ProcIdent **;**

**procedure** ProcNull **;**

**procedure** ProcSpace **;**

**procedure** ProcUnknown **;**

**end ;**

**implementation**

**constructor** TSynMiColor.Create **(** AOwner **:** TComponent **);**

**//Constructor** of the class. Here you must create the attributes to use.

**begin**

**inherited** Create **(** AOwner **);**

**//** comments attribute

fAtriComment **:=** TSynHighlighterAttributes.Create **(** 'Comment' **);**

fAtriComment.Style **:=** [fsItalic] **;** **//** in italics

fAtriComment.Foreground **:=** clGray **;** **//** gray font color

AddAttribute **(** fAtriComment **);**

**//** keyword attribute

fAtriClave **:=** TSynHighlighterAttributes.Create **(** 'Key' **);**

fAtriClave.Style **:=** [fsBold] **;** **//** in bold

fAtriClave.Foreground **:=** clGreen **;** **//** green font color

AddAttribute **(** fAtriKey **);**

**//** numbers attribute

fAtriNumber **:=** TSynHighlighterAttributes.Create **(** 'Number' **);**

fAtriNumero.Foreground **:=** clFuchsia **;**

AddAttribute **(** fAtriNumber **);**

**//** spaces attribute. No attributes

fAtriEspac **:=** TSynHighlighterAttributes.Create **(** 'space' **);**

AddAttribute **(** fAtriEspac **);**

**//** strings attribute

fAtriString **:=** TSynHighlighterAttributes.Create **(** 'String' **);**

fAtriString.Foreground **:=** clBlue **;** **//** blue font color

AddAttribute **(** fAtriString **);**

CreateMethodTable **;** **//** Build method table

**end ;**

**procedure** TSynMyColor.CreateMethodTable **;**

{Build the table of the functions used for each initial character of the token to be processed.

Provides a quick way to process a token by the initial character}

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

'\_' ,'A'..'Z','a'..'z' **:** fProcTable[I] **:=** @ProcIdent **;**

#0 **:** fProcTable[I] **:=** @ProcNull **;** **//** Read the end of string mark character

#1..#9, #11, #12, #14..#32 **:** fProcTable[I] **:=** @ProcSpace **;**

**else** fProcTable[I] **:=** @ProcUnknown **;**

**end ;**

**end ;**

**procedure** TSynMiColor.ProcIdent **;**

**//** Processes an identifier or keyword

**begin**

**while** linAct[posFin] **in** [ '\_' ,'A'..'Z','a'..'z'] **do**

Inc **(** posEnd **);**

fTokenID **:=** tkKey **;**

**end ;**

**procedure** TSynMiColor.ProcNull **;**

**//** Processes the occurrence of character #0

**begin**

fTokenID **:=** tkNull **;** **//** You only need this to indicate that the end of the line has been reached

**end ;**

**procedure** TSynMiColor.ProcSpace **;**

**//** Processes character that is start of space

**begin**

fTokenID **:=** tkSpace **;**

**repeat**

Inc **(** posEnd **);**

**until (** linAct[posEnd] **>** #32 **)** **or (** linAct[posEnd] **in** [#0, #10, #13] **);**

**end ;**

**procedure** TSynMiColor.ProcUnknown **;**

**begin**

inc **(** posEnd **);**

**while (** linAct[posEnd] **in** [#128..#191] **)** **OR //** continued utf8 subcode

**((** linAct[posEnd] **<>** #0 **)** **and (** fProcTable[linAct[posFin]] **=** @ProcUnknown **))** **do** inc **(** posEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**procedure** TSynMiColor.SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);**

{It is called by the editor, whenever you need to update the information of

colored on a line. After calling this function, it is expected that

GetTokenEx, return the current token. And also after every call to

"Next".}

**begin**

**inherited ;**

linAct **:=** PChar **(** NewValue **);** **//** copy the current line

endPos **:=** 0 **;** **//** points to the first character

Next **;**

**end ;**

**procedure** TSynMiColor.Next **;**

**//** Called by SynEdit, to access the next Token.

**begin**

StartPos **:=** EndPos **;** **//** point to the next token

fProcTable[linAct[posEnd]] **;** **//** The corresponding function is executed.

**end ;**

**function** TSynMiColor.GetEol **:** Boolean **;**

{Indicates when the end of the line has been reached}

**begin**

Result **:=** fTokenId **=** tkNull **;**

**end ;**

**procedure** TSynMiColor.GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

{Returns information about the current token}

**begin**

TokenLength **:=** posEnd **-** posIni **;**

TokenStart **:=** linAct **+** posIni **;**

**end ;**

**function** TSynMiColor.GetTokenAttribute **:** TSynHighlighterAttributes **;**

**//** Returns information about the current token

**begin**

**case** fTokenID **of**

tkComment **:** Result **:=** fAtriComment **;**

tkKey **:** Result **:=** fAtriClave **;**

tkNumber **:** Result **:=** fAtriNumber **;**

tkSpace **:** Result **:=** fAtriSpace **;**

tkString **:** Result **:=** fAtString **;**

**else** Result **:=** **nil ;** **//** tkUnknown, tkNull

**end ;**

**end ;**

{The following functions are used by SynEdit to manage the

braces, brackets, parentheses and quotes. They are not crucial for coloring

of tokens, but they must respond well.}

**function** TSynMiColor.GetToken **:** **String ;**

**begin**

Result **:=** '' **;**

**end ;**

**function** TSynMiColor.GetTokenPos **:** Integer **;**

**begin**

Result **:=** posIni **-** 1 **;**

**end ;**

**function** TSynMiColor.GetTokenKind **:** integer **;**

**begin**

Result **:=** 0 **;**

**end ;**

**end** .

### GetDefaultAttribute property

Someone may have asked themselves: How to access, from outside the class, the attributes of, for example, keywords? Remember that the attributes of the tokens must be declared in the highlighter and not in the parent class “TSynCustomHighlighter”.

A simple answer would be “we set the attribute properties to public, and then we can reference it as any property of our highlighter.

And it's true, that would work, but if the question were: How to access the attributes of the tokens from the editor? So there the situation gets a little complicated, because, even though the editor (of the TSyenEdit class) has the “HighLighter” property, it only refers to the “TSynCustomHighlighter” class and not to the derived class (highlighter). that we always use to implement coloring.

To partially solve this difficulty, there is an additional method that is recommended to be implemented. This method is “GetDefaultAttribute” and will allow our highlighter to respond to requests for access to the attributes generated by “TSynCustomHighlighter”.

Although the “TSynCustomHighlighter” class does not include attribute type properties (the programmer is left free to create the ones they want), it does include a way to access the main attributes of the tokens. Fixed properties have been defined in the class:

**property** CommentAttribute **:** TSynHighlighterAttributes **;**

**property** IdentifierAttribute **:** TSynHighlighterAttributes **;**

**property** KeywordAttribute **:** TSynHighlighterAttributes **;**

**property** StringAttribute **:** TSynHighlighterAttributes **;**

**property** SymbolAttribute **:** TSynHighlighterAttributes **;**

**property** WhitespaceAttribute **:** TSynHighlighterAttributes **;**

Which allow reading or modifying the indicated attributes. However, for these properties to work, we must override our highlighter with the following method:

**function** TSynMiColor.GetDefaultAttribute **( Index :** integer **):** TSynHighlighterAttributes **;**

{This method is called by the "TSynCustomHighlighter" class, when accessing any of

its properties **:** CommentAttribute, IdentifierAttribute, KeywordAttribute, StringAttribute,

SymbolAttribute or WhitespaceAttribute.

}

**begin**

**case Index of**

SYN\_ATTR\_COMMENT **:** Result **:=** fCommentAttri **;**

SYN\_ATTR\_IDENTIFIER **:** Result **:=** fIdentifierAttri **;**

SYN\_ATTR\_KEYWORD **:** Result **:=** fKeyAttri **;**

SYN\_ATTR\_WHITESPACE **:** Result **:=** fSpaceAttri **;**

SYN\_ATTR\_ **STRING :** Result **:=** fStringAttri **;**

**else** Result **:=** **nil ;**

**end ;**

**end ;**

As you can see, the idea is to give access to our attributes, according to the type of attribute requested. Of course, if we haven't defined a specific attribute, we could return NIL. Likewise, we may have defined additional attributes that might not be accessible from outside the class, because they are not in the requested category.

When someone accesses the “CommentAttribute” property of “TSynCustomHighlighter”, it calls “GetDefaultAttribute”, passing the “SYN\_ATTR\_COMMENT” parameter. It is the programmer's decision to return the attribute that he considers necessary. The usual thing would be to return the attribute that represents the comments, but the class will not do any subsequent validation. In theory we could return any attribute we want.

If there will be no access to the aforementioned properties of “TSynCustomHighlighter”, it is not necessary to implement “GetDefaultAttribute”, however, it is advisable to always implement it.

### Recognizing keywords.

In the previous example we marked all identifiers as keywords by assigning them the “tkKey” attribute. We did this in the “ProcIdent” method:

**procedure** TSynMiColor.ProcIdent **;**

**//** Processes an identifier or keyword

**begin**

**while** linAct[posFin] **in** [ '\_' ,'A'..'Z','a'..'z'] **do**

Inc **(** posEnd **);**

fTokenID **:=** tkKey **;**

**end ;**

But in a normal case, only some identifiers will be marked as keywords. To do this, the simplest way could be to compare the current token with a group of keywords, and only if they match, mark them as keywords.

The code to use could be like this:

**procedure** TSynMiColor.ProcIdent **;**

**//** Processes an identifier or keyword

**var** tam **:** integer **;**

**begin**

**while** linAct[posFin] **in** [ '\_' ,'A'..'Z','a'..'z'] **do**

Inc **(** posEnd **);**

tam **:=** EndPos **-** StartPos **;**

**if** strlcomp **(** linAct **+** posIni, 'EN' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'DE' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'LA' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

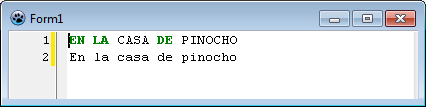
**if** strlcomp **(** linAct **+** posIni, 'LOS' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

The string comparison is done using the “strlcomp” function, because we are handling a “PChar” variable.

In this code, only the words “EN”, “LA” and “DE” are recognized as reserved words. By applying this modification we could have a screen like this.



Note that the routines only recognize capital letters, because the string comparison is done this way.

To add more words, you can increase the list and choose the attributes given to each category of words. However, this method becomes cumbersome as the number of words and conditionals to add grows and therefore is not the ideal way to follow in implementing an adequate syntax.

Later we will see how we can optimize the detection of identifiers.

### Optimizing the syntax.

As we have been insisting throughout this description, it is important to maintain fast code for the good performance of the editor. To do this we must identify the points where we can reduce execution cycles.

Analyzing the previous code you can see that the “ProcIdent” procedure is the heaviest in terms of processing. Due to its implementation, it requires making multiple comparisons and verifications to detect the identifiers to be colored.

The first optimization we will do has to do with the condition:

**while** linAct[posFin] **in** [ '\_' ,'A'..'Z','a'..'z'] **do**

Although using sets is efficient, this code can be considerably optimized by using a selection table.

Being able to quickly identify if a character is on a list is easy if we approach the problem from another perspective. Let's imagine that each character is associated with a table that contains as a value a simple flag that indicates whether or not the variable is part of the list. Such a structure would be of the type:

Identifiers **:** **array** [#0..#255] **of** ByteBool **;**

Now we create a fill procedure that checks only the boxes for valid characters for identifiers, such as “true”:

**procedure** CreateTableIdentif **;**

**var**

i **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**begin**

**Case i** **of**

'\_' , '0'..'9', 'a'..'z', 'A'..'Z' **:** Identifiers[i] **:=** **True ;**

**else**

Identifiers[i] **:=** **False ;**

**end ;**

**end ;**

**end ;**

Once this table is filled, we can use it to quickly detect which characters are considered part of an identifier, again using a simple while:

**procedure** TSynMiColor.ProcIdent **;**

**//** Processes an identifier or keyword

**var** tam **:** integer **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** Inc **(** posFin **);**

tam **:=** EndPos **-** StartPos **;**

**if** strlcomp **(** linAct **+** posIni, 'EN' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'DE' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'LA' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'LOS' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

The next point to optimize is multiple comparisons. Logically in this example, there are only 4 comparisons, but normally we can be working with more than 100. In these conditions, although it may not seem like it, valuable time may be wasted in chain detection, often doing redundant verifications.

The problem comes down to optimizing the comparison of one string in a list of several. There are various methods to carry out the optimization of this task.

Most of the Lazarus syntax components use the Hash-functions method, which is a bit complex, but it basically involves assigning each keyword to be detected a numerical value, more or less. unique, allowing it to be categorized into a small number of groups (See Appendix for more detail on this method).

Although this method is fast, it is not readable and easily confusing. Additionally, simple modifications, such as adding a new keyword, require careful calculation before modifying the code.

Here we will use a method that is generally faster and much more readable and easier to modify. This algorithm is a simplified form of a prefix tree. It can be seen as a tree in which only the first level is implemented. The first character of the identifier to be searched is used as a prefix. We will call this method the First Character as Prefix algorithm.

The method is implemented by creating a first categorization of the words using the same method table created in “CreateMethodTable”, creating a function for each initial letter of the identifier. Thus the code for “CreaTableOfMethods” would have the following form:

**procedure** TSynMiColor. CreateMethodTable **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

**...**

'A' ,'a' **:** fProcTable[I] **:=** @ProcA **;**

'B' ,'b' **:** fProcTable[I] **:=** @ProcB **;**

'C' ,'c' **:** fProcTable[I] **:=** @ProcC **;**

'D' ,'d' **:** fProcTable[I] **:=** @ProcD **;**

'E' ,'e' **:** fProcTable[I] **:=** @ProcE **;**

'F' ,'f' **:** fProcTable[I] **:=** @ProcF **;**

'G' ,'g' **:** fProcTable[I] **:=** @ProcG **;**

'H' ,'h' **:** fProcTable[I] **:=** @ProcH **;**

...

'H' ,'h' **:** fProcTable[I] **:=** @ProcH **;**

**end ;**

Then, in the procedures ProcA, ProcB, ... etc., the processing of a reduced group of identifiers is carried out, significantly reducing the number of comparisons.

For example, the procedure in charge of identifying the keywords, starting with “L”, would be:

**procedure** TSynMiColor.ProcL **;**

**var** tam **:** integer **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** Inc **(** posFin **);**

tam **:=** EndPos **-** StartPos **;**

**if** strlcomp **(** linAct **+** posIni, 'LA' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

**if** strlcomp **(** linAct **+** posIni, 'LOS' , tam **)** **=** 0 **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** no attributes

**end ;**

It is noted that the number of comparisons to be made is considerably reduced. In fact, if the keywords to be compared were evenly distributed in the alphabet, the number of comparisons would be reduced by 26 times.

As it stands, this procedure will only detect uppercase reserved words [[7]](#footnote-7). To make it case insensitive, additional processing would have to be added.

We took advantage of this missing functionality to optimize comparisons, using a quick compare function that also ignores the box (upper or lower case). For this we will once again use the invaluable help of the tables. The method will consist of creating a table that assigns an ordinal to each alphabetical character, regardless of its case. We will call this table “mHashTable”, and we will take the opportunity to fill it in “CreaTableIdentif”:

**procedure** CreateTableIdentif **;**

**var**

i, j **:** Char **;**

**begin**

**for** i **:=** #0 **to** #255 **do**

**begin**

**Case** i **of**

'\_' , '0'..'9', 'a'..'z', 'A'..'Z' **:** Identifiers[i] **:=** **True ;**

**else** Identifiers[i] **:=** **False ;**

**end ;**

j **:=** UpCase **(** i **);**

**Case** i **in** [ '\_' , 'A'..'Z', 'a'..'z'] **of**

**True :** mHashTable[i] **:=** Ord **(** j **)** **-** 64

**else**

mHashTable[i] **:=** 0 **;**

**end ;**

**end ;**

**end ;**

Now with this function created, we can now create a function, for quick comparisons. We will use the one used in the Lazarus libraries:

**function** TSynMiColor.KeyComp **( const** aKey **:** **String ):** Boolean **;**

**//** Quickly compares a string to the current token, pointed to by "fToIden".

**//** Token size must be in "fStringLen"

**var**

I **:** Integer **;**

Temp **:** PChar **;**

**begin**

Temp **:=** fToIdent **;**

**if** Length **(** aKey **)** **=** fStringLen **then**

**begin**

Result **:=** **True ;**

**for** i **:=** 1 **to** fStringLen **do**

**begin**

**if** mHashTable[Temp^] **<>** mHashTable[aKey[i]] **then**

**begin**

Result **:=** **False ;**

**break ;**

**end ;**

inc **(** Temp **);**

**end ;**

**end else** Result **:=** **False ;**

**end ;**

This comparison function uses the pointer “fToIdent” and the variable “fStringLen” to evaluate the comparison. The only parameter it requires is the string to compare.

With the help of the “mHashTable” table, the comparison will be done ignoring the box.

Now we can rethink the “ProcL” procedure:

**procedure** TSynMiColor.ProcL **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'A' **)** **then** fTokenID **:=** tkKey **else**

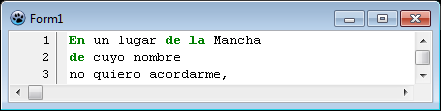
**if** KeyComp **(** 'OS' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** no attributes

**end ;**

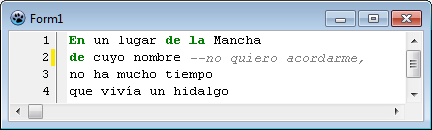
This procedure checks whether the words “LA” or “LOS” are detected. As a further optimization, the comparison of the first character is skipped, as it has been detected before calling this function.

Now we have our basic highlighter ready. A test of the program with a few more keywords will give us the following result:



### Single Line Comment Coloring

It should be done in highlighter. The procedure is simple. The sequence of characters must first be detected, which correspond to the beginning of the comment, and then locate the end of the line.



In our example we added to our “MakeMethodTables” function, the detection of comments, identifying the hyphen character “-”, since the token for single-line comments is the double hyphen “--”.

**procedure** TSynMiColor. CreateMethodTable **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

...

' **-** ' **:** fProcTable[I] **:=** @ProcMinus **;**

...

**end ;**

**end ;**

We can only detect one character in “ CreateMethodTable ”, so it is necessary to detect the next character in the “ProcMinus” function.

This function must have the following form:

**procedure** TSynMiColor.ProcMinus **;**

**//** Processes the ' **-** ' symbol

**begin**

**case** LinAct[EndPos **+** 1] **of //** see next character

' **-** ' **:** **//** is a single line comment

**begin**

fTokenID **:=** tkComment **;**

inc **(** PosFin, 2 **);** **//** jump to next token

**while not (** linAct[EndPos] **in** [#0, #10, #13] **)** **do** Inc **(** PosFin **);**

**end ;**

**else //** must be the "less" operator.

**begin**

inc **(** PosEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**end**

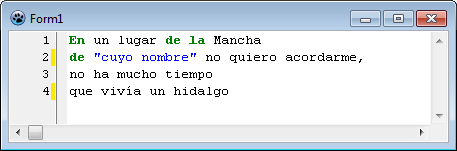
**end ;**

You need to see the next character to determine if it is the comment token. If so, the end of the line is searched to consider that entire block as a single token with the “tkComment” attribute.

If it is not the comment token, we simply move on to the next token, marking the “-” symbol as being of type “tkUnknown”. If we want to consider the “-” sign in a special category, this is the point where it should be done.

### Chain coloring

The next case of coloring corresponds to the coloring of strings. This is a simple case, because the strings will occupy at most one line. A string has a delimiter, which is used for both the start and the end of the string.



In our case, we will use the strings delimited by quotes.

First we must include its detection in the “CreateMethodTable” procedure:

**procedure** TSynMiColor. CreateMethodTable **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

...

'"' **:** fProcTable[I] **:=** @ProcString **;**

...

**end ;**

**end ;**

When the quote character is detected, control will be passed to “ProcString”, who will be in charge of finding the final delimiter of the string, and marking the entire string as a single token:

**procedure** TSynMiColor.ProcString **;**

**//** Processes the quote character.

**begin**

fTokenID **:=** tkString **;** **//** mark as string

Inc **(** PosFin **);**

**while ( not (** linAct[EndPos] **in** [#0, #10, #13] **))** **do begin**

**if** linAct[EndPos] **=** '"' **then begin //** search for end of string

Inc **(** PosFin **);**

**if (** linAct[EndPos] **<>** '"' **)** **then break ;** **//** if not double quote

**end ;**

Inc **(** PosFin **);**

**end ;**

**end ;**

Note that before determining whether the end of the string has been found, it is first verified that it is not a case of double quotes. Normally a double quote “in quotes” represents the quote character.

It can also be deduced that the string token necessarily ends on the same line where it started. It is possible to generate coloring of multi-line strings, as if they were multi-line comments, which is the case that we will see below.

### Range management

Before seeing how range coloring is implemented, it is convenient to know how ranges are handled in the SynEdit editor.

Using ranges allows you to color elements that could extend beyond a line. Such is the case of comments or multi-line strings that many languages implement.

To implement this functionality, the editor handles three methods that are defined in the “TSynCustomHighlighter” class of the “SynEditHighlighter” unit:

TSynCustomHighlighter **=** **class (** TComponent **)**

...

**function** GetRange **:** Pointer **;** **virtual ;**

**procedure** SetRange **(** Value **:** Pointer **);** **virtual ;**

**procedure** ResetRange **;** **virtual ;**

...

**end ;**

**function** TSynCustomHighlighter.GetRange **:** pointer **;**

**begin**

Result **:=** **nil ;**

**end ;**

**procedure** TSynCustomHighlighter.SetRange **(** Value **:** Pointer **);**

**begin**

**end ;**

**procedure** TSynCustomHighlighter.ResetRange **;**

**begin**

**end ;**

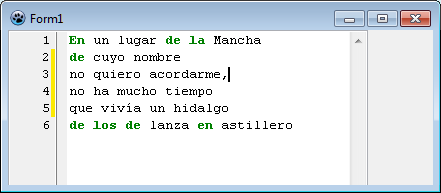
These methods have the following function:

* ResetRange.- It is executed before scanning the first line of the text, since there are no previous lines that affect the range.
* GetRange.- It is called after completing the exploration of a line, to obtain the level at the end of the line. This value is stored internally.
* SetRange.- It is called before scanning a line. Change the level using the level from the previous line.

The sequence of execution depends on the action performed. The first time a document is opened, SynEdit scans all the lines with GetRange() (reading all the tokens), to have the value that corresponds to each line. The read value will be stored internally for later comparisons.

Every time a part of the document is modified, SynEdit makes several calls to GetRange() and SetRange(), to reconstruct the new state of the document. The scan can go from the current line to the end of the document, if SynEdit deems it necessary. However, it is common for the document to only be scanned for a few lines.

The following example shows an editor, and the calls to the ranges and “SetLine” methods, when line 3 is modified:



1. SetRange
2. SetLine: I don't want to remember,
3. GetRange
4. SetLine: not long ago
5. GetRange
6. SetLine: who lived a gentleman
7. GetRange
8. SetRange
9. SetLine: whose name
10. SetRange
11. SetLine: I don't want to remember,
12. SetRange
13. SetLine: not long ago
14. SetRange
15. SetLine: who lived a gentleman

The editor usually scans the text from a line before the modified line, until it finds that a line returns the same level as it had previously.

If no syntax processing is to be performed, it is not necessary to override these methods. They should only be modified when range coloring or code folding is going to be implemented.

The value that “SetRange” sends, in the “Value” parameter, is a pointer, as well as the value that “GetRange” expects to receive, because they have been designed to work with objects. But it is not necessary to work with pointers. In practice, an enumerated type is often used to identify the levels of the ranges, taking care to make the necessary conversions.

**Type**

...

TRangeState **=** **(** rsUnknown, rsComment **);**

...

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

...

fRange **:** TRangeState **;**

...

**end ;**

...

**function** TSynMiColor.GetRange **:** Pointer **;**

**begin**

Result **:=** Pointer **(** PtrInt **(** fRange **));**

**end ;**

**procedure** TSynMiColor.SetRange **(** Value **:** Pointer **);**

**begin**

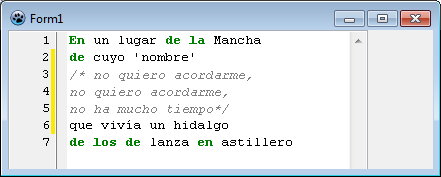
fRange **:=** TRangeState **(** PtrUInt **(** Value **));**

**end ;**

The PtrInt and PtrUInt functions convert a pointer to an integer of the same size as the pointer.

The value of the pointers is not important in itself [[8]](#footnote-8), because the objects pointed to by them are not accessed under normal conditions. The important thing is that they take different values when a particular range is found (comments, blocks, etc.), than when there are no active ranges.

In the following example, comment coloring (/\* … \*/) has been implemented, and the values of the parameters passed and read are displayed, when a comment is inserted starting from line 3:



1. SetRange: 0
2. SetLine: /\*I don't want to remember,
3. GetRange: 1
4. SetLine: I don't want to remember,
5. GetRange: 1
6. SetLine: not long ago\*/
7. GetRange: 0
8. SetRange: 0
9. SetLine: whose 'name'
10. SetRange: 0
11. SetLine: /\*I don't want to remember,
12. SetRange: 1
13. SetLine: I don't want to remember,
14. SetRange: 1
15. SetLine: not long ago\*/

Next to “SetRange” or “GetRange”, the ordinal of “fRange” is displayed, as a visual aid to see how it is changing. We clarify that the change in “fRange” does not have to be consecutive, it is enough for “fRange” to take different values for the coloring functionality to work.

The concept of ranges can be a little difficult to assimilate at first. It may help to think of them as a way to store additional information that corresponds to each line. Therefore there is a range value (pointer to an object), for each line of text scanned.

The range is not associated with the concept of “level” [[9]](#footnote-9), or “scope”, it is simply: Additional information associated with each line that can be used to save the state of a line, at the end of it. Of course, if you do not want to use this information, just leave it with the default values.

There may be a lot of additional information that you want to associate with each line. Ranges are a useful way to store this information. The editor reads the range of each line at the beginning (through the highlighter), when it finishes scanning the line and saves this information.

Then when the text is modified, the editor makes successive scans to “update” the ranges in the lines that may be affected.

As a rule, the range of previous lines is not affected by the change in any line. But successive lines can be altered in their range. Therefore, if this is the case, the editor will explore the following lines until it finds that the range that corresponds to it is similar to the one it had, and the update stops.

If the information we want to associate does not refer to a line, but rather to smaller elements, ranges will not help directly [[10]](#footnote-10). They are most useful when the information to be stored can be obtained directly or easily from the state of the previous line.

When managing ranges, it must be fulfilled that:

“Knowing the final state of the previous line (range or object pointed to by the range) is all that is needed to work correctly with the current line”

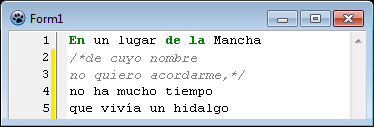
Another way of seeing the usefulness of ranges is to think of them as an aid to transfer information from one line to another, considering that the editor does not explore the entire text in an orderly manner, but instead tries to make the least amount of explorations, performing [[11]](#footnote-11)several small explorations in various parts of the text, according to the modified text.

This means that if we alter, for example, the variable XXX, while scanning line “n”, we cannot be sure that the value of XXX will be read when scanning line “n+1”, because not necessarily the next line to explore will be “n+1”. If we want the desired value of the XXX variable to be seen when scanning the “n+1” line, we must see how to save it and recover it as part of the range.

The next section will help in understanding ranges by seeing an actual implementation in syntax coloring.

### Range or context coloring

Context coloring involves considering an interval of text, which can be on one or several lines, as if it were a single token. Logically, due to the hierarchy used, a token cannot be greater than one line, therefore, if the range extends to more than one line, it will be identified as several tokens (one for each line) of the same category.



Given our class, this coloring can be done using a class derived from the highlighter used, or the functionality can be included in the original syntax code itself.

If you want to create a derived class, it must have a structure similar to this:

TSynDemoHlContextFoldBase **=** **class (** TSynMiColor **)**

**protected**

FCurRange **:** Integer **;**

**public**

**procedure** Next **;** **override ;**

**function** GetTokenAttribute **:** TSynHighlighterAttributes **;** **override ;**

**public**

In the “Next” and “GetTokenAttribute” methods, the additional behavior needed to color the text ranges should be added.

For more information, it is recommended to see the example that comes with Lazarus: in \examples\SynEdit\NewHighlighterTutorial\

However, the most efficient way would be to include this functionality in the highlighter itself.

Now let's consider the case of coloring a range of one or several lines. For this example let's consider coloring multi-line comments.

First we choose the comment delimiter characters. For our example we will use the typical C characters: “/\*” and “\*/”. All text between these characters will be considered a comment and will have the “fStringAttri” attribute.

We must detect the beginning delimiter by the “/” character, but doing the validation, since it could be the division operator.

Again we do the interception in the “ CreateMethodTable ” function:

**procedure** TSynMiColor. CreateMethodTable **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

...

'/' **:** fProcTable[I] **:=** @ProcSlash **;**

...

**end ;**

**end ;**

And we identify comments in “ProcSlash”:

**procedure** TSynMiColor.ProcSlash **;**

**//** Processes the ' **/** ' symbol

**begin**

**case** linAct[EndPos **+** 1] **of**

' **\*** ' **:** **// multi -** line comment

**begin**

fRange **:=** rsComment **;** **//** mark range

inc **(** PosFin, 2 **);**

CommentProc; //Process in comment mode

**end ;**

**else //** must be the "between" operator.

**begin**

inc **(** PosEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**end**

**end ;**

We note that we are working with a procedure “CommentProc” and with a new flag, called “fRange”. Which should be declared as shown:

**Type**

...

TRangeState **=** **(** rsUnknown, rsComment **);**

...

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

...

fRange **:** TRangeState **;**

...

This statement is important for managing ranges. The detection of coloring in ranges requires this type of management.

Various types of ranges can be created in “TRangeState”, according to the needs of the syntax. The order of those listed is not important.

Additionally, to implement the ranges functionality, you must override the three ranges methods:

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

...

**function** GetRange **:** Pointer **;** **override ;**

**procedure** SetRange **(** Value **:** Pointer **);** **override ;**

**procedure** ResetRange **;** **override ;**

...

**end ;**

...

{Implementation of range functionalities}

**procedure** TSynMiColor.ReSetRange **;**

**begin**

fRange **:=** rsUnknown **;**

**end ;**

**function** TSynMiColor.GetRange **:** Pointer **;**

**begin**

Result **:=** Pointer **(** PtrInt **(** fRange **));**

**end ;**

**procedure** TSynMiColor.SetRange **(** Value **:** Pointer **);**

**begin**

fRange **:=** TRangeState **(** PtrUInt **(** Value **));**

**end ;**

Once the behavior of these methods is defined. The editor will be in charge of managing your calls, saving us the work of controlling the status of the lines.

But it is still necessary to process the lines that are in the comments range. To do this, we implement the “CommentProc” method:

**procedure** TSynMiColor.CommentProc **;**

**begin**

fTokenID **:=** tkComment **;**

**case** linAct[PosFin] **of**

#0 **:**

**begin**

ProcNull **;**

**exit ;**

**end ;**

**end ;**

**while** linAct[EndPos] **<>** #0 **do**

**case** linAct[PosFin] **of**

' **\*** ' **:**

**if** linAct[EndPos **+** 1] **=** ' **/** ' **then**

**begin**

inc **(** PosFin, 2 **);**

fRange **:=** rsUnknown **;**

**break ;**

**end**

**else** inc **(** PosEnd **);**

#10 **:** **break ;**

#13 **:** **break ;**

**else** inc **(** PosEnd **);**

**end ;**

**end ;**

This method scans the lines for the final delimiter of the comment. If it is not found, it considers everything scanned (including the entire line) as a single token of type “tkComment”. Note that the end-of-comment delimiter “\*/” is not being detected anywhere else in the class.

Which should be called when it is detected that we are in the middle of a comment, as is done in “ProcSlash”, but we must also include it in “Next”:

**procedure** TSynMiColor.Next **;**

**begin**

posIni **:=** posEnd **;** **//** points to the first element

**if** fRange **=** rsComment **then**

CommentProc

**else**

**begin**

fRange **:=** rsUnknown **;**

fProcTable[linAct[PosEnd]] **;** **//** The corresponding function is executed.

**end ;**

**end ;**

In this way, we pass control to “CommentProc” when we are in the middle of a context comment.

## The TSynCustomHighlighter class

So far, we have shown how to implement a simple highlighter, using only the necessary properties and methods of TSynCustomHighlighter. Now we are going to talk a little about the class itself and some additional functionalities that it brings.

A quick look at the class code will indicate that it is a more or less extensive class, considering it is an abstract class. But, despite everything, the TSynCustomHighlighter class does not store information itself. All the information it handles is saved in the editor, specifically in Lines[].

Any editor that wants to implement syntax highlighting must be associated with a highlighter. The relationships between an editor and a highlighter are:

* An editor can be associated with one and only one highlighter.
* A highlighter can serve one or more editors.

This relationship can be deduced, taking into consideration that the highlighter itself does not store information about the text it explores.

A consequence of the above relationships is that a highlighter is not fixedly associated with a particular editor.

But under normal conditions, when an editor uses a single highlighter, the simplified relationship is one to one.

A standout feature of TSynCustomHighlighter is the way it explores lines using the concept of range.

The range information, which is generated by the highlighter, is saved in the editor and used to access each line, with the appropriate initial state.

### CurrentLines[] and CurrentRanges[]

One of the most useful properties of TSynCustomHighlighter is CurrentLines[]. This arrangement allows access to the “buffer” of the current editor, that is, the one that is using the highlighter at that moment.

The CurrentLines[] array is assigned to Lines[] before the editor uses the highlighter, so CurrentLines[] will always be guaranteed to reference the current editor.

In the same way as CurrentLines[], allows us to access the information of the current lines, the CurrentRanges[] property [[12]](#footnote-12)allows us to access the range values, which corresponds to each line of the text.

CurrentRanges[], is a complex object, but for practical purposes, we can see it as a simple table of pointers. These values are the ones assigned to each line when TSynCustomHighlighter requests it from the highlighter using GetRange().

It is necessary to know that CurrentLines[] and CurrentRanges[], are like tables that start at zero. If for example we wanted to see the range value of line 3, we would have to look at CurrentRanges[2].

Logically CurrentLines[]and CurrentRanges[] have the same size at all times, and this size matches the number of lines that the current editor has (when the highlighter is assigned to one).

Attempting to access CurrentLines[] or CurrentRanges[], when the highlighter is not assigned to an editor, will result in a runtime error.

The CurrentLines[] table returns strings, but CurrentRanges[] returns pointers. Returns the same pointers that are assigned when using the methods:

* TSynCustomHighlighter.GetRange()
* TSynCustomHighlighter.SetRange()

That is, CurrentRanges[] allows us to recover the Range information stored in each line.

The following diagram will clarify what has been said:



Of course, the meaning of this pointer is the same as that used in the highlighter (enumerated, integer, object reference, etc.). It will be up to the programmer to apply the type conversion corresponding to each case.

### Some Methods and Properties

The TSynCustomHighlighter class additionally has various useful properties that may be useful to us at some point.

The ScanAllRanges() method causes the editor to scan all the lines again to update the references to the range that each line has. This method is useful, for example, when changing the syntax of the working language.

The “LanguageName” property returns a string with text corresponding to the name of the language for which the highlighter is prepared. It is the highlighter's responsibility to activate this property, overriding the GetLanguageName() method.

The “SampleSource” property returns a string with a piece of example code, in the language used, as a sample to test the highlighter. Again, it is the highlighter's responsibility to activate this property, overriding the GetLanguageName() method.

There is an internal property that can be used for working with identifiers: IdentChars. Internally, it is a simple reference to GetIdentChars(), which has the following definition:

**function** TSynCustomHighlighter.GetIdentChars **:** TSynIdentChars **;**

**begin**

Result **:=** [#33..#255] **;**

**end ;**

It may be necessary to override this method to suit our needs.

The StartAtLineIndex() method allows you to position the highlighter on any line of the text, to start exploring that line. This is useful when we want to perform an additional scan (outside of the normal highlighter scanning process), to obtain some additional information from the highlighter.

An example of using this function can be seen in the method: TSynEdit. GetHighlighterAttriAtRowCol(), from the editor, which allows obtaining the attribute and token for any position in the text.

### Attributes

Much of the code in TSynCustomHighlighter is related to attribute management. These allow you to configure how the tokens will be seen in the editor.

Attributes are objects that should be created before using the highlighter. To create a new attribute, execute:

fKeyAttri **:=** TSynHighlighterAttributes.Create **(** SYNS\_AttrKey, SYNS\_XML\_AttrKey **);**

fKeyAttri.Style **:=** [fsBold] **;**

AddAttribute **(** fKeyAttri **);** **//** save the reference to the attribute

Typically, this code is always placed in the highlighter's constructor. But they could be created dynamically anywhere in the process.

References to attributes are stored in an internal structure of the class (fAttributes). As an additional facility, they are released when destroying the class, so it is not necessary (nor should you) destroy the attributes added to the highlighter.

All highlighters have several attributes defined (statically or dynamically), because they are necessary to assign the highlighting properties of the text.

Attributes are objects of the TSynHighlighterAttributes class, defined in the same SynEditHighlighter unit, that contains the TSynCustomHighlighter class.

The TSynCustomHighlighter class does not have attributes defined internally, as is the way it works, but it has defined properties that allow access to some of the most common attributes:

TSynCustomHighlighter **=** **class (** TComponent **)**

...

**public**

**property** AttrCount **:** integer **read** GetAttribCount **;**

**property** Attribute[idx **:** integer] **:** TSynHighlighterAttributes

**read** GetAttribute **;**

**property** Capabilities **:** TSynHighlighterCapabilities

**read** {$IFDEF SYN\_LAZARUS}FCapabilities{$ **ELSE** }GetCapabilities{$ENDIF} **;**

**property** SampleSource **:** **string read** GetSampleSource write SetSampleSource **;**

**property** CommentAttribute **:** TSynHighlighterAttributes

**index** SYN\_ATTR\_COMMENT **read** GetDefaultAttribute **;**

**property** IdentifierAttribute **:** TSynHighlighterAttributes

**index** SYN\_ATTR\_IDENTIFIER **read** GetDefaultAttribute **;**

**property** KeywordAttribute **:** TSynHighlighterAttributes

**index** SYN\_ATTR\_KEYWORD **read** GetDefaultAttribute **;**

**property** StringAttribute **:** TSynHighlighterAttributes

**index** SYN\_ATTR\_ **STRING read** GetDefaultAttribute **;**

**property** SymbolAttribute **:** TSynHighlighterAttributes **//** mh 2001 **-** 09 **-** 13

**index** SYN\_ATTR\_SYMBOL **read** GetDefaultAttribute **;**

**property** WhitespaceAttribute **:** TSynHighlighterAttributes

**index** SYN\_ATTR\_WHITESPACE **read** GetDefaultAttribute **;**

This access depends on correctly implementing the GetDefaultAttribute() method. For ease of implementation, there are some predefined constants in the “SynEditHighlighter” unit:

**const**

SYN\_ATTR\_COMMENT **=** 0 **;**

SYN\_ATTR\_IDENTIFIER **=** 1 **;**

SYN\_ATTR\_KEYWORD **=** 2 **;**

SYN\_ATTR\_STRING **=** 3 **;**

SYN\_ATTR\_WHITESPACE **=** 4 **;**

SYN\_ATTR\_SYMBOL **=** 5 **;**

These constants do not cover the number of attributes that a fairly complete highlighter can handle, but they are a simple help to access the most common attributes.

A typical implementation of GetDefaultAttribute(), in a highlighter is:

**function** TSynLFMSyn.GetDefaultAttribute **( Index :** integer **):** TSynHighlighterAttributes **;**

**begin**

**case Index of**

SYN\_ATTR\_COMMENT **:** Result **:=** fCommentAttri **;**

SYN\_ATTR\_IDENTIFIER **:** Result **:=** fIdentifierAttri **;**

SYN\_ATTR\_KEYWORD **:** Result **:=** fKeyAttri **;**

SYN\_ATTR\_STRING **:** Result **:=** fStringAttri **;**

SYN\_ATTR\_WHITESPACE **:** Result **:=** fSpaceAttri **;**

SYN\_ATTR\_SYMBOL **:** Result **:=** fSymbolAttri **;**

**else**

Result **:=** **nil ;**

**end ;**

**end ;**

The idea is to return the appropriate reference to the highlighter, when requested via GetDefaultAttribute(). The attributes are defined in the highlighter.

From what we have seen about the attributes, it can be deduced that there are various ways to access the attributes of a highlighter.

The easiest way to access attributes is through GetDefaultAttribute():

**var** attribute **:** TSynHighlighterAttributes **;**

...

attribute **:=** SynLFMSyn1.GetDefaultAttribute **(** SYN\_ATTR\_KEYWORD **);**

attribute.Foreground **:=** clRed **;**

However, this way will only work for attributes that have been correctly entered in the GetDefaultAttribute() method, which depends on the correct implementation of the highlighter. Additionally, only some of the most common attributes are visible.

Another way to access attributes is to use the public properties that many highlighters implement. These are:

TSynLFMSyn **=** **class (** TSynCustomFoldHighlighter **)**

...

**private**

fCommentAttri **:** TSynHighlighterAttributes **;**

fIdentifierAttri **:** TSynHighlighterAttributes **;**

fKeyAttri **:** TSynHighlighterAttributes **;**

fNumberAttri **:** TSynHighlighterAttributes **;**

fSpaceAttri **:** TSynHighlighterAttributes **;**

fStringAttri **:** TSynHighlighterAttributes **;**

fSymbolAttri **:** TSynHighlighterAttributes **;**

...

**published**

**property** CommentAttri **:** TSynHighlighterAttributes **read** fCommentAttri

write fCommentAttri **;**

**property** IdentifierAttri **:** TSynHighlighterAttributes **read** fIdentifierAttri

write fIdentifierAttri **;**

**property** KeyAttri **:** TSynHighlighterAttributes **read** fKeyAttri write fKeyAttri **;**

**property** NumberAttri **:** TSynHighlighterAttributes **read** fNumberAttri

write fNumberAttri **;**

**property** SpaceAttri **:** TSynHighlighterAttributes **read** fSpaceAttri

write fSpaceAttri **;**

**property** StringAttri **:** TSynHighlighterAttributes **read** fStringAttri

write fStringAttri **;**

**end ;**

The attributes themselves are hidden, but their respective properties are published, so they could be changed from the Object Inspector.

Depending on the implementation of the highlighter, these properties could allow us greater visibility to access the attributes of the highlighter.

So one way to access the attributes, using these properties would be:

**var** attribute **:** TSynHighlighterAttributes **;**

**...**

attribute **:=** SynLFMSyn1.KeywordAttribute **;**

attribute.Foreground **:=** clRed **;**

Another way to access all of the highlighter's attributes would be to use the Attribute[] attribute table, which is a simple reference to the protected GetAttribute() method:

**function** TSynCustomHighlighter.GetAttribute **(** idx **:** integer **):**

TSynHighlighterAttributes **;**

**begin**

Result **:=** **nil ;**

**if (** idx **>=** 0 **)** **and (** idx **<** fAttributes.Count **)** **then**

Result **:=** TSynHighlighterAttributes **(** fAttributes.Objects[idx] **);**

**end ;**

Attributes are stored internally in fAttributes, a TStringList, in the object field.

A typical access to an attribute, through the highlighter would be:

**var** attribute **:** TSynHighlighterAttributes **;**

**...**

attribute **:=** SynLFMSyn1.Attribute[2] **;**

attribute.Foreground **:=** clRed **;**

The disadvantage is that we need to know the index of the attribute with which we are going to work. As a help we could use the name of the attribute.

Thus, a typical iteration of attributes would be:

**var** i **:** integer **;**

**begin**

...

**for** i **:=** 0 **to** SynLFMSyn1.AttrCount **-** 1 **do**

ShowMessage **(** SynLFMSyn1.Attribute[i]. **Name );**

...

The name stored in the “Name” property is the name that is assigned to the attribute when it is created using its constructor:

Attribute **:=** TSynHighlighterAttributes.Create **(** 'Name' **);**

At the same time, we can also use the second parameter of the constructor (aStoredName):

**constructor** TSynHighlighterAttributes.Create **(** aCaption **:** **string ;** aStoredName **:** **String =** '' **);**

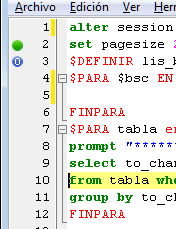
Which allows you to assign another internal name to the attribute. We can then access this name through the “StoredName” property of the attribute.

To know the number of attributes, we have the “AttrCount” property.

Iterating through Attribute[], allows you to effectively access all attributes that have been created in the highlighter.

## Code Folding Functionality

The code folding feature refers to the ability to display the length of a predefined text block in the left bar of the editor:



Marcas de plegado

Most of the syntax components, which come in Lazarus, do not include folding functionality, so it will have to be created in code. The code for folding must be in the same highlighter.

### Basic Folding

Let's consider adding code folding, starting with a simple highlighter, like the one we described above.

Firstly, to implement folding, you need to derive the highlighter class from the “TSynCustomFoldHighlighter” class (defined in the “SynEditHighlighterFoldBase” unit), instead of using the “TSynCustomHighlighter” class.

This new class contains the processing code for folding.

First we change the declaration of the highlighter class that we use:

TSynMiColor **=** **class (** TSynCustomHighlighter **)**

...

By:

TSynDemoHlFold **=** **class (** TSynCustomFoldHighlighter **)**

...

But we must also modify the range treatment methods:

{Implementation of range functionalities}

**procedure** TSynMiColor.ReSetRange **;**

**begin**

**inherited ;**

fRange **:=** rsUnknown **;**

**end ;**

**function** TSynMiColor.GetRange **:** Pointer **;**

**begin**

CodeFoldRange.RangeType **:=** Pointer **(** PtrInt **(** fRange **));**

Result **:=** **inherited ;**

**end ;**

**procedure** TSynMiColor.SetRange **(** Value **:** Pointer **);**

**begin**

**inherited ;**

fRange **:=** TRangeState **(** PtrUInt **(** CodeFoldRange.RangeType **));**

**end ;**

To respond appropriately to the requirements of the class. “inherited” takes care of that, and we move the context information to “CodeFoldRange”, because that is where “TSynCustomFoldHighlighter” now works.

Once these modifications have been implemented, we are ready to add the “folding”, to do this we must detect the beginning and end of the block.

The easiest way is to call the methods: “StartCodeFoldBlock” and “EndCodeFoldBlock”, when the start and end tokens are detected respectively:

For example, if we added the beginning with the detection of the word BEGIN and the end of the block with the word END, the code would be:

**procedure** TSynMiColor.ProcB **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'EGIN' **) then begin**

fTokenID **:=** tkKey **;** StartCodeFoldBlock **( nil );** **end**

**else**

**if** KeyComp **(** 'Y' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

**...**

**procedure** TSynMiColor.ProcE **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'N' **)** **then** fTokenID **:=** tkKey **else**

**if** KeyComp **(** 'ND' **)** **then begin**

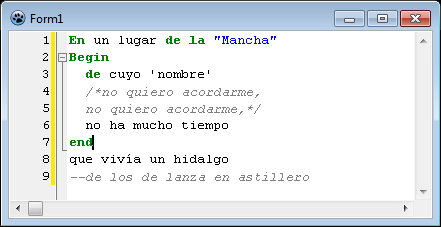
fTokenID **:=** tkKey **;** EndCodeFoldBlock **();** **end**

**else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

A test text might look like this:



### Example code

A complete example code with syntax coloring of identifiers, strings, single-line and multi-line comments, and folding is shown below:

{Minimal functional unit that demonstrates the structure of a simple class

which implements full syntax coloring and code folding.

By Tito Hinostroza: 08/07/2013

}

**unit** uSyntax **;** {$mode objfpc}{$H **+** }

**interface**

**uses**

Classes, SysUtils, Graphics, SynEditHighlighter, SynEditHighlighterFoldBase **;**

**type**

{Class for creating a highlighter}

TRangeState **=** **(** rsUnknown, rsComment **);**

**//** ID to categorize tokens

TtkTokenKind **=** **(** tkComment, tkKey, tkNull, tkSpace, tkString, tkUnknown **);**

TProcTableProc **=** **procedure of object ;** **//** Procedure type to process the

**//** token for the initial character.

{TSynMiColor}

TSynMiColor **=** **class (** TSynCustomFoldHighlighter **)**

**protected**

posIni, posEnd **:** Integer **;**

fStringLen **:** Integer **;** **//** Size of current token

fToIdent **:** PChar **;** **//** Pointer to identifier

linAct **:** PChar **;**

fProcTable **:** **array** [#0..#255] **of** TProcTableProc **;** **//** procedure table

fTokenID **:** TtkTokenKind **;** **//** Id of the current token

fRange **:** TRangeState **;**

**//** define the categories of the "tokens"

fAtriComment **:** TSynHighlighterAttributes **;**

fAtriKey **:** TSynHighlighterAttributes **;**

fAtriEspac **:** TSynHighlighterAttributes **;**

fAtriString **:** TSynHighlighterAttributes **;**

**public**

**procedure** SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);** **override ;**

**procedure** Next **;** **override ;**

**function** GetEol **:** Boolean **;** **override ;**

**procedure** GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

**override ;**

**function** GetTokenAttribute **:** TSynHighlighterAttributes **;** **override ;**

**public**

GetToken **function :** **String ;** **override ;**

**function** GetTokenPos **:** Integer **;** **override ;**

**function** GetTokenKind **:** integer **;** **override ;**

**constructor** Create **(** AOwner **:** TComponent **);** **override ;**

**private**

**procedure** CommentProc **;**

**procedure** CreateMethodTable **;**

**function** KeyComp **( const** aKey **:** **String ):** Boolean **;**

**procedure** ProcMinus **;**

**//** Identifier processing functions

**procedure** ProcNull **;**

**procedure** ProcSlash **;**

**procedure** ProcSpace **;**

**procedure** ProcString **;**

**procedure** ProcUnknown **;**

**procedure** ProcB **;**

**procedure** ProcC **;**

**procedure** ProcD **;**

**procedure** ProcE **;**

**procedure** ProcL **;**

**function** GetRange **:** Pointer **;** **override ;**

**procedure** SetRange **(** Value **:** Pointer **);** **override ;**

**procedure** ResetRange **;** **override ;**

**end ;**

**implementation**

**var**

Identifiers **:** **array** [#0..#255] **of** ByteBool **;**

mHashTable **:** **array** [#0..#255] **of** Integer **;**

**procedure** CreateTableIdentif **;**

**var** i, j **:** Char **;**

**begin**

**for** i **:=** #0 **to** #255 **do**

**begin**

**Case** i **of**

'\_' , '0'..'9', 'a'..'z', 'A'..'Z' **:** Identifiers[i] **:=** **True ;**

**else** Identifiers[i] **:=** **False ;**

**end ;**

j **:=** UpCase **(** i **);**

**Case** i **in** [ '\_' , 'A'..'Z', 'a'..'z'] **of**

**True :** mHashTable[i] **:=** Ord **(** j **)** **-** 64

**else**

mHashTable[i] **:=** 0 **;**

**end ;**

**end ;**

**end ;**

**constructor** TSynMiColor.Create **(** AOwner **:** TComponent **);**

**//Constructor** of the class. Here you must create the attributes to use.

**begin**

**inherited** Create **(** AOwner **);**

**//** comments attribute

fAtriComment **:=** TSynHighlighterAttributes.Create **(** 'Comment' **);**

fAtriComment.Style **:=** [fsItalic] **;** **//** in italics

fAtriComment.Foreground **:=** clGray **;** **//** gray font color

AddAttribute **(** fAtriComment **);**

**//** keyword attribute

fAtriClave **:=** TSynHighlighterAttributes.Create **(** 'Key' **);**

fAtriClave.Style **:=** [fsBold] **;** **//** in bold

fAtriClave.Foreground **:=** clGreen **;** **//** green font color

AddAttribute **(** fAtriKey **);**

**//** spaces attribute. No attributes

fAtriEspac **:=** TSynHighlighterAttributes.Create **(** 'space' **);**

AddAttribute **(** fAtriEspac **);**

**//** strings attribute

fAtriString **:=** TSynHighlighterAttributes.Create **(** 'String' **);**

fAtriString.Foreground **:=** clBlue **;** **//** blue font color

AddAttribute **(** fAtriString **);**

CreateMethodTable **;** **//** Build method table

**end ;**

**procedure** TSynMyColor.CreateMethodTable **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

' **-** ' **:** fProcTable[I] **:=** @ProcMinus **;**

'"' **:** fProcTable[I] **:=** @ProcString **;**

' **/** ' **:** fProcTable[I] **:=** @ProcSlash **;**

'B' ,'b' **:** fProcTable[I] **:=** @ProcB **;**

'C' ,'c' **:** fProcTable[I] **:=** @ProcC **;**

'D' ,'d' **:** fProcTable[I] **:=** @ProcD **;**

'E' ,'e' **:** fProcTable[I] **:=** @ProcE **;**

'L' ,'l' **:** fProcTable[I] **:=** @ProcL **;**

#0 **:** fProcTable[I] **:=** @ProcNull **;** **//** Read the end of string mark character

#1..#9, #11, #12, #14..#32 **:** fProcTable[I] **:=** @ProcSpace **;**

**else** fProcTable[I] **:=** @ProcUnknown **;**

**end ;**

**end ;**

**function** TSynMiColor.KeyComp **( const** aKey **:** **String ):** Boolean **;**

**var**

I **:** Integer **;**

Temp **:** PChar **;**

**begin**

Temp **:=** fToIdent **;**

**if** Length **(** aKey **)** **=** fStringLen **then**

**begin**

Result **:=** **True ;**

**for** i **:=** 1 **to** fStringLen **do**

**begin**

**if** mHashTable[Temp^] **<>** mHashTable[aKey[i]] **then**

**begin**

Result **:=** **False ;**

**break ;**

**end ;**

inc **(** Temp **);**

**end ;**

**end else** Result **:=** **False ;**

**end ;**

**procedure** TSynMiColor.ProcMinus **;**

**//** Processes the ' **-** ' symbol

**begin**

**case** LinAct[EndPos **+** 1] **of //** see next character

' **-** ' **:** **//** is a single line comment

**begin**

fTokenID **:=** tkComment **;**

inc **(** PosFin, 2 **);** **//** jump to next token

**while not (** linAct[EndPos] **in** [#0, #10, #13] **)** **do** Inc **(** PosFin **);**

**end ;**

**else //** must be the "less" operator.

**begin**

inc **(** PosEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**end**

**end ;**

**procedure** TSynMiColor.ProcString **;**

**//** Processes the quote character.

**begin**

fTokenID **:=** tkString **;** **//** mark as string

Inc **(** PosFin **);**

**while ( not (** linAct[EndPos] **in** [#0, #10, #13] **))** **do begin**

**if** linAct[EndPos] **=** '"' **then begin //** search for end of string

Inc **(** PosFin **);**

**if (** linAct[EndPos] **<>** '"' **)** **then break ;** **//** if not double quote

**end ;**

Inc **(** PosFin **);**

**end ;**

**end ;**

**procedure** TSynMiColor.ProcSlash **;**

**//** Processes the ' **/** ' symbol

**begin**

**case** linAct[EndPos **+** 1] **of**

' **\*** ' **:** **// multi -** line comment

**begin**

fRange **:=** rsComment **;** **//** mark range

fTokenID **:=** tkComment **;**

inc **(** PosFin, 2 **);**

**while** linAct[EndPos] **<>** #0 **do**

**case** linAct[PosFin] **of**

' **\*** ' **:**

**if** linAct[EndPos **+** 1] **=** ' **/** ' **then**

**begin**

inc **(** PosFin, 2 **);**

fRange **:=** rsUnknown **;**

**break ;**

**end else** inc **(** EndPos **);**

#10 **:** **break ;**

#13 **:** **break ;**

**else**

inc **(** PosEnd **);**

**end ;**

**end ;**

**else //** must be the "between" operator.

**begin**

inc **(** PosEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**end**

**end ;**

**procedure** TSynMiColor.ProcB **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'EGIN' **)** **then begin** fTokenID **:=** tkKey **;** StartCodeFoldBlock **( nil );** **end else**

**if** KeyComp **(** 'Y' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

**procedure** TSynMiColor.ProcC **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

**procedure** TSynMiColor.ProcD **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'E' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

**procedure** TSynMiColor.ProcE **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'N' **)** **then** fTokenID **:=** tkKey **else**

**if** KeyComp **(** 'ND' **)** **then begin** fTokenID **:=** tkKey **;** EndCodeFoldBlock **();** **end else**

fTokenID **:=** tkUnknown **;** **//** common identifier

**end ;**

**procedure** TSynMiColor.ProcL **;**

**begin**

**while** Identifiers[linAct[posFin]] **do** inc **(** posFin **);**

fStringLen **:=** EndPos **-** StartPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** linAct **+** posIni **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'A' **)** **then** fTokenID **:=** tkKey **else**

**if** KeyComp **(** 'OS' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkUnknown **;** **//** no attributes

**end ;**

**procedure** TSynMiColor.ProcNull **;**

**//** Processes the occurrence of character #0

**begin**

fTokenID **:=** tkNull **;** **//** You only need this to indicate that the end of the line has been reached

**end ;**

**procedure** TSynMiColor.ProcSpace **;**

**//** Processes character that is start of space

**begin**

fTokenID **:=** tkSpace **;**

**repeat**

Inc **(** posEnd **);**

**until (** linAct[posEnd] **>** #32 **)** **or (** linAct[posEnd] **in** [#0, #10, #13] **);**

**end ;**

**procedure** TSynMiColor.ProcUnknown **;**

**begin**

inc **(** posEnd **);**

**while (** linAct[posEnd] **in** [#128..#191] **)** **OR //** continued utf8 subcode

**((** linAct[posEnd] **<>** #0 **)** **and (** fProcTable[linAct[posFin]] **=** @ProcUnknown **))** **do** inc **(** posEnd **);**

fTokenID **:=** tkUnknown **;**

**end ;**

**procedure** TSynMiColor.SetLine **( const** NewValue **:** **String ;** LineNumber **:** Integer **);**

**begin**

**inherited ;**

linAct **:=** PChar **(** NewValue **);** **//** copy the current line

endPos **:=** 0 **;** **//** points to the first character

Next **;**

**end ;**

**procedure** TSynMiColor.Next **;**

**begin**

posIni **:=** posEnd **;** **//** points to the first element

**if** fRange **=** rsComment **then**

CommentProc

**else**

**begin**

fRange **:=** rsUnknown **;**

fProcTable[linAct[PosEnd]] **;** **//** The corresponding function is executed.

**end ;**

**end ;**

**function** TSynMiColor.GetEol **:** Boolean **;**

{Indicates when the end of the line has been reached}

**begin**

Result **:=** fTokenId **=** tkNull **;**

**end ;**

**procedure** TSynMiColor.GetTokenEx **(** out TokenStart **:** PChar **;** out TokenLength **:** integer **);**

{Returns information about the current token}

**begin**

TokenLength **:=** posEnd **-** posIni **;**

TokenStart **:=** linAct **+** posIni **;**

**end ;**

**function** TSynMiColor.GetTokenAttribute **:** TSynHighlighterAttributes **;**

**//** Returns information about the current token

**begin**

**case** fTokenID **of**

tkComment **:** Result **:=** fAtriComment **;**

tkKey **:** Result **:=** fAtriClave **;**

tkSpace **:** Result **:=** fAtriSpace **;**

tkString **:** Result **:=** fAtString **;**

**else** Result **:=** **nil ;** **//** tkUnknown, tkNull

**end ;**

**end ;**

{The following functions are used by SynEdit to manage the

braces, brackets, parentheses and quotes. They are not crucial for coloring

of tokens, but they must respond well.}

**function** TSynMiColor.GetToken **:** **String ;**

**begin**

Result **:=** '' **;**

**end ;**

**function** TSynMiColor.GetTokenPos **:** Integer **;**

**begin**

Result **:=** posIni **-** 1 **;**

**end ;**

**function** TSynMiColor.GetTokenKind **:** integer **;**

**begin**

Result **:=** 0 **;**

**end ;**

**procedure** TSynMiColor.CommentProc **;**

**begin**

fTokenID **:=** tkComment **;**

**case** linAct[PosFin] **of**

#0 **:**

**begin**

ProcNull **;**

**exit ;**

**end ;**

**end ;**

**while** linAct[EndPos] **<>** #0 **do**

**case** linAct[PosFin] **of**

' **\*** ' **:**

**if** linAct[EndPos **+** 1] **=** ' **/** ' **then**

**begin**

inc **(** PosFin, 2 **);**

fRange **:=** rsUnknown **;**

**break ;**

**end**

**else** inc **(** PosEnd **);**

#10 **:** **break ;**

#13 **:** **break ;**

**else** inc **(** PosEnd **);**

**end ;**

**end ;**

**/////////** Implementation of range functionalities **//////////**

**procedure** TSynMiColor.ReSetRange **;**

**begin**

**inherited ;**

fRange **:=** rsUnknown **;**

**end ;**

**function** TSynMiColor.GetRange **:** Pointer **;**

**begin**

CodeFoldRange.RangeType **:=** Pointer **(** PtrInt **(** fRange **));**

Result **:=** **inherited ;**

**end ;**

**procedure** TSynMiColor.SetRange **(** Value **:** Pointer **);**

**begin**

**inherited ;**

fRange **:=** TRangeState **(** PtrUInt **(** CodeFoldRange.RangeType **));**

**end ;**

**initialization**

CreateTableIdentif **;** **//** Create the table for quick lookup

**end** .

Not all comments have been included, due to space issues.

### Improving Folding

The way the folding has been shown, it is a fairly simple process. However, folding in SynEdit is quite complete and includes additional functionalities.

So far we can summarize that to control the folding, the StartCodeFoldBlock() and EndCodeFoldBlock() methods are required, which have the following declarations:

**function** StartCodeFoldBlock **(** ABlockType **:** Pointer **;**

IncreaseLevel **:** Boolean **=** **true ):** TSynCustomCodeFoldBlock **;** **virtual ;**

**procedure** EndCodeFoldBlock **(** DecreaseLevel **:** Boolean **=** **True );** **virtual ;**

The operation of these methods is simple. StartCodeFoldBlock() adds a folding block and EndCodeFoldBlock() removes the last added method.

There is no greater magic in this. Each call to EndCodeFoldBlock() will always delete the last fold added with StartCodeFoldBlock().

In its simplest form, to start a folding block, we would use:

StartCodeFoldBlock **( nil );**

And to close a folding block we would do:

EndCodeFoldBlock **();**

This way of working would be sufficient for a simple fold. However, we might need to know which block we are handling in order to decide if it is valid or not, close the current block. For example, we know that in Pascal the reserved word UNTIL closes the REPEAT block, but not a BEGIN block.

In order to know which block we are handling, we could create an auxiliary structure, such as a queue (since folding blocks can be nested), and always be able to read which is the last block we are in.

But such work is not necessary, because such a structure already exists in the “TSynCustomFoldHighlighter” class, and is prepared especially for this work.

To identify our current folding block, we must use the “ABlockType” parameter of StartCodeFoldBlock().

To do this, it might be convenient to have a list of folding block identifiers in our highlighter:

TMyBlockType **=** **(**

cfbt\_BeginEnd,

cfbt\_RepeatUntil,

cfbt\_RecordEnd,

cfbt\_uses,

cfbt\_var

**);**

Then when we want to indicate the type of block, we would use the following code:

StartCodeFoldBlock **(** Pointer **(** PtrInt **(** ABlockType **)));**

The type conversion is necessary, because the parameter “ABlockType” is of type pointer.

Closing this block makes no difference to any other block. A simple call to EndCodeFoldBlock() would be enough:

EndCodeFoldBlock **();**

No further information is necessary, because it is known that it will always be the last block, the one that will be closed.

To know which is the last block in the fold stack, we have the TopCodeFoldBlockType() method:

**function** TopCodeFoldBlockType **(** DownIndex **:** Integer **=** 0 **):** Pointer **;**

The parameter is optional, and should generally be left at zero, unless you want to obtain another innermost folding block.

Now, with the methods: StartCodeFoldBlock(), EndCodeFoldBlock() and TopCodeFoldBlockType(), we have everything we need to create complex folding structures within a highlighter.

Type comparisons can now be made:

**var**

p **:** Pointer **;**

...

p **:=** TopCodeFoldBlockType **;** **//** read last block entered

**//** checks if it is appropriate to close the current block

**if** TMyBlockType **(** PtrUInt **(** p **))** **in** [cfbt\_BeginEnd, cfbt\_RecordEnd] **then**

EndCodeFoldBlock **(** DecreaseLevel **);**

...

### More about Folding

In the StartCodeFoldBlock() and EndCodeFoldBlock() methods, there is a parameter that allows managing the visibility of the folding mark. This is “IncreaseLevel” for StartCodeFoldBlock() and “DecreaseLevel” for EndCodeFoldBlock().

Calling StartCodeFoldBlock(), with “IncreaseLevel” set to FALSE, will not display the typical fold mark on the left side of the editor (Gutter). However, the folding will be done internally, or rather, internal information will be generated in the same way as it would be done with a visible fold, but without allowing the code to be folded.

This can be confusing. Someone might ask why would I do such a thing? Why would you create a fold that is not visible? What use would a folding block that cannot be folded have?

The simple answer would be: Because in this way it is possible to enable or disable folding without altering the structure of a text.

If we handle several folding blocks, and there are relationships between these blocks, it is not advisable to remove any of them because the structure of the other folding blocks could be altered. Therefore, when you want to “eliminate” a folding block, what you actually do is simply hide it, setting “IncreaseLevel” to FALSE, when calling StartCodeFoldBlock().

In the same way, to be consistent, you should always set “DecreaseLevel” to FALSE, in EndCodeFoldBlock(), when the block has been created with “IncreaseLevel” in FALSE.

Let's get to know the main methods a little better. The StartCodeFoldBlock() method has the following implementation:

**function** TSynCustomFoldHighlighter.StartCodeFoldBlock **(** ABlockType **:** Pointer **;**

IncreaseLevel **:** Boolean **=** **True ):** TSynCustomCodeFoldBlock **;**

**begin**

Result **:=** CodeFoldRange.Add **(** ABlockType, IncreaseLevel **);**

**end ;**

The EndCodeFoldBlock() method has the following implementation:

**procedure** TSynCustomFoldHighlighter.EndCodeFoldBlock **(** DecreaseLevel **:** Boolean **=** **True );**

**begin**

CodeFoldRange.Pop **(** DecreaseLevel **);**

**end ;**

The “CodeFoldRange” object works as a stack to which folding information is added and removed.

The “CodeFoldRange” property is a reference to “FCodeFoldRange” of the TSynCustomHighlighterRange class:

TSynCustomHighlighterRange **=** **class**

**private**

FCodeFoldStackSize **:** integer **;** **//** EndLevel

FMinimumCodeFoldBlockLevel **:** integer **;**

FRangeType **:** Pointer **;**

FTop **:** TSynCustomCodeFoldBlock **;**

**public**

**constructor** Create **(** Template **:** TSynCustomHighlighterRange **);** **virtual ;**

**destroyer** Destroy **;** **override ;**

**function** Compare **(** Range **:** TSynCustomHighlighterRange **):** integer **;** **virtual ;**

**function** Add **(** ABlockType **:** Pointer **=** **nil ;** IncreaseLevel **:** Boolean **=** **True ):**

TSynCustomCodeFoldBlock **;** **virtual ;**

**procedure** Pop **(** DecreaseLevel **:** Boolean **=** **True );** **virtual ;**

**function** MaxFoldLevel **:** Integer **;** **virtual ;**

**procedure** Clear **;** **virtual ;**

**procedure** Assign **(** Src **:** TSynCustomHighlighterRange **);** **virtual ;**

**procedure** WriteDebugReport **;**

**property** FoldRoot **:** TSynCustomCodeFoldBlock **read** FTop write FTop **;**

**public**

**property** RangeType **:** Pointer **read** FRangeType write FRangeType **;**

**property** CodeFoldStackSize **:** integer **read** FCodeFoldStackSize **;**

**property** MinimumCodeFoldBlockLevel **:** integer

**read** FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel **;**

**property** Top **:** TSynCustomCodeFoldBlock **read** FTop **;**

**end ;**

This object works as a stack with respect to the folded ones. The Add() method adds a new element and Pop() extracts the last element.

The Pop() method has a protection against trying to delete a fold that does not exist. So it could be executed without fear of overflowing “CodeFoldRange”.

To see the size of the stack, you can call “CodeFoldStackSize” at any time. This counter will only be updated when StartCodeFoldBlock() is called with the “IncreaseLevel” parameter set to TRUE or EndCodeFoldBlock() is called with the “DecreaseLevel” parameter set to TRUE, as is the default.

### Enable and disable folding blocks

A desirable feature of the folding functionality is that certain folding blocks can be enabled or disabled.

In the previous section, we saw how you can disable folding blocks, in the highlighter, using the “IncreaseLevel” and “DecreaseLevel” parameters. Now we will see how to use some predefined structures in the highlighter, to facilitate the management of enabling/disabling folding blocks.

The “TSynCustomFoldHighlighter” class includes methods and properties to handle what are called “Folding Settings”:

TSynCustomFoldHighlighter **=** **class (** TSynCustomHighlighter **)**

**protected**

**//** FoldConfig

FFoldConfig **:** **Array of** TSynCustomFoldConfig **;**

**function** GetFoldConfig **( Index :** Integer **):** TSynCustomFoldConfig **;** **virtual ;**

**procedure** SetFoldConfig **( Index :** Integer **;** **const** AValue **:** TSynCustomFoldConfig **);** **virtual ;**

**function** GetFoldConfigCount **:** Integer **;** **virtual ;**

**function** GetFoldConfigInternalCount **:** Integer **;** **virtual ;**

**function** GetFoldConfigInstance **( Index :** Integer **):** TSynCustomFoldConfig **;** **virtual ;**

**procedure** InitFoldConfig **;**

**procedure** DestroyFoldConfig **;**

**procedure** DoFoldConfigChanged **(** Sender **:** TObject **);** **virtual ;**

**private**

...

**protected**

...

**public**

**property** FoldConfig[ **Index :** Integer] **:** TSynCustomFoldConfig

**read** GetFoldConfig write SetFoldConfig **;**

**property** FoldConfigCount **:** Integer **read** GetFoldConfigCount **;**

**end ;**

These properties provide us with an internal structure for storing the folding configuration properties, and also exposes the FoldConfig and FoldConfigCount properties to access these properties.

It is possible to use any other custom structure, but TSynCustomFoldHighlighter includes this class and the methods and properties necessary for its management, so that it saves us the work of managing configurations of the folding ranges. Furthermore, since they are included in the same TSynCustomFoldHighlighter class, they are always accessible from the highlighters.

The configurations of a folding range are saved in the FFoldConfig array. A FoldConfig object is an instance of TSynCustomFoldConfig:

TSynCustomFoldConfig **=** **class (** TPersistent **)**

**private**

FEnabled **:** Boolean **;**

FFoldActions **:** TSynFoldActions **;**

FModes **:** TSynCustomFoldConfigModes **;**

FOnChange **:** TNotifyEvent **;**

FSupportedModes **:** TSynCustomFoldConfigModes **;**

...

**published**

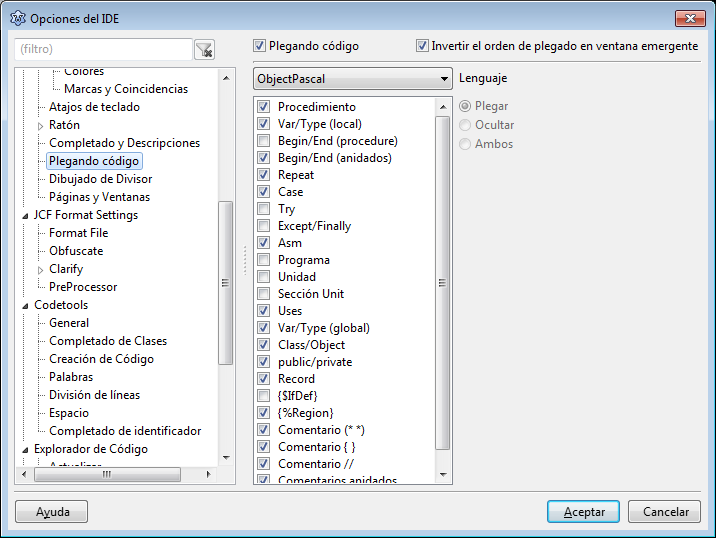
**property** Enabled **:** Boolean **read** FEnabled write SetFEnabled **;**

**property** Modes **:** TSynCustomFoldConfigModes **read** FModes write SetModes **default** [fmFold] **;**

**end ;**

The most important property is perhaps “Enabled”, because it allows us to decide when a folding range is enabled or not.

This structure is a base with the basic properties for a folding range. To get an idea of how they are used, you can go to the Lazarus environment, and select the menu “Tools>Options>Editor>Code Folding>”:



In this case, the folding blocks for the Lazarus Pascal highlighter are being shown. When you activate or deactivate any of these boxes, you are changing the “Enabled” property of one of the “ TSynCustomFoldConfig ” objects that exist in the highlighter.

In the implementation of most highlighters with folding in Lazarus, you will see that, to decide whether or not to make the fold visible, the FFoldConfig[] array is first consulted.

**function** TSynLFMSyn.StartLfmCodeFoldBlock **(** ABlockType **:** TLfmCodeFoldBlockType **):** TSynCustomCodeFoldBlock **;**

**var**

FoldBlock **:** Boolean **;**

p **:** PtrInt **;**

**begin**

FoldBlock **:=** FFoldConfig[ord **(** ABlockType **)** ].Enabled **;**

p **:=** 0 **;**

**if not** FoldBlock **then**

p **:=** PtrInt **(** CountLfmCodeFoldBlockOffset **);**

Result **:=** StartCodeFoldBlock **(** p **+** Pointer **(** PtrInt **(** ABlockType **))** , FoldBlock **);**

**end ;**

**procedure** TSynLFMSyn.EndLfmCodeFoldBlock **;**

**var**

DecreaseLevel **:** Boolean **;**

**begin**

DecreaseLevel **:=** TopCodeFoldBlockType **<** CountLfmCodeFoldBlockOffset **;**

EndCodeFoldBlock **(** DecreaseLevel **);**

**end ;**

In this code, the trick of giving “p”, a value offset from the original value, when using a non-visible fold, is used in StartLfmCodeFoldBlock(). So that you can later recover its original value and know if the fold was visible or not, in EndLfmCodeFoldBlock().

This method is a way to encode additional information into an enumeration (disguised as a pointer). As a consequence, this decoding must also be applied, overriding TopLfmCodeFoldBlockType():

**function** TSynLFMSyn.TopLfmCodeFoldBlockType **(** DownIndex **:** Integer **):** TLfmCodeFoldBlockType **;**

**var**

p **:** Pointer **;**

**begin**

p **:=** TopCodeFoldBlockType **(** DownIndex **);**

**if** p **>=** CountLfmCodeFoldBlockOffset **then**

p **:=** p **-** PtrUInt **(** CountLfmCodeFoldBlockOffset **);**

Result **:=** TLfmCodeFoldBlockType **(** PtrUInt **(** p **));**

**end ;**

As a rule, it is expected that there will be as many TSynCustomFoldConfig objects as there are different folding blocks. For example, in the SynLFMSyn highlighter, there are the following blocks:

TLfmCodeFoldBlockType **=** **(**

cfbtLfmObject, **//** **object** , **inherited** , **inline**

cfbtLfmList, **//** **<>**

cfbtLfmItem, **//** Item

cfbtLfmNone

**);**

Therefore, the GetFoldConfigInternalCount() and GetFoldConfigInstance() methods must be overridden so that the necessary configurations are created:

**function** TSynLFMSyn.GetFoldConfigInternalCount **:** Integer **;**

**begin**

Result **:=** ord **(** high **(** TLfmCodeFoldBlockType **))** **-ord** ( low **( TLfmCodeFoldBlockType** ) **)** **+** 1 **;**

**end ;**

**function** TSynLFMSyn.GetFoldConfigInstance **( Index :** Integer **):** TSynCustomFoldConfig **;**

**begin**

Result **:=** **inherited** GetFoldConfigInstance **( Index );**

Result.Enabled **:=** **True ;**

**end ;**

These methods are executed at startup, to create the configuration structures in FFoldConfig[].

## The TSynCustomFoldHighlighter class

As we have seen, all highlighters that wish to implement code folding must derive from the TSynCustomFoldHighlighter class instead of TSynCustomHighlighter.

You could say that, the TSynCustomFoldHighlighter class is a descendant of TSynCustomHighlighter, which simply adds code folding functionality.

As expected, the class declaration adds the information necessary for code folding and some useful methods to obtain information about the folding.

### Low Level Folding

To better understand how folding is implemented in the classroom, we will describe the working mechanics at a low level.

Let's consider a typical code in Pascal, with folding:



In this code you can see that the entire procedure is within a folding block. The body of the procedure opens another folding block. Inside, the IF structure opens two folding blocks: the IF body and the ELSE body.

As is common, we can save a few lines by joining some keywords on the same line:



In this code, the body of the IF ends on the same line where the body of the ELSE begins, therefore, a kind of “overlapping” of the blocks can be seen, but this is not the case. In fact, you could say that the body of the IF “ends” after the reserved word END, and that the body of the ELSE begins with the reserved word BEGIN.

Like folding blocks, they can be nested, so we are introducing the concept of “level”. If we have not opened any folding blocks, we will say that we are at level zero. When a folding block is opened, we will say that we are at level 1 and so on. That is why commonly, we will be talking about “nesting levels” within a folding block.

As in the management of ranges, for reasons of economy and simplicity, the information related to folding blocks is stored for each line, such as the state of the highlighter when it finishes exploring each line.

In order to correctly process the code folding, the TSynCustomFoldHighlighter class saves two values for each line:

* EndLevel.- It is the nesting level of blocks at the end of the line.
* MinLevel.- It is the minimum nesting level anywhere on the line.

These variables, and more, are saved as part of the information that the highlighter stores in each line of the editor.

A visual exercise will help us understand how code folding is processed within TSynCustomFoldHighlighter:



In the first line the minimum level is 0, because before the reserved word PROCEDURE, no block has been opened yet.

In the second line, a new block is opened with the reserved word BEGIN. For code folding purposes, it is not so important to know whether the block starts before or after BEGIN.

The other lines follow the same logic.

This information is all the editor needs to display the fold marks in the side panel. Whenever EndLevel > MinLevel is found, it means that one or more folding blocks have been opened, and the block flag should be displayed. If EndLevel, in the previous line is greater than MinLevel, in one line, it means that the block of the previous line has been closed. This information allows you to quickly scan the lines to obtain information about the folding blocks.

Of course, TSynCustomFoldHighlighter handles more information to deal with the blocks, but for mark display purposes, this information would be enough, and it is part of the information that is stored in each line of the editor.

It may seem that folding information is lost if a block is opened and closed within the same line, but it must be considered that for code folding purposes, that block would be inconsequential.

To view the EndLevel and MinLevel values, you can access the properties:

**function** TSynCustomFoldHighlighter.FoldBlockEndLevel **(** ALineIndex **:** TLineIdx **;**

**const** AFilter **:** TSynFoldBlockFilter **):** integer **;**

**function** TSynCustomFoldHighlighter.FoldBlockMinLevel **(** ALineIndex **:** TLineIdx **;**

**const** AFilter **:** TSynFoldBlockFilter **):** integer **;**

It is only necessary to pass the required line number (starting at zero). The “AFilter” field is not used, so it should be left at NIL.

### CurrentLines[] and CurrentRanges[]

These TSynCustomFoldHighlighter properties return the same information that TSynCustomHighlighter would return, more or less.

CurrentLines[], is exactly the same and allows access to the lines of the current editor. Until then everything works fine.

However CurrentRanges[], has changed. It is now a pointer to objects of type TSynCustomHighlighterRange.

The fact that TSynCustomFoldHighlighter not only stores Range information makes it necessary to have a way to add additional information to each line of the editor.

The handling of folding blocks requires additional information, so a special structure has been created that allows packaging the range information (the same pointer as always), and additionally other fields.

That is why the TSynCustomHighlighterRange class exists, defined as follows:

TSynCustomHighlighterRange **=** **class**

**private**

FCodeFoldStackSize **:** integer **;** **//** EndLevel

FMinimumCodeFoldBlockLevel **:** integer **;**

FRangeType **:** Pointer **;**

FTop **:** TSynCustomCodeFoldBlock **;**

**public**

**constructor** Create **(** Template **:** TSynCustomHighlighterRange **);** **virtual ;**

**destroyer** Destroy **;** **override ;**

**function** Compare **(** Range **:** TSynCustomHighlighterRange **):** integer **;** **virtual ;**

**function** Add **(** ABlockType **:** Pointer **=** **nil ;** IncreaseLevel **:** Boolean **=** **True ):**

TSynCustomCodeFoldBlock **;** **virtual ;**

**procedure** Pop **(** DecreaseLevel **:** Boolean **=** **True );** **virtual ;**

**function** MaxFoldLevel **:** Integer **;** **virtual ;**

**procedure** Clear **;** **virtual ;**

**procedure** Assign **(** Src **:** TSynCustomHighlighterRange **);** **virtual ;**

**procedure** WriteDebugReport **;**

**property** FoldRoot **:** TSynCustomCodeFoldBlock **read** FTop write FTop **;**

**public**

**property** RangeType **:** Pointer **read** FRangeType write FRangeType **;**

**property** CodeFoldStackSize **:** integer **read** FCodeFoldStackSize **;**

**property** MinimumCodeFoldBlockLevel **:** integer

**read** FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel **;**

**property** Top **:** TSynCustomCodeFoldBlock **read** FTop **;**

**end ;**

This means that now in TSynCustomFoldHighlighter, when CurrentRanges[] is read for a line, we will not directly read the range value, but rather we will obtain the reference to a TSynCustomHighlighterRange object, with additional information (a lot of additional information), necessary for handling the folding blocks.

The following diagram clarifies the situation better:



The range information now becomes another humble field of this new object. The field is RangeType. That is why, in a highlighter with folding, when the methods are implemented:

TSynFacilSyn.GetRange: Pointer;

TSynFacilSyn.SetRange(Value: Pointer);

, you must work with the “CodeFoldRange.RangeType” field, instead of the direct value of the function.

CurrentRanges[] still stores pointers, as it does in TSynCustomHighlighter, but now these pointers are references to objects of type TSynCustomFoldHighlighterRange.

So we can apply type conversion when reading CurrentRanges[]:

p **:=** TSynCustomHighlighterRange **(** CurrentRanges[SynEdit1.CaretY **-** 1] **)**

Arguably the most important properties of a TSynCustomFoldHighlighterRange object are RangeType and Top.

The Top property is a reference to a TSynCustomCodeFoldBlock object, which represents the last Folding Block that has been opened, at the end of the corresponding line of CurrentRanges[].

A TSynCustomCodeFoldBlock object is not exactly the folding block (the one we opened with StartCodeFoldBlock()), but rather it is an object that functions as the node of a linked list.

This is because all blocks opened, with StartCodeFoldBlock(), are stored in a linked list, and not in a LIFO stack, as you might expect.



The Add() and Pop() methods of TSynCustomHighlighterRange allow you to add or remove nodes from the list.

Each node in the list is an object of type TSynCustomCodeFoldBlock, which has the following definition:

TSynCustomCodeFoldBlock **=** **class**

**private**

FBlockType **:** Pointer **;**

FParent, FChildren **:** TSynCustomCodeFoldBlock **;**

FRight, FLeft **:** TSynCustomCodeFoldBlock **;**

FBalance **:** Integer **;**

**function** GetChild **(** ABlockType **:** Pointer **):** TSynCustomCodeFoldBlock **;**

**protected**

**function** GetOrCreateSibling **(** ABlockType **:** Pointer **):** TSynCustomCodeFoldBlock **;**

**property** Right **:** TSynCustomCodeFoldBlock **read** FRight **;**

**property** Left **:** TSynCustomCodeFoldBlock **read** FLeft **;**

**property** Children **:** TSynCustomCodeFoldBlock **read** FChildren **;**

**public**

**destroyer** Destroy **;** **override ;**

**procedure** WriteDebugReport **;**

**public**

**procedure** InitRootBlockType **(** AType **:** Pointer **);**

**property** BlockType **:** Pointer **read** FBlockType **;**

**property** Parent **:** TSynCustomCodeFoldBlock **read** FParent **;**

**property** Child[ABlockType **:** Pointer] **:** TSynCustomCodeFoldBlock **read** GetChild **;**

**end ;**

The links to the other nodes are found in the “Parent” and “Children” properties.

The most important data that this node contains is BlockType, which is the reference to the folding block that we indicate when we use StartCodeFoldBlock(). There is one BlockType for each node, that is, for each open block.

As with the RangeType, the BlockType pointer can refer to a real object or be an integer or enumeration encoded in a data pointer type. This will depend on how the highlighter that makes use of code folding is implemented.

### Some Methods and Properties

To handle folding, TSynCustomFoldHighlighter has several methods available, which can be accessed from inside and outside the class.

The FoldEndLine() method, with the declaration:

**function** TSynCustomFoldHighlighter.FoldEndLine **(** ALineIndex, FoldIndex **:** Integer **):** integer **;**

, allows returning the final line of the block, at the end of the ALineIndex line (which starts at 0 for the first line). The second parameter allows you to specify the level of the block to use. For the highest level block, it should be left at zero.

Let's consider the following code:



If we apply the FoldEndLine(1,0) function, we will get 8, because the highest level open block, at the end of line 2, ends on line 9.

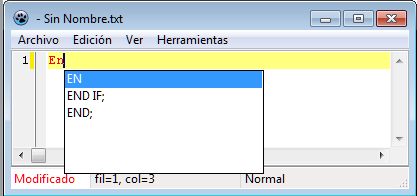
The FoldLineLength() method returns the number of lines of the block opened at the end of the indicated line. It has the same parameters as FoldEndLine():

**function** TSynCustomFoldHighlighter.FoldLineLength **(** ALineIndex, FoldIndex **:** Integer **):** integer **;**

Likewise, ALineIndex starts at zero for line 1.

## Autocomplete

Autocomplete is the feature that SynEdit has that allows it to display, while typing, a list of words that we can choose to complete the missing text in the current word. This way we will save ourselves the trouble of having to write the entire word.



Autocompletion in SynEdit is not as developed as the folding or syntax highlighting feature. There are basic functionalities, but it does its job.

Like several other features of SynEdit, it is possible to implement autocompletion using components ("TSynCompletion"), from the components bar, or you can use code to create it.

Autocompletion has been defined to have the following *modus operandi* :

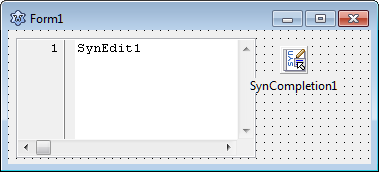
1. It is always associated with an editor of type SynEdit. This association can be done in design, or by code.
2. It is activated every time the key combination associated with completion is pressed. The most common combination used is <Ctrl> + <space>.
3. When activated, a window should appear, at the cursor position, with a list of words (or phrases), from which one of them must be selected.
4. When the selection window is displayed, the current word, the one before the cursor, is taken as the working word. This is the word that will be replaced.
5. If when the selection window is displayed, it only has one option, the work word will be automatically replaced with this only option and the selection window will be closed.
6. When the selection window is displayed, you can still type in the editor, but the selection window will take control of some keys, such as the arrow keys. Additionally, deleting the work word will not be allowed, because it must have at least one character.
7. In the displayed list, you can use the arrow keys, <Page Up>, <Page Down>, <Home>, or <End> to scroll through the options. These keys will no longer act on the editor while the list of options is displayed.
8. To select one of the options you must press the <enter> key, <space>, or a symbol such as '+', or '-'.
9. When you select one of the options from the list, the current word will be replaced by the selected one, and the selection window will disappear.
10. IF you do not want to select any option in the options window, you must press <escape> to make the options window disappear and continue normal editing.
11. The options window can only be made to disappear by selecting one of its items or by pressing <escape>.

This is normal behavior for the autocomplete functionality. To change this behavior, you must manipulate the properties of the completion component (TSynCompletion), either through the design window or through code.

For major modifications, it must be done entirely by code.

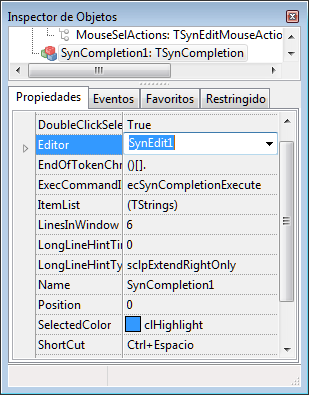
### Autocompletion using component

The procedure is simple. The TSynEdit editor is added to the form, naturally, and then the “TSynCompletion” component is also added:



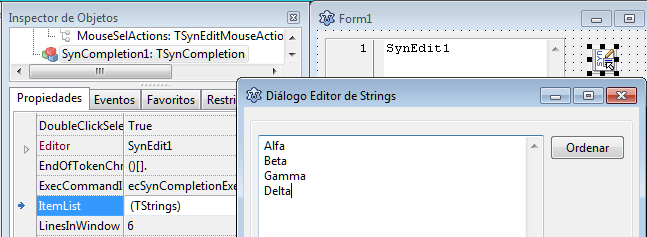
The “TSynCompletion” component is found in the components palette, where “TSynEdit” is also found.

Once we have the working components, we need to associate “SynCompletion1” to “SynEdit1”. To do this, configure the “Editor” property of “SynCompletion1” in the object explorer to point to “SynEdit1”:



Note that the default property for “ShortCut” is “Ctrl + Space”, which means that this combination will be the one that activates the autocomplete drop-down list.

Once this property is associated, the functionality will be ready to work in the editor. But you still need to add the list of words that will be displayed in the dropdown menu. To do this we can modify the “ItemList” property directly from the Object Inspector:



Now, when the program is run, and the combination <Ctrl> + <space> is pressed, a drop-down list will appear with the indicated words, which will allow us to replace the current word with one of the words on the list.

This working method is simple and quick to implement, but it does not allow more elaborate working options such as filtering the list according to the word being written. For this and other functionalities, it is necessary to work with code.

### Autocomplete using code

For autocompletion, an object of class “TSynCompletion” must be created:

MenuContex **:** TSynCompletion **;** **//** context menu

Initialization is simple:

MenuContex **:=** TSynCompletion.Create **(** Self0 **);** **//** create menu

MenuContex.Editor **:=** ed **;** **//** assign editor

MenuContex.ShortCut **:=** Menus.ShortCut **(** VK\_SPACE, [ssCtrl] **);** **//** assign keyboard shortcut

Under normal conditions, the completion window will only be activated when the shortcut key combination is pressed. If you want to show the window at any other time, you can use the execute method:

The following procedure displays the “TSynCompletion1” completion window for the “ed” editor:

**procedure** Show **;**

**var** p **:** TPoint **;**

**begin**

P **:=** Point **(** ed.CaretXPix,ed.CaretYPix **+** ed.LineHeight **);**

PX **:=** Max **(** 0,Min **(** PX, ed.ClientWidth **-** TSynCompletion1.Width **));**

P **:=** ed.ClientToScreen **(** p **);**

**//** Open context menu. It will only be shown if it has items.

SynCompletion1.Execute **(** '' , px, py **);**

**end ;**

The calculation of px and py is done so that the window appears at the cursor position. It is important to note that the window will only be displayed if “TSynCompletion1” has elements in its list.

To add items to “TSynCompletion1”, you must use the “ItemList” property, which is a “StringList”:

SynCompletion1.ItemList.add **(** 'Alpha' **);**

SynCompletion1.ItemList.add **(** 'Beta' **);**

SynCompletion1.ItemList.add **(** 'Gamma' **);**

The strings that are added are the ones that will appear when the “TSynCompletion” window is displayed.

If the word list is to be fixed, then it can be filled only once at the start of the program. If the list depends on the current word, or other information, then you can make use of one of the “TSynCompletion” events. The event in question is “OnExecute”. This event will be executed before opening the word list for completion. Here you can use it to choose the list of words to fill out. As an additional help, the string “SynCompletion1.CurrentString” can be used. The identifier that is before the cursor is stored here, that is, the working word.

The “OnExecute” event will be raised regardless of whether the list was activated by key combination or using the “Execute” method. But only when activated with a key combination is the “CurrentString” string automatically updated. When “Execute” is used, the first parameter is the initial value that is assigned to “CurrentString”.

With each key pressed, while the selection window is active, the “CurrentString” string is updated.

Another important event is “OnSearchPosition”. This event is called every time a key is pressed while the “TSynCompletion” window is open. As a parameter, it provides an integer that indicates the selected element. The first element is 0. This value can be changed from within the event, to choose another element to select. If you do not use this event, by default, the element that matches “CurrentString” is selected.

The “CurrentString” property indicates the current word that is behind the cursor. It doesn't matter if there are characters in front of the cursor, “CurrentString” only considers the previous characters, until it finds a space. This property is only updated when invoking “TSynCompletion” from the keyboard. If opened with “Execute()”, “CurrentString” will start empty.

# APPENDIX

## Algorithms for implementing highlighters.

They can be implemented in various ways. The main objective is to have a quick response and they focus mainly on the quick comparison of strings. The algorithms that have been tested the most are:

### Algorithm based on the use of Hash functions

This is the algorithm that has been used to implement most of the pre-defined highlighters that come with Lazarus. It is fast to process, but has a complicated implementation.

It works with a function table “fProcTable[]”, which is used to direct each character, according to its type, to a specific function that will be responsible for identifying the analyzed token.

Filling this table is done before using the highlighter, in the MakeMethodTables() function:

**procedure** TSynPerlSyn.MakeMethodTables **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

#0 **:** fProcTable[I] **:=** @NullProc **;**

#1..#9,#11,#12,#14..#32 **:** fProcTable[I] **:=** @SpaceProc **;**

#10 **:** fProcTable[I] **:=** @LFProc **;**

#13 **:** fProcTable[I] **:=** @CRProc **;**

' **:** ' **:** fProcTable[I] **:=** @ColonProc **;**

'#' **:** fProcTable[I] **:=** @CommentProc **;**

' **=** ' **:** fProcTable[I] **:=** @EqualProc **;**

' **>** ' **:** fProcTable[I] **:=** @GreaterProc **;**

' **<** ' **:** fProcTable[I] **:=** @LowerProc **;**

'0' ..'9', '.' **:** fProcTable[I] **:=** @NumberProc **;**

'$' , 'A'..'Z', 'a'..'z', '\_' **:** fProcTable[I] **:=** @IdentProc **;**

' **-** ' **:** fProcTable[I] **:=** @MinusProc **;**

' **+** ' **:** fProcTable[I] **:=** @PlusProc **;**

' **/** ' **:** fProcTable[I] **:=** @SlashProc **;**

' **\*** ' **:** fProcTable[I] **:=** @StarProc **;**

#34 **:** fProcTable[I] **:=** @StringInterpProc **;**

#39 **:** fProcTable[I] **:=** @StringLiteralProc **;**

**else**

fProcTable[I] **:=** @UnknownProc **;**

**end ;**

**end ;**

Here alphabetic characters are always considered the beginning of identifiers and are pointed to the IdentProc() function that will be responsible for extracting the identifier and verifying if it is a keyword.

To detect key identifiers, follow the following process:

The token corresponding to the identifier is extracted and the result of its hash function is calculated [[13]](#footnote-13). With this value, a function table is addressed, called fIdentFuncTable[], which must have entries enabled only in the hash function values that correspond to keywords.

The fIdentFuncTable[] table is populated in the InitIdent method, and must be done before using the highlighter:

**procedure** TSynPerlSyn.InitIdent **;**

**var**

I **:** Integer **;**

**begin**

**for** I **:=** 0 **to** 2167 **do**

**Case** I **of**

109 **:** fIdentFuncTable[I] **:=** @Func109 **;**

113 **:** fIdentFuncTable[I] **:=** @func113 **;**

196 **:** fIdentFuncTable[I] **:=** @func196 **;**

201 **:** fIdentFuncTable[I] **:=** @func201 **;**

204 **:** fIdentFuncTable[I] **:=** @func204 **;**

207 **:** fIdentFuncTable[I] **:=** @func207 **;**

209 **:** fIdentFuncTable[I] **:=** @func209 **;**

211 **:** fIdentFuncTable[I] **:=** @func211 **;**

230 **:** fIdentFuncTable[I] **:=** @func230 **;**

...

**else**

fIdentFuncTable[I] **:=** @AltFunc **;**

**end ;**

**end ;**

The addressed functions are of the form “funcXXX”, where XXX is the corresponding “hash” value. For a group of keywords, there are only a number of hash values in the range.

The other entries only return the value “tkIdentifier” (via AltFunc), indicating that it is a simple identifier.

If there are several keywords that have the same hash value, an additional comparison is implemented within the corresponding function:

**function** TSynPerlSyn.Func230 **:** TtkTokenKind **;**

**begin**

**if** KeyComp **(** 'tr' **)** **then** Result **:=** tkKey **else**

**if** KeyComp **(** 'my' **)** **then** Result **:=** tkKey **else** Result **:=** tkIdentifier **;**

**end ;**

### Algorithm based on the First character as a prefix.

This algorithm is the one we describe in this document and is based on using the first character of an identifier as a prefix, to speed up the detection of keywords.

It also uses a function table “fProcTable[]”, which is used to direct each character, according to its type, to a specific function that will be responsible for identifying the analyzed token. The difference is that this table will point to a specific function for each alphabetic character:

**procedure** TSynMiColorSF.MakeMethodTables **;**

**var**

I **:** Char **;**

**begin**

**for** I **:=** #0 **to** #255 **do**

**case** I **of**

' **-** ' **:** fProcTable[I] **:=** @ProcMinus **;**

' **/** ' **:** fProcTable[I] **:=** @ProcSlash **;**

#39 **:** fProcTable[I] **:=** @ProcString **;**

'"' **:** fProcTable[I] **:=** @ProcString2 **;**

'0' ..'9' **:** fProcTable[I] **:=** @ProcNumber **;**

'A' ,'a' **:** fProcTable[I] **:=** @ProcA **;**

'B' ,'b' **:** fProcTable[I] **:=** @ProcB **;**

'C' ,'c' **:** fProcTable[I] **:=** @ProcC **;**

'D' ,'d' **:** fProcTable[I] **:=** @ProcD **;**

...

'Z' ,'z' **:** fProcTable[I] **:=** @ProcZ **;**

'\_' **:** fProcTable[I] **:=** @ProcUnder **;**

'$' **:** fProcTable[I] **:=** @ProcMacro **;**

#13 **:** fProcTable[I] **:=** @ProcCR **;**

#10 **:** fProcTable[I] **:=** @ProcLF **;**

#0 **:** fProcTable[I] **:=** @ProcNull **;**

#1..#9, #11, #12, #14..#32 **:** fProcTable[I] **:=** @ProcSpace **;**

**else** fProcTable[I] **:=** @ProcUnknown **;**

**end ;**

**end ;**

Then each function of type ProcA() or ProcB() directly makes the comparisons to identify the keywords:

**procedure** TSynMiColorSF.ProcA **;**

**begin**

**while** Identifiers[fLine[Run]] **do** inc **(** Run **);**

fStringLen **:=** Run **-** fTokenPos **-** 1 **;** **//** calculate size **-** 1

fToIdent **:=** Fline **+** fTokenPos **+** 1 **;** **//** pointer to identifier **+** 1

**if** KeyComp **(** 'nd' **)** **then** fTokenID **:=** tkKey **else**

**if** KeyComp **(** 'rray' **)** **then** fTokenID **:=** tkKey **else**

fTokenID **:=** tkIdentifier **;** **//** common identifier

**end ;**

Since the first character of the identifier has already been identified, the comparison is done with one less character, which speeds up the comparison.

### Comparison between algorithms

To verify the performance of both algorithms, I have performed two sets of comparisons. One of them using the Lazarus 1.0.12 PHP highlighter and the other using the Perl highlighter.

Both highlighters (implemented with the hash algorithm) were compared in speed to a highlighter implemented with the First Character as Prefix algorithm.

These were the results:

|  |  |
| --- | --- |
| PHP highlighter (with hashing algorithm): | 1.1sec, 1.1sec, 1.1sec. |
| PHP Highlighter (with prefix algorithm): | 1.0 sec, 1.0 sec, 1.0 sec. |

|  |  |
| --- | --- |
| Resaltador Perl (con algoritmo hash): | 1.84 seg, 1.82 seg, 1.83 seg |
| Resaltador Perl (con algoritmo de prefijo): | 1.68 seg, 1.68 seg, 1.68 seg. |

The tests show the times it takes to process an entire file a given number of times. The test with Perl performed 5000 simple scans of a file. The scan routine looked like this:

**procedure** BrowseFile **(** lines **:** TStringList **;** hlt **:** TSynCustomHighlighter **);**

**//** Explore a file using the indicated highlighter.

**var**

p **:** PChar **;**

tam **:** integer **;**

Lin **:** **string ;**

**begin**

**for** lin **in** lines **do begin**

hlt.SetLine **(** lin,1 **);**

**while not** hlt.GetEol **do begin**

hlt.Next **;**

hlt.GetTokenEx **(** p,tam **);**

hlt.GetTokenAttribute **;**

**end ;**

**end ;**

**end ;**

Both tests were done taking a PHP and Perl source file as examples. Perl's had 427 lines. The comparison was made on 32-bit Windows, on Intel architecture.

After carrying out the tests, it is concluded that the prefix algorithm is generally faster than the hash function algorithm. There may be particular cases where this is not true, but the implementation of the prefix algorithm can always be improved to make it comparatively faster.

Considering other criteria for comparison, the following table can be prepared:

|  |  |  |
| --- | --- | --- |
| **CRITERION** | **HASH FUNCTION ALGORITHM** | **ALGORITHM OF THE FIRST CHARACTER AS A PREFIX** |
| Speed | Slower in most cases | Faster in most cases |
| Code size | Greater, due to the fact that it requires an additional function table and a large number of functions. | Minor, because it only uses a table of functions and simple comparisons. |
| Optimizable | Difficult to optimize. It requires improving the hash function, but can complicate processing time. | Easier to optimize. More levels can be added in the form of a prefix tree or comparisons can be improved. |
| Readability | Not very readable. Difficult to follow the code. | It is more readable. |
| Maintenance. | Difficult to maintain. Adding new keywords involves recalculating your hash function. | Easy to maintain. Adding new keywords just means putting them in the corresponding function. |
| Possibility of using external syntax files. | Very few. Its inherent structure complicates making it dynamic. | More manageable. By ordering its identifiers, it allows it to be better adapted to the use of external syntax files. |

According to the comparison made, it is NOT RECOMMENDED to implement syntax highlighters with the Hash function algorithm, which has been used in SynEdit.

### Optimization criteria

I have repeatedly stressed the need to use fast response algorithms in highlighter implementation.

Here I want to indicate some points to take into account in the implementation of algorithms for highlighters, considering that we are using the Free Pascal compiler in its version 2.6.2. Some of these criteria may not be true in other compilers or even in other versions of Free Pascal.

The first point we must consider is that strings of type PChar are generally faster than strings of type String, since they are handled as pointers.

Another consideration regarding pointers is that, if you have a string of type PChar, declared like this:

fLine: PChar;

The quickest way to access a character is: fLine[i]

The form: (fLine+i)^ is noticeably slower (it has been estimated 7% slower).

In highlighters, it is common to have to advance a pointer as long as the pointed character belongs to a valid character set.

To quickly compare whether a character is in a character set, three methods are discussed:

Array of boolean values

A typical algorithm would be:

**var** CharsIdentif **:** **array** [#0..#255] **of** ByteBool **;**

...

**while** CharsIdentif[fLine[posEnd]] **do**

inc **(** posEnd **);**

Here we assume that the CharsIdentif[] array has been initialized for the characters you want to consider valid.

Compare to Sets

A typical algorithm would be:

**var** letters **:** **set of** char **;**

...

**while** fLine[posFin] **in** letters **do**

inc **(** posEnd **);**

Here it is assumed that the letter set has started with the characters that you want to consider valid.

Compare to Case...Of

A quick comparison method can also be implemented with the following structure:

**while true do begin**

**case** fLine[posFin] **of**

'$' ,'\_','0'..'9','a'..'z','A'..'Z' : **inc** ( **posEnd** ) **;**

**else break ;** **//** exit

**end**

**end ;**

A comparative table shows the response speeds of these three methods, tested on Windows with 32-bit x86 architecture:

|  |  |
| --- | --- |
| **METHOD** | **TIME** |
| Array of boolean values | 39 |
| Comparison with sets | 48 |
| Compare to Case...Of | Four. Five |

It can be seen that the comparison using an array of booleans is about 20% faster than the comparison using sets and about 13% faster than using Case ..Of.

A comparison was also tried using an array of integers, instead of Boolean values, obtaining a longer delay. Using integers is about 1% or 2% slower.

# INDEX

[1](#__RefHeading___Toc392668336)  [Syntax-aware editor: SynEdit.](#__RefHeading___Toc392668336)  [5](#__RefHeading___Toc392668336)

[1.1](#__RefHeading___Toc392668337)  [What is SynEdit?](#__RefHeading___Toc392668337)  [5](#__RefHeading___Toc392668337)

[1.2](#__RefHeading___Toc392668338)  [Features of SynEdit](#__RefHeading___Toc392668338)  [6](#__RefHeading___Toc392668338)

[1.3](#__RefHeading___Toc392668339)  [Appearance](#__RefHeading___Toc392668339)  [6](#__RefHeading___Toc392668339)

[1.3.1](#__RefHeading___Toc392668340)  [Vertical Panel](#__RefHeading___Toc392668340)  [8](#__RefHeading___Toc392668340)

[1.3.2](#__RefHeading___Toc392668341)  [Right Margin](#__RefHeading___Toc392668341)  [9](#__RefHeading___Toc392668341)

[1.3.3](#__RefHeading___Toc392668342)  [Typography](#__RefHeading___Toc392668342)  [10](#__RefHeading___Toc392668342)

[1.4](#__RefHeading___Toc392668343)  [Operation](#__RefHeading___Toc392668343)  [13](#__RefHeading___Toc392668343)

[1.4.1](#__RefHeading___Toc392668344)  [Editor](#__RefHeading___Toc392668344)  [13 Coordinates](#__RefHeading___Toc392668344)

[1.4.2](#__RefHeading___Toc392668345)  [Cursor handling](#__RefHeading___Toc392668345)  [16](#__RefHeading___Toc392668345)

[1.4.3](#__RefHeading___Toc392668346)  [Line delimiter](#__RefHeading___Toc392668346)  [17](#__RefHeading___Toc392668346)

[1.5](#__RefHeading___Toc392668347)  [Modify content](#__RefHeading___Toc392668347)  [19](#__RefHeading___Toc392668347)

[1.5.1](#__RefHeading___Toc392668348)  [Execute commands](#__RefHeading___Toc392668348)  [20](#__RefHeading___Toc392668348)

[1.5.2](#__RefHeading___Toc392668349)  [Accessing Lines[]](#__RefHeading___Toc392668349)  [22](#__RefHeading___Toc392668349)

[1.5.3](#__RefHeading___Toc392668350)  [The Clipboard](#__RefHeading___Toc392668350)  [23](#__RefHeading___Toc392668350)

[1.5.4](#__RefHeading___Toc392668351)  [Do and Undo.](#__RefHeading___Toc392668351)  [25](#__RefHeading___Toc392668351)

[1.6](#__RefHeading___Toc392668352)  [Selection management](#__RefHeading___Toc392668352)  [27](#__RefHeading___Toc392668352)

[1.6.1](#__RefHeading___Toc392668353)  [Selection in Column Mode](#__RefHeading___Toc392668353)  [29](#__RefHeading___Toc392668353)

[1.6.2](#__RefHeading___Toc392668354)  [BlockBegin and BlockEnd](#__RefHeading___Toc392668354)  [31](#__RefHeading___Toc392668354)

[1.6.3](#__RefHeading___Toc392668355)  [BlockBegin and BlockEnd in Normal Selection](#__RefHeading___Toc392668355)  [34](#__RefHeading___Toc392668355)

[1.6.4](#__RefHeading___Toc392668356)  [BlockBegin and BlockEnd in Selection in Column Mode](#__RefHeading___Toc392668356)  [34](#__RefHeading___Toc392668356)

[1.7](#__RefHeading___Toc392668357)  [Search and Replace](#__RefHeading___Toc392668357)  [36](#__RefHeading___Toc392668357)

[1.7.1](#__RefHeading___Toc392668358)  [Search](#__RefHeading___Toc392668358)  [37](#__RefHeading___Toc392668358)

[1.7.2](#__RefHeading___Toc392668359)  [Search using TFindDialog](#__RefHeading___Toc392668359)  [38](#__RefHeading___Toc392668359)

[1.7.3](#__RefHeading___Toc392668360)  [Replacement](#__RefHeading___Toc392668360)  [39](#__RefHeading___Toc392668360)

[1.7.4](#__RefHeading___Toc392668361)  [Replacement using TReplaceDialog](#__RefHeading___Toc392668361)  [41](#__RefHeading___Toc392668361)

[1.8](#__RefHeading___Toc392668362)  [Redialing options](#__RefHeading___Toc392668362)  [44](#__RefHeading___Toc392668362)

[1.8.1](#__RefHeading___Toc392668363)  [Highlighting a text.](#__RefHeading___Toc392668363)  [44](#__RefHeading___Toc392668363)

[1.8.2](#__RefHeading___Toc392668364)  [Highlighting the current word](#__RefHeading___Toc392668364)  [45](#__RefHeading___Toc392668364)

[1.8.3](#__RefHeading___Toc392668365)  [Highlighting current line](#__RefHeading___Toc392668365)  [46](#__RefHeading___Toc392668365)

[1.8.4](#__RefHeading___Toc392668366)  [Highlighting any line](#__RefHeading___Toc392668366)  [48](#__RefHeading___Toc392668366)

[1.8.5](#__RefHeading___Toc392668367)  [Text Markers](#__RefHeading___Toc392668367)  [49](#__RefHeading___Toc392668367)

[1.8.6](#__RefHeading___Toc392668368)  [More about bookmarks](#__RefHeading___Toc392668368)  [54](#__RefHeading___Toc392668368)

[1.9](#__RefHeading___Toc392668369)  [Summary of Properties and Methods](#__RefHeading___Toc392668369)  [61](#__RefHeading___Toc392668369)

[1.9.1](#__RefHeading___Toc392668370)  [Options and Options2 Property](#__RefHeading___Toc392668370)  [64](#__RefHeading___Toc392668370)

[2](#__RefHeading___Toc392668371)  [Syntax Highlighting and Autocomplete with SynEdit.](#__RefHeading___Toc392668371)  [66](#__RefHeading___Toc392668371)

[2.1](#__RefHeading___Toc392668372)  [Introduction](#__RefHeading___Toc392668372)  [66](#__RefHeading___Toc392668372)

[2.1.1](#__RefHeading___Toc392668373)  [Key Concepts](#__RefHeading___Toc392668373)  [67](#__RefHeading___Toc392668373)

[2.2](#__RefHeading___Toc392668374)  [Syntax Coloring Using Predefined Components](#__RefHeading___Toc392668374)  [69](#__RefHeading___Toc392668374)

[2.2.1](#__RefHeading___Toc392668375)  [Using a predefined language](#__RefHeading___Toc392668375)  [69](#__RefHeading___Toc392668375)

[2.2.2](#__RefHeading___Toc392668376)  [Using a custom language](#__RefHeading___Toc392668376)  [69](#__RefHeading___Toc392668376)

[2.3](#__RefHeading___Toc392668377)  [Syntax Coloring Using Code](#__RefHeading___Toc392668377)  [70](#__RefHeading___Toc392668377)

[2.3.1](#__RefHeading___Toc392668378)  [Syntax Highlighting Cases.](#__RefHeading___Toc392668378)  [71](#__RefHeading___Toc392668378)

[2.3.2](#__RefHeading___Toc392668379)  [Line Scan](#__RefHeading___Toc392668379)  [71](#__RefHeading___Toc392668379)

[2.3.3](#__RefHeading___Toc392668380)  [Getting started](#__RefHeading___Toc392668380)  [75](#__RefHeading___Toc392668380)

[2.3.4](#__RefHeading___Toc392668381)  [Adding functionality to the syntax](#__RefHeading___Toc392668381)  [78](#__RefHeading___Toc392668381)

[2.3.5](#__RefHeading___Toc392668382)  [GetDefaultAttribute](#__RefHeading___Toc392668382)  [90 property](#__RefHeading___Toc392668382)

[2.3.6](#__RefHeading___Toc392668383)  [Recognizing key words.](#__RefHeading___Toc392668383)  [92](#__RefHeading___Toc392668383)

[2.3.7](#__RefHeading___Toc392668384)  [Optimizing the syntax.](#__RefHeading___Toc392668384)  [93](#__RefHeading___Toc392668384)

[2.3.8](#__RefHeading___Toc392668385)  [Single Line Comment Coloring](#__RefHeading___Toc392668385)  [97](#__RefHeading___Toc392668385)

[2.3.9](#__RefHeading___Toc392668386)  [String Coloring](#__RefHeading___Toc392668386)  [99](#__RefHeading___Toc392668386)

[2.3.10](#__RefHeading___Toc392668387)  [Management of ranges](#__RefHeading___Toc392668387)  [100](#__RefHeading___Toc392668387)

[2.3.11](#__RefHeading___Toc392668388)  [Range or context coloring](#__RefHeading___Toc392668388)  [104](#__RefHeading___Toc392668388)

[2.4](#__RefHeading___Toc392668389)  [The TSynCustomHighlighter class](#__RefHeading___Toc392668389)  [109](#__RefHeading___Toc392668389)

[2.4.1](#__RefHeading___Toc392668390)  [CurrentLines[] and CurrentRanges[]](#__RefHeading___Toc392668390)  [109](#__RefHeading___Toc392668390)

[2.4.2](#__RefHeading___Toc392668391)  [Some Methods and Properties](#__RefHeading___Toc392668391)  [111](#__RefHeading___Toc392668391)

[2.4.3](#__RefHeading___Toc392668392)  [Attributes](#__RefHeading___Toc392668392)  [111](#__RefHeading___Toc392668392)

[2.5](#__RefHeading___Toc392668393)  [Code Folding Functionality](#__RefHeading___Toc392668393)  [116](#__RefHeading___Toc392668393)

[2.5.1](#__RefHeading___Toc392668394)  [Basic Folding](#__RefHeading___Toc392668394)  [116](#__RefHeading___Toc392668394)

[2.5.2](#__RefHeading___Toc392668395)  [Example code](#__RefHeading___Toc392668395)  [118](#__RefHeading___Toc392668395)

[2.5.3](#__RefHeading___Toc392668396)  [Improving Folding](#__RefHeading___Toc392668396)  [126](#__RefHeading___Toc392668396)

[2.5.4](#__RefHeading___Toc392668397)  [More about Folding](#__RefHeading___Toc392668397)  [128](#__RefHeading___Toc392668397)

[2.5.5](#__RefHeading___Toc392668398)  [Enable and disable folding blocks](#__RefHeading___Toc392668398)  [129](#__RefHeading___Toc392668398)

[2.6](#__RefHeading___Toc392668399)  [The TSynCustomFoldHighlighter class](#__RefHeading___Toc392668399)  [133](#__RefHeading___Toc392668399)

[2.6.1](#__RefHeading___Toc392668400)  [Low folding Level](#__RefHeading___Toc392668400)  [133](#__RefHeading___Toc392668400)

[2.6.2](#__RefHeading___Toc392668401)  [CurrentLines[] and CurrentRanges[]](#__RefHeading___Toc392668401)  [135](#__RefHeading___Toc392668401)

[2.6.3](#__RefHeading___Toc392668402)  [Some Methods and Properties](#__RefHeading___Toc392668402)  [139](#__RefHeading___Toc392668402)

[2.7](#__RefHeading___Toc392668403)  [Autocomplete](#__RefHeading___Toc392668403)  [140](#__RefHeading___Toc392668403)

[2.7.1](#__RefHeading___Toc392668404)  [Autocompletion using component](#__RefHeading___Toc392668404)  [141](#__RefHeading___Toc392668404)

[2.7.2](#__RefHeading___Toc392668405)  [Autocompletion using code](#__RefHeading___Toc392668405)  [143](#__RefHeading___Toc392668405)

[3](#__RefHeading___Toc392668406)  [APPENDIX](#__RefHeading___Toc392668406)  [145](#__RefHeading___Toc392668406)

[3.1](#__RefHeading___Toc392668407)  [Algorithms for highlighter implementation.](#__RefHeading___Toc392668407)  [145](#__RefHeading___Toc392668407)

[3.1.1](#__RefHeading___Toc392668408)  [Algorithm based on the use of Hash functions](#__RefHeading___Toc392668408)  [145](#__RefHeading___Toc392668408)

[3.1.2](#__RefHeading___Toc392668409)  [Algorithm based on the First character as a prefix.](#__RefHeading___Toc392668409)  [147](#__RefHeading___Toc392668409)

[3.1.3](#__RefHeading___Toc392668410)  [Comparison between algorithms](#__RefHeading___Toc392668410)  [148](#__RefHeading___Toc392668410)

[3.1.4](#__RefHeading___Toc392668411)  [Optimization criteria](#__RefHeading___Toc392668411)  [149](#__RefHeading___Toc392668411)

[INDEX](#__RefHeading___Toc392668412)  [152](#__RefHeading___Toc392668412)

1. This is not entirely true, because some oriental characters can occupy two editor cells (full width). [↑](#footnote-ref-1)
2. Initially in the first text editors, a 1 to 1 correspondence was always maintained (except when tabulations were supported), using ASCCI encoding or something similar. [↑](#footnote-ref-2)
3. To complicate matters, considering that a character on the screen can occupy two cells in the editor, in general, one or more string bytes can represent one or two cells on the screen. [↑](#footnote-ref-3)
4. Due to the inequality in character size in UTF-8, there are specific functions for handling strings in UTF-8, such as Utf8Length, and Utf8Pos. [↑](#footnote-ref-4)
5. There are editors that can handle more than one selection block, but the vast majority only work with a single selection block. [↑](#footnote-ref-5)
6. The possibility of converting a highlighter into a syntax analyzer is left to the programmer's freedom. The pre-defined highlighters in Lazarus only identify tokens and in some cases, block delimiters such as BEGIN ..END (for folding), but are not designed to serve as lexers. Doing them this way would involve more processing and result in slower analysis, which defeats the purpose of a highlighter. [↑](#footnote-ref-6)
7. You can also see that word recognition is not effective because it will recognize words even if they only match the first few characters. [↑](#footnote-ref-7)
8. In the design of “TSynCustomHighlighter”, the use of pointers has been defined, in order to be able to use references to real objects, which can be associated with a specific range. For most cases, it will be enough to handle a simple integer or enumeration. However, in certain developments, such as the “TSynCustomFoldHighlighter” class, objects are used for its functionality. The use of pointers to manage ranges is confusing at first, as the feeling remains that using a simple integer would have been enough, but the current design allows for greater freedom. [↑](#footnote-ref-8)
9. It may be given that meaning, in a particular implementation, but it is not required. [↑](#footnote-ref-9)
10. Although ways could be devised that allow the information inside the line to be reconstructed in a range object. [↑](#footnote-ref-10)
11. Only within a line, tokens are scanned, always sequentially until the end of the line. [↑](#footnote-ref-11)
12. It should be clarified that CurrentRanges[], is declared as PROTECTED, so it is not directly accessible from outside the class, but it can be made accessible when creating our own highlighter. [↑](#footnote-ref-12)
13. This value is obtained by creating a table that assigns a value to each letter of the English alphabet (usually in the mHashTable[], and taking advantage of it to fill it in the “MakeIdentTable” function). With this table, the value corresponding to each keyword is calculated, adding the value of each letter in the identifier. The value obtained usually depends on the letters of the identifier and its size, but usually does not exceed 200 in normal syntax. This value is not unique for each word, as several different words can share the same value, but it allows you to effectively categorize identifiers to restrict the search to a much smaller group. [↑](#footnote-ref-13)