sgcRichEdit

Getting Started

Put a small word processor on a Delphi form, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want to add a rich text editor to it with sgcRichEdit. You do not need to know anything about RTF, HTML or document formats in advance. If you have used a word processor, you already know what the editor does; this guide shows you how to drive it from Delphi.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from an editor that shows a formatted document loaded from one line of code to a small mail merge application with a template, PDF export and print preview. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, type and format some text, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcRichEdit is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\RichEdit\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloRichEdit` to `05.RichLetterApp`, with a Delphi project (`.dpr`) that you can open and run directly. All of them are VCL applications for Windows.

The screenshots were taken on a machine with Spanish regional settings. The demos themselves contain fixed English text, but wherever a program shows numbers or dates formatted by Delphi, they follow the Windows regional settings of the machine that runs it.

Contents

1. What sgcRichEdit is

The idea in one minute
Your first editor needs one line of code
What you need
Part of sgcUI
Coming from TRichEdit

2. Install and the palette

Installing
The packages
The palette page
The standard actions
Help at design time
Editions
Where the demos are
Running your first demo

3. Core concepts

The editor and its document
Formats are units
Loading and saving
Commands
Command state and OnSelectionChange
Standard actions
Points and ARGB colours
Look: theme, zoom, width and read only
Printing and export
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello RichEdit
4.2 Demo 02: Rich Formatting
4.3 Demo 03: Rich Tables
4.4 Demo 04: Rich Formats
4.5 Demo 05: Rich Letter App

5. Main components

Reference table
The editor
The data aware editor
The actions
Documents without a control
FireMonkey

6. Common problems

Formats and export
Formatting from code
Layout and fonts

7. Where next

The other Delphi demos
Help topics
On the website
Getting help

1. What sgcRichEdit is

sgcRichEdit is a rich text editor for Delphi applications, for the VCL and for FireMonkey. It holds a document made of paragraphs, headings, lists, tables, pictures and links, paints it and lets the user edit it with the keyboard and the mouse, like a small word processor inside your form. Undo and redo, the clipboard and the usual shortcuts work without a line of code.

You drop a `TsgcVCLRichEdit` on a form, load a document into it from HTML, Markdown, RTF, plain text or its own JSON format, and save it back in any of those formats. From code you format the selection with commands, such as bold, a font size or a new table, and ask the editor about the text under the caret to keep your toolbar in sync. When the document is ready you can print it, preview it, or export it to DOCX, PDF or PNG.

This guide is for Delphi developers who are new to sgcRichEdit. By the end you will know how the pieces fit together, which properties and methods to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

A rich text editor has three jobs, and sgcRichEdit keeps them apart:

- **The document.** A tree of blocks: paragraphs with a style such as Heading 1 or Quote, lists, tables with cells, pictures and links. Every piece of text carries its character format: font, size, colour, bold, italic and so on. The document lives in the `Document` property of the editor.
- **The editing.** Typing, deleting, the selection, undo and redo, and about fifty *commands* such as `recBold`, `recFontSize`, `recInsertTable` or `recStyle`. The same commands drive the keyboard shortcuts, the standard actions and your own buttons.
- **The formats.** Readers and writers that turn the document into text and back: the native JSON, HTML, Markdown, RTF and plain text, plus DOCX, PDF and PNG for export.

Loading a document is one call, `LoadFromString` or `LoadFromFile`. Saving is one call, `SaveToString` or `SaveToFile`. Formatting is one call, `ExecuteCommand`. That is the core of the whole product; the rest of this guide fills in the details.

Your first editor needs one line of code

The first QuickStart demo has a single `TsgcVCLRichEdit` on the form, aligned to the client area. The only code loads the startup document from an HTML string:

```
uses
  Classes, Controls, Forms,
  // sgc
  sgcRichEdit_Formats, sgcRichEdit_VCL, sgcRichEdit_HTML;
```

```
procedure TfrmHelloRichEdit.FormCreate(Sender: TObject);
begin
  { refHTML tells the editor which reader to use. refAuto (the default)
    detects the format from the text itself. }
  Editor.LoadFromString(
    '<h1>Hello, sgcRichEdit</h1>' +
    '<p>This is a <b>rich text editor</b> for Delphi. The text can be ' +
    '<b>bold</b>, <i>italic</i>, <u>underlined</u>, ' +
    '<span style="color:#c0392b">red</span>, ' +
    '<span style="background-color:#fff59d">highlighted</span> or ' +
    '<span style="font-size:16pt">larger</span>.</p>' +
    '<h2>Try it now</h2>' +
    // ...
    '<p style="text-align:center"><i>The whole document was loaded from ' +
    'one HTML string with LoadFromString.</i></p>', refHTML);
  { a freshly loaded document is not a change made by the user }
  Editor.Modified := False;
end;
```

Run it and you have a working editor: click into the text and type, select a word and press Ctrl+B, press Ctrl+Z to undo. The headings, the coloured words and the bullet list all come from ordinary HTML tags. The unit `sgcRichEdit_HTML` in the `uses` clause is what teaches the editor to read HTML; you will meet that idea again in chapter 3.

What you need

- **Delphi 7 or later.** There are packages for Delphi 7, 2007, 2009, 2010, XE to XE8, 10 Seattle to 10.4 Sydney, 11, 12 and 13.
- **VCL:** `TsgcVCLRichEdit` and `TsgcVCLDBRichEdit`, for Windows, in every supported Delphi version. The QuickStart demos are VCL applications and compile for Win32 and Win64.
- **FireMonkey:** `TsgcFMXRichEdit` and `TsgcFMXDBRichEdit`, with their own packages, for Delphi XE6 and later. This guide uses the VCL, but the FireMonkey controls edit the same document with the same commands and formats.
- **sgcUICore**, the shared base of every sgcUI product, which the installer adds for you, and **sgcPDF** for the PDF export, which the installer also adds when you select sgcRichEdit.

Part of sgcUI

sgcRichEdit is one of the sgcUI products, and it shares their foundations:

- **Themes.** The `Theme` property takes the sgcUICore themes, `utDefault`, `utLight`, `utDark`, `utCustom` and `utHighContrast`, so the editor matches the other sgcUI controls of your application. `utDefault` follows the application theme.
- **Printing.** Print, the print preview and the page settings come from sgcUICore, the same as for the other sgcUI controls.
- **PDF.** The PDF export is drawn by sgcPDF through the sgcUIPDF bridge, the same bridge every sgcUI control uses.
- **Other products build on it.** sgcChat uses sgcRichEdit, and the `Editor` demo of chapter 7 puts the editor under the sgcUI ribbon control.

Coming from TRichEdit

If you have used the `TRichEdit` of the VCL, the biggest difference is that sgcRichEdit does not wrap a Windows control. It has its own document model, its own layout and its own painting, which is why the VCL and FireMonkey controls behave the same and why a document looks the same on the screen, on the printer and in the PDF.

In practice that means three things for your code. You load and save documents in several formats, not only RTF. You format text through commands and read the state of the selection through `CommandState`, instead of setting `SelAttributes`. And tables and pictures are part of the document: the user can click into a table cell and add rows and columns, and your code can do the same with a command.

2. Install and the palette

This chapter gets sgcRichEdit onto your Tool Palette and the first demo running. If the components are already installed, skim the palette section and jump to Running your first demo.

Installing

sgcRichEdit is installed by the sgcUI installer, the same installer for every sgcUI product:

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, select **sgcRichEdit (installs sgcPDF)**. The installer shows one checkbox per product and enables the products your licence includes. Selecting sgcRichEdit also selects sgcPDF, the PDF engine it needs for `ExportToPDF`, and sgcUICore is installed with it.
4. The installer copies the files, compiles the packages on your machine for each selected Delphi version and installs the design time packages.

Everything goes into one folder, by default `Documents\esegece\sgcUI`. The installer adds its `Source` folder and the folder of the compiled units to the library path of each selected Delphi version, so a new project finds the sgcRichEdit units without any setting on your side.

The packages

You only need this section if you build the packages yourself. sgcRichEdit is split into a framework neutral package and one package per framework, so a FireMonkey application never links the VCL and the other way round. Every package name ends with the suffix of the Delphi version, for example `D13` for Delphi 13 or `D7` for Delphi 7.

Package	Kind	Contains
`sgcRichEdit<V>`	Runtime	The core: document model, editing, commands, layout, renderer, the formats, printing and accessibility. Requires `sgcUICore`. No VCL, no FireMonkey.
`sgcRichEditVCL<V>`	Runtime	`TsgcVCLRichEdit`, `TsgcVCLDBRichEdit` and the VCL actions.
`sgcRichEditFMX<V>`	Runtime, XE6 and later	`TsgcFMXRichEdit`, `TsgcFMXDBRichEdit` and the FireMonkey actions.
`dclsgcRichEdit<V>`	Design time	Registers the VCL controls on the palette, the standard actions and the component editor.
`dclsgcRichEditFMX<V>`	Design time, XE6 and later	Registers the FireMonkey controls and actions.

The packages do not depend on sgcPDF. The PDF export is linked into your application only when you add the unit `sgcUIPDF_Export` to a `uses` clause, as chapter 3 explains.

The palette page

After the installation the Tool Palette has a page named **SGC RichEdit**. In a VCL project it shows two components:

Component	What it does
`TsgcVCLRichEdit`	The rich text editor. Every QuickStart demo uses it.
`TsgcVCLDBRichEdit`	The same editor bound to a field of a dataset through `DataSource` and `DataField`. It is not used in the QuickStart; the `DataAware` demo of chapter 7 shows it.

In a FireMonkey project the same page shows `TsgcFMXRichEdit` and `TsgcFMXDBRichEdit` instead.

The standard actions

sgcRichEdit also registers standard actions, which you find in the action list editor rather than on the palette. Double click a `TActionList`, open the drop down of the New Action button, choose **New Standard Action...** and look for the **SGC RichEdit** category:

Group	Actions
Editing	`TsgcRichEditCut`, `TsgcRichEditCopy`, `TsgcRichEditPaste`, `TsgcRichEditPastePlainText`, `TsgcRichEditUndo`, `TsgcRichEditRedo`, `TsgcRichEditSelectAll`, `TsgcRichEditFind`
Character format	`TsgcRichEditBold`, `TsgcRichEditItalic`, `TsgcRichEditUnderline`
Paragraph format	`TsgcRichEditAlignLeft`, `TsgcRichEditAlignCenter`, `TsgcRichEditAlignRight`, `TsgcRichEditAlignJustify`, `TsgcRichEditBullets`, `TsgcRichEditNumbering`, `TsgcRichEditIndent`, `TsgcRichEditOutdent`
Insert	`TsgcRichEditInsertTable`, `TsgcRichEditInsertLink`, `TsgcRichEditInsertImage`
Any command	`TsgcRichEditCommandAction`, which runs the command chosen in its `Command` property

Demo 02 builds its whole formatting toolbar from these actions.

Help at design time

Right click a `TsgcVCLRichEdit` on the form to find its component editor:

- **Edit document...** opens an editor where you type and format the startup document at design time. A double click on the component does the same. The document is stored in the form file, so the editor shows it at run time without any code.
- **Load from file...** loads a document from a file into the component at design time.
- **Clear** empties the design time document.

The QuickStart demos load their documents in code instead, so you can read the content in the source, but for a fixed text, such as a help page or a welcome note, the design time document saves you a long string.

Editions

sgcRichEdit is a Registered product. You can buy it as a single product, and it is included in sgcUI Enterprise and in the All-Access bundle. There is no free Community edition. Every licence includes the complete product: the VCL and FireMonkey controls and their data aware versions, all the file formats, printing, print preview and the DOCX, PDF and PNG export. The trial, part of the sgcUI trial installer, is for evaluation.

Where the demos are

The demos live in `Demos\RichEdit\Delphi` under your sgcUI folder. The QuickStart is the first subfolder:

Folder	Project	What it shows
01.HelloRichEdit	`HelloRichEdit.dpr`	The smallest editor: a formatted document loaded from one HTML string
02.RichFormatting	`RichFormatting.dpr`	A formatting toolbar built from standard actions, plus font, size and colour from code
03.RichTables	`RichTables.dpr`	Paragraph styles, table commands and a picture inserted from code
04.RichFormats	`RichFormats.dpr`	Every text format, a file round trip, DOCX and PDF export
05.RichLetterApp	`RichLetterApp.dpr`	A small mail merge application with a template, themes, PDF and print preview

Every demo folder holds the project, its unit and form, and a `readme.html` with a short description. The `readme.html` in `00.QuickStart` itself is the index of the five demos. The other folders in `Demos\RichEdit\Delphi` are larger demos, listed in chapter 7.

Running your first demo

1. In Delphi, choose **File** > **Open Project** and open `Demos\RichEdit\Delphi\00.QuickStart\01.HelloRichEdit\HelloRichEdit.dpr`.
2. Press **F9**. The window opens with a formatted document.
3. Click into the text and type. Select a word and press Ctrl+B. Press Ctrl+Z to undo it.
4. Back in the IDE, open the form and select `Editor`. Only `Align` and `BorderStyle` differ from their defaults; everything you saw comes from the HTML string in `FormCreate`.

> **Tip.** To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this guide come from that view. Choose **View as Form** to go back to the designer.

3. Core concepts

This chapter explains the ideas you need before the demos make full sense. Each one is short, and each one comes back in chapter 4. If you prefer to learn by doing, run the demos first and come back here when a section of chapter 4 sends you.

The editor and its document

`TsgcVCLRichEdit` is the control you put on the form. It owns a document, published in code as the `Document` property (a `TsgcRichDocument`). The document holds the paragraphs, tables, pictures, links, lists, the paragraph and character styles and the page settings. The control paints the document as one continuous page, turns keyboard and mouse input into edits, and keeps the undo history.

Most of the time you never touch `Document` directly. You load and save it through the methods of the control, you change it through commands, and you read it back through the methods of the control again. The exception in the QuickStart is the image store, `Document.Images`, which demo 03 uses to add a picture.

Three members of the control tell you about the content:

- `Modified` is `True` when the document changed. Set it to `False` after a load or a save, as every demo does.
- `PlainText` (also available as `Text`) is the document without formatting.
- `SelText` is the selected text.

Two events tell you that something happened: `OnChange` fires after every edit of the document, and `OnSelectionChange` fires every time the caret moves or the selection changes.

Formats are units

This is the idea that surprises most people on their first day. In Delphi, a file format is available only when its unit is linked into your application. Each format unit registers its reader or its writer when the application starts, so adding the unit to a `uses` clause is all it takes, and leaving it out means the format is missing:

Unit	Registers
(always available)	The native JSON format (`refJSON`) and plain text (`refText`)
`sgcRichEdit_HTML`	The HTML reader
`sgcRichEdit_HTML_Writer`	The HTML writer
`sgcRichEdit_Markdown`	The Markdown reader and writer
`sgcRichEdit_RTF_Reader`	The RTF reader
`sgcRichEdit_RTF_Writer`	The RTF writer
`sgcRichEdit_DOCX`	The DOCX writer (export only)
`sgcUIPDF_Export`	The PDF writer used by `ExportToPDF` (export only)

The formats are values of `TsgcRichEditFormat`, declared in `sgcRichEdit_Formats`: `refAuto`, `refJSON`, `refHTML`, `refMarkdown`, `refRTF`, `refText`, `refDOCX` and `refPDF`. A format whose unit is missing raises an exception with the message "Import of ... is not supported" or "Export to ... is not supported".

Which format should you store your documents in? The native JSON format keeps everything in the document exactly: styles, lists, pictures, links and tables. It is the format the control uses for the design time document in the form file, and it is the right choice whenever your own application reads the document back, as in demo 05. Use HTML, Markdown, RTF or DOCX when another program has to read the document.

> **Note.** Two formats are never detected by their content. Markdown is recognised only from a `.md` or `.markdown` file extension or an explicit `refMarkdown`; a Markdown string loaded with `refAuto` is read as plain text. DOCX and PDF are export only, so they cannot be loaded at all.

Loading and saving

The control has three loaders and three savers, and a property per text format:

Member	What it does
`LoadFromString(Text, Format)`	Replaces the document with the text. Returns the format that was read.
`LoadFromFile(FileName, Format)`, `LoadFromStream`	The same from a file or a stream. With `refAuto`, the default, the format is detected from the content and, for a file, from the extension.
`SaveToString(Format)`	Writes the document in the format. The default is `refJSON`.
`SaveToFile(FileName, Format)`, `SaveToStream`	The same to a file or a stream. For a file, `refAuto` takes the format from the extension.
`AsJSON`, `AsHTML`, `AsRTF`, `AsMarkdown`, `AsText`	Read the document in that format, or replace it when you assign a value.

Here are the members together, with the format units they need in the `uses` clause:

```
uses
  SysUtils, Classes, StdCtrls,
  // sgc
  sgcUICore_Theme, sgcRichEdit_Commands, sgcRichEdit_Formats,
  sgcRichEdit_VCL, sgcRichEdit_HTML, sgcRichEdit_HTML_Writer,
  sgcRichEdit_Markdown, sgcRichEdit_RTF_Reader, sgcRichEdit_RTF_Writer,
  sgcRichEdit_DOCX,
  sgcUIPDF_Export; // the PDF writer used by ExportToPDF

procedure ConvertNotes(aEdit: TsgcVCLRichEdit; aMemo: TMemo);
var
  vFormat: TsgcRichEditFormat;
begin
  vFormat := aEdit.LoadFromFile('notes.md'); // Markdown, from the extension
  if vFormat = refMarkdown then
    aEdit.SaveToFile('notes.rtf', refRTF);
  aMemo.Text := aEdit.AsHTML;                // the same document as HTML
  aEdit.AsMarkdown := '# A new document' + #13#10#13#10 + 'Written in **Markdown**.';
end;
```

The `uses` clause above is shared by the other snippets of this chapter, which is why it also lists the theme, command and PDF units.

Commands

Everything the user can do to the format of the text is a command, a value of `TsgcRichEditCommand` in the unit `sgcRichEdit_Commands`. There are about fifty of them, in a few families:

Family	Commands
Character format	`recBold`, `recItalic`, `recUnderline`, `recStrikeout`, `recSuperscript`, `recSubscript`, `recFontName`, `recFontSize`, `recFontColor`, `recHighlight`, `recClearFormatting`
Paragraph format	`recAlignLeft`, `recAlignCenter`, `recAlignRight`, `recAlignJustify`, `recLeftToRight`, `recRightToLeft`, `recIndent`, `recOutdent`, `recLineSpacing`, `recStyle`
Lists	`recBullets`, `recNumbering`, `recListLevelUp`, `recListLevelDown`
Tables	`recInsertTable`, `recInsertRowAbove`, `recInsertRowBelow`, `recInsertColumnBefore`, `recInsertColumnAfter`, `recDeleteRow`, `recDeleteColumn`, `recDeleteTable`, `recMergeCells`, `recSplitCell`, `recCellProperties`
Insert	`recInsertImage`, `recInsertLink`, `recRemoveLink`, `recPageBreak`
Editing	`recUndo`, `recRedo`, `recCut`, `recCopy`, `recPaste`, `recPastePlainText`, `recSelectAll`, `recFind`, `recReplace`, `recReplaceAll`

A command works on the selection, or on the paragraph or table under the caret. You run one with `ExecuteCommand`. Commands that are a simple switch, such as bold or a new table row, need nothing else:

```
procedure MakeBold(aEdit: TsgcVCLRichEdit);
begin
  { a command without a value: like pressing Ctrl+B }
  if aEdit.CommandState(recBold).Enabled then
    aEdit.ExecuteCommand(recBold);
  { Checked is True while the selection is already bold }
  if aEdit.CommandState(recBold).Checked then
    aEdit.Hint := 'The selection is bold';
end;
```

Commands that need a value, such as a font name, a size, a colour or the number of rows of a new table, take a second parameter, a `TsgcRichCommandParam` record. Clear it with `sgcRichCommandParamClear` first, then fill the field the command reads:

Field	Used for
`Text`	Font name, style name, link URL, find text, the alternative text of a picture
`Text2`	Link title, replace text
`Value`, `Value2`	Font size; line spacing; picture width and height, all in points
`Int1`, `Int2`	Table rows and columns; the id of a picture in `Document.Images`
`Color`	Font colour and highlight, as an ARGB colour

Demo 02 runs `recFontName`, `recFontSize`, `recFontColor` and `recHighlight` this way, and demo 03 runs `recStyle`, `recInsertTable` and `recInsertImage`.

Command state and OnSelectionChange

A toolbar should show what is under the caret: the Bold button pressed inside bold text, the font box showing the current font, the table buttons disabled outside a table. You get that information from `CommandState`, which returns a `TsgcRichCommandState` record for one command:

Field	Meaning
`Enabled`	The command can run now. The table commands, for example, are enabled only with the caret in a table.
`Checked`	The selection already has the format, for example bold text for `recBold`. It is `False` when `Mixed` is `True`.
`Mixed`	The selection holds different values, for example two fonts. The other fields then hold the first value.
`Text`, `Value`, `Color`	The current value: the font name or style name, the font size, the colour.

Read it in the `OnSelectionChange` event, which fires every time the caret moves or the selection changes. Demo 02 updates its font and size boxes there, and demo 03 its style box and table buttons.

Standard actions

For the common commands you do not even need `ExecuteCommand`. The standard actions of chapter 2 each know their command. Put them in a `TActionList`, set their `Control` property to the editor, and connect them to buttons or menu items. An action then does three things on its own: it runs its command when clicked, it disables itself when the command is not available (Undo with nothing to undo), and it is `Checked` while the selection already has the format. When `Control` is empty, the action works on the rich edit that has the focus, which is handy for a form with several editors.

Points and ARGB colours

Two units of measure are easy to get wrong:

- **Sizes are in points**, as in a word processor: the font size, the size of a picture and `DocumentWidth`. A point is 1/72 inch, so a 96 dpi screen pixel is 0.75 points. Demo 03 inserts a 320 x 90 pixel picture as 240 x 67.5 points.
- **Colours are ARGB**, written `$AARRGGBB`, with `$FF` as the alpha of an opaque colour. Red is `$FFC0392B`, not the VCL `TColor` `$002B39C0`. A `TColor` such as `clRed` does not give red here, because its bytes are read in a different order.

Look: theme, zoom, width and read only

A handful of published properties change how the editor looks and behaves:

Property	What it does
`Theme`	The sgcUICore theme: `utDefault` (follows the application theme), `utLight`, `utDark`, `utCustom` or `utHighContrast`. The type lives in the unit `sgcUICore_Theme`.
`Zoom`	The zoom factor, from 0.5 to 4. The user can also zoom with Ctrl and the mouse wheel.
`DocumentWidth`	The width of the text in points. With 0 the text wraps at the width of the control; with a value it wraps like a page of that width.
`ReadOnly`	Turns the editor into a viewer: the user can read, select and click links, but cannot change the document.
`BorderStyle`, `Color`	The border and the background, as for any VCL control.

```
procedure SetupViewer(aEdit: TsgcVCLRichEdit);
begin
  aEdit.ReadOnly := True;   // a viewer: the user cannot change the text
  aEdit.Zoom := 1.25;       // 125 %
  aEdit.Theme := utDark;    // the sgcUICore dark theme
  aEdit.DocumentWidth := 453.5; // wrap the text at 453.5 points
end;
```

Printing and export

On the screen the document is one long page. Printing, the print preview and the PDF and PNG export cut it into pages, using the page settings of the document (`Document.PageSettings`: paper, orientation and margins). The control has one method for each:

Method	What it does
`Print(PrinterName)`	Prints without a dialog on the named printer, or the default printer when the name is empty, and returns the number of pages.
`PrintPreview(Caption)`	Shows the sgcUI print preview.
`ExportToPDF(FileName)`	Writes a PDF file, tagged for screen readers by default. Needs `sgcUIPDF_Export` in a `uses` clause.
`ExportToPNG(FileName, Page, DPI)`	Writes one page (0 based) as a PNG image and returns the number of pages.
`SaveToFile(FileName, refDOCX)`	Writes a Word document. Needs `sgcRichEdit_DOCX`.

```
procedure ExportAll(aEdit: TsgcVCLRichEdit);
var
  vPages: Integer;
begin
  aEdit.SaveToFile('report.docx', refDOCX);       // Word, export only
  aEdit.ExportToPDF('report.pdf');                // needs sgcUIPDF_Export
  vPages := aEdit.ExportToPNG('report_page1.png', 0, 150); // page 1 at 150 dpi
  aEdit.Hint := IntToStr(vPages) + ' pages';
end;
```

> **Important.** `ExportToPDF` raises `EsgcUIPDFExportUnavailable` when the unit `sgcUIPDF_Export` is not linked into the application. The sgcRichEdit units do not link it for you, so add it to a `uses` clause of every project that exports PDF.

Summary: who does what

You want to	Use
Show a document	`LoadFromString`, `LoadFromFile`, or a design time document
Support a format	Its unit in the `uses` clause
Store a document for your own application	`SaveToString(refJSON)` or `AsJSON`
Hand a document to another program	`SaveToFile` with HTML, Markdown, RTF or DOCX
Add a formatting toolbar	The standard actions, with `Control` set to the editor
Format text from code	`ExecuteCommand`, with a `TsgcRichCommandParam` for values
Mirror the selection in your UI	`OnSelectionChange` and `CommandState`
React to edits	`OnChange` and `Modified`
Print, preview or export	`Print`, `PrintPreview`, `ExportToPDF` (with `sgcUIPDF_Export`), `ExportToPNG`

4. The QuickStart demos

The five QuickStart demos build on each other. Open them in order: each section below says what you will build, which components are on the form, shows the code, explains how it works and suggests a few changes to try.

Demo	Project	New ideas
01.HelloRichEdit	`HelloRichEdit.dpr`	The editor, a document loaded from HTML, format units
02.RichFormatting	`RichFormatting.dpr`	Standard actions, commands with values, `OnSelectionChange` and `CommandState`, ARGB colours
03.RichTables	`RichTables.dpr`	Paragraph styles, table commands, the image store, `CommandState.Enabled`, find
04.RichFormats	`RichFormats.dpr`	Every text format, `SaveToString`, a file round trip, DOCX and PDF export
05.RichLetterApp	`RichLetterApp.dpr`	Native JSON as a template, `ReplaceAll`, `DocumentWidth`, themes, PDF and print preview

> **Note.** The demos need sgcRichEdit installed, so the components appear on the SGC RichEdit palette page and the `sgcUI\Source` folder is on the library path. Each demo writes its files, when it writes any, into the folder of its executable.

4.1 Demo 01: Hello RichEdit

Folder `01.HelloRichEdit`, project `HelloRichEdit.dpr`.

What you will build

The smallest sgcRichEdit application: a window filled by the editor, showing a formatted document with two headings, bold, italic, underlined, coloured, highlighted and larger words, a bullet list and a centred paragraph. Everything is editable at once.

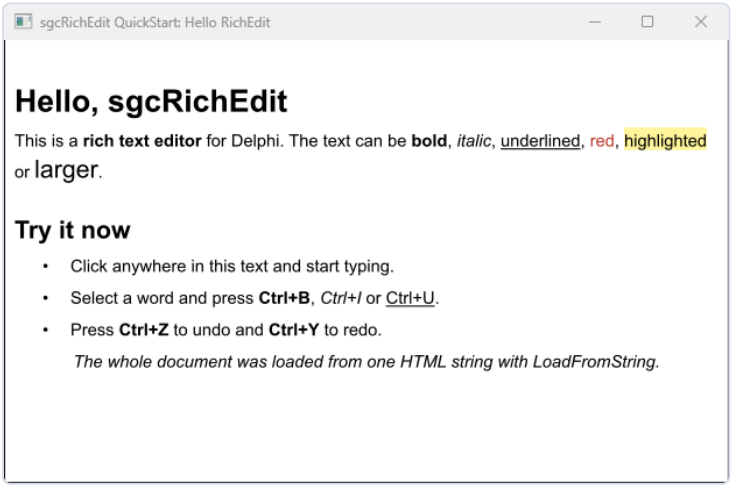


Figure 4.1. Demo 01. The whole document comes from one HTML string.

Components on the form

Component	Class	Properties you set
`Editor`	`TsgcVCLRichEdit`	`Align = alClient`, `BorderStyle = bsNone`. Everything else keeps its default value.

```
  object Editor: TsgcVCLRichEdit
    Left = 0
    Top = 0
    Width = 620
    Height = 380
    Align = alClient
    BorderStyle = bsNone
    TabOrder = 0
  end
```

The code

The `uses` clause names the unit of the format types, the unit of the control and the HTML reader:

```
uses
  Classes, Controls, Forms,
  // sgc
  sgcRichEdit_Formats, sgcRichEdit_VCL, sgcRichEdit_HTML;
```

And this is the only event handler of the demo:

```
procedure TfrmHelloRichEdit.FormCreate(Sender: TObject);
begin
  { refHTML tells the editor which reader to use. refAuto (the default)
    detects the format from the text itself. }
  Editor.LoadFromString(
    '<h1>Hello, sgcRichEdit</h1>' +
    '<p>This is a <b>rich text editor</b> for Delphi. The text can be ' +
    '<b>bold</b>, <i>italic</i>, <u>underlined</u>, ' +
    '<span style="color:#c0392b">red</span>, ' +
    '<span style="background-color:#fff59d">highlighted</span> or ' +
    '<span style="font-size:16pt">larger</span>.</p>' +
    '<h2>Try it now</h2>' +
    '<ul>' +
    '<li>Click anywhere in this text and start typing.</li>' +
    '<li>Select a word and press <b>Ctrl+B</b>, <i>Ctrl+I</i> or ' +
    '<u>Ctrl+U</u>.</li>' +
    '<li>Press <b>Ctrl+Z</b> to undo and <b>Ctrl+Y</b> to redo.</li>' +
    '</ul>' +
    '<p style="text-align:center"><i>The whole document was loaded from ' +
    'one HTML string with LoadFromString.</i></p>', refHTML);
  { a freshly loaded document is not a change made by the user }
  Editor.Modified := False;
end;
```

How it works

1. When the application starts, the unit `sgcRichEdit_HTML` registers its HTML reader. Nothing in the demo calls it directly; being in the `uses` clause is enough.
2. The form loads from its DFM and the editor starts with an empty document.

3. `FormCreate` calls `LoadFromString` with the HTML text and `refHTML`. The editor picks the HTML reader, builds the document from the tags (`<h1>` and `<h2>` become the Heading 1 and Heading 2 styles, `<ul>` a bullet list, the `style` attributes colours, highlight, size and alignment) and paints it.
4. `Modified := False` marks the loaded document as unchanged, so a later check of `Modified` reports only the edits of the user.
5. From now on the editor handles the keyboard, the mouse, the clipboard and the undo history on its own.

Try this

- Load Markdown instead of HTML: add `sgcRichEdit_Markdown` to the `uses` clause and call `Editor.LoadFromString('# Hello' + #13#10#13#10 + 'Some **bold** text', refMarkdown)`.
- Set `ReadOnly = True` in the Object Inspector: the same window becomes a document viewer.
- Set `Zoom = 1.5`, or `Theme = utDark`, and run again.
- Remove `sgcRichEdit_HTML` from the `uses` clause and run: `LoadFromString` now fails with “Import of ... is not supported”. Put it back.

4.2 Demo 02: Rich Formatting

Folder `02.RichFormatting`, project `RichFormatting.dpr`.

What you will build

A formatting toolbar above the editor. The first row holds Undo, Redo, Bold, Italic, Underline, the three alignments, Bullets, Numbering, Outdent and Indent, all standard actions without a line of code. The second row holds a font box, a size box and two buttons that colour the selection from code. Move the caret through the document and watch the toolbar follow: Bold is pressed inside bold text, the alignment buttons show the alignment of the paragraph, and the font and size boxes show the format under the caret.

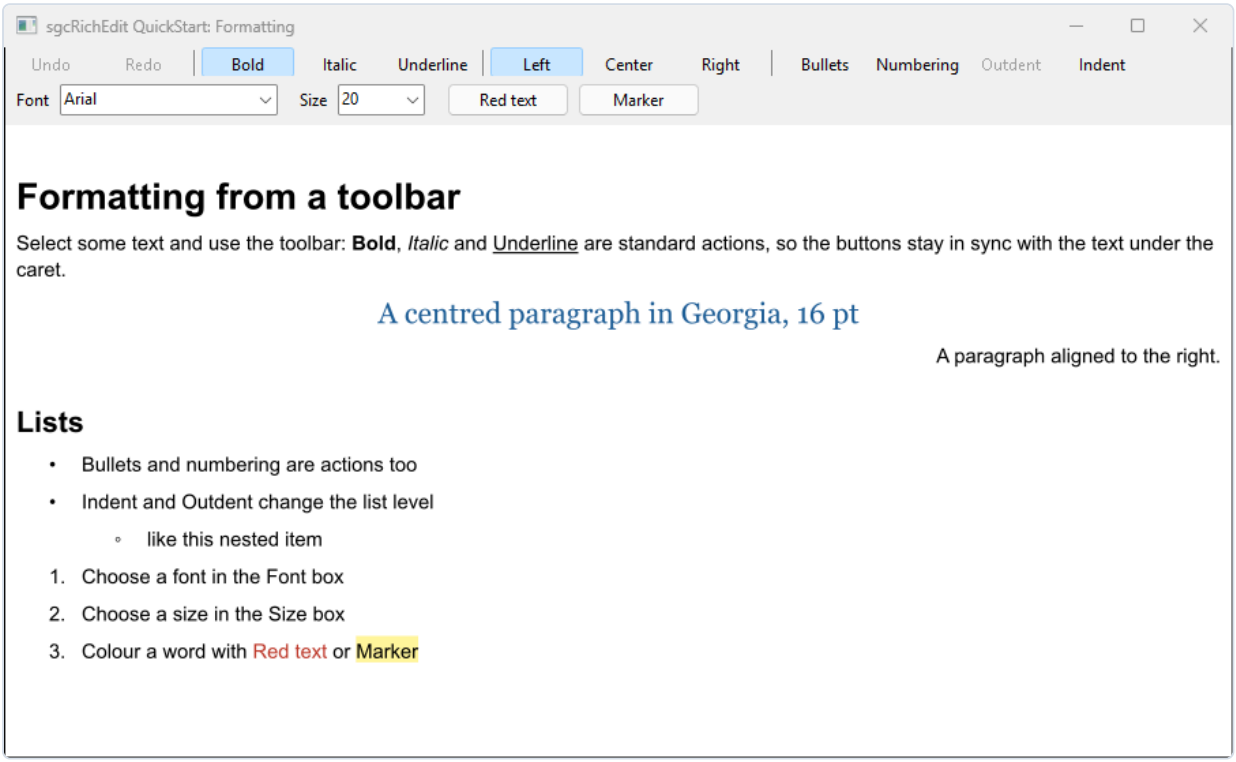


Figure 4.2. Demo 02 at startup. The caret is in the heading, so Bold and Left are pressed and the boxes show Arial 20. Undo, Redo and Outdent are disabled because they cannot run yet.

Components on the form

Component	Class	Properties you set
`Actions`	`TActionList`	Twelve standard actions from the SGC RichEdit category: `actUndo` (`TsgcRichEditUndo`), `actRedo`, `actBold`, `actItalic`, `actUnderline`, `actAlignLeft`, `actAlignCenter`, `actAlignRight`, `actBullets`, `actNumbering`, `actOutdent`, `actIndent`. Each has `Control = Editor` and a `Caption`.
`ToolBar`	`TToolBar`	`ShowCaptions = True`. One `TToolButton` per action (`Action` property); the toggles have `Style = tbsCheck`.
`cmbFont`, `cmbSize`	`TComboBox`	`Style = csDropDown`, a list of font names and sizes, `OnSelect`.
`btnRed`, `btnMarker`	`TButton`	`OnClick`.
`Editor`	`TsgcVCLRichEdit`	`Align = alClient`, `BorderStyle = bsNone`, `OnSelectionChange = EditorSelectionChange`.

An action and its button need only these lines in the form file:

```
  object actBold: TsgcRichEditBold
    Caption = 'Bold'
    Control = Editor
  end
  ...
  object btnBold: TToolButton
    Left = 136
    Top = 0
    Action = actBold
    Style = tbsCheck
  end
```

The code

The `uses` clause adds the commands unit and the actions unit:

```
uses
  SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls, ComCtrls, ToolWin,
  ActnList,
  // sgc
  sgcRichEdit_Commands, sgcRichEdit_Formats, sgcRichEdit_VCL,
  sgcRichEdit_VCL_Actions, sgcRichEdit_HTML;
```

The font and size boxes run a command with a value. The record is cleared first, then the field the command reads is filled:

```
procedure TfrmRichFormatting.cmbFontSelect(Sender: TObject);
var
  vParam: TsgcRichCommandParam;
begin
  if cmbFont.Text = '' then
    Exit;
  { always clear the record first, then fill the field the command reads }
  sgcRichCommandParamClear(vParam);
  vParam.Text := cmbFont.Text;
  Editor.ExecuteCommand(recFontName, vParam);
  Editor.SetFocus;
end;

procedure TfrmRichFormatting.cmbSizeSelect(Sender: TObject);
var
  vParam: TsgcRichCommandParam;
begin
  sgcRichCommandParamClear(vParam);
  vParam.Value := StrToIntDef(cmbSize.Text, 0);
  if vParam.Value <= 0 then
    Exit;
  Editor.ExecuteCommand(recFontSize, vParam);
  Editor.SetFocus;
end;
```

The two colour buttons share one helper. The colours are ARGB values:

```
procedure TfrmRichFormatting.DoApplyColor(aCommand: TsgcRichEditCommand;
  aColor: Cardinal);
var
  vParam: TsgcRichCommandParam;
begin
  sgcRichCommandParamClear(vParam);
  vParam.Color := aColor;
  Editor.ExecuteCommand(aCommand, vParam);
  Editor.SetFocus;
end;

procedure TfrmRichFormatting.btnRedClick(Sender: TObject);
begin
  { the font colour of the selection: opaque red, $FF + RRGGBB }
  DoApplyColor(recFontColor, $FFC0392B);
end;

procedure TfrmRichFormatting.btnMarkerClick(Sender: TObject);
begin
  { the background of the selection, like a yellow marker pen }
  DoApplyColor(recHighlight, $FFFFF59D);
end;
```

Finally, the selection event mirrors the format under the caret in the two boxes:

```
procedure TfrmRichFormatting.EditorSelectionChange(Sender: TObject);
var
  vState: TsgcRichCommandState;
begin
  { CommandState describes the selection for one command. Mixed is True
    when the selection holds different values, for example two fonts. }
  vState := Editor.CommandState(recFontName);
  if vState.Mixed then
    cmbFont.Text := ''
  else
    cmbFont.Text := vState.Text;
  vState := Editor.CommandState(recFontSize);
  if vState.Mixed or (vState.Value <= 0) then
    cmbSize.Text := ''
  else
    cmbSize.Text := IntToStr(Round(vState.Value));
end;
```

How it works

1. Each standard action knows its command: `TsgcRichEditBold` runs `recBold`, `TsgcRichEditUndo` runs `recUndo`, and so on. Because `Control = Editor`, they always act on this editor, even while the focus is on a combo box.
2. While the application is idle, the actions ask the editor for the state of their command, enable or disable themselves and set `Checked`. The toggle buttons look pressed while their action is checked. This is why there is no code for the first toolbar row at all.
3. Choosing a font calls `cmbFontSelect`, which runs `recFontName` with the name in `Param.Text`. The size box runs `recFontSize` with the size in points in `Param.Value`.
4. The colour buttons run `recFontColor` and `recHighlight` with an ARGB colour in `Param.Color`.
5. Each handler gives the focus back to the editor with `Editor.SetFocus`, so the user can keep typing.
6. Every caret move fires `OnSelectionChange`. `EditorSelectionChange` reads `CommandState(recFontName)` and `CommandState(recFontSize)` and shows an empty box when the selection is `Mixed`, for example a selection across two fonts. `FormCreate` calls it once so the boxes are filled at startup.

> **Tip.** With an action list holding `TsgcRichEditBold` and its friends, Ctrl+B is handled by the action (a form shortcut) instead of the editor itself. The result is the same, and the toolbar updates either way.

Try this

- Add a `TsgcRichEditAlignJustify` action with `Control = Editor` and a toolbar button for it.
- Add a button that runs `Editor.ExecuteCommand(recStrikeout)`, or one for `recClearFormatting`.
- Add a green button with `DoApplyColor(recFontColor, $FF2E7D32)`.

4.3 Demo 03: Rich Tables

Folder `03.RichTables`, project `RichTables.dpr`.

What you will build

A small quarterly report with paragraph styles, a table and a picture. A Style box shows and changes the style of the paragraph under the caret. Buttons insert a new table and add or delete rows and columns in the table under the caret; they are enabled only while the caret is inside a table. Another button inserts a picture drawn at run time, and the same code puts a picture into the document at startup.

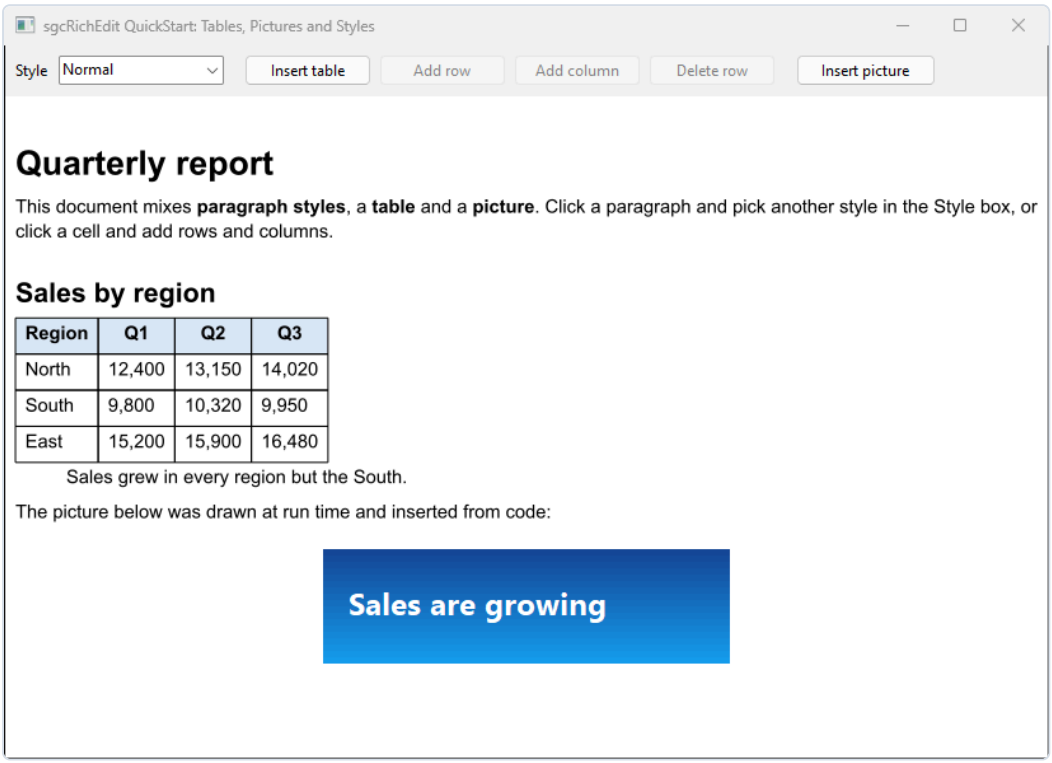


Figure 4.3. Demo 03 at startup. The caret is outside the table, so Add row, Add column and Delete row are disabled. The blue banner was drawn and inserted by code.

Components on the form

Component	Class	Properties you set
`cmbStyle`	`TComboBox`	Items Normal, Title, Subtitle, Heading 1, Heading 2, Heading 3, Quote and Code; `OnSelect`.
`btnTable`, `btnAddRow`, `btnAddColumn`, `btnDeleteRow`, `btnPicture`	`TButton`	`OnClick`.
`pnlTools`	`TPanel`	`Align = alTop`, holds the box and the buttons.
`Editor`	`TsgcVCLRichEdit`	`Align = alClient`, `BorderStyle = bsNone`, `OnSelectionChange = EditorSelectionChange`.

The code

`FormCreate` loads an HTML document with a table and a `[picture]` placeholder, then finds the placeholder and puts the picture over it:

```
procedure TfrmRichTables.FormCreate(Sender: TObject);
begin
  Editor.LoadFromString(
    '<h1>Quarterly report</h1>' +
    '<p>This document mixes <b>paragraph styles</b>, a <b>table</b> and a ' +
    '<b>picture</b>. Click a paragraph and pick another style in the Style ' +
    'box, or click a cell and add rows and columns.</p>' +
    '<h2>Sales by region</h2>' +
    // ...
    '<blockquote>Sales grew in every region but the South.</blockquote>' +
    '<p>The picture below was drawn at run time and inserted from code:</p>' +
    '<p style="text-align:center">[picture]</p>', refHTML);
  { select the [picture] placeholder with the find command of the editor,
    then insert the picture: like typing, an insert replaces the
    selection }
  Editor.Commands.FindText := '[picture]';
  Editor.Commands.FindNext;
  btnPictureClick(nil);
  Editor.Modified := False;
  EditorSelectionChange(nil);
end;
```

A style is applied by name, and a table is inserted with its size in `Int1` and `Int2`. The row and column buttons run commands without a value through a small helper:

```
procedure TfrmRichTables.DoCommand(aCommand: TsgcRichEditCommand);
begin
  Editor.ExecuteCommand(aCommand);
  Editor.SetFocus;
end;

procedure TfrmRichTables.cmbStyleSelect(Sender: TObject);
var
  vParam: TsgcRichCommandParam;
begin
  sgcRichCommandParamClear(vParam);
  vParam.Text := cmbStyle.Text;
  Editor.ExecuteCommand(recStyle, vParam);
  Editor.SetFocus;
end;

procedure TfrmRichTables.btnTableClick(Sender: TObject);
var
  vParam: TsgcRichCommandParam;
begin
  sgcRichCommandParamClear(vParam);
  vParam.Int1 := 3; // rows
  vParam.Int2 := 3; // columns
  Editor.ExecuteCommand(recInsertTable, vParam);
  Editor.SetFocus;
end;

procedure TfrmRichTables.btnAddRowClick(Sender: TObject);
begin
  DoCommand(recInsertRowBelow);
end;
```

The picture is drawn on a `TBitmap`, its bytes go into the image store of the document, and `recInsertImage` puts it at the caret:

```
procedure TfrmRichTables.btnPictureClick(Sender: TObject);
var
  oBitmap: TBitmap;
  oStream: TMemoryStream;
  vBytes: TsgcRichBytes;
  vParam: TsgcRichCommandParam;
  y: Integer;
begin
  oBitmap := TBitmap.Create;
  oStream := TMemoryStream.Create;
  try
    { draw a small banner: a blue gradient with a title }
    oBitmap.PixelFormat := pf24bit;
    oBitmap.Width := 320;
    // ...
    oBitmap.SaveToStream(oStream);

    { the picture bytes, unchanged, go into the image store of the
      document; PNG, JPEG, GIF and BMP are accepted }
    SetLength(vBytes, oStream.Size);
    Move(oStream.Memory^, vBytes[0], oStream.Size);
    sgcRichCommandParamClear(vParam);
    vParam.Int1 := Editor.Document.Images.Add('image/bmp', vBytes,
      oBitmap.Width, oBitmap.Height);
    { the size in the document, in points (a 96 dpi pixel is 0.75 pt) }
    vParam.Value := oBitmap.Width * 0.75;
    vParam.Value2 := oBitmap.Height * 0.75;
    vParam.Text := 'Sales banner'; // the alternative text
    Editor.ExecuteCommand(recInsertImage, vParam);
  finally
    oStream.Free;
    oBitmap.Free;
  end;
end;
```

The selection event shows the style of the paragraph and enables the table buttons:

```
procedure TfrmRichTables.EditorSelectionChange(Sender: TObject);
begin
  { the style of the paragraph under the caret }
  cmbStyle.Text := Editor.CommandState(recStyle).Text;
  if cmbStyle.Text = '' then
    cmbStyle.Text := 'Normal';
  { the table commands are enabled only inside a table }
  btnAddRow.Enabled := Editor.CommandState(recInsertRowBelow).Enabled;
  btnAddColumn.Enabled := Editor.CommandState(recInsertColumnAfter).Enabled;
  btnDeleteRow.Enabled := Editor.CommandState(recDeleteRow).Enabled;
end;
```

How it works

1. The HTML reader turns `<h1>`, `<h2>` and `<blockquote>` into the Heading 1, Heading 2 and Quote styles, and the `<table>` into an editable table with the cell backgrounds of the header row.
2. `Editor.Commands` is the command object of the editor. Its `FindText` and `FindNext` select the placeholder, and since an insert replaces the selection, like typing does, the picture takes the place of `[picture]`.
3. `Document.Images.Add` stores the bytes of the picture unchanged, with their MIME type and pixel size, and returns an id. `recInsertImage` places the picture with that id at the caret, at the size in points given in `Value` and `Value2`, with `Text` as its alternative text for screen readers.

4. The Style box runs `recStyle` with the style name in `Param.Text`. The style changes the whole paragraph under the caret.
5. `btnTableClick` runs `recInsertTable` for 3 rows and 3 columns. The row and column buttons run their commands on the table under the caret.
6. In `EditorSelectionChange`, `CommandState(recStyle).Text` is the style of the paragraph, an empty string for a paragraph without a style, which the demo shows as Normal. `CommandState(...).Enabled` is `True` for the table commands only inside a table, so the editor itself decides which buttons are available.

Try this

- Change `btnTableClick` to insert a table with 5 rows and 2 columns.
- Add a button that runs `DoCommand(recMergeCells)`, then select two cells and click it.
- Insert the picture at half size: multiply `Value` and `Value2` by 0.375 instead of 0.75.

4.4 Demo 04: Rich Formats

Folder `04.RichFormats`, project `RichFormats.dpr`.

What you will build

An editor on the left and, on the right, the same document written as text in the format chosen in the Format box: the native JSON, HTML, Markdown, RTF or plain text. Edit the document and the source view follows after every change. Buttons save the document to a file in the chosen format, load it back, and export it to DOCX and PDF.

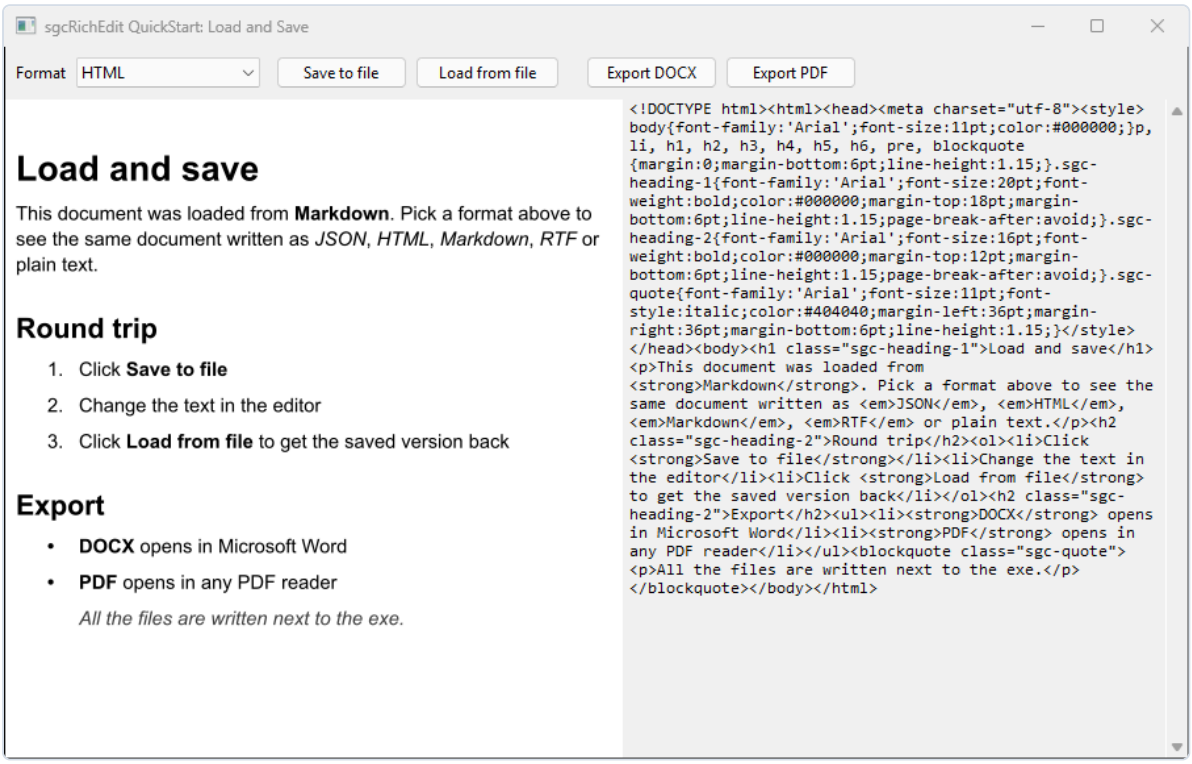


Figure 4.4. Demo 04 at startup. The document was loaded from Markdown; the right side shows it written as HTML.

Components on the form

Component	Class	Properties you set
`cmbFormat`	`TComboBox`	`Style = csDropDownList`, items JSON, HTML, Markdown, RTF and Plain text, `ItemIndex = 1`, `OnChange`.
`btnSave`, `btnLoad`, `btnDOCX`, `btnPDF`	`TButton`	`OnClick`.
`lblStatus`	`TLabel`	Shows what the last button did.
`Editor`	`TsgcVCLRichEdit`	`Align = alLeft`, `Width = 470`, `BorderStyle = bsNone`, `OnChange = EditorChange`.
`Splitter`	`TSplitter`	Between the editor and the source view.
`memSource`	`TMemo`	`Align = alClient`, read only, Consolas font.

The code

The `uses` clause is where this demo decides which formats it supports:

```
uses
  SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls,
  // sgc
  sgcRichEdit_Formats, sgcRichEdit_VCL, sgcRichEdit_HTML,
  sgcRichEdit_HTML_Writer, sgcRichEdit_Markdown, sgcRichEdit_RTF_Reader,
  sgcRichEdit_RTF_Writer, sgcRichEdit_DOCX,
  // linking this unit enables ExportToPDF (it uses sgcPDF)
  sgcUIPDF_Export;
```

Two constant arrays map the items of the Format box to formats and file extensions, and two small helpers use them:

```
const
  { the formats of cmbFormat, in the order of its items }
  CS_FORMATS: array [0 .. 4] of TsgcRichEditFormat = (refJSON, refHTML,
    refMarkdown, refRTF, refText);
  CS_EXTENSIONS: array [0 .. 4] of string = ('.json', '.html', '.md',
    '.rtf', '.txt');
  // ...
function TfrmRichFormats.DoFormat: TsgcRichEditFormat;
begin
  Result := CS_FORMATS[cmbFormat.ItemIndex];
end;

function TfrmRichFormats.DoFileName(const aExt: string): string;
begin
  Result := ExtractFilePath(ParamStr(0)) + 'RichFormats_sample' + aExt;
end;
```

`FormCreate` loads a Markdown string, and the source view is refreshed with `SaveToString` whenever the format or the document changes:

```
procedure TfrmRichFormats.FormCreate(Sender: TObject);
begin
  Editor.LoadFromString(
    '# Load and save' + #13#10#13#10 +
    'This document was loaded from **Markdown**. Pick a format above to ' +
    'see the same document written as *JSON*, *HTML*, *Markdown*, *RTF* ' +
    'or plain text.' + #13#10#13#10 +
    '## Round trip' + #13#10#13#10 +
    // ...
    '> All the files are written next to the exe.', refMarkdown);
  Editor.Modified := False;
  DoShowSource;
end;

procedure TfrmRichFormats.DoShowSource;
begin
  { SaveToString writes the document in any text format }
  memSource.Text := Editor.SaveToString(DoFormat);
end;

procedure TfrmRichFormats.EditorChange(Sender: TObject);
begin
  DoShowSource;
end;
```

Save and load use the extension of the chosen format:

```
procedure TfrmRichFormats.btnSaveClick(Sender: TObject);
var
  vFile: string;
begin
  vFile := DoFileName(CS_EXTENSIONS[cmbFormat.ItemIndex]);
  Editor.SaveToFile(vFile, DoFormat);
  Editor.Modified := False;
  lblStatus.Caption := 'Saved ' + ExtractFileName(vFile);
end;

procedure TfrmRichFormats.btnLoadClick(Sender: TObject);
var
  vFile: string;
  vFormat: TsgcRichEditFormat;
begin
  vFile := DoFileName(CS_EXTENSIONS[cmbFormat.ItemIndex]);
  if not FileExists(vFile) then
  begin
    lblStatus.Caption := 'Save the file first';
    Exit;
  end;
  { refAuto: the format is detected; the result says which one was used }
  vFormat := Editor.LoadFromFile(vFile);
  Editor.Modified := False;
  lblStatus.Caption := 'Loaded ' + ExtractFileName(vFile) + ' as ' +
    cmbFormat.Items[cmbFormat.ItemIndex];
  if vFormat = refAuto then
    lblStatus.Caption := 'Unknown format';
  DoShowSource;
end;
```

And the two export buttons:

```
procedure TfrmRichFormats.btnDOCXClick(Sender: TObject);
var
  vFile: string;
begin
  vFile := DoFileName('.docx');
  Editor.SaveToFile(vFile, refDOCX);
  lblStatus.Caption := 'Saved ' + ExtractFileName(vFile);
end;

procedure TfrmRichFormats.btnPDFClick(Sender: TObject);
var
  vFile: string;
begin
  vFile := DoFileName('.pdf');
  Editor.ExportToPDF(vFile);
  lblStatus.Caption := 'Saved ' + ExtractFileName(vFile);
end;
```

How it works

1. Each unit in the `uses` clause registers a reader or a writer at startup: HTML in both directions, Markdown, RTF in both directions and the DOCX writer. JSON and plain text need no unit.
2. `FormCreate` loads the document with an explicit `refMarkdown`, because Markdown is never detected from the content of a string.
3. `DoShowSource` writes the document in the chosen format with `SaveToString`. Since `OnChange` fires after every edit, the source view always matches the editor.
4. `btnSaveClick` writes `RichFormats_sample` plus the extension of the format next to the executable. `btnLoadClick` reads the file back with `LoadFromFile` and the default `refAuto`, which detects the format, and returns the format it used.
5. `btnDOCXClick` writes a Word document with `SaveToFile(..., refDOCX)`, and `btnPDFClick` a PDF file with `ExportToPDF`. Both are export only formats, so the Load button cannot read them back.

Watch the source view while you switch formats: the JSON shows every detail of the document, the HTML carries the styles as CSS classes, and the Markdown keeps the structure but not the colours, sizes and fonts, since Markdown has no syntax for them.

> **Important.** `ExportToPDF` needs the PDF writer of sgcPDF, which is linked by the unit `sgcUIPDF_Export`. The demo lists it in the `uses` clause shown above, which enables `ExportToPDF`. Your own application must do the same, otherwise the call raises `EsgcUIPDFExportUnavailable` (see chapter 6).

Try this

- Remove `sgcRichEdit_RTF_Writer` from the `uses` clause, run, and choose RTF in the Format box: you get the exception “Export to RTF is not supported”. Put it back.
- Replace `Editor.SaveToString(DoFormat)` with the properties `Editor.AsMarkdown`, `Editor.AsHTML`, `Editor.AsRTF`, `Editor.AsJSON` or `Editor.AsText`.
- Add a button that writes the first page as an image with `Editor.ExportToPNG(DoFileName('.png'))`.

4.5 Demo 05: Rich Letter App

Folder `05.RichLetterApp`, project `RichLetterApp.dpr`.

What you will build

A small mail merge application. A list on the left holds four customers; clicking one shows the letter for that customer, with the name, company, city and amount filled in and their formatting kept. A checkbox switches to the template itself, where you can change the wording or the formatting for every letter at once. A Theme box switches the editor between the default, light and dark themes, and two buttons export the letter to PDF and show the print preview.

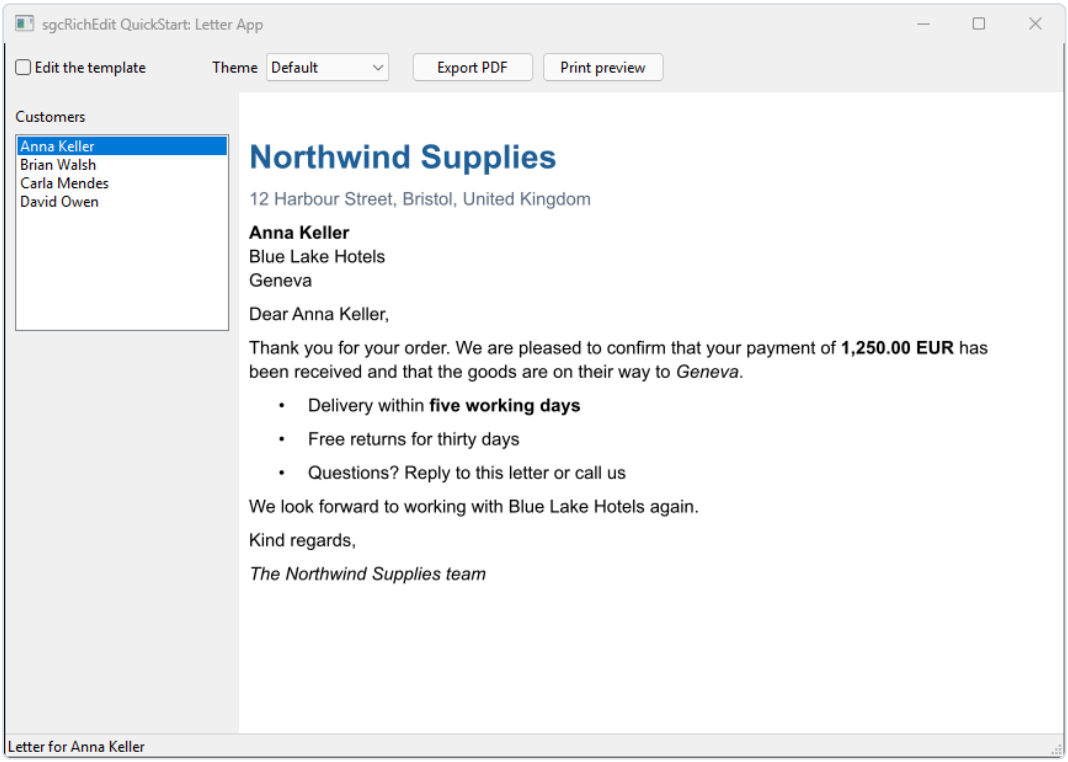


Figure 4.5. Demo 05 at startup, showing the letter for the first customer. The text wraps at the width of an A4 page, not at the width of the window.

Components on the form

Component	Class	Properties you set
`lstCustomers`	`TListBox`	Four customer names, `OnClick`.
`chkTemplate`	`TCheckBox`	`Caption = 'Edit the template'`, `OnClick`.
`cmbTheme`	`TComboBox`	Items Default, Light and Dark, `OnChange`.
`btnPDF`, `btnPreview`	`TButton`	`OnClick`.
`Editor`	`TsgcVCLRichEdit`	`Align = alClient`, `BorderStyle = bsNone`, `DocumentWidth = 453.5`, `OnChange = EditorChange`.
`StatusBar`	`TStatusBar`	`SimplePanel = True`.

```
  object Editor: TsgcVCLRichEdit
    Left = 190
    Top = 40
    Width = 670
    Height = 521
    Align = alClient
    BorderStyle = bsNone
    DocumentWidth = 453.500000000000000000
    TabOrder = 2
    OnChange = EditorChange
  end
```

The value of `DocumentWidth`, 453.5, is the text width of an A4 page with 2.5 cm margins, in points.

The code

The `uses` clause adds `sgcUICore_Theme` for the theme values. The customer data lives in constant arrays, one entry per item of the list:

```
uses
  SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls, ComCtrls,
  // sgc
  sgcUICore_Theme, sgcRichEdit_Formats, sgcRichEdit_VCL, sgcRichEdit_HTML,
  // linking this unit enables ExportToPDF (it uses sgcPDF)
  sgcUIPDF_Export;
  // ...
const
  { the customers: one entry per item of lstCustomers }
  CS_COMPANY: array [0 .. 3] of string = ('Blue Lake Hotels',
    'Ridge Engineering', 'Harbour Books', 'Sunfield Farms');
  CS_CITY: array [0 .. 3] of string = ('Geneva', 'Denver', 'Lisbon',
    'Adelaide');
  CS_AMOUNT: array [0 .. 3] of string = ('1,250.00 EUR', '980.40 USD',
    '312.75 EUR', '2,040.00 AUD');
```

`FormCreate` writes the template once in HTML, loads it and keeps it as native JSON:

```
procedure TfrmRichLetterApp.FormCreate(Sender: TObject);
begin
  { the template is written once in HTML, loaded, and kept as JSON }
  Editor.LoadFromString(
    '<h1 style="color:#1f5f99">Northwind Supplies</h1>' +
    '<p style="color:#5b6b7f">12 Harbour Street, Bristol, United Kingdom' +
    '</p>' +
    '<p><b>[Name]</b><br>[Company]<br>[City]</p>' +
    '<p>Dear [Name],</p>' +
    // ...
    '<p>We look forward to working with [Company] again.</p>' +
    '<p>Kind regards,</p>' +
    '<p><i>The Northwind Supplies team</i></p>', refHTML);
  FTemplate := Editor.SaveToString(refJSON);
  lstCustomers.ItemIndex := 0;
  DoShowLetter;
end;
```

A letter is the template with the placeholders replaced:

```
procedure TfrmRichLetterApp.DoShowLetter;
var
  i: Integer;
begin
  i := lstCustomers.ItemIndex;
  if i < 0 then
    Exit;
  Editor.LoadFromString(FTemplate, refJSON);
  { ReplaceAll returns how many times the text was found; [] means: any
    case, also inside words }
  Editor.Commands.ReplaceAll('[Name]', lstCustomers.Items[i], []);
  Editor.Commands.ReplaceAll('[Company]', CS_COMPANY[i], []);
  Editor.Commands.ReplaceAll('[City]', CS_CITY[i], []);
  Editor.Commands.ReplaceAll('[Amount]', CS_AMOUNT[i], []);
  Editor.Modified := False;
  StatusBar.SimpleText := 'Letter for ' + lstCustomers.Items[i];
end;
```

The checkbox switches between the template and the letters, and an edited template replaces the stored one:

```
procedure TfrmRichLetterApp.chkTemplateClick(Sender: TObject);
begin
  if chkTemplate.Checked then
  begin
    Editor.LoadFromString(FTemplate, refJSON);
    StatusBar.SimpleText := 'Editing the template: keep the placeholders ' +
      '[Name], [Company], [City] and [Amount]';
  end
  else
  begin
    { the edited template becomes the template of every letter }
    FTemplate := Editor.SaveToString(refJSON);
    DoShowLetter;
  end;
  btnPDF.Enabled := not chkTemplate.Checked;
end;
```

The theme, the PDF export, the print preview and the status bar are one or two lines each:

```
procedure TfrmRichLetterApp.cmbThemeChange(Sender: TObject);
begin
  case cmbTheme.ItemIndex of
    1:
      Editor.Theme := utLight;
    2:
      Editor.Theme := utDark;
  else
    Editor.Theme := utDefault;
  end;
end;

procedure TfrmRichLetterApp.btnPDFClick(Sender: TObject);
var
  vFile: string;
begin
  vFile := ExtractFilePath(ParamStr(0)) + 'Letter ' +
    lstCustomers.Items[lstCustomers.ItemIndex] + '.pdf';
  Editor.ExportToPDF(vFile);
  StatusBar.SimpleText := 'Saved ' + vFile;
end;

procedure TfrmRichLetterApp.btnPreviewClick(Sender: TObject);
begin
  Editor.PrintPreview('Letter');
end;

procedure TfrmRichLetterApp.EditorChange(Sender: TObject);
begin
  if Editor.Modified then
    StatusBar.SimpleText := 'Modified, ' +
      IntToStr(Length(Editor.PlainText)) + ' characters';
end;
```

How it works

1. The template is written once in HTML, with `[Name]`, `[Company]`, `[City]` and `[Amount]` where the customer data goes. The name in the address and the amount are bold, the city in the text is italic.
2. `SaveToString(refJSON)` keeps the template in the native JSON format, which stores every format and picture exactly. Loading it back with `LoadFromString(FTemplate, refJSON)` gives the identical document, ready for the next letter.
3. `Editor.Commands.ReplaceAll` replaces every occurrence of a placeholder and returns how many it found. A replacement takes the format of the text it replaces, so the bold `[Amount]` becomes a bold amount. The empty set `[]` means any case, also inside words.
4. While Edit the template is checked, the editor shows the template. Unchecking it saves the edited template with `SaveToString(refJSON)`, and every letter from then on uses it.
5. `DocumentWidth = 453.5` makes the text wrap like the printed page. It limits only the width of the text: the editor does not draw a sheet of paper around it.
6. `cmbThemeChange` sets `Editor.Theme` to one of the sgcUICore themes.
7. `btnPDFClick` exports the current letter with `ExportToPDF`, and `btnPreviewClick` opens the sgcUI print preview with `PrintPreview`, where the letter is cut into pages using the page settings of the document.
8. `EditorChange` shows the number of characters of the plain text while the document is modified.

> **Tip.** The placeholders use square brackets on purpose. A placeholder written with braces, such as `{Name}`, cannot be mentioned inside a `{ }` Pascal comment, because its closing brace ends the comment. Square brackets avoid the problem in comments and are easy to spot in the document.

> **Important.** As in demo 04, this demo links the unit `sgcUIPDF_Export` to enable `ExportToPDF`. Your own application must link it too, otherwise `ExportToPDF` raises `EsgcUIPDFExportUnavailable`. The print preview does not need it.

Try this

- Add a `[Date]` placeholder to the template and replace it in `DoShowLetter` with `Editor.Commands.ReplaceAll('[Date]', DateToStr(Date), [])`. The date follows the regional settings of Windows.
- Set `DocumentWidth = 0` and resize the window: the letter now wraps at the window width.
- Add a button that saves every letter as a Word document in a loop: select each customer, call `DoShowLetter` and then `Editor.SaveToFile(..., refDOCX)`, with `sgcRichEdit_DOCX` in the `uses` clause.

5. Main components

This chapter is a short reference of the classes you meet in a VCL application. The help has one topic per class with every property, method and event; chapter 7 has the links.

Reference table

Class	Where you find it	What it is for
`TsgcVCLRichEdit`	SGC RichEdit palette page	The rich text editor control.
`TsgcVCLDBRichEdit`	SGC RichEdit palette page	The editor bound to a field of a dataset.
`TsgcRichEditBold`, `TsgcRichEditUndo`, ...	SGC RichEdit category of the standard actions	Ready made actions for a toolbar, a menu or a ribbon.
`TsgcRichEditCommandAction`	SGC RichEdit category of the standard actions	An action for any command, chosen in its `Command` property.
`TsgcRichDocument`	Unit `sgcRichEdit_Model`, the `Document` property	The document: paragraphs, tables, images, links, styles and page settings.
`TsgcRichEditCommands`	Unit `sgcRichEdit_Commands`, the `Commands` property	The command object of an editor, with find and replace.
`TsgcRichEditPageSource`	Unit `sgcRichEdit_Print`, `CreatePrintSource`	Cuts a document into pages for printing, preview and export.
`TsgcRichFontRegistry`	Unit `sgcRichEdit_Fonts`, the `FontRegistry` property	The fonts used to measure and draw text.
`TsgcFMXRichEdit`, `TsgcFMXDBRichEdit`	SGC RichEdit palette page in FireMonkey projects	The FireMonkey editors.

The editor

`TsgcVCLRichEdit` is the control of every QuickStart demo. Its members fall into a few groups:

- **Content:** `LoadFromString`, `LoadFromFile`, `LoadFromStream`, `SaveToString`, `SaveToFile`, `SaveToStream`, the `AsJSON`, `AsHTML`, `AsRTF`, `AsMarkdown` and `AsText` properties, `PlainText`, `SelText`, `Modified`, `Clear` and `Document`.
- **Editing:** `ExecuteCommand`, `CommandState`, `Commands`, `Undo`, `Redo`, `SelectAll`, `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard` and `ScrollCaretIntoView`.
- **Look and behaviour:** `Theme`, `Zoom`, `DocumentWidth`, `ReadOnly`, `ShowSpelling`, `AutoOpenLinks`, `BorderStyle` and `Color`.
- **Output:** `Print`, `PrintPreview`, `ExportToPDF`, `ExportToPNG` and `CreatePrintSource`.
- **Events:** `OnChange`, `OnSelectionChange`, `OnLinkClick` and `OnContextMenu`, plus the usual mouse, keyboard and focus events of a VCL control.

The data aware editor

`TsgcVCLDBRichEdit` is the same editor bound to a field of a dataset. Set `DataSource` and `DataField` as for any data aware control, and `FieldFormat` to the format the field is stored in: the native JSON keeps everything, while RTF, HTML or plain text suit fields that other programs read as well. With `FieldFormat = refAuto` the format of each value is detected from its content. For large documents, `AutoDisplay = False` loads a document only when the user asks for it. The `DataAware` demo of chapter 7 shows the control with a `TClientDataSet`.

The actions

The standard actions of chapter 2 cover the clipboard, undo and redo, select all, find, bold, italic, underline, the four alignments, bullets, numbering, indent, outdent and the insertion of tables, links and images. Every action has a `Control` property: set it to an editor to bind the action to that editor, or leave it empty to act on the rich edit that has the focus. For a command without its own action, use `TsgcRichEditCommandAction` and choose the command in its `Command` property.

Documents without a control

The format functions also work on a `TsgcRichDocument` without any control: `sgcRichLoadFromFile`, `sgcRichLoadFromString`, `sgcRichSaveToFile`, `sgcRichSaveToString` and their stream versions in `sgcRichEdit_Formats`, and `sgcRichExportPDF` in `sgcRichEdit_Print`. A service or a console program can convert documents this way, for example RTF to PDF, without a window. The help topic Formats and the topic Printing and PDF show complete examples.

FireMonkey

`TsgcFMXRichEdit` and `TsgcFMXDBRichEdit` are the FireMonkey versions, in the `sgcRichEditFMX` and `dclsgcRichEditFMX` packages for Delphi XE6 and later, with their own standard actions. They use the same document model, the same commands and the same formats, so the code of this guide applies to them almost unchanged. The `EditorFMX` and `DataAwareFMX` demos of chapter 7 show them.

6. Common problems

These are the problems that come up most often in a first sgcRichEdit project. Each entry says what you see, why it happens and how to fix it.

Formats and export

"Import of ... is not supported" or "Export to ... is not supported"

What you see. `LoadFromString`, `LoadFromFile`, `SaveToString` or `SaveToFile` raises an exception that names a format.

Why. In Delphi a format exists only when its unit is linked into the application. The unit registers its reader or writer at startup; without it, the format is unknown. Readers and writers are separate units for HTML and RTF, so an application that reads HTML may still be unable to write it.

Fix. Add the unit to a `uses` clause: `sgcRichEdit_HTML` to read HTML, `sgcRichEdit_HTML_Writer` to write it, `sgcRichEdit_Markdown` for Markdown, `sgcRichEdit_RTF_Reader` and `sgcRichEdit_RTF_Writer` for RTF, `sgcRichEdit_DOCX` for DOCX. The table in Formats are units lists them all.

ExportToPDF raises EsgcUIPDFExportUnavailable

What you see. Printing and the print preview work, but `ExportToPDF` raises `EsgcUIPDFExportUnavailable`.

Why. The PDF is written by sgcPDF through the sgcUIPDF bridge, and the sgcRichEdit units do not link the bridge on their own, so an application that does not need PDF does not carry the PDF engine.

Fix. Add `sgcUIPDF_Export` to a `uses` clause of the application, for example the one of the main form. Demos 04 and 05 already list the unit for their Export PDF buttons, and your own application needs the same line. sgcPDF is installed together with sgcRichEdit, so the unit is on the library path.

A Markdown document shows its # and ** signs

What you see. You load a Markdown string and the editor shows it as plain text, with the Markdown syntax visible.

Why. With `refAuto`, Markdown is never detected from the content. Only the `.md` and `.markdown` file extensions, or an explicit format, select the Markdown reader; anything that is not recognised as RTF, JSON or HTML is read as plain text.

Fix. Pass `refMarkdown`, as demo 04 does, and make sure `sgcRichEdit_Markdown` is in the `uses` clause. For files, use the `.md` extension.

The Load button cannot read a DOCX or PDF file

What you see. A file written with `refDOCX` or `ExportToPDF` cannot be loaded back.

Why. DOCX, PDF and PNG are export only formats.

Fix. Keep a copy in the native JSON format (or HTML, Markdown or RTF) whenever the document has to be edited again, and export DOCX or PDF from it when needed.

Formatting from code

A colour comes out wrong or invisible

What you see. `recFontColor` or `recHighlight` gives a different colour from the one you meant, or no visible colour at all.

Why. The editor uses ARGB colours, `$AARRGGBB`. A VCL `TColor` such as `clRed` or `$002B39C0` stores its bytes in a different order and has 0 in the alpha byte.

Fix. Write the colour as `$FF` followed by the red, green and blue bytes, for example `$FFC0392B` for the red of demo 02.

A command does nothing, or uses an old value

What you see. `ExecuteCommand` with a `TsgcRichCommandParam` has no effect or applies an unexpected value.

Why. The record is a local variable, so its fields hold whatever was in memory, and each command reads a specific field: `Text` for a font or style name, `Value` for a size, `Int1` and `Int2` for table rows and columns, `Color` for colours.

Fix. Call `sgcRichCommandParamClear` first, then fill the field the command reads, as every demo does. Check `CommandState(Command).Enabled` when a command may not apply where the caret is, such as the table commands outside a table.

The Style box shows nothing for a normal paragraph

What you see. `CommandState(recStyle).Text` returns an empty string.

Why. A paragraph without an explicit style has no style name.

Fix. Treat an empty string as Normal, as demo 03 does in `EditorSelectionChange`.

A placeholder in a comment breaks the unit

What you see. After you mention a placeholder such as `{Name}` in a comment, the compiler reports errors in the lines after it.

Why. A `{ }` Pascal comment ends at the first closing brace, so the brace of the placeholder closes the comment early and the rest of the comment is read as code.

Fix. Use placeholders with square brackets, as demo 05 does (`[Name]`), or write such comments with `//` or `(* *)`.

Pictures and fonts have the wrong size

What you see. An inserted picture is larger or smaller than expected, or a font size from code does not match your pixel measurements.

Why. Sizes in the document are in points, not pixels.

Fix. Multiply pixels at 96 dpi by 0.75 to get points, as demo 03 does for its picture.

Layout and fonts

DocumentWidth does not show a page

What you see. With `DocumentWidth` set, the text wraps at the right width, but there is no sheet of paper, no page shadow and no page breaks on the screen.

Why. `DocumentWidth` only limits the width of the text. The editor always shows the document as one continuous page.

Fix. That is the intended behaviour. To see the document as pages, use `PrintPreview`, which applies the page settings of `Document.PageSettings`.

Text shows in another font

What you see. A document asks for a font that is installed, but the editor draws it with a different one.

Why. sgcRichEdit measures and draws text with its own font engine, which reads TrueType outlines only (`.ttf` files and `.ttc` collections). Fonts with CFF outlines, which are most `.otf` files, are skipped, and the text falls back to an alias of the family and finally to the Roboto font embedded in sgcUICore.

Fix. Use the TrueType version of the font, add a font file with `Editor.FontRegistry.AddFontFile`, or map the family to an installed TrueType family with `Editor.FontRegistry.AddAlias`.

Ctrl+B does not reach my OnKeyDown handler

What you see. On a form with a `TActionList` holding `TsgcRichEditBold`, a key handler of the editor never sees Ctrl+B.

Why. The action owns the shortcut, so the form runs the action before the key reaches the editor.

Fix. Nothing to fix for the formatting itself: the action runs the same command. If you need to react to the shortcut, use the `OnExecute` event of the action, or clear its `ShortCut`.

7. Where next

Once the five QuickStart demos feel familiar, you know the core of sgcRichEdit. This chapter points you to the larger demos, to the documentation and to support.

The other Delphi demos

The other folders in `Demos\RichEdit\Delphi` are complete applications. Each one has a `readme.html` that explains what it shows and how to run it.

Demo	What it shows
`Editor`	A word processor with a ribbon: a `TsgcVCLRichEdit` under a `TsgcVCLRibbon` with Home, Insert and View tabs and a File backstage. Every ribbon item runs an editor command and follows its state, a find and replace bar searches the document, and the status bar shows the pages, the words and the zoom. The natural next step after demo 05.
`DataAware`	A `TsgcVCLDBRichEdit` bound to the memo field of an in memory `TClientDataSet`, next to a `TDBGrid` and a `TDBNavigator`. The notes are stored as native JSON, RTF, HTML and plain text, and a combo box chooses the format that Post writes.
`EditorFMX`	A FireMonkey word processor with `TsgcFMXRichEdit` and a `TsgcFMXRibbon`: formatting, lists, tables, images and links, five file formats, PDF and PNG export, zoom and a touch mode with selection handles.
`DataAwareFMX`	A `TsgcFMXDBRichEdit` bound to a memo field of a `TClientDataSet`, with four notes stored as JSON, RTF, HTML and Markdown and a FieldFormat selector.

The ribbon of the `Editor` and `EditorFMX` demos is another sgcUI control; their readme files say what they need.

Help topics

The sgcRichEdit help is part of the sgcUI help at https://www.esegece.com/help/sgcUI, in the RichEdit section. These topics are the best continuation of this guide:

Topic	What it covers
Introduction	What sgcRichEdit does and where to start.
Overview, Installation, Editions	The layers of the product, the packages, the installation and the editions.
Document model, Styles and lists	The document, its blocks and runs, paragraph and character styles, lists.
Tables, Images and links, Find and replace	Table editing, the image store, links and searching.
Formats, Printing and PDF	What each format keeps, format detection, pagination, PDF, PNG and fonts.
Clipboard, Editing shortcuts, Data aware editing	Copy and paste, the keyboard, and the data aware controls.
Actions, TsgcVCLRichEdit	The standard actions, and the editor with every property, method and event.
Troubleshooting	More problems and their causes.

On the website

- https://www.esegece.com/products/sgcrichedit/: the product page, with the features, the editions and the downloads.

Getting help

If you are stuck, write to info@esegece.com or open a ticket in the helpdesk of your account at https://www.esegece.com/. It helps to include your Delphi version, your sgcRichEdit version and edition, and the demo or a small project that shows the problem.

Enjoy writing your first documents.

---

sgcRichEdit and sgcUI are products of eSeGeCe. https://www.esegece.com