sgcToolbar

Getting Started

Put a themed toolbar and status bar 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 give it a modern command bar at the top and a status bar at the bottom with sgcToolbar. You do not need to know anything about sgcToolbar in advance. If you have used the standard `TToolBar` and `TStatusBar` of the VCL, many ideas will feel familiar, and chapter 1 explains the differences.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from a toolbar with three buttons to a small text editor with actions, a zoom menu, a dark mode toggle and a status bar that shows the caret position and the saved state. 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, click the buttons, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcToolbar 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\Toolbar\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloToolbar` to `05.ToolbarNotepad`, with a Delphi project (`.dpr`) that you can open and run directly. All of them are VCL applications for Windows. Demos 01, 02 and 04 work with every edition; demos 03 and 05 link toolbar items to actions, a feature of the Registered editions, as chapter 2 explains.

The screenshots were taken on a Windows machine with Spanish regional settings. Numbers and dates in the demos follow the Windows regional settings of the machine that runs them, so your windows may show them slightly differently.

Contents

1. What sgcToolbar is

The idea in one minute
Your first toolbar needs one event handler
What you need
Part of sgcUI
Coming from TToolBar and TStatusBar

2. Install and the palette

Installing
The packages
The palette page
Editions
Where the demos are
Running your first demo

3. Core concepts

The items are a collection
What an item can be: Kind
Toggles and radio groups
One event for every item
Items linked to actions
The status bar and its panels
Updating the status bar from a thread
Themes and the accent colour
Size, captions and the keyboard
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello Toolbar
4.2 Demo 02: Toolbar Item Kinds
4.3 Demo 03: Toolbar Actions
4.4 Demo 04: Toolbar Themes
4.5 Demo 05: Toolbar Notepad

5. Main components

Reference table
The toolbar
The status bar
The badge link
The ribbon
FireMonkey and beyond

6. Common problems

Actions and events
Layout and look
Editions, threads and compiling

7. Where next

The other Delphi demos
Help topics
On the website
Getting help

1. What sgcToolbar is

sgcToolbar gives Delphi applications the frame of a modern window: a command bar at the top and a status bar at the bottom, for the VCL and for FireMonkey. The toolbar shows a row of buttons, toggles, radio groups, dropdown buttons, labels, separators and spacers. The status bar shows text panels, progress bars and coloured state indicators. Both are drawn by sgcUICore, the shared engine of the sgcUI suite, so they follow the light and dark themes of the suite and look the same as the other sgcUI controls next to them.

You drop a `TsgcVCLToolbar` on a form, add items to its `Items` collection in the Object Inspector, and handle one event, `OnItemClick`, to react to the clicks. Drop a `TsgcVCLStatusBar` as well, add panels to its `Panels` collection, and change their text from code with one method call.

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

The idea in one minute

A toolbar is a list of commands. In sgcToolbar every command is an *item*: a `TsgcToolbarItem` in the `Items` collection of the bar. An item is not a separate control on the form, it is a row in a collection, like a column of a `TListView` or a field of a dataset. That keeps the form tidy and lets the bar measure, lay out and paint all its items in one go.

Every item is described by a few properties:

- **What it is**: `Kind` is a push button, a toggle, a dropdown button, a separator, a label or a spacer.
- **What it shows**: `Caption` is the text, `Glyph` a short text drawn in the icon slot (or `ImageIndex` for a real image from an image list), `Hint` the tooltip and `Badge` a small count on the icon.
- **What state it is in**: `Enabled`, `Visible` and, for a toggle, `Down`.

When the user clicks an item, the bar raises `OnItemClick` and passes the item. The status bar works the same way: its `Panels` collection holds `TsgcStatusBarPanel` items, each with a `Kind` (text, progress bar or indicator), a `Text` and a width. That is the entire idea; everything else is detail.

Your first toolbar needs one event handler

The first QuickStart demo is the smallest toolbar there is. The three buttons are set at design time, in the `Items` collection, and stored in the form file like this:

```
object Toolbar: TsgcVCLToolbar
  Left = 0
  Top = 0
  Width = 520
  Height = 44
  Items = <
    item
      Caption = 'New'
      Glyph = 'N'
      Hint = 'Create a new document'
    end
    item
      Caption = 'Open'
      Glyph = 'O'
      Hint = 'Open a document'
    end
    item
      Caption = 'Save'
      Glyph = 'S'
      Hint = 'Save the document'
    end>
  ShowHint = True
  OnItemClick = ToolbarItemClick
end
```

And this is the only code of the demo, the `OnItemClick` handler of the bar. It receives the item that was clicked, so one handler serves every button:

```
procedure TfrmHelloToolbar.ToolbarItemClick(Sender: TObject;
  aItem: TsgcToolbarItem);
begin
  { aItem is the TsgcToolbarItem that was clicked: its Caption, its Index
    in Toolbar.Items or its Tag tell the buttons apart. }
  lblInfo.Caption := 'You clicked ' + aItem.Caption;
end;
```

There is no `Align` line in the form file because `alTop` is the default of the toolbar, and no image list because the `Glyph` letters are the icons. Chapter 4 walks through this demo in detail.

What you need

- **Delphi.** Delphi 7, 2007, 2009, 2010, XE to XE8, 10 Seattle to 10.4 Sydney, 11, 12 and 13. There is a set of packages for each of these versions.
- **VCL.** `TsgcVCLToolbar` and `TsgcVCLStatusBar` run on Windows, Win32 and Win64 (Win64 from XE2, the first Delphi with a 64 bit compiler).
- **FireMonkey.** `TsgcFMXToolbar` and `TsgcFMXStatusBar`, from Delphi XE6, for the FireMonkey platforms. This guide uses the VCL controls; the FireMonkey controls have the same items, panels, properties and events.
- **sgcUICore.** The shared runtime of the sgcUI suite. It is installed together with sgcToolbar, you do not install it separately.

Part of sgcUI

sgcToolbar is one of the products of sgcUI, the eSeGeCe suite of visual components. All of them are drawn by sgcUICore, which provides the canvas, the fonts and the themes. In practice this means three things for you:

- One call such as `sgcUISetApplicationTheme(utDark, True)` switches every sgcUI control of the application to the dark theme at once: the toolbar, the status bar, and a grid, a navigation bar or a dashboard next to them.
- Theme presets such as Classic and Fluent change the shape and colours of every sgcUI control together. Demo 04 shows both.
- The toolbar and the status bar are the natural frame of the other products: a toolbar at the top and a status bar at the bottom of a window that hosts an sgcGrid grid, an sgcDashboard dashboard or an sgcNavigation navigation bar.

Coming from TToolBar and TStatusBar

Most Delphi developers know the toolbar and status bar of the VCL. sgcToolbar covers the same ground with a different shape, and it is worth knowing the differences before you start:

Topic	VCL `TToolBar` and `TStatusBar`	sgcToolbar
Buttons	Each button is a `TToolButton` control on the form	Each button is a `TsgcToolbarItem` in the `Items` collection, with one `OnItemClick` for all of them
Icons	An image list is needed	A text `Glyph` works without an image list; `Images` and `ImageIndex` are there for real images
Look	Follows the Windows or VCL style	Follows the sgcUI themes, light, dark and high contrast, Classic and Fluent presets, with an `AccentColor`
Extras		Badges, toggles in radio groups, spacers that push items to the right, status panels with a progress bar or a coloured indicator
Threads	Main thread only	`SetPanelText`, `SetProgress`, `SetIndicator` and `SetPanelVisible` of the status bar can be called from any thread
Frameworks	VCL only	The same model on VCL and FireMonkey

What stays the same: the toolbar aligns to the top, the status bar to the bottom, items can be linked to the actions of a `TActionList`, and dropdown buttons open an ordinary `TPopupMenu`. You can keep your action lists and popup menus.

2. Install and the palette

This chapter installs sgcToolbar, shows what lands on the component palette and runs the first demo. It takes a few minutes.

Installing

sgcToolbar 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 **sgcToolbar**. The installer shows one checkbox per product and enables the products your licence includes. sgcUICore has no checkbox: it is installed automatically with sgcToolbar.
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 sgcToolbar units without any setting on your side.

The packages

sgcToolbar is split into a framework neutral runtime package, one runtime package per framework and one design time package per framework. Each package name ends with the suffix of the Delphi version, for example `D13` for Delphi 13 or `D7` for Delphi 7:

Package (Delphi 13 name)	Kind	Contents
`sgcToolbarD13`	Runtime, framework neutral	The models of the toolbar and the status bar, the items, the panels and the renderer. Uses neither the VCL nor FireMonkey.
`sgcToolbarVCLD13`	Runtime, VCL	`TsgcVCLToolbar`, `TsgcVCLStatusBar` and the VCL ribbon
`dclsgcToolbarD13`	Design time, VCL	Registers the VCL components on the palette
`sgcToolbarFMXD13`	Runtime, FireMonkey (XE6 and later)	`TsgcFMXToolbar`, `TsgcFMXStatusBar` and the FireMonkey ribbon
`dclsgcToolbarFMXD13`	Design time, FireMonkey (XE6 and later)	Registers the FireMonkey components on the palette

The runtime packages need the sgcUICore packages, which the installer installs too. A VCL application never links FireMonkey, and a FireMonkey application never links the VCL.

The palette page

After the installation, the Tool Palette has a page called **SGC Toolbar**. With a VCL form open it shows:

Component	What it is
`TsgcVCLToolbar`	The toolbar, aligned to the top by default. Used by every QuickStart demo.
`TsgcVCLStatusBar`	The status bar, aligned to the bottom by default. Used by demos 03, 04 and 05.
`TsgcToolbarBadgeLink`	A non visual component that shows a count from another sgcUI component, for example the unread notifications of sgcNavigation, as the `Badge` of a toolbar item.
`TsgcVCLRibbon`	An Office style ribbon with tabs, groups, a Quick Access Toolbar and KeyTips. Registered editions only.
`TsgcVCLRibbonBackstage`	The full window File page of a ribbon. Its pages are added with the **New Page** command of the component editor. Registered editions only.

With a FireMonkey form open, the same page shows the FireMonkey components instead: `TsgcFMXToolbar`, `TsgcFMXStatusBar`, the FireMonkey ribbon and backstage, and the badge link, which works with both frameworks.

The QuickStart demos use only the toolbar and the status bar. Chapter 5 describes all the components.

Editions

sgcToolbar is available as a Registered edition, a Trial edition for evaluation and a Community edition. In the source code the difference is one compiler define, `SGC_TOOLBAR_ADVANCED`, which the Community edition does not define. Without it, three features are not compiled:

- The `Action` property of the items, the link to a `TActionList`.
- The overflow menu, which lists the items that do not fit in the width of the bar. In the Community edition those items are clipped.
- The ribbon, `TsgcVCLRibbon` and `TsgcVCLRibbonBackstage`, and their FireMonkey versions.

Everything else, the toolbar, the status bar, every item kind, badges, themes, images, keyboard navigation and the thread safe status bar methods, is in every edition.

> **Note.** Demos 01, 02 and 04 compile and run with every edition. Demos 03 and 05 link items to actions, so they need a Registered edition or the Trial. With the Community edition you can still read their sections: chapter 6 shows how to get the same result with `OnItemClick`.

Where the demos are

The demos are installed with the product, in `Demos\Toolbar\Delphi` under the sgcUI folder:

Folder	What is in it
`00.QuickStart`	The five demos of this guide, from `01.HelloToolbar` to `05.ToolbarNotepad`, and a `readme.html` that lists them.
`EditorShell`, `LiveStatus`, `RibbonShell`	Larger VCL demos: an editor shell with the overflow menu, a status bar fed by a worker thread, and a ribbon editor.
`EditorShellFMX`, `RibbonShellFMX`	The FireMonkey versions of the editor shell and the ribbon editor.

Chapter 7 describes the larger demos one by one.

Running your first demo

1. In Delphi, choose **File** > **Open Project** and open `Demos\Toolbar\Delphi\00.QuickStart\01.HelloToolbar\HelloToolbar.dpr`.
2. Press **F9**. The demo compiles and a small window opens with a toolbar of three buttons: New, Open and Save.
3. Click a button. The text in the middle of the window shows its name.
4. Rest the mouse on a button to see its tooltip, then press the arrow keys and Enter: the toolbar has the keyboard focus and moves a focus frame between the buttons.

If Delphi reports that it cannot find `sgcToolbar_VCL.dcu` or a similar unit, the library path does not contain the sgcUI `Source` folder. Chapter 6 explains the fix.

> **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

sgcToolbar is small, and a handful of ideas explain almost everything the demos do. This chapter goes through them in the order you meet them. Each idea comes back in one of the demos of chapter 4.

The items are a collection

The buttons of a `TsgcVCLToolbar` live in its `Items` property, a collection of `TsgcToolbarItem`. At design time you edit it with the collection editor: click the ellipsis button of `Items` in the Object Inspector and use **Add New**. Each item then shows its own properties in the Object Inspector.

In code, `Toolbar.Items[i]` returns an item by its position, `Toolbar.Items.Count` is the number of items and `Toolbar.Items.Add` creates a new one at the end. This method builds three items at run time:

```
procedure TfrmMain.BuildToolbar;
var
  oItem: TsgcToolbarItem;
begin
  oItem := Toolbar.Items.Add;
  oItem.Caption := 'Refresh';
  oItem.Glyph := 'R';
  oItem.Hint := 'Reload the data';

  oItem := Toolbar.Items.Add;
  oItem.Kind := tikSeparator;

  oItem := Toolbar.Items.Add;
  oItem.Kind := tikToggle;
  oItem.Caption := 'Live';
  oItem.Glyph := 'L';
  oItem.Tag := 1;

  Toolbar.OnItemClick := ToolbarItemClick;
end;
```

Since an item is not a component, it has no name in the form class. You tell items apart by their `Index` (the demos keep constants such as `CS_ITEM_BOLD = 0`), by their `Tag`, or by their `Caption`. Changing any property of an item at run time repaints the bar; demo 02 changes the caption of a dropdown button after a menu choice.

The icon of an item is either its `Glyph`, a short text such as `'N'` drawn in the icon slot, or an image: assign a `TImageList` to the `Images` property of the toolbar and set `ImageIndex` on the items. The QuickStart demos use glyphs only, so they need no image files.

What an item can be: Kind

The `Kind` property of an item decides what it is and how it behaves:

Kind	What it is	Used in
`tikButton`	A push button, the default kind	Demo 01
`tikToggle`	A button that stays pressed; `Down` tells whether it is pressed	Demos 02 to 05
`tikDropdown`	A button with an arrow that opens its `DropdownMenu`, an ordinary `TPopupMenu`	Demos 02 and 05
`tikSeparator`	A thin vertical line between two groups of items	Demos 02 to 05
`tikLabel`	Plain text that cannot be clicked	Demo 02
`tikSpacer`	An empty space that takes the free width, so the items after it sit at the right end of the bar	Demos 02 to 05

Any clickable item can also carry a `Badge`, a short text drawn as a small pill on the icon, typically a count such as `'3'`. Set it to an empty string to hide it. `BadgeColor` changes its colour; 0, the default, uses the accent colour of the bar.

Toggles and radio groups

A `tikToggle` item flips its `Down` property on every click, like the Bold button of a word processor. Give several toggles the same `GroupIndex` above 0 and they become a radio group: pressing one releases the others, and a click on the pressed one does not release it, so exactly one stays down. A toggle with `GroupIndex = 0` is independent.

When your handler runs, `Down` already has its new value. Demo 02 relies on that: its handler simply reads the `Down` state of every toggle and applies it to a memo. To set the state from code, assign `Down`; this is also the only way to release every item of a radio group.

One event for every item

A click on an item is reported in up to two places, from the general to the specific:

1. `OnItemClick` of the toolbar fires for every item that is clicked, with the item as the `aItem` parameter. For a toggle it fires after `Down` has changed, for a dropdown button before its menu is shown.
2. Then the item runs its own command: its `OnClick` event when one is assigned, otherwise the `OnExecute` of its action, if it has one.

For most toolbars one `OnItemClick` handler is all you need. The item `OnClick` is handy for an item that does something unrelated to the others; you assign it in the Object Inspector after selecting the item in the collection editor. Disabled items (`Enabled = False`) are drawn dimmed and do not react, and separators, labels and spacers are never clicked.

Items linked to actions

Delphi applications often keep their commands in a `TActionList`, so the menu, the toolbar and the shortcuts share one `OnExecute` and one `OnUpdate`. A toolbar item has an `Action` property for that. An item linked to an action takes its `Caption`, `Hint`, `Enabled`, `Visible` and `ImageIndex` from the action, and a toggle takes its

`Down` state from the `Checked` property of the action. A click on the item runs the action, unless the item has its own `OnClick`.

The `OnUpdate` events of the actions are what keep the buttons up to date: Delphi calls them when the application is idle, and the action enables or disables itself. The toolbar takes part in these updates, so the `OnUpdate` of an action used only by the toolbar runs too.

The `Action` property belongs to the Registered editions, see chapter 2.

The status bar and its panels

`TsgcVCLStatusBar` aligns itself to the bottom of the form and shows the panels of its `Panels` collection from left to right. Each `TsgcStatusBarPanel` has a `Kind`:

- `spkText`, the default: the panel shows its `Text`, aligned by `Alignment`.
- `spkProgress`: a progress bar filled to `Progress`, a value from 0 to 100.
- `spkIndicator`: a coloured dot in front of the text. `IndicatorState` is `sisOff`, `sisOk` (green), `sisWarning` (amber) or `sisError` (red).

The width of a panel comes from one of three settings. `Width` is a fixed width in logical pixels, 100 by default. `AutoSize` makes the panel as wide as its text. `Spring` panels share whatever width the other panels leave, so a message panel with `Spring = True` fills the bar. The same can be set up in code:

```
procedure TfrmMain.BuildStatusBar;
var
  oPanel: TsgcStatusBarPanel;
begin
  oPanel := StatusBar.Panels.Add;
  oPanel.Text := 'Ready';
  oPanel.Spring := True;

  oPanel := StatusBar.Panels.Add;
  oPanel.Kind := spkProgress;
  oPanel.Width := 150;

  oPanel := StatusBar.Panels.Add;
  oPanel.Kind := spkIndicator;
  oPanel.Text := 'Online';
  oPanel.Width := 100;
  oPanel.IndicatorState := sisOff;
end;
```

To change a panel while the program runs, use the methods of the status bar, which take the index of the panel: `SetPanelText(aIndex, aText)`, `SetProgress(aIndex, aValue)`, `SetIndicator(aIndex, aState)` and `SetPanelVisible(aIndex, aValue)`. An index out of range is ignored. A click on a panel raises `OnPanelClick` with the panel as a parameter. The size grip at the right end, `SizeGrip = True` by default, resizes a form with a sizeable border.

Updating the status bar from a thread

A status bar often reports work done in the background: a download, an import, a long calculation. The four `Set...` methods of the previous section can be called from any thread, with no `Synchronize` or `Queue`. The status bar stores the new value under a lock and asks its window to repaint, so the VCL is only touched in the

main thread. This worker thread reports a simulated download on a status bar with a text panel, a progress panel and an indicator:

```
constructor TDownloadWorker.Create(aStatusBar: TsgcVCLStatusBar);
begin
  FStatusBar := aStatusBar;
  FreeOnTerminate := True;
  inherited Create(False);
end;

procedure TDownloadWorker.Execute;
var
  i: Integer;
begin
  for i := 1 to 50 do
  begin
    if Terminated then
      Exit;
    Sleep(100);
    // no Synchronize: these three setters are thread-safe
    FStatusBar.SetPanelText(0, Format('Chunk %d of 50', [i]));
    FStatusBar.SetProgress(1, i * 2);
  end;
  FStatusBar.SetIndicator(2, sisOk);
end;
```

Only these four methods are thread safe. Every other property, and adding or deleting panels, belongs in the main thread, and a worker must stop before the form that owns the status bar is destroyed. The toolbar itself is main thread only. The `LiveStatus` demo, listed in chapter 7, is a complete example.

Themes and the accent colour

The colours of both bars come from the sgcUICore theme. Each bar has a `Theme` property:

- `utDefault`, the default value, means “paint with the application theme”. Change the application theme and every sgcUI control that keeps `utDefault` follows.
- `utLight`, `utDark` and `utHighContrast` fix the theme of that one bar, whatever the application theme is.

The application theme is set with two procedures of the unit `sgcUICore_Theme`. `sgcUISetApplicationThemePreset` chooses the shape and the colours, `CS_UI_THEME_CLASSIC` or `CS_UI_THEME_FLUENT` (rounder corners, the Windows 11 look). `sgcUISetApplicationTheme` chooses `utLight` or `utDark`; with `True` as the second parameter it also themes the standard VCL controls of the forms, such as panels, labels and combo boxes. That second part is done by the unit `sgcUICore_Theme_VCL`, so add it to the `uses` clause of one unit of your project, as demos 04 and 05 do.

`AccentColor` replaces the accent colour of the theme on one bar: the tint of a pressed toggle, the focus frame and the badges on the toolbar, the progress bars on the status bar. Like every colour in sgcUI it is an ARGB value written as `$AARRGGBB`, and 0, the default, uses the accent of the theme. A fixed theme and accent for both bars looks like this:

```
procedure TfrmMain.SetDarkBars;
begin
  Toolbar.Theme := utDark;
  StatusBar.Theme := utDark;
  // $AARRGGBB, 0 (the default) uses the accent of the theme
  Toolbar.AccentColor := $FF2E7D32;
  StatusBar.AccentColor := $FF2E7D32;
end;
```

Note the leading `$FF`: it is the alpha channel, fully opaque. A colour written as `$2E7D32` has an alpha of 0 and is transparent.

Size, captions and the keyboard

- **Height.** The toolbar calculates its own height from `ButtonSize` (32 by default) and `Padding` (6 by default): `ButtonSize + 2 * Padding`, scaled to the screen DPI. Any other height you set is ignored, so change `ButtonSize` instead. `Spacing` is the gap between items.
- **Captions.** `ShowCaptions = False` shows only the icons. An item whose `Hint` is empty then shows its `Caption` as the tooltip, so the user can still find out what an icon does. Tooltips appear only when `ShowHint` is `True`.
- **Keyboard.** `TabStop` is `True` by default, so the toolbar takes part in the tab order. When it has the focus, Left, Right, Home and End move a focus frame between the buttons and Enter or Space clicks the focused one. In an editor you usually want the focus to stay in the text, so demos 02 to 05 set `TabStop = False`.

Summary: who does what

You want to	Use
Add a button	An item in `Toolbar.Items`, with `Caption`, `Glyph` and `Hint`
React to clicks	`Toolbar.OnItemClick`, or the `OnClick` of one item
Make an on and off button, or a radio group	`Kind = tikToggle`, `Down`, `GroupIndex`
Open a menu from a button	`Kind = tikDropdown` and `DropdownMenu`
Push items to the right	A `tikSpacer` item in front of them
Share commands with a menu	The item `Action` (Registered editions)
Show text, progress or a state at the bottom	Panels in `StatusBar.Panels`, changed with `SetPanelText`, `SetProgress` and `SetIndicator`
Report from a worker thread	The same `Set...` methods, no `Synchronize`
Change the look of the whole application	`sgcUISetApplicationThemePreset` and `sgcUISetApplicationTheme`, plus `sgcUICore_Theme_VCL` in the uses clause
Change the look of one bar	`Theme`, `AccentColor`, `ShowCaptions`, `ButtonSize`

4. The QuickStart demos

The five demos build on each other. Run them in order, and read each section with the demo open in Delphi. Every section has the same parts: what you will build, the components on the form, the code, how it works and a few things to try.

Demo	Project	New ideas
01.HelloToolbar	`HelloToolbar.dpr`	Items set at design time, `Caption`, `Glyph` and `Hint`, one `OnItemClick` for all items
02.ToolbarItemKinds	`ToolbarItemKinds.dpr`	Every item kind, toggles, a radio group, a dropdown menu, a badge, changing an item at run time
03.ToolbarActions	`ToolbarActions.dpr`	Items linked to actions shared with a main menu, `OnUpdate`, the item `OnClick`, the status bar
04.ToolbarThemes	`ToolbarThemes.dpr`	Application theme and preset, `AccentColor`, `ShowCaptions`, `ButtonSize`, progress and indicator panels
05.ToolbarNotepad	`ToolbarNotepad.dpr`	Everything together in a small text editor

> **Note.** The demos need sgcToolbar installed, so the components appear on the SGC Toolbar palette page and the `sgcUI\Source` folder is on the library path. Demos 01, 02 and 04 use only features that are part of every edition. Demos 03 and 05 use the `Action` property of the items and need a Registered edition or the Trial.

4.1 Demo 01: Hello Toolbar

Folder `01.HelloToolbar`, project `HelloToolbar.dpr`.

What you will build

The smallest sgcToolbar application: a window with a toolbar of three buttons, New, Open and Save, and a label that shows the name of the last button clicked.

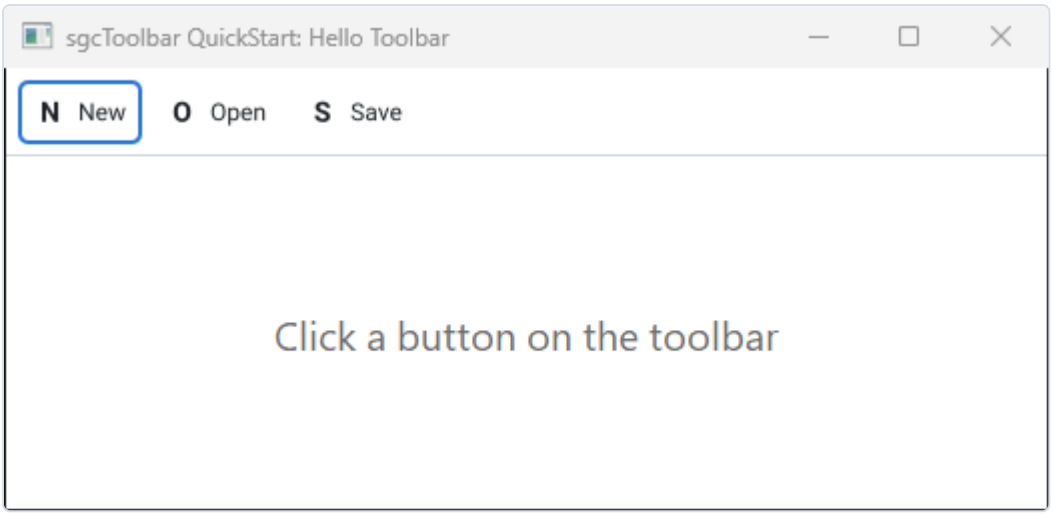


Figure 4.1. Demo 01. The letters N, O and S are the `Glyph` of each item, drawn in the icon slot. The frame around New is the keyboard focus.

Components on the form

Component	Class	Properties you set
`Toolbar`	`TsgcVCLToolbar`	`Items`: New, Open and Save, with `Glyph` N, O and S and a `Hint` each. `ShowHint = True`. `OnItemClick = ToolbarItemClick`.
`lblInfo`	`TLabel`	`Align = alClient`, centred text.

The toolbar has no `Align` line in the form file because `alTop` is its default, and its height of 44 pixels is not chosen by hand: it is `ButtonSize` (32) plus twice `Padding` (6). The items are the same three you saw in chapter 1:

```
Items = <
  item
    Caption = 'New'
    Glyph = 'N'
    Hint = 'Create a new document'
  end
  item
    Caption = 'Open'
    Glyph = 'O'
    Hint = 'Open a document'
  end
  item
    Caption = 'Save'
    Glyph = 'S'
    Hint = 'Save the document'
  end>
ShowHint = True
OnItemClick = ToolbarItemClick
```

The code

The `uses` clause names the unit of the controls and the unit of the item class, `TsgcToolbarItem`:

```
uses
  Classes, Controls, Forms, StdCtrls,
  // sgc
  sgcToolbar_Model, sgcToolbar_VCL;
```

And this is the only event handler of the demo:

```
procedure TfrmHelloToolbar.ToolbarItemClick(Sender: TObject;
  aItem: TsgcToolbarItem);
begin
  { aItem is the TsgcToolbarItem that was clicked: its Caption, its Index
    in Toolbar.Items or its Tag tell the buttons apart. }
  lblInfo.Caption := 'You clicked ' + aItem.Caption;
end;
```

How it works

1. The form loads from its DFM. The toolbar reads its three items, aligns itself to the top and measures each button from its glyph and caption.
2. The toolbar is the only control of the form that can take the focus, so it starts focused and draws a focus frame around its first button.
3. When you click a button, or press Enter or Space on the focused one, the toolbar raises `OnItemClick` with that item.
4. The handler reads `aItem.Caption` and shows it in the label.

Try this

- Add a fourth item in the collection editor of `Items`, Caption Print, Glyph P. Run again: it works with no new code, because `OnItemClick` serves every item.
- Set `ShowCaptions` to `False`. Only the glyphs remain. Clear the `Hint` of one item and rest the mouse on it: its caption becomes the tooltip.
- Set `Enabled` of the Save item to `False`. It is drawn dimmed and ignores clicks.

4.2 Demo 02: Toolbar Item Kinds

Folder `02.ToolbarItemKinds`, project `ToolbarItemKinds.dpr`.

What you will build

A formatting toolbar above a memo, with every kind of item: Bold, Italic and Underline toggles, a Left, Center and Right radio group, a Size dropdown button with a popup menu, a label, a spacer and a Comments button with a badge. The toolbar formats the text of the memo.

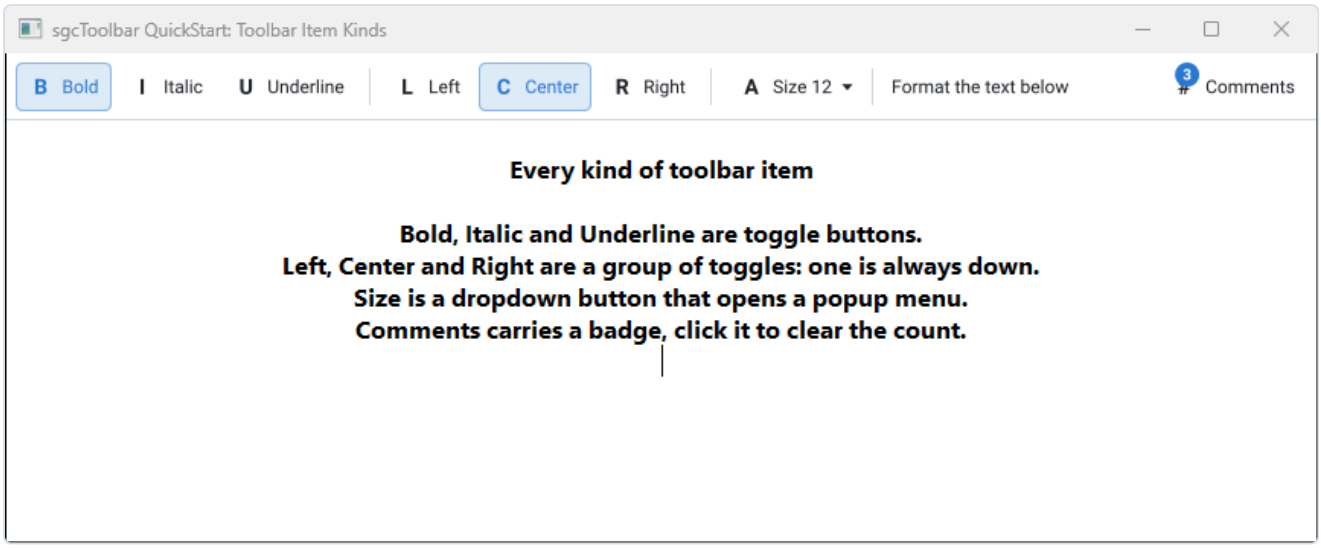


Figure 4.2. Demo 02. Bold and Center are down. The spacer pushes the Comments button, with its badge, to the right end.

Components on the form

Component	Class	Properties you set
`Toolbar`	`TsgcVCLToolbar`	Thirteen items (see below). `ShowHint = True`, `TabStop = False`, `OnItemClick = ToolbarItemClick`.
`memText`	`TMemo`	`Align = alClient`, a few lines of text.
`pmSize`	`TPopupMenu`	Four radio menu items, Size 10 to Size 20, with the size in `Tag` and `OnClick = mnuSizeClick`.

The items show one kind after the other. Here is an excerpt of the collection with one item of each new kind:

```
      item
        Kind = tikToggle
        Caption = 'Bold'
        Glyph = 'B'
        Hint = 'Bold'
        Down = True
      end
      // ...
      item
        Kind = tikToggle
        Caption = 'Center'
        Glyph = 'C'
        Hint = 'Center'
        Down = True
        GroupIndex = 1
      end
      // ...
      item
        Kind = tikDropdown
        Caption = 'Size 12'
        Glyph = 'A'
        Hint = 'Font size'
        DropdownMenu = pmSize
      end
      item
        Kind = tikSeparator
      end
      item
        Kind = tikLabel
        Caption = 'Format the text below'
      end
      item
        Kind = tikSpacer
      end
      item
        Caption = 'Comments'
        Glyph = '#'
        Hint = 'Three new comments, click to clear the badge'
        Badge = '3'
      end>
```

`TabStop = False` keeps the keyboard focus in the memo when you click a button, as in most editors.

The code

The items have no names, so the demo keeps their positions in constants:

```
const
  { the position of the items in Toolbar.Items }
  CS_ITEM_BOLD = 0;
  CS_ITEM_ITALIC = 1;
  CS_ITEM_UNDERLINE = 2;
  CS_ITEM_LEFT = 4;
  CS_ITEM_CENTER = 5;
  CS_ITEM_RIGHT = 6;
  CS_ITEM_SIZE = 8;
  CS_ITEM_COMMENTS = 12;
```

`DoApplyFormat` reads the `Down` state of the toggles and applies it to the memo. `FormCreate` calls it once, so the memo matches the items that are down in the DFM:

```
procedure TfrmToolbarItemKinds.FormCreate(Sender: TObject);
begin
  { Bold and Center are Down in the DFM: format the memo to match. }
  DoApplyFormat;
end;


procedure TfrmToolbarItemKinds.DoApplyFormat;
var
  vStyle: TFontStyles;
begin
  { the toggles: one font style each }
  vStyle := [];
  if Toolbar.Items[CS_ITEM_BOLD].Down then
    Include(vStyle, fsBold);
  if Toolbar.Items[CS_ITEM_ITALIC].Down then
    Include(vStyle, fsItalic);
  if Toolbar.Items[CS_ITEM_UNDERLINE].Down then
    Include(vStyle, fsUnderline);
  memText.Font.Style := vStyle;

  { the radio group: exactly one of the three is Down }
  if Toolbar.Items[CS_ITEM_CENTER].Down then
    memText.Alignment := taCenter
  else if Toolbar.Items[CS_ITEM_RIGHT].Down then
    memText.Alignment := taRightJustify
  else
    memText.Alignment := taLeftJustify;
end;
```

The `OnItemClick` handler clears the badge of the Comments button and reformats the memo for every other item:

```
procedure TfrmToolbarItemKinds.ToolbarItemClick(Sender: TObject;
  aItem: TsgcToolbarItem);
begin
  { the Comments button: a click clears its badge }
  if aItem.Index = CS_ITEM_COMMENTS then
    aItem.Badge := ''
  else
    DoApplyFormat;
end;
```

The four menu items of `pmSize` share one handler, which also writes the chosen size into the caption of the dropdown button:

```
procedure TfrmToolbarItemKinds.mnuSizeClick(Sender: TObject);
var
  oMenuItem: TMenuItem;
begin
  { the four menu items share this handler, their Tag holds the size }
  oMenuItem := Sender as TMenuItem;
  oMenuItem.Checked := True;
  memText.Font.Size := oMenuItem.Tag;
  { an item can be changed at run time like any other property }
  Toolbar.Items[CS_ITEM_SIZE].Caption := oMenuItem.Caption;
end;
```

How it works

1. A click on Bold flips its `Down` state, then raises `OnItemClick`. The handler calls `DoApplyFormat`, which sees the new state and changes the font of the memo.
2. A click on Right presses it and releases Center, because both have `GroupIndex = 1`. A second click on Right does nothing: a radio toggle can only be switched on.
3. A click on the Size button raises `OnItemClick` first (the handler reformats, harmlessly) and then opens `pmSize` below the button. Choosing a size runs `mnuSizeClick`.
4. Assigning `Caption` of the Size item repaints the toolbar; the button grows or shrinks to fit the new text.
5. A click on Comments sets its `Badge` to an empty string, and the pill disappears.

Try this

- Give Bold and Italic `GroupIndex = 2`. They now exclude each other, like the alignment buttons.
- Set `BadgeColor` of the Comments item to `$FFD32F2F`, an opaque red in ARGB notation.
- Set `Width` of the spacer to 40. For a spacer, `Width` is the minimum width: it still stretches, but when you make the window narrow the gap before Comments never drops below 40 pixels.

4.3 Demo 03: Toolbar Actions

Folder `03.ToolbarActions`, project `ToolbarActions.dpr`.

> **Edition.** This demo links items to actions through the `Action` property, which is part of the Registered editions. It does not compile with the Community edition.

What you will build

An Edit menu and a toolbar that share the same five actions: Cut, Copy, Paste, Select All and Word Wrap. Cut and Copy are enabled only while some text is selected, Paste only while the clipboard holds text, and the toolbar button and the menu item of each command always agree. A `TsgcVCLStatusBar` at the bottom shows the last command and the size of the selection.

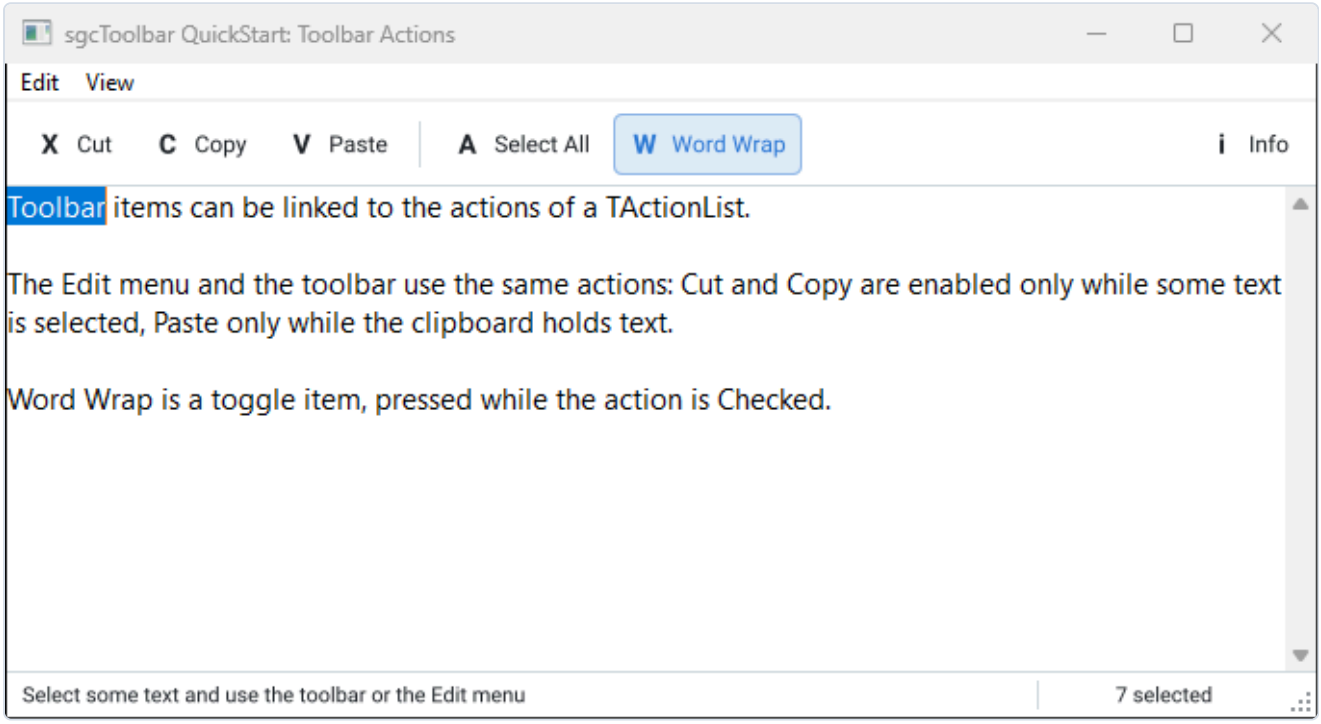


Figure 4.3. Demo 03. Word Wrap is down because its action is `Checked`. The right panel of the status bar shows the size of the selection.

Components on the form

Component	Class	Properties you set
`ActionList`	`TActionList`	`actCut`, `actCopy`, `actPaste`, `actSelectAll` with shortcuts, and `actWordWrap` with `Checked = True`. Cut and Copy share `OnUpdate = actSelectionUpdate`, Paste has `OnUpdate = actPasteUpdate`.
`MainMenu`	`TMainMenu`	Edit and View menus whose items use the same actions.
`Toolbar`	`TsgcVCLToolbar`	Items with `Action` set; Word Wrap is a `tikToggle`. An Info item without action, with `OnClick = ToolbarInfoClick`. `TabStop = False`, `OnItemClick = ToolbarItemClick`.
`StatusBar`	`TsgcVCLStatusBar`	Two panels: a message panel with `Spring = True`, and a centred panel of 130 pixels.
`memText`	`TMemo`	`Align = alClient`, the form's `ActiveControl`.

The toolbar items name their action. The Info item has no action, but an `OnClick` of its own, assigned in the Object Inspector after selecting the item in the collection editor:

```
      item
        Caption = 'Cut'
        Glyph = 'X'
        Hint = 'Cut the selection'
        Action = actCut
      end
      // ...
      item
        Kind = tikToggle
        Caption = 'Word Wrap'
        Glyph = 'W'
        Hint = 'Wrap the long lines'
        Down = True
        Action = actWordWrap
      end
      item
        Kind = tikSpacer
      end
      item
        Caption = 'Info'
        Glyph = 'i'
        Hint = 'An item with its own OnClick event'
        OnClick = ToolbarInfoClick
      end>
```

The status bar has two panels. The second one has no text in the DFM, the code fills it:

```
  object StatusBar: TsgcVCLStatusBar
    // ...
    Panels = <
      item
        Text = 'Select some text and use the toolbar or the Edit menu'
        Spring = True
      end
      item
        Width = 130
        Alignment = taCenter
      end>
  end
```

The code

`FormCreate` selects the first word of the memo:

```
procedure TfrmToolbarActions.FormCreate(Sender: TObject);
begin
  // ...
  { select the first word, so Cut and Copy start enabled }
  memText.SelStart := 0;
  memText.SelLength := 7;
end;
```

The `OnExecute` handlers are one line each. Word Wrap flips its own `Checked` property, which the menu shows as a check mark and the toolbar as the `Down` state of the toggle:

```
procedure TfrmToolbarActions.actCutExecute(Sender: TObject);
begin
  memText.CutToClipboard;
end;
// ...

procedure TfrmToolbarActions.actWordWrapExecute(Sender: TObject);
begin
  { Checked drives the menu check mark and the Down state of the toggle }
  actWordWrap.Checked := not actWordWrap.Checked;
  memText.WordWrap := actWordWrap.Checked;
end;
```

The `OnUpdate` handlers enable the commands. `actSelectionUpdate` also writes the size of the selection into panel 1 of the status bar:

```
procedure TfrmToolbarActions.actSelectionUpdate(Sender: TObject);
begin
  { the OnUpdate of Cut and Copy: only with some selected text. It also
    keeps the selection panel of the status bar up to date. }
  (Sender as TAction).Enabled := memText.SelLength > 0;
  StatusBar.SetPanelText(CS_PANEL_SELECTION,
    Format('%d selected', [memText.SelLength]));
end;


procedure TfrmToolbarActions.actPasteUpdate(Sender: TObject);
begin
  actPaste.Enabled := Clipboard.HasFormat(CF_TEXT);
end;
```

Finally the two toolbar events. `OnItemClick` fires for every item, the Info item's `OnClick` only for that item:

```
procedure TfrmToolbarActions.ToolbarItemClick(Sender: TObject;
  aItem: TsgcToolbarItem);
begin
  { fires before the action runs, for every item of the toolbar }
  StatusBar.SetPanelText(CS_PANEL_MESSAGE, 'Toolbar: ' + aItem.Caption);
end;


procedure TfrmToolbarActions.ToolbarInfoClick(Sender: TObject);
begin
  { the OnClick of the Info item, assigned in Toolbar.Items in the
    Object Inspector }
  StatusBar.SetPanelText(CS_PANEL_MESSAGE,
    'The menu and the toolbar share the same five actions');
end;
```

How it works

1. When the form loads, every linked item copies the `Caption`, `Hint`, `Enabled` and `Visible` of its action, and Word Wrap copies `Checked` into `Down`.
2. Whenever the application is idle, Delphi calls the `OnUpdate` of the actions. Clear the selection and Cut and Copy become disabled, on the toolbar and in the menu at the same time.
3. A click on Cut raises `OnItemClick`, which writes “Toolbar: Cut” in the status bar, and then runs `actCutExecute`. Choosing Cut in the menu or pressing Ctrl+X runs the same action, without `OnItemClick`.

4. A click on Info raises `OnItemClick` and then the item's own `OnClick`, which overwrites the message.
5. `SetPanelText` changes the text of a panel by its index; the constants `CS_PANEL_MESSAGE = 0` and `CS_PANEL_SELECTION = 1` name the panels.

Try this

- Give `actWordWrap` the `ShortCut` Ctrl+W. Press it: the menu check mark and the toolbar toggle follow.
- In `actPasteUpdate`, also set `actPaste.Visible := Clipboard.HasFormat(CF_TEXT)`. Copy nothing, and the Paste button disappears from the toolbar.
- Handle `OnPanelClick` of the status bar, `procedure(Sender: TObject; aPanel: TsgcStatusBarPanel)`, and show `aPanel.Index` in the message panel.

4.4 Demo 04: Toolbar Themes

Folder `04.ToolbarThemes`, project `ToolbarThemes.dpr`.

What you will build

A toolbar and a status bar that you restyle while the program runs. A combo box switches the application theme between Classic and Fluent, light and dark; another one changes the accent colour; two check boxes hide the captions and make the buttons larger. The status bar shows the three kinds of panels: text, progress and indicator.

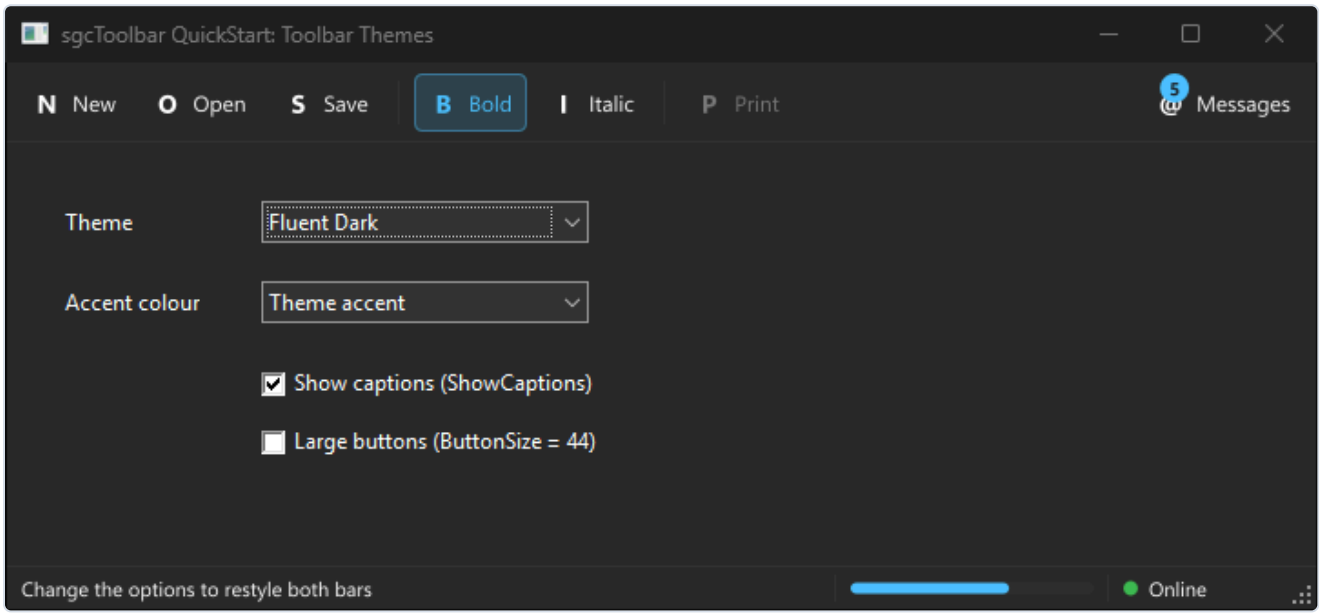


Figure 4.4. Demo 04 starts in Fluent Dark. The standard panel, labels and combo boxes are dark too, because the unit `sgcUICore_Theme_VCL` is in the uses clause.

Components on the form

Component	Class	Properties you set
`Toolbar`	`TsgcVCLToolbar`	Sample items, including a disabled Print button and a Messages button with `Badge = '5'`. `TabStop = False`. `Theme` keeps its default, `utDefault`.
`StatusBar`	`TsgcVCLStatusBar`	A `Spring` text panel, an `spkProgress` panel with `Progress = 65` and `Width = 150`, and an `spkIndicator` panel "Online" with `IndicatorState = sisOk` and `Width = 100`.
`pnlOptions`	`TPanel`	Holds `cboTheme` (four presets, Fluent Dark selected), `cboAccent`, `chkCaptions` and `chkLarge`.

The panels of the status bar, with their `Kind`:

```
    Panels = <
      item
        Text = 'Change the options to restyle both bars'
        Spring = True
      end
      item
        Kind = spkProgress
        Width = 150
        Progress = 65
      end
      item
        Kind = spkIndicator
        Text = 'Online'
        Width = 100
        IndicatorState = sisOk
      end>
```

The code

The `uses` clause adds three sgcUICore units: `sgcUICore_Canvas` for the colour type `TsgcUIColor`, `sgcUICore_Theme` for the theme procedures, and `sgcUICore_Theme_VCL`, which themes the standard VCL controls:

```
uses
  Classes, Controls, Forms, StdCtrls, ExtCtrls,
  // sgc
  sgcUICore_Canvas, sgcUICore_Theme, sgcUICore_Theme_VCL, sgcToolbar_Model,
  sgcToolbar_VCL;
```

The accent colours are ARGB values. The first one, 0, means “the accent of the theme”:

```
const
  { the accents of cboAccent, ARGB: Default, Green, Orange and Purple }
  CS_ACCENTS: array [0 .. 3] of TsgcUIColor = (0, $FF2E7D32, $FFE65100,
    $FF7B1FA2);
```

The theme combo box sets the preset and the light or dark theme of the whole application. `FormCreate` calls it once, so the theme selected in the DFM is applied at start:

```
procedure TfrmToolbarThemes.FormCreate(Sender: TObject);
begin
  { the DFM selects Fluent Dark, apply it now }
  cboThemeChange(nil);
end;


procedure TfrmToolbarThemes.cboThemeChange(Sender: TObject);
begin
  { the items of the combo box: 0 Classic Light, 1 Classic Dark,
    2 Fluent Light, 3 Fluent Dark }
  if cboTheme.ItemIndex >= 2 then
    sgcUISetApplicationThemePreset(CS_UI_THEME_FLUENT)
  else
    sgcUISetApplicationThemePreset(CS_UI_THEME_CLASSIC);
  if Odd(cboTheme.ItemIndex) then
    sgcUISetApplicationTheme(utDark, True)
  else
    sgcUISetApplicationTheme(utLight, True);
end;
```

The other three options are plain properties of the bars:

```
procedure TfrmToolbarThemes.cboAccentChange(Sender: TObject);
begin
  Toolbar.AccentColor := CS_ACCENTS[cboAccent.ItemIndex];
  StatusBar.AccentColor := CS_ACCENTS[cboAccent.ItemIndex];
end;


procedure TfrmToolbarThemes.chkCaptionsClick(Sender: TObject);
begin
  Toolbar.ShowCaptions := chkCaptions.Checked;
end;


procedure TfrmToolbarThemes.chkLargeClick(Sender: TObject);
begin
  if chkLarge.Checked then
    Toolbar.ButtonSize := 44
  else
    Toolbar.ButtonSize := 32;
end;
```

How it works

1. Both bars keep `Theme = utDefault`, so they paint with the application theme.
2. `sgcUISetApplicationThemePreset` changes the preset, the shapes and colours of every sgcUI control. `sgcUISetApplicationTheme` switches between light and dark and repaints every sgcUI control that follows the application theme.
3. The `True` parameter also themes the standard controls of the open forms. That work is done by `sgcUICore_Theme_VCL`; without that unit in the uses clause the panel, the labels and the combo boxes would stay light.
4. `AccentColor` changes the tint of the pressed Bold toggle, the focus frame and the badge of Messages on the toolbar, and the progress bar on the status bar. An item with its own `BadgeColor` keeps that colour for its badge.

5. `ButtonSize = 44` makes the toolbar taller: its height is always `ButtonSize + 2 * Padding`.

Try this

- Set `StatusBar.Theme := utLight` in `FormCreate`. Switch to a dark theme: the toolbar follows, the status bar stays light.
- Call `sgcUISetApplicationAccentColor($FF0078D4)` from `sgcUICore_Theme`. It changes the accent of the application theme, so every control with `AccentColor = 0` picks it up.
- Add a button that calls `StatusBar.SetProgress(1, 100)` and `StatusBar.SetIndicator(2, sisError)`: the bar fills and the dot turns red.

4.5 Demo 05: Toolbar Notepad

Folder `05.ToolbarNotepad`, project `ToolbarNotepad.dpr`.

> **Edition.** Like demo 03, this demo links items to actions, a feature of the Registered editions. It does not compile with the Community edition.

What you will build

A small text editor that puts the previous demos together. The toolbar has New, Open and Save, Cut, Copy and Paste, a Word Wrap toggle, a Zoom dropdown and, at the right end, a Dark toggle that switches the whole application between light and dark. The status bar shows the caret position, a message, the zoom level and an indicator that turns from Saved to Modified as soon as you type.

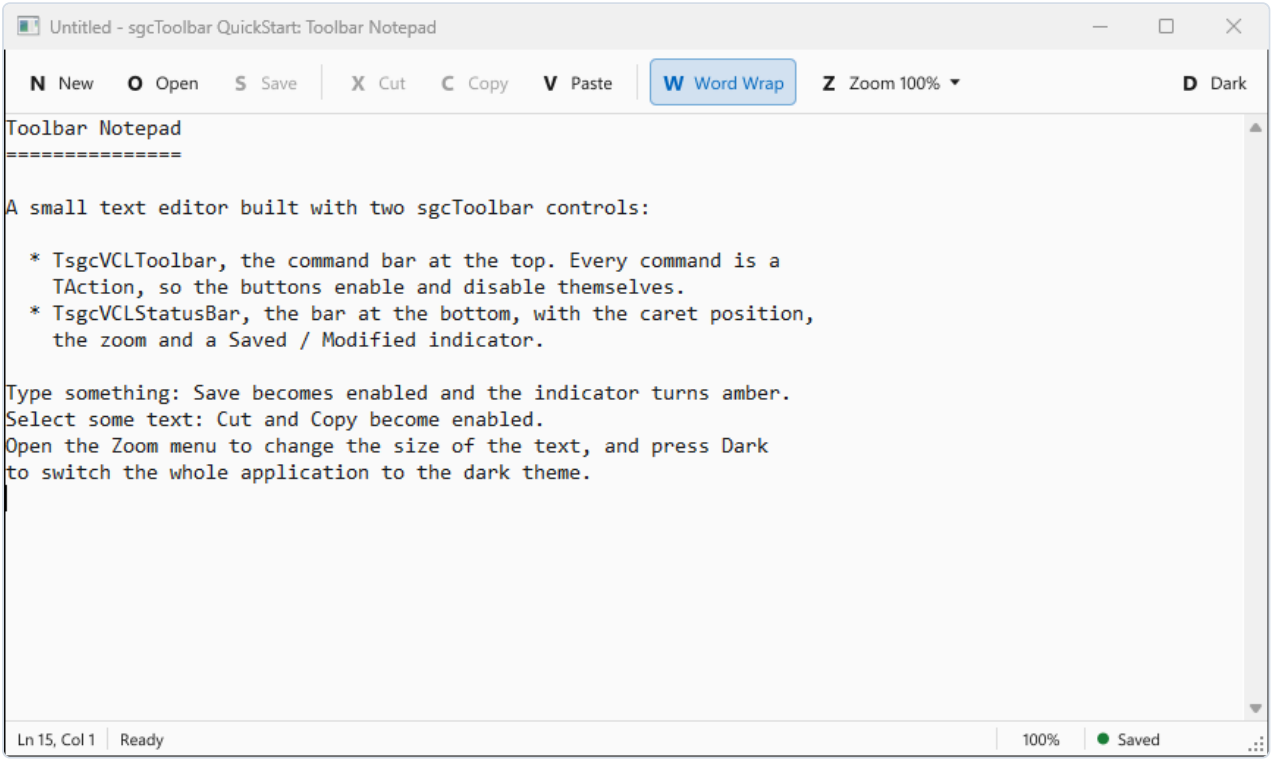


Figure 4.5. Demo 05 in the Fluent light theme. Save is dimmed because there is nothing new to save, and Cut and Copy because no text is selected.

Components on the form

Component	Class	Properties you set
`Toolbar`	`TsgcVCLToolbar`	Items linked to the actions; a `tikDropdown` Zoom item with `DropdownMenu = pmZoom`; a `tikSpacer`; a `tikToggle` Dark item with `Tag = 1`. `TabStop = False`, `OnItemClick = ToolbarItemClick`.
`StatusBar`	`TsgcVCLStatusBar`	Four panels: the caret (`AutoSize`), a message (`Spring`), the zoom (60 pixels, centred) and an `spkIndicator` panel.
`memText`	`TMemo`	`Align = alClient`, Consolas 11, some text in the DFM. `OnChange`, `OnClick` and `OnKeyUp` update the status bar.
`ActionList`	`TActionList`	`actNew`, `actOpen`, `actSave`, `actCut`, `actCopy`, `actPaste`, `actWordWrap`. Save, Cut, Copy and Paste have an `OnUpdate`.
`pmZoom`	`TPopupMenu`	75%, 100%, 125% and 150%, with the percentage in `Tag`.
`dlgOpen`, `dlgSave`	`TOpenDialog`, `TSaveDialog`	Filter for text files.

The last items of the toolbar: the Zoom dropdown, the spacer and the Dark toggle, which has no action and is recognised by its `Tag`:

```
      item
        Kind = tikDropdown
        Caption = 'Zoom 100%'
        Glyph = 'Z'
        Hint = 'Zoom the text'
        DropdownMenu = pmZoom
      end
      item
        Kind = tikSpacer
      end
      item
        Kind = tikToggle
        Caption = 'Dark'
        Glyph = 'D'
        Hint = 'Switch between the light and the dark theme'
        Tag = 1
      end>
```

The four panels of the status bar:

```
    Panels = <
      item
        Text = 'Ln 1, Col 1'
        AutoSize = True
      end
      item
        Text = 'Ready'
        Spring = True
      end
      item
        Text = '100%'
        Width = 60
        Alignment = taCenter
      end
      item
        Kind = spkIndicator
        Text = 'Saved'
        Width = 110
        IndicatorState = sisOk
      end>
```

The code

Constants name the panels, the position of the Zoom item and the `Tag` of the Dark toggle:

```
const
  { the panels of StatusBar.Panels }
  CS_PANEL_CARET = 0;
  CS_PANEL_MESSAGE = 1;
  CS_PANEL_ZOOM = 2;
  CS_PANEL_STATE = 3;
  { the position of the Zoom dropdown in Toolbar.Items }
  CS_ITEM_ZOOM = 9;
  { the Tag of the Dark toggle }
  CS_TAG_DARK = 1;
  { the font size of the memo at 100% }
  CS_BASE_FONT_SIZE = 11;
```

`FormCreate` selects the Fluent preset with the light theme and fills the status bar:

```
procedure TfrmToolbarNotepad.FormCreate(Sender: TObject);
begin
  // ...
  sgcUISetApplicationThemePreset(CS_UI_THEME_FLUENT);
  sgcUISetApplicationTheme(utLight, True);
  DoSetFileName('');
  DoSetSaved(True);
  DoUpdateCaret;
end;
```

Two helpers keep the status bar in step with the editor. `DoSetSaved` switches the indicator between a green Saved and an amber Modified; `DoUpdateCaret` writes the caret position:

```
procedure TfrmToolbarNotepad.DoSetSaved(aSaved: Boolean);
begin
  { the indicator panel: a green dot when saved, an amber one when not }
  memText.Modified := not aSaved;
  if aSaved then
  begin
    StatusBar.SetIndicator(CS_PANEL_STATE, sisOk);
    StatusBar.SetPanelText(CS_PANEL_STATE, 'Saved');
  end
  else
  begin
    StatusBar.SetIndicator(CS_PANEL_STATE, sisWarning);
    StatusBar.SetPanelText(CS_PANEL_STATE, 'Modified');
  end;
end;


procedure TfrmToolbarNotepad.DoUpdateCaret;
begin
  StatusBar.SetPanelText(CS_PANEL_CARET, Format('Ln %d, Col %d',
    [memText.CaretPos.Y + 1, memText.CaretPos.X + 1]));
end;
```

The Save action, and the `OnUpdate` that enables it only when the memo has changes:

```
procedure TfrmToolbarNotepad.actSaveExecute(Sender: TObject);
begin
  if FFileName = '' then
  begin
    if not dlgSave.Execute then
      Exit;
    DoSetFileName(dlgSave.FileName);
  end;
  memText.Lines.SaveToFile(FFileName);
  DoSetSaved(True);
  StatusBar.SetPanelText(CS_PANEL_MESSAGE, 'Saved ' + FFileName);
end;


procedure TfrmToolbarNotepad.actSaveUpdate(Sender: TObject);
begin
  { Save is enabled only when there is something new to save }
  actSave.Enabled := memText.Modified;
end;
```

The zoom menu changes the font size, the caption of the Zoom item and the zoom panel:

```
procedure TfrmToolbarNotepad.mnuZoomClick(Sender: TObject);
var
  oMenuItem: TMenuItem;
begin
  oMenuItem := Sender as TMenuItem;
  oMenuItem.Checked := True;
  memText.Font.Size := Round(CS_BASE_FONT_SIZE * oMenuItem.Tag / 100);
  Toolbar.Items[CS_ITEM_ZOOM].Caption := Format('Zoom %d%%', [oMenuItem.Tag]);
  StatusBar.SetPanelText(CS_PANEL_ZOOM, Format('%d%%', [oMenuItem.Tag]));
end;
```

The Dark toggle is handled in `OnItemClick`. The memo events refresh the indicator and the caret:

```
procedure TfrmToolbarNotepad.ToolbarItemClick(Sender: TObject;
  aItem: TsgcToolbarItem);
begin
  { the Dark toggle has no action: its Down state picks the theme }
  if aItem.Tag = CS_TAG_DARK then
  begin
    if aItem.Down then
      sgcUISetApplicationTheme(utDark, True)
    else
      sgcUISetApplicationTheme(utLight, True);
  end;
end;


procedure TfrmToolbarNotepad.memTextChange(Sender: TObject);
begin
  DoSetSaved(False);
  DoUpdateCaret;
end;
```

How it works

1. At start the memo is unmodified, so `actSaveUpdate` disables Save and the indicator shows a green Saved.
2. Typing fires `memTextChange`: the indicator turns amber and reads Modified, and at the next idle moment `actSaveUpdate` enables the Save button.
3. Save asks for a file name the first time, writes the file and sets the state back to Saved, which disables the button again.
4. A click on Dark flips its `Down` state; `OnItemClick` recognises it by its `Tag` and switches the application theme, so the toolbar, the status bar and, through `sgcUICore_Theme_VCL`, the memo turn dark together.
5. The spacer before Dark takes the free width, so Dark stays at the right end however wide the window is.

Try this

- Add a Find item with a `TFindDialog`: an action `actFind` with the shortcut Ctrl+F, and an item linked to it.
- Add a fifth panel that shows the number of characters, updated in `memTextChange` with `StatusBar.SetPanelText`.
- Replace the Dark toggle with a Theme dropdown whose menu offers Classic Light, Classic Dark, Fluent Light and Fluent Dark, as in demo 04.

5. Main components

The SGC Toolbar palette page is short, and two of its components cover a typical first application: the toolbar and the status bar. This chapter introduces them, then the badge link, the ribbon and the FireMonkey versions.

Every class has its own help topic, named after the class, such as `TsgcVCLToolbar.htm` or `TsgcToolbarItem.htm`. Each topic lists the properties, methods and events with their types and defaults.

Reference table

Class	Palette page	What it is for	Key members
`TsgcVCLToolbar`	SGC Toolbar	The VCL toolbar	`Items`, `Images`, `ShowCaptions`, `ButtonSize`, `Theme`, `AccentColor`, `OnItemClick`
`TsgcVCLStatusBar`	SGC Toolbar	The VCL status bar	`Panels`, `SizeGrip`, `Theme`, `AccentColor`, `SetPanelText`, `SetProgress`, `SetIndicator`, `OnPanelClick`
`TsgcToolbarBadgeLink`	SGC Toolbar	Shows a count from another component as the badge of an item	`Source`, `Target`, `ItemIndex`, `MaxCount`, `ShowZero`, `OnFormat`
`TsgcVCLRibbon`	SGC Toolbar	An Office style ribbon (Registered editions)	Tabs, groups, Quick Access Toolbar, KeyTips
`TsgcVCLRibbonBackstage`	SGC Toolbar	The File page of a ribbon (Registered editions)	Pages added with the New Page command
`TsgcFMXToolbar`, `TsgcFMXStatusBar`	SGC Toolbar (FireMonkey)	The FireMonkey toolbar and status bar	The same as the VCL bars
`TsgcFMXRibbon`, `TsgcFMXRibbonBackstage`	SGC Toolbar (FireMonkey)	The FireMonkey ribbon and backstage (Registered editions)	The same as the VCL ribbon
`TsgcToolbarItem`	none, an item of `Items`	One button, toggle, dropdown, separator, label or spacer	`Kind`, `Caption`, `Glyph`, `ImageIndex`, `Hint`, `Down`, `GroupIndex`, `Badge`, `DropdownMenu`, `Action`, `OnClick`
`TsgcStatusBarPanel`	none, an item of `Panels`	One panel of the status bar	`Kind`, `Text`, `Width`, `AutoSize`, `Spring`, `Progress`, `IndicatorState`

The toolbar

`TsgcVCLToolbar` is in the unit `sgcToolbar_VCL`; the item class, the item kinds and the panel enumerations are in `sgcToolbar_Model`, so a form that handles `OnItemClick` uses both. Beyond the properties of chapter 3, the toolbar has:

- `Images`, a `TCustomImageList`, for real icons through the `ImageIndex` of the items. The icon slot shows the image at `ImageIndex` when there is one, and the `Glyph` otherwise, so you can start with glyphs and add images later.
- The overflow menu (Registered editions): when the items do not fit, the last ones move into a menu behind a chevron button at the right end, and choosing an entry works exactly like a click.
- Right to left support: with a right to left `BiDiMode` the items are laid out from the right, and `MirrorGlyph` mirrors the icon of an item that points in a direction.
- Accessibility: the items are reported to screen readers with their caption, state and badge.

The status bar

`TsgcVCLStatusBar`, in the same unit, holds the panels of chapter 3. Remember the two rules that matter most: the width of a panel comes from `Width`, `AutoSize` or `Spring`, and the four `Set...` methods are the thread safe way to change a panel. A panel also has `Hint`, `Visible` and `Tag`; a hidden panel takes no width, and the `Spring` panels share what it leaves.

The badge link

`TsgcToolbarBadgeLink` is a non visual component that copies a number into the `Badge` of a toolbar item. Its `Source` is any component that publishes a count, such as the notification model of sgcNavigation; its `Target` is the toolbar and `ItemIndex` the item. A count above `MaxCount` (99 by default) shows as "99+", a count of 0 clears the badge unless `ShowZero` is `True`, and `OnFormat` has the last word on the text. You need it only when the number comes from another sgcUI product; for your own numbers, set `Badge` directly, as demo 02 does.

The ribbon

The Registered editions add `TsgcVCLRibbon`, an Office style ribbon built on the same renderer and themes: tabs and groups that adapt to the width of the window, split buttons and galleries, contextual tabs, KeyTips, a customizable Quick Access Toolbar, a minimized mode and a layout that can be saved and restored. `TsgcVCLRibbonBackstage` is the full window File page of the ribbon. The `RibbonShell` demo is the place to start, and the Ribbon overview help topic describes the model.

FireMonkey and beyond

`TsgcFMXToolbar` and `TsgcFMXStatusBar`, in `sgcToolbar_FMX`, have the same items, panels, properties and events as the VCL bars, from Delphi XE6. Dropdown buttons open a FireMonkey `TPopupMenu`. The `EditorShellFMX` and `RibbonShellFMX` demos show them.

Because the bars are drawn by a framework neutral renderer, a console application, a service or a test can also paint a toolbar or a status bar to a PNG or SVG image without any window. The Headless rendering help topic explains how.

6. Common problems

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

Actions and events

An item does not react to a click

What you see. Clicking an item does nothing, or it is drawn dimmed.

Why. A disabled item (`Enabled = False`) ignores clicks and the keyboard. An item linked to an action follows the `Enabled` property of the action, so the action's `OnUpdate` may be disabling it. Separators, spacers and labels are never clicked.

Fix. Check `Enabled` of the item, the `OnUpdate` of its action, and its `Kind` .

A toggle of a radio group cannot be switched off

What you see. Clicking the pressed toggle of a group leaves it pressed.

Why. A toggle with `GroupIndex` above 0 is a radio button: a click only switches it on.

Fix. Set `Down := False` from code, or use `GroupIndex = 0` for an independent toggle.

The handler reads the wrong item

What you see. After adding an item in the designer, `OnItemClick` reacts to the wrong buttons.

Why. Code that tells items apart by `Index` , like the constants of demos 02 and 05, depends on their order. Inserting an item shifts the indexes of the items after it.

Fix. Update the constants, or give the items a `Tag` and test that instead, as demo 05 does for the Dark toggle. Tags do not change when items move.

Layout and look

The toolbar ignores the Height I set

What you see. You change `Height` in the Object Inspector and it jumps back.

Why. The height of the toolbar always follows `ButtonSize + 2 * Padding` , scaled to the DPI of the screen.

Fix. Change `ButtonSize` or `Padding` instead.

The standard controls do not change with the theme

What you see. `sgcUISetApplicationTheme(utDark, True)` makes the toolbar and the status bar dark, but the panels, labels, edits and memos of the form stay light.

Why. The VCL side of the `True` parameter lives in the unit `sgcUICore_Theme_VCL`, which registers itself when it is linked into the program. `sgcToolbar_VCL` does not use it, so a program that uses only sgcToolbar does not contain it.

Fix. Add `sgcUICore_Theme_VCL` to the `uses` clause of one unit, as demos 04 and 05 do.

An accent colour shows as transparent or wrong

What you see. After setting `AccentColor` or `BadgeColor`, the colour is missing or not the one you expected.

Why. sgcUI colours are ARGB values, `$AARRGGBB`, not VCL `TColor` values. `$2E7D32` has an alpha of 0, and a VCL constant such as `clRed` has its bytes in a different order.

Fix. Write the colour with an opaque alpha, for example `$FF2E7D32`. The type `TsgcUIColor` is declared in `sgcUICore_Canvas`, add that unit when you declare colour constants, as demo 04 does.

An icon has no tooltip when captions are hidden

What you see. With `ShowCaptions = False`, some icons show a tooltip and others do not.

Why. Tooltips appear only when `ShowHint` is `True`. With captions hidden, an item shows its `Hint`, and its `Caption` only when the `Hint` is empty.

Fix. Set `ShowHint = True` on the toolbar, and give the items a `Hint` or leave it empty to reuse the caption.

The keyboard focus jumps to the toolbar

What you see. Pressing Tab in an editor moves the focus to the toolbar, or the toolbar starts focused.

Why. `TabStop` is `True` by default, so the toolbar is part of the tab order.

Fix. Set `TabStop = False`, as demos 02 to 05 do, and set the `ActiveControl` of the form to the editor.

Editions, threads and compiling

Undeclared identifier Action, or an error reading the form

What you see. With the Community edition, demo 03 or 05 fails to compile with `E2003 Undeclared identifier`, or a form raises an “Error reading” exception when it loads.

Why. The Community edition compiles without `SGC_TOOLBAR_ADVANCED`, so the `Action` property of the items does not exist. A form saved with an item linked to an action cannot be read.

Fix. Use a Registered edition or the Trial. With the Community edition, remove the `Action` lines and run the commands from `OnItemClick` or the item `OnClick` instead.

The last items are cut off at the right edge

What you see. On a narrow window, the last items of the toolbar are clipped.

Why. The Community edition has no overflow menu, so the items that do not fit are clipped.

Fix. Make the window wider, set `ShowCaptions` to `False`, or use a Registered edition, where the hidden items move into the overflow menu.

The status bar misbehaves when updated from a thread

What you see. Random errors or a frozen status bar while a worker thread updates it.

Why. Only `SetPanelText`, `SetProgress`, `SetIndicator` and `SetPanelVisible` are thread safe. Setting `Panels[i].Text` directly, adding panels or touching the toolbar from a worker thread uses the VCL outside the main thread.

Fix. Use the four methods, and stop the worker before the form that owns the status bar is destroyed.

Delphi cannot find sgcToolbar_VCL

What you see. `F2613 Unit 'sgcToolbar_VCL' not found` or a similar message when you compile a demo.

Why. The library path of the active platform does not contain the sgcUI `Source` folder. This happens when sgcToolbar was not installed for that Delphi version, or when you compile for a platform, such as Win64, whose library path was not set.

Fix. Run the installer again for that Delphi version, or add the sgcUI `Source` folder to **Tools** > **Options** > **Language** > **Delphi** > **Library** for the selected platform.

7. Where next

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

The other Delphi demos

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

Demo	What it shows
`EditorShell`	The shell of a small text editor: a toolbar with action links, toggles, a radio group, a dropdown and an overflow menu, and a status bar with the caret position and a saved state indicator. The natural next step after demo 05.
`LiveStatus`	A worker thread simulates a download and updates the progress, the indicator and the text panels of the status bar directly, with the thread safe setters.
`RibbonShell`	A small text editor driven by a ribbon: tabs and groups that adapt to the window width, a split button, a gallery, a contextual tab, KeyTips, a customizable Quick Access Toolbar, a backstage with pages, and a layout that is kept between sessions.
`EditorShellFMX`	The EditorShell demo with a `TsgcFMXToolbar` and a `TsgcFMXStatusBar`: action links, toggles, a radio group, a dropdown, an overflow menu and a status bar.
`RibbonShellFMX`	A FireMonkey editor driven by a `TsgcFMXRibbon`: tabs, adaptive groups, a styles gallery, a dialog launcher, the Quick Access Toolbar, KeyTips, a backstage, themes and layout persistence.

EditorShell and EditorShellFMX use action links and the overflow menu, and the ribbon demos use the ribbon, so these four need a Registered edition or the Trial. LiveStatus builds with every edition.

Help topics

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

Topic	What it covers
Introduction	What sgcToolbar does and where to start.
Architecture, Platforms	The models, the renderer and the controls; the Delphi versions, the platforms and the packages.
Item kinds, Toggles and radio groups, Dropdown menus, Badges	The items in depth.
Action links, Overflow, Images and glyphs, Keyboard	Actions, the overflow menu, icons and keyboard navigation.
Status panels, Progress and indicators, Thread-safe updates	The status bar in depth.
Themes, Headless rendering	Colours, presets and accent colours; images of the bars without a window.
Ribbon overview	The ribbon of the Registered editions.
Troubleshooting	More problems and their causes.
One topic per class	`TsgcVCLToolbar`, `TsgcVCLStatusBar`, `TsgcToolbarItem`, `TsgcStatusBarPanel`, `TsgcToolbarBadgeLink` and the others, with every property, its type and its default.

On the website

- https://www.esegece.com/products/sgctoolbar/: 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 sgcToolbar version and edition, and the demo or a small project that shows the problem.

Enjoy building your next command bar.

---

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