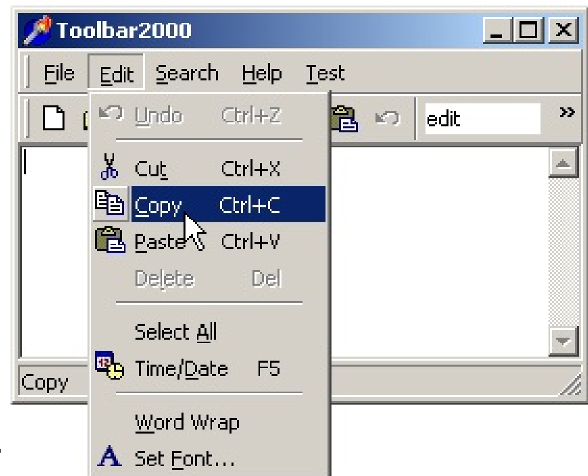
Copyright © 1998-2004 by Jordan Russell. All rights reserved.

*web site:* http://www.jrsoftware.org/
*newsgroups:* http://www.jrsoftware.org/newsgroups.php
*e-mail:* http://www.jrsoftware.org/contact.php

---




Toolbar2000 is a set of components for Borland Delphi and C++Builder (4.0 and later) designed to mimic the Office 2000 look and behavior. It is shareware (register).

Toolbar2000 is nearly a complete rewrite of the classic Toolbar97 component set. As such, I consider it to be a new and separate product, not a mere upgrade. (However, all currently registered users of Toolbar97 are licensed to Toolbar2000 as well, at no extra charge.)

Some of the features of Toolbar2000 include:

- **Office 2000-style draggable, dockable toolbars**
  - Toolbars can smoothly move as they are dragged (no dragging rectangle).
  - Toolbars that go partially off the edge of the form can display a chevron button that brings up a popup window exposing the obscured items.
  - Toolbar items can optionally wrap into multiple rows like Office's menu bar.
  - Unlike Toolbar97, toolbars are not required to be placed on a dock; they can be placed anywhere you need them.
  - Vertical text on vertically-docked toolbars.
- **Office 2000-style menus**
  There is no clear distinction between menus and toolbars, giving you the utmost in flexibility. Toolbars can contain menus, menu bars can contain

buttons, and they can share the same items.

- Full compatibility with Windows 95/98/2000/Me/XP and NT 4.0 *without* requiring a recent version of COMCTL32.DLL, unlike Delphi's TToolBar and TCoolBar components.
- Slide & fade animation (when enabled in Windows).
- On Windows XP, support for shadows on menus and flat menus (when enabled in the OS).
- Microsoft Active Accessibility (MSAA) support.
- Scrolling menu support.
- Multi-monitor support.

---

Toolbar2000 License

**All files included in the Toolbar2000 archive are Copyright © 1998-2003 Jordan Russell. Use and/or distribution of them requires acceptance of the following License Agreement.**

---

Toolbar2000 License Agreement v3.1

"Author" herein refers to Jordan Russell (the creator of the Toolbar2000 components).
"Software" refers to all files bearing this notice, as well as any other files and source code included with Toolbar2000 (typically extracted from a .zip archive), and all content in them, regardless of whether any modifications have been made.

Except where otherwise noted, all of the documentation and Software included in the Toolbar2000 package is copyrighted by Jordan Russell (the Author).

Copyright © 1998-2003 Jordan Russell. All rights reserved.

Use and distribution of the software is permitted provided that all of the following terms are accepted:

1. The Software is provided "as-is," without any express or implied warranty. In no event shall the Author be held liable for any damages arising from the use of the Software.
2. All redistributions of the Software's files must be in their original, unmodified form. Distributions of modified versions of the files is not permitted without express written permission of the Author.
3. All redistributions of the Software's files must retain all copyright notices and web site addresses that are currently in place, and must include this list of conditions without modification.
4. None of the Software's files may be redistributed for profit or as part of another software package without express written permission of the Author.
5. You are permitted to Compile the Software into any kind of applications. ("Compile" here refers to the automatic process of translating the Software's source code into executable machine code by a compiler such as the one included with Borland's Delphi or C++Builder.)
   However, compilation into commercial or shareware applications, or any

applications you are profiting from, requires registration (payment) of the software.
For information on registering, see the Toolbar2000 documentation or this web page:
http://www.jrsoftware.org/tb2kreg.php

6. Redistribution of any of the Software's files in object form (including but not limited to .DCU and .OBJ formats) is strictly prohibited without express written permission of the Author.
7. Full backward compatibility in future versions of the Software is not guaranteed. In no event shall the Author be held liable for any inconvenience or damages arising from lack of backward compatibility.

If you do not agree to all of the above terms, you are not permitted to use the Software in any way, and all copies of it must be deleted from your system(s).

Register

Toolbar2000 is shareware, and is the result of countless hours of hard work. If you use the components, please show your support by registering (paying for it).

**Why register?**

- Registration is required if you use Toolbar2000 in any applications that you or your organization are profiting from (i.e. commercial or shareware).
- Registered users receive higher-priority support.
- Registered users are always guaranteed access to the source code.
- Your support serves as motivation for me to continue improving the components.

**How do I register?**

To register, please visit the following web page:
http://www.jrsoftware.org/tb2kreg.php

The cost of a single user license is currently only US$35. This license permits development using Toolbar2000 by a single person only. A site license costs US$89, which permits development using Toolbar2000 by any number of persons at your place of work. Both licenses allow distribution of Toolbar2000 in compiled form in any type of application - be it commericial, shareware, or freeware.

Registration may done using credit card, check, money order, or cash (many foreign currencies accepted).

Installation

The first step in using Toolbar2000 is to install it in Delphi or C++Builder. However, before using Toolbar2000 be sure to read over the License Agreement.

Toolbar2000 may be used in Delphi 4.0-7.0 or C++Builder 4.0-6.0. Previous versions of Delphi and C++Builder are not supported (see FAQ).

**IMPORTANT:** When unzipping Toolbar2000, make sure your unzip program is configured to recreate the directory structure (in WinZip, check *Use Folder Names*).

**Delphi 4.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* if you have not already done so. The *Library Path* field should then look similar to this:

   $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;c:\tb2k\source

   Click *OK*.
2. Select *File* | *Open...* on the menu bar. Set *Files of type* to *Delphi package source*, locate and select the *tb2k_d4* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.
5. Repeat steps 2 through 4 with the *tb2kdsgn_d4* package.

   **NOTE:** The order in which you install the two packages is important; if you install them in the wrong order, Delphi will probably complain about not being able to find tb2k_d4.bpl the next time it is started. To fix this, select *Component* | *Install Packages*, remove the two packages, and reinstall them by repeating the above steps.

**Delphi 5.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library*

*Path* if you have not already done so. The *Library Path* field should then look similar to this:

$(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;c:\tb2k\source

Click *OK*.
2. Select *File* | *Open...* on the menu bar. Set *Files of type* to *Delphi package source*, locate and select the *tb2kdsgn_d5* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.

**Delphi 6.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* if you have not already done so. The *Library Path* field should then look similar to this:

   $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;c:\tb2k\source

   Click *OK*.
2. Select *File* | *Open...* on the menu bar. Set *Files of type* to *Delphi package source*, locate and select the *tb2kdsgn_d6* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.

**Delphi 7.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* if you have not already done so. The *Library Path* field should then look similar to this:

   $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;c:\tb2k\source

Click *OK*.
2. Select *File* | *Open...* on the menu bar. Set *Files of type* to *Delphi package source*, locate and select the *tb2kdsgn_d7* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.

**C++Builder 4.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* field if you have not already done so. The *Library Path* field should then look similar to this:

   $(BCB)\LIB;$(BCB)\LIB\OBJ;c:\tb2k\source

   Click *OK*.
2. Select *File* | *Open Project...* on the menu bar. Locate and select the *tb2k_cb4* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.
5. Repeat steps 2 through 4 with the *tb2kdsgn_cb4* package.

**C++Builder 5.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* field if you have not already done so. The *Library Path* field should then look similar to this:

   $(BCB)\Lib;$(BCB)\Bin;$(BCB)\Imports;$(BCB)\Projects\Bpl;c:\tb2k\sour

   Click *OK*.
2. Select *File* | *Open Project...* on the menu bar. Locate and select the *tb2k_cb5* package source file in your Toolbar2000 *Packages* directory, and click *Open*.

3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.
5. Repeat steps 2 through 4 with the *tb2kdsgn_cb5* package.

**C++Builder 6.0 installation / upgrade:**

1. Select *Tools* | *Environment Options...* on the menu bar. Go to *Library* tab and add the full path of your Toolbar2000 *Source* directory to the *Library Path* field if you have not already done so. The *Library Path* field should then look similar to this:

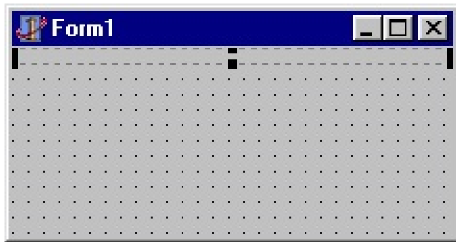
   $(BCB)\Lib;$(BCB)\Bin;$(BCB)\Imports;$(BCB)\Projects\Bpl;c:\tb2k\sour

   Click *OK*.
2. Select *File* | *Open Project...* on the menu bar. Locate and select the *tb2k_cb6* package source file in your Toolbar2000 *Packages* directory, and click *Open*.
3. A package editor window will appear. Click *Compile*, then click *Install*.
4. Close the package editor window. If you are asked if you want to save changes to the package, answer *No*.
5. Repeat steps 2 through 4 with the *tb2kdsgn_cb6* package.

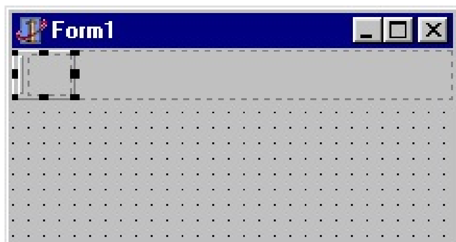
# Getting Started

Now that you have installed Toolbar2000, how do you go about creating dockable toolbars and menu bars? Here is a brief introduction.

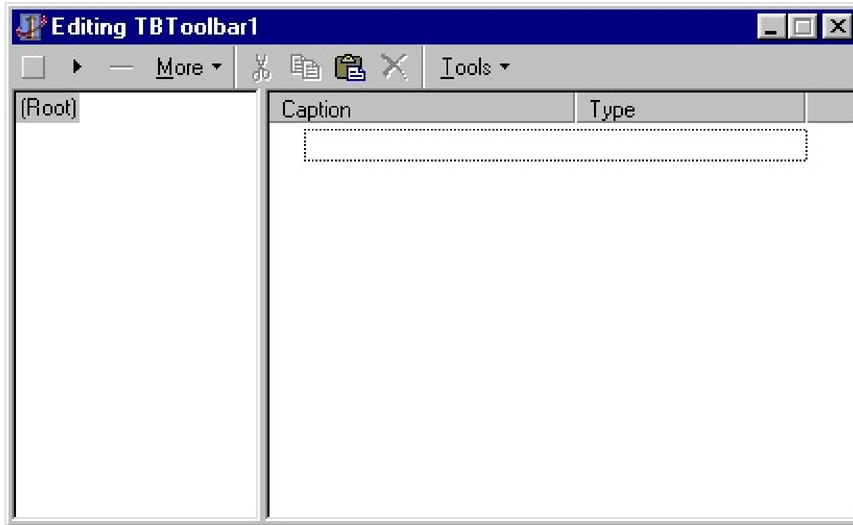
1. The first step is to create one or more TTBDock components. Docks will contain the toolbar(s) you create. To move a dock to a different side of the form, change its `Position` property.



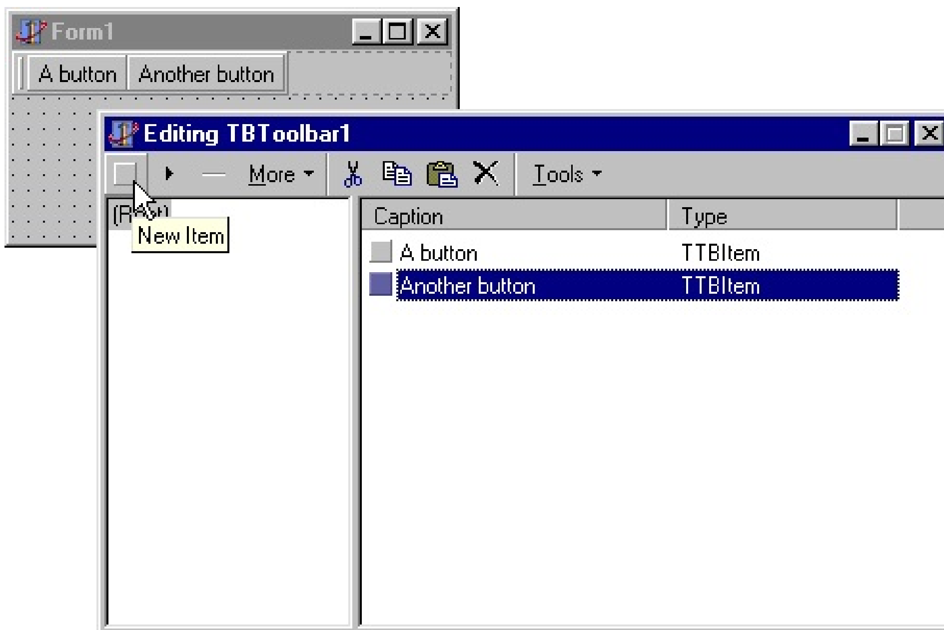

2. Next, select a TTBDock component you have created (by clicking on it), then drop a TTBToolbar component onto the dock (by double-clicking TTBToolbar on the component palette). If the toolbar you created is to be used as a menu bar, set the `MenuBar` property to `True`. If you have an image list you want to use with the toolbar, assign it to the `Images` property.




3. You will probably want to add some items to the toolbar you just created. To do this, invoke the Toolbar Editor by either double-clicking the toolbar or by clicking the "..." button next to the Items property in the Object Inspector.




4. Once inside the Toolbar Editor, use the "New" buttons on the Toolbar Editor's toolbar to create new items. Then modify them to your liking using Object Inspector.




For a ready-to-run example, open the *Demo* project in the *DemoProj* subdirectory. If you're using C++Builder, open the *DemoBCB* project instead, or *DemoBCB6* if you're using C++Builder 6.

## Migrating from TMainMenu/TPopupMenu

Toolbar2000 comes with a TMainMenu/TPopupMenu to TTBToolbar conversion tool, so it is not necessary to re-create all of your menus from scratch when implementing Toolbar2000 in an existing project.

Here is how to use it. It is recommended that you make a backup copy of your form before proceeding.

1. Open the form containing the TMainMenu or TTBPopupMenu component you want to convert.

2. Create a TTBToolbar component on the form (if you haven't already done so), and double-click it to invoke the Toolbar Editor.

3. On the Toolbar Editor's toolbar, click *Tools* then *Convert TMainMenu/TPopupMenu*.

4. Follow the on-screen instructions to proceed with the conversion. If it encounters any properties or events that it is not able to convert, warning messages will be displayed.

5. If you are satisfied with the results, you may delete the TMainMenu or TPopupMenu component.

MSAA Support

Toolbar2000 contains integrated support for Microsoft Active Accessibility (MSAA), a COM-based technology which provides a consistent mechanism for accessibility aids -- notably, screen readers for the visually impaired -- to query applications for information on their user interface elements.

## Enabling MSAA support

When Toolbar2000 is compiled into an EXE or run-time package, MSAA support should work out of the box, provided you have not removed the *Application.Initialize* call from your project's .dpr file.

When Toolbar2000 is compiled into a DLL, the host application must initialize the COM library in order for the MSAA support to function. If the host application is a Delphi application, the simplest way to ensure the COM library is initialized is to add the *ComObj* unit to the project's or main form's "uses" clause. Alternatively, the application can manually call the *CoInitialize* function at startup, and *CoUninitialize* at shutdown.

## Testing MSAA support

Windows 2000 and XP come with a basic MSAA-based screen reader called Narrator which can be used for cursory testing of the MSAA support on Toolbar2000's menus. On English editions of Windows, Narrator may be accessed by going to Start > Programs > Accessories > Accessibility > Narrator. Non-English editions of Windows may not list Narrator on the Start menu, but it should still be possible to start Narrator by running *narrator.exe* manually.

After starting Narrator, try opening and navigating menus in Toolbar2000. You should hear the captions of menu items announced as you move the mouse over them, the same as with standard menus.

For more sophisticated testing, try the Inspect, AccExplorer, and AccEvent tools from the MSAA SDK, available from Microsoft's web site, and also in the Platform SDK.

## Tips for MSAA-friendliness

- Every menu and toolbar item (with the exception of separators) should have a caption assigned. Toolbar2000 will make the caption available to MSAA clients regardless of whether the item has a visible text label. If you do not give your items captions, then visually impaired users may be unable to determine their purpose.

## Compatibility

Toolbar2000's MSAA support has been tested on the following operating system and MSAA version combinations:

- Windows 95 with MSAA 1.3
- Windows 98 with MSAA 1.3 and 2.0
- Windows Me with MSAA 1.3 and 2.0
- Windows NT 4.0 SP6 with MSAA 1.3 and 2.0
- Windows 2000 with MSAA 1.3 and 2.0
- Windows XP with MSAA 2.0

Global Functions and Variables

## TB2Dock unit

### Functions:

- **procedure** `TBIniLoadPositions(`**const** `OwnerComponent: TComponent;` **const** `Filename, SectionNamePrefix:` **string**`);` Loads the positions of all toolbars owned by OwnerComponent from the .INI file specified by Filename. Normally, OwnerComponent will be a form. This function is provided for backwards compatibility; 32-bit applications should use the registry instead. This function should be called when your application starts (usually in the OnCreate handler of your form). If the positions were not previously saved in the .INI file, TBIniLoadPositions has no effect. Each toolbar's data is loaded from a section whose name is the *Name* property of the toolbar prefixed by *SectionNamePrefix*.

  *Example:*

  ```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    TBIniLoadPositions(Self, 'test.ini', '');
  end;
  ```

- **procedure** `TBIniSavePositions(`**const** `OwnerComponent: TComponent;` **const** `Filename, SectionNamePrefix:` **string**`);`
  Saves the positions of all toolbars owned by OwnerComponent to the .INI file specified by Filename. Normally, OwnerComponent will be a form. This function is provided for backwards compatibility; 32-bit applications should use the registry instead. This function should be called when your application exits (usually in the OnDestroy handler of your form). Each toolbar's data is saved to a section whose name is the *Name* property of the toolbar prefixed by *SectionNamePrefix*.

  *Example:*

  ```
  procedure TForm1.FormDestroy(Sender: TObject);
  begin
    TBIniSavePositions(Self, 'test.ini', '');
  end;
  ```

- **procedure** `TBRegLoadPositions(`**const** `OwnerComponent: TComponent;`

**const** RootKey: DWORD; **const** BaseRegistryKey: **string**);
Loads the positions of all toolbars owned by OwnerComponent from the registry. Normally, OwnerComponent will be a form. This function should be called when your application starts (usually in the OnCreate handler of your form). If the positions were not previously saved in the registry, TBRegLoadPositions has no effect.

RootKey and BaseRegistryKey specify the root key and subkey that it loads the data from. Normally, you should use HKEY_CURRENT_USER as the root key. TBRegLoadPositions will append the Name of the toolbars onto this. For example, if BaseRegistryKey is *Software\My Company\My Program\Toolbars* and the Name of a toolbar is *MyToolbar*, it loads the data from the *Software\My Company\My Program\Toolbars\MyToolbar* key.

*Delphi example:*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TBRegLoadPositions(Self, HKEY_CURRENT_USER,
    'Software\My Company\My Program\Toolbars');
end;
```

*C++Builder example:*

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TBRegLoadPositions(this, (DWORD)HKEY_CURRENT_USER,
    "Software\\My Company\\My Program\\Toolbars");
}
```

- **procedure** TBRegSavePositions(**const** OwnerComponent: TComponent; **const** RootKey: DWORD; **const** BaseRegistryKey: **string**);
Saves the positions of all toolbars owned by OwnerComponent to the registry. Normally, OwnerComponent will be a form. This function should be called when your application exits (usually in the OnDestroy handler of your form).

  RootKey and BaseRegistryKey specify the root key and subkey that it saves the data to. Normally, you should use HKEY_CURRENT_USER as the root key. TBRegSavePositions will append the Name of the toolbars onto this. For example, if BaseRegistryKey is *Software\My Company\My Program\Toolbars* and the Name of a toolbar is *MyToolbar*, it saves the data

from the *Software\My Company\My Program\Toolbars\MyToolbar* key.

*Delphi example:*

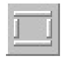
```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  TBRegSavePositions(Self, HKEY_CURRENT_USER, 'Software\My Compa
end;
```

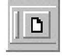
*C++Builder example:*

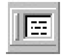
```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  TBRegSavePositions(this, (DWORD)HKEY_CURRENT_USER,
    "Software\\My Company\\My Program\\Toolbars");
}
```

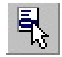
Components

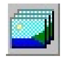
Toolbar2000 provides these components on the component palette:

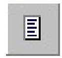
 TTBDock

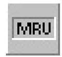
 TTBToolbar

 TTBToolWindow

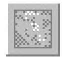
 TTBPopupMenu

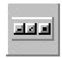
 TTBImageList

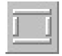
 TTBItemContainer



TTBMRUList

 TTBBackground

 TTBMDIHandler

 TTBDock

Properties | Events | Methods

**Description:**

Create TTBDock controls at locations you want toolbars to be able to dock at. These automatically resize as toolbars are docked onto them. Set the Position property to designate which side of the form the dock is to be located.

**Key Properties:**

- `property AllowDrag: Boolean default True;` When True, toolbars on the dock can be dragged. But when it is False, there are several noteworthy differences: child toolbars are not draggable, the positions of child toolbars are neither loaded nor saved, and toolbars from other docks with AllowDrag set to True cannot be docked to it. Remember you are permitted to create two docks with the same Position, so you could create one dock with AllowDrag set to False and another dock with AllowDrag set to True.
- `property Background: TTBBasicBackground;`
  Specifies an optional background bitmap to be displayed on the dock. TTBBackground components are assigned to this property.
- `property BackgroundOnToolbars: Boolean default True;`
  When True, the Background "shines through" onto docked toolbars.
- `property BoundLines: TTBDockBoundLines;`
  `TTBDockBoundLines = set of (blTop, blBottom, blLeft, blRight);`
  Use this to add extra lines to the sides of the dock. For docks Positioned at the top of the form, it looks best if you set this to [blTop].
- `property FixAlign: Boolean default False;`
  If at run-time you notice a dock not appearing in the location it should, enabling this should correct the problem. This problem only occurs when you have a TTBDock and another control with the same Align setting (i.e. a dock and a list view both set to alLeft). When True, this adds an extra pixel to the width or height so that the VCL is able to align it correctly.
- `property LimitToOneRow: Boolean default False;`
  Set this to True if you want to prevent the user from having more than one row of docked toolbars. I generally don't recommend you enable this (since Office doesn't do this) unless absolutely necessary. If you have fixed-size form that would look wrong with too many rows of toolbars, you should

instead respond to the OnResize event of TTBDock to make your form resize itself.

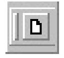
- **property** `Position: TTBDockPosition;`
  `TTBDockPosition = (dpTop, dpBottom, dpLeft, dpRight);`
  Determines where the dock is located on the form.
- **property** `ToolbarCount: Integer;`
  The number of visible toolbars currently docked.
- **property** `Toolbars[Index: Integer]: TTBCustomDockableWindow;`
  Zero-based array of all the visible toolbars that are currently docked.

**Events:**

- **property** `OnInsertRemoveBar: TTBInsertRemoveEvent;`
  `TTBInsertRemoveEvent = procedure(Sender: TObject; Inserting: Boolean; Bar: TTBToolbar) of object;`
  Occurs after a toolbar is docked (Inserting = True) or undocked (Inserting = False).
- **property** `OnRequestDock: TTBRequestDockEvent;`
  `TTBRequestDockEvent = procedure(Sender: TObject; Bar: TTBCustomDockableWindow; var Accept: Boolean) of object;`
  Occurs whenever a toolbar is moved over the dock as it is being dragged. By setting Accept to False you can prevent a particular toolbar from being docked.
- **property** `OnResize: TNotifyEvent;`
  Occurs whenever the dock is resized.

**Key Methods:**

- **procedure** `BeginUpdate;`
  Disables toolbar arrangement. You may call this when moving multiple toolbars on a dock to reduce flicker. Once the changes are complete, call EndUpdate.
- **procedure** `EndUpdate;`
  Re-enables toolbar arrangement after a call to BeginUpdate.

 TTBToolbar

Properties | Events | Methods

**Description:**

This is the toolbar component. To add/edit/delete items on a toolbar, double-click it in the form designer to invoke the Item Editor. (See the Getting Started page for instructions on creating docks and toolbars.)

In addition to Toolbar2000's built-in "items," regular controls may also be placed at the top level of a TTBToolbar. Simply drop them on a toolbar the same way you would any other control. Note, however, that there are several limitations of using controls instead of items on a toolbar. For example, controls will not be displayed in a chevron popup menu.

**Remarks:**

When the *CloseButton* property is True (the default), the toolbar can hide itself if the user clicks the close button on a floating toolbar. Because of this, you should always include an item on a menu that toggles the *Visible* property so the user can get it back if it is closed. See the demo application source code for an example of this.

While toolbars are typically placed on docks, it is not required. When placed outside of a dock, toolbars will not display a drag handle or border. To get an undocked toolbar to wrap or display a chevron, you must do at least one of the following: set AutoResize to False, set Align to one of [alTop, alBottom, alClient], or add [akLeft, akRight] to Anchors.

At run-time, any new items/controls created on a TTBToolbar are initially positioned at the end of the toolbar. To change positions of individual items/controls at run-time, call `toolbar.Items.Move`.

**Key Properties:**

- `property ActivateParent: Boolean default True;` Determines whether the parent form is activated when a floating toolbar is clicked.
- `property AutoResize: Boolean default True;`
  Determines whether the toolbar will automatically adjust its width and

height when items are added/deleted/changed. However, if Anchors includes both *akLeft* and *akRight* then it will not change the toolbar's width, and if Anchors includes both *akTop* and *akBottom* it will not change the height. This property is not applicable to docked or floating toolbars.

- `property BorderStyle: TBorderStyle default bsSingle;`
  When set to bsSingle, the toolbar has a 3-D border.
- `property Caption;`
  What appears in the title bar of a floating toolbar.
- `property ChevronHint: string;`
  The hint displayed when the mouse is moved over the toolbar's chevron (»). This defaults to the value of *STBChevronItemMoreButtonsHint* in TB2Consts.pas.
- `property ChevronMoveItems: Boolean default True;`
  Normally, when an item on the toolbar's chevron popup menu (») is clicked, the clicked item will move into the visible area of the toolbar in place of the least recently clicked item. This behavior is consistent with Office 2000 and XP. Setting this property to False will disable it.
- `property ChevronPriorityForNewItems: TTBChevronPriorityForNewItems default tbcpHighest;`
  `TTBChevronPriorityForNewItems = (tbcpHighest, tbcpLowest);`
  If set to *tbcpHighest*, items created at run-time are given the "highest priority", meaning items that were created previously are hidden first (as they have lower priority). Setting this to *tbcpLowest* gives the opposite behavior: items created at run-time will be hidden first.
- `property CloseButton: Boolean default True;`
  When True, a close button appears in the title bar when the toolbar is floating.
- `property CloseButtonWhenDocked: Boolean default False;`
  When True, a close button is displayed when the toolbar is docked.
- `property CurrentDock: TTBDock;`
  The TTBDock control that the toolbar is currently docked to. To move the toolbar to another dock, you can assign to this property any TTBDock control on the form. To make a toolbar floating at run time, assign to the Floating property.
- `property DefaultDock: TTBDock;`
  (Note: The *LastDock* property is meant to supersede this property.)
  The default dock location. This is used when the user double-clicks a floating toolbar. If neither this property nor *LastDock* are set, nothing will happen.

- **property** DockableTo: TTBDockableTo **default** [dpTop, dpBottom, dpLeft, dpRight];
  TTBDockableTo = set of (dpTop, dpBottom, dpLeft, dpRight);
  Specifies which positions the toolbar may be docked at.
- **property** DockMode: TTBDockMode;
  TTBDockMode = (dmCanFloat, dmCannotFloat, dmCannotFloatOrChangeDocks);
  Determines where the user can drag the toolbar. If this is *dmCanFloat*, the default, the toolbar can float or dock to any dock matching the criteria set by *DockableTo*. If this is *dmCannotFloat*, the toolbar can dock to the same docks but it cannot float. If the user moves the mouse outside a dock, the "Unavailable" mouse cursor is displayed. Also, the toolbar does not respond to double-clicks. If this is *dmCannotFloatOrChangeDocks*, the user cannot drag the toolbar anywhere outside its current dock.
- **property** DockPos: Integer;
  This is only valid if the toolbar is currently docked (CurrentDock <> nil). This is its current horizontal (or vertical, if docked to a left or right dock) position in pixels.
- **property** DockRow: Integer;
  This is only valid if the toolbar is currently docked (CurrentDock <> nil). This is the row the toolbar is currently docked at.
- **property** DragHandleStyle: TTBDragHandleStyle **default** dhDouble;
  TTBDragHandleStyle = (dhDouble, dhNone, dhSingle);
  Determines the type of drag handle displayed on the left (or top, when docked vertically) of the toolbar.
- **property** Floating: Boolean;
  *Run-time only.* This property is True if the toolbar is currently in a floating state. If it is not True, setting it to True will make it floating.
- **property** FloatingMode: TTBFloatingMode;
  TTBFloatingMode = (fmOnTopOfParentForm, fmOnTopOfAllForms);
  By default, this is set to *fmOnTopOfParentForm*, meaning when floating the toolbar only stays on top of its parent form. If this is set to *fmOnTopOfAllForms*, the toolbar will stay above all other forms in the project (except those that are also set to stay on top).
- **property** FloatingPosition: TPoint;
  *Run-time only.* The X,Y coordinates the toolbar appears at when it is in a floating state (Floating = True).
- **property** FloatingWidth: Integer;
  The desired X coordinate at which the toolbar wraps its items when it is in a

floating state (Floating = True).

- **property** FullSize: Boolean **default** False;
  When True, the toolbar always fills the entire width (or height, if vertically docked) of the dock, much like Office's menu bar.
- **property** HideWhenInactive: Boolean **default** True;
  When True, the toolbar is hidden whenever the application is deactivated (a characteristic of Office's toolbars).
- **property** LastDock: TTBDock;
  The TTBDock control that the toolbar was last docked to, or if the toolbar is currently docked and not being dragged, this is equal to the *CurrentDock* property. Double-clicking a floating toolbar will restore it back to the dock specified by this property, and at the same position it previously was docked at. This property overrides and is meant to be a replacement for *DefaultDock*. If you want to disable the use of this property, set *UseLastDock* to False.
- **property** MenuBar: Boolean **default** False;
  When True, the toolbar will act like a menu bar. The MenuBar property itself is used for determining how Alt keypresses are handled, and if accelerator keys may be hidden. Also, for convenience, changing the MenuBar property automatically sets the FullSize and ShrinkMode properties. When MenuBar is set to True, FullSize will be set to True, ShrinkMode will be set to tbsmWrap, CloseButton will be set to False, and ProcessShortCuts will be set to True. This emulates the look and behavior of Office's menu bars.
- **property** ProcessShortCuts: Boolean **default** False;
  When True, shortcut keys specified by the items' ShortCut properties will be processed when the owning form is enabled.
- **property** ShowCaption: Boolean **default** True;
  When True, the toolbar displays a caption bar and a close button (if *CloseButton* is True) when floating.
- **property** ShrinkMode: TTBShrinkMode **default** tbsmChevron;
  TTBShrinkMode = (tbsmNone, tbsmWrap, tbsmChevron);
  Determines how the toolbar will "shrink" when it is too wide to fit on a horizontal dock, or too tall to fit on a vertical dock. *tbsmNone* prevents the toolbar from being shrunk. *tbsmWrap* causes the toolbar's items to wrap into multiple rows when all items cannot fit on a single row. *tbsmChevron* displays a chevron (») at the end of the toolbar when all items cannot fit. Clicking the chevron causes a popup menu to be shown which displays the invisible items.

- **property** SmoothDrag: Boolean **default** True;
  When True, the toolbar's position will be constantly updated while the toolbar is being dragged, like Office 2000. When False, a rectangle will be displayed while the toolbar is being dragged, and the actual toolbar will only move once the mouse button released.
- **property** Stretch: Boolean **default** False;
  When True, the toolbar, when docked, will stretch to fill any unused space on the row. This creates a TCoolBar-like effect.
  *Note:* If you wish to use an aligned (Align <> alNone) or anchored control inside a toolbar, you will need to use TTBToolWindow, since TTBToolbar does not support aligned/anchored controls.
- **property** SystemFont: Boolean **default** True;
  When True, the toolbar will use the menu font set in Windows' Display Properties for its font, instead of the font specified by the *Font* property.
- **property** UpdateActions: Boolean **default** True;
  When True, Actions associated with the toolbar's top-level items will have their Update methods called every time the application becomes idle. This behavior is consistent with other VCL controls, like TToolBar. However, the calls to Update can waste much CPU time if you have a *lot* of items with associated Actions, so it may make sense to change this property to False if that is a concern.
- **property** UseLastDock: Boolean **default** True;
  When True, the toolbar saves the last dock it was docked to in the property *LastDock*, and internally preserves the position it was docked at. See the description of *LastDock* for more information.
- **property** View: TTBView;
  *Run-time only.* The TTBView component that the toolbar uses to render items on the screen.

### Events:

- **property** OnClose: TNotifyEvent;
  Occurs after the toolbar is hidden in response to the user clicking the toolbar's Close button, or the application calling the Close method.
- **property** OnCloseQuery: TCloseQueryEvent;
  TCloseQueryEvent = **procedure**(Sender: TObject; var CanClose: Boolean) **of object**;
  Same in function as a form's OnCloseQuery event. Setting *CanClose* to False will cancel the requested close operation.

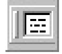
- **property** `OnDockChanged: TNotifyEvent;`
  Occurs after the toolbar has changed between docks, or from a docked to floating state or vice versa.
- **property** `OnDockChanging: TTBDockChangingEvent;`
  `TTBDockChangingEvent =` **procedure**`(Sender: TObject; Floating: Boolean; DockingTo: TTBDock)` **of object**`;`
  Occurs immediately before the toolbar changes between docks, or from a docked to floating state or vice versa. *Floating* specifies whether the toolbar is about to go into a floating state. If Floating is False, *DockingTo* specifies where the toolbar is about to dock to.
- **property** `OnDockChangingHidden: TTBDockChangingEvent;`
  `TTBDockChangingEvent =` **procedure**`(Sender: TObject; Floating: Boolean; DockingTo: TTBDock)` **of object**`;`
  Similar to the OnDockChanging event, but this event is called after the toolbar has already been hidden from the screen in preparation to be moved to another dock. This can be useful if, for example, you want to make changes in the ordering of the toolbar's controls whenever it moves between docks, but don't want any visible flickering during this time.
- **property** `OnMove: TNotifyEvent;`
  Occurs each time the toolbar moves. Note that when *SmoothDrag* is True, the event is fired repeatedly as the toolbar is dragged.
- **property** `OnRecreated: TNotifyEvent;`
  Occurs immediately after the toolbar recreates itself. This usually happens when it changes between a docked and non-docked state.
- **property** `OnRecreating: TNotifyEvent;`
  Occurs immediately before the toolbar recreates itself. This usually happens when it changes between a docked and non-docked state.
- **property** `OnResize: TNotifyEvent;`
  Occurs after the toolbar's size changes. Note that this event is fired after any size change, not only when the user resizes a floating toolbar.
- **property** `OnVisibleChanged: TNotifyEvent;`
  Occurs after the toolbar's visibility changes. This event will occur if the application manually changes the visibility of the toolbar (e.g., by toggling the *Visible* property), or if the user closes the toolbar using its Close button.

**Key Methods:**

- **procedure** `AddDockForm (`**const** `Form: TCustomForm);`
  Adds a form to the list of forms that the toolbar can be docked to besides

the current parent's parent form. Keep in mind that moving a toolbar to another form does not change its Owner property; therefore it will still be destroyed when its owner component is destroyed. To change a toolbar's owner, use TComponent's RemoveComponent and InsertComponent methods.

- **procedure** `BeginMoving (`**const** `InitX, InitY: Integer);`
  `TTBSizeHandle = (twshLeft, twshRight, twshTop, twshTopLeft, twshTopRight, twshBottom, twshBottomLeft, twshBottomRight);`
  Forces the toolbar to enter "move" mode. This is called internally whenever the user clicks the drag handle of a docked toolbar or the caption bar of a floating toolbar, and will only work properly if it is called while the left mouse button is still down. The *InitX* and *InitY* parameters specify the client coordinates where the mouse button went down at, which determine where the dragging rectangle appears initially. In most cases (0, 0) should be fine.
- **procedure** `BeginSizing (`**const** `ASizeHandle: TToolWindowSizeHandle);`
  `TTBSizeHandle = (twshLeft, twshRight, twshTop, twshTopLeft, twshTopRight, twshBottom, twshBottomLeft, twshBottomRight);`
  Forces the toolbar to enter "size" mode. This is called internally whenever the user clicks one of the resizing handles on the border of a floating toolbar, and will only work properly if it is called while the left mouse button is still down. The *ASizeHandle* parameter specifies which resizing handle was clicked.
- **procedure** `BeginUpdate;`
  Disables arrangement of items/controls on the toolbar. You may call this when moving multiple items/controls on the toolbar to reduce flicker. Once the changes are complete, call EndUpdate.
- **procedure** `EndUpdate;`
  Re-enables item/control arrangement after a call to BeginUpdate.
- **procedure** `RemoveDockForm (const Form: TCustomForm);`
  Removes a form from the list of forms AddDockForm adds to.

 TTBToolWindow

Properties

**Description:**

This component is very similar to TTBToolbar, but has several key differences:

- Only controls are accepted; there is no possibility to drop "items" (unless you put a TTBToolbar inside the TTBToolWindow).
- Contained controls are not arranged automatically.
- Contained controls may be aligned (Align <> alNone).
- Floating tool windows may be resized freely.

See the help for the TTBToolbar component for explanations of the properties and events not listed here.

**Key Properties:**

- `property MaxClientHeight: Integer default 32;` The maximum height, in client pixels, that the user can resize the tool window to when floating.
- `property MaxClientWidth: Integer default 32;`
  The maximum width, in client pixels, that the user can resize the tool window to when floating.
- `property MinClientHeight: Integer default 32;`
  The minimum height, in client pixels, that the user can resize the tool window to when floating.
- `property MinClientWidth: Integer default 32;`
  The minimum width, in client pixels, that the user can resize the tool window to when floating.
- `property Resizable: Boolean default True;`
  When True, the user may resize the tool window when floating.

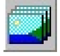
TTBPopupMenu

**Description:**

This component is the Toolbar2000 version of TPopupMenu. It may be used whereever a TPopupMenu would be used. Double-click it in the form designer to invoke the Item Editor.

**Remarks:**

Note that since TTBPopupMenu is a descendant of TPopupMenu, it inherits all of TPopupMenu's properties. However, not all of these properties are applicable to Toolbar2000, and thus are ignored. Examples of ignored properties include AutoHotkeys, AutoLineReduction, and MenuAnimatation.

 TTBImageList

[Properties](#)

**Description:**

This component is an enhanced version of the standard *TImageList* component, designed for use with the Toolbar2000 components.

**Key Properties:**

- `property CheckedImages: TCustomImageList;` Pointer to another image list which contains images that are to be shown on checked (Checked = True) items. If this property is not set, it will not use different images on checked items.
- `property DisabledImages: TCustomImageList;`
  Pointer to another image list which contains images that are to be shown on disabled (Enabled = False) items. If this property is not set, it will generate its own "disabled" images.
- `property HotImages: TCustomImageList;`
  Pointer to another image list which contains images that are to be shown on items when they are selected. If this property is not set, it will not use different images on selected items.
- `property ImagesBitmap: TBitmap;`
  *ImagesBitmap* lets you use bypass the standard TImageList streaming mechanism which suffers from these problems:
  - On systems with older COMCTL32.DLL versions (e.g. Windows 95) the images don't show up.
  - It saves images in your desktop's color depth. Thus, even if your images use only 16 colors, if you're running in 32-bit color mode the images will be saved in 32-bit color -- a big waste of space.

  How does one create a bitmap suitable for assigning to *ImagesBitmap*?
  1. The simplest way to get started is to double-click your existing image list in the form designer and click *Export*.
  2. *(optional)* Load the exported bitmap into an image editing program (Paint will do) and save it in the smallest color depth necessary. If your image list uses Office-style images, then 4-bit color (i.e. 16 colors) is all you need.
  3. Go to the *ImagesBitmap* property, click the "..." button and load in the

bitmap file. Then save your form. (You may want to make a backup of your form files first.)

From then on, the image list will get its images from the *ImagesBitmap* property. Don't modify the image list using Delphi's Image List Editor; as long as a bitmap is assigned to the *ImagesBitmap* property, the changes won't be preserved when the form is saved.

- `property ImagesBitmapMaskColor: TColor default clFuchsia;`
  The color in *ImagesBitmap* which is considered transparent.

TTBItemContainer

**Description:**

This non-visual component is a container for items. Like TTBToolbar, double-click it in the form designer to invoke the Item Editor.

By itself, TTBItemContainer is not useful. It is intended to be "linked" to another toolbar item via the *LinkSubitems* property. You cannot directly assign a TTBItemContainer to a LinkSubitems property, but you can create a TTBSubmenuItem-type item inside the TTBItemContainer, and assign that to a LinkSubitems property.

TTBMRUList

Properties | Methods

**Description:**

This component holds items for a Most Recently Used list. It is intended to be linked to a TTBMRUListItem item on a menu via its `MRUList` property.

**Key Properties:**

- **`property`** `AddFullPath: Boolean` **`default`** `True;` When True, items added to the list via the Add method will be expanded into fully qualified pathnames. If your MRU list doesn't contain filenames, this property should be changed to False.
- **`property`** `HidePathExtension: Boolean` **`default`** `True;`
  When True, pathnames will be hidden when the items are displayed, as well as file extensions if Explorer is configured to "hide file extensions for known file types." If your MRU list doesn't contain filenames, this property should be changed to False.
- **`property`** `Items: TStrings;`
  The items in the MRU list. You can read and directly manipulate the MRU list items via this property.
- **`property`** `MaxItems: Integer` **`default`** `4;`
  The maximum number of items that the list may contain. When `MaxItems` is exceeded, one or more items are deleted from the end of the list.
- **`property`** `Prefix:` **`string`**`;`
  The string prefixed to the names of values read and written by the LoadFrom* and SaveTo* methods. The default is "MRU", making the name of the first value "MRU0".

**Key Methods:**

- **`procedure`** `Add(`**`const`** `Filename:` **`string`**`);`
  Adds a new filename to the top of the MRU list. If the specified filename already exists in the MRU list, it will be moved to the top.
  The `Add` method is the preferred way of adding new items to the MRU list.
- **`procedure`** `LoadFromIni(Ini: TCustomIniFile;` **`const`** `Section:` **`string`**`);`

Loads the MRU items from a TIniFile component or other TCustomIniFile descendant. *Section* specifies the section name to read from.
Usage example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create('MyProgram.ini');
  try
    MRUList.LoadFromIni(Ini, 'MRUList');
  finally
    Ini.Free;
  end;
end;
```

- **procedure** LoadFromRegIni(Ini: TRegIniFile; **const** Section: **string**);
  Loads the MRU items from a TRegIniFile component. *Section* specifies the section name to read from.
  Usage example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Ini: TRegIniFile;
begin
  Ini := TRegIniFile.Create('Software\My Company\My Program');
  try
    MRUList.LoadFromRegIni(Ini, 'MRUList');
  finally
    Ini.Free;
  end;
end;
```

- **procedure** Remove(**const** Filename: **string**);
  Removes the specified filename from the MRU list if it exists.
- **procedure** SaveToIni(Ini: TCustomIniFile; **const** Section: **string**);
  Saves the MRU items to a TIniFile component or other TCustomIniFile descendant. *Section* specifies the section name to save to.
  Usage example:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  Ini: TIniFile;
begin
```

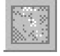
```
  Ini := TIniFile.Create('MyProgram.ini');
  try
    MRUList.SaveToIni(Ini, 'MRUList');
  finally
    Ini.Free;
  end;
end;
```

- **procedure** `SaveToRegIni(Ini: TRegIniFile;` **const** `Section:` **string**`);`

  Saves the MRU items to a TRegIniFile component. *Section* specifies the section name to save to.

  Usage example:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  Ini: TRegIniFile;
begin
  Ini := TRegIniFile.Create('Software\My Company\My Program');
  try
    MRUList.SaveToRegIni(Ini, 'MRUList');
  finally
    Ini.Free;
  end;
end;
```

 TTBBackground

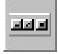
Properties

**Description:**

This component holds a background bitmap which may be associated with TTBDock components via their *Background* properties.

**Key Properties:**

- **`property`** `Bitmap: TBitmap;` The bitmap to use. It is tiled across the length of the dock.
- **`property`** `BkColor: TColor` **`default`** `clBtnFace;`
  See description of *Transparent*.
- **`property`** `Transparent: Boolean` **`default`** `False;`
  When True, the color of the bottom-left pixel of the bitmap is considered transparent, and is replaced with the color specified by the *BkColor* property.

 TTBMDIHandler

Properties

**Description:**

This component adds the MDI system menu and minimize/restore/close buttons to a menu bar when an MDI child window in the application is maximized. The *Toolbar* property specifies the menu bar.

Any MDI application which uses Toolbar2000 for its menus should use this component.

**Remarks:**

Currently TTBMDIHandler is only designed to work with wrapped (*ShrinkMode = tbsmWrap*) menu bars.

The items on the system menu are generated from the actual MDI child form's system menu. Therefore, the captions of the items are properly localized.

**Key Properties:**

- **`property`** `Toolbar: TTBCustomToolbar;` The menu bar that the MDI system menu and buttons should be added to. This menu bar's *ShrinkMode* property should be set to *tbsmWrap*.

## Item Components

This section documents the various "item" components that may be placed on toolbars.

TTBCustomItem

Properties | Methods | Events

**Description:**

This is the base component which all other Toolbar2000 items inherit from.

**Key Properties:**

- `property AutoCheck: Boolean default False;` When True, the item's Checked state toggles automatically when it is clicked, before any OnClick event is fired.
- `property Caption: string;`
  The item's caption. Whether it is displayed or not depends on the setting of the *DisplayMode* and *Options* properties, and whether the item is on a menu or a toolbar.
  Regardless of whether the caption is displayed it is always made available to MSAA clients, so it is a good idea to set *Caption* on every item you create (with the exception of separators).
- `property Checked: Boolean;`
  When True, the button or menu item is drawn has a "down" or "checked" appearance. This property is similar to TMenuItem.Checked and TSpeedButton.Down.
- `property Count: Integer;`
  *(Run-time only)* The number of subitems the item has.
- `property DisplayMode: TTBItemDisplayMode default nbdmDefault;`
  `TTBItemDisplayMode = (nbdmDefault, nbdmTextOnly, nbdmTextOnlyInMenus, nbdmImageAndText);`
  Determines whether the item's image is to be displayed. *nbdmDefault* means show both the image and text in menus, but only an image on toolbars. *nbdmTextOnly* means never show an image. *nbdmTextOnlyInMenus* means never show an image if the item is on a menu. *nbdmImageAndText* means always show the image.
- `property EffectiveOptions: TTBItemOptions;`
  *(Run-time only)* The effective Options for the item; that is, the Options after having been combined with inherited options and masked.
- `property GroupIndex: Integer default 0;`
  When GroupIndex is non-zero and True is assigned to the Checked

property, all other items with the same parent and GroupIndex setting will have their Checked properties set to False automatically.
This is similar to TSpeedButton's GroupIndex property.

- `property Hint: string;`
  The popup text that is to be displayed when the user rests the mouse cursor over the item. On toolbars, you must set the *ShowHint* property to *True* in order to see popup hints on top-level items. To see popup hints on popup menus, you must add *tboShowHint* to the *Options* property of the items or one of their ancestors.
  This property works essentially the same as the *Hint* property used throughout the VCL, but with one Toolbar2000-specific enhancement: if you don't assign a short hint to the *Hint* property, Toolbar2000 will generate one itself by taking the value of the *Caption* property and stripping any accelerator keys and any trailing colon or ellipsis from it. However, in the case of TTBSubmenuItem, Toolbar2000 will not generate a hint unless the item has the *DropdownCombo* property set to *True* or the item has no visible caption.
  The automatic hint generation may be disabled by setting *tboNoAutoHint* on the *Options* property.
- `property InheritOptions: Boolean default True;`
  When True, the item will automatically combine its parent's Options with its own Options. Use the *MaskOptions* property to prevent certain Options from being inherited.
- `property ImageIndex: Integer;`
  The image index to use, or -1 if the item has no image.
  By default, when a toolbar item has an image, its Caption is hidden. Set the DisplayMode property to *nbdmImageAndText* to have it display both the image and caption.
- `property Images: TCustomImageList;`
  The image list that holds the image for the item. If *Images* is nil (i.e. left blank at design time), it will inherit the images from the parent toolbar or menu.
- `property Items[Index: Integer]: TTBCustomItem; default;`
  *(Run-time only)* Use Items to access to a subitem by its position in the list of subitems.
- `property LinkSubitems: TTBCustomItem;`
  This property allows you to have the item take its list of subitems from a different item instead of from itself. By using this property, you could have several TTBSubmenuItems sharing the same set of subitems.

- **property** `MaskOptions: TTBItemOptions` **default** `[];`
  Determines which Options will *not* be inherited. Has no effect if *InheritOptions* is False.
- **property** `Options: TTBItemOptions` **default** `[];`
  Miscellaneous options:
  - tboDefault - When set, the item's Caption will be displayed in a bold font, and the item will be automatically executed when the parent item is double-clicked. Like TMenuItem's Default property.
  - tboDropdownArrow - When set, TTBSubmenuItems on toolbars will display an arrow on their right side. This option does not change the way the item functions.
  - tboImageAboveCaption - When set, the item's image will be displayed above its caption, similar to default look in older versions of Internet Explorer. This option has no effect if the item is on a popup menu.
  - tboLongHintInMenuOnly - When set, the item's long hint will only be used when in a modal menu loop.
  - tboNoAutoHint - When set, automatic hint generation will be disabled. See the description of the *Hint* property for more information.
  - tboNoRotation - When set, the item's caption will not be rotated 90 degrees when its parent toolbar is docked vertically.
  - tboSameWidth - When set along with *tboImageAboveCaption* on two or more items on the same view, the items will be stretched as necessary so that they all have the width in pixels.
  - tboToolbarStyle - When set, the item will be displayed in the style of a toolbar button when on a popup menu: it will have a raised border, the text will be centered, and when Checked the filled pattern will extend across the entire item.
  - tboToolbarSize - When set, the item will be displayed on popup menus with the same size as a toolbar button, and its text will be hidden by default like a toolbar button. Also, consecutive items with tboToolbarSize set will be arranged horizontally like a toolbar.
    In order to set tboToolbarSize, you must first set tboToolbarStyle.
  - tboShowHint - When set, the item can display popup hints when on a popup menu (similar to IE5's Favorites menu). (To enable popup hints on a toolbar, set the toolbar's ShowHint property to True.)

  By default, options set here are inherited by child items. See also the *InheritOptions* and *MaskOptions* properties.
- **property** `ShortCut: TShortCut` **default** `0;`
  The keyboard shortcut for the item. Note that the ProcessShortCuts

property of the parent toolbar must be set to True for the shortcut to be functional.

- **property** `SubMenuImages: TCustomImageList;`
  The image list that holds the images for the item's subitems. If *SubMenuImages* is nil (i.e. left blank at design time), it will inherit the images from the parent toolbar or menu.

**Key Methods:**

- **procedure** `Add(AItem: TTBCustomItem);`
  Adds a new subitem to the end.
- **procedure** `Clear;`
  Removes and frees all of the item's subitems.
- **function** `ContainsItem(AItem: TTBCustomItem): Boolean;`
  Returns True if *AItem* is one of the item's subitems. AItem need not be an immediate child of the item.
- **procedure** `Delete(Index: Integer);`
  Deletes the subitem at position *Index*. See also Remove.
- **functions** `IndexOf(AItem: TTBCustomItem): Integer;`
  Returns the position of the subitem *AItem*, or -1 if it isn't one of the item's subitems.
- **procedure** `Insert(NewIndex: Integer; AItem: TTBCustomItem);`
  Inserts a new subitem *AItem* at position *NewIndex*.
- **procedure** `Move(CurIndex, NewIndex: Integer);`
  Moves the item at position *CurIndex* to position *NewIndex*.
- **procedure** `Popup(X, Y: Integer; TrackRightButton: Boolean;`
  `Alignment: TTBPopupAlignment = tbpaLeft);`
  `TTBPopupAlignment = (tbpaLeft, tbpaRight, tbpaCenter);`
  Creates and displays a popup menu containing the item's subitems. If *TrackRightButton* is True, items may be selected with the right mouse button. *Alignment* determines how the popup menu is aligned relative to the X,Y coordinate.
- **procedure** `Remove(Item: TTBCustomItem);`
  Removes the subitem *Item*. Nothing happens if *Item* isn't one of the item's subitems.

**Events:**

- **property** `OnClick: TNotifyEvent;`

Occurs after a button or menu item is clicked.

If the item is a

TTBSubmenuItem with the *DropdownCombo* property set to False (the default), this event will also be fired when the submenu is opened, after the OnPopup event.

- **property** OnPopup: TTBPopupEvent;

```
TTBPopupEvent = procedure(Sender: TTBCustomItem; FromLink: Boolean)
of object;
```

Occurs before a submenu is opened. You could use this event to initialize the appearance of the submenu by writing code to add, delete, or modify items.

If the LinkSubitems property is set, an OnPopup event will be sent first to the item referenced by LinkSubitems. The FromLink parameter will be True in such events.

- **property** OnSelect: TTBSelectEvent;

```
TTBSelectEvent = procedure(Sender: TTBCustomItem; Viewer:
TTBItemViewer; Selecting: Boolean) of object;
```

Occurs when the item becomes, or ceases to be, the selected item. *Selecting* will be True if the item is the selected item, or False if the item is no longer the selected item.

TTBItem

**Description:**

This is the primary item component.

It is essentially identical to TTBCustomItem; see its help topic for details on the properties and methods.

TTBSubmenuItem

Properties

**Description:**

Use TTBSubmenuItem to create top-level menus or submenus.

TTBSubmenuItem inherits many properties, methods, and events from TTBCustomItem; see its help topic for details on the properties, methods, and events not listed here.

**Key Properties:**

- `property DropdownCombo: Boolean default False;` When True, the item will behave as a both clickable button and a dropdown menu, similar to MS Office's Font Color button. Clicking the left side of the button generates an OnClick event; clicking the right side of the button (the part with the arrow) generates an OnPopup event, and displays a dropdown menu.

TTBSeparatorItem

Properties

**Description:**

Use TTBSeparatorItem to create a separator.

**Remarks:**

Separators that are at the front or end of a toolbar/menu will be hidden automatically, as will consecutive separators.

**Key Properties:**

- `property Blank: Boolean default False;` When True, the item won't draw a line; it will just be an empty space.

TTBEditItem

Properties | Events

**Description:**

Use TTBEditItem to create top-level menus or submenus.

TTBEditItem emulates the appearance and behavior of the edits on MS Office's toolbars.

A key functional difference between a TEdit and a TTBEditItem is the Text property doesn't change in real time as the user modifies the text. Instead, the text is only saved in the Text property once the user presses Enter. (Additionally, an OnAcceptText event is fired when Enter is pressed.) If the user presses Escape, it discards the changes.

Also, by default, TTBEditItem will change into a button when its parent toolbar is docked vertically. This behavior can be disabled by adjusting the EditOptions property (see below).

TTBEditItem inherits many properties, methods, and events from TTBCustomItem; see its help topic for details on the properties, methods, and events not listed here.

**Key Properties:**

- `property CharCase: TEditCharCase;`
  `TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);`
  Use CharCase to force the item's text to assume a particular case.
- `property EditCaption: string;`
  The caption displayed to the left of the Edit when the item is on a menu.
- `property EditOptions: TTBEditItemOptions default [];`
  Miscellaneous options:
  - tboUseEditWhenVertical - When set, the item will remain displayed as an edit instead of a button when the parent toolbar is docked vertically.
- `property EditWidth: Integer default 64;`
  The width in pixels of the Edit.
- `property MaxLength: Integer default 0;`
  The maximum number of characters the user may enter. If this is 0, no limit

is imposed.

- **property** `Text`: **string**;
  The text.

**Key Events:**

- **property** `OnAcceptText: TTBAcceptTextEvent;`
  `TTBAcceptTextEvent =` **procedure**`(Sender: TObject;` **var** `NewText: String;` **var** `Accept: Boolean)` **of object**;
  Occurs when the user presses Enter after editing the text. *NewText* is the new text that will be assigned to the Text property, which the event handler may optionally modify. Set *Accept* to False to prevent the new text from being assigned to the Text property.

- **property** `OnBeginEdit: TTBBeginEditEvent;`
  `TTBBeginEditEvent =` **procedure**`(Sender: TTBEditItem; Viewer: TTBEditItemViewer; EditControl: TEdit)` **of object**;
  When you click a TTBEditItem or press Enter on it, a TEdit control is created on the fly to handle the input. This event occurs after the TEdit control is created. The TEdit control is specified in the *EditControl* parameter. The code in the event handler may modify the properties of *EditControl* to customize its appearance, among other things.

TTBGroupItem

**Description:**

TTBGroupItem allows you to create your own group item.

TTBGroupItem inherits many properties, methods, and events from TTBCustomItem; see its help topic for details on the properties, methods, and events not listed here.

TTBMRUListItem

Properties

**Description:**

TTBMRUListItem is a group item which displays a Most Recently Used list.

To use this item, you first need to create a TTBMRUList component.

**Key Properties:**

- `property MRUList: TTBMRUList;` The TTBMRUList component to get the MRU items from.

TTBMDIWindowItem

Events

**Description:**

TTBMDIWindowItem is a group item which expands to a list of available MDI child windows at run-time. This is intended to be placed at the end of an MDI application's Window menu, following a separator.

**Remarks:**

This component modifies the main form's *WindowMenu* property at run-time.

The list of windows comes from the system. Thus, the caption of the "More Windows" item is properly localized.

**Events:**

- `property OnUpdate: TNotifyEvent;` Occurs after the subitems are created and initialized.

TTBVisibilityToggleItem

Properties

**Description:**

TTBVisibilityToggleItem is a special type of item which toggles the *Visible* property of a control (specified by the *Control* property) when clicked. It also at run-time automatically sets its *Checked* property equal to control's *Visible* property.

This item class is primarily useful for creating menu items which toggle the visibility of toolbars.

TTBVisibilityToggleItem inherits many properties, methods, and events from TTBCustomItem; see its help topic for details on the properties, methods, and events not listed here.

**Key Properties:**

- **`property`** `Control: TControl;` The associated control.

TTBControlItem

Properties

**Description:**

A TTBControlItem component is automatically created when you drop a control on a TTBToolbar. It is essentially a "wrapper" that allows the control to be managed by the toolbar in the same way as Toolbar2000's native items.

**Key Properties:**

- `property Control: TControl;` The control it is managing. You should not modify this property; it is set automatically.

TTBView

Properties | Methods

**Description:**

Views arrange items for display and create item viewers (TTBItemViewer) to render individual items.

Each TTBToolbar component has its own TTBView instance, accessible via the View property. Popup menus also have views when they are displayed.

**Usage example:**

If you have a toolbar named TBToolbar1 and an item on the toolbar named TBItem1, here is how to get the bounding rectangle of TBItem1:

```
var
  R: TRect;
begin
  R := TBToolbar1.View.Find(TBItem1).BoundsRect;
end;
```

**Key Properties:**

- **`property`** `ParentItem: TTBCustomItem;` *(Read-only)* The item which the view looks in to find items to display. The view then creates item viewers for each item it is going to display.
- **`property`** `ParentView: TTBView;`
  *(Read-only)* The parent view, or *nil* if the view is a top-level view.
- **`property`** `Selected: TTBItemViewer;`
  The currently selected item viewer in the view, or *nil* if there is no selected item viewer. An item viewer becomes "selected" when the mouse is moved over it, or when it is highlighted with the keyboard.
- **`property`** `Viewers: PTBItemViewerArray;`
  *(Read-only)* The zero-based of array of item viewers (TTBItemViewer) that the view currently owns.
  Note: Access to this property is not range checked.
- **`property`** `ViewerCount: Integer;`
  *(Read-only)* The number of item viewers available in the Viewers array.
- **`property`** `Window: TWinControl;`

*(Read-only)* Specifies the control associated with the view. TTBView will invalidate this control when necessary.

**Key Methods:**

- **`procedure`** `BeginUpdate;`
  Disables arrangement of item viewers on the view. (TTBToolbar.BeginUpdate calls this.)
- **`function`** `ContainsView(AView: TTBView): Boolean;`
  Returns True if *AView* is a child view of the view. *AView* doesn't have to be an immediate child for the function to return True.
- **`procedure`** `CloseChildPopups;`
  Destroys any child views (typically popup menus) that the view is currently displaying.
- **`procedure`** `DrawSubitems(ACanvas: TCanvas);`
  Draws the entire contents of the view on a canvas. TTBToolbar calls this in its Paint method.
- **`procedure`** `EndUpdate;`
  Re-enables item viewer arrangement after a call to BeginUpdate. (TTBToolbar.EndUpdate calls this.)
- **`function`** `Find(Item: TTBCustomItem): TTBItemViewer;`
  Returns the item viewer associated with *Item*. If the item does not exist on the view, an exception is raised. Therefore it will not return *nil*.
- **`function`** `IndexOf(AViewer: TTBItemViewer): Integer;`
  Returns the index of the specified item viewer. If *AViewer* is *nil* or does not exist, the function returns -1.
- **`function`** `Invalidate(AViewer: TTBItemViewer);`
  Invalidates the area on Window occupied by the item viewer. There is normally no need to call this function directly.
- **`procedure`** `InvalidatePositions;`
  Marks the current item viewer positions as "invalid," causing them to be recalculated and redrawn later. There is normally no need to call this function directly.
- **`function`** `NextSelectable(CurViewer: TTBItemViewer; GoForward: Boolean): TTBItemViewer;`
  Returns the next selectable item viewer following CurViewer (if GoForward is True) or preceding CurViewer (if GoForward is False). CurViewer can be *nil*, in which case the function will return the first or last selectable item viewer. If no selectable item viewer is found, the function

returns *nil*.

- **procedure** `ScrollSelectedIntoView;`
  If the view is a scrolling menu, this procedure adjusts the scroll position if necessary so that the currently selected item viewer is visible.
- **procedure** `TryValidatePositions;`
  Same as *ValidatePositions*, but does not validate if *BeginUpdate* was called or if *ParentItem* is currently being loaded.
- **procedure** `ValidatePositions;`
  Ensures that all item viewer positions are valid, recalculating them if necessary. There is normally no need to call this function directly.
- **function** `ViewerFromPoint(`**const** `P: TPoint): TTBItemViewer;`
  Returns the item viewer at the specified point, or *nil* if no item viewer exists at the specified point. The point is in client coordinates.

TTBItemViewer

Properties | Methods

**Description:**

An item viewer is responsible for drawing a single item on the screen, and handling mouse and keyboard input directed to the item.

TTBView creates and manages item viewers.

**Key Properties:**

- **`property`** `BoundsRect: TRect;` *(Read-only)* The bounding rectangle of the item. This is not valid when Show is False.
- **`property`** `Clipped: Boolean;`
  *(Read-only)* This is True if the item would normally have been shown but the parent menu has scrolling enabled and the item could not fit in the available space. *Show* will always be False when *Clipped* is True.
- **`property`** `Item: TTBCustomItem;`
  *(Read-only)* The item being rendered.
- **`property`** `OffEdge: Boolean;`
  *(Read-only)* This is True if the item could not fit in the available space and is to be displayed on the parent toolbar's chevron popup menu. *Show* will always be False when *OffEdge* is True.
- **`property`** `Show: Boolean;`
  *(Read-only)* This is True if the item will be shown, or False if it is hidden.
- **`property`** `View: TTBView;`
  *(Read-only)* The owning TTBView component.

**Key Methods:**

*Note:* There are a lot of protected methods in TTBItemViewer which can be overridden in descendant classes to change appearance or behavior of the item viewer. Documentation for them has not been completed yet.

- **`function`** `ScreenToClient(`**`const`** `P: TPoint): TPoint;`
  Converts a screen coordinate to a coordinate relative to the top-left corner of the item viewer.

# Glossary

The following are definitions of some of the special terms used in Toolbar2000.

**group item**
: A special type of item. The child items of a *group item* are rendered on the group item's parent item, as if they were actually children of the parent item.
: Group items may be nested up to approximately 9 levels deep.

**item**
: *Item* components are used for toolbar and menu items. Items descend from the TTBCustomItem class.

**item viewer**
: *Item viewer* refers to a TTBItemViewer component or a descendant. Instances of TTBItemViewer draw individual items on the screen and process mouse and keyboard input.

**view**
: A *view* refers to a TTBView component. The TTBView class arranges items for display and manages item viewers.

Frequently Asked Questions

- Features
- Problems

---

## Features

### Will it have _____ in the future?

Please check the online Feature Request Tracker for a list of some features I intend to add, and features that users have requested be added.

### Will Delphi 2/3 and C++Builder 1/3 be supported?

No; sorry. There are several reasons why supporting them would be difficult because of key features their VCLs lack. If you need a good dockable toolbar system now which supports these older Delphi/C++Builder versions, you can always use Toolbar97.

### Where is the Down property on toolbar buttons?

It's there but it's called Checked.

### Can I get the positions of my toolbars to be preserved when my application exits, and restored the next time it is started?

Yes, Toolbar2000 provides functions to do this. See Global Functions and Variables.

### How can I get the bounding rectangle of an item on a toolbar?

If you have a toolbar named TBToolbar1 and an item on the toolbar named TBItem1, here is how to get the bounding rectangle of TBItem1:

```
var
  R: TRect;
begin
```

```
  R := TBToolbar1.View.Find(TBItem1).BoundsRect;
end;
```

---

## Problems

### Why don't controls on toolbars appear in the chevron popup menu?

Toolbar2000's proprietary "items" have been designed from the ground up to have the ability to be displayed in multiple places simultaneously. Controls do not have this ability. It would have to change the Parent property of the controls to move them to the popup menu. However, it's not quite that simple. Because Toolbar2000 takes the mouse capture while a popup menu is up, it would need some mechanism for delivering mouse and keyboard events to controls on a popup menu. I think this would be rather complicated to do, and it's not something high on the list of my priorities at the moment.

### Why don't I see Minimize, Restore, and Close buttons on my menu bar when an MDI child form is maximized?

Use a TTBMDIHandler component to make a menu bar MDI-aware.