

The **mitchenall.com**
Toolbar Component

*Draft Developer Documentation
for Version 1.0*

Table of Contents

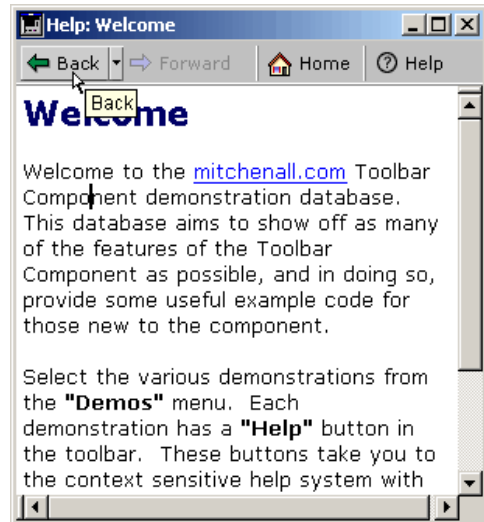
Table of Contents	2
Introduction	4
Toolbar Components.....	5
Installing the Components	6
On Startup	7
Process Stack Requirements	7
Adding a Toolbar to a 4D Form.....	8
Minimum Requirements for a Form	9
Pre-Defined Standard Toolbars.....	10
Defining your own toolbars	12
One Toolbar Per Form	12
Adding Buttons and Button Names	12
Button Types	14
Button and Popup Actions	14
toolbar_Redraw.....	16
Basic Command Reference	17
Initialisation Commands	18
toolbar_Startup.....	18
toolbar_Clear.....	18
toolbar_FormEvt.....	19
toolbar_Preferences.....	19
toolbar_ProcessInit	20
toolbar_Redraw.....	20
Adding Buttons and Popups	21
toolbar_Button_Add	21
toolbar_Popup_Add.....	23
toolbar_SPopup_Add.....	24
toolbar_Line_Add.....	24

Toolbar and Button Properties	25
toolbar_Button_Enabled_Get	25
toolbar_Button_Enabled_Set	25
toolbar_Button_Icon_Get	26
toolbar_Button_Icon_Set	26
toolbar_Button_Importance_Get	26
toolbar_Button_Importance_Set	27
toolbar_Button_Method_Get	28
toolbar_Button_Method_Set	28
toolbar_Button_Selected_Get	28
toolbar_Button_Selected_Set	29
toolbar_Button_Shortcut_Get	30
toolbar_Button_Shortcut_Set	30
toolbar_Button_Text_Get	31
toolbar_Button_Text_Set	31
toolbar_Button_Tooltip_Get	31
toolbar_Button_Tooltip_Set	32
toolbar_Button_Visible_Get	32
toolbar_Button_Visible_Set	32
toolbar_Popup_Method_Get	33
toolbar_Popup_Method_Set	33
toolbar_Popup_Shadow_Get	33
toolbar_Popup_Shadow_Set	34
toolbar_SetBackgroundPicture	35
Customising the Toolbar	36
Demo Database	37
Contributing to the Component	38

Introduction

The Toolbar Component allows developers to create dynamic toolbars of rollover buttons on the forms and dialogs of their 4D applications.

Since 4D version 6.5 it has been possible to create rollover buttons in 4D using picture buttons and a picture containing each frame of the rollover. Although this works, it's not an ideal arrangement and anyone who has had to create lots of rollovers for an application will know exactly what I'm talking about.



Then there is the problem of dynamic toolbars. Until recently, developers could create dynamic toolbars using Coolbar, although with MacOS X, this will unfortunately be no longer possible.

This component allows developers to get the best of both worlds by giving them the ability to create dynamic toolbars using dynamic rollover buttons, all with a simple API so that a little code as possible is require.

Although the source is provided with this component, this doesn't necessarily mean that developers should install the module directly, rather than as a component. The source is provided as an example to developers, but also to allow developers to evaluate the source, fix bugs if they find them (rather than wait for updates) and contribute their own ideas to the project.

In it's component form, only those methods which need to be called appear in the Explorer window which makes it appear far more compact, but also, installing the component is far easier than moving the module from the source database.

Toolbar Components

The Toolbar Component is actually made from more than one component, giving the developers the choice over how much or little they want to install. To get the basic functionality of the toolbar component, the developer must install both the Rollover Component and the Toolbar Component, plus have some pictures available for use as icons, button backgrounds and toolbar backgrounds (provided in the Toolbar Picture Library). The following details the components provided.



Rollover Component

This component (available separately on the mitchenall.com website, but included with the toolbar component) is used to dynamically create rollover buttons and popups for display in the toolbar. This **MUST** be installed for the toolbar component to function.



Toolbar Component

This contains the core functionality of the component including the toolbar master form. This **MUST** be installed for the toolbar component to function.



Toolbar Extras Component

Contains the standard input and output form toolbars, and the associated methods. These have all been left public methods so that you can customise them to your heart's content. It is not necessary to install this component, but new users might find the default toolbars useful, especially as sample code.



Toolbar Workshop Component

Contains the workshop dialog. This new dialog allows you to test various features of the component by building toolbars using the dialog, but also creates the code needed to generate the toolbar in the dialog. Like the “Extras” component, it is not necessary to install this into your structure.



Toolbar Picture Library

Contains all the sample pictures we've used in the standard toolbars and other sample toolbars, including the icons, the button backgrounds and the toolbar backgrounds.

Installing the Components

Installing the component(s) in your application is the same as with other components, however, as there is more than one component involved, you need to ensure that the components are installed in the correct order.

The components are fairly small in size, so for the core functionality, your structure file will only grow about 400-500k after installation.

The following steps should allow you to easily install the necessary items.

1. Create a back-up of your structure file (you should do this whenever moving objects to a structure file using 4D Insider).
2. Ensure the structure file is in good condition using 4D Tools, and if possible, SanityCheck from Committed Software.
3. Put the TextProperties, RolloverAffix and ToolbarAffix plug-ins into your Mac4DX and Win4DX folders, then open your structure with 4D Insider.
4. Choose “Install/Update...” from the “Component” menu and install the Rollover Component. This contains the code required to dynamically create rollover buttons.
5. Next, install the Toolbar Component. This component contains all the core toolbar code and the toolbar master form and is the minimum you must install to use the component in anyway.
6. To install the Toolbar Picture Library, simply open it with 4D Insider and drag the require pictures into your structure.

At this point, the core elements for the toolbar component to function have been installed. If you want to install the Toolbar Extras and Toolbar Workshop component, you can now do so.

In order to initialise the component in your application, you must call the *toolbar_Startup* routine from your “On Startup” method, before any toolbars are used.

On Startup

It is extremely important that the *toolbar_Startup* method is called before any toolbar functionality is used in an application. You can either call *toolbar_Startup* from directly from your On Startup method, or another method that is called from it. It must be called.

Process Stack Requirements

During testing, we found that there is a reasonably large stack requirement for the toolbar in interpreted mode. This is not a problem in a compiled database, only in interpreted mode.

If you create processes using the New Process command, you may need to give the process at least 64k of stack. If you let 4D create processes for you, you may need to change the default stack allocation using 4D Customizer.

Adding a Toolbar to a 4D Form

The component contains a master form that needs to be included on any form that is going to have a toolbar. The master form contains all the buttons, lines, etc, which are necessary.

Example 1 – Creating an Output Form with a Toolbar

1. Open the form wizard and select a “List form” and the template “Look 3D (Subform)”, and choose the fields you want to display.
2. Click the “Advanced...” button.
3. From the “Options” tab, choose to display a Form Title on the output form.
4. Edit the new form in the Form editor and if you wish, save the wizard settings as a new template.
5. Delete the Form Title, this was just added as it is just the right height to allow our toolbar to fit above the column titles.
6. Choose the `Toolbar_Master` form as the inherited form.
7. Ensure that the form events [On load](#), [On Unload](#), [On Resize](#) and [On Timer](#) are selected for the form. This is extremely important.
8. In the form method, call `toolbar_FormEvt` passing the current form event at the top of the method (i.e. for all form events).
9. In the On load event of the form, call the method `toolbar_Std_OutputForm`.
10. You have now create a new output form with a cool looking standard toolbar. If you display this form in it’s own window in the run-time environment, the buttons will resize dynamically according to the window size.

Minimum Requirements for a Form

Any form needing a toolbar must inherit the `Toolbar_Master` form. If the form you want to add a toolbar to is already inheriting a form, you need to make this form inherit the `Toolbar_Master`. This form contains all the picture buttons, lines and other objects needed to create the toolbar.

The toolbar needs to trap certain form events in order to keep the internal state up to date and to refresh the toolbar at the correct times. In the form properties, ensure that the follow events have been checked:

- [On Load](#) (initialises current toolbar)
- [On Unload](#) (clears current toolbar)
- [On Resize](#) (resize buttons to fit windows size)
- [On Timer](#) (used when a semi-popup is clicked)

It is extremely important that these events have been activated in order for the toolbar to function properly.

Then in the form method, you must call `toolbar_FormEvt` and load whichever toolbar you want for the form, e.g.

```
`Form Method: [Contact]Output  
`Author: Mark Mitchenall
```

```
C_LONGINT($formEvent)  
$formEvent:=Form event
```

```
`call the standard toolbar form events  
toolbar_FormEvt($formEvent)
```

```
Case of
```

```
: ($formEvent=On load)  
`load standard output form toolbar  
toolbar_Std_OutputForm
```

```
End case
```

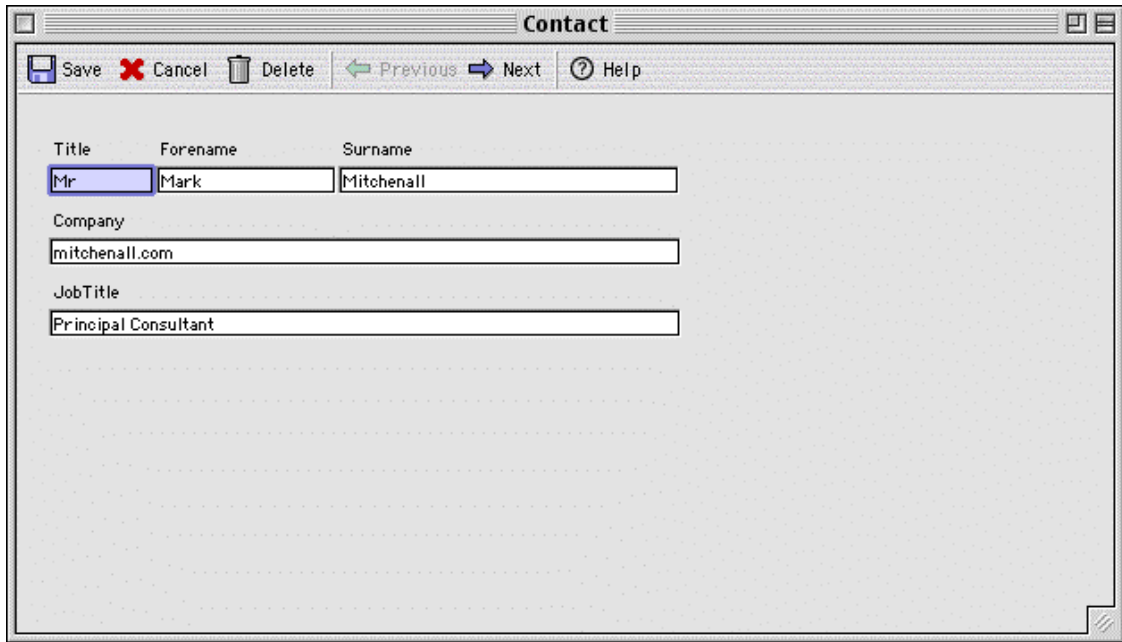



Figure 2- toolbar_Std_InputForm

Defining your own toolbars

When you create a toolbar, it is easiest to create a new method containing all the declarations for the toolbar, rather than putting them directly in the form method. This makes it easy to put all the toolbar display code in one place so that it can be easily called from other objects on your form if necessary.

The best way to see how a toolbar is defined is to look at the two example toolbars provided as public routines in the component.

When the On Load event for a form runs, the *toolbar_FormEvt* method is run and creates a new blank toolbar reference for the form. You can then add buttons, popups, semi-popups and lines to this toolbar using the simple commands provided.

Because the toolbars are created at run-time, you can dynamically modify the toolbar at any time.

One Toolbar Per Form

Just one toolbar is allowed per form at this time. However, as stated previously, you can change the contents of this toolbar at runtime, even while the toolbar is being displayed.

Adding Buttons and Button Names

To define a new toolbar, you must first call *toolbar_Clear* to ensure that the toolbar is completely initialised for this form. Then each button must be added to the toolbar in the order in which they will appear from left to right. To make referencing the buttons easier, each button is given a name. A good idea is to define a set of string constants for your button names to ensure that they're easier to cross-reference.

Often, the name of the button is not important, but if you want to change the state of a button, e.g. to disable it, or to change the button text, you need to know the button's or popup's name. The name can be of any length, but ideally it should be short and easy to remember.

To create a toolbar with just a "Done" button, you could use the following code snippet.

```
`Form Method: [Dialog]Dialog
`Author: Mark Mitchenall
```

```
C_LONGINT($formEvent)
$formEvent:=Form event
```

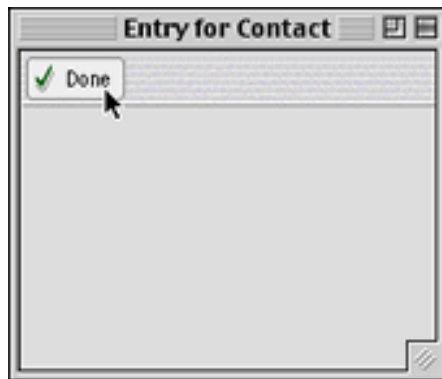
```
`call the standard toolbar form events
toolbar_FormEvt($formEvent)
```

Case of

```
 : ($formEvent=On load)
   `display just a "Done" button
   toolbar_Clear
   toolbar_Button_Add ("done"; "Done"; "smIcon_Done"; "CANCEL")
   toolbar_Redraw
```

End case

The above code when placed in the form method of a form that has inherited the Toolbar Master will display a single button in the toolbar using the icon called "smIcon_Done" from the picture library and the text "Done". If you click on the the button, "CANCEL" will be called and the dialog will close.



In the above example, the button's name is "done". This means that if we later need to disable the button, we can simply call...

```
toolbar_Button_Enabled_Set ("done"; False)
```

Button Types

The component supports 3 types of buttons, each generated by the rollover component. These are buttons, popups and semi-popups.

Buttons work the same as any other button on your form. They only react to a click and call a single method when that happens.

Popups are slightly more complicated. Like a button, they can call a single method when they're clicked and then display a popup menu using the **Pop up menu** command. It is also possible to associate a text array with the popup and have the component automatically display the popup based on this array. In this case, you can optionally have your own routine called after an item has been successfully selected from the popup.

Semi-popups consist of a button and a popup menu. If the user simply clicks the button portion, the button action is called. If the user clicks and holds the button portion, the popup action is called. However, if the user clicks the popup portion, the popup action is called immediately. They support all the features of a button and a popup and are useful in a few circumstances.

Button and Popup Actions

When you add a button to a toolbar, a *clickedMethod* is defined for the button. If no method is specified, a standard method will be called based on the name of the button when the button is clicked. You can change the *clickedMethod* for a button after the toolbar has been defined, but there should be a valid method.

With popup menus, you can set a *popupMethod* that is called when the user clicks on the popup menu. Popup menus are implemented using a picture button with an invisible button over the top. The reason for this is that the On Clicked event runs on the mouse down, rather than after the user releases the mouse. This allows us to display a popup menu using 4D's **Pop up menu** function.

Alternatively, you could also set a shadowArray for a popup menu and let the toolbar component build the menu for you. Using the shadowArray you still have the option to set a *popupMethod* that is called after the shadowArray popup has been displayed and the user selected something. This is covered under the command, *toolbar_Popup_Shadow_Set*.

With semi-popups, i.e. buttons that have both a popup action and a button action, you need to define both the *popupMethod* and *clickedMethod*. Future versions of the toolbar Component will let you define a semi-popup with just a shadowArray, but it was important to get a freeze on features for version 1.0. It is still OK to use a shadowArray without a *popupMethod* with semi-popups, but the *clickedMethod* must be defined.

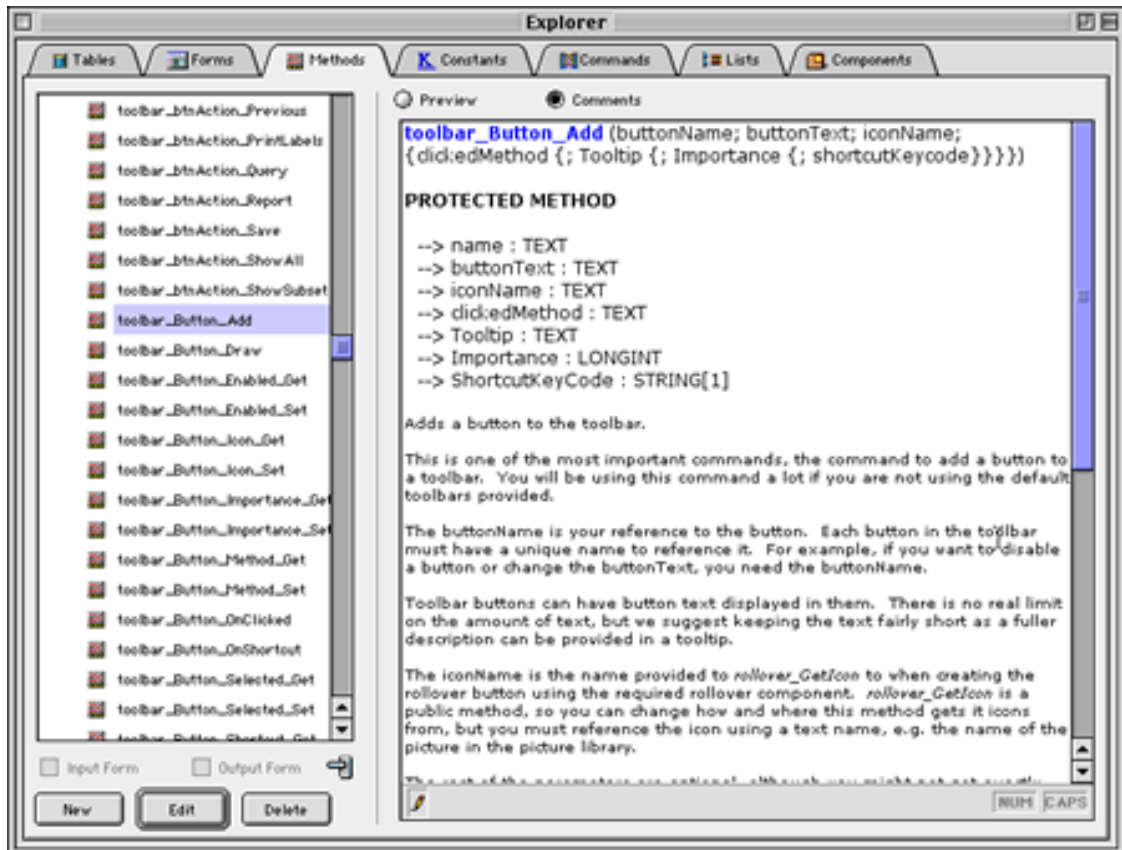
toolbar_Redraw

This command redraws the toolbar and should be called after nearly any command that modifies the appearance of the toolbar. If you modify any toolbar settings in the toolbar arrays directly, calling *toolbar_Redraw* will not necessarily redraw the toolbar. This is why it is extremely important to only modify toolbar button settings using the commands provided. Also, by sticking to the commands provided, it makes it easier to change the implementation without affecting the API.

Once the toolbar has been initialised on the form and drawn for the first time, it should only need to be manually redrawn if something changes in the appearance of the toolbar, e.g. changing a button's text, hiding a button, etc. The only exception is *toolbar_Button_Enabled_Set* as this routine can disabled the button without the toolbar needing to be redrawn.

Basic Command Reference

The following part of the documentation covers the commands in the component. For up to date information about each routine in the component, you can also look at the “Explorer Comments” which have been included. These are normally more up to date as they’re updated as commands are created and modified.



Initialisation Commands

toolbar_Startup

toolbar_Startup

Starts up the toolbar Component. This **MUST** be called in the **On Startup** method of your database if you intend to use this component in any way whatsoever. This method initialises all the variables used by the component and loads the global preferences.

toolbar_Clear

toolbar_Clear

Clears the current toolbar of all items. This should be called before adding any buttons to a form, e.g. in the [On Load](#) event before calling *toolbar_Button_Add* or any of the other commands for adding items to the toolbar.

toolbar_FormEvt

toolbar_FormEvt ({formEvent})

→	FormEvent	Longint
---	-----------	---------

This is one of the most important routines in the component. It must be included in the form method of any form that contains a toolbar. The form should also have the following events activate and this method called for all of them.

- [On Load](#)
- [On Unload](#)
- [On Resize](#)
- [On Timer](#)

If called with no parameters, this method uses the current form event. If you pass a form event value, this event will be called instead. You should not pass anything other than the current form event value unless you know exactly what you're doing.

toolbar_Preferences

toolbar_Preferences

This method is called by `toolbar_Startup` to configure the options for the toolbar component. This is a public method, so that it's developers can set their own options easily.

See also: `toolbar_SetBackgroundPicture`

toolbar_ProcessInit

toolbar_ProcessInit

This method should be called in any new process which uses any dialogs which contain toolbars. If you use the [Process Component](#), another freeware component from mitchenall.com, calling this method at the start of each new process doesn't have to be difficult to remember.

toolbar_Redraw

toolbar_Redraw

Redraws the current toolbar. This should be called whenever you feel the toolbar needs redrawing. The component is clever enough to know when the toolbar really does need redrawing, so you won't be wasting too many cycles by over calling this method when the toolbar doesn't actually need a redraw.

One time when you should call this method is after opening a separate window of a different size that has a toolbar from a form or dialog which is also displaying a toolbar. e.g.

` Open a dialog from the current form

```
$winRef: =Open form window ([myTable];"myDialog";Regular Window)  
DIALOG ([myTable];"myDialog")  
CLOSE WINDOW
```

` redraw the current toolbar after the previous window has closed

```
toolbar_Redraw
```

Adding Buttons and Popups

The commands in this section are for adding buttons, popups and semi-popups to the toolbar. These routines would normally be called in the On Load event of a form, but can also be called after the toolbar has been drawn.

There is a limit on the number of items which can be added to a toolbar, currently 20. When this limit is reached, these commands will raise an error using *toolbar_ERROR*.

toolbar_Button_Add

toolbar_Button_Add (name; buttonText; iconName; actionMethod; toolTip; importance; keystroke)

→ name	Text
→ buttonText	Text
→ iconName	Text
→ actionMethod	Text
→ toolTip	Text
→ importance	Longint
→ keystroke	String[1]

Adds a button to the toolbar.

This is one of the most important commands, the command to add a button to a toolbar. You will be using this command a lot if you are not using the default toolbars provided.

The *buttonName* is your reference to the button. Each button in the toolbar must have a unique name to reference it. For example, if you want to disable a button or change the *buttonText*, you need the *buttonName*.

Toolbar buttons can have button text displayed in them. There is no real limit on the amount of text, but we suggest keeping the text fairly short as a fuller description can be provided in a tooltip.

The *iconName* is the name provided to *rollover_GetIcon* to when creating the rollover button using the required rollover component. *rollover_GetIcon* is a public method, so you can change how and where

this method gets it icons from, but you must reference the icon using a text name, e.g. the name of the picture in the picture library.

The rest of the parameters are optional, although you might not get exactly what you want unless you specify at least the clickMethod.

The clickedMethod is the method which is **EXECUTE**'d when a user clicks on the button. If this parameter is omitted, the component automatically sets a method called "toolbar_btnAction_" plus the name of the button. If you use the provided example button actions, these are then easy to use because you only need to specify the buttonName, buttonText and iconName to have a button set for your toolbar, e.g.

```
` add a button with the clickedMethod "toolbar_btnAction_Cancel"  
toolbar_Button_Add ("Cancel"; "Cancel"; "smIcon_Cancel")
```

The tooltip parameter does exactly what it says, i.e. adds a tooltip to the button. If a tooltip is set for a button, this is automatically added into the tooltip, so there is no need to put this in your tooltip text yourself.

The Importance of a button relates to how the toolbar gets resized and which buttons get resized first. For more information about button importance, there is a fuller description in *toolbar_Button_Importance_Set*.

The ShortcutKeyCode is a 1 character string which represents the shortcut key for the button. For more information see *toolbar_Button_Shortcut_Set*.

toolbar_Popup_Add

toolbar_Popup_Add (name; buttonText; iconName; popupMethod; toolTip; importance)

→ name	Text
→ buttonText	Text
→ iconName	Text
→ popupMethod	Text
→ toolTip	Text
→ importance	Longint
→ keystroke	String[1]

Adds a popup to the toolbar. This method is similar to *toolbar_Button_Add*, except that no keyboard shortcut can be assigned to a popup and instead of a clickedMethod, there is a popupMethod parameter. This parameter is almost identical except that popup methods are expected to display a popup menu using the 4D built-in **Pop up menu** command.

See also: [toolbar_Button_Add](#), [toolbar_SPopup_Add](#)

toolbar_SPopup_Add

toolbar_SPopup_Add (name; buttonText; iconName; clickedMethod; popupMethod; toolTip; importance; keystroke)

→ name	Text
→ buttonText	Text
→ iconName	Text
→ clickedMethod	Text
→ popupMethod	Text
→ toolTip	Text
→ importance	Longint
→ keystroke	String[1]

Adds a semi-popup to the toolbar. This type of toolbar item requires both a `clickedMethod` and a `popupMethod`. This is because the user could either click the button section (to get the button action), click and hold the button section (to get a popup), or click the popup portion (to get the popup quicker).

All the other parameters are the same as for Buttons and Popups.

Semi-Popups can also have shadow variables defined for them. For more details see the command *toolbar_Popup_Shadow_Set*.

toolbar_Line_Add

toolbar_Line_Add

Adds a separator line to the toolbar. These are the vertical lines which can appear between buttons to help group buttons of a similar nature together.

Toolbar and Button Properties

These commands allow you to read and modify the properties of buttons and popups on the toolbar. Because some properties are shared by all types of item, the

toolbar_Button_Enabled_Get

toolbar_Button_Enabled_Get (buttonName) --> enabledState

→	buttonName	Text	
←	EnabledState	Boolean	True if the button is enabled

Gets the enabled state of a button on the toolbar. This method can be applied to buttons, popups and semi-popups with the same effect.

toolbar_Button_Enabled_Set

toolbar_Button_Enabled_Set (buttonName; enabledState) --> enabledState

→	buttonName	Text	
→	EnabledState	Boolean	

Sets the enabled state of a button on the toolbar. This method can be applied to buttons, popups and semi-popups with the same effect.

The button can be disabled/enabled without needing to redraw the whole toolbar.

toolbar_Button_Icon_Get

toolbar_Button_Icon_Get (buttonName) --> iconName

→	buttonName	Text
←	iconName	Text

Returns the iconName set for the button. Can also be applied to popups and semi-popups.

toolbar_Button_Icon_Set

toolbar_Button_Icon_Set (buttonName; iconName)

→	buttonName	Text
→	iconName	Text

Sets the icon of the button and if different than current icon, invalidates the toolbar and redraws it.

toolbar_Button_Importance_Get

toolbar_Button_Importance_Get (buttonName) --> importance

→	buttonName	Text
←	importance	Longint

Returns the importance of a button.

This routine probably isn't that useful in everyday usage, but it is used by the *toolbar_Workshop* dialog in the test database and has been included in the API for completeness.

toolbar_Button_Importance_Set

`toolbar_Button_Importance_Set (buttonName) --> importance`

→	buttonName	Text
→	importance	Longint

Sets the importance of a button. This is the value which is used to decide which buttons to minimize first on a toolbar when the window can't hold all the buttons at full size.

For example, if you feel that your "Save Record" button is far more important than the other buttons, you can give it a high importance number so that it is the last button to resize. However, there will still be odd times when this button isn't the last to resize if there are other less importance, but smaller buttons, which can fill up a gap made by resizing the larger button.

If you have a "Help" button on each for, it might be nice to make this button most important for new users, but less important for experienced ones. You could also make the importance of buttons vary depending on the users button usage.

If you don't worry about the importance of buttons, the buttons will simply be resized from right to left until the toolbar fits. If after resizing a group of buttons from the right, there is space for one of the smaller buttons which has resized to fit the gap, it will be shown maximised so that the toolbar fits as snugly as possible.

toolbar_Button_Method_Get

toolbar_Button_Method_Get (buttonName) --> onClickedMethod

→	buttonName	Text
←	onClickedMethod	Text

Returns the onClickedMethod set for the button, i.e. the method which is called when the button is clicked. Can also be applied to semi-popups. To get the onPopupMethod of a semi-popup, you need to call *toolbar_Popup_Method_Get* (buttonName).

toolbar_Button_Method_Set

toolbar_Button_Method_Set (buttonName; onClickedMethod)

→	buttonName	Text
→	onClickedMethod	Text

Set the 4D method that will be called for the button or clicked action of a semi-popup. This doesn't change the appearance of the toolbar, so the toolbar does not need redrawing after a call to this command.

toolbar_Button_Selected_Get

toolbar_Button_Selected_Get (buttonName) --> selected

→	buttonName	Text
←	selected	Boolean

Returns the selected state of a button.

See also: [toolbar_Button_Selected_Set](#)

toolbar_Button_Selected_Set

toolbar_Button_Selected_Set (buttonName; selected)

→	buttonName	Text
→	selected	Boolean

Sets the selected state of a button. For example, you may have a toolbar for use with a 4D Write area. If this toolbar has a "Bold" and "Italic" button, you may want to show these buttons in their depressed state if the selected text is bold or italic. This method lets you do this by setting whether the button is selected or not.

e.g.

```
If (wr get text property ($area;wr bold) = 1)
  toolbar_Button_Selected_Set ("bold"; True)
Else
  toolbar_Button_Selected_Set ("bold"; False)
End if
```

```
If (wr get text property ($area;wr italic) = 1)
  toolbar_Button_Selected_Set ("italic"; True)
Else
  toolbar_Button_Selected_Set ("italic"; False)
End if
```

toolbar_Button_Shortcut_Get

toolbar_Button_Shortcut_Get (buttonName) --> keyCodeChar

→	buttonName	Text
←	keyCodeChar	Text

Returns the keyCodeChar set as a shortcut for the button. Can also be applied to semi-popups, but not to popups as these do not have keyboard shortcuts.

Currently, only a limited set of shortcuts are allowed in the toolbar and they're referenced using a simple character.

See also: [toolbar_Button_Shortcut_Set](#)

toolbar_Button_Shortcut_Set

toolbar_Button_Shortcut_Set (buttonName; keyCodeChar)

→	buttonName	Text
→	keyCodeChar	Text

Sets the shortcut key of the button. This method can also be used to set the shortcut key for the semi-popup button actions.

The shortcut key must be passed as 1 character string. For special keystrokes, the following keycodes can be used.

- Cmd-Left Arrow - Char(28)
- Cmd-Right Arrow - Char(29)
- Cmd-Up Arrow - Char(30)
- Cmd-Down Arrow - Char(31)
- Enter Key - Char(3)
- Escape - Char(27)
- Cmd-Backspace - Char(8)
- Cmd-Del - Char(127)

toolbar_Button_Text_Get

toolbar_Button_Text_Get (buttonName) --> buttonText

→	buttonName	Text
←	buttonText	Text

Returns the buttonText set for the button, i.e. the text displayed directly in the button. Can also be applied to popups and semi-popups.

toolbar_Button_Text_Set

toolbar_Button_Text_Set (buttonName; buttonText)

→	buttonName	Text
→	buttonText	Text

Sets the text of the button and if different than current text, invalidates the toolbar and redraws it.

toolbar_Button_Tooltip_Get

toolbar_Button_Tooltip_Get (buttonName) --> tooltip

→	buttonName	Text
←	tooltip	Text

Returns the tooltip set for the button. Can also be applied to popups and semi-popups.

toolbar_Button_Tooltip_Set

toolbar_Button_Tooltip_Set (buttonName; tooltip)

→	buttonName	Text
→	tooltip	Text

Sets the tooltip for a button.

toolbar_Button_Visible_Get

toolbar_Button_Visible_Get (buttonName) --> visibilityState

→	buttonName	Text	
←	visibilityState	Boolean	True if the button is current visible

Returns the visibility state of the button.

toolbar_Button_Visible_Set

toolbar_Button_Visible_Set (buttonName; visibilityState)

→	buttonName	Text	
→	visibilityState	Boolean	True if the button is current visible

A bit like 4D's own **SET VISIBLE** command, only this sets the visibility of an individual button the form.

After calling this routine, you must call *toolbar_Redraw*.

toolbar_Popup_Method_Get

toolbar_Popup_Method_Get (buttonName) --> onPopupMethod

→	buttonName	Text
←	onPopupMethod	Text

Returns the onPopupMethod set for the popup, i.e. the method which is called when the button is clicked. Can also be applied to semi-popups. To get the onClickedMethod of a semi-popup, you need to call *toolbar_Button_Method_Get* (buttonName).

toolbar_Popup_Method_Set

toolbar_Popup_Method_Set (buttonName; onPopupMethod)

→	buttonName	Text
→	onPopupMethod	Text

Sets the method which will be called for the popup for semi-popup when the popup action is activated.

toolbar_Popup_Shadow_Get

toolbar_Popup_Shadow_Get (popupName) --> shadowArray

→	buttonName	Text
←	shadowArray	Pointer

Returns a pointer to the shadow array for the popup.

toolbar_Popup_Shadow_Set

toolbar_Popup_Shadow_Get (popupName; shadowArray)

→	buttonName	Text
→	ShadowArray	Pointer

The Shadow array is a Text Array that the popup menu on the toolbar uses for the popup. This is a time-saving feature which allows the you to create a text array with the items you want for the popup, store the currently selected item in the normal way, and have the toolbar component create the popup for you. You can still set an onClicked method, but now the method will be called after the popup is displayed and selected from and passed the number of the item that was selected from the popup.

The optional parameter, showZeroElement, allows you to choose whether you want whatever is stored in the zero element of the ShadowArray in the popup in the toolbar. The default is True, i.e. when you select an item from the popup, the popup button text will change to the value of that item. If you want to display the button text instead, you can pass False for this parameter.

Example Usage:

```
ARRAY TEXT(myPopupArray;4)  
myPopupArray{1}:="Popup Item 1"  
myPopupArray{2}:="Popup Item 2"  
myPopupArray{3}:="Popup Item 3"  
myPopupArray{4}:="The Last Item"
```

```
toolbar_Popup_Add ("Sample";"buttonText";"icon";"popupMethod")  
toolbar_Popup_Shadow_Set ("Sample";->myPopupArray)
```

toolbar_SetBackgroundPicture

toolbar_SetBackgroundPicture (backgroundPicture)

→ backgroundPicture	Picture
---------------------	---------

Allows you to set the picture used as the background image for the toolbar. This could be different depending on platform, e.g. on Mac, you might have the stripy MacOS X style background, but on Windows, you could use a Windows XP style background.

The method automatically "tiles" the picture across the toolbar background.

Normally called from *toolbar_Preferences*.

Customising the Toolbar

Most of the customisations that can be made are in the buttons themselves, and this is completely controlled by the Rollover Component. However, there are a few Toolbar specific customisations that can be made using the *toolbar_SetProperty* command.

The appearance of the buttons is completely controlled by the Rollover Component. These settings can be modified either at startup or runtime in the *rollover_Preferences* method. Be aware, however, that if you change any rollover settings, you must clear the rollover cache in order to see the changes.

Demo Database

The demo database shows how all the various features of the toolbar component can be used. To see nearly all the features of the component in action, and find example code showing how to implement, the demo is the best place to look.

Contributing to the Component

Like many mitchenall.com components, the Toolbar Component is completely open-source. For us, the beauty of open-source is that it enables other developers to use the code in other ways, and so that the code can act as example code. It also means that if a developer finds a bug in the component, they have the ability to dive into the code and fix it for themselves, rather than having to rely on us. We can't always give a great deal of time to our freeware offerings, so by providing the source code, we can let others provide support when we're unable to.

We're always interested in seeing enhancements which people have made to the component, and where applicable, add them to the component so that others can also get the benefit of your work. All contributions are accredited to the developer who made them.

Whether you have some examples of the component in use, some new button designs or icons you want to share, additional documentation or corrections, bug-fixes or enhancements, we would love to hear from you.

Also, if you have other translations for the standard buttons to include in the demo database as samples, we'd be happy to receive a copy of the STR# resource for inclusion. So far, we have English, and Chinese, but we'd love to be able to add many other languages. If the language requires a specific font to be display correctly, please let us know the font settings you use (on both Mac and Windows) so that we can include this also.