MV_MENU-TUTORIAL

VER 2.01

This tutorial will teach you how to use the mv_Menu component in dbase.
mv_Menu is a dbase custom class, with which you can build a menu for your dbase forms.

Why another menu ? you may ask, dbase has a built-in class to create menus, it even comes with a designer to be able to do the design work visually.
But the mv_Menu has some important advantages over the stockmenu:

- you can use any icon or bmp as picture in front of the menuitems.
- the mv_Menu can be docked at the top, left, right or bottom of your form.
- not only SDI Windows can have a menu, but all your MDI-Forms can have their own custom menu.
- most important of all the mv_Menu mimics the behaviour of the more modern menus like in Office2003,Office2007 and even Vista (called Ribbon-menu)

There is one drawback however, you will have to code the menu by hand. There is no designer available yet...

Read on, to learn how to do that.

A) Registering the ocx

The working horse of the mv_Menu is an activex object. Like every other COM-Object, you need to register it before you can use it.
This has to be done on your developing machine but also on your clients machine.

You need to run the system program regsvr32 with the path of the ocx as parameter.

regsvr32 c:\vdblogic\com\mv_menu.ocx

Where you should replace "c:\vdblogic\com\" with the path where you placed the ocx file.If the file path contains a space, enclose it in double quotes, e.g.

Regsvr32 "c:\Program Files\com\mv_menu.ocx"

Please refer to your install program (Inno, Installshield,...) to see how to register the component automatically with your application.

B) Getting started

The dbase part of the component is a custom classfile (delivered as mv_Menu.co) that holds all the classes you will need to build a sophisticated modern-looking menu.

At the base of the hierarchy there is the mv_Menu class itself: it is the container that represents the menu bar at the top of your form.
At the beginning, this menu bar is completely empty and the act of building your menu will consist in filling this empty menu bar with items and sub items and sub-sub-items…

Let's do it together, step by step. We will first build our starting point: Open the source editor and type the following:

```
#include  mv_menu.h
Class own_menu(oparent) of mv_menu(oparent) custom

Endclass
```

Well it is not much, but we just made a subclass from our main-menu-object. Every guru in the dbase field will agree that it is a good idea to never use the native control itself, but rather use subclassed controls. All we've done is: follow the rules! The include directive will help us with some predefined constants in order to make the code easier to read. Note the usage of the "Custom" descriptor, this will prevent the form designer to stream properties of our menu each time we change something in the form.
Of course we want to see it in action, so our next step will be to put our menu on a form.

Close the sourceeditor and save the code as own.cc, choose the tab “Other” in the dbase navigator, click on the file mv_menu.co and press <F2>, do the same with own.cc. Alternatively, you can type “set procedure to <filename.ext>” in the command window. (Depending on your settings, files with the extension *.co may not be visible in the dbase navigator.) You just brought the components into scope for the form designer.
Now start the form designer with a new empty form. Press the F12-key to bring up the source code editor. You’ll see something like:

```
class NeuForm of FORM
   with (this)
      height = 16
      left   = 38
      top    = 1
      width  = 40
      text   = ""
   endwith
endclass
```

Go to the top of the code, place the cursor above the line that says: “ ** End Header” and add the following…

```
Set procedure to mv_Menu.co additive
Set procedure to own.cc additive
```

Now go to the line between endwith and endclass and add :

```
This.my_menu = new own_menu(this)
```

We have instantiated an object of the class own_menu and stored a reference in the property my_menu of the object “this”. We are in the constructor of the form, so “this” is nothing else than the form itself. With that line, we have created the Menu and attached it to the form.

- Note : *the parameter passed to the new operator must be a form-object. If you try to attach the menu to something else (entryfield or pushbutton) you’ll get an error message.*

- Note : *you can’t drag the menu-control from the component palette. Most of its properties are hidden and the streaming engine would get confused if you try to do so. You have to add the menu in the source code.*

- Note : *the onopen event of the form is to late to attach the menu. It has to be done in the constructor.*

Change back to the designer view (F12) and you'll see … nothing. All the drawing work is performed at run-time, during design, the control will be invisible. Ok, then let's run the form, save it as test.wfm and start it. Bummer, nothing. Well actually, there is something but you can't see it. Open the form in the designer again and change the form's colornormal property to: silver and run the form. Now you should see something like:

Here we can see our empty menu bar. Nothing special, except that it is bound to a MDI-form. If you find it useful, you could easily open two mdi-forms simultaneously, with two different menu bars (depending for instance on some data shown in the form). This kind of thing can't be done with the stock-menu-object.

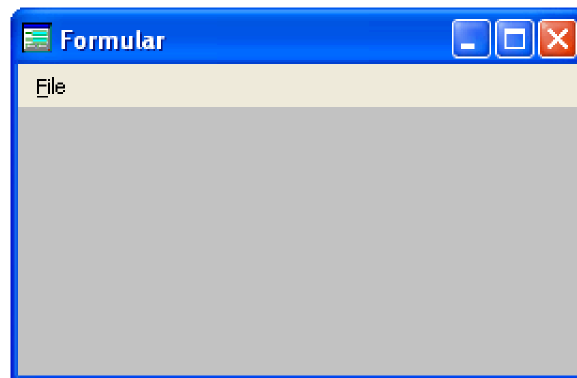
The next step will be to populate the mv_menu.

C) Adding items to the menu bar

The mv_Menu is built with two different objects: the menuitem-object and the popupitem-object. The menuitem-object is used as a container for more objects (menuitems or popupitems) and the popupitem-object is used as the object that will launch a function, when it is clicked. The menuitem will open submenus if it is clicked, but has no separate programmable eventhandler. Open the own.cc file in the source editor and add some lines to get the following.

```
#include  mv_menu.h
Class own_menu(oparent) of mv_menu(oparent) custom
   Function initialize
      This.m_item1 = new menuitem(this, "&File")
   return
Endclass
```

Close the own.cc and run your testform again. AHA, now we start getting something.

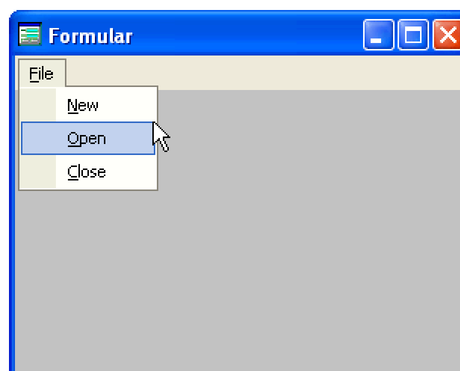


We added a method to the own_menu class with the name “initialize”. This method is very important, because it will be called automatically when the menu is created (it is a sort of onopen-eventhandler for the menu bar) In this method, we can write all the construction code for the items and sub items, like: this.m_item1 = new menuitem(this,”&File”)

The syntax follows exactly the rules of the normal dbase OOP language, the new operator will instantiate a new object of the class menuitem, its parent is “this” and its caption is “File”. Because “this” is the menubar, the menuitem object is now a child of the menubar. As said earlier, the menuitem itself is a container and can contain other items; we will add some popupitem objects. Add the following code in the initialize method of the own_menu class

```
This.m_item1.p1 = new popupitem(this.m_item1,  "&New")
This.m_item1.p2 = new popupitem(this.m_item1,  "&Open")
This.m_item1.p3 = new popupitem(this.m_item1,  "&Close")
```

If you run the form, you will see that we slowly approach the look and feel of a modern menu.



Next we will look at the beauty of object oriented programming.

D) What about inheritance ?

In the previous chapter, we learned how to populate our menu bar. One way of building a menu for an application would be to put all the necessary items in the .cc file and create it with new( ). The drawback of this method is that your menu would be unique to this one application; it would be difficult to reuse the code (menu) in other applications. Fortunately, dbase is a full featured object oriented programming language and we will use its capabilities to make our life easy: we will add a new class that inherits from our own_menu. Open the own.cc file in the source-editor and go to the bottom of the file. There, after the “endclass” statement, you add the following:

```
Class next_menu(oparent) of own_menu(oparent) custom
   Function initialize
      Super::initialize( )
      This.m_item2 = new menuitem(this, "&Edit")
   return
Endclass
```

Here we subclassed the next_menu from the own_menu, that means that all the code and all the objects of our own_menu are now part of the next_menu and we can easily add more functionality to this next_menu.
The line : super::initialize( ) insures, that that all the initialization code of the father is executed correctly. Each time you build such sort of inheritancechain, you must make sure, that all the parents and grand-parents have their chance to be built properly, hence the call of super::initialize.
Next step is adding another menuitem to the menubar with caption “Edit”

- Note : *the ampersand is used as with the stock menu class, the letter that is preceded by the “&” sign will get an underscore and can be used as a shortcut with the ALT-key*

Before we look at the menu in the form, we add some more items.

```
This.m_item2.m1 = new menuitem(this.m_item2, "&Action")
This.m_item2.m1.p1 = new popupitem(This.m_item2.m1, "&Copy")
This.m_item2.m1.p2 = new popupitem(This.m_item2.m1, "&Paste")

This.m_item2.Cancel = new popupitem(This.m_item2, "Ca&ncel")
```

Close the file and run the form. Hmm, nothing changed. You have to change the class in the form too : this.MY_MENU = new next_MENU(this)

If you run the form now, you will see the updated menu. It is also clear now, what it means that a menuitem is a container for other items. The object m_item2.m1 holds two popupitems as sub-menus. It is important to know that the main menu bar can only hold menuitems.
I leave it to you to add a few more items to the menu (maybe a “Tools” and a “Help” menu).
Now that we have something to work with, we will take a closer look at the properties and methods of the mv_Menu.

E) The members of the menu bar

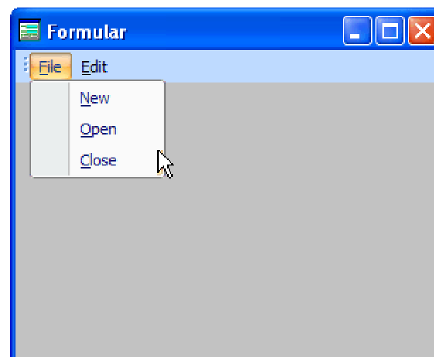
1) The Theme property and the Set_Theme methods:

With the mv_Menu you have the choice between seven different styles. These “themes” mimic the look and feel of the well known Office packages or the new “Glass-Look” introduced with Vista. In the include file, you can find the following theme-constants:

```
#define    theme_2000       0    // Office 2000
#define    theme_XP         1    // Office XP
#define    theme_2003       2    // Office 2003
#define    theme_WinXP      3    // Native WinXP
#define    theme_Whidbey    4    // Whidbey
#define    theme_2007       5    // Office 2007
#define    theme_Ribbon     6    // Vista-Look
```

If you call the Set_Theme method with one of these constants, the menu will immediately change to the look of the selected theme. The theme-property holds a numeric value between 0 and 6, which represents the currently selected theme. Add the following code at the end of the initialize method of the next_menu class in own.cc and see the result in your testform…

```
This.Set_Theme( theme_ribbon )
```



2) The attach_to_framewin property

This property will bind the menu to the _app.framewin object. Its default value is false, which means, that the menu is not bound to the framewin. If you set it to true in your class definition, the menu will appear at the top of the framewin. However, you still need to have a normal form as the parent of the control. This “parent-form” will be responsible for the lifetime of the menu. If you close this form, the mv_menu will be released and replaced by the standard dbase menu. The form property of the items will contain a reference to this parent form.

```
This.attach_to_framewin = true
```

3) The Set_Docking method

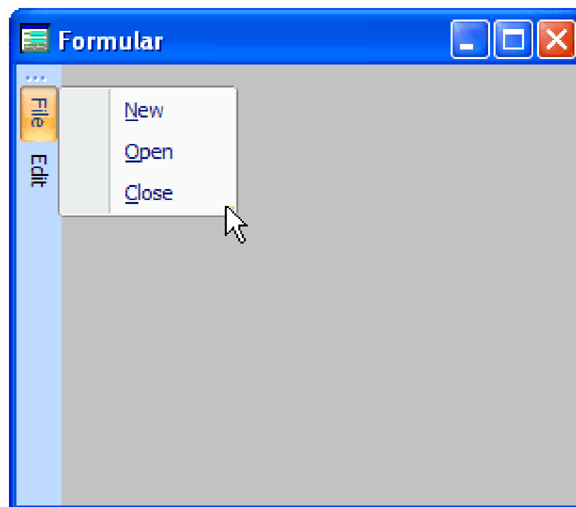
This method takes two parameters. The first is a Boolean value that indicates if the menu bar can be docked to the sides of its parent window. If this value is false, the menu takes its classic position at the top of the window and can't be moved (in this case the second parameter is ignored). If the passed value is true, the second (numeric) parameter determines where the menu bar can be docked. The following constants are defined in the include file:

```
#define    Dock_Top     0x1
#define    Dock_Bottom  0x2
#define    Dock_Left    0x4
#define    Dock_Right   0x8
#define    Dock_Float   0x10
#define    Dock_All     0x1F
```

If you want for instance, that the menu can be docked to the top and bottom line of the form, you call the Set_Docking method as follows…

```
This.Set_Docking(true, Dock_top + Dock_bottom )
```

Note that you can not change the orientation of the Captions when the menu bar is docked to the left or to the right of the window.



4) The Get_MenuHeight method

Sometimes it is handy, to know the exact height of the menu, f.i. if you want to calculate the remaining space in your form. The Get_MenuHeight method returns this height in pixels.

5) The Add_Toolbar method

Part of the mv_menu is a modern looking toolbar. This toolbar has a lot of properties, events and methods. We will discuss these later in this tutorial.
The link between the menu and the toolbar is provided by the add_toolbar method. A toolbar cannot exist without a menu. Therefore, you first have to create a menu (it could be completely empty) and then attach the toolbar to this menu with the above method.
Add_Toolbar takes two parameters : a string, which will be assigned to the toolbar and a numeric value, that controls the starting position of the tool bar (for more information, see the set_pos method of the toolbar.

```
This.t_bar = this.add_toolbar(“File” , 0)
```

This.tbar contains now a reference to the toolbar and can be used, to add controls to it.

6) The Set_Visible method

Sometimes you don’t want a menu in the form, but you want the toolbar; because a toolbar can’t exist without a menu, we need a way to hide the unwanted menu. This is achieved by the Set_Visible method. The default is set to true, if you call the function with a value of false, it will hide the menubar and the topmost control of your form will be the toolbar.

```
This.Set_Visible(false)
```

The method goes together with the m_visible property. If you want to know, if a menubar is visible, you can read this property.
If form.mv_menu.m_visible = true // the menubar is visible.

7) The Set_Iconsize method

One feature of the toolbar is, that it can contain buttons of any size and this buttons can display icons of any size. However, the size of the buttons is controlled by the mv_menu. You cannot have one button with a 16x16 pixel icon (the default) and another button with a 24x24 pixel icon. You specify the size of the toolbar-icons globally with the above function.
If you have an icon, that supports multiple screen resolutions and contains the same image in different sizes, the toolbar will automatically choose the most appropriate. If your icon contains only one image, the toolbar will inflate or shrink it to the dimension, passed to the set_iconsize method.

```
This.Set_Iconsize(24)
```

8) The Set_State and Get_State method

As you and the user can dock the menu at different positions in the form or drag the toolbar around, we need a way to save those positions. The user should find the position of the different parts of the mv_menu the same as when he left the form.
This is achieved with the get_state and set_state methods.

Get_State returns a string in XML-format, that holds all the position information of the mv_menu. You could write this string to an INI-File, or if you have many menus and toolbars in an application, you could write them to a memofield in a table. Once you open the form anew, you can call the set_state method and pass it the saved stringvalue, to position all the elements at the place where they were, when the user closed the form.
Typically you would write the following code in the onopen-event of the form

```
Function OnOpen
     Local initpos,q
     Q = new query()
     Q.sql = “select * from init where id = ‘men1’”
     q.active = true
     Initpos = q.rowset.fields[“pos”].value
     Form.mv_menu.set_state(initpos)
     q.active = false
return
```

Men1 is the key under which you saved the state of the menu and toolbar for this form.
In the canclose-event of the form, you might write something like the following :

```
Function Form_canclose
     Local initpos
     Initpos = form.mv_menu.get_state()
     ** now store the string in the table …
return
```

9) The OnUpdate event

Like the original dbase toolbar, the mv_menu has an onupdate-event. This event fires every 100 ms and can be used to update the visual appearance of menu- or toolbar-items.

For instance one could disable a navigation menuitem, if the corresponding rowset is at the endofset. It is enough to write a function with the name of “OnUpdate” in the class definition and the mv_menu will call this function on a regular basis, when the form is idle

10) Set_Fontsize and Set_FontName Methods

In some circumstances, it can be handy to change the size and name of the font that is used in the mv_menus (you should use this carefully). This can be done with the two methods Set_Fontsize, which awaits a numeric parameter, and Set_FontName, which takes a string parameter. If you pass a string, that is not in the list of available Fonts, the method automatically sets the fontname to Arial.

11) Set_MDIBackColor

This method has an effect only if the menu is bound to the _app.framewin. It controls the background colour of the MDI-client window. Three options are available:

- Automatic option (parameter –1) (Default). With this option, the mv_menu adapts the background colour of the mdi-client automatically according Microsoft standards, that is if you chose the Office2003 or Ribbon theme, the background changes to a lighter grey or a light blue.
- dbase option (pass –2). In this case, the background keeps its well known dark grey colour independent of the selected theme.
- Colour-option (parameter : 0xBBGGRR). If you pass a positive numeric value, it will be interpreted as a colour reference, the background is painted in this colour again independent of the selected theme.

So far we have seen all the members of the baseclass which is the mv_menu.
Now we'll have a closer look at the members of the menu and popupitems…

E) The members of the menu- and popup-items

1) The Clicked event

The most important member of the item controls is the clicked event. The purpose of a menu is to give the user an easy and quick access to certain functionality. This functionality has to be provided by you, the developer, by writing a function that does what the user expects. The clicked event is there, to bind the functions you have written to the different (popup) items.
You connect the event with its handler by assigning a function pointer to the event, exactly as with any other event in dbase. In the initialize-method of the own_menu-class, write the following:

```
This.m_item1.p3.clicked = class::closeform
```

With the line above you just added an event handler to the menu's close button.
Now you have to write the eventhandler; because you used the keyword "CLASS" with the scope resolution operator :: , dbase awaits the function closeform in the same class definition. Place the cursor just before the endclass statement of the own_menu class and type the following:

```
Function closeform
  this.form.close( )
return
```

This piece of code will simply close the form, so that the close-menu-item will do what its name suggests.

- Note : *both the popupitem and the menuitem have a clicked-event, but only the eventhandler of the popupitem will be called automatically when you click on the corresponding item. If you click on a menuitem, it will only open its submenu but not run the function associated with the clicked event. This is done to keep the two objects as symmetric as possible and it gives you the possibility to call the event programmatically: this.m_item1.clicked( )*

2) The form property

The function that closes the form above, uses this line of code: this.form.close( )
Each menu-item has a property called "form" that is a direct reference to the form to which the menubar was attached. With this property, you can build a reference to every object on the particular form, like: this.form.entryfield1.copy( ) which will copy the content of entryfield1 to the clipboard.

Inside of the eventhandlers, you can't use "form" directly (form.close( ) will throw an error) . The items are subclassed from the dbase objectclass, which is a non-visual class and therefore doesn't know anything about a thing called "form". It is similar to the rowset-object, where you have to use this.parent.parent to get to the form. However, with the form-property of the items, you can skip the parents and simply write this.form

3) The enabled and the visible properties

These properties go together with the Set_Enabled and Set_Visible methods. You can read these properties i.e. ? this.enabled (will return *true* or *false*), but in order to change a value you have to use the appropriate method. To make sure they take effect, write the following line after the instantiation of the open-item :

```
This.m_item1.p2.set_enabled(false)
```

If you run your form you will see that the open-item is still there, but that you can't select it anymore. Use the same syntax with Set_Visible, to hide an item completely. This is particularly useful if you want to customize your menus depending on user rights. Just define the complete menu and hide those parts (with Set_Visible(*false*)) that don't fit with the currently logged-in user.

4) The text property

This property goes together with the Set_Text method. At any time you can easily change the caption of a particular item. In the next _menu class, write the following :

```
this.m_item1.set_text("&Data")
```

This code will change the caption of our first menuitem to : "Data"
The initial caption was given during instantiation, as second parameter of the new operator.

5) The type property

This property is readonly and indicates the type of the item :

Menuitem = 0
Popupitem = 1

6) The shortcut property

There are in fact two properties related to shortcuts: the shortcut property itself and the shortcut-modifier property. The former identifies a keystroke that will launch the clicked event of the corresponding item directly; the latter indicates whether a modifier-key (shift, control, alt) has to be pressed simultaneously.

The include file defines the following constants:

```
#define     FNONE     0
#define     FSHIFT    4
#define     FCTRL     8
#define     FALT      16
```

As with the other properties, we use the Set_Shortcut-method to actually change the property's value. The method takes two parameters, the first being the keycode for the shortcut-property and the second the value for the modifier. Keycodes are defined as the ASCII-Codes of the normal keys. For special keys you can find the complete list in the include file. For instance, the <END> key is defined as:

```
#define     VK_END    0x23
```

Add the following line to the next_menu class

```
this.m_item1.p3.Set_Shortcut(VK_END,FCTRL)
```

The shortcut is designed to display the localized abbreviations for the keys, so on English systems you may see: Ctrl + End , while on German systems you might see Strg + Ende .

The effect is the same on all operating systems, if you press the End-key together with the control key, the form will close.

7) The checked property

The property goes together with the Set_Checked method and is used to draw a checkmark in front of the corresponding item. Try the line below, just after the instantiation of the cancel-item in the next_menu class

```
This.m_item2.cancel.set_checked(true)
```

If you run the form, you will see a check-mark in front of the Cancel-item

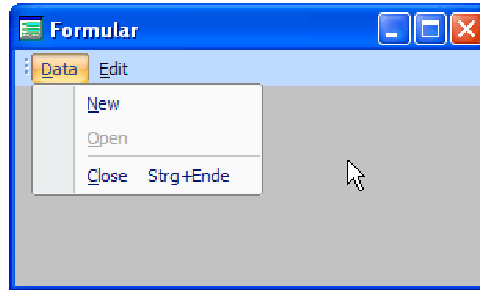
There are a few methods that aren't associated with properties. Let's have a look at them now.

8) The Set_Separator method

This method is used to separate an item from the rest of the items. A line will be drawn between the separated item and the rest. In the own_menu class, add the following:

```
this.m_item1.p3.set_separator( true )
```

Running the form, you will see something like:

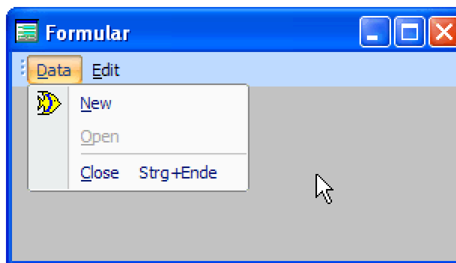


9) The Set_Icon method

As the name indicates, this method will attach an icon to a menu item.
You have to pass a valid filename as parameter. The filename can be a relative or an absolute path. The relative path starts from the location where your cc. File resides. In the executable it starts from the path where the exe resides. Generally, absolute paths are no good idea, so we concentrate on the relative paths.
If you write: Set_Icon(“fish.ico”) , the control will look in the folder, where the cc is located, to find a file named fish.ico. If it can’t find it, no icon will be displayed. It is a common habit, to group pictures and icons in a separate folder. You can access these files as follows: Set_icon(“ .\images\fish.ico”). The pathname follows the well known rules for relative paths: “.” is the current folder and “..” is the parent folder.
Let’s try this in our demo-class, by adding the following codeline:

```
This.m_item1.p1.set_icon("fish.ico")
```

If you made sure that the fish.ico was copied from the samples-folder to your own folder, you should see the following:

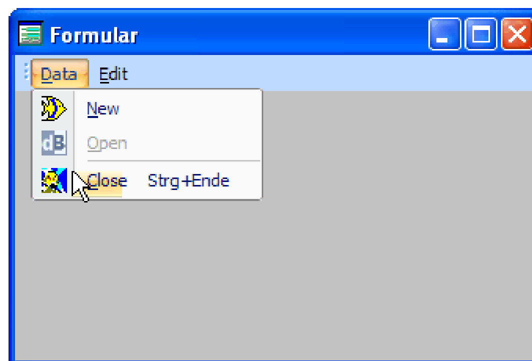


10) The Set_Resource method

If you have a lot of icons and/or bitmaps in your application, it is a good idea to group them in a resource dll. You don't have to copy dozens of small files during installation. The same if you want to change some of the icons with newer ones; you just replace them in the DLL file and send the (one) file to your customer. The Set_Resource method is used, to extract icons from these resource files.
It takes two parameters: the first is the filename of the Resource-DLL, the second is the name or ordinal of the icon (both passed as string-type).
The filename can contain the complete path to the resource, if not, the class will look in the actual directory then in the directory of the runtime and also in the Windows and Windows\system32 directory. If it can't find the file, the icon will not be displayed.

```
This.m_item1.p2.set_resource("plus_en.dll","31233")
This.m_item1.p3.set_resource("resource.dll","ROSWELL")
```

The first line will set the icon to the icon resource with ordinal number 31233 of the plus_en.dll in the dbase home directory (which is the dBi icon) the second line will use the name of an icon "ROSWELL". The result looking like :



11) The OnShowPopup event

Menu items (not popup items) have a special event called onshowpopup.
This event fires, every time you click on a menu item but before the popup menu is actually opened. That can be very handy, if you want to enable or disable an item depending on some general condition.
The standard example for the use of this event is the enabling/disabling of a copy-menu-item whether there is something selected or not.
The complete source for a standard "Edit"-Menu can be found in the demo cc that is part of the package

If you followed this tutorial till here, your own.cc file will look similar to the following, all those who don't like typing, can just copy and paste the below, to see the mv_menu in action.

```
#include mv_menu.h
Class own_menu(oparent) of mv_menu(oparent) Custom
   This.attach_to_framewin = true
   Function initialize
      This.m_item1    = new menuitem(this,"&File")
      This.m_item1.p1 = new popupitem(this.m_item1,"&New")
      This.m_item1.p2 = new popupitem(this.m_item1,"&Open")
      This.m_item1.p2.set_enabled(false)
      This.m_item1.p3 = new popupitem(this.m_item1,"&Close")
      This.m_item1.p3.clicked = class::closeform
      This.m_item1.p3.set_separator(true)
      This.m_item1.p1.set_icon("fish.ico")
   return

   Function closeform
      this.form.close()
   return
endclass

Class next_menu(oparent) of own_menu(oparent) custom
   Function initialize
      Super::initialize( )
      This.m_item2 = new menuitem(this,"&Edit")
      This.m_item2.m1 = new menuitem(this.m_item2,"&Action")
      This.m_item2.m1.p1 = new popupitem(This.m_item2.m1,"&Copy")
      This.m_item2.m1.p2 = new popupitem(This.m_item2.m1,"&Paste")
      This.m_item2.Cancel = new popupitem(This.m_item2,"Ca&ncel")
      This.m_item2.cancel.set_checked(true)
      This.set_theme(theme_ribbon)
      This.set_docking(true,dock_top + dock_bottom)
      This.m_item1.set_text("&Data")
      This.m_item1.p3.Set_Shortcut(VK_END,FCTRL)
      This.m_item1.p2.set_resource("plus_en.dll","31233")
      This.m_item1.p3.set_resource("resource.dll","ROSWELL")
   return
Endclass
```

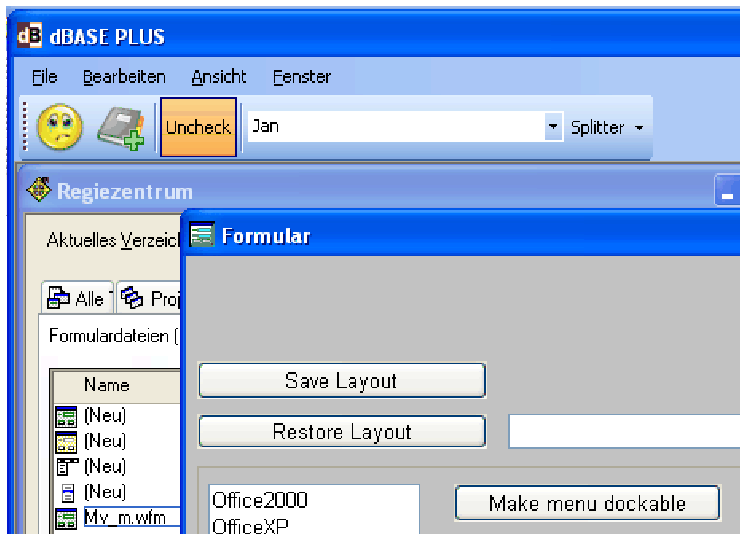
Lets now start the tour of the toolbar.

F) The mv_Toolbar

The mv_toolbar has a lot in common with the native dbase toolbar, however there are a few important ameliorations:

- first of all there is the more modern look and feel. The mv_toolbar will take the same visual styles as the mv_menu.
- You can assign icons with sizes other than the standard 16 x16 pixels.
- The Toolbar buttons can have text or icons or both.
- The mv_toolbar is not limited to tool buttons, it can contain mv_comboboxes and mv_splitbuttons.

It could look like the picture below…



The usage of the toolbar is similar to the usage of the mv_menu. You can add controls to it with the new operator.

```
This.t_bar = this.add_toolbar("File" , 0)
this.t_bar.t1  = new mv_toolbutton(this.t_bar,"FILE")
this.t_bar.cb1 = new mv_toolcombo(this.t_bar,"Combobox")
this.t_bar.s1  = new mv_splitbutton(this.t_bar,"Splitter")
```

The above code will add three different controls to the toolbar, a normal tool button, a combo box and a split button. A split button is a tool button, that has a drilldown button at its side. If the user clicks on the "button part" of the split button, the clicked event is fired. If the user clicks on the drilldown, a popup menu will be shown.

You can find all the members of the toolbar and its controls below in the summary.

G) Summary

Below you can find a short reference to all properties and methods of the mv_Menu class

Menubar :

- **Theme / Set_Theme** : gets/sets the visual style of the menu.
- **Set_Docking** : specifies the way, the menu can be docked to its parent form.
- **Version** : this property is not explained in detail, it just contains the actual version of the control.
- **Attach_to_framewin** : if you set this property to true, the menu will be bound to the _app.framewin (default = false)
- **Add_toolbar(ctitle,npos)** : adds a toolbar to the form, that holds the mv_menu.
- **Set_Visible** : shows/hides the menubar (default = true).
- **Get_State / Set_State(cstate)** : retrieves and restores the exact position of the menu and if present, the positon of the toolbars.
- **Set_IconSize(nsize)** : controls the size of the icons in the toolbars. This method acts globally.
- **Set_FontSize,Set_Fontname** : controls the font of the menu
- **Set_MdiBackColor**: -1 = automatic, -2 = dbase default, any other numeric value sets the background color.
- **OnUpdate** : Event, that fires every 100 ms, if the form is idle.
- **Get_MenuHeight** : returns the exact height of the menu in pixels.

Menuitem/Popupitem :

- **Clicked** : assign a dbase-function to this event. The function will be executed by clicking the corresponding popup item.
- **Form** : each menu- and popup item has a reference to its parent form.
- **Enabled / Set_Enabled** : gets/sets whether the item is enabled or not.
- **Visible / Set_Visible** : gets/sets whether the item is visible or not.
- **Text / Set_Text** : gets/sets the caption of the item.
- **Type** : readonly property that holds the type of the item (menu/popup)
- **Shortcut / Shortcut_Modifier / Set_Shortcut** : to handle the accelerator-keys.
- **Checked / Set_checked** : whether the item is checked or not.
- **Set_Separator** : whether the item has a separator line above itself or not.
- **Set_Icon** : load an icon from a file.
- **Set_Resource** : load an icon from a resource file by name or ordinal number.
- **OnShowPopup** : this event is fired, when the user clicks a menu item, that will open a popup, but before the popup is actually drawn.

Toolbar :

- **Set_Closeable** : controls, if the floating toolbar may be closed (default = false)
- **Set_Gripper** : controls, if the gripper (enabling the toolbar to be dragged around) is shown or not.
- **Set_Pos** : controls the position of the toolbar, 0 = top, 1= bottom, 2= left, 3 = right , 4 = float
- **Set_Visible** : show/hide the entire toolbar.

- **Set_Text** : this is a text to identify a toolbar, if you right click on a menu, the toolbars are listed by this text.

Toolbar Button:

- **Set_Width : (method)** goes together with the **width** property, it awaits a numeric parameter, that specifies the width in pixel of the control
- **Set_Height : (method) / height**-property, awaits a numeric parameter specifying the height in pixels of the control
- **Set_Tooltip : (method) / tooltip** property, awaits a string parameter. This string will be displayed as a tool tip, when the cursor stays over the control.
- **Set_Text : (method) / Text** property, awaits a string parameter, this string will be displayed in the tool button, when the style of the tool button is 1 or 3 (see set_style)
- **Set_Separator : (method)** true or false: indicates if the control will be preceded by a vertical line. Used to give a visible separation between parts of the toolbar.
- **Set_enabled : (method) / enabled** property (default = true). If you use set_enabled(false), the control will not react on user actions and will be displayed in a lighter colour.
- **Set_Checked : (method) / checked** property, (default=false). Gives a visual feedback, if a certain condition is true or false. If you use set_checked(true), the tool_button will remain in a signal-colour.
- **Set_Icon : (method)** , awaits a string parameter, containing the path and name of an icon file. This icon will be displayed in the tool button, if its style is 2 (see set_style). Many professional icons contain images in different sizes. The set_icon method will automatically choose the most appropriate size, according to the size, preset by the mv_menu-method Set_Iconsize. If no appropriate size is available, the method will expand or shrink the existing image (resulting in a loss of quality of the image).
- **Set_Resource : (method)** , identical to the set_icon method but awaiting two parameters, the first being the path + name of a resource file, the second being the name or index of the image in the resource file.
- **Set_Shortcut : (method) / shortcut and shortcut_modifier** property, awaits two parameters, the first being the ASCII value of the key to launch the clicked event of the tool button, the second being the modifier (SHIFT , ALT , CONTROL see explanation for the menu items).
- **Set_Style : (method) / style** property , controls the appearance of the tool button. If you pass the numeric value 1 as parameter, the tool button will contain only the text, assigned with the set_text method ; the value 2 means only the icon is shown and 3 shows the icon and the text side by side.
- **Clicked : (event)** , the control will call the function (event-handler) that was assigned to this member, each time the control is clicked. Codeblocks are not allowed.

ToolbarSplitbutton:

- The ToolbarSplitbutton has the same members as the Toolbar button, however you can assign a popup menu to it. This popup menu is the same as you would use with a menu item (including all its members). If you click on the split button, its clicked event-handler will be called, when you click on the little triangle at the side of the button, the assigned popup menu will be shown. Because the split button acts like a menu item, you can assign it a onshowpopup-eventhandler, that will be called just before the popup is actually drawn (see onshowpopup for menu items)

ToolbarCombobox:
The following methods/properties are the same as with the toolbar button : Set_Width, Set_Height, Set_Tooltip, Set_Separator, Set_Enabled and Set_Visible.

The members below are unique to the combo box.

- **Set_Datasource : (method)** this function takes an array as parameter. All the items in the array will be displayed in the combo box.
- **Set_canEdit : (method) / canedit** property takes a logical value to control, if the user can enter text in the edit-field of the combobox. If canedit is false (default), only the items present in the datasource-array can be selected in the combo box.
- **Get_Data : (method) / cursel** property. This function returns the text, that is actually visible in the edit field of the combo box.
- **Get_Index : (method) / index** property. The return value of that function is the index of the visible text in the datasource array. If canedit is true, get_index might return 0, if the visible text is not part of the original array.
- **Set_Index : (method)** if you want to preset a value in the combo box, you can call this method with a numeric parameter, representing the position of the text in the array.
- **Clicked : (event)** this is the same event as with every other mv_control (in this case, it can also be used as an ongotfocus-event)
- **LostFocus : (event)** this event fires, when the combo box loses focus.
- **OnKey : (event)** this event fires, each time a key is pressed in the edit field of the combo box.

ToolbarEditField:
The following methodsare the same as for the mv_toolbutton : Set_Width, Set_Height, Set_Tooltip, Set_Separator, Set_Enabled, Set_Visible

Special for the EditField

- **Set_Text: (Methode)** awaits a string that will be displayed in the editfield.
- **Get_Text: (Methode)** retrieves the actual text of the editfield.

You can see all these methods,properties and events in action, in the demo form (Mv_Demo / Mv_Demo_english), that is part of the mv_menu package. This demo form uses a generic class (startmenu.) , that contains the code for a complete “Edit”-Menu and a complete “Windows”- Menu along with a lot of comments on how to get the most out of the mv_menu.

Special thanks goes to Jan Hoelterling, Ivar B. Jessen and Roland Wingerter for their great help with this document!

The mv_Menu will be continuously enhanced in the future. Registered user will be informed per e-mail. If you have questions, you can contact me at : info @ vdblogic . de

Marc Van den Berghen

www.vdblogic.de/dbl