Clarion Magazine

This edition includes all articles, news items and blog posts from July 1 2011 to August 31 2011.

Clarion News

Read 7 Clarion news items.

Articles

Source Code Library Updated

July 4 2011

The 2011 Q2 Source Code Library update is now available.

ClarionMag's Summer Schedule

July 9 2011

Clarion Magazine is on its annual summer break; July and August will be a combined issue. The office will be closed from July 10-17 and 20-22, and email response may be delayed one or two days at other times. Thanks for your patience.

Tip of the Week: Make them longer, you'll be happier

August 16 2011

Dave Harms takes a swing at short, cryptic labels.

MDI Tab Control or Open Window Toolbar?

August 17 2011

When Nardus Swanevelder saw the MDI Tab Option in the application manifest settings, he knew this was something that he could use in his applications. But he wishes this feature had been called the "Open Window Toolbar."

Tip of the Week: Take care when refactoring variable names

August 24 2011

Making variable names longer and more descriptive can pay huge dividends, but be careful when renaming your variables. Here's how the compiler can help.

Clarion 8 Gold Released, CSPs Up For Renewal

August 29 2011

SoftVelocity has shipped the gold release of Clarion 8. If you have a CSP subscription you'll need to renew in order to continue to receive updates through to the gold release of Clarion 9.

A Windows 7 Alt-Lockup Fix For Clarion 6

August 30 2011

Clarion 7/8 users have a fix in hand for the Windows 7 Alt key lockup problem. But what do you do if you're still on Clarion 6? If you're Carl Barnes, you go into the heart of the Windows message system, you find the problem, and you fix it. Part 1 of 2.

A Fresh Look At ClarionTest

August 30 2011

It's almost two years since Dave Harms wrote ClarionTest 1.0, an NUnit-inspired testing framework for Clarion Win32 programming. And although ClarionTest has been shown on ClarionLive, it hasn't had much coverage here at home. Dave remedies that with the first article in a series on this increasingly popular utility.

A Windows 7 Alt-Lockup Fix For Clarion 6, Part 2

August 31 2011

Clarion 7/8 users have a fix in hand for the Windows 7 Alt key lockup problem. But what do you do if you're still on Clarion 6? If you're Carl Barnes, you go into the heart of the Windows message system, you find the problem, and you fix it. Part 2 of 2.

Using Clarion.NET's Easier Interop

August 31 2011

Clarion.NET now provides a much easier way to call .NET code from your Clarion Win32 applications. Graham Dawson provides an example, and then uses that example to show exactly how Clarion.NET does its magic.

Clarion 8's New Gradients

August 31 2011

SoftVelocity recently gave Clarion 8 the ability to display gradients for TOOLBAR, PANEL, BOX and ELLIPSE controls. Mike Hanson explains, and provides a utility for exploring this new feature.

Clarion News

DockingPane Wrapper Template 2.00

Version 2.00 of the DockingPane wrapper template is now available. This is a major release. It contains the following updates: Codejock v15.1.0 compatibility added; Multi Threading Enhanced; OCX Registration Enhanced; Support for Visual Theme Resource files added (Office 2007, 2010 + Win7); Support for Outlook 2007 Theme added; Support for Visual Studio 2005 Beta2 Theme added; Support for Visual Studio 2008 Theme added; Support for Visual Studio 2010 Theme added; Support for Visual Studio 6 Theme added; The control can now be added to the main AppFrame multiple times, and in multiple positions; Class modified to add Thin@ Support; Save / Restore layout facility added; Restore default layout facility added; Action events can now be blocked on an individual pane basis via the "Action" derived method. A new example has been added to the demo application; 2 New Methods added to help with other 3rd Party compatability; Pane Height and Width properties can now be optional variables instead of just constant values; Panes can now be defined in any order and still be attached and / or docked to another Pane; Trappable OCX Events can now be added and removed at runtime; Trappable Keystrokes can now be added and removed at runtime; Pane ID variable increased; Template / Class Optimized; New Option - "Keyboard Navigation"; New Option - "Display Pin Option on Floating Panes"; New Option - "Display Size Cursor While Dragging"; New Option - "Arrow Sticker Style"; New Method : 'EnableKeyboardNavigate'; New Method : 'Action'; New Method : 'AttachToWindow'; New Method : 'EnableKeyboardNavigate'; New Method : 'DockPane'; New Method : 'FloatPane'; New Method : 'GetPaneObj'; New Method : 'HidePane'; New Method : 'NormalizeSplitters'; New Method : 'RecalcLayout'; New Method : 'RestoreDefaultLayout'; New Method : 'RestoreLayout'; New Method : 'SaveLayout'; New Method : 'UpdatePanes'; Method Removed: 'AddCtrl'; Method Removed: 'GetCtrlCType'; Method Removed: 'GetCtrlObj'; Method Removed: 'GetCtrlType'; Method Removed: 'LoadState'; Method Removed: 'SaveState'; Method Removed: 'SetCtrlDescription'; Method Removed: 'SetCtrlProperty'; Method Removed: 'SetCtrlTooltip'. The update is free to all users who have an Active Maintenance Plan in place, or costs $47.50 for all users who have a Lapsed Maintenance Plan. It can be downloaded via the Products page within the members area of the web site.

Posted August 16 2011 (permanent link)

HTML Editor Wrapper Template 1.02

Version 1.02 of the HTML Editor wrapper template is now available. This is a major release. It contains the following updates: C8 compatibility added; Multi Threading Enhanced; OCX Registration Enhanced; Template / Class Optimized; The control can now be added to the main AppFrame multiple times, and in multiple positions; Class modified to add Thin@ Support; 2 New Methods added to help with other 3rd Party compatibility; New method "GetProperty'; New method "SendCR'; New method "SelectAll'; New method "SelectText'; New method "SetProperty'; BUG FIX: Keyboard Shortcuts now working correctly; BUG FIX: Some toolbar buttons could still be displayed even though the Toolbar

was set to "Don't Display"; BUG FIX: Merge Process Extension not working correctly in Legacy apps. The update is free to all users who have an Active Maintenance Plan in place, or costs $47.50 for all users who have a Lapsed Maintenance Plan. It can be downloaded via the Products page within the members area of the web site.

Posted August 16 2011 (permanent link)

MustBeInFileOrNull Template Update

A new version of the MustBeInFileOrNull template is available. This release contains a fix for compatibility with Clarion 8.

Posted August 16 2011 (permanent link)

DMC 2.4.2.1469

DMC version 2.4.2.1469 is now available. This release is available, free of charge, to all DMC customers who have an active DMC maintenance and support plan subscription (and to all evaluation level users). You can update to this new version from within DMC or you can use the full installer. Changes include: added code so as have the system tray icon reflect what is happening during a txa parsing (dmc being hidden); added code to protect values within quotes during txa parsing of a port to sql task; added a new color definition in the themes to allow you to define what color you want your buttons (background) to be; added new feature during port to sql : during the first dctx reading phase which displays the tobe sql dct - now you will have a report list of any duplicate key names found in your dct; added a new table level option token - dmclevel - during a port to sql define the order in which the tables will be treated and mainly the data transfer profiles order (Parent-Child); Added in CSV export a stop or continue message; Corrected the BLOB Temp Table in IMDD support to work properly (with a HUGE improvement in time taken to process Tables with Blobs); Added support for a concat detail during data transfers the necessary code to accept a value of SP to be used as a separator for a SPACE; Changed SQL reserved words browse - form to open up a form and not EIP and added a filter on this browse to display - ALL - level 1 or level 2; Changed the Port to SQL default project NAME to also reflect the full path name of the DCTX used; Removed from "select task" the manage your profiles as that was redundant with the navigation panel link; Changed the override defaults on Date and Time detections during a clone to SQL task to display icons showing what will happen; Corrected the code used to LOCATE a record in Table Lists (main wizard); Corrected a bug when in Viewer mode and the table contained less than 50 records - the Viewer would report "all records read"; Corrected a bug in ODBC driver where a string(2000) in PostGreSQL was seen as a memo (a TEXT in sql language); Corrected a bug in ODBC driver which sees a PostGreSQL ByteA blob as a string; Corrected a bug during SQL normalizations on "Next" and "Previous" buttons; Corrected a bug in Port to SQL on data type when using MySQL which could break the dctx generated; Corrected a bug in CSV export for the progress bar to display properly; Corrected a bug in SQL DB Backup edit and restore scripts; Corrected a bug during Clone to SQL when a BLOB was not placed at end of table structure in SQL; Corrected a bug during Port to SQL on a DCT without any relations - then columns were badly renamed and ended up with empty screens; Corrected a bug during Port to SQL on MySQL PRIMARY keys; Corrected a bug during Clone to ASCII from a TPS; Corrected a bug during SQL to TPS cloning on table names (tps side would be "dbo_customers" - now it is simply "customers"); Corrected a bug so as to display the progress bar during data transfers when DMC reads the SOURCE side data after clicking on the Transfer button; Corrected a bug in displaying properly lifetime versions when using american dates and when US Dates were used (on expiry dates); Corrected a bug in ODBC connections on the test server; Corrected a bug when PostGreSQL was used with Binary Blobs;

Posted August 16 2011 (permanent link)

New ClarionAddins web site

Brahn Partridge has created a new web site just for Clarion addins. Currently listed addins on the site are: CancelBuildButtons; ClarionEditorContextHelp (Free!); ClarionMagSearch (Free!); ClearErrors; EditorMacros; InsertClarionColor; KeyboardShortcuts (Coming Soon); MainToolbarExtras; OptionsAutoSave (Free!); ProjectBrowserExtras; PropertyGridExtras; RemoveLine; SearchPadExtras; SetTheme; SolutionIcons; StartPageV2 (Coming Soon). Releases and updates announced via twitter: https://twitter.com/clarionaddins.

Posted August 16 2011 (permanent link)

1st Logo Design Back-to-School Sale

All 1st Logo Design Icon Collections are up to 80% off.

Posted August 16 2011 (permanent link)

ClarionLive Weekly Webinar for August 19, 2011: ProScan & ProImage

This week Charles Edmonds is going to talk about the features and benefits of using ProScan and ProImage to enhance your applications. He will also talk about why you would want to use these tools instead of just "rolling your own" imaging solution and what this means to the average Clarion programmer. There will also be a "first look" at a new Clarion template that is being released and some special discount prices for ClarionLive attendees.

Posted August 18 2011 (permanent link)

The ClarionMag Blog

Tip of the Week: Make them longer, you'll be happier

By Dave Harms

Posted August 16 2011

Right, let's get one thing out of the way at the start. This isn't a tip. It's a rant. My tip is, don't do this and I won't rant about it.

I'm not blaming you. I've written this kind of code too. We all have, probably. Well, all except for Andrew Guidroz II. But it's got to stop.

I'm talking about those teeny tiny ultra-short variable names, like `RQ:B`, and `tlRVn`, and other labels too small to see without the aid of Coke-bottle lenses.

Someday someone's going to need to read your code, and they're not going to have a clue what you were on about. It might even be you, and you won't have a clue what you were on about either. Just. Say. No. You should be able to read code. With labels like `RQ:B` you don't need to read code, you need to read minds.

Use descriptive labels and procedure/method/class names. Make 'em every bit as long as necessary. Don't fret over the typing, just use code completion: it's not perfect, but it's way better than unreadable code.

While I'm ranting, I think it's time to give those old style naming conventions like Hungarian notation the boot. I used to be a big fan of type notations. And in combination with long labels they're probably not so bad. But I don't bother wtih them any more. I can always hover my mouse over a variable if I want to know its data type. I'd rather focus on making code readable, and those funky prefixes just get in the way.

Take the time to give meaningful labels to things; it may be a little extra work (and some additional typing) now but it will pay off down the road.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

by Austin Drum on August 17 2011 (comment link)

RefL:Amen + llop:To + segyq:That + lkkk:Brother

by Russell Eggen on August 17 2011 (comment link)

I could not agree more. The only concession I make to Hungarian notation is my prefixes tend to indicate where one would find the original declaration, for example GLO:SomeVariable is in my dictionary, whereas AGLO:SomeVariable is found in this application file, global scope. Yeah, I know, that is a stretch to call

that a concession .

Class properties are easily identified, so I don't really do anything special for them.

On naming controls, I get very explicit. I can't tell you how annoying it is to find ?Insert:2 {PROP:Disable} = True inside embed code. I always rename controls from their default: ? InsertLineButton. Or ?TAB:4 to ?JobsTab and so forth.

All ABC class instances I rename too. From BRW1 to CustListMgr. All my classes have "Mgr" in the label. Nothing else else does. ?Browse:1 to ?CustList; Browse:1:Queue to CustListQ and so on.

When I turn over such code to anotehr developer or a client, I rarely get questions as to what I did. This is how you write self-documenting code. And I do a lot of commenting too! ;-)

---

by Dave Harms on August 17 2011 (comment link)

Austin - heh heh

Russ - yeah, those numbered equates can be especially bad news.

---

by Graham Dawson on August 18 2011 (comment link)

Hi Dave,

Absolutely agree.

Can I add another rant: this time against 'magic numbers' in parameters. We have these all over the place and they drive me to distraction!

You know the kind of thing:-

TenancyProcedure(1)

TenancyProcedure(2)

Just what does the procedure do?

Either use EQUATEs (which I agree can be difficult to manage) or pass strings.

TenancyProcedure('COMMENCE')

TenancyProcedure('TERMINATE')

it's self explanatory now.

Graham

---

by Dave Harms on August 18 2011 (comment link)

Graham, I'm a big fan of equates for both numeric and string constants.

Dave

---

by surfersteve on August 24 2011 (comment link)

dumped naming convention years ago. so these days i dont even bother with camel case in csharp either. big long words describe by procedures.

for example if i create a word for use in expert templates. i will say something like, texttargetnewparts()

this uses the current textartget class in focus and create a new object for creating text line parts

i like simple statements and words, after all i have to read them as a human. i dont read hex and i hate computer languages that are not simple to read

BACK TO TOP

MDI Tab Control or Open Window Toolbar?

By Nardus Swanevelder

Posted August 17 2011

Recently I looked at the Application Manifest settings for one of my APPs, and I saw the Extended UI button. I clicked on it and decided to read the Help to understand what is happening on this screen. Yes I do sometimes read the help…

When I saw the MDI Tab Option I knew that this was something that I might be able to use in my applications. MDI Tab in my opinion is not the correct name for this function. I would call it something like the Open Window Toolbar.

In this article I'll show you how to use this option, and how to make it customizable by your users.

What is the Open Window Toolbar?

The Open Window Toolbar is a special toolbar, introduced in Clarion 7.2, that keeps track of all active MDI windows in a tabbed format (specifically, all threads launching MDI windows). Figure 1 is an example from one of my applications.

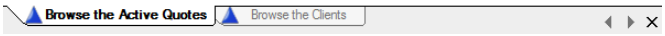
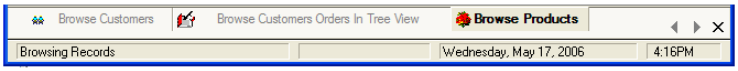


Figure 1. Example of a Tab Toolbar

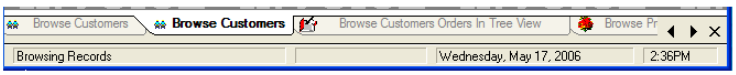
Standard features of the Open Window Toolbar are:

- You can position your Toolbar at the Top or Bottom of the Application's Frame.
- This feature offers you different Styles for the Tabs on the Toolbar. Below is an example of the different styles:

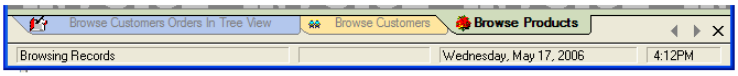
Default uses the current tab style set by the XP operating system. For example:



B&W displays a black and white tab style:



Colored displays colored tabs using a set of predefined colors defined in the target window.



Boxed displays a boxed type of tab style:

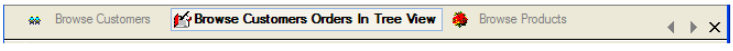


Figure 2. Sample Styles for the Tabs

Enabling the toolbar

I more or less showed how to enable the toolbar in the opening paragraph, but I'll go through it in more detail. The first step is to click on the Global Properties Tab in your app file where the Application's Frame is situated. This is normally the app file that is your exe.

The second step is to click on the Actions button that opens the Global Properties screen. On this screen click on the App Settings Tab and then click on the Extended UI button. Figure 3 shows the available Options. Under the MDI Tab section you will see two drop down boxes. MDI Tab has three options namely Top, Bottom and Disable. As you'd expect, these indicate if the Toolbar opens up at the Top of the screen, bottom of the screen or if it is disabled. The MDI Tab Style drop down gives you four options and if you look at Figure 2 you can see what the different settings will do to the Toolbar.

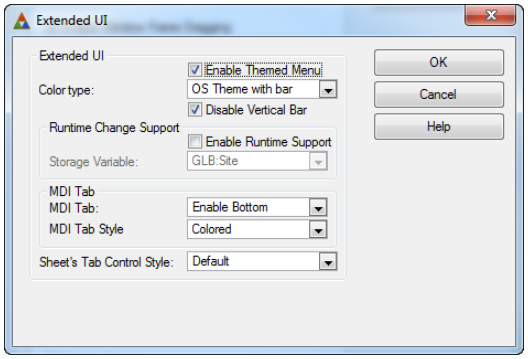


Figure 3. Extended UI Settings

What code is added to my application?

To see the code that is added to your application, go to the procedure that contains the Frame. Find the `ThisWindow.Init` Embed and go to Priority 9300.

```
COMPILE ('**CW7**',_VER_C70)
    AppFrame{PROP:TabBarLocation} = MDITabLocation:Bottom
    AppFrame{PROP:TabBarStyle} = TabStyle:BlackAndWhite
    AppFrame{PROP:TabBarVisible}  = True
    MenuStyleMgr.Init(?Menubar)
    MenuStyleMgr.SuspendRefresh()
    MenuStyleMgr.SetThemeColors('AUTO')
    MenuStyleMgr.SetImageBar(False)
    MenuStyleMgr.ApplyTheme()
    MenuStyleMgr.Refresh(TRUE)
!**CW7**
```

Figure 4. Listing of Toolbar code

The magic happens in the first three lines of the code as seen in Figure 4. Line one assigns an equate that specifies the Location of the Toolbar, line two specifies which style the application is going to use, and line three enables or disables the Toolbar.

That is it! That is all you need to do (or, more accurately, the templates do) to enable this Open Window Toolbar. Or is it?

How do I change this at runtime?

It is very easy to add the Open Window Toolbar to your application, but what if the user doesn't want it at the top or the user doesn't like the colored tabs?

Luckily it is not difficult to add the option to change the settings at runtime. Here are the steps you need

to take.

1. Disable the MDI Tab code

Disable the MDI Tab code in the Extended UI template. Yes that is correct!

2. Call a local method

In the Main procedure (which contains the application Frame) at the `ThisWindow.Init` Embed at Priority 9700 add the following code:

```
MyLocal.ApplyTabbar()
```

(If you want to understand how a Local Class work please see my previous article on Local Classes.)

This code calls a method in a local class, and I will look at the code that goes into the method in Write the method code. Make sure that the code goes into the embed point with a priority of 9700 and not 9300. The method has to be called after Clarion's code as shown in Figure 4. (You could also use a Local Routine to achieve the same functionality)

3. Create the menu options

Create two new Menu options under the Window Menu. I have named the first new Menu option "Tab Toolbar Settings" and the second "Apply Tab Toolbar Settings". On the first Menu option call a Form procedure and name the procedure "TabBarSettings". This procedure will be used to give the end-user a method of selecting and saving the settings that they might want to change. On the second menu option open an embed point and add the same code as per Step 2.

4. Create two local variables

Fourth, add two local variables of type Byte to the procedure. I have named mine `LOC:InvalidFlag` and `LOC:TabBarStyle`. The first local variable will be used to determine if the system should enable the Toolbar or not and the second variable is used to determine which Style the user has selected.

5. Write the method code

I have added the following code into the method that I call:

```
MyLocal.ApplyTabBar     Procedure()
   Code

   If IniMgr.Fetch('TabBar','Visible') = 'True'
      LOC:InvalidFlag = False

      !TabBar Location
      !0 = Top
      !1 = Bottom
      If IniMgr.Fetch('TabBar','Location') <> ''
         AppFrame{PROP:TabBarLocation} = IniMgr.Fetch('TabBar','Location')
      Else
         LOC:InvalidFlag = True
      End

      !TabBar Style
      !0 = default
```

```
      !1 = Black and white
      !2 = colored
      !3 = squared
      !4 = boxed
      LOC:TabBarStyle = IniMgr.Fetch('TabBar','Style')
      If LOC:TabBarStyle < 5
         AppFrame{PROP:TabBarStyle} = LOC:TabBarStyle
      Else
         LOC:InvalidFlag = True
      End

      !TabBar Visible
      If LOC:InvalidFlag = False
         AppFrame{PROP:TabBarVisible} = True
      Else
         AppFrame{PROP:TabBarVisible} = False
      End

      !!Add this code if you want to override the default colors for the tabs
      !If LOC:TabBarStyle = 0 or LOC:TabBarStyle = 2

      !   !Color - Set background color of the Tab in the Toolbar
      !   AppFrame{PROP:TabBarColor,1}  = Color:Blue

      !   !Color - Set your own colors for the tabs on the toolbar
      !   AppFrame{PROP:TabBarColor,2}  = Color:Green        !Tab1
      !   AppFrame{PROP:TabBarColor,3}  = Color:Red          !Tab2
      !   AppFrame{PROP:TabBarColor,5}  = Color:Aqua         !Tab4
      !   AppFrame{PROP:TabBarColor,6}  = Color:Fuchsia      !Tab5
      !   AppFrame{PROP:TabBarColor,7}  = Color:Teal         !Tab6
      !   AppFrame{PROP:TabBarColor,8}  = Color:Purple       !Tab7
      !Else
      !End

   Else
      !Disbale
      AppFrame{PROP:TabBarVisible} = False
   End
```

From the above code it is evident that there are four properties that can be used to activate the Toolbar.

Properties

Here's an overview of the different properties I'm using.

PROP:TabBarLocation

This specifies the location of the Toolbar and the following options are available

Setting	Value	Equate

Top	0	MDITabLocation:Top
Bottom	1	MDITabLocation:Bottom

PROP:TabBarStyle

This property specifies the visual style of the Tabs on the Toolbar. You can use any one option:

Please see Figure 2 for a graphic view of the different styles.

Setting	Value	Equate
Default	0	TabStyle:Default
“Black and White” in Office 2003 Style	1	TabStyle:BlackAndWhite
Colored in Office 2003 Style	2	TabStyle:Colored
Squared tab	3	TabStyle: Squared
Boxed tab	4	TabStyle: Boxed

PROP:TabBarVisible

This enables or disables the Toolbar or in other words Hide or Show the Toolbar

Setting	Value	Equate
Hide	0	False
Show	1	True

PROP:TabColor

This property has some default values but it is possible to override them with user specific values if the defaults are not acceptable.

An example of the default colors looks like this:

Figure 5. Default colors for Tabs on Toolbar

To change the default colors it is important to understand that the following code needs to be called after the Toolbar was made visible, and it will only have an effect if the `TabBarStyle` is set to Default or Colored. The Index number of 1 is used to set the background color of the Tabs in the toolbar and index number 2 – 8 refers to Tabs 1 to 7. If a Tab 8 is created it will use the same color as Tab 1 and so forth.

```
!This is the background color of the Tabs
AppFrame{PROP:TabBarColor,1} = Color:Blue
!This is the First Tab’s color
AppFrame{PROP:TabBarColor,2} = Color:Green
AppFrame{PROP:TabBarColor,3} = Color:Red
AppFrame{PROP:TabBarColor,4} = Color:Yellow
AppFrame{PROP:TabBarColor,5} = Color:Aqua
AppFrame{PROP:TabBarColor,6} = Color:Fuchsia
AppFrame{PROP:TabBarColor,7} = Color:Teal
AppFrame{PROP:TabBarColor,8} = Color:Purple
```

If the code above is executed the tab bar will be displayed as follow:

Figure 6. Changed colors for Tabs on Toolbar

The configuration procedure

Earlier you created a stub for the `TabBarSettings` procedure; it's now time to flesh that procedure out. Create a screen similar to this or import the screen from the code attached to this article.

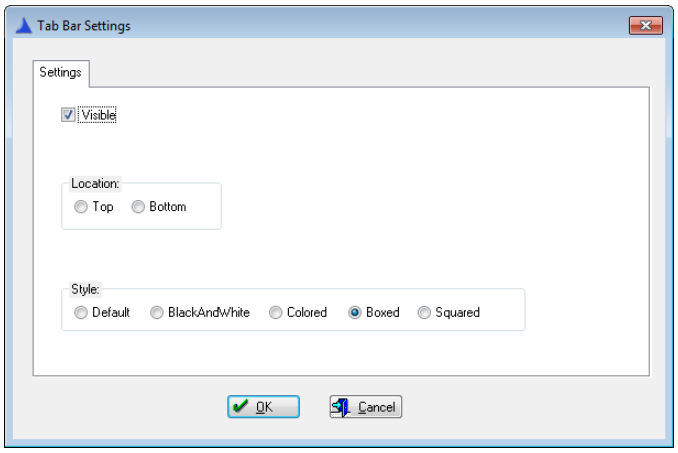


Figure 7. Tab Bar Settings screen

Add the following code to the `ThisWindow.Init` Embed at priority 8001

```
LOC:TabBarVisibleTemp = IniMgr.Fetch('TabBar','Visible')
If LOC:TabBarVisibleTemp = 'True'
   LOC:TabBarVisible = True
Else
   LOC:TabBarVisible = False
End
LOC:TabBarLocation = IniMgr.Fetch('TabBar','Location')
LOC:TabBarStyle    = IniMgr.Fetch('TabBar','Style')
Also add the following code to the ThisWondow.TakeCompleted Embed at priority 4500
If LOC:TabBarVisible = True
   LOC:TabBarVisibleTemp = 'True'
Else
   LOC:TabBarVisibleTemp = 'False'
End
IniMgr.Update('TabBar','Visible',LOC:TabBarVisibleTemp)
INIMgr.Update('TabBar','Location',LOC:TabBarLocation)
IniMgr.Update('TabBar','Style',LOC:TabBarStyle)
Post(EVENT:CloseWindow)
```

INI file options

Click on the Global Properties Tab in your app file where the Application's Frame is situated. Next click on the Actions button that opens the Global Properties screen. Now click on the INI File Options Button. The default setting for the INI File to use is: "Program Name.INI" and the default for Location of Program Name. Ini is App Directory. If the above settings have to be user specific the setting needs to be changed to a CSIDL Folder. I have chosen `CSIDL_LOCAL_APPDATA` (non-roaming).

Below is an example of what the screen might look like:

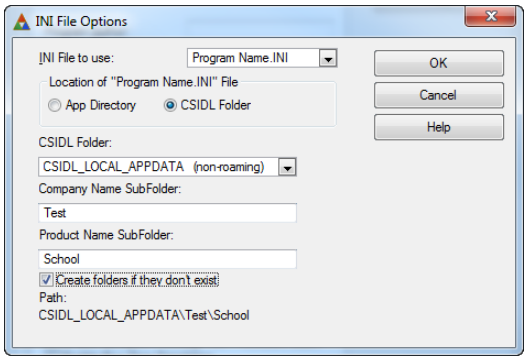


Figure 8. INI File Settings

If you need to understand a bit more on the meaning of the different CSIDL locations go to this website and scroll down to: Where to put data under Vista?

You should now be able to run the application and change the MDI tab options on the fly.

Summary

Following the easy steps explained in this article is all you have to do to enable the Show Window Toolbar in your applications while also giving the end-user the option to change the settings for the Toolbar at runtime.

Download the source

Nardus Swanevelder was born and raised in South Africa. He was a networking engineer for seven years before he moved over to the commercial side of the business. Nardus has developed a Sale Cycle Management system for the Information and Communication Technology industry. He has been programming in Clarion since 1989, and holds B.Com and MBA degrees. In his spare time Nardus lectures Financial Management to B. Com Hons students at North-West University.

Article comments

---

by P.G.M. Bouma on August 17 2011 (comment link)

Nice ! Are those nice tabs also possible a browse with tabs on it ?

---

by douglas johnson on August 19 2011 (comment link)

Any font, font size, settings? As best I can tell, it is not determined by the frame.

---

by Nardus Swanevelder on August 22 2011 (comment link)

I don't see any SV template that will change the Tabs on a Browse for you, the only solution I can see is to change it manually unless you purchase some of the third party tools to help you with that. CapeSoft's MakeOver might help with that.

---

by Nardus Swanevelder on August 22 2011 (comment link)

I looked at various ways to set the font of the TabBar but it seems SV will have to give us a way to do that, unless someone else knows of a clever way to do it.

If you run through the controls on the Window you don't get a PROP:Type for the TabBar. You do get Prop:Type = 22 which is Create:MenuBar and Prop:Type = 128 which is Create:Toolbar. This Create:Toolbar is however not the TabBar.

FindTabBar = 0 Loop FindTabBar = AppFrame{prop:nextfield,FindTabBar} If FindTabBar = 0 then break. Stop(FindTabBar{prop:type}) End

The code above is what I used to try and retrieve the TabBar's equate.

I thought that if I could get the equate for the TabBar maybe something like SetFont()could work, but as I was unable to find the equate I could not test setfont().

Sorry I could not be of more assistance.

BACK TO TOP

Tip of the Week: Take care when refactoring variable names

By Dave Harms

Posted August 24 2011

In the previous tip I ranted about cryptic and too-short variable names. I've been spending a lot of time in code like that, and I've done a lot of renaming of variables to try to clean things up. Some of these variables are generated by templates, so I've been making template changes as well.

If you find yourself needing to make a whole lot of changes, you'll make your life easier if you don't change too many variables at once. If you can afford the time, just change one variable name at a time. You may want to do a search and replace to change all usages of that variable before compiling, or you can just change the variable name in the declaration and let the compiler find the instances for you.

There's one important advantage to doing variable name changes this way. If you end up changing the variable name to something that already exists in the same scope, you'll get a compiler warning. If you made a whole bunch of changes before you compiled, and especially if you made those changes in embedded code, you may not be able to back the changes out. Now you don't know which uses of the variable are the renamed variable and which are the original variable.

You could also search through all of your application's code to find potential conflicts; if you take that approach, remember to also search through any source files that your source may include. But I generally let the compiler tell me if I have a problem.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

BACK TO TOP

A Windows 7 Alt-Lockup Fix For Clarion 6

By Carl Barnes

Posted August 30 2011

As Windows 7 became popular reports started coming in to our office of users experiencing the application locking up (i.e. freezing, hanging, etc) while they were performing normal data entry. The last thing many users recalled doing was pressing an Alt+ key accelerator, but they could not reproduce the lockup. Eventually a user remembered exactly what they had done, pressed the Alt key then released it without pressing an accelerator key. So we had the exact steps to reproduce the lockup: press and release Alt.

Initially the only option was shutdown the app via Task Manager. The task status didn't show as "not responding", and the CPU was not pegged at 100%; this was unusual. Visually the frozen app still displayed fine. The File menu was highlighted or latched (an important clue), but keyboard and mouse input did not work.

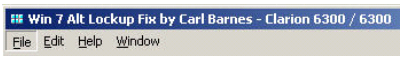


Figure 1. The locked up menu

Eventually someone found the app could be unlocked by pressing Alt+Esc. So at least tech support could help a customer that called, but I suspect many did not call and were frustrated. Unfortunately the tech support manager decided Alt+Esc was an acceptable solution and meant we were not wasting development time on the problem, so I didn't find out about it until recently. Educating customers on the unlock keystroke was not, however, an acceptable solution for a commercial product, and we were not prepared to move to a more recent version of the Clarion runtime. We had to come up with an alternative.

In this article I'll give you an easy fix to trap the Windows API message that causes the problem and prevent the lockup from occurring, plus what I find to be an improvement in menu behavior.

Standard Windows Behavior

The standard Windows behavior on pressing the Alt key and releasing it is to activate the menu bar. The first menu is highlighted (normally the file menu). This menu takes focus, but does not drop down. That is visually what is seen in a locked up Clarion app. In newer Microsoft applications (e.g. Internet Explorer 7, 8 and 9) the menu bar is completely hidden until the Alt key is pressed and released. On Apple Macs running Windows the Option key functions as the Alt key.

The F10 key has the same standard behavior to select the menu bar and will also lock up a Clarion app that is susceptible to the Alt key problem. Also every combination of Shift, Ctrl or Alt with F10 selects the menu bar, and locks up. Of course if a child window has altered F10, or used it as a `KEY()`, the

frame menu will not be activated.

In my testing I found the lockup only occurred when the Clarion app has an MDI frame and a child window open. SDI windows are unaffected. The problem also does not happen on all Windows 7 systems. Newsgroup posts (links at end) by some testers found it required Outlook, Live Mail or Messenger be installed, and some said one of those had to be running. Other testers reported the lockup on clean installs without those programs installed.

The details don't matter, the problem occurs on many systems so it must be fixed. Almost all of our in house systems had the problem so I was able to test and verify the lockup occurs in apps created with Clarion 5, 5.5 and 6.

SoftVelocity reported fixing the problem for Clarion 7.1. A February 18 blog post titled "Update for 7.1" stated:

> Latest update for 7.1 has just been made available. This build works around a problem Clarion developers found on Windows 7 when the Alt Key was pressed. The problem reported was an apparent conflict with Windows 7 "Live Mail" or Outlook 2007. Both of these apps default to hiding the main menu and activating it upon pressing the Alt Key. The problem wasn't unique to Clarion 7, the same conflict happens under Windows 7 with version 6 Clarion apps (and most likely prior versions too). We are testing a backport of the workaround for C6.3.x.

A fix for C6.3 has not been released by SV, but below you can download my fix. If you are using 7.1, or later, you do not need the fix, but you may find the altered menu behavior more appealing.

The template fix option

A template fix was published in the news groups by Marius van den Berg that does prevent the lockup. The template works by simply altering the Alt key (`ALERT(1024)`) and F10 key in each MDI child window so the keystrokes never get to the frame and lock it up. In my opinion the template solution has several negatives that make it undesirable for a commercial or sizable project:

1. The global template must be populated into every App. It adds code to every window generated by template: window, browse, form, etc. I like "squeezing the last drop" and would prefer the least code possible; I'm not happy at the thought of code in every window.

2. Only template generated Windows are fixed, so the following types of windows are typically not fixed: hand coded, source procedure, ABC class, legacy standard, and third party. Also windows with merged MDI menus do not get fixed.

3. The menu bar becomes harder to use. The Alt key and F10 key will not activate it. Windows 7 does not display the menu accelerator key underlines (e.g. File Edit..) until it "sees" the Alt key is pressed. The template prevents the Alt key down message from getting to the frame so the underlines never display, the user will not know what Alt+ letter to press to drop a menu.

4. Alerting F10 interferes with child window existing code that uses F10 as an Alert or Key. Some additional template programming could probably fix that problem.

5. C5, and earlier, do not get fixed due to `ALERT(1024)` not working.

An API fix in the frame

The fix I desired was to handle it all in one place, not every window. I guessed that there were probably Windows messages going to the frame that I could intercept to prevent the problem. My research turned up that when the menu is being activated by the Alt key or F10 key a `WM_SYSCOMMAND` message is sent to the frame with parameters of `SC_KEYMENU` and zero. With about 20 lines of code I could intercept the message and prevent the lockup. Actually the solution only takes about five lines of code; the rest of the code is the overhead required to get the message before the RTL.

The Clarion Accept / Event paradigm is a layer on top of the Windows message system so you'll never see true Windows messages unless you "subclass" the Frame window.

In Part 2 I'll explain how to intercept Windows messages and strip out the one message that causes the lockup.

Carl Barnes is an independent consultant working in the Chicago area. He has been using Clarion since 1990, is a member of Team TopSpeed and a TopSpeed Certified Support Professional. He is the author of a number of Clarion utilities including CW Assistant, The CHM help class CHM4Clarion, and Clarion Source Search.

Article comments

BACK TO TOP

A Fresh Look At ClarionTest

By Dave Harms

Posted August 30 2011

If you've participated in Arnold Young and John Hickey's excellent ClarionLive webinars, you've probably seen John or me or more recently Mike Hanson demonstrating a utility called ClarionTest. It is, as you'd expect, a tool that somehow helps you test your Clarion code.

But what does testing code mean, really? All too often in the Clarion world, testing means sitting down to a running program, opening windows, clicking on buttons and typing text, and observing what happens. It is of course very important to verify that a program behaves the way the user expects it to behave. It's just enormously difficult and time consuming to test every aspect of a program's functionality this way. You have to remember to try all the different menu options, you have to enter both valid and invalid data and check for the results, and when something goes wrong you have to be able to repeat the test conditions so as to verify the bug is fixed.

Testing an application with the keyboard and mouse is necessary, but not sufficient.

But it gets worse. Verifying your program's behavior by pretending to be a user is rather like building a car out of a bunch of untested parts, and then testing the car by getting behind the wheel and going for a drive. Imagine if the engine were a brand new design being run for the very first time, or no one had ever verified that the transmission would actually shift as expected, or the braking system used a previously unknown technique to bring the vehicle to a stop.

It's likely that some vehicles were tested in just this way in the very early days of the automotive industry (and quite probably a few test drivers were seriously injured if not killed). But nobody does that anymore. The individual components that make up the vehicle are tested, stressed and evaluated in isolation long before they're assembled into larger systems, and eventually into an entire vehicle. As a result, cars and trucks are far more reliable and durable than they were fifty or even twenty years ago.

Software development has evolved along similar lines, in part because of the widespread acceptance of object-oriented programming. The behavior of individual objects can be verified using automated testing techniques; applications that are composed of these objects are much less likely to have bugs because the individual components have been thoroughly tested.

And that's where ClarionTest comes in (although it isn't limited to just object-oriented code). ClarionTest isn't about taking your app for a drive. It's about making sure the chunks of code that make up your app are working as they should.

Of course in Clarion there's a caveat to that statement. Because of the AppGen, much of the code in our apps is tried and true. So I'm not talking about generated code, I'm talking about the code that you add to your app. That's both the code that gives the greatest value to your app (because it makes the app do all those non-standard things your clients value) and it's the code that's the most prone to failure, simply because it's usually new and because nobody writes perfect code all the time.

In this article I'll introduce the concepts behind unit testing with ClarionTest. In Part 2 I'll walk through a unit testing example, and I'll go into ClarionTest's internal workings in more detail.

About unit testing

I wrote the original ClarionTest almost two years ago now. At the time I was writing quite a lot of C# code and I began using NUnit, a popular unit testing framework for .NET. Wikipedia defines unit testing, in part, as

> a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. In object-oriented programming a unit is usually an interface, such as a class. Unit tests are created by programmers or occasionally by white box testers during the development process.
>
> Ideally, each test case is independent from the others: substitutes like method stubs, mock objects,[1] fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and

behaves as intended

C# is an object-oriented language, so I use NUnit to verify the behavior of my C# classes. Actually I also used NUnit to help me design my classes, but test-driven development is another topic entirely....

In C# I typically write one test class for every "real" class. For instance, the web application that I wrote to deliver ClarionMag has a store, and one of the classes that makes up the store is called `ShoppingCart`. To make sure that the shopping cart worked the way I expected it to work, I created a class which I called `ShoppingCartTests`. Here's a sample test method from that C# class:

```
[Test]
public void AddOneProduct_WithSingleQuantityAndDiscountMaxValue_VerifyTotal()
{
    ShoppingCart cart = new ShoppingCart();
    Product p = CreateProduct("SUB12", 159);
    Coupon coupon = CreateCouponForProduct(Coupon.Type.MaximumProductValue, 123.45m,"SUB12",t
    cart.Add(coupon);
    cart.Add(new CartItem(p));
    Assert.That(cart.GetSavings(), Is.EqualTo(35.55m), "Wrong savings amount");
    Assert.That(cart.GetTotal(), Is.EqualTo(123.45m), "Wrong total");
}
```

To run the test I simply compile the code and then invoke NUnit, usually via the Visual Studio IDE. Or, if NUnit is already running, I just click on a button in the NUnit application to rerun a previously loaded test. NUnit loads up the DLL containing the class, locates the test method and executes the test, showing me the results.

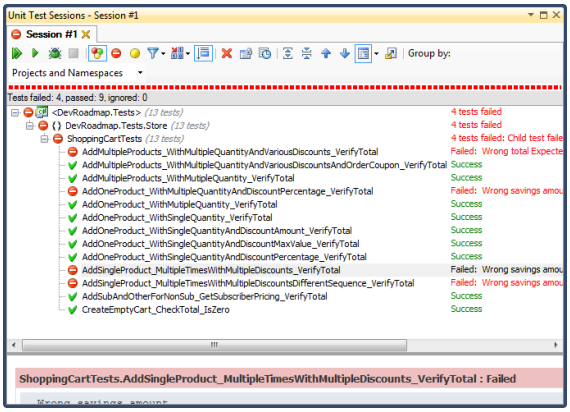


Figure 1. Using NUnit

In an instant I can see whether or not my test has passed or failed. And, sometimes more importantly, I run an entire set of tests and find out if some change I'd made had caused collateral damage, making some other test fail unexpectedly.

I really liked using NUnit. Being able to test my code quickly and repeatedly, and instantly see the results, gives me a warm, fuzzy feeling.

Unit testing the hard way

I really wanted to be able to do this kind of unit testing in Clarion. But .NET has some advantages that make unit testing a whole lot easier.

In the code listing above you may have noticed this line of text:

```
[Test]
```

This is an attribute, which is sort of like a tag. It essentially attaches a bit of code and/or data to whatever follows. When NUnit runs and loads up a DLL, it looks for methods with the [Test] attribute. It then runs those methods and looks for the result of a special set of method calls including the `Assert.That()` statements in the above listing. (Actually NUnit first looks for a class with a [TestFixture] attribute - when it finds such a class it then examines that class for methods decorated with the [Test] attribute. But either way it's the same principle: it's the attribute that tells NUnit how to find test methods.)

NUnit can do its job because .NET has the ability to self-examine. That is, you can write code in .NET to look at running code and determine what classes are present, what methods are available, and so

forth. And then your code can dynamically call those methods and look for certain kinds of results.

Clarion has a degree of self-examination built in, primarily via the `WHO`, `WHAT` and `WHERE` functions. But it doesn't have anything like .NET's capabilities (what .NET calls *reflection*), and `WHO`, `WHAT` and `WHERE` are primarily useful for structures, not procedures and methods.

Clarion can dynamically load up a DLL (via the Windows API) and, using some brute force examination of the DLL, determine which procedures are available to be called. But there isn't any mechanism like attributes that lets you the developer tag certain procedures within the DLL as test procedures. There also isn't any mechanism for determining what kinds of data those test procedures might return.

In the end I settled on the following strategy. When you tell ClarionTest to run any tests available in a given DLL, ClarionTest goes through the following steps:

1. It looks through the test DLL for a specially named procedure called `CLARIONTEST_GETLISTOFTESTPROCEDURES`. I think you'll agree that this is a procedure name that is unlikely to be used for any other purpose.
2. If `CLARIONTEST_GETLISTOFTESTPROCEDURES` is found, ClarionTest calls that procedure and obtains a list of test procedures in that DLL.
3. Based on input from the ClarionTest user, or based on command line parameters passed in, ClarionTest executes one or more of the available test procedures using the Windows API. Those test procedures have a standard prototype which allows ClarionTest to obtain an instance of a class containing the test results.
4. Each test procedure contains test code along with one or more `AssertThat` statements. `AssertThat` evaluates two inputs and sets the test result class data accordingly.
5. When the test procedure returns, ClarionTest displays the result of the test.

Now, if you had to build all of that infrastructure just to create one unit test, you'd never do it. It'd be way too much work.

The easier hard way

Happily, Clarion has this thing called an AppGen that makes the whole process so much simpler. All you really need to do is create an application, populate one global extension, and create a test procedure using a special procedure template. You can then plug in whatever test code you like into that test procedure using the embeditor, build the DLL, and tell ClarionTest to load up the DLL and run any or all of the test procedures contained therein.

Figure 2 shows the

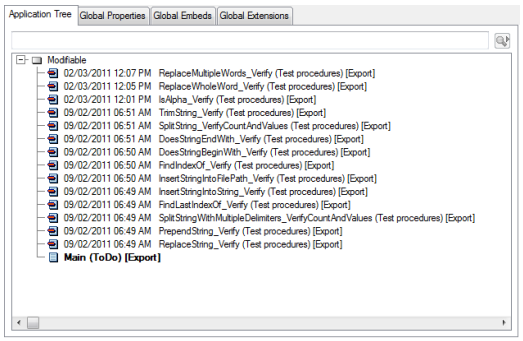


Figure 2. The test DLL

Figure 3 shows the ClarionTest UI with the test DLL from Figure 2 loaded, and Figure 4 shows the detail of the test results.

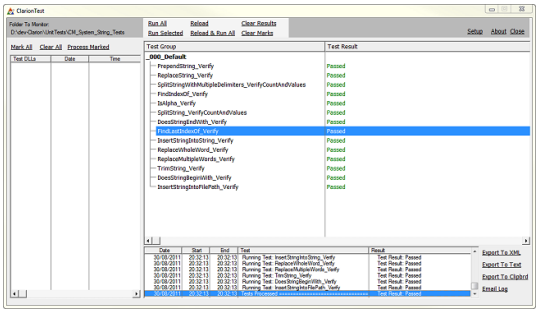


Figure 3. ClarionTest's UI (with enhancements by John Hickey)

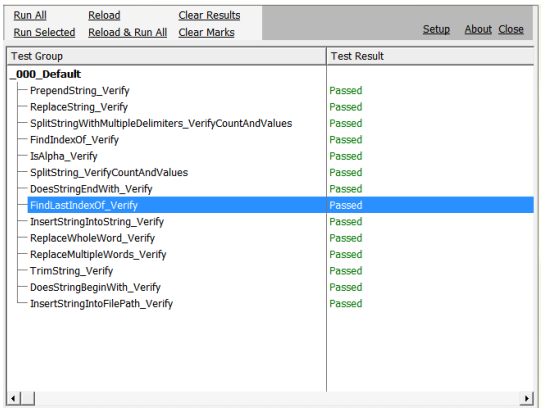


Figure 4. Detail of test results

Compared to available tooling in.NET, ClarionTest is still a little crude. But it does enable automated unit testing under Clarion, and that's pretty useful. John Hickey, in particular, has made some very nice improvements to ClarionTest which he and I will write about in upcoming articles.

I continue to use ClarionTest on an almost daily basis, and the larger and more complex the system the more I use ClarionTest to help me build the small, highly tested components which I can use to construct those larger systems.

In Part 2 I'll provide a link to the latest version of ClarionTest. I'll walk through a simple test DLL, and I'll also show some of the code that enables ClarionTest to do its work. Not surprisingly, a lot of that code comes straight from the pages of Clarion Magazine.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

BACK TO TOP

A Windows 7 Alt-Lockup Fix For Clarion 6, Part 2

By Carl Barnes

Posted August 31 2011

In Part 1 I explained the Windows 7 Alt lockup problem affecting Clarion 6, and I argued that the best approach to fixing the problem is to intercept the Windows message that causes the lockup. I also said that the solution involved subclassing the frame window.

The subclassing solution

What is subclassing? Each window receives and processes messages in a special procedure named the "Window Procedure" or WndProc for short. The Clarion Run Time Library (RTL) implements the WndProc, takes those messages and either sends them to the Accept loop as a Clarion event, or takes other action. To subclass a window is to insert your own WndProc procedure to receive the Windows messages before the RTL WndProc can get the messages. Your WndProc can prevent the RTL from getting the message, which in this case will prevent the application from locking up.

Enough talking about this problem, let's get to this solution's code. Below is the frame subclass procedure that must be added to your frame app.

```
SubClassFrame  PROCEDURE(SIGNED _hWnd, UNSIGNED _wMsg, UNSIGNED _wParam, SIGNED _LParam),SIGNED,P
WM_SYSCOMMAND  EQUATE(0112h)
SC_KEYMENU     EQUATE(0F100h)
Alt1stMenu     EQUATE(val('f')) !&File is Alt+f
  CODE
  CASE _wMsg
  OF WM_SYSCOMMAND
     IF BAND(_wParam,0FFF0h)=SC_KEYMENU | !0Fh internal
     AND _LParam=0 THEN     !LParam=0 if ALT or F10
         !Send Alt+F message to frame to drop file menu
         PostMessage(_hWnd, WM_SYSCOMMAND, SC_KEYMENU, Alt1stMenu)
         RETURN(True)  !Cutoff RTL or it will lockup
     END
  END
  RETURN(CallWindowProc(AppFrame_OrigWndProc, _hWnd, _wMsg, _wParam, _LParam))
!SubClassFrame() must be in the same Module as the Frame
!AppFrame_OrigWndProc LONG is in same Module as data
```

The Windows default WndProc for the MDI child window (`DefMDIChildProc`) spots the F10 or Alt `WM_Keyup` message in the child window and sends the frame a `WM_SYSCOMMAND` message with the wParam as `SC_KEYMENU` and the `lParam` as zero. My testing found this message is what causes the menu bar to take focus, and it's the message that locks up a Clarion app.

To prevent the lockup this message must be stopped. If you simply `RETURN(True)` to stop the frame from getting the message the lockup won't occur, but the Alt key will no longer act normally and select the menu bar. My preferred solution is to post a message to the frame, sending the "f" key so the file menu is dropped. (You will need to change the equate if `Menu('&File')` is not your first menu.) Here's what the application looks like after pressing and releasing Alt or F10:

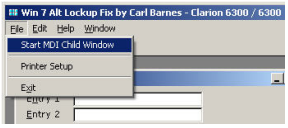


Figure 1. The File menu forced to drop down

In the MSDN WM_SYSCOMMAND documentation page the remarks section spells out clearly how menu messages work:

> If the wParam is SC_KEYMENU, lParam contains the character code of the key that is used with the ALT key to display the popup menu. For example, pressing ALT+F to display the File popup will cause a WM_SYSCOMMAND with wParam equal to SC_KEYMENU and lParam equal to 'f'.

Finally, the fix code does a `Return(True)` telling Windows the message was handled by the sublass procedure. All other messages fall through to the `Return( CallWindowProc (AppFrame_OrigWndProc …))` that forward the message to the RTL `WndProc` for the normal event processing Clarion developers expect from `ACCEPT`.

Originally instead of the `PostMessage()` I tried a slightly simpler approach to send the "f" message to drop the file menu. I changed the original message key with `_LParam=VAL('f')` then let it fall through and call `CallWindowProc()` passing the "f" key. This worked perfectly in my tests, except on one system. In retrospect I think that test was done incorrectly. I decided I liked the `PostMessage()` code as cleaner and simpler, so I kept it. Here's the changed code I decided *not* to use:

```
    IF BAND(_wParam,0FFF0h)=SC_KEYMENU | !0Fh internal
    AND _LParam=0 THEN     !LParam=0 if ALT or F10
        !Send Alt+F message to frame to drop file menu
        _LParam = Alt1stMenu
    END
END
RETURN(CallWindowProc(AppFrame_OrigWndProc, _hWnd, _wMsg, _wParam, _LParam))
```

One tricky detail in this code is that the wParam must have the lower four bits stripped (BAND (W,0FFF0h)) before testing it for equalness to SC_KeyMenu. I found that detail reading MSDN about `WM_SYSCOMMAND` all the way down in the remarks section where it states "the four low-order bits of the wParam parameter are used internally by the system…an application must combine the value 0xFFF0 with the wParam value by using the bitwise AND". When calling the API it's always a good idea to read the complete MSDN page on the function, plus the linked pages on the parameters and enumerations. I think the Windows SDK should define an SC_Mask of `0FFF0h` equate; seeing that declarion would have made the requirement to strip the low four bits more obvious. In my testing the low four bits were always zero, so maybe they are not normally sent in an `SC_KeyMenu` message.

Having Alt (or F10) drop the File menu is not standard Windows behavior but I prefer it as much more visual to the user. I feel most users don't intentionally active the menu bar using the Alt key and most likely do it accidentally when they press Alt with the intent of pushing a child window prompt accelerator &key, but decide not to and release Alt. When that causes the file menu to take focus the child window loses focus and confuses many users because it is so visually subtle. To remove this comment out the PostMessage line and the Alt key will not drop the menu, which is also not standard behavior. You could make this menu behavior user selectable.

When I first implemented this fix I checked `GetVersionEx()` and only did it on Windows 7 systems. When I ran into Windows 7 systems still locking up it took me some time to figure out the reason my fix did not work was the shortcut had been setup with Compatibility set to Windows XP. That causes `GetVersionEx()` to return XP. I decided to remove the version check and implement this fix for all versions of Windows. This way the behavior of Alt dropping the File menu will be standard across all platforms.

Connecting the subclass procedure to the frame

The `SubClassFrame` procedure should be added to your App in the same Module as the Frame procedure (which is typically named `Main`) so it can have the best chance of being next to the frame object code in memory. That also lets you use module data instead of global data for the `LONG` variable AppFrame_OrigWndProc. Both are identical static data, declaring it in the module limits the scope to that module. The variable is only required by the two procedures so it's best to minimize its scope.

To actually make this work and have the Frame window messages go to the `SubClassFrame` procedure you must insert your call into the WndProc chain with the below two lines code. This code must be added after the frame window is open:

```
>>> OPEN(AppFrame) <<< find this existing  line
AppFrame_OrigWndProc   = AppFrame{PROP:WndProc}
AppFrame{PROP:WndProc} = Address(SubClassFrame)
```

The final step is to define the prototypes for the two Windows API functions called. Open the Global embeds and find the "Inside the Global Map" embed point, then add this code:

```
MODULE('Win32')
```

```
    CallWindowProc(SIGNED lpPrevWndProc,UNSIGNED hWnd,|
      UNSIGNED wMsg,UNSIGNED wParam,SIGNED lParam),SIGNED,|
      RAW,PASCAL,DLL(1),NAME('CallWindowProcA')
    PostMessage(UNSIGNED hWnd,UNSIGNED uMsg,UNSIGNED wParam,|
      LONG lParam),BOOL,PASCAL,PROC,DLL(1),NAME('PostMessageA')
  END
```

That's it, compile and test. If you get an "Indistinguishable new prototype" compile error referencing `CallWindowProc` or `PostMessage` that means they have already been defined in the map. Many third party templates use these API calls so don't be surprised. Reopen the embed point and comment out the declaration. That error might also indicate your frame was already subclassed.

Frame already subclassed

If your frame is already subclassed the best way to implement this fix is to find your existing subclass procedure and add the five lines of code to handle the `WM_SYSCOMMAND` message. The best way to check if your frame is subclassed is to open the frame source and search for `WndProc`. If you find the code `AppFrame{PROP:WndProc} = Address(SubXxx)` then `SubXxx` is the name of the procedure you want to modify. Try to find it in your procedure tree and view its source. If you can't find it or can't embed code into that existing subclass procedure then it is ok to implement another subclass procedure as described above. They call each other in a chain until one of them returns true.

How I found the message

First, how did I know the lockup was caused by a message? Windows' approach to receiving user input is mostly based on messages being posted, queued, then processed, so messages are a logical place to start.

I knew how to reproduce the problem. I wanted to press and release the Alt key, then view all the messages. A crude way to see the messages is to add the subclass procedure to the frame and have it dump all the messages to DebugView using `OutputDebugString`. The trouble is for each message the output will simply be four numbers; you will have to decode them using the SDK or MSDN. That can be time consuming, and most messages will be a dead end. There is also a chance the message might be affected by another `WndProc`.

Microsoft makes a tool named Spy++ (spyxx.exe) that will hook all the messages for a Window and decode every number into an SDK equate name. It will not decode Clarion Event loop messages, they will show up as `WM_App+Number`. The "Number" is an Offset + Event Equate. The actual offset used varies by runtime version. You should be able to reverse engineer a way to decode these by posting a few events.

Spy++ makes it an easy drag and drop process to spy on a window's messages, no programming required. Below is a screen capture of Spy++ setup to hook the frame's messages. I pressed the F10 key on a child window and Spy++ nicely decoded messages to show the equate name for each parameter number. Scanning these messages the text "KeyMenu" caught my attention as probably menu related. And it was one of the last messages processed.

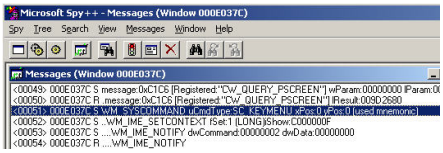


Figure 2. Using Spy++

My copy of Spy++ came with Visual Studio in the Tools folder. It may also come with the free Windows Platform SDK or be downloadable.

To use Spy++ select File, then Messages. Drag and drop the finder icon onto your application's frame.

Spy++ has many options to filter out messages you've determined do not apply. Spy++ does not sublass the window, it gets the messages via the SetWindowsHookEx() API so it will always see all the messages. It will also let you spy on all of the controls on a window to figure out their structure, settings, properties and design. Spy++ is a very useful tool that should be in everyone's toolkit. Older versions of Spy++ occasionally locked up the entire system, so be sure to get the most recent version possible, save your work before using it, and don't leave it running when you don't need it.

Attached example

The downloadable source zip contains a working example (screenshot below) that has a frame and child

window to let to test the fix. You have the option to turn off the fix to see the application lock up, plus you can alert F10 and Alt+F on the child window to verify the fix allows local use of those keys to work normally.

The AltFix7.CLW file contains the four snippets of code you need to insert into your application plus detailed step-by-step instructions in the comments. As a bonus the example also includes the four lines of code you need in the subclass procedure to have an application close without user intervention when Windows shuts down.

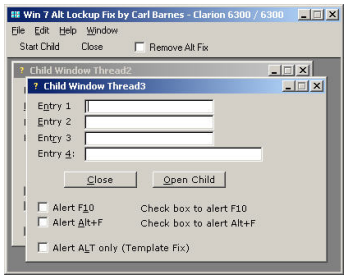


Figure 3. The demo app

Summary

If your Clarion application was developed using Clarion 6 or before and is multithreaded MDI, then you should implement the attached fix to prevent users from locking up when they press and release the Alt key, or press F10. To implement the fix in your program open AltFix7.CLW from the download zip, read the instructions in the comments and copy/paste the code into your frame app. There are just four code snippets to embed. If you have any doubts test your app on a few Windows 7 machines, especially one with Outlook or Live Mail installed and running. Be sure to test with a child window open. The fix should work all the way back to Clarion 1.5. I've tested it under 5, 5.5 and 6. If you are using Clarion 7.1 or later you probably do not need the fix; however, you may want to implement the code if you like the behavior of the Alt key dropping the first menu as being very visually obvious to the user his child window has lost focus.

Links

- Applications hang on Windows 7 after pressing Alt key - also with Clarion 7.1? (sv.clarion.clarion7)
- SV - Clarion 6/7 and ALT-KEY Bug (sv.clarion.clarion7)
- Hardcore Clarion - Sub-Classed Windows - This 1998 article uses `SetWindowLong` to install the subclass; you must use the `Clarion Prop:WndProc` for the RTL to work properly.

Download the source

Carl Barnes is an independent consultant working in the Chicago area. He has been using Clarion since 1990, is a member of Team TopSpeed and a TopSpeed Certified Support Professional. He is the author of a number of Clarion utilities including CW Assistant, The CHM help class CHM4Clarion, and Clarion Source Search.

Article comments

BACK TO TOP

Using Clarion.NET's Easier Interop

By Graham Dawson

Posted August 31 2011

The latest Clarion.NET release 8394 came with an important enhancement, the ability to easily expose procedures so that they are visible to Clarion Win32 applications.

The key word there is *easily*. It's always been possible, but you needed to be able to use the .NET IL disassembler and assembler programs, and you had to know exactly what to do. Not a task for the faint hearted.

So just how easy does the new release make it?

Well it turns out to be surprisingly simple.

.NET is just like Win32 in that you can only export things from a DLL (assembly in .NET speak) not an EXE file, so make sure the Clarion.NET project type is set to Class Library.

Let's take a simple procedure, one that takes no parameters and returns no value. Say you have a procedure in Clarion.NET

```
                    MAP !Interop Exported procedures
EnableWatching          PROCEDURE()
                    END
```

You need to add some attributes to the declaration, as follows

```
                    MAP !Interop Exported procedures
EnableWatching          PROCEDURE(),NAME('EnableWatching'),PUBLIC
                    END
```

It helps if you have a separate MAP structure for each of the three types of procedures

- Interop Exported procedures (procedures you will call from a Clarion Win32 application – the subject of this article)
- PInvoke Imported procedures (Clarion Win32 procedures to call from your .NET application)
- and well, normal procedures

All that remains is to tell the IDE to act on those additional attributes. You do that by going into the Application tab of the project Properties and setting the checkbox 'Export global named procedures and create Clarion Win32 LIB file' to ticked…

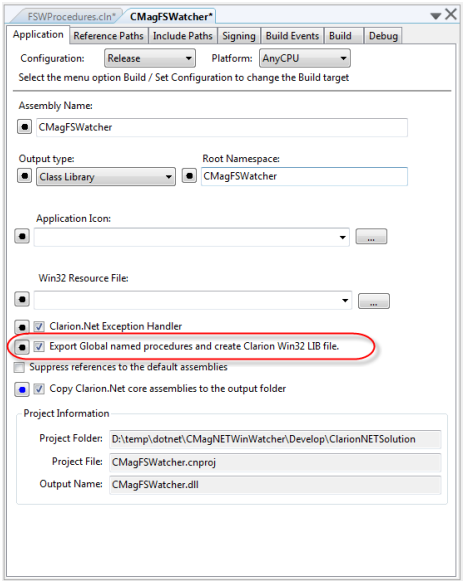


Figure 1. Creating the Win32 LIB

Now if you build the project the created assembly will be amended (see below) to make it usable from Clarion Win32 and `EnableWatching` will be exported.

Just like any other DLL you need a LIB file in order to be able to use it from Clarion Win32. The build creates the LIB file for you, but there are a couple of 'gotchas' here.

First, the build process (by default) puts the output files in the Debug or Release project sub-folders, however the LIB file itself gets created in the Project folder. If you're like me you'll add the LIB file into the Project as an Existing Item with the 'Copy to output directory' property set to Always.

Second - the build process doesn't always seem to update the LIB file correctly . The solution is to always do a clean before the build.

OK, so that's the .NET end done, how do you use the newly created DLL from Clarion Win32?

Well just like 'normal' DLL usage you need to add the LIB file into the project and declare the procedure.

```
                MODULE('CMagFSWatcher.DLL')
EnableWatching      PROCEDURE(),NAME('EnableWatching'),PASCAL,DLL(True)
                END
```

Note the additional `PASCAL` attribute that sets the calling convention used by the DLL.

Now assuming you have a body to the procedure in Clarion.NET and a call in Clarion Win32 you just compile and run, and the .NET procedure will be called just like any other.

It gets a little more complicated when you start to add procedure parameters or return values, but not much !

To call a .NET procedure with an integer parameter you declare an int32 type in Clarion.NET:

```
                MAP !Interop Exported procedures
EnableWatching      PROCEDURE(),NAME('EnableWatching'),PUBLIC
SetupDebug          PROCEDURE(int32 parDebug),NAME('SetupDebug'),PUBLIC
                END
```

and a long type in Clarion Win32:

```
                MODULE('CMagFSWatcher.DLL')
EnableWatching      PROCEDURE(),NAME('EnableWatching'),PASCAL,DLL(True)
SetupDebug          PROCEDURE(long parDebug),NAME('SetupDebug'),PASCAL,DLL(True)
                END
```

Strings are treated slightly differently. Clarion.NET uses the .NET `STRING` datatype, which Clarion Win32 matches up to a *CString. Here's the .NET prototype:

```
                MAP !Interop Exported procedures
```

```
EnableWatching          PROCEDURE(),NAME('EnableWatching'),PUBLIC
SetupDebug              PROCEDURE(int32 parDebug),NAME('SetupDebug'),PUBLIC
SetupWatchedFolder      PROCEDURE(STRING parFolder),NAME('SetupWatchedFolder'),PUBLIC
                    END
```

And here's the Win32 prototype:

```
                    MODULE('CMagFSWatcher.DLL')
EnableWatching          PROCEDURE(),NAME('EnableWatching'),PASCAL,DLL(True)
SetupDebug              PROCEDURE(long parDebug),NAME('SetupDebug'),PASCAL,DLL(True)
SetupWatchedFolder      PROCEDURE(*CSTRING parFolder),NAME('SetupWatchedFolder'),PASCAL,RAW,DLL(T
                    END
```

As well as the datatype difference you also need to add the `RAW` attribute at the Win32 side.

There's another little 'gotcha' here: you may think that because the parameter is a reference that any changes you make at the Clarion.NET end will be passed back into Clarion Win32.

Well they aren't, the caller just needs the address that's all.

If you want to pass strings back then you have to use a return value as below.

In Clarion.NET

```
                    MAP !Interop Exported procedures
EnableWatching          PROCEDURE(),NAME('EnableWatching'),PUBLIC
SetupDebug              PROCEDURE(int32 parDebug),NAME('SetupDebug'),PUBLIC
SetupWatchedFolder      PROCEDURE(STRING parFolder),NAME('SetupWatchedFolder'),PUBLIC
ReturnFilename          PROCEDURE(),String,NAME('ReturnFilename'),PUBLIC
                    END
```

becomes in Clarion Win32

```
                    MODULE('CMagFSWatcher.DLL')
EnableWatching          PROCEDURE(),NAME('EnableWatching'),PASCAL,DLL(True)
SetupDebug              PROCEDURE(long parDebug),NAME('SetupDebug'),PASCAL,DLL(True)
SetupWatchedFolder      PROCEDURE(*CSTRING parFolder),NAME('SetupWatchedFolder'),PASCAL,RAW,DLL(T
ReturnFilename          PROCEDURE(),*CString,NAME('ReturnFilename'),PASCAL,DLL(True)
                    END
```

That covers the basics of Clarion.NET's new easy interop.

OK So what's it good for?

So you've learnt how to easily call Clarion.NET procedures from Clarion Win32. But why would you want to do this. In other words what's it good for?

Well there are things that .NET does easily that are difficult to accomplish in Win32.

Take for example the common task of monitoring a folder for the creation of a file. Let's say that when the file arrives you rename it and copy it to a holding folder for later processing.

Normally you'd use some sort of timer and poll the folder doing a `DIRECTORY` on each `Event:Timer` to look for the new arrivals. Not very efficient, you either poll often, and waste resources, or too infrequently and end up processing the file late or worse still having the file overwritten by another new arrival.

Well, .NETs vast class library has a `FileSystemWatcher` component that will handle the above specification with no timers, no hogging processor time, and instant reaction when selected changes occur.

In the downloadable source zip you'll find a program made up of three elements:

1. A Clarion.NET solution creating a Clarion Win32 callable Class Library
2. A Clarion Win32 project that will setup the FileSystemWatcher via the Clarion.NET DLL and receive the results
3. A Clarion Win32 DLL which will provide the method of sending events from Clarion.NET into Clarion Win32

I'm not going to go into every procedure in the source, it's all attached and some of the procedures have already been outlined in Part 1.

A few of points need explaining though.

a) Why is there a form in the Clarion.NET project?

The FileSystemWatcher requires something called a SynchronizingObject, usually a form. You need to actually show the form which introduces a new problem because if the results are being shown in the Clarion Win32 program you don't really want a .NET form around.

So you need to hide the form.

In the Form1.cln file you'll see that the parent SetVisibleCore method has been overridden so that when debug is false the form will not be shown. (Thanks to Dennis Evans for helping me sort that one out.) Note that hiding the form in any other way results in a noticeable flash onscreen.

b) How are events passed from Clarion.NET into Clarion Win32

This is the big one: you need a way of signalling to the Win32 program that a file has been placed in the holding folder. If the program was wholly Win32 you'd probably use POST to send an event from the polling thread to the processing thread. There isn't a POST command in Clarion.NET, but what if you use the Clarion Win32 RTL? Clarion.NET can use something called Platform Invoke (PInvoke) to call Win32 exported procedures. So can you declare the POST procedure, put the Clarion RTL DLL in the folder and use it in that way?

Well no, it turns out the Clarion RTL cannot be used in this way, Diego knows the full gory details but take it from me I've tried, GPFs and unhandled exceptions all over the shop !

However it is possible to 'wrap' the Clarion RTL POST procedure so that it is callable.

Simply create a Clarion Win32 DLL project that exports a procedure that takes four arguments and calls POST. Here's the complete program source:

```
    PROGRAM


    MAP
        MyPost(UNSIGNED parEvent, SIGNED parControl, SIGNED parThread, |
            SIGNED parPosition),PASCAL
    END


  CODE


MyPost      PROCEDURE(UNSIGNED parEvent, SIGNED parControl, |
                SIGNED parThread, SIGNED parPosition)

    CODE
    POST(parEvent,  parControl,  parThread,  parPosition)
```

You'll need a custom .EXP file to go along with the code; one is included in the project. You may also want to compile the project with the Local option so you'll have no versioning problems when used with other releases of Clarion.

```
LIBRARY 'WrappedPost' GUI
EXPORTS
  MyPost @?
```

To use the wrapped POST procedure at the Clarion.NET end you have to prototype it:

```
                    MAP !PInvoke Imported procedures
                        [DllImport('WrappedPost.DLL')]
MyPost                  PROCEDURE(CLALONG parEvent ,CLALONG parControl ,|
                          CLALONG parThread ,CLALONG parPosition),NAME('MyPost')
                    END
```

Note: You don't need a LIB file for Clarion.NET to use a PInvoke exported procedure, just the DLL.

To use the wrapped POST simply call it with

```
MyPost(theEvent,theControl,theThread,thePriority)
```

See the HandleFSWEvent procedure in the Clarion.NET project for an example of how the POST wrapper is used.. The details of the post event to post, control, thread etc having been setup earlier with

the `SetupPostDetails` procedure.

c) Why are filenames stored in a queue at the Clarion.NET side?

The filewatcher is posting events back from Clarion.NET into Clarion Win32 but there's no guarantee that they'll be handled by the receiving `ACCEPT` loop immediately (even if the priority is set to 1), and by the time the `ACCEPT` loop handles the next event a new file may have arrived and the filename changed. So the filenames are stored in a queue and retrieved from the top by the ReturnFilename procedure.

To use NETWinWatcher just set up a folder to watch and a folder to copy the resultant files to; both are required.

Then click Enable to begin watching. As files are copied from watched folder to copy folder they are prefixed with a number so you can clearly see the creation sequence. Selecting an item in the listbox and double-clicking will open that file up in Notepad which is surprisingly good for examining just about any type of file, text or binary.

Clear will remove (permanently) the copied files from the copy folder.

Debug (only available before you Enable for the first time) will make visible the form in the Clarion.NET project which echoes the files found.

Now you know how to easily call .NET code from Win32, and hopefully you have an idea of why you'd want to do that. Now it's time to see how Clarion.NET works this magic.

Clever SoftVelocity - so how's it done?

The ClarionSharp Blog reveals most of the details:

> In order to expose the Procedures for use in a Win32 program the binary Clarion# DLL file has to be disassembled back to IL code, the IL code is then modified, and then the IL code has to be assembled back to a binary (PE format) file. To do this there are two programs from the .Net SDK that have to be found; Ilasm and Ildasm. In the current implementation the location for those files is stored in an external .config file, and if they are not found (Ilasm and Ildasm), you'll get an error message telling you to edit the path stored in Dllexport.exe.config.

Can we see that in action?

Well you now have the NETWinWatcher.exe program, and while it isn't exactly designed to catch the very temporary files created by this process, it will do the job.

Which folder needs to be watched?

Normally temporary files are created in the users temp folder, i.e.

```
SystemDrive:\Users\YourUsername\AppData\Local\Temp
```

But in this case the temporary files aren't created there but in the project's `\Obj\Debug` or `\Obj\Release` folder, depending on the build configuration.

Lets look at the files created when the shipping ManagedDLL example project is built. Start NETWinWatcher use the locator buttons set the watched and copy folders and the click on Enable to start watching…

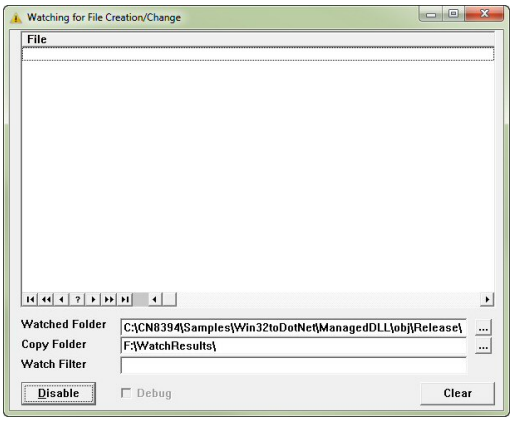


Figure 2. NetWinWatcher

Now clean the ManagedDLL project followed by a build.

NB if you receive errors from Clarion.NET just click on Continue (as stated above NETWinWatcher wasn't really designed for this particular task)

These are the results…

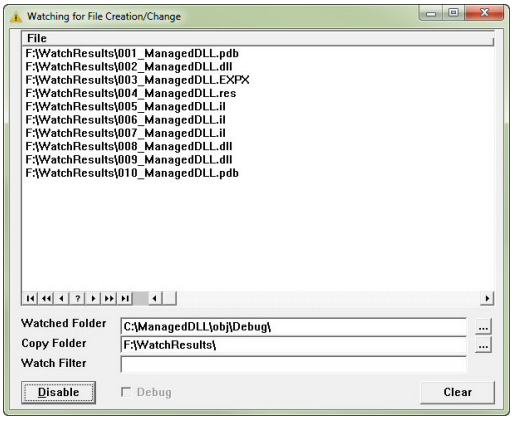


Figure 3. NetWinWatcher displaying results

It's clear that the output ManagedDLL.dll is actually built twice and in between build one and two there are IL (.NET Intermediate Language) files produced along with a mysterious .EXPX file.

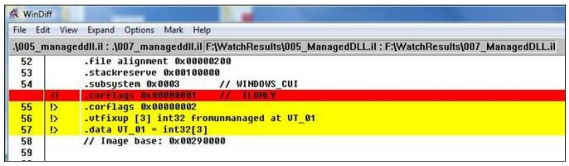
The EXPX file is an XML file that tells the DLLExport.EXE program (mentioned in the ClarionSharp Blog referred to above) which procedures to amend for export.

```
<?xml version="1.0" encoding="utf-8"?>
<export>
  <class name="ManagedDLL.GlobalSomeFunctions">
    <method name="SumTwo" export="SumTwo" call="System.Runtime.CompilerServices.CallConvStdcall" />
    <method name="SayHello" export="SayHello" call="System.Runtime.CompilerServices.CallConvStdcall" />
    <method name="SayHelloName" export="SayHelloName" call="System.Runtime.CompilerServices.CallConvStdcall" />
  </class>
</export>
```

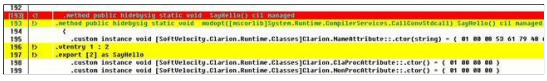
Figure 4. The EXPX file

It's the DLLExport.EXE program that actually does the business, first calling ILDasm to disassemble the newly created assembly, then amend it to enable procedures to be exported, and finally calling ILasm to re-assemble the amended IL language into a new version of the DLL.

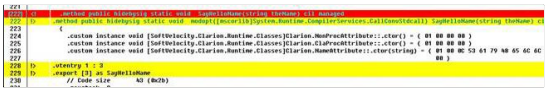
If you compare the first IL file with the last IL file in a differencing program (I use WinDiff) you can see the changes. (Red items are before, yellow items are after change)



`SayHello` is exported:



`SayHelloName` is exported



and finally `SumTwo`.

Don't ask me how the IL works but there are plenty of resources on the Web that deal with this method, e.g.

http://www.codeproject.com/KB/dotnet/DllExport.aspx

So there you have it you can now call .NET procedures and make use of both visual and non-visual components from Clarion Win32 programs. The possibilities are endless!

For instance, one of the changes I made to the initial release of NETWinWatcher was to add an entry field (rather than just a display string) to enable the watched folder to be typed in manually rather than located with a FileDialog.

Why? Well, Clarion Win32s FileDialog won't display hidden folders, but Clarion.NET's equivalent the FileBrowserDialog will…

Download the source

Graham Dawson has been tinkering with computers since the days of the Sinclair ZX81. After a career as a TV and Radio service engineer he got a job with SDM building and configuring PCs and mini computers, which led him to study computer programming at the Open University. Graham began using Clarion with version 2.003. He "retired" in 2006 but was persuaded to return to SDM a year later. He is the company's ".NET guy", and SDM had a Clarion.NET PDA application in production. He lives in Chester in the UK with his lovely wife Rachel.

Article comments

by surfersteve on September 2 2011 (comment link)

many thanks graham, its nice to see the two way post being used in this way. i was looking foward to your article and apperciate your time and effects. many thanks again!

---

by Graham Dawson on September 2 2011 (comment link)

Hi Steve,

I'd love to see if Pierre Tremblay has any comments on the 'wrapped POST' method, just in case there are any hidden problems. But in my testing at it does seem to work without issue, and it's so simple.

Glad you liked the article.

Graham

---

by milovan radosevic on September 2 2011 (comment link)

Thanks for the post. I will probably test it in the following months.

Nenad

---

by Paul Konyk on September 2 2011 (comment link)

My thanks too Graham. I've have a project where I will be putting all of this to work.

BACK TO TOP

Clarion 8's New Gradients

By Mike Hanson

Posted August 31 2011

SoftVelocity recently gave Clarion 8 the ability to display gradients for TOOLBAR, PANEL, BOX and ELLIPSE controls. The window designer doesn't yet let us specify them directly, so the only way to use this feature is to via three new runtime properties:

- PROP:GradientType
- PROP:GradientFromColor
- PROP:GradientToColor

The first one gets one of the following values (as defined in ...\Clarion8\LibSrc\Win\Equates.clw):

```
GradientTypes     ITEMIZE(0),PRE
Off                 EQUATE
Vertical            EQUATE
Horizontal          EQUATE
VerticalCylinder    EQUATE
HorizontalCylinder  EQUATE
DiagonalTopLeft     EQUATE
DiagonalBottomLeft  EQUATE
DiagonalTopRight    EQUATE
DiagonalBottomRight EQUATE
                  END
```

The FromColor and ToColor properties are standard Clarion 0BBGGRRh (blue, green, red) colors, or you can use the COLOR:* equates that are also defined in Equates.clw.

> **NOTE:** If you want a gradient for a control that doesn't support it, make the control transparent and place a gradient box behind it. You'll probably also want to set the border color to NONE.

Applying a gradient at runtime is simply a matter of assigning the properties for your control:

```
?Panel{PROP:GradientType     } = GradientTypes:Vertical
?Panel{PROP:GradientFromColor} = COLOR:White
?Panel{PROP:GradientToColor  } = COLOR:Black
```

Even though this syntax is not difficult, it gets verbose and tedious when you have to apply it to a bunch of controls, so I created a simple function to reduce the code to this:

```
ST::SetGradient(?Panel, GradientTypes:Vertical, |
                COLOR:White, COLOR:Black)
```

Of course, deciding what colors look good is a hit and miss affair. There are a number of good color management programs out there (Color Schemer Studio is my favorite), but almost none of them understand Clarion's BGR color values, its built-in COLOR:Equates, or the way it likes to format hexadecimal numbers as 0xxxxxxh. Many of them don't include a feature to preview gradients, and even if they do they won't look the same as those produced by Clarion's own algorithms.

To remedy this I built a little utility that accommodates these issues, which looks like this:

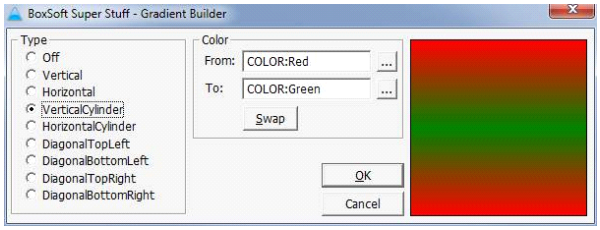


Figure 1. The Gradient Builder

The procedures are prototyped inside StMhGrad.inc:

```
                    PRAGMA('project(#compile StMhGrad.clw)')
                    MODULE('StMhGrad.inc')
 ST::SetGradient      PROCEDURE(SIGNED FEQ,
                                BYTE GradientType,
                                LONG GradientFromColor,
                                LONG GradientToColor)
 ST::BuildGradient    PROCEDURE(*BYTE Type,
                                *LONG FromColor,
                                *LONG ToColor,
                                <BOOL CommandToClipboard>),BYTE
                    END
```

You include it in your program by mentioning it in the embed "Inside the Global Map":

```
INCLUDE('StMhGrad.inc')
```

There's one noteworthy item in the INC file above: the PRAGMA directive tells it to compile and link the module into your EXE/DLL. The alternative is to manually add it to the list of modules in your project settings, but automatic is always better.

You'll usually use ST::SetGradient in your running programs, whereas ST::BuildGradient is more likely to be called as a utility while you're designing your windows. (You could also call it in your running apps, if you want your users to change the gradients at runtime.)

ST::BuildGradient has three output parameters (passed by referenced and used to return multiple values to the caller). Note that it supplies the raw values rather than the equated ones that you would see in your code. There are situations where either could be useful, so to streamline coding there's a fourth optional parameter: CommandToClipboard. If you pass it TRUE, it will format a sample command and copy it onto the Windows clipboard; you can then paste the command to the source editor.

The function returns TRUE if the user presses OK, or FALSE otherwise. Here's a sample of all the output:

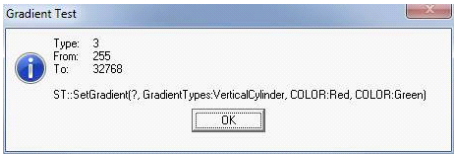


Figure 2. Test output

If you examine StMhGrad.clw, you'll find algorithms for doing conversions between color names and color values, as well as between LONGs and hexadecimal STRINGs.

The Pragma trick works great for procedural code, but things are even easier when you're writing object orienting code.

If you were declaring a CLASS module rather than regular procedures, then the PRAGMA would be unnecessary. Instead you would do this:

```
MyClass  CLASS,TYPE,|
  MODULE('MyClass.clw'),
  LINK('MyClass.clw')
```

The LINK attribute tells the compiler to compile the specified file, and it tells the linker to include the

resulting OBJ file in the project.

Eventually Clarion's own window and report designers will be able to specify the gradient parameters, so you can preview them in your own windows. Until that happens, `ST::BuildGradient` can help you to visualize them. Even when the designers accommodate gradients, the `ST::SetGradient` procedure will help to keep your code clear and concise.

Download the source

Mike Hanson is affiliated with BoxSoft, which produces the "Super" series of templates, distributed through Mitten Software. He has been creating add-on products for Clarion since his Public Domain Models for CPD 2.0 back in 1988. He's also written articles for every Clarion-related publication, and has spoken at numerous conferences and training seminars. If you have any questions, you can reach him via www.boxsoft.net.

Article comments

by surfersteve on August 31 2011 (comment link)

thanks mike, i assume the gold release of clarion 8 does let us do it in designer mode, id hope so

by Dave Harms on September 1 2011 (comment link)

Steve, to my knowledge there's no gradient support in the designer as of the C8 Gold release.

by surfersteve on September 2 2011 (comment link)

its ok, im very grateful for your article and your sample code. im learning a lot.

by Mike Hanson on September 6 2011 (comment link)

Thanks! I'm glad you enjoyed it.

BACK TO TOP