Best practices when coding EwE6

This page describes standard solutions for recurring oddities and nasties that we have ran into over the years when building EwE6.

The following issues are discussed on this page:

Robustness

Try / Catch ==

Compatibility

Mono

.NET is in theory system-independent, and .NET applications can be deployed under Unix, Linux, MacOS, etc using runtime environments such as ?Mono. However, not all .NET features work under Mono, most notably the Microsoft.VisualBasic assembly. This can prove problematic since every Visual Basic project by default receives a reference to this assembly.

In EwE we have worked around this problem by redefining key types and constants from the Microsoft.VisualBasic assembly in ?EwEUtils. You should not use Chr, Asc, IIF, TriState, MsgBox, UBound, InStr, vbTab, vbCrLF and a whole whack of other old and familiar constructs if you wish your application to run on any other OS than Windows.

To remove reliance on Microsoft.VisualBasic simply remove the auto-generated reference in the project properties References page. For most constructs .NET offers a solid alternative (for instance Array.GetUpperBound()), but for other constructs you may have to be creative (vcTab becomes Convert.ToChar(9).ToString() - or do you have a better idea?).

Target processor

The EwE source code now fully utilizes 64-bit capabilities. In order to make sure that Windows finds the correct 32 or 64 bit Access drivers make sure you always compile your EwE6 main project against x86 or x64, never against AnyCPU in *Menu* > *Build* > *Configuration Manager*.

Note that 64-bit EwE will not be able to find 32-bit Access database drivers and vice-versa.

User Interface design guidelines

Please adhere to the User Interface Guidelines when building user interfaces.

Resources and localization

Theoretically, the EwE6 scientific interface can be localized to any language. In intention all language-specific elements in EwE are provided in either localizable forms or in localizable resource tables. We have tried to consistently implement this but exceptions may exist; please let us know if you find any.

The ScientificInterfaceShared assembly offers a whack of shared resources, such as strings and images, for plug-ins and the main Scientific Interface to share to reduce the amount of scattered resources that need localizing. In your assembly simply add a statement such as import

ScientificInterfaceShared.My.Resources = SharedResources, and access all shared resources on the imported SharedResources thingy.

When you develop your own plug-ins with a user interface, please try to stick to the following resource guidelines:

- Use resources provided in ScientificInterfaceShared when possible,
- Set the 'localizable' property of any forms that you develop to True.

Nasty experiences

- Always override *Dispose(bDisposing)* to clean up sub-classed UI elements, do not use *OnHandleDestroyed* because the .NET framework, which wraps Win32 controls, may call this method during the regular life span of a .NET control to do housekeeping. The call may be followed by *OnHandleCreated* - it's simply not a valid trigger to assume your control is dying.
- Note that the Visual Studio designer automagically places a Dispose method in its *.designer.vb files which is blocked from debugging. You may want to manually move this method to your main vb file and strip off *DebuggerNonUserCode* tags that prevent the debugger from stepping through the code.