# Creating a GUI using WTL

**UW Bothell, CSS**
**Chris Traina**
**August 2004**

---

Pre-requisites:

1) Visual Studio .NET 7.1

2) WTL 7.1 -- you can download it here:
   http://www.microsoft.com/downloads/details.aspx?FamilyId=1BE1EB52-AA96-4685-99A5-4256737781C5&displaylang=en)
   You will need to add the path to the WTL 7.1 include directory to your project's "Additional include directories" property (both for C/C++ and Resources, once the latter exists).

**WtlFrame:**

1) This project demonstrates how to create the basic framework for a WTL window. This creates the main window framework that additional resources/controls are added to.

2) stdafx.h: this file contains all the WTL (WTL is simply an extension of ATL, so all the files names start with atl) header files as well as the CAppModule variable required for all WTL applications.

3) WtlFrame.h: this is where the CWtlFrame class is declared
   a. CWtlFrame inherits from CFrameWindowImpl, a WTL base class.
   b. IDR_MAINFRAME is a resource ID that would normally go in resource.h. Because we don't have any resources that Visual Studio recognizes, we will not have the resource.h header until later (see WtlWindow below).
   c. The section between BEGIN_MSG_MAP(CWtlFrame) and END_MSG_MAP() maps all the Windows message/control handlers for this class. For example, OnCreate is called when the class is instantiated and can be used for initialization purposes.
4) WtlFrame.cpp: this is where the CWtlFrame class methods are implemented.
   a. OnCreate creates a simple status bar and calls the GetMessageLoop function to register itself to receive certain Windows messages.
   b. OnFileExit posts a message telling Windows to close the window.

5) main.cpp: this is where the CWtlFrame class is instantiated and the function that creates the window (Create) is called.

6) Once you have properly installed WTL 7.1 and modifed the project to include the WTL include directory, you should be able to compile and run this project. When it is run, it will display a plain window frame. The next step is to add a window to the frame.

**WtlWindow:**

1) WtlFrame.h: two modifications have been made to this file:
   a. A CWindowView member variable has been added to the class for creation of the "child" window.
   b. An additional line has been added to PreTranslateMessage so that, if the frame class cannot process a message, the message is passed to the child window.

2) WtlFrame.cpp: a crucial line is added to this file to call the function (Create) that creates the child window.

3) WtlWindow.h: this is where the child window class is declared
    a. Note that IDD_WTL_VIEW is a resource ID that is stored in resource.h. Because we have added resources (as you will soon see), Visual Studio has automatically created the resource.h file for us and populated it with the resource IDs. We had to manually re-locate the IDR_MAINFRAME #define to this file.
    b. A message handler (OnShow) has been added to handle any intialization of the child window.

4) WtlWindow.cpp: this is where the child window class methods are implemented

5) If you change from the "Solution Explorer" tab to the "Resource View" tab (or double-click on WtlTutorial.rc), you will see that a dialog resource called IDD_WTL_VIEW has been added. This resource is linked to the CWtlWindow class and is created/displayed when the class is created by the CWtlFrame class. We will use the Resource View more when we add additional controls.

6) You should now be able to compile and run this project. Now, instead of displaying an empty framework, it displays the CWtlWindow (IDD_WTL_VIEW) form within the the CWtlFrame.

7) Notes on Resources:
    a. Adding the first resource to the previous WtlFrame project is fairly simple. Simply go to the Resource View tab, right-click the root entity of the tree view, select Add, and select "Add Resource…". Expand "Dialog" and select IDD_FORMVIEW. The default ID (IDD_FORMVIEW) was changed to IDD_WTL_VIEW in the tutorial.
    b. How is resource ID linked to class? In the WtlWindow.h file, you will see that the class sets its internal IDD value to IDD_WTL_VIEW. This links the FORMVIEW resource that was just created to the CWtlWindow class. Thus, when you instantiate a CWtlWindow object (as WtlFrame does when it is created), a FORMVIEW object is created with it.
    c. Resource IDs are the way you get a handle to any control. If you want a handle to a button with the ID IDC_BN_MYBUTTON, you can do this:

```
CButton bnButton = GetDlgItem(IDC_BN_MYBUTTON);
```

**WtlWindowWithControls:**

1) WtlWindow.h:
    a. You will see that several new message handlers have been added, using a slightly different macro (COMMAND_HANDLER). These are handlers for controls that have been added to the window. For example, OnBnClickedModify has been added as the command handler for when the IDD_BN_MODIFY button is clicked. The BN_CLICKED parameter is a notification code that specifies what type of control event the handler will deal with. Other possible parameters (e.g. BN_PUSHED, BN_DISABLED, BN_SETFOCUS, etc.) can be found in winuser.h. Other options are available for different control types. For example, LBN_SELCHANGE is the notification code when the selected value in a list box changes and CBN_DBLCLK is the code for when a combo box is double-clicked.
    b. A new map has also been added:

```
BEGIN_DDX_MAP(CWtlWindow)
    DDX_TEXT(IDC_STATIC_MSG, m_StaticMsg)
END_DDX_MAP()
```

The DDX stands for Do Data eXchange and this code maps the contents of m_StaticMsg to the text for the IDC_STATIC_MSG control. This makes it possible to load the controls current text value into m_StaticMsg, using DoDataExchange(true, IDC_STATIC_MSG), or to modify m_StaticMsg and update the controls text value with the new m_StaticMsg value, using

DoDataExchange(false, IDC_STATIC_MSG). NOTE: if you want to update all the controls for which data exchanges are mapped, simply omit the control ID, e.g. DoDataExchange(false).

c. You can now compile and run the project. You can see how the combination of command handlers and data exchange mappings allow you to, among many other things, update a button's text, count the number of button clicks, update a static text control, read data out of an edit box, and start and monitor a timer.

2) Creating your own control:
   a. Go to the Resource View, select the IDD_WTL_VIEW control, and drag a new button from the Toolbox onto the form.
   b. With the new button selected, change the ID (in the Properties window) to IDD_BN_CLOSE and the Caption to Close.
   c. In the resource.h file, you will see that there is an automatically created #define for IDD_BN_CLOSE.
   d. In WtlWindow.h, add a COMMAND_HANDLER for the BN_CLICKED event for your new Close button.
   e. In the implementation of your command handler, add the following code:

```
GetParent().PostMessage(WM_CLOSE);
```

   f. Compile and run your updated program. Now, the Close button will close the window.