Document/View Architecture in Visual C++ Test and Measurement Applications

Back to Document

As a Visual C++ developer, you probably develop many applications that have a dialog-based architecture. While great for simple applications that do not require customized toolbars and menus, the dialog-based architecture does not meet the criteria for more complex applications that require elaborate user interfaces. National Instruments recommends that you use the document/view architecture to develop more complex applications. This application note leads you through using Measurement Studio Visual C++ class libraries to create a complex measurement application with the document/view architecture.

Table of Contents:

Compare Document/View Architecture with Dialog-Based Architecture

The MFC document/view architecture includes a combination of a document, in which data is stored, and a view, which has privileged access to the data. The document/view architecture separates the storage and maintenance of data from the display of data. This separation facilitates development of applications that display multiple views of a single set of data simultaneously. When the application user changes the data in one view, the architecture handles the task of notifying the other views that they must update to display the modified data.
The following table compares the architectures.

Architecture	Menus	Toolbars	Benefits
Document/View	· System menu similar to the one in dialog-based architecture. · Application menu that includes the following options. - New - Open - Save - Save As - Print - Print Preview - Print Setup - Undo - Cut - Copy	· Fully developed toolbar that includes buttons for all default menu options. · Option to add custom toolbar buttons and organize the buttons. · Includes a status bar at the bottom of the frame.	With the fully developed menu and toolbar in the document/view architecture, you can provide users with an easy-to-use, intuitive application. These expanded menus and toolbars make it easy for users to navigate a complex application.

	- Paste		
Dialog-Based	Minimal system menu that includes options for moving the dialog box, viewing the About information, and exiting the application.	No integrated tools or status bars.	With the dialog-based architecture, you can provide users with a simple, clean interface for less complex applications.

Choose Document/View Architecture for Appropriate Applications

You might choose to create a dialog-based application when the following scenarios are applicable:

- The application user interface is simple.
- You are using code that already integrates data management with data viewing.

When you use the document/view architecture to implement a test and measurement application, you must design the user interface differently than when you use the dialog-based architecture. For example, you must consider how changes to the document affect the view and how changes to the view affect the document. Additionally, you might need to provide different sets of menu items and toolbar buttons for each view.

When you create a Visual C++ application, you can create the application with either a Single Document Interface (SDI) or a Multiple Document Interface (MDI). When you use an application in which you can work with only one document at a time, you are using an application that uses the SDI architecture. For example, Notepad is an SDI application. If you are working in one Notepad document, you must close it before you can work in another Notepad document. When you use an application in which you can work with more than one document at a time, you are using an application that uses the MDI architecture. For example, National Instruments TestStand is an MDI application. You can open several sequence files in one session of TestStand and move between the files, changing and saving information in each one separately, without closing one file to work in another file.

Build an Application Using the Document/View Architecture

This application note leads you through the process of creating an SDI application called Waveform that uses the document/view architecture to plot sine waves, sawtooth waves, square waves, and white noise on a graph. Waveform offers a menu that includes choices associated with each type of waveform data you want to plot on the graph and additional toolbar buttons associated with the menu items. Waveform also includes a graph control on which to display the waveform data.

Waveform uses a Frame class, a Document class, and a View class. Visual C++ provides these classes automatically when you select the option to create an application with document/view architecture in the MFC Application Wizard. These classes make up the functionality of the document/view architecture. The Frame class encompasses the toolbar, menus, and status bar. The Document class contains the data that the application uses. The View class displays the data that the application uses. In Waveform, the Frame class determines what type of waveform data to plot; the Document class determines if the buffer contains data and whether to clear the buffer for new documents; and the View class updates the graph with the appropriate plots.
This application note guides you through the following processes to design the Waveform application:

1. Create an SDI Application with the Document/View Architecture
2. Add the Code--Waveform Example
    - Add the Graph Control
    - Add the Custom Menu
    - Add Buttons to the Toolbar
    - Add the Member Variable and Event Handlers
    - Add the Event Handler Code
    - Add the Code to Update the View
    - Add Advanced Settings

3. Add Code for Saving and Loading Data in the Graph--Serialization Code Example

Refer to Figure 1 for an example of how the finished application appears. Notice the custom menu and toolbar buttons.

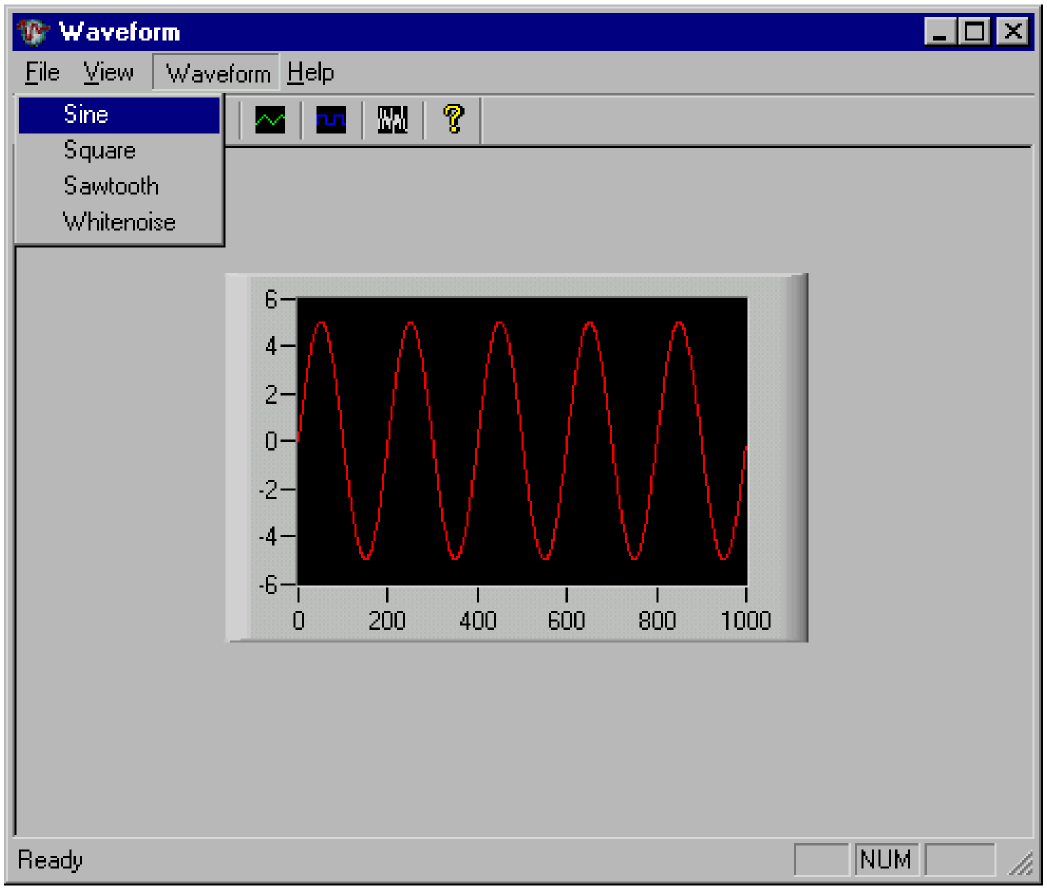

Figure 1. User Interface of Completed Project

Create an SDI Application with the Document/View Architecture

Complete the following steps to create the Waveform application:

1. Open Visual Studio .NET 2003.
2. Select **File»New»Project** to launch the New Project dialog box.
3. In the Project Types pane, expand the **Visual C++ Projects** folder. Select **MFC**.
4. In the Templates pane, select **MFC Application** and enter Waveform in **Name**.
5. Click **OK** to launch the MFC Application Wizard.
6. Select **Application Type** in the wizard.
7. In the Application Type page, select **Single document** and ensure that **Document/View architecture support** is selected.
8. Select **Document Template Strings** in the wizard.
9. In the Document Template Strings page, enter the file extension that you want to use for the waveform data that the application saves and loads. National Instruments recommends that you use a three-letter extension that represents the type of file you are saving and loading. Ensure that you do not use an extension that is already used by other applications, such as doc, xls, or exe.
10. Select **Advanced Features** in the wizard.
11. In the Advance Features page, disable **Printing and print preview** to exclude these options from the application File menu.
12. Select **Generated Classes** in the wizard.
13. In the **Generated classes** list, select the View class. For **Base class**, select **CFormView.** The CFormView class is a View class that uses a dialog resource similar to the one you use when you develop a dialog-based application. You design the user interface in the resource editor the same way you do for a dialog-based application.
14. Click **Finish**. Click **Yes** in the dialog box that opens to notify you that no printing support will be available for CFormView.

15. From the Visual Studio .NET menu, select **Measurement Studio»Add/Remove Class Libraries Wizard** to add Measurement Studio class libraries to the application. Select the **Analysis**, **Common**, and **User Interface** components, then click **Finish**.

Working with the CMainFrame Class

One difference between a dialog-based application and an application built with the document/view architecture is that the document/view application window contains a frame window that surrounds the views. In a dialog-based application, the user interface contains only the dialog. In the document/view architecture, the frame window is derived from the CMainFrame class. In Figure 2, the frame is the area that contains the menus and toolbar at the top and the status bar at the bottom.

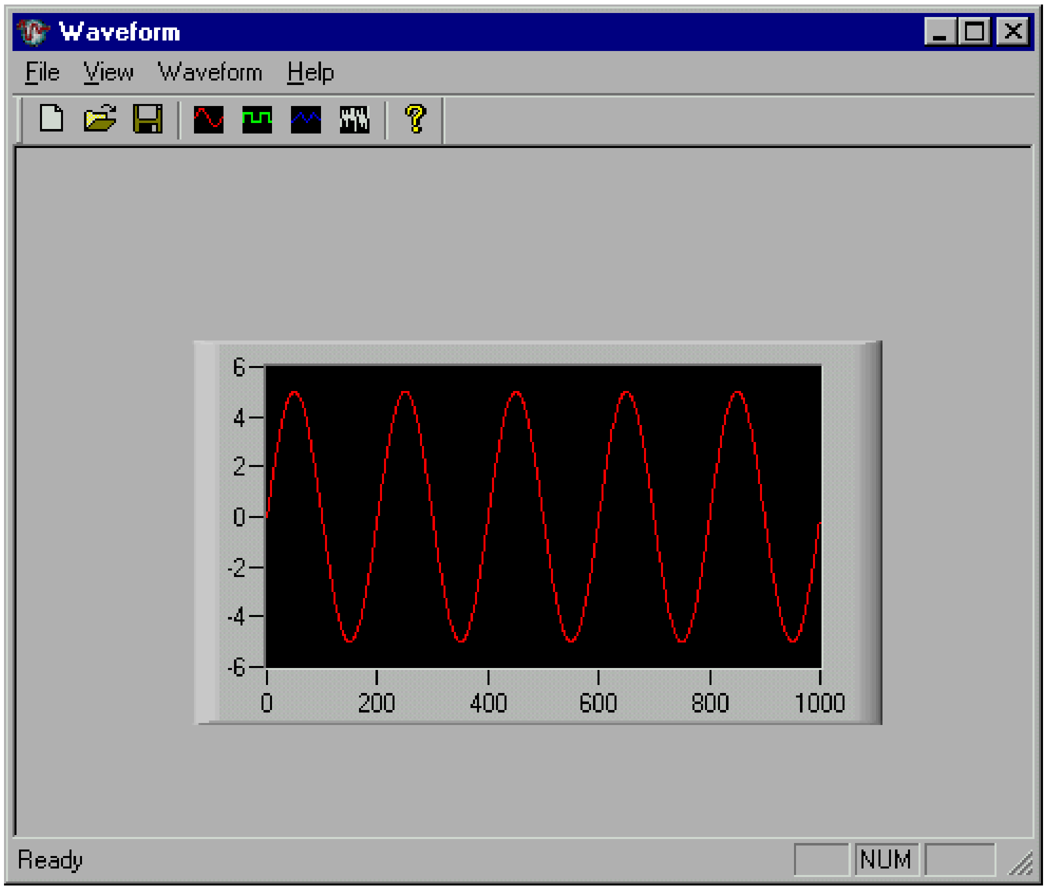

Figure 2. Waveform User Interface

When you create Waveform with the document/view architecture, you must decide if the event handlers for the items in the frame are going to interact with the view and, if so, how. For example, in Figure 2, the toolbar buttons specify the type of waveform to generate and display on the graph, which is in the view. So, the event handlers for these toolbar buttons are in the CFormView-derived class rather than in the CMainFrame-derived class.

Note: Visual C++ creates some of the items in the frame automatically. For example, Visual C++ automatically creates the File, View, and Help menus and generates the event handlers for these items. After the event handlers are generated, you can fill in the code in the .cpp source file for the class in which you declared these functions. In this example, the event handlers are in the WaveformView.cpp file.

Create the User Interface and Add the Code

The following sections include steps to design the user interface, custom menu, and custom toolbars and to add event handlers and event handler code to Waveform.

Add the Graph Control

1. Drag and drop a graph control from the Measurement Studio C++ Tools Tab in the Toolbox to the dialog resource.
2. Resize the graph control as appropriate.

Add the Custom Menu

1. Open the Menu resource for the project.
2. Delete the **Edit** menu by selecting it and pressing <Delete>. Visual Studio displays a warning about removing the pop-up menu and all its contents. Click **OK** to close the warning.
3. Click the empty box on the menu bar to add a menu for the waveform types.
4. Place the cursor in the box on the menu bar that is entitled Type here and enter a name, such as Waveform, for the new menu item.
5. Enter the names for the Waveform menu subitems, as shown in Figure 1.
6. Using the ID_MENUNAME_ITEM naming convention, enter an **ID** for the menu item in the Properties window. For example, if you set up a menu called Waveform to list the waveform types to display on the graph, the ID for the sine wave item might be ID_WAVEFORM_SINE.
7. In the Properties window, for each menu item's **Prompt** property, enter the information you want to appear in the status bar followed by \n and the information you want to appear in the tooltip for the menu item. For example, for the sine wave item, enter Generates a sine wave\nSine Wave.
8. Click and drag the menu to the appropriate location on the menu bar.

Add Buttons to the Toolbar

In this section, you create toolbar buttons and associate each with its corresponding item in the Waveform menu.

1. Open the Toolbar resource for the project.
2. Delete the Print, Cut, Copy, and Paste toolbar buttons. To delete the button, click and drag it off the toolbar. Pressing <Delete**>** deletes only the button image rather than the button.
3. Click the empty box on the toolbar to add a button.
4. In the Properties window, set **ID** by selecting the waveform menu item ID that you want to associate with the button.
5. Using the graphics tools to the left of the toolbar workspace, create an icon for the toolbar button. Use different colors for each of the waveforms. If the graphics tools are not visible, right-click in the toolbar workspace and click **Show Colors Window** to display the Graphics toolbar.
6. Repeat steps 3 through 5 for each Waveform menu item.

Add the Member Variable and Event Handlers

This section includes steps for adding a variable for the graph control that you placed on the dialog resource and adding event handlers for the items that you added to the Waveform menu.

1. Open the dialog resource on which you placed the graph control.
2. Right-click the graph control and select **Add Variable** from the right-click menu to launch the Add Variable wizard. Add a member variable for the graph. Typically, the naming convention is m_name of control, so in this case, specify m_graph. Click **Finish**.
3. Re-open the menu resource for the project.
4. Right-click the **Sine** waveform menu item. Select **Add Event Handler** from the right-click menu to launch the Event Handler wizard.
5. In **Message type**, click **COMMAND**. When you use the **COMMAND** option, the application framework sends a command message to the application to notify it that the user has executed a command.
6. In **Class list**, select the View class. Click **Add and edit** in the wizard.
7. Repeat steps 3 through 6 for each of the menu items that you created.

Add the Event Handler Code

This section includes steps to add the event handler code for each Waveform menu item.

1. Open the Document class .h file and declare a CNiReal64Vector variable named m_wave in the //Attributes section. You use m_wave in the functions that display waveforms on the graph.

2. Open the View class .cpp file and scroll to the bottom of the file to view the skeleton event handlers for the menu items.

3. In the sine wave event handler, declare a double variable named phase and initialize it to zero.

Call the appropriate waveform generation function in the CNiMath class. Pass the appropriate parameters to the waveform function. Use the GetDocument function to get the m_wave variable from the Document class. You also typically pass phase, amplitude, and frequency in addition to m_wave.

4. Use the GetDocument function and the SetModifiedFlag function to specify when the document has changed. Using the SetModifiedFlag ensures that the framework prompts the user to save data when appropriate.

5. Add a statement to call the PlotY function on the graph, using the GetDocument function to get the m_wave value from the Document class.

Refer to the following code sample for the code that you add for the SineWave function:

```
void CWaveformView::OnWaveformSine()
{
  double phase = 0.0;
  CNiMath::SineWave(GetDocument()->m_wave, 1000, phase, 5.0, 0.005);
  GetDocument()->SetModifiedFlag();
  m_graph.PlotY(GetDocument()->m_wave);
}
```

6. Repeat steps 3 through 5 for the square wave and sawtooth wave event handlers. For white noise, pass the seed parameter rather than the frequency parameter and do not declare the phase.

Note: Each function takes different parameters. Review the online help information for each function to determine the parameters and appropriate values to pass.

7. Build and run Waveform.

Note: Because you have not yet added code to handle clearing and/or updating the view, you are unable to use the **New** item in the **File** menu and/or on the toolbar. Refer to the *Add the Code to Update the View* section for information about adding the **File»New** functionality.

Add the Code to Update the View

When you display a waveform on the graph, the information that is plotted is held in a CNiReal64Vector object. To ensure that accurate information is displayed on the graph, you must add code to Waveform to clear the graph and object of old waveform data before displaying new data. Complete the following steps to properly clear the graph and object.

Clear the Object

1. Open the Document class .cpp file and scroll to the OnNewDocument function.
2. In place of the // TODO: add reinitialization code here marker, add m_wave.SetSize(0); to clear the contents of the m_wave object.

Clear the Graph

1. In the Solution Explorer, double-click the View class header file.
2. In the Properties window, click the overrides icon.
3. Select the OnUpdate function in the Properties window to generate skeleton code for the function.
4. Add the following code to the OnUpdate function that you added:

```
void CWaveformView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
  if(GetDocument()->m_wave.GetSize() > 0)
  {
     m_graph.PlotY(GetDocument()->m_wave);
  }
  else
  {
     m_graph.ClearData();
  }
}
```

Add Advanced Settings
You can set the line color of each waveform to match the color of the icons you designed for each custom button on the toolbar.

1. Open the Document class .h file and declare a public CNiColor variable called m_plotColor in the //Attributes section. You use this variable to specify the color of the waveforms you display on the graph.

2. Open the View .cpp file. In the event handler code for a menu item, add a statement to set the value of the m_plotColor variable. Add a statement using the CNiPlot::LineColor property to set the color of the plot to the value of the m_plotColor variable.

The following code excerpt shows how to get the waveform data from the document and plot it on the graph in a particular color. In this example, the code sets the sine wave plot color to red:

```
void CWaveformView::OnWaveformSine()
{
  double phase = 0.0;
  CNiMath::SineWave(GetDocument()->m_wave, 1000, phase, 5.0, 0.005);
  GetDocument()->m_plotColor = CNiColor(255, 0, 0);
  GetDocument()->SetModifiedFlag();
  m_graph.Plots.Item(1).LineColor = GetDocument()->m_plotColor;
  m_graph.PlotY(GetDocument()->m_wave);
}
```

3. Repeat step 2 for each menu item in the Waveform menu.
4. Modify the View class OnUpdate function to set the color of the plot to the value of the Document class m_plotColor variable before the call to PlotY.

Add Code for Saving and Loading Data in the Graph
Ideally, you want users to save waveform data and load it in the graph at a later time. Use serialization functions or streaming operators to handle the input and output functionality required to save and load data in the graph. Because you used the document/view architecture, the Save and Open functionality is built into the Waveform application. To enable this functionality, you must modify the Document Serialize function as shown in the following example:

```
void CWaveformDoc::Serialize(CArchive& ar)
{
  if(ar.IsStoring())
  {
     m_wave.Serialize(ar);
     ar << m_plotColor;
  }
  else
  {
     m_wave.Serialize(ar);
     ar >> m_plotColor;
     UpdateAllViews(NULL);
  }
```

```
    }
```

The code sample above uses the CArchive object, which specifies if the serialization operation is an input operation or an output operation. If the operation is an output operation, this function writes the values in m_wave and the value of m_plotColor into the archive. If the operation is an input operation, this function extracts the values of m_wave and m_plotColor from the archive and writes the values into m_wave and m_plotColor.

Finished Application

You have created an SDI application that uses the document/view architecture to plot several different types of waveforms, in different colors, on a graph. Waveform uses the Frame, Document, and View classes to efficiently maintain and update the waveform data and display the data in the view.

Build and run Waveform, then complete the following steps to test the application:

1. Use each of the menu items and toolbar buttons to ensure that they generate and plot the correct waveform in the correct color.

2. Save and load waveform files to ensure that the **Save** and **Open** items work correctly.

3. Close the application without saving to ensure that Waveform prompts you to save the data.

To demonstrate the benefits of using the document/view architecture for the Waveform application, the following table shows a side-by-side comparison of creating Waveform using dialog-based architecture and document/view architecture.

Dialog-Based	**Document/View**	**Notes**
No built-in functionality to save and load data. You have to write the code to support this functionality.	Built-in functionality to save and load data.	Using the document/view architecture saves you time and effort in providing the code to save and load data.
Forced to modify existing view to handle additional data in the document. Might lead to a cluttered, difficult-to-use interface.	Ability to create additional views to handle additional data in the document.	Using the document/view architecture helps you provide users with a clean, easy-to-use interface that displays data efficiently.
To prevent the user interface from becoming cluttered and difficult to use, you must limit the amount and/or types of data you add to the document.	Ability to add as much data to the document as you need. Because you can add additional views to handle additional data, you are not limited to the type of data you can add to the document.	Using the document/view architecture gives you the opportunity to update an application easily and efficiently. Because you can add data to the document without affecting the original view or views, you can easily update the document without reorganizing the view. Instead, you just create a new view to handle the new data.

My Profile | Privacy | Legal | Contact NI © 2005 National Instruments Corporation. All rights reserved.