InstrumentLab 5.0

Visual C++ Quick Start

www.openwire.org
www.mitov.com

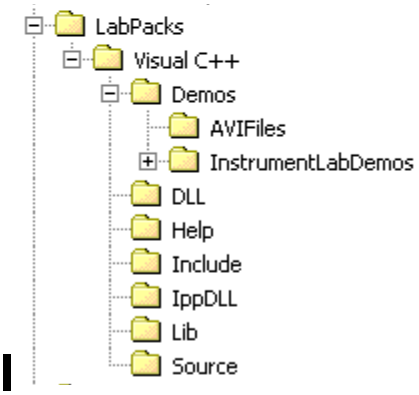
Index

Installation

InstrumentLab comes with an installation program. Just start the installation by double-clicking on the Setup.exe file and follow the installation instructions.

Where is InstrumentLab

After the installation InstrumentLab is located under a single root directory. The default location is C:\Program Files\ LabPacks. During the installation the user has the option to select alternative directory.
Here is how the directory structure should look like after the installation:



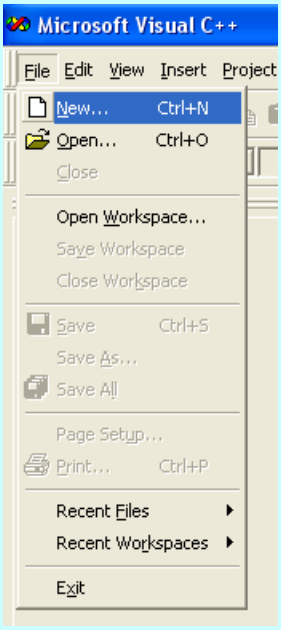
Under the InstrumentLabDemos directory are located the demo files. The help files and the documentation are located under the Help directory. The DLL directory contains the redistributable DLL files. The header files needed for your projects are located under the Include directory. The Release and Debug version of the library is located under the Lib directory.
It is a great idea to start by opening and compiling the demo files. The demo projects ware designed with Visual C++ 6.0. They can be opened and compiled under Visual C++.NET as well, in this case the IDE will create the necessary solution files.

Creating a new InstrumentLab project in Visual C++

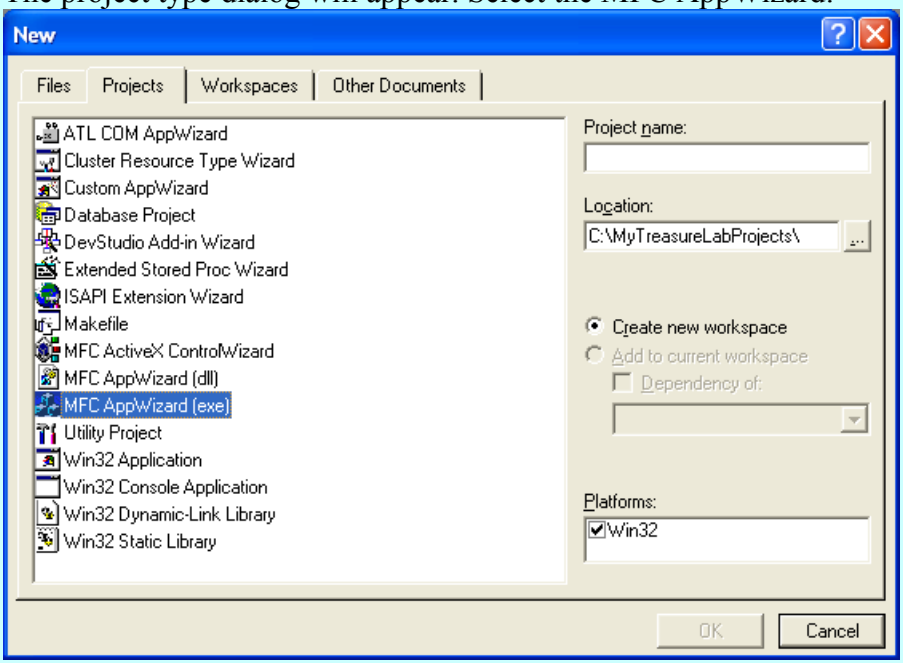
All of the examples in this manual start with creating a MFC Dialog based project. This is not an InstrumentLab requirement, but using the resource editor to design the application makes writing the examples much easier.
The following chapters will assume that you have created the project and will teach you how to add specific InstrumentLab functionality.

Visual C++ 6.0:

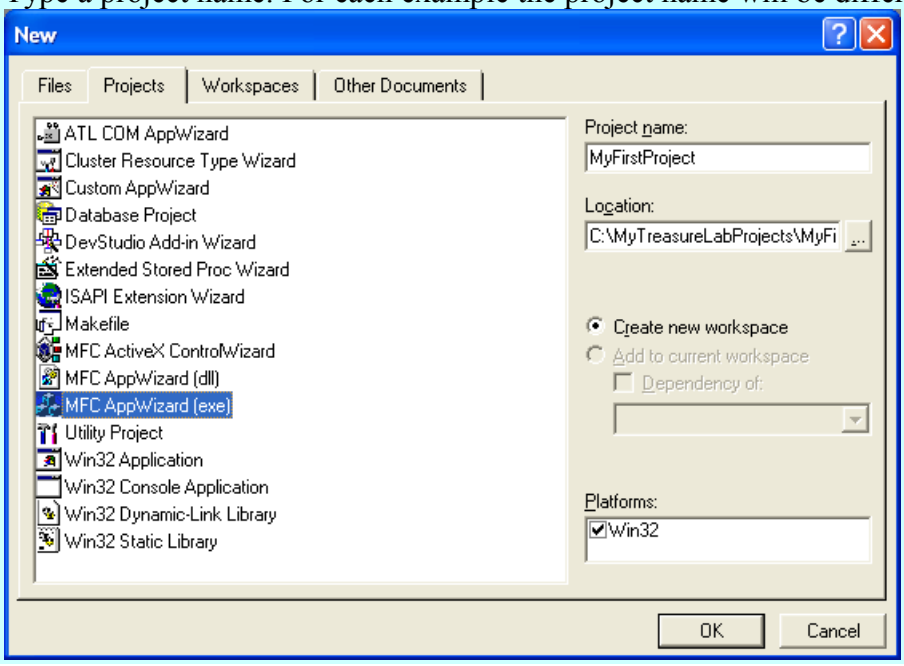
Start by creating a new project.
From the VC++ menu, select | File | New…|



The project type dialog will appear. Select the MFC AppWizard:

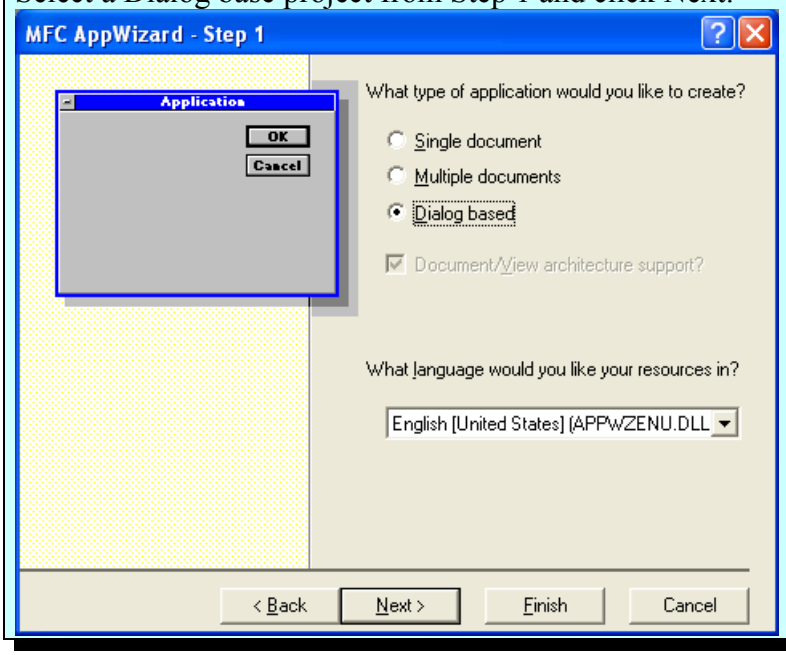


Type a project name. For each example the project name will be different:

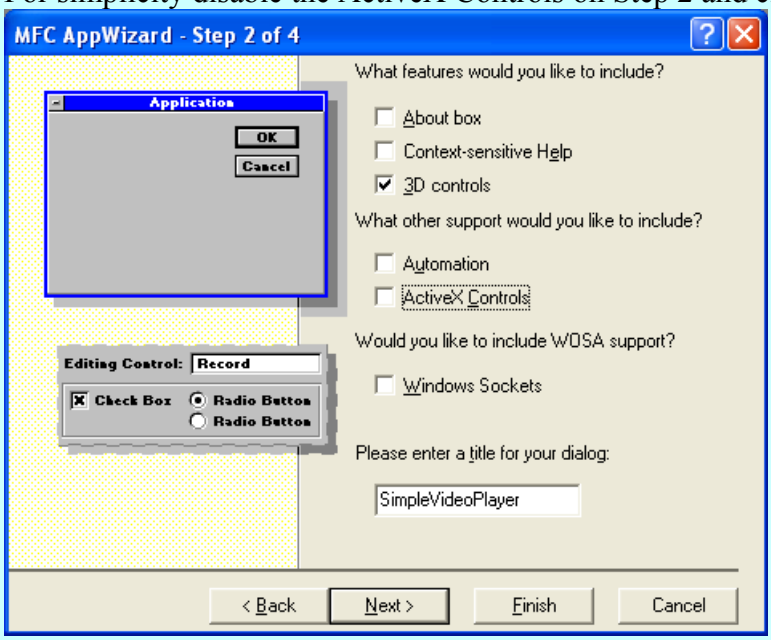


Click OK.

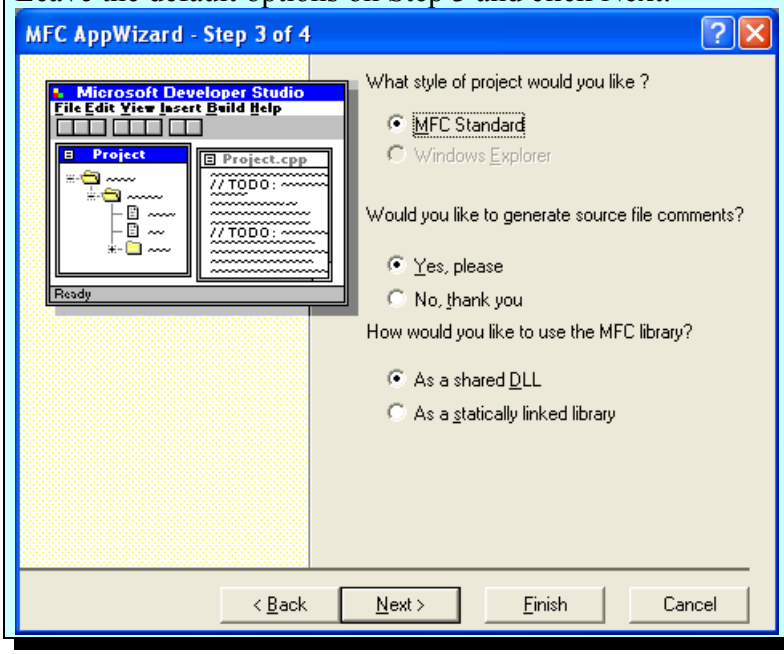
Select a Dialog base project from Step 1 and click Next:



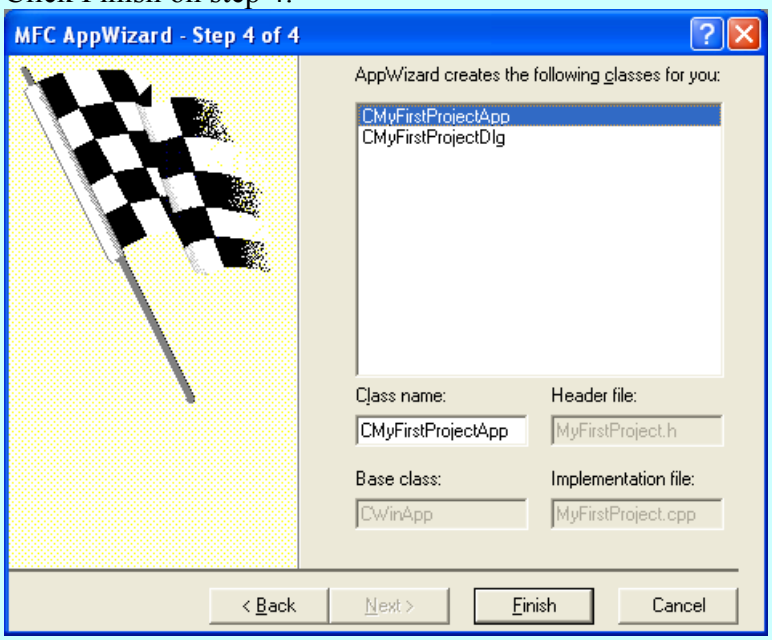
For simplicity disable the ActiveX Controls on Step 2 and click Next:



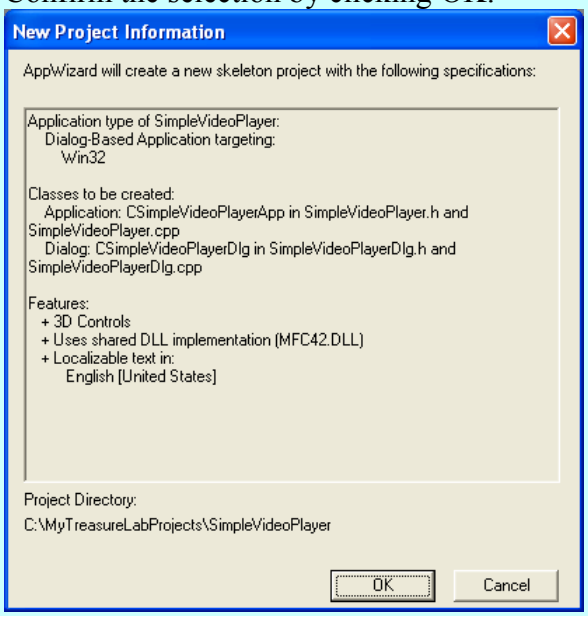
Leave the default options on Step 3 and click Next:



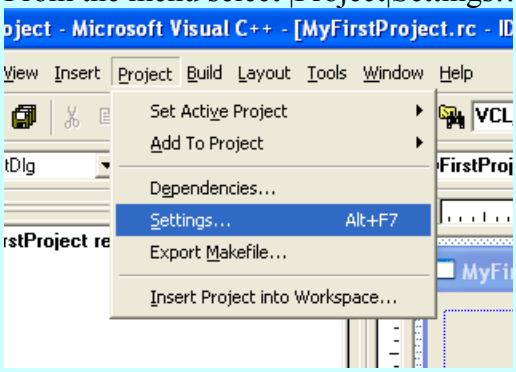
Click Finish on step 4:



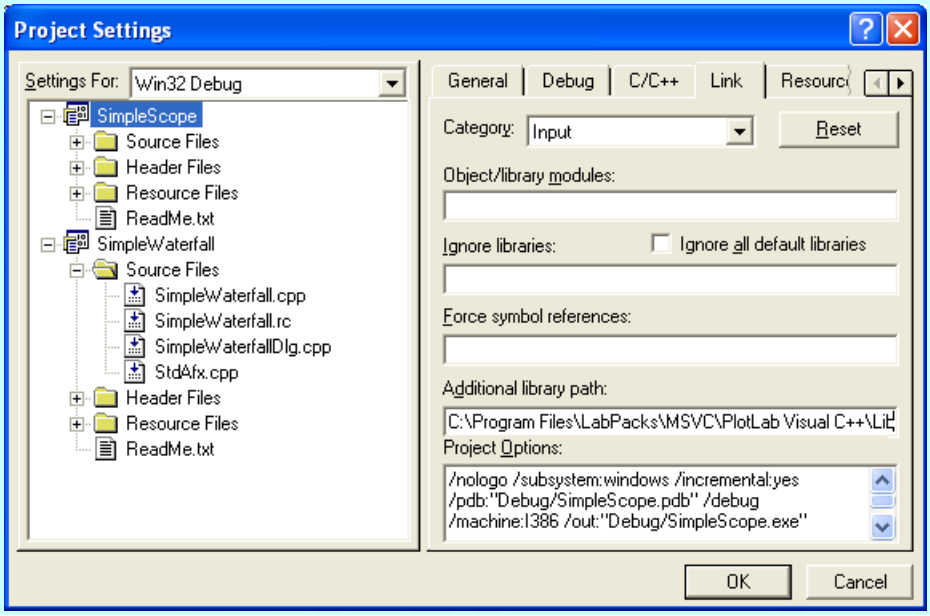
Confirm the selection by clicking OK:



At this point you should have a new project created.
From the menu select |Project|Settings…| :

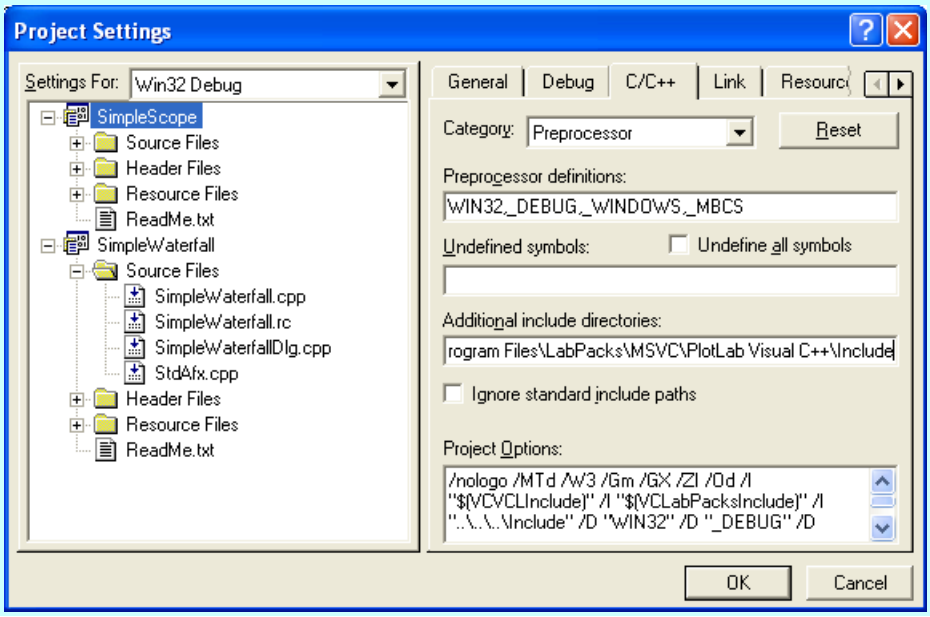


In the Project Settings dialog select the | Link | tab and in the “. Switch to the “Input” cathegory. In the “Additional library path:” edit box add the path to the library files. If you have followed the default installation it should be located at C:\Program Files\LabPacks\Visual C++\Lib:



Switch to the |C/C++| tab.
In the “Additional include directories:” edit box add the path to the header files. If you have followed the default installation they should be located at C:\Program

Files\LabPacks\Visual C++\Include:

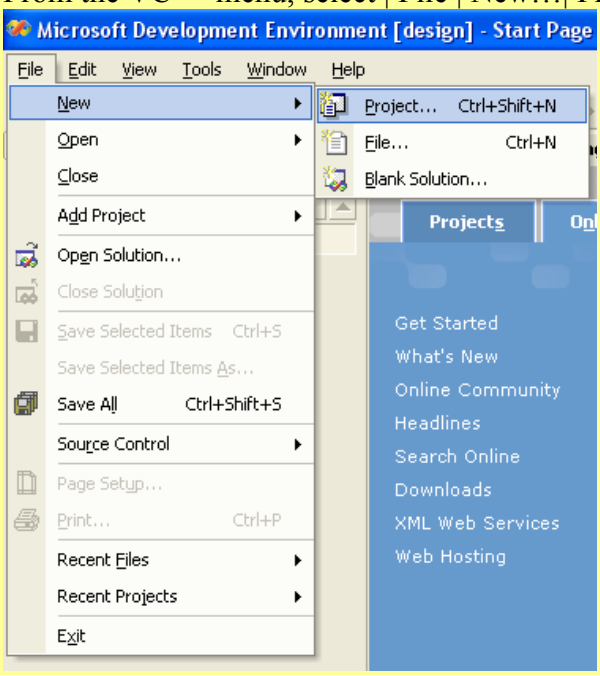


Click OK.

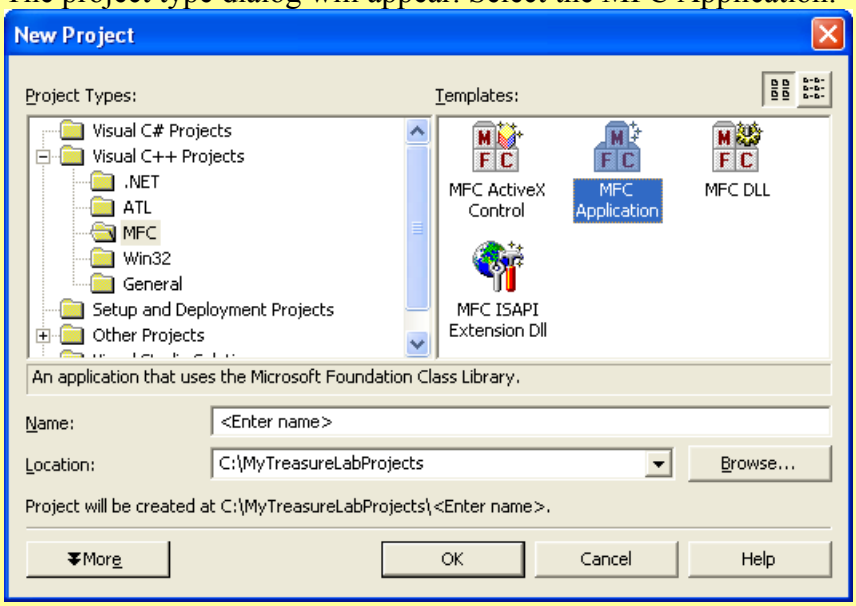
Now you have fully configured project, and you can start writing the actual code.

Visual C++ 2003:

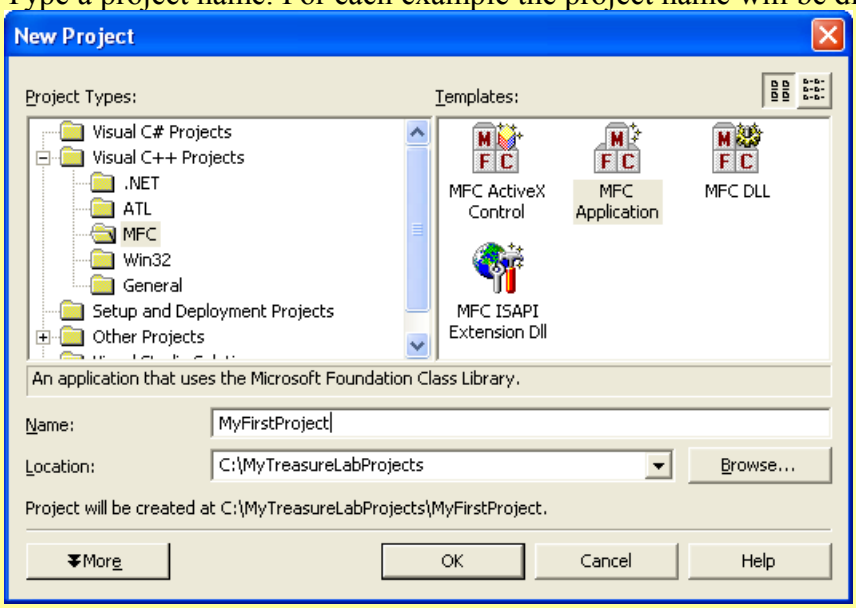
Start by creating a new project.
From the VC++ menu, select | File | New…| Project… |



The project type dialog will appear. Select the MFC Application:

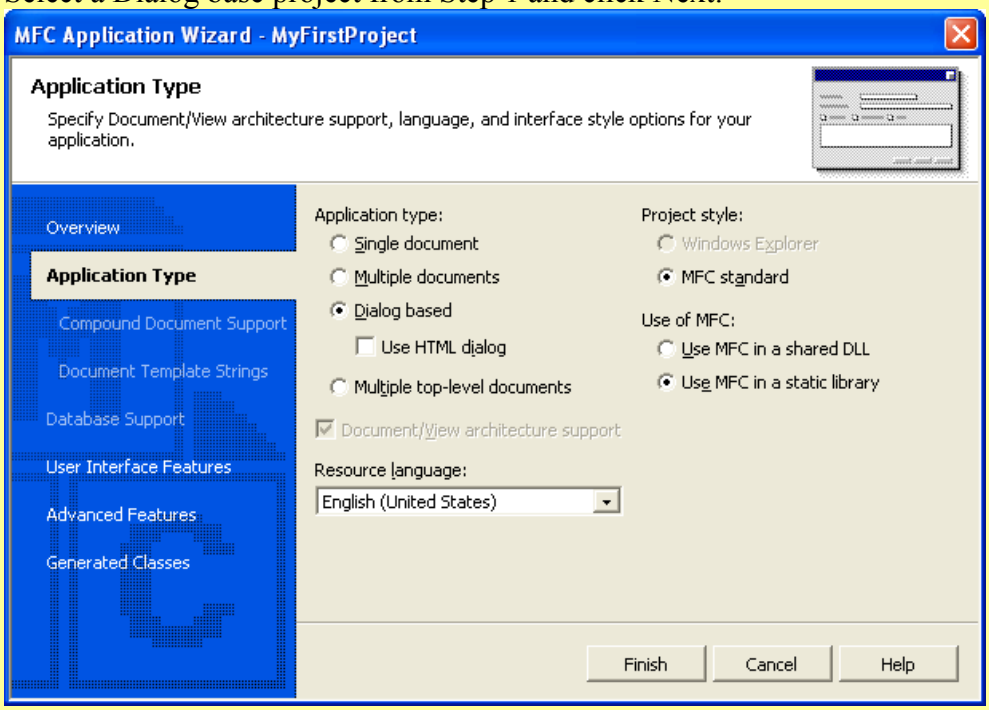


Type a project name. For each example the project name will be different:

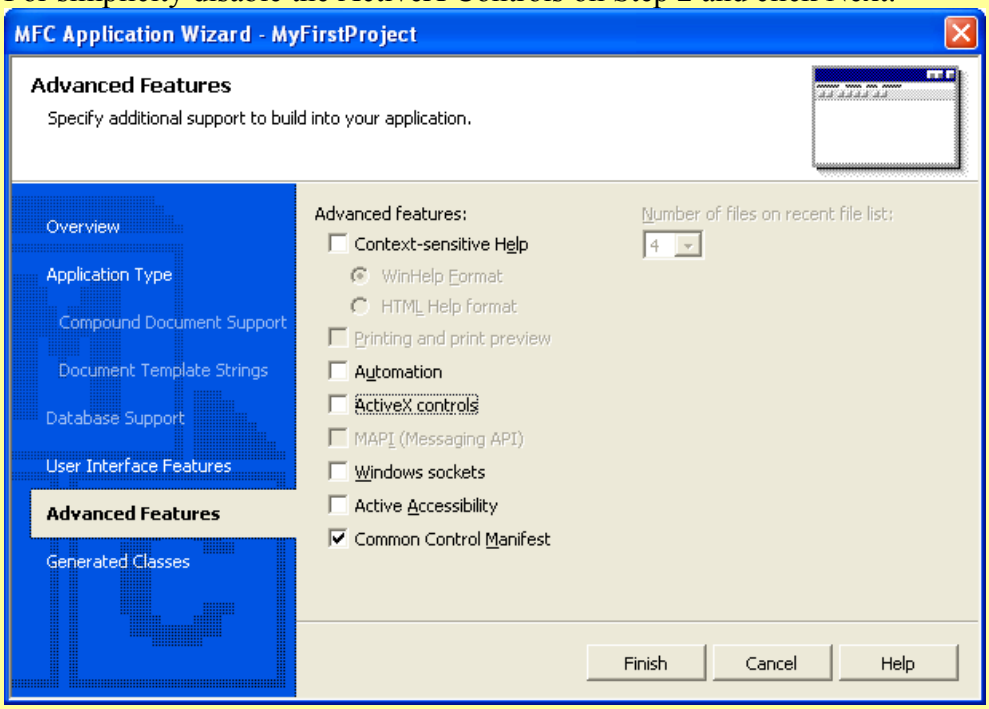


Click OK.

Select a Dialog base project from Step 1 and click Next:

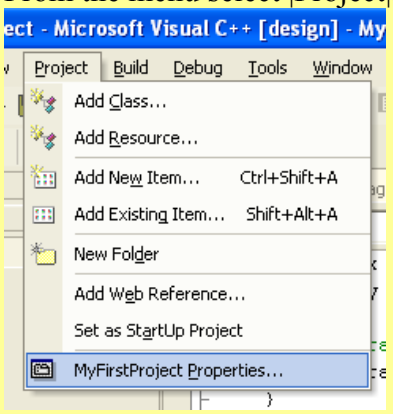


For simplicity disable the ActiveX Controls on Step 2 and click Next:

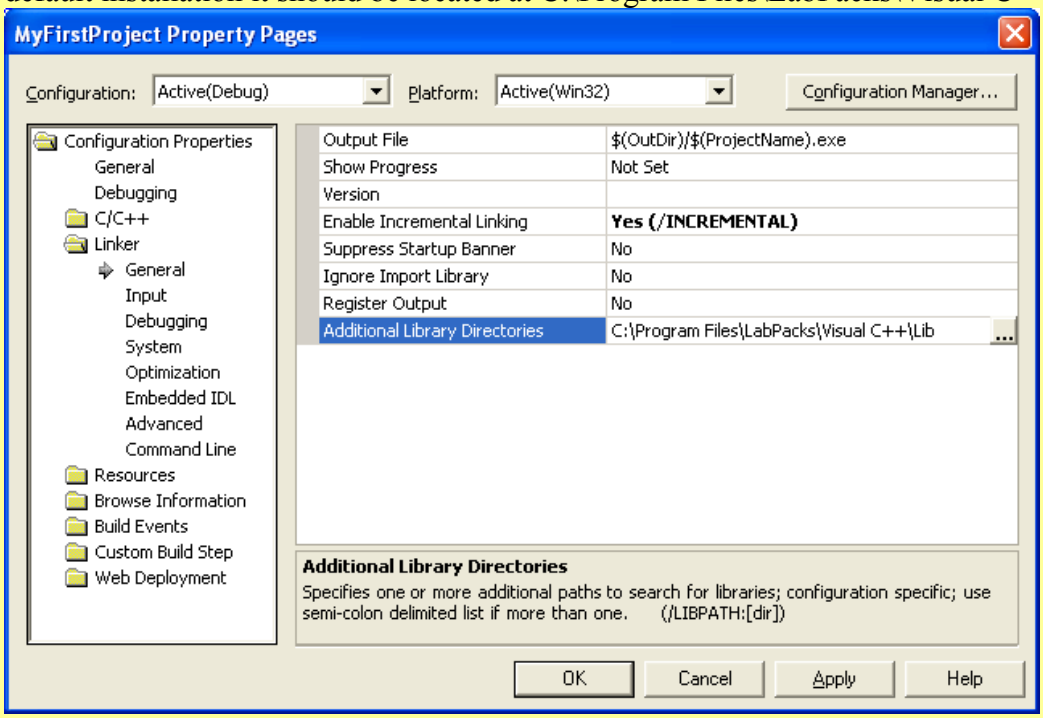


Click Finish.

At this point you should have a new project created.
From the menu select |Project|Settings…| :

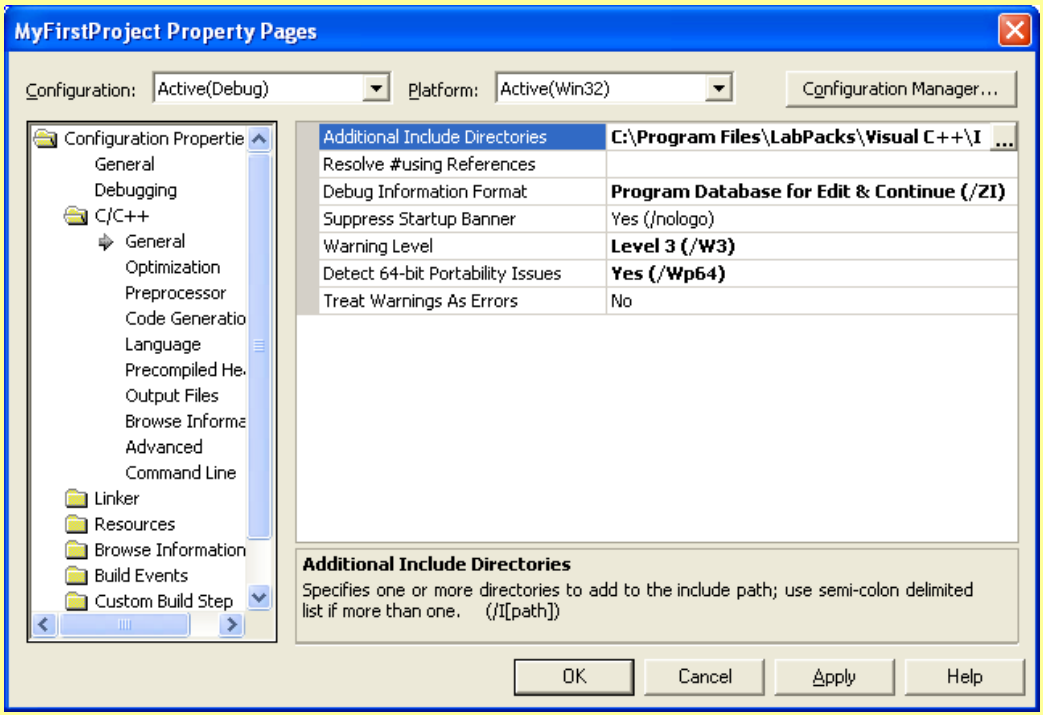


In the Project Property dialog select the Linker General page. In the "Additional library directories:" edit box add the path to the library files. If you have followed the default installation it should be located at C:\Program Files\LabPacks\Visual C++\Lib:



Switch to the C/C++ General page.
In the "Additional include directories:" edit box add the path to the header files. If you have followed the default installation they should be located at C:\Program

Files\LabPacks\Visual C++\Include:



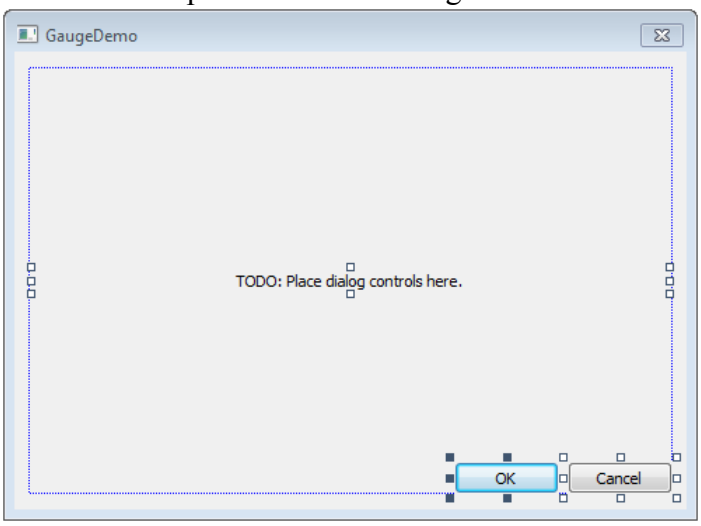
Click OK.

Now you have fully configured project, and you can start writing the actual code.

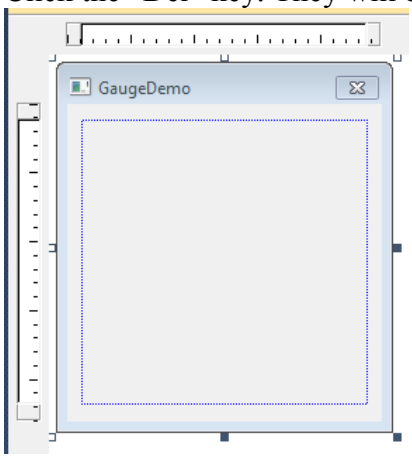
Creating application with angular gauge

Create and setup a new project named GaugeDemo as described in the “Creating a new InstrumentLab project in Visual C++” chapter.

Select the components on the dialog form:



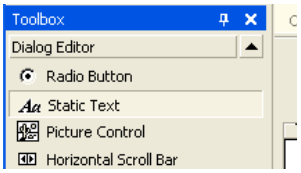
Click the “Del” key. They will be deleted from the form:



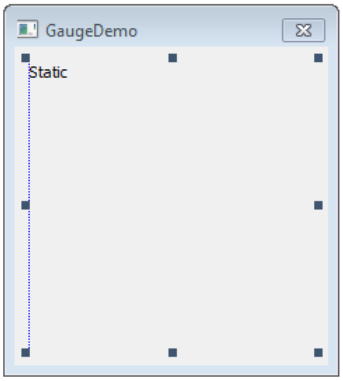
From the controls toolbar select a “Static Text” control:

VC++ 6:

VC++ 2003/2005:

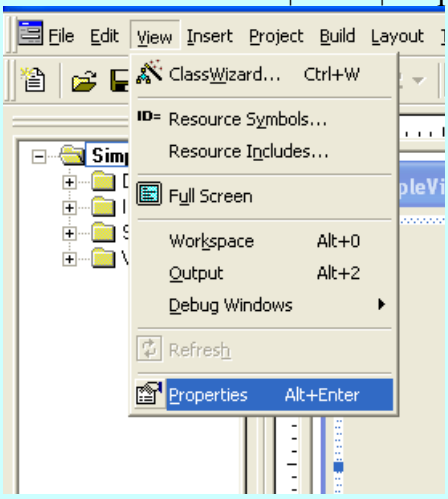


Place the control on the form:

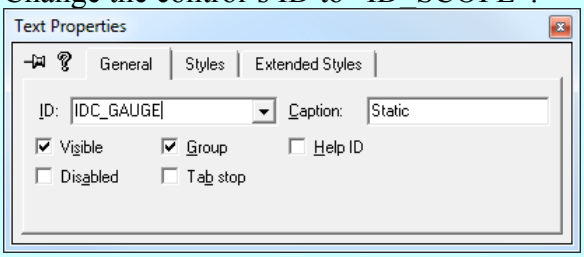


In Visual C++ 6.0:

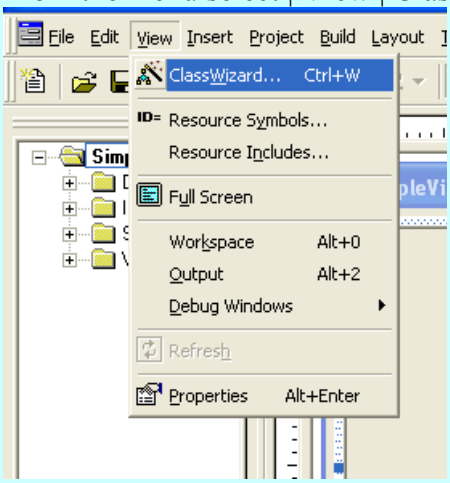
From the menu select | View | Properties |:



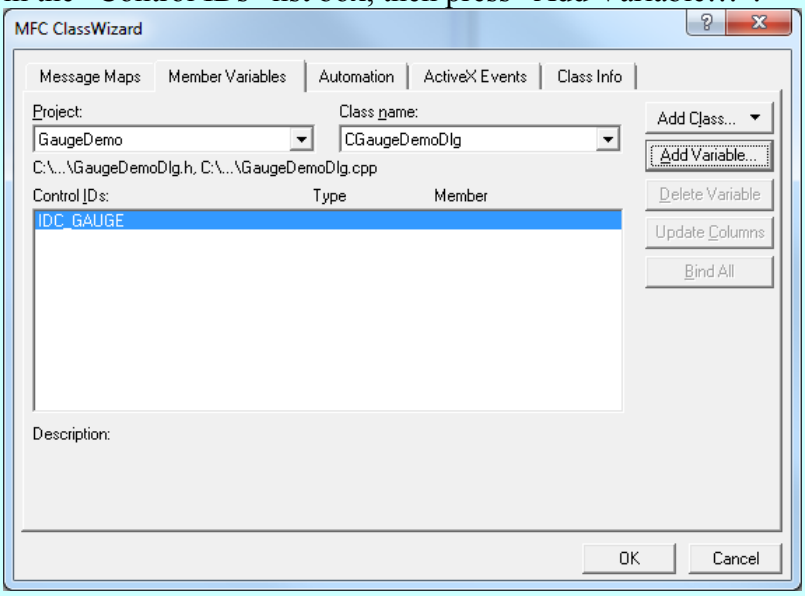
Change the control's ID to "ID_SCOPE":



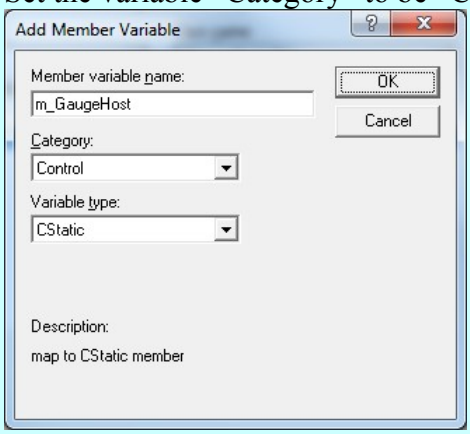
From the menu select | View | ClassWizard… |:



In the "ClassWizard" select the "Member Variables" tab and select the "ID_GAUGE" in the "Control IDs" list box, then press "Add Variable…":

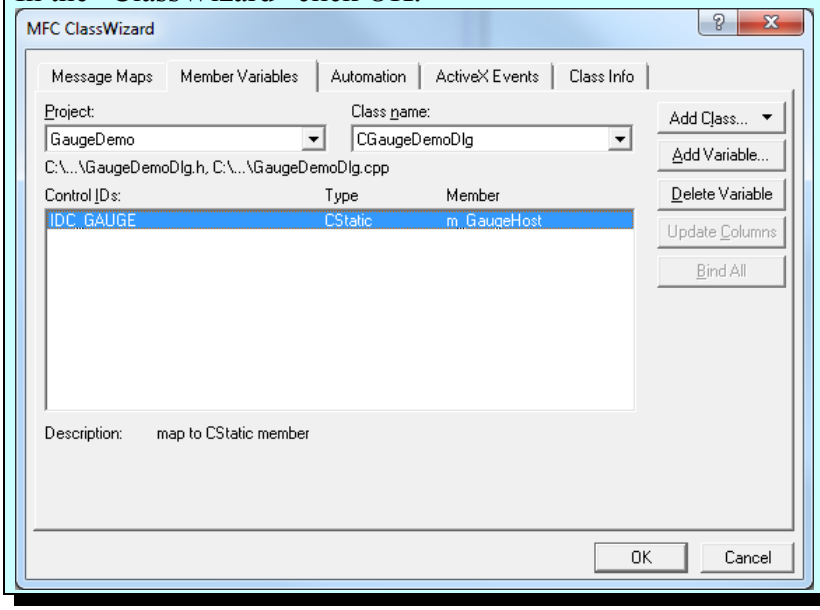


Set the variable “Category” to be “Control”, and set the name to be m_GaugeHost:



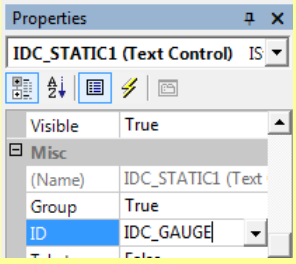
Click OK.

In the “ClassWizard” click OK:

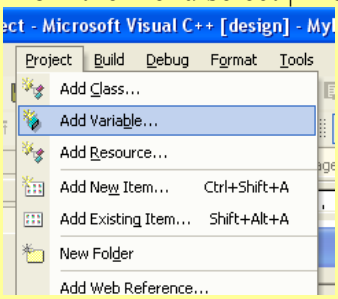


In Visual C++ 2003/2005:

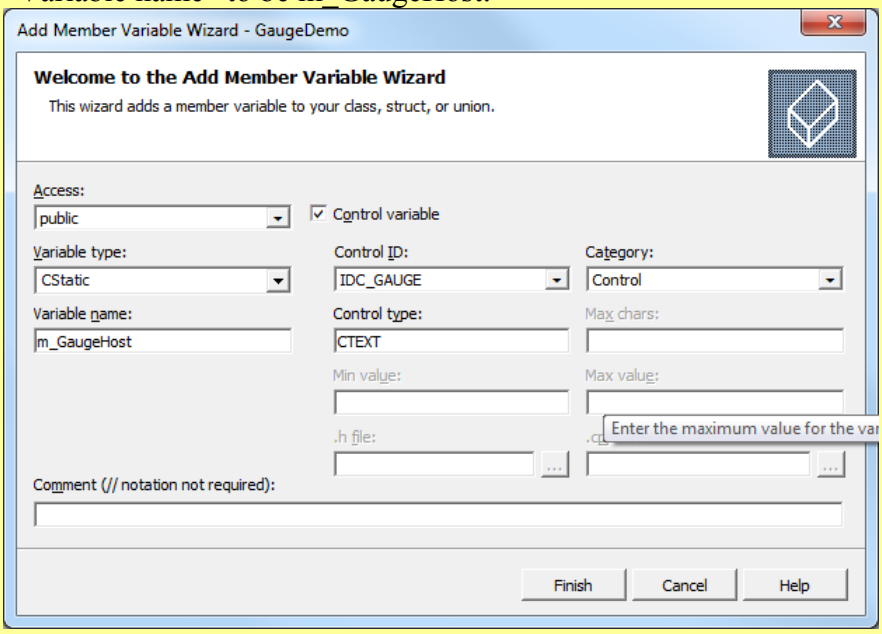
Change the control’s ID to “ID_GAUGE”:



From the menu select | Project | Add Variable… |:



In the "Add Member Variable Wizard" Set the "Variable type" to CStatic, and set the "Variable name" to be m_GaugeHost:



Click Finish.

Add the highlighted lines in the GaugeDemoDlg.h header file:

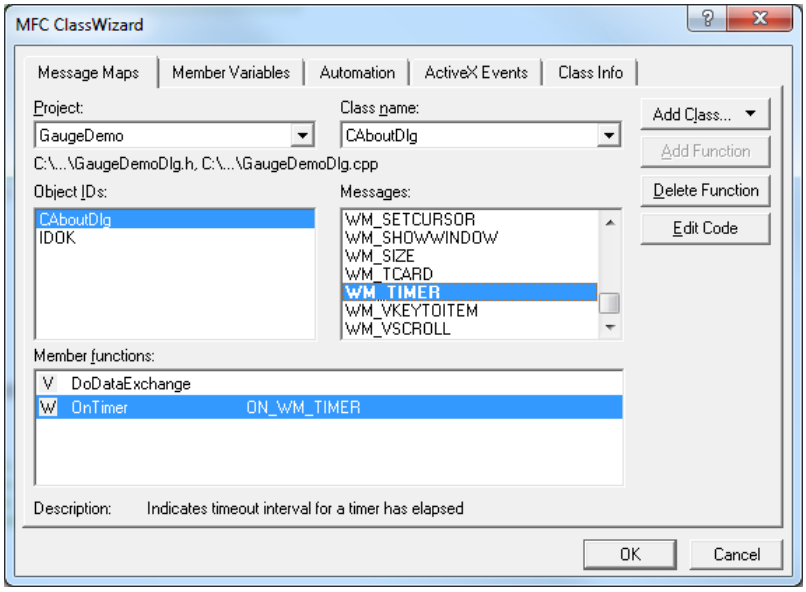
```
// GaugeDemoDlg.h : header file
//

#pragma once
#include "afxwin.h"

#include <CILAngularGauge.h>
// CGaugeDemoDlg dialog
class CGaugeDemoDlg : public CDialogEx
{
// Construction
public:
	CGaugeDemoDlg(CWnd* pParent = NULL);	// standard
constructor
```

```
// Dialog Data
	enum { IDD = IDD_GAUGEDEMO_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support


// Implementation
protected:
	CTILAngularGauge Gauge;

protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	CStatic m_GaugeHost;
	afx_msg void OnTimer(UINT_PTR nIDEvent);
};
```

Add the highlighted lines in the CScopeDemoDlg::OnInitDialog of the ScopeDemoDlg.cpp source file:

```
BOOL CGaugeDemoDlg::OnInitDialog()
{
	CDialogEx::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		BOOL bNameValid;
		CString strAboutMenu;
		bNameValid = strAboutMenu.LoadString(IDS_ABOUTBOX);
		ASSERT(bNameValid);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}
```

```
      // Set the icon for this dialog.  The framework does this
automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);            // Set big icon
      SetIcon(m_hIcon, FALSE);           // Set small icon

      // TODO: Add extra initialization here
      VCL_InitControls( m_hWnd );
      Gauge.Open( m_GaugeHost.m_hWnd );
      VCL_Loaded();

      // Start data flow.
      SetTimer( 1, 100, NULL );

      return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

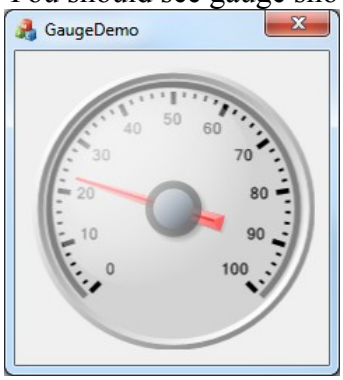


Add the highlighted lines in the OnTimer event handler:

```
void CGaugeDemoDlg::OnTimer(UINT_PTR nIDEvent)
{
      // TODO: Add your message handler code here and/or call default

      Gauge.Value = rand() * 100 / RAND_MAX;

      CDialogEx::OnTimer(nIDEvent);
}
```

Compile and run the application.

You should see gauge showing the random values:



You have just learned how to use InstumentLab gauges.

Adding component elements to a gauge

Open the application already created in the "Creating application with angular gauge" chapter.

Add the highlighted lines in the ScopeDemoDlg.h header file:

```
// GaugeDemoDlg.h : header file
//

#pragma once
#include "afxwin.h"

#include <CILAngularGauge.h>
// CGaugeDemoDlg dialog
class CGaugeDemoDlg : public CDialogEx
{
// Construction
public:
	CGaugeDemoDlg(CWnd* pParent = NULL);	// standard
constructor

// Dialog Data
	enum { IDD = IDD_GAUGEDEMO_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support


// Implementation
protected:
	CTILAngularGauge Gauge;
	CTILExternalGaugeHandElement  GaugeHand;

protected:
	HICON m_hIcon;

	// Generated message map functions
```

```
	//{{AFX_MSG(CScopeDemoDlg)
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	afx_msg void OnDataButton();
	afx_msg void OnClearButton();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Add the highlighted lines in the CScopeDemoDlg::OnInitDialog of the ScopeDemoDlg.cpp source file:

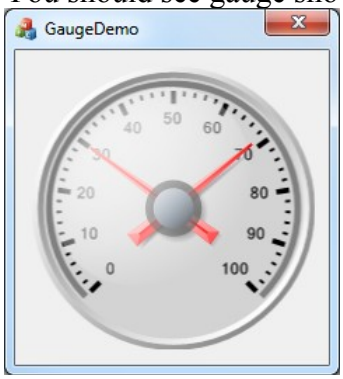
```
BOOL CGaugeDemoDlg::OnInitDialog()
{
	CDialogEx::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		BOOL bNameValid;
		CString strAboutMenu;
		bNameValid = strAboutMenu.LoadString(IDS_ABOUTBOX);
		ASSERT(bNameValid);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this
automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	VCL_InitControls( m_hWnd );
	Gauge.Open( m_GaugeHost.m_hWnd );
	Gauge.Elements.Add( GaugeHand );
	VCL_Loaded();

	// Start data flow.
	SetTimer( 1, 100, NULL );
```

```
	return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

Add the highlighted lines in the OnTimer event handler:

```
void CGaugeDemoDlg::OnTimer(UINT_PTR nIDEvent)
{
	// TODO: Add your message handler code here and/or call default

	Gauge.Value = rand() * 100 / RAND_MAX;
	GaugeHand.Value = rand() * 100 / RAND_MAX;

	CDialogEx::OnTimer(nIDEvent);
}
```

Compile and run the application.
You should see gauge showing the random values with both the hands:



Stop the application.

Add the highlighted lines in the ScopeDemoDlg.h header file:

```
// GaugeDemoDlg.h : header file
//

#pragma once
#include "afxwin.h"

#include <CILAngularGauge.h>
#include <CILThermometer.h>
#include <CILLed.h>

// CGaugeDemoDlg dialog
class CGaugeDemoDlg : public CDialogEx
{
// Construction
public:
	CGaugeDemoDlg(CWnd* pParent = NULL);	// standard
constructor

// Dialog Data
```

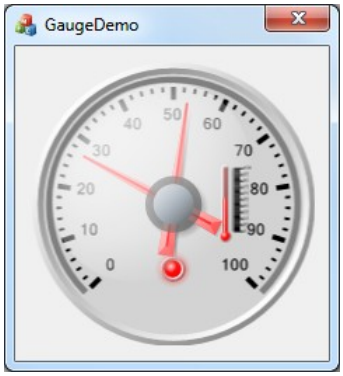
```
	enum { IDD = IDD_GAUGEDEMO_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support


// Implementation
protected:
	CTILAngularGauge Gauge;
	CTILExternalGaugeHandElement  GaugeHand;
	CTILThermometerElement		GaugeThermometerElement;
	CTILLevelDetectLedElement	GaugeLevelDetectLedElement;

protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	CStatic m_GaugeHost;
	afx_msg void OnTimer(UINT_PTR nIDEvent);
};
```

Add the highlighted lines in the CScopeDemoDlg::OnInitDialog of the ScopeDemoDlg.cpp source file:

```
BOOL CGaugeDemoDlg::OnInitDialog()
{
	CDialogEx::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		BOOL bNameValid;
		CString strAboutMenu;
		bNameValid = strAboutMenu.LoadString(IDS_ABOUTBOX);
		ASSERT(bNameValid);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}
```

```
     // Set the icon for this dialog.  The framework does this
automatically
     //  when the application's main window is not a dialog
     SetIcon(m_hIcon, TRUE);             // Set big icon
     SetIcon(m_hIcon, FALSE);            // Set small icon

     // TODO: Add extra initialization here
     VCL_InitControls( m_hWnd );
     Gauge.Open( m_GaugeHost.m_hWnd );
     GaugeThermometerElement.Open( Gauge );
     Gauge.Elements.Add( GaugeThermometerElement );
     GaugeLevelDetectLedElement.Open( Gauge );
     Gauge.Elements.Add( GaugeLevelDetectLedElement );
     GaugeLevelDetectLedElement.Level = 50;
     Gauge.Elements.Add( GaugeHand );
     VCL_Loaded();

     // Start data flow.
     SetTimer( 1, 100, NULL );

     return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

Compile and run the application.
You should see gauge showing the random values with both the hands, the LED will be On when the first hand is above 50 and the Thermometer will show the value of the first hand:



You have just learned how to create composite components in InstumentLab.

Using the TSLCRealBuffer in C++ Builder and Visual C++

The C++ Builder version of the library comes with a powerful data buffer class, called TSLCRealBuffer.
The TSLCRealBuffer is capable of performing basic math operations over the data as well as some basic signal processing functions. The data buffer also uses copy on write algorithm improving dramatically the application performance.

The TSLCRealBuffer is an essential part of the SignalLab generators and filters, but it can be used independently in your code.
You have seen already some examples of using TSLCRealBuffer in the previous chapters. Here we will go into a little bit more details about how TSLCRealBuffer can be used.
In order to use TSLCRealBuffer you must include SLCRealBuffer.h directly or indirectly (trough another include file):

```
#include <SLCRealBuffer.h>
```

Once the file is included you can declare a buffer:
Here is how you can declare a 1024 samples buffer:

```
TSLCRealBuffer Buffer( 1024 );
```

Version 4.0 and up does not require the usage of data access objects. The data objects are now obsolete and have been removed from the library.

You can obtain the current size of a buffer by calling the GetSize method:

```
Int ASize = Buffer.GetSize(); // Obtains the size of the buffers
```

You can resize (change the size of) a buffer:

```
Buffer.Resize( 2048 ); // Changes the size to 2048
```

You can set all of the elements (samples) of the buffer to a value:

```
Buffer.Set( 30 ); // Sets all of the elements to 30.
```

You can access individual elements (samples) in the buffer:

```
Buffer [ 5 ] = 3.7; // Sets the fifth elment to 3.7

Double AValue = Buffer [ 5 ]; // Assigns the fifth element to a
variable
```

You can obtain read, write or modify pointer to the buffer data:

```
const double *data = Buffer.Read() // Starts reading only
double *data = Buffer.Write()// Starts writing only
double *data = Buffer.Modify()// Starts reading and writing
```

Sometimes you need a very fast way of accessing the buffer items. In this case, you can obtain a direct pointer to the internal data buffer. The buffer is based on copy on write technology for high performance. The mechanism is encapsulated inside the buffer, so when working with individual items you don't have to worry about it. If you want to access the internal buffer for speed however, you will have to specify up front if you are planning to modify the data or just to read it. The TSLCRealBuffer has 3 methods for accessing the data Read(), Write(), and Modify (). Read() will return a constant pointer to the data. You should use this method when you don't intend to modify the data and just

need to read it. If you want to create new data from scratch and don't intend to preserve the existing buffer data, use Write(). If you need to modify the data you should use Modify (). Modify () returns a non constant pointer to the data, but often works slower than Read() or Write(). Here are some examples:

```
const double *pcData = Buffer.Read(); // read only data pointer

double Value = *pcData; // OK!
*pcData = 3.5; // Wrong!


double *pData = Buffer.Write(); // generic data pointer

double Value = *pData; // OK!
*pData = 3.5; // OK!
```

You can assign one buffer to another:

```
Buffer1 = Buffer2;
```

You can do basic buffer arithmetic:

```
TSLCRealBuffer Buffer1( 1024 );
TSLCRealBuffer Buffer2( 1024 );
TSLCRealBuffer Buffer3( 1024 );

Buffer1.Set( 20.5 );
Buffer2.Set( 5 );

Buffer3 = Buffer1 + Buffer2;
Buffer3 = Buffer1 - Buffer2;
Buffer3 = Buffer1 * Buffer2;
Buffer3 = Buffer1 / Buffer2;
```

In this example the elements of the Buffer3 will be result of the operation (+,-,* or /) between the corresponding elements of Buffer1 and Buffer2.
You can add, subtract, multiply or divide by constant:

```
// Adds 4.5 to each element of the buffer
Buffer1 = Buffer2 + 4.5;

// Subtracts 4.5 to each element of the buffer
Buffer1 = Buffer2 - 4.5;

// Multiplies the elements by 4.5
Buffer1 = Buffer2 * 4.5;

// Divides the elements by 4.5
Buffer1 = Buffer2 / 4.5;
```

You can do "in place" operations as well:

```
Buffer1 += Buffer2;
Buffer1 += 4.5;
```

```
Buffer1 -= Buffer2;
Buffer1 -= 4.5;

Buffer1 *= Buffer2;
Buffer1 *= 4.5;

Buffer1 /= Buffer2;
Buffer1 /= 4.5;
```

Those are just some of the basic buffer operations provided by SignalLab.
If you are planning to use some of the more advanced features of TSLCRealBuffer please refer to the online help.
SignalLab also provides TSLCComplexBuffer and TSLCIntegerBuffer. They work similar to the TSLCRealBuffer but are intended to be used with Complex and Integer data. For more information on TSLCComplexBuffer and TSLCIntegerBuffer please refer to the online help.

Distributing your application

Once you have finished the development of your application you most likely will need to distribute it to other systems. In order for the built application to work, you will have to include a set of DLL files together with the distribution. The necessary files can be found under the [install path]\DLL directory([install path] is the location where the library was installed).
You can distribute them to the [Windows]\System32 ([Windows]\SysWOW64 in 64 bit Windows) directory, or to the distribution directory of your application([Windows] is the Windows directory - usually C:\WINNT or C:\WINDOWS).

Deploying your application with the IPP DLLs

The application will work, however the performance can be improved by also copying the Intel IPP DLLs provided with the library.
The DLLs are under the [install path]\LabPacks\IppDLL directory([install path] is the location where the library was installed).
In 32 bit Windows to deploy IPP, copy the files to the [Windows]\System32 directory on the target system.
In 64 bit Windows to deploy IPP, copy the files to the [Windows]\SysWOW64 directory on the target system.
[Windows] is the Windows directory - usually C:\WINNT or C:\WINDOWS
This will improve the performance of your application on the target system.