Mach3 Plugin Development Tutorial

Mach3 is the most versatile, supported and cost-effective PC based CNC controller in the world. The addition of the plugin SDK has made it also the most configurable. This document is an introduction to plugin development for Mach3.

Mach3 Plugin Development Tutorial

Introduction

Mach3 is the small shop / home shop industry standard for PC based CNC controllers. It is in use by thousands of advanced hobbyists, semi-pro and pro shops. It has created a revolution all of its own. Now Mach3 plugins will add to this revolution by allowing unrivaled customization of this already rich and powerful tool.

Getting Started

In order to develop Mach3 plugins you need to have the correct C++ development environment and then the Mach3 SDK. The C++ development environment is Visual Studio 2003. Since this product is no longer in production it must be obtained through third party sources. I got mine on eBay. Once you have Visual Studio 2003, it should be installed in the default location on drive C. All the subsequent sections of this tutorial will assume this. The default installation location is:

C:\Program Files\Microsoft Visual Studio .NET 2003

Please verify this before going on with this tutorial.

Installing the Mach3 SDK

First, download the latest development version of Mach3 from http://www.machsupport.com. Be sure to install and test this first. It is highly recommended that you DO NOT try to develop and initially test plugins on a copy of Mach3 that is connected to a live CNC tool. This can be a very dangerous activity.

This tutorial is currently based on Mach3 v2.48 and the SDK2.03.00.zip. I will update these version numbers in appendix A as appropriate

I have my SDK installed in the folder:

C:\CNC\Mach3Development

By unzipping the SDK2.03.00.zip in this folder and making sure the WINZIP option 'Use Folder Names' is checked I get the following folder structure:

C:\CNC\Mach3Development
  SDK2.03.00
    Blank Plugin
    BlankMovement
    GalilPlugin
    JoyStickPlugIn
    MachIncludes
    ncPod Oringial
    ShuttlePro

If all has gone well and everything looks like I have said it should look, then you can proceed to testing the SDK installation. If not either figure out what went wrong or start over, removing files and folders that were located in the wrong places before trying again.

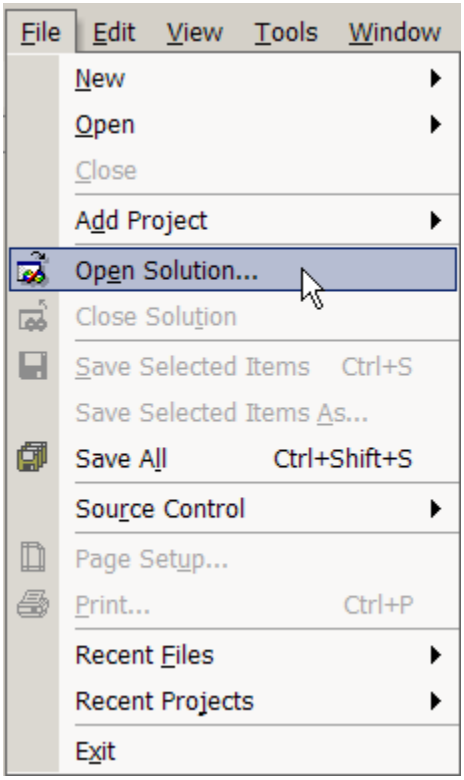
Building the SDK samples

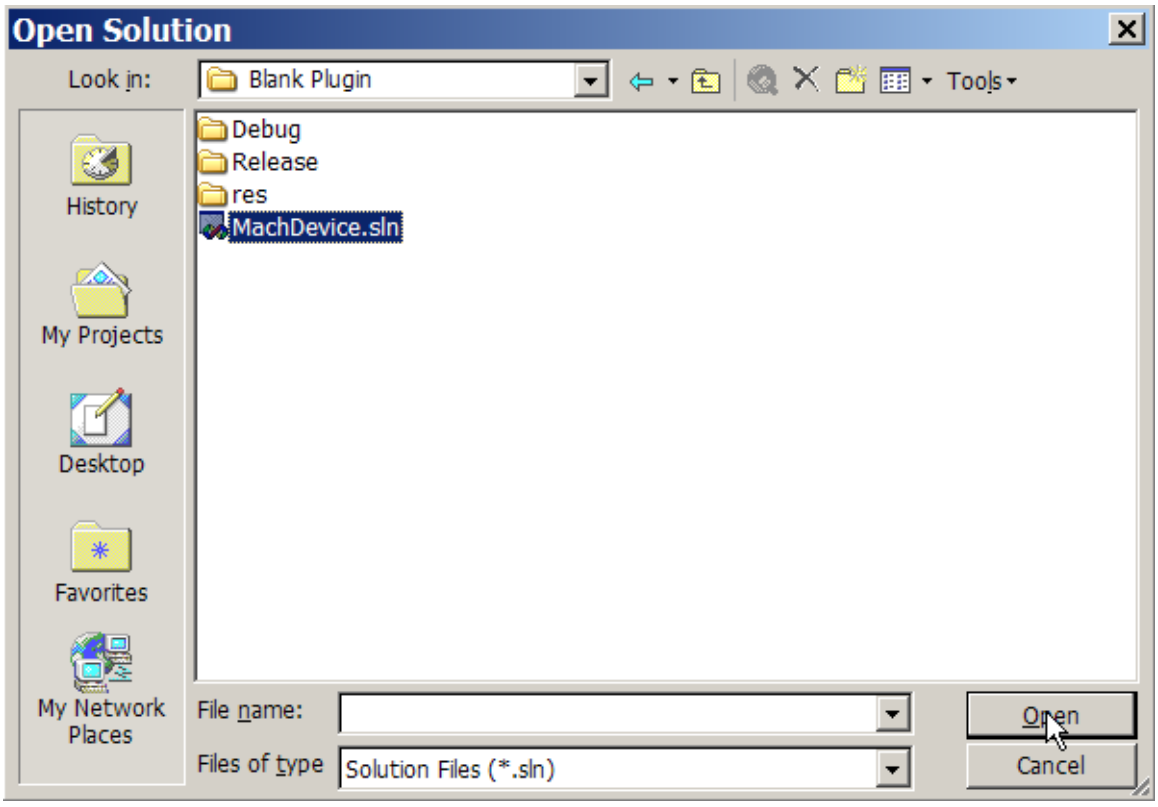
Building the samples that come with the Mach3 SDK is a vital first step that will verify your installation and your ability to build (create) a functional plugin that will work with Mach3.

The first plugin we will build is 'BlankPlugin'. To do this open Visual C++ from the Visual Studio menu and then use the 'File' menu. On this menu you will find 'Open Solution'. If you click on this you can browse for the solution to open. In this case it is:

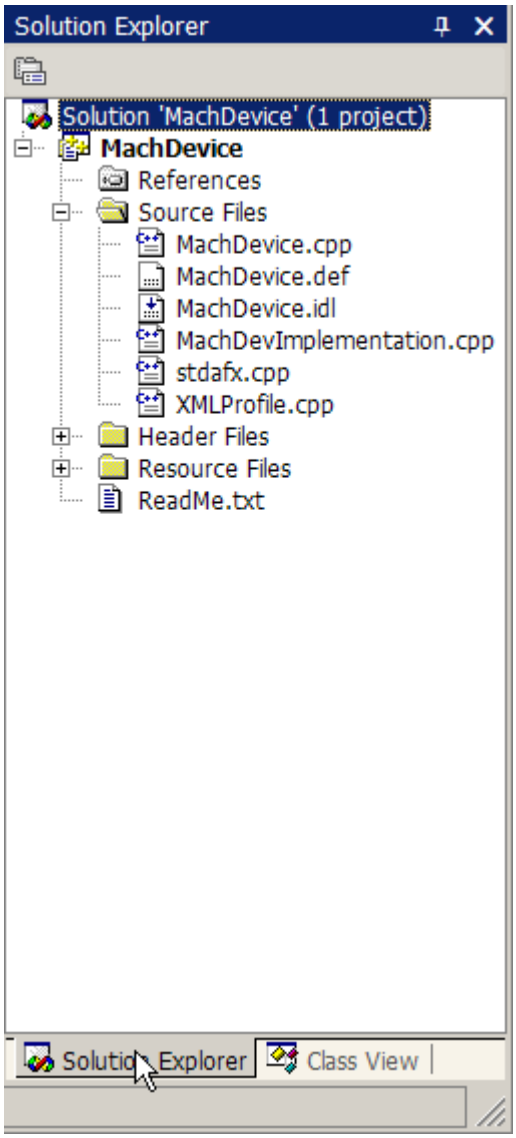
C:\CNC\Mach3Development
  SDK2.03.00
    Blank Plugin

And the solution is named ‘MachDevice.sln’. By selecting this and pressing the ‘Open’ button you will see the main solution explorer view on the right of the screen.

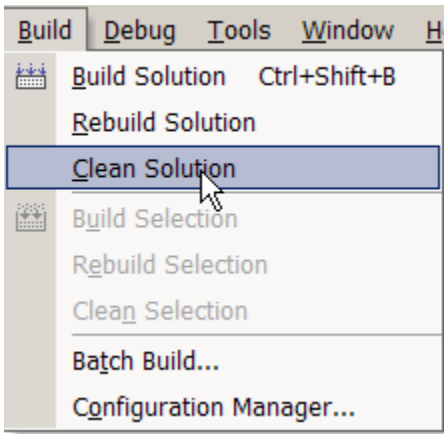




This is the solution explorer screen with the ‘Solution Explorer’ tab selected at the bottom of the screen.



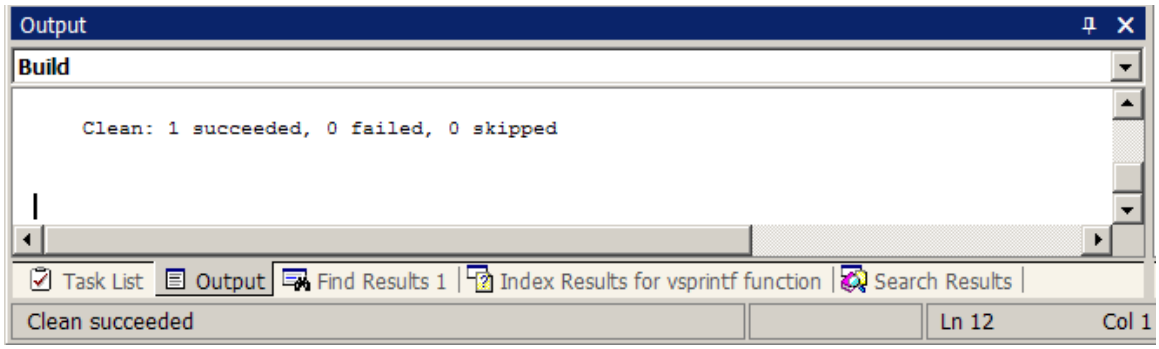
At the top of the screen there is a ‘Build’ menu. On this menu there is a ‘Clean Solution’ option.



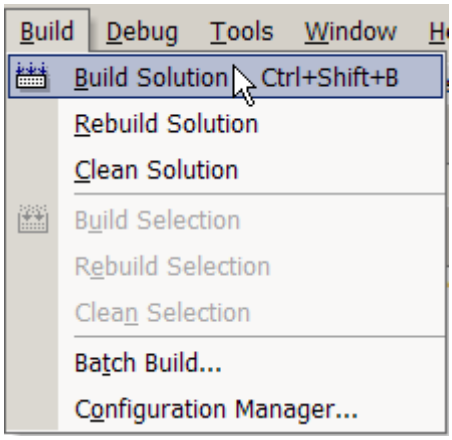
Use this to establish an initial default state for this plugin. It should work a little then display:

```
Clean: 1 succeeded, 0 failed, 0 skipped
```

On the output panel at the bottom of the screen.



Now you can use the ‘Build’ menu command ‘Build Solution’ to create the plugin.



This plugin will be called 'digtizer.dll' and it will be found in this folder:

C:\CNC\Mach3Development
  SDK2.03.00
    Digitizer.dll

To test this first plugin copy it to the C:\Mach3\plugins folder and then start Mach3. Open the 'Config' menu and select 'Config Plugins'. The plugin configuration dialog will display. The 'Digitizer – Digitizing Plugin' entry will have a red X in the first column that is labeled 'Enabled'. Click on this red X with your mouse and it will turn into a green checkmark. This means the plugin will initialize on the next Mach3 load. Stop Mach3 and restart it. Since this plugin is 'blank' it actually does nothing that can be externally observed.

Ok, how do we know the plugin REALLY is running? Let's add something that will display when we click on the 'Config' button in the plugin configuration dialog.

In the Visual C++ project explorer window on the right side of the screen, click on the '+' in the little box by the entry 'Source Files'. These files are the actual C++ code that is compiled into a Mach3 plugin. Locate the source code file 'MachDevImplementation' and double click on it to open the file in the Visual Studio editor. Now find the function 'myConfig'. This is where we will some code to display a Message Box. Here is the code we will add:

```
MessageBox(NULL,"It's ALIVE","Config OK",MB_OK);
```

```
Here is the 'myConfig' function.

void   myConfig (CXMLProfile *DevProf)
// Called to configure the device
// Has read/write access to Mach XML profile to remember what it needs
to.

{

} // myConfig
```

This function is called when the 'Config' button is pressed in the plugin configuration dialog. We will add the 'MessageBox' that displays text to show that all is indeed well.

```
void   myConfig (CXMLProfile *DevProf)
// Called to configure the device
// Has read/write access to Mach XML profile to remember what it needs
to.

{
    MessageBox(NULL,"It's ALIVE","Config OK",MB_OK);

} // myConfig
```

Build this plugin again and copy it to the C:\Mach3\plugins folder. Now when you open the plugin configuration dialog and click on the 'Config' button for this plugin you should see a Message Box with the text "It's ALIVE" with a caption (title bar) that says "Config OK". You now have made your first customized Mach3 plugin!

Your first plugin

To make a plugin of your own you must either use 'BlankPlugin' as a starting point or you can use the prototype file set and instructions that I am providing. This document will only discuss the second method.

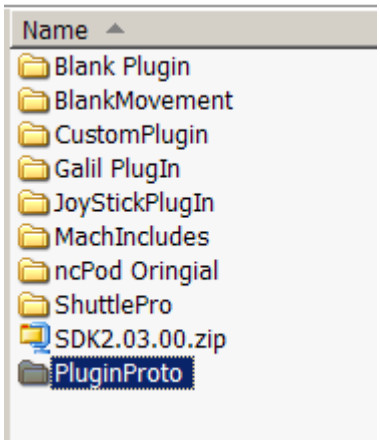
This Mach3 plugin file kit allows you to easily make a new plugin that has a different name, a new GUID and debug messages that can be displayed with the SysInternals DebugView FREE debug message viewer utility.

To make a new Mach3 plugin project use the Visual Studio appwizard to create a 'new project'. This will be a MFC DLL project. This project MUST be located in the directory immediately below the MachIncludes directory. This allows the directory heirarchy that is present in these prototype files to operate correctly.

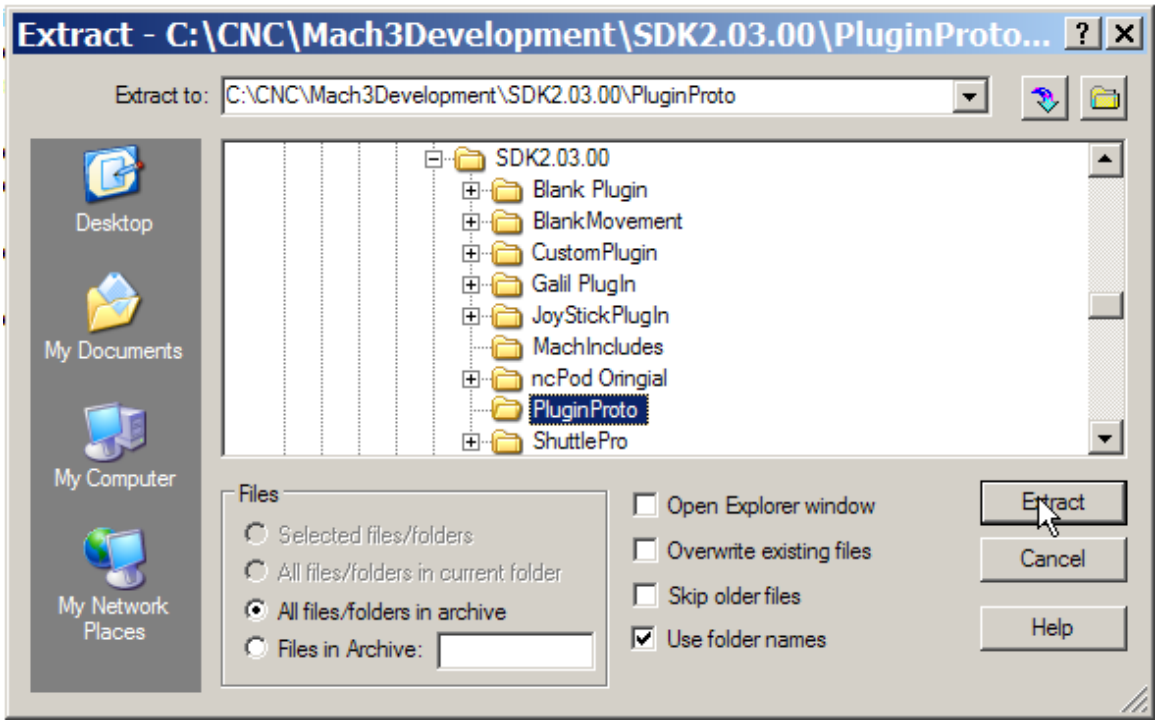
Example:

For Visual Studio 2003:

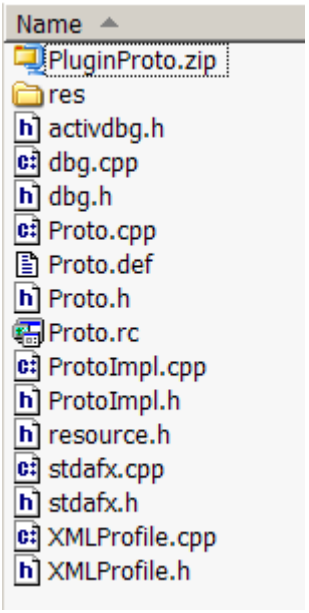
To create a new project 'CustomPlugin' use these steps. First, create a new folder under the SDK installation folder and name it 'PluginProto'.



Now, place the PluginProto.zip file that you downloaded in this folder and unzip it there.



Here are the files that will be extracted to that folder



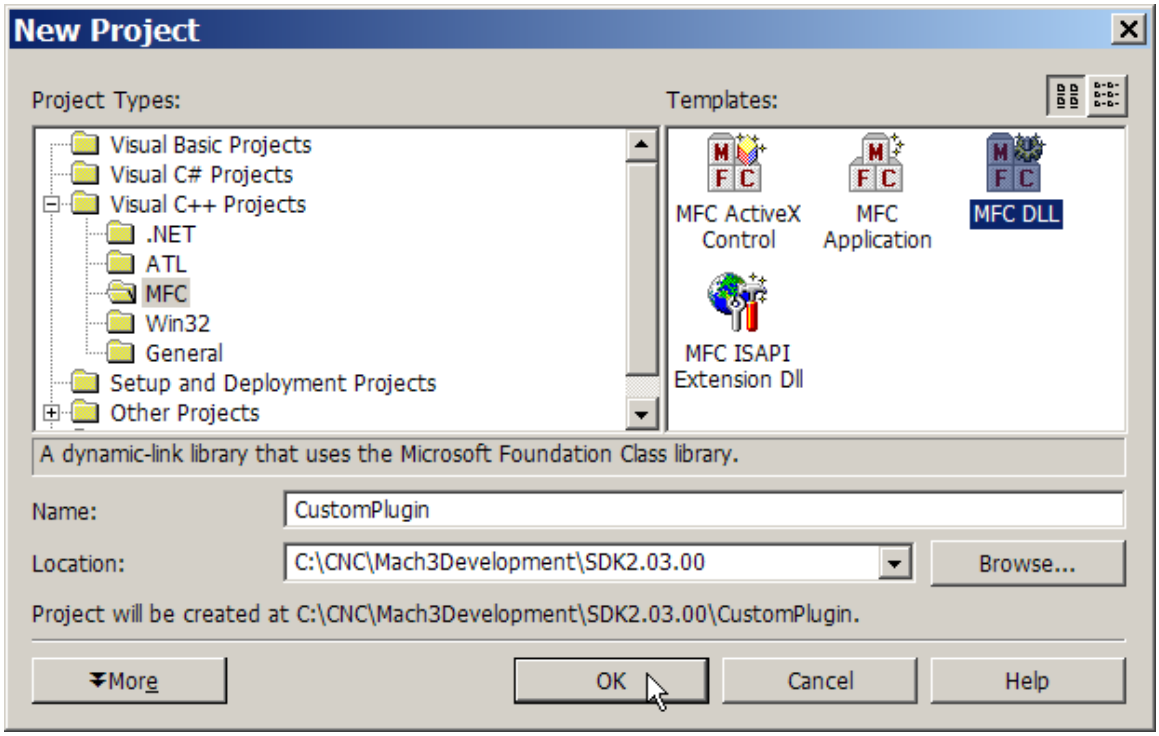
These files will be used later in the creation of this new plugin. WARNING, be sure that the tool that you use to ‘unzip’ this ZIP archive will ‘use folder names’ and create the ‘res’ folder. This is important for the plugin archirecture. I use WINZIP for all of my work.

Creating the New Plugin DLL

Now we will start creating the new plugin project that will be a unique and original plugin for Mach3.

In Visual Studio 2003 use File->New->Project and select

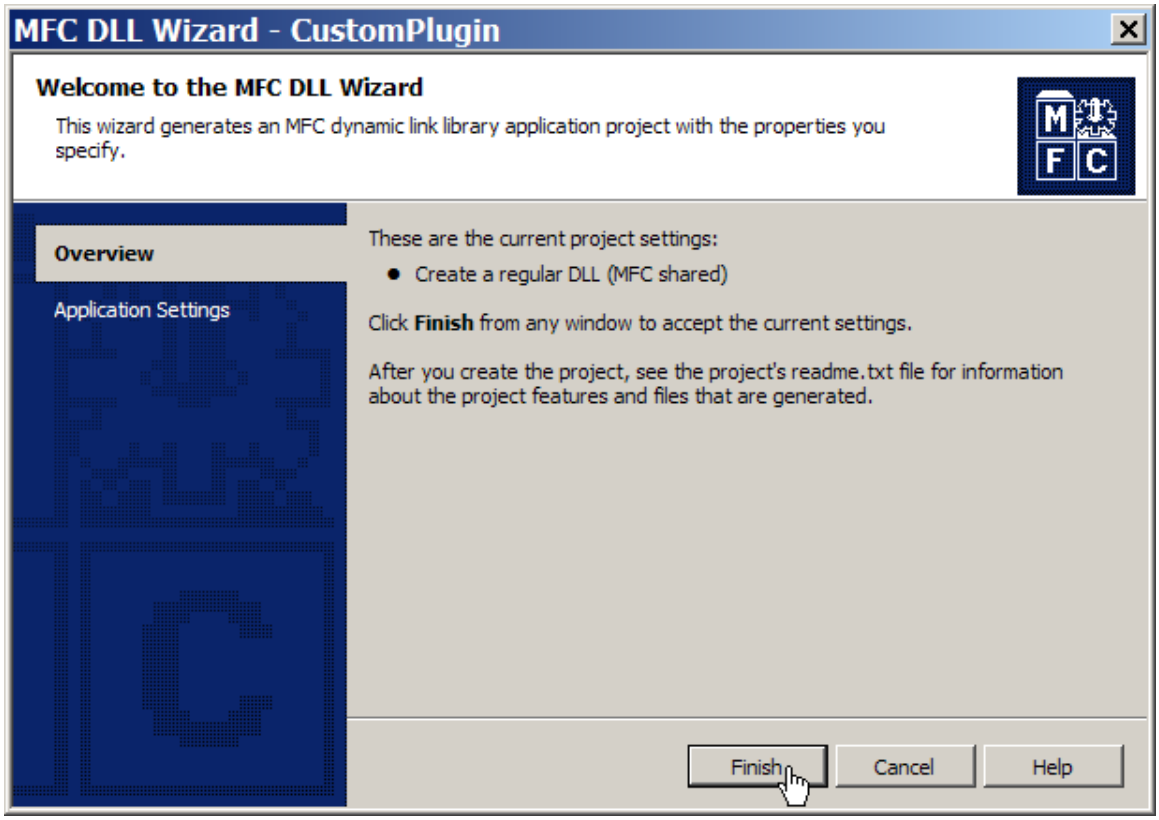
Visual C++ Projects->MFC->MFC DLL



Then browse for the directory where the Mach3 SDK was installed. In this case it is C:\CNC\Mach3Development\SDK2.03.00

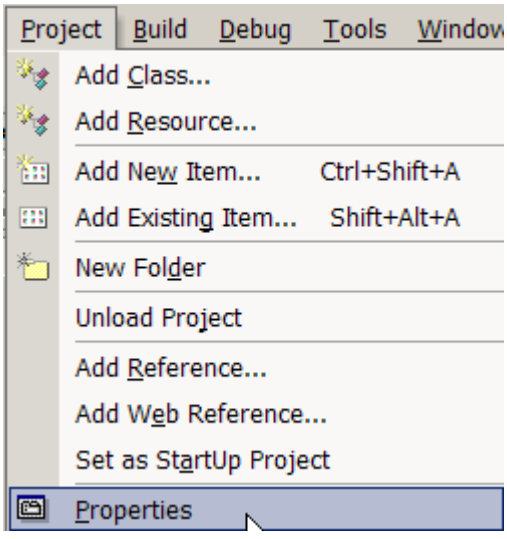
Enter the project name that you want created. In this case it is CustomPlugin. Press the 'Ok' button and the project files in a new solution will be created.

Press the 'Finish' button to complete the project and solution creation.



The MFC DLL AppWizard uses a shared MFC DLL which is USUALLY a bad thing, so we will change that to using the MFC static library first. Do this by selecting

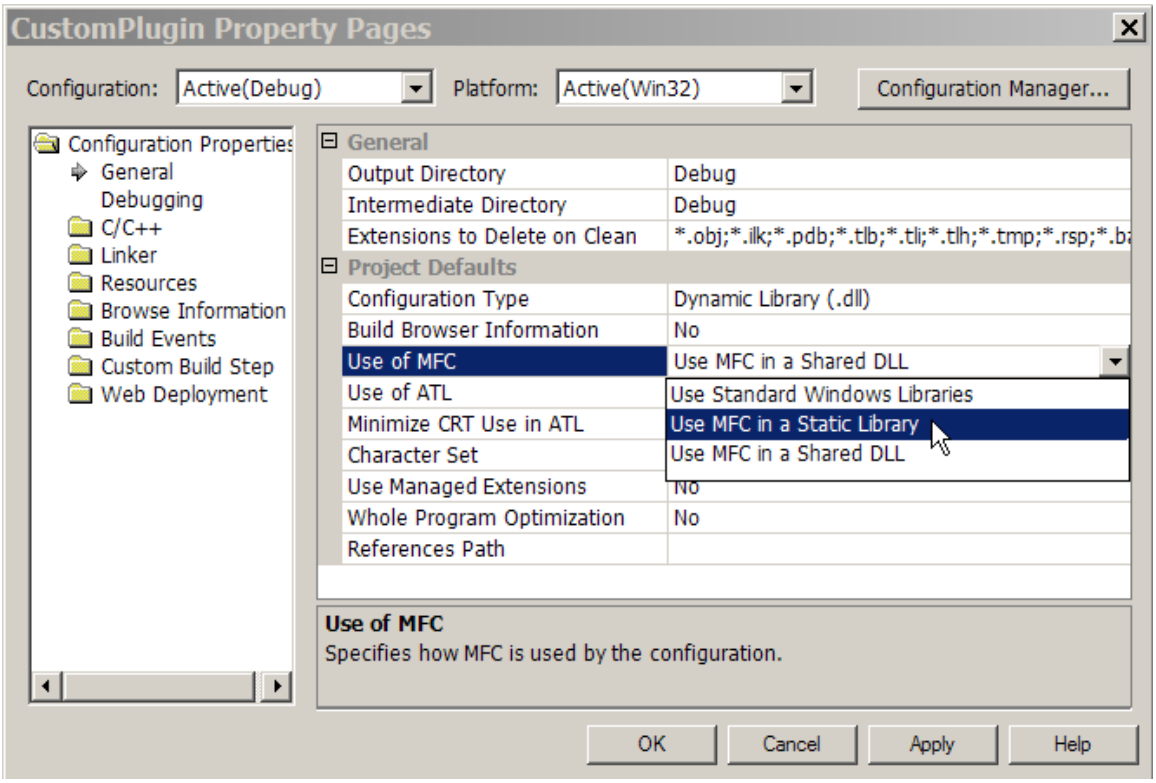
Project->Properties



Then select:

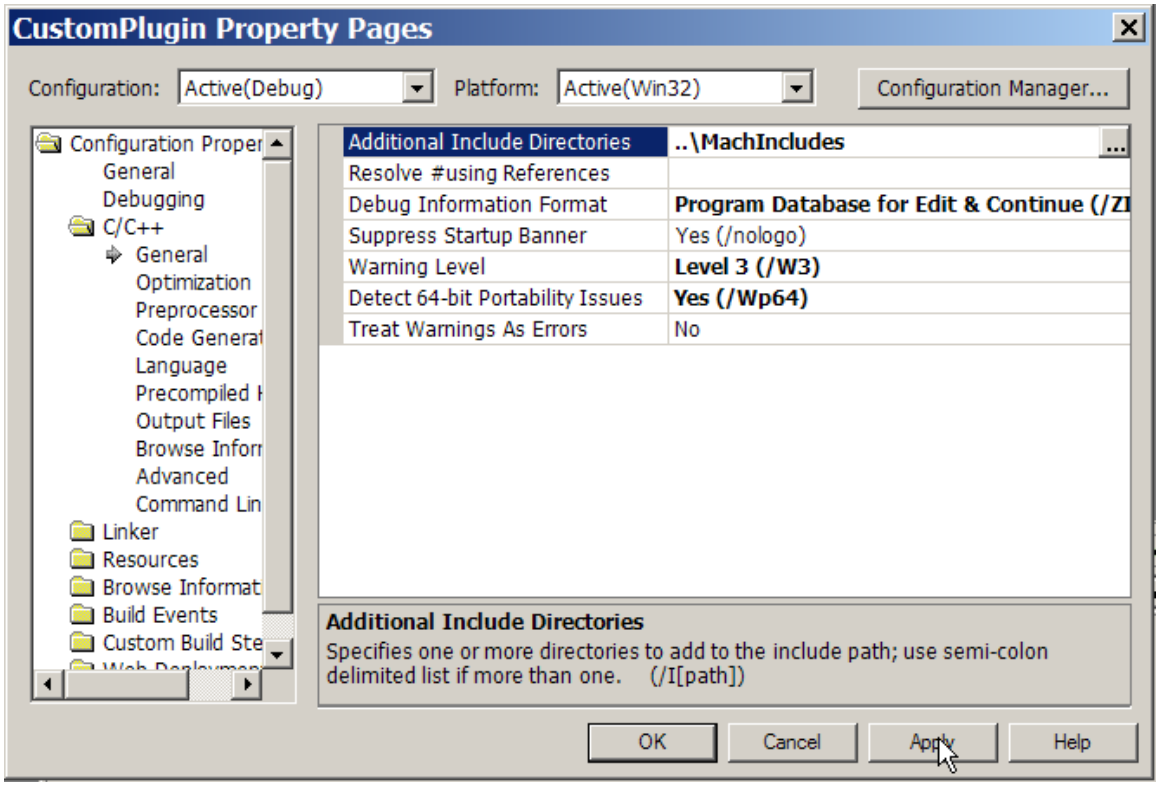
Configuration Properties->General

Then change 'Use of MFC' from 'Use MFC in a Shared DLL' to 'Use MFC in a Static Library'.

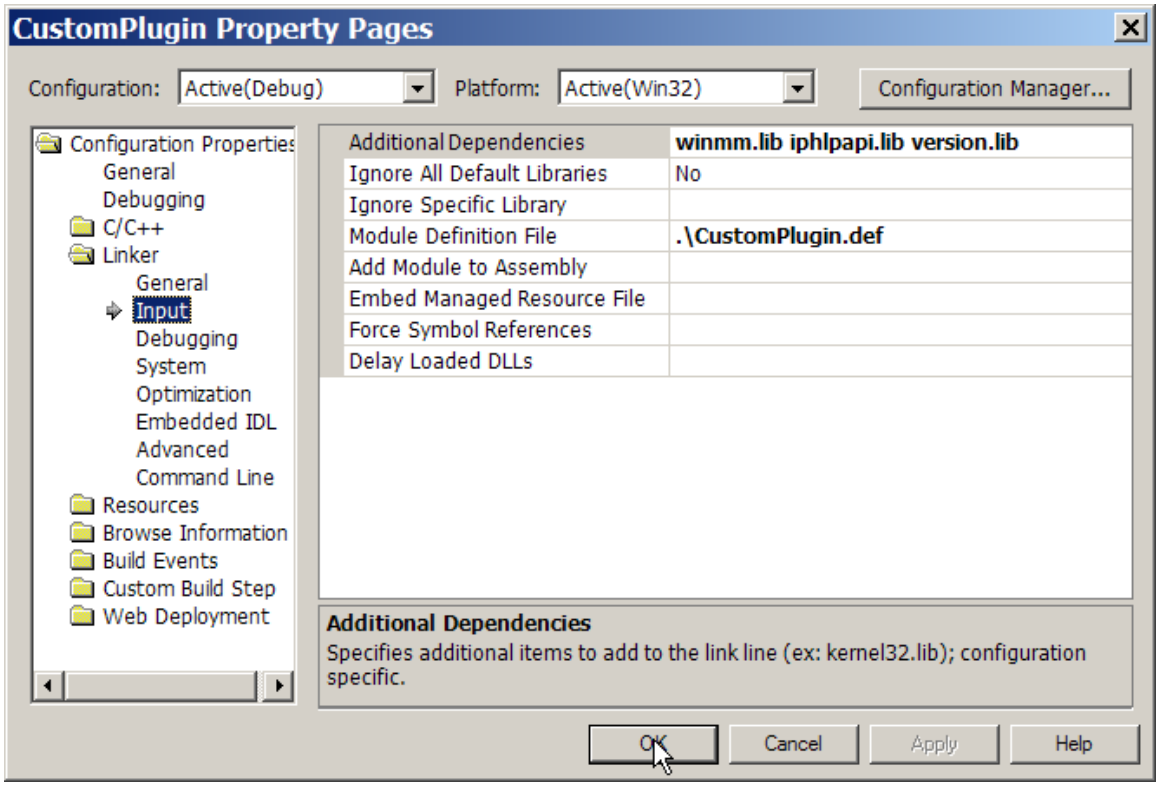


Insure that the following settings are made to the project:

additional include directories ..\MachIncludes

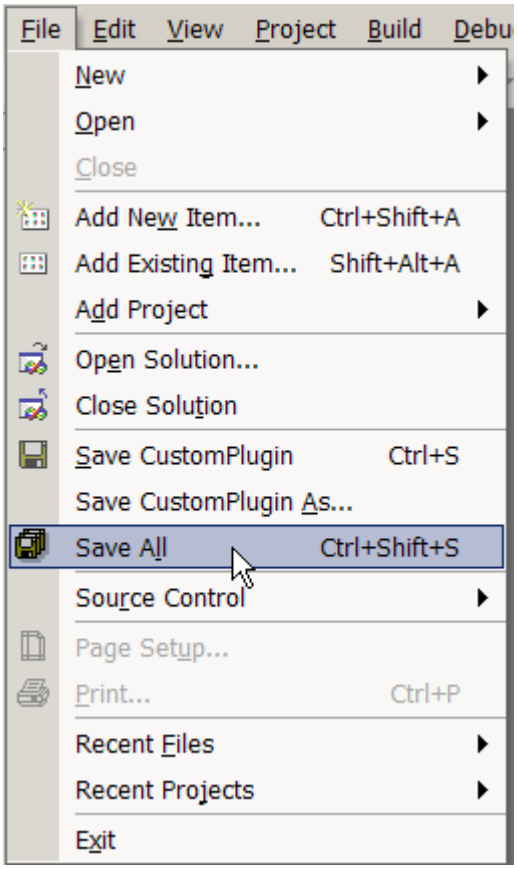


additional dependencies winmm.lib iphlpapi.lib version.lib

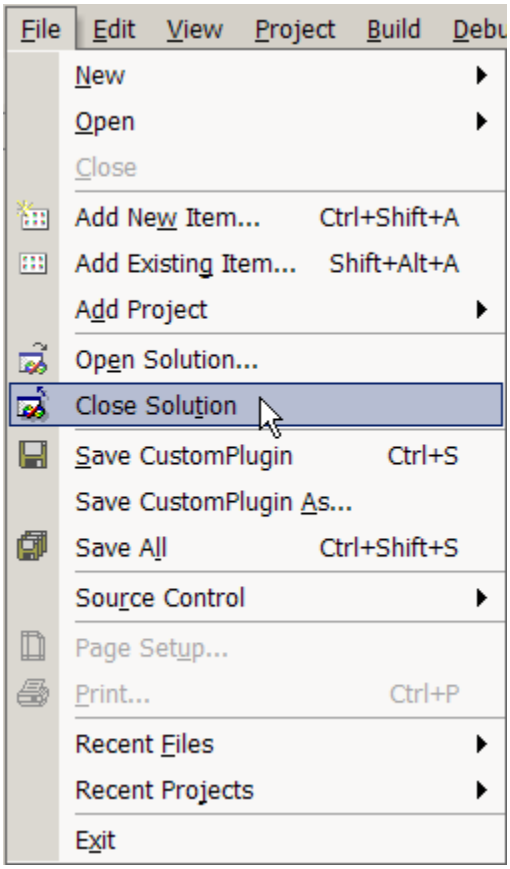


Press ‘OK’ to complete this step.

Use File->Save All to write the project settings to disk



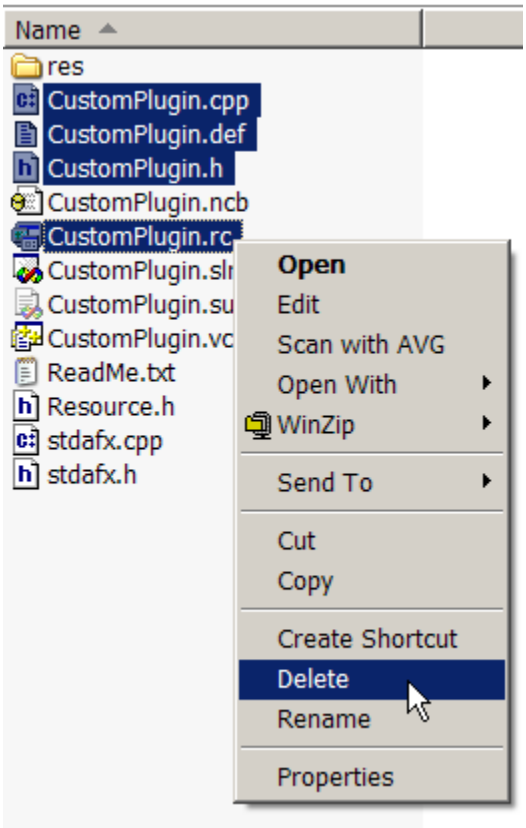
Then close the solution for the next steps



Replacing the Generated Files with the PluginProto Files

Now delete the files named 'CustomPlugin.cpp', 'CustomPlugin.def', 'CustomPlugin.h' and 'CustomPlugin.rc' that were created in the project directory:

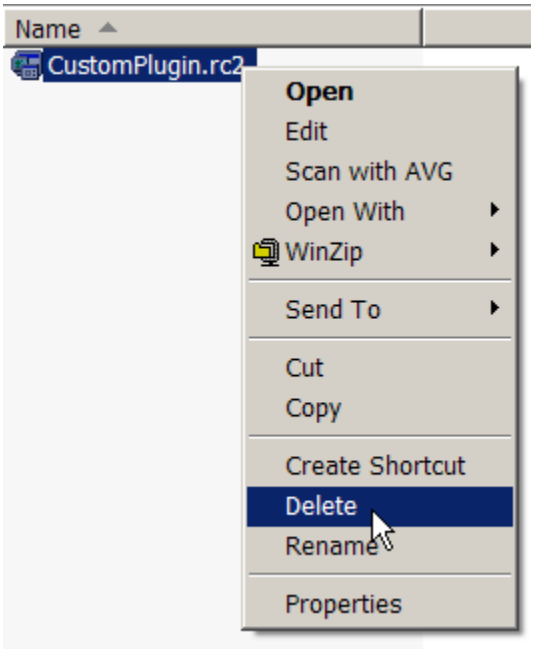
C:\CNC\Mach3Development\SDK2.03.00\CustomPlugin



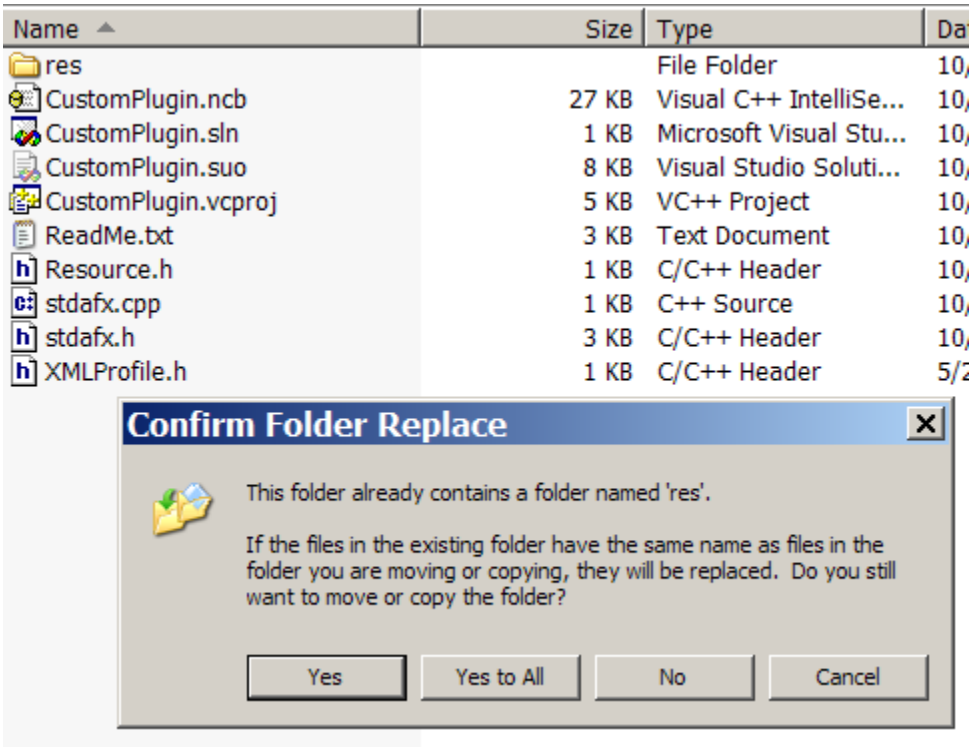
These files will be replaced with files taken from the ‘PluginProto.zip’ archive that you unzipped at the beginning of this tutorial.

AND delete the file named 'CustomPlugin.rc2' in the RES subdirectory.

C:\CNC\Mach3Development\SDK2.03.00\CustomPlugin\res

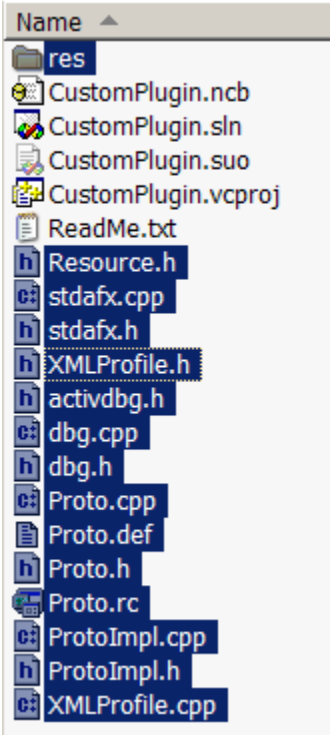


Then copy all the files and RES subdirectory from the PluginPrototypeFiles directory.

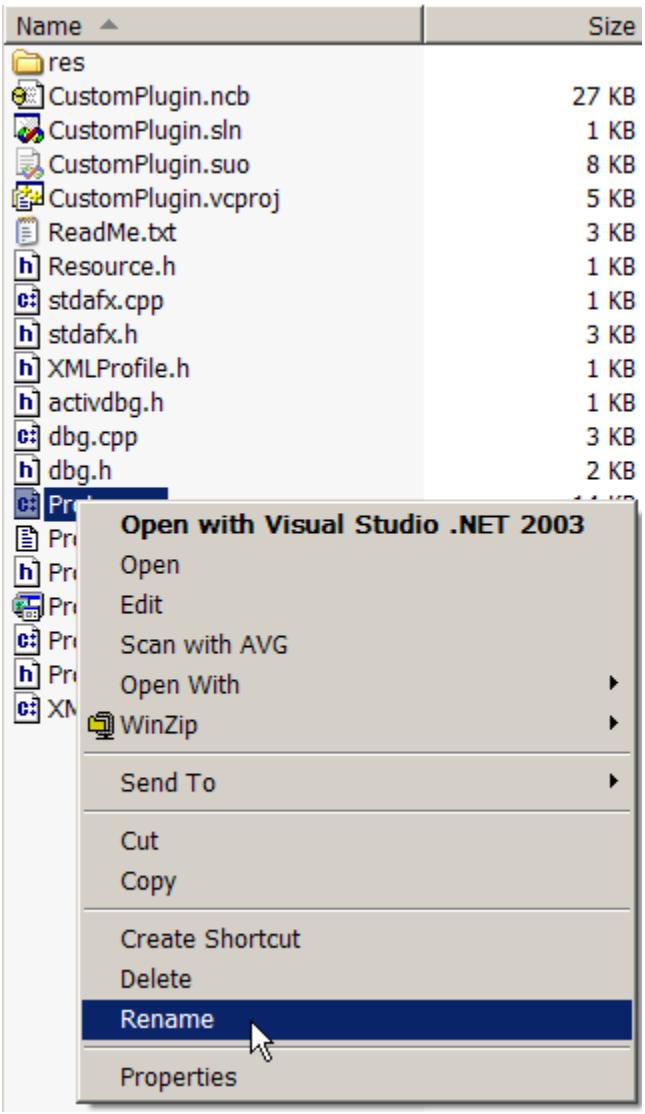


Answer ‘Yes to all’ here to completely update everything

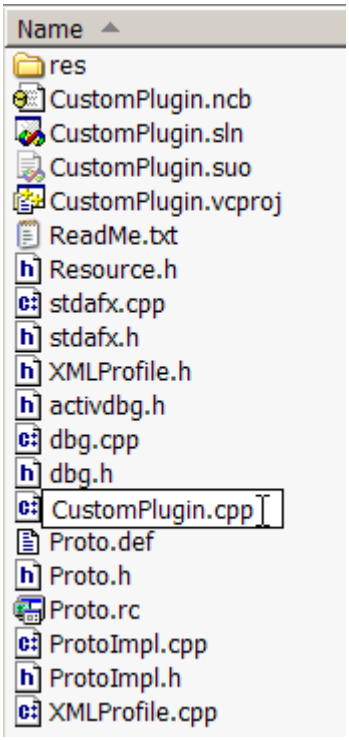
Here is the folder after the copy is completed.



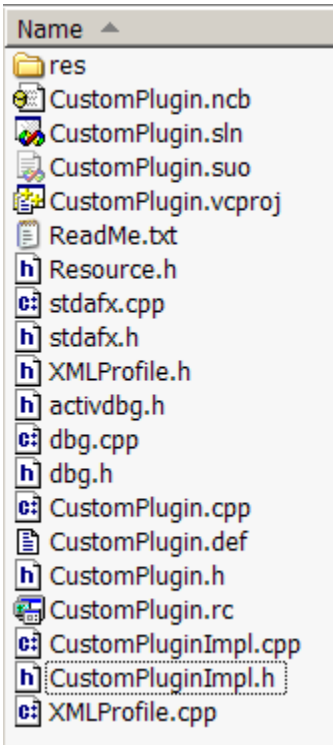
Rename all the 'Proto*.*' files to CustomPlugin*.* after they have been copied to the project folder.



Here is the file ‘Proto.cpp’ renamed to ‘CustomPlugin.cpp’.

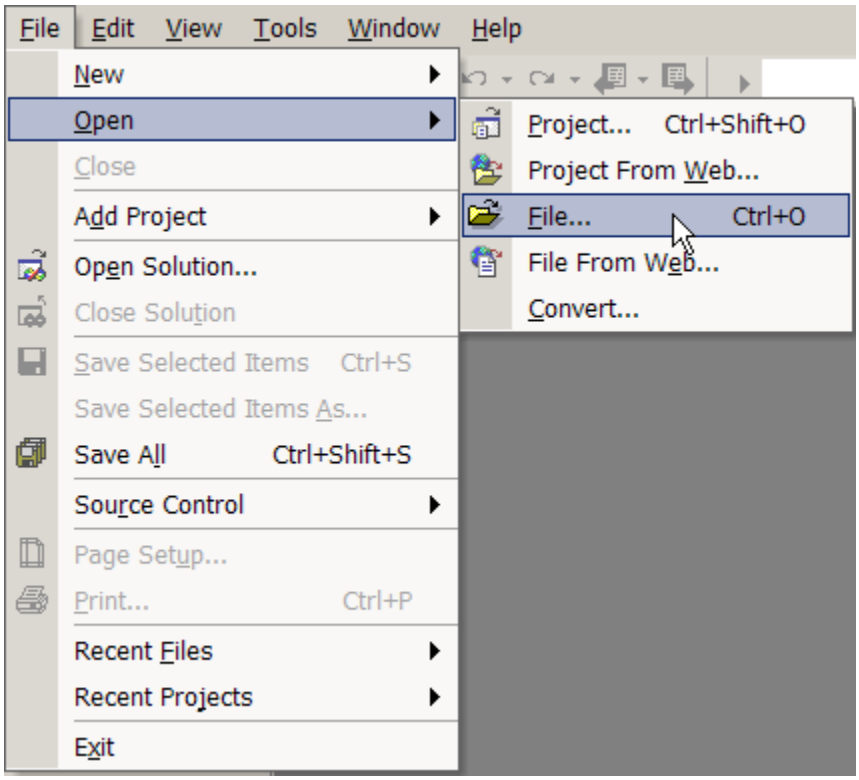


Here is the finished folder of renamed files. Be sure to open the ‘res’ folder and rename ‘proto.rc2’ to ‘CustomPlugin.rc2’ also.

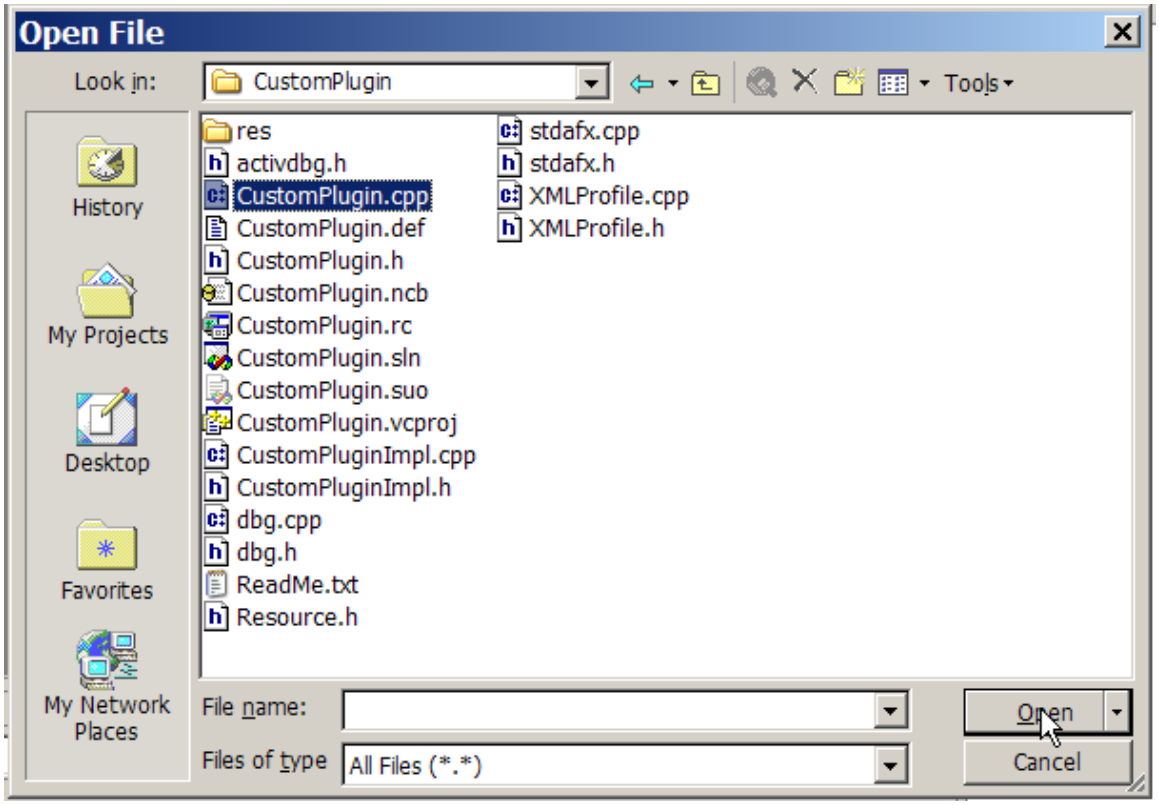


Editing The Replaced Files

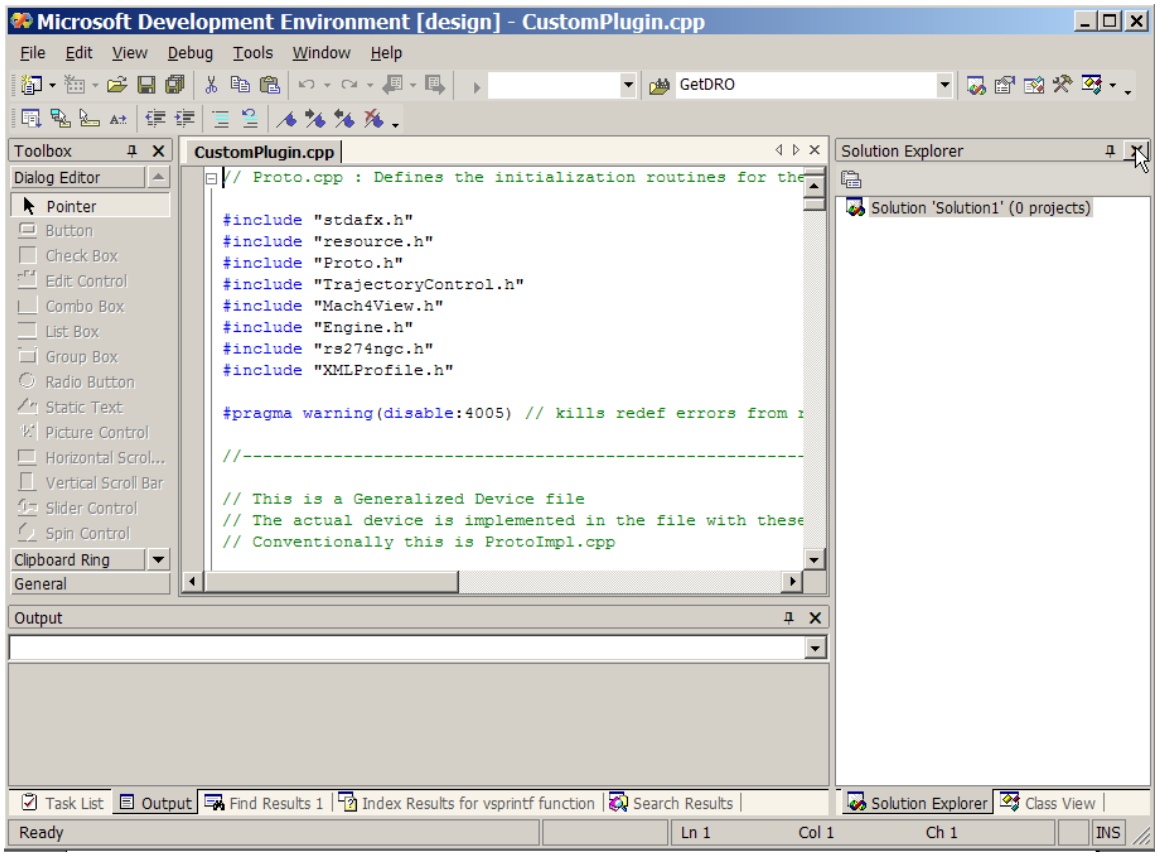
Edit all the files so that all references to 'Proto' become 'CustomPlugin'



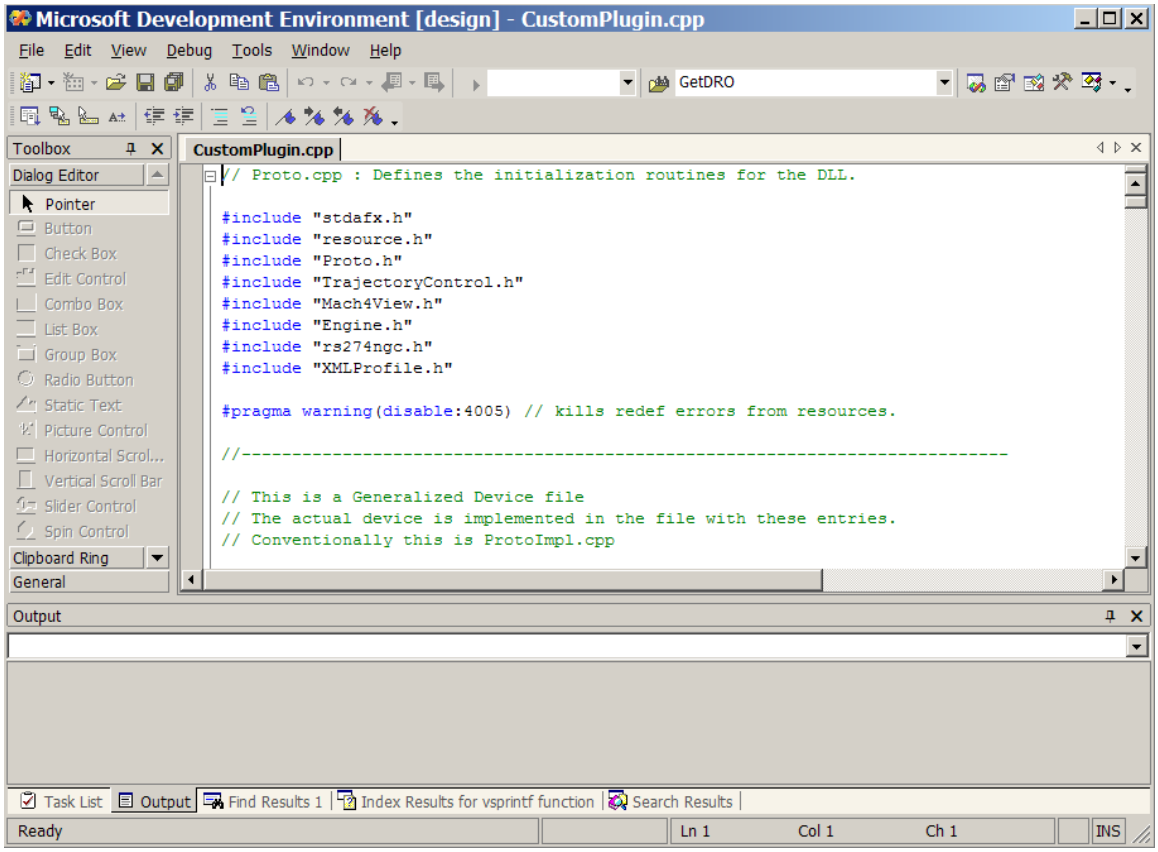
Open the file ‘CustomPlugin.cpp’



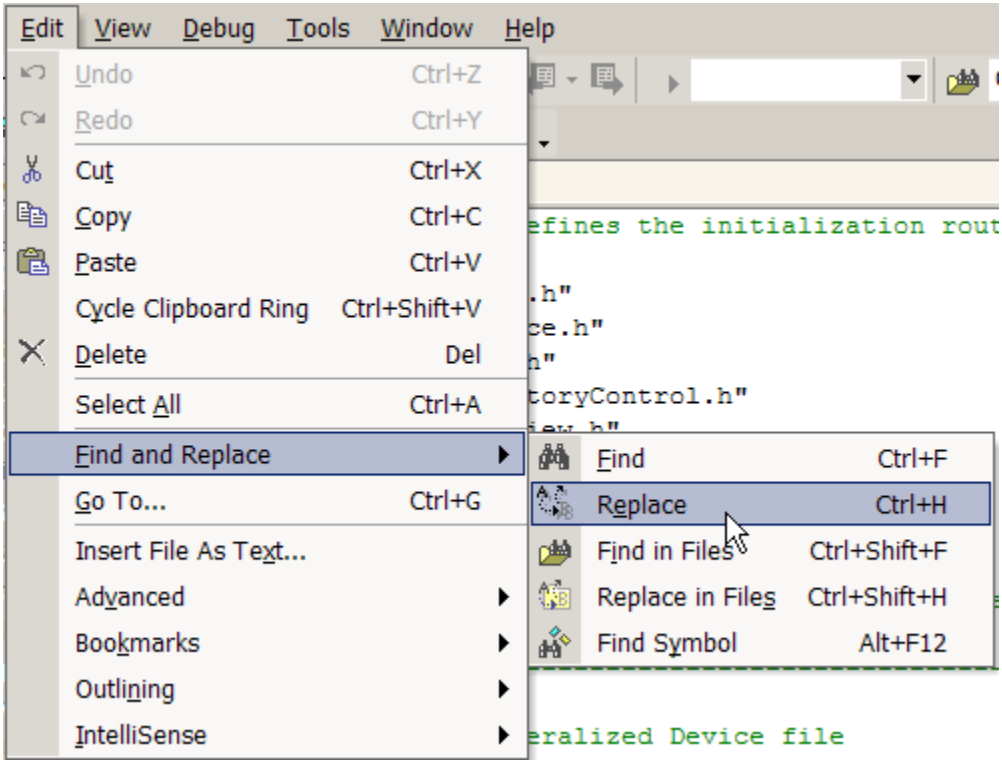
Close the solution explorer that was opened by default, it will not be needed. You will have to click on the 'X' twice to completely close this window.

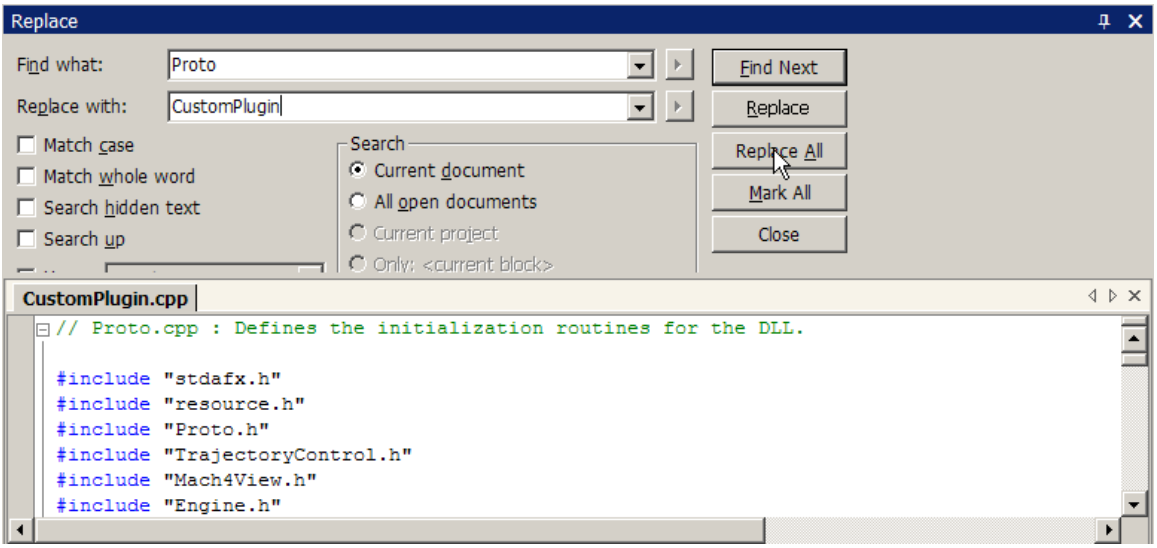


Now we have the edit in a much bigger space

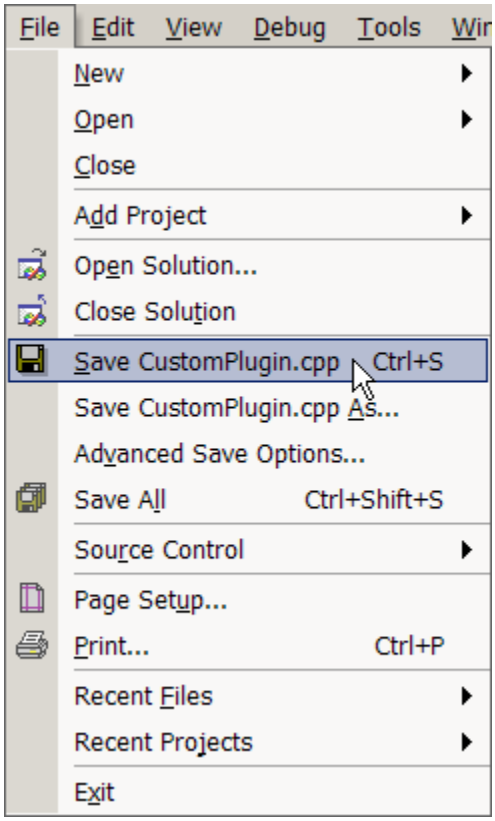


Use the ‘Find and replace’ feature of the editor to locate all instances of ‘Proto’ and change it to ‘CustomPlugin’.



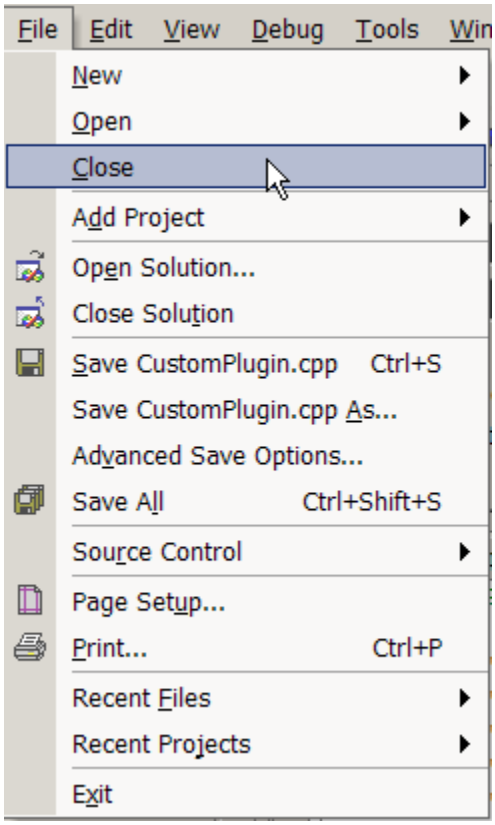


Now save this file.



Note that is you don't see the name of the file you want to save, just move the mouse out side the list and release the button. The click on the text that was edited in the editor window and try again. This should make the file name appear instead of 'Solution1'.

And close it.



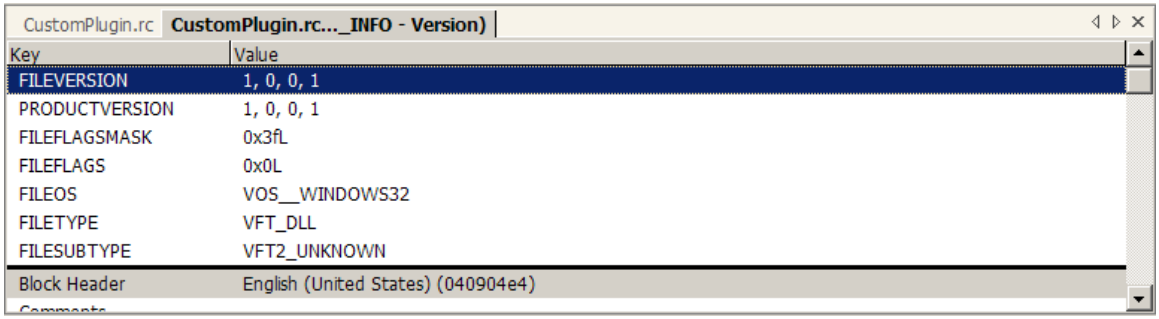
Edit CustomPlugin.def , CustomPlugin.h , CustomPluginImpl.cpp and CustomPluginImpl.h in the same manner.

Edit resource.h before editing CustomPlugin.rc. These files have a relationship that must be maintained.

CustomPlugin.rc requires manual editing since 'Find and Replace' does not work on a windows resource file that is opened with the resource editor. Resources will be discussed later in this document. When you open CustomPlugin.rc you will see the following screen.

Click on the '+' beside 'Version' and you will see this screen.

Double click on VS_VERSION_INFO and then you will see this screen.

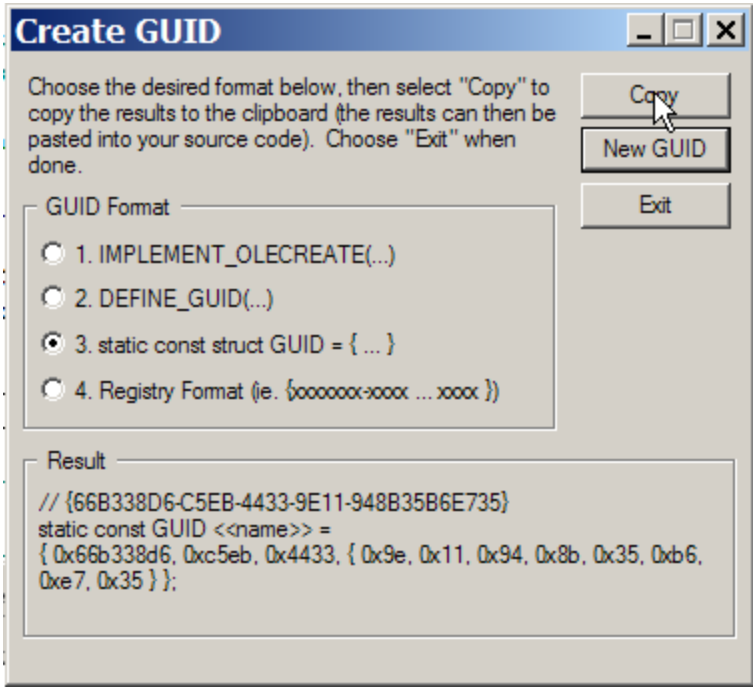


Now you can scroll down and just change each instance of 'Proto' to 'CustomPlugin'. You do this by double clicking on each item which gives you an edit box to change the text in.

Change the plugin name and GUID associated with the plugin. This is contained in CustomPlugin.cpp The new GUID is produced by using the utility GUIDGEN which should be found in:

C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools

You use GuidGen by double clicking on it with Windows Explorer. This will give you this screen. Select the third radio button '3. static const struct GUID = {…}' and press the 'Copy' button.



The result can then be pasted into the CustomPlugin.cpp source code where the old GUID is and the extra lines of code produced by GuidGen removed.

The goal is to replace this line:

```
{ 0xC9FB259, 0xB864, 0x40A5, { 0xB5, 0x9F, 0x65, 0xE1, 0x1E, 0x20, 0x9F, 0xC4 } };
```

With the new GUID, which in this case only is:

```
{ 0xd52a7ae5, 0x4740, 0x4b28, { 0xa9, 0xf1, 0x8c, 0xb, 0x72, 0x92, 0x6d, 0xa9 } };
```

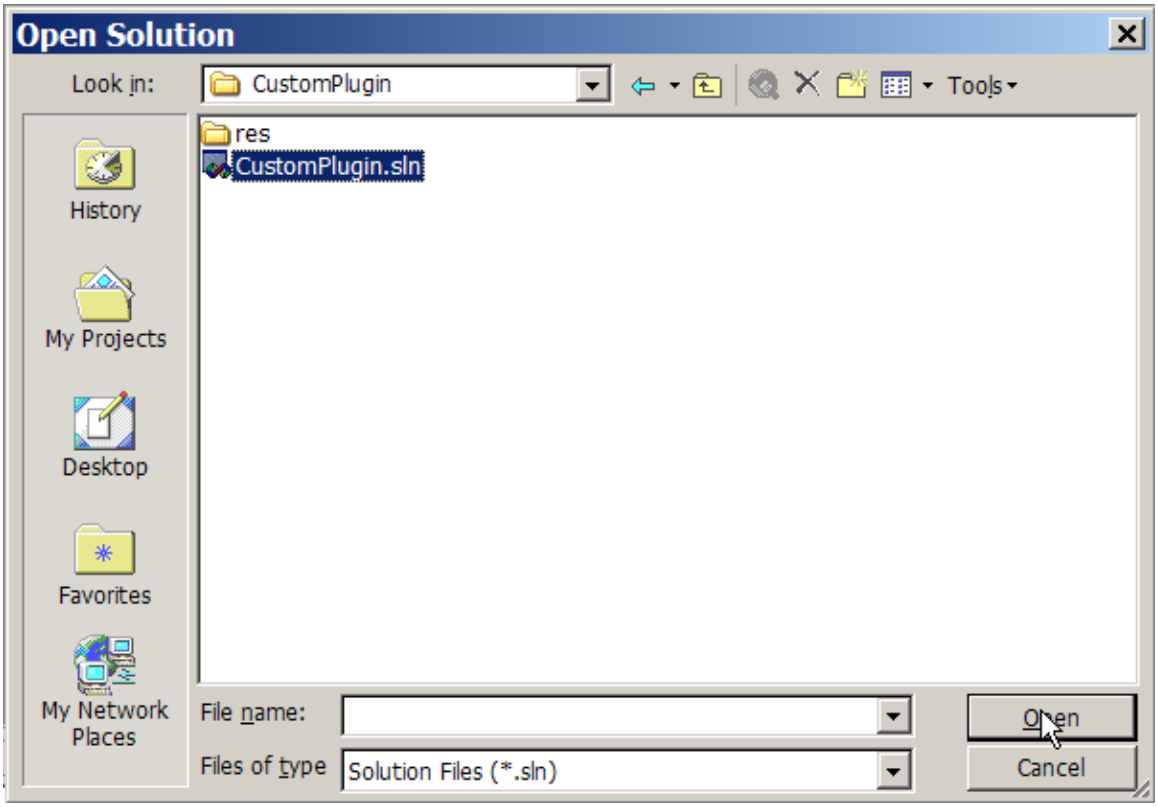
```
//----------------------------------------------------------------------------

// BE SURE to use GuidGen to make a GUID for each plugin that you derive
// from this CustomPlugintype file set

// The one and only CCustomPluginApp object

CCustomPluginApp theApp;

const GUID CDECL BASED_CODE _tlid =

// {D52A7AE5-4740-4b28-A9F1-8C0B72926DA9}
static const GUID <<name>> =
{ 0xd52a7ae5, 0x4740, 0x4b28, { 0xa9, 0xf1, 0x8c, 0xb, 0x72, 0x92, 0x6d, 0xa9 } };


        { 0xC9FB259, 0xB864, 0x40A5, { 0xB5, 0x9F, 0x65, 0xE1, 0x1E, 0x20, 0x9F, 0xC4 } };

const WORD _wVerMajor = 1;
const WORD _wVerMinor = 0;
```

Remove the old GUID and the ‘static const GUID <<name>> =’ and you will end up with this screen, except you will have different GUID values, of course.

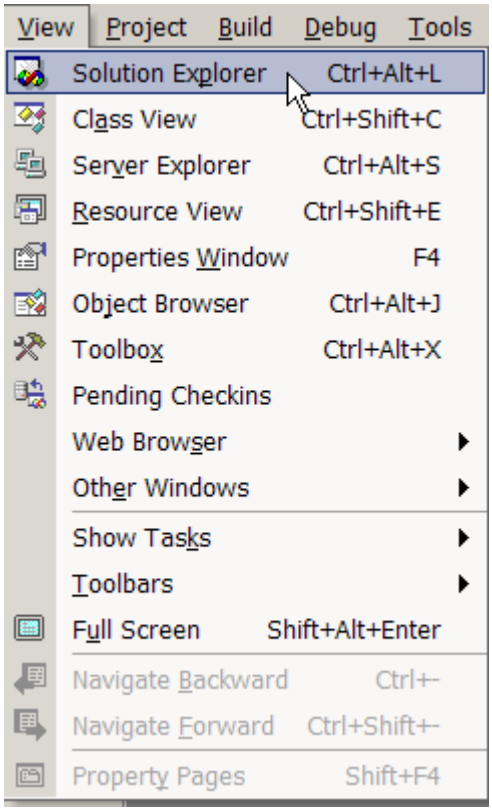
```
//----------------------------------------------------------------------------

// BE SURE to use GuidGen to make a GUID for each plugin that you derive
// from this CustomPlugintype file set

// The one and only CCustomPluginApp object

CCustomPluginApp theApp;

const GUID CDECL BASED_CODE _tlid =
    { 0xd52a7ae5, 0x4740, 0x4b28, { 0xa9, 0xf1, 0x8c, 0xb, 0x72, 0x92, 0x6d, 0xa9 } };

const WORD _wVerMajor = 1;
const WORD _wVerMinor = 0;

//----------------------------------------------------------------------------

// CCustomPluginApp initialization

BOOL CCustomPluginApp::InitInstance()
```

Adding the Remaining Project Files

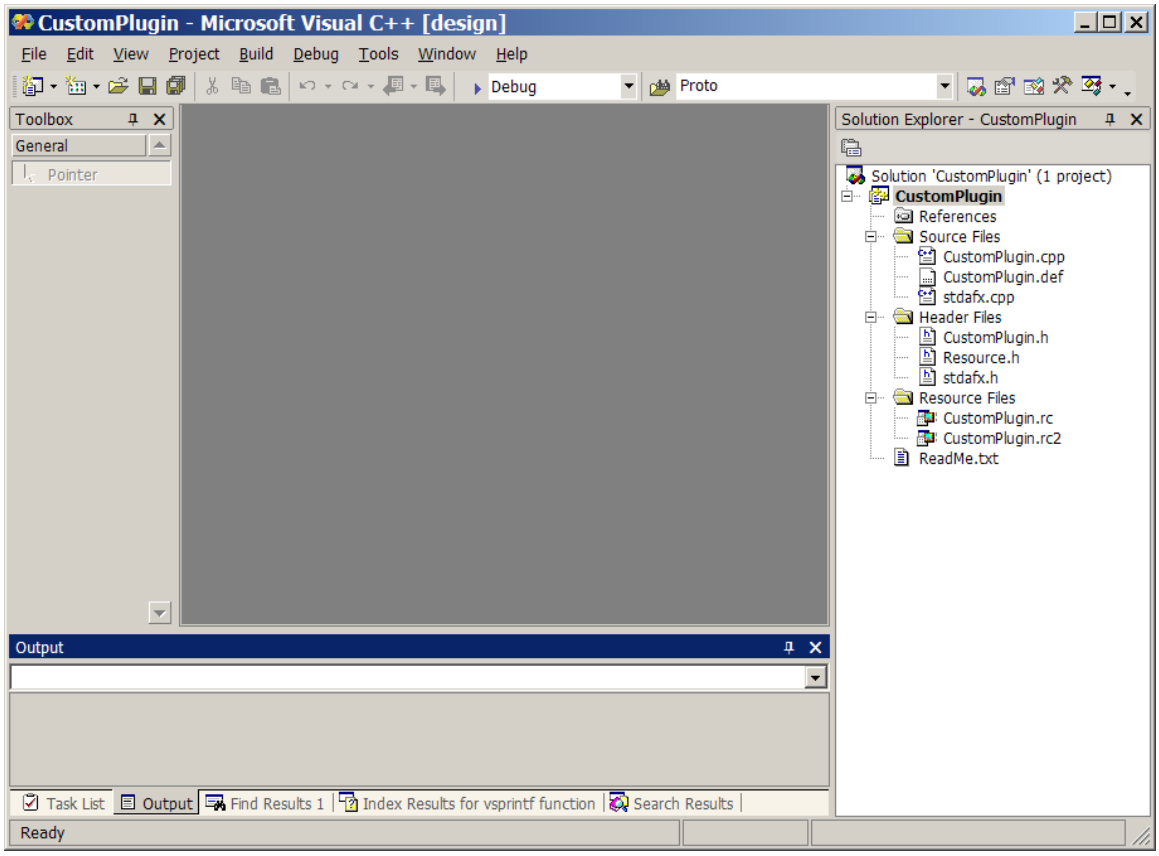
Close this file and then reopen the original solution file that was generated when we created the DLL originally with the AppWizard.



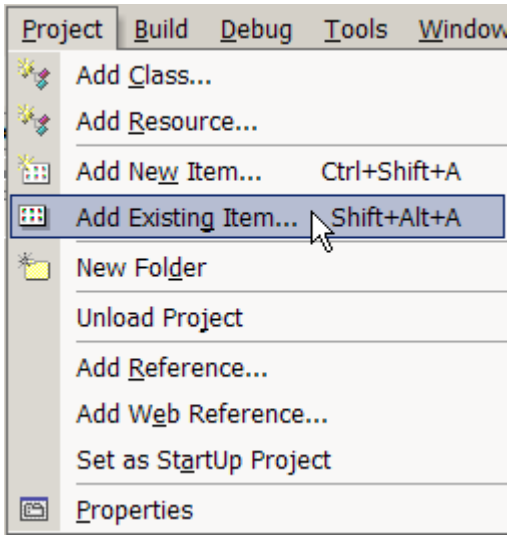
Reopen the solution explorer window so you can navigate around the project easily.



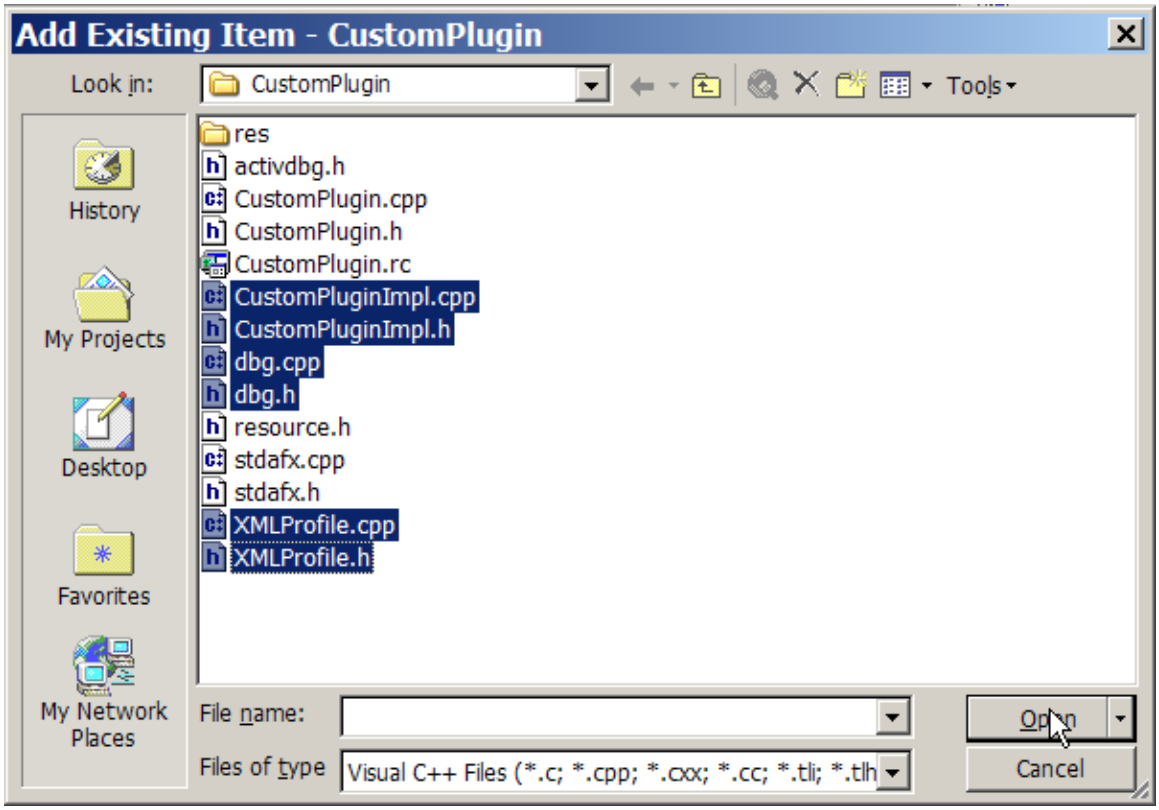
You will now see this screen.



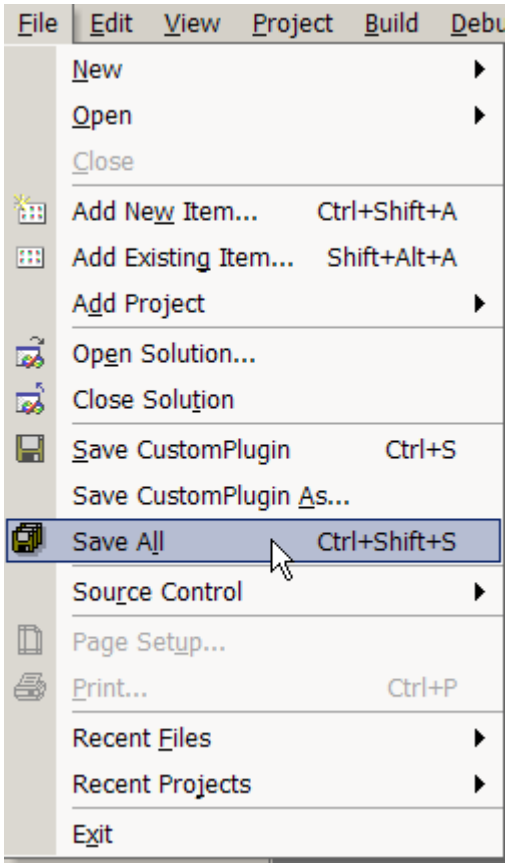
Add the files 'dbg.cpp', 'dbg.h', 'XMLProfile.cpp' and 'XMLProfile.h' to the project. Add the user code implementation files also. In this case they would be 'CustomPluginImpl.cpp' and 'CustomPluginImpl.h'



Select the files to add and press ‘Open’.



Use ‘File->Save All’ to save the current project state.



Testing The Custom Plugin

Test build and then copy the finished CustomPlugin.dll from the folder

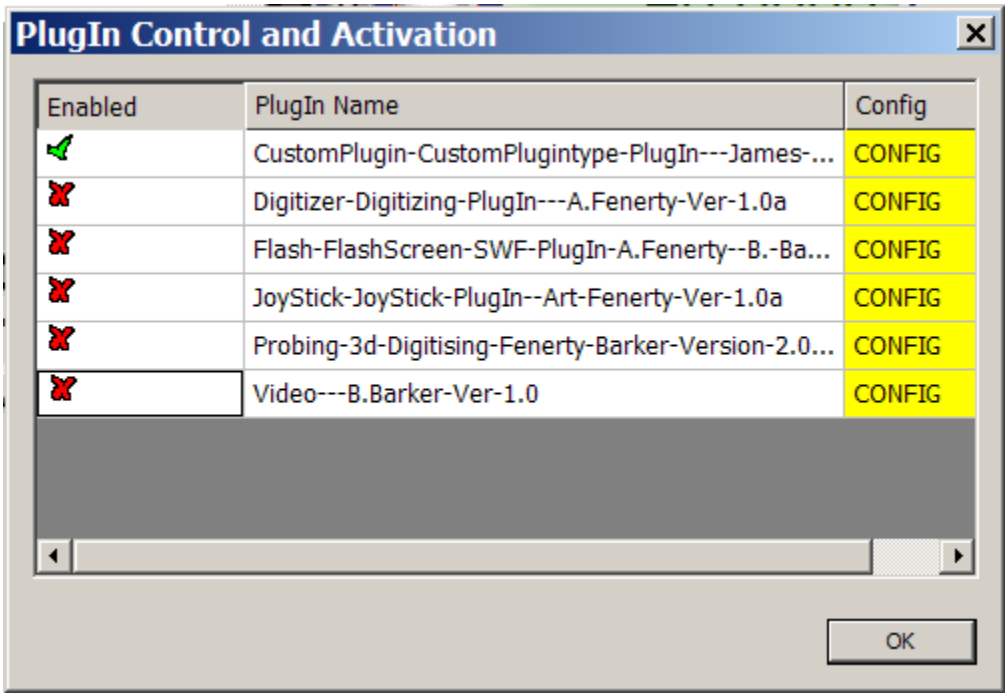
C:\CNC\Mach3Development
SDK2.03.00
CustomPlugin
Debug

To the Mach3 plugins folder:

C:\Mach3\plugins

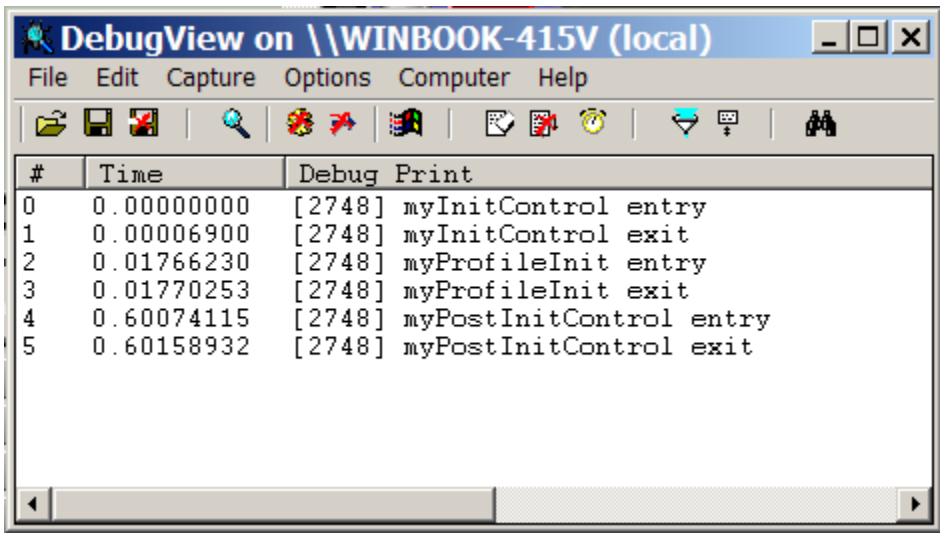
Mach3 Plugin Development Tutorial

Start Mach3 and configure the installed plugins so that only the new CustomPlugin is enabled. This will make the testing of the new plugin very simple.



Close Mach3 and then restart it so that all the DbgMsg messages that are in the CustomPluginImpl.cpp file can be seen at startup. Confirm debug messages are seen in DebugView so that you are sure the plugin loads and runs successfully.

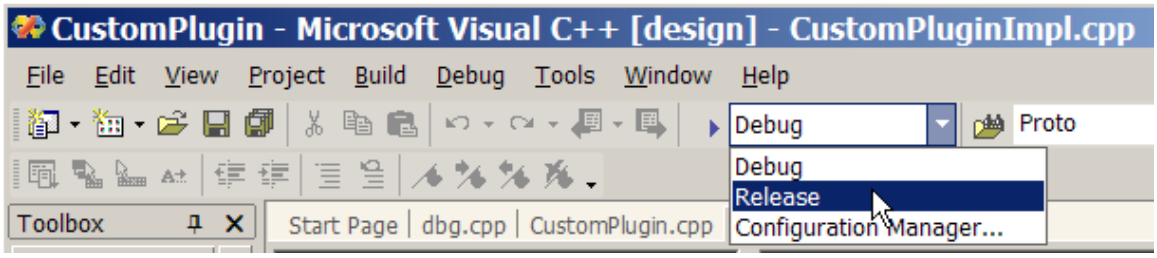
In order to use DebugView you must download it from the Microsft Sysinternals webpage and then install it. Here is what a test run on my laptop with Mach3 looks like at startup time:

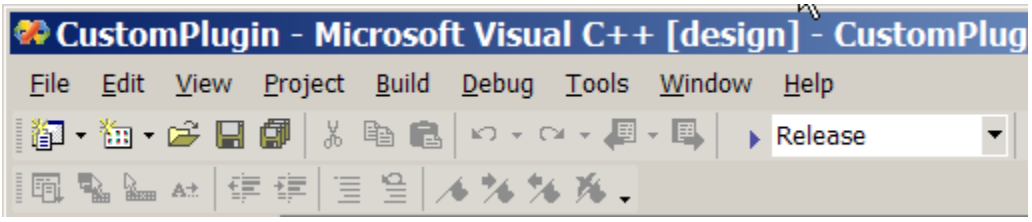


Release Builds of the Custom Plugin

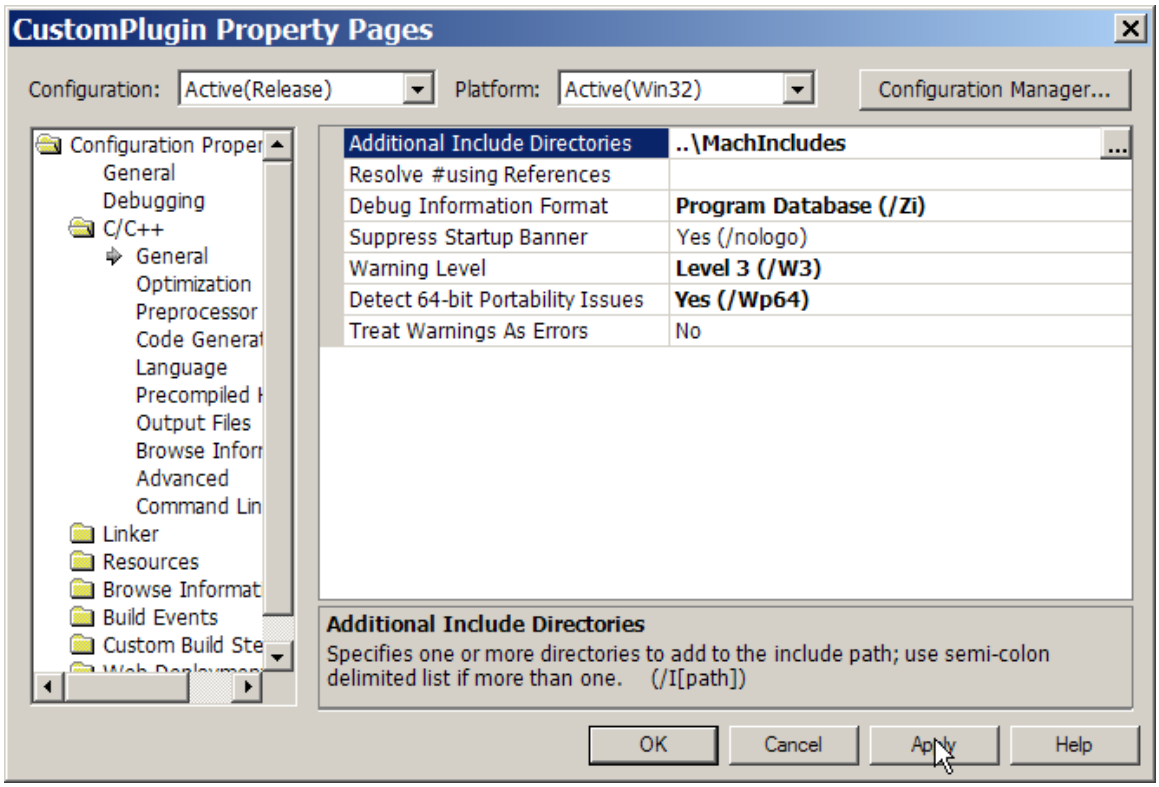
The final step in the first part of this tutorial is adding the necessary configuration to Visual Studio to allow a release build of your new plugin. Release builds remove ALL DbgMsg debugging outputs and overhead leaving only RelMsg and ErrMsg output to DebugView. A release build is also smaller and faster, it is the form that you want to ship to your customers.

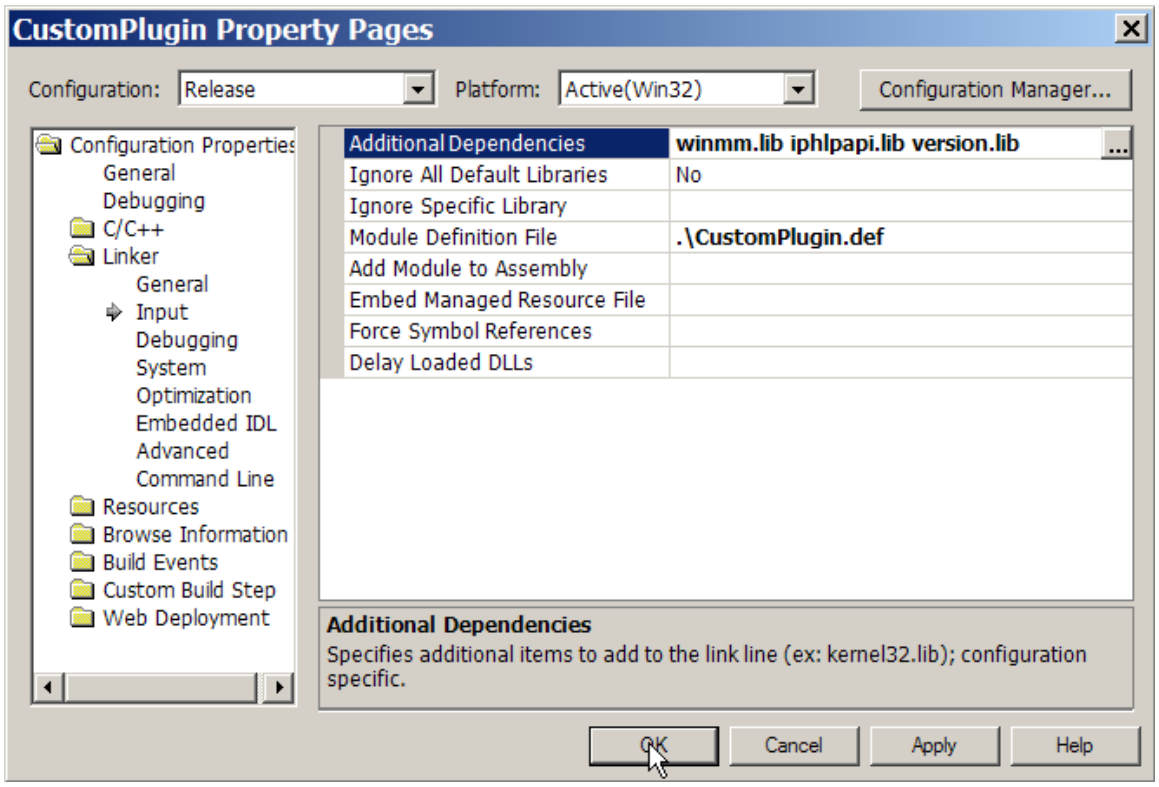
Open Visual Studio and change the build type from 'Debug' to 'Release'.





Now repeat the configuration that was done initially to give the location of the MachIncludes folders and the added dependencies.



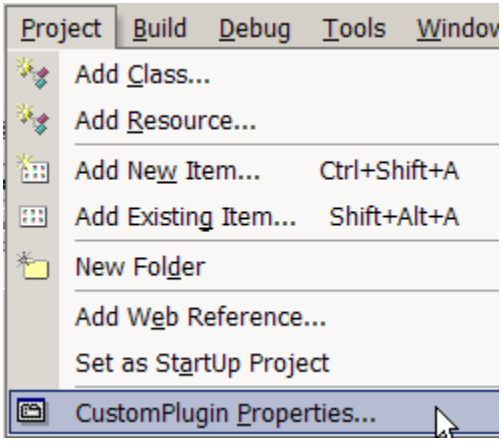


Now when you make the release build the finished DLL will be in:

C:\CNC\Mach3Development
SDK2.03.00
CustomPlugin
Release

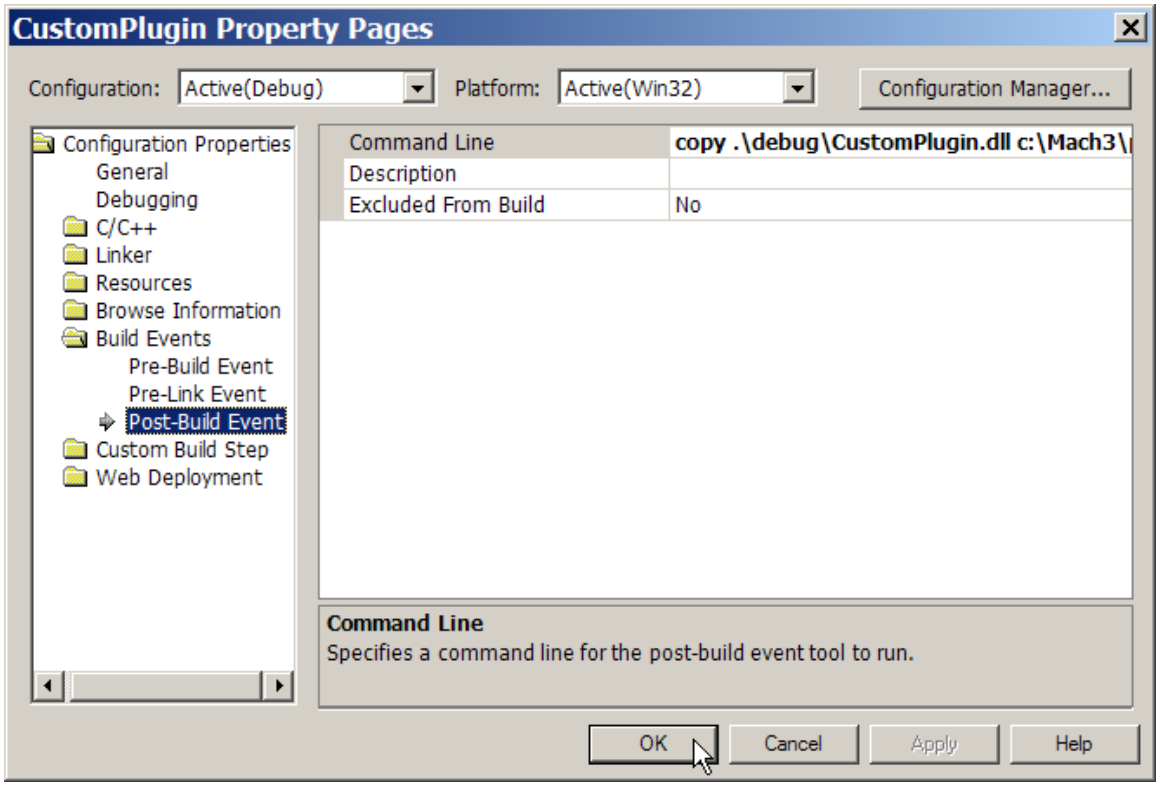
Making Plugin Testing Easier

Up to this point we have been copying the compiled plugin from either the Debug or Release folders associated with our CustomPlugin project. There is a much easier way to do this, it is called a "Post Build Event". A Post Build Event will let you execute a command line application (copy in this case) to perform some extra processing that is only done for a successful build. What we will do is set this up to automatically copy the CustomPlugin.dll to the Mach3 plugins folder so that we can test it. The post build events are set by opening the Project->Properties menu.



For Debug builds we need to select “Post Build Event” and add the following Command Line:

copy .\debug\CustomPlugin.dll c:\Mach3\plugins



Repeat this process for Release builds except you should change the Command Line to:

copy .\release\CustomPlugin.dll c:\Mach3\plugins

In order to avoid being puzzled by accidentally testing a Release build when you were expecting a Debug build (and DbgMsg output of course) add this to the ‘myPositInitControl’ function in CustomPluginImpl.cpp.

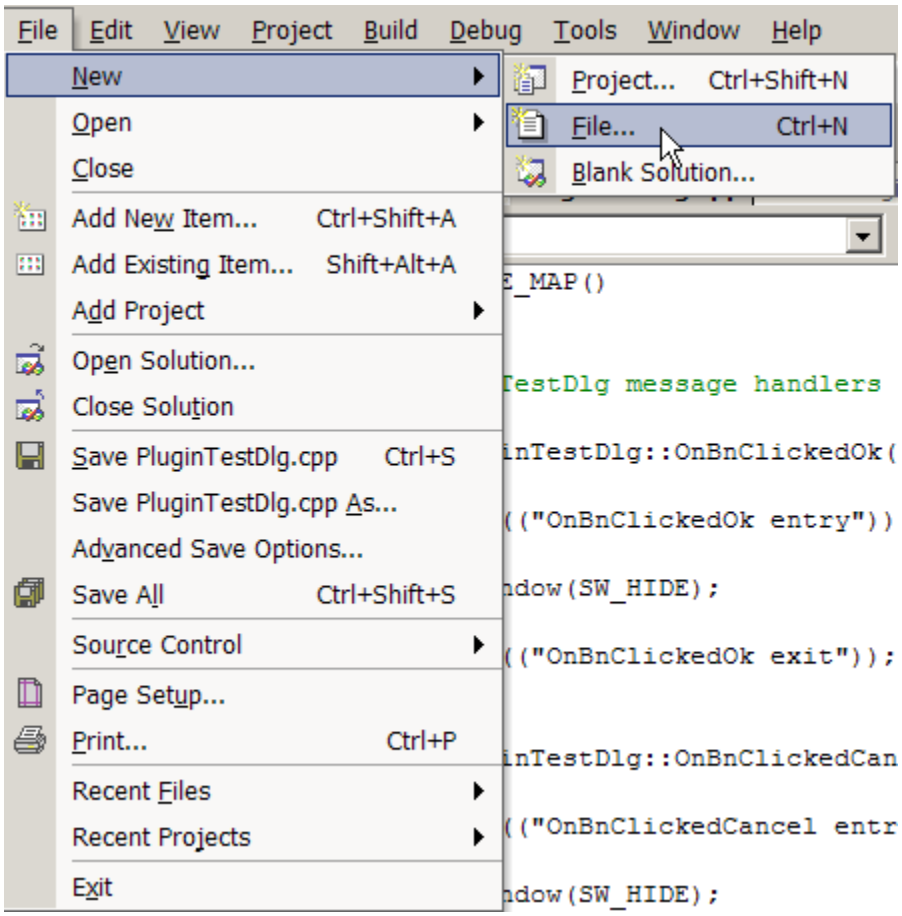
```
#ifndef _DEBUG

RelMsg(("Release build CustomPlugin"));

#endif
```

You will only see this message if you are running a release build of the CustomPlugin.

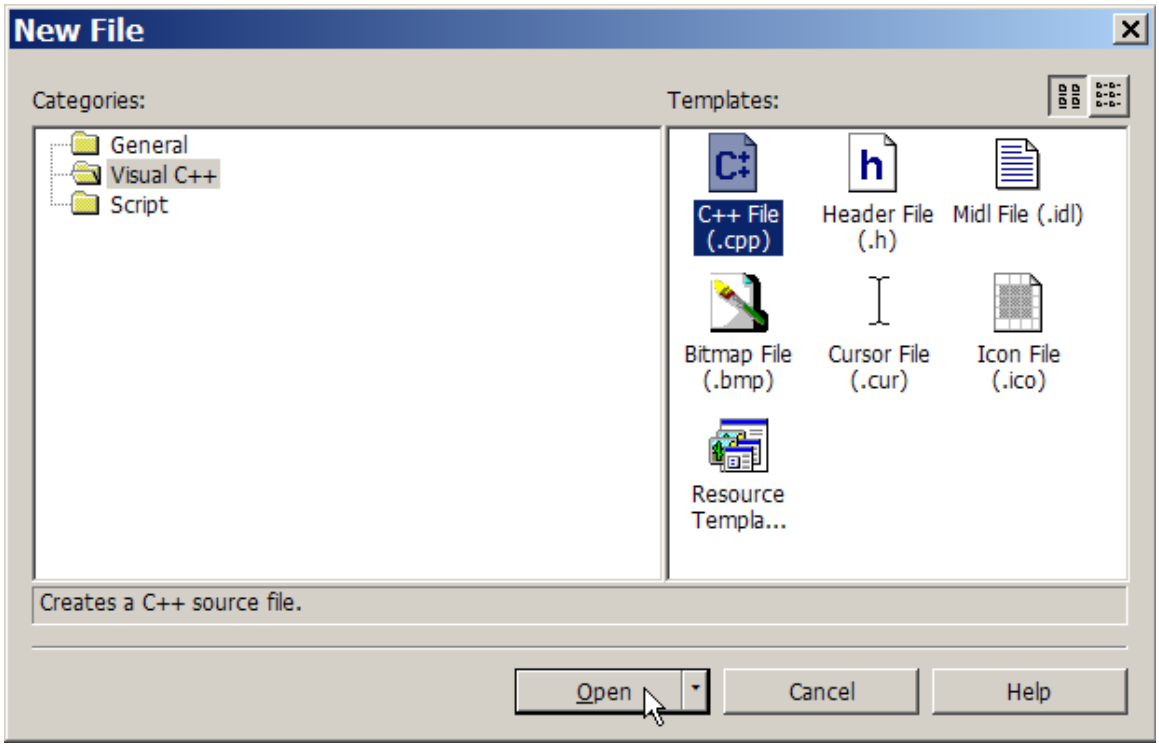
Adding a Utility Library

It is desirable to have a single place for 'helper' or utility functions to be coded. Since we need at least one such helper function for the next section, we will create this now, and add to it as we need to.

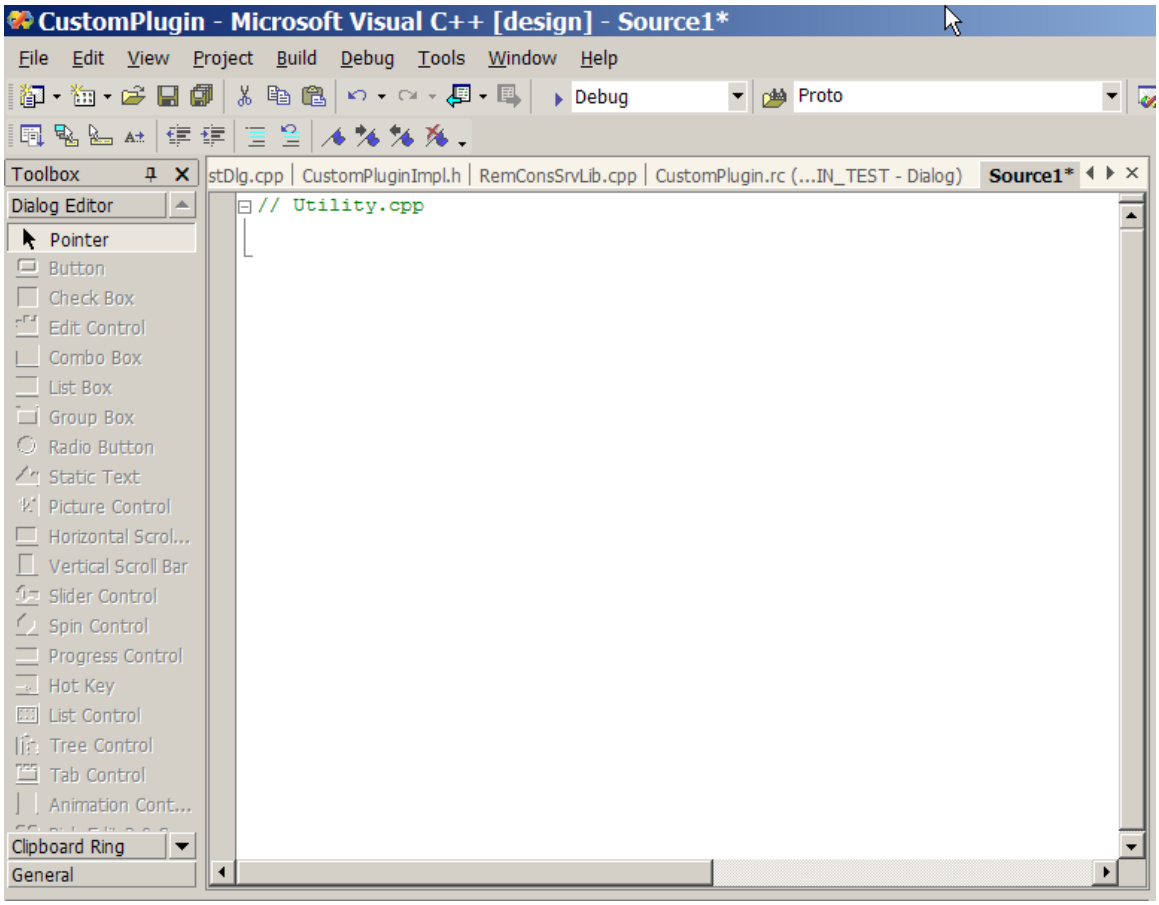
Use the 'File->New' menu to create a 'Utility.cpp' and a 'Utility.h' file.



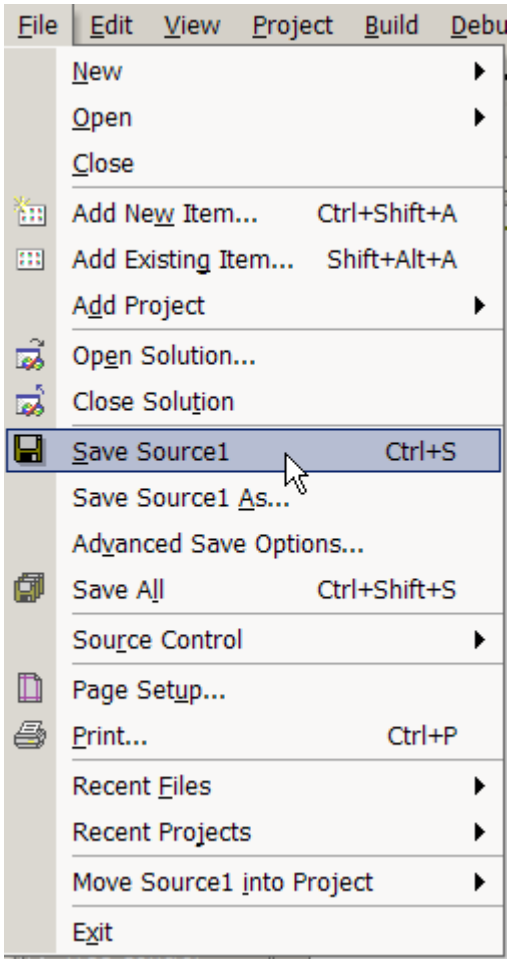
Mach3 Plugin Development Tutorial



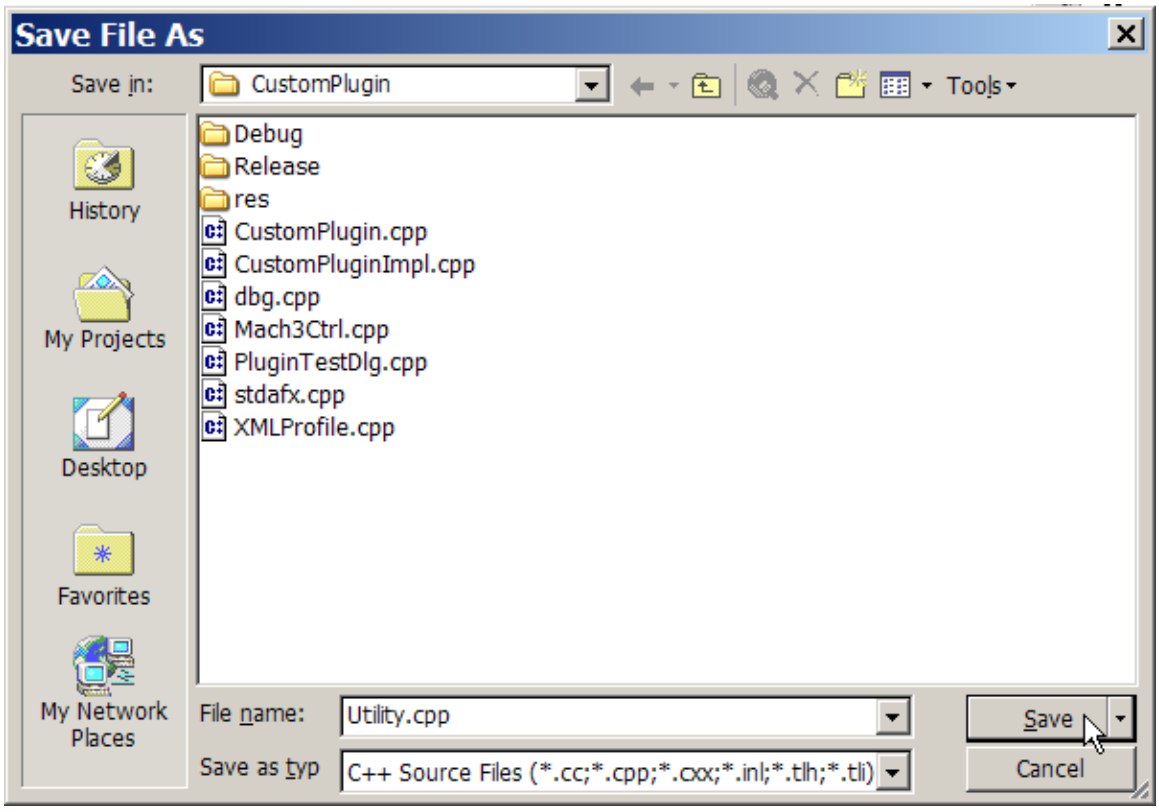
Add the text '// Utility.cpp' then save the file. The file will initially be named 'Source1.cpp' but you need to change that to 'Utility.cpp'.



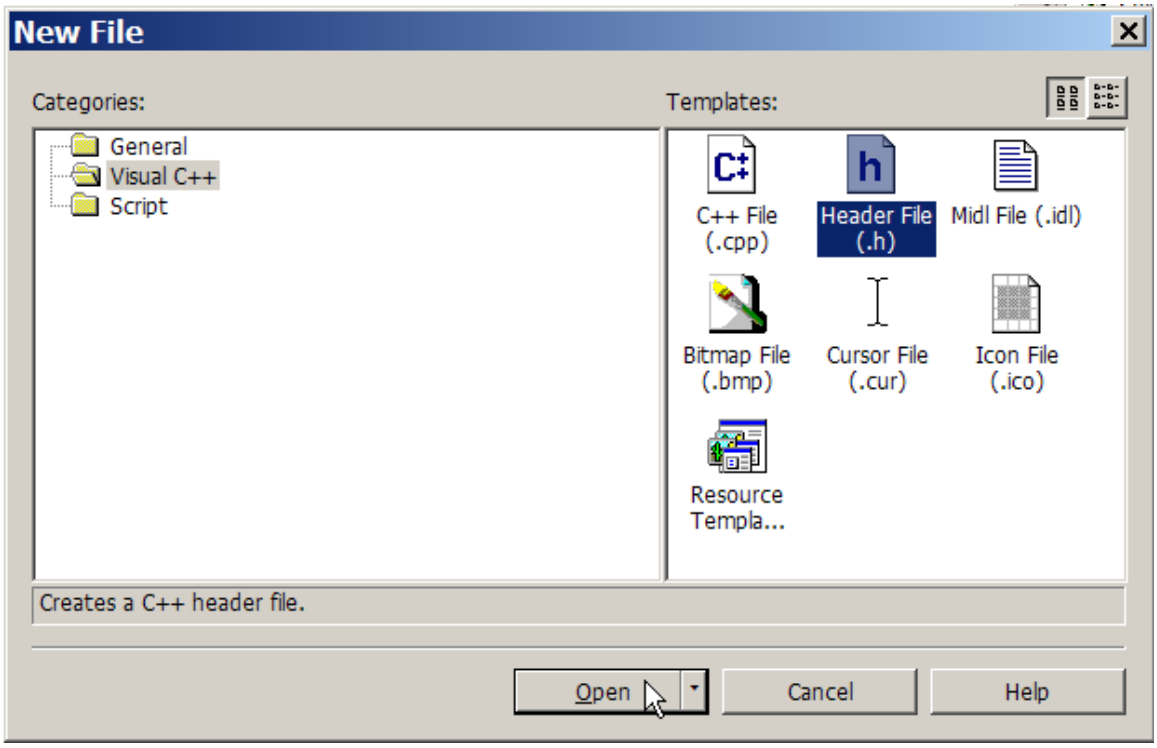
Use the ‘File->Save’ menu. Select ‘Save Source1’.



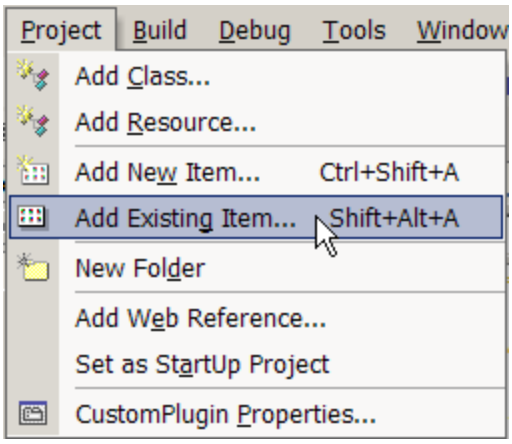
Now change the file name to ‘Utility.cpp’ and then press ‘Save’.



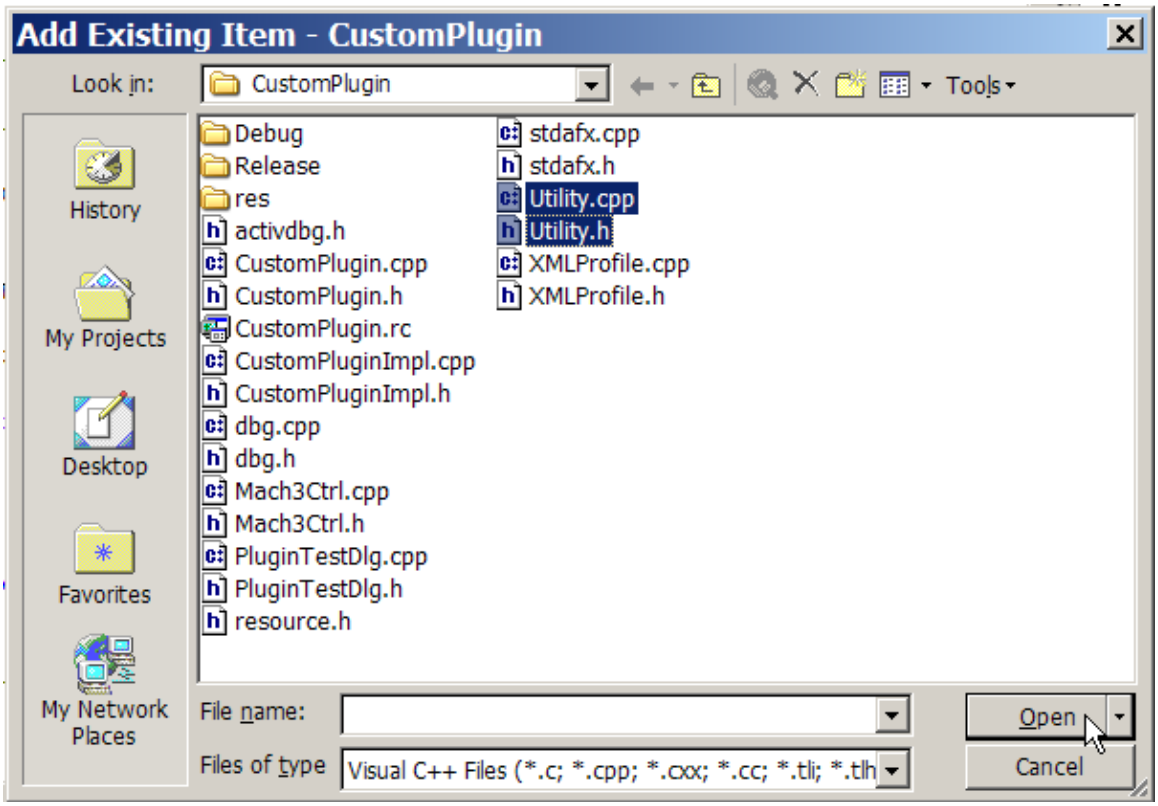
Repeat this process but select ‘Header File (.h)’ instead. Name this file ‘Utility.h’



The file creation wizard does NOT make the files you just created part of the CustomPlugin project, so you must add them yourself. Use the 'Project->AddExisting Item' menu to open a dialog where you can select these files to be added to the current project.



Press the 'Open button to finish adding the new files for the Utility library to the project.



Adding Functions to the Utility Library

The first utility library function that will be needed is a function to get the HWND of Mach3's main window. This will be used in the next section to make Mach3 the 'parent' of a dialog. This makes the dialog owned by Mach3 and allows us to manage it's lifetime in terms of Mach3's lifetime. We will call this function GetMach3MainWindow. In general it is a good practice to name functions according to what they do. This is called 'self documenting code'. Here is the code for GetMach3MainWindow.

```
//-----------------------------------------------------------------------------

HWND GetMach3MainWindow(VOID)

{
    HWND m3 = NULL;

    m3 = FindWindow(NULL,"Mach3 CNC Controller ");

    if (NULL != m3) {

        DbgMsg(("found Mach3 window"));
    }

    return(m3);
}

//-----------------------------------------------------------------------------
```

In order to add this to the library, you must put a 'function prototype' in "Utility.h". This tells the Visual C++ compiler the details of how the function is to be called or invoked. Here is the code for the "Utility.h" header file:

```
// Utility.h

#pragma once

HWND GetMach3MainWindow(VOID);
```

Here is the “Utility.cpp” source code file:

```
// Utility.cpp
// Utility / helper functions for Mach3 plugin authoring

#include "stdafx.h"

#include "Utility.h"
#include "dbg.h"

//----------------------------------------------------------------------------

HWND GetMach3MainWindow(VOID)

{
    HWND m3 = NULL;

    m3 = FindWindow(NULL,"Mach3 CNC Controller ");

    if (NULL != m3) {

        DbgMsg(("found Mach3 window"));
    }

    return(m3);
}

//----------------------------------------------------------------------------
```

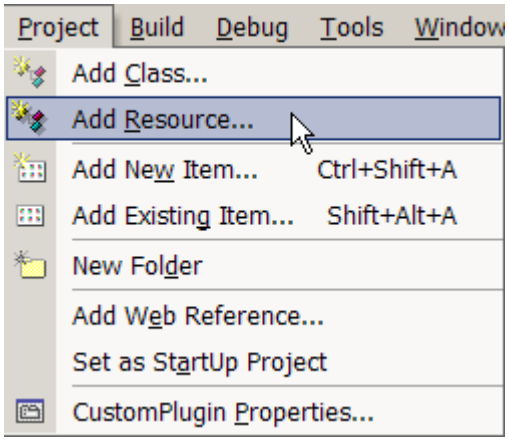
Please note the addition of the three header files at the beginning. They are:

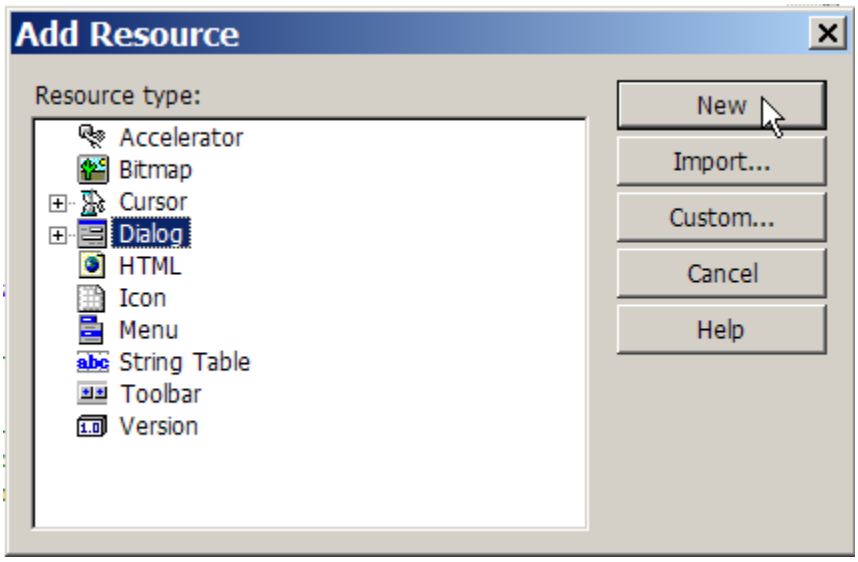
```
#include "stdafx.h"

#include "Utility.h"
#include "dbg.h"
```

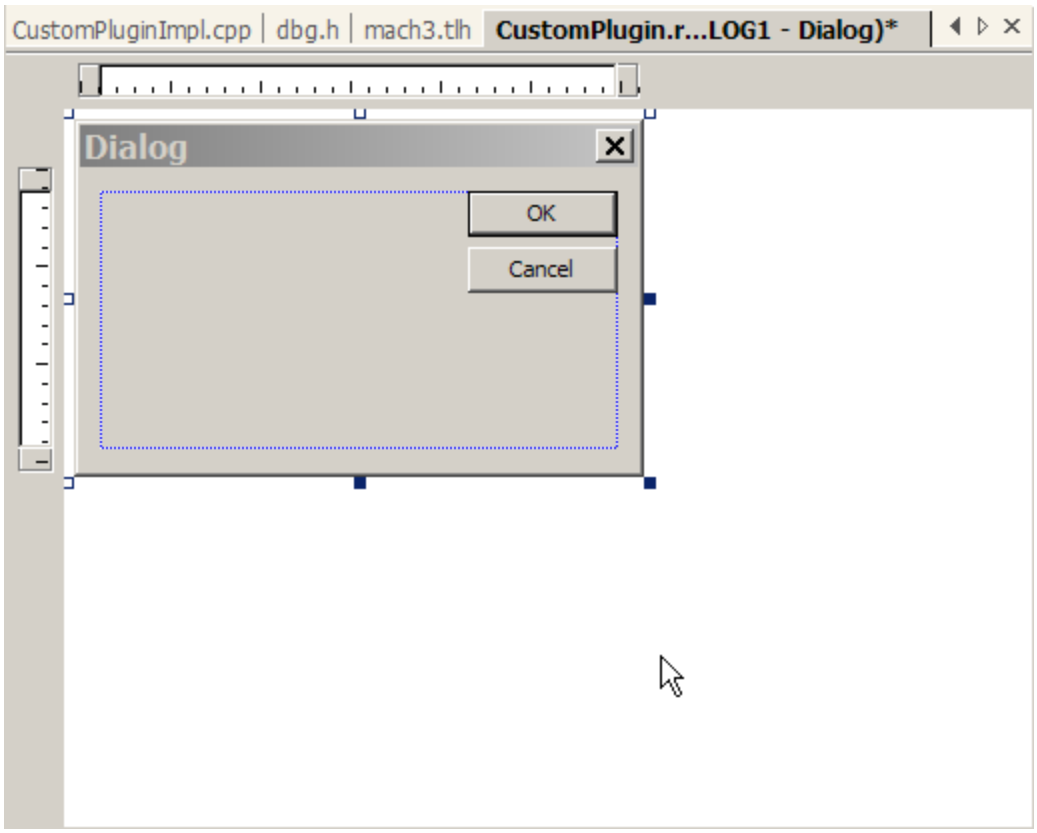
These are needed for correct compilation of this library source file.

Adding A Dialog To The Custom Plugin

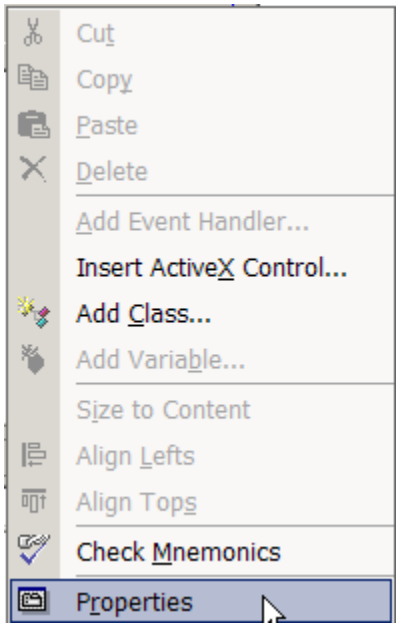
Many plugins will require a dialog for configuration data, or other uses. In order to add a dialog and a class file to use it you start by creating the dialog with the resource editor.



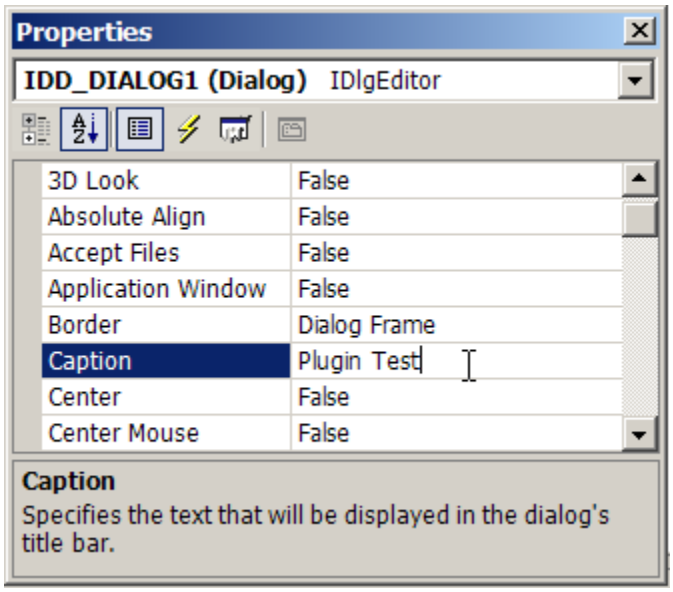


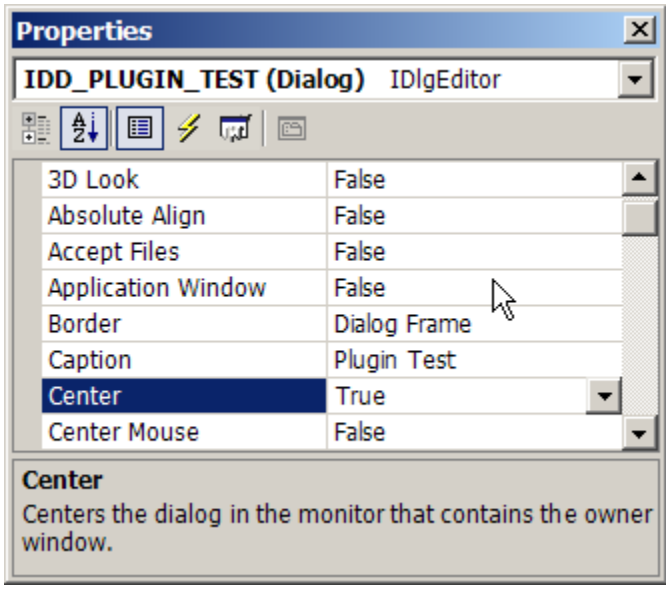


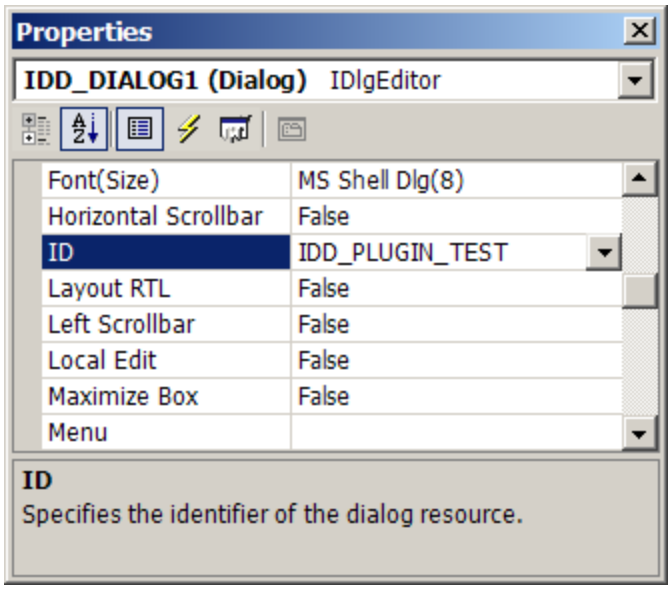
Change the properties of the dialog so the caption (title bar) says ‘Plugin Test’ and the ID of the dialog is IDD_PLUGIN_TEST.



There are three properties that need to be changed. They are “Caption”, “Center” and “ID”. I am showing each one individually.





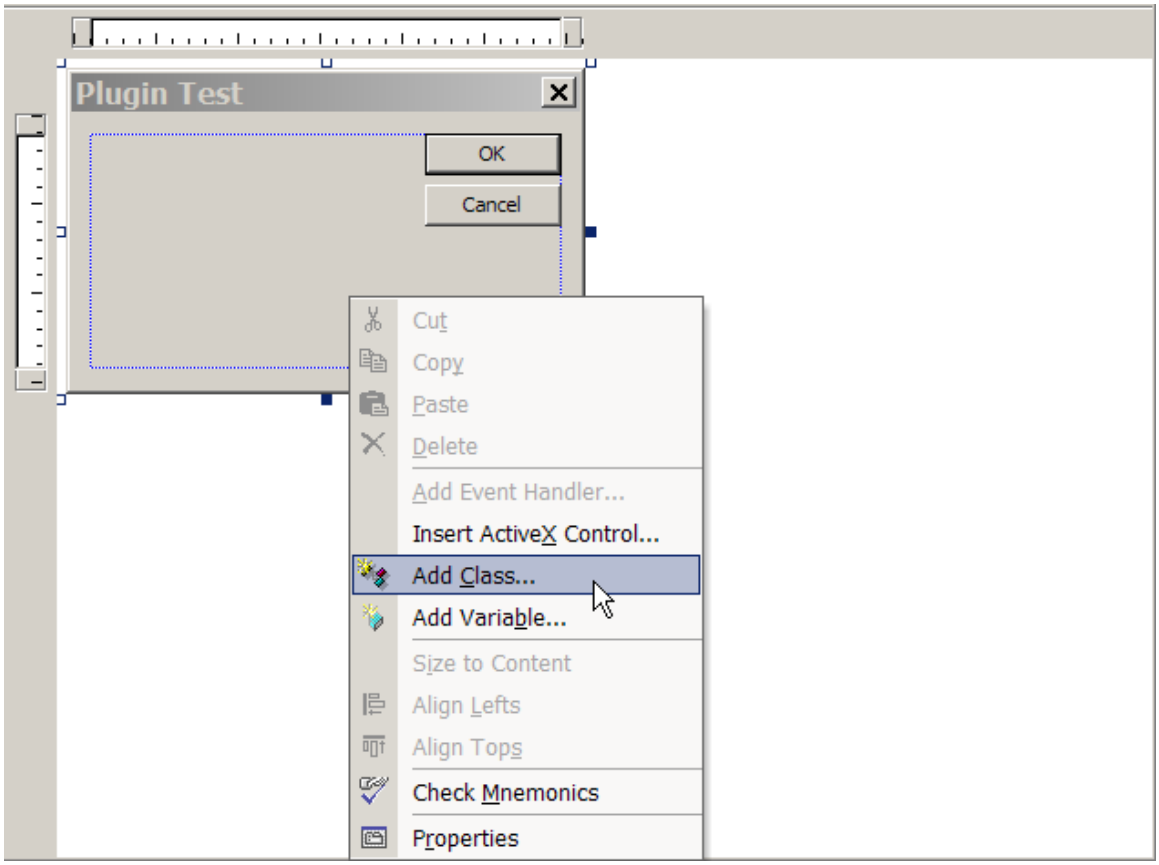


Close the properties editor by clicking on the 'X' in the upper right hander corner. This will save all the changes that you made.

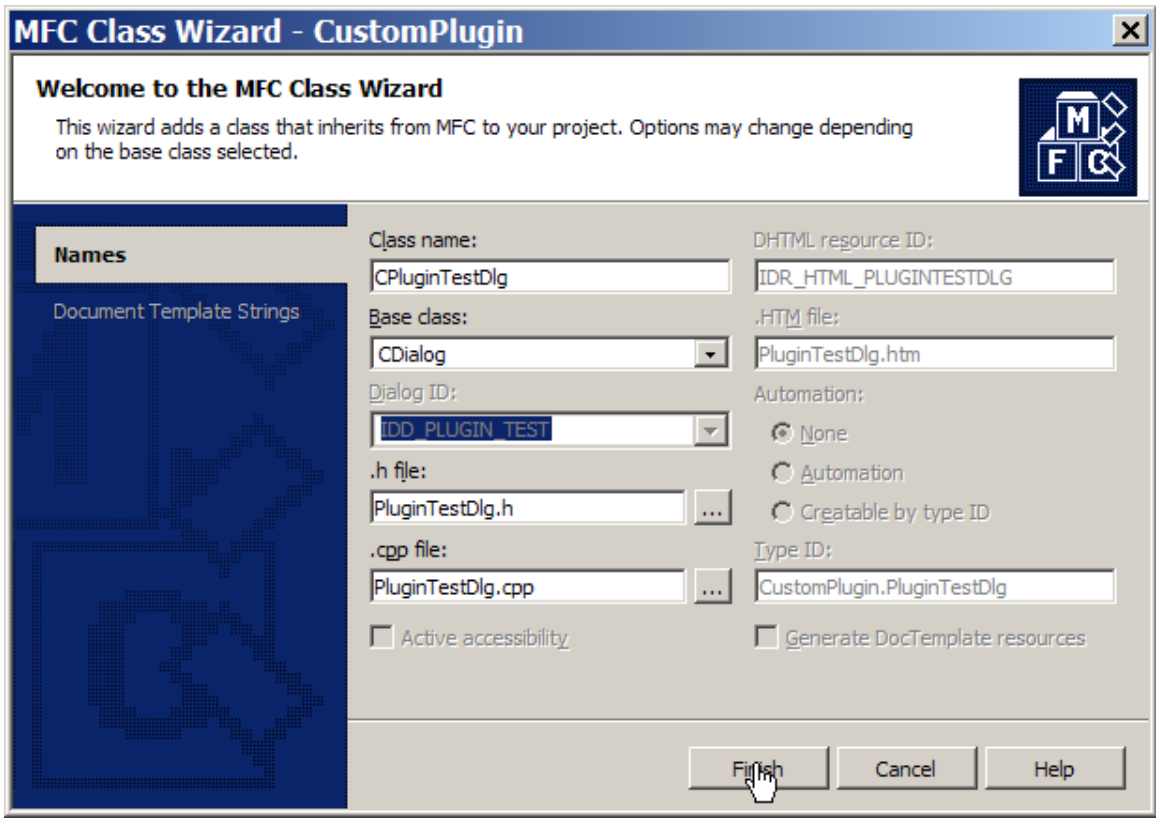
Adding a MFC Class to Use the Dialog

In order to display the dialog and use the buttons and other controls that we will add we must create a C++ class that is derived from the MFC base class 'CDialog'. I will add a reference section with links to what all that means, so for now just follow along and we will end up with a reliable modeless dialog that can use both the Intrinsic functions and the Object Model methods and properties.

Right click on dialog and select ‘Add Class’.



This screen will appear. Select “CDialog” for the “Base class” then enter CPluginTestDlg in the “Class name” edit box. Note that it is a Microsoft convention that all class names should begin with ‘C’ (it means class). It is a good idea to follow this. After you have entered the class name you will see that the “.h file” and “.cpp file” edit boxes are now filled in for you. Press finish to create these files.



This results in two new files, PluginTestDlg.cpp and PluginTestDlg.h. They will be used to create a modeless dialog.

What is a ‘modeless dialog’? It is a popup screen that sits on top of the Mach3 main screen. It is ‘modeless’ since it does not stop you from using the menus and buttons on Mach3’s main screen. This is the best way to enter data from a plugin, but modeless dialogs are much more trouble to manage. So, we are establishing a set of rules for creating and managing these dialogs inside of a Mach3 plugin.

PluginTestDlg.cpp needs to have additional headers (.h files) added for integration with the CustomPlugin project and to use the DbgMsg system. When you are done this section of PluginTestDlg.cpp looks like this:

```
#include "stdafx.h"
#include "resource.h"
#include "CustomPlugin.h"
#include "PluginTestDlg.h"
#include "dbg.h"
```

```
If the AppWizard adds a redundant header ".\plugintestdlg.h" it may be
removed or left in as you wish.  Other that these headers,
"PluginTestDlg.cpp" can be used as it was generated by the AppWizard.
```

The "PluginTestDlg.h" file does not need any changes at this time.

Since we have added four new files (Utility.cpp, Utilty.h PluginTestDlg.cpp and PluginTestDlg.h) it is a good idea to try a test build now to make sure that all has gone well for these steps. If it has not, please review this section up to this point to see what has gone wrong.

If all went well with your test build, now we can add a little more code and make the Plugin Test dialog display from Mach3.

The first things we must add are the declarations for the HWND and CWnd that will hold the Mach3 main window handles. The HWND is a fundamental windows 'type' and the CWnd is a MFC class that encapsulates a HWND and provides added functionality,

In the file CustomPluginImpl.cpp add these lines after the header files section:

```
HWND mach3Wnd;
CWnd mach3CWnd;
```

```
Now we must add the declaration of the Plugin Test dialog class so that
we can make an 'instance' of it.  We will explain further at that
point.
```

```
CPluginTestDlg *dlg;
```

```
We also add the header file "Utility.h" to the header file section so
we can use the utility function "GetMach3MainWindow".  And we also need the
header file "PluginTestDlg.h" to declare the CPluginTestDlg class itself.
```

Here is the finished upper section of CustomPluginImpl.cpp.

```
// CustomPluginImpl.cpp

#include "stdafx.h"
#include "resource.h"
#include "TrajectoryControl.h"
#include "Mach4View.h"
#include "Engine.h"
#include "rs274ngc.h"
#include "XMLProfile.h"

#include "CustomPluginImpl.h"
#include "PluginTestDlg.h"
#include "Utility.h"
#include "dbg.h"

#include <mmsystem.h>
#include <math.h>

HWND mach3Wnd;
CWnd mach3CWnd;

CPluginTestDlg *dlg;

// ============================================================================
```

In the “myPostInitControl()” function in CustomPluginImpl.cpp we add the following code. This will make a new ‘instance’ of the CPluginTestDlg class and call the “Create() method to create an invisible dialog box. We want the dialog to be invisible for now so that we can show and hide it on command from Mach3.

```
//----------------------------------------------------------------------------

void myPostInitControl()

{
    // called when mach fully set up so all data can be
    // used but initialization outcomes are not affected

    DbgMsg(("myPostInitControl entry"));

    mach3Wnd = GetMach3MainWindow();

    mach3CWnd.Attach(mach3Wnd);

    dlg = new CPluginTestDlg;

    dlg->Create(IDD_PLUGIN_TEST,&mach3CWnd);

    DbgMsg(("myPostInitControl exit"));
}

//----------------------------------------------------------------------------
```

Let's look at each line of code in turn since this is a very important function and the first real programming that we have done in this tutorial.

```
DbgMsg(("myPostInitControl entry"));
```

This simply outputs the string "myPostInitControl entry" to the DebugView console so that we know where we are in the flow of the plugin initialization.

```
mach3Wnd = GetMach3MainWindow();
```

This calls the utility function "GetMach3MainWindow()" which looks up Mach3's main window handle (HWND) and then returns it to us here.

```
mach3CWnd.Attach(mach3Wnd);
```

This takes the HWND that was returned and attaches it to the CWnd class object. Now the CWnd is fully prepared for making a new "instance" of CPluginTestDlg.

```
dlg = new CPluginTestDlg;
```

This uses the "new" operator (built-in C++ language feature) to allocate memory on the "heap" for "dlg" which is an "instance" of CPluginTestDlg.

```
dlg->Create(IDD_PLUGIN_TEST,&mach3CWnd);
```

This calls the "Create" object method which creates the actual dialog box from the template named by IDD_PLUGIN_TEST. This template is in the file "CustomPlugin.rc" and was created when we created dialog at the beginning of the "Adding A Dialog To The Custom Plugin" section.

```
DbgMsg(("myPostInitControl exit"));
```

This outputs the string "myPostInitControl exit" to the DebugView console so that we know that we are done with the plugin initialization.

This takes care of 'startup' processing for our modeless dialog. Now we need to look at 'shutdown' processing since we always need to clean up our use of resources and memory. In this case failure to do will crash Mach3 on exit (AND maybe give you the Blue Screen Of Death also).

In “myCleanUp” we release all the memory and resources used by the modeless dialog that we created. Here is what it looks like.

```
//------------------------------------------------------------------------

// Used for destruction of variables prior to exit.
// Called as Mach3 shuts down.

void myCleanUp()

{
    DbgMsg(("myCleanUp entry"));

    dlg->DestroyWindow();

    mach3CWnd.Detach();

    delete dlg;

    DbgMsg(("myCleanUp exit"));
}

//------------------------------------------------------------------------
```

And this deserves a line-by-line explanation also.

```
    DbgMsg(("myCleanUp entry"));
```

This is output to DebugView and shows that this function has been entered.

```
    dlg->DestroyWindow();
```

This destroys the modeless dialog box but does not free the memory for the “dlg” object.

```
    mach3CWnd.Detach();
```

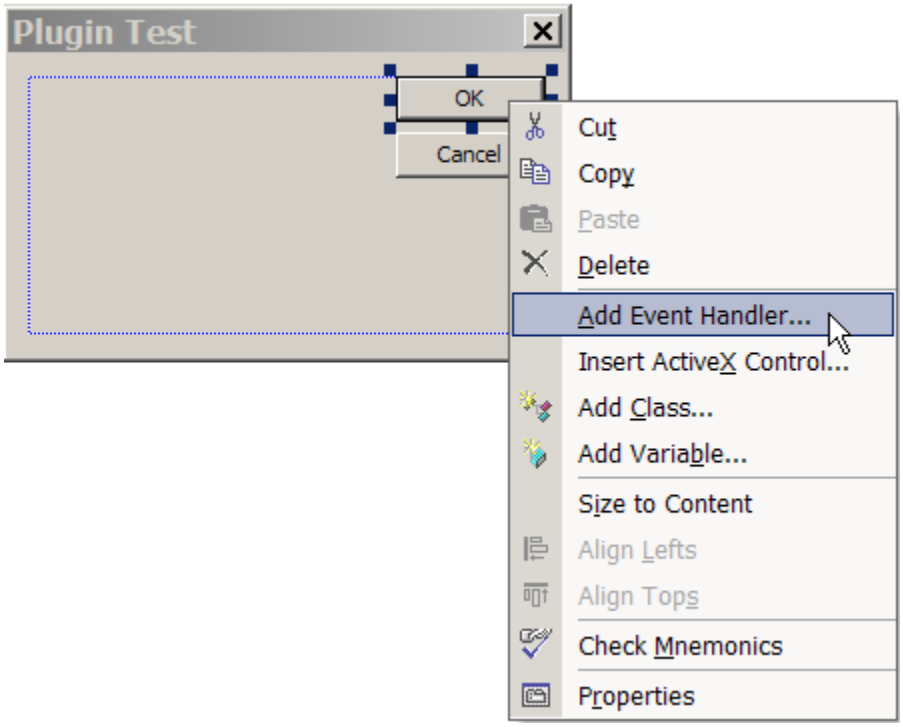
This detaches the Mach3 main window handle from the Cwnd that was used to create the dialog box. If this is not done Mach3 will crash on exit.

```
    delete dlg;

This free the memory for the CPluginTestDlg object that was allocated by the
statement “dlg = new CPluginTestDlg”.  ‘new” and “delete” MUST always be used
in pairs or memory will NOT be freed.  This is called a “memory leak”.


    DbgMsg(("myCleanUp exit"));
```

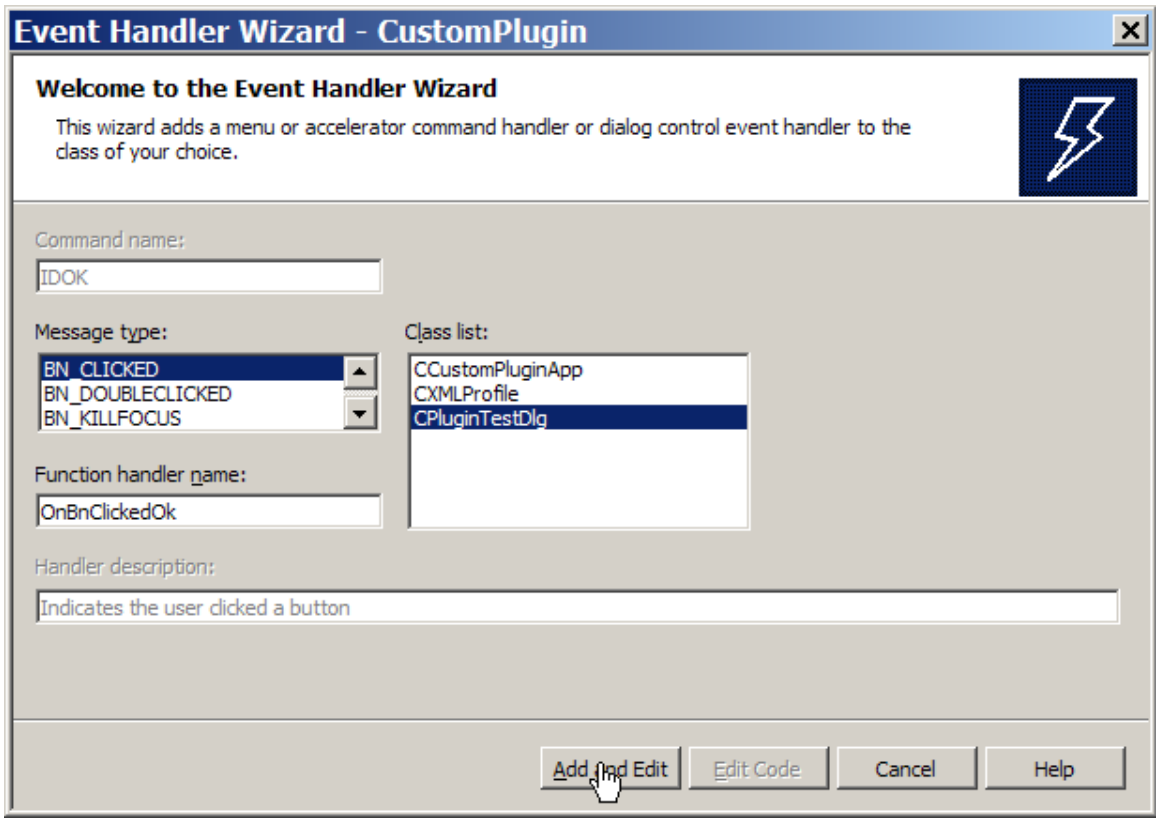
This is output to DebugView and shows that this function is now complete and will exit..

Adding Button Handlers to the Dialog

In order to use the buttons on the plugin test dialog you must add “Event Handlers” to the CPluginTestDlg class. Open the source editor and display the ‘Plugin Test’ dialog. Right click on the ‘OK’ button and select ‘Add Event Handler’.



Make sure that you have “BN_CLICKED” and “CPluginTestDlg” selected. Press the “Add And Edit” button when you are ready.



Edit the AppWizard generated code so that ‘OnOk()’ is not called. Instead we will just hide this window for both the ‘OK’ and ‘Cancel’ buttons.

```
void CPluginTestDlg::OnBnClickedOk()
{
    DbgMsg(("OnBnClickedOk entry"));

    ShowWindow(SW_HIDE);

    DbgMsg(("OnBnClickedOk exit"));
}
```

Add a handler for the ‘Cancel’ button and put the same changes in the ‘OnBnClickedCancel’ handler code.

```
void CPluginTestDlg::OnBnClickedCancel()
{
    DbgMsg(("OnBnClickedCancel entry"));

    ShowWindow(SW_HIDE);

    DbgMsg(("OnBnClickedCancel exit"));
}
```

The last thing we have to do before we can test all of this is to modify the "myConfig" function in CustomPluginImpl.cpp. Here is what looks like.

```
//-----------------------------------------------------------------------------

// Called to configure the device
// Has read/write access to Mach XML profile to remember what it needs to.

void myConfig(CXMLProfile *DevProf)

{
    DbgMsg(("myConfig entry"));

    dlg->ShowWindow(SW_SHOW);

    DbgMsg(("myConfig exit"));
}

//-----------------------------------------------------------------------------
```

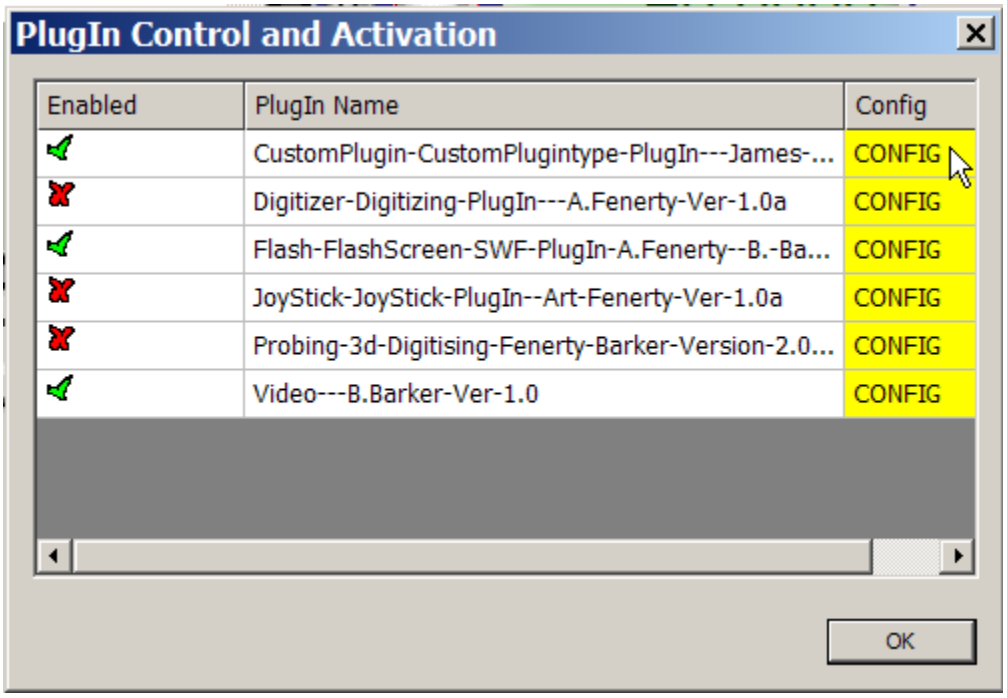
Ok, we have the standard DbgMsg entry and exit trace functions. Not much more detail needed for those.

```
    dlg->ShowWindow(SW_SHOW);
```

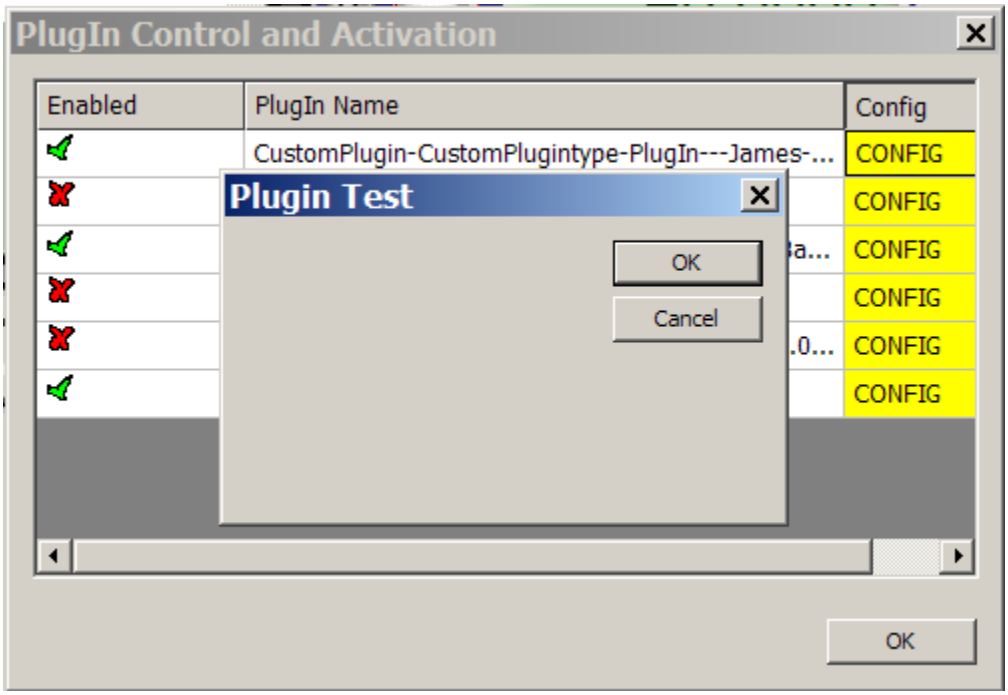
This line shows (makes visible) the modeless dialog by calling the ShowWindow method in the CWnd base class for CDialog.

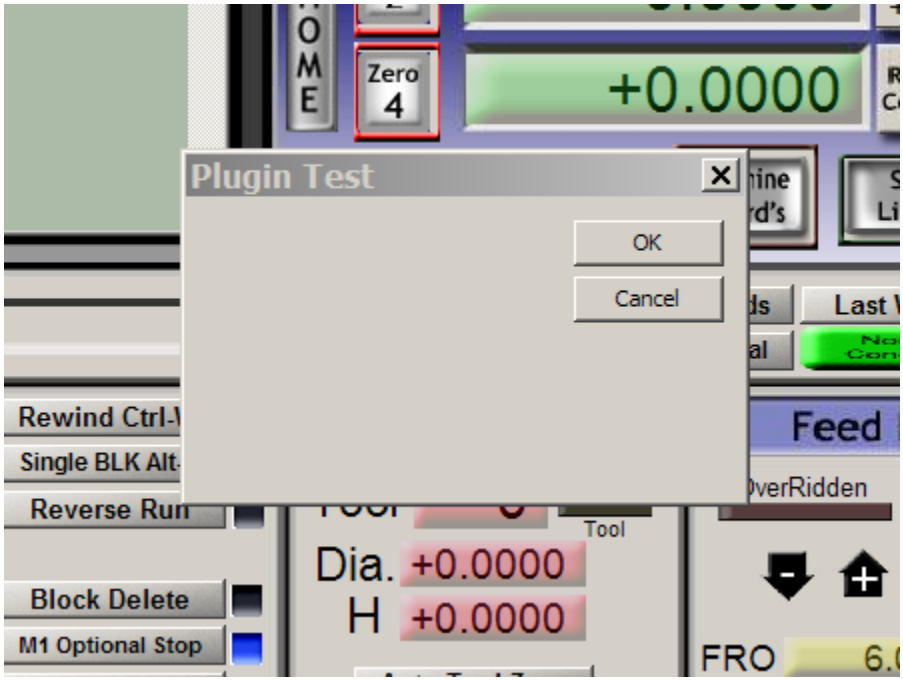
Testing The Dialog

Now it is time to test your dialog. Build the Debug version of the CustomPlugin.dll and copy it to the Mach3 plugins folder like you did earlier. If all was well with the build and copy you can then start Mach3 and open the "Plugin Control and Activation" dialog in Mach3. If you press the big yellow CONFIG button on the right hand side AND everything was done well you will see your modeless dialog pop up in the center on the Mach3 screen.



This is what you should see. Since this is a modeless dialog you can now close the "Plugin Control and Activation" dialog without affecting the "Plugin Test" dialog at all.

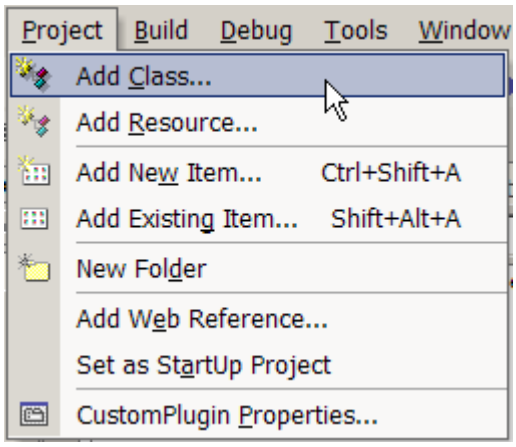




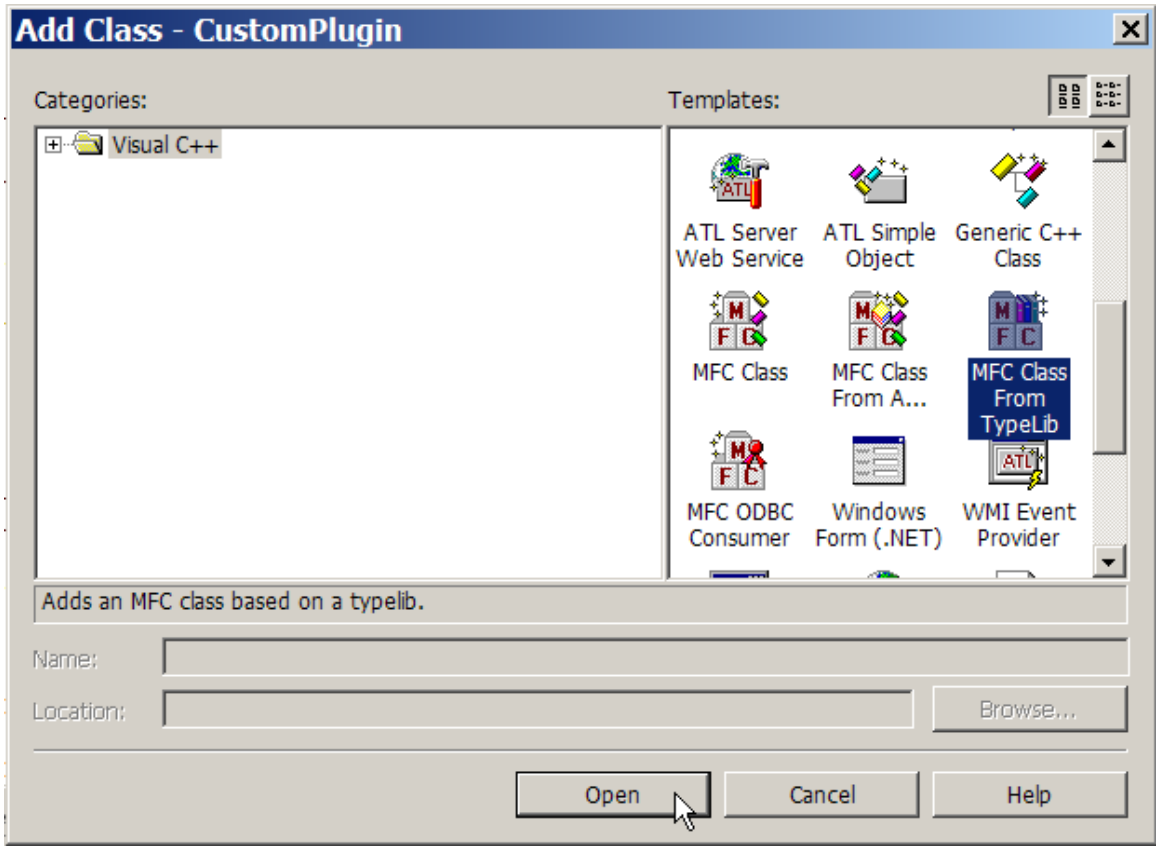
Adding Object Model Support

The Mach3 Object Model is the set of programming interfaces and data that supports the Mach3 script facility (the "macros"). Support has been added for external program access to this object. This also works for plugins and now gives access to ALL of the programmability that Mach3 offers the system integrator / customizer.

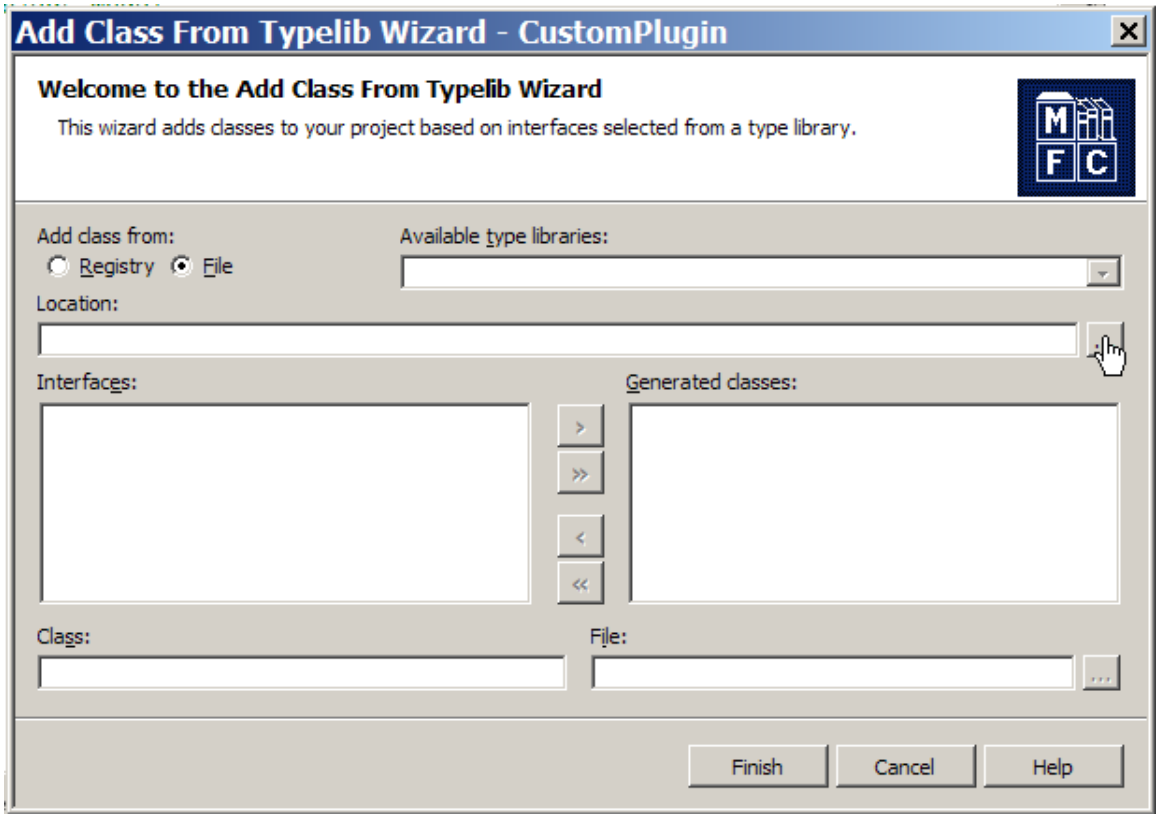
Mach3 has an embedded TypeLib that completely define all of the methods and properties that the Mach3 object model makes available. In order to use this in the CustomPlugin we need to create the header (.h) files that define these interfaces and provide helper functions for using them. We start with the "Project->Add Class" menu.



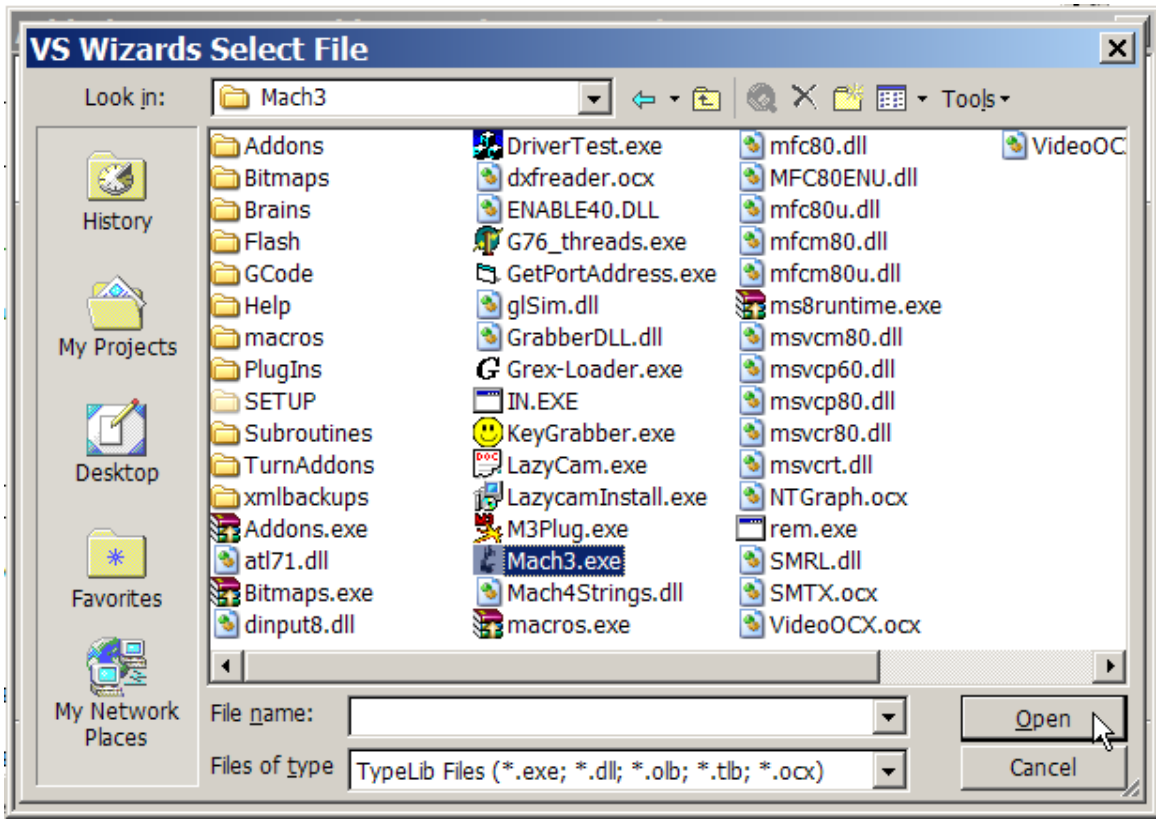
We now can specify ‘MFC Class From TypeLib”.



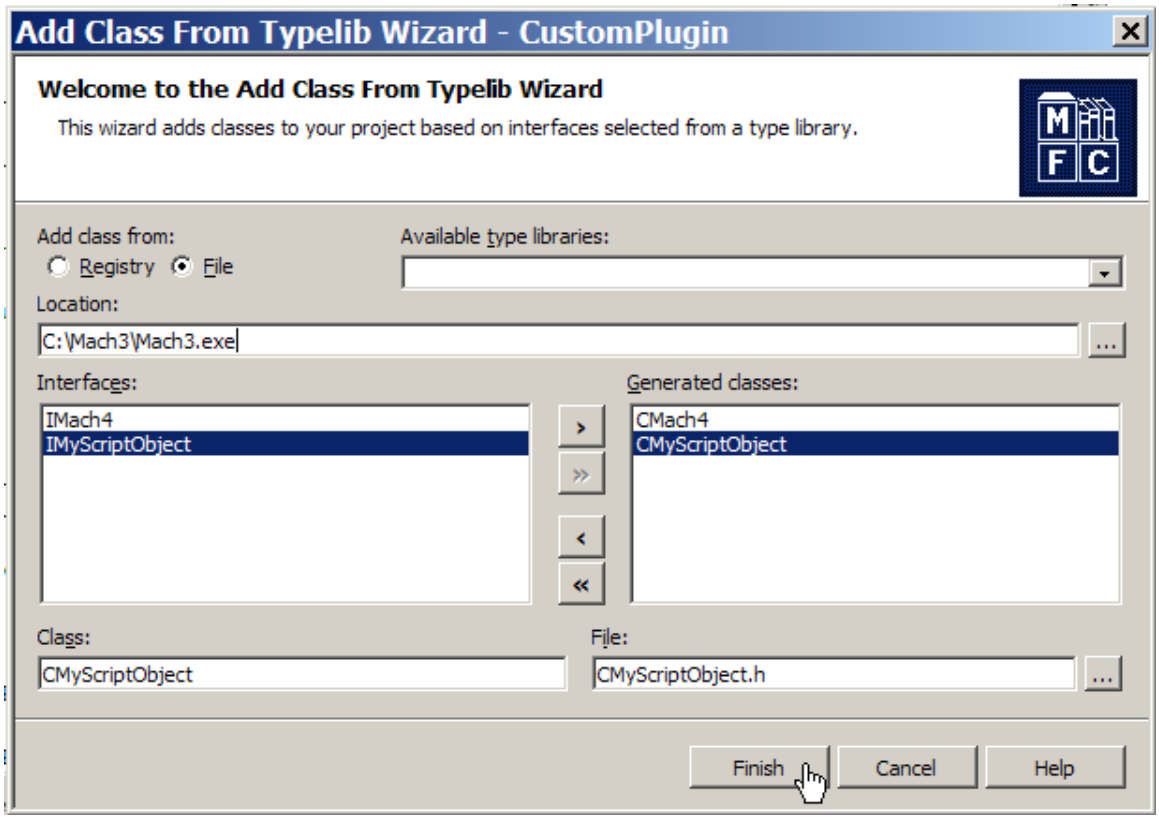
This display this dialog. Select the “Add class from File” radio button and then press the button to the right of the “Location” edit box.



Select the file ‘Mach3.exe’ and then press the “Open” button



Now you can select both available interfaces “IMach4” and “IMyScriptObject” and add then to the “Generated Classes” window with the “>>” button



Press the “Finish” button and two files will be generated, “CMach4.h” and “CMyScriptObject.h”. Add these files to the CustomPlugin project. These files will be used by the Object Model Library that we will now create.

Since you now have quite a bit of experience with Mach3 plugins we will start referring to sections that have come before. Initially this will all be very similar to what was done to add the Utility Library support, so see that section for how to create new files, name them and add them to the CustomPlugin project. The files that you will create will be “Mach3ObjectModel.cpp” and “Mach3ObjectModel.h”.

After you create these files add this code to the “Mach3ObjectModel.h” file.

```
// Mach3ObjectModel.h
// This library allows access to the Mach3 scripting engine
// object model

VOID Mach3ObjectModelStartup(VOID);
VOID Mach3ObjectModelShutdown(VOID);
BOOL LoadGcodeFile(CHAR *filePath);
BOOL RunGcodeFile(CHAR *filePath);
BOOL CloseGcodeFile(VOID);
BOOL CycleStart(VOID);
BOOL PushOEMButton(short button);
```

And add all of this code to the “Mach3ObjectModel.cpp” file.

```
// Mach3ObjectModel.cpp
// This library allows access to the Mach3 scripting engine
// object model

#include "stdafx.h"

#include "CMach4.h"
#include "CMyScriptObject.h"
#include "Mach3ObjectModel.h"
#include "Mach3DRO.h"
#include "Mach3Button.h"
#include "dbg.h"

CMach4 mach4;
CMyScriptObject scripter;
bool connected = FALSE;
```

```
//-----------------------------------------------------------------------------

VOID Mach3ObjectModelStartup(VOID)

{
    static const IID IID_IMyScriptObject =  { 0xf1d3ee6c, 0xab32, 0x4996,
    { 0xb2, 0x70, 0xf4, 0x15, 0x61, 0x3f, 0x5b, 0xa3 } };
    CLSID clsid;

    PushDbgMode();

    DbgOn
//  DbgOff

    CoInitialize(NULL);

    LPUNKNOWN lpUnk = NULL;
    LPDISPATCH lpDispatch = NULL;
    HRESULT res;

    DbgMsg(("Mach3ObjectModelStartup entry"));

    try {

        if (CLSIDFromProgID(OLESTR("Mach4.Document"),&clsid) == NOERROR) {

            if (res = GetActiveObject(clsid,NULL,&lpUnk) == NOERROR) {

                HRESULT hr = lpUnk->QueryInterface(IID_IDispatch,
                           (LPVOID*)&lpDispatch);

                lpUnk->Release();

                if (hr == NOERROR) {

                    mach4.AttachDispatch(lpDispatch, TRUE);

                    lpDispatch = mach4.GetScriptDispatch();

                    scripter.AttachDispatch(lpDispatch, TRUE);

                    connected = TRUE;

                    DbgMsg(("Mach3 control OK"));
                }
            }
        }
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("Mach3ObjectModelStartup exit"));

    PopDbgMode();
}

//-----------------------------------------------------------------------------
```

```
VOID Mach3ObjectModelShutdown(VOID)

{
    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("Mach3ObjectModelShutdown entry"));

    CoUninitialize();

    DbgMsg(("Mach3ObjectModelShutdown exit"));

    PopDbgMode();
}

//-----------------------------------------------------------------------------

BOOL LoadGcodeFile(CHAR *filePath)

{
    BOOL retVal = FALSE;

    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("LoadGcodeFile entry"));

    try {

        DbgMsg(("LoadGcodeFile: filePath = %s",filePath));

        scripter.LoadFile(filePath);
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("LoadGcodeFile exit"));

    PopDbgMode();

    return(retVal);
}

//-----------------------------------------------------------------------------
```

```
BOOL RunGcodeFile(CHAR *filePath)

{
    BOOL retVal = FALSE;

    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("RunGcodeFile entry"));

    try {

        DbgMsg(("RunGcodeFile: filePath = %s",filePath));

        scripter.LoadRun(filePath);
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("RunGcodeFile exit"));

    PopDbgMode();

    return(retVal);
}

//-----------------------------------------------------------------------------

BOOL CloseGcodeFile(VOID)

{
    BOOL retVal = FALSE;

    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("CloseGcodeFile entry"));

    try {

        scripter.DoOEMButton(CLOSE_GCODE);
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("CloseGcodeFile exit"));

    PopDbgMode();

    return(retVal);
}

//-----------------------------------------------------------------------------
```

```
BOOL CycleStart(VOID)

{
    BOOL retVal = FALSE;

    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("CycleStart entry"));

    try {

        scripter.DoOEMButton(CYCLE_START);
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("CycleStart exit"));

    PopDbgMode();

    return(retVal);
}

//-----------------------------------------------------------------------------

BOOL PushOEMButton(short button)

{
    BOOL retVal = FALSE;

    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg(("PushOEMButton entry"));

    try {

        DbgMsg(("PushOEMButton: button = %d",button));

        scripter.DoOEMButton(button);
    }

    catch(_com_error &e) {

        com_error_msg(e);
    }

    DbgMsg(("PushOEMButton exit"));

    PopDbgMode();

    return(retVal);
}
```

```
//-----------------------------------------------------------------------
```

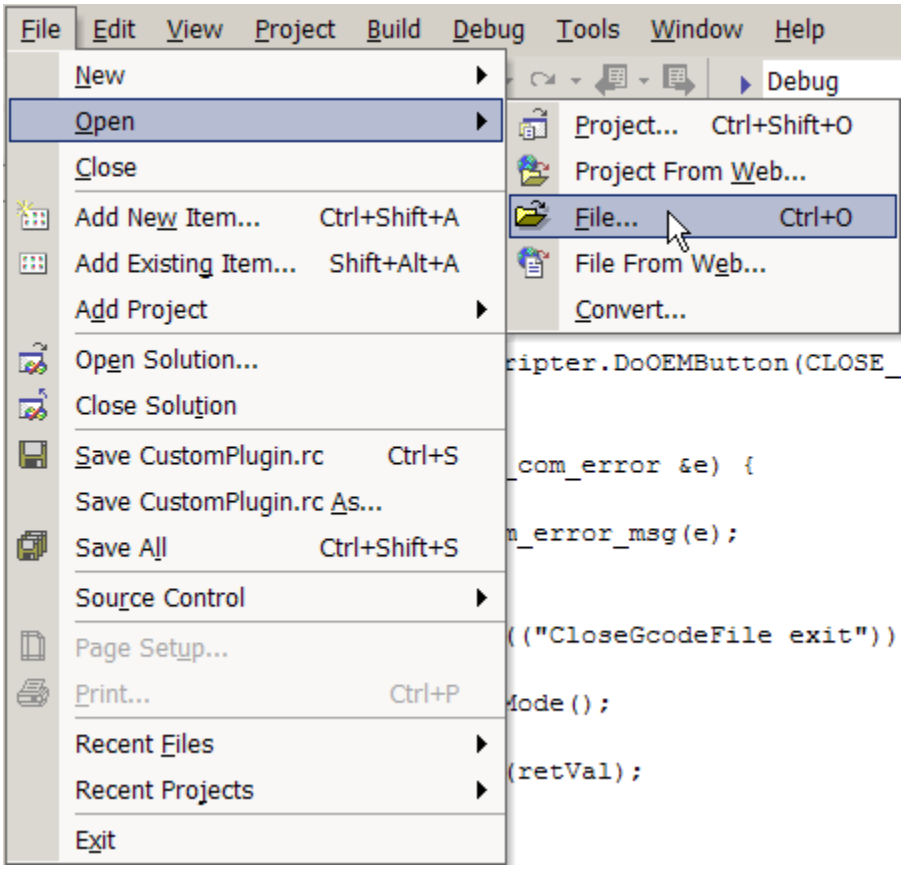
Using the Intrinsics and Object Model to Actually Do Something

Well, that is probably the most whimsical section title so far but a perfect description! We are going to add buttons and code to our test modeless dialog to make Mach3 ready to run, then load and run the infamous roadrunner.tap GCODE file.

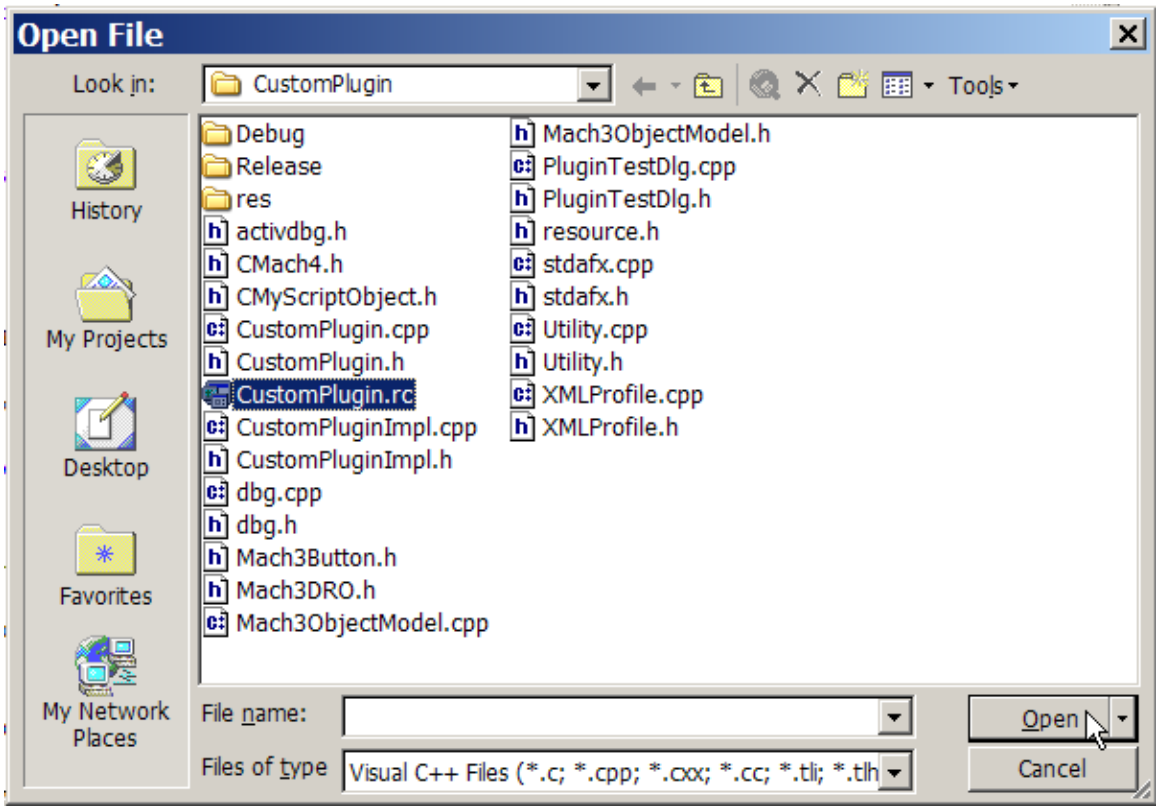
// FIXIT
Mach3ObjectStartup

Mach3ObjectShutdown

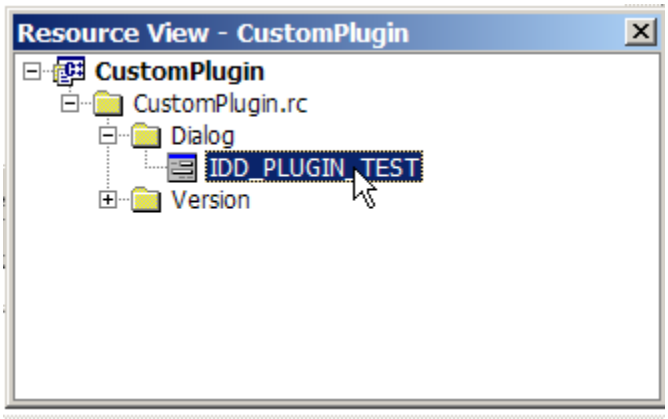
First, open the dialog and add three new buttons. They will be labeled “Reset”, “Load RR” and “Run RR”.



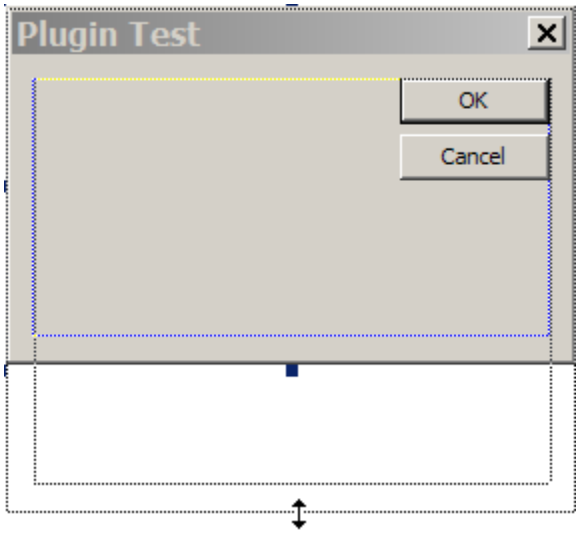
The dialog is contained in the file CustomPlugin.rc

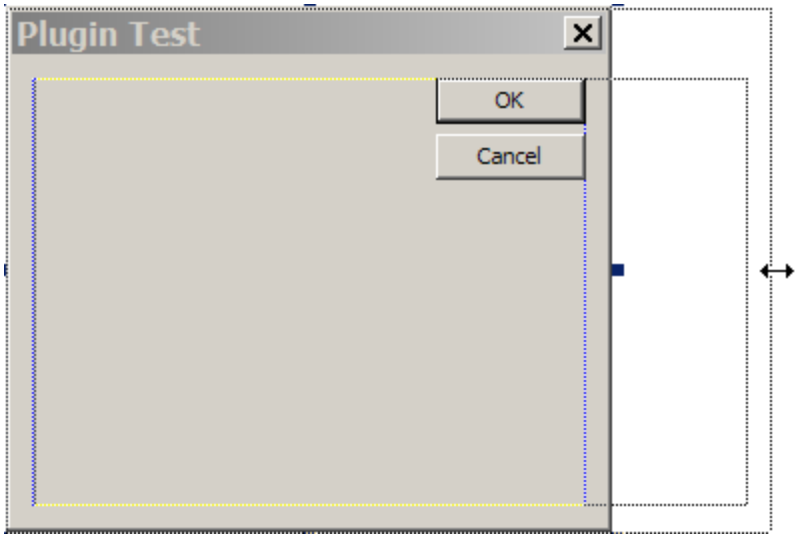


When you open CustomPlugin.rc you will see this dialog. Double click on IDD_PLUGIN_TEST .

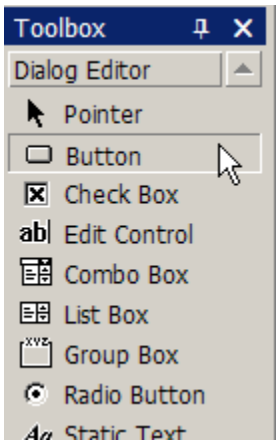


Now you can stretch the “Plugin Test” dialog so that we can easily add some new buttons.

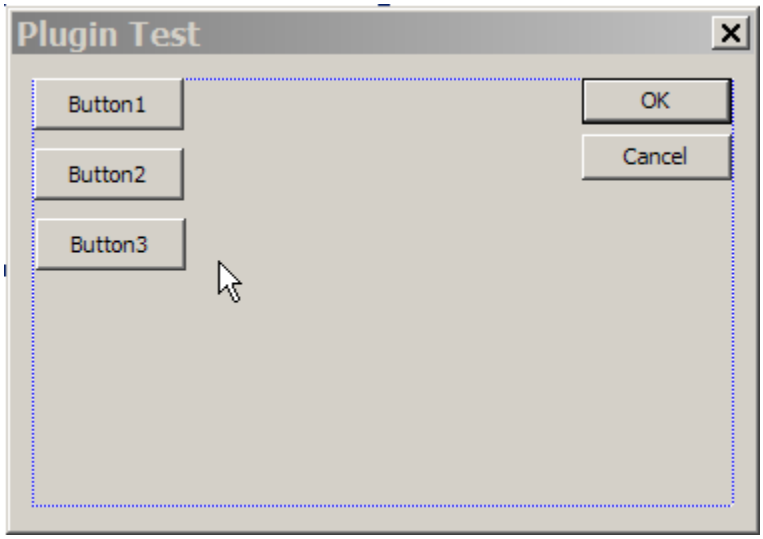




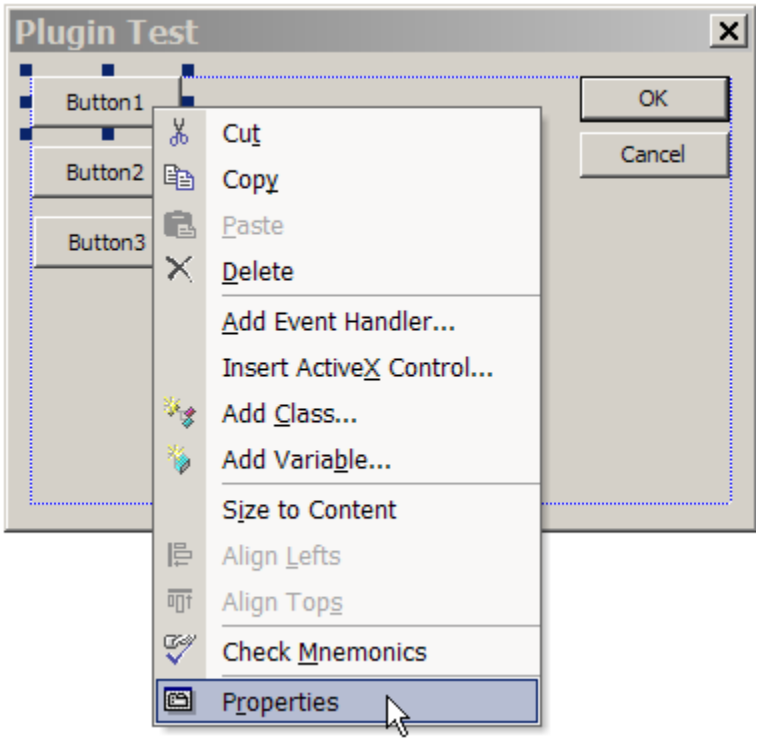
After stretching the “Plugin Test” dialog select the ‘Button’ tool from the toolbox on the left.



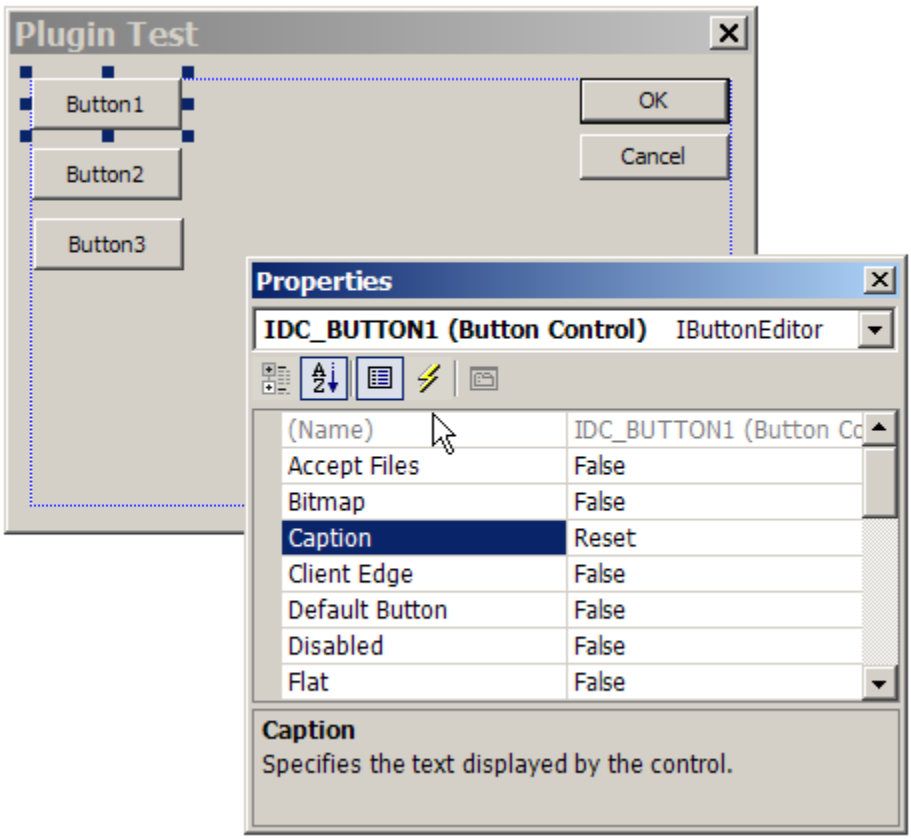
Click on the dialog where you want your button to be located. The default button size is 50 x 14 but I like 50 x 16 usually so I make my buttons a little taller.



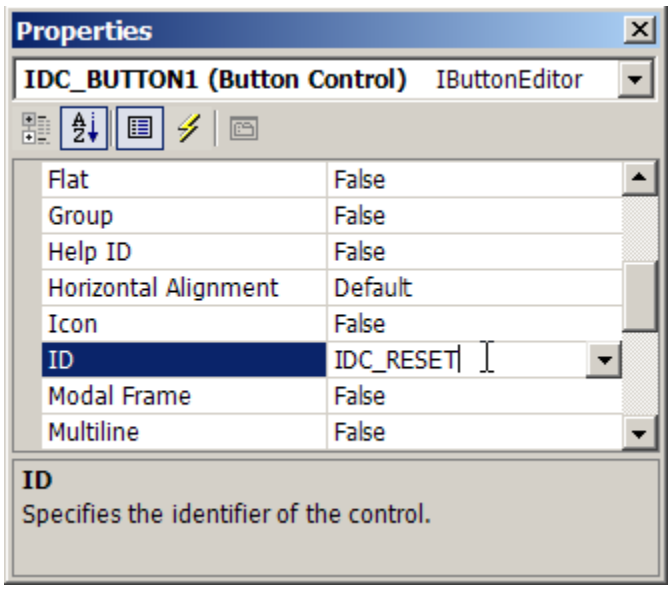
After you have added all three buttons you need to see the button caption and the ID. These are both button properties, so we right click on the first button and select "Properties" from the menu.



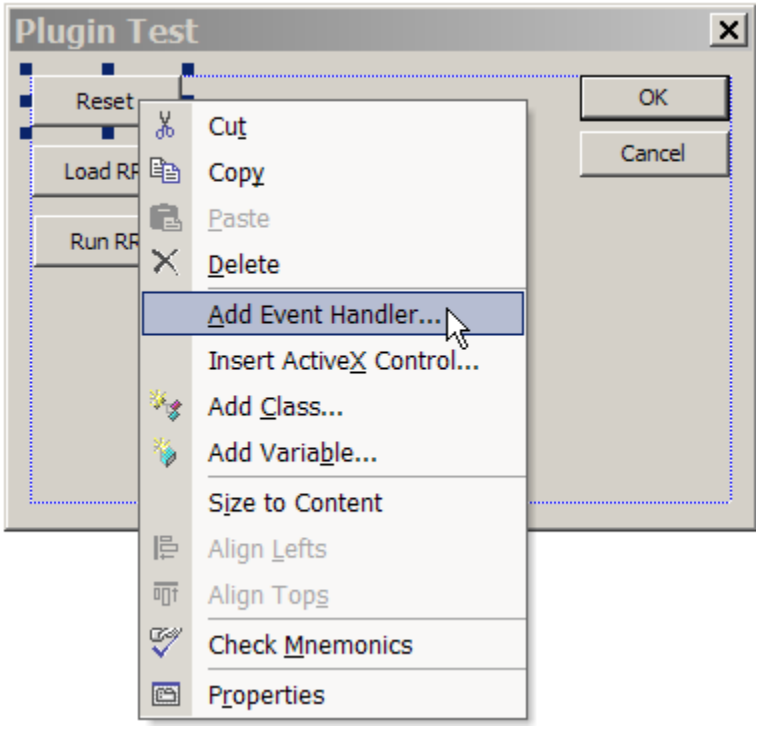
Set the caption of the first button to “Reset”.



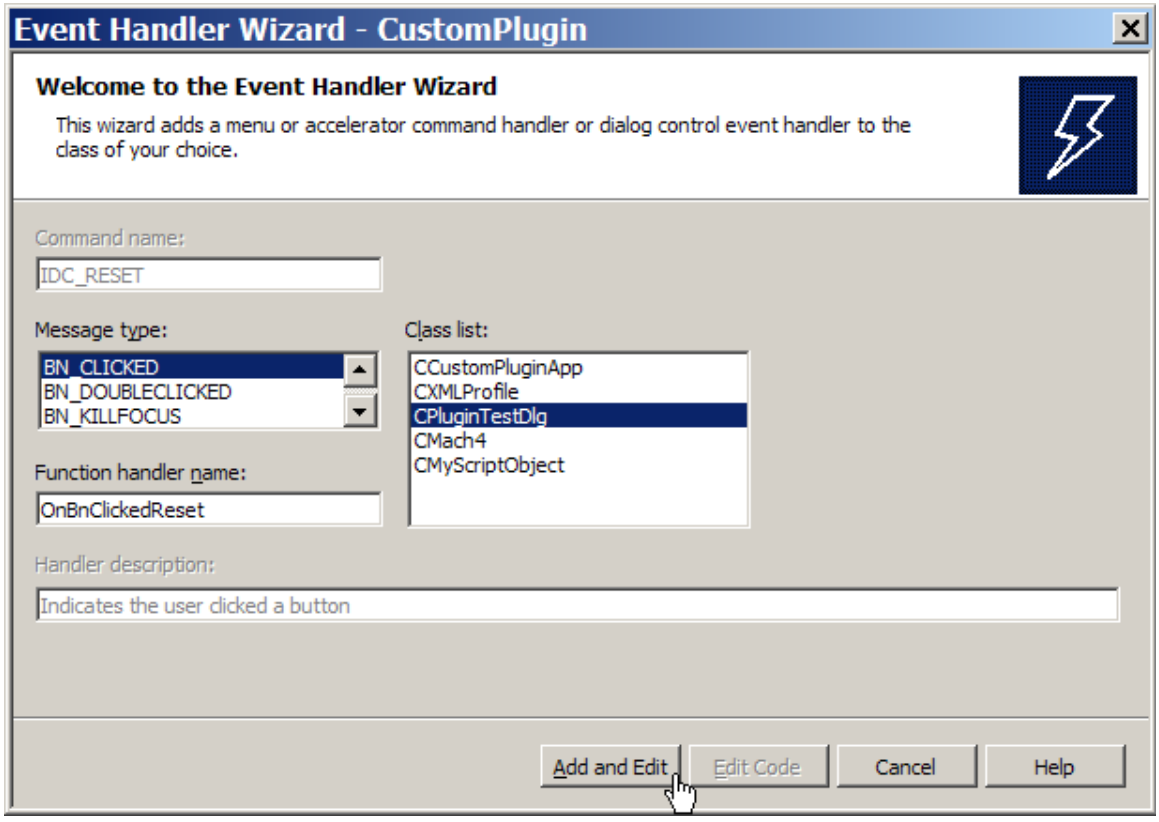
And set the ID to IDC_RESET.



Add an event handler for the “Reset” button.



Select BN_CLICKED and CPluginTestDlg then press “Add and Edit”.



Now you can add the code to make this button perform a “Reset” operation on Mach3. This is done by using the DoButton Intrinsic to push the Mach3 “Reset button”.

```
void CPluginTestDlg::OnBnClickedReset()
{
    DoButton(MACH3_RESET);    // Reset (1021)
}
```

Now set the button captions and ID values for the next two buttons. They will be:

Button 2 “ Load RR” IDC_LOAD_RR

Button 3 “Run RR” IDC_RUN_RR

Then add event handlers for these buttons.

```
void CPluginTestDlg::OnBnClickedLoadRr()
{
    scripter.LoadFile("C:\\Mach3\\Gcode\\RoadRunner.tap");
}

void CPluginTestDlg::OnBnClickedRunRr()
{
    DoButton(CYCLE_START);
}
```

Finished File Sets

There are xxx finished files sets. They represent the progress made at each stage of this tutorial.

CustomPlugin_V01.zip is the tutorial up to doing the first release build.

CustomPlugin_V02.zip is the next steps to add a modeless dialog.

To Be Enhanced

Plugin Development Reference

This section documents the currently available intrinsic functions and object model methods and properties that can be used to develop Mach3 plugins.

Mach3 Plugin Intrinsic Functions

A Mach3 plugin intrinsic is a function that is initialized by Mach3 when the plugin is loaded and that may be called directly. Note that the names of the intrinsics are arbitrary, I.E. they are the named function pointers declared that are set to the internal functions in Mach3. The 'Set' functions that are exported from the plugin DLL are NOT arbitrary names since Mach3 needs to 'soft' link to the functions in order to set the function pointers. This list will document the standard function pointer names that were originally conceived by John Prentice. It also shows the associated exported DLL functions.

Function Pointer	Exported Function
DoButton	SetDoButton
SetDRO	SetSetDRO
GetDRO	SetGetDRO
SetLED	SetSetLED
GetLED	SetGetLED
Code	SetCode

These intrinsic are used by just calling them as if they were a function in the current source code. I.E. DoButton(CYCLE_START); emulates a 'push' (mouse click) on the Cycle Start button on Mach3's main screen.

Each of these intrinsics is documented here

DoButton(SHORT button)

Pushes the button that is specified by the parameter 'button'. See the list of button codes for details as to this parameter.

SetDRO(SHORT dro, DOUBLE value)

Sets the value into the Mach3 DRO specified by the dro parameter. Note that you CANNOT set the system DROs in this manner. You can only set USER DROs with new values. Mach3's DROs reflect the internal state of Mach3 and only change when that state changes. I.E. the X position DRO can only be affected by changing the current X axis position. See the list of DRO codes for details as to the dro parameter.

DOUBLE GetDRO(SHORT dro)

Gets (returns) the current value of the specified DRO as a double precision floating point value. In general you shoud try to use the engine block variables instead of reading the DROs directly since the engine data is always current, the DROs may lag somewhat. See the list of DRO codes for details as to the dro parameter.

To Be Enhanced

Mach3 Plugin Control and Utility Functions

Mach3 also calls exported functions in the plugin to perform various tasks such as start up, shut down and reset. These functions are called asynchronously, I.E. Mach3 calls them when the action is performed in Mach3 with no apparent delay. This is especially important for the reset control function since this signals either the sudden stoppage of Mach3 or it's availability to start running.

To Be Enhanced

Mach3 Engine Block Data

Mach3 sets a pointer to the Engine data in a variable named Engine. This allows you to directly access the main Mach3 engine runtime parameters. You may freely read any data contained here, but writing data back to the engine block must be done VERY carefully (if at all).

To Be Enhanced

Mach3 Object Model

The Mach3 Object Model is a set of methods and properties that were originally developed for the Mach3 scripting environment. Through the use of the 'GetDispatchPtr' method all of the object model is available for use inside a Mach3 plugin.

To Be Enhanced

The DbgMsg Library

The DbgMsg facility is a set of functions that were developed for DLL debugging based upon my experiences as a Windows CE embedded software developer. The main point to grasp about the DbgMsg API is that it can be used as much as you wish throughout your code without adding the overhead of the strings to a release build. I.E. ALL DbgMsg strings are completely removed from a release build giving zero added runtime overhead in your product. There are three message functions in the library currently. DbgMsg, RelMsg and ErrMsg. Here are the functions and notes on their use:

Note that the examples parameters are declared:

CHAR *string_parm;	formatted with %s
WORD integer_parm;	formatted with %d
LONG long_integer_parm;	formatted with %ld
DOUBLE double_parm;	formatted with %f

DbgMsg((“message with parms %s %d”,string_parm, integer_parm));

All three message output functions use the same parameters in the same way. DbgMsg uses printf format specifications, all the formatting that you can find in the online MSDN documentation is valid..

There are helper functions that make DbgMsg use much more versatile. They are:

PushDbgMode(), PopDbgMode(), DbgOn, DbgOff

These functions allow the selective disabling and enabling of message output and the maintenance of the current output states across nested function calls.

DbgOff will stop DbgMsg and output. RelMsg and ErrMsg ALWAYS work, they are unaffected. This applies to debug build only. In release builds only RelMsg, com_error_msg and ApiDbgMsg will output messages to DebugView.

DbgOn will start DbgMsg output again.

PushDbgMode() saves the current debug mode on a 'stack'. This allows you to change the output mode in functions you are calling.

PopDbgMode() retrieves the current debug state from the 'stack'. This restores the LAST output mode that was in use when PushDbgMode() was called.

There are also two special debug messages for dealing with Windows API failures and COM exception handling.

ApiDbgMsg(("message with parms %s %d",string_parm, integer_parm));

ApiDbgMsg is used to display your message and any parameters you format with this message. It then calls GetLastError(), formats the error data returned and appends that to your message. This is provided for ease of debugging Windows API use in the plugins.

com_error_msg(_com_error &e)

This error message routine is used in the 'catch' block of a 'try – catch' pair for implementing exception handling for use with the Mach3 object model.

Now I will demonstrate the use of the DbgMsg Library in a sample pair of functions, FunctionOne and FunctionTwo. FunctionOne is (hypothetically) called by plugin code to p;rovide some service. FunctionTwo is called by FunctionOne to support FunctionOne in some way.

```
//-----------------------------------------------------------------------------------------------------

VOID FunctionOne(ImportantStuff *stuff)

{
    PushDbgMode();

    DbgOn
//  DbgOff

    DbgMsg((“FunctionOne entry”));

    stuff->nice_number = FunctionTwo();

    DbgMsg((“FunctionOne exit”));

    PopDbgMode();
}

//-----------------------------------------------------------------------------------------------------

DOUBLE FunctionTwo()

{
    DOUBLE a_unique_nice_number = 0.0;

    PushDbgMode();

//  DbgOn
    DbgOff

    DbgMsg((“FunctionTwo entry”));

    a_unique_nice_number = (DOUBLE)(rand() * 3.1416));

    DbgMsg((“FunctionTwo exit”));

    PopDbgMode();

    Return(a_unique_nice_number);
}

//-----------------------------------------------------------------------------------------------------
```

This pair of functions shows how the DbgMsg system should be used in plugin functions. Note that even though FunctionTwo disable DbgMsg output with the DbgOff macro, since it calls PushDbgMode() and PopDbgMode() the original operating mode of FunctionOne is restored. So, we get all of the debug messages from FunctionOne and none of the debug messages from FunctionTwo. If we want to see the debug messages from FunctionTwo then we comment out DbgOff and uncomment DbgOn.

Appendix A – Current Versions and Notes

11/17/07

Initial version of this document for:

Mach3 v2.48 and the SDK2.03.00

Appendix B – Mach3 Plugin Development Resources

Sysinternals DebugView is now owned by Microsoft. It is a free download and it may be obtained here:

http://www.microsoft.com/technet/sysinternals/Miscellaneous/DebugView.mspx

Winzip is available for download at

http://www.winzip.com

It is highly recommended and will be used to compress files for all of the plugin tutorials and commercial software deliveries. It currently costs $29.95.

Appendix C – Useful Websites

Here are some useful websites with tutorials and references for C++ and MFC (Microsoft Foundation Classes)..

Microsoft MSDN online, the first place to look for the real documentation for the Windows API and MFC.

http://msdn2.microsoft.com/en-us/default.aspx

The CodeProject, a really great resource for all kinds of code and tutorials.

http://www.codeproject.com/

Bear in mind that very little of what you see in my plugin tutorial is considered to be 'pure C++'. I am largely an embedded C programmer and I prefer that over strict object oriented C++. C will also run faster, and is what you get when you download the Mach3 SDK. No objects are used outside of the object model interfaces where it is really not optional, I have seen COM / IDispatch programming done in pure C but it is really only for masochists.

A good tutorial C++ site.

http://www.cplusplus.com/doc/tutorial/

Another good tutorial site with C language coverage and advanced topics.

http://www.cprogramming.com/tutorial.html

Very good tutorials on MFC in the context of Visual C++.

http://www.functionx.com/visualc/

Appendix D – Credits and Congratulations

The biggest credit goes to Art Fenerty, the tireless creator of Mach3. Without him an entire industry would not exist. Brian Barker is doing an increasing amount of work with Art. The creator of the first plugin abstraction and the basis for subsequent plugins is John A. Prentice who we all owe much to. You all have my thanks and my admiration.

Appendix E – Contact Information

You may contact me at jemmyell@yahoo.com to inquire about Mach3 plugin development.

I provide consulting and contract programming to the Mach3 community. I can provide everything from pilot plugin projects to complete deliverables including documentation and installers. Licensing modules that lock your plugin to an individual computer are also available. Please contact me for details and a quote to jump start YOUR plugin project.

In-house native Chinese Language (Mandarin) translation services are also available.