Building a DLL with Visual C++

Publish Date: Aug 03, 2013

Overview

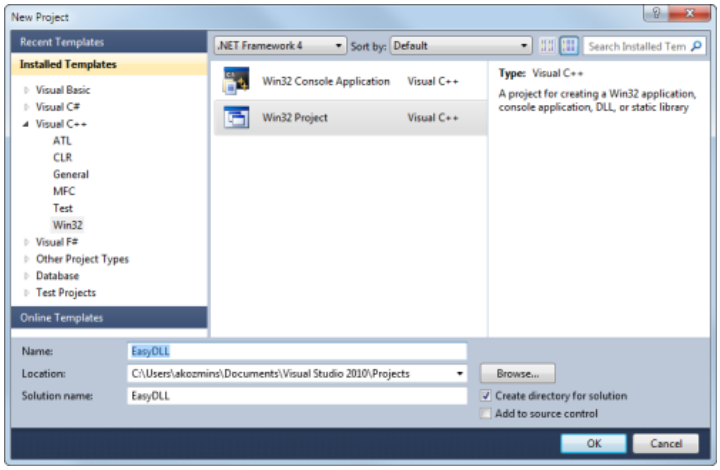
Microsoft's Visual C++ (MSVC) integrated development environment (IDE) can be overwhelming if the programmer has never used it. This document is designed to aid those wanting to compile a DLL for use with LabVIEW.

Note: This document applies to MSVC 2010.

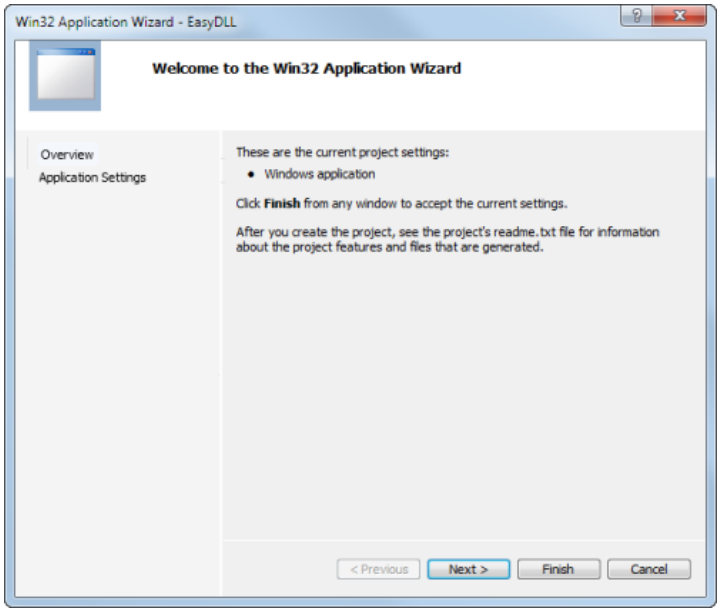
Table of Contents

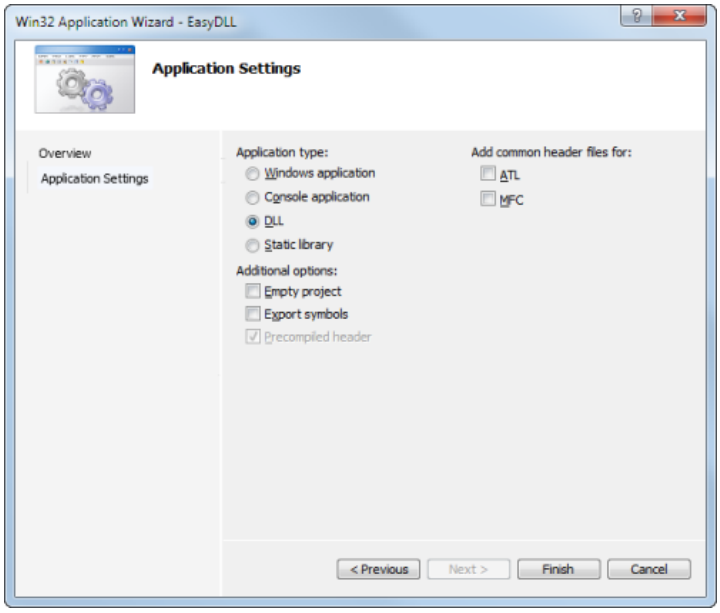
1. Step 1: Creating a DLL Project

Select **File»New Project** to open the New Project dialog box. From the **Visual C++ Templates** list, select **Win32 Project**, name your project and click **OK**.



In the next dialog box, you may see the current project settings to be Windows Application. Click **Next** to change the Application Type to **DLL**.





MSVC creates a DLL project with one source (`.cpp`) file, which has the same name as the project. It also generates a `stdafx.cpp` file. The `stdafx.cpp` file is necessary, but you do not generally need to edit it.

2. Step 2: Editing the Source File

Every DLL file must have a `DllMain` function, which is the entry point for the library. Unless you must do a specific initialization of the library, the default `DllMain` that MSVC created is sufficient. Notice that this function does nothing.

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
    return TRUE;
}
```

If a library initialization is required, you might need a more complete `DllMain`:

```
BOOL WINAPI DllMain(
         HINSTANCEhinstDLL,  // handle to DLL module
         DWORD fdwReason,     // reason for calling function
         LPVOID lpReserved )  // reserved
{
    // Perform actions based on the reason for calling.
    switch( fdwReason )
    {
   case DLL_PROCESS_ATTACH:
       // Initialize once for each new process.
       // Return FALSE to fail DLL load.
       break;

   case DLL_THREAD_ATTACH:
       // Do thread-specific initialization.
       break;

   case DLL_THREAD_DETACH:
       // Do thread-specific cleanup.
       break;

   case DLL_PROCESS_DETACH:
       // Perform any necessary cleanup.
       break;
    }
       return TRUE;
}
```

Once the `DllMain` function is complete, write the routines that you intend to access from the DLL.

```
//Function declarations
int GetSphereSAandVol(double radius, double* sa, double* vol);
double GetSA(double radius);
double GetVol(double radius);

...

int GetSphereSAandVol(double radius, double* sa, double* vol)
//Calculate the surface area and volume of a sphere with given radius
{
   if(radius < 0)
```

```
    return false; //return false (0) if radius is negative
        *sa = GetSA(radius);
        *vol = GetVol(radius);
        return true;
}

double GetSA(double radius)
{
    return 4 * M_PI * radius * radius;
}

double GetVol(double radius)
{
    return 4.0/3.0 * M_PI * pow(radius, 3.0);
}
```

For the DLL to compile correctly, you must declare the `pow` function (i.e. power, `pow(x,y)` is equivalent to x^y) and the constant M_PI (i.e. 3.14159).
Do this by inserting two lines of code below `#include "stdafx.h"` at the top of the `.cpp` file. The code should look as follows:

```
#include "stdafx.h"
#include "math.h"    //library that defines the pow function
#define M_PI 3.14159 //declare our M_PI constant
```

At this point, you can compile and link the DLL. However, if you do so, the DLL will not export any functions, and thus, will not really be useful.

3. Step 3: Exporting Symbols

To access the functions within the DLL, it is necessary to tell the compiler to export the desired symbols. However, you first must address the issue of C++ name decoration. MSVC compiles your source as C++ if it has a `.cpp` or `.cxx` extension. If the source file has a `.c` extension, then MSVC compiles it as C. If you compile your file as C++, then the function names are normally decorated in the output code. This might be problematic because the function name has extra characters added to it. To avoid this problem, declare the function as 'extern "C"' in the function declaration, as follows:

```
extern "C" int GetSphereSAandVol(double radius, double* sa, double* vol);
```

This prevents the compiler from decorating the name with C++ decorations.

Warning: Without C++ decoration, polymorphic functions are not possible.

When you finish with the C++ decorations, you can actually export the functions. There are two methods to inform the linker which functions to export. The first, and most simple, is to use the `__declspec(dllexport)` tag in the function prototype for any function you want to export. To do this, add the tag to the declaration and definition, as follows:

```
extern "C" __declspec(dllexport) int GetSphereSAandVol(double radius, double* sa, double* vol);
...
__declspec(dllexport) int GetSphereSAandVol(double radius, double* sa, double* vol)
{
     ...
}
```

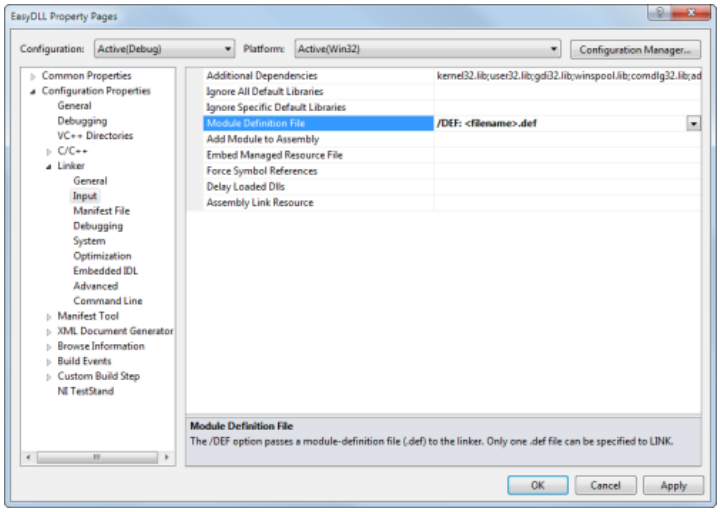
The second method is to use a `.def` file to explicitly declare which functions to export. The `.def` file is a text file that contains information the linker uses to decide what to export. It has the following format:

```
LIBRARY   <Name to use inside DLL>
DESCRIPTION "<Description>"
EXPORTS
    <First export>   @1
    <Second export>  @2
    <Third export>   @3
    ...
```

For the example DLL, the `.def` file will look like this:

```
LIBRARY   EasyDLL
DESCRIPTION "Does some sphere stuff."
EXPORTS
    GetSphereSAandVol   @1
```

If you have properly created your DLL project, then the linker automatically looks for a `.def` file of the same name as the project in the project directory. To change this option, select **Project»Properties**. In the **Linker** folder, click the **Input** property page and modify the **Module Definition File** property to **/DEF: <filename>.def** .



See Also:
Microsoft's .DEF file method documentation

4. Step 4: Specifying the Calling Convention

The last thing that you might need to do before compiling the DLL is to specify the calling convention for the functions that you want to export. Usually, there are two choices: C calling convention or standard calling conventions, also called Pascal and WINAPI. Most DLL functions use standard calling conventions, but LabVIEW can call either.

To specify C calling conventions, you do not need to do anything. This is the default unless you specify otherwise in **Project»Properties»C/C++»Advanced**. If you want to explicitly declare the function as a C call, use the `__cdecl` keyword in the function declaration and definition:

```
extern "C" __declspec(dllexport) int __cdecl GetSphereSAandVol(double radius, double* sa, double* vol);
...
__declspec(dllexport) int __cdecl GetSphereSAandVol(doublt radius, double* sa, double* vol)
{
     ...
}
```

To specify standard calling conventions, place the `__stdcall` keyword in the function declaration and definition:

```
extern "C" int __stdcall GetSphereSAandVol(double radius, double* sa, double* vol);
...
int __stdcall GetSphereSAandVol(doublt radius, double* sa, double* vol)
{
     ...
}
```

When using standard calling conventions, the function name is decorated in the DLL. You can avoid this by using the `.def` file method of exporting functions, rather than the `__declspec(dllexport)` method. Therefore, National Instruments recommends that you use the `.def` file method to export stdcall functions.

5. Step 5: Building the DLL

Once you write the code, declare what functions to export, and set the calling conventions, you are ready to build your DLL. Select **Build»Build <Your project>** to compile and link your DLL. You are now ready to use or debug your DLL from LabVIEW. The attached `EasyDLL.zip` file contains the Visual C++ workspace used to create this DLL and a LabVIEW VI that accesses the DLL.