Using Visual C++ DLLs in a C++Builder Project

It is likely that one day your boss will ask you if you can create a GUI with C++Builder that interfaces to an existing 32 bit DLL compiled with Microsoft Visual C++. Often times, the original DLL source code won't be available to you, either because the DLL comes from a third party vendor, or because the 22 year old intern just deleted the `\DLL\SRC` directory from the network. Given a DLL and a header file, this article shows you how to call the DLL from your C++Builder project.

Calling DLL functions from a C++Builder project

Calling a DLL that was created with Visual C++ presents C++Builder programmers with some unique challenges. Before we attempt to tackle DLLs generated by Visual C++, it may be beneficial to review how you call a DLL that was created with C++Builder. A DLL that was created with C++Builder presents fewer roadblocks than one that was made by Visual C++.

You need to gather three ingredients in order to call a DLL function from your C++Builder program: the DLL itself, a header file with function prototypes, and an import library (you could load the library at runtime instead of using an import library, but we will stick to the import library method for simplicity). To call a DLL function, add the import library to your C++Builder project by selecting the *Project | Add To Project* menu option in C++Builder. Next, insert a `#include` statement for the DLL header file in the C++ source file that needs to call one of the DLL functions. The last step is to add the code that calls the DLL function.

Listings A and B contain source code for a DLL that can serve as a test DLL. Notice that the test code implements two different calling conventions (`__stdcall` and `__cdecl`). This is for a very good reason. When you try to call a DLL that was compiled with Visual C++, most of your headaches will result from disagreements due to calling conventions. Also notice that one function does not explicitly list the calling convention that it uses. This unknown function will act as a measuring stick for DLL functions that don't list their calling convention.

```
//----------------------------------------
// Listing A: DLL.H

#ifdef __cplusplus
extern "C" {
#endif

#ifdef _BUILD_DLL_
```

```
#define FUNCTION __declspec(dllexport)
#else
#define FUNCTION __declspec(dllimport)
#endif

FUNCTION int __stdcall   StdCallFunction(int Value);
FUNCTION int __cdecl     CdeclFunction  (int Value);
FUNCTION int             UnknownFunction(int Value);

#ifdef __cplusplus
}
#endif


//---------------------------------------
//Listing B: DLL.C

#define _BUILD_DLL_
#include "dll.h"

FUNCTION int __stdcall StdCallFunction(int Value)
{
    return Value + 1;
}

FUNCTION int __cdecl   CdeclFunction(int Value)
{
    return Value + 2;
}

FUNCTION int UnknownFunction(int Value)
{
    return Value;
}
```

To create a test DLL from Listing A and Listing B, open up C++Builder and bring up the Object Repository by selecting the *File* | *New* menu option. Select the DLL icon and click the OK button. C++Builder responds by creating a new project with a single source file. That file will contain a DLL entry point function and some include statements. Now select *File* | *New Unit*. Save the new unit as `DLL.CPP`. Cut and paste the text from Listing A and insert it into the header file `DLL.H`. Then copy the code from Listing B and insert it into `DLL.CPP`. Make sure that the `#define` for `_BUILD_DLL_` is placed above the include statement for `DLL.H`.

Save the project as `BCBDLL.BPR`. Next, compile the project and take a look at the files produced. C++Builder generates both a DLL and an import library with a `.LIB` extension.

At this point, you have the three ingredients needed to call a DLL from a C++Builder project: the DLL itself, a header file for function prototypes, and an import library to link with. Now we need a C++Builder project that will try to call the DLL functions. Create a new project in C++Builder and save it to your hard drive. Copy the DLL, the import library, and the `DLL.H` header file from DLL project to this new project. Select the *Project* | *Add To Project* menu option and add the LIB file to the project. Next, add a `#include` statement in the main unit that includes `DLL.H`. Finally, add code that calls the DLL functions. Listing C shows code that calls each the DLL functions from Listing A and B.

```
//---------------------------------------
// Listing C: MAINFORM.CPP - DLLTest program
#include <vcl\vcl.h>
#pragma hdrstop
```

```
#include "MAINFORM.h"
#include "dll.h"
//---------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int Value = StrToInt(Edit1->Text);
    int Result= StdCallFunction(Value);
    ResultLabel->Caption = IntToStr(Result);
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    int Value = StrToInt(Edit1->Text);
    int Result= CdeclFunction(Value);
    ResultLabel->Caption = IntToStr(Result);
}
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    int Value = StrToInt(Edit1->Text);
    int Result= UnknownFunction(Value);
    ResultLabel->Caption = IntToStr(Result);
}
```

The problem with Visual C++ DLLs

In an ideal world, calling a DLL created with Visual C++ would be no more difficult than calling a DLL built with C++Builder. Unfortunately, Borland and Microsoft disagree on several points. For starters, Borland and Microsoft disagree on file formats for OBJs and import library files (Visual C++ uses the COFF library format while Borland uses OMF). This means that you can't add a Microsoft generated import library to a C++Builder project. Thanks to the Borland `IMPLIB` utility, the file format differences are surmountable.

The two products also disagree on linker naming conventions. This turns out to be the primary hurdle when trying to call a Visual C++ DLL from C++Builder. Every function in a DLL or OBJ has a linker name. The linker uses the linker name to resolve functions that were protoyped during compile time. The linker will generate an unresolved external error if it can't find a function with a linker name that it thinks is needed by the program.

With regard to linker function names, Borland and Microsoft disagree on these points:

- 1- Visual C++ sometimes decorates exported `__stdcall` functions.
- 2- Borland C++Builder expects imported `__cdecl` functions to be decorated.

So why is this such a big deal? Take disagreement #1 regarding the `__stdcall` calling convention. If you create a DLL with Visual C++ that contains a `__stdcall` function called `MyFunction()`, Visual C++ will give the function a linker name that looks like `_MyFunction@4`. When the Borland linker tries to resolve calls made to this function, it expects to find a function with the name `MyFunction`. Since the import library for the Visual C++ DLL doesn't contain a function called

MyFunction, the Borland linker reports an unresolved external, which means it couldn't find the function.

Your attack strategy for overcoming these three problems will depend on how the Visual C++ DLL was compiled. I have broken the process into four steps.

Step 1: Identify calling conventions used by the Visual C++ DLL

In order to combat the naming convention entanglements, you must first determine what calling conventions are used by functions in the DLL. You can determine this by investigating the header file for the DLL. The function prototypes in the DLL header should look something like this:

```
__declspec(dllimport) void CALLING_CONVENTION MyFunction(int nArg);
```

CALLING_CONVENTION should be __stdcall or __cdecl (see Listing A for concrete examples). In many cases, the calling convention won't be specified, in which case it defaults to __cdecl.

Step 2: Examine the linker names in the DLL

If step 1 reveals that the DLL utilizes the __stdcall calling convention, you will need to examine the DLL to determine the naming convention that Visual C++ followed when it created the DLL. Visual C++ decorates __stdcall functions by default, but the DLL programmer can prohibit name decorations if they add a DEF file to their project. Your work will be slightly more tedious if the DLL supplier did not use a DEF file.

The command line TDUMP utility allows you to examine the linker names of functions exported by the DLL. The following command invokes TDUMP on a DLL.

```
TDUMP -ee -m MYDLL.DLL > MYDLL.LST
```

TDUMP can report a ton of information about the DLL. We're only interested in functions exported by the DLL. The -ee command option instructs TDUMP to list only export information. The -m switch tells TDUMP to show the DLL functions in their raw format. Without the -m switch, TDUMP would attempt to de-mangle decorated functions into a human readable format. If the DLL is large, you may want to redirect the output of TDUMP to a file (via the > MYDLL.LST appendage).

The TDUMP output for the test DLL in Listing A and B looks like this:

```
Turbo Dump  Version 5.0.16.4 Copyright (c) 1988, 1998 Borland International
                      Display of File DLL.DLL

EXPORT ord:0000='CdeclFunction'
EXPORT ord:0002='UnknownFunction'
EXPORT ord:0001='_StdCallFunction@4'
```

Notice the leading underscore and the trailing @4 on the __stdcall function. The __cdecl and the unknown function don't contain any decorations. If the Visual C++ DLL had been compiled with a DEF file, the decorations on the __stdcall function would not be present.

Step 3: Generate an import library for the Visual C++ DLL

Here comes the hard part. Due to the library file format differences between C++Builder and Visual C++, you cannot add an import library created with Visual C++ to your C++Builder project. You must create an OMF import library using the command line tools that come with C++Builder. Depending out what you found in the first two steps, this step will either go smoothly, or it could take some time.

As stated earlier, C++Builder and Visual C++ don't agree on how functions should be named in a DLL. Due to naming convention differences, you will need to create an aliased import library if the DLL implements calling conventions where C++Builder and Visual C++ disagree. Table A lists the areas of disagreement.

Table A: Visual C++ and C++Builder naming conventions

Calling convention	VC++ name	VC++ (DEF used)	C++Builder Name
__stdcall	_MyFunction@4	MyFunction	MyFunction
__cdecl	MyFunction	MyFunction	_MyFunction

The C++Builder column lists function names that the Borland linker expects to see. The first Visual C++ column lists the linker names that Visual C++ generates when the Visual C++ project does not utilize a DEF file. The second Visual C++ column contains linker names that Visual C++ creates when a DEF file is used. For things to go smoothly, the C++Builder name should agree with the Visual C++ name. Notice that the two products agree in only one place: `__stdcall` functions where the Visual C++ project contained a DEF file. For the remaining scenarios, you will need to create an import library that aliases the Visual C++ name to a C++Builder compatible name.

Table A shows that there are several combinations that you may need to deal with when creating the import library. I have separated the combinations into two cases.

Case 1: The DLL contains only `__stdcall` functions and the DLL vendor utilized a DEF file.

Table A reveals that VC++ and C++Builder agree only when the DLL uses `__stdcall` functions. Furthermore, the DLL must be compiled with a DEF file to prevent VC++ from decorating the linker names. The header file will tell you if the `__stdcall` calling convention was used (Step 1), and `TDUMP` will reveal whether or not the functions are decorated (Step 2). If the DLL contains `__stdcall` functions that are not decorated, then Visual C++ and C++Builder agree on how the functions should be named. You can create an import library by running `IMPLIB` on the DLL. No aliases are needed.

`IMPLIB` works like this:

```
IMPLIB (destination lib name) (source dll)
```

For example,

```
IMPLIB mydll.lib mydll.dll
```

Create the import library and move on to step 4.

Case 2: The DLL contains `__cdecl` functions or decorated `__stdcall` functions.

If your DLL vendor is adamant about creating DLLs that are compiler independent, then you have a good chance of falling into the Case 1 category. Unfortunately, odds are you won't fall into the Case 1 group for several reasons. For one, the calling convention defaults to `__cdecl` if the DLL

vendor omits a calling convention when prototyping the functions, and `__cdecl` forces you into Case 2. Secondly, even if your vendor has utilized the `__stdcall` calling convention, they probably neglected to utilize a DEF file to strip the Visual C++ decorartions.

However you got here, Good Day, and welcome to Case 2. You're stuck with a DLL whose function names don't agree with C++Builder. Your only way out of this mess is to create an import library that aliases the Visual C++ function names into a format compatible with C++Builder. Fortunately, the C++Builder command line tools allow you to create an aliased import library.

The first step is to create a DEF file from the Visual C++ DLL by using the `IMPDEF` program that comes with C++Builder. `IMPDEF` creates a DEF file that lists all of the functions exported by the DLL. You invoke `IMPDEF` like this:

```
IMPDEF (Destination DEF file) (source DLL file).
```

For example

```
IMPDEF mydll.def mydll.dll
```

After running `IMPDEF`, open the resulting DEF file using the editor of your choice. When the DLL source in Listing A and B is compiled with Visual C++, the DEF file created by `IMPDEF` looks like this:

```
LIBRARY     DLL.DLL

EXPORTS
    CdeclFunction        @1
    UnknownFunction      @3
    _StdCallFunction@4   =_StdCallFunction      @2
```

The next step is to alter the DEF file so it aliases the DLL functions into names that C++Builder will like. You can alias a function by listing a C++Builder compatible name followed by the original Visual C++ linker name. For the test DLL in Listing A and B, the aliased DEF looks like this:

```
EXPORTS
  ; use this type of aliasing
  ; (Borland name)   = (Name exported by Visual C++)
    _CdeclFunction   = CdeclFunction
    _UnknownFunction = UnknownFunction
    StdCallFunction  = _StdCallFunction@4
```

Notice that the function names on the left match the Borland compatible names from Table A. The function names on the right are the actual linker names of the functions in the Visual C++ DLL.

The final step is to create an aliased import library from the aliased DEF file. Once again, you rely on the `IMPLIB` utility, except that this time, you pass `IMPLIB` the aliased DEF file as its source file instead of the original DLL. The format is

```
IMPLIB (dest lib file) (source def file)
```

For example

```
IMPLIB mydll.lib mydll.def
```

Create the import library and move on to step 4. You may want to examine the import library first

to ensure that each DLL function appears in a naming format that C++Builder agrees with. You can use the `TLIB` utility to inspect the import library.

```
TLIB mydll.lib, mydll.lst
```

The list file for the test DLL looks like this:

```
Publics by module

StdCallFunction size = 0
        StdCallFunction

_CdeclFunction  size = 0
        _CdeclFunction

_UnknownFunction size = 0
        _UnknownFunction
```

Step 4: Add the import library to your project

Once you create an import library for the Visual C++ DLL, you can add the import library to your C++Builder project using the *Project | Add to Project* menu option. You use the import library without regard to whether the import library contains aliases or not. After adding the import library to your project, build your project and see if you can successfully link.

Conclusion:

This article demonstrated how you can call functions in a Visual C++ DLL from a C++Builder project. The techniques work with C++Builder 1 and C++Builder 3, and DLLs built with Visual C++ 4.X or Visual C++ 5 (I haven't tested Visual C++ 6 yet).

You may have noticed that this article only discusses how to call C style functions in a DLL. No attempt is made to call methods of an object where the code for the class resides in a Visual C++ DLL. C++ DLLs present an even greater array of problems because linker names for member functions are mangled. The compiler employs a name mangling scheme in order to support function overloading. Unfortunately, the C++ standard does not specify how a compiler should mangle class methods. Without a strict standard in place, Borland and Microsoft have each developed their own techniques for name mangling, and the two conventions are not compatible. In theory, you could use the same aliasing technique to call member functions of a class that resides in a DLL. However, you may want to consider creating a COM object instead. COM introduces many of its own problems, but it does enforce a standard way of calling methods of an object. A COM object created by Visual C++ can be called from any development environment, including both Delphi and C++Builder.

C++Builder 3.0 introduced a new command line utility called `COFFtoOMF.EXE`. This utility can convert a Visual C++ import library to a C++Builder import library. Furthermore, the program will automatically alias `__cdecl` functions from the Visual C++ format to the C++Builder format. The automatic aliasing can simplify Step 3 if the DLL exclusively uses the `__cdecl` calling convention.
