A Practical Guide to Writing Dynamically Loadable Modules for IDL

By: Richie Zeng

Rev 1.0

For use by the Center for X-Ray Optics at Lawrence Berkeley National Lab

About

This guide is intended for users who are relatively familiar with C/C++ and know at least a little about IDL. The goal of this guide is to get you up and running with your own DLM very quickly and simply, and as a result I skipped a lot of details when writing this.

You will see me refer to Ronn Kling's "Calling C and C++ from IDL" (from here on referred to as CCCPPFIDL) book or the IDL External Development Guide (EDG) if you're looking for additional details.

Introduction

There are many different ways to interface with IDL from other languages. The two main ones are with CALL_EXTERNAL and with DLMs (Dynamically Loadable Modules).

CALL_EXTERNAL essentially allows you to build an IDL library in C, while having no knowledge of IDL internals. The functions that you write in C can be called from another C program without having to make any changes. The main disadvantage of this is that you have to load the library every time you want to call a method from the library. The code ends up looking less elegant as a result.

Dynamically Loadable Modules are more difficult to write than CALL_EXTERNAL libraries because they involve extensive knowledge of IDL internals and the memory management behind them. They are, however, much faster because the libraries are loaded only once (when a function/procedure is called) and don't have to be loaded again until after IDL is closed. The code is also much more elegant, as it behaves exactly like a built in IDL method. In fact, many of IDL's methods are written in DLMs already. See for yourself!
Go to: {IDL Root Folder}/bin/bin.x86

We decided to go with writing DLMs because we felt that speed was the highest priority, and the ease of coding on the IDL side makes up for the extra trouble on the C side. Plus, the C side isn't so bad once you learn the basic rules.

Dynamically Loadable Modules

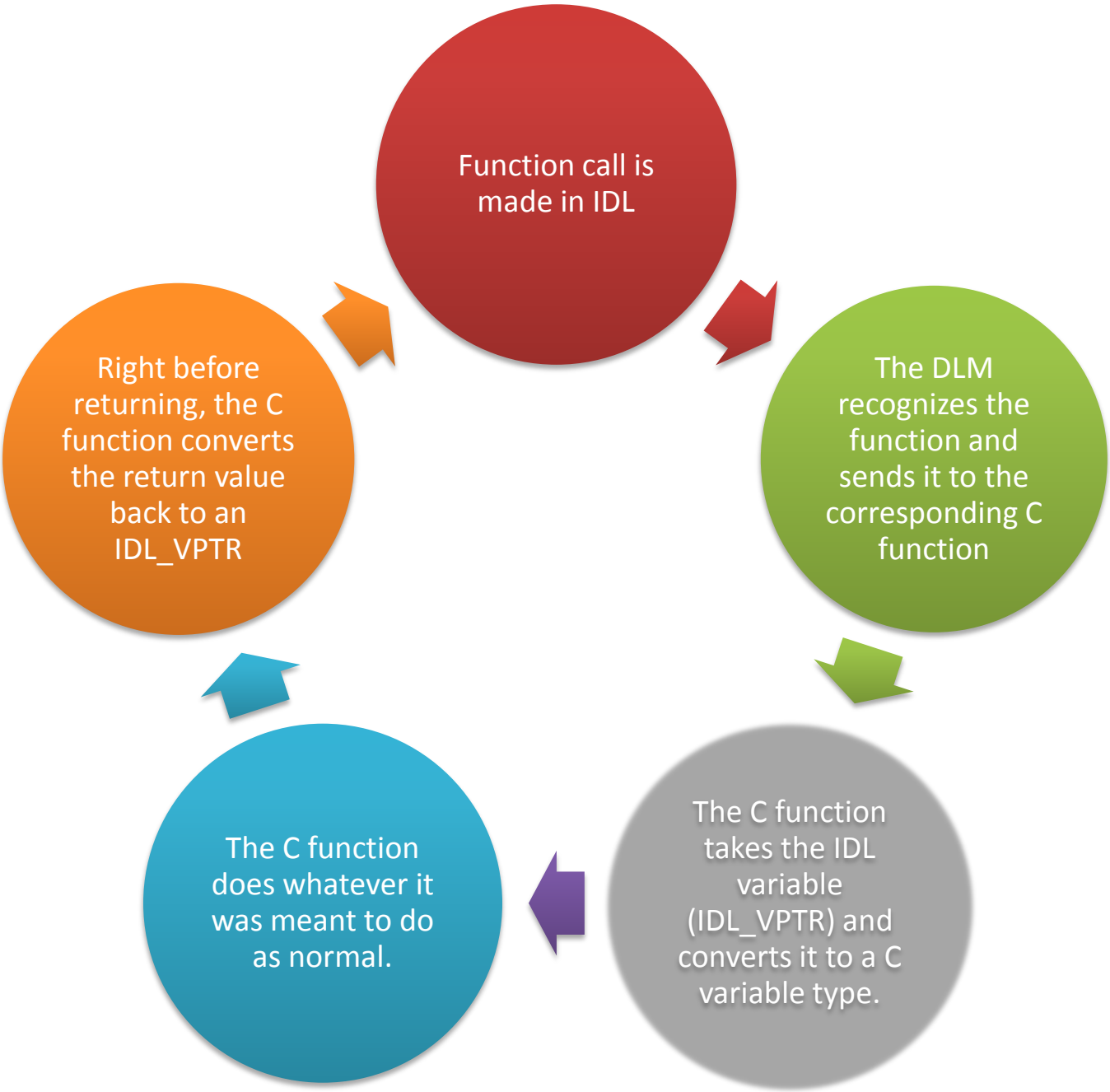
DLMs essentially have two components. One is the .dll file, which is the library that contains all the methods. You should be decently familiar with writing these before writing a DLM. The second component is a .dlm file, which is essentially a list of functions and procedures that IDL can read.

When IDL starts up, it looks at all the DLMs in its DLM_PATH (more on this later), and keeps a record of all the functions and methods. Its important to note that these libraries are NOT yet loaded, which is why this step barely affects runtime at all.

IDL only loads a DLM when a function/method that it recognizes is called. Once a DLM is loaded, it never needs to be loaded again until you restart IDL.

A .dlm file is just a text file! The real meat of writing a DLM is in how you construct your libraries, which is what the main part of this guide is about.

Architecture



Components

DLM File

IDL_Load function

Export Definition File

Corresponding C functions

IDL Wrapper Code

The DLM File

```
MODULE EXAMPLEDLM
DESCRIPTION Several examples of simple DLM code
VERSION 1.0
SOURCE RICHIE ZENG


FUNCTION EX_HELLO_FUNC 0 1
PROCEDURE EX_HELLO_PROC 0 1
FUNCTION EX_SIMPLE_INT 1 1
FUNCTION EX_SIMPLE_UINT 1 1

PROCEDURE EX_INPLACEARR 1 1
```

The DLM is just a text file formatted in a specific way. You need to put this in the same folder as the .dll file that you created.

Only two parts of this example are critical to have, the module name and the list of functions and procedures. See pg. 6 of CCCPPFIDL for more details.

MODULE: The name of the module.

FUNCTIONS/PROCEDURES: The list of methods that IDL will track. When you call one of these from the IDL side it will load the module.

Format: FUNCTION Name [MinArgs] [MaxArgs] [Options]
PROCEDURE Name [MinArgs] [MaxArgs] [Options]

The name of the function/procedure must be in all caps (IDL is case insensitive). The only option that you might use is KEYWORDS for when you want a function to have keywords.

IDL_Load

```
int IDL_Load(void)
{
  /* Call the startup function to add the routines to IDL. */

  /* Functions names in the strings are recognized by IDL and must be in a
  /* NOTE: Make sure to rename the placeholder functions so they don't pol
  static IDL_SYSFUN_DEF2 ex_functions[] = {
    {(IDL_FUN_RET)hello_func, "EX_HELLO_FUNC", 0, 1, 0, 0},
    {(IDL_FUN_RET)simple_int, "EX_SIMPLE_INT", 1, 1, 0, 0},
    {(IDL_FUN_RET)simple_uint, "EX_SIMPLE_UINT", 1, 1, 0, 0},
  };

  static IDL_SYSFUN_DEF2 ex_procedures[] = {
    {(IDL_SYSRTN_GENERIC)hello_proc, "EX_HELLO_PROC", 0, 1, 0, 0},
    {(IDL_SYSRTN_GENERIC)in_place_array, "INPLACEARR", 1, 1, 0, 0},
    };

    return IDL_SysRtnAdd(ex_functions, IDL_TRUE, ARRLEN(ex_functions)) &&
        IDL_SysRtnAdd(ex_procedures, IDL_FALSE, ARRLEN(ex_procedures));
}
```

The IDL_Load function essentially maps the IDL functions to their corresponding C functions. (CCCPPFIDL pg. 8)

Note that there are two arrays, one for procedures and one for functions, this does not have to be the case for your functions. This is important because in the IDL_SysRtnAdd() one of the parameters is IDL_TRUE or IDL_FALSE depending on whether it is a procedure or a function. Beyond this, you can have as many different arrays as you want.

Let’s break down the formatting behind each entry in the array.

Export Definition File

```
LIBRARY                 EXAMPLEDLM
DESCRIPTION             'Simple sample functions for writing DLMs'

EXPORTS                 IDL_Load        @1
```

For development on windows, an export definition file is required if you don't use the __declspec(dllexport) keyword to export the DLL functions.

The only function you need to put in the DLL for exporting is the IDL_Load function, because that is the only function directly accessed by IDL. IDL accesses the functions that you write on the C side by going through the IDL_Load.

This is a very simple file to have, the only thing you might have different from the above example is the name of the LIBRARY and the (optional) DESCRIPTION. Don't forget the "@1" after the IDL_Load.

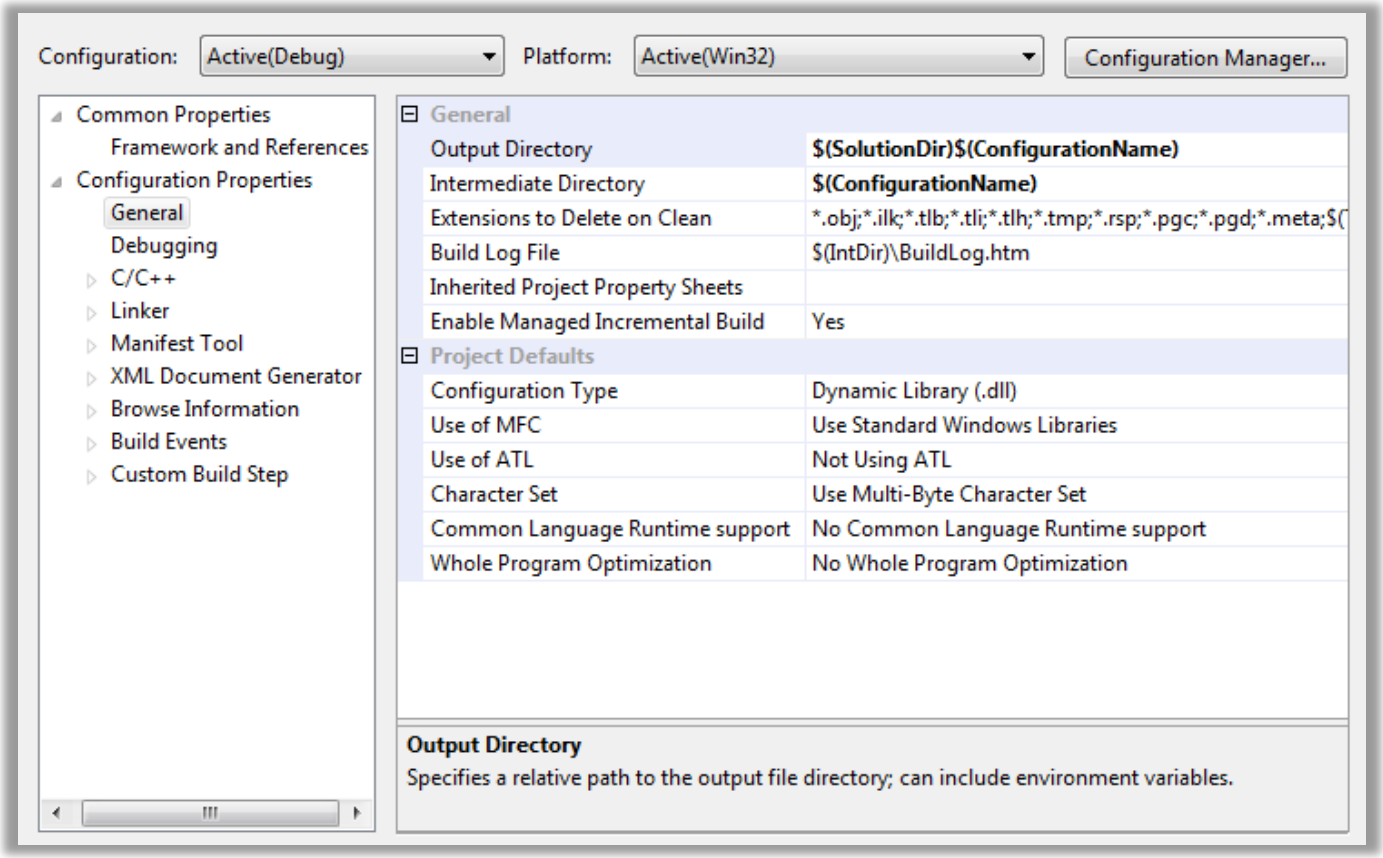
NOTE: While this file is very simple to write, it is very important to have. IDL will look for the IDL_Load function as soon as you make a call to a DLM function, but if you don't export the IDL_Load function IDL won't be able to find it!

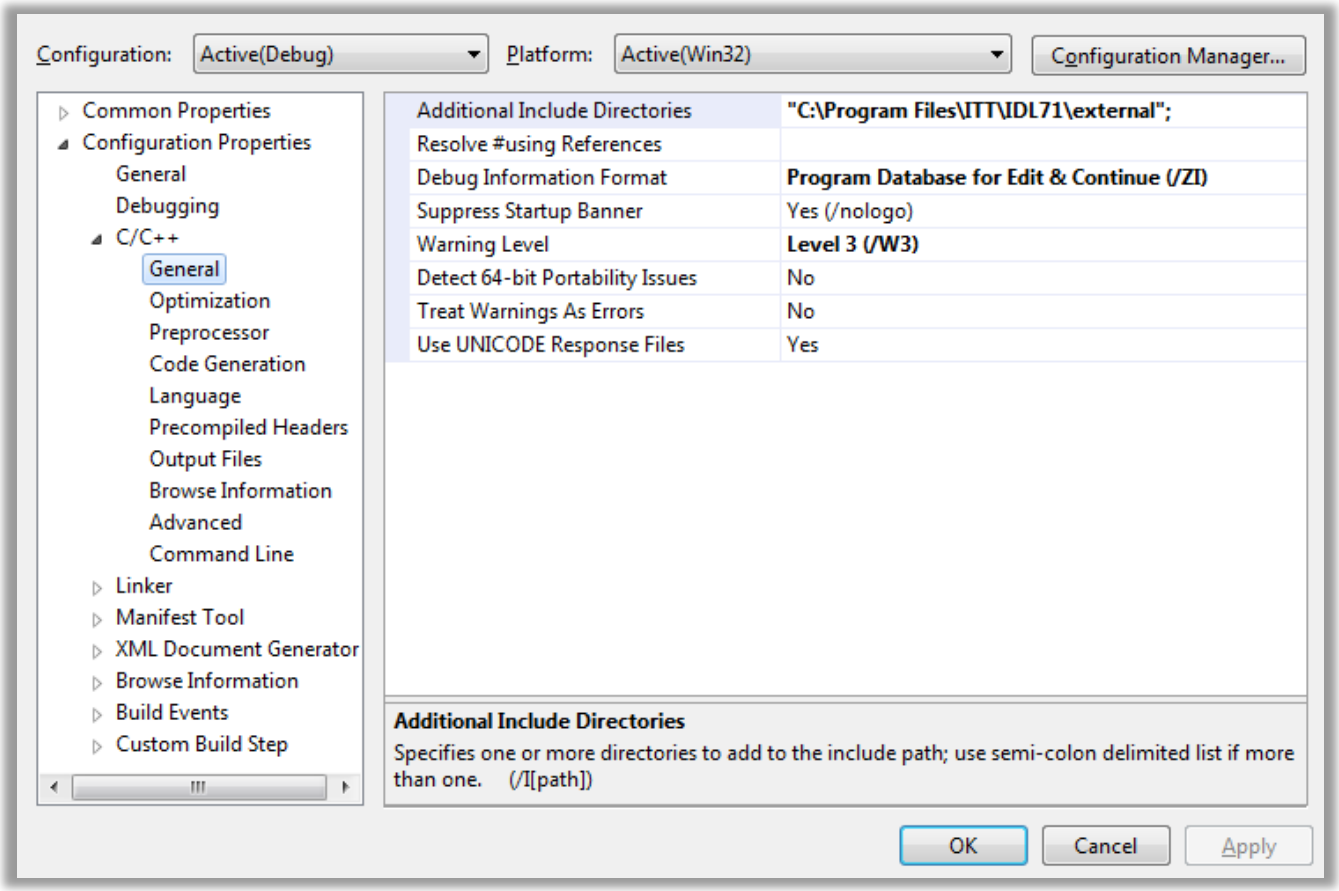
Configuring Your Development Environment

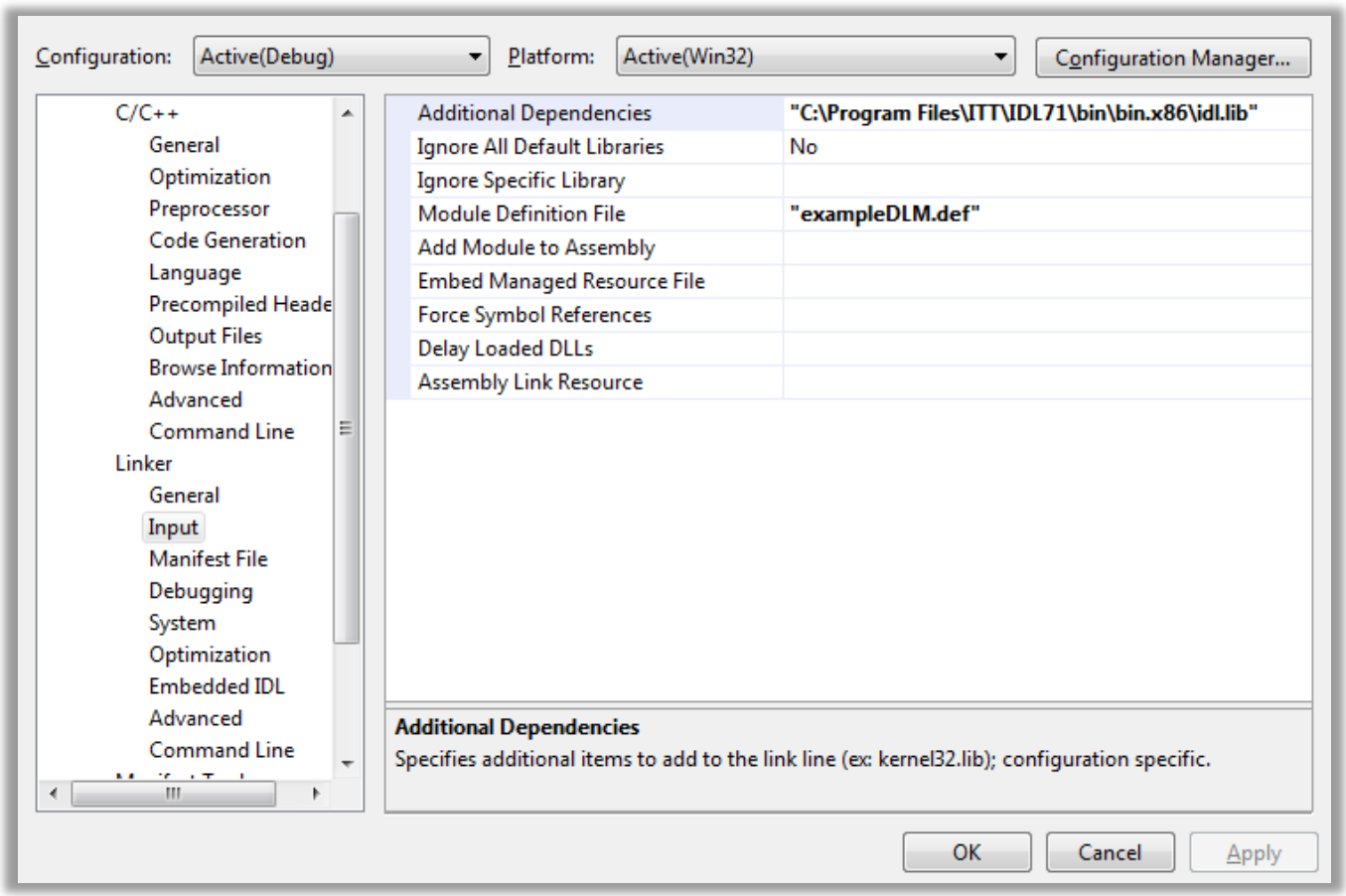
There are several settings you must have set properly so your library compiles properly. The following pages show you important settings to have for your project. These example images are from Visual Studio 2008.

Any include directories I have are the default installation location for most things. If you installed your software/libraries somewhere else, adjust your import paths as needed.

Go to Project -> {Project name} Properties (Hotkey: Alt+F7)




Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Additional Include Directories "C:\Program Files\ITT\IDL71\external";
Resolve #using References
Debug Information Format Program Database for Edit & Continue (/ZI)
Suppress Startup Banner Yes (/nologo)
Warning Level Level 3 (/W3)
Detect 64-bit Portability Issues No
Treat Warnings As Errors No
Use UNICODE Response Files Yes
Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])
OK
Cancel
Apply


Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Heade
Output Files
Browse Information
Advanced
Command Line
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Additional Dependencies "C:\Program Files\ITT\IDL71\bin\bin.x86\idl.lib"
Ignore All Default Libraries No
Ignore Specific Library
Module Definition File "exampleDLM.def"
Add Module to Assembly
Embed Managed Resource File
Force Symbol References
Delay Loaded DLLs
Assembly Link Resource
Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.
OK
Cancel
Apply

Code Design Process

Now you're ready to write some code! Here is a suggested design process for when you're writing your code to make sure you have all the right pieces.

What does your IDL program need?

- Think about what kinds of function/procedure calls you want in your IDL program.
- Write a DLM file so you have a good plan of what you want to do.
- Map the DLM function/procedure names to C/C++ function names in your IDL_Load function.

What can your C library do?

- Begin by writing your C library completely ignoring the IDL side.
- These library methods should be able to be called by any other C program and have it work.

Put the two components together!

- After your C library is perfectly functional, you can wrap it in your IDL code.
- THESE functions are the ones that IDL_Load maps the DLM functions to.
- You should make the necessary type conversions and call the C library.

Some Things to Consider Before Writing Code

- IDL DLM support code can get very long, especially if you use keywords. Consider splitting your code into multiple files, at the very least put your IDL_Load in a separate file so you always know where it is.

- The more you write on the C side the faster your code will be. Remember, IDL is an interpreted language, meaning it is SLOW compared to C, but much easier to write. Weigh the pros and cons of easy coding with runtime speed.