

Calling dlls From An IVI-COM Driver

REVISION HISTORY

Date	Description	Author	Revision Letter
11/26/02	Original Draft	John Rudderham	A

Abstract

This application note describes the procedure for calling functions contained in another dll from an IVI-COM driver. The case where the other dll is a VXI plug and play driver is discussed.

What you will need

Three things:

- ❑ A header (.h) file. This is included in your project so that the compiler has information about the functions in the dll.
- ❑ A library (.lib) file. This file gives the linker information about the dll.
- ❑ The dll itself. Used at runtime to execute the functions that it contains.

Modifications to the IVI-COM driver project

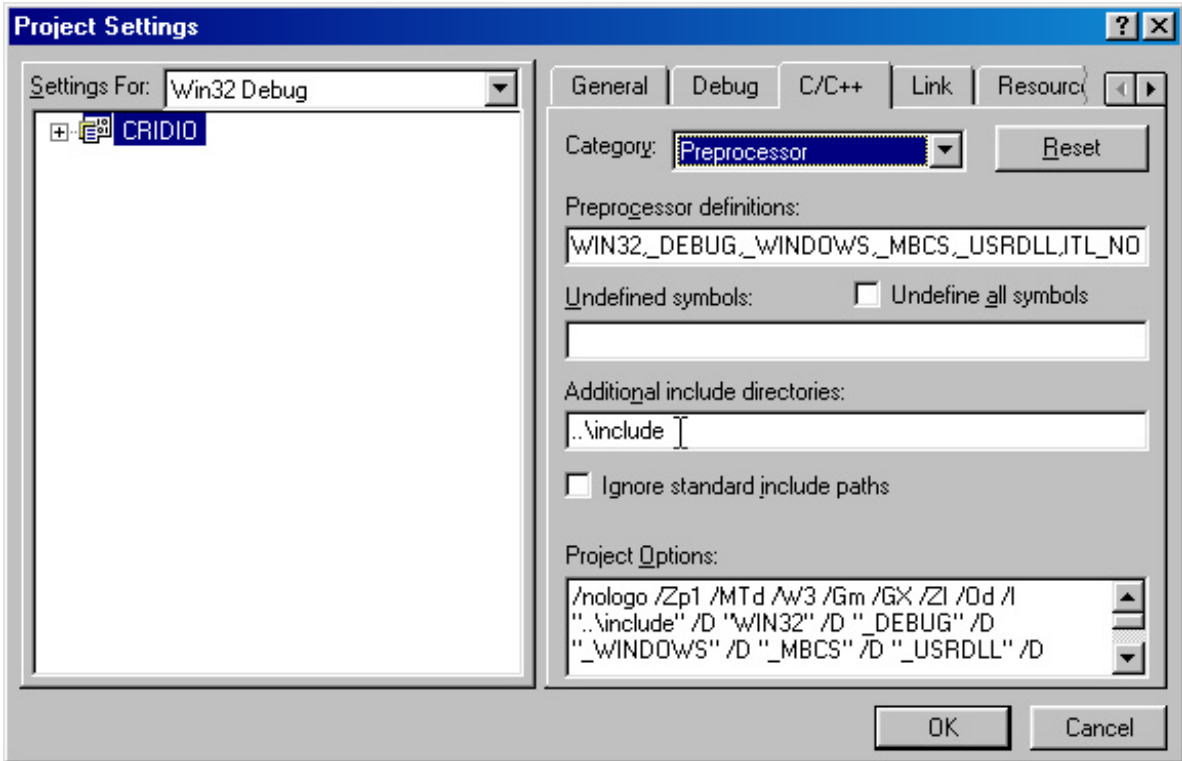
Once the driver project has been created (using VIVID Center), here are the steps to access the dll:

- ❑ Include the header file. The best place to do this is the StdAfx.h file, as this is in turn included in the other source code files. A typical StdAfx.h is shown below. The new line is highlighted.

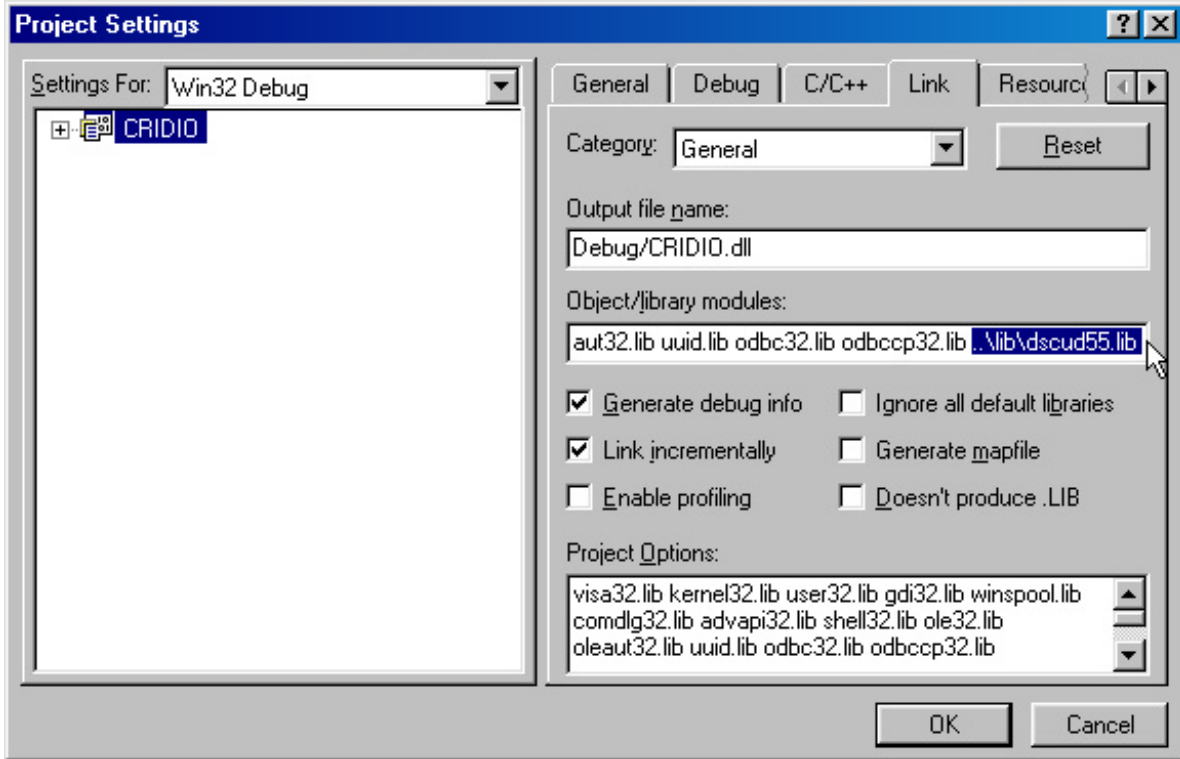
```
// stdafx.h : include file for standard system include files,  
// or project specific include files that are used frequently,  
// but are changed infrequently  
  
#if !defined(AFX_STDAFX_H__362EB0C4_0A09_11D4_AB5B_0050DACCF940__INCLUDED_)  
#define AFX_STDAFX_H__362EB0C4_0A09_11D4_AB5B_0050DACCF940__INCLUDED_  
  
#if _MSC_VER > 1000  
#pragma once  
#endif // _MSC_VER > 1000  
  
#define STRICT  
#ifndef _WIN32_WINNT  
#define _WIN32_WINNT 0x0400  
#endif  
#define _ATL_APARTMENT_THREADED  
  
#define _ATL_DEBUG_INTERFACES  
#define _ATL_DEBUG_QI  
  
#include <atlbase.h>  
//You may derive a class from CComModule and use it if you want to override  
//something, but do not change the name of _Module  
extern CComModule _Module;  
#include <atlcom.h>  
  
#pragma warning(disable : 4503) // standard disable for long STL names
```

```
#include "Itl.h"  
#include "dscud.h" // Header file for dll which will be accessed from the driver  
#import "IviDriverTypeLib.dll" raw_interfaces_only, no_namespace, named_guids  
  
//{{AFX_INSERT_LOCATION}}  
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.  
  
#endif // !defined(AFX_STDAFX_H_362EB0C4_0A09_11D4_AB5B_0050DACC940__INCLUDED)
```

This compiler must be able to find this include file. Add its directory to the *Additional Include Directories* in the project settings as shown below if necessary.



- Add the library file to the list of *Object/library modules* in the project settings.



- ❑ Call the functions from the dll in your code. After completing the steps above, the dll functions can be used as if they were “local”. If any global variables need to be declared, they can be added as member variables to the driver class in the main driver header file.
- ❑ Once the IVI-COM driver dll has been compiled, the original driver dll must be placed in the same directory as the IVI_COM driver dll before an instance of the IVI-COM driver can be created.

Considerations for Wrapping a VXI Plug and Play Driver

A common reason to want to call a dll from an IVI-COM driver is to make use of an existing VXI plug and play driver that has been written for the instrument. In this case, VIVID Center can be used to design an IVI-COM driver, and calls into the plug and play driver can be made from the IVI-COM driver’s methods and properties.

Initialization

Since the Initialize method needs to perform the same tasks in all normal IVI-COM drivers, the method can make use of a standard implementation in the IVI Template Library (ITL). The implementation of the driver’s Initialize method is shown below. This implementation is stored in the *liviDriver.cpp* file in the driver project.

```

HRESULT CCRIScope::Initialize(BSTR ResourceName, VARIANT_BOOL IdQuery, VARIANT_BOOL
    Reset, BSTR OptionString)
{
    ITL_METHOD_START_INIT_NOT_REQUIRED(IlviDriver, Initialize);
    return IlviDriverImpl<CVektrexScope>::Initialize(ResourceName, IdQuery, Reset, OptionString);
}
    
```

This is an ideal place to call the Initialize function of the plug and play driver. However, the implementation of Initialize in ITL performs several tasks that still need to be done. These include setting a flag to say the driver has been initialized (other methods and properties will not run if this flag has not been set) and setting up the Configuration Server to provide logical name/ resource descriptor and

repeated capability mapping. Calling ITL's Initialize with the "simulate=true" option allows these features to be set up.

Calling ITL's initialize with simulate=true causes a flag to be set that means that the simulation code that is generated into each of the IVI-COM driver's methods and properties will be run. Although the driver has get and put simulation properties, IVI rules do not allow a driver that has been initialized with simulation = true to use the put_simulation property to turn simulation off. Since the *m_bSimulate* variable that stores this flag is declared as private in the ITL, it cannot be accessed directly. In VIVID version 2.0 it is necessary to make a change to the ITL to allow simulation to be turned off. The file that needs to be changed is *c:\program files\ivi\include\itlbase.h* and the change is to make *m_bSimulate* a protected variable. Once this change is made, *m_bSimulate* can be set to false after ITL's initialize has been called. In future releases of VIVID *m_bSimulate* will be made protected.

The code below shows an Initialize method that follows these principles and calls into a VXI plug and play driver. The passed in option string is parsed and the simulation option is set to true. The ITL version of initialize is then called using the modified option string. Once this function returns the simulation flag is restored to the value that the client passed in. The initialize function in the VXI plug and play driver is then called, as long as simulation wasn't originally selected.

```

HRESULT CCRIDIO::Initialize(BSTR ResourceName, VARIANT_BOOL IdQuery, VARIANT_BOOL
Reset, BSTR OptionString)
{
    ITL_METHOD_START_INIT_NOT_REQUIRED(IlviDriver, Initialize);
    USES_CONVERSION;

    HRESULT hr = S_OK;

    string strOpts = OLE2A(OptionString);
    string strValue, strBefore, strAfter;
    int nPos = 0;
    int comma = 0;
    bool simulate;
    int i;
    int nTrue = 0;

    nPos = strOpts.find("=");

    if (nPos == strOpts.npos)
    {
        //Option string is blank if '=' is not found
        strOpts = "simulate=true";
        simulate = false;
    }
    else //at least one option is present - do we have simulate ?
    {
        // First make the string all lower case
        for (i = 0; i < strOpts.length(); i++)
        {
            strOpts[i] = tolower(strOpts[i]);
        }

        nPos = strOpts.find("simulate");

        if (nPos == strOpts.npos)
        {
            //option string does not contain simulate

```

```

        strOpts += ",simulate=true";
        simulate = false;
    }
    else //simulate found!
    {
        nPos = strOpts.find ("=", nPos); //find location of '=' after simulate
        comma = strOpts.find ("", nPos); //find location of ',' after '='

        if (comma == strOpts.npos) //no comma found - simulate must be last option
            //in string
        {
            strValue = strOpts.substr(nPos + 1);
        }
        else
        {
            strValue = strOpts.substr(nPos + 1, comma - nPos - 1);
        }

        nTrue = strValue.find ("true");

        if (nTrue != strValue.npos) //option string already has simulate=true
        {
            simulate = true;
        }
        else //simulate = false
        {
            simulate = false;
            if (comma == strOpts.npos) //need to replace false at the end of the
                //string with true
            {
                strOpts = strOpts.substr (0, nPos + 1);
                strOpts += "true";
            }
            else //need to replace false in the middle of the string with true
            {
                strBefore = strOpts.substr (0,nPos + 1);
                strAfter = strOpts.substr (comma);
                strOpts = strBefore;
                strOpts += "true";
                strOpts += strAfter;
            }
        }
    }
}

BSTR newOptions = A2OLE(strOpts.c_str());

//now run the initialize command in ITL with simulate on - this sets up the driver's initialized flag
//and will use the IVI ConfigServer to look up the resource descriptor if a logical name has been
//used.
hr = IlviDriverImpl<CCRIDIO>::Initialize(ResourceName, IdQuery, Reset, newOptions);

//reset the simulate flag to the value that was passed in.
m_bSimulate = simulate;

if (!m_bSimulate) //if the user hasn't requested simulation.....

```

```
{
    //after ITL's initialize has been run we can use the driver property to determine
    //the resource descriptor.
    BSTR ResDescriptor;

    if (SUCCEEDED(hr))
    {
        hr = GetRoot()->get_IoResourceDescriptor(&ResDescriptor);
    }

    //convert reset and id query parameters to plug and play Boolean data type...
    ViBoolean VI_IdQuery, VI_Reset;

    if (IdQuery == VARIANT_FALSE)
    {
        VI_IdQuery = VI_FALSE;
    }
    else
    {
        VI_IdQuery = VI_TRUE;
    }

    if (Reset == VARIANT_FALSE)
    {
        VI_Reset = VI_FALSE;
    }
    else
    {
        VI_Reset = VI_TRUE;
    }

    //now run the VXI plug and play driver's initialise
    if (SUCCEEDED(hr))
    {
        hp34401_init (OLE2A(ResDescriptor),VI_IdQuery,VI_Reset,vi);
    }
}
}
```

IVI Configuration Server

If the steps mentioned in the above section on driver initialization are taken the IVI configuration server will be available for use as normal. If the driver is initialized using a logical name, the IVI configuration server can be used to find out the resource descriptor and repeated capability mappings from the IVI configuration server XML file.

Simulation

Simulation is available to drivers created using this wrapping scheme. In IVI-COM drivers, if simulation is turned on, simulation macros in each method and property will take care of returning a value and not executing the rest of the code in the method or property (in this case the call to the plug and play driver). In the wrapped driver case the simulate flag has a special use during initialization – so code in the driver's initialize method must decide whether the user has passed in the simulate=true option and not turn off simulation at the end of the method if this is the case.

VIVID Center Coding Wizards

The features in VIVID Center that allow complete driver functions to be generated without further coding in Visual C++ are not designed to generate code for a wrapped driver. In this case it is best to generate the driver skeleton and implement the calls to the plug and play dll in Visual C++.

Parameter Types

Both IVI-COM and VXI plug and play drivers support a limited set of data types. IVI-COM supports LONG, DOUBLE, BSTR, VARIANT_BOOL, BYTE and SAFEARRAY. The plug and play data types are defined in the file *visatype.h*. One of the jobs of the IVI-COM driver method or property is to convert the IVI-COM data type into the appropriate plug and play type, before the function in the plug and play driver is called. Data Type issues include:

1. Numeric Types. VXI plug and play drivers support signed and unsigned 8,16 and 32-bit integer types. IVI-COM only has LONG and BYTE so range issues may occur.
2. Arrays. IVI-COM drivers use SAFEARRAYS so conversion between these and regular C-type arrays will be needed.
3. Strings. IVI-COM drivers use the BSTR type, so conversion between these and C-type char * will be needed.