IVI Instrument Driver Programming Guide (Visual C++/CLI Edition)

June 2012   Revision 2.0

1- Overview

1-1 Recommendation Of IVI-COM Driver

Because Visual C++/CLI is a managed environment, IVI-COM instrument drivers that are executed under the unmanaged environment (as native codes) cannot be used directly. In general, in order to use a COM object from the managed environment, an assembly that is called "COM Interop" is required. When installing an IVI-COM instrument driver, the corresponding interop assembly is also automatically installed. Therefore this guidebook recommends using IVI-COM instrument drivers through the interop assemblies.

Notes:

- This guidebook shows examples that use KikusuiPwx IVI instrument driver (KIKUSUI PWX series DC Power Supply). You can also use IVI drivers for other vendors and other models in the same manner.

- This guidebook describes how to create 32bit (x86) programs that run under Windows7 (x64), using Visual Studio 2010 (C++/CLI).

1-2 IVI Instrument Class Interface

When using an IVI instrument driver, there are two approaches – using specific interfaces and using class interfaces. The former is to use interfaces that are specific to an instrument driver and you can utilize the most of features of the instrument. The later is to utilize instrument class interfaces that are defined in the IVI specifications allowing to utilize interchangeability features, but instrument specific features are restricted.

Notes:

- The instrument class to which the instrument driver belongs is documented in Readme.txt for each of drivers. The Readme document can be viewed from Start button➜All Programs➜Kikusui➜KikusuiPwx menu.

- If the instrument driver does not belong to any instrument classes, you can't utilize class interfaces. This means that you cannot develop applications that utilize interchangeability features.

2- Example Using Specific Interface

Here we introduce an example using specific interfaces. By using specific interfaces, you can utilize the maximum feature (or model specific functions) provided by the driver but you have to spoil interchangeability.

2-1 Creating Application Project

To simplify explanation, this guidebook shows you an example of the simplest console application. After launching Visual Studio IDE, choose **File | New | Project** menu to bring up the **New Project** dialogue. Select **CLR** under **Visual C++** language from **Installed**

Templates, then select **.NET Framework 2.0** and **CLR Console Application**. Also specify a project name (guideAppCppCli in this example) then click OK. A new application project will be then created.
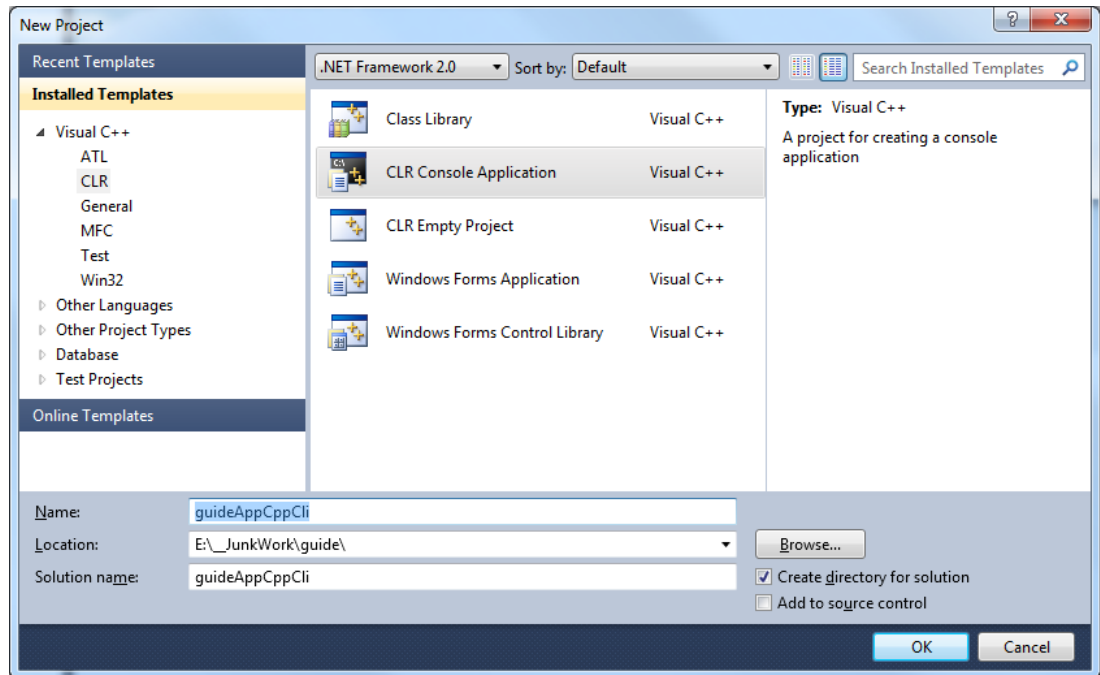

Figure 2-1 New Project Dialogue

2-2 Importing Type Libraries

What you should do first after creating a new project is, reference to interop assemblies for the IVI-COM instrument driver that you want to use. Choose **Project | Add References** menu to bring up the **Add References** dialogue, and then choose the **Browse** tab.
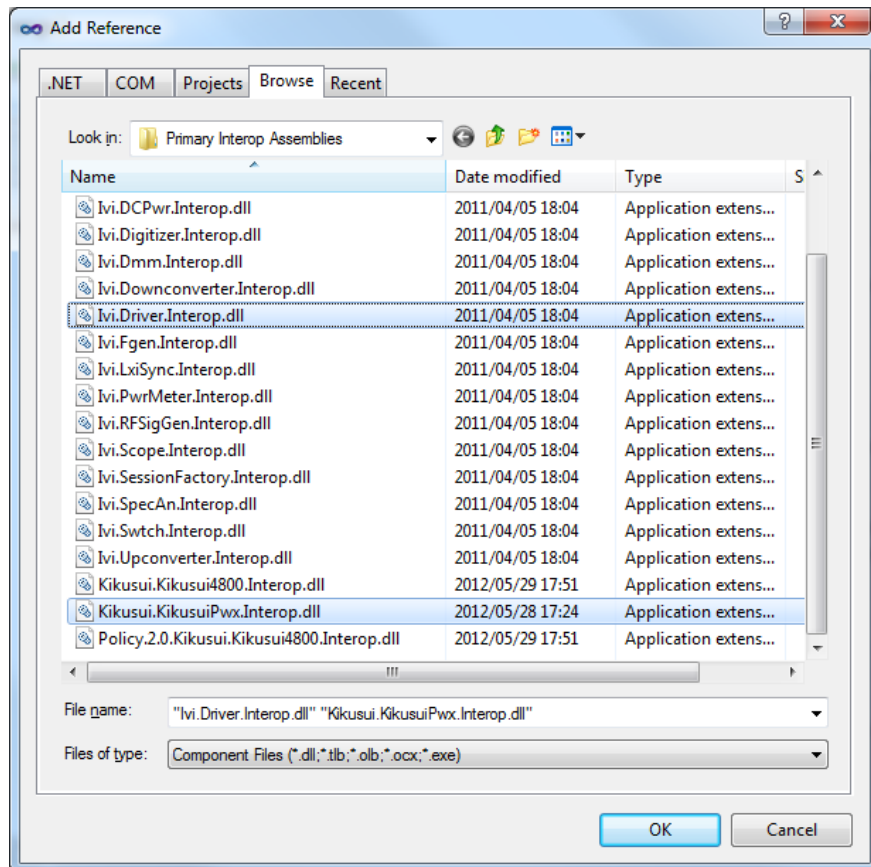
Figure 2-2 Add Reference Dialogue

Since this example uses interop assembly for the KikusuiPwx IVI-COM driver, select all of **Kikusui.KikusuiPwx.Interop.dll**, **Ivi.Driver.Interop.dll**, and **Ivi.DCPwr.Interop.dll** located in the **C:/Program Files (x86)/IVI Foundation/IVI/Bin/Primary Interop Assemblies** directory. (Multiple select is click while pressing the Ctrl key.)

Notes:

● Do not directly import the IVI driver's type library from COM tab. If doing so, your project may not be able to reference correct assemblies because name spaces by Strong Name are failed to create.. Make sure to reference the DLLS.

2-3 Specifying Name Spaces

On the source code where the `main` function is placed, declare to use name spaces for the interop assemblies that are specified to reference in the previous section. Add the following two lines in the beginning.

```
using namespace Ivi::Driver::Interop;
using namespace Kikusui::KikusuiPwx::Interop;
```

2-4 Creating Object and Initializing Session

Continuously you write codes in the `main` function. Write the following code fragments that opens a session for instrument driver object and close it. Here assume that an instrument (Kikusui PWX series DC supply) having IP address 192.168.1.5 connected with LAN interface.

Test & Measurement Instruments

```
IKikusuiPwx^ instr = gcnew KikusuiPwx();

instr->Initialize( "TCPIP::192.168.1.5::INSTR", true, true, "" );
instr->Close();
```

When creating a driver object, use **gcnew** operator for the **KikusuiPwx** component class to create its instance. Use the variable **instr** as **IKikusuiPwx** interface type to receive the created object. (The default interface of **KikusuiPwx** component is **IKikusuiPwx**, so type-casting is not needed.)

When declaring a variable as interface type, you need caret (^) to specify "referenced type". To access properties and methods of referenced types, use referecing operator (->).

Just creating the object does not communicate with instrument, so furthermore invoke **Initialize** method and **Close** method.

No need to write codes for releasing the created object. It is automatically destroyed when the scope of variable **instr** has been lost (when ending **main** method). (However, internal memory releasing timing relies on the .NET garbage collector because it is .NET environment.)

```cpp
// guideAppCppCli.cpp : main project file.

#include "stdafx.h"

using namespace System;
using namespace Ivi::Driver::Interop;
using namespace Kikusui::KikusuiPwx::Interop;

int main(array<System::String ^> ^args)
{
    //Console::WriteLine(L"Hello World");

    IKikusuiPwx^ instr = gcnew KikusuiPwx();

    instr->Initialize( "TCPIP::192.168.1.5::INSTR", true, true, "" );
    instr->Close();

    return 0;
}
```

Figure 2-3 Creating Object and Initializing Session

Now let's talk about the parameters for the **Initialize** method. Every IVI-COM instrument driver has an **Initialize** method that is defined in the IVI specifications. This method has the following parameters.

Table 2-1   Params of Initialize Method

Parameter	Type	Description
ResourceName	string	VISA resource name string. This is decided according to the I/O interface and/or address through which the instrument is connected. For example, a LAN-based instrument having IP address 192.168.1.5 will be `TCPIP::192.168.1.5::INSTR` (when VXI-11case).
IdQuery	Boolean	Specifying TRUE performs ID query to the instrument.
Reset	Boolean	Specifying TRUE resets the instrument settings.

OptionString	string	Overrides the following settings instead of default:
		
		`RangeCheck`
		`Cache`
		`Simulate`
		`QueryInstrStatus`
		`RecordCoercions`
		`Interchange Check`
		
		Furthermore you can specify driver-specific options if the driver supports `DriverSetup` features.

ResourceName specifies a VISA resource. If **IdQuery** is TRUE, the driver queries the instrument identities using a query command such as **"*IDN?"**. If **Reset** is TRUE, the driver resets the instrument settings using a reset command such as **"*RST"**.

OptionString has two features. One is what configures IVI-defined behaviours such as **RangeCheck**, **Cache**, **Simulate**, **QueryInstrStatus**, **RecordCoercions**, and **Interchange Check**. Another one is what specifies **DriverSetup** that may be differently defined by each of instrument drivers. Because the **OptionString** is a string parameter, these settings must be written as like the following example:

```
QueryInstrStatus = TRUE , Cache = TRUE , DriverSetup=12345
```

(DriverSetup=12345 is only an imaginary parameter for explanation.)

Names and setting values for the features being set are case-insensitive. Since the setting values are Boolean type, you can use any of TRUE, FALSE, 1, and 0. Use commas for splitting multiple items. If an item is not explicitly specified in the `OptionString` parameter, the IVI-defined default value is applied for the item. The IVI-defined default values are TRUE for **RangeCheck** and **Cache**, and FALSE for others.

Some instrument drivers may have special meanings for the **DriverSetup** parameter. It can specify items that are not defined by the IVI specifications when invoking the **Initialize** method, and its purpose and syntax are driver-specific. Therefore, specifying the **DriverSetup** must be at the last part on the **OptionString** parameter. Because the contents of **DriverSetup** are different depending on each driver, refer to driver's Readme document or online help.

2-5 Closing Session

To close instrument driver session, use the **Close** method.

2-6 Execution

You can execute the previous codes for the time being.

```
// guideAppCppCli.cpp : main project file.
#include "stdafx.h"
```

```
using namespace System;
using namespace Ivi::Driver::Interop;
using namespace Kikusui::KikusuiPwx::Interop;

int main(array<System::String ^> ^args)
{
    //Console::WriteLine(L"Hello World");

    IKikusuiPwx^ instr = gcnew KikusuiPwx();

    instr->Initialize( "TCPIP::192.168.1.5::INSTR", true, true, "");
    instr->Close();

    return 0;
}
```

In this example codes, content of the **main** method is executed linearly. If the instrument is actually connected and the **Initialize** method call has succeeded, the program will finish silently. However, if a communication problem has occurred or the VISA library is not configured properly, a COM exception (**System.Runtime.InteropServices.COMException**) will be generated. .
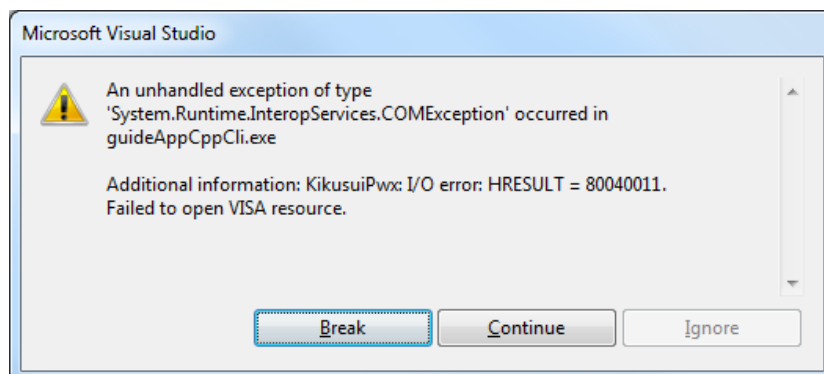
How to handle errors (exception) is explained later.

Figure 2-4 COM Exception

2-7 Repeated Capabilities, Output Collection

In case of IVI drivers for such as power supply or oscilloscope, the driver is designed assuming the instrument has multiple channels. Therefore for properties and methods that access instrument settings , there are a lot of cases that Repeated Capabilities (or Collection) are implemented. As for instrument drivers of DC power supplies, it is the Output collection.

For the case of KikusuiPwx IVI-COM driver, its concept is in **KikusuiPwxOutputs** and **KikusuiPwxOutput**. The plural name is the collection and singular name is each item (1 or more items) which may exist in the collection. In general an IVI instrument driver for DC power supply is designed assuming the instrument is a multi-track model.

They have the same name except for differences plural and singular forms. Like this, a component that has a plural name is generally called as Repeated Capabilities in the IVI spec. (Also called as Collection in COM terminology). The COM interface having plural name such as **IKikusuiPwxOutputs** normally has **Count**, **Name**, and **Item** properties (all read-only). **Count** property returns number of objects, **Name** property returns the name of the indexed object, and **Item** property returns reference to the object specified by a name. .

The following code example controls the output channel that is identified by **"Output0"** for the Kikusui PWX series DC supply.

```
...
IKikusuiPwxOutput^ output = instr->Outputs->Item["Output0"];
output->VoltageLevel = 20.0;
output->CurrentLimit = 2.0;
output->Enabled = true;
...
```

Once the **IKikusuiPwxOutput** interface has been acquired, there is no difficulty at all. The **VoltageLevel** and the **CurrentLimit** properties set voltage level and current limit settings respectively. The **Enabled** property switches output ON/OFF state.

Mind the grammar for acquiring the **IKikusuiPwxOutput** interface. This example here acquires the **IKikusuiPwxOutputs** interface though the **Output** property of the **IKikusuiPwx** interface, then acquires **IKikusuiPwxOutput** interface by using the Item property.

Now mind the parameter passed to the **Item** property. This parameter specifies the name of the single **Output** object to be referenced. Actual available names (Output Name) are however different depending on drivers. For example, KikusuiPwx IVI-COM driver uses an expression like **"Output0"**. However other drivers, even if being IviDCPwr class-compliant, may have different names. One instrument driver, for example, may use an expression like **"Channel1"**. Although available names on a particular instrument driver are normally documented in the driver's online help, you can also check them out by writing some test codes shown below.

```
IKikusuiPwxOutputs^ outputs = instr->Outputs;
int n;
int c = outputs->Count;

 for( n=1; n<=c; n++) {
     String^ name;
     name = outputs->Name[n];
     System::Diagnostics::Debug::WriteLine(name);
 }
```

The **Count** property returns number of single objects that the repeated capabilities have. The **Name** property returns the name of single object for the given index. The name is exactly the one that can be passed to the **Item** property as a parameter. In the above example, the codes iterate from the index 1 to **Count** by using the **for** block. Mind that the index numbers for the **Name** parameter is one-based, not zero-based.

3- Error Handling

In the previous examples, there was no error handling processed. However, setting an out-of-range value to a property or invoking an unsupported function may generate an error from the instrument driver. Furthermore, no matter how the application is designed and implemented robustly, it is impossible to avoid instrument I/O communication errors.

When using IVI-COM instrument drivers, every error generated in the instrument driver is transmitted to the client program as a COM exception. In case of C#, a COM exception can be handled by using **try, catch, finally** blocks.

Now let's change the example of setting voltage and current as follows.

```
try {
    IKikusuiPwx^ instr = gcnew KikusuiPwx();

    instr->Initialize( "TCPIP::192.168.1.5::INSTR", true, true, "");
```

```
        IKikusuiPwxOutput^ output = instr->Outputs->Item["Output0"];
        output->VoltageLevel = 20.0;
        output->CurrentLimit = 2.0;
        output->Enabled = true;


        instr->Close();
    }
    catch( System::Runtime::InteropServices::COMException^ e) {
        String^ msg =
            String::Format( "{0} {1}",
                e->Message,
                "0x" +  Convert::ToString( e->ErrorCode, 16));
        System::Diagnostics::Debug::WriteLine( msg);
    }

    catch( Exception^ e) {
    }
```

In this example, codes that may generates exceptions are written in the **try** block. For example, if the name passed to the **Item** property is wrong, if an out-of-range value is passed to **VoltageLevel**, or if an instrument communication error is generated, a COM exception will be generated in the instrument driver. Any exceptions generated inside the **try** block are handled in a **catch** block if corresponding block exists. (If no corresponding **Catch** block is fond, the process will be handled as an Unhandled Exception and the program will crash.) Above example just displays a simple message in the console when an exception has occurred.

4- Example Using Class Interface

Now we explain how to use class interfaces. By using class interfaces, you can swap the instruments without recompiling/relinking your application codes. In this case, however, IVI-COM instrument drivers for both pre-swap and post-swap models must be provided, and these drivers both must belong to the same instrument class. There is no interchangeability available between different instrument classes.

4-1 Virtual Instrument

What you have to do before creating an application that utilizes interchangeability features is create a virtual instrument. To realise interchangeability features, you should not write codes that are very specific to a particular IVI-COM instrument driver (e.g. creating an object instance directly as KikusuiPwx type) and should not write a specific VISA resource name such as "TCPIP::192.168.1.5::INSTR". Writing them directly in the application spoils interchangeability.

Instead, the IVI-COM specifications define methods to realise interchangeability by placing an external IVI configuration store. The application indirectly selects an instrument driver according to contents of the IVI Configuration Store, and accesses the indirectly loaded driver through the class interfaces.

The IVI Configuration Store is normally **C:/ProgramData/IVI Foundation/IVI /IviConfigurationStore.XML** file and is accessed through the IVI Configuration Server DLL. This DLL is mainly used by IVI instrument drivers and some VISA/IVI configuration tools, not by end-user applications. Instead, you can edit IVI driver configuration by using NI-MAX (NI Measurement and Automation Explorer) bundled with NI-VISA or IVI Configuration Utility bundled with KI-VISA.

Notes:

● As for how to edit virtual instrument settings using NI-MAX, refer to "IVI Instrument Driver Programming Guide (LabVIEW Edition or LabWindows/CVI Edition)".

This guidebook assumes that a virtual instrument having the logical name `mySupply` is already created, using KikusuiPwx driver, and using a VISA resource " TCPIP::192.168.1.5::INSTR ".

4-2 Importing Type Libraries

Similarly to the example using specific interfaces, create a new project of CLR Console Application as C++ language.

Similarly to the example using specific interfaces, create a new project of Console Application as C# language.

What you should do first after creating a new project is, reference to interop assemblies for the IVI-COM instrument driver that you want to use and the IVI Session Factory. The IVI Session Factory is one of indispensable IVI standard libraries, which allows to load the appropriate IVI driver parsing an IVI logical name (virtual instrument) configured in the IVI Configuration Store.

Choose **Project | Add References** menu to bring up the **Add References** dialogue, and then choose the **Browse** tab.
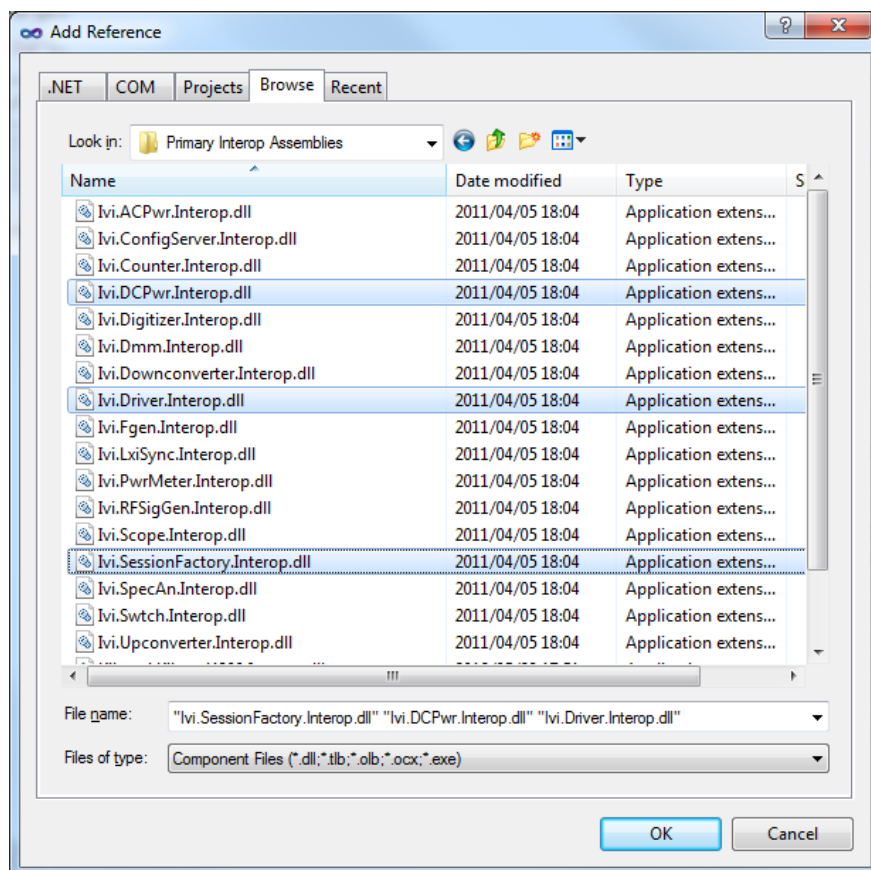

Figure 4-1 Add Reference Dialogue

Since this example uses interop assembly for the KikusuiPwx IVI-COM driver, select all of **Ivi.Driver.Interop.dll**, **Ivi.DCPwr.Interop.dll**, and **Ivi.SessionFactory.Interop.dll** located in the **C:/Program Files (x86)/IVI Foundation/IVI/Bin/Primary Interop Assemblies** directory. (Multiple select is click while pressing the Ctrl key.)

Once reference setting is completed, write the following code fragments. (Here write everything at once including the exception handlers mentioned before.)

```
#include "stdafx.h"

using namespace System;
using namespace Ivi::Driver::Interop;
using namespace Ivi::DCPwr::Interop;
using namespace Ivi::SessionFactory::Interop;

int main(array<System::String ^> ^args)
{
    //Console::WriteLine(L"Hello World");

    IIviSessionFactory^ sf = gcnew IviSessionFactory();

    try {
        IIviDCPwr^ instr = (IIviDCPwr^)sf->CreateDriver( "mySupply");

        instr->Initialize( "mySupply", true, true, "");

        IIviDCPwrOutput^ output = instr->Outputs->Item["Track_A"];
        output->VoltageLevel = 20.0;
        output->CurrentLimit = 2.0;
        output->Enabled = true;

        instr->Close();
    }
    catch( System::Runtime::InteropServices::COMException^ e) {
        String^ msg =
            String::Format( "{0} {1}",
                e->Message,
                "0x" +  Convert::ToString( e->ErrorCode, 16));
        System::Diagnostics::Debug::WriteLine( msg);
    }

    return 0;
}
```

Let's explain from the beginning.

4-3 Specifying Name Spaces

In the program source code, declare to use name spaces for the interop assemblies that are specified to reference in the previous section. Add the following three lines in the beginning.

```
using namespace Ivi::Driver::Interop;
using namespace Ivi::DCPwr::Interop;
using namespace Ivi::SessionFactory::Interop;
```

At first, notice that any type names beginning with Kikusui are not used. This example code no longer has dependency on KikusuiPwx. Instead, the IVI class interfaces of **IviDriver** and **IviDCPwr**, and **SessionFactory** object are used.

4-4 Creating Object and Initializing Session

In contrast using specific interfaces, any dependency to specific components such as KikusuiPwx cannot be written. Instead, it creates an instance of **SessionFactory** object,

and indirectly create a driver object that is configured in the IVI Configuration Store by using **CreateDriver** method..

Now create an **IviSessionFactory** object, then obtain the reference to **IIviSessionFactory** interface.

```
IIviSessionFactory^ sf = gcnew IviSessionFactory();
```

Next, invoke the **CreateDriver** method passing the IVI Logical Name (Virtual Instrument). The created object is actually an instance of KikusuiPwx driver, but here store the reference to **IIviDCPwr** interface into the variable **instr**.

```
IIviDCPwr^ instr = (IIviDCPwr^)sf->CreateDriver( "mySupply");
```

If IVI Configuration Store is properly configured, the code will execute without generating exceptions. However, at this point of time, it has not communicate with the instrument yet. The DLL of IVI -COM driver is just loaded.

Then invoke **Initialize** method. At this point of time, communications with the instrument begins. The 1st parameter to **Initialize** method was originally a VISA address (VISA IO resource) but, here it shall be the IVI Logical Name. The IVI Configuration Store already knows the linked info concerning to this Logical Name, such as Hardware Asset, therefore the VISA address specified there will be actually applied.

```
instr->Initialize( "mySupply", true, true, "");
```

As for IviDCPwr class, the **Output** object of DC power supply is found in the **Outputs** collection. Similarly to the example of using specific interface, it obtains the reference to the single **Output** object from the collection. In this case, the interface type is **IIviDCPwrOutput** instead of **IKikusuiPwxOutput**.

```
IIviDCPwrOutput^ output = instr->Outputs->Item["Track_A"];
output->VoltageLevel = 20.0;
output->CurrentLimit = 2.0;
output->Enabled = true;
```

Mind the parameter that is passed to **Item** parameter. This parameter specifies the name of single **Output** object that you want to reference to. In the example using specific interfaces it passed Physical Name that may be different by driver implementation basis, but not here. This example cannot use such Physical Names very specific to an instrument driver implementation (in fact it is possible to use but shall not to avoid spoiling interchangeability), so we use a Virtual Name.

The virtual name **"Track_A"** that is used in the above example is what specified to map to the physical name **"Output0"** in the IVI Configuration Store.

4-5 Exchanging Instrument

Example shown so far were set to use kipwx instrument driver as the virtual instrument configuration. Now what happens if changing the instrument to the one that is hosted by AgN57xx driver (Agilent N5700 series DC Power Supply)? In this case, you don't have to

recompile/relink your application, however you have to change the configuration for IVI Logical Name (virtual instrument). Basically the configuration shall change:

- Software Module in Driver Session tab (kipwx➜AgN57xx)

- map target of Virtual Names (Output0➜Output1)

- IO Resource Descriptor in Hardware Asset (changing to the VISAaddress of post-swap instrument)

- Once the configuration is properly set, the above example will function with the post-swap instrument without having to recompile.

Once the configuration is properly set, the above example will function with the post-swap instrument without having to recompile.

Notes:

- For how to configure virtual instruments, refer to "IVI Instrument Driver Programming Guide (LabVIEW Edition or LabWindows/CVI Edition)".

- The interchangeablity feature utilizing IVI class drivers does not guarantee the correct operation between pre-swapping and post-swapping instruments. Please make sure to confirm that your system correctly functions after swapping the instruments.