IVI 計測器ドライバ プログラミング・ガイド (Visual C++/CLI 編)

June 2012　Revision 2.0

1- 概要

1-1 IVI-COM ドライバの推奨

C++/CLI はマネージド環境なので、アンマネージド環境(ネイティブ・コード)で実行される IVI-COM 計測器ドライバを直接使用する事はできません。一般に COM オブジェクトをマネージド環境から使う場合は、COM Interop と呼ばれるアセンブリが必要になります。IVI-COM 計測器ドライバをインストールした際に、インタロップ・アセンブリも自動的にインストールされています。従って、本ガイドブックでは IVI-COM 計測器ドライバをインタロップ経由で使用する事を推奨します。

Notes:

- 本ガイドブックでは、IVI-COM KikusuiPwx 計測器ドライバ(KIKUSUI PWX シリーズ直流電源)を使用する例を示します。他社メーカー及び他機種用の IVI 計測器ドライバでも、ほぼ同様の手順で使用できます。
- 本ガイドブックでは、Visual Studio 2010 (C++/CLI)を使用し、Windows7 (x64)上で動作する 32bit(x86) .NET プログラムを作成する場合を例に説明します。

1-2 IVI 計測器クラス・インターフェース

IVI-COM 計測器ドライバを利用する場合、スペシフィック・インターフェースを利用する方法とクラス・インターフェースを利用する方法の 2 種類があります。前者は計測器ドライバの固有インターフェースを利用するもので、使用する計測器の機能を最大限に利用する事ができます。後者は IVI 仕様書で定義されている計測器クラスのインターフェースを利用するもので、インターチェンジャビリティ機能を利用する事ができますが、機種固有の機能を使うことは制限されます。

Notes:

- 計測器ドライバが所属する計測器クラスについては、ドライバ毎の Readme.txt に記載されています。Readme 文書は、Start➔All Programs➔Kikusui➔KikusuiPwx メニューから開く事ができます。
- 計測器ドライバが如何なる計測器クラスにも属していない場合、クラス・インターフェースを利用する事はできません。つまりこの場合、インターチェンジャビリティ機能を利用するアプリケーションを作成する事は出来ません。

2- スペシフィック・インターフェースを使用するサンプル

ここでは、スペシフィック・インターフェースを使用したサンプルを示します。スペシフィック・インターフェースを使用すると、計測器ドライバで提供される機能を最大限に利用する事ができますが、インターチェンジャビリティを実現する事はできません。

2-1 アプリケーション・プロジェクトの作成

ここでは説明を簡略化する為、最も簡素なコンソール・アプリケーションを例に説明します。Visual Studio 統合環境を起動したら、**File | New | Project** メニューを選択して **New Project** ダイアログを表示します。**Installed Templates** から **Visual C++言語の CLR を選択し、.NET**

Framework 2.0、**CLR Console Application** を選択します。更にプロジェクト名(この例では guideAppCppCli)を指定して **OK** をクリックします。アプリケーションのプロジェクトが新規に作成されます。

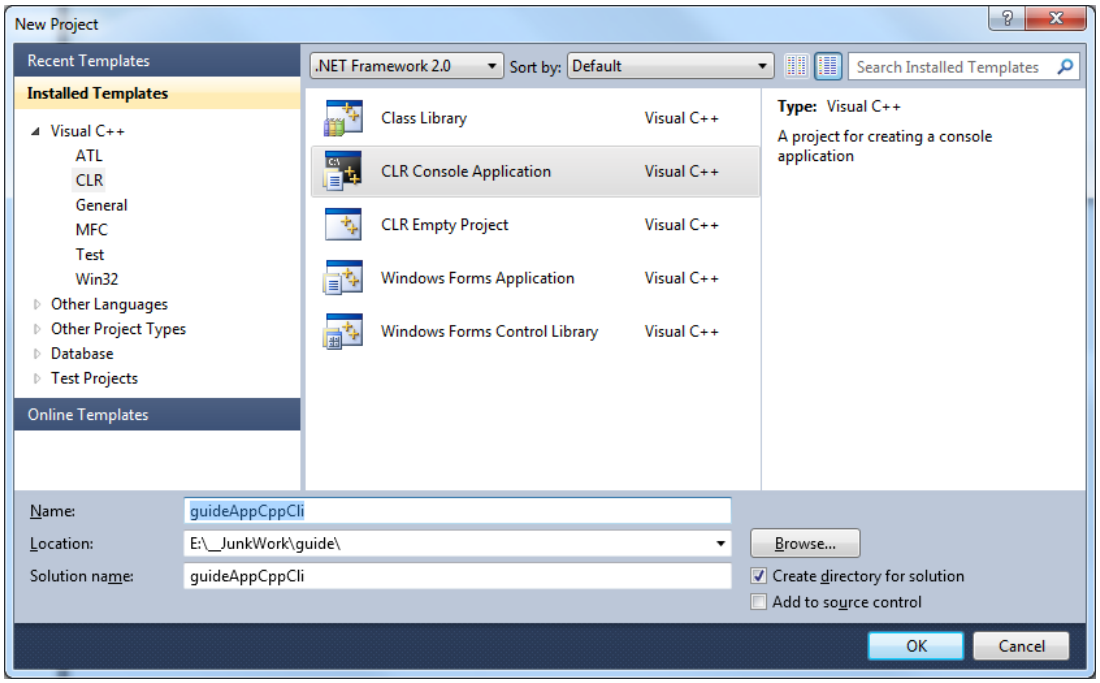

Figure 2-1 New Project ダイアログ

2-2 タイプライブラリのインポート

新規プロジェクトを作成したあと最初にすべき事は、利用したい IVI-COM 計測器ドライバのインタロップ・アセンブリを参照する事です。**Project | References** メニューを選択してプロジェクトのプロパティを表示し、**Add New References**...ボタンをクリックして **Add Reference** ダイアログを表示します。そこで **Browse** タブを選択します。

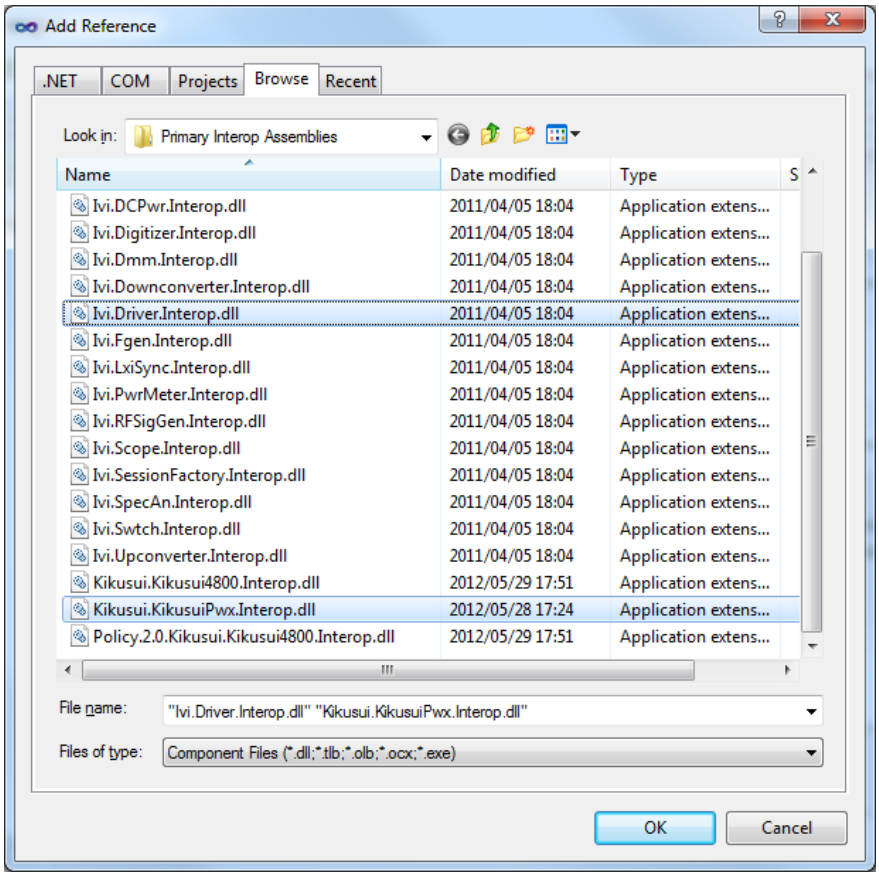

Figure 2-2 Add Reference ダイアログ

ここでは KikusuiPwx IVI-COM ドライバのインタロップ・アセンブリを使用するので、**C:/Program Files (x86)/IVI Foundation/IVI/Bin/Primary Interop Assemblies** ディレクトリにある **Kikusui.KikusuiPwx.Interop.dll** と **Ivi.Driver.Interop.dll** の両方を同時選択して下さい。(複数同時選択は Ctrl キーを押しながらクリック。)

Notes:

- COM タブから IVI ドライバのタイプ・ライブラリを直接インポートしないで下さい。そのようにすると、ストロング・ネームによる名前空間が作成されず、正しいインタロップ・アセンブリを参照できない場合があります。必ず IVI 標準ディレクトリ傘下の **Primary Interop Assemblies** サブ・ディレクトリにある DLL を参照して下さい。

2-3 名前空間の指定

main 関数のあるソースコード上で、前項で参照設定したインタロップ・アセンブリに対して名前空間の利用を宣言します。下記の 2 行を冒頭に追加して下さい。

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

2-4 計測器オブジェクトの作成とセッションのイニシャライズ

引き続き **main** 関数内にコードを書いていきます。下記のような、計測器ドライバ・オブジェクトの作成とセッションのオープン、クローズを行うコードを書きます。ここでは、IP アドレス 192.168.1.5 を持つ計測器(Kikusui PWX シリーズ直流電源)が LAN で接続されていると仮定します。

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

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

ドライバ・オブジェクトを作成する際は、**KikusuiPwx** コンポーネント・クラスに対して **gcnew** 演算子を使ってそのインスタンスを作成します。作成されたオブジェクトは、**IKikusuiPwx** インターフェース型の変数 **instr** で受け取ります。(**KikusuiPwx** コンポーネントのデフォルト・インターフェースは **IKikusuiPwx** なので、タイプ・キャストは不要です。)

インターフェース型で変数を宣言する際は「参照型」であることを示すためキャレット(^)が付きます。また参照型のプロパティやメソッドには参照演算子(->)を使ってアクセスします。

オブジェクトを作成しただけでは計測器と通信しないので、更に Initialize()メソッドと Close()メソッドを呼び出します。

作成したオブジェクトの解放処理は特に記述しません。変数 **instr** のスコープが失われた段階(**main** 関数の終了時)で自動的に廃棄されます。(但し.NET 環境なので、内部のメモリ解放等のタイミングは.NET ガベージ・コレクタの動作に依存します。)

```
// 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 オブジェクトの作成とセッションの初期化

ここで、**Initialize** メソッドのパラメータについて説明しましょう。全ての IVI-COM 計測器ドライバは、IVI 仕様書で定義された **Initialize** メソッドを持っています。このメソッドには、以下のようなパラメータがあります。

Table 2-1 Initialize メソッドのパラメータ

パラメータ	タイプ	説明
ResourceName	string	VISA リソース名の文字列。計測器が接続されている I/O インターフェース、アドレスなどによって決定される。例えば、"TCPIP::192.168.1.5::INSTR"の例では、IP アドレス 192.168.1.5 を持つ LAN 接続の計測器で VXI-11 インターフェースを使用する事を意味する。
IdQuery	Boolean	true を指定した場合、計測器に対して ID クエリを行う。
Reset	Boolean	true を指定した場合、計測器の設定をリセットする。
OptionString	string	RangeCheck Cache Simulate QueryInstrStatus RecordCoercions Interchange Check に関する設定を、デフォルト以外に指定できる。更に、計測器ドライバが DriverSetup 機能をサポートする場合、その設定を行うことができる。

ResourceName には VISA リソースを指定します。**IdQuery** に true を指定した場合は、計測器に対して**"*IDN?"**クエリなどを発行して機種情報を問い合わせます。**Reset** に true を指定した場合は、**"*RST"**コマンドなどを発行して計測器の設定をリセットします。

OptionString には、2 つの機能があります。1 つは **RangeCheck**, **Cache**, **Simulate**, **QueryInstrStatus**, **RecordCoercions**, **Interchange Check**,などの IVI 定義の動作を設定します。もう 1 つは、計測器ドライバ毎に独自に定義される **DriverSetup** を指定します。**OptionString** は文字列パラメータなので、これらの設定は下のサンプルのような書式でなければなりません。

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

(DriverSetup=12345 はあくまでも説明上の内容であり、架空のパラメータです。)

設定したい機能の名称及び設定値はケース・インセンシティブ(大文字と小文字の区別なし)です。設定値は Boolean 型なので、TRUE、FALSE、1、0 の何れかが有効です。複数の項目を設定する場合は、カンマで区切ります。**OptionString** パラメータで特に設定値を指定しない場合、IVI 仕様書で定義されたデフォルト値が適用されます。IVI 仕様書で定義されたデフォルト値は、**RangeCheck** と **Cache** だけが TRUE で、その他は全て FALSE です。

計測器ドライバによっては、**DriverSetup** パラメータが意味を持つ場合もあります。これは、IVI 仕様書では定義されない項目を **Initialize** の呼び出し時に指定するもので、利用目的や書式はドライバ依存です。従って **DriverSetup** の指定を行う場合、それは **OptionString** の最後の項目として指定される必要があります。**DriverSetup** の指定内容はドライバ毎に異なるので、ドライバの Readme 文書又はオンライン・ヘルプなどを参照してください。

2-5 セッションのクローズ

計測器ドライバによるセッションをクローズするには、**Close** メソッドを使います。

2-6 実行

ここまでのコードだけで、とりあえず実行する事は可能です。

```
// 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;
}
```

このサンプルコードでは **main** 関数の内容が直線的に実行されます。実際に計測器が接続されていて **Initialize** メソッドが成功すれば何事もなくプログラムは終了しますが、通信に失敗した場合や、VISA ライブラリの設定が正しく行われていない場合などは、COM 例外(**System.Runtime.InteropServices.COMException**)を発生します。

エラー(例外)の処理方法については後述します。

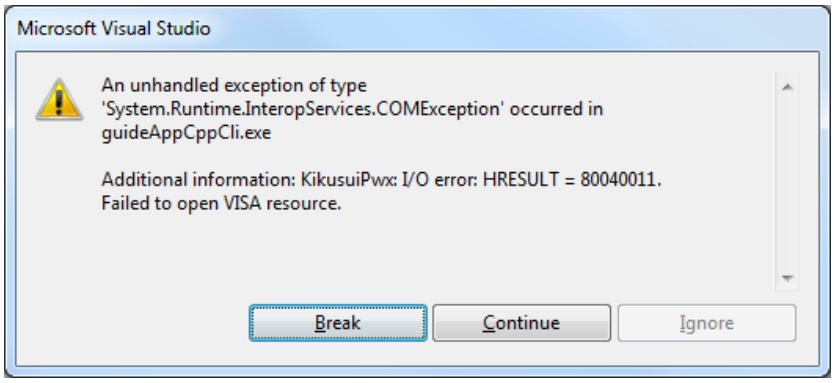

Figure 2-4 COM 例外

2-7 リピーテッド・キャパビリティ、Output コレクション

電源装置やオシロスコープなどの IVI ドライバでは、計測器が複数のチャンネルが装備されている事を前提に設計されています。従って計測器の設定に関する操作を行うプロパティやメソッドは、リピーテッド・キャパビリティ(或いはコレクション)と呼ばれるオブジェクト配列の概念を実装するケースが多く見られます。直流電源の計測器ドライバの場合は Output コレクションがそれに該当します。

KikusuiPwx IVI-COM ドライバの場合、**KikusuiPwxOutputs** と **KikusuiPwxOutput** がそうです。複数形のオブジェクトがコレクションであり、単数形の物がその中に複数(1 個以上)生息します。一般に直流電源用の計測器ドライバは、(実際のチャンネル数とは関係なく)複数の出力チャンネルを持つマルチ・トラック電源装置の概念を前提に設計されています。

これらは単数形と複数形の違いを除いて同じ名前になっています。このように複数形の名前を持つコンポーネントは、IVI 仕様書では一般にリピーテッド・キャパビリティ(一般的な COM の用語ではコレクション)と呼ばれます。**IKikusuiPwxOutputs** のような複数形の名前を持つ COM インターフェースは通常 **Count**、**Name**、**Item** プロパティ(いずれもリード・オンリー)を持ちます。**Count** プロパティはオブジェクトの個数を返し、**Name** プロパティはインデックス番号で指定されたオブジェクトの名前を返します。**Item** プロパティは、指定された名前を持つオブジェクトへの参照を返します。

下記のコード例は、Kikusui PWX シリーズ直流電源の"Output0"という名前で識別される出力チャンネルを制御するものです。

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

一旦 IKikusuiPwxOutput インターフェースを取得してしまえば、あとは難しい事はありません VoltageLevel プロパティは電圧レベル設定を、CurrentLimit プロパティは電流リミット設定を、それぞれ行います。Enabled プロパティは出力の ON/OFF 設定を行います。

IKikusuiPwxOutput インターフェースを取得する際の記述に注意してください。ここでは、IKikusuiPwx インターフェースの Outputs プロパティを通じて IKikusuiPwxOutputs を取得し、Item プロパティを使って IKikusuiPwxOutput インターフェースを取得しています。

ここで、Item プロパティに渡しているパラメータに注意する必要があります。このパラメータは参照したい単品の Output オブジェクトの名前を指定しています。しかしここで使える名前は、ドライバごとにそれぞれ違います。例えば KikusuiPwx IVI-COM ドライバでは、"Output0"のような表現になっていますが、他のドライバでは(たとえ、同じ IviDCPwr クラスであっても)違ったものになります。例えば、他の計測器ドライバでは、"Channel1"のような表現かも知れません。特定の計測器ドライバで使用可能な名前は、通常はドライバのオンライン・ヘルプなどに記載されていますが、下記のようなコードを書くことでそれらを調べる事も可能です。

```
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);
 }
```

Count プロパティは、リピーテッド・キャパビリティが持つ単品オブジェクトの個数を返します。Name プロパティは、与えられたインデックス番号の単品オブジェクトが持つ名前を返します。この名前こそが、Item プロパティに渡す事のできるパラメータになるのです。上記の例では、for 文を使って、インデックス 1 から Count までを反復処理しています。Name パラメータに渡すインデックス番号は 0 ベースではなく 1 ベースである事に注意してください。

3- エラー処理

これまで示したサンプルでは、エラー処理を何も行っていませんでした。しかし実際には、範囲外の値をプロパティに設定したり、サポートされていない機能を呼び出したりすると、計測器ドライバがエラーを発生する事があります。また、どんなに堅牢に設計・実装されたアプリケーションでも、計測器との I/O 通信エラーは避けることが出来ません。

IVI-COM 計測器ドライバでは、計測器ドライバ内で発生したエラーは全て COM 例外としてクライアント・プログラムに伝えられます。C++/CLI の場合、COM 例外は try、catch、finally ブロックを使って処理する事が出来ます。

先ほど示した、電圧・電流を設定するコードを下記のように変更してみましょう。

```
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) {
    }
```

ここでは、例外を発生するかもしれないコードを **try** ブロックの中に書いています。例えば、**Item** プロパティに渡した名前が間違っている場合、**VoltageLevel** に設定する値が適正範囲から外れている場合、或いは計測器との通信に失敗した場合などはいずれも、計測器ドライバ内で COM 例外が発生します。**try** ブロックの中で発生した例外は、対応する **catch** ブロックがあればそこで処理されます。(対応する **catch** ブロックがない場合は Unhandled Exception として処理され、プログラムはクラッシュします。)上記の例では、例外が発生した場合に簡単なメッセージを組み立て、コンソールに表示しています。

4- クラス・インターフェースを使用するサンプル

ここでは、計測器クラス・インターフェースを使用したサンプルを示します。計測器クラス・インターフェースを使用すると、アプリケーションを再度コンパイル・リンクすることなく、計測器を別の機種に交換する事ができます。但しその場合、交換前後の両機種に対して IVI-COM 計測器ドライバが提供されており、且つそれらのドライバが同じ計測器クラスに属している必要があります。異なる計測器クラス間でのインターチェンジャビリティは実現できません。

4-1 仮想インストルメント

インターチェンジャビリティ機能を利用するアプリケーションの作成を行う前にやっておかなければならない事は、仮想インストルメントの作成です。インターチェンジャビリティ機能を実現するには、アプリケーション・コード内に特定の IVI-COM 計測器ドライバに依存した記述(例えば KikusuiPwx 型で直接オブジェクトを生成)したり、"TCPIP::192.168.1.5::INSTR"のような特定 VISA アドレス(リソース名)の記述などをするべきではありません。これらの事柄をアプリケーション内に直接記述すると、インターチェンジャビリティを損ないます。

その代わりに、IVI 仕様では計測器ドライバとアプリケーションの外部に IVI コンフィグレーション・ストアを置く事によってインターチェンジャビリティを実現します。アプリケーションは特定機種用の計測器ドライバを直接使うのではなく、計測器クラス・インターフェースを使います。その際に、IVI コンフィグレーション・ストアの内容に従って計測器ドライバ DLL の選択を行い、間接的にロードされた計測器ドライバを特定機種に依存しないクラス・インターフェースを通じてアクセスします。

IVI コンフィグレーション・ストアは通常、**C:/ProgramData/IVI Foundation/IVI/IviConfigurationStore.xml** ファイルで、IVI Configuration Server DLL を通じてアクセスされます。この DLL を利用するのは、主に IVI 計測器ドライバや一部の VISA/IVI コンフィグレーション・ツ

ールであって、アプリケーションからは通常は使いません。その代わりに、NI-VISA に付属の NI-MAX (NI Measurement and Automation Explorer)か又は KI-VISA に付属の IVI Configuration Utility を使用して IVI ドライバのコンフィグレーションを行います。

Notes:

- NI-MAX を使用して仮想インストルメントの設定を行う手順に関しては、「IVI 計測器ドライバ・プログラミング・ガイド(LabVIEW 編又は LabWindows/CVI 編)」を参照してください。

このガイドブックでは、mySupply という IVI ロジカル・ネームで仮想インストルメントが既に作成されていて、KikusuiPwx ドライバを使用し、VISA リソース"TCPIP::192.168.1.5::INSTR"を使用する、という設定が行われているものとします。

4-2 タイプ・ライブラリのインポート

スペシフィック・インターフェースを使用するサンプルと同様に、C++言語の CLR Console Application で新規にプロジェクトを作ります。

新規プロジェクトを作成したあと最初にすべき事は、利用したい IVI 計測器クラスのインタロップ・アセンブリと、IVI Session Factory のインタロップ・アセンブリを参照設定する事です。IVI Session Factory は、IVI コンフィグレーション・ストアに設定された IVI ロジカル・ネーム(仮想インストルメント)から適切な IVI ドライバ・モジュールをロードするのに不可欠な IVI 標準ライブラリです。

Project | References メニューを選択してプロジェクトのプロパティを表示し、**Add New References**...ボタンをクリックして **Add Reference** ダイアログを表示します。そこで **Browse** タブを選択します。

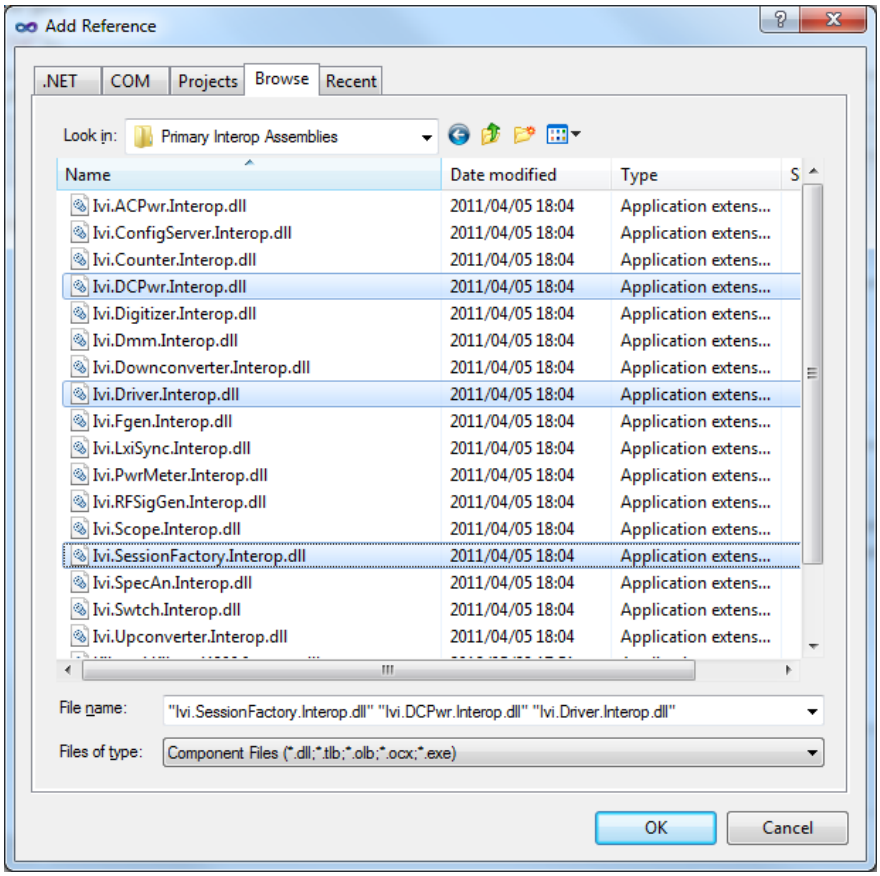

Figure 4-1 Add Reference ダイアログ

ここでは IviDCPwr のインタロップ・アセンブリを使用するので、**C:/Program Files (x86)/IVI Foundation/IVI/Bin/Primary Interop Assemblies** ディレクトリにある

Ivi.Driver.Interop.dll、**Ivi.DCPwr.Interop.dll**、**Ivi.SessionFactory.Interop.dll**、の全てを同時選択して下さい。(複数同時選択は Ctrl キーを押しながらクリック。)

参照設定を終えたら、下記のようなコードを書いていきます。(ここでは、既に説明した例外処理も含めて一気に書きます。)

```
#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;
}
```

順番に説明していきましょう。

4-3 名前空間の指定

前項で参照設定したインタロップ・アセンブリに対して名前空間の利用を宣言します。今回は下記の3行を冒頭に追加しました。

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

Kikusui で始まる名前空間が宣言されていない点に注意して下さい。このサンプル・コードはもはや KikusuiPwx への依存を含んでいません。その代わりに、**IviDriver**、**IviDCPwr** の IVI クラス・インターフェースと **SeesionFactory** の名前空間が宣言されています。

4-4 オブジェクトの作成とセッションのイニシャライズ

スペシフィック・インターフェースを使う場合と異なり、KikusuiPwx のような特定コンポーネントへの依存を記述することは出来ません。その代わりに、**SessionFactory** オブジェクトのインスタンスを作成し、**CreateDriver** メソッドを呼び出すことで、IVI コンフィグレーション・ストアに設定されているドライバ・オブジェクトを間接的に作ります。

まず **IviSessionFactory** オブジェクトを作成し、**IIviSessionFactory** インターフェースへの参照を取得します。

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

次に、既に作成した IVI ロジカル・ネーム(仮想インストルメント)を指定して **CreateDriver** メソッドを呼び出します。作成されたドライバ・オブジェクトは実際には KikusuiPwx ドライバのオブジェクトですが、ここでは **IIviDCPwr** インターフェースへの参照を変数 **instr** に保持します。

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

IVI Configuration Store が正しく設定されていれば、例外を発生することなく実行できるはずです。但し、この時点ではまだ計測器とは通信していません。IVI-COM ドライバの DLL がロードされただけです。

そして **Initialize** メソッドを呼び出します。この時点で計測器との通信が開始します。**Initialize** メソッドに渡す最初のパラメータは本来 VISA アドレス(VISA IO リソース)ですが、ここでは IVI ロジカル・ネームを渡します。IVI コンフィグレーション・ストアにはこのロジカル・ネームとリンクする Hardware Asset 情報があるので、そこで指定した VISA アドレスが実際には適用されます。

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

IviDCPwr クラスでは直流電源の「アウトプット」オブジェクトは **Outputs** コレクションの中にあります。スペシフィック・インターフェースでの例題と同様に、コレクションから単一の **Output** オブジェクトへの参照を取得します。この場合、**IKikusuiPwxOutput** インターフェースではなく、**IIviDCPwrOutput** インターフェース型となります。

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

ここで、**Item** プロパティに渡しているパラメータに注意する必要があります。このパラメータは参照したい単品の **Output** オブジェクトの名前を指定しています。スペシフィック・インターフェースを使用した例ではドライバごとにそれぞれ異なる名前(フィジカル・ネーム)を直接渡していしましたが、ここでは違います。ここでは特定の計測器ドライバに依存したフィジカル・ネームは使えない(使っても動作するが、それではインターチェンジャビリティを損なう)ので、バーチャル・ネームを指定します。

上記の例で使用しているバーチャル・ネーム**"Track_A"**は IVI コンフィグレーション・ストアで**"Output0"**というフィジカル・ネームにマップされるように指定された物です。

4-5 計測器の交換

これまでの例では、仮想インストルメントのコンフィグレーションとして KikusuiPwx(kipwx)計測器ドライバを使うように設定しましたが、ここで計測器を例えば AgilentN57xx(又は AgN57xx)ドライバでホストされるもの(Agilent N5700 シリーズ直流電源)に交換するとどうなるでしょう。その場合には、アプリケーションを再度コンパイル・リンクする必要はありませんが、"mySupply"という IVI ロジカル・ネーム(仮想インストルメント)のコンフィグレーション内容を変更する必要があります。

変更しなければならないコンフィグレーションは基本的には、

- Driver Session タブにある Software Module の変更(kipwx➔AgN57xx)
- Virtual Names の展開先マップ変更(Output0➔Output1)
- Hardware Asset タブにある IO Resource Descriptor の変更(交換後の接続先 VISA アドレスへ)

という具合になります。コンフィグレーションが正しく設定されれば、上記のサンプルは再度コンパイル・リンクをせずにそのまま交換後の計測器でも動作します。

Notes:

- 仮想インストルメントのコンフィグレーション方法については、「計測器ドライバ・プログラミング・ガイド(LabVIEW 編又は LabWindows/CVI 編)」を参照してください。
- IVI クラス・インターフェースを利用したインターチェンジャビリティ機能は、計測器交換前後での動作を保証するわけではありません。交換後のシステムが正常に機能するかどうか十分に検証してから運用して下さい。