IVI-COM Instrument Driver Programming Guide (C#.NET Edition)

Dec 2003 Revision 1.1

1- Overview

1-1 Using IVI-COM Drivers in C#.NET

Because C#.NET is a managed environment, IVI-COM instrument drivers that are executed under the unmanaged environment cannot be used directly. In general, in order to use a COM object from the managed environment, an assembly that is called RCW(Runtime Callable Wrapper) associated with it is necessary. Fortunately this assembly can be automatically generated from the type library by using Add Reference menu in the Visual Studio.NET integrated development environment, or by using the command-line utility TLBIMP.EXE (Type Library Importer).

When using an IVI-COM 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 utilise the most of features of the instrument you use. The later is to utilise instrument class interfaces that are defined in the IVI specifications allowing utilising interchangability 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➔Program➔IVI folder.
>
> If the instrument driver does not belong to any instrument classes, you can't utilise class interfaces. This means that you cannot develop applications that utilise interchangability features.

1-2 Creating An Application Project

This document explains how to develop a form-oriented application that is the most popular style in C#.NET. After launching the Visual Studio.NET integrated development environment, choose **File | New | Project** menu to bring up the **New Project** dialogue. Select **C# Project** from **Project Types**, select **Windows Application** from **Templates**, give a project name, and then click **OK**. A new application project will be then created.

> Notes:
>
> This guidebook assumes that you use IVI-COM Kikusui4800 instrument driver (for KIKUSUI PIA4800 series DC Power Supply Controller). You can also use IVI-COM instrument drivers for other models in the same manner.

2- Example Using Specific Interfaces

Here we introduce an example using specific interfaces. By using specific interfaces, you can utilise the maximum power of driver features but you have to spoil interchangability.

2-1 Importing Type Libraries

What you should do first after creating a new project is, generate an interop assembly from the type library of the IVI-COM instrument driver you want to use, and then reference to it. Choose **Project | Add References** menu to bring up the **Add References** dialogue, then choose the **COM** tab.

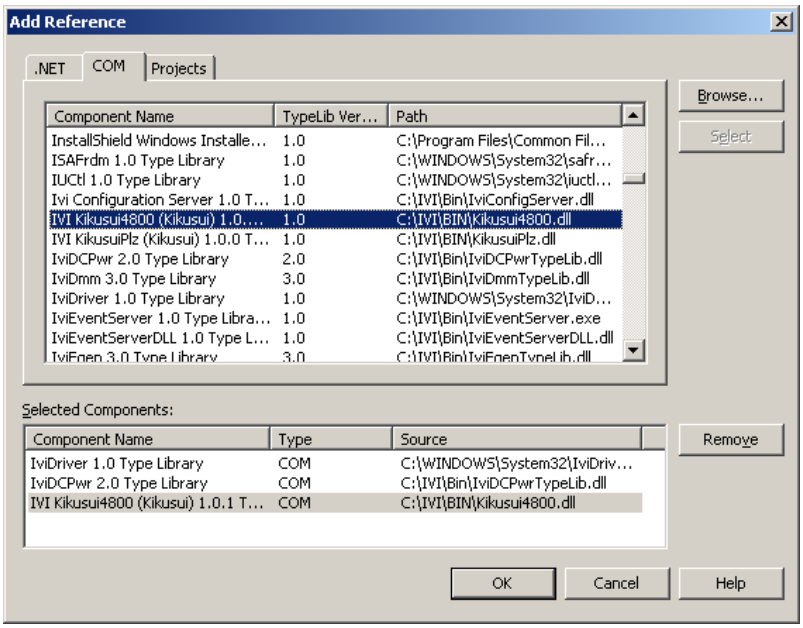

Figure 2-1 Add Reference dialogue

Since this example assumes that you use Kikusui4800 IVI-COM driver, choose **Kikusui4800 (Kikusui) 1.0 Type Library**. Furthermore choose also **IviDriver 1.0 Type Library** and **IviDCPwr 2.0 Type Library**.

Because **IviDriver** is a common item for every IVI-COM instrument driver, you need choose it regardless what instrument driver you use. **IviDCPwr** is necessary because the Kikusui4800 driver belongs to the IviDCPwr class. For example, if the instrument driver you actually use belongs to the IviDmm class, you need select the IviDmm Type Library. After selecting them with the **Select** button, click the **OK** button.

2-2 Object Browser

By adding references to the assemblies, you can confirm available syntaxes through the Object Browser of the Visual Studio.NET integrated development environment. To launch the Object Browser, choose **View | Object Browser** menu.

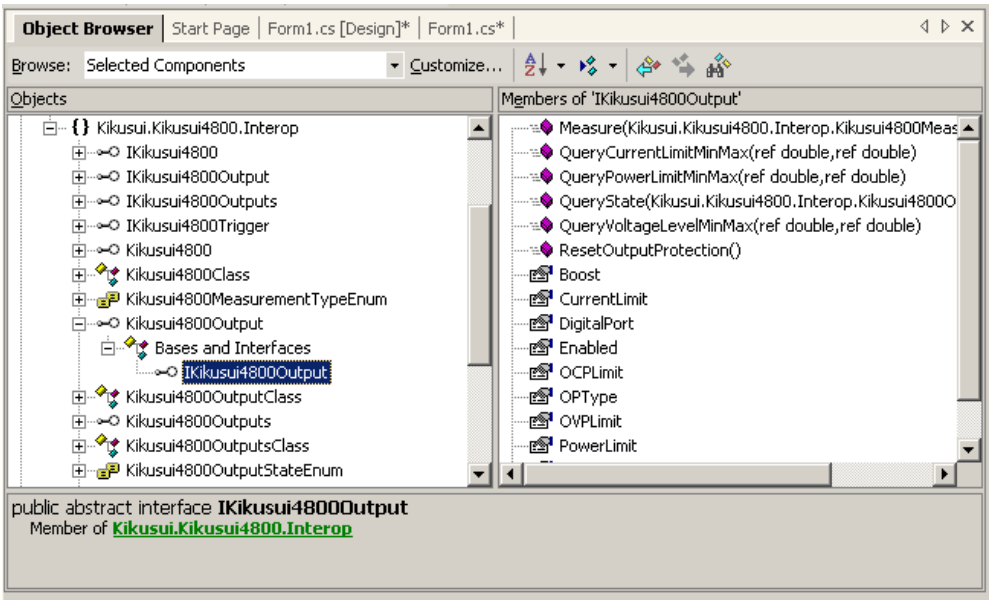

Figure 2-2 Object Browser

2-3 Creating Object and Initialising Session

First, doubleclick on the design-time form with the mouse. Then the `Form1_Load` event handler having only a skeleton code will be shown. Then write the following `using` directive, which allows you to omit the namespace references.

```
using Kikusui.Kikusui4800.Interop;
```

Declare `m_dcpwr` as a form's data member variable as `Kikusui4800Class` type. Do not forget to write the `new` operator since you also create an instrument driver object here.

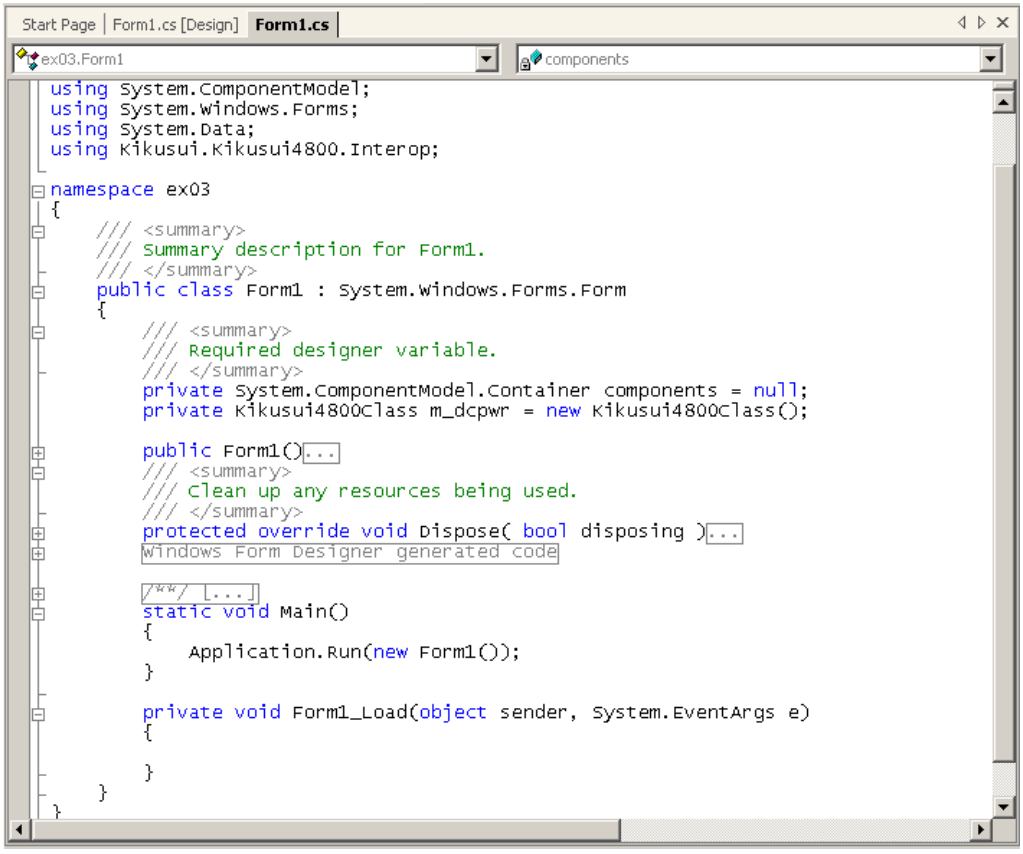

Figure 2-3 Form1_Load handler

Just creating the object does not perform any instrument I/Os. To initiate I/Os with the instrument, you use the `Initialize` method. As you type `m_dcpwr.Ini` in the `Form1_Load` handler, the IntelliSense feature of C#.NET will show the method/property list for `Kikusui4800Class` type. Since you have typed until Ini here, the `Initialize` method that is the closest name candidate will be highlighted.

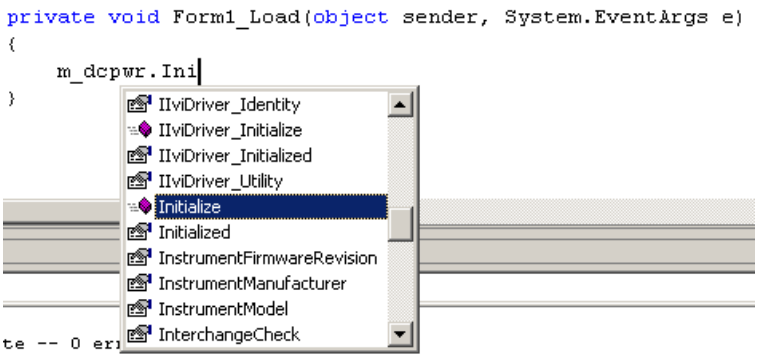

Figure 2-4 IntelliSense (Method/Property List)

By pressing the Tab key then typing a left parenthesis "(", IntelliSense will show all the parameters for the method.

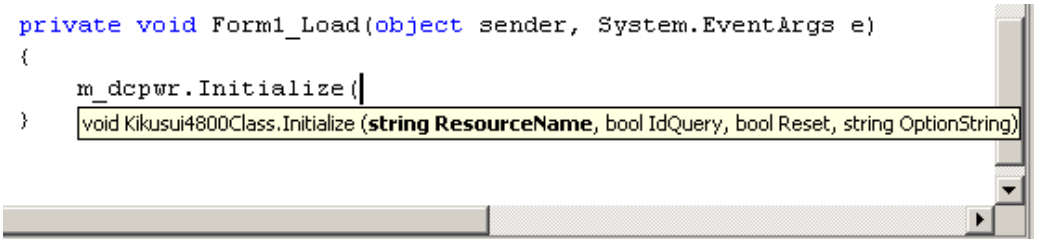

Figure 2-5 IntelliSense (Parameter List)

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 Parameters for 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. If the instrument has the address 3 on the GPIB board #0, for example, it can be `GPIB0::3::INSTR`.
`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
```

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.

Now try to write `Initialize` call. The `OptionString` parameter is optional and you can specify `null` to it. (`OptionString` is an optional parameter in Visual Basic languages, but can't be skipped in the C# language.)

```
uing Kikusui.Kikusui4800.Interop;

...(snip)...

public class Form1 : System.Windows.Forms.Form
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;
    private Kikusui4800Class m_dcpwr = new Kikusui4800Class();

...(snip)...

    private Form1_Load( object sender, System.EventArgs e)
    {
       m_dcpwr.Initialize("GPIB0::3::INSTR", true, true, null);
    }
}
```

2-4 Closing Session

To close the instrument driver session, use the Close method. Since this example wrote the Initialize method call in the `Form1_Load` handler, it is better to write the `Close` method call in the `Dispose` handler that is overridden from the `Form1` class. To override the `Dispose` handler, select Form1 at the upper-left combobox, and then select "Dispose(bool disposing)" at the right-side combobox.

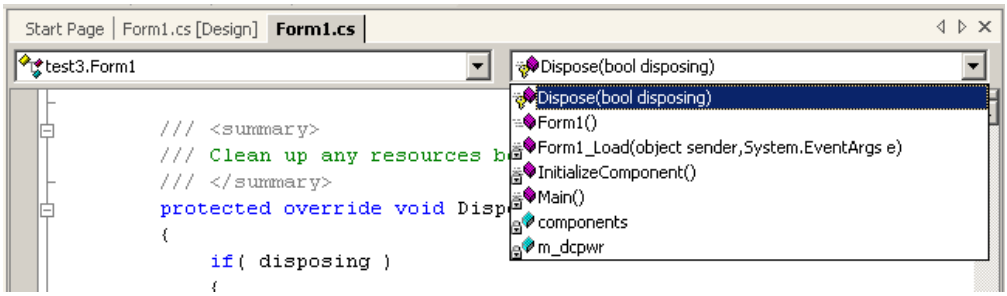

Figure 2-6 Overriding Dispose handler

Now the codes for the overridden `Dispose` method will appear. Add some codes so that they become as like below.

```
protected override void Dispose(bool disposing)
{
    if(disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
        m_dcpwr.Close();
        m_dcpwr = null;
    }
    base.Dispose( disposing );
```

```
}
```

2-5 Execution

You can execute the previous codes for the time being. Since the `Initialize` method is invoked in the `Form1_Load` handler, communications with the instrument immediately start as you launch the program. If the instrument is actually connected and the `Initialize` method call has succeeded, the form screen appears. If a communication problem has occurred or the VISA library is not configured properly, a COM exception (System.Runtime.InteropServices.COMException) will be generated.

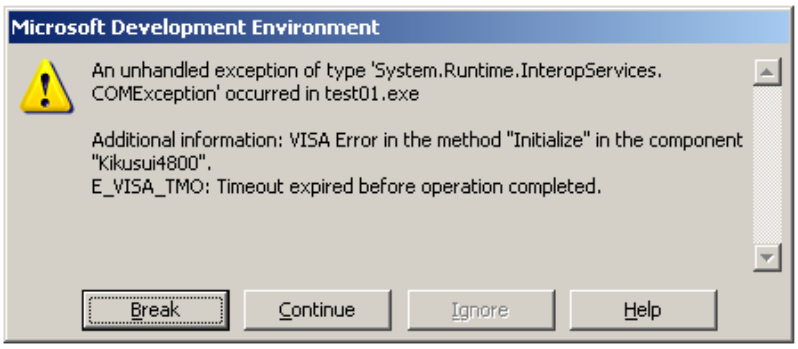

Figure 2-7 COM exception

2-6 Repeated Capabilities

In case of Kikusui4800 IVI-COM driver, output settings for the DC power supplies are performed through the Output interfaces as the same concept defined by the IviDCPwr class. In case of specific interfaces provided by the Kikusui4800 driver, they are the `IKikusui4800Output` and `IKikusui4800Outputs` interfaces. An instrument driver that is compliant with the IviDCPwr class is designed assuming that the instrument is a multi-track power supply equipping multiple output channels.

These COM interfaces have the same name with an exception of differences between singular and plural forms. An interface having this kind of plural name is generally called "repeated capabilities" in the IVI specifications. Repeated capabilities are something like a container that is defined for handling equivalent or similar multiple objects, and a COM interface having a plural name such as `IKikusui4800Outputs` normally has the `Count`, `Name`, and `Item` properties (all are read-only). Plus, a singular object can be referenced through the `Item` property.

First look at the following example, which controls an output channel identified by the name "`N5!C1`" on the power supply instrument (actually a Kikusui PIA4800 series DC Power Supply Controller) hosted by the Kikusui4800 IVI-COM driver. This example writes the codes in the event handler assuming you put a command button (button1) on the form.

```
private void button1_Click( object sender, System.EventArgs e)
{
    IKikusui4800Output output;
    output = m_dcpwr.Outputs.get_Item("N5!C1");
    output.VoltageLevel = 10.5;
    output.CurrentLimit = 1.2;
    output.Enabled = true;
}
```

Once the `IKikusui4800Output` 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 `IKikusui4800Output` interface. This example here acquires the `IKikusui4800Outputs` interface though the Output property of the `IKikusui4800` interface, then acquires `IKikusui4800Output` interface by using the `Item` property.

```
IKikusui4800Output output;
output = m_dcpwr.Outputs.get_Item("N5!C1");
```

The codes can also be written as like below.

```
IKikusui4800Outputs outputs = m_dcpwr.Outputs;
IKikusui4800Output output = outputs.get_Item("N5!C1");
```

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, Kikusui4800 IVI-COM driver uses an expression like "`N1!C1`" specifying NODE and CH. However other drivers, even if being IviDCPwr class-compliant, may have different names. One instrument driver, for example, may use an expression like "`Track1`". 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.

```
IKikusui4800Outputs outputs = m_dcpwr.Outputs;
int cnt = outputs.Count;
int ndx;
for( ndx=1; ndx<=cnt; ndx++)
{
    string strName;
    strName = outputs.Name(ndx);
    System.Diagnostics.Debug.WriteLine(strName);
}
```

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` statement. Mind that the index numbers for the `Name` parameter is one-based, not zero-based.

3- Example Using Class Interfaces

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 interchangability available between different instrument classes.

3-1 Virtual Instrument

What you have to do before creating an application that utilises interchangability features is create a virtual instrument. To realise interchangability 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 Kikusui4800 type) and should not write a specific VISA resource name such as "GPIB0::3::INSTR". Writing them directly in the application spoils interchangability.

Instead, the IVI-COM specifications define methods to realise interchangability 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 /Program Files/IVI/Data/IviConfigurationStore.XML file and is accessed through the IVI Configuration Server DLL. This DLL is mainly used by IVI-COM instrument drivers and some configuration tools provided by instrument driver vendors, not by end-user applications. KIKUSUI provides a configuration tool called **Kikusui IVI Config Utility** that allows you to configure virtual instrument settings.

> Notes:
>
> As for how to configure virtual instruments by using Kikusui IVI Config Utility, refer to "Programming Guide, (IVI Config Utility Edition)."

This guidebook assumes that a virtual instrument having the logical name "`MySupply`" is already created, using Kikusui4800 driver, and using a VISA resource "GPIB0::3::INSTR".

3-2 Importing Type Libraries

What you should do first after creating a new project is, generate an interop assembly from the type library of the IVI-COM instrument driver you want to use, and then reference to it. Choose **Project | Add References** menu to bring up the **Add References** dialogue, and then choose the **COM** tab.

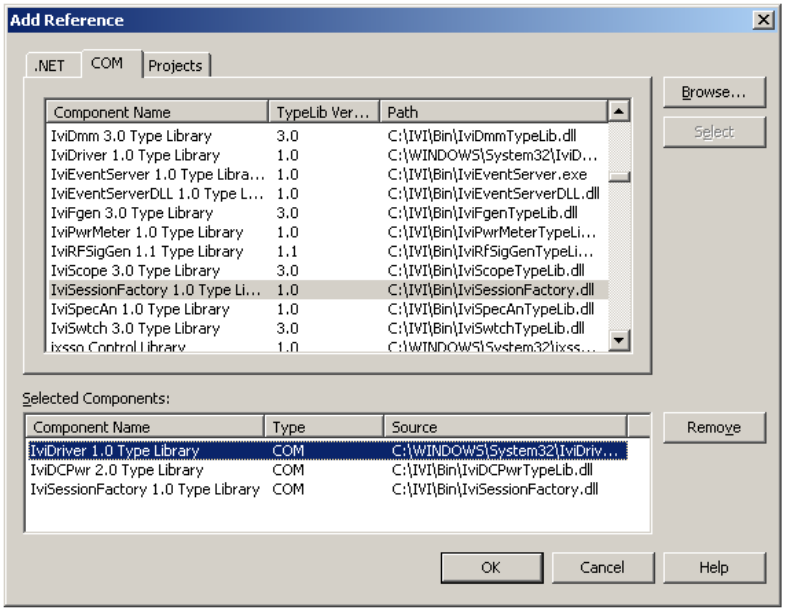

Figure 3-1 Add Reference dialogue

Since this example assumes that you use the IviDCPwr class interface, select **IviDCPwr 2.0 TypeLibrary**. Furthermore, make sure select **IviDriver 1.0 Type Library** and **IviSessionFactory 1.0 TypeLibrary** regardless instrument classes you use. After selecting one or more items with the Select button, then click the **OK** button.

Importing type libraries are now completed. Your application will be able to use arbitrary instrument drivers through the IviDCPwr class interfaces.

3-3 Object Browser

By adding references to the assemblies, you can confirm available syntaxes through the Object Browser of the Visual Studio.NET integrated development environment. To launch the Object Browser, choose **View | Object Browser** menu.

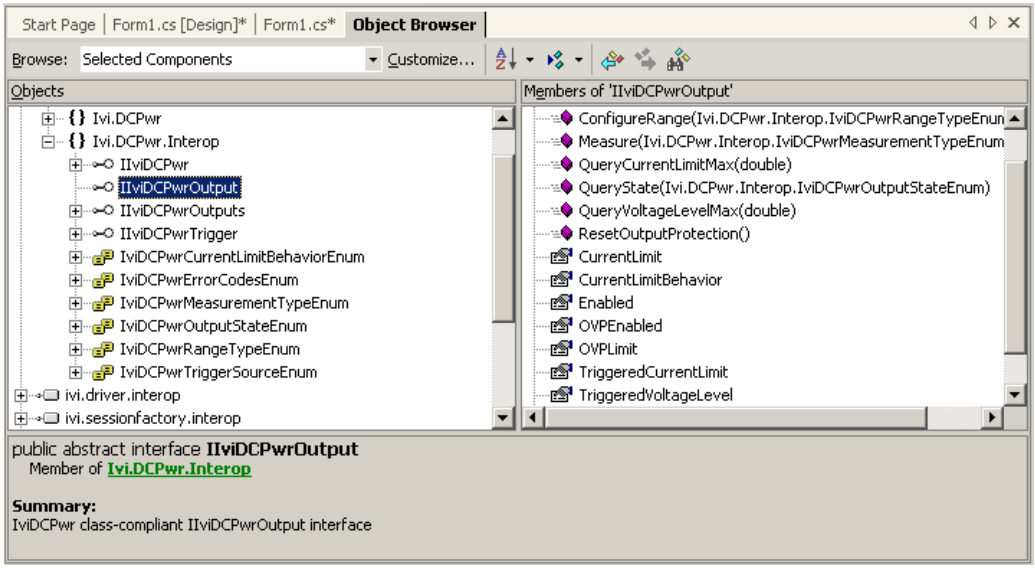

Figure 3-2 Object Browser

3-4 Creating Object and Initialising Session

First, doubleclick on the design-time form with the mouse. Then the `Form1_Load` event handler having only a skeleton code will be shown. Then write the following `using` directive, which allows you to omit the namespace references.

```
using Ivi.SessionFactory.Interop;
using Ivi.Driver.Interop;
using Ivi.DCPwr.Interop;
```

Declare `m_dcpwr` as a form's data member variable as `IIviDCPwr` type. A type that begins with capital "I" such as `IIviDCPwr` is in general a COM interface type, therefore you cannot create an object with the `New` operator.

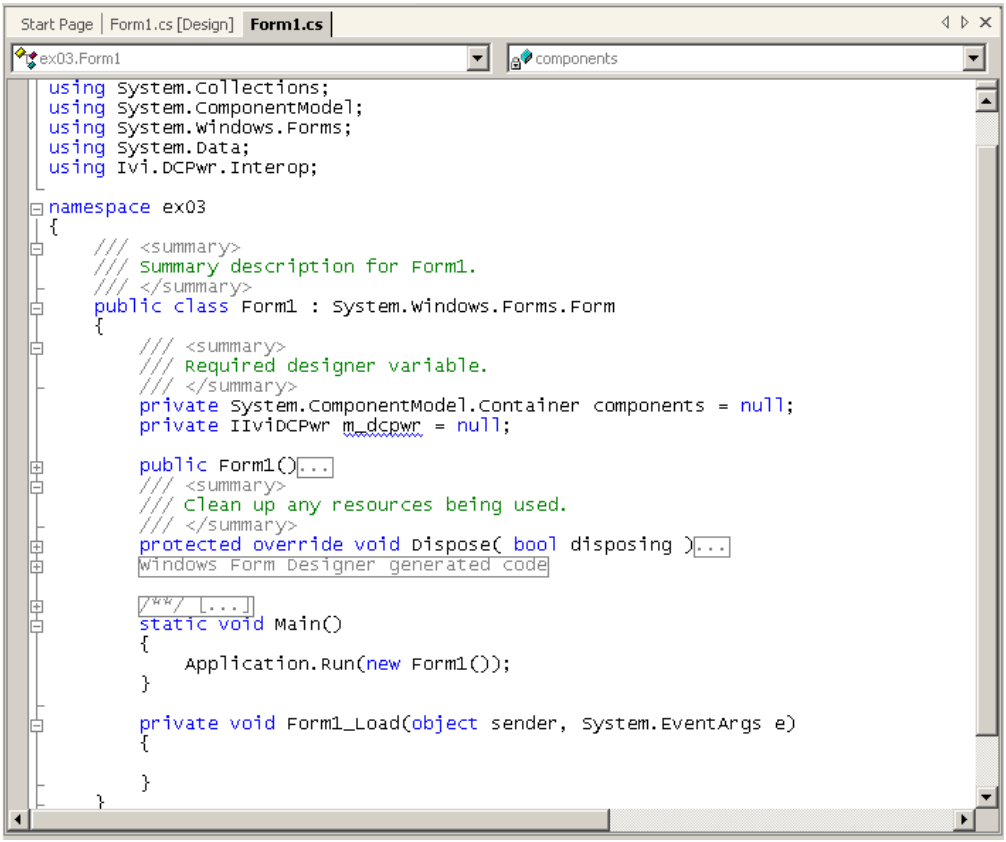

Figure 3-3 Form1_Load handler

Next you must create an instrument driver object. To create an instrument driver object, use IVI Session Factory. The IVI Session Factory is a DLL server that comes with the IVI Shared Components. It extracts configuration information of the virtual instrument specified by the given logical name, loads the appropriate instrument driver software, then creates an instrument driver object.

```
private Form1_Load( object sender, System.EventArgs e)
{
    IviSessionFactoryClass sf = new IviSessionFactoryClass();
    m_dcpwr = (IIviDCPwr)sf.CreateDriver("MySupply");

}
```

The example here creates an `IviSessionFactory` object and then creates an instrument object with the `CreateDriver` method. The parameter "`MySupply`" must already be configured as a valid logical name. The instrument driver DLL to be loaded will be what specified by the virtual instrument `MySupply`.

Mind that an explicit typecast is required when assigning the return value from the `CreateDriver` method to the variable m_dcpwr. The COM interface that is returned from the `CreateDriver` method is strictly declared as `IUnknown` type, therefore the declaration in the interop assembly that corresponds to this interface is `object` type. In this case, it is necessary to invoke `QueryInterface` to acquire the `IIviDCPwr` interface. In case of C#.NET language, an explicit typecast makes the `QueryInterface` be invoked behind the scenes.

After creating the driver object, invoke the `Initialize` method. Although parameters of the `Initialize` method are exactly the same as the case of using specific interfaces, you should specify the logical name for the `ResourceName` parameter instead of specifying a VISA resource.

Again let 's talk about parameters of the `Initialize` method. Every IVI-COM instrument driver has an `Initialize` method that is defined by the IVI specifications. This method has the following parameters.

Table 3-1 Parameters for 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. If the instrument has the address 3 on the GPIB board #0, for example, it can be `GPIB0::3::INSTR`. If specifying a logical name, the VISA resource that is described in the logical name's Hardware Asset configuration will be indirectly specified.
`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.

In general, applications that are aware of interchangability use a logical name for the `ResourceName` parameter rather than a VISA resource. Actually it is possible to specify a VISA resource, however, doing so slightly spoils abstraction of virtual instrument.

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
```

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.

Now try to write `Initialize` call. The `OptionString` parameter is optional and you can specify `null` to it. (`OptionString` is an optional parameter in Visual Basic languages, but can't be skipped in the C# language.)

```
m_dcpwr.Initialize( "MySupply", true, true, null);
```

3-5 Closing Session

To close the instrument driver session, use the Close method. Since this example wrote the Initialize method call in the `Form1_Load` handler, it is better to write the `Close` method call in the `Dispose` handler that is overridden from the `Form1` class. To override the `Dispose` handler, select Form1 at the upper-left combobox, and then select "Dispose(bool disposing)" at the right-side combobox.

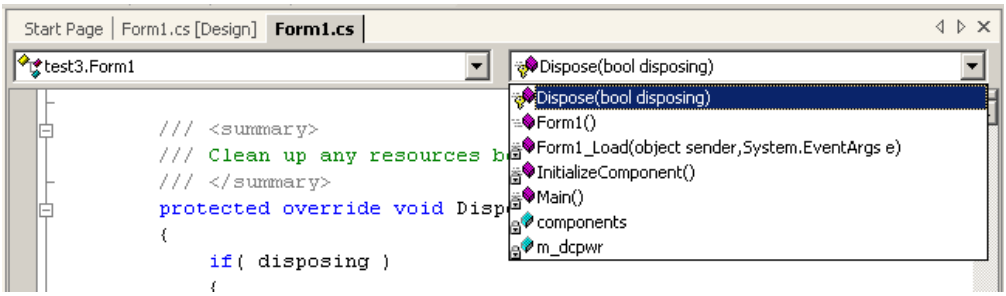

Figure 3-4 Overriding Dispose handler

Now the codes for the overridden `Dispose` method will appear. Add some codes so that they become as like below.

```
protected override void Dispose(bool disposing)
{
    if(disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
        m_dcpwr.Close();
        m_dcpwr = null;
    }
    base.Dispose( disposing );
}
```

3-6 Execution

You can execute the previous codes for the time being. Since the `Initialize` method is invoked in the `Form1_Load` handler, communications with the instrument immediately start as you launch the program. If the instrument is actually connected and the `Initialize` method call has succeeded, the form screen appears. If a communication problem has occurred or the VISA library is not configured properly, a COM exception (System.Runtime.InteropServices.COMException) will be generated.

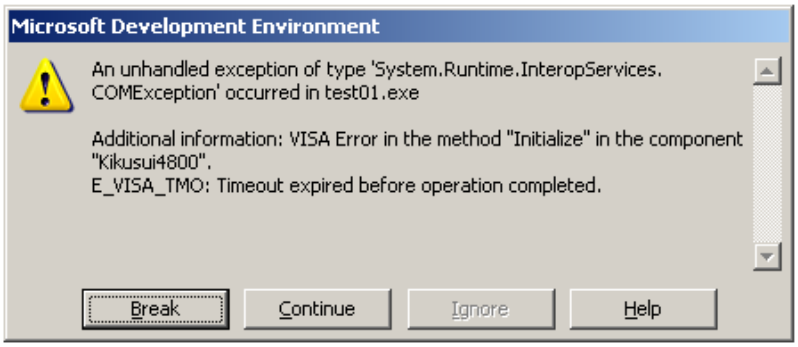

Figure 3-5 COM exception

3-7 Repeated Capabilities

In case of IviDCPwr class interfaces, output settings for the DC power supplies are performed through the Output interfaces. The interfaces used here are `IIviDCPwrOutput` and `IIviDCPwrOutputs`. An instrument driver that is compliant with the IviDCPwr class is designed assuming that the instrument is a multi-track power supply equipping multiple output channels.

These COM interfaces have the same name with an exception of differences between singular and plural forms. An interface having this kind of plural name is generally called "repeated capabilities" in the IVI specifications. Repeated capabilities are something like a container that is defined for handling equivalent or similar multiple objects, and a COM interface having a plural name such as `IIviDCPwrOutputs` normally has the `Count`, `Name`, and `Item` properties (all are read-only). Plus, a singular object can be referenced through the `Item` property.

First look at the following example, which controls an output channel identified by the name "`Track_A`" registered as in the virtual instrument's virtual name. This example writes the codes in the event handler assuming you put a command button (button1) on the form.

```
private void button1_Click( object sender, System.EventArgs e)
{
    IKikusui4800Output output;
    output = m_dcpwr.Outputs.get_Item("Track_A");
    output.VoltageLevel = 10.5;
    output.CurrentLimit = 1.2;
    output.Enabled = true;
}
```

Once the `IIviDCPwrOutput` 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 `IIviDCPwrOutput` interface. This example here acquires the `IIviDCPwrOutputs` interface though the Output property of the `IIviDCPwr` interface, then acquires `IIviDCPwrOutput` interface by using the `Item` property.

```
    IIviDCPwrOutput output;
    output = m_dcpwr.Outputs.get_Item("Track_A");
```

The codes can also be written as like below.

```
    IIviDCPwrOutputs outputs = m_dcpwr.Outputs;
    IIviDCPwrOutput output = outputs.get_Item("Track_A");
```

Now mind the parameter passed to the `Item` property. This parameter specifies the name of the single Output object to be referenced. In the example that used specific interfaces, we specified a name that was very dependent upon each driver (physical name), however we use a different approach. Because a physical name that depends on a particular instrument driver should not be used, we specify a virtual name instead.

3-8 Swapping Instruments

In the previous examples, we used the Kikusui4800 instrument driver for the virtual instrument configurations. Now what will happen if you replace the instrument with the one that is hosted by the AgilentE36xx driver? In this case you do not have to recompile/relink your application, but you need change the virtual instrument configurations.

The configurations you have to change are basically Software Module selection on the Driver Session tab, and virtual name mappings on the Virtual Names tab (because physical names of the map target are changed). Replacing instruments may not allow using the same I/O interface (such as changing from a GPIB-only instrument to an RS232-only instrument), so you may have to change IO Resource Descriptor on the Hardware Asset tab as need.

> Notes:
>
> As for how to configure virtual instruments by using Kikusui IVI Config Utility, refer to "Programming Guide, (IVI Config Utility Edition)."

4- 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#.NET, a COM exception can be handled by using `try`, `catch`, and `finally` blocks.

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

```
private void button1_Click( object sender, System.EventArgs e)
{
    try
    {
        IKikusui4800Output output;
        output = m_dcpwr.Outputs.Item("N5!C1");
        output.VoltageLevel = 10.5;
        output.CurrentLimit = 1.2;
        output.Enabled = true;
    }
    catch(System.Runtime.InteropServices.COMException ex)
    {
        MessageBox.Show(
              ex.Message, "error 0x" + Convert.ToString(ex.ErrorCode, 16));
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message, "error");
    }

}
```

In this example, errors are handled by using `try`, `catch`, and `finally` blocks. 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. Above example just displays a simple message box when an exception has occurred.

Detail about the error (COM exception) can be acquired through the parameter of `catch` block. This example sets the error code (hexadecimal) obtained from `ErrorCode` property to the message box caption, and sets the description string obtained from the `Message` property to the main body text.

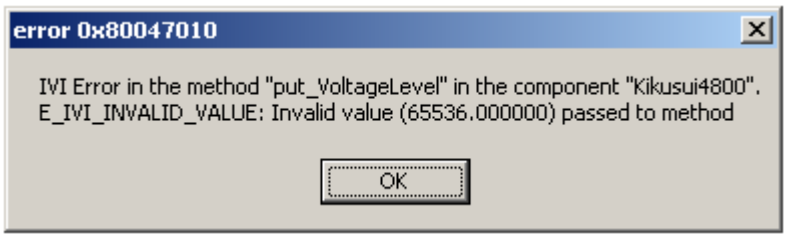

Figure 4-1 Message box by error handling