Cslow – a C# library for the IOWarrior

Cslow is a .NET library for the IOWarrior produced by CodeMercenaries Hard- und Software GmbH. It is entirely written in C# (with calls into unmanaged code of course). The library was developed with the Visual C# Express 2005 Edition.

Applications to be developed with the library must meet the following restrictions :

- The Operating System must be Windows XP or newer. There is no and never will be support for Windows98, Win NT etc.
- The .Net Framework 2.0 must be installed
- Your application has to be a Windows Forms application. There is no and never will be support for pure console applications.

As a developer you should be familiar with (or willing to read up on) the following programming topics

- Threads
- Implementing Interfaces
- Delegates

I also expect that you read the datasheet for the IOWarrior.

Here is a short rundown of the features implemented in the library

- Dynamic plug/unplug notification for IOWarriors through callback-functions
- Dynamic data notification through callback-functions
- Optional filtering for specific IOWarriors
- Optional filtering of reports
- Open source, non-restrictive license

Where to get help

After you have read this document thoroughly and didn't find an answer to your question, bug reports, comments and feature requests should be addressed to <iow@wayoda.org>

Licensing Terms for the Cslow-library

1 Overview

Everything in the library is callback-driven. There are for instance no functions for retrieving a list of currently connected IOWarriors, or functions for reading reports from an IOWarrior. All you have to do is to create a class that implements a specific Interface and then wait until the library calls you back whenever something happens.

1.1 The `org.wayoda.csiow` namespace

All the classes and interfaces of the library are defined in the namespace `org.wayoda.csiow` . An application gets access to the library through the

```
using org.wayoda.csiow;
```

statement. (I guess you know that you have to add a reference to the file `CsIow.dll` located `bin`-directory of the project.)

Please, do **not** develop your application in namespace `org.wayoda.csiow`. The Intellisense feature of Visual Studio would reveal all internal classes and methods of the CsIow Library to your application. Calling a method that is declared internal inside the `org.wayoda.csiow` namespace will most likely mess up your application. There are no hidden features or anything else your application could benefit from.

1.2 Overview of classes/interfaces

The library exposes 6 classes and 2 interfaces to your application-code. Here's a short description for each of them (more detailed information and code examples later on in this document):

`class IowManager`

This class handles everything about device-detection, i.e. plug/unplug of IOWarriors. You will have to register a class that implements interface `IowDeviceChangeListener` with the manager, and it will notify you whenever a new IOWarrior is plugged in, or an existing device is unplugged.

`interface IowDeviceChangeListener`

You have to implement this interface in one of your own application-classes. After you have registered your class with the `IowManager`, you will be notified for device-changes through a callback into the methods defined by the interface.

`class IowDeviceFilter`

This class manages a list of filters based on the product-id and/or serial-number of the IOWarriors. Setting up the filter allows you to get callbacks only for specific IOWarriors, ignoring all others that do not match the criteria in the filter.

`class IOWarrior`

An instance of this class represents an IOWarrior found on your system. As long as it is not unplugged, you can write new reports to it. Reports coming in from the device will be sent to application-classes implementing the interface `IowReportListener` after they have been registered with the IOWarrior. You can register more than one listener with an IOWarrior and there are ways to filter the types of reports sent to you. You can for instance add one listener that receives all updates on the status of the IO-Pins and

another one listening only for changes on a Switch-Matrix. There are also several functions for retrieving static device capabilities, like the product-id, serial-number, revision etc.

`interface IowReportListener`

You have to implement this interface in one of your own application-classes. After you have registered your class with an IOWarrior, you will be notified for new reports sent by the device and also when the IOWarrior was unplugged.

`class IowReport`

This class that encapsulates the data that is exchanged between your application and the IOWarriors. The datasheet of the IOWarrior explains exactly the how many bytes make up a report to be written to an IOWarrior or read from it. This number varies along the different products and modes of the IOWarriors. In the CsIow-Lib the IOWarrior class will take care of this problem. All you have to do is create a new `IowReport`, fill in the report-id and data, and send it off to the device.

`class IowError`

Things go wrong. If that happens we return an instance of this class which provides some information why the operation failed.

`class IowSpecialMode`

From the datasheet of the IOWarrior you already know about the different SpecialMode-commands it supports. You also read that not very IOWarrior (or even a different revision of the same product) supports the same SpecialModes. The class implements a few typed constants that can be used for checking which SpecialModes are supported by a specific device, which type of SpecialMode an `IowReport` belongs to, or for filtering reports in an `IowReportListener`.

2 Handling plug/unplug events for IOWarriors

The first thing your application has to do is setting up a listener for device changes that gets called whenever a new IOWarrior gets plugged in or is removed. The most basic one I can think of, looks like this

```
public class MyDeviceListner : IowDeviceChangeListener {
       public MyDeviceListener() {
       }

       public void deviceAdd(IOWarrior iow) {
              Console.WriteLine("Add : "+iow);
       }

       public void deviceRemoved(IOWarrior iow) {
              Console.WriteLine("Removed : "+iow);
       }
}
```

All it does is print out a message when an IOWarrior is plugged in or removed in the two methods which implement the interface `IowDeviceChangeListener`. Your application will only need a single instance of this class because only one Listener can be registered with the library in the static method `IowManager.open()`.

Here's how to register our listener with the library

```
IowError err;
MyDeviceListener devListener=new MyDeviceListener();

err=IowManager.open(devListener);
if(err!=IowError.OK) {
       Console.WriteLine("Error in open : "+err.getMessage());
....
```

That's it. Create the listener and hand it over to the `open()` call. I put in a bit of error handling in case things go wrong. An instance of class IowError is returned from the `open()` call. If everything went fine the constant `IowError.OK` is returned. Otherwise a new Instance of `IowError` was created inside the library and you can request some information about the error that occurred.

If the method returns an `error!=IowError.OK` you're already done, because it signals something went seriously wrong and your application will never be able to access any IOWarriors.

Instead of returning an error the `IowManager.open()` will throw Exceptions for all of the following situations.

The OperatingSystem is not Windows XP or newer

A `System.PlatformNotSupportedException` will be thrown. CsIow uses some OS libraries not available on older platforms. So you cannot create an application that runs on one of these platforms.

You called `IowManager.open()` twice without closing it in between

A `System.InvalidOperationException` is thrown. Each successful `open()` call must be followed by a `close()` call later (when you are done with the library).

You called `IowManager.open()` with a null-argument for the listener

> Obviously the library needs someone to talk to on device changes, so this would be a serious programming error, and a `System.ArgumentNullException` is therefore thrown.

But lets assume the `open()` call was successful. This is what happens inside the library:

All IOWarriors currently connected will be opened and for each device your listeners `deviceAdd(IOWarrior iow)` method will be called.

The library will create an internal handler that will call your `deviceAdd()` or `deviceRemoved()` methods whenever an IOWarrior is plugged in or removed later. This handler will be active until your application closes the library. So your listener must be prepared to handle new devices every time between the `IowManager.open()` call until the `IowManager.close()` call.

If there are no IOWarriors connected by the time `IowManger.open()` is called, this is not regarded as an error. The method will still return `IowError.OK`. If your application relies on the presence of a specific device, it is your task to tell the user to plug it in.

2.1 Filtering for specific IOWarriors

With the code shown so far your application will get an `deviceAdd()` event for every IOWarrior plugged into your computer. If your application relies on the presence of a specific IOWarrior or maybe supports only the IOWarrior24 you can setup a filter for devices before you call `IowManager.open()`. An example:

You have 2 IOWarriors on your desktop

a) An IOWarrior24 with a serial-number of 0x123 to which an LCD-Display is connected

b) An IOWarrior56 with a serial-number of 0x456 that does some AD-conversion

Now you want to write an application for device a that prints the system status to the LCD. As your are not interested in the IOWarrior56 you can setup a device-filter for the library

```
IowDeviceFilter filter=new IowDeviceFilter;
filter.add(IOWarrior.PID_IOW24,0x123);

MyDeviceListner devListener=new MyDeviceListner();

IowError err=IowManager.open(devListener,filter);
```

Here we created a new filter for the library which will force it to ignore all IOWarriors but the IOWarrior24 with the serial-number 0x123. You will never get a callback to your listener for the IOWarrior56 that is also plugged into your computer. You can add more than just one IOWarrior to the filter. If you happen to have 3 IOWarriors and want to use 2 of them just call `filter.add()` again with the product-id and serial-number for the second device.

(By the way : as you see in the code, all product-id's for IOWarriors are available as static constants in the class `IOWarrior`.)

If you want to setup you filter-conditions to accept all IOWarriors of a specific product-type there is the `addProduct(int)` method in class `IowDeviceFilter`

```
IowDeviceFilter filter=new IowDeviceFilter;
filter.addProduct(IOWarrior.PID_IOW56);
```

That would make the library accept only IOWarrior56 devices.

If `IowManager.open(devListner,filter)` is called with an empty filter (no device/product added to it) or null, the library does not regard this as an error. It will simply accepts and reports every IOWarrior found.

2.2 Closing the library

There is only one thing to say about this : *You have to do it!*

You cannot simply exit your application and hope that the library takes care of itself. There might be threads running inside the library which will not terminate until `IowManager.close()` is called. Consequently your application might not be able to shutdown. There are also resources which are only freed inside the `close()` call. In `close()` your device listeners `deviceRemoved()` method will be called for each IOWarrior. So whatever your code did with the IOWarrior, it will know that the device is not available any more.

2.3 Inside the device event handler

Sorry, here comes the tough part!

Each event for device plug/unplug is delivered in a new Thread. The library can not and does not care what you do inside this Thread. It also does not care wether the Thread that was started for the device will ever terminate. The library will not wait for it to die (fire-and-forget)!

From your applications point of view this is what happens

- The library detects that a new IOWarrior was plugged in
- It creates a new instance of class IOWarrior
- It starts up a new Thread that calls the `deviceAdd(IOWarrior iow)` method of your listener for device changes

Now it's up to you to decide what you do inside your handlers callback. The simple example from above is just perfect. It prints a message to the console an returns. The Thread that was started from the library will terminate right after `deviceAdd()` returned.

But it is very easy to turn the code from above into a very bad example:

```
public class MyDeviceListner : IowDeviceChangeListener {
       public MyDeviceListener() {
       }

       public void deviceAdd(IOWarrior iow) {
              Console.WriteLine("Add : "+iow);
              for(int i=0;i<3600;i++) {
                     Thread.Sleep(1000);
                     Console.WriteLine("I'm still alive!");
              }
       }

       public void deviceRemoved(IOWarrior iow) {
              Console.WriteLine("Removed : "+iow);
       }
}
```

Now the `deviceAdd()` message is printed like before but then the Thread will go on for an hour printing "I'm still alive!" every second. The library does not stop the Thread from doing so. If your applications code does not stop the Thread by calling `Abort()`, your application will run for one hour no matter how often the user clicked the close button on your window. So there are obviously things you cannot put into the event handler. A rough guideline for successful programming goes like this:

- Terminate quickly. Lengthy operations should be run in a new Thread which you control inside your applications code.
- Save the IOWarrior object for later use inside one of your own datastructures.
- If you want to call methods for a component that was derived from `Windows.Forms.Control` you have to use the `Invoke()` method of your component.

The first point should be obvious from the example above.

The second point is based on the fact that you will never see this specific IOWarrior again, once you returned from `deviceAdd()`. The `IowManager` class does not implement methods that lets your application retrieve a list of devices later on. IOWarriors are reported once, that's it.

The explanation for the third point can be found in every serious book about programming .NET applications. Every UserControl in an application can only be accessed from the Thread in which it was created. Since the event handler runs in its own Thread, you will get an exception if you try to call a method of an UserControl. In the .NET environment this problem is resolved by calling the `Invoke()` method of the Control.

A simple example should clarify this. We want add every IOWarrior that is reported in the callback to a `ListBox-Control` located on the applications main form. If the IOWarrior is unplugged later, it shall be removed from the list in the `deviceRemoved()` handler.

Since we must use Invoke we have to define a few Delegates for the `ListBox`-methods we need to call:

```
//used to access ListBox.Items.Add
delegate int AddIowToListBox(IOWarrior iow);
//used to access ListBox.Items.RemoveAt
delegate void RemoveIowFromListBox(int i);
```

Here is the code for the event handlers to make this work

```
// a ListBox “deviceListBox” has already been added to our main form

public void deviceAdd(IOWarrior iow) {
	//Create a delegate for the method we want to call
	AddIowToListBox del=new AddIowToListBox(deviceListBox.Items.Add);
	//call the listbox function with our IOWarrior
	deviceListBox.Invoke(del, new Object [] {iow});
}

public void deviceRemoved(IOWarrior iow) {
	//create the delegate
	RemoveIowFromListBox del=newRemoveIowFromListBox(deviceListBox.Items.RemoveAt);
	//iterate through the items in the listbox
	for(int i=0;i<deviceListBox.Items.Count;i++) {
		// The test for the device can be done without
		// Invoke, because the visual appearance of the
		// control is unchanged
		if(iow.Equals(deviceListBox.Items[i])) {
			//Here we need to use Invoke for device removal
			deviceListBox.Invoke(ris, new object[] { i });
		}
	}
}
```

This it all it takes to fulfill the 3 requirements for the event handlers.

- Both methods return right after the IOWarrior was added to the `ListBox` (or removed from it).
- If we want to do something with the IOWarrior later, we can request the device from the `ListBox` where it is stored.
- Using `Invoke()` puts the calls to the `ListBox` on the main thread of the application.

The example works fine since adding the IOWarrior to the `ListBox`-Control doesn't take very long. But our handler still waits for the `deviceListBox.Invoke()` method to return.
If we had to do something even more time-consuming in the handler, we could have used `BeginInvoke()` instead, which puts the call to the `ListBox`-methods into a new thread that executes on the main thread of the application.

3 Operating an IOWarrior

When your device event handler is called with a new `IOWarrior`, it provides you with a fully functional object. If you look at the source code of the library you will notice that `class IOWarrior` does not expose a public constructor to your applications code. IOWarriors are created inside the library code and then reported to your application when they are ready for use. There is also no method for closing an IOWarrior. Your application should simply forget about the device when you're done with it.

3.1 What kind of device did I get ?

This will obviously the first question that has to be coped with in your application. The `IOWarrior` class provides the usual methods for getting the product-id, serial-number and revision-number of the IOWarrior. But since the `IOWarrior` class provides you only with a unified object no matter what kind of product from the IOWarrior-family is actually represented, there are a few more methods:

int `getReportSizeIO`()
int `getReportSizeSM`()
: These two methods tell you how many bytes make up an report to/from the IO-Pins and SpecialMode functions. Since the library uses only `IowReports` for data exchange with the device, your application needs to know how many bytes in an `IowReport` have to be processed on reads and writes.

int `getReportLatency`()
: Tells you the maximum data rate at which reports can be send by the device to your application.

`IowSpecialMode [] getSupportedSpecialModes()`
: This method returns an array of all the SpecialModes that are available on this device. An `IowSpecialMode`-object is just one of the typed constants defined in class `IowSpecialMode`. If you want to check if your device supports a SwitchMatrix you would have to retrieve the array and test wether the object `IowSpecialMode.SWITCH_MATRIX` is found in the array.

`bool isSupportedMode(IowSpecialMode mode)`
: Makes the previous test even more easy. Just call this function with one of the constants from class `IowSpecialMode` to find out if it is supported by your device.

`bool isConnected()`
: Simply tests if your device is still connected (i.e. not unplugged).

3.2 IowReport

Before we get to the read and write functions I have to introduce `class IowReport`. The basic structure for all data that gets written to or read from an IOWarrior is an array of bytes with the report-id in the first element and the actual data following. The size of a report varies along the different products. The library uses a unified datastructure for all devices. An `IowReport` provides you with enough memory for even the biggest report to be exchanged with an IOWarrior (that is 64 bytes for an IOWarrior56). Since the library is well aware how many of the bytes in an `IowReport` have to be send to the device, you don't have specify the length of a report.

The datasheet of the IOWarrior mentions two distinctive pipes to which reports get written. If you are familiar with the IowKit-Library from CodeMercs you know that you always have to specify one of these pipes on reads and writes. In the CsIow-Library I dropped the concept of pipes. Writing to the IO-Pins is treated as just another SpecialMode with the report-id set to 0x00. (Consequently there is also a typed constant `IowSpecialMode.IO`). So if you want your data to go to the IO-Pins set the report-id to 0x00 in the `IowReport`. For reports coming in from the device the same applies. If the report carries a report-id of 0x00 it came from the IO-Pins.

There are several constructors for an `IowReport` that initialize the data in it and also setters and getters for the data. I guess the most useful feature is the index-operator that can be used on the individual bytes in the report. Just keep in mind that the byte at index 0 in the report is the report-id.

```
//Create a report with every byte set to 0x00
IowReport rep=new IowReport();
rep[0]=0x0C; //set the report-id for RemoteControl anIow24
rep[1]=0x01; //and switch the specialmode on
```

3.3 Writing to the device

This is simply done by creating an `IowReport` with the data you want to send and then calling the `write(IowReport rep)` method of your device. If you want to write more than one report to the device with a single call, you can create an array of reports and send them off with the `write(IowReport [] reps)` call. Both calls block until the data is successfully written or an error is detected. The return value is an instance of `class IowError`. A successful write will always return the constant `IowError.OK`. Otherwise you will have to look at the type of error that was reported :

```
// iow is the device we want to write the data to

// lets set all IO-Pin to LOW
IowReport rep=new IowReport();
IowError err=iow.write(rep);
if(err=IowError.OK)
      Console.WriteLine("Success");
else if(err.getType() == IowError.Type.WRITE_FAILED)
      Console.WriteLine("Write failed because : "+err.getMessage());
else if(err.getType() == IowError.Type.WRITE_UNSUPPORTED_MODE)
      ....
else if(....
```

The `getType()` method of class `IowError` will return one of the following constants:

`IowError.Type.NO_ERROR`

The report was successfully written to the device.

`IowError.Type.WRITE_NULL`

Error in your application code. You tried to write an `IowReport` that is null.

`IowError.Type.WRITE_UNSUPPORTED_MODE`

Error in your application code. You tried to write an `IowReport` with a report-id for a SpecialMode that is not available on this device. You would see this error for instance if you try to enable the KEY_MATRIX mode on an IOWarrior24.

`IowError.Type.UNPLUGGED`

The library was unable to deliver the report, because the device is already unplugged.

`IowError.Type.WRITE_TIMEOUT`

The library sets an internal timeout of 5 seconds for each write. The write did not complete inside this limit.

`IowError.Type.WRITE_FAILED`

Somehow the write failed!? This sounds rather vague, but our library is not able to cope with this error in a more detailed way. You can inspect the value returned from `err.getWin32Code()` if you want to handle it in your applications code.

The `write(IowReport [] reps)` method for an array of reports will either return `IowError.OK` if all reports were successfully written, or the error for the first report it failed to write. You will not be able to know how many reports (if any) actually made it to the device though.

3.4 Reading from the device

Reading is a bit more complicated than writing, but the whole thing is more or less a repetition of the concepts explained in the chapter about handling device changes.

Here again you will have to setup a class that implements an interface, and then register that class with one (or more) IOWarrior(s).

The `interface IowReportListener` defines to methods :

`void reportUpdate(IowReport rep, IOWarrior iow)`

This method will be called from the IOWarrior whenever a new report came in from the device. Since you can register your class with more than one IOWarrior the callback will provide you not only with the data itself but also with the device from which the data is reported.

`void deviceRemoved(IOWarrior iow)`

This will be called from one of the IOWarriors to which your class was registered. It simply tells you that this specific device was unplugged and no more reports are to be expected from it.

A basic implementation for an `IowReportListener` looks like this

```
public class MyReportListner : IowReportListener {
	public MyReportListener() {
	}

	public void reportUpdate(IowReport rep, IOWarrior iow) {
		Console.WriteLine("New Report from : "+iow);
		Console.WriteLine(rep.ToString());
	}

	public void deviceRemoved(IOWarrior iow) {
		Console.WriteLine("Removed : "+iow);
	}
}
```

It should be obvious what our listener does.

Now we have to register our class with a device. The `class IOWarrior` implements the method `addReportListener(IowReportListener listener,IowSpecialMode [] modes)` for this. The first argument to the call is the listener we just created, or to be more specific: an instance of a class that implements `interface IowReportListener`. The second argument, an array of `IowSpecialMode` constants, allows you to register only for specific reports in which you are interested.

```
//this one will be called on IO-Pin changes
MyReportListener forIO=new MyReportListener();
IowSpecialMode [] IOmode= { IowSpecialMode.IO };

//and this one for the switch-matrix only
MyReportListener forSwitches=new MyReportListener();
IowSpecialMode [] Switchmode= { IowSpecialMode.SWITCH_MATRIX };

//they are both added to the same device
iow.addReportListener(forIO,IOmode);
iow.addReportListener(forSwitches,Switchmode);
```

Here we create two instances of our listener class. Both are registered with the same IOWarrior. The first one will notified when one of the IO-Pins changes, the second one will be called for SwitchMatrix events only. Since we didn't register any more listener all other reports will be silently discarded inside the library.

If you want to receive every type of report coming in from the device, use `null` or an empty array for the second argument of `addReportListener()`.

If you are tired of waiting for reports from an IOWarrior, the `removeIowListener(IowReportListener listener)` method allows you to unregister you class from a device.

If an IOWarrior you have registered with is unplugged, the `deviceRemoved()` method of your handler is called. You don't have to unregister yourself on unplug. The IOWarrior will forget your listener after the final `deviceRemoved()` call has returned.

3.4.1 Inside the report event handler

Here are the restrictions for your applications code inside the callback methods for reports.

- **Terminate quickly**. Lengthy operations should be run in a new Thread which you control inside your applications code.
- If you want to call methods for a component that was derived from `Windows.Forms.Control` you have to use the `Invoke()` method of your component.

Sounds familiar, here's the background story.

When a new instance of `class IOWarrior` is created the library starts an internal Thread that waits for new reports from the device. As soon as a new report is available the `reportUpdate(IowReport rep, IOWarrior iow)` methods of all registered IowReportListeners will be called. At this point the Thread waits for the callbacks of your application code to return. The Operating System provides an internal buffer for reports from the device, but if your handlers code blocks for too long, you might loose some reports from the IOWarrior without notice.

3.4.2 `getIOStatus()` Handle with care!

The only method from the `class IOWarrior` we haven't mentioned yet is `getIOStatus()`. It returns the last `IowReport` read from the IO-Pins of the device. In the previous chapter we saw that reading from the device can be delayed or even stopped if the code in your report event handler blocks for too long. This affects `getIOStatus()` too, since the report to be returned, is updated from the same Thread that calling into your event handlers.

4 CsDemo

I created a small demo project `CsDemo`. The application opens a window which displays all connected IOWarriors in a `ListBox`. A report can be send to the selected device by editing the data-fields at the bottom of the window. Every report that comes in from any of the devices is printed into a `TextBox`.

There are only three classes in the project:

`HexTextBox`

A simple class extending `TextBox` that restricts the users input to hex values. This class does not use any functionality from the CsIow library.

`WritePanel`

A user control that provides 64 `HexTextBox` fields and a write button. The only relation with the CsIow library is that a new `IowReport` is created from the textbox values when the user clicks the write button. A delegate from the applications main form is called with this report and the data is written to the currently selected device.

`Form1`

This is the applications main form and all code dealing with the IOWarriors is implemented here.

Let's have a look at the code of `Form1` to see how the library is put into practice.

At the top of the file we include our library to the project and set the namespace to something unequal to he libraries namespace

```
using org.wayoda.csiow;
namespace CsDemo
```

Since we want to handle all callbacks from the library in the forms code we need a class that implements the interfaces `IowDeviceChangeListener` and `IowReportListener`. The statement

```
public partial class Form1 : Form, IowDeviceChangeListener,IowReportListener {
```

shows that all the event handlers will be defined inside the forms code itself.

Now we need a place to open and close the library. The `Form1_Load` event is a good place to do that:

```
private void Form1_Load(object sender, EventArgs e) {
	IowDeviceFilter f=new IowDeviceFilter();
	//uncomment the next line if you want to see only IOWarrior24's
	//f.addProduct(IOWarrior.PID_IOW24);
	IowError err;
	err=IowManager.open(this,f);
	if(err!=IowError.OK) {
		StringBuilder sb=new StringBuilder();
		sb.AppendLine("Opening the CsIow-Lib failed");
		sb.AppendLine("("+err.ToString()+")");
		MessageBox.Show(sb.ToString(), "SimpleIow Failed!");
		Application.Exit();
	}
}
```

We create a new `IowDeviceFilter` but we do not set any conditions for it. For the demo we simply accept every IOWarrior that gets plugged in.

In the `IowManager.open(this,f)` call we register the main form itself as the listener for device changes.

The library will finally be closed when the user closes our form.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e) {
      IowManager.close();
}
```

Now we need to implement the callbacks for device changes defined in interface `IowDeviceChangeListener`.

```
public void deviceAdd(IOWarrior iow) {
      AddIowToControl ais=new AddIowToControl(deviceListBox.Items.Add);
      deviceListBox.Invoke(ais, new Object [] {iow});
      iow.addReportListener(this, null);
}
```

The first two lines in the handler create a delegate and use the `Invoke()` call of the `ListBox` to add our new device to that listbox.

The statement `iow.addReportListener(this, null)` registers the form as the eventhandler for all reports that come in from the device. Since we want to register for every type of report coming in, the filter for `IowSpecialMode` is null.

Here's the method that handles unplug events from the library

```
public void deviceRemoved(IOWarrior iow) {
      RemoveIowFromControl ris=new RemoveIowFromControl(deviceListBox.
                                                Items.RemoveAt);
      lock(deviceListBox) {
            for(int i=0;i<deviceListBox.Items.Count;i++) {
                  // The test for the device can be done without
                  // Invoke, because the visual appearance of the
                  // control is unchanged
                  if(iow.Equals(deviceListBox.Items[i])) {
                        //Here we need to use Invoke for device removal
                        deviceListBox.Invoke(ris, new object[] { i });
                  }
            }
      }
}
```

Again we create a delegate to access the `ListBox`. Than we iterate through the items in the listbox and remove the IOWarrior that was unplugged. As you see not every method from the `ListBox` needs to be wrapped with Invoke. We have access to the number of items in the list and can also check the items in the list against the IOWarrior that was removed. But actually removing the device will repaint the listbox on the window. So that must be wrapped by `Invoke`.

That's all we had to implement for device plug/unplug events. Lets get on to the reports send by IOWarriors.

I guess you remember that interface `IowReportListener` also defines the method `deviceRemoved(IOWarrior iow)`. But in our example we don't have to implement it a second time because both interfaces use the same signature. Our method from above

simply gets called twice on each removal. One call comes from the `IowManager` to which we registered for device changes. The second one comes from the device itself. Don't expect these calls to come in any order. They are coming from different threads. Since both calls change the items in the `ListBox` we protect access to the control by locking it. The first call (thread) coming in will have exclusive access to the list. The second call will have to wait until the first one is done with the list.

Reports coming in from the device will simply be printed to a `TextBox` on the window.

```
public void reportUpdate(IowReport rep, IOWarrior iow) {
	// Create a nice message to be printed in the ReportTextBox
	int reportLength;
	String dev=iow.ToString();
	StringBuilder sb=new StringBuilder();
	sb.Append("R ");
	sb.Append(iow.getProductName());
	sb.Append(" ");
	sb.AppendFormat("{0:X} ", iow.getSerialNumber());
	if(rep.getSpecialMode()==IowSpecialMode.IO)
		reportLength=iow.getReportSizeIO();
	else
		reportLength=iow.getReportSizeSM();
	for(int i=0;i<reportLength;i++) {
		sb.AppendFormat("{0:X2} ", rep[i]);
	}
	sb.Append("\n");
	// And now add this text using a delegate and Invoke again.
	AddReportToControl adds=new AddReportToControl(ReportList.AppendText);
	ReportList.Invoke(adds, new object[] { sb.ToString() });
}
```

Inside the method we create a nicely formatted String for the data just read. We put the name of the device and its serial-number in front. Then we print the individual bytes from the report. As you know an `IowReport` will always provide you with 64 bytes of data, but not all of them carry any meaning in a report. Let's say the report came from the IO-Pins of an IOWarrior40. For this device only the first 5 bytes in the `IowReport` make up the whole data. So we want to print out only these 5 bytes on the window. This is taken care of by requesting the type of SpecialMode from the report itself. Than we ask the device how many bytes make up a report of the specific type. Only these bytes will than actually be printed on the form. Adding the message to the textbox has again to be wrapped by Invoke().

The only thing that's missing now is a method that writes a new report to one of the devices.

```
public void writeToIow(IowReport rep) {
	IowError werr;
	IOWarrior iow=deviceListBox.SelectedItem as IOWarrior;
	if(iow!=null) {
	werr=iow.write(rep);
	if(werr!=IowError.OK) {
		StringBuilder sb=new StringBuilder();
		sb.Append("Write for device ");sb.Append(iow.getProductName());
		sb.Append(" ");sb.Append(iow.getSerialNumberString());
		sb.AppendLine(" failed!");sb.AppendLine(werr.getMessage());
		MessageBox.Show(sb.ToString(), "Write Error");
	}
}
```

The method is called from the `WritePanel`-class which created the `IowReport` when the user clicked the write-button. It retrieves the currently selected IOWarrior from the list of devices and writes the report to it. If an error is detected we show a MessageBox with a description of the error.

That was all we had to implement concerning the IOWarriors. I didn't show you the delegates we used for `Invoke()` on the `ListBox` and `TextBox` but I guess you find your way around by looking at the sources.

The binary for the demo can be installed from directory publish in the projects source-tree (or it can be rebuild and started in the debugger, if you prefer that.)

I also packaged the CsIow-library DLL into the projects source-tree and added a reference to it.

In you own projects you should add a reference to the CsIow-library project itself so you're always using the latest version of the library.

Revision History of this document:

12/11/2006 First public release

Cslow library changelog

12/11/2006 Version 0.1.0.0 First public release