JCamNet

API Documentation

JoeScan Inc.

July 2015

Contents

1 Main Page

1.1 Introduction

JCamNet is the implementation of an API (Application Programming Interface) for the JoeScan line of Laserscanners. It can be used to write applications that make use of one or more scanners, and is the same interface that is used internally at JoeScan to build custom and diagnostic applications.

JCamNet is delivered as an assembly for the .NET framework. Even though it is implemented in C#, the functionality can be accessed through any language targeting the CLR 4.0 platform, such as VisualBasic for .NET. The provided examples are written in C# 4.0.

1.2 Online documentation

JoeScan provides a wide range of support documents on our Support Website (http://joescan.-com/support). Please refer to the documents there for

- mechanical,
- electrical,
- and Performance specifications,
- the configuration utility JSDiag,
- and a parameters reference.

The developer documentation for the C/C++ library (**JCamDll**) may also be helpful. **JCamNet** and **JCamDll** contain roughly the same functionality, but have different calling conventions, and in some cases, different naming. Nonetheless, the examples and documentation can help to clarify concepts and techniques.

The most current version of this document can also always be found on the support page: http://joescan.-com/jcamnet .

1.3 Examples

2 Getting Started

2.1 Prerequisites

2.1.1 A Short Note Before We Begin

The code for the examples has been simplified to show a specific area of interest, and does not include proper error checking. As such, the example code is not ready for production purposes, and should be seen as an illustration for a specific concept only. It is up to you, the developer, to catch exceptions and handle error situations appropriately. Your application should react gracefully to those conditions and make no assumptions about undocumented behaviour. You have been warned.

2.1.2 Visual Studio and .NET

- Visual Studio 2010 (SP1 recommended) or Visual C# Express 2010
- .NET Framework 4.0 (in Windows 7, installed by default)

The example projects were built in C#4.0 with Visual Studio 2010 (SP1). Using Visual Basic.NET or other languages targeting the CLR 4.0 is possible but not supported at this time.

2.2 Referencing the JCamNet assembly from a project

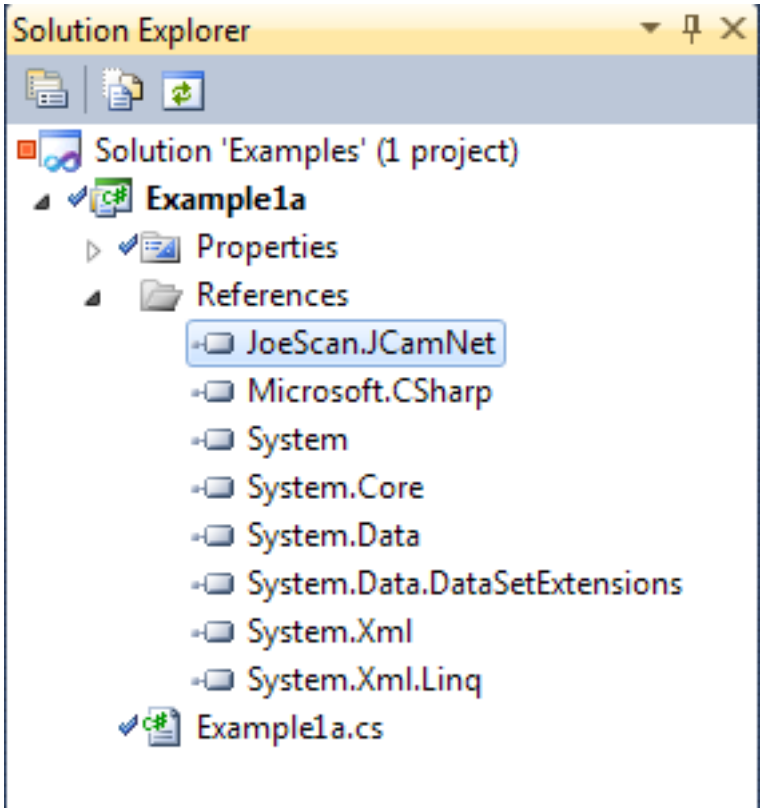
The **JCamNet** assembly is delivered as a DLL named **JoeScan.JCamNet.dll** in the **lib** subfolder of the **JCamNet** distribution. At the same location you will find **JoeScan.JCamNet.xml**, which aides Visual Studio in providing IntelliSense help, such as parameter info and function names.

2.2.1 Adding a Reference

You need to add a reference to the **JCamNet** assembly to your project. Here's how:

- In Visual Studio, right-click on your project in the Solution Explorer, and select "Add Reference...",
- Switch to the "Browse" tab,
- navigate to the directory where **JoeScan.JCamNet.dll** is stored, and click "OK".

The result should look like this:



You project will now list JoeScan.JCamNet under its References, and all publicly accessible symbols in the API are available to you. If the **JoeScan.JCamNet.xml** file is at the same file location (you don't need to reference it), IntelliSense in Visual Studio will automatically show the help when using symbols from the **JCamNet** API. In the example projects, this has already been done for you.

2.3 The JoeScan.JCamNet namespace

Including the line

```
using JoeScan.JCamNet;
```

at the top of your source file will import the namespace and allow you to use the symbols from the assembly with their short names, e.g. Scanner instead of JoeScan.JCamNet.Scanner. In the example code, this has already been done for you.

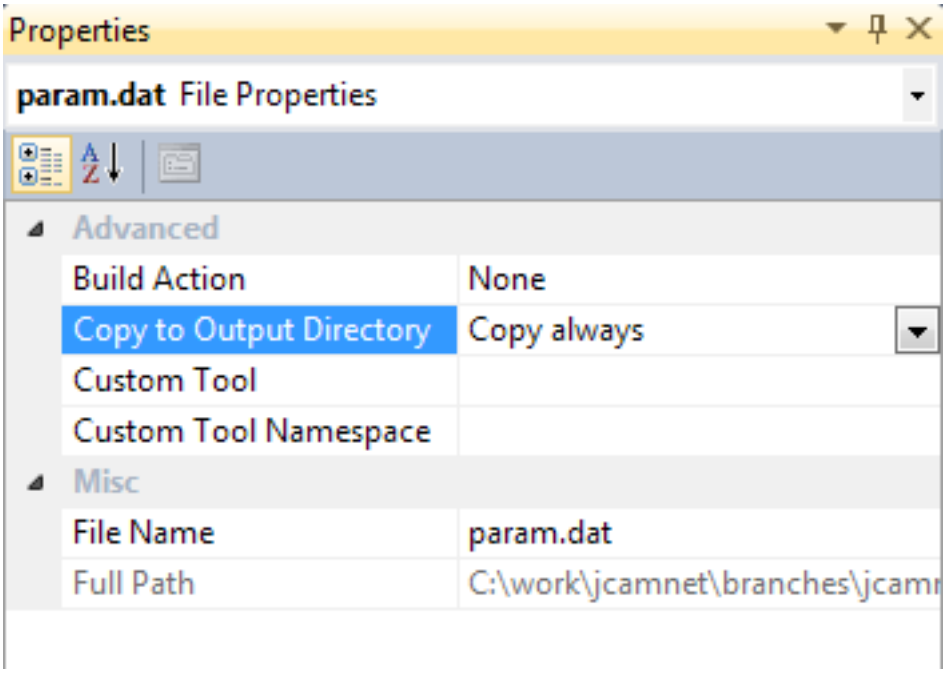
2.4 The param.dat file

Scan heads are configured by sending a text string of parameters to them. Almost all applications keep this data in a text file, and per convention it typically is named param.dat. For the meaning and format of the parameters, please see the documentation at the JoeScan support site (`http://joescan.com/Parameters.html`). The parameters file used to configure the heads needs to be accessible from your application, so it can be read and it's contents sent to the scanner. You can either keep this file in a fixed location on your hard drive and refer to it by an absolute path, or keep it co-located with your application. In the example projects, the latter approach was taken. To do the same in your projects:

- In Visual Studio, right click on your project in the Solution Explorer, and select "Add", then "Existing Item",
- In the file dialog opening, change the file type from "Visual C# Files (...)" to "All Files (.)",
- Navigate to and pick the "param.dat" file you want to use, and hit "Add".

The file "param.dat" is now part of your project, however, Visual Studio still needs to be told what to do with the file when the project is built. In our case, we want the file to be copied to the output directory where your application (exe) is located. To do so:

- Right click on the "param.dat" and select "Properties"
- In the Property Editor that opens, set the property "Copy to Output Directory" to "Copy always".



Remember that any changes you make to the parameter file in the JSDiag application will not be reflected in your local copy. Therefore it is advised to deploy your application in such a way that JSDiag and your application run from the same directory and share the same param.dat.

3 On-Demand and Synchronized Scanning

3.1 On-Demand Mode

After first connecting to a scanner, it will be in what is called On-Demand mode. In this mode, you can send a request, and it will execute the measuring process and return a Profile. The call to GetProfile() will block, i.e. hold the execution of your application, for as long as it takes to excute the request. Similarly, the call pair BeginGetProfile()/ EndGetProfile() will block in EndGetProfile() until the measured profile is returned.

This mode is most suitable for applications with low scan rates and a single head. You can not "overdrive" the scanner in this mode, i.e. force it to go too fast and lose data. A major downside of On-Demand mode is the fact that you will not be able to achieve high scan speeds. Because the time a scan is taken can be affected by network timing and PC speed, On-Demand mode is also not suitable for interleaving multiple scanners to avoid interference.

3.2 Synchronized Scanning

Synchronized Scanning (also referred to as Synchronized Mode) is another way of getting profile data from the scanner. The primary difference to On-Demand Mode is that the trigger for a scan does not come from the controlling computer but from a timer, an encoder or a pulse on the StartScan input line. Any of these events can trigger a scan, after which the resulting profile is stored onboard the scanner head until the controlling computer requests a transfer.

The three subtypes of Synchronized Scanning Mode are:

- TimedSyncMode
- EncoderSyncMode
- PulseSyncMode

In TimedSyncMode or EncoderSyncMode, the scanner will automatically scan at preset intervals (from the parameters file), based on elapsed time or encoder value, respectively.

Note

TimeSyncMode is not suitable for a multi-head setup, because each head uses its own internal clock, which may drift compared to the others.

As stated above, a scanner starts in On-Demand Mode. Enter Synchronized Mode by calling one of these functions:

- EnterEncoderSyncMode()
- EnterTimedSyncMode()
- EnterPulseSyncMode()

In Synchronized Mode, retrieve profiles from the scanner by calling any of these functions:

- GetQueuedProfiles()
- GetQueuedProfiles(int maxProfiles)
- BeginGetQueuedProfiles() or BeginGetQueuedProfiles(int maxProfiles) , followed by EndGetQueuedProfiles()

Restart Synchronized Mode:

Call EnterEncoderSyncMode() , EnterTimedSyncMode() , or EnterPulseSyncMode() to discard any remaining unread profiles and restart Synchronized Mode.

Leave Synchronized Mode:

Call ExitSyncMode() () to discard any remaining unread profiles and fall back to Non-Synchronized (On-Demand) Mode.

If you call a non-synchronized function during Synchronized Mode, an exception of type JCamNet.ScannerOperations-Exception will be thrown.

Note

The term Synchronized Scanning is somewhat misleading, because the scan and readout cycles are run independently (asynchronously). The naming instead comes from the fact that all scan heads are triggered by a signal from an encoder or timer, and thusly operate "in sync" with each other.

4 Examples 1a, 1b, 1c: Finding Scanners on the network

Scanner discovery follows a request/response model, whereby the user's application requests scanners that meet certain criteria (it can also request all available scanners), and the scanner(s) respond to the request by sending their ID information. The responses are not guaranteed to be returned in real-time; they're sent via UDP. Consequently, after the request, the functions used for discovering scanners block for approximately one second in order to reasonably ensure that all scanners get their responses in.

The available discovery functions are located in the class ScannerConfig:

- public static List<DiscoveryResponse> ScannerConfig.FindAllScanners()
- public static DiscoveryResponse ScannerConfig.FindScanner(int id)
- public static void ScannerConfig.FindAllScanners(ScannerResponseDelegate srh)

The code in project Example1a :

```
using System.Collections.Generic;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            List<DiscoveryResponse> responses = ScannerConfig.FindAllScanners();
            foreach (var discoveryResponse in responses)
            {
                System.Console.WriteLine("Base IP Address: {0}, CableId: {1}, Serial Number: {2}",
                    discoveryResponse.BaseIPAddress,
                    discoveryResponse.CableId,
                    discoveryResponse.SerialNumber );
            }
        }
    }
}
```

Sample Output:

```
 Base IP Address: 192.168.1.105, CableId: 0, Serial Number: 374
 Base IP Address: 192.168.1.150, CableId: 0, Serial Number: 1465
 Base IP Address: 192.168.1.105, CableId: 22, Serial Number: 1941
 Base IP Address: 192.168.1.105, CableId: 28, Serial Number: 1940
 Base IP Address: 192.168.1.105, CableId: 30, Serial Number: 1747
```

If you want to query for a specific scanner, either by it's CableId or Serial number, use the second function above:

The parameter **id** is either a CableId or a serial number. Because no hardware with serial numbers below 128 exists, the API assumes any argument < 128 is a CableId, and a serial number otherwise. However, there may be multiple scanners on your network that have the same CableId (and a different IP Base Address). In this case, it is not safe to use this function, as it only returns one DiscoveryResponse and may throw an exception.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        private static int serialNumber = 1465;
        private static int cableId = 0;

        static void Main(string[] args)
        {
            DiscoveryResponse response = ScannerConfig.FindScanner(serialNumber);
            System.Console.WriteLine("Base IP Address: {0}, CableId: {1}, Serial Number: {2}",
                    response.BaseIPAddress, response.CableId, response.SerialNumber);

           // don't do this: if multiple scanners respond, you will receive an exception
           // of type JoeScan.JCamNet.ScannerCommunicationException

           // response = ScannerConfig.FindScanner(cableId);
           // System.Console.WriteLine("Base IP Address: {0}, CableId: {1}, Serial Number: {2}",
           //         response.BaseIPAddress, response.CableId, response.SerialNumber);

        }
    }
}
```

Sample Output:

```
Base IP Address: 192.168.1.150, CableId: 0, Serial Number: 1465
```

Another possibility to find scanners is to use a delegate method of type ScannerResponseDelegate, which is called for each DiscoveryResponse received fom the network. This is useful when you have specific criteria to build the list of scanners you want to adress, e.g. all scanners on a specific subnet.

In the following example, we only want to work with scanners that have a "1" in their network address at the 3rd quad.

Note

For proper subnet testing instead of this simplified version, see: http://en.wikipedia.org/wiki/-Subnetwork

```
using JoeScan.JCamNet;
using System;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            ScannerConfig.FindAllScanners(MyDelegate);
        }

        public static void MyDelegate(DiscoveryResponse response)
        {
            // filter here by your criteria, for instance the subnet:
            if (response.BaseIPAddress.GetAddressBytes().GetValue(2).ToString() == "1")
            {
                System.Console.WriteLine("Base IP Address: {0}, CableId: {1}, Serial Number: {2}",
                    response.BaseIPAddress, response.CableId, response.SerialNumber);
            }
        }
    }
}
```

Sample Output:

```
Base IP Address: 192.168.1.150, CableId: 0, Serial Number: 1465
Base IP Address: 192.168.1.105, CableId: 22, Serial Number: 1940
Base IP Address: 192.168.1.105, CableId: 30, Serial Number: 1941
```

Note

In general, it is recommended that you not use the discovery functions in a production application, it is better to configure your scanners once and store IP addresses and CableIds with your application. For diagnostics and configuration applications, the usage of the discovery functions is fine.

5 Examples 2a, 2b, 2c : Connecting to scanners

A scanner on the network is represented by an object of type Scanner. Several static functions in the Scanner class will produce a Scanner for you:

- Scanner.Connect(IPAddress)
- Scanner.Connect(IPAddress baseIpAddress, short cableId)
- Scanner.Connect(IPAddress baseIpAddress, params short[] cableIds)

Note

Creating a connection to a scanner head will disconnect it from any other application that may hold a connection. If you are using the JSDiag diagnostic tool, at the same time as you run your application, your connection may fail. The JSConfig tool is safe to use.

5.1 Example2a

Example 2a shows you how to connect to a scanner that has a statically configured IP address:

```
using System;
using System.Net;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            IPAddress ip = new IPAddress(new byte[] {192,168,1,150});
            Scanner s = null;
            try
            {
                 s = Scanner.Connect(ip);
            }
            catch (Exception e)
            {
                Console.WriteLine("Failed to connect to scanner: {0}, Reason: {1}",
                    ip.ToString(),
                    e.Message);
                return;
            }
            Console.WriteLine("Connected to Scanner! Serial #: {0}", s.SerialNumber);
        }
    }
}
```

Sample output:

```
Connected to Scanner! Serial #: 1465
```

Note

If the scanner can't be reached, the function will time out and throw an exception. It is up to you to properly handle this exception. The time before the function times out can be several seconds, so for an application with a Graphical User Interface it is advisable to run this code in a way that doesn't block the main UI, and handles the failed connection gracefully.

5.2 Example2b

The more common case is a multi-scanner setup, where all scanners have the base IP set, and adressed via their CableIds. In this example, we connect sequentially to all scanners with the BaseIp Address 192.168.1.150 and CableIds 0 and 1. Both scanners respond.

```
using System.Net;
using JoeScan.JCamNet;
using System;


namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            short[] cableIds = new short[] { 0, 1 };
            IPAddress baseIpAddress = new IPAddress(new byte[] { 192, 168, 1, 150 });
            foreach (var cableId in cableIds)
            {
                Scanner s = null;
                try
                {
                    s = Scanner.Connect(baseIpAddress, cableId);
                    Console.WriteLine("Connected to Scanner! Serial #: {0}", s.SerialNumber);
```

```
                }
                catch (Exception e)
                {
                    Console.WriteLine("Failed to connect to scanner: {0}, Reason: {1}",
                        baseIpAddress.ToString(),
                        e.Message);
                }
            }
        }
    }
}
```

Sample output is:

```
Connected to Scanner! Serial #: 1465
Connected to Scanner! Serial #: 1329
```

5.3 Example2c

A convenience function Scanner.Connect(IPAddress baseIpAddress, params short[] cableIds) is provided that will connect to all scanners for a given BaseIP and an array of CableIds. All scanners must be configured to use this addressing method. The scanners are contacted sequentially, so there is no speed advantage over using Scanner.-Connect(IPAddress baseIpAddress, short cableId) , however, it is important to note that this function will not throw an exception in case a scanner is unreachable. Instead, the returned list of scanner objects must be checked to determine if all requested CableIds have responded:

```
using System;
using System.Net;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            short[] cableIds = new short[] { 0, 1 , 2};
            IPAddress baseIpAddress = new IPAddress(new byte[] { 192, 168, 1, 150 });
            Scanner[] found = Scanner.Connect(baseIpAddress, cableIds);
            foreach (var scanner in found)
            {
                Console.WriteLine("Connected to Scanner! Serial #: {0}", scanner.SerialNumber);
            }
            if (cableIds.Length != found.Length)
            {
                Console.WriteLine("Not all CableIds responded.");
            }
        }
    }
}
```

Sample output is:

```
Connected to Scanner! Serial #: 1465
Connected to Scanner! Serial #: 1329
Not all CableIds responded.
```

6 Example 3: Configuring Scanners

Once you have connected to the scanner head(s), you are ready to scan. In most cases however, the heads may not yet be configured to your requirements, so this example looks at the procedure to configure a scanner head. Even though the scanner head will retain configuration information in its nonvolatile memory, it is recommended practice to send the configuration regardless.

The configuration parameters are stored in a simple text format that is documented in the Parameter Reference .

```
using System;
using System.IO;
using System.Net;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            IPAddress baseIpAddress = new IPAddress(new byte[] { 192, 168, 1, 150 });
            // the scanner we want to configure
            Scanner scanner = Scanner.Connect(baseIpAddress, 0);

            TextReader parametersReader;
            // Make sure your application can find the parameters file. In Visual Studio,
            // you can add the file to your project, and use the "Copy to Output Directory" property,
            // forcing Visual Studio to copy the file to your output directory every time you build.
            using (parametersReader = File.OpenText("param.dat"))
            {
                // the parameters are passed as one long string
                string parametersString = parametersReader.ReadToEnd();
                try
                {
                    // send the configuration parameters, and tell the head to store it permanently
                    scanner.SetParameters(parametersString, true);
                    Console.WriteLine("Scanner {0}:{1} was successfully configured!",
                       scanner.IPAddress.ToString(), scanner.CableID);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Scanner {0}:{1} did not accept configuration parameters!",
                        scanner.IPAddress.ToString(), scanner.CableID);
                }
            }
        }
    }
}
```

The parameters are passed as a simple string, containing all of the file at once. Depending on your application, you can provide this string any way you like, e.g. retrieve it from a database, read it from a file, or build it 'on the fly'.

Note

Be aware that some text editors change the encoding of the file and store it as 'UTF-8', in which case the SetParameters() function may fail.

7 Example 4: On-Demand Scanning

After connecting to a scanner head and configuring it, you can now request data. There are several ways to do this, but the simplest is On-Demand Scanning. By default, the scanner head starts in this mode, and all you have to do to get data is to call Scanner.GetProfile() .

There are a number of issues to consider when using On-Demand Scanning.

- Speed: the scan speed in this mode is lower than in Synchronized Mode, because the head can not effectively parallelize readouts and scan requests.
- Timing: the timing is entirely dependent on the readout loop of your controlling computer. If the object in the scanner's view is moving, the distance between consecutive profiles may be different, depending on the processing time needed in the readout loop.
- Synchronization: in On-Demand mode, synchronizing multiple scan heads is not possible.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using JoeScan.JCamNet;
```

```
namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            IPAddress baseIpAddress = new IPAddress(new byte[] { 192, 168, 1, 150 });
            Scanner scanner = null;
            try
            {
                scanner = Scanner.Connect(baseIpAddress, 0);
            }
            catch (Exception e)
            {
                Console.WriteLine("Failed to connect to scanner: {0}, Reason: {1}",
                    baseIpAddress.ToString(), e.Message);
                return;
            }
            TextReader parametersReader;
            using (parametersReader = File.OpenText("param.dat"))
            {
                string parametersString = parametersReader.ReadToEnd();
                try
                {
                    scanner.SetParameters(parametersString, true);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Scanner {0}:{1} did not accept configuration parameters!",
                        scanner.IPAddress.ToString(), scanner.CableID);
                    return;
                }
            }
            // Now we're ready to scan. In this example, we retrieve a fixed number of Profiles from
            // the scan head. In practice, this would be done in a loop.
            List<Profile> profiles = new List<Profile>();
            for (int i = 0; i < 10; i++)
            {
                try
                {
                    Profile p = scanner.GetProfile();
                    Console.WriteLine("Profile {0}: measured {1} points.", i, p.Count);
                }
                catch(Exception e)
                {
                    Console.WriteLine("Errror retrieving profile from scanner. Exiting.");
                    return;
                }
            }
        }
    }
}
```

Sample output is:

```
Profile 0: measured 218 points.
Profile 1: measured 218 points.
Profile 2: measured 218 points.
Profile 3: measured 218 points.
Profile 4: measured 217 points.
Profile 5: measured 218 points.
Profile 6: measured 218 points.
Profile 7: measured 217 points.
Profile 8: measured 217 points.
Profile 9: measured 218 points.
```

Note

The number of measured points in the example above fluctuates due to external influences, e.g. ambient light.

8 Examples 5a, 5b: Synchronized Scanning

Example 5a illustrates the usage of Synchronized Mode. The loop reading out the data will only be ended when a certain number (10) of Profiles are read. In practice, this approach is not recommended, because a tight loop like this

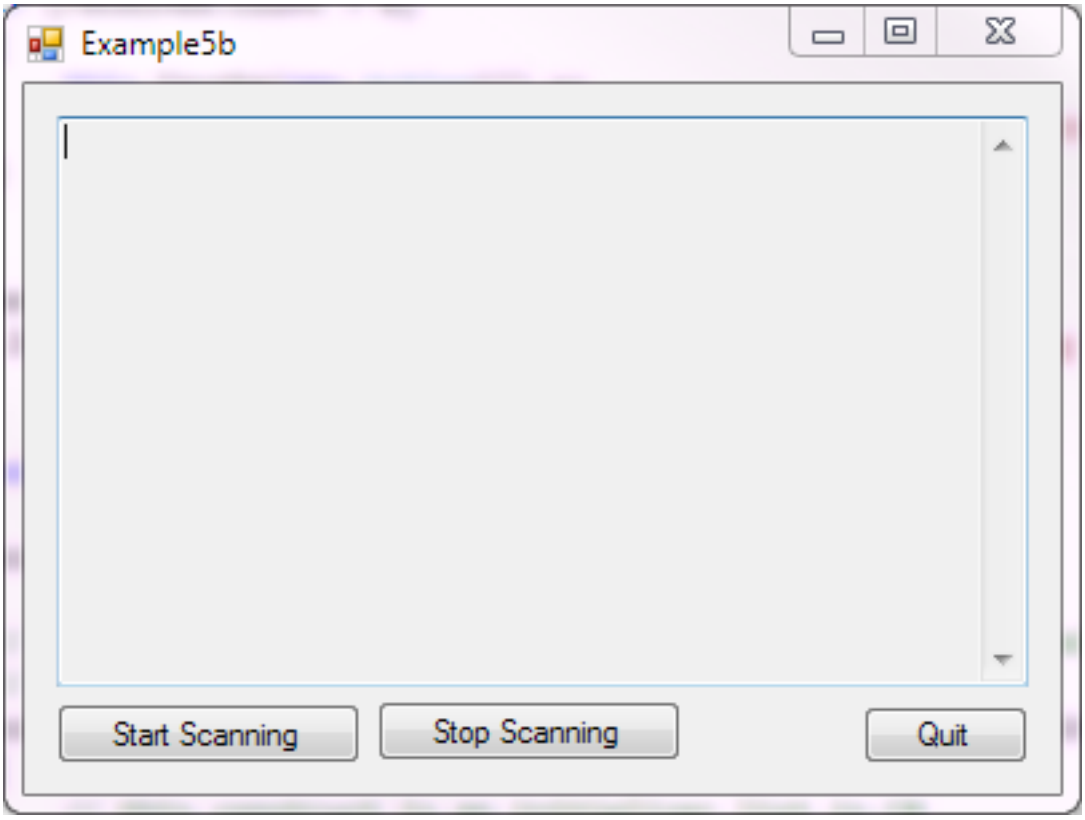
would block the rest of your program. Also, if for some reason the scanner can't measure any points, your program will be stuck in an endless loop. In the next example we will look at a better solution, using threads.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            IPAddress baseIpAddress = new IPAddress(new byte[] { 192, 168, 1, 150 });
            Scanner scanner = null;
            try
            {
                scanner = Scanner.Connect(baseIpAddress, 0);
            }
            catch (Exception e)
            {
                Console.WriteLine("Failed to connect to scanner: {0}, Reason: {1}",
                    baseIpAddress.ToString(), e.Message);
                return;
            }
            TextReader parametersReader;
            using (parametersReader = File.OpenText("param.dat"))
            {
                string parametersString = parametersReader.ReadToEnd();
                try
                {
                    scanner.SetParameters(parametersString, true);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Scanner {0}:{1} did not accept configuration parameters!",
                        scanner.IPAddress.ToString(), scanner.CableID);
                    return;
                }
            }
            // Now we're ready to scan. In this example, we scan until we have received ten Profiles.
            // Be aware that this loop will not end until the ten profiles have been measured, so if you
            // try this code on a real scan head that for some reason can't measure any points
            // (blocked view, wrong window settings etc.), the program will appear to hang!

            scanner.EnterEncoderSyncMode();
            int profileCount = 0;
            while (profileCount < 10)
            {
                try
                {
                    // In Sync Mode, you must use GetQueuedProfiles() instead of GetProfile()
                    // Here we request one profile at a time. If the encoder hasn't triggered a scan yet,
                    // the returned list is empty.
                    List<Profile> received = scanner.GetQueuedProfiles(1);
                    if (received.Count > 0)
                    {
                        Console.WriteLine("Profile {0}: measured {1} points.",
                            profileCount, received[0].Count);
                        profileCount++;
                    }
                }
                catch (Exception e)
                {
                    Console.WriteLine("Errror retrieving profile from scanner. Exiting.");
                    break;
                }
            }
            // exiting sync mode will clean up all unread profiles from the head
            scanner.ExitSyncMode();
        }
    }
}
```

A better alternative to continually polling the scanners in your application is to use a dedicated thread. Example 5b shows you how to accomplish just that. The code becomes quite a bit more complex, but most of it is actually boilerplate that

can be reused.

In Example 5b, we create a Windows Forms application and show a simple Form:



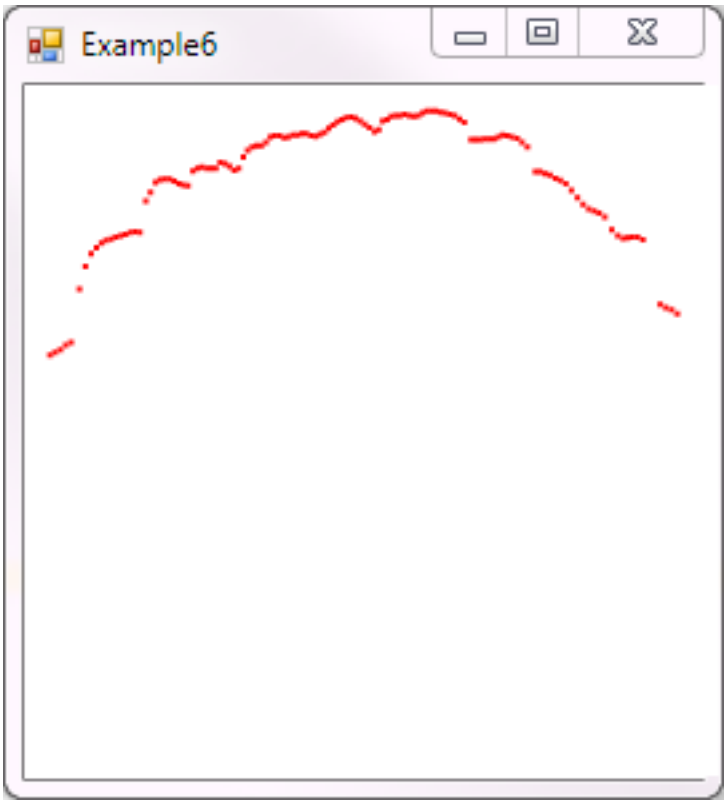
In response to a click on the "Start Scanning" button, the application creates a new thread, and assigns it a function (ScanThreadMain) to run. The code there contains an endless loop that polls the scanners again and again. The main UI runs in it's own thread and does nothing but wait for user interaction. As soon as a profile was read from a scan head, the ScanThreadMain uses Invoke() to execute code on the UI thread. The Invoke() call is necessary, because threads are not allowed to make calls directly across thread boundaries. The benefit of this design is that the UI is always responsive and reacts timely. Go ahead and try it out -- the window can be resized and moved even under a heavy load.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;
using System.Windows.Forms;
using JoeScan.JCamNet;

namespace Examples
{
    public partial class Form1 : Form
    {
        private Thread scanThread;
        private volatile bool isRunning;
        private Scanner scanner;

        public Form1()
        {
            InitializeComponent();
            scanner = SetupScanner();
        }

        // ScanThreadMain contains the code that is executed in a thread, i.e. separate
        // from the User Interface
        private void ScanThreadMain()
        {
```

```
            // we will use encoder sync mode here
            scanner.EnterEncoderSyncMode();
            // the code on a thread can not directly make calls on the
            // UI thread, hence the need for Invoke()
            this.Invoke(new Action(() => this.textBox.Clear()));
            this.Invoke(new Action(() => this.textBox.AppendText("Scanning Started.\n")));

            // this is an endless loop until the isRunning variable is changed
            // to false from the outside of this thread
            while (isRunning)
            {
                // GetQueuedProfiles() is a shortcut for BeginGetQueuedProfiles()/EndGetQueuedProfiles()
                // See the Example on Hi-Speed Scanning for more information
                List<Profile> received = scanner.GetQueuedProfiles(1);
                if (received.Count > 0)
                {
                    this.Invoke(new Action(() =>
                        this.textBox.AppendText(String.Format("Profile with {0} points measured.\n",
                        received[0].Count))));
                }
            }

            scanner.ExitSyncMode();
            this.Invoke(new Action(() => this.textBox.AppendText("Scanning Stopped.\n")));
        }

        private void StartButton_Click(object sender, EventArgs e)
        {
            if (scanThread == null || !scanThread.IsAlive)
            {
                // we create a new thread here and assign the ScanThreadMain as the code path
                // to be executed when the thread is started
                scanThread = new Thread(new ThreadStart(this.ScanThreadMain))
                {
                    // this construct is an initializer list in C#
                    IsBackground = true,
                    Name = "ScanThreadMain",
                    Priority = ThreadPriority.Highest
                };
                isRunning = true;
                scanThread.Start();
            }
        }

        private void StopButton_Click(object sender, EventArgs e)
        {
            StopScanning();
        }

        private void StopScanning()
        {
            isRunning = false;
            if (scanThread != null)
            {
                // wait a moment for the thread to close down
                scanThread.Join(100);
                scanThread = null;
            }
        }

        private void QuitButton_Click(object sender, EventArgs e)
        {
            StopScanning();
            Close();
        }

        private Scanner SetupScanner()
        {
            // for clarity, no error checking here
            Scanner s = Scanner.Connect(new IPAddress(new byte[] {192, 168, 1, 150}));
            using (TextReader parametersReader = File.OpenText("param.dat"))
            {
                string parametersString = parametersReader.ReadToEnd();
                s.SetParameters(parametersString, true);
            }
            return s;
        }
    }
}
```

9 Example 6: A Simple Realtime Display

Example 6 is similar to Example 5b, but this time we use the obtained profiles to show a very simple realtime display of ProfileDataPoints on a Windows form:



The scanning thread forces the scan head into TimedSyncMode, and then proceeds to poll for Profiles, just like in the previous examples. Meanwhile, in the UI a standard Charting component (System.Windows.Forms.DataVisualization.-Charting.Chart) is updated every 80 milliseconds with data from a Profile. The important part is the locking: both the main thread and the scan thread have access to the variable latestProfile, so in order to avoid a conflict, we have to lock access to the variable for the short time that it takes to write new data into it.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;
using JoeScan.JCamNet;
using Timer = System.Windows.Forms.Timer;

namespace Examples
{
    public partial class Form1 : Form
    {
        private Thread scanThread;
        private volatile bool isRunning;
        private Scanner scanner;
        private Profile latestProfile;

        public Form1()
        {
            InitializeComponent();
            scanner = SetupScanner();
            Timer liveUpdateTimer = new Timer {Interval = 80};
            liveUpdateTimer.Tick += new EventHandler(UpdateLiveProfiles);
```

```
        liveUpdateTimer.Start();
        StartScanning();
    }

    private void UpdateLiveProfiles(object sender, EventArgs e)
    {
        if (latestProfile != null)
        {
            Series series = LiveView.Series[0];
            series.Points.Clear();
            // to prevent that the scanning thread overwrites
            // latestProfile, we lock around the code that reads the variable
            lock (this)
            {
                LaserOnTimeLabel.Text = String.Format("{0} ms", latestProfile.LaserOnTime/1000.0);
                foreach (var pt in latestProfile)
                {
                    series.Points.AddXY(pt.X, pt.Y);
                }
            }
        }
    }

    private void ScanThreadMain()
    {
        scanner.EnterTimedSyncMode();
        while (isRunning)
        {
            List<Profile> received = scanner.GetQueuedProfiles(1);
            if (received.Count > 0)
            {
                // to prevent overwriting the variable while it's being
                // read in the GUI thread, we need to lock
                lock (this)
                {
                    latestProfile = received[0];
                }
            }
        }

        scanner.ExitSyncMode();
    }

    private void StartScanning()
    {
        if (scanThread == null || !scanThread.IsAlive)
        {
            scanThread = new Thread(new ThreadStart(this.ScanThreadMain))
            {
                IsBackground = true,
                Name = "ScanThreadMain",
                Priority = ThreadPriority.Highest
            };
            isRunning = true;
            scanThread.Start();
        }
    }

    private void StopScanning()
    {
        isRunning = false;
        if (scanThread != null)
        {
            scanThread.Join(100);
            scanThread = null;
        }
    }

    private Scanner SetupScanner()
    {
        // for clarity, no error checking here
        Scanner s = Scanner.Connect(new IPAddress(new byte[] { 192, 168, 1, 150 }));
        using (TextReader parametersReader = File.OpenText("param.dat"))
        {
            string parametersString = parametersReader.ReadToEnd();
            s.SetParameters(parametersString, true);
        }
        return s;
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        StopScanning();
    }
```

```
    }
}
```

10 Example 7: Hi-Speed Scanning with Multiple Heads

Example 7 shows three important concepts:

- how to deal with multiple scan heads,
- how to interleave calls for maximum scan speed and
- how to use pulse sync mode.

Using multiple heads is as simple as keeping a connection open (i.e. retaining a Scanner object) for each of them. If the objects are kept in an enumerable collection, it is very convenient to loop over them with a foreach statement. The factory function Scanner.Connect() will give you back an array of Scanners, but be aware that you must check that you connected to as many heads as you requested. In the example code, we skip this check.

If you have connections to lots of scanners (more than six), the total roundtrip for requesting and reading a profile from each one can take a while. Fortunately, there's an easy way to parallelize requests and reads from groups of scanners. The key is to send all the requests at once, and then read all the profiles back at once. This causes all the scanners' processing time and network travel time to happen simultaneously.

In the example, we execute two foreach loops, one for requesting profiles, and one for reads.

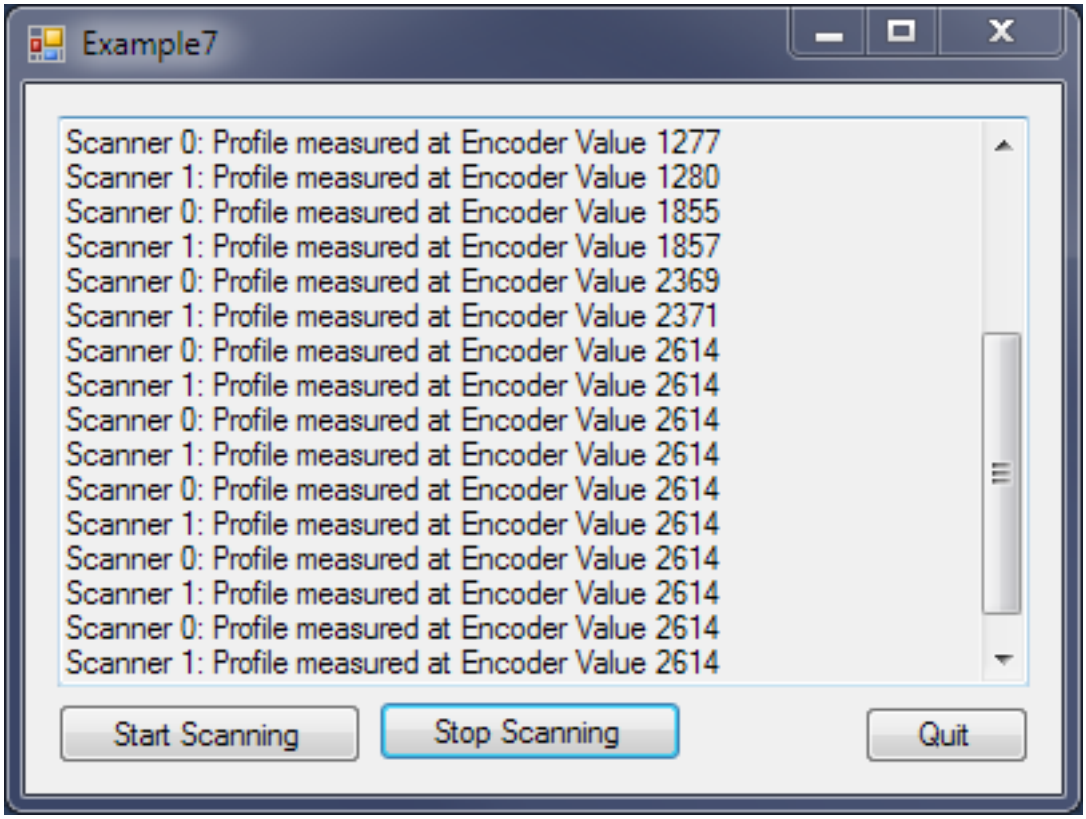
In the code, we also demonstrate how to use Pulse Sync Mode. This mode allows for a master head to drive slave heads. The slave heads wait for a StartIO pulse on the cable. The code designates the first head the master, and the second a slave.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;
using System.Windows.Forms;
using JoeScan.JCamNet;

namespace Examples
{
    public partial class Form1 : Form
    {
        private Thread scanThread;
        private volatile bool isRunning;
        private Scanner[] scanners;

        public Form1()
        {
            InitializeComponent();
            scanners = SetupScanners();
        }

        private void ScanThreadMain()
        {
            foreach (var scanner in scanners)
            {
                // reset encoders
                scanner.SetEncoder(0);
                // EnterPulsSyncMode() will make all heads trigger on receiving a
                // Start Scan pulse, that is emitted by the "master" head.
                scanner.EnterPulseSyncMode();
            }
            // sleep for a millisecond to give all heads to get ready
            Thread.Sleep(1);
            // and tell the "master" head to send out a pulse train every second
            scanners[0].StartPulseMaster(1000 * 1000);
```

```
        // this is called on a thread, so it can't manipulate objects on the UI thread, hence the
        // need to use Invoke()
        this.Invoke(new Action(() => this.textBox.Clear()));
        this.Invoke(new Action(() => this.textBox.AppendText("Scanning Started.\n")));

        while (isRunning)
        {
            // This is the recommended way to parallelize the readout from multiple scanners.
            // first, request profiles
            foreach (var scanner in scanners)
            {
                scanner.BeginGetQueuedProfiles(1);
            }
            // then, read them out
            foreach (var scanner in scanners)
            {
                List<Profile> received = scanner.EndGetQueuedProfiles();
                if (received.Count > 0)
                {
                    // we need to make temp copies because of the way
                    // captured variables work with lambda expressions
                    int tempShort = scanner.CableID;
                    int tempInt = received[0].Location;
                    this.Invoke(new Action(() =>
                        this.textBox.AppendText(String.Format
                        ("Scanner {0}: Profile measured at Encoder Value {1}\n",
                        tempShort, tempInt))));
                }
            }
            // Keep processing time in this loop to a minimum if you want maximum scan speed.
            // Key to a fast system is to only gather the data in this loop, and let another thread
            // (e.g. your main GUI thread) do heavy processing asynchronously. You must use thread
            //  safe methods of exchanging data or interacting with the UI.
        }
        // and make sure every scanner is out of sync mode again
        foreach (var scanner in scanners)
        {
            scanner.ExitSyncMode();
        }
    }

    private void StartButton_Click(object sender, EventArgs e)
    {
        if (scanThread == null || !scanThread.IsAlive)
        {
            scanThread = new Thread(new ThreadStart(this.ScanThreadMain))
            {
                IsBackground = true,
                Name = "ScanThreadMain",
                Priority = ThreadPriority.Highest
            };
            isRunning = true;
            scanThread.Start();
        }
    }

    private void StopButton_Click(object sender, EventArgs e)
    {
        StopScanning();
    }

    private void StopScanning()
    {
        isRunning = false;
        if (scanThread != null)
        {
            scanThread.Join(100);
            scanThread = null;
        }
    }

    private void QuitButton_Click(object sender, EventArgs e)
    {
        StopScanning();
        Close();
    }

    private Scanner[] SetupScanners()
    {
        // for clarity, no error checking here
        Scanner[] scanners = Scanner.Connect(new IPAddress(new byte[] {192, 168, 1, 150}),
            new short[] {0,1});
        using (TextReader parametersReader = File.OpenText("param.dat"))
        {
```

```
                string parametersString = parametersReader.ReadToEnd();
                foreach (var scanner in scanners)
                {
                    scanner.SetParameters(parametersString, true);
                }
            }
            return scanners;
        }
    }
}
```

The running application from Example 7 looks like this:



11 Example 8: Reading Multiple Lasers

Some models of JoeScan Scanners of the JS20 Series (JS-20 X2, JS-20 X3) have more than one laser, each of which can be separately addressed and read out. Example 8 shows how to do that in On-Demand mode using the GetProfile(int laserIndex) function.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using JoeScan.JCamNet;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            Scanner scanner = Scanner.Connect(new IPAddress(new byte[] { 192, 168, 1, 152 }), 0);
            TextReader parametersReader;
            using (parametersReader = File.OpenText("param.dat"))
            {
```

```
                scanner.SetParameters(parametersReader.ReadToEnd(), true);
            }
            // this will only work on an X2 or X3 model!
            if (scanner.LaserCount < 2)
            {
                System.Console.WriteLine("Demo code only for models with multiple lasers.");
                return;
            }
            for (int i = 0; i < scanner.LaserCount; i++)
            {
                Profile p = scanner.GetProfile(i);
                System.Console.WriteLine("Laser {0}: Profile contains {1} points.", i,p.Count);
            }
        }
    }
}
```

Note

Reading out multiple lasers in Synchronized Mode is only supported when configured with the `TimeStaggered-Scanning` keyword.

12 Example 9: Accessing the Camera

Even though not required for measurement purposes, it is possible to read the camera image out of a scan head, for instance, as a setup help. Example 9 does just that and fills a System.Windows.Forms.PictureBox every second with the bitmap obtained from the sensor head. As a shortcut, he laser will actually fire during the exposure as well, so you can verify and troubleshoot the position of the laser line easily.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Net;
using System.Text;
using System.Windows.Forms;
using JoeScan.JCamNet;

namespace Examples
{
    public partial class Form1 : Form
    {
        private Scanner scanner = null;

        public Form1()
        {
            InitializeComponent();
            scanner = Scanner.Connect(new IPAddress(new byte[] { 192, 168, 1, 150 }));
            Timer liveUpdateTimer = new Timer { Interval = 1000 };
            liveUpdateTimer.Tick += (o, args) => { pictureBox.Image = scanner.GetImage().GetBitmap(); };
            liveUpdateTimer.Start();
        }
    }
}
```

13 Namespace Documentation

13.1 Package JoeScan

Packages

- package JCamNet

  Namespace for all API related symbols.

13.2 Package JoeScan.JCamNet

Namespace for all API related symbols.

Classes

- class DiscoveryResponse

  DiscoveryResponse objects contain information reported by scanners on the network when using the discovery functions.
- class ScannerCommunicationException

  Exception thrown when a communication error occurs between the scanner and the client.

- class ScannerOperationException

  Exception thrown when a scanner operation fails.
- struct OldCalibrationValue

  Old position calibration parameters sent to the scanner. More...
- class Profile

  The Profile class contains measured profile data from the scanner.
- struct ProfileDataPoint

  A single data point in a profile suitable for measurement. For efficiency, in JCamNet this is implemented as a struct, not a class, so it follows value-type semantics.
- class RawScan
- struct RawScanPoint

  A data point in a raw (untransformed) scan used for diagnostics only.
- class Scanner

  The Scanner object represents a connection to a JoeScan scanhead accessible through the network.
- class ScannerConfig

  ScannerConfig provides access to the JoeScan scan head detection routines.
- class ScannerImage

  ScannerImage objects represent the camera images from the scanner used primarily for debugging issues with the scanner. The image is a black and white 8 bit image.

Enumerations

- enum ScannerIpSetup

  Enumeration which controls the method used to determine a scanner's IP address when configuring a scanner.
- enum InputFlags

  Flags that indicate the status of the signal lines on the scan head when the scan was taken. StartScan by default is triggered on a falling edge, (see `http://joescan.com/SynchronizedScanningParameters.html#-StartScanTriggerOnHigh`*), so the flag is typically set.*
- enum DataPointStatus

13.2.1 Detailed Description

Namespace for all API related symbols. This namespace is the top level namespace for all symbols in the JCamNet library. Your code should not define any symbols in this namespace.

13.2.2 Class Documentation

13.2.2.1 struct JoeScan::JCamNet::OldCalibrationValue

Old position calibration parameters sent to the scanner.

13.2.3 Enumeration Type Documentation

13.2.3.1 enum DataPointStatus

A status indicator for scans which indicates the brightness of a detected point in a scan.

13.2.3.2 enum InputFlags

Flags that indicate the status of the signal lines on the scan head when the scan was taken. StartScan by default is triggered on a falling edge, (see `http://joescan.com/SynchronizedScanningParameters.html#-StartScanTriggerOnHigh`), so the flag is typically set.

13.2.3.3 enum ScannerIpSetup

Enumeration which controls the method used to determine a scanner's IP address when configuring a scanner.

14 Class Documentation

14.1 DiscoveryResponse Class Reference

DiscoveryResponse objects contain information reported by scanners on the network when using the discovery functions.

Properties

- byte[] MacAddress [get]
  Returns the assigned MAC address for this scanner.
- byte Status [get]
  Currently unused.
- ScannerIpSetup CurrentIpSetup [get]
- IPAddress IPAddress [get]
  The current static or computed IPAddress assigned to the scanner.
- IPAddress BaseIPAddress [get]
  The current static or base IPAddress assigned to the scanner.
- int SerialNumber [get]
- int CableId [get]
  The cable id of the scanner.
- int FirmwareVersion [get]
  The firmware version of the scanner.

14.1.1 Detailed Description

DiscoveryResponse objects contain information reported by scanners on the network when using the discovery functions.

```
The discovery functions <see cref="ScannerConfig.FindAllScanners()" />,   <see cref="ScannerConfig.FindScanner(in
```

ScannerConfig.FindAllScanners(JoeScan.JCamNet.ScannerConfig.ScannerResponseDelegate). will return one DiscoveryResponse per scanner found.

14.1.2 Property Documentation

14.1.2.1 IPAddress BaseIPAddress [get]

The current static or base IPAddress assigned to the scanner.

14.1.2.2 int CableId [get]

The cable id of the scanner.

If static addressing is used, this will always be 0.

14.1.2.3 ScannerIpSetup CurrentIpSetup [get]

Returns the current ScannerIpSetup mode being used by the scanner.

14.1.2.4 int FirmwareVersion [get]

The firmware version of the scanner.

This value is only intended for diagnostic purposes. Your application should not rely on this value.

14.1.2.5 IPAddress IPAddress [get]

The current static or computed IPAddress assigned to the scanner.

If CurrentIpSetup is set to ScannerIpSetup.ByCableId, then this will return BaseIPAddress + CableId, otherwise it will return the same as BaseIPAddress.

14.1.2.6 byte [] MacAddress [get]

Returns the assigned MAC address for this scanner.

See http://en.wikipedia.org/wiki/MAC_address for more details.

14.1.2.7 int SerialNumber [get]

The serial number of the scanner.

14.1.2.8 byte Status [get]

Currently unused.

14.2 Profile Class Reference

The Profile class contains measured profile data from the scanner.

Public Member Functions

- IEnumerator< ProfileDataPoint > GetEnumerator ()

  Get an enumerator over the ProfileDataPoint objects in this profile.

Properties

- int SequenceNumber [get]

  Get the sequence number of the scan. This number is restarted when syncronized scanning is restarted.
- int Location [get]

  Get the encoder value when the scan was taken.
- int SendLocation [get]

  Get the encoder value when the scan was retrieved from the scanner.
- int LaserOnTime [get]

  Get the amount of time the laser was on for this scan. Unit: microseconds.
- InputFlags Inputs [get]
- int TimeInHead [get]

  The time since the scanner was powered on when the scan was retrieved. Unit: milliseconds.
- ScanFlags Flags [get, set]

  The flags attached to this scan.

- int Count [get]

  The number of data points in the profile.
- double Z [get, set]

  Z value of this profile.
- int LaserIndex [get]

  Index of the laser used for this profile. For single-laser models, this is always 0, for multi-laser models, the laser at end farthest away from the camera has the index 0.
- ProfileDataPoint this[int index] [get]

14.2.1 Detailed Description

The Profile class contains measured profile data from the scanner.

The primary data object used by scanners, the Profile class gives access to fully transformed profile data, e.g. points in the coordinate system that this head was calibrated in. The ProfileDataPoint elements in the profile contain the profile points detected from the scanner during an individual scan. The Profile object includes additional information about the scan taken, i.e. encoder data and exposure information.

14.2.2 Member Function Documentation

14.2.2.1 IEnumerator<ProfileDataPoint> GetEnumerator ( )

Get an enumerator over the ProfileDataPoint objects in this profile.

Returns

An enumerator over the ProfileDataPoint objects in this profile.

14.2.3 Property Documentation

14.2.3.1 int Count [get]

The number of data points in the profile.

14.2.3.2 ScanFlags Flags [get, set]

The flags attached to this scan.

Flags generally indicate failures when taking the scan, such as overdriving the scanner. Only if the returned value is ScanFlags.None, the data can be assumed to be valid.

14.2.3.3 InputFlags Inputs [get]

The values of the input lines (Channel A/Channel B/Start Scan) when the scan was taken.

14.2.3.4 int LaserIndex [get]

Index of the laser used for this profile. For single-laser models, this is always 0, for multi-laser models, the laser at end farthest away from the camera has the index 0.

14.2.3.5 int LaserOnTime [get]

Get the amount of time the laser was on for this scan. Unit: microseconds.

14.2.3.6 int Location [get]

Get the encoder value when the scan was taken.

14.2.3.7 int SendLocation [get]

Get the encoder value when the scan was retrieved from the scanner.

14.2.3.8 int SequenceNumber [get]

Get the sequence number of the scan. This number is restarted when syncronized scanning is restarted.

14.2.3.9 ProfileDataPoint this[int index] [get]

Access a ProfileDataPoint by index in the collection.

```
<param name="index"></param>
<returns></returns>
```

14.2.3.10 int TimeInHead [get]

The time since the scanner was powered on when the scan was retrieved. Unit: milliseconds.

14.2.3.11 double Z [get, set]

Z value of this profile.

14.3 ProfileDataPoint Struct Reference

A single data point in a profile suitable for measurement. For efficiency, in JCamNet this is implemented as a struct, not a class, so it follows value-type semantics.

Properties

- double X [get]
  X coordinate.
- double Y [get]
  Y coordinate.
- double Brightness [get]
  Brightness of the detected data point (values range from 0.0 - 1.0)
- double RawBrightness [get]
  Brightness of the detected data point, range not limited to 1.0.
- System.Drawing.Color GrayScale [get]
  Color representing the brightness of the point.

14.3.1 Detailed Description

A single data point in a profile suitable for measurement. For efficiency, in JCamNet this is implemented as a struct, not a class, so it follows value-type semantics.

14.3.2 Property Documentation

14.3.2.1 double Brightness [get]

Brightness of the detected data point (values range from 0.0 - 1.0)

14.3.2.2 System.Drawing.Color GrayScale [get]

Color representing the brightness of the point.

14.3.2.3 double RawBrightness [get]

Brightness of the detected data point, range not limited to 1.0.

14.3.2.4 double X [get]

X coordinate.

14.3.2.5 double Y [get]

Y coordinate.

14.4 RawScan Class Reference

Public Member Functions

- IEnumerator< RawScanPoint > GetEnumerator ()

  Gets an enumerator over all RawScanPoint objects in this RawScan.

Properties

- int Location [get]

  Encoder count when the RawScan was taken.
- int SendLocation [get]

  Encoder count when the RawScan was returned from the scan head.
- int LaserOnTime [get]

  The amount of time the laser was on, in microseconds.
- int TimeInHead [get]

  The number of milliseconds from the time the scan head was started until the scan was requested.
- InputFlags Inputs [get]

  The values on the input lines of the head when the scan was taken.
- ScanFlags Flags [get]

  The warning flags encountered on this scan.
- int NumberOfPoints [get]

  The number of points in this RawScan.
- RawScanPoint this[int index] [get]

  Access a RawScanPoint by index in the collection.

14.4.1 Detailed Description

A RawScan contains raw, untransformed data points detected by the scan head. The data contained in a RawScan is not useful for measurement.

See also

Profile

14.4.2 Member Function Documentation

14.4.2.1 IEnumerator<RawScanPoint> GetEnumerator ( )

Gets an enumerator over all RawScanPoint objects in this `RawScan`.

Returns

An enumerator over all RawScanPoint objects in this `RawScan`.

14.4.3 Property Documentation

14.4.3.1 ScanFlags Flags [get]

The warning flags encountered on this scan.

14.4.3.2 InputFlags Inputs [get]

The values on the input lines of the head when the scan was taken.

14.4.3.3 int LaserOnTime [get]

The amount of time the laser was on, in microseconds.

14.4.3.4 int Location [get]

Encoder count when the `RawScan` was taken.

14.4.3.5 int NumberOfPoints [get]

The number of points in this RawScan.

14.4.3.6 int SendLocation [get]

Encoder count when the `RawScan` was returned from the scan head.

The correct encoder count will be stored in Location. This property should not be used for measurement.

See also

Location

14.4.3.7 RawScanPoint this[int index] [get]

Access a RawScanPoint by index in the collection.

Parameters

index	

Returns

14.4.3.8 int TimeInHead [get]

The number of milliseconds from the time the scan head was started until the scan was requested.

14.5 RawScanPoint Struct Reference

A data point in a raw (untransformed) scan used for diagnostics only.

Properties

- int Data [get]

  The camera coordinate of the detected data point, in 1/100th of a pixel.
- DataPointStatus Status [get]

  The brightness level of the data point.
- int Brightness [get]

  The actual brightness of the data point.

14.5.1 Detailed Description

A data point in a raw (untransformed) scan used for diagnostics only.

Scans and ScanDataPoints are not usable for measurement. Use ProfileDataPoint and Profile for measurement data.

14.5.2 Property Documentation

14.5.2.1 int Brightness [get]

The actual brightness of the data point.

14.5.2.2 int Data [get]

The camera coordinate of the detected data point, in 1/100th of a pixel.

14.5.2.3 DataPointStatus Status [get]

The brightness level of the data point.

14.6 Scanner Class Reference

The Scanner object represents a connection to a JoeScan scanhead accessible through the network.

Inherits IProfileProvider.

Public Member Functions

- string GetParameters ()

  Retrieve the parameters file from the scanner.
- void SetParameters (string contents, bool saveToDisk)

  Send a parameter file to the scanner.
- uint[] GetStatusValues ()

  Get an array of status values supported by this head.
- string GetStatusDescription (int i)

  Get a human readable description for a status value index. See GetStatusValues for more information.
- OldCalibrationValue[] GetOldCalibrationValues (int laserIndex, int numCalibrations)

  Get an array of previous calibration Values from the head.

- void SetOrientation (int laserIndex, double xOffset, double yOffset, double roll)

  Send scanner orientation calibration values to scanner.
- RawScan GetScan ()

  Request a raw, untransformed scan from the scanner with default laser 0.
- RawScan GetScan (uint laserIndex)

  Request a raw, untransformed scan from the scanner.
- ScannerImage GetImage ()

  Request a camera image from the scanner. This can be useful when debugging issues where the scanner is encountering unexpected interference or unable to see the laser due to objects blocking the view of the scanner.
- void GetLaserImage (out ScannerImage image, out RawScan rawScan)

  Retrieve an image of the laser exposure and the corresponding scan.
- Profile GetProfile ()

  Request a profile, including transformed data.
- Profile GetProfile (int laserIndex)

  Request a profile, including transformed data.
- void BeginGetProfile ()

  Send a request to the scanner to start the a profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetProfile() method call or the scanner communications stream will be corrupted.
- void BeginGetProfile (int laserIndex)

  Send a request to the scanner to start the a profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetProfile() method call or the scanner communications stream will be corrupted.
- Profile EndGetProfile ()

  Complete the BeginGetProfile call and retrieve the profile. This method must be proceded by a call to BeginGetProfile() or BeginGetProfile(int)
- void SetEncoder (int encoder)

  Reset the encoder count.
- void EnterEncoderSyncMode ()

  Begin running in encoder synchronized scan mode.
- void EnterPulseSyncMode ()

  Begin running in pulse synchronized scan mode.
- void EnterTimedSyncMode ()

  Begin running in time synchronized scan mode.
- void ExitSyncMode ()

  Exit synchronized scanning mode (both time and encoder modes).
- void StartPulseMaster (int pulseInterval, int pulseCount=0)

  Tell the scanner to generate a pulse train on Start Scan I/O.
- void StopPulses ()

  Stop the pulse train, but the PulseMaster scanner will still drive a constant value to the Start Scan output.
- void SetAlternatingExposure (bool enable, int exposure1, int exposure2)

  Set the alternating exposure parameters. Only works in sync mode. Settings are lost when exiting sync mode and must be resent after re-entering syncmode.
- List< Profile > GetQueuedProfiles ()

  Retrieve the queued profiles from the scanner when running in synchronized scanning mode.
- List< Profile > GetQueuedProfiles (int maxProfiles)

  Retrieve the queued profiles from the scanner when running in synchronized scanning mode.
- void BeginGetQueuedProfiles ()

  Send a request to the scanner to start the profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetQueuedProfiles() method call or the scanner communications stream will be corrupted.

- void BeginGetQueuedProfiles (int maxProfiles)

  Send a request to the scanner to start the profile download. This can be used to parallelize the transfer of data from multiple scanners. This method must be followed by the EndGetQueuedProfiles() method call or the scanner communications stream will be corrupted.
- List< Profile > EndGetQueuedProfiles ()

  Complete the GetQueuedProfiles call and retrieve the profiles. This method must be proceded by a call to BeginGetQueuedProfiles() or BeginGetQueuedProfiles(int)
- void Dispose ()

  Dispose the scanner object.

Static Public Member Functions

- static Scanner Connect (IPAddress ipAddress)

  Connect to a JoeScan scan head at the specified IP address.
- static Scanner Connect (IPAddress baseIpAddress, short cableId)

  Connect to a JoeScan scan head by specifying the base IP address and cable ID.
- static Scanner[] Connect (IPAddress baseIpAddress, params short[] cableIds)

  Connect to a JoeScan scan head by specifying the base IP address and an array of cable IDs.
- static void InitializePerformanceCounters ()

  Install/Reinstall the performance counters for this class library.

Protected Member Functions

- void Dispose (bool disposing)

  Dispose the scanner object.

Properties

- bool SyncMode `[get]`

  Is true if the scan head is in Sync Mode.
- IPAddress IPAddress `[get]`

  Get the IP address used to connect to the scanner.
- int SerialNumber `[get]`

  Scanner serial number.
- Version ServerVersion `[get]`

  Version of the scan server software.
- Version FpgaVersion `[get]`

  The version of the FPGA program on the scanner.
- short CableID `[get]`

  The scanner cable ID.
- double WindowLeft `[get]`
- double WindowRight `[get]`

  Right window bound of the scanner.
- double WindowTop `[get]`

  Top window bound of the scanner.
- double WindowBottom `[get]`

  Bottom window bound of the scanner.
- bool LaserUpstream `[get]`

  Is the laser upstream to the direction of the belt or chain.

- int LaserCount [get]

  The number of lasers in this scanner.
- int SendTimeOut [get, set]

  The TCP timeout for sending data to the scanner.
- int ReceiveTimeOut [get, set]

  The TCP timeout for receiving data from the scanner.
- double InchesPerPulse [get]

14.6.1 Detailed Description

The Scanner object represents a connection to a JoeScan scanhead accessible through the network.

It allows access to the configuration, profile collection, synchronized scanning and debugging functions of an individual scanner.

14.6.2 Member Function Documentation

14.6.2.1 void BeginGetProfile ( )

Send a request to the scanner to start the a profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetProfile() method call or the scanner communications stream will be corrupted.

14.6.2.2 void BeginGetProfile (int *laserIndex*)

Send a request to the scanner to start the a profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetProfile() method call or the scanner communications stream will be corrupted.

Parameters

laserIndex	Laser index to request

14.6.2.3 void BeginGetQueuedProfiles ( )

Send a request to the scanner to start the profile download. This can be used to parallelize the transfer of data from multiple scanners. This API must be followed by the EndGetQueuedProfiles() method call or the scanner communications stream will be corrupted.

Exceptions

ScannerOperationException	Throws if head is not in Sync Mode.

14.6.2.4 void BeginGetQueuedProfiles (int *maxProfiles*)

Send a request to the scanner to start the profile download. This can be used to parallelize the transfer of data from multiple scanners. This method must be followed by the EndGetQueuedProfiles() method call or the scanner communications stream will be corrupted.

Parameters

maxProfiles	Specify the maximum number of profiles to retrieve.

Exceptions

ScannerOperationException	Throws if head is not in Sync Mode.

14.6.2.5 static Scanner Connect (IPAddress *ipAddress*) [static]

Connect to a JoeScan scan head at the specified IP address.

Parameters

ipAddress	The IP address of the scan head to connect.

Returns

A Scanner object connected to the scan head at the specified location.

14.6.2.6 static Scanner Connect (IPAddress *baseIpAddress*, short *cableId*) [static]

Connect to a JoeScan scan head by specifying the base IP address and cable ID.

This method will only work if the scan heads are configured for base+cable ID IP address mode.

Parameters

baseIpAddress	The base IP address.
cableId	The scanner cable ID to connect.

Returns

14.6.2.7 static Scanner [] Connect (IPAddress *baseIpAddress*, params short[] *cableIds*) [static]

Connect to a JoeScan scan head by specifying the base IP address and an array of cable IDs.

```
This method will only work if the scan heads are configured for base+cable ID
IP address mode.

Caller must check number of scanners returned to determine if any connections failed.

<param name="baseIpAddress">The base IP address.</param>
<param name="cableIds">The scanner cable ID to connect.</param>
```

Returns

14.6.2.8 void Dispose ( )

Dispose the scanner object.

14.6.2.9 void Dispose (bool *disposing*) [protected]

Dispose the scanner object.

14.6.2.10 Profile EndGetProfile ( )

Complete the BeginGetProfile call and retrieve the profile. This method must be proceded by a call to BeginGetProfile() or BeginGetProfile(int)

Returns

A profile from the scanner.

14.6.2.11 List<Profile> EndGetQueuedProfiles ( )

Complete the GetQueuedProfiles call and retrieve the profiles. This method must be proceded by a call to BeginGetQueuedProfiles() or BeginGetQueuedProfiles(int)

```
        <returns>A collection of the profiles queued in the scanner at the time of the call
        to BeginGetQueuedProfiles.</returns>
```

/

Exceptions

ScannerOperationException	Throws if head is not in Sync Mode.

14.6.2.12 void EnterEncoderSyncMode ( )

Begin running in encoder synchronized scan mode.

14.6.2.13 void EnterPulseSyncMode ( )

Begin running in pulse synchronized scan mode.

14.6.2.14 void EnterTimedSyncMode ( )

Begin running in time synchronized scan mode.

Time synchronized scan mode should not be used on multiple zone scan systems because internal clock drift will cause the scanners to fall out of time.

14.6.2.15 void ExitSyncMode ( )

Exit synchronized scanning mode (both time and encoder modes).

14.6.2.16 ScannerImage GetImage ( )

Request a camera image from the scanner. This can be useful when debugging issues where the scanner is encountering unexpected interference or unable to see the laser due to objects blocking the view of the scanner.

Returns

A camera image from the scanner.

14.6.2.17 void GetLaserImage (out ScannerImage *image*, out RawScan *rawScan*)

Retrieve an image of the laser exposure and the corresponding scan.

Parameters

image	The image of the scan
rawScan	The scan

14.6.2.18 OldCalibrationValue [] GetOldCalibrationValues (int *laserIndex*, int *numCalibrations*)

Get an array of previous calibration Values from the head.

Parameters

laserIndex	the laser number to query
numCalibrations	the maximum count of old calibrations to return

Returns

An array of OldCalibrationValue structs

Exceptions

ScannerCommunication-Exception	

14.6.2.19 string GetParameters ( )

Retrieve the parameters file from the scanner.

Returns

A string containing the parameters file.

14.6.2.20 Profile GetProfile ( )

Request a profile, including transformed data.

Returns

A transformed profile containing data points suitable for measurement.

14.6.2.21 Profile GetProfile (int *laserIndex*)

Request a profile, including transformed data.

Returns

A transformed profile containing data points suitable for measurement.

14.6.2.22 List<Profile> GetQueuedProfiles ( )

Retrieve the queued profiles from the scanner when running in synchronized scanning mode.

This method is not applicable when the scanner is not running in synchronized scan mode.

Returns

A list of all of the queued profiles from the scanner. (This list may be empty).

Exceptions

ScannerOperationException	Throws if head is not in Sync Mode.

14.6.2.23 List<Profile> GetQueuedProfiles (int *maxProfiles*)

Retrieve the queued profiles from the scanner when running in synchronized scanning mode.

This method is not applicable when the scanner is not running in synchronized scan mode.

Parameters

maxProfiles	maximum number of profiles to retrieve

Returns

A list of the queued profiles from the scanner. (This list may be empty).

Exceptions

ScannerOperationException	Throws if head is not in Sync Mode.

14.6.2.24 RawScan GetScan ( )

Request a raw, untransformed scan from the scanner with default laser 0.

Scans are generally used for debugging purposes and are not useful for measurement. Use GetProfile() for measurement.

Returns

An untransformed scan from the scanner.

14.6.2.25 RawScan GetScan (uint *laserIndex*)

Request a raw, untransformed scan from the scanner.

Scans are generally used for debugging purposes and are not useful for measurement. Use GetProfile(int) for measurement.

Parameters

laserIndex	Laser index to request.

Returns

An untransformed scan from the scanner.

14.6.2.26 string GetStatusDescription (int *i*)

Get a human readable description for a status value index. See GetStatusValues for more information.

Parameters

i	The status value index

Returns

A string with the description

Exceptions

ScannerCommunication-Exception	

14.6.2.27 uint [ ] GetStatusValues ( )

Get an array of status values supported by this head.

```
<returns>
                An array of uints containing the status values. The index of the value can be used
                to get a textual description from <see cref="GetStatusDescription"/>.
</returns>   <exception cref="ScannerCommunicationException">a</exception>   When using GetStatusValu
                use is available, as not all hardware configurations support all status codes.
                Currently, the following status values are supported:
                <list type="table">            <listheader> <term>Index</term><term>Description</te
```

/ 0DSP Version Code [OBSOLETE]0 1Mode: 0-init, 1-idle, 2-expose, 3-scan [OBSOLETE]0 2Host Request: 0-none,1-scan cam, 2-laser scan [OBSOLETE]0 3Current camera exposure time205895 4Current laser on time250 5Current encoder count4294966296 6Optically Isolated inputs415 7Results of last Host Interface communication [OBSOLETE]0 8Encoder value, reset to zero at scan start4294966296 9FPGA Version517 10Cable ID0 11Scan Server Version133295 12Serial Number1465 13Scale Factor. Divide by this to convert to inches.25400 14Temp in 0.001 deg. C1000000

The order of items is guaranteed not to change. The temperature sensor is not available on all hardware and will return the value shown above as a default instead.

14.6.2.28 static void InitializePerformanceCounters ( ) [static]

Install/Reinstall the performance counters for this class library.

14.6.2.29 void SetAlternatingExposure (bool *enable,* int *exposure1,* int *exposure2*)

Set the alternating exposure parameters. Only works in sync mode. Settings are lost when exiting sync mode and must be resent after re-entering syncmode.

Parameters

enable	If true enable the alternating exposure mode, if false return to normal exposure mode
exposure1	One of the two exposure times, must be between 10 and 650000
exposure2	One of the two exposure times, must be between 10 and 650000

14.6.2.30 void SetEncoder (int *encoder*)

Reset the encoder count.

Parameters

encoder	Current encoder count value to use.

14.6.2.31 void SetOrientation (int *laserIndex,* double *xOffset,* double *yOffset,* double *roll*)

Send scanner orientation calibration values to scanner.

/

Parameters

laserIndex	which laser to send the position calibration for
xOffset	The new value of the scanner's x offset.
yOffset	The new value of the scanner's y offset.
roll	The new value of the scanner's rotation.

Exceptions

ScannerCommunication-Exception	

14.6.2.32 void SetParameters (string *contents*, bool *saveToDisk*)

Send a parameter file to the scanner.

Parameters

contents	The parameter file data.
saveToDisk	True to save the parameters to persistent storage.

14.6.2.33 void StartPulseMaster (int *pulseInterval*, int *pulseCount* = 0)

Tell the scanner to generate a pulse train on Start Scan I/O.

Parameters

pulseInterval	Period of the pulse trainin microseconds, must be between 1,000 and 5,000,000
pulseCount	Number of pulses, zero for continuous, must be between 0 and 511

14.6.2.34 void StopPulses ( )

Stop the pulse train, but the PulseMaster scanner will still drive a constant value to the Start Scan output.

14.6.3 Property Documentation

14.6.3.1 short CableID [get]

The scanner cable ID.

This value may not be available in older scanners if you connect to the scanner using only the IP Address.

14.6.3.2 Version FpgaVersion [get]

The version of the FPGA program on the scanner.

14.6.3.3 double InchesPerPulse [get]

Get the InchesPerPulse value (encoder scale used to calculate the profile Z component).

14.6.3.4 IPAddress IPAddress [get]

Get the IP address used to connect to the scanner.

14.6.3.5 int LaserCount [get]

The number of lasers in this scanner.

14.6.3.6 bool LaserUpstream [get]

Is the laser upstream to the direction of the belt or chain.

14.6.3.7 int ReceiveTimeOut [get, set]

The TCP timeout for receiving data from the scanner.

14.6.3.8 int SendTimeOut [get, set]

The TCP timeout for sending data to the scanner.

14.6.3.9 int SerialNumber [get]

Scanner serial number.

14.6.3.10 Version ServerVersion [get]

Version of the scan server software.

14.6.3.11 bool SyncMode [get]

Is true if the scan head is in Sync Mode.

14.6.3.12 double WindowBottom [get]

Bottom window bound of the scanner.

14.6.3.13 double WindowLeft [get]

Left window bound of the scanner

14.6.3.14 double WindowRight [get]

Right window bound of the scanner.

14.6.3.15 double WindowTop [get]

Top window bound of the scanner.

14.7 ScannerCommunicationException Class Reference

Exception thrown when a communication error occurs between the scanner and the client.

Public Member Functions

- ScannerCommunicationException ()
  Construct a ScannerCommunicationException.
- ScannerCommunicationException (string message)
  Construct a ScannerCommunicationException.
- ScannerCommunicationException (string message, Exception innerException)
  Construct a ScannerCommunicationException.

14.7.1 Detailed Description

Exception thrown when a communication error occurs between the scanner and the client.

14.7.2 Constructor & Destructor Documentation

14.7.2.1 ScannerCommunicationException ( )

Construct a ScannerCommunicationException.

14.7.2.2 ScannerCommunicationException (string *message*)

Construct a ScannerCommunicationException.

Parameters

message	The message.

14.7.2.3 ScannerCommunicationException (string *message,* Exception *innerException*)

Construct a ScannerCommunicationException.

Parameters

message	The message.
innerException	An inner exception which in turn caused this exception.

14.8 ScannerConfig Class Reference

ScannerConfig provides access to the JoeScan scan head detection routines.

Public Member Functions

- delegate void ScannerResponseDelegate (DiscoveryResponse response)

  ScannerResponseDelegate provides a callback delegate to be called when a scanner is discovered on the network.

Static Public Member Functions

- static List< DiscoveryResponse > FindAllScanners ()

  Locate all scanners on the local network.
- static DiscoveryResponse FindScanner (int id)

  Locate a specific scanner on the network (by cable ID or serial number).
- static void FindAllScanners (ScannerResponseDelegate srh)

  Locate a specific scanners on the local network using a callback delegate.
- static void SetScannerIpAddress (int idNumber, ScannerIpSetup ipMode, IPAddress ipAddress, IPAddress netMask)

  Set a scanner's IP address configuration.

14.8.1 Detailed Description

ScannerConfig provides access to the JoeScan scan head detection routines.

14.8.2 Member Function Documentation

14.8.2.1 static List<DiscoveryResponse> FindAllScanners ( ) [static]

Locate all scanners on the local network.

Returns

A list of responses describing the scanners found on the network.

14.8.2.2 static void FindAllScanners (ScannerResponseDelegate *srh*) [static]

Locate a specific scanners on the local network using a callback delegate.

Parameters

srh	The callback delegate to call when a scanner responds.

14.8.2.3 static DiscoveryResponse FindScanner (int *id*) [static]

Locate a specific scanner on the network (by cable ID or serial number).

Returns

A list of responses describing the scanners found on the network.

14.8.2.4 delegate void ScannerResponseDelegate (DiscoveryResponse *response*)

ScannerResponseDelegate provides a callback delegate to be called when a scanner is discovered on the network.

Parameters

response	The scanner discovery response.

14.8.2.5 static void SetScannerIpAddress (int *idNumber,* ScannerIpSetup *ipMode,* IPAddress *ipAddress,* IPAddress *netMask*) [static]

Set a scanner's IP address configuration.

Parameters

idNumber	The ID number of the scanner to configure.
ipMode	The IP address selection mode to use.
ipAddress	The Static or Base IP address.
netMask	The IP Netmask for the network.

14.9 ScannerImage Class Reference

ScannerImage objects represent the camera images from the scanner used primarily for debugging issues with the scanner. The image is a black and white 8 bit image.

Public Member Functions

- System.Drawing.Bitmap GetBitmap ()

  Create a Bitmap of this image.
- byte GetPixel (int x, int y)

  Return a single pixel from the image.

Properties

- int ExposureTime [get]

  The exposure time for this image in microseconds.
- int Width [get]

The width of the image in pixels.

- int Height [get]

  The height of the image in pixels.

14.9.1 Detailed Description

ScannerImage objects represent the camera images from the scanner used primarily for debugging issues with the scanner. The image is a black and white 8 bit image.

See also

Scanner.GetImage

14.9.2 Member Function Documentation

14.9.2.1 System.Drawing.Bitmap GetBitmap ( )

Create a Bitmap of this image.

Returns

A black and white System.Drawing.Bitmap containing the image.

14.9.2.2 byte GetPixel (int *x*, int *y*)

Return a single pixel from the image.

Using this method to access the data is not recommended for performance reasons.

Parameters

x	X, sample
y	Y, line

Returns

14.9.3 Property Documentation

14.9.3.1 int ExposureTime [get]

The exposure time for this image in microseconds.

14.9.3.2 int Height [get]

The height of the image in pixels.

14.9.3.3 int Width [get]

The width of the image in pixels.

14.10 ScannerOperationException Class Reference

Exception thrown when a scanner operation fails.

Public Member Functions

- ScannerOperationException ()

  Construct a ScannerOperationException.
- ScannerOperationException (string message)

  Construct a ScannerOperationException.
- ScannerOperationException (string message, Exception innerException)

  Construct a ScannerOperationException.

14.10.1 Detailed Description

Exception thrown when a scanner operation fails.

14.10.2 Constructor & Destructor Documentation

14.10.2.1 ScannerOperationException ( )

Construct a ScannerOperationException.

14.10.2.2 ScannerOperationException (string *message*)

Construct a ScannerOperationException.

Parameters

message	The message.

14.10.2.3 ScannerOperationException (string *message*, Exception *innerException*)

Construct a ScannerOperationException.

Parameters

message	The message.
innerException	An inner exception which in turn caused this exception.