MenloSystems

Remote Control Documentation
-
THz Spectrometers
-
Interface Control Document
(qwebchannel based interface)

Precision in Photonics. Together we shape light.

Menlo Systems, a leading developer and global supplier of instrumentation for high-precision metrology, was founded 2001 as spin-off of the Max-Planck-Institute of Quantum Optics. Known for the Nobel-Prize-winning Optical Frequency Comb technology, the Munich based company offers complete solutions based on ultrafast lasers, synchronization electronics and THz systems for applications in industry and research.

I. IMPRINT

Title:	Remote Control Documentation - Interface Control Document (ICD)
Applicable for spectrometers:	TeraSmart, TERA K15, TERA SYNC, TERA ASOPS
Applicable for software:	ScanControl Software 1.3.0 and newer
Date:	31.10.2019
Author:	Menlo Systems GmbH, Munich, Germany

Changes: Subject to technical changes without prior notice.

II. TABLE OF CONTENTS

1. INTRODUCTION

Menlo System’s THz spectrometers can be controlled from various programming languages either on the local computer or from the network.

These capabilities depend on the hardware and software generation of the systems.

Two different communication protocols are available. **Please check which protocol is supported by your software:**

- **Command-based TCP socket protocol:** For THz systems shipped with TeraScan software prior to 2018 (TeraSmart/TERA SYNC/K15/K8 Mark III/TERA ASOPS), the TeraScan software features an integrated TCP Socket communication interface implemented as TCP socket server that receives commands from Ethernet and returns information based on to these requests. The client sends a command such as “START”, and the server receives this and reacts accordingly. The interface is not event-driven and relies on polling the server continuously. ***This interface is deprecated and no new commands will be implemented for this protocol. It is supported both by the software “Menlo Systems TeraScan” and “Menlo Systems ScanControl”***

- **Event-based QWebChannel:** The “Qt WebChannel” application programming interface enables a peer-to-peer communication between a server and a client (HTML/JavaScript/Python/C++/…) on an object-oriented and event-driven basis. This interface has the nature of an API (Application Programming Interface) and works also over Ethernet. This enables the user to control the spectrometer from remote in a way as if it was local. For an easy usage and customer convenience, and for communication in e.g. LabView, there is an easy-to-understand .NET library implementation of the interface. ***This protocol is only supported by the software “Menlo Systems ScanControl”.***

Note: This documentation covers only the new Qt WebChannel interface. This is the interface recommended for ScanControl users. For users willing to use the command-based TCP/IP interface, please refer to the corresponding documentation.

2. ABBREVIATIONS AND NAMES

Abbrev./Names	Description
DHCP	Dynamic Host Configuration Protocol The near-universal way of assigning IP addresses dynamically
DNS	Domain Name System The near-universal way of translating human-readable computer names into IP addresses.
IP	Internet Protocol A very common network protocol, basis of the internet.
NA	Not applicable
TCP	Transmission Control Protocol A network protocol providing reliability, connection-orientation in an IP network.
US-ASCII	United States American Standard Code for Information Interchange Here: used to refer to the 7-bit character-set commonly known under that name.
USB	Universal Serial Bus
ODU	Optical Delay Unit
API	Application Programming Interface
QT	Widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various platforms such as Linux, Windows, macOS, Android or embedded systems.
QML	Qt Modeling Language: a user interface markup language. declarative language for user interface–centric applications.
SUBVI	LabVIEW files used to encapsulate a particular subroutine. It is called in a main LabVIEW file as a sub unit, in analogy to text-based languages.

3. NETWORK REQUIREMENTS

When using server and client on separate PCs, both PCs must be connected to a 10/100Base-TX Ethernet network using a standard 8P8C modular connector. Communication happens via TCP/IP, specifically IPv4. If server and client software are running on the same PC, no network is required.

The user has the option of providing a DHCP server to assign the measurement PC a dynamic IP address. In this case the customer must also provide a DNS server that translates a fixed name into the dynamically assigned IP address. Alternatively, the PC can be configured to a static IP address, which requires no extra infrastructure on the customer side. The external IP address of the PC can be found using the `ipconfig` command in the Windows command line.

NOTE: **Make sure that the measurement computer and the connecting computer have a low delay connection with high available data rate, preferably in the same subnet. Ask your system administrator about routing options and if you have problems connecting.**

4. QWEBCHANNEL INTERFACE

4.1 QWEBCHANNEL: AN OBJECT-ORIENTED AND EVENT - DRIVEN INTERFACE

Note: This interface is only supported by the software "Menlo Systems ScanControl".

The "Qt WebChannel", or QWebChannel, is an application programming interface which enables a peer-to-peer communication between a server and a client application written in HTML/JavaScript/Python/C++/… on an object-oriented and event-driven basis. This interface has the nature of an API (Application Programming Interface). This enables the user to communicate between programs on the same or on different PCs, and, for example, to control the spectrometer from remote in a way as if it was local.

The API is part of the Qt application framework, which is extensively used by the ScanControl.

1. Object-oriented: ScanControl is represented as a class. The public methods of the class allow the remote controlling of ScanControl
2. Event-driven: signals triggered by events are defined. Appropriate event handlers are connected to the signals in order to deal with the event that triggered the signal. In such a modern paradigm the flow of the program is determined by event and actions, and not by the order of the instructions.

For more information about Qt and QWebChannel, see the Qt documentation: https://doc.qt.io/.

To use this API, a communication layer has to be established between a client and the server (ScanControl in this case).

The resulting architecture is schematized in Fig 1.

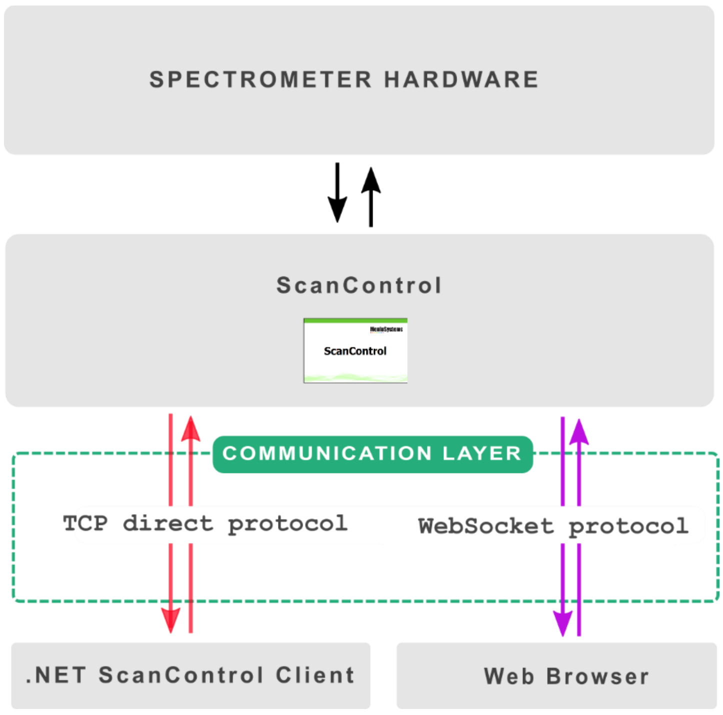


Figure 1: Complete hardware and software architecture of the Spectrometer.

As shown in Fig 1, the communication layer offers two complementary protocols to connect:

- WebSocket protocol at default port 8002.
- Direct TCP Socket protocol at default port 8003

The same commands are available for both protocols. To remotely control the instrument, one of the two protocols need to be implemented, to send commands to the system by calling the methods that are provided.

WebSocket protocol is preferred for Python, and the direct TCP protocol for C#.

4.2 SHORT DESCRIPTION OF THE PROTOCOLS

Direct TCP protocol: TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network.

WebSocket protocol: The WebSocket Protocol enables two-way communication between a client running a code in a controlled environment to a remote host that has opted-in to communications from that code. The protocol consists of an opening handshake followed by basic message framing, layered over TCP.

The two protocols are used to exchange message at the communication layer level. The background remoting architecture relies on Qt WebChannel.

Qt WebChannel comes with native support for JavaScript, C++ and QML.

Menlo provides a .NET, a C++ and a python implementation of the Qt WebChannel protocol. Those implementations are open source. The source code is available for download at:

https://github.com/MenloSystems

Separate sections are provided below for the individual programming languages. In the following we will often use the abbreviation qwebchannel when referring to Qt WebChannel.

4.3 CONNECTION SETUP AND GENERAL INFOS

Make sure that no firewall blocks the ethernet connection of the software and that the ports set are available on the system.

Start the software ScanControl on the THz computer. At its startup, the ScanControl software provides two servers, a WebSocket server and a TCP Socket server.

The TCP socket server is started at the TCP port 8003.

A successful startup of the TCP socket server will lead to the following entry in the ScanControl.log file (available from the software menu "Help->Show log…":

```
(remoting.webChannel): Setting up WebChannel TcpSocket server on port 8003
to listen on all interfaces
```

If an error occurs during startup the Windows sockets error code will be logged to the file `ScanControl.log`.

The WebSocket socket server: is started at the port 8002.

A successful startup of the WebSocket socket server will lead to the following entry in the ScanControl.log file (available from the software menu "Help->Show log…":

```
(remoting.webChannel): Setting up WebChannel WebSocket server on port 8002
to listen on all interfaces
```

If an error occurs during startup the Windows sockets error code will be logged to the file `ScanControl.log`.

Note: ScanControl can handle multiple connections from any network user and does not have security features. We recommend to set up a firewall if needed. The firewall should not prevent the connection between THz computer and client computer, if they are different machines.

Note that no virtual terminal is provided, i.e. no character echo or support for editing is available in the log viewer.

4.3.1REPRESENTATION OF SCANCONTROL IN QWEBCHANNEL

ScanControl is represented as a state machine in the QWebChannel based remote interface. The corresponding possible states are listed in the following table:

State Name	Value	
`Uninitialized`	`0`	Software is not yet connected to devices
`Initializing`	`1`	Software attempts to connect to devices
`Idle`	`2`	Not doing anything. Waiting for user input.
`Acquiring`	`3`	Software is in measurement mode (active).
`Busy`	`4`	System is busy. This can mean that e.g. something is moving to a start position, or waiting for a trigger (ASOPS)…
`Error`	`5`	An error occurred. No measurement possible.

4.3.2INFORMATION EXCHANGE IN QWEBCHANNEL: PULSEFLAG

In the QWebChannel interface some information, in form of flag descriptors, are attached to the data cube of each pulse. Such information is enclosed in the `enum PulseFlags:`

Key	Hex Value	Binary Value	Description
`NoPulseFlags`	`0x0`	`00000000`	
`FlippedPulse`	`0x1`	`00000001`	Bit 0 tells if the pulse was flipped (time inverted)
`Trigger1`	`0x10`	`00010000`	Bit 4 tells if trigger 1 was activated
`Trigger2`	`0x20`	`00100000`	Bit 5 tells if trigger 1 was activated
`Trigger3`	`0x40`	`01000000`	Bit 6 tells if trigger 1 was activated
`Trigger4`	`0x80`	`10000000`	Bit 7 tells if trigger 1 was activated
`TriggerMask`	`0xF0`	`11110000`	Mask for filtering

4.3.3GETTING MEASUREMENT DATA

ScanControl uses event handlers based on signals to process measurement data. There is no direct command to get the current pulse information. To get data, one must connect to such signals (see later sections). See e.g. https://en.wikipedia.org/wiki/Signals_and_slots for details on the concept.

For a list of the properties supported by the ScanControl state machine, see later sections (from pag. 9).

5. PROGRAMMING TUTORIAL

5.1 INTRODUCTION

The QWebChannel interface allows communication between ScanControl and client applications written in a variety of programming languages. The interface is thus extremely general, and can be easily adapted to different language families.

In this tutorial we provide a detailed implementation of the QWebChannel remote interface of ScanControl in three languages:

.NET programming Languages: Languages belonging to the Common Language Infrastructure (CLI). These includes but are not limited to: C#, C++. As an example we use C#. Menlo systems provides a .NET implementation “webchannel.net” of QWebChannel interface, communicating via the TCP socket protocol.

Python 3: Menlo provides a python 3 implementation “PyWebChannel” of QWebChannel interface communicating via the WebSocket protocol.

LabVIEW: For communication in LabView, the .NET implementation is packed in a library `.dll` file that can be loaded in the LabVIEW programming environment. A similar approach can be translated to other environments such as MATLAB.

5.2 .NET PROGRAMMING LANGUAGES

Menlo Systems provides a .NET implementation “webchannel.net” of QWebChannel interface for the TCP socket protocol. The library is provided as a file `qwebchannel.dll` which can be loaded by any programming environment that support .NET. Please contact Menlo Systems to get that library. If you are unsure how to use this programming environment, skip to the following subsection and go directly for an introduction to simplified usage of the programming interface with included examples.

5.2.1.NET SCANCONTROL CLIENT

This chapter refers to the library `scancontrolclient.dll`. It serves as a client program (written in C#) to communicate with ScanControl in an easy way, for users with limited programming skills. The functionality is enclosed in the namespace `MenloSystems`.

The library uses the QWebChannel implementation `qwebchannel.dll` to build a client wrapper. This means that, for the proper functioning of `scancontrolclient.dll` the file `qwebchannel.dll` needs to be available, as well as the file `Newtonsoft.Json.dll`.

The .NET ScanControl client library `scancontrolclient.dll` offers the following properties, classes, and methods:

Enum ScanControlStatus

The `enum` structure `ScanControlStatus` represents the states of the state machine of ScanControl. See the table above in 4.3.1 for the keys and values and their meaning.

Enum PulseFlags

The `enum` structure`PulseFlags` implements the pulse flags properties (see 4.3.2) as C# enum object.

Class PulseReadyEventArgs

This class is derived from `System.EventArgs`, and it offers the full information regarding a measured pulse. All the measured pulses are provided in a format corresponding to this structure. The structure contains the properties in the following table. The columns “Get” and “Set” indicate if such properties are readable (Get) and/or writable (Set).

Property	Description	Get	Set
`double[] TimeAxis`	Timeaxis as double array	☑	☑
`List<double[]> Amplitudes`	Pulse data as double array	☑	☑
`long Timestamp`	Timestamp of the measurement. Equal to system time of the computer running ScanControl at the moment of processing the pulse in memory (does not correspond to the exact sampling timestamp)	☑	☑
`PulseFlags Flags`	Pulseflag-Information, see `Enum Pulseflags` definition above	☑	☑

Note: The TeraSmart and TERA K15 systems currently only support one pulse trigger. Please contact Menlo Systems about how to activate this function. Trigger 2-4 are supported by software for future releases.

Class ScanControl

This is the main client instance of ScanControl. It contains the necessary properties and functionality, such as methods and event handlers to remotely control ScanControl.

Properties of the class ScanControl

Property	Description	Get	Set
`bool IsConnected`	True if connected to ScanControl	☑	
`double Begin`	Begin value/start value for measurement in ps	☑	☑
`double End`	End value for the measurement in ps	☑	☑
`uint DesiredAverages`	Wanted averages, “No. of averages” in ScanControl	☑	☑
`uint CurrentAverages`	Currently finished averages	☑	
`ScanControlStatus Status`	Current status of ScanControl, see section 4.3.1	☑	

To set the settable properties, simply assign a new value to the variables. The backend will make sure they’re communicated to ScanControl.

Methods of the class ScanControl

Property	Description
`void Start()`	Start the measurement
`void Stop()`	Stop the measurement
`void ResetAveraging()`	Resets/Restarts the averaging by deleting the averaging buffer, so that averaging starts again from 0%. Results in a value for CurrentAverages = 0
`static double[] DecodeDoubleArray(string b64array)`	Decodes an b64-encoded string at returns it as double array. Pulse data is usually sent in b64 encoding, so this function is used to decode the data.
`void Disconnect()`	Disconnects the client from the server.
`void Connect()`	Connects to ScanControl using default values (to localhost at port 8003). Does not wait for the connection to be established. For a function that waits for this use BlockingConnect()
`void Connect(string host)`	Connects to ScanControl using a specified host and default port (port 8003)
`void Connect(string host, int port)`	Connects to ScanControl using a specified host and port
`void BlockingConnect()`	Connects to ScanControl using default values (to localhost at port 8003). The method finishes only after the connection is successfully established
`void BlockingConnect(string host)`	Connects to ScanControl using a specified host and default port (port 8003). The method finishes only after the connection is successfully established
`void BlockingConnect(string host, int port)`	Connects to ScanControl using a specified host and port. The method finishes only after the connection is successfully established

Event Handlers

Property	Description
`OnPulseReady(object sender, PulseReadyEventArgs e)`	Is triggered whenever a pulse is received from the program. Gives the sender object (ScanControl class instance) and the pulse information including timeaxis (see `PulseReadyEventArgs` properties above)
`OnDisplayPulseReady(object sender, PulseReadyEventArgs e)`	Is triggered whenever a pulse is received from the program, that is meant for display. The rate is 20 Hz at maximum. Gives the sender object (ScanControl class instance) and the pulse information including timeaxis (see `PulseReadyEventArgs` properties above) ➔Use this event handler if the data is only used for displaying it to the user. It is not needed to get all the pulses if they’re at a very high rate just for displaying them e.g. for alignment or bandwidth check.
`OnConnected(object sender, Eventargs e)`	Is triggered when the connection is established. Doesn’t provide any further information.
`OnDisconnected(object sender, Eventargs e)`	Is triggered when the connection is lost. Doesn’t provide any further information.
`OnBeginChanged(object sender, ValueChangedEventArgs<double>)`	Is triggered when the begin/start value changed. Also delivers the new begin value in ps through the `ValueChangedEventArgs` class as `double Value` (get and set possible).
`OnEndChanged(object sender, ValueChangedEventArgs<double>)`	Is triggered when the end value changed. Also delivers the new end value through the `ValueChangedEventArgs` class as `double Value` (get and set possible).

`OnDesiredAveragesChanged(object sender, ValueChangedEventArgs<uint>)`	Is triggered when the value for desired averages changes. Also delivers the new value through the `ValueChangedEventArgs` class as `uint` **`Value`** (get and set possible).
`OnCurrentAveragesChanged(object sender, ValueChangedEventArgs<uint>)`	Is triggered when the number of (actually measured) averages changes. Also delivers the new value through the `ValueChangedEventArgs` class as `uint` **`Value`** (get and set possible). ➔Using this handler only makes sense if the measurement speed is very low (>> 1 s), because the averages will change for every pulse measured
`OnStatusChanged(object sender, ValueChangedEventArgs<ScanControl Status>)`	Is triggered when the status of ScanControl changes. Also delivers the new value through the `ValueChangedEventArgs` class as **ScanControlStatus Value** (see `Enum ScanControlStatus` above)

5.2.2C# .NET SCANCONTROL CLIENT TUTORIAL

As presented above, the file `scancontrolclient.dll` provides a ready and easy-to-use library for remotely controlling the ScanControl software. The following files need to be available in the compiler environment for the library to work:

- `Newtonsoft.Json.dll`
- `qwebchannel.dll`

The first example shows the usage of the library directly in C# .NET. By importing the `MenloSystems` namespace, a `ScanControl`-Object is available, which lets us send commands and receive data from the **running ScanControl program** on the **same computer** (localhost).

The first example (“example1.cs”) demonstrates how to start the measurement.

Example 1 (example1.cs):

```
using MenloSystems;

public class HelloScanControlClass
{
        public static void Main(string[] args)
        {
            ScanControl sc = new ScanControl();
            sc.BlockingConnect();
            sc.Start();
        }
}
```

First the `ScanControl` object is created, then the connection is established. `BlockingConnect()` waits for the connection to be established before the program continues. The function `Start()` then simply starts the measurement.

In order to compile the file, the following command can be used (valid for Windows 7 and .net Framework 3.5:

```
%Path of .NET Framework Installation%/csc.exe -out:"example1.exe" example1.cs -r:"Newtonsoft.Json.dll" -r:"qwebchannel.dll" -r:"scancontrolclient.dll"
```

Where %Path of .NET Framework Installation% is the path of the .NET framework installed on the PC.

Notes:

- For more sophisticated programs, the function `Connect()` enables the experienced user to keep the program responsive while the connection is established.
- If ScanControl runs on a remote computer, the `BlockingConnect()` and `Connect()` commands accept specifying an IP-address and optionally a port:

```
sc.BlockingConnect(″127.0.0.1″);
sc.BlockingConnect(″127.0.0.1″,8003);
```

In the ScanControl logfile (available through “Help->Show Log”), the following lines will be written:

```
[DATE TIME I] (remoting.webChannel): New WebChannel connection: QHostAddress("::ffff:127.0.0.1") port: 5664 name: ""
[DATE TIME I] (remoting.webChannel): WebChannel connection lost: QHostAddress("::ffff:127.0.0.1") port: 5664 name: ""
```

The first line is printed when the `BlockingConnect()`-command is received, the second line appears when the program terminates, as this automatically closes the socket connection. If you have troubles with the connection, look into the log file to see if connections get interrupted.

Note: Do not open too many parallel connections. There is no need to have more than one connection, and this will lead to higher memory requirements and more traffic.

Now if we want to receive data, we need to use event handlers. The following is a simple example of how to work with the `OnPulseReady` event handler to acquire data.

Example 2 (example2.cs):

```
using System;
using System.Threading;

using MenloSystems;

public class MainClass
{
    static void UpdateInfo(object sender, PulseReadyEventArgs e)
    {
        ScanControl sc = (ScanControl) sender;

        Console.Write(string.Format("\rTimestamp: {0}, Averages: {1}/{2}", e.Timestamp, sc.CurrentAverages, sc.DesiredAverages));
    }

    public static void Main(string[] args)
    {
        ScanControl sc = new ScanControl();
        sc.BlockingConnect();

        sc.OnPulseReady += UpdateInfo;
        sc.OnDisconnected += (sender, e) => Console.Error.WriteLine("\nLost con
nection");

        sc.Start();
        Thread.Sleep(5000);

        sc.Stop();

        Thread.Sleep(100);
    }
}
```

Before sending the command `Start()` an event handler is defined for the event `OnPulseReady`. The `UpdateInfo` function is requested to be called when a pulse was measured. This function receives the measurement data as `PulseReadyEventArgs` object, and in addition the `ScanControl` object. This way, even more information than just the pulse data is available: the current averages and desired averages can be extracted from the ScanControl object. The function is only called when new data is measured, so no polling for data is needed.

Moreover, to handle cases where the connection is lost, an event handler for the event `OnDisconnected` is added in-line, which communicated the lost connection to the console.

After defining the event handlers, the measurement is started, and the main thread is halted for 5 seconds. The event handler is called anyway.

The program produces the following output:

```
Timestamp: 1541597133499, Averages: 1/1
```

The timestamp changes for 5 seconds until the program terminates.

In case of a lost connection during the 5 seconds of sleep time, the output is as follows:

```
Timestamp: 1541598064369, Averages: 1/1
Lost connection
```

Next, a bigger example is given, where a graphical user interface (GUI) is used to display time-domain data.

Example 3 (example3.cs):

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;
using System.Threading;
using System.Runtime.InteropServices;

using MenloSystems;

public class MainForm : Form
{
    ScanControl scanControl;
    Series dataSeries;
    object sync = new object();

    void GotPulse(object sender, PulseReadyEventArgs e)
    {
        lock (e) {
            var pulse = e.Amplitudes[0];
            var timeaxis = e.TimeAxis;
            // Plotting must be done in main thread/same thread where plot was created
            // this can be done using an invocation
            Invoke((Action) (() => plotNewPulse(timeaxis, pulse)));
        }
    }

    public MainForm(ScanControl sc)
    {
        var chart = new Chart();

        Height = 480;
        Width = 640;

        Controls.Add(chart);

        chart.Location = new Point(15, 15);
        chart.Width = Width - 50;
        chart.Height = Height - 110;
        chart.Anchor = AnchorStyles.Left | AnchorStyles.Top | AnchorStyles.Right | AnchorStyles.Bottom;

        var startBtn = new Button { Text = "Start" };
        startBtn.Location = new Point(15, chart.Top + chart.Height + 15);
        startBtn.Anchor = AnchorStyles.Left | AnchorStyles.Bottom;
        Controls.Add(startBtn);

        var stopBtn = new Button { Text = "Stop" };
        stopBtn.Location = new Point(startBtn.Left + startBtn.Width + 15, chart.Top + chart.Height + 15);
        stopBtn.Anchor = AnchorStyles.Left | AnchorStyles.Bottom;
        Controls.Add(stopBtn);

        chart.ChartAreas.Add(new ChartArea {
            Name = "Area"
        });

        // delete previous data
        chart.Series.Clear();

        dataSeries = new Series {
            Name = "Data",
            ChartType = SeriesChartType.Line
        };

        // add new graph
        chart.Series.Add(dataSeries);

        this.scanControl = sc;

        // Connect actions
        startBtn.Click += delegate(object sender, EventArgs e) { sc.Start(); };

        stopBtn.Click += delegate(object sender, EventArgs e) { sc.Stop(); };

        // Connect and init data
        scanControl.OnDisplayPulseReady += GotPulse;
    }

    void plotNewPulse(double[] timeaxis, double[] pulse)
    {

        lock (pulse) {
            if (timeaxis.Length != pulse.Length) {
                return;
            }
            dataSeries.Points.DataBindXY(timeaxis, pulse);
        }
    }


    [STAThread]
    public static void Main(string[] args)
    {
        ScanControl sc = new ScanControl();
        sc.BlockingConnect();

        Application.EnableVisualStyles();
        Application.Run(new MainForm(sc));

        sc.Disconnect();
    }
}
```

The program mainly consists of a class that is derived from the Windows `Form` object. As members, there is the `ScanControl` object, and a `Series` object that holds the data.

In the program's main function, first the connection is established and then the `Application` object is used to call the method `MainForm`, which builds up the GUI. The `ScanControl` object `sc` is handed so that the connection can be used to get the data inside the chart.

The key behavior is implemented by sending `start` and `stop` commands using `Click` event handlers for the buttons, and by setting the function `GotPulse` to be called when new display data is available. Inside this function, the first pulse data line (Channel 1) and the timeaxis are extraced, and then the plotting function is called.

This needs to be done using an invoke due to thread safety. For details about Threads in C# and Charts, see the .NET documentation.

5.3 PYTHON PROGRAMMING LANGUAGE

Menlo provides a python 3 implementation “PyWebChannel”, or “pywebchannel” in the following of the QWebChannel interface for the websocket socket protocol. The newest version is always available under:

https://github.com/MenloSystems/pywebchannel

From Python 3.4 onwards, this module has no dependencies. In Python < 3.4 you need the backport enum34 package. The `pywebchannel.async` submodule provides an `asyncio` compatibility layer (Python 3.5+).

A simple, newline-delimited raw TCP/IP Transport and Protocol for use with `asyncio` is provided in `pywebchannel.asyncio`.

5.3.1PYTHON SCANCONTROL CLIENT

This section refers to the file `scancontrolclient.py`. It functions as a client program to communicate with ScanControl in an easy way, for users with limited programming skills.

The library `scancontrolclient.py` uses the package pywebchannel, and establishes the connection to ScanControl. This means that the package pywebchannel needs to be available. Please contact Menlo systems to obtain the file `scancontrolclient.py.` After a connection is established, the functions and properties of ScanControl are created in a member object of the class ScanControlClient, which is the main class defined in the file `scancontrolclient.py.`

Note: The file scancontrol.py makes extensive use of the module asyncio for asynchronous programming in Python. Please refer to the Python documentation for a reference on the features and functionalities of the module, such as the extensively used concept of event loop.

The file `scancontrolclient.py` offers the classes, properties and methods described in the following.

Class ScanControlClient

This is a helper class to establish connections to ScanControl through QWebChannel. It uses the asyncio event loops to send/receive information.

Methods

Method	Description
`ScanControlClient(loop = None)`	Constructor class that takes an `asyncio` event loop as argument and returns the client instance. If no event loop is proved, it creates one and saves it in its `loop` property. This event loop (either provided or created inside the constructor) then needs to be run in order to get the connection going.
`connect(host = ”localhost”, port = “8002”)`	Establishes the connection to the webchannel server. Host and port are optional (default values given). After the connection is established, the member variable `scancontrol` is available.
`void run()`	Runs a method in the event loop (see property loop below)

Properties

Property	Description
`loop`	`Asyncio` event loop that is used for the communication. No communication is happening if the loop isn't run.
`scancontrol`	Provides the instance of ScanControl. This property is not set before a successful connection is established.

Enum class ScanControlStatus

This `enum.IntEnum` structure `ScanControlStatus` represents the states of the state machine of ScanControl. See the table above in 4.3.1 for the keys and values and their meaning.

Enum class PulseFlags

This enum class represents a pulse flags implementation (see section 4.3.2).

class QWebChannelWebSocketProtocol

Protocol for communicating with webchannel hosts using websocket in python.

Derived from `websockets.client.WebSocketClientProtocol`.

Bridges WebSocketClientProtocol and QWebChannel.

Continuously reads messages in a task and invokes QWebChannel.message_received() for each. Calls QWebChannel.connection_open() when connected.

Also patches QWebChannel.send() to run the websocket's send() in a task.

This class is for internal use of scancontrolclient.py and is not further described here. It may, however, be of help for custom client implementations for skilled programmers.

5.3.2SCANCONTROL OBJECT

To access the ScanControl program functionality, first a connection must be established. Then the object properties and methods are dynamically created in the `scancontrol` member object and are accessible. The object `scancontrol` contains the necessary methods, properties and signal for the actual interaction of Python with ScanControl. Such methods, properties and Signal are described in the following.

Methods of the object `scancontrol`

Since the commands must be executed on a remote program, all functions need to be executed by the internal event loop of the client. More details on this on the description of the first example, example1.py.

Currently (ScanControl Version 1.3.3), the following methods are available:

Method	Description
`start()`	Starts the measurement.
`stop()`	Stops the measurement
`reset()`	Resets the hardware configuration. This leads to a reinitialization of the ODU for TeraSmart/Tera K15 systems.
`resetAveraging()`	Resets/Restarts the averaging by deleting the averaging buffer, so that averaging starts again from 0%. Results in a value for `CurrentAverages = 0`
`setBegin(double begin)`	Sets a new begin/start value for the measurement in ps
`setEnd(double end)`	Sets a new end value for the measurement in ps
`setRange(double range)`	Sets a new end value that equals begin + **`range`**
`setDesiredAverages(int avg)`	Wanted averages, "No. of averages" in ScanControl "Measurement setup" menu.
`setRate(double rate)`	Sets a new measurement speed.

Note: When the data type, or the number of arguments is wrong, a message will be displayed in the ScanControl logfile.

e.g. for setBegin(1,1), the following message appears:

```
[2018/11/09 17:51:37.477 W] (default): Ignoring additional arguments while
invoking method "setBegin" on object
Remoting::WebChannelReadyAdaptor(0x3b63100) : 2 arguments given, but method
only takes 1
```

Properties of the object `scancontrol`

The object properties are all of the get type. Setting them does not change the value in ScanControl. In order to change them, methods must be used instead.

Property	Description
`float begin`	Begin/start value for the measurement in ps
`float end`	End value for the measurement in ps
`uint desiredAverages`	Wanted averages, "No. of averages" in ScanControl
`uint currentAverages`	Currently finished averages
`float range`	Scan range, equal to **`end - begin`**
`float rate`	Current measurement speed
`uint status`	Current status of ScanControl, see section 4.3.1
`str timeAxis`	Time axis of the measurement as string that is base64 encoded

Signals (Event Handling)

The ScanControl object offers signals that are suited for event handling in Qt style. See the documentation at https://doc.qt.io/qt-5/signalsandslots.html for more details on Qt signals.

Signals of the object `scancontrol`

Signal Name	Arguments passed to callback	Description
`beginChanged`	`float newBegin`	Is triggered when the begin/start value changed and gives the new begin value
`endChanged`	`float newEnd`	Is triggered when the end value changed and gives the new end value
`rangeChanged`	`float newRange`	Is triggered when the begin or end value changes and gives the new range (range = end - begin)
`timeAxisChanged`	`float newTimeAxis`	Is triggered when the time axis changed. Gives the new timeaxis as (`double array`) as encoded base64 string (see description of timeaxis property above)
`displayPulseReady`	`dict data`	Is triggered whenever a pulse is received from the program that is meant for display/preview. The rate is 20 Hz at maximum. Gives the pulse information as **`dict`** structure, including • Amplitude (pulse) information of channel 1 and 2 as `double data['amplitude'][0-1]` • timeaxis: `double data['timeaxis']` • pulse flags as `int data['flags']` • timestamp as `int data['timestamp']` ***Note: Use this event handler if the data is only used for displaying it to the user. It is not needed to get all the pulses if they're at a very high rate just for displaying them e.g. for alignment or bandwidth check.***
`pulseReady`	`dict data`	Is triggered whenever a pulse is received from the program. Gives the pulse information as **`dict`** structure, including • Amplitude (pulse) information of channel 1 and 2 as `double data['amplitude'][0-1]` • timeaxis: `double data['timeaxis']` • pulse flags as `int data['flags']` • timestamp as `int data['timestamp']` ***Note: Use this event handler if you want to receive all data. If you want to preview a measurement, we recommend to use `displayPulseReady`***
`desiredAveragesChanged`	`float newAvg`	Is triggered when the value for desired averages changes.
`currentAveragesChanged`	`float newAvg`	Is triggered when the number of (really measured) averages changes. Also delivers the new value through the `ValueChangedEventArgs` class as `uint Value` (get and set possible). ➔Using this handler only makes sense if the measurement speed is very low (>> 1 s), because the averages will change for every pulse measured
`statusChanged`	`int newStatus`	Is triggered when the status of ScanControl changes. Gives the status as `int newStatus`.
`rateChanged`	`float newRange`	Is triggered when the scan rate changes. Gives the new range as `float newRange.`

`pulseReadyEncoded`	**`dict data`**	Similar to `pulseReady` but the pulse data is not decoded, and no timeaxis is given. The **`dict`** structure therefore includes • Amplitude (pulse) information of channel 1 and 2 as base 64 encoded `str data['amplitude'][0-1]` • pulse flags as `int data['flags']` • timestamp as `int data['timestamp']`
`displayPulseReadyEncoded`	**`dict data`**	Similar to `displayPulseReady`, but the pulse data is not decoded, and no timeaxis is given. For the dict structure, see `pulseReadyEncoded`.
`destroyed`		Internal signal for qwebchannel; triggered when a qwebchannel object is destroyed; does not matter in this context
`objectNameChanged`		Internal signal for qwebchannel; triggered when a qwebchannel object changes its name; does not apply in this context

In the event based paradigm, events trigger the generation of the signal in the previous table.To handle events, simply define a function as callback that receives the passed arguments and connect it to the signal, as in the Python example collection in the following.

Note: Decoding encoded data

Decoding of pulse/timeaxis data is possible by e.g. using the `base64 package (import base64)`

```
import base64
base64.b64decode(timeAxis)
```

The resulting string can then be converted to a `numpy` array

```
import numpy
axis = numpy.frombuffer(base64.b64decode(timeAxis), dtype=np.float64)
```

Note: The timeaxis is always transferred with the signals/event handlers `pulseReady` und `displayPulseReady` (see above). There is no need to decode the data yourself.

5.3.3PYTHON TUTORIAL

By importing the `ScanControlClient` class from scancontrolclient.py, a client program is available, which offers an interface for remotely controlling the **running ScanControl program.** The following examples connect to a ScanControl instance on the **same computer** (localhost, so the default values for connecting are used.

Example 1 (example1.py):

The first example demonstrates how to start the measurement.

```
1. from scancontrolclient import ScanControlClient
2.
3. client = ScanControlClient()
4. client.connect()
5. ScanControl = client.scancontrol
6. client.run(ScanControl.start())
```

In Line 3, the `ScanControlClient` object is created. By calling the constructor without arguments, an event loop is created automatically.

The connection is established by executing the function `connect()`. It finishes after a connection was established. Then all the properties and methods of ScanControl are dynamically created in the member object `scancontrol`.

So in **Line 5**, the `ScanControl` object is created as a local copy of `client.scancontrol`.

In **Line 6**, the measurement is started by executing the `start()` function of `ScanControl`. This is done using the `run()` function of `ScanControlClient`.

Note: Functions that run on the remote software must be put into an event loop. To use the client's internal event loop, the function `run()` is available. Alternatively, the loop can be directly accessed using `client.loop` , and the method of `loop .run_until_complete`.

In the ScanControl logfile (available through "Help->Show Log"), the following lines will be written:

```
[2018/11/21 11:21:53.620 I] (remoting.webChannel): New WebChannel
connection: QHostAddress("::1") port: 2060 name: ""
[2018/11/21 11:21:53.671 I] (remoting.webChannel): WebChannel connection
lost: QHostAddress("::1") port: 2060 name: ""
```

The first line is printed when the `connect()`-function is executed, the second line appears when the program terminates, as this automatically closes the websocket connection. If you have troubles with the connection, look into the log file to see if connections get interrupted.

Note: Do not open too many parallel connections. There is no need to have more than one connection, and this will lead to higher memory requirements and more traffic.

Now if we want to receive data, we need to use signals and slots. The following is a simple example of how to work with the `pulseReady` signal to acquire data:

Example 2 (example2.py):

```python
from scancontrolclient import ScanControlClient

def gotPulse(data):
    print('Got Pulse:')
    print(data['amplitude'][0])

client = ScanControlClient()
client.connect()
ScanControl = client.scancontrol

ScanControl.pulseReady.connect(gotPulse)
client.loop.run_until_complete(ScanControl.start())
client.loop.run_forever()
```

Before sending the command `start()` a function is set to be called when the Signal `pulseReady` is received. This is done by the `connect()` function, that is available for any signal in that is available in the ScanControl object. As argument, it takes the receiving function. The number of arguments must match the number of arguments passed to the callback function, which is listed in the documentation above.

The `gotPulse` function is requested to be called when a pulse was measured. This function receives the measurement data, the time axis, the pulse flags and the timestamp as `dict` object. The function is only called when new data is measured, so no polling for data is needed.

After sending the start command, the loop is run forever, so that data can come in and is processed.

The program produces the following output:

```
Connected.
Got Pulse:
[  1.68579373e-03   5.64836892e-04   1.47932259e-03 ...,   5.97677333e-03
  -6.39934833e-03  -8.95167218e-05]
Got Pulse:
[ 0.0041957  -0.00302731  0.00406636 ..., -0.00595427 -0.00339114
 -0.00448904]
.
.
.
```

The program runs until it is terminated from outside (interruption by Ctrl + C, or is otherwise stopped).

Next, a bigger example is given, where a graphical user interface (GUI) is used to display time-domain data.

Example 3 (example3.py):

```python
from scancontrolclient import ScanControlClient, ScanControlStatus
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QHBoxLayout, QV
BoxLayout, QStatusBar, QMainWindow
from PyQt5.QtCore import pyqtSignal, QObject
import pyqtgraph as pg
import asyncio


class CommunicationBackend(QObject):

    displayPulseReady = pyqtSignal(dict)
    statusChanged = pyqtSignal(int)

    def __init__(self, loop = None):
        super().__init__()
        self.loop = loop
        if self.loop is None:
            self.loop = asyncio.get_event_loop()

    def _got_display_pulse(self, data):
        self.displayPulseReady.emit(data)

    def _status_changed(self, newStatus):
        self.statusChanged.emit(newStatus)

    def establish_connection(self):
        client = ScanControlClient(self.loop)
        client.connect()
        self.ScanControl = client.scancontrol
        self.ScanControl.displayPulseReady.connect(self._got_display_pulse)
        self.ScanControl.statusChanged.connect(self._status_changed)

    async def start(self):
        await self.ScanControl.start()

    async def stop(self):
        await self.ScanControl.stop()


class mainWindow(QMainWindow):

    def __init__(self, backend = None, loop = None):
        super().__init__()
        self.initUI()
        self.loop = loop
        if self.loop is None:
            self.loop = asyncio.get_event_loop()

        if backend is not None:
            self.connect_signals(backend)

    def initUI(self):
        self.setGeometry(300, 300, 300, 220)
        self.setWindowTitle('Example3')

        hbox = QHBoxLayout()
        hbox.addStretch(1)
        self.start_btn = QPushButton('Start', self)
        self.stop_btn = QPushButton('Stop', self)
        hbox.addWidget(self.start_btn)
        hbox.addWidget(self.stop_btn)
        vbox = QVBoxLayout()
        vbox.addStretch(1)
        self.guiplot = pg.PlotWidget()
        vbox.addWidget(self.guiplot)
        vbox.addLayout(hbox)
        self.statusBar = QStatusBar()
        self.setStatusBar(self.statusBar)
        widget = QWidget()
        widget.setLayout(vbox)
        self.setCentralWidget(widget)
        self.show()

    def _plot(self, data):
        self.guiplot.clear()
        self.guiplot.plot(data['amplitude'][0])

    def _status_changed(self, status):
        self.statusBar.showMessage("Status: " + str(ScanControlStatus(status).name))

    def connect_signals(self, backend):
        self.start_btn.clicked.connect(lambda: self.loop.create_task(backend.start()))
        self.stop_btn.clicked.connect(lambda: self.loop.create_task(backend.stop()))
        backend.displayPulseReady.connect(self._plot)
        backend.statusChanged.connect(self._status_changed)


async def process_events():
    while True:
        app.processEvents()
        await asyncio.sleep(0.1)

if __name__ == '__main__':
    app = QApplication(sys.argv)

    loop = asyncio.get_event_loop()

    backend = CommunicationBackend(loop)
    backend.establish_connection()

    window = mainWindow(backend)

    loop.run_until_complete(process_events())
```

The program separates the GUI from the communication backend, and makes heavy use of custom Qt signals.

One event loop is used for all the events that are triggered in the program. This is done by calling the constructors of both backend and the GUI window. The signals of `scancontrolclient` are received by the backend (including the pulse data when measuring), which in turn forwards them to the GUI. The GUI is able to send `start()` and `stop()`-commands to the `backend`, which then forwards those commands to `scancontrolclient`.

For details about python and `asyncio`, see the Python documentation.

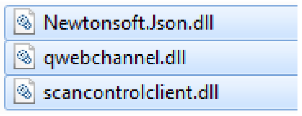
5.4 LABVIEW PROGRAMMING LANGUAGE

For communication in LabView, the .NET library implementation encapsulated in the file `scancontrolclient.dll` is used. The available functions, variables and definitions are therefore found in the .NET Section 5.2.1. Please contact Menlo Systems to get the library files.

Note: ScanControl must be running on a local or remote computer, initializing and running the spectrometer. Direct access to the hardware is not supported.

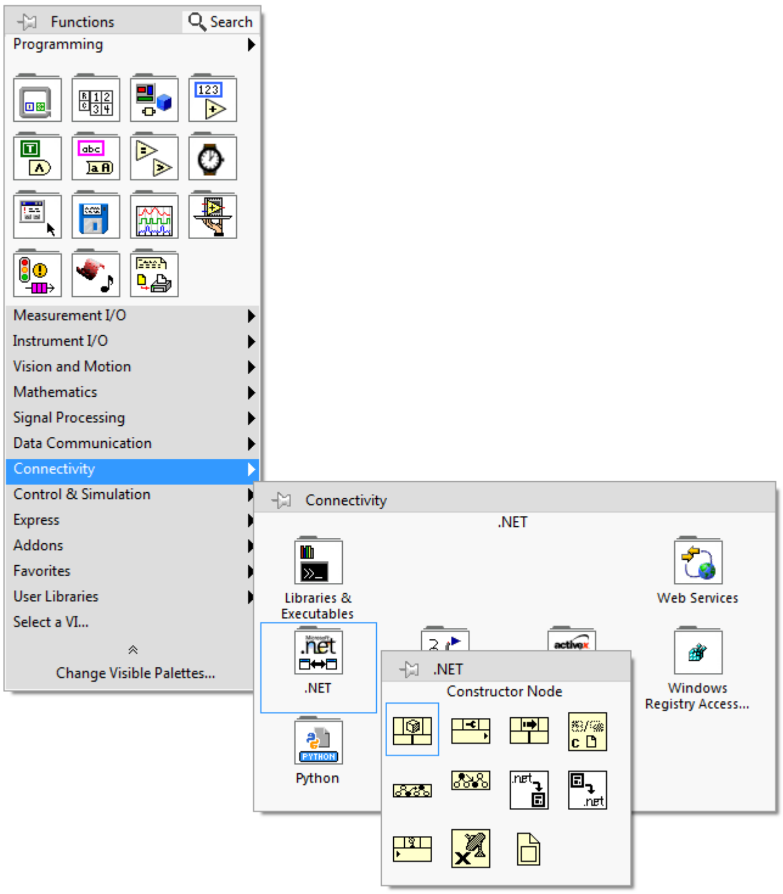
5.4.1LABVIEW .NET TUTORIAL

To access the program functionality, the three `.dll` files, `Newtonsoft.Json.dll`, `qwebchannel.dll` and `scancontrolclient.dll` must be available for LabView.


Newtonsoft.Json.dll
qwebchannel.dll
scancontrolclient.dll

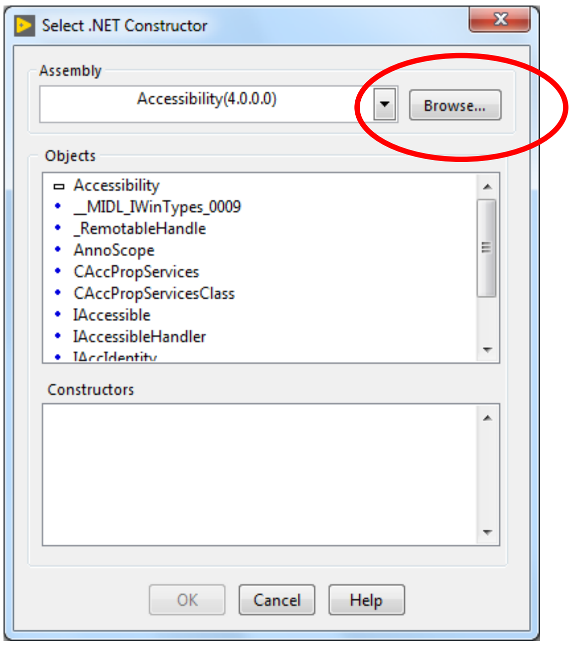
The following steps in LabView enable the user to call .NET functions:

1. Go to the LabView block diagram
2. Go to functions dialog -> “Connectivity”->.NET and select “Constructor Node”

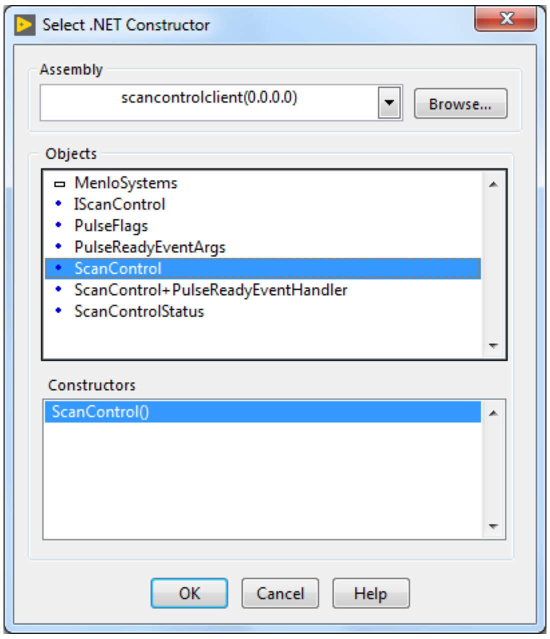


3. Place the node in the block diagram. A dialog appears.

4. Press “Browse”



5. Select “scancontrolclient.dll” from the directory. The following selection will appear:

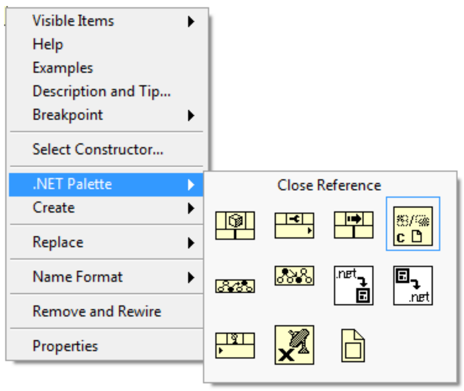


6. Select “ScanControl” constructor and press OK.

Now the `ScanControl` object is available.

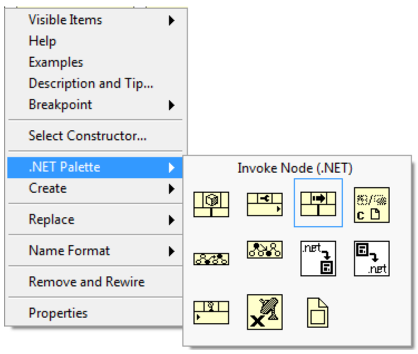
Deleting the reference

Whenever a ScanControl object is not needed anymore, the reference should be deleted in order to free unnecessarily occupied memory. This is done by using the block “Close Reference” from the LabVIEW .NET palette:

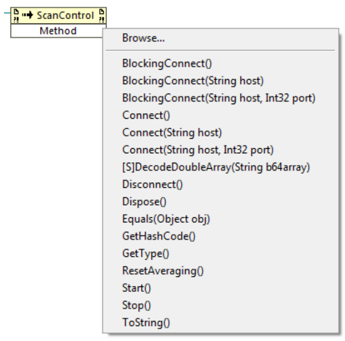


Calling methods

By right-clicking the reference node, the .NET palette can be accessed directly. Functions/Methods can be called by the “Invoke Node (.NET)” node.



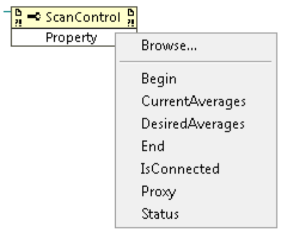
When connecting the ScanControl reference, a list of public methods is automatically displayed:



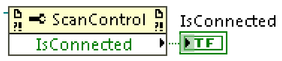
If parameters are needed for the function, LabVIEW automatically displays parameter nodes. ***Note: Don’t use the functions Dispose(), Disconnect(), and BlockingConnect() in LabVIEW. Exceptions cause by those functions cannot be caught in LabVIEW, which lead to a total crash of LabVIEW. Similar problems for this functions could arise in a MATLAB environment.***

Setting/Getting properties

By right-clicking the reference node, the .NET palette can be accessed directly. Properties can be get/set “Property Node (.NET)” node:

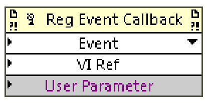


By selecting on of the options, the datatype is automatically detected and in/output can be connected to the node. This shows an example for an indicator that displays whether the connection is established:



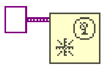
Using .NET event handlers

In the following we describe a step by step procedure on how to wire a LabVIEW diagram for the handling of an event. As example for an event we take the built in ScanControl event “`OnConnected`”.

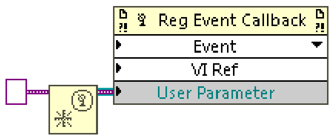


The node “Reg Event Callbacks” defines what happens, when a particular event is triggered in .NET. In Example 2 below, the function is used. The following steps must be taken when creating an event:

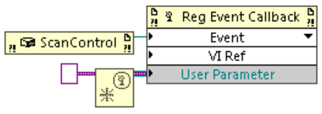
1. Create a user event with an input data type suitable for the data transferred by the event. Here the data type is variant.



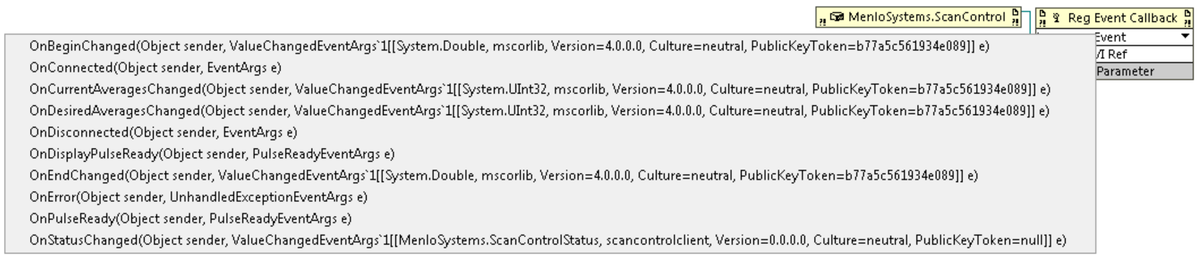
2. Connect the “user event out” connector to the “User Parameter” input.



3. Connect a .NET constructor reference to the “Event” input.

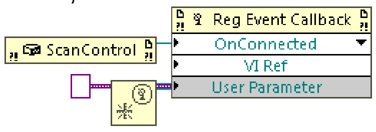


4. Select the appropriate event from the drop-down list:

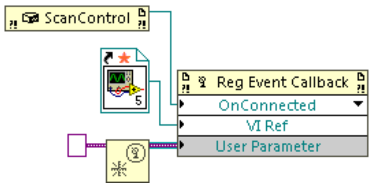


```
OnBeginChanged(Object sender, ValueChangedEventArgs`1[[System.Double, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] e)
OnConnected(Object sender, EventArgs e)
OnCurrentAveragesChanged(Object sender, ValueChangedEventArgs`1[[System.UInt32, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] e)
OnDesiredAveragesChanged(Object sender, ValueChangedEventArgs`1[[System.UInt32, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] e)
OnDisconnected(Object sender, EventArgs e)
OnDisplayPulseReady(Object sender, PulseReadyEventArgs e)
OnEndChanged(Object sender, ValueChangedEventArgs`1[[System.Double, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] e)
OnError(Object sender, UnhandledExceptionEventArgs e)
OnPulseReady(Object sender, PulseReadyEventArgs e)
OnStatusChanged(Object sender, ValueChangedEventArgs`1[[MenloSystems.ScanControlStatus, scancontrolclient, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]] e)
```

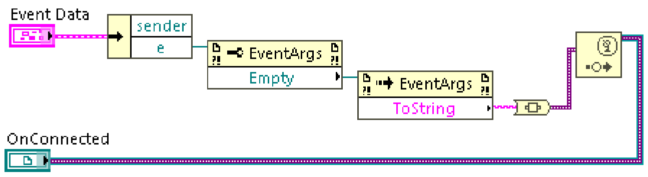
Here, the event “`OnConnected`” is used:



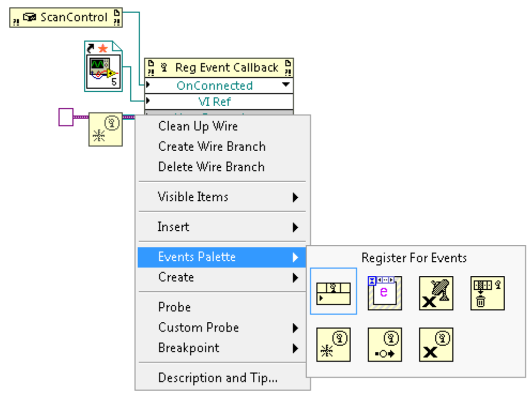
5. To transmit properly the data coming with the signal triggered by the event, we need to create an appropriate “Callback VI”. Right-Click on “VI Ref” and select “Create Callback VI”:
6. A new VI window will appear. Save the VI under an appropriate name.
   In the main window, the callback will be connected to the node:



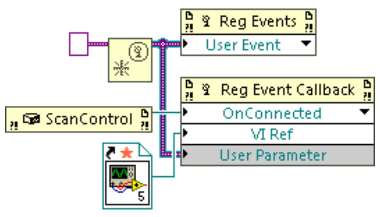
7. Inside the Callback VI, use the “Generate User Event” node to generate the event. For this, the user event is available in the SubVI, in this case as “`OnConnected`”. Data can be given to In this example, the .NET `.toString()` function is simply used and then the string is converted to `variant`:



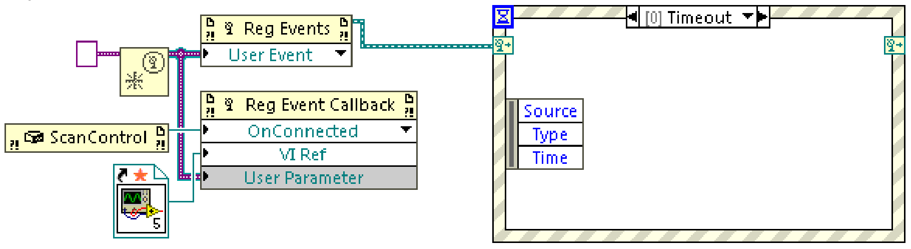
8. To register the occurring of the Event in the main VI, a “Register For Events” node is needed.



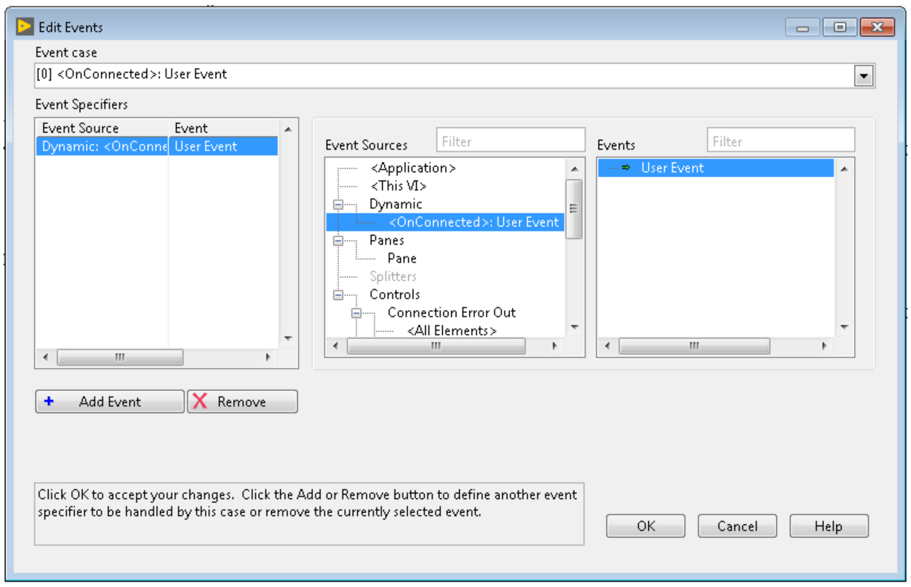
   It must be connected to the user event wire:



9. The output reference of the “Reg Events” node can be used in an event structure in the main VI to trigger an action. Right click on the event structure border and select “Show Dynamic Event Terminals”. Then the user event can be connected to the structure:



10. Customize the event structure to handle the user event:


Edit Events

Event case
[0] <OnConnected>: User Event

Event Specifiers

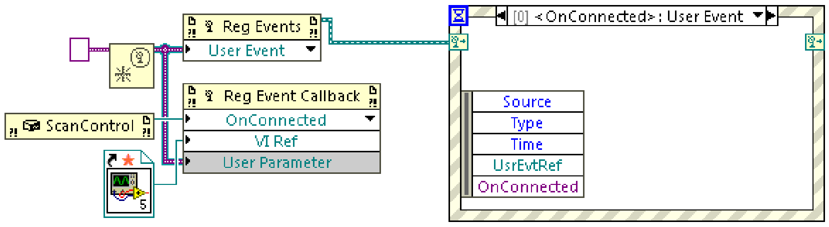
Event Source	Event
Dynamic: <OnConne	User Event

Event Sources
<Application>
<This VI>
Dynamic
<OnConnected>: User Event
Panes
Pane
Splitters
Controls
Connection Error Out
<All Elements>

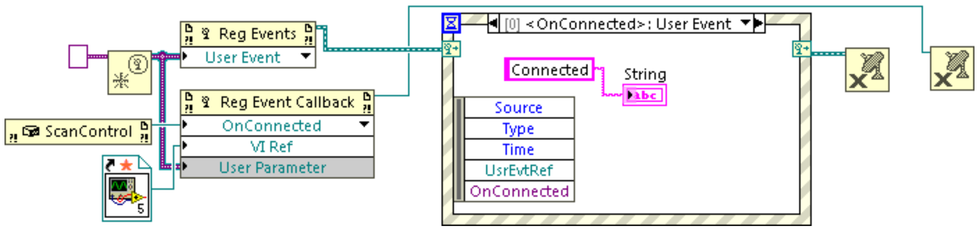
Events
User Event

Click OK to accept your changes. Click the Add or Remove button to define another event specifier to be handled by this case or remove the currently selected event.

When pressing OK, the data generated in the callback is available in the event structure (see purple variant object “OnConnected”):



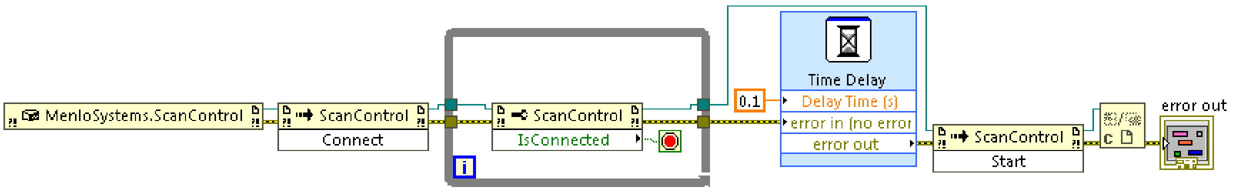
11. Make sure to disconnect all events when they are not needed anymore:



Now follows a list of three example VI, where we employ the concepts seen so far in a useful way.

5.4.2LABVIEW EXAMPLES

Example 1 (example1.vi):



In the first example, a connection is established to a `ScanControl` instance on the same computer, and then a single `Start` command is sent.

First the `ScanControl` object is created by calling the constructor, then the `Connect` function is called.

Since establishing connections over network takes some time, one must make sure that the connection is established before commands are sent. Checking if the connection is stable is done by checking the property `IsConnected`. When the status of `IsConnected` is true, the program continues. An additional delay is needed for building up the necessary object memory in the background. Finally, the `Start` command is sent, following by the “close reference” node.

Note: Do not use the `BlockingConnect` method in LabView. This function blocks further execution until the connection is established. If there is an error (e.g. ScanControl is not running), the function never finishes and LabVIEW is deadlocked, meaning that it’s impossible to stop the VI execution. Use the described connection method in this example.

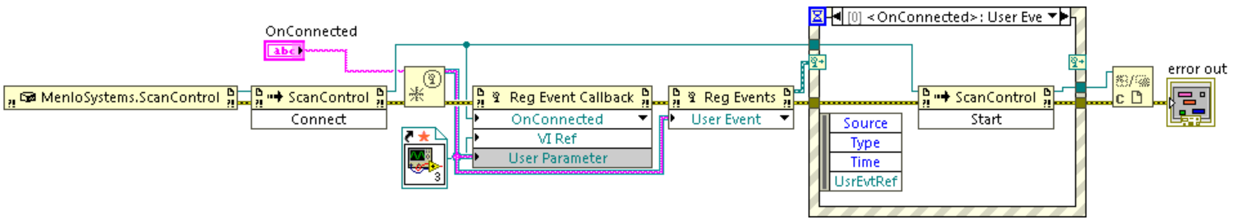
If an error/exception occurs, this is displayed in the `error out` node.

When running this example without ScanControl running on the target machine, the condition for `IsConnected` will never be met, which leads to an infinite loop.

Note: Always close the reference to ScanControl when the program finishes. Otherwise, memory will not be freed. This might cause memory requirements over time.

Example 2 (example2.vi):

A more complex, but also more sophisticated way to establish a connection is by using the event handler `OnConnected`. This way, no wait loops or delays need to be integrated as nodes.



After the `Connect`-function, an event is defined using the `Create user event` node. It needs a data type as input (left top input). This data type must match the data type if the information one wants to get as data when the event structure is entered. Since at this stage, we do not need further information except that the connection was successful, a string named “`OnConnected`” was fed into this input.

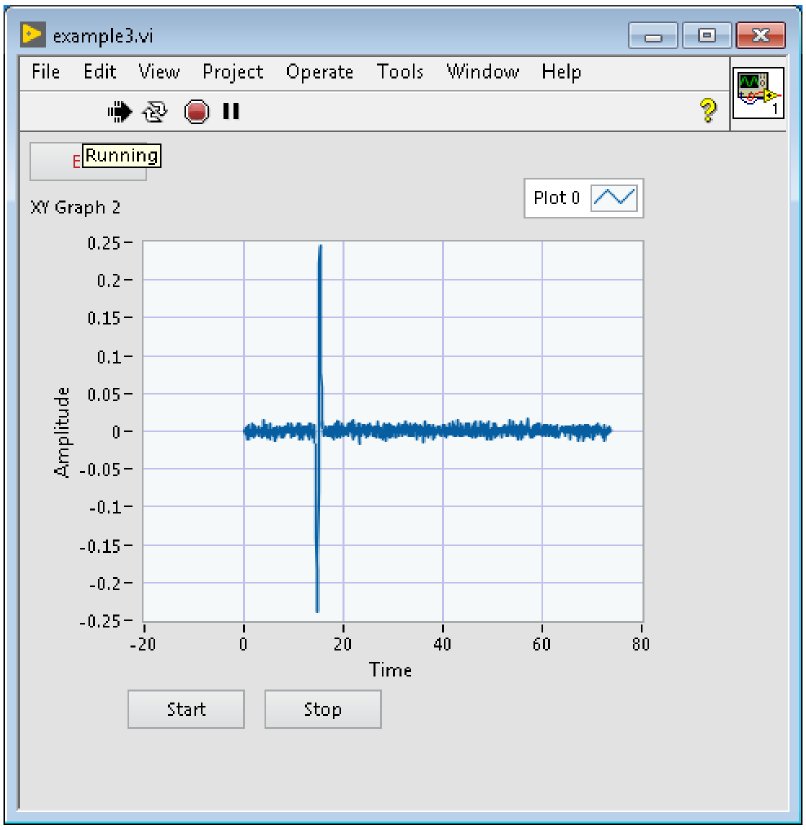
Next, an Event Callback is be registered. This node defines what happens, whenever a particular event is triggered in .NET. The callback VI OnConnected.vi is linked to “`Vi Ref`”. The user event is linked to “`User Parameter`”, and the reference of `ScanControl` to the Event node. As event, “`OnConnected`” was selected.

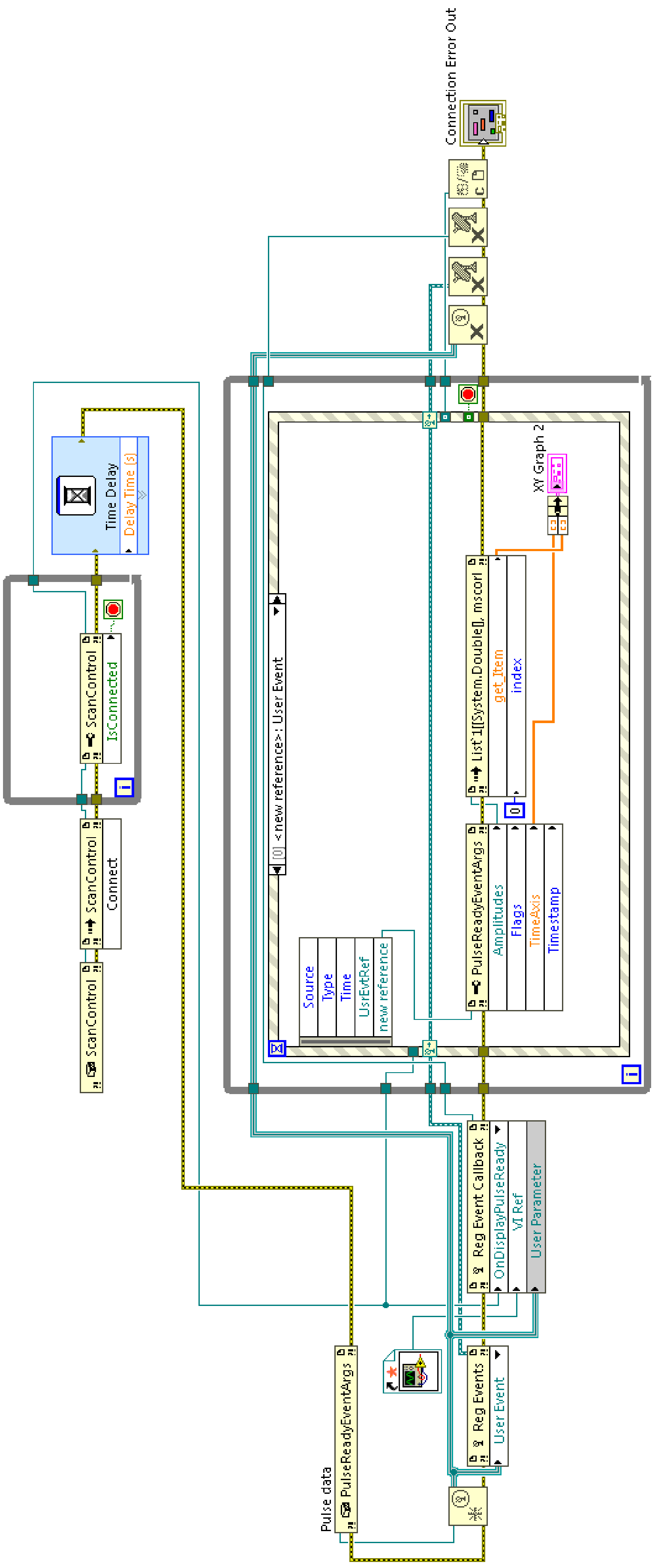
Then, the user event is registered and a custom event structure is linked to it. Inside, the `Start` command is sent. This makes sure, that the `Start` command is only sent when the connection was successful.

Note: For more detailed step-by-step instruction about event callbacks, see the section on event callbacks above.

Example 3 (example3.vi):

Next, an example of a simple program that displays pulse data and allows the user to Start and Stop the program is shown.

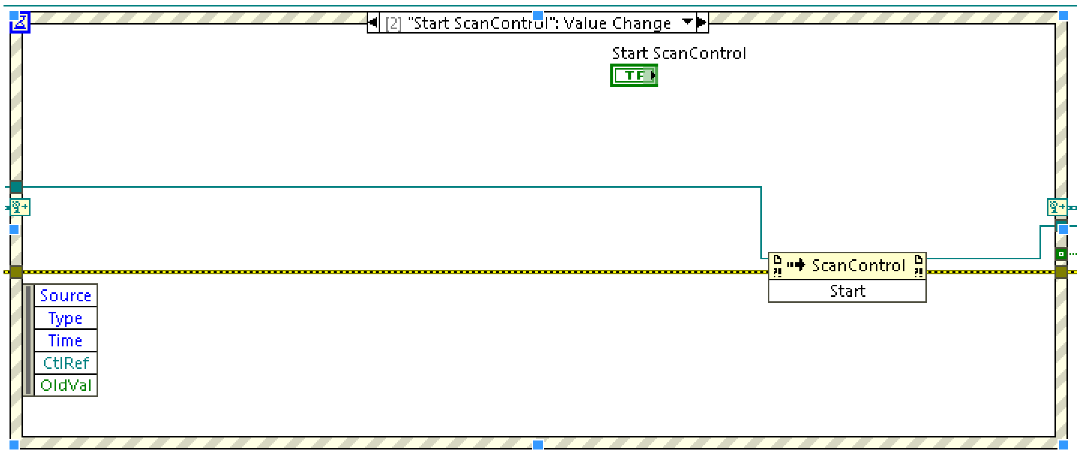




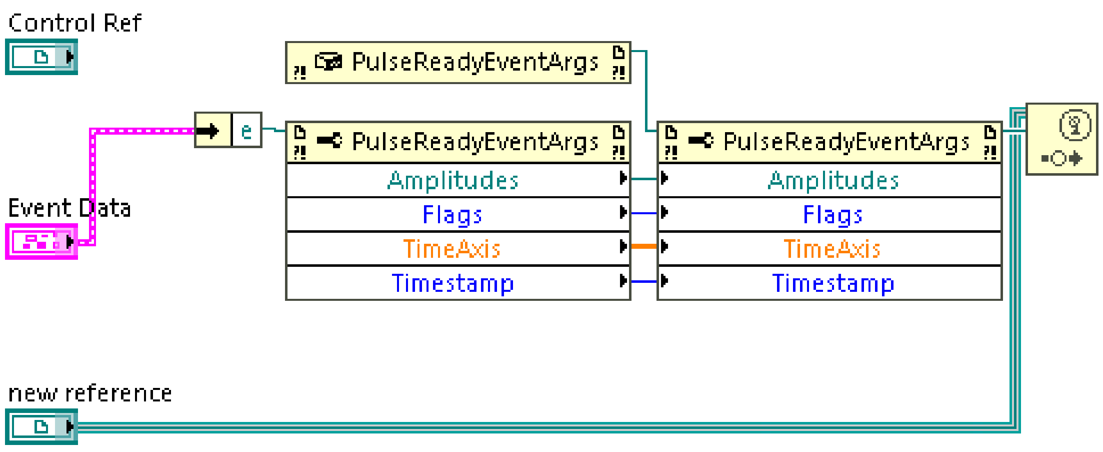
Connection Error Out
XY Graph 2
Time Delay
Delay Time (s)
ScanControl
IsConnected
ScanControl
Connect
ScanControl
[0] <new reference>: User Event
Source
Type
Time
UsrEvtRef
new reference
PulseReadyEventArgs
Amplitudes
Flags
TimeAxis
Timestamp
List`1[System.Double[], mscorl
get_Item
index
0
Reg Event Callback
OnDisplayPulseReady
VI Ref
User Parameter
Reg Events
User Event
Pulse data
PulseReadyEventArgs

The program starts at the top, again with the ScanControl constructor, followed by establishing the connection. This is then followed by defining a callback for the event "`OnDisplayPulseReady`", to which the callback VI "OnDisplayPulseReady.vi" is connected (see below).

Inside the loop, the custom event for the data is caught. The data is extracted using the "`get_Item(int)`" function of .net to extract the double array data of the amplitudes of type `List[]`. The input of channel 1 is extracted, and plotted on an XY graph, together with the `TimeAxis` as `x` axis. The other cases of the event loop send the `Start()` and `Stop()` commands, e.g.



The following shows the contents of the callback "OnDisplayPulseReady.vi":



The PulseReadyEventArgs object is copied to a new instance of it.

Note: The PulseReadyEventArgs object must be copied to a new array inside the callback, because otherwise only the reference (pointer, memory address) is transferred to the event structure, which results in wrong or zero data.

Note: Using event handlers is the only way to get pulse data from ScanControl using the webchannel interface.

The program demonstrates how event handlers can be used in order to design very simple and elegant programs.

Headquarters

Menlo Systems GmbH

Am Klopferspitz 19a
D-82152 Martinsried, Germany

service@menlosystems.com
sales@menlosystems.com

Phone:	+49 89 189 166 0
Fax:	+49 89 189 166 111

US Office

Menlo Systems, Inc
56 Sparta Avenue
Newton, NJ 07860, USA

service@menlosystems.com
ussales@menlosystems.com

Phone:	+1 973 300 4490
Fax:	+1 973 300 3600