RsInstrument C#

Release 1.19.0.80

Rohde Schwarz

Aug 09, 2023

CONTENTS:

RsInstrument is a .NET component that provides convenient way of communicating with Rohde & Schwarz instruments.

Basic Hello-World code:

```
using System;
using System.Text;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.122::hislip0");
            string idn = instr.Query("*IDN?");
            Console.WriteLine("\nHello, I am: " + idn);
            instr.Dispose();

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();
        }
    }
}
```

More examples on Rohde & Schwarz Github repository:

- Miscelaneous C#
- Miscelaneous VB.NET
- Oscilloscopes
- Powersensors
- Spectrum Analyzers

Preconditions

- Installed R&S VISA 5.12+ or NI VISA 18.0+
- No VISA installation is necessary if you select the plugin SocketIO

Supported Frameworks

- .NET Core 3.1
- .NET Standard 2.1
- .NET Standard 2.0
- .NET Framework 4.8
- .NET Framework 4.5

CHAPTER

ONE

REVISION HISTORY

1.1 Version 1.18.1.79 - 20.04.2023

- Fixed NuGet `readme.md` file.

1.2 Version 1.18.0.78 - 20.04.2023

- Changed the accepted `*IDN?` response to more permissive.
- Added SkipStatusSystemSettings to the options string, default value is false.
- Added methods `Utilities.GoToLocal()`, `Utilities.GoToRemote()`.

1.3 Version 1.17.0.75 - 30.05.2022

- Added platform - dependent Visa DLL load for .NET Core builds. The loading now works for Linux and OSX.
- Added mikro to the list of known SI-prefixes for double, int32, int64 conversions.
- Added Session settings string tokens DisableStbQuery (false), DisableOpcQuery (false).
- Changed parsing of SYST:ERR? response to tolerate +0,"No Error" response.

1.4 Version 1.15.0.67 - 21.10.2021

- Added .NET Standard 2.0 allowing targeting .NET Core and .NET Framework with one assembly.
- Added RohdeSchwarz.RsInstrument.Conversions namespace with double,integer,boolean conversion extention methods.

1.5 Older Versions

Version 1.14.0.65 - 15.10.2021

- Fixed CheckStatus() which was skipped if the QueryInstrumentStatus was false. Now the error checking is performed regardless that settings
- Added correct conversion of strings with SI suffixes (e.g.: MHz, KHz, THz, GHz, ms) to double, int32, int64
- Fixed VISA read buffer in case of multi-threading access

Version 1.13.0.64 - 28.09.2021

- Fixed bug where the NuGet packages contained debug versions of the assemblies with file version 1.0.0.0
- Additional changes only relevant to auto-generated drivers

Version 1.11.0.61 - 19.05.2021

- Added constructor RsInstrument(string resourceName, string optionString)
- improved options string help
- added checking for empty or null resourceName in the constructor

Version 1.10.1.60 - 18.04.2021

- **Added alias methods:**
  - Query() = QueryString()
  - Write() = WriteString()
  - QueryWithOpc() = QueryStringWithOpc()
  - WriteWithOpc() = WriteStringWithOpc()

Version 1.10.0.57 - 19.01.2021

- Added documentation on https://rsinstrumentcsharp.readthedocs.io/
- Changes relevant to auto-generated drivers only

Version 1.9.0.56 - 14.01.2021

- Added QueryOpc(int visaTimeout)
- Fixed error where the System.TimeoutException was thrown instead of the RsInstrument.VisaTimeoutException
- Cosmetic changes

Version 1.8.0.55 - 14.12.2020

- Fixed setting of VISA Timeout by init to 10000ms
- Added “DTX”, “Dtx”, “dtx” to a list of values that are represented as NaN

Version 1.7.3.53 - 25.11.2020

- NuGet package signed with Rohde Schwarz certificate
- Core change: Only relevant for auto-generated instrument drivers

Version 1.7.2.51 - 16.11.2020

- Changed NuGet icon
- Adjusted Company name and copyright
- Core change: Only relevant for auto-generated instrument drivers

Version 1.7.0.50 - 11.11.2020

- Changed authors and copyright information
- Core change: Conversion of the empty returned string to array returns empty array. Before, the empty string was converted to an array of one empty element.
- Added QueryStringList(), QueryStringListWithOpc()
- Added QueryBooleanList(), QueryBooleanListWithOpc()

Version 1.6.4.48 - 09.11.2020

- Fixed parsing of the instrument errors when an error message contains two double quotes

Version 1.6.3.47 - 22.10.2020

- Changes only relevant for auto-generated instrument drivers
- Added 'UND' to the list of numbers that are represented as NaN

Version 1.6.0.43 - 05.10.2020

- New Core with added OptionsString token 'TermChar' for setting a custom termination character
- Added 'Hameg' to the list of supported instruments
- Added static method AssertMinVersion() for checking the RsInstrument minimum version

Version 1.5.2.42 - 17.09.2020

- Changes only relevant for auto-generated instrument drivers

Version 1.5.1.41 - 04.09.2020

- New Core 1.8.2.41 with the fix for instrument that do not support OPT? query

Version 1.5.1.40 - 24.08.2020

- New Core 1.8.1.40 with the fixed simulation mode issues

Version 1.5.0.39 - 11.08.2020

- Multi-target frameworks .NET Standard 2.1, .NET Core 3.1, .NET Framework 4.5 and 4.8
- New Core 1.8.0.38 with these features:
- Implemented SocketIO Visa Plugin that does not need VISA
- New Options token: 'SelectVisa' with parameters: NativeVisa | RsVisa | RsVisaPrio | Socket
- Options token 'PreferRsVisa' is now obsolete (but still supported)
- Added new static function RsInstrument.FindResources()

Version 1.4.2.38 - 04.08.2020

- Fixed buffer size for Nrp-Z sessions
- Added and corrected examples

Version 1.4.0.36 - 20.07.2020

- Distributed as NuGet package
- Changed Core to allow for AnyCPU build
- Added Session Settings bool AssureResponseEndWithLF

Version 1.3.0.34 - 19.06.2020

- Added invoking read_segmented event for the first chunk of the ReadUnknownLength()
- New Core with RepeatedCapabilities for command groups

Version 1.2.0.32 - 10.01.2020

- New Core with reworked session settings
- Support for NRP-Zxx instruments

Version 1.1.0.30 - 29.11.2019

- Reorganized Utilities interface to sub-groups
- Added Write/Query With Opc Event
- Added locking for multithreading safety
- Added segmented read / write events

Version 1.0.0.20

- First released version

CHAPTER

TWO

WELCOME TO THE RSINSTRUMENT C# STEP-BY-STEP GUIDE

CHAPTER

THREE

1. INTRODUCTION

RsInstrument is a C# remote-control communication module for Rohde & Schwarz SCPI-based Test and Measurement Instruments. The original title of this document was **"10 Tips and Tricks…"**, but there were just too many cool features to fit into 10 chapters. We wanted to skip thirteen, so we ended up with fourteen.

Some of the RsInstrument's key features:

- You can select which VISA to use or even not use any VISA at all
- Initialization of a new session is straight-forward, no need to set any other properties
- Many useful features are already implemented - reset, self-test, opc-synchronization, error checking, option checking
- Binary data blocks transfer in both directions
- Transfer of arrays of numbers in binary or ASCII format
- File transfers in both directions
- Events generation in case of error, sent data, received data, chunk data (in case of big data transfer)
- Multithreading session locking - you can use multiple threads talking to one instrument at the same time

CHAPTER

FOUR

2. INSTALLATION

RsInstrument is hosted on NuGet.org. You can install it with **Visual Studio Package Manager** or **Visual Studio Packet Manager Console**:

```
PM> Update-Package -Id RsInstrument -reinstall
```

CHAPTER

FIVE

3. FINDING AVAILABLE INSTRUMENTS

RsInstrument can search for available instruments:

```
// Find the instruments in your environment

using System;
using System.Collections.Generic;
using System.Linq;
// Install as NuGet package from www.nuget.org repository
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            // Use the instrList items later as resource names in the RsInstrument␣
↪constructor
            IEnumerable<string> instrList = RsInstrument.FindResources("?*");
            instrList.ToList().ForEach(Console.WriteLine);
        }
    }
}
```

If you have more VISAs installed, the one actually used by default is defined by a secret widget called VISA Conflict Manager. You can force your program to use R&S VISA:

```
// Find any USB instruments in your environment with the VISA preference

using System;
using System.Collections.Generic;
using System.Linq;
// Install as NuGet package from www.nuget.org
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
```

(continues on next page)

(continued from previous page)

```
            // 'vxi11' and 'lxi' boolean arguments have only effect for R&S VISA
            // As 'plugin' you can use:
            // - 'NativeVisa' (default system VISA)
            // - 'RsVisa' (Rohde & Schwarz VISA)
            // - 'RsVisaPrio' (Prefer Rohde & Schwarz VISA, fall back to default VISA)
            string plugin = "RsVisa";
            var instrList = RsInstrument.FindResources("USB?*", false, false, plugin);
            instrList.ToList().ForEach(Console.WriteLine);
        }
    }
}
```

Tip: We believe our R&S VISA is the best choice for our customers. Here are the reasons why:

- Small footprint
- Superior VXI-11 and HiSLIP performance
- Integrated legacy sensors NRP-Zxx support
- Additional VXI-11 and LXI devices search
- Availability for Windows, Linux, Mac OS

CHAPTER

SIX

4. INITIATING INSTRUMENT SESSION

RsInstrument offers four different types of starting your remote-control session. We begin with the most typical case, and progress with more special ones.

6.1 Standard Session Initialization

Initiating new instrument session happens, when you instantiate the RsInstrument object. Below, is a Hello World example. Different resource names are examples for different physical interfaces.

```
// Hello World example for any R&S Instrument

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            // A good practice is to assure that you have a certain minimum version␣
↪installed
            RsInstrument.AssertMinVersion("1.19.0");
            // Standard LAN connection (also called VXI-11)
            var resourceString1 = "TCPIP::192.168.1.100::INSTR";
            // Hi-Speed LAN connection - see 1MA208
            var resourceString2 = "TCPIP::192.168.1.100::hislip0";
            // GPIB Connection
            var resourceString3 = "GPIB::20::INSTR";
            // USB-TMC (Test and Measurement Class)
            var resourceString4 = "USB::0x0AAD::0x0119::022019943::INSTR";
            // R&S Powersensor NRP-Z86 (needs NRP-Toolkit installed)
            var resourceString5 = "RSNRP::0x0095::104015::INSTR";

            // Initializing the session
            RsInstrument instr = new RsInstrument(resourceString1);
```

(continues on next page)

(continued from previous page)

```
            Console.WriteLine("RsInstrument Driver Version: " + instr.Identification.
↪DriverVersion);
            Console.WriteLine("Visa Manufacturer: " + instr.Identification.
↪VisaManufacturer);
            Console.WriteLine("Instrument Full Name: " + instr.Identification.
↪InstrumentFullName);
            Console.WriteLine("Installed Options: " + string.Join(",", instr.
↪Identification.InstrumentOptions));
            string idn = instr.QueryString("*IDN?");
            Console.WriteLine("\nHello, I am: " + idn);

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();

            // Close the session
            instr.Dispose();
        }
    }
}
```

Note: If you are wondering about the missing `ASRL1::INSTR`, yes, it works too, but come on… it's 2021.

Do not care about specialty of each session kind; RsInstrument handles all the necessary session settings for you. You immediately have access to many identification properties. Here are same of them:

```
string instr.Identification.IdnString;
Version instr.Identification.DriverVersion;
string instr.Identification.VisaManufacturer;
string instr.Identification.InstrumentFullName;
string instr.Identification.InstrumentSerialNumber;
string instr.Identification.InstrumentFirmwareVersion;
List<string> instr.Identification.InstrumentOptions;
```

The constructor also contains optional boolean arguments `idQuery` and `resetDevice`:

```
bool idQuery = true;
bool resetDevice = true;
RsInstrument instr = new RsInstrument("TCPIP::192.168.56.101::hislip0", idQuery,␣
↪resetDevice);
```

- Setting `idQuery` to true (default is true) checks, whether your instrument can be used with the RsInstrument module.
- Setting `resetDevice` to true (default is false) resets your instrument. It is equivalent to calling the `Reset()` method.

6.2 Selecting Specific VISA

Just like in the static method `FindResources()`, RsInstrument allows you to prefer R&S VISA:

```
// Open your instrument with preferred Rohde & Schwarz VISA

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            var resourceString = "TCPIP::192.168.1.100::INSTR";
            // Force use of the Rs Visa. For default system VISA, use the
↪"SelectVisa=NativeVisa"
            RsInstrument instr = new RsInstrument(resourceString, true, false,
↪"SelectVisa=RsVisa");

            string idn = instr.QueryString("*IDN?");
            Console.WriteLine("\nHello, I am: " + idn);
            Console.WriteLine("\nI am using VISA from " + instr.Identification.
↪VisaManufacturer);

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();

            // Close the session
            instr.Dispose();
        }
    }
}
```

6.3 No VISA Session

We recommend using VISA whenever possible, preferably with HiSlip session because of its low latency. However, if you are a strict VISA denier, RsInstrument has something for you too - **no Visa installation raw LAN socket**:

```
// Using RsInstrument without VISA for LAN Raw socket communication

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RohdeSchwarz.RsInstrument;
```

(continues on next page)

(continued from previous page)

```
namespace Examples
{
    class Program
    {
        static void Main()
        {
            var resourceString = "TCPIP::192.168.1.100::5025::SOCKET";  // SOCKET-like␣
↪resource name
            RsInstrument instr = new RsInstrument(resourceString, true, false,␣
↪"SelectVisa=SocketIo");
            string idn = instr.QueryString("*IDN?");
            Console.WriteLine("\nHello, I am: " + idn);

            Console.WriteLine("\nNo VISA has been harmed or even used in this example.");
            Console.WriteLine("Press any key ...");
            Console.ReadKey();

            // Close the session
            instr.Dispose();
        }
    }
}
```

Warning: Not using VISA can cause problems by debugging when you want to use the communication Trace Tool. The good news is, you can easily switch to use VISA and back just by changing the constructor arguments. The rest of your code stays unchanged.

6.4 Simulating Session

If a colleague is currently occupying your instrument, leave him in peace, and open a simulating session:

```
var instr = new RsInstrument("TCPIP::192.168.56.101::HISLIP", true, false, "Simulate=True
↪");
```

More `optionString` tokens are separated by comma:

```
var optionString = "SelectVisa='rs', Simulate=True";
var instr = new RsInstrument("TCPIP::192.168.56.101::HISLIP", true, false, optionString);
```

6.5 Shared Session

In some scenarios, you want to have two independent objects talking to the same instrument. Rather than opening a second VISA connection, share the same one between two or more RsInstrument objects:

```
// Sharing the same physical VISA session by two different RsInstrument objects

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            var instr1 = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            var instr2 = new RsInstrument(instr1.Session);

            Console.WriteLine("instr1: " + instr1.Identification.IdnString);
            Console.WriteLine("instr2: " + instr2.Identification.IdnString);

            // Closing the instr2 session does not close the instr1 session.
            // instr1 is the 'session master'
            instr2.Dispose();
            Console.WriteLine("instr2: I am closed now");

            Console.WriteLine("instr1: I am  still opened and working.");
            Console.WriteLine("Look, I can prove it: " + instr1.Identification.
↪IdnString);

            instr1.Dispose();
            Console.WriteLine("instr1: Only now I am closed.");

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();
        }
    }
}
```

Note: The `instr1` is the object holding the 'master' session. If you call the instr1.Dispose(), the `instr2` loses its instrument session as well, and becomes pretty much useless.

CHAPTER

SEVEN

5. BASIC I/O COMMUNICATION

Now we have opened the session, it's time to do some work. RsInstrument provides two basic methods for communication:

- `WriteString()` - writing a command without an answer e.g.: ***RST**
- `QueryString()` - querying your instrument, for example with the ***IDN?** query

You may ask a question. Actually, two questions:

- **Q1**: Why there are not called `Write()` and `Query()` ?
- **Q2**: Where is the `Read()` ?

Answer 1: Actually, there are: `Write()` = `WriteString()` and `Query()` = `QueryString()`, they are aliases. To avoid mixing string and binary communication, we promote the ones with `String` names. All the method names for binary transfer contain `Bin` in their names.

Answer 2: Short answer - you do not need it. Long answer - your instrument never sends unsolicited responses. If you send a set-command, you use `WriteString()`. For a query-command, you use `QueryString()`. So, you really do not need it...

Enough with the theory, let us look at an example. Simple write and query:

```
// Basic string WriteString / QueryString

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            instr.WriteString("*RST");
            var response = instr.QueryString("*IDN?");
            Console.WriteLine(response);

            Console.WriteLine("\nPress any key ...");
```

(continues on next page)

(continued from previous page)

```
            Console.ReadKey();

            // Close the session
            instr.Dispose();
        }
    }
}
```

This example is so-called "*University-Professor-Example*" - good to show a principle, but never used in praxis. The abovementioned commands are already a part of the driver's API. Here is another example, achieving the same goal:

```
// Basic string WriteString / QueryString

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            instr.Reset();
            Console.WriteLine(instr.Identification.IdnString);

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();

            // Close the session
            instr.Dispose();
        }
    }
}
```

One additional feature we need to mention here: **VISA Timeout**. To simplify, VISA Timeout plays a role in each `QueryXxx()`, where the controller (your PC) has to prevent waiting forever for an answer from your instrument. VISA Timeout defines that maximum waiting time. You can set/read it with the `VisaTimeout` property:

```
// Timeout in milliseconds
instr.VisaTimeout = 3000;
```

After this time, RsInstrument raises an exception. Speaking of exceptions, an important feature of the RsInstrument is **Instrument Status Checking**. Check out the next chapter that describes the error checking in details.

Tip: If you're dying for the `Write()` and `Query()`, use the cool extensions feature of C#:

```
public static class RsInstrumentExtensions
{
    // Mapping Write() to WriteString()
    public static void Write(this RsInstrument instr, string command)
    {
        instr.WriteString(command);
    }

    // Mapping Query() to QueryString()
    public static string Query(this RsInstrument instr, string query)
    {
        return instr.QueryString(query);
    }
}
```

For completion, we mention other string-based `QueryXxx()` methods, all in one example. They are convenient extensions providing type-safe double/boolean/integer querying features:

```
// Other query methods

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            instr.VisaTimeout = 5000;
            instr.InstrumentStatusChecking = true; // Default is true
            int scount = instr.QueryInteger("SWEEP:COUNT?"); // returning integer number
            bool output = instr.QueryBool("SOURCE:RF:OUTPUT:STATE?");  // returning␣
↪boolean value
            double freq = instr.QueryDouble("SOURCE:RF:FREQUENCY?");  // returning float␣
↪number

            // Close the session
            instr.Dispose();
        }
    }
}
```

Lastly, a method providing basic synchronization: `QueryOpc()`. It sends query ***OPC?** to your instrument. The instrument waits with the answer until all the tasks it currently has in the execution queue are finished. This way your program waits too, and it is synchronized with actions in the instrument. Remember to have the VISA timeout set to an appropriate value to prevent the timeout exception. Here's a snippet:

```
var oldTout = instr.VisaTimeout;
    instr.VisaTimeout = 3000;
instr.WriteString("INIT");
instr.QueryOpc();
    instr.VisaTimeout = oldTout;

// The results are ready now to fetch
results = instr.QueryString("FETCH:MEASUREMENT?");
```

You can also specify the `VisaTimeout` just for one `QueryOpc()` call. The following example is equivalent to the one above:

```
instr.WriteString("INIT");
    // Set the VISA Timeout for this call only, then set it back to the original value
instr.QueryOpc(3000);

// The results are ready now to fetch
results = instr.QueryString("FETCH:MEASUREMENT?");
```

Tip: Wait, there's more: you can send the ***OPC?** after each `WriteString()` automatically:

```
// Default value after init is false
instr.OpcQueryAfterEachSetting = true;
```

CHAPTER

EIGHT

6. ERROR CHECKING

RsInstrument has a built-in mechanism that after each command/query checks the instrument's status subsystem, and throws an exception if it detects an error. For those who are already screaming: **Speed Performance Penalty!!!**, don't worry, you can disable it.

Instrument status checking is very useful since in case your command/query caused an error, you are immediately informed about it. Status checking has in most cases no practical effect on the speed performance of your program. However, if for example, you do many repetitions of short write/query sequences, it might make a difference to switch it off:

```
// Default value after init is true
instr.InstrumentStatusChecking = false;
```

To clear the instrument status subsystem of all errors, call this method:

```
instr.ClearStatus();
```

Instrument's status system error queue is clear-on-read. It means, if you query its content, you clear it at the same time. To query and clear list of all the current errors, use the following:

```
IEnumerable<string> errorsList = instr.QueryAllErrors();
```

See the next chapter on how to react on write/query errors.

CHAPTER

NINE

7. EXCEPTION HANDLING

The base class for all the exceptions thrown by the RsInstrument is `RsInstrumentException`. Inherited exception classes:

- `InstrumentStatusException` thrown if a command or a query generated error in the instrument's error queue
- `VisaException` thrown if the underlying VISA component throws an error
- `OperationTimeoutException` thrown if OPC timeout is reached
- `VisaTimeoutException` thrown if visa timeout is reached

In this example we show usage of all of them:

```
// How to deal with RsInstrument exceptions

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr;
            try // Separate try-catch for initialization prevents accessing␣
↪uninitialized object
            {
                // Adjust the VISA Resource string to fit your instrument
                instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            }
            catch (RsInstrumentException e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("Your instrument is probably OFF...");
                // Exit now, no point of continuing
                Console.WriteLine("Press any key to finish.");
                Console.ReadKey();
                return;
```

(continues on next page)

(continued from previous page)

```
            }

            // Dealing with commands that potentially generate errors OPTION 1:
            // Switching the status checking OFF temporarily
            instr.InstrumentStatusChecking = false;
            instr.WriteString("MY:MISSpelled:COMMand");
            // Clear the error queue
            instr.ClearStatus();
            // Status checking ON again
            instr.InstrumentStatusChecking = true;

            // Dealing with queries that potentially generate errors OPTION 2:
            try
            {
                instr.VisaTimeout = 1000;
                instr.QueryString("MY:OTHEr:WRONg:QUERy?");
            }
            catch (InstrumentStatusException e)
            {
                // Instrument status error
                Console.WriteLine(e.Message);
                Console.WriteLine("Nothing to see here, moving on...");
            }
            catch (VisaException e)
            {
                // General Visa error
                Console.WriteLine(e.Message);
                Console.WriteLine("Somethin's seriously wrong...");
            }
            catch (VisaTimeoutException e)
            {
                // Visa Timeout error
                Console.WriteLine(e.Message);
                Console.WriteLine("That took a long time... longer than the VISA timeout
↪");
            }
            catch (OperationTimeoutException e)
            {
                // OPC Timeout error
                Console.WriteLine(e.Message);
                Console.WriteLine("You called some method with OPC, and that took a long␣
↪time...");
            }
            catch (RsInstrumentException e)
            {
                // General RsInstrument error.
                // RsInstrumentException is a base class for all the RsInstrument␣
↪exceptions
                Console.WriteLine(e.Message);
                Console.WriteLine("Some other RsInstrument error...");
            }
            finally
```

(continues on next page)

(continued from previous page)

```
            {
                instr.VisaTimeout = 5000;
                // Close the session in any case
                instr.Dispose();
            }
        }
    }
}
```

Tip: General rules for exception handling:

- If you are sending commands that might generate errors in the instrument, for example deleting a file which does not exist, use the **OPTION 1** - temporarily disable status checking, send the command, clear the error queue and enable the status checking again.
- If you are sending queries that might generate errors or timeouts, for example querying measurement that cannot be performed at the moment, use the **OPTION 2** - try/catch with optionally adjusting timeouts.

CHAPTER

TEN

8. OPC-SYNCHRONIZED I/O COMMUNICATION

Now we are getting to the cool stuff: OPC-synchronized communication. OPC stands for OPeration Completed. The idea is: use one method (write or query), which sends the command, and polls the instrument's status subsystem until it indicates: **"I'm finished"**. The main advantage is, you can use this mechanism for commands that take several seconds, or minutes to complete, and you are still able to interrupt it if needed. You can also perform other operations with the instrument in a parallel thread.

Now, you might say: **"This sounds complicated, I'll never use it"**. That is where the RsInstrument comes in: all the **Write/Query** methods we learned in the previous chapter have their `WithOpc` siblings. For example: `WriteString()` has `WriteStringWithOpc()`. You can use them just like the normal write/query with one difference: They all have an optional parameter `timeout`, where you define the maximum time to wait. If you omit it, it uses value from `OpcTimeout` property. Important difference between the meaning of `VisaTimeout` and `OpcTimeout`:

- `VisaTimeout` is a VISA IO communication timeout. **It does not play any role in the** `WithOpc()` methods. It only defines timeout for the standard `QueryXxx()` methods. We recommend to keep it to maximum of 10000 ms.
- `OpcTimeout` is a RsInstrument internal timeout, that serves as a default value to all the `WithOpc()` methods. If you explicitly define it in the method API, it is valid only for that one method call.

That was too much theory... Now an example:

```
// Write / Query with OPC
// The SCPI commands syntax is for demonstration only

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            instr.VisaTimeout = 3000;
            // OpcTimeout default value is 10000 ms
            instr.OpcTimeout = 20000;

            // Send Reset command and wait for it to finish
```

(continues on next page)

(continued from previous page)

```
            instr.WriteStringWithOpc("*RST");

            // Initiate the measurement and wait for it to finish, define the timeout 50␣
↪secs
            // Notice no changing of the VISA timeout
            instr.WriteStringWithOpc("INIT", 50000);
            // The results are ready, simple fetch returns the results
            // Waiting here is not necessary
            var result1 = instr.QueryString("FETCH:MEASUREMENT?");

            // READ command starts the measurement,
            // We use QueryStringWithOpc to wait for the measurement to finish
            var result2 = instr.QueryStringWithOpc("READ:MEASUREMENT?", 50000);

            // Close the session
            instr.Dispose();
        }
    }
}
```

CHAPTER

ELEVEN

9. QUERYING ARRAYS

Often you need to query an array of numbers from your instrument, for example a spectrum analyzer trace or an oscilloscope waveform. Many programmers stick to transferring such arrays in ASCII format, because of the simplicity. Although simple, it is quite inefficient: one float 32-bit number can take up to 12 characters (bytes), compared to 4 bytes in a binary form. Well, with RsInstrument do not worry about the complexity: we have one method for binary or ascii array transfer.

11.1 Querying Float Arrays

Let us look at the example below. The method doing all the magic is `Binary.QueryBinOrAsciiFloatArray()`. In the 'waveform' variable, we get back an array of float numbers:

```
// Querying ASCII float arrays

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument rto = new RsInstrument("TCPIP::192.168.1.100::INSTR", true, true);
            // Initiate a single acquisition and wait for it to finish
            rto.WriteStringWithOpc("SINGle", 20000);
            double[] waveform = rto.Binary.QueryBinOrAsciiFloatArray("FORM ASC;:CHAN1:DATA?");
            Console.WriteLine("Instrument returned points: " + waveform.Length);

            // Close the session
            rto.Dispose();
        }
    }
}
```

You might say: *I would do this with a simple 'query-string-and-split-on-commas'...* and you are right. The magic happens when we want the same waveform in binary form. One additional setting we need though - the binary data from the instrument does not contain information about its encoding. Is it 4 bytes float, or 8 bytes float? Low Endian or Big Endian? This, we have to specify with the property `Binary.FloatNumbersFormat`:

```
// Querying binary float arrays

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument rto = new RsInstrument("TCPIP::192.168.1.100::INSTR", true, true);

            // Initiate a single acquisition and wait for it to finish
            rto.WriteStringWithOpc("SINGle", 20000);

            // This tells the RsInstrument in which format to expect the binary float data
            rto.Binary.FloatNumbersFormat = InstrBinaryFloatNumbersFormat.Single4Bytes;
            // If your instrument sends the data with the swapped endianness, use the following value:
            // rto.Binary.FloatNumbersFormat = InstrBinaryFloatNumbersFormat.Single4BytesSwapped;

            double[] waveform = rto.Binary.QueryBinOrAsciiFloatArray("FORM REAL,32;:CHAN1:DATA?");
            Console.WriteLine("Instrument returned points: " + waveform.Length);

            // Close the session
            rto.Dispose();
        }
    }
}
```

Tip: To find out in which format your instrument sends the binary data, check out the format settings: **FORM REAL,32** means floats, 4 bytes per number. It might be tricky to find out whether to swap the endianness. We recommend you simply try it out - there are only two options. If you see too many NaN values returned, you probably chose the wrong one:

- `InstrBinaryFloatNumbersFormat.Single4Bytes` means the instrument and the control PC use the same endianness
- `InstrBinaryFloatNumbersFormat.Single4BytesSwapped` means they use opposite endianness

The same is valid for double arrays: settings **FORM REAL,64** corresponds to either

`InstrBinaryFloatNumbersFormat.Double8Bytes` or `InstrBinaryFloatNumbersFormat.Double8BytesSwapped`

11.2 Querying Integer Arrays

For performance reasons, we split querying float and integer arrays into two separate methods. The following example shows both ascii and binary array query. Here, the magic method is `Binary.QueryBinOrAsciiIntegerArray()` returning an array of integers:

CHAPTER

TWELVE

10. QUERYING BINARY DATA

A common question from customers: How do I read binary data to a byte stream, or a file?

If you want to transfer files between PC and your instrument, check out the following chapter: 12_Transferring_Files.

12.1 Querying to a Byte Array

Let us say you want to get raw RTO waveform data. Call this method:

```
byte[] data = instr.Binary.QueryData("FORM REAL,32;:CHAN1:DATA?");
```

12.2 Querying to PC files

Modern instrument can acquire gigabytes of data, which is often more than your program can hold in memory. The solution may be to save this data to a file. RsInstrument is smart enough to read big data in chunks, which it immediately writes into a file stream. This way, at any given moment your program only holds one chunk of data in memory. You can set the chunk size with the property `IoSegmentSize`. The initial value is 100 000 bytes.

We are going to read the RTO waveform into a PC file *c:\temp\rto_waveform_data.bin*:

```
rto.IoSegmentSize = 100000;
var myPcFile = new FileStream(@"c:\temp\rto_waveform_data.bin", FileMode.OpenOrCreate);
instr.Binary.QueryData("FORM REAL,32;:CHAN1:DATA?", myPcFile);
```

CHAPTER

THIRTEEN

11. WRITING BINARY DATA

13.1 Writing from bytes data

We take example for a Signal generator waveform data file. First, we construct a `waveformData` as `byte[]`, and then send it with `WriteData()`:

```
// MyWaveform.wv is an instrument file name under which this data is stored
smw.Binary.WriteData("SOUR:BB:ARB:WAV:DATA 'MyWaveform.wv',", waveformData);
```

Notice the `WriteData()` has two parameters:

- `string` parameter `command` for the SCPI command
- `byte[]` parameter `binaryData` for the payload data

13.2 Writing from PC files

The `WriteData()` has two overloads - either you provide data as `byte[]` or as a `Stream`, for example a `FileStream` connected to your source PC file:

```
FileStream mySourcePcFile = new FileStream(@"c:\temp\rto_waveform_data.bin", FileMode.
↪Open);
smw.Binary.WriteData("SOUR:BB:ARB:WAV:DATA 'MyWaveform.wv',", mySourcePcFile);
```

CHAPTER

FOURTEEN

12. TRANSFERRING FILES

14.1 Instrument -> PC

You just did a perfect measurement, saved the results as a screenshot to the instrument's storage drive. Now you want to transfer it to your PC. With RsInstrument, no problem, just figure out where the screenshot was stored on the instrument. In our case, it is `var/user/instr_screenshot.png`:

```
instr.File.FromInstrumentToPc(@"var/user/instr_screenshot.png", @"c:\temp\pc_screenshot.
↪png");
```

14.2 PC -> Instrument

Another common scenario: Your cool test program contains a setup file you want to transfer to your instrument: Here is the RsInstrument one-liner:

```
instr.File.FromPcToInstrument(@"c:\MyCoolTestProgram\instr_setup.sav", @"var/appdata/
↪instr_setup.sav");
```

CHAPTER

FIFTEEN

13. TRANSFERRING BIG DATA WITH PROGRESS

We can agree that it can be annoying using an application that shows no progress for long-lasting operations. The same is true for remote-control programs. Luckily, the RsInstrument has this covered. And, this feature is quite universal - not just for big files transfer, but for any data in both directions.

RsInstrument allows you to register a function (programmers fancy name is '*callback*'), which it invokes after each transfer of one data chunk. Moreover, because you can set the chunk size, you can pace it as you desire. You can even slow down the transfer speed, if you want to process the data as they arrive (direction instrument -> PC). To show this in praxis, we are going to use another *University-Professor-Example*: querying the ***IDN?** with chunk size of 2 bytes and delay of 200ms between each chunk read:

```
// Event handlers by reading - querying string data

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void MyTransferHandler(object sender, InstrSegmentEventArgs args)
        {
            Console.Write("\nSegment: " + args.SegmentIx);
            Console.Write(", transferred bytes: " + args.TransferredSize);
            Console.Write(", total size: " + args.TotalSize);

            // Get the current segment data
            args.DataStream.Position = args.SegmentDataOffset;
            string text = new StreamReader(args.DataStream).ReadToEnd();
            Console.Write(", data: '" + text + "'");
            args.DataStream.Position = args.DataStream.Length;

            if (args.Finished)
            {
                Console.WriteLine("\n\nEnd of Transfer");
            }
```

(continues on next page)

(continued from previous page)

```
            else
            {
                // Pause here for 100ms after each segment read
                Thread.Sleep(100);
            }
        }

        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            instr.Events.ReadSegmentHandler = MyTransferHandler;
            // Set data chunk size to 2 bytes
            instr.IoSegmentSize = 2;
            var response = instr.QueryString("*IDN?");
            instr.Events.ReadSegmentHandler = null;

            // Close the session
            instr.Dispose();
        }
    }
}
```

If you start it, you might wonder (or maybe not): why is the `args.TotalSize = None`? The reason is, in this particular case the RsInstrument does not know the size of the complete response up-front. However, if you use the same mechanism for transfer of a known data size (for example, a file transfer), you get the information about the total size too, and hence you can calculate the progress as:

*progress [pct] = 100 * args.TransferredSize / args.TotalSize*

The following example transfers a file from instrument to the PC and back. In the callback, we skip showing the data, since it's a quite big binary stream. Depending on the file size, adjust the `IoSegmentSize` and the `Thread.Sleep()` delay in the callback:

```
// Event handlers by reading - writing and reading big files

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void MyRwHandler(object sender, InstrSegmentEventArgs args)
        {
            Console.Write("\nSegment: " + args.SegmentIx);
            Console.Write(", transferred bytes: " + args.TransferredSize);
            Console.Write(", total size: " + args.TotalSize);
```

(continues on next page)

(continued from previous page)

```
            if (args.Finished)
            {
                Console.WriteLine("\n\nEnd of Transfer");
            }
            else
            {
                // Pause here for 1ms after each segment read
                Thread.Sleep(1);
            }
        }

        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            // Set data chunk size to 100 000 bytes
            instr.IoSegmentSize = 100000;
            string pcFile = @"c:\temp\bigFile.bin";
            string pcFileBack = @"c:\temp\bigFileBack.bin";
            string instrFile = @"/var/user/bigFile.bin";

            Console.WriteLine("\n\nTransferring file from PC to the Instrument");

            instr.Events.WriteSegmentHandler = MyRwHandler;
            instr.File.FromPcToInstrument(pcFile, instrFile);
            instr.Events.WriteSegmentHandler = null;

            Console.WriteLine("\n\nTransferring the file back from Instrument to the PC
↪");

            instr.Events.ReadSegmentHandler = MyRwHandler;
            instr.File.FromInstrumentToPc(instrFile, pcFileBack);
            instr.Events.ReadSegmentHandler = null;

            // Close the session
            instr.Dispose();
        }
    }
}
```

CHAPTER

SIXTEEN

14. MULTITHREADING

You are at the party, many people talking over each other. Not every person can deal with such crosstalk, neither can measurement instruments. For this reason, RsInstrument has a feature of scheduling the access to your instrument by using so-called **Locks**. Locks make sure that there can be just one client at a time 'talking' to your instrument. Talking in this context means completing one communication step - one command write or write/read or write/read/error check.

To describe how it works, and where it matters, we take two typical multithread scenarios:

16.1 One instrument session, accessed from multiple threads

You are all set - the lock is a part of your instrument session. Check out the following example - it executes properly, although the instrument might get 10 queries at the same time:

```
// Multiple threads are accessing one RsInstrument object

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr = new RsInstrument("TCPIP::192.168.1.100::INSTR");

            Parallel.For(1, 10,
                x =>
                {
                    Console.WriteLine("Query {0} started...", x);
                    var response = instr.QueryString("*IDN?");
                    Console.WriteLine("...Query {0} finished.", x);
                });

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();
```

(continues on next page)

(continued from previous page)

```
            // Close the session
            instr.Dispose();
        }
    }
}
```

16.2 More instrument sessions, accessed from multiple threads

We have two objects, `instr1` and `instr2` talking to the same instrument from multiple threads. The `instr1` and `instr2` objects have two independent locks. We bring them in sync with this call:

```
instr2.AssignLock(instr1.GetLock());
```

Check out the following example:

```
// Two RsInstrument objects are used in multiple threads and are accessing the same␣
↪instrument

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.RsInstrument;

namespace Examples
{
    class Program
    {
        static void Main()
        {
            RsInstrument instr1 = new RsInstrument("TCPIP::192.168.1.100::INSTR");
            RsInstrument instr2 = new RsInstrument(instr1.Session);
            instr1.VisaTimeout = 200;
            // Synchronize the locks of both sessions
            instr2.AssignLock(instr1.GetLock());

            Parallel.For(1, 20,
                x =>
                {
                    if (x % 10 == 0)
                    {
                        // For each even x number, query the instr1
                        Console.WriteLine("Instr1 {0} started...", x);
                        var response = instr1.QueryString("*IDN?");
                        Console.WriteLine("...Instr1 Query {0} finished.", x);
                    }
                    else
                    {
```

(continues on next page)

(continued from previous page)

```
                        // For each odd x number, query the instr2
                        Console.WriteLine("Instr2 {0} started...", x);
                        var response = instr2.QueryString("*IDN?");
                        Console.WriteLine("...Instr2 Query {0} finished.", x);
                    }
                });

            Console.WriteLine("\nPress any key ...");
            Console.ReadKey();

            // Close the sessions
            instr2.Dispose();
            instr1.Dispose();
        }
    }
}
```

Note: If you want to simulate some party crosstalk, delete this line:

```
instr2.AssignLock(instr1.GetLock());
```

Although the `instr1` will still schedule its instrument access, the `instr2` will be doing the same at the same time, which will then lead to all that fun stuff happening.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search