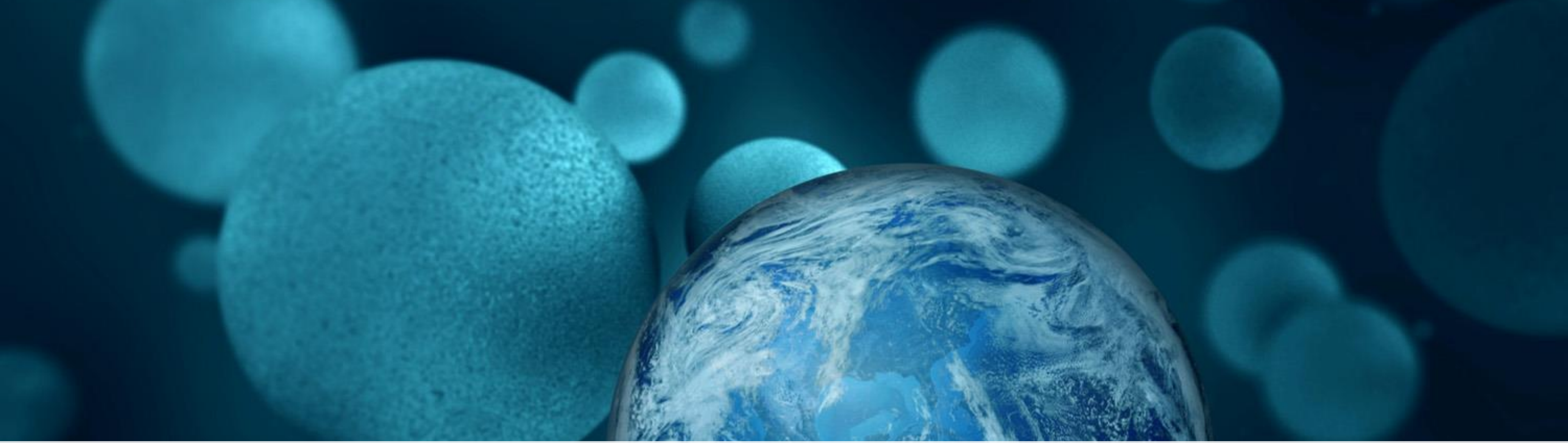


Thermo Fisher
SCIENTIFIC

Applied API Training for Exactive Series software

Revised version of May 2018, FGC/AK

The world leader in serving science

Introduction

- **API = Application programming interface**
  - Something that allows third-party applications to link a provided feature set
  - Examples: Windows programming interface; browser plugins; bluetooth remote control in your car

- **Exactive Series instruments introduced an API in 2.1 (ASMS 2012)**
  - enable customers to develop new features / techniques
  - inspect data on the fly

Aim of the training

- **Understand how deep you can dig into instrument details**
- **Understand how to drive the instrument**
  - and what is the best approach
- **Learn how to write API-accessing programs**
- **Find new ideas what you can do with current instrumentation**

The training will not help for (detailed) questions about

- Xcalibur / Foundation
- Running / operating / calibrating the instrument
- Specific hardware elements in the MS

Topics

- Recap of .NET and C# features needed to understand the talk
- Setting up a work bench
- Connecting to the API
- Handling the connection to the instrument
- Gathering data
- Placing scans
- Replacing inclusion and exclusion lists
- Setting individual values
- Running methods

Recap of .NET and C# features needed to understand the talk

Some understanding should be present on

- **Windows OS**
- **Xcalibur**
  - QualBrowser
  - RAW files in general
  - setting up a method
  - setting up an acquisition
- **.NET**
  - 2.x knowledge will do for reading
  - 4.x is best for writing, Exactive still uses 2.x interfaces, to support this use app.config entry `<startup useLegacyV2RuntimeActivationPolicy="true">`
- **C# >= 2.0**
  - Solid knowledge required on events, event handlers,
  - knowledge about multithreading helpful
  - Other languages supported: MatLab, PowerShell, Managed C++, all .NET languages

Setting up a work bench

- **Always install newest software!**
- **Exactive Series instrument is required**
- **Windows software**
  - Windows 7 or Windows 10 running the MS driver, use Tune for testing communication
  - Visual Studio 2010+ (2015 is recommended)
  - Requires an Exactive Series API license
  - Example program helps
- **Use present documentation**
  - Regular Exactive software installs C:\Xcalibur\system\Exactive\bin\ESAPI_Help.zip containing documentation about call interfaces etc.

Connecting to the API (Hands-on)

- **Fire up instrument and Exactive Tune, make sure there is a connection**
- **Open VS, create a project (C#, Windows, Classic Desktop, Console application, implies MSIL, 32 bit)**
- **Edit app.config**

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- If starting under CLR4, we want to load all CLR2 assemblies -->
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <!– Optional: Prefer CLR4 over CLR2. This is a bit odd for CLR4 present with Foundation 3.0 installed. But if Foundation 3.1 is installed, it requires CLR4 for Foundation.IO. -->
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
    <supportedRuntime version="v2.0.50727"/>
  </startup>
</configuration>
```

- **Open ESAPI_Help.zip, open chm file and copy the initial example into a new class, e.g. „Connect“**
- **References are missing: Add some dependencies from Exactive bin folder**
  - API-1.0.dll
  - API-1.1.dll
  - ESAPI-1.0.dll
  - ESAPI-1.1.dll
- **Resolve missing references**
- **Instantiate and execute class in Program.cs**

```
new Connect().DoJob();
Console.WriteLine("Press any key to continue...");
Console.ReadKey();
```

- **Run**

Connecting to the API the old way (Hands-on)

- **Replace GetApiInstance code by**

  ```
  Type type = Type.GetTypeFromProgID("Thermo Exactive.API_Clr2_32_V1", true);
  object o = Activator.CreateInstance(type);
  return (IInstrumentAccessContainer) o;
  ```

- **Run**

- **Background**

  ProgIDs refer to the so called COM concept of Microsoft, a CORBA-like implementation of remote access architecture.

  COM allows to link even into other processes, breaking the boundary of 32bit/64bit and .NET2/.NET4, but also throw errors that are not understandable and can be influenced by system-wide configuration settings, that may stop proper working.

  Moreover, Exactive‘s architecture makes the COM server being an inproc-server, meaning that the bitness and .NET-version restrictions are still in place with the advantage of immediate responses and low system resources usage. COM magic inside .NET let you just *assume* that it would work.

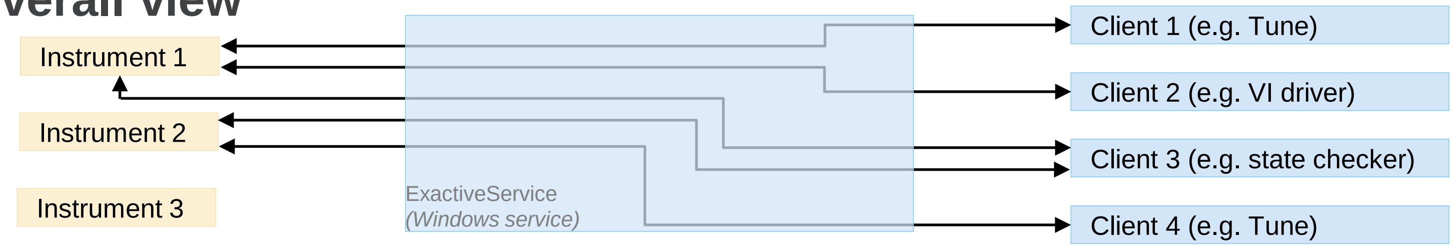
Connecting to the API (best-practice for training)

- **Establish your own standard routine to get access to the instrument**
    - Safely assume that the first instrument is the only instrument

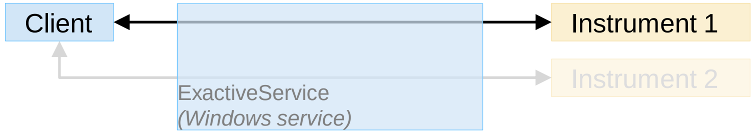
```
using System;
using System.Reflection;
using Microsoft.Win32;
using Thermo.Interfaces.ExactiveAccess_V1;
using Thermo.Interfaces.InstrumentAccess_V1;

namespace ProgramNameSpace
{
    static class Connection
    {
        const string InstrumentName = "Thermo Exactive";
        const string ApiFileNameDescriptor = "ApiFileName_Clr2_32_V1";
        const string ApiClassNameDescriptor = "ApiClassName_Clr2_32_V1";
        static private IInstrumentAccessContainer m_container = null;

        static private IInstrumentAccessContainer GetContainer()
        {
            string baseName = @"SOFTWARE\Finnigan\Xcalibur\Devices\" + InstrumentName;
            using (RegistryKey key = Registry.LocalMachine.OpenSubKey(baseName))
            {
                if (key != null)
                {
                    string asmName = (string) key.GetValue(ApiFileNameDescriptor, null);
                    string typeName = (string) key.GetValue(ApiClassNameDescriptor, null);
                    if (!string.IsNullOrEmpty(asmName) && !string.IsNullOrEmpty(typeName))
                    {
                        Assembly asm = Assembly.LoadFrom(asmName);
                        return (IInstrumentAccessContainer) asm.CreateInstance(typeName);
                    }
                }
            }
            throw new Exception("Cannot find API information of instrument \"" + InstrumentName + "\" in the registry.");
        }

        // Returns an implementation of the first instrument of the API that REQUIRES a call to Dispose finally.
        static internal IExactiveInstrumentAccess GetFirstInstrument()
        {
            if (m_container == null)
                m_container = GetContainer();

            return (IExactiveInstrumentAccess) m_container.Get(1);
        }
    }
}
```

Handling the connection to the instrument (background)

- **ExactiveService is„man in the middle“**
  - Connections are established on demand of clients and instruments independently
  - Service doesn‘t keep (much) state information of instrument → clients have to wait for feedback of actions
  - Special classes in API maintain some overall state, but individual values require event-driven waits
- **Overall view**



- **Individual view (client‘s perspective)**
  - typically, only one instrument is connected



- **Good practice**
  - Never use Thread.Sleep (`Thread.Sleep(n) → Thread.CurrentThread.Join(n)`)
  - Don‘t poll (use event driven model)
  - Always consider a disconnected instrument

Handling the connection to the instrument (Hands-on)

- **Create a standard API program and write a handler for instrument connection changes and compare with Tune‘s instrument state on instrument reboot**

```
class EventDriven
{
    // In Main(), call new EventDriven().DoJob()
    internal void DoJob()
    {
        using (IExactiveInstrumentAccess instrument = Connection.GetFirstInstrument())
        {
            Console.WriteLine("Waiting 60 seconds for connection changes using event listening...");
            // Check the connection initially (after setting up the handler for race conditions), but let the API announce connection changes
            instrument.ConnectionChanged += Instrument_ConnectionChanged;
            Thread.CurrentThread.Join(60000);
            instrument.ConnectionChanged -= Instrument_ConnectionChanged;
            Console.WriteLine("\n" + DateTime.Now.ToString("HH:mm:ss,fff ") + instrument.InstrumentName + " connected:" + instrument.Connected);
        }
    }

    private void Instrument_ConnectionChanged(object sender, EventArgs e)
    {
        IExactiveInstrumentAccess instrument = (IExactiveInstrumentAccess) sender;
        Console.WriteLine("\n" + DateTime.Now.ToString("HH:mm:ss,fff ") + instrument.InstrumentName + " connected:" + instrument.Connected);
    }
}
```

- **First result: Connected doesn‘t mean we can use the system as with Tune, it is just „connected“**

  Tune grants user access later than instrument has been connected.

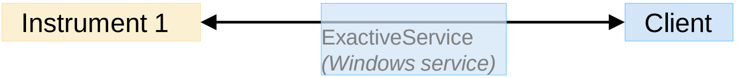
Handling the connection to the instrument properly (Hands-on)

- **Rely on Control.Acquisition to maintain a proper connection, observe in Tune a reboot of the instrument**

```
class Properly
{
   internal void DoJob()
   {
      using (IExactiveInstrumentAccess instrument = Connection.GetFirstInstrument())
      {
         Console.WriteLine("Waiting 60 seconds for connection changes using event listening...");
         // Best way is to use instrument.Control.Acquisition.WaitFor or a pure event handler of StateChange, but for demonstration, we display the changes in ConnectionChanged and StateChanged.
         instrument.Control.Acquisition.StateChanged += Acquisition_StateChanged;
         instrument.ConnectionChanged += Instrument_ConnectionChanged;
         Thread.CurrentThread.Join(60000);
         instrument.ConnectionChanged -= Instrument_ConnectionChanged;
         instrument.Control.Acquisition.StateChanged -= Acquisition_StateChanged;
      }
   }

   private void Instrument_ConnectionChanged(object sender, EventArgs e)
   {
      IExactiveInstrumentAccess instrument = (IExactiveInstrumentAccess) sender;
      Console.WriteLine("\n" + DateTime.Now.ToString("HH:mm:ss,fff ") + instrument.InstrumentName + " connected:" + instrument.Connected);
   }

   private void Acquisition_StateChanged(object sender, StateChangedEventArgs e)
   {
      Console.WriteLine(DateTime.Now.ToString("HH:mm:ss,fff ") + "Overall description:     " + e.State.Description);
      Console.WriteLine(DateTime.Now.ToString("HH:mm:ss,fff ") + "Instrument mode:         " + e.State.SystemMode);
      Console.WriteLine(DateTime.Now.ToString("HH:mm:ss,fff ") + "Instrument system state: " + e.State.SystemState);
   }
}
```

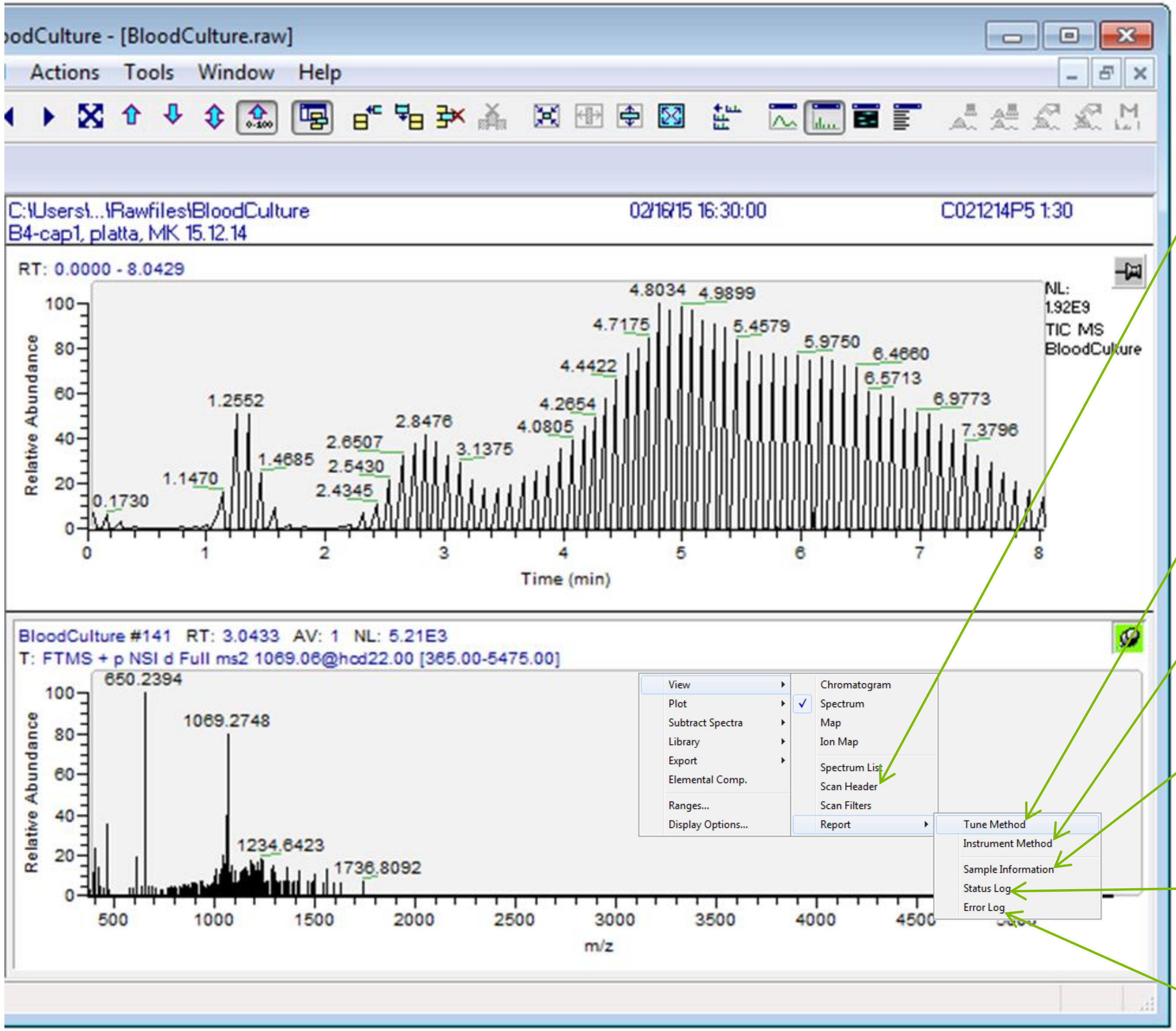
Gathering data of Exactive (Background)

- **Instrument gathers data all the time except if Off/Standby**
- **„External“ programs write data on their own to raw files**
- **Service distributes gathered data <u>concurrently</u> to all clients**



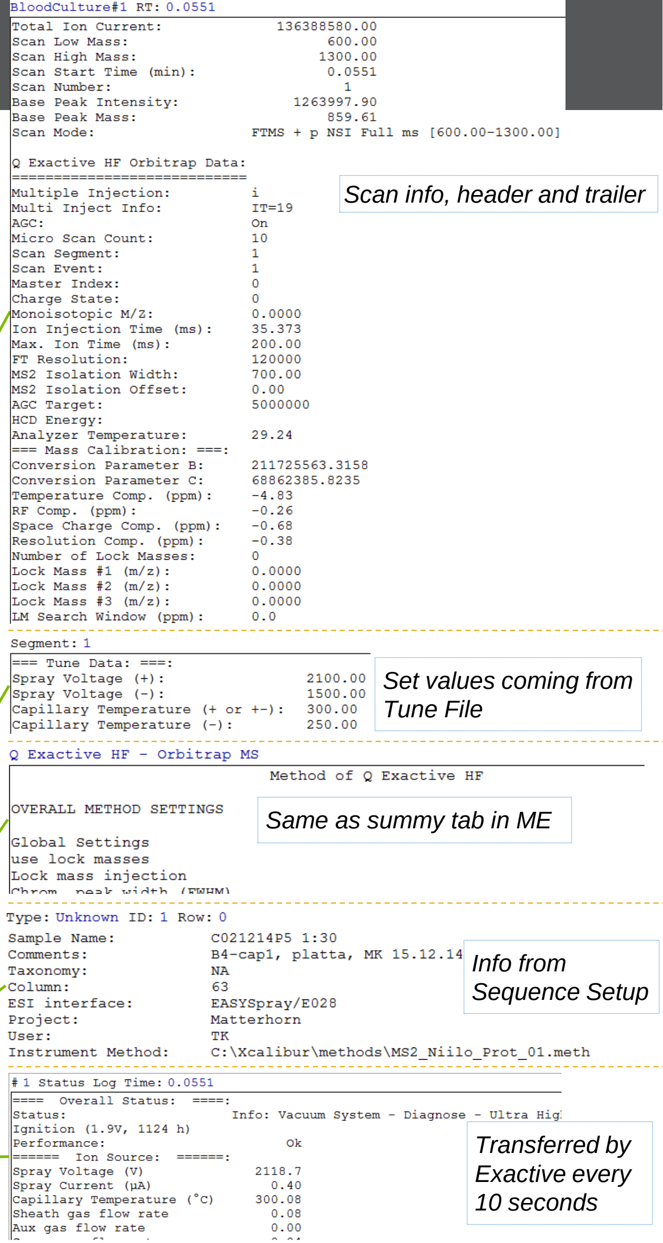
  - includes MS spectra
  - includes analog channels
- **<u>No license</u> is <u>required</u> for API programs to listen to data streams**
- **Data is sent in packets, packets are placed in shared memory**
  - data is compressed in a proprietary format
  - data is expanded in API during access
  - instrument sends data for all scans shown in Tune
- **For an acquisition, start and end yield separate events**

Gathering data of Exactive (Background)



C:\Users\...\Rawfiles\BloodCulture 02/16/15 16:30:00 C021214P5 1:30
B4-cap1, platta, MK 15.12.14

BloodCulture #141 RT: 3.0433 AV: 1 NL: 5.21E3
T: FTMS + p NSI d Full ms2 1069.06@hcd22.00 [365.00-5475.00]



```
BloodCulture#1 RT: 0.0551
Total Ion Current:              136388580.00
Scan Low Mass:                        600.00
Scan High Mass:                      1300.00
Scan Start Time (min):                0.0551
Scan Number:                               1
Base Peak Intensity:              1263997.90
Base Peak Mass:                       859.61
Scan Mode:                  FTMS + p NSI Full ms [600.00-1300.00]

Q Exactive HF Orbitrap Data:
============================
Multiple Injection:         i
Multi Inject Info:          IT=19
AGC:                        On
Micro Scan Count:           10
Scan Segment:               1
Scan Event:                 1
Master Index:               0
Charge State:               0
Monoisotopic M/Z:           0.0000
Ion Injection Time (ms):    35.373
Max. Ion Time (ms):         200.00
FT Resolution:              120000
MS2 Isolation Width:        700.00
MS2 Isolation Offset:       0.00
AGC Target:                 5000000
HCD Energy:
Analyzer Temperature:       29.24
=== Mass Calibration: ===:
Conversion Parameter B:     211725563.3158
Conversion Parameter C:     68862385.8235
Temperature Comp. (ppm):    -4.83
RF Comp. (ppm):             -0.26
Space Charge Comp. (ppm):   -0.68
Resolution Comp. (ppm):     -0.38
Number of Lock Masses:      0
Lock Mass #1 (m/z):         0.0000
Lock Mass #2 (m/z):         0.0000
Lock Mass #3 (m/z):         0.0000
LM Search Window (ppm):     0.0
```

Scan info, header and trailer

```
Segment: 1
=== Tune Data: ===:
Spray Voltage (+):                     2100.00
Spray Voltage (-):                     1500.00
Capillary Temperature (+ or +-):       300.00
Capillary Temperature (-):             250.00
```

Set values coming from Tune File

```
Q Exactive HF - Orbitrap MS
                        Method of Q Exactive HF

OVERALL METHOD SETTINGS

Global Settings
use lock masses
Lock mass injection
```

Same as summy tab in ME

```
Type: Unknown ID: 1 Row: 0
Sample Name:            C021214P5 1:30
Comments:               B4-cap1, platta, MK 15.12.14
Taxonomy:               NA
Column:                 63
ESI interface:          EASYSpray/E028
Project:                Matterhorn
User:                   TK
Instrument Method:      C:\Xcalibur\methods\MS2_Niilo_Prot_01.meth
```

Info from Sequence Setup

```
# 1 Status Log Time: 0.0551
====  Overall Status:  ====:
Status:                    Info: Vacuum System - Diagnose - Ultra Hig
Ignition (1.9V, 1124 h)
Performance:                           Ok
======  Ion Source:  ======:
Spray Voltage (V)              2118.7
Spray Current (µA)               0.40
Capillary Temperature (°C)     300.08
Sheath gas flow rate             0.08
Aux gas flow rate                0.00
```

Transferred by Exactive every 10 seconds

May contain info about issues messing up the RAW file

Gathering data of Exactive (Hands-on, excercise)

- **Bind to instrument and do an initial listening to acquistion start, end, and scan generation**

```
using IMsScan = Thermo.Interfaces.InstrumentAccess_V2.MsScanContainer.IMsScan;
namespace KeepInstrumentConnection
{
    class DataReceiver
    {
        internal void DoJob()
        {
            using (IExactiveInstrumentAccess instrument = Connection.GetFirstInstrument())
            {
                IMsScanContainer orbitrap = instrument.GetMsScanContainer(0);
                Console.WriteLine("Waiting 60 seconds for scans on detector " + orbitrap.DetectorClass + "...");
                orbitrap.AcquisitionStreamOpening += Orbitrap_AcquisitionStreamOpening;
                orbitrap.AcquisitionStreamClosing += Orbitrap_AcquisitionStreamClosing;
                orbitrap.MsScanArrived += Orbitrap_MsScanArrived;
                Thread.CurrentThread.Join(60000);
                orbitrap.MsScanArrived -= Orbitrap_MsScanArrived;
                orbitrap.AcquisitionStreamClosing -= Orbitrap_AcquisitionStreamClosing;
                orbitrap.AcquisitionStreamOpening -= Orbitrap_AcquisitionStreamOpening;
            }
        }

        private void Orbitrap_MsScanArrived(object sender, MsScanEventArgs e)
        {
            using (IMsScan scan = (IMsScan) e.GetScan())    // caution! You must dispose this, or you block shared memory!
            {
                Console.WriteLine("\n{0:HH:mm:ss,fff} scan with {1} centroids arrived", DateTime.Now, scan.CentroidCount);
            }
        }

        private void Orbitrap_AcquisitionStreamClosing(object sender, EventArgs e)
        {
            Console.WriteLine("\n{0:HH:mm:ss,fff} {1}", DateTime.Now, "Acquisition stream closed (end of method)");
        }

        private void Orbitrap_AcquisitionStreamOpening(object sender, MsAcquisitionOpeningEventArgs e)
        {
            Console.WriteLine("\n{0:HH:mm:ss,fff} {1}", DateTime.Now, "Acquisition stream opens (start of method)");
        }
    }
}
```

Gathering data of Exactive Series instruments (Background)

- **Exactive scan data (IMsScan : IDisposable) contains**
  - spectrum (IEnumerable of centroids)
    - m/z
    - intensity
    - charge
    - resolution
    - profile shape (optional, must be accessed **during** access of spectrum enumeration)
    - flags: IsExceptional, IsFragmented, IsMerged, IsMonoisotopic, IsReferenced
  - noise band (IEnumerable of noise-nodes, to be considered as polygon)
    - m/z
    - intensity
    - baseline (empty for Exactive)
  - meta data
    - Generic header (common)
    - Trailer
    - Status
    - TuneData
    - fixed for Acquisition

Gathering data (Hands-on, exercise and demonstration)

- **Output of scan without meta data and profiles is straight-forward**

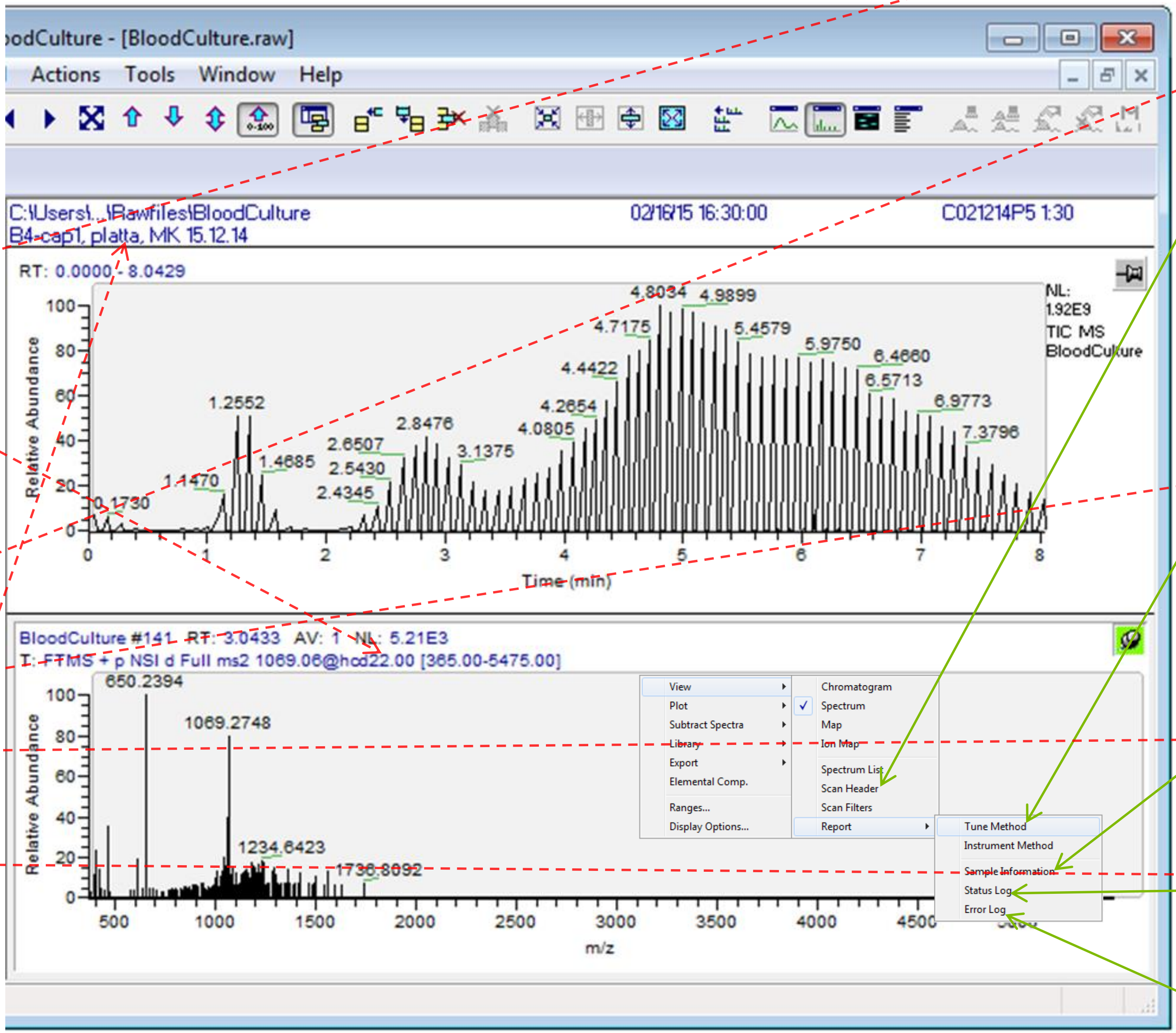
```
private void Orbitrap_MsScanArrived(object sender, MsScanEventArgs e)
{
   using (IMsScan scan = (IMsScan) e.GetScan())    // caution! You must dispose this, or you block shared memory!
   {  // block to replace below under “Profile:”
      Console.WriteLine("\n{0:HH:mm:ss,fff} scan with {1} centroids arrived", DateTime.Now, scan.CentroidCount);
      Console.WriteLine("Noise: " + string.Join("; ", scan.NoiseBand.Take(5).Select(n => string.Format("{0:F2},{1:0.0e0}", n.Mz, n.Intensity))));
      Console.WriteLine("Centroids: " + string.Join("; ", scan.Centroids.Take(5).Select(n => string.Format("{0:F2},{1:0.0e0},z={2},R={3}", n.Mz, n.Intensity, n.Charge, n.Resolution))));
   }
}
```

- **Maintain a separate thread for data processing (e.g. using a queue): you can block the service otherwise**
- **Profile data must be accessed only within enumeration of centroids: Consider copying data**

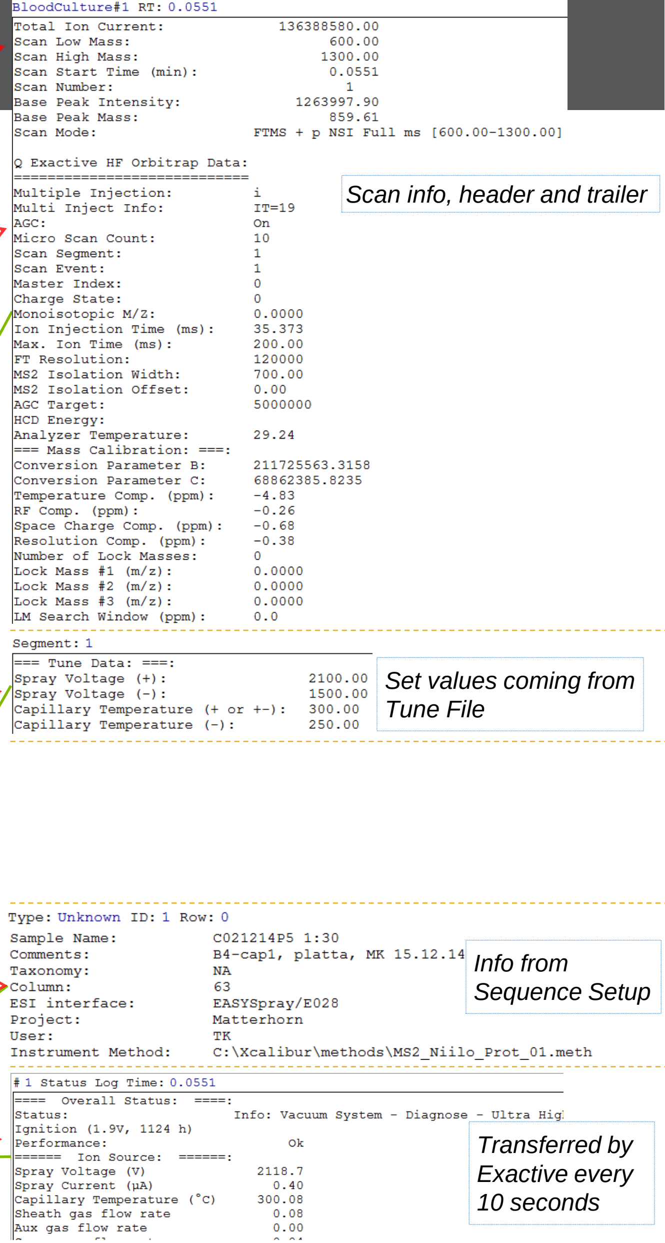
```
Console.WriteLine("\n{0:HH:mm:ss,fff} scan with {1} centroids arrived", DateTime.Now, scan.CentroidCount);
#if ProfileAccessInsideEnumeration
   int max = 5;
   foreach (ICentroid c in scan.Centroids)
   {
      if (max-- == 0)
         break;
      Console.Write("{0:F4},{1:0.0e0},z={2}  :  ", c.Mz, c.Intensity, c.Charge);
      Console.WriteLine(string.Join("; ", c.Profile.Take(5).Select(n => string.Format("{0:F4},{1:0.0e0}", n.Mz, n.Intensity))));
   }
#elif ProfileAccessOutsideEnumeration
   ICentroid[] list = scan.Centroids.ToArray();
   for (int i = 0; i < Math.Min(5, list.Length); i++)
   {
      ICentroid c = list[i];
      Console.Write("{0:F4},{1:0.0e0},z={2}  :  ", c.Mz, c.Intensity, c.Charge);// works
      Console.WriteLine(string.Join("; ", c.Profile.Take(5).Select(n => string.Format("{0:F4},{1:0.0e0}", n.Mz, n.Intensity)))); // crashes enumeration happened in ToArray call
   }
#else
   // Create an array where the profile is copied on enumeration
   Tuple<ICentroid,IMassIntensity[]>[] list = scan.Centroids.Select(n => new Tuple<ICentroid, IMassIntensity[]>(n, (IMassIntensity[]) n.Profile.Clone())).ToArray();
   for (int i = 0; i < Math.Min(5, list.Length); i++)
   {
      Tuple<ICentroid, IMassIntensity[]> tuple = list[i];
      Console.Write("{0:F4},{1:0.0e0},z={2}  :  ", tuple.Item1.Mz, tuple.Item1.Intensity, tuple.Item1.Charge);  // works
      Console.WriteLine(string.Join("; ", tuple.Item2.Take(5).Select(n => string.Format("{0:F4},{1:0.0e0}", n.Mz, n.Intensity)))); // works
   }
#endif
```

Gathering meta data (Background)

- **Meta data is send by the instrument**
  - on start of an acquisition (resend on instrument restart)
  - on arrival of a new scan
- **CommonInformation**
  - common for all instruments
  - once per scan
  - Also defines the scan filter on top of each spectrum
- **SpecificInformation**
  - merges the following
    - Trailer (per scan)
    - TuneData (from the last occurance of the **first** scan of an acquisition)
    - AcquisitionFixed (from last acquisition stream opening)
    - status log (comes with a scan roughly every 10 seconds)



```
odCulture - [BloodCulture.raw]
C:\Users\...\Rawfiles\BloodCulture        02/16/15 16:30:00        C021214P5 1:30
B4-cap1, platta, MK 15.12.14
RT: 0.0000 - 8.0429
BloodCulture #141  RT: 3.0433  AV: 1  NL: 5.21E3
T: FTMS + p NSI d Full ms2 1069.06@hcd22.00 [365.00-5475.00]
```



```
BloodCulture#1 RT: 0.0551
Total Ion Current:                   136388580.00
Scan Low Mass:                             600.00
Scan High Mass:                           1300.00
Scan Start Time (min):                     0.0551
Scan Number:                                    1
Base Peak Intensity:                   1263997.90
Base Peak Mass:                            859.61
Scan Mode:                   FTMS + p NSI Full ms [600.00-1300.00]

Q Exactive HF Orbitrap Data:
============================
Multiple Injection:          i
Multi Inject Info:           IT=19
AGC:                         On
Micro Scan Count:            10
Scan Segment:                1
Scan Event:                  1
Master Index:                0
Charge State:                0
Monoisotopic M/Z:            0.0000
Ion Injection Time (ms):     35.373
Max. Ion Time (ms):          200.00
FT Resolution:               120000
MS2 Isolation Width:         700.00
MS2 Isolation Offset:        0.00
AGC Target:                  5000000
HCD Energy:
Analyzer Temperature:        29.24
=== Mass Calibration: ===:
Conversion Parameter B:      211725563.3158
Conversion Parameter C:      68862385.8235
Temperature Comp. (ppm):     -4.83
RF Comp. (ppm):              -0.26
Space Charge Comp. (ppm):    -0.68
Resolution Comp. (ppm):      -0.38
Number of Lock Masses:       0
Lock Mass #1 (m/z):          0.0000
Lock Mass #2 (m/z):          0.0000
Lock Mass #3 (m/z):          0.0000
LM Search Window (ppm):      0.0
```

Scan info, header and trailer

```
Segment: 1
=== Tune Data: ===:
Spray Voltage (+):                  2100.00
Spray Voltage (-):                  1500.00
Capillary Temperature (+ or +-):    300.00
Capillary Temperature (-):          250.00
```

Set values coming from Tune File

```
Type: Unknown ID: 1 Row: 0
Sample Name:            C021214P5 1:30
Comments:               B4-cap1, platta, MK 15.12.14
Taxonomy:               NA
Column:                 63
ESI interface:          EASYSpray/E028
Project:                Matterhorn
User:                   TK
Instrument Method:      C:\Xcalibur\methods\MS2_Niilo_Prot_01.meth
```

Info from Sequence Setup

```
# 1 Status Log Time: 0.0551
====  Overall Status:  ====:
Status:                          Info: Vacuum System - Diagnose - Ultra Hig
Ignition (1.9V, 1124 h)
Performance:                             Ok
======  Ion Source:  ======:
Spray Voltage (V)                    2118.7
Spray Current (µA)                     0.40
Capillary Temperature (°C)           300.08
Sheath gas flow rate                   0.08
Aux gas flow rate                      0.00
```

Transferred by Exactive every 10 seconds

May contain info about issues messing up the RAW file

Gathering meta data of Exactive Series instruments

- **Bind to instrument and establish meta data listeners**

```
using IMsScan = Thermo.Interfaces.InstrumentAccess_V2.MsScanContainer.IMsScan;
internal void DoJob()
{
   …
   IMsScanContainer orbitrap = instrument.GetMsScanContainer(0);
   orbitrap.AcquisitionStreamOpening += Orbitrap_AcquisitionStreamOpening;
   orbitrap.MsScanArrived += Orbitrap_MsScanArrived;
   Thread.CurrentThread.Join(60000);
   orbitrap.MsScanArrived -= Orbitrap_MsScanArrived;
   orbitrap.AcquisitionStreamOpening -= Orbitrap_AcquisitionStreamOpening;
   …
}

private void Orbitrap_MsScanArrived(object sender, MsScanEventArgs e)
{
   using (IMsScan scan = (IMsScan) e.GetScan())// caution! You must dispose this, or you block shared memory!
   {
      Console.WriteLine("\n{0:HH:mm:ss,fff} scan with {1} centroids arrived", DateTime.Now, scan.CentroidCount);
      Dump("Common", scan.CommonInformation);
      Dump("Specific", scan.SpecificInformation);
   }
}

private void Orbitrap_AcquisitionStreamOpening(object sender, MsAcquisitionOpeningEventArgs e)
{
   Console.WriteLine("\n{0:HH:mm:ss,fff} Acquisition stream opening", DateTime.Now);
   Dump("Specific", e.SpecificInformation);
}

private void Dump(string title, IInfoContainer container)
{
   Console.WriteLine(title);
   foreach (string key in container.Names)
   {
      string value;
      MsScanInformationSource source = MsScanInformationSource.Unknown;// has to match or must be Unknown
      if (container.TryGetValue(key, out value, ref source))
      {
         string descr = source.ToString();
         descr = descr.Substring(0, Math.Min(11, descr.Length));
         Console.WriteLine("   {0,-11} {1,-35} = {2}", descr, key, value);
      }
   }
}
```

Placing scans (Background)

- Requires API license
- Active userrole dictates values and value ranges of scan properties
- Possible values, ranges and help can be interrogated
- Instrument‘s execution engine pulls values for the next scan from available sources in following order
  - Custom API Scan
  - Repeating API Scan
  - Method experiment
  - Method global settings
  - Tunefile
  - Calibration file
- A Repeating API scan requires cancellation if been set
- A Custom API scan allows setting a delay time to allow setting the next Custom scan w/o internal scan
- Short before end of custom scan, an event is send by instrument to place further scans
- API scans produce same data as methods (see gathering data)

Placing scans

- **Scan properties for a new scan needs to be defined ~20ms before scan is scheduled**
  One or two scans can be already scheduled in the current execution queue.
  Use „Access Id“ property to identify the individual scans.
- **Defaults of the current tunefile are taken if specific settings are missing in scan definition**
- **Language for decimals in strings is the independent locale (similar to US)**
- **Study the examples!**

Replacing inclusion and exclusion lists (Background)

- **Tables are used to define what precursors are selected or avoided on scan construction**
  - one inclusion table shared by all experiments
  - one exclusion table shared by all experiments
  - inclusion and exclusion table don‘t share same properties / columns
- **API license required for using replace mechanism**
- **Instrument type may have an influence on properties of an inclusion/exclusion table**
- **Language for decimals in strings is the independent locale**

Replacing inclusion and exclusion lists (Hands-on)

- **Create a PRM method of one minute, set dynamic exclusion to 0, set top 5 peaks of Tune‘s current spectrum on inclusion list**
  - observe mandatory fields
- **Write a handler to replace the inclusion list (e.g. replace with new list containing only two masses)**
  - Use missing fields for defaults where possible as thumb rule
- **Run method using Tune with rawfile output**
- **Apply handler and observe values (Access Id!) in QualBrowser**
- **Check out the example**