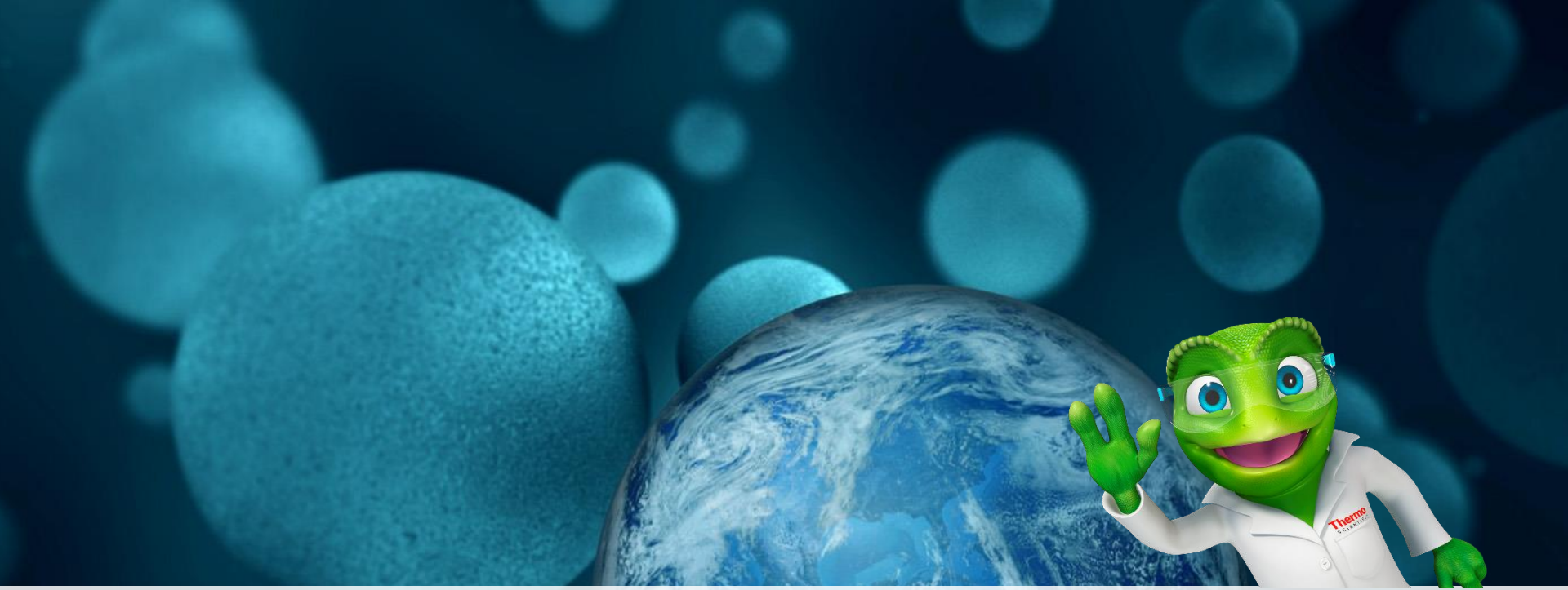


ThermoFisher SCIENTIFIC

Chromeleon DDK Development Training – Day2

Anton Kyosev, CMD Senior Staff Software Engineer
Yubo Dong, CMD Software Engineering Manager

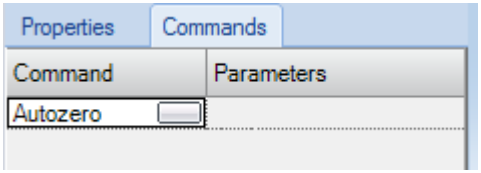
Day 2 Morning Agenda

- ❑ Driver
- ❑ Audit Trail
- ❑ Properties, Commands
- ❑ Preflight

DDK Driver Basic

- The **Driver** has one or more **Devices**. The Device represents any functional group like detector (CDet, EDet , QDet), pump (Pump_TC), Autosampler, etc. The Device (CDet) can have **Sub-Devices** (CD, CD_Analog_Out, etc.)
- A **Device** has **Properties** and **Commands**
- Devices, Properties and Commands are part of the Symbol Table
  - **Settings**: Read/Write properties
  - **Status**: Read only properties
  - **Tasks**: Commands
  - Command are executed one by one, not possible to hace two commands at the same time

Firmware 1:1 match

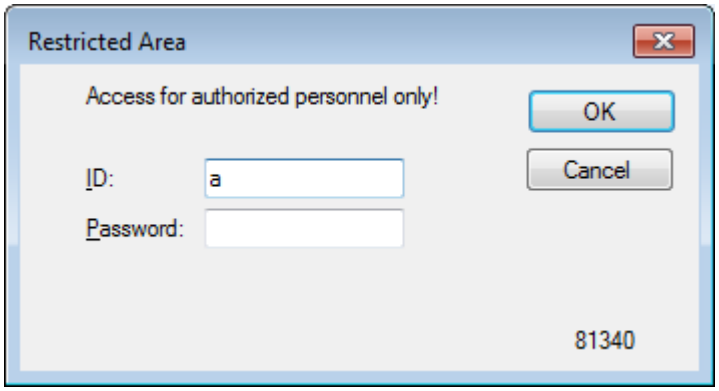


A few common identification

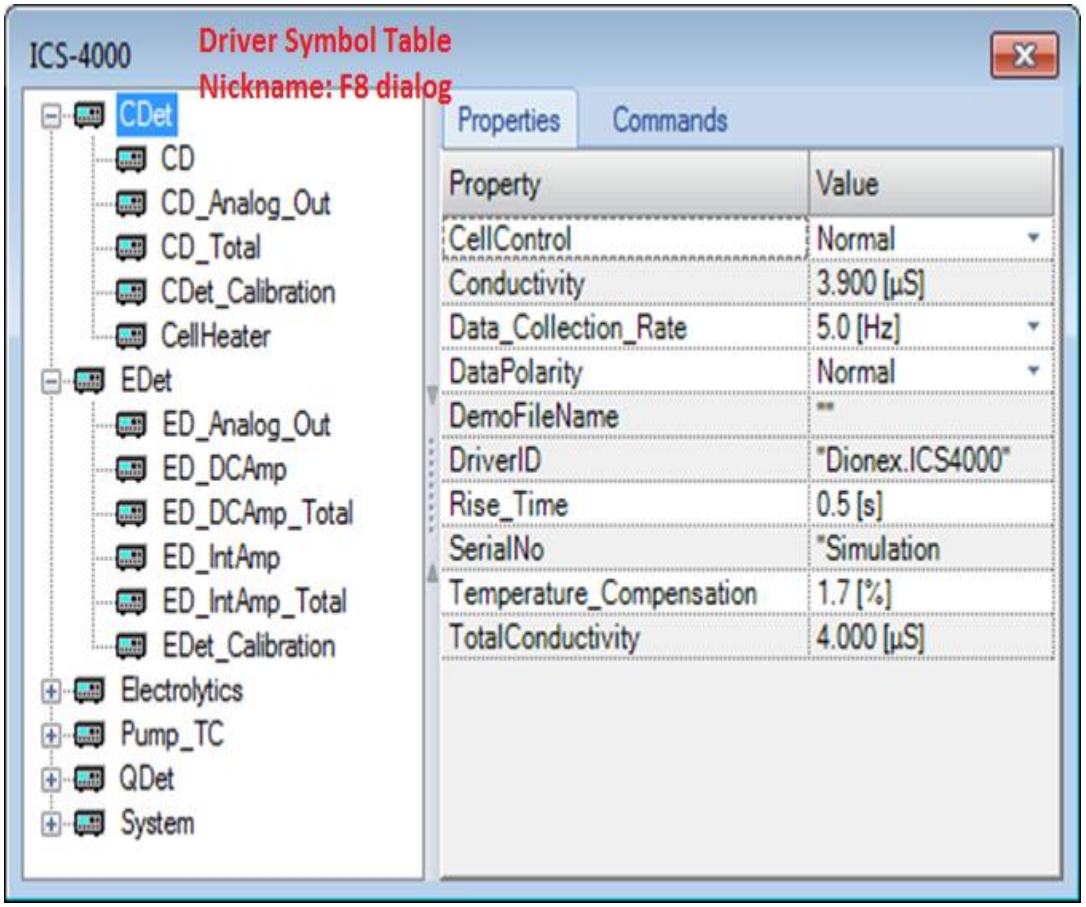
- **ModelNo** or **DeviceType:** unique name to identify a module or a device
- **Instrument Name** and **Device Name:** MUST be unique and editable
- **DriverID:** Unique for the driver, identification of a driver (coding)
- **SerialNo or SerialNumber:** only unique after physical module is shipped from manufacturer, should NOT be used as unique identification for driver code development

Driver Symbol Table

Change service level: Ctrl+Shift+Help - About Chromeleon..



F8: driver symbol table
F9: Sequence editor column content fill down



DDK Driver Symbol View Tools

- Internal developers:
  - TestHarness: InstrumentServiceWCF.Testharness.exe
- External developers:
  - SymbolTreeDump.exe

DDK Driver Namespaces

DDK Namespaces to be used by the driver and configuration plug-in

- **Dionex.Chromeleon.DDK**
  - DDK Host Interface, DDK Driver Interface, Configuration Interface
  - To be used only by driver and configuration plug-in
  - Reference CmDDK.dll

- **Dionex.Chromeleon.Symbols**
  - Chromeleon symbol Interfaces
  - Reference CmSymbols.dll

DDK Driver Interface - IDriver

- **Configuration**

  `string IDriver.Configuration` – Get/Set the driver's configuration

- **Initialization and Shutdown:**

  `void IDriver.Init(IDDK ddk)` – A driver must create its devices and properties (symbols), etc.

  `void IDriver.Exit()` – Last function called before unload

- **Connect / Disconnect:**

  `void IDriver.Connect()` – Allocate communication resources and connect to the instrument

  `void IDriver.Disconnect()` – Disconnect from instrument and free resources. Aborts a running sequence.

DDK Driver Interface – Driver Class

```
using Dionex.Chromeleon.DDK;

public interface IDriver
{
    string Configuration { get; set; }
    void Init(IDDK ddk);
    void Connect();
    void Disconnect();
    void Exit();
}

[DriverIDAttribute("MyCompany.Demo")]
public class Driver : IDriver
```

DDK Driver Interface – Configuration

```
private Config.Driver m_Config;
private bool m_IsSimulated;

string IDriver.Configuration
{
    get
    {
        if (m_Config != null)
        {
            return m_Config.XmlText;
        }
        return Config.Driver.DefaultXmlText();
    }
    set
    {
        m_Config = new Config.Driver(value);
        m_IsSimulated = m_Config.Demo.IsSimulated;
    }
}
```

If the configuration changes, the driver is destroyed and created again.

DDK Driver Interface – Create Device

```
public interface IDriver
{
    string Configuration { get; set; }
    void Init(IDDK ddk);
    void Connect();
    void Disconnect();
    void Exit();
}

public void Init(IDDK ddk)
{
  IDevice device = ddk.CreateDevice("Device_Name", "Device help text");
}
```

DDK Driver Interface – Connect & Disconnect

```
public interface IDriver
{
    string Configuration { get; set; }
    void Init(IDDK ddk);
    void Connect();
    void Disconnect();
    void Exit();  // Called when the driver is about to be destroyed
}

void IDriver.Connect()
{
    // Sets the standard driver property Connected = True
}

void IDriver.Disconnect()
{
    // Sets the standard driver property Connected = False and
    // calls AuditMessage(AuditLevel.Abort) – aborts any running Injection
    m_DDK.Disconnect();
}
```

DDK Driver Execution Audit Trails

Driver reports errors only with audit messages via `IDDK` or `IDevice`:

```
m_DDK.AuditMessage(AuditLevel.Warning, "Text");
```

```
m_Device.AuditMessage(AuditLevel.Error, "Text");
```

preferred

If an injection is running, calling audit message with audit level abort aborts the injection and the sequence.

```
m_Device.AuditMessage(AuditLevel.Abort, "Text");
```

Note: that messages sent by **`m_DDK.AuditMessage`** are only visible in the CM instrument controller configuration. Drivers should therefore use **`m_Device.AuditMessage`** where possible as **`m_Device.AuditMessage`** also writes the name of the device into the message and the CM client also shows the audit messages of all devices installed on the instrument.

Driver Property

Symbol Table – Properties

3 types:

- IIntProperty
- IDoubleProperty
- IStringProperty

```
IStringProperty Name = device.CreateProperty("Name",
                                    "Help", ddk.CreateString(100));
Name.AuditLevel = AuditLevel.Normal;
Name.Update("Value");
Trace.WriteLine(Name.Value);
```

All values (like Name.Value) are Nullable

Driver Standard Property

Chromeleon knows a set of standard properties defined in

```
public enum StandardPropertyID
{
  DriverID,
  ModelNo,
  SerialNo,
  Ready,
  . . .
}
```

The standard properties have a special meaning for Chromeleon and are expected to behave in a certain way.

Example of creating the Boolean property Ready:

```
ITypeInt type = ddk.CreateInt(0, 1);
type.AddNamedValue(false.ToString(), 0);
type.AddNamedValue(true .ToString(), 1);
IIntProperty ready = device.CreateStandardProperty(StandardPropertyID.Ready,
                                                   type);
```

Driver commands

Symbol Table – Commands

```
ICommand CommandXxx = Device.CreateCommand("Xxx",
                                           "Help Text");

CommandXxx.AuditLevel = AuditLevel.Normal;

CommandXxx.AddParameter("ParameterName",
                        "Help Text",
                         DDK.CreateString(100));
```

Driver Preflight and Events

```
private void OnCommandXxxPreflight(CommandEventArgs args)
{
  if (!IsCommunicating)
  {
    AuditMessage(AuditLevel.Error, "Not connected");
    return;
  }
}

private void OnCommandXxx(CommandEventArgs args)
{
  try
  {
    // Execute the command
  }
  catch (Exception ex)
  {
    AuditMessage(AuditLevel.Error, ex.Message);
  }
}
```

Preflighting and Real Execution

Preflighting a program with a property assignment:

```
0.000  MyProperty = 1
       End
```

Preflight Begin

- At the beginning of the preflight a run context object is created passed via `IRunContext`
- The DDK takes a snapshot of all properties in the driver and saves it to the run context (see `IRunContext.Precondition`)
- OnPreflightBegin handler is called

Preflighting a new Timestep:

- New time step is detected in DDK driver instance
- This information is sent to the DDK
- OnPreflightLatch handler is called

Preflighting and Real Execution

Preflighting the Property Assignment:

- Property assignment detected in DDK driver instance
- DDK driver instance performs basic preflight checks (check Connected property, range check)
- New property value is sent to the DDK
- Property assignment is recorded in IRunContext.ProgramSteps
- OnPreflightSetProperty is called

Preflighting Last line of a timestep

- DDK driver instance detects that current line is the last one of a given time step
- Property assignment preflight is done as described before
- Information ‘last line of this time step’ is sent to the DDK
- Plug-In’s OnPreflightSync handler is called if implemented.

Preflight End

- Plug-In’s OnPreflightEnd handler is called if implemented

Preflighting and Real Execution

Preflighting the Property Assignment:

- DDK performs basic preflight checks (check the Connected property, range check)
- Driver‘s `OnPreflightSetProperty` is called if implemented where additional verification of the new value can be made
- Property value is set in the device’s property `OnSetProperty` event handler
- New property value is sent to the DDK
- Property assignment is recorded in `IRunContext`

Preflight End

- Driver’s OnPreflightEnd handler is called if implemented

Instrument Method's Real Execution Phase:

- Driver's OnTransferPreflightToRun is called if implemented
  - Called in CmDDKHost.exe main thread
  - Run context argument contains every call into the driver since preflight begin -> Use this for download drivers!
- A new RunContext is created and sent to the DDK
- The DDK takes a snapshot of all properties in the driver and saves it to the run context (Precondition)
- Property assignment detected in device's setProperty()
- DDK performs basic online checks (check Connected property, range check)
- New property value is sent to the DDK
- Driver's OnSetProperty is called if implemented
- Driver must update property with Update(), not set by the DDK!

Preflight Example

- CmDDKExamples\Preflight\
  Implements warnings in all preflight handlers so the events can be watched

Download Driver Example

- CmDDKExamples\Download\
  Use the `IProgramStep` interface to walk through the list of events in the instrument method. The complete method will be sent to the hardware in one go in the OnTransferPreflightToRun handler.

Queue Preflighting:

- During BatchPreflightBegin a batch preflight object is created by CmDDKDrv.cdd and sent to the DDK
- Batch Preflight might contain several sequences and stand-alone instrument/emergency/smart startup/smart shutdown methods
- Driver's OnBatchPreflightXXX handlers are called if implemented
- `ISamplePreflight` provides access to the sample's properties

Queue Preflighting and Real Sequence Processing – cont'd

Sequence Pre-Execution Check:

Only injections of the sequence being started

- **Sequence Real Execution**
  - After successful pre-execution-check sequence is started
  - Driver's OnSequenceStart is called if implemented. The driver can get a reference to `ISequencePreflight` during sequence runtime from args.SequencePreflight;
  - Driver's OnSequenceChange is called if the running sequence is changed. ISequencePreflight becomes invalid after this call. The driver must update the reference to `ISequencePreflight` with the new one newSequenceArgs.SequencePreflight
  - Driver's OnSequenceEnd is called. ISequencePreflight becomes invalid after this call.

Queue Preflight Example

- CmDDKExamples\PreparingSampler\
  Implements a sampler that performs sample preparation

```
private void OnDeviceSequenceChange(SequencePreflightEventArgs oldSequenceArgs,
                                    SequencePreflightEventArgs newSequenceArgs)
{
    ISequencePreflight sequence = newSequenceArgs.SequencePreflight;

    foreach (IBatchEntryPreflight batchEntry in newSequenceArgs.SequencePreflight.Entries)
    {
        ISamplePreflight injection = (ISamplePreflight)batchEntry;
        Trace.WriteLine("Injection \"" + injection.Name + "\" -  " +
                        "Position = "  + injection.Position + ", " +
                        "Volume = "    + injection.InjectVolume.ToString());
    }
}
```

Day 2 Afternoon Agenda

- ❑ Shared Driver
- ❑ Debugging

Shared Driver

- A driver can be shared between 2 and more instruments

- Configuration

```
class MainForm : Form, IConfigurationPlugin, IConfigurationDescriptors,
                      IConfigurationPluginSharedInstruments

        // All instruments IDs the driver is attached to
        // If AttachedTo = 0011, then the driver is attached to 2
        instruments with IDs
        //                  0001 and 0010
        long IConfigurationPluginSharedInstruments.AttachedTo
        {
            get { return m_InstrumentsMap; }
            set { m_InstrumentsMap = value; }
        }
```

- Configuration

```
private IInstrumentInfo m_InstrumentInfo;  // All instruments info

IInstrumentInfo IConfigurationPluginSharedInstruments.InstrumentConfiguration
{
    set
    {
        if (m_InstrumentInfo == value)
            return;
        m_InstrumentInfo = value;
        // Initialize using:
        // m_InstrumentInfo.InstrumentMap;  // All instruments' Ids
        // m_InstrumentInfo.GetInstrumentName(InstrumentID id)
    }
}
```

Driver

ddk.InstrumentMap contains all instruments' ids the driver is shared with:

```
void IDriver.Init(IDDK ddk)
{
    long instrumentsMap = ddk.InstrumentMap.Value;

    if (instrumentsMap == (long)InstrumentID.None)
        throw new InvalidOperationException("ddk.InstrumentMap is 0");
```

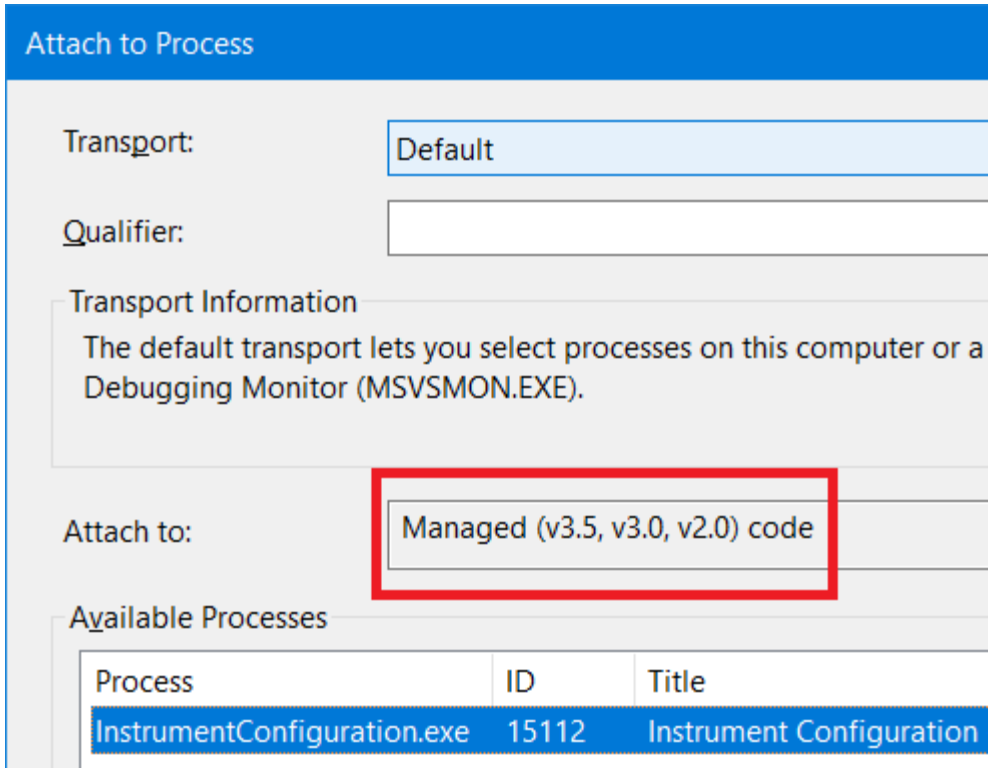
To get the instruments names use `ddk.GetInstrumentName(InstrumentID id)`

Debugging a DDK driver

- Start Instrument Controller using Chromeleon Services Manager
- Start Instrument Configuration
- Add an instrument
- Add the driver
- Use Debugger.Launch() to break into the debugger or attach the debugger to **CmDDKHost.exe** after loading the driver

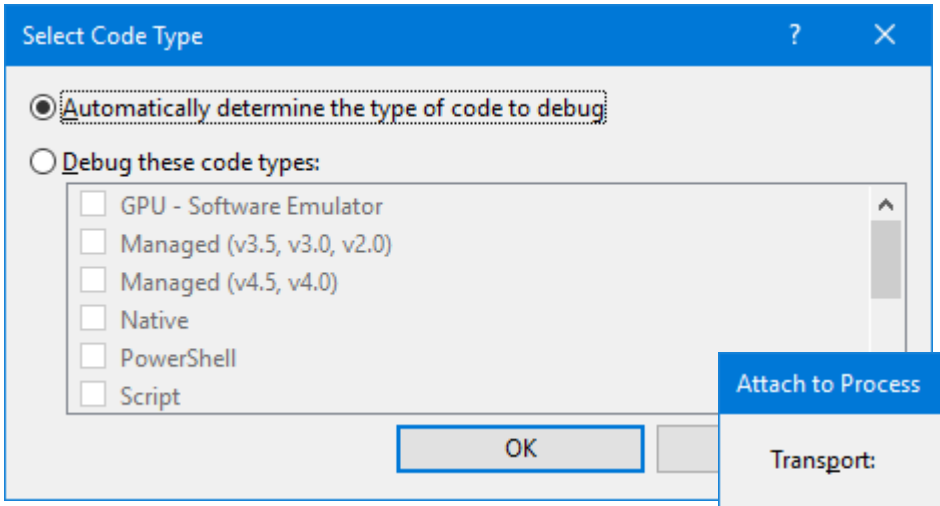
Debugging Driver Configuration

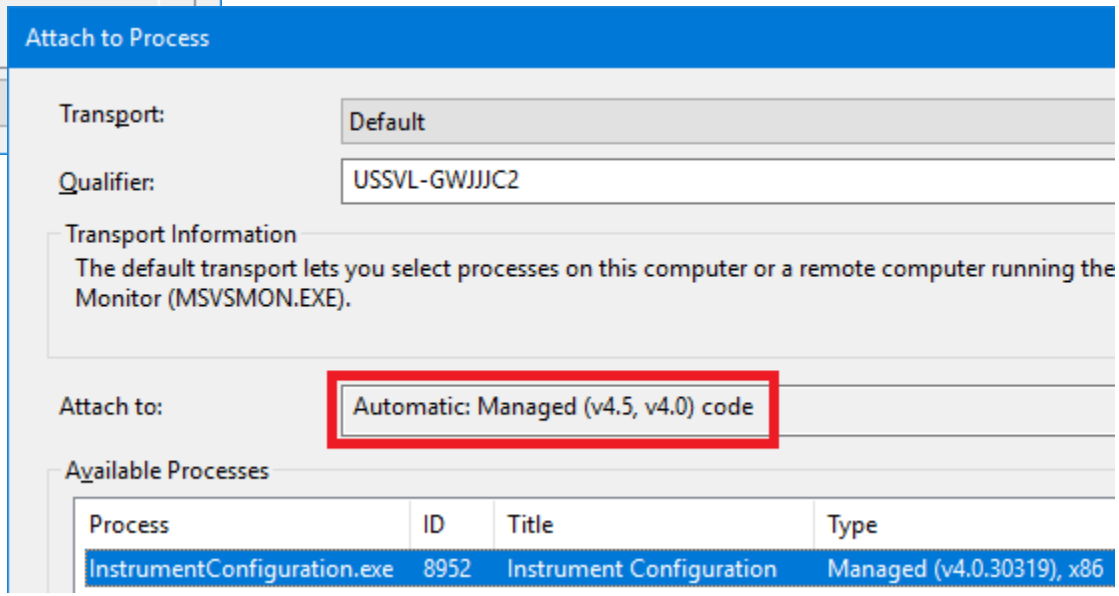
- Run Visual Studio must be run as **administrator**
- Run **InstrumentConfiguration.exe** and attach to it from Visual Studio
- Some versions of Visual Studio (before 2013) may require the debugged version of .NET to be explicitly specified:



Debugging Driver Configuration

Otherwise it is not necessary to specify the .NET version in the attached to dialog – this can be determined automatically.





If using Windows 10 and the version has to be selected, then select Managed (v4.5, v4.0)

Debugging Driver

1. Attach to the **CmDDKHost.exe**

2. Add `Debugger.Launch()` to the driver constructor to break into the debugger

```
#if DEBUG
    Debugger.Launch();
#endif
```

3. Consider using infinite timeouts for debug versions

```
public static int GetTimeout(int value)
{
#if DEBUG
    if (Debugger.IsAttached)
    {
        value = int.MaxValue;
    }
#endif
    return value;
}
```

Debugging Driver

Using `Debug.Assert` or `Debug.Fail` causes undesired behavior or does not work at all.

Use this instead:

```
#if DEBUG
    if (SomethingIsWrong)
        throw new Exception("Something Is Wrong");
#endif
```

Debugging Driver

```
[Conditional("DEBUG")]
[DebuggerStepThrough]
public static void DebuggerBreak(string text = null)
{
    Trace.WriteLine("DEBUG DebuggerBreak " + text);
    if (Debugger.IsAttached)
    {
        Debugger.Break();
    }
    else
    {
        Debugger.Launch();
    }
}

[Conditional("DEBUG")]
[DebuggerStepThrough]
public static void DebuggerBreakIfIsAttached(string text = null)
{
    Trace.WriteLine("DEBUG DebuggerBreakIfIsAttached " + text));
    if (Debugger.IsAttached)
    {
        Debugger.Break();
    }
}
```

Tools Used for Debugging with Firmware

- **Each development could have different tool to debug commands sent to firmware. Here are some tool we use in Sunnyvale team:**
  - **Internal USBTester:** Each firmware development may have a different tool for DDK driver to test firmware
  - **Bus Hound:** The Bus Hound from Perisoft is an USB and serial port monitoring software
  - **USB Monitor:** The HHD Software Device Monitoring Studio – USB Monitor is a powerful tool to monitor the communication traffic between driver and firmware. This tool must be used when the monitoring will be longer than 5 hours and all monitoring data must be preserved. For short term monitoring **Bus Hound** is preferred, because of its simplicity and easier to work with UI