

Chromeleon DDK Cookbook and Developer Guidelines

© Copyright by Dionex Softron GmbH a part of Thermo Fisher Scientific

Topics:

File:	Chromeleon DDK Cookbook and Developer Guidelines.docx
Release:	Version 1.6, 04-January-2017
Author(s):	Ohlhaut, Marcus;Jochum, Michael
Distribution:	All internal and external developers writing DDK based Chromeleon drivers
Keywords:	Chromeleon DDK Cookbook and Developer Guidelines

Contents

Contents	1
1 Scope of this Document	3
2 References	3
3 Terminology and Abbreviations.....	4
4 DDK Design Goals	5
5 System Architecture	6
5.1 Chromeleon Software Architecture.....	6
5.2 Client/Server functionality.....	7
5.3 The CmDDKHost.exe process	8
6 Configuration Interface	9
6.1 Instrument Configuration Manager	9
6.2 Initial driver startup	11
6.3 Configuration Report	12
6.4 ISendReceive	15
6.4.1 Configuration wizard.....	15
6.5 Extended Functionality	17
6.5.1 Generic Configuration Plug-In	17
6.5.2 IConfigurationDescriptors	18
6.6 Practice Hints.....	19
6.6.1 Handling Simulation Mode and Connect Settings	19
7 Basic IDriver operation	19
7.1 Life cycle of a DDK Driver	19
7.2 Setting up the Symbol Table	20
8 Method Execution.....	21
8.1 Five ways of executing a method	21
8.2 Events during Script Execution.....	22
8.3 Aborting Injection runs	23
8.4 Runtime Messages to Instrument Audit Trail	23
9 Preflighting and Ready Check.....	24
9.1 Preflighting Scenarios.....	24

9.2	Preflight Types.....	26
9.3	Preflight Results.....	28
9.4	Preflighting Events.....	28
9.5	Preflighting and Ready Checks: Do's and Don'ts (applies in general to CM6 as well)	28
10	Some Examples of Instrument Control.....	29
10.1	Simple property update	29
10.2	Dependent Properties.....	30
10.3	Simple command.....	31
11	Method Download.....	32
12	Ramp Syntax	33
13	Generating and Editing Of Script.....	35
13.1	Accessing the method directly (only CM7)	36
13.2	Using UI components, data binding, constraint checkers	39
14	ePanel Support.....	46
15	Sampler Drivers.....	48
15.1	Setting up sampler components	48
15.1.1	Setting up the symbol table	48
15.1.2	Required <code>IInjectHandler</code> interfaces and their implementation	48
15.1.3	Creating a minimum auto-sampler/injector device	49
15.2	Supporting the New CM7 Sequence Wizard.....	49
15.3	Supporting the Injection List	49
15.4	Overwriting Position and Volume in the Instrument Method	50
15.5	Batch Preflighting.....	50
15.6	Blank Runs	52
15.7	Manually triggered Injections	53
15.8	Autosamplers and dynamic Tray Changes	53
15.9	Overlapped Sampler Preparation	55
15.10	Other considerations.....	55
16	Pump Drivers	56
16.1	Setting up the symbol table	56
16.2	Broadcast handling.....	57
16.3	Special interfaces	58
17	Delivering Data	58
17.1	2D Signal Processing	58
17.2	3D Signal Processing	62
17.2.1	<code>IPDACHannel</code>	62
17.2.2	<code>ISpectrumWriter</code>	63
17.3	General Questions.....	64
18	Smart Startup/Shutdown Support.....	65
19	Localization.....	65
20	Online Help Support	66
21	Reporting Support.....	67

21.1	Precondition Log entries	67
21.2	Audit Trail report variables	68
21.3	Signal metadata	69
22	Data Audit Trail Support	72
23	Simulation Mode	72
24	Update of a Released Driver	72
25	Debugging and Troubleshooting	73
25.1	Debugging with Visual Studio	73
25.2	Typical Problems with a DDK Driver	75
25.3	Debugging tools	75
26	Deployment	76
26.1	Build	76
26.2	Source tree layout and projects	76
26.3	Signature	77
26.4	Certification	77
26.5	Setup and Distribution	78
27	Miscellaneous	79
27.1	Do's and Don'ts	79
27.2	Frequently Asked Questions	80
27.3	Miscellaneous CM6 hints	81

1 Scope of this Document

This document is intended for developers using the Chromeleon 7 Device Driver Development Kit (CM7 DDK) to write driver packages for the Chromeleon 7 product family. Readers should already have received a face-to-face training, where the concepts illustrated in this document have been presented.

In addition to the reference documentation (see section 2), which is automatically generated from all the interfaces that the DDK provides, this document contains additional information and background to help developers gain a thorough understanding of how instrument control works in CM7, what typical tasks are to be implemented and what our expectations of driver package behavior are.

DDK V1 offers almost the same functionality in CM6 and CM7. Differences are stated in the CM6 documentation. As long as not stated otherwise in the document the content only refers to CM7.

We intend to update this document in response to questions arising during the training sessions or sent to us via email (ddksupport@thermofisher.com), so frequent updates are to be expected. Updates will not be distributed automatically, but only on request or with each new DDK release.

2 References

1. DDK V1 Documentation:
<ChromeleonInstallationFolder>\DDK\Documentation\DDKV1.chm
2. DDK V2 Documentation:
<ChromeleonInstallationFolder>\DDK\Documentation\DDKV2.chm
3. Chromeleon DDK Driver Certification Requirements.pdf
4. Chromeleon 7 DDK ePanel Selectors.pdf

3 Terminology and Abbreviations

Term	Meaning
Instrument	Refers to a set of →Modules that share a common clock. This could be a stack built of individual UltiMate 3000 modules, as well as a stack of Agilent 1200 modules, or any other mix of modules. It could also be a single module, such as an Agilent GC, a Shimadzu LC-2010 compact system or any other module that provides a complete chromatography system in one module.
Driver	Refers to the set of software that is needed to control a →Module. Each driver can belong to one →Instrument or can be shared between instruments. Typically, a driver has a single communication resource that it uses to control its →Module. Note that for each UltiMate 3000 module we would have one driver (as each uses its own USB connection), whereas for a stack of Agilent 1200 modules we would have a single driver (as the whole stack uses a common TCP/IP connection). For an Agilent GC, we would also have one driver, as well as for a Shimadzu LC-2010 compact system.
Module	A piece of chromatographic instrumentation that can be controlled via a single communication resource. Note, that for a stack of Agilent 1200 modules, the whole stack is represented as a single →Module in Chromeleon.
Device	Functional part of a →Module. This can represent any functional group, such as a UV detector as a whole, a relay offered by a →Module, or one of the two flows delivered by a dual pump. Devices can be grouped into a hierarchy.
Script	A series of property assignments, commands (with parameters), time marks and other instructions controlling the execution order. The Chromeleon way of representing full-fledged Instrument Methods as well as single user interactions with the →Instrument.
Injection (CM7; in CM6: Samples)	A single run - equilibration, sample preparation, sample injection, timetable execution (pump gradients, wavelength switches, temperature changes ...) and data acquisition. Represented by a single line in the Injection List of a →Sequence.
Instrument Controller (ICT)	Part of Chromeleon, responsible for forwarding commands to the →Drivers. It also controls →Batch runs and manages the incoming Data.
Real Time Kernel (RTK)	Part of Chromeleon. A service, which hosts and manages the individual →Driver instances
Sequence	A collection of Injections. Each injection within a sequence may use a different sample position, injection volume and/or injection method.
Queue (CM7; in CM6: Batch)	A collection of →Sequences. Each item within the queue might have been added by a different user. The ICT executes sequence by sequence. For a queue, an emergency method can be defined that gets executed after the Queue has been aborted due to an [Abort] error.
Daily Audit Trail (DAT)	A collection of audit trail messages that were generated during instrument controller operation. It contains all executed script lines, any warnings and error messages issued by the drivers and their devices, as well as additional messages from queue and sequence control. One DAT is generated per →Instrument and per day.
Injection Audit Trail (IAT)	A sub set of audit trail messages that were generated during instrument controller operation. It contains all executed script lines, any warnings and error messages issued by the drivers and their devices during an →Injection. One IAT is generated per →Instrument and →Injection. All these messages are also written to the →DAT.
Data Audit Trail	Documents changes to versioned items in a →Data Vault (DV). For the purpose of this document, Instrument Methods that are saved in a DV are of interest.

Data Vault (DV)	A storage location for Chromeleon data. Stores →Sequences and their Injection Lists in a hierarchical structure. Stores →Daily Audit Trails, →Injection Audit Trails, Instrument Methods, all generated raw data and other items. A DV consists of a file system part (where large objects such as raw data are stored) and a data base (MS SQL Express, MS SQL, Oracle) that holds smaller items and metadata, and namely all versioned items.
Channel	A →Device that produces data during an Injection. It has AcqOn and AcqOff commands.
Signal	Data produced by a →Channel device. It can be 2D (such as Pump.Pressure, Oven.Temperature or UV.UV_VIS_1) or 3D such as UV.3DFIELD or FLD.3DFIELD.
Batch	Chromeleon 6 term for a collection of sequences and standalone →PGMs →Queue
CM	Chromeleon
[CM6]	Information specific to Chromeleon 6
[CM7]	Information specific to Chromeleon 7
DDK	Chromeleon Device Driver Development Kit
GC	Gas Chromatography
IC	Ion Chromatography
LC	Liquid Chromatography
METH	Chromeleon 7 Instrument Method
PGM	Chromeleon 6 Program File (Instrument Method)
SDK	Chromeleon Software Development Kit
Timebase	A collection of devices that share a common clock (CM6)
IME	Instrument Method Editor
IMW	Instrument Method Wizard

4 DDK Design Goals

Chromeleon Driver Development Kit – what is it good for, why did we make it?

The DDK provides a possibility to implement drivers, configuration and editor plug-ins for CM7 (and CM6, but this is not covered here). We want it to be

- used by both internal developers and external developers/third party instrument vendors
- easy to learn
- well documented, including example code
- robust against “bad” drivers
- designed for extension and compatibility

There is very little must-have code: simple drivers are easy to implement. Since the .NET infrastructure avoids a lot of issues with binary API compatibility, drivers can more easily be ported/reused in different CM versions; this should speed up driver development significantly.

As the DDK is used by Thermo Fisher Scientific as well (all new drivers implemented by us are based on the DDK), we are confident that the interfaces are useful, complete and work according to their specification.

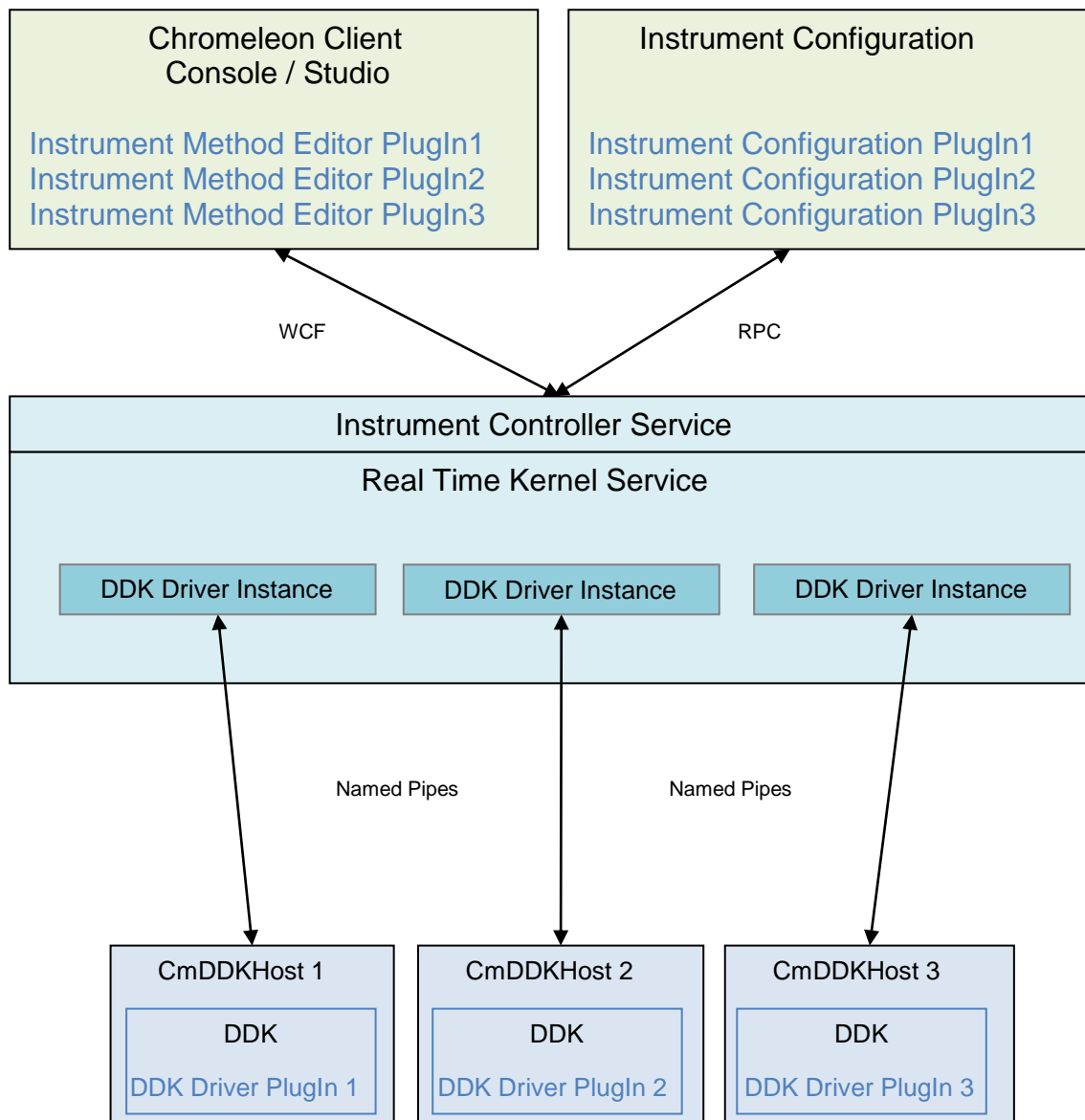
One of the major goals was to reduce complexity compared to the legacy C++ interface; the DDK interfaces are much smaller and more structured.

As all DDK interfaces are CLS compliant, any .NET language can be used. We strongly recommend C#.

5 System Architecture

5.1 Chromeleon Software Architecture

The following diagram shows the main components of the Chromeleon application family. The items with blue text are components covered by this documentation.



The Instrument Controller is actually two services for historical reasons: Instrument Controller Service ICS (ServiceHost.exe) and Real Time Kernel Service RTK (CmDriver.exe).

Some comments about the usual workflows:

- Instrument Method Editor (CM6: Program Editor) UI Interface (DDK V2): Console/Studio hosts a (mandatory) UI for creating and editing of instrument methods. No direct access to driver/hardware, all I/O is carried out via the symbol table and execution of method scripts.
- Direct control (controlling the instrument when no injection is running) is via ePanels (or Command dialog) hosted in the client application. All user interactions in the ePanels are forwarded to the CmDDKHost process.
- Configuration Interface (DDK V1): Instrument Configuration hosts Configuration UI. Uses a plug-in mechanism. Reports and validates configuration. It persists configuration through Instrument Controller.
- Driver Interface (DDK V1): CmDDKHost process hosts driver implementation. No UI. One process per driver instance. Performs all I/O with hardware.
- Some processes run in user session (Console, Studio, Instrument Configuration), some as LocalSystem (ICS, RTK, CmDDKHost).
- Many more processes for administrative tasks: Discovery Setup, Data Vault Setup, Administration Console (User Management, License Configuration).
- Licensing (see Help > About > Details in the Chromeleon Console) determines what features of Chromeleon can be used. For drivers, the DDK Developer License (see 26.4) and the 3D Data Acquisition License (see 17.2) are usually the most important.

5.2 Client/Server functionality

Be aware that Chromeleon is fully Client/Server compatible. This means that your driver instance (talking to the actual device hardware), the corresponding configuration plug-in and the corresponding instrument method editor plug-in may run on three different computers. Also be aware that it is not guaranteed that both sides run the same version of Chromeleon.

- Communication between the driver and the configuration plug-in is solely via the exchanged configuration XML and the **ISendReceive** interface. The configuration plug-in must not perform local actions (e.g. browsing the local file system for support files, enumerating local communication resources or plug and play devices) if these actions logically belong to the driver component. Use ISendReceive to submit such actions to the driver to execute them on the server PC on behalf of the configuration plug-in. Ensure that you can handle the case where both sides support a different set of ISendReceive functionality and document any backward/forward compatibility issues that arise from additional features added to this interface.
- Also make sure that both driver and configuration plug-in deal gracefully with changes to the content of the XML configuration information (e.g. ignore, but preserve additional nodes; use suitable defaults for missing nodes, but don't add them if not needed; provide some version number in interfaces and exchanged data).
- Also be prepared for users loading a configuration (InstrumentConfiguration.cm6 in CM7, cmserver.cfg in CM6) from another computer that might be older or newer than the version the running code has been compiled against. Similar graceful fallback handling as stated in the previous item is needed.
- Communication between the driver and the instrument method editor plug-in is solely via the **symbol table** (presence or absence of commands and properties, attributes on symbols; all static) and current **property values** (all dynamic). This is a one-way data flow from server to client. The instrument method editor plug-in should, in general, not require that the driver is actually connected to the hardware device. It should, however, assume that the driver has initialized all properties to useful values (or null). The instrument method editor plug-in must be prepared to be connected to a driver that provides a different symbol table (maybe some symbols have been added recently, but are not present in an older version of the driver running on an older instrument controller), typically by disabling the corresponding UI elements and suppression of method step generation for these absent symbols.
- Also be aware that multiple instances of the Chromeleon client can run at the same time (either by the same or different users, on the same or different Windows session (fast user switching or terminal services), on the same or different PCs). While only one client will be able to actively control the server and submit methods for execution, all other clients can monitor the driver's state, its symbols and their current values all at the same time. Make sure

that you save user specific settings (e.g. window position or size) in a user-specific location (HKCU or %USERPROFILE%) or avoid this altogether.

- Make sure that you don't transport information containing file paths or UNC paths between the components. When running on different computers, these paths may not be meaningful or even not accessible on some of these computers. The same applies to other system resources, e.g. registry keys.
- [CM6]: Be aware that in order to submit sequences from a remote client, one needs to have the appropriate Chromeleon licenses available: Multiple Network Control on the server; Server Control on the client. Both PCs need to have access to the same datasource. On the server PC, create a Windows file share for C:\chromeleon\data and provide Read/Write access to "Everyone". On the client PC, mount the datasource (File > Mount Datasource > Browse to the share, and select the cm_local.mdb. When working in the client, place all sequences that you want to run on the server PC in that datasource so that the server can access the samples and programs.

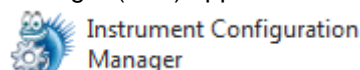
5.3 The CmDDKHost.exe process

- Assume that each instance of a driver is loaded into its own instance of a CmDDKHost.exe process. Thus, in case you need to communicate with another instance of your driver (e.g. for instance counting), standard Windows interprocess communication features need to be used.
- Do not interpret the contents of the command line of the process you run in. Parts of it may be derived from user's input
- Typically, CmDDKHost.exe runs as LocalSystem, spawned by the real time kernel (CmDriver.exe), which runs as LocalSystem, too. The process has no way to present UI or receive user input directly, as it does not have access to the user's desktop session. Don't rely on being run as LocalSystem when requesting access to objects subject to Windows security. For instance, don't request Key.AllAccess when just reading from a registry value. Don't read from/write to user specific data locations such as HKCU or %USERPROFILE%.
- For debugging purposes and specific use cases (MSQ support), all processes should also be runnable as standard Windows applications with the user rights of the currently logged on user.
- Be Terminal Services-aware; CmDDKHost.exe typically runs spawned from a Windows service, thus it runs in session 0.
- Be threading safe. Not all calls into a driver are from the same thread, plus, you will most likely implement a few threads of your own in your implementation. Make sure that you protect all accesses to global data and that you don't risk any deadlocks. Expect concurrent and reentrant function calls. Don't rely on the current implementation (what gets called on what thread) to be guaranteed behavior unless stated in the DDK documentation.
- Be multi-instance aware. For example, when you store files or create Windows kernel objects, make sure that the object name that your driver instance is unique (there could be another instance of your driver) and recognizable (e.g. use the name of your instrument and/or main device, or an instance count, or the friendly driver name as the unique identifier). Assume that one instance gets stopped and started anew while other instances keep running.
- Since the Instrument Controller's processes run as Windows Services, no user interface can be presented to the user by these processes directly.
- When spawning multiple CmDDKHost.exe instances, the RTK tries to distribute these instances evenly across all available cores. However, this is not guaranteed. Your driver should work on a low-performing single-core PC as well, if you require significant CPU resources, the driver's release notes/documentation should stress this and detail the minimum requirements.

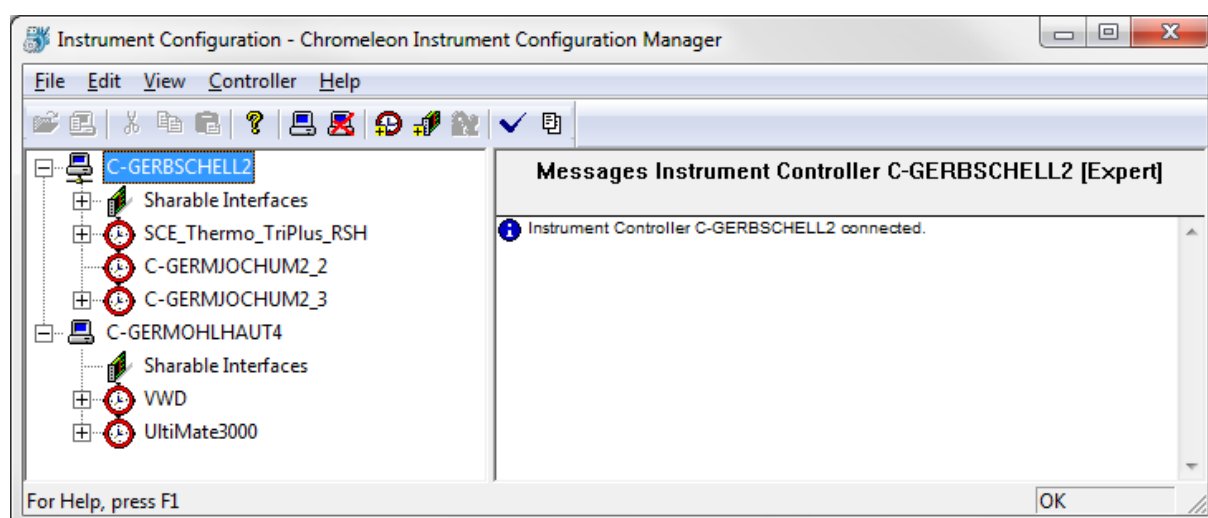
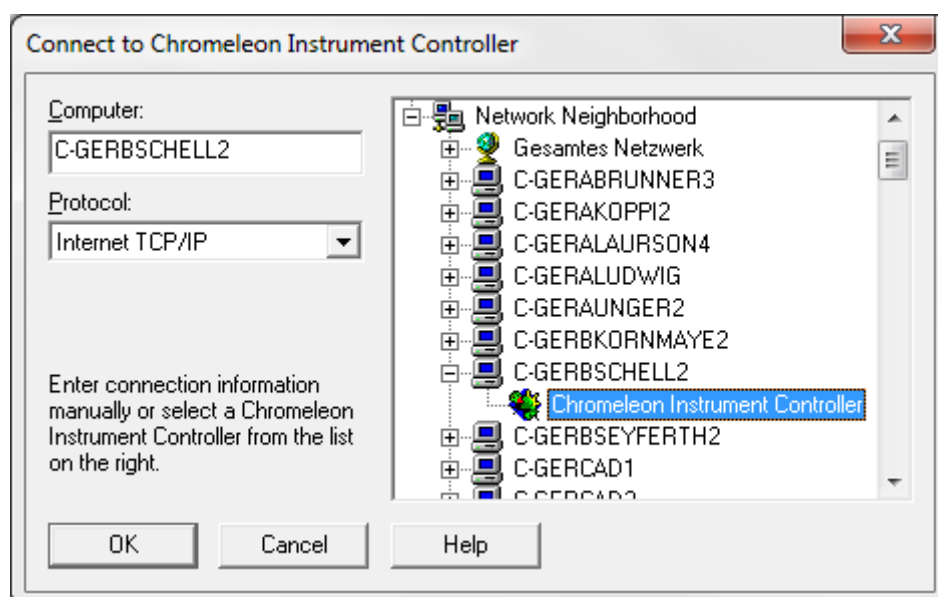
6 Configuration Interface

6.1 Instrument Configuration Manager

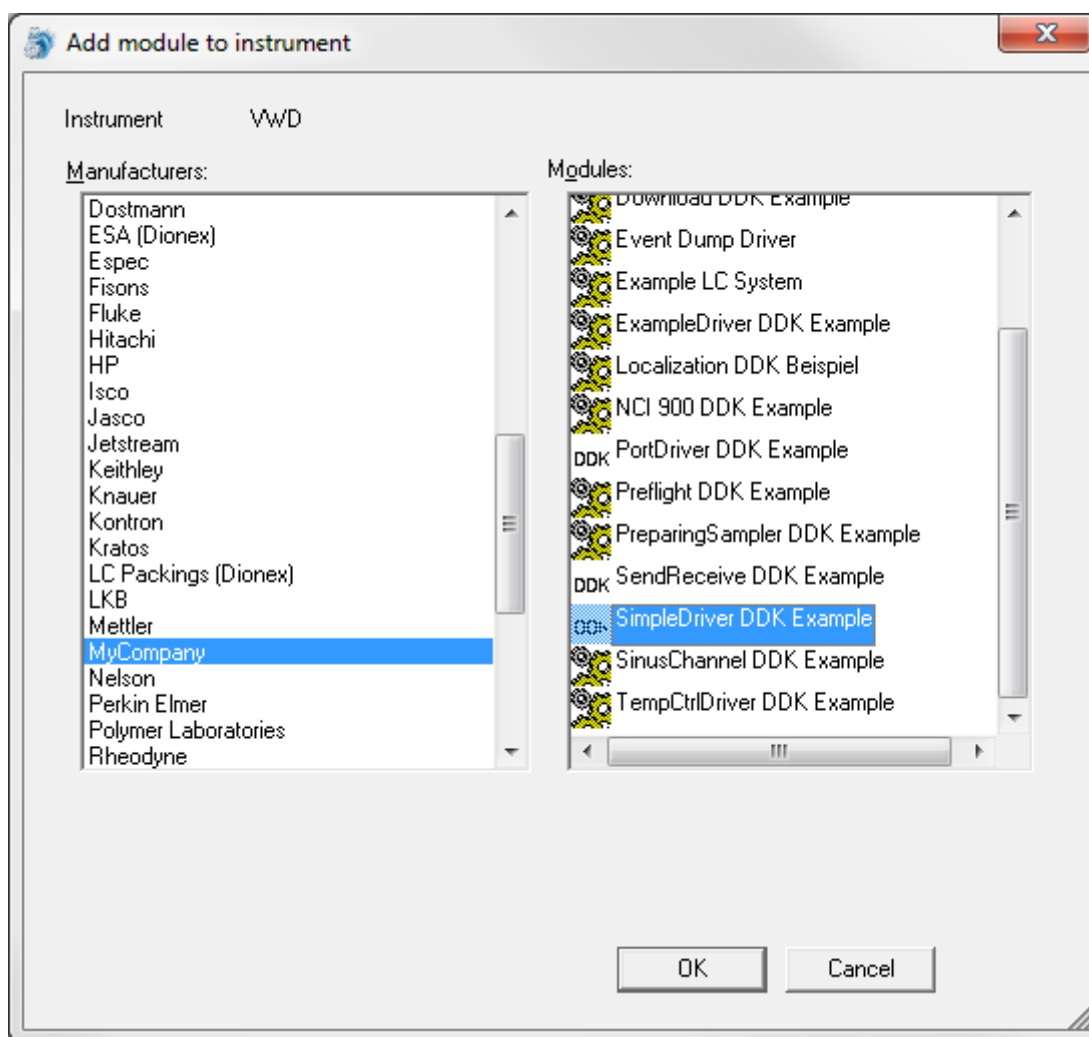
To support changing the configuration of the Instrument Controller (ICT), the Instrument Configuration Manager (ICM) application is available:



This application runs with the privileges of the Windows user that starts it. It can be used to configure both the Instrument Controller running on the same PC (local configuration) as well on a different PC (remote configuration). When the application starts up, it attempts to connect to the local ICT (if one is running.) Connections to remote ICTs need to be established manually via menu item Controller > Connect to Chromeleon Instrument Controller:



Users select a driver to be added to the configuration by first adding an instrument (Edit > Add Instrument) or by selecting an instrument that has already been created, then calling for the Add module to instrument dialog (Edit > Add module):



Manufacturer and Module names are defined by the driver plug-in, in the DriverInfo.resx:

Name	Value	Comment
DeviceManufacturer	MyCompany	Required string resource: Manufacturer of the hardware device being controlled by the driver.
FriendlyName	SimpleDriver DDK Example	Required string resource: Name of the driver (also used to list the driver in the Instrument Configuration Manager).

If more than one instance of a driver is added to an instrument, the FriendlyName is suffixed with #n, e.g. "SimpleDriver DDK Example #2" for the second instance.

NB: After adding a module, users who may wish to edit the friendly name can do so from the tree view. Friendly name must be < 40 characters.

The minimum settings that a user must have available in the configuration UI are:

- Specify device names for all devices that the driver is about to create. This is needed so that more than one driver instance can be added to an instrument without the two instances attempting to create device symbols with identical names.
- Specify which communication resource (COM/RS232 port, TCP/IP address, GPIB address, USB ...) is to be used by the driver for communication with the module.

All UI used in the ICM should be localizable, but do not localize device names or other components related to symbol names..

The configuration plug-in exchanges the (driver-private) configuration settings with the corresponding driver via an XML document. In its simplest form, this document looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
  <Driver>
    <Device name="Simple Device">
      <Parameter name="Device Name">SimpleDevice</Parameter>
    </Device>
  </Driver>
</Configuration>
```

The nodes `<Configuration>` and `<Driver>` are mandatory; the contents of the `<Driver>` node are treated as driver-private data that is in transparent to Chromeleon.

Caution:

- Make sure that when this document contains any numbers that they are represented in InvariantCulture - the locale settings in ICM and CmDDKHost.exe might not be the same.
- The DDK adds some internal information as attributes to the outer hull of the `<Configuration>` and `<Driver>` nodes. When you adapt the configuration XML in the configuration plug-in and/or driver, or if you convert it to an internal type of configuration structure: Never try to recreate the configuration XML from scratch if a DDK function asks for the current configuration, as then this additional information may get lost. Always load the configuration XML into an XmlDocument instance and make local adaptations there.

The instrument controller is responsible for persisting the configuration information. Upon closing the ICM application (or by File > Save Installation), the list of instruments, list of drivers and each driver's private configuration information is saved to

C:\ProgramData\Dionex\Chromeleon\InstrumentControllerConfiguration.cmhc

on the PC that runs the ICT (not the ICM). When the ICT is stopped and restarted, it will read this file, use it to load the individual drivers and provide each driver with its private configuration information.

The configuration plug-in also needs to provide a formatted text snippet that describes all settings that have been made in the configuration UI in human readable form. This snippet will be used as part of the Configuration Report (Controller > Create Report). See chapter 6.3 for details.

Configuration Check (Controller > Check Configuration) is implemented by the framework. It attempts to identify basic configuration errors (instruments without an injector, duplicate device names ...) and is also automatically executed prior to saving the configuration information.

Input validation: Device names must not be longer than 24 characters, may consist of [A-Za-z0-9_] only and must not start with [0-9]. Please ensure that all device names handled by a single configuration plug-in are unique at the time of leaving the dialog with OK.

6.2 Initial driver startup

Initially loading a driver results in the following series of actions:

1. User selects the desired driver from the Add module to instrument dialog.
2. RTK spawns a new CmDDKHost.exe and tells it to load the corresponding driver assembly.
3. CmDDKHost.exe creates an instance of the type implementing `IDriver` in this assembly.
4. `IDriver.get_Configuration` is called. The implementation needs to return an XML string specifying the default configuration.
5. This XML string is passed from the ICT to the ICM.
6. ICM loads the corresponding configuration plug-in assembly.
7. ICM creates an instance of the type implementing `IConfigurationPlugin` in this assembly.
8. ICM calls `IConfigurationPlugin.set_Configuration` and provides the XML string.
9. ICM calls `IConfigurationPlugin.ShowConfigurationDialog`.
10. The implementation should now present the configuration UI initialized to settings as described by the received XML.

After the user has reviewed the settings in the configuration UI, clicking OK will result in the following series of actions:

1. ICM calls `ICConfigurationPlugin.get_Configuration`. The implementation needs to return an XML string specifying the details of the configuration as has been defined by the user in the UI.
2. ICM passes this XML string to the ICT.
3. RTK unloads the `IDriver` assembly and stops `CmDDKHost.exe`.
4. RTK spawns a new `CmDDKHost.exe` and tells it to load the corresponding driver assembly.
5. `CmDDKHost.exe` creates an instance of the type implementing `IDriver` in this assembly.
6. `IDriver.set_Configuration` is called with the XML string from step 2.
7. `IDriver.Init` is called. The implementation needs to interpret the configuration XML and use the information provided therein to set up the symbol table accordingly.

Aborting the configuration UI dialog by clicking Cancel will simply close the configuration UI, unload the `IDriver` assembly, stop `CmDDKHost.exe` and unload the `ICConfigurationPlugin` assembly.

If the user opens the configuration UI dialog for a driver that is already up and running (by double-clicking the corresponding node in the tree view of ICM or via Edit > Properties), the following series of actions result:

1. `IDriver.get_Configuration` is called on the running instance. The implementation needs to return an XML string specifying the current configuration
2. This XML string is passed from the ICT to the ICM.
3. ICM loads the corresponding configuration plug-in assembly.
4. ICM creates an instance of the type implementing `ICConfigurationPlugin` in this assembly.
5. ICM calls `ICConfigurationPlugin.set_Configuration` and provides the XML string.
6. ICM calls `ICConfigurationPlugin.ShowConfigurationDialog`. The implementation should now present the configuration UI initialized to settings as described by the received XML.

Clicking Cancel at this point will simply close the configuration UI.

Clicking OK will result in the same steps as described above (i.e., the driver is torn down and re-spawned with the XML that the configuration UI has provided. This is independent of whether the settings have changed or not.)

6.3 Configuration Report

With `ICConfigurationPlugin.ConfigurationReport` the DDK asks the configuration plug-in for a formatted string that lists the current driver's configuration settings. While the format of this text was not defined in the past, almost all drivers set it up in a standard format that is described below. Starting with 7.2 SR5, the configuration report is also parsed by other administrative tools of Chromeleon so that it becomes mandatory to use the following format.

- For a setting, use the following format `<Description>:<Tab><Value>`. Example:

```
Device Name: UV.
```

Do not use ':' within the description so that it is clear for a parser where the description ends.

- Use a new line for every setting. Example:

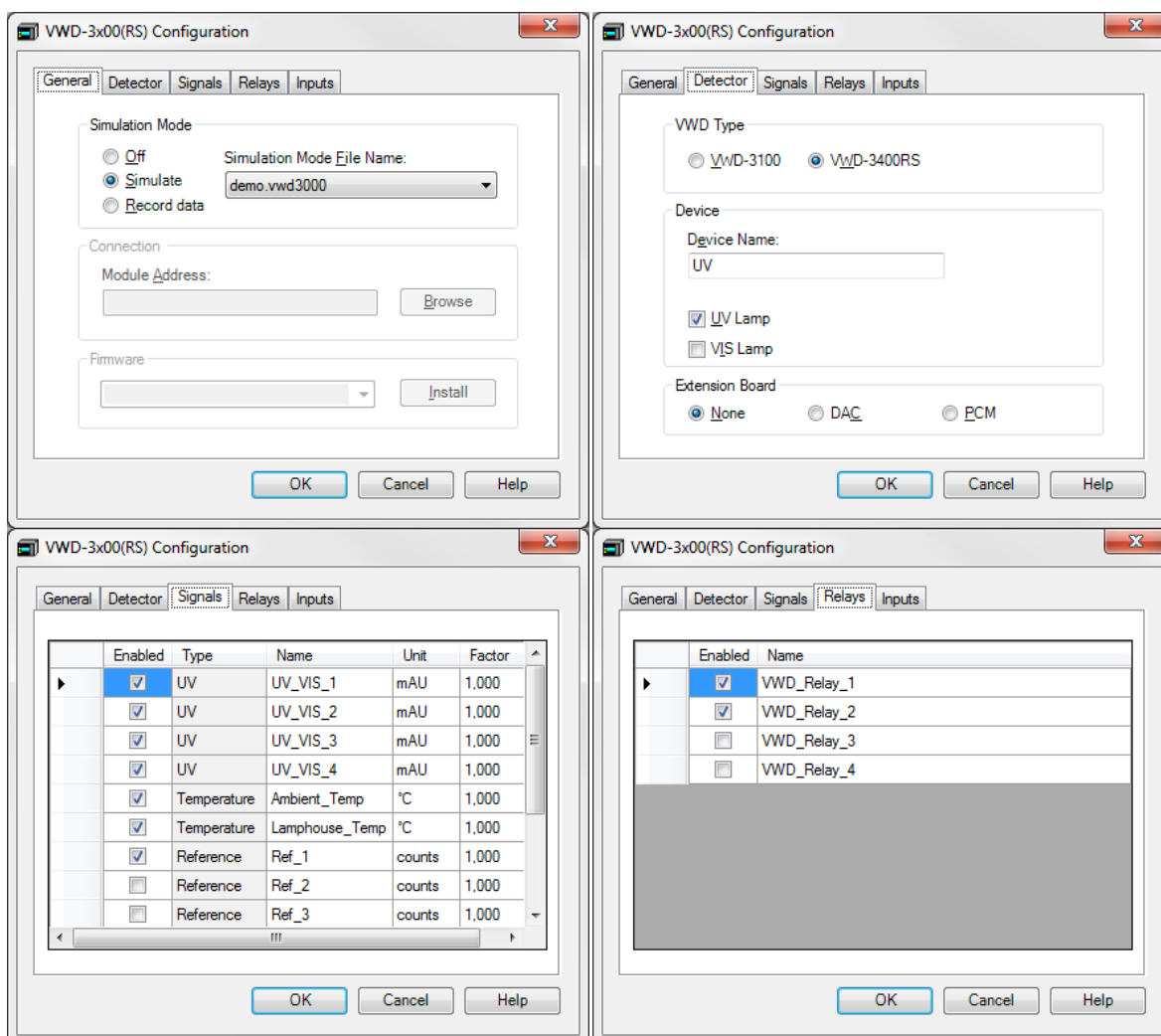
```
...
Device Name: UV
USB Address: USB-12345
Simulation Mode: Reading from file 'demo.vwd3000'
...
```

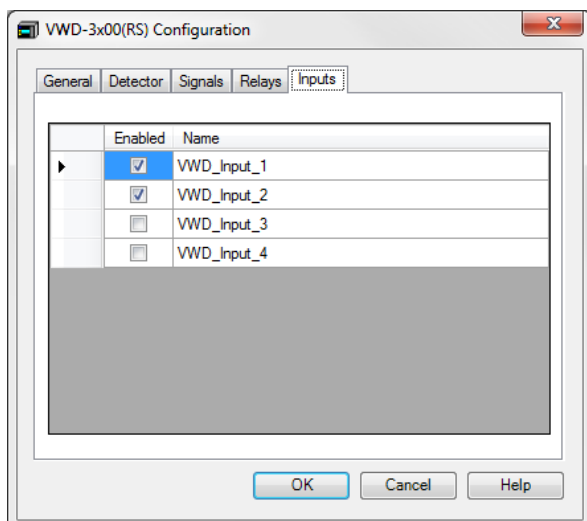
- Group settings for sub modules and use tabs for indentations. Use a separate line with a title that describes the sub module. Example:

```
...
Signals:
    UV_VIS_1:
        Enabled:      Yes
        Unit:         mV
        Factor:       1
    UV_VIS_2:
        Enabled:      Yes
...
```

Grouping should start with the driver node: <Tab><Driver Name>, as shown in the example below.

For instance, the following UI settings





are represented as:

```
VWD-3x00 (RS) Detector
  Device Name: UV
  USB Address: USB-12345
  Simulation Mode: Reading from file 'demo.vwd3000'
  Used Detector:
    Type: VWD-3400RS
    UV Lamp: Installed
    VIS Lamp: Not Installed
    DAC board: Not Installed
    PCM board: Not Installed
  Used Signals:
    UV_VIS_1
      Type: UV
      Unit: mAU
      Factor: 1,000
    UV_VIS_2
      Type: UV
      Unit: mAU
      Factor: 1,000
    UV_VIS_3
      Type: UV
      Unit: mAU
      Factor: 1,000
    UV_VIS_4
      Type: UV
      Unit: mAU
      Factor: 1,000
    Ambient_Temp
      Type: Temperature
      Unit: °C
      Factor: 1,000
    Lamphouse_Temp
      Type: Temperature
      Unit: °C
      Factor: 1,000
    Ref_1
      Type: Reference
      Unit: counts
      Factor: 1,000
    Measm_1
      Type: Measurement
      Unit: counts
      Factor: 1,000
  Used Relays:
    VWD_Relay_1
    VWD_Relay_2
  Used Inputs:
    VWD_Input_1
    VWD_Input_2
```

Note: The Configuration report uses the local regional settings as numbers format.

6.4 ISendReceive

Note that (to support remote configuration scenarios) the configuration plug-in must **never** attempt to communicate with the (hardware) module directly. Nor must it access **local OS or PC resources** for the purpose of figuring out details about modules automatically. You cannot, for example, ask Windows to enumerate the list of available COM ports or devices known to Plug&Play within the implementation of the configuration plug-in. Such code must be implemented in the driver because only this code is guaranteed to run on the computer where the Instrument Controller runs and where your hardware is connected.

The DDK provides the `IDriverSendReceive` interface to allow you to execute code by the driver plug-in on behalf of the configuration plug-in.

Example: To provide a drop list that has been initialized with the available COM ports to which your module can be connected:

1. In the configuration plug-in, have


```
private IConfigSendReceive m_SendReceive;

public bool ShowConfigurationDialog(IConfigDriverExchange
configDriverExchange)
{
    m_SendReceive = configDriverExchange as IConfigSendReceive;
    string outputString;
    m_SendReceive.SendReceiveEx("GetComPorts", out outputString);
    // Parse information returned and initialize drop list.
}
```
2. In the driver plug-in, have

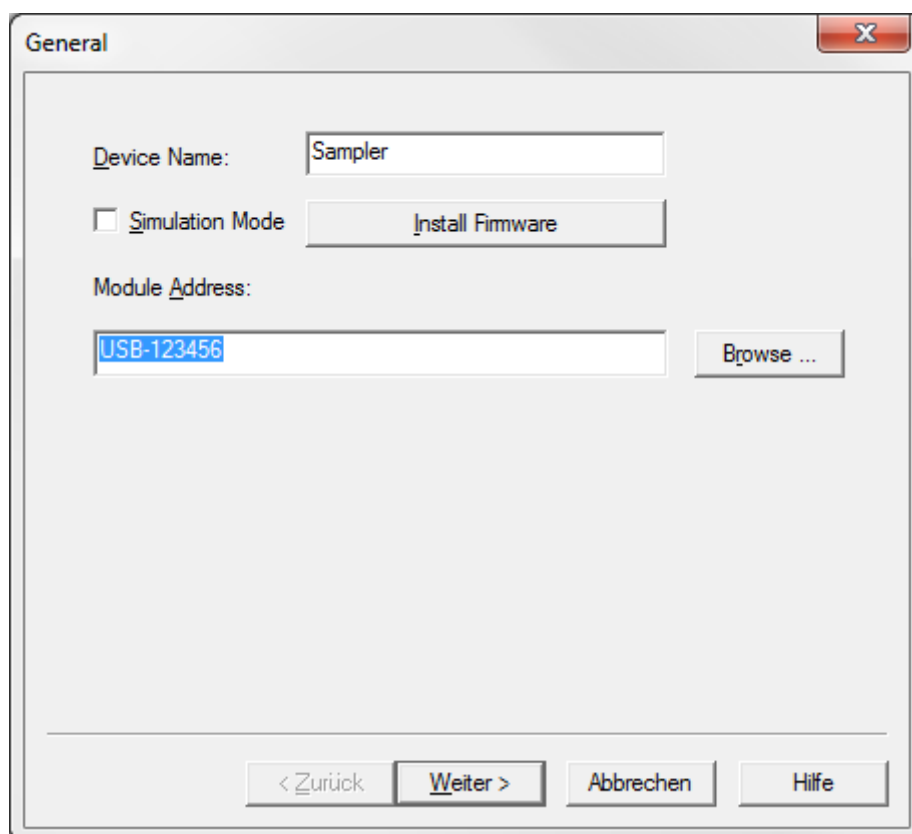

```
public class Driver: IDriver, IDriverSendReceive {...}

public void OnSendReceive(DDK cmDDK, string inputString, out string
outputString)
{
    outputString = "";
    if (inputString == "GetComPorts")
    {
        // SerialPort.GetPortNames() or however you determine it.
        outputString = "COM1#COM2";
    }
}
```

Another use case would be additional functionality to trigger a firmware download from within the configuration UI. This would use an `IConfigSendReceive` call to ask the driver to perform this download using the communication resource that the driver has available anyway. (Of course, the implementation on the driver side should ensure that it is safe to do such a download to avoid interfering with runs, and the firmware file needs to be available on the PC that runs the Instrument Controller. You can't simply browse the file system on the PC running the ICM and forward the file path to the ICM; you could, however, transmit the file content via the send/receive mechanism.)

6.4.1 Configuration wizard

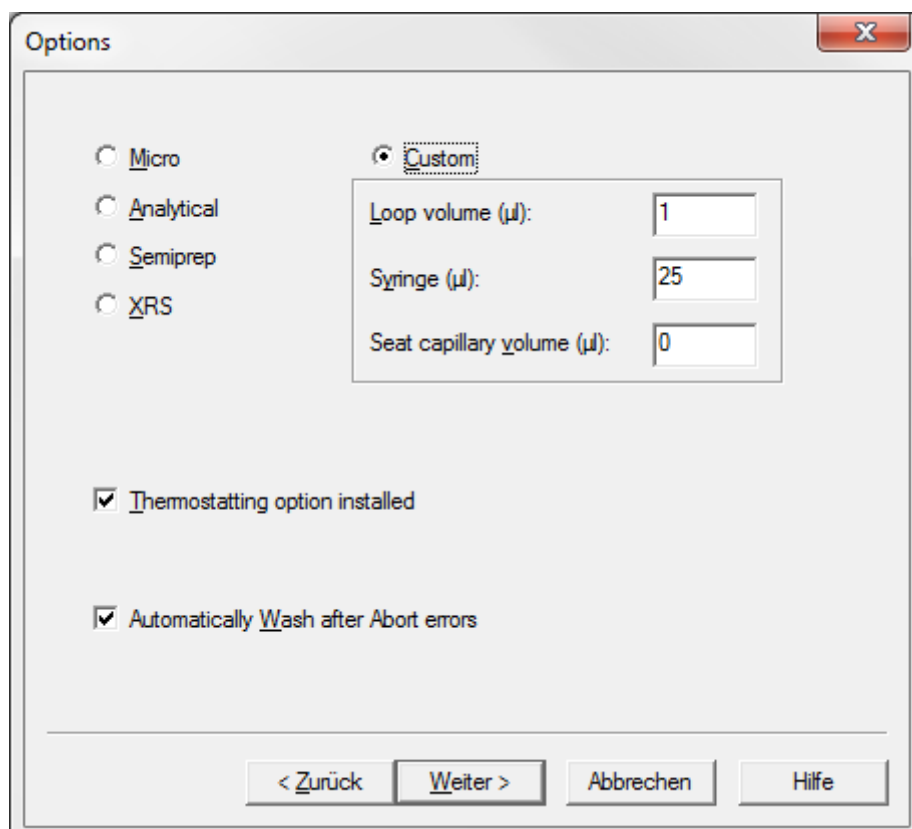
A clever use of `ISendReceive` functionality could allow easing initial driver configuration. Rather than presenting an initial configuration dialog filled with some defaults, show a wizard instead for initial driver configuration. (You could flag this by returning a default configuration XML that has `<Configuration>` and `<Driver>` nodes only and is otherwise empty.) The first page of the wizard should simply ask for the communication resource and the default device name(s):



The 'General' window contains the following fields and controls:

- Device Name:** A text box containing 'Sampler'.
- Simulation Mode:** An unchecked checkbox.
- Install Firmware:** A button located next to the Simulation Mode checkbox.
- Module Address:** A text box containing 'USB-123456'.
- Browse ...:** A button located next to the Module Address text box.
- Navigation buttons:** '< Zurück', 'Weiter >', 'Abbrechen', and 'Hilfe' are located at the bottom of the window.

On Next, perform an `IConfigSendReceive` call that informs the driver about the communication resource that is to be used. The driver implementation can open the communication resource, talk to the module, figure out what particular options this module provides and return back information to the wizard so that the subsequent pages can automatically be initialized with the correct settings:



The 'Options' window contains the following fields and controls:

- Radio buttons:**
 - ☐ Micro
 - ☐ Analytical
 - ☐ Semiprep
 - ☐ XRS
 - ☒ Custom
- Custom settings (when Custom is selected):**
 - Loop volume (µl):** A text box containing '1'.
 - Syringe (µl):** A text box containing '25'.
 - Seat capillary volume (µl):** A text box containing '0'.
- Checkboxes:**
 - ☒ Thermostating option installed
 - ☒ Automatically Wash after Abort errors
- Navigation buttons:** '< Zurück', 'Weiter >', 'Abbrechen', and 'Hilfe' are located at the bottom of the window.

After the initial configuration, present the standard dialog UI for review or changes:

The image displays two screenshots of the 'WPS-3000(RS) Autosampler' configuration dialog box.

The top screenshot shows the 'General' tab. It includes a 'Device Name' field with the value 'Sampler', a 'Simulation Mode' checkbox (unchecked), and a 'Module Address' field with the value 'USB-123456'. There are buttons for 'Install Firmware', 'Browse ...', 'Retrieve configuration from module', and 'Send configuration to module'. At the bottom are 'OK', 'Abbrechen', 'Übernehmen', and 'Hilfe' buttons.

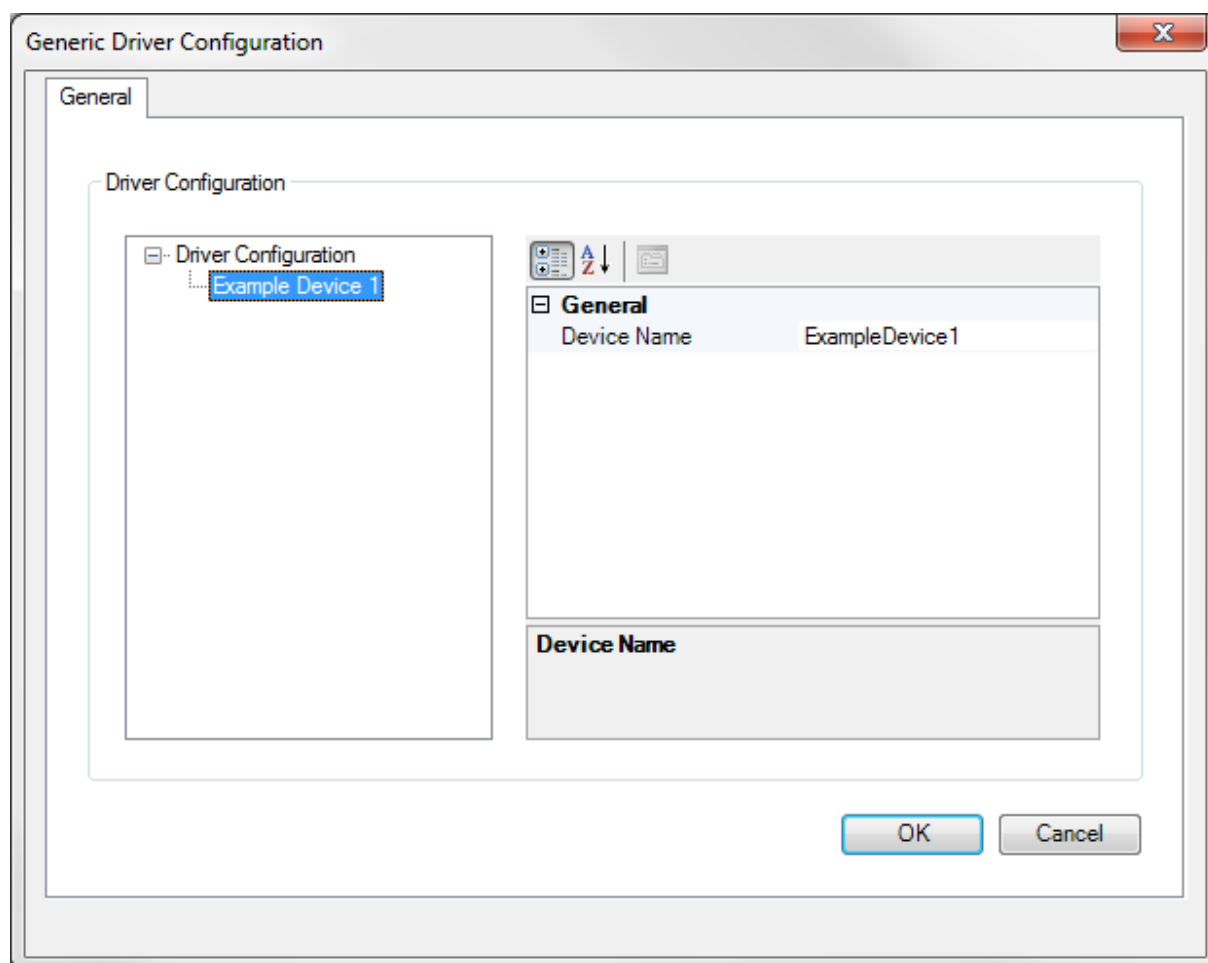
The bottom screenshot shows the 'Options' tab. It features radio buttons for 'Micro', 'Analytical', 'Semiprep', and 'XRS' (selected). To the right, there are input fields for 'Loop volume (µl)' (1), 'Syringe (µl)' (25), and 'Seat capillary volume (µl)' (0). Below these are checkboxes for 'Thermostating option installed' and 'Automatically Wash after Abort errors'. At the bottom are 'OK', 'Abbrechen', 'Übernehmen', and 'Hilfe' buttons.

You can also use this functionality in editor mode; i.e. for an initialized driver. A use case would be to try to connect the driver to a different device or to revert manual changes. Take care, however, that you may send this request to a driver with an existing connection. If the communication parameters match the one of the existing connection, you can use it normally. If not, you have to ensure that the existing connection is not closed or is at least restored afterwards.

6.5 Extended Functionality

6.5.1 Generic Configuration Plug-In

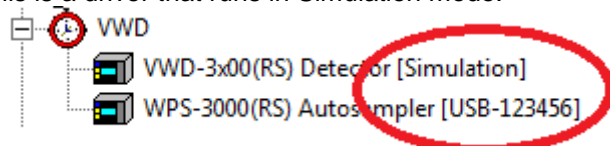
To speed up initial development, having a working configuration plug-in is not required during early development phases. As long as your configuration string is a well-formed XML document, you can use the built-in generic configuration dialog with your driver. It will be used if, for a particular DriverID, no matching configuration plug-in can be found, and will simply allow you to edit all the nodes in the XML document:



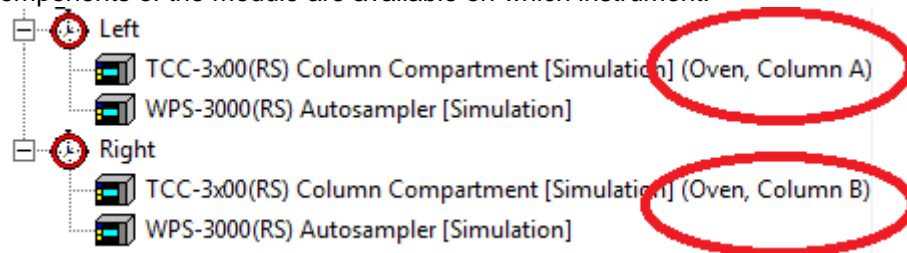
6.5.2 IConfigurationDescriptors

The optional interface `IConfigurationDescriptors` can be used to add additional information to the driver name as it appears in the tree view. This information cannot be edited by the user.

Use `ConnectInfo` to add information such as the IP or USB address used by the driver, or whether this is a driver that runs in Simulation mode:



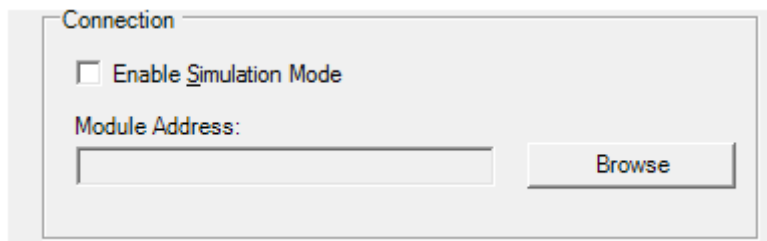
Use `InstrumentInfo` if a driver is shared between instruments and you want to indicate which components of the module are available on which instrument:



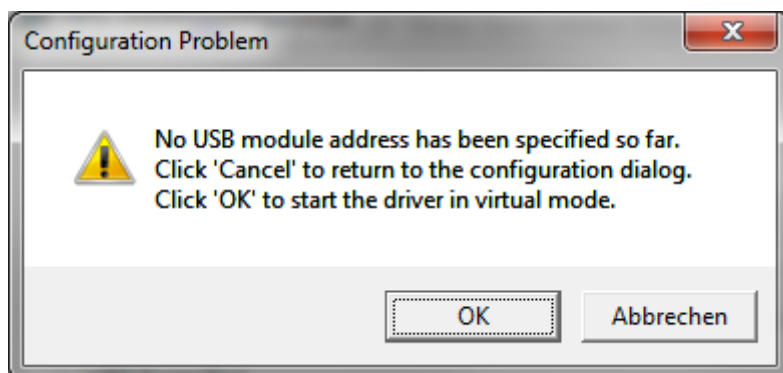
6.6 Practice Hints

6.6.1 Handling Simulation Mode and Connect Settings

The first page that is displayed for both initial configuration and reconfiguration should contain a check box for the simulation mode (CM6: "virtual mode") and connect settings (i.e. RS232 COM port, USB port, LAN address). In the default configuration, simulation mode should be off and the connect settings should be empty. Example:



Of course, this means that the initial configuration is an inconsistent one, one that the driver can't work with. So if a user tries to leave the initial configuration without enabling the simulation mode or specifying connection information, the configuration plug-in should be able to cope with that; e.g. by popping up a dialog similar to that one:



7 Basic IDriver operation

7.1 Life cycle of a DDK Driver

Except from the initial driver startup (see 6.2), an object implementing `IDriver` has the following life cycle:

1. RTK determines that a particular driver is part of the configuration
2. RTK spawns a new instance of `CmDDKHost.exe`.
The command line includes the Driver ID¹ as well as an *instance identification*, e.g., `MyCompany.AutoSampler|AutoSampler DDK Example #2`
3. `CmDDKHost.exe` uses the Driver ID to determine the correct assembly by enumerating and inspecting all available `DriverCert.xml`
4. An instance of the object implementing `IDriver` is created
5. `IDriver.set_Configuration` is called and the configuration XML is passed to the object
6. `IDriver.Init` is called. Implementation needs to set up the symbol table now. See 7.2 for details.
7. The driver is now up and running, but has not yet opened the communication resource and does not talk to the hardware.
8. `IDriver.Connect` is called. The driver should open the communication resource, verify that the hardware found matches the configuration specified by the configuration XML, and

¹ Make sure that there are no spaces in your DriverID. Otherwise, command line parsing will fail.

synchronize all Chromeleon properties with the firmware properties.

Note that there are a couple of timeouts for this call. If your driver doesn't return quickly (<60 s), the DDK assumes that the operation failed and your driver is treated as being in Disconnected state.

Note that there is no way to represent any timetables that might be stored in the module as each Chromeleon property can only represent a single value at any time. Use the current value, the initial value or the out-of-run value, for initializing the Chromeleon property.

Connecting should not have any effects on the hardware (except, maybe locking the hardware's keypad or setting an indication that the hardware is now under computer control).

It may happen that you read a property value from the module that is outside the range for this property as you have defined within Chromeleon. A good practice would be to clip the value read from the module to be within the Chromeleon range (maybe the driver issues a Warning to document this) and then update the value stored in the module to be the same.

9. During normal operation, the driver forwards any actions requested by the RTK to the hardware.

A driver should not modify the state of the hardware on its own. All changes should only be done in response to the method executed by the RTK (which results in events fired in the driver).

10. `IDriver.Disconnect` is called. The driver should unlock the hardware's keypad or clear the indication, stop all communication threads and close the communication resource.

Chromeleon properties can remain at the last known value, or, if appropriate, be set to null.

11. At this point, further Connect/Operation/Disconnect cycles may follow.

12. `IDriver.Exit` is called. The driver should clean up all other resources that the implementation has used and dispose of any unmanaged resources as well.

13. `CmDDKHost.exe` terminates. (This could be due to various reasons - instrument controller gets shut down, instrument gets deleted, driver gets deleted, driver gets reconfigured. You don't need to care about the reason.)

7.2 Setting up the Symbol Table

All client UI uses an instrument's Symbol Table (ST) to learn about what an instrument offers in terms of instrument status and control. The ST is a hierarchical structure of devices, sub-devices, structs, properties and commands. Properties always have a value (special value <null> means value is currently unavailable), Commands can have parameters. The client code knows nothing about the specific semantics of these properties or commands.

The driver implementation needs to set up the desired ST as part of the `IDriver.Init()` function:

- Each **Driver** creates one or more **Devices**
- A Device offers **Properties** and **Commands**
- Properties and command parameters use a separate **Type** description for the variable
- Devices, Properties and Commands are part of the **Symbol Table**

The `IDDK` reference that is passed into `IDriver.Init()` can be used to create new device objects, providing an `IDevice` interface. On the `IDevice` interface, you'll find methods to create new data types (`ITypeDouble`, `ITypeInt`, `ITypeString`) and to create properties (`IProperty`), commands (`ICommand`) and parameters (`IParameter`, also using a `Type` description).

Note: The first device you create in the `IDriver.Init()` function will get the Connect, Disconnect commands. So this should be the 'main' device of your driver instance.

At this point, you'll likely need to inspect the Configuration XML that has been passed into the driver for:

- names of devices (these should be configurable to allow more than one driver instance to be added to one instrument)
- presence or absence of hardware options (don't provide properties and commands for optional components like an additional valve or temperature control if the option is not present in your module at this time)
- the valid range of property types (e.g. a data collection rate property's range might depend on what firmware the module is running, a volume property's range might depend on what syringe has been installed in a sampler)

Some property/command names are reserved by the system because the DDK needs to provide additional support for these symbols to guarantee that they follow specific semantics. For these properties, you need to call `IDevice.CreateStandardProperty()` rather than `IDevice.CreateProperty()`. Refer to [1] for details.

Also, for some types of devices, rather than using `IDDK.CreateDevice()`, you need to call more specific create functions. See further down in this document for more details.

`IDevice.SetOwner` can be used to create a device hierarchy if desired. This is typically used to group the 2D and 3D channel devices offered by a detector under a common device that represents the detector as a whole.

`IDevice.CreateStruct` can be used to group several related properties together. This is typically used for properties like:

`Temperature.Nominal`
`Temperature.Value`

`Pressure.LowerLimit`
`Pressure.UpperLimit`
`Pressure.Nominal`

`%B.Value`
`%B.Equate`

etc.

8 Method Execution

8.1 Five ways of executing a method

CM7 supports five different user interactions that result in the RTK executing a method:

1. When the user operates an ePanel control that is linked to a property.
This usually results in a single line method that specifies a property assignment:

`Pump.Flow = 1.000`
2. When the user operates a script button on an ePanel.
This results in arbitrary script; what script gets executed is defined by whoever created the ePanel by defining the script button's Script property.
3. When the user operates the Monitor Baseline button.
For "On", this results in a script with `AcqOn` commands for the selected channels.
For "Stop", this results in a script with `AcqOff` commands for the selected channels.
4. During Queue processing. Sequences are worked off one by one, and for each Sequence, Injections are worked off one by one. For each Injection, an Instrument Method has been defined, and the script of which is executed as part of the processing of that individual Injection.
5. During Queue processing, when any driver generates an [Abort] error, the Emergency Method (if one has been set for the Queue), will be run. Emergency Methods will usually contain a few lines of script only:

```
Pump.Flow = 0.000
UV.Lamp = Off
Oven.TemperatureControl = Off
FractionationValve.Position = Waste
```

to set the instrument into a safe state.

A driver needs to be prepared for some of these actions executing in parallel. For instance, while an Injection is running (and the corresponding Instrument Method executes), a user might operate an ePanel control or click on an ePanel's script button. There will be a warning that this interferes with a running Queue, but users can choose to execute the interactive action nevertheless.

At the same time, triggers that have been defined in the Instrument Method may determine that their run condition is now true, so additional scripts for the trigger blocks will start to be executed. Drivers (especially those using method download) that cannot support concurrent execution need to implement additional preflight checks to reject this.

The script language also supports conditional execution (If, Else, While, Trigger). Unfortunately, script actions executed conditionally are not preflighted, so drivers may also need to implement additional run-time checks if they can't handle conditional execution.

8.2 Events during Script Execution

Consider the following script:

```
{InitialTime} Sampler.Position = H12
                Sampler.Volume = 10
0.000          UV.Autozero
                Wait UV.Ready
                Sampler.Inject
                UV_VIS_1.AcqOn
1.000          UV_VIS_1.AcqOff
End
```

It is a stripped down version of a typical instrument method used for an Injection. The following is the list of events that are dispatched to the drivers when this script executes:

OnBroadcast (SampleStart)	to all drivers - this is an injection run
OnBroadcast (ClockStart)	to all drivers - script clock has started
OnLatch	to all drivers - "InitialTime" commands follow
OnSetProperty (Position, H12)	only to driver owning "Sampler" device
OnSetProperty (Volume, 10)	only to driver owning "Sampler" device
OnSync	to all drivers - "InitialTime" commands complete
OnLatch	to all drivers - "0.000" commands follow
OnCommand (Autozero)	only to driver owning "UV" device
OnSync	to all drivers - no more "0.000" commands for now
OnBroadcast (Hold)	to all drivers - instrument is now waiting for UV.Ready
<i><time passes while detector performs autozero operation></i>	
OnBroadcast (Continue)	to all drivers - Waiting ended, execution resumes
OnLatch	to all drivers - more "0.000" commands follow
OnCommand (Inject)	only to driver owning "Sampler" device
OnSync	to all drivers - no more "0.000" commands for now
OnBroadcast (Hold)	to all drivers - instrument is now waiting for inject response
OnBroadcast (Inject)	to all drivers - an injection is in progress
<i><time passes while sampler performs injection></i>	
OnBroadcast (InjectResponse)	to all drivers - inject response
OnBroadcast (Continue)	to all drivers - Waiting ended, execution resumes
OnCommand (AcqOn)	only to driver owning "UV_VIS_1" device
OnSync	to all drivers - "0.000" commands complete
OnBroadcast (ZeroCrossed)	to all drivers - script clock is now at a time >0.000
<i><60 seconds pass></i>	
OnLatch	to all drivers - "1.000" commands follow
OnCommand (AcqOff)	only to driver owning "UV_VIS_1" device
OnSync	to all drivers - no more "1.000" commands for now
OnBroadcast (ClockStop)	to all drivers - script is complete, script clock stopped
OnBroadcast (SampleEnd)	to all drivers - injection run is complete

8.3 Aborting Injection runs

- During an injection run, any driver can abort the queue by issuing a message with level Abort. The injection will be set to Interrupted; if an emergency program has been set for the queue, it will run next, then, processing will stop.
- During an injection run, any driver can abort the current injection run by calling `AbortSample`. The injection will be set to Interrupted; queue processing will continue with the next injection.
- During “Wait for Injection Response”, autosampler drivers can inform the system that the vial was not found (**NotifyMissingVial**), which will generate a tailored error message and otherwise behave similar to `AbortSample` as stated in the previous item.
- Warnings and Errors during any script execution will only be logged, but won't terminate script execution. Queue execution will continue unaffected.
- Aborts during any script execution will terminate script execution. Outside queue processing, nothing else happens after an abort error; especially no emergency method will be run.
- When you lose communication to the device, issue an Abort message giving as much detail as possible, and then call `IDDK.Disconnect`. Apart from this, let the user manage Connected/Disconnected state explicitly and simply perform all necessary tasks in `IDriver.Connect` and `IDriver.Disconnect`. Failure during these operations should be reported to the DDK via a `System.Exception`. Make sure that whenever the driver disconnects, unmanaged resources are cleaned up properly so that a reconnect is possible at all times.
- An easy way to simulate an Abort error during testing is to execute an assignment of a null value, such as `UV_VIS_1.Wavelength = UV_VIS_1.Retention`. (`UV_VIS_1.Retention` is null if the channel doesn't currently acquire data)

8.4 Runtime Messages to Instrument Audit Trail

The DDK offers methods which allow the output of text messages to the Chromeleon Instrument Audit Trail so that the user can read them. Chromeleon writes one Instrument Audit Trail for each instrument and an additional Instrument Audit Trail for global events.

The main purpose of these audit trails is to document the instrument run. For instance, the part of the Instrument Audit Trail which was written during an injection run is stored with that injection in the according sequence.

The preferred method to output messages into the Instrument Audit Trail is writing the message for a specific device:

```
void IDevice.AuditMessage(AuditLevel auditLevel, string messageText);
```

Then the message is included in the instrument-specific audit trail. The name of the device raising this message will be recorded together with the message text.

The message text should be localizable.

The `auditLevel` parameter specifies the error level, one of the following four options:

AuditLevel.Message:

This level should be used for common informational messages. Please avoid writing messages which are not relevant to the user (messages important for the developer only).

AuditLevel.Warning:

The warning level should be used for any event which might be unexpected to the user and might influence the sequence run. However, no immediate danger to the instrument hardware and no kind of data loss is expected. Examples:

- Wear parts should be replaced in the near future
- Sequence run is delayed until a door will be closed
- The instrument cannot use a method parameter as specified. A slightly different value is used while no effect on the analysis output is expected (otherwise this must be abort level or, even better, this should be rejected in ready check)

AuditLevel.Error:

The error level is often used for messages which are directly raised by the firmware in one of the following situations:

- A serious instrument hardware problem occurred. However, the current analysis is expected to be finished correctly. Example: Leak sensor reports problem, might be defective.
- A temporary problem occurred during the instrument run. While effects on the analysis and the data output cannot be excluded it is not expected that the problem will endure. Example: Temporary communication problems, commands might being lost (without command loss this should be informational level or should not be reported at all)

In the second situation described above the driver can call `IDevice.AbortSample()` to abort the current Injection. However, this is not a general rule.

AuditLevel.Abort:

This level must be used in the following situations:

- A serious instrument problem occurred. To prevent a danger from the hardware the run must be aborted.
- A serious instrument problem occurred. While there isn't an immediate danger to the instrument hardware, a correct analysis is no longer possible.
- Chromatography results will be distorted because data acquired from the instrument got lost. It must be assumed that this isn't a one-time problem (otherwise error level would be acceptable).
- The instrument method or command cannot be executed in the current instrument's state (e.g. UV detector acquisition is started while lamps are off).

The instrument run will be aborted.

When viewing the Instrument Audit Trail later, filters can be set for the device name and the error level.

There is also a method to output messages into the global Instrument Audit Trail:

```
void IDDK.AuditMessage(AuditLevel auditLevel, string messageText);
```

These messages are used rarely as they are visible to the user in Instrument Configuration application only.

9 Preflighting and Ready Check

9.1 Preflighting Scenarios

Any request by the user for a driver to do something will be submitted to the driver via execution of an instrument method. Before a method is run, a preflight happens to allow the driver to check the method for validity. You need to be aware of the following scenarios:

- In the Instrument Method Editor ([CM6] PGM Editor), the user executes the Method Check ([CM6] Check) command. This is referred to as a **Semantic Check**. At this point, you don't know whether the method will be run as part of an injection run or not, and you don't know anything about what might have happened before the method actually gets executed. Thus, only identify errors that can be spotted when just looking at the method from top to bottom.

Don't evaluate anything against the current device or driver state. Typical examples:

- Upper and lower limit for a property are both specified in the script, but are inconsistent.
 - The number of channels to be acquired doesn't work at the data collection rate that the script specifies.
 - A wavelength is requested for acquisition that requires a lamp that has been explicitly turned off in the script.
- Users can place Script buttons on ePanels. The button's property sheet has a page where the script is specified that the system runs when the button is clicked. On that page, a Check button is available, and it also triggers a Semantic Check with the same considerations as stated in the previous item. Note that the script can also comprise a single line, e.g. for executing a particular command with or without parameters.
 - Users can place interactive elements (check boxes, edit boxes) on ePanels that are linked against a particular (writeable) property. When the user changes the state of the control and the focus leaves the control, a single line script representing the change is generated. There is no semantic check. The single line script is immediately preflighted (a **pre-exec check**) and executed if the preflight didn't return messages with level error or abort. If a message with level warning is returned, the user has to confirm whether execution should take place.
 - Users can use the Inject, Flow, and Acquisition dialogs from the CM6 toolbar to specify a set of related parameters for these functions ([CM7]: only Monitor Baseline is available). When the property sheets are closed with OK, a multi line script corresponding to the settings is generated (it only contains those settings that are different to the current state of the driver) and submitted to a pre-exec preflight. Again, there is no semantic check available, and execution will take place immediately if no messages with level error or abort result from the preflight. If a message with level warning is returned, the user has to confirm whether execution should take place.
 - Queue ([CM6] batch) preflighting. In the Batch dialog [CM6] or on the 'Queue' tab of the 'Instruments' view [CM7], users can add sequences. In CM6, they can also add standalone PGMs to the batch (Standalone PGMs are rarely used – they were designed to allow users to submit a startup/equilibration procedure prior to the first sequence or a shutdown procedure after the final sequence). Also, an emergency method and a program to recover from power failure (only CM6) can be defined. In the queue dialog, users can explicitly request a Queue/Batch Ready Check by clicking on the Ready Check button or (implicitly) by clicking on the Start button. **Batch preflighting** will first iterate over all sequences to access all injection records. Each injection record references a particular method. For efficiency, each method is preflighted exactly once at this point, even if more than one injection uses the same method. Then, batch preflighting processes the queue in the execution order – any standalone PGM is preflighted (only CM6), any sequence is preflighted by iterating over all injections, providing injection type, position and volume plus a reference to the corresponding method preflight for that injection. Finally, emergency methods and power-failure PGM (only CM6) are preflighted. At this point, your preflighting should perform ready checking based on the current device/driver state, answering the question *Will this method run successfully on the instrument as it is right now?* Ideally, you might want to track some state variables (e.g. eluent consumption, a waste container filling level, disk usage or part wear) for the complete batch ready check, summing up each injection's contribution. Again, once the batch ready check is complete, any errors or aborts from the ready check will prevent the batch from being started. Any warnings have to be confirmed by the user. The batch starts, and the method for the first sequence's first injection (or the initial standalone PGM) is again preflighted as a pre-exec preflight. Note that, as some time may have been passed, the device/driver state may now be different to what it was at batch ready check time, so the preflighting results may be different now. Any error/abort message will terminate the queue. All following methods (and in CM6 standalone PGMs) are pre-exec preflighted as soon as the previous item has been processed and the queue engine has moved on to the next item. Note that the transition from one sequence to the next is in general not visible to drivers; and that

the items in the queue may change while the queue runs (users can add/modify/delete items except the currently running injection) and that you might need to rerun the batch ready check for all remaining items at some point during batch processing. See `UpdatesWanted`.

- It is important that you do not modify the device/driver state during any preflight. It might happen that more than one preflight runs at the same time (consider e.g. user changing a value on a panel while a lengthy batch preflight runs.) If you need to save private state during preflight, do so by using the `IRunContext.CustomData` object.
`OnTransferPreflightToRun()` is the earliest possibility to modify the global device/driver state because at this point, all preflighting/ready checking has been completed and only one `OnTransferPreflightToRun()` call will be processed at any particular time.
- During a preflight run, Abort and Error level messages are treated alike.
- Be prepared that during preflighting, the "NewValue" passed in for set property and command parameters can be **null**. This is the case when the method contains an arithmetic expression rather than a constant. The expression will only be evaluated at run time; so for such commands, you cannot perform any up-front validation. If your driver cannot cope with such expression assignments (e.g. because the device supports method download only so that the method must be known at injection start) your ready check must reject such methods. [CM7] If formulas use values from Custom Variables (CV), these values are generally treated as expressions during preflighting, the execution engine will, however, attempt to provide the resolved value as a constant at the earliest possible time (i.e., Sequence CVs are already resolved during Queue preflighting, whereas Injection CVs will only be resolved for the pre-exec preflight of each individual injection.)

9.2 Preflight Types

Depending on the user's interaction with the software, different types of preflights may get executed. The `args.IRunContext` that is available in all preflight event handlers indicates what the type of the current preflight is.

1. When the user operates an ePanel control that is linked to a property or a single command:
`IRunContext` has `IsManual` and `IsPreExec` set.
`OnTransferPreflightToRun` will follow immediately if the preflight is successful.
2. When the user operates a script button on an ePanel:
`IRunContext` has `IsManual` and `IsPreExec` set.
`OnTransferPreflightToRun` will follow immediately if the preflight is successful.
3. When the user operates the Monitor Baseline button:
`IRunContext` has `IsManual` and `IsPreExec` set.
`OnTransferPreflightToRun` will follow immediately if the preflight is successful.
4. When the user edits an Instrument Method in the IME and clicks on the "Check Method" button:
`IRunContext` has `IsSemanticCheck` set.
No `TransferPreflightToRun` will follow in this case.
5. When the user selects an Instrument Method to be used as the Emergency Method (on the Instrument's Queue tab):
`IRunContext` has `IsReadyCheck` and `IsEmergencyMethod` set.
No `TransferPreflightToRun` will follow in this case.
6. When the user clicks the "Ready Check" button on the Instrument's Queue tab:
`IRunContext` has `IsReadyCheck` and `IsSample` set.
NB: For each Instrument Method that is referred to by one of the Injections in all the Sequences that are currently in the Queue, a preflight gets executed (once per Instrument Method, even if it is used by more than one Injection. Injection CVs are not resolved at this time.)
No `TransferPreflightToRun` will follow in this case.
7. When the user clicks the "Start" button on the Instrument's Queue tab (and 6. has been omitted):
Same as for 6.

8. During Queue processing. Sequences are worked off one by one, and for each Sequence, Injections are worked off one by one. Prior to each Injection, the corresponding Instrument Method will be preflighted:
`IRunContext` has `IsSample` and `IsPreExec` set. Injection CVs will be resolved at this time.
`OnTransferPreflightToRun` will follow immediately if the preflight is successful.

9.3 Preflight Results

For any preflight, if any driver emits an audit message with level Error (or Abort) in one of the `OnPreflightXxxx` event handlers, the message is shown to the user (in Check Results or the daily audit trail), preflighting is aborted and no further processing for the preflied method will occur. In particular, `OnTransferPreflightToRun` will never be called and the corresponding `IRunContext` object will be removed.

If only audit messages with level Warning have been emitted, these messages will be shown to the user. For `IsReadyCheck/IsPreExec` preflied, the user can decide whether the method should nevertheless be executed. If the warnings are accepted by the user, the script will eventually run.

If no audit messages or only audit messages with level Message have been emitted, these messages will be shown to the user. The script will eventually run.

9.4 Preflighting Events

Preflighting a method with a property assignment:

```
0.000 MyProperty = 1
      End
```

Preflight Begin

- A run context object is created by the DDK driver instance and sent to the DDK, see `IRunContext`
- The DDK takes a snapshot of all properties in the driver and saves it to the run context, see `IRunContext.Precondition`
- Plug-In's `OnPreflightBegin` handler is called if implemented

Preflighting a new Timestep:

- New time step is detected in DDK driver instance
- This information is sent to the DDK
- Plug-In's `OnPreflightLatch` handler is called if implemented

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`
- Plug-In's `OnPreflightSetProperty` is called if implemented

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

9.5 Preflighting and Ready Checks: Do's and Don'ts (applies in general to CM6 as well)

- Correct syntax and range for constant values already enforced by system
- Check Method (semantic check) - Look only at script. Any contradicting specifications?
 - Lamp state vs. AcqOn vs. Wavelength?

- TempCtrl vs. Temp.Nominal?
 - Lower < Nominal < Upper Limit?
 - Hold (Wait, Message) after inject/zero crossed broadcast?
- Ready Check (ready check for queue) - Same checks as for semantic check. Take current instrument state into account.
Caveat: The same method may be used for more than one injection run. This will result in only one Ready Check Preflight for the script, so one can only check against current instrument state once. If instrument conditions change due to sequence run, this can only be detected in subsequent PreExec Checks or during the Batch ready check procedures.
 - PreExec Check (about to be run) - Same checks as for ready check. Take current instrument state into account.
Caveat: if instrument conditions have changed since Queue Ready Check (due to sequence run or manual actions), results may differ from Queue Ready Check. Results are documented in Daily Audit Trail and Injection Audit Trail.
 - **Caveat:** Formulas in property assignments/command parameters are only evaluated at execution time. All preflights will see <null> for the corresponding value. However, formulas that contain references to Custom Variables will get resolved to constants as early as possible.
 - **Caveat:** Some checks may not make sense for emergency methods. (NB: Emergency Methods will always be run, even if the PreExec preflight emits errors. However, queue start is not possible if Queue Ready Check reports errors for the Emergency Method). Be less strict when checking scripts that are flagged as `IsEmergencyMethod`.

10 Some Examples of Instrument Control

10.1 Simple property update

Let's consider a driver that, in its `IDriver.Init`, has:

```
IDevice m_Device = m_DDK.CreateDevice("Sampler", ...);
IProperty m_washVolumeProperty = m_Device.CreateProperty("washVolume", ...);
m_washVolumeProperty.OnSetProperty += new
SetPropertyEventHandler(washVolume_OnSetProperty);
void washVolume_OnSetProperty(SetPropertyEventArgs args)
```

When the method line

```
Sampler.washVolume = 20
```

executes, the RTK dispatches this to the driver plug-in and fires the `OnSetProperty` event. The event handler `washVolume_OnSetProperty` gets called. What should the implementation for this event handler do?

1. Determine the new value for the property from `args.NewValue`.
2. Perform range validation on the new value, and possibly any other state checks that might be needed. NB: See remark on calculated values below.
3. Forward the change request to the communication class. (This could be called directly, or via posting to some internal job queue, which the communication class polls. While you might block in `OnSetProperty`, we recommend executing all blocking code on a separate communication thread and returning quickly from these event handlers.)

4. The communication class forwards the change request to the firmware, using the communication resource.
5. The firmware processes the change request. If the value is visible at the hardware's display, the new value should now show there.
6. The firmware acknowledges the change request using the communication resource.
7. When receiving the acknowledgement, the communication class calls `m_washVolumeProperty.Update()` with the new value.
8. DDK and RTK ensure that the new value is forwarded to all connected clients so that the new value is visible in all UI.

If any step fails, do not call `m_washVolumeProperty.Update()`, but issue an audit message (with level Error or Abort) instead.

Remember that CM7 also allows formula to be used in scripts, e.g.

```
Sampler.washVolume = 2 * Sampler.Volume
```

Such formulas are only evaluated when the method line executes. This has the following consequences:

- IMW/IME input validation doesn't check whether the value is within the allowed range.
- In all preflights, `args.NewValue` will be null. Preflighting can't therefore check whether the value is within the allowed range either.
- Drivers using method download must reject such scripts as they need to access the value already at preflight time.
- Drivers using direct control should therefore perform their own range validation in `OnSetProperty` (unless they are sure that the firmware will finally reject an out-of-range value, in which case a rejection by the firmware is also acceptable).

10.2 Dependent Properties

Let's consider a driver that offers two properties that depend on each other. Depending means that either there is a constraint such as `UpperLimit > LowerLimit` or that you need to send both of them to the firmware in one operation.

Consider these methods

```
0.000 Oven.Temperature.LowerLimit = 20
      Oven.Temperature.UpperLimit = 40
1.000 ...
```

```
0.000 Oven.Temperature.UpperLimit = 40
      Oven.Temperature.LowerLimit = 20
1.000 ...
```

executing when the current values are `LowerLimit = 50` and `UpperLimit = 60` (which is consistent with the constraint). In both cases we want to end up with `LowerLimit = 20` and `UpperLimit = 40` (which is also consistent with the constraint). The second script would, however, temporarily violate the constraint when executed on a line by line basis.

For these scenarios, it is recommended performing checks in `OnPreflightSync` and operations in `OnSync`:

1. Remember the current values of the two properties in `OnPreflightLatch resp. OnLatch` (e.g. with the `IRunContext.CustomData` object). This essentially tells you that script execution has progressed to a new time step.
2. Update the remembered values in `OnPreflightSetProperty resp. OnSetProperty`.
3. `OnPreflightSync resp. OnSync` will inform you that, for the current time step, no more script lines will need to be executed by your driver. At this point, check for the values and whether the constraint is violated or not. If the latter, forward the change request to your communication class, using the two remembered values. Otherwise, report the violation via a suitable audit message.
4. Once the communication class has acknowledged the change, update the two Chromeleon properties to the new values.

A similar strategy can be used for firmwares that only allow you to update a larger number of related properties in one operation, such as for a sampler that offers a struct that defines all speed settings and that can only be updated in the firmware as a whole. Or for a pump, where a ramp step includes all flow and composition values in one command.

10.3 Simple command

Let's consider a driver that, in its `IDriver.Init`, has:

```
IDevice m_Device = m_DDK.CreateDevice("UV", ...);
IProperty m_readyProperty = m_Device.CreateStandardProperty(StandardPropertyID.Ready, ...);
ICommand m_autoZeroCommand = m_Device.CreateCommand("AutoZero", ...);
m_autoZeroCommand.OnCommand += new CommandEventHandler(autoZero_OnCommand);
m_autoZeroCommand.ImmediateNotReady = true;
void autoZero_OnCommand(CommandEventArgs args)
```

When the method line

```
UV.Autozero
```

executes, the RTK dispatches this to the driver and fires the `OnCommand` event. The event handler `autoZero_OnCommand` gets called. What should the implementation for this event handler do?

1. Perform range validation on the command parameters (if any), and possibly any other state checks that might be needed.
2. As the autozero operation probably takes a significant amount of time, and the detector will not be ready for run while the operation takes place, set the Ready property to false:
`m_readyProperty.Update(false)`
3. Forward the request to the communication class. (This could be called directly, or via posting to some internal job queue, which the communication class polls.)
4. The communication class forwards the request to the firmware, using the communication resource.
5. The firmware starts the operation. The hardware's display should now show that the operation is in progress.
6. The firmware acknowledges the start of the operation using the communication resource.
7. After some time, the operation completes in the firmware. The hardware's display should now no longer show that the operation is in progress.
8. The firmware signals the end of the operation using the communication resource.

9. When receiving the signal, the communication class calls `m_readyProperty.Update(true)`.
10. DDK and RTK ensure that the new value is forwarded to all connected clients so that the new value is visible in all UI.

If any step fails, ensure to call `m_readyProperty.Update(true)`, and issue an audit message (with level Error or Abort) instead.

Setting `ImmediateNotReady` on `m_autoZeroCommand` is needed to ensure that a method similar to

```
UV.Autozero
Wait UV.Ready
```

waits correctly. Without the additional flag, the change to the Ready property (step 2 above) might not have been processed quickly enough, and there is a chance that the Wait command (which is handled by the system) sees an outdated value (true). If this ever happened, the expected waiting would not occur.

11 Method Download

Drivers for modules that support method download only may use the following general approach for instrument control:

1. During preflight, build up the desired method. Typically, you would iterate over the `ProgramSteps` collection in `OnPreflightEnd`. Store the resulting method in `runContext.CustomData` in your own format, if necessary.
Notes:
 - You can also do this later on in step 2!
 - Do not store the resulting method anywhere else, as multiple preflights might run at the same time and you wouldn't be able to keep things apart properly.
 - Do not interact with the module in any way during preflighting. At this point you cannot assume that the method that is being preflighted will get executed soon (or even at all).
2. After the Pre-Execution preflight has been successful, the method is handed over to the RTK for execution. Your driver will be informed of this by a call to the `OnTransferPreflightToRun` event handler.
3. In this event handler, set `m_readyProperty.Update(false)` and start to download the method to the module. Also, instruct the firmware to apply the initial settings of the method immediately. Sample preparation and/or injection should not be started yet, and timetable execution in the firmware should also not be started yet.
4. When the method download/method apply has completed fine, set `m_readyProperty.Update(true)` to signal that your module is ready for run.
5. As part of your implementation of the Inject command, start sample preparation/injection. Timetable execution in the firmware should not be started yet.
6. When the module has injected, generate the inject response and start timetable execution in the firmware.
7. (If there is no Inject command, or you are not the driver that provides the Inject command, start timetable execution in the firmware on `Broadcast.RetentionZeroCrossed`.)
8. As soon as your driver determines that the method is running in the module, set `IDevice.DelayTermination` to true.

9. As soon as your driver determines that the module has completed running the method and is idle again, set `IDevice.DelayTermination` to false. This will ensure that the Injection will not change to Finished until your module has completed processing.

You must set `IDevice.ImmediateNotReady` (in `IDriver.Init`) to true to ensure that waiting for the download to be complete works reliably. Injection will not be triggered before all drivers have completed their method downloads (via the `Wait xxx.Ready AND yyy.Ready` line generated in the injection method script)

Be aware that your driver may only delay termination for a short time. The DDK will force termination after the time specified by `IDevice.TerminationTimeout`, which defaults to 10 seconds. If your device takes a significant extra time to complete processing, set `IDevice.TerminationTimeout` to a reasonable value.

Make sure that you reset `IDevice.DelayTermination` to false in case an Abort error occurs. Stop the method execution at the module, wait for any cleanup operations to be complete, and then reset the flag.

Make sure that your `OnCommand` and `OnSetProperty` handlers don't interfere with the module while downloading/running an instrument method. If possible, these handlers should nevertheless execute the desired actions when no instrument method is running to support interactive device control using ePanels.

Make sure that once you allow termination, nothing should happen in driver, device or firmware that will make the device non-Idle (even temporarily) to prevent the next pre-exec ready check from seeing a non-idle, non-ready device.

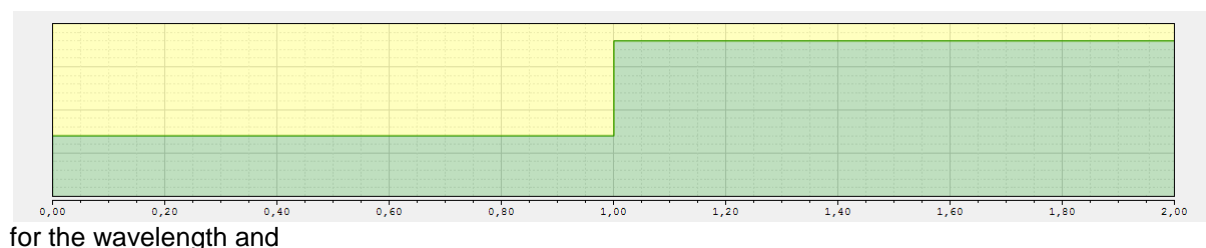
12 Ramp Syntax

Consider the following two examples:

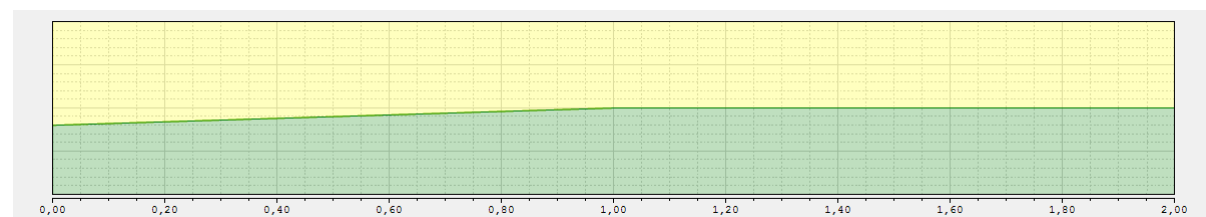
```
0.000 Wavelength = 200
1.000 Wavelength = 220
2.000 End
```

```
0.000 Flow = 0.4
1.000 Flow = 0.5
2.000 End
```

Clearly, the user expects the following behavior of the system



for the wavelength and



for the flow change.

In order to specify which behavior is intended, you need to set `IsRamped = true` for the `IProperty` that you have created for the Flow property.

Note:

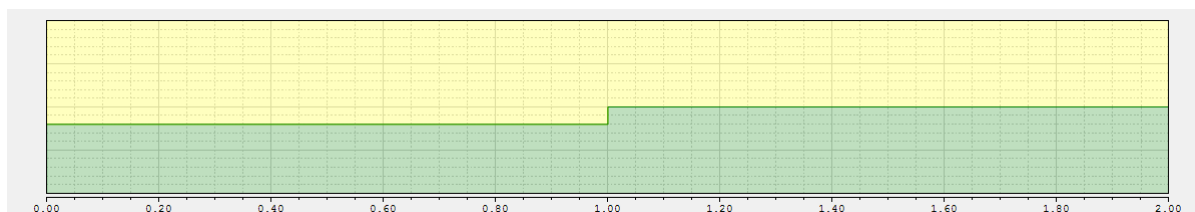
- This must be an `IDoubleProperty` so that intermediate values can be represented properly.
- This must be set during the `IDriver.Init` function; i.e. when the symbols are defined. It is not possible to set this later on or dynamically!

Based on whether `IsRamped` is set, the method execution engine generates different events during method execution:

Execution Time	IsRamped = false (Wavelength)	IsRamped = true (Flow)
0.000	OnSetProperty() with args.NewValue = 200	OnSetRamp() with args.Start = 0.4 args.End = 0.5 args.Duration = 1.000
1.000	OnSetProperty() with args.NewValue = 220	OnSetProperty() with args.NewValue = 0.5
2.000	<Nothing>	<Nothing>

In the (rare) case that a step-like change for a property with ramping behavior is intended, the method can express this by

```
0.000 Flow = 0.4
1.000 Flow = 0.4
      Flow = 0.5
2.000 End
```



Execution Time		IsRamped = true (Flow)
0.000		OnSetRamp() with args.Start = 0.4 args.End = 0.4 args.Duration = 1.000
1.000		OnSetRamp() with args.Start = 0.4 args.End = 0.5 args.Duration = 0.001
1.001		OnSetProperty() with args.NewValue = 0.5
2.000	<Nothing>	<Nothing>

- Be prepared to receive a mix of `SetProperty` and `SetRamp` for a single property if it has `IsRamped` set.
- Watch out for Hold/Continue broadcasts
- Watch out for ramps requested prior to `InjectResponse/ZeroCrossed` broadcasts. Instruments using method download may not be capable of executing them.

Ramped GC oven temperatures are usually specified by stating an initial value, a change rate, and duration. The IME plug-in should provide this representation in the UI, but must convert to/from the start/end representation used by the method.

Initial Temperature: 300 °C
Increase Temperature by 50 K/min for 2 minutes
->

```
0.000 Oven.Temperature.Nominal = 300 [°C]
2.000 Oven.Temperature.Nominal = 400 [°C]
```

Future versions of the DDK will have additional helper classes to ease working with ramped properties.

Non-linear ramps (as provided by a few LC and IC pumps) are not supported by the DDK for now, as each vendor typically uses a different curve function, and the exact function needs to be known not only to the driver but also to other Chromeleon components, such as the gradient plot and the reporting engine.

13 Generating and Editing Of Script

When the user interacts with an ePanel or operates the Monitor Baseline button, the client application will automatically generate suitable script and pass it to the Instrument Controller for execution, transport is via the WCF connection that the client has for each Instrument.

Instrument, SmartX and Emergency Methods are created and edited in the client via the New Instrument Method Wizard (IMW) resp. the Instrument Method Editor (IME). These methods are stored in the data vault, from which the Instrument Controller will read them when they are being used during processing of the Queue.

The IMW generates script based on the currently selected instrument. All wizard pages should default all property values presented in the IMW UI to the current state of the corresponding property value that is currently active in the instrument (i.e., the value that an ePanel would show for this property).

Drivers using method download might want to implement a property (most likely, this would be a rather large string property containing some driver-private XML) that represents the method that is currently active in the module, and implement an IMW plug-in that accesses this XML to fill in the defaults for the IMW. This would have the benefit of being able to pre-fill time-tables as well.

Note that in order to support true client-server architecture, the IMW/IME plug-in is not allowed to talk to the module directly, nor does it have a way to actively poll for the current method through the driver.

IMW/IME plug-ins might also need to know exactly what the configuration is that the driver currently uses. Ideally, the plug-in determines this solely from the symbol table (e.g., in order to find out whether the sampler has a tray cooling option, check whether a symbol `Sampler.TrayTemperature` is present), but the driver could also provide the driver-private configuration XML as part of the symbol table that is available in the client.

```
using Dionex.Chromeleon.Symbols.Extensions;
const short cat_xml3rdPartyData = 293;
IAttributeDefinition attributeDefinition = device as IAttributeDefinition;
attributeDefinition.AddString(cat_xml3rdPartyData, ConfigurationXML);
```

```
using Dionex.Chromeleon.Symbols.Extensions;
const short cat_xml3rdPartyData = 293;
ISymbol deviceSymbol = symbolTable.Child(deviceName);
ISymbolExtension extension = deviceSymbol as ISymbolExtension;
IAttributeList attributeList = extension.Attributes;
IAttribute attribute = attributeList[cat_xml3rdPartyData];
ConfigurationXML = attribute.Value as String;
```

Note: In Chromeleon 7.2 or newer one can use the `ISymbol.StoreAdditionalInformation` and `IDevice.UpdateAdditionalInformation` functions to store the additional information.

The IMW/IME should provide some basic validation:

- Text boxes or other input fields should check for valid input and whether values are within range
- Individual plug-in pages should check for additional constraints (such as whether a minimum and a maximum value specified on the same page are consistent)

Be aware that in the IME, the user might be modifying the script content directly. The script grid only checks for correct script syntax and performs range validation when constant numbers are assigned to properties and parameters. Additional constraint checks are therefore necessary in the semantic check preflighting, which the driver needs to implement. The semantic check is triggered when the user clicks "Check Method" in the IME.

Do not expect that all instrument methods conform to whatever the IME plug-in would generate and validate. Be prepared that users might edit the generated scripts.

The system provides contextual information to the IME plug-in that allows to determine whether the IMW or the IME is shown. For the wizard, the UI should enforce that the user specifies all values (thus, check boxes should be two-state, and empty input boxes should be rejected by input validation), whereas for the editor, the UI must be able to cope with information that is missing from the script (thus, check boxes should be tri-state, and empty input boxes are acceptable).

[CM7] The preferred UI is the dialog-based Device View which all drivers must provide. The Script View is seen as a tool for expert users, its uses should not normally be needed.

[CM6] The preferred UI is the Commands view, where the user can edit the PGM script directly, as not all drivers provide a Device View, and the DDK framework is less sophisticated in CM6.

13.1 Accessing the method directly (only CM7)

In Chromeleon 7 the instrument method is grouped into several stages. Depending on the method you create, different stages are available.

For separation methods (these are used for Injections) the following stages exist:

Stage name	Meaning/Content
Instrument Setup	Initial time commands. Contains all settings defined by SimpleControls
Equilibration	Commands with negative times. Define start parameters for gradient and ramps.
Inject Preparation	Prepare system for Inject command (Wait xx.Ready and Autozero)
Inject	Inject command (or sequence)
Start Run	Start Data Acquisition
Run	Time tabled commands (gradient and ramps steps, wavelength switching, valve position changes)
Stop Run	Stop Data Acquisition
Post Run	Commands executed after data acquisition (e.g. prepare system for next injection, cool-down)

Each stage has a time and a default time step with the same time. All time steps within a stage refer to the method time and not to the start time of the stage.

	Time	Command	Value
0	▷ {Initial Time}	Instrument Setup	
10	▲ -4,000	Equilibration	Duration = 4,000 [min]
11		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
12		PumpModule.Pump.%B.Value	0,0 [%]
13		PumpModule.Pump.%C.Value	0,0 [%]
14		PumpModule.Pump.%D.Value	0,0 [%]
15		PumpModule.Pump.Curve	5
16	▲ 0,000	Inject Preparation	
17		Wait	FLD.Ready And PumpModule.Pump.Ready
18	▲ 0,000	Start Run	
19		FLD.FLD_FlowCell.AcqOn	
20		PumpModule.Pump.Pump_Pressure.AcqOn	
21	▲ 0,000	Run	Duration = 20,000 [min]
22		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
23		PumpModule.Pump.%B.Value	0,0 [%]
24		PumpModule.Pump.%C.Value	0,0 [%]
25		PumpModule.Pump.%D.Value	0,0 [%]
26		PumpModule.Pump.Curve	5
27	▲ 20,000		
28		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
29		PumpModule.Pump.%B.Value	0,0 [%]
30		PumpModule.Pump.%C.Value	0,0 [%]
31		PumpModule.Pump.%D.Value	0,0 [%]
32		PumpModule.Pump.Curve	5
33	▲ 20,000	Stop Run	
34		FLD.FLD_FlowCell.AcqOff	
35		PumpModule.Pump.Pump_Pressure.AcqOff	
36	End		

The “Instrument Setup” stage has the time “Initial Time” which is represented as negative infinity. The Chromeleon runtime engine will ignore the time span between the initial stage and the following stage, so the clock will start at the time defined by the first time mark of that stage. The stages “Inject Preparation”, “Start Run” and “Run” all have the same time of zero. The duration of a stage is determined by the time step with the greatest time. For the above shown example the AcqOff commands will be executed at ~ 24 min after the start of the injection run (4 min for equilibration + time needed for injection + 20 min run time after inject).

The following time steps are pre-defined by the Chromeleon instrument method editor:

- MethodTime.Initial (Double.NegativeInfinity)
- MethodTime.Zero (0.0)
- MethodTime.Minimum (Int.Minimum+1)
- MethodTime.Maximum (Int.Maximum)

Property and command steps are accessed via the stage and the time step. Each stage offers a collection of time steps; and each time step offers a collection of symbol steps (ISymbolStep is the base of IPropertyStep and ICommandStep).

In order to access a symbol step, retrieve the stage which contains the step and then choose the correct time step. If the stage and time step are unknown one has to iterate over all existing time steps in the method. See the following code snippet for details:

```

/// <summary>
/// Returns the first step found in the method for the given symbol.
/// </summary>
/// <param name="sym">The symbol of the command or porperty for which a
/// step should be found.
/// </param>
/// <returns>A symbol step if the method contains a step for the given
/// symbol; otherwise null.
/// </returns>

```

```
private ISymbolStep GetSymbolStep(ISymbol sym)
{
    if (_editMethod != null && sym!=null)
    {
        var timeSteps = _editMethod.Stages.GetAllTimeSteps();
        if (timeSteps != null)
        {
            foreach (var timeStep in timeSteps)
            {
                var symStep = from step in timeStep.SymbolSteps
                               where step.Symbol == sym
                               select step;

                if (symStep.Count() >0 )
                    return symStep.First(elem => elem !=null);
            }
        }
        return null;
    }
    return null;
}
```

Once the step for a given property was retrieved the symbol step can be casted to `IPropertyStep`. The `IPropertyStep` provides access to the “SymbolValue” of the step. With the help of the “SymbolValue” one gets access to the value assigned to the step. The symbol value of the property step can be cast to an `INumericSymbolValue` or `IStringSymbolValue` depending on the property type. The value of the step will be null for expressions/formulas. In order to check the expression assigned to a property step one has to use the `ValueExpression` object. If the expression evaluates to a constant value, the value can be retrieved via one of the `ValueAsConstant` properties of the `ValueExpression` object.

The following example shows how to retrieve a constant value for a property step where the right hand side of the assignment is a formula:

```
private double? GetInitialValue(ISymbol symbol)
{
    var stage = EditMethod.Stages.GetStage(SeparationMethodStage.InstrumentSetup);
    var found = stage.TimeSteps[0].SymbolSteps
        .Where(step => step.Symbol == symbol)
        .OfType<IPropertyStep>()
        .LastOrDefault(pstep =>
            pstep.ValueExpression.ValueAsConstantDouble.HasValue);

    return found == null ? null : found.ValueExpression.ValueAsConstantDouble;
}
```

Command symbol steps are slightly different than property steps. Instead of offering a symbol value a parameter step list is available. For each parameter step the value can be accessed via the `SymbolValue` or the `ValueExpression` property depending on the right hand side of the assignment.

The DDK V2 framework offers two different approaches for creating symbol steps. The first approach covers the step creation via the `IStepLink` interface. In order to create an object implementing this interface use the `IEditMethod` function `CreateStepLink()`. This interface allows you to define the position for the step within the time step. However, the framework cannot guarantee that the symbol step will be placed exactly at the defined position. The reason therefore is that other plug-ins can also place their steps at the same position in the time step. In this case the loading order of the plug-ins will decide where exactly the step is placed. (However, since individual drivers operate in parallel anyway, this is usually not a problem. If you need to control the order of commands, use the order provided by the stage model and ensure commands end up in the correct stages.)

Consider the following example:

There are two plug-ins named `imeA` and `imeB`. Both plug-ins create one step in the Equilibration stage and use the step link position “First”. Assume that `imeA` was loaded before `imeB`. The resulting method will have the following entries in the Equilibration stage:

```
0.0 Equilibration
    imeAStep = "a"
    imeBStep = "b"
```

If the loading order of both plug-ins is inverse, the method will look like this:

```
0.0 Equilibration
    imeBStep = "b"
    imeAStep = "a"
```

The DDK V2 framework will however assure that the steps created by a single plug-in will be created in the defined order. Furthermore all steps created via this interface need to be created during the `Initialize()` of the `IPage` object. `IStepLink` interface is used by all Simple Controls shipped with Chromeleon.

If more flexibility is required in terms of defining the time step within a stage and the position of the step within a certain time step is of less importance the step creation via the `IEditMethod` function `CreateStep()`. After creating a step one has to add this step to a time step. Any existing time step can be used or a new one can be created.

13.2 Using UI components, data binding, constraint checkers

Design Overview

Instrument method editor plug-ins are loaded into the DDK host which is part of the Chromeleon client. In order to find the corresponding plug-in for a driver, the DDK host iterates over all plug-ins found in the directory "DDK\V2" and its sub directories. For each plug-in the `DriverCert.xml` file will be analyzed and the driver ID is compared to the drivers available on the instrument for which a method is created. If the driver ID of the plug-in matches a driver ID, the plug-in will be loaded. So in order to load a plug-in during method creation the following prerequisites need to be fulfilled:

- A driver with the same driver ID is part of the instrument
- A connection to the instrument is available
- A valid `DriverCert.xml` for the plug-in exists
- Plug-in provides a public class that implements the `IInitEditorPlugIn` interface

Furthermore the framework holds several collections for system components. Each plug-in should register system components like pumps, detectors, diagnostic channels and inject devices. The registration of components allows the framework to display additional controls in order to guide the user through the method creation process, e.g. if two plug-ins register an inject device the framework will automatically display an injector selection page in the Wizard. In order to gain access to the system components one can use the `IEditorPlugIn.System` or `IPage.Component.SystemComponent` properties. Notice that some of the functions for registering system components return objects which are required by some complex controls, e.g. registering a pump will return an `IPumpDescription` which is used by the `PumpGradientPage` control.

As mentioned above each plug-in needs to implement the `IInitEditorPlugIn` interface. Via this interface the part of the symbol table for this plug-in becomes accessible. After analyzing the symbol table the plug-in should create a page for each device. Each page needs to implement the interface `IPage` in order to add the page to the Wizard and editor and to gain access to the instrument method.

Walking the symbol tree

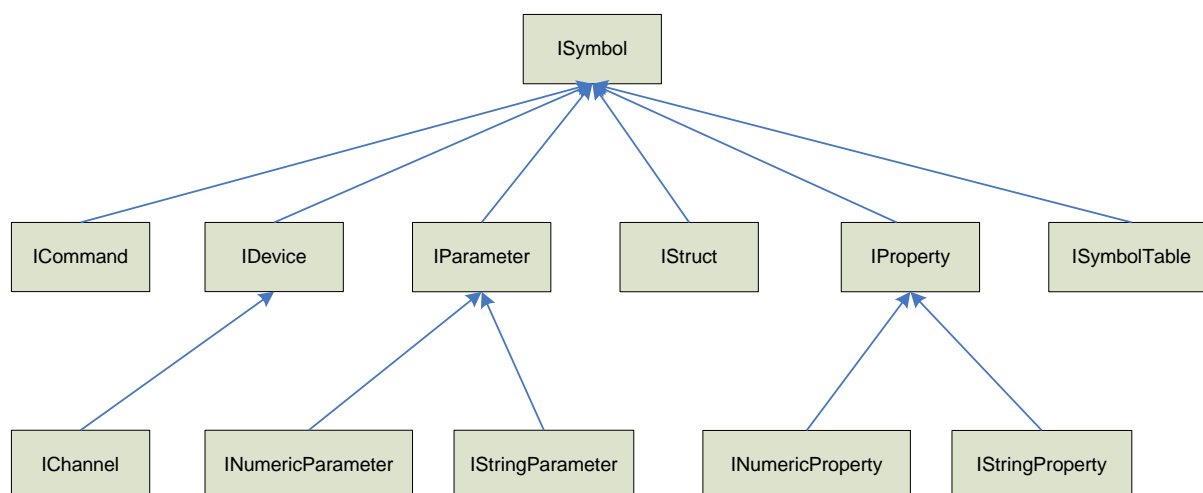
Each editor plug-in gains access to the symbol table via the `IEditorPlugIn` object provided by the `Initialize` of the `IInitEditorPlugIn`. The framework will only pass the relevant part of the symbol table to the plug-in. Given the following symbol table:

```
<Device Name="FLD" Level="Normal" Description="FLD device" Type="general">
  <Property Name="DriverVersion" Level="Expert"/>
  <Property Name="Ready" Level="Normal" >
    <Enum-Value value="0" Name="NotReady" />
```

```
<Enum-Value value="1" Name="Ready" />
</Property>
</Device>
<Device Name="PumpModule" Level="Normal" Description="HPG3X00RS Pump Module">
  <Command Name="GeneralLeak_StartFlow" Level="Expert" />
  <Command Name="GeneralLeak_MoveCamDelivery" Level="Expert" />
  <Property Name="Pressure_StopPhase_Initial" Level="Expert" />
  <Property Name="Pressure_StopPhase_Drop" Level="Expert" />
</Device>
<Device Name="Sampler" Level="Normal" Description="Sampler." Type="general">
  <Command Name="RotorSealChanged" Level="Expert" />
  <Property Name="InjectValveSwitches" Level="Expert" />
  <Property Name="InjectValveErrors" Level="Service" />
  <Struct Name="Temperature" Level="Normal">
    <Property Name="Nominal" Level="Normal"/>
    <Property Name="Value" Level="Normal"/>
  </Struct>
</Device>
```

the plug-in for the pump will only retrieve the part of the symbol table starting at element “<Device Name=“PumpModule” .../>” with all its child elements. The sampler plug-in will only retrieve the element “<Device Name=“Sampler”.../>” with all its children.

Each XML element will be mapped to an `ISymbol` object or a derived class. See the following inheritance diagram for details:



In order to access properties and commands below the main device symbol the `Child` function of the `ISymbol` object needs to be called. This function supports also the component notation; e.g. to access to temperature nominal value of the sampler call `symb.Child("Temperature.Nominal")`. Furthermore it is possible to get an enumeration of all children having a certain type by calling `ISymbol.ChildrenOfType(SymbolType type)`. `Type` can be `IDevice`, `IStructure`, `IProperty` and so on.

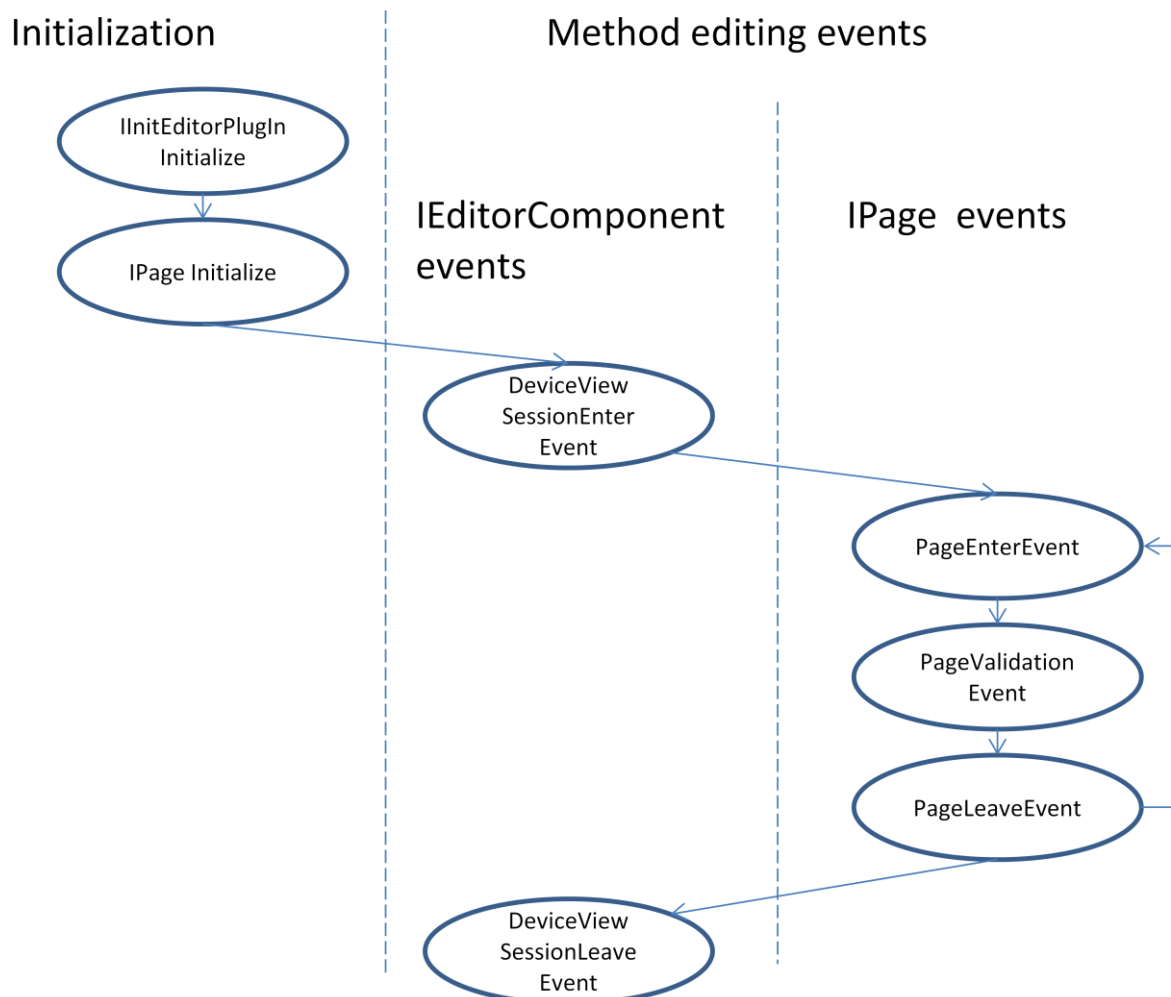
For retrieving data stored as an attribute by the driver for a symbol one has to use the function `GetAttribute` of `ISymbol`. Currently two different functions are available for accessing attributes in the symbol table. If the driver stores a value for the attributes with number 293 (third party data) or 0x0402-0x0416 (GC ramp attributes) then the `GetAttribute` function with parameter `AttributeCategory` has to be used. Currently the following categories are supported:

```
public enum AttributeCategory
{
    GcRampAttributes,
    Xml3rdParty,
}
```

For all other attributes just specify the XML attribute name in order to retrieve the value.

Note: Not all attributes available for DDK drivers will also be available in the Chromeleon client. If an attribute is missing in the Chromeleon client symbol table contact the DDK hotline.

Editor plug-in and page lifecycle



1. **IInitEditorPlugIn.Initialize**
Inspect the symbol tree. Create pages for each device. Register system components.
2. **IPage.Initialize**
Create user controls, constraint controller and method steps via IStepLink
3. **DeviceViewSessionEnterEvent**
This event notifies plug-ins that the device view session has been entered and the method can now be modified by the plug-in.
4. **PageEnterEvent**
Is raised when user activates the page or when the device view session is reopened in consequence of the following actions:
 - method check operation
 - 'save' or 'save as' operation
 Read back settings from the method and update controls. Attach to events. Run validation checks (user could have changed settings in the script view). Do not modify method.

5. **PageValidationEvent**

Is raised when the page is active and the user:

- selects another page
- performs a method check operation
- performs a 'save' or 'save as' operation

Execute validation checks. Displaying an error will stop user from leaving page. Do not rely on Windows.Forms events like Validating and Validated. These will not be raised.

6. **PageLeaveEvent**

This event is raised after successful validation of the page.

Write back settings to method. Detach from events. Create additional method steps if needed.

7. **DeviceViewSessionLeaveEvent**

The device view session has been closed and the method can't be modified by the plug-in any longer.

Please note that the device view session is closed and reopened each time the method is saved by the user - even if the device specific editor page keeps open. In this case the events 5, 6, 7, 3 and 4 are raised.

The device view session will also be closed when the user switches to the script view and it will be reopened when switching back to the plug-in page. This is because write access to the method cannot be granted to the script view and the DDK plug-in concurrently.

Controller Components

In general the DDK V2 differentiates between two different types of controllers. One the one hand there are the UI controllers and one the other hand there are the constraint controllers.

UI controllers (PropertyStepControllers) are used with several user controls shipped with Chromeleon. These user controls can be reused for creating editor plug-ins. A PropertyStepController controls a PropertyStep in the instrument method. In Wizard mode it will create a new step and apply a default value (value from the symbol table for the given property just before launching the instrument method editor). In editor mode the controller attempts to connect an existing property step. If successful, it will display and edit its value; if not an empty control will be displayed. See the next sub chapter "UI Components" for a complete list of reusable controls that make use of PropertyStepControllers.

Constraint controllers are not linked to any user control. These controllers can be used for validating user input. Currently four different controller types are available defined in the namespace `Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components` (add a reference to `Dionex.DDK.V2.InstrumentMethodEditor.Components.dll` from the `bin\DDK\V2` directory).

Controller name	Description
BinaryConstraintController	Checks constraints between two values that are controlled by PropertyStepControllers. Displays an error message when the check fails and prevents the user from leaving the page. The constraints can be defined by the plug-in developer. The following predefined constraints exist: <code>LowerThanConstraint</code> , <code>LowerOrEqualConstraint</code> , <code>GreaterThanConstraint</code> , <code>GreaterOrEqualConstraint</code> .
EnableController	Controls enabling and disabling a collection of controls. The enable condition can be defined by the plug-in developer. If the condition evaluates to true the controls are enabled, otherwise disabled.
MinMaxController	Defines a controller for device settings with upper limit and lower limit value properties. Typical examples for using the MinMaxController are: - Temperature setting - Pump pressure settings
MinMaxDeviationController	Defines a controller for device settings with upper limit, lower limit, nominal value and maximum deviation properties.

MinMaxNominalController	Defines a controller for device settings with upper limit, lower limit and nominal value properties.
-------------------------	--

Refer to the following editor plug-in examples shipped with the DDK setup for a usage example:

- BinaryConstraintController: ExampleLCSystem.EditorPlugIn\PumpGeneralPage.cs
- MinMaxNominalController: TempCtrlDriver.EditorPlugIn\TempCtrlPage.cs

UI Components

The DDK V2 framework offers several UI controls from simple controls for text input to complete pages with plots and grids. Please reuse these controls where possible in order to achieve a common look and feel throughout the entire product.

Simple Controls

All simple controls have in common that they wrap existing Windows Forms controls and create a method step in the “Instrument Setup” stage. The user input is automatically validated against the type information (range, number of digits) defined in the symbol table. If the user input is invalid the background color of the input field is set to red and an error provider with a suitable error message is displayed. On the right side of the control a help bubble is displayed. The tool tip that becomes visible, when hovering the mouse above the help bubble, displays the content of the description attribute found in the symbol table. The following simple controls are available in the namespace (Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components)

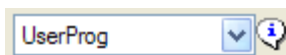
PropertyStepTextBox

A text box controlling an instrument setup property step.



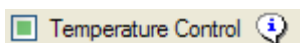
PropertyStepComboBox

A combo box controlling an instrument setup property step.



PropertyStepCheckBox

A check box controlling an instrument setup property step.



PropertyRangeLabel

A label that connects to a Chromeleon symbol in order to display the allowed range of a Chromeleon symbol.



All simple controls can be used within the Windows Forms Designer. The property “SymbolPath” needs to be set to the name of the property. If the property is not a direct child of the main device symbol use the component notation, e.g.

```
<Device Name="PumpModule" Level="Normal" Description="HPG3X00RS Pump Module">
  <Struct Name="Pressure" Level="Normal" Description="The current pump pressure.">
    <Property Name="Value" Level="Normal" Description="The current pump pressure."/>
    <Property Name="LowerLimit" Type="int" Min="0" Max="1034" Unit="bar"/>
    <Property Name="UpperLimit" Type="int" Min="0" Max="1034" Unit="bar"/>
  </Struct>
</Device>
```

In order to link a PropertyStepTextBox to the “LowerLimit” property of the pump set the symbol path property in the designer to “Pressure.LowerLimit”.

Complex Controls

Complex controls can be grouped by module types. For almost every complex control the editor plug-in examples show how to use it. See please refer to the examples for detailed code examples.

Detector/Channel controls

ChannelGridControl

A component containing a grid that allows editing acquisition times and instrument setup parameters for a range of channels.

No	Channel	Excitation WL [nm]	Emission WL [nm]	Sensitivity	PMT	Filter wheel
1	<input checked="" type="checkbox"/> Emission_1	300,0	350,0	1	Pmt1	Auto
2	<input checked="" type="checkbox"/> Emission_2	300,0	350,0	1	Pmt1	Auto
3	<input checked="" type="checkbox"/> Emission_3	300,0	350,0	1	Pmt1	Auto
4	<input type="checkbox"/> Emission_4					

Note: The ChannelGridControl has a property `AcquisitionTimesVisible`. If it is set to true, one can also change the acquisition start and end times. If the device/driver allows to specify the data acquisition time window independent from the analysis run time, we recommend to link this property with a checkbox 'Edit acquisition times' in the upper right corner of the UI. Default should be 'deselected', i.e. the data acquisition time should match the analysis time.

Channel start settings:							<input checked="" type="checkbox"/> Edit acquisition times
No	Channel	Wavelength [nm]	Bandwidth [nm]	RefWavelength [nm]	RefBandwidth [nm]	Acquisition on [min]	Acquisition off [min]
1	<input checked="" type="checkbox"/> UV_VIS_1	230,0	1	Off	1	Start Run	Stop Run
2	<input checked="" type="checkbox"/> UV_VIS_2	254,0	1	Off	1	Start Run	Stop Run
3	<input checked="" type="checkbox"/> UV_VIS_3	270,0	1	Off	1	Start Run	Stop Run
4	<input checked="" type="checkbox"/> UV_VIS_4	300,0	1	Off	1	Start Run	Stop Run
5	<input checked="" type="checkbox"/> 3DFIELD			Off	1	Start Run	Stop Run

ChannelTimeGridControl

A component that allows changing channel properties during the run of an instrument method.

No	Time [min]	Channel	Excitation WL [nm]	Emission WL [nm]	Sensitivity	PMT	Filter wheel
1	0,000	Emission_1		350,0			
2	1,000	Emission_1			4		
3	2,000	Emission_1					





Pump controls





For pumps two complete pages are available which can be reused:

PumpGeneralPage

A complete page to edit general settings for a pump flow device.

Solvents

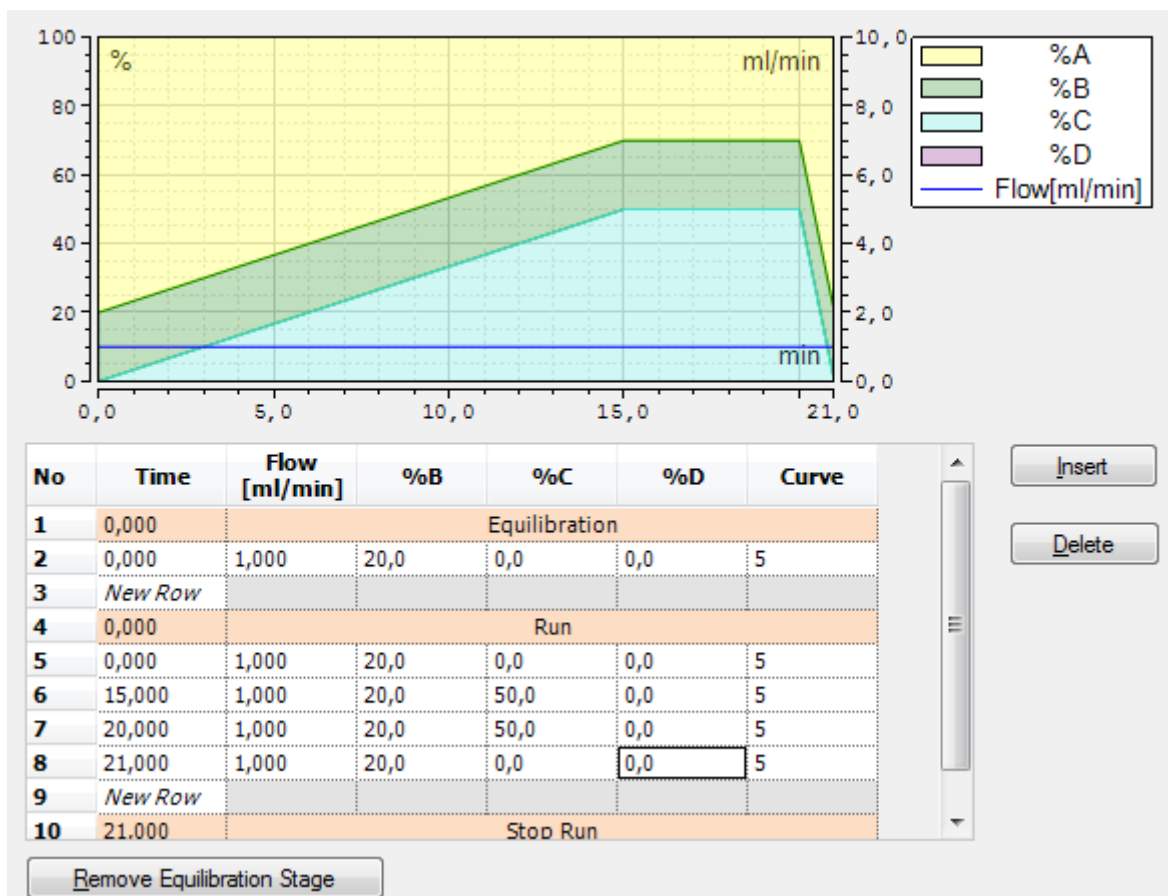
	Name
%A:	<input type="text" value="%A"/> 
%B:	<input type="text" value="%B"/> 
%C:	<input type="text" value="%C"/> 
%D:	<input type="text" value="%D"/> 

Pressure Limits		Maximum Flow Acceleration/Deceleration	
Lower Limit:	<input type="text" value="0"/>  [0...620 bar]	Up:	<input type="text" value="Infinite"/>  [Infinite...9999,999 ml/min ²]
Upper Limit:	<input type="text" value="620"/>  [0...620 bar]	Down:	<input type="text" value="Infinite"/>  [Infinite...9999,999 ml/min ²]

PumpGradientPage

A complete page for editing flow and solvent composition for a pump flow device. In order to use this page a PumpDescription object is required. The pump description contains information about the pump, like flow symbol, number of solvents, curve property.

Hint: Make use of the `IFlowHandler` in the driver.



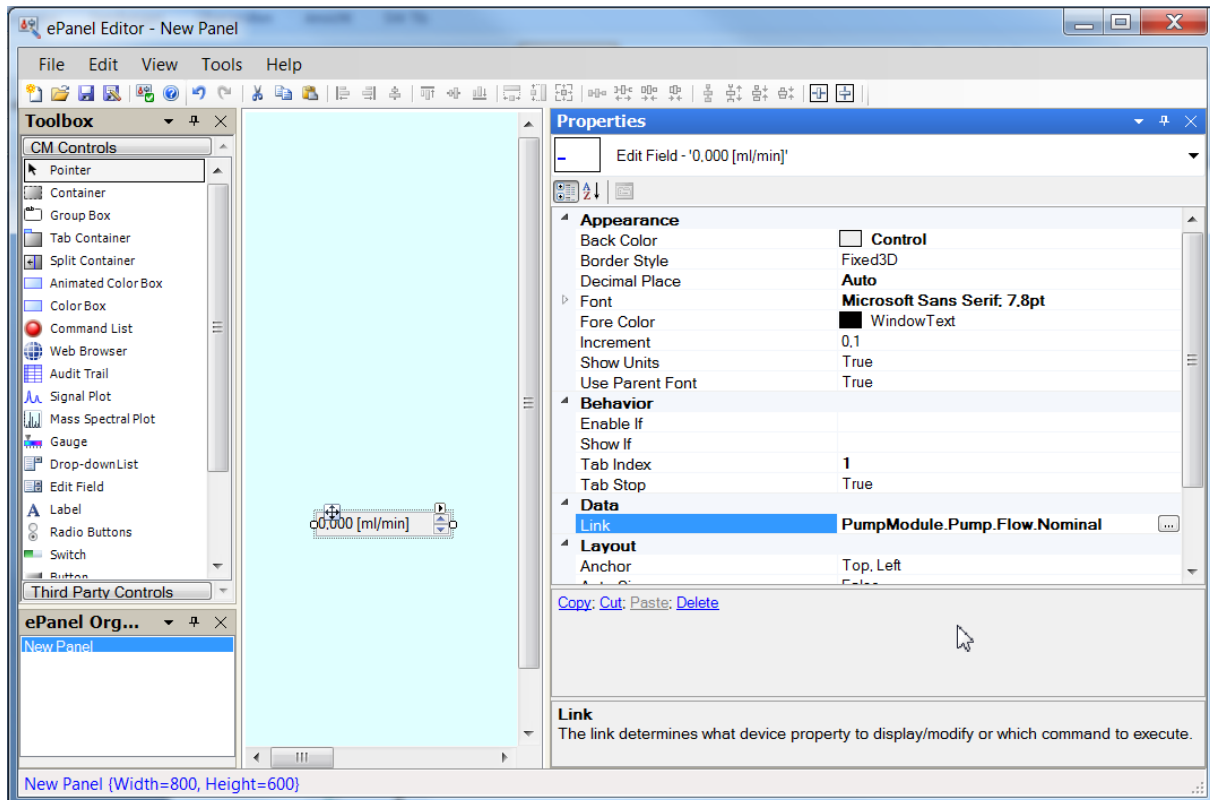
General controls

For each property that can change its value during the run a time table grid is offered:

No	Time	ValveLeft	ValveRight
1	{Initial Time}	6_1	1_2
2	0,000	6_1	6_1
3	1,000	1_2	6_1

14 ePanel Support

In the ePanel editor CM Controls can be linked to Properties directly. Using macros inside the link is not discussed in this section. For using macros please see 4.



But controls can be linked to a Struct too. In this case the usage of DefaultGetProperty and DefaultSetProperty are of importance. The following examples show common use cases:

Temperature Example:

Temperature.Nominal
Temperature.Value

Nominal is the DefaultSetProperty. Value is the DefaultGetProperty. Value is a read only Property.

Linking to an Edit Field the following behavior will be observed: Entering a new set point Temperature.Nominal will see that change. Immediately the Edit Field will display the Temperature.Value property. So the slowly changing Temperature can be observed via this control. Note that for an Edit Field to work properly with a Struct both DefaultSetProperty and DefaultGetProperty need to be assigned to Properties.

Linking to a Gauge controls Nominal and Value are represented. Via the slider on the right side the Nominal set point can be changed.

Pressure Example:

Pressure.LowerLimit
Pressure.UpperLimit
Pressure.Value

Value is the DefaultGetProperty. Value is a read only Property. The Pressure Struct should not be linked to an Edit Field. For the Gauge control two sliders are provided for LowerLimit and UpperLimit automatically.

IDevice.CreateFlowHandler() does create Properties and Structs with appropriate DefaultSetProperty and DefaultGetProperty settings.

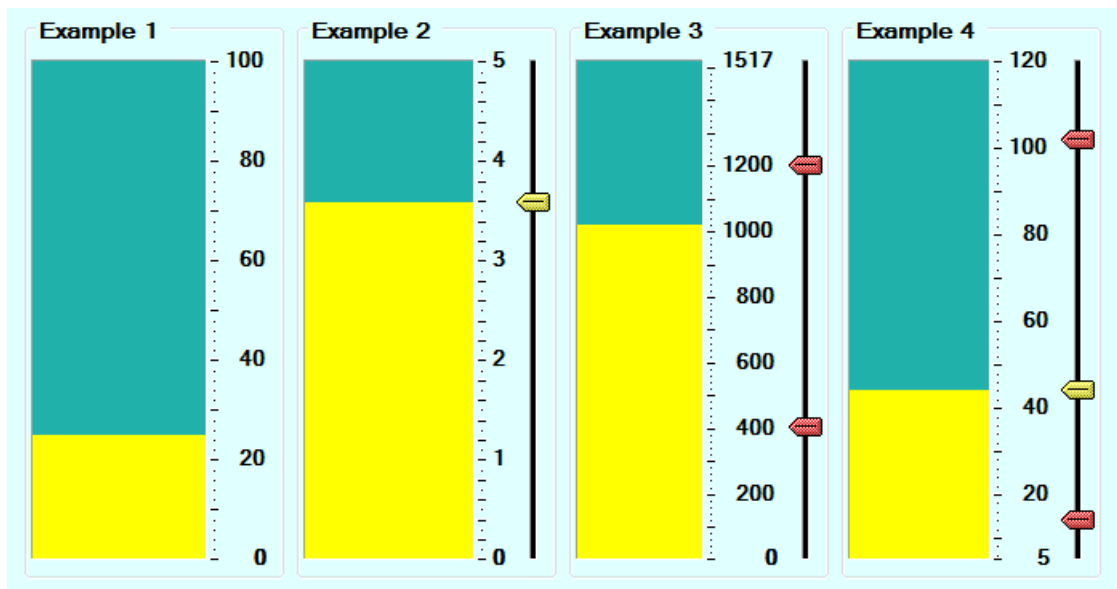
Gauges linked to different Struct setups are seen below:

Example 1: Struct with assigned DefaultGetProperty.

Example 2: Struct with assigned DefaultGetProperty and DefaultSetProperty.

Example 3: Struct with assigned DefaultGetProperty. LowerLimit and UpperLimit Property

Example 4: Struct with assigned DefaultGetProperty and DefaultSetProperty. LowerLimit and UpperLimit Property



15 Sampler Drivers

15.1 Setting up sampler components

15.1.1 Setting up the symbol table

IInjectHandler: Central Interface for auto-sampler or injector functionality

An instance of this interface can be created with `IDevice.CreateInjectHandler`. This function also creates a 'Position' property (`IInjectHandler PositionProperty`), a 'Volume' property (`IInjectHandler VolumeProperty`) and an 'Inject' command (`IInjectHandler InjectCommand`) in the corresponding device.

These three symbols have special semantics:

- The `Position` and `Volume` properties are initialized from the values entered in the Injection List at the beginning of an injection; i.e. before the corresponding instrument method is started
- The `Inject` command will temporarily suspend method execution until the sampler driver has signaled that injection has taken place and analysis can be started.

The `Position` property type must be defined appropriately so that it reflects the number and layout of available vials and trays in the sampler. See chapters 15.2 and 15.3 for details.

15.1.2 Required `IInjectHandler` interfaces and their implementation

- `NotifyInjectResponse()`: To be called by the driver after successful injection.
Note: The driver needs to call this as soon as possible after the injection valve has switched from Load to Inject. Don't wait until any post-switch operation (such as a wash or homing) is complete.
Timing is critical here, as this will start the clock that the RTK uses during method execution. If there is a significant time jitter (e.g., because your driver has to poll the module for this information, or your communication resource has random timing (such as a TCP/IP

connection)), then you need to support routing the inject response from the module into your driver by some hardware means (e.g., using a relay controlled by the sampler's firmware and a TTL input provided by another driver. See `IInputPortProvider` and `IInputPortUser` for details.)

- `NotifyMissingVial()`: To be called by the driver when the module has determined that there is no vial at the specified position. This will abort the current Injection (state changes to Interrupted), but sequence processing continues with the next Injection.
- The driver can also call `IDevice.AbortSample()` to abort the current Injection. Use this for other fatal situations during injection. In addition, issue an audit message with details about the error cause before calling the function.

15.1.3 Creating a minimum auto-sampler/injector device

- Create `IDevice`
- Create `IInjectHandler` with `IDevice.CreateInjectHandler`
- Implement `IInjectHandler.InjectCommand.OnCommand` to perform the injection
- When injection has finished call `IInjectHandler.NotifyInjectResponse()`
- For example code see
 - `CmDDKExamples\AutoSampler\`

15.2 Supporting the New CM7 Sequence Wizard

The CM7 Sequence Wizard (NSW) guides users through the creation of new sequences. It will first ask the user for instrument selection. If the instrument has more than one device providing an Inject command (i.e., two samplers/injectors are present), a dialog to select one of these samplers is shown next. After this step, the NSW knows what sampler will be used for the sequence.

If that sampler provides a `_TrayDescription`² property, the current contents of this property are evaluated and transformed into a graphical representation of the tray as well as into an ordered list of valid positions. If the property is absent, no graphical representation of the tray is available. However, an ordered list of valid positions can be determined from the named values associated with the `IInjectHandler.Position` property. If the property has no named values, the list of valid positions is simply all numbers that are within the numerical range of the property.

For details on how to set up `_TrayDescription`, see [1]. In Chromeleon 7 a set of helper classes and interfaces exists to ease this task:

- With `IInjectHandler.CreateRackLayoutDescriptionProperty(Int32 size)` one can create a `_TrayDescription` property with the required string length
- With `IInjectHandler.CreateRackLayoutDescriptionBuilder()` one can create a `IRackLayoutDescriptionBuilder` helper interface instance, that allows to set up a rack layout description without knowing the low level rack layout description syntax.
- For further details check reference[1] and the Autosampler example project in the DDK examples solution provided with the DDK.

The UI also validates user's input for injection volume against the range of the `IInjectHandler.Volume` property.

15.3 Supporting the Injection List

The Injection List validates input for Position against the Tray Description (if there is one) resp. the `IInjectHandler.Position` property (numerical range or named values if the symbol has any). The Injection List validates input for Volume against the range of the `IInjectHandler.Volume` property.

² To avoid confusion: Naming is for historical reasons; today we would call it `_RackDescription`

[CM7] If there is more than one sampler present on the currently assigned Instrument, all ranges are joined together.

[CM6] A heuristic is used to determine which sampler is used; essentially, it is the device used for injection in the PGM for the second sample of the sequence.

15.4 Overwriting Position and Volume in the Instrument Method

Instrument methods can use any of the following to override the values for Position and Volume that have been specified in the Injection List:

```
Sampler.Position = <value>
Sampler.Volume = <value>
Sampler.Inject
```

```
Sampler.Inject Position = <value>, Volume = <value>
```

If your driver implements direct control, you don't need to worry about this. The DDK will (at the start of the Injection) fire `OnSetProperty()` for Position and Volume with the values that have been specified in the Injection List; then (when the method commands get executed), additional `OnSetProperty()` for Position and Volume events will be fired. Thus, when `OnCommand()` for Inject is being fired, your driver already has been informed about the values to use.

After the Injection has finished, the values in the Injection List will be updated automatically to whatever has been used by the Inject command so that Reporting and Calibration functionality operate on the correct data.

Be aware, however, that whenever you inspect Position and Volume at preflight time, you might need to keep track whether the Inject command will use the values from the Injection List or the values resulting from overriding in the script.

If your driver implements a method download you have to check the program steps for 'Position' and 'Volume' property assignments and for possible parameter assignments with the 'Inject' command step. Also take care that the value of the property assignment steps or the parameter assignment is available at preflight time (i.e. no formulas or custom variables (CM6: user defined columns) are used). Otherwise such instrument methods must be rejected.

15.5 Batch Preflighting

Method preflighting will not give you access to the items (Sequences and Injections) that users have placed into the queue. It will only inform the driver about the contents of instrument methods used for injection runs. However, there are scenarios where drivers need to know about Sequences and Injections (and fields such as Injection Type, Position, Volume for each Injection), too.

For this, the system supports Batch Preflighting (this is CM6 terminology for compatibility reasons, and should be Queue Preflighting in CM7). The following events are sent to drivers when the user clicks the "Ready Check" button on the instrument's Queue tab, or when a Queue gets started, or when a running Queue has been modified (sequences added, removed, rearranged, injections added, removed, rearranged, injection type/volume/position/injection method changed).

Relevant events are:

```
IDevice.OnBatchPreflightBegin
IDevice.OnBatchPreflightSample
IDevice.OnBatchPreflightEmergencyProgram
IDevice.OnBatchPreflightEnd
```

In `OnBatchPreflightBegin`, you can set up and preflight-local state information that you might need during the batch preflight (`args.IRunContext.CustomData`).

`OnBatchPreflightSample` is called for each injection that is present in the queue.

`ISamplePreflight` provides access to the Injection's properties (Injection Type, Position, and Volume).

`OnBatchPreflightEmergencyProgram` is called once if, and only if, an emergency method has been specified for the queue.

In `OnBatchPreflightEnd`, you can perform any additional checking before the batch preflight results are returned to the user.

As usual, any audit trail messages that the driver emits in any of these event handlers will be collected and shown to the user in the Check Results list on the Queue tab. If any [Abort] or [Error] messages are present, the Queue cannot be started (or will be aborted if it is already running.)

When it is time to actually execute one of the Sequences in the Queue (note that Queue start might be at any later time, and users can specify that a particular Sequence is only started at some given time in the future), additional events are generated:

- Driver's `OnSequenceStart` is called if implemented whenever a sequence starts
- Driver may store a reference to `ISequencePreflight` (contains information on all Injections that are part of the current sequence), this is valid throughout the whole sequence
- Driver's `OnSequenceEnd` is called if implemented when the sequence has been worked off
- Driver's `OnSequenceChange` is called and the running sequence has changed in any respect
- Old `ISequencePreflight` becomes invalid after `OnSequenceEnd` and `OnSequenceChange`

Note that the drivers do not get informed about the Queue being started or stopped.

Batch preflighting is mainly intended to be used by drivers that support overlapped sample preparation, where a look-ahead is needed so that the driver knows position, volume and instrument method for the Injection(s) that are scheduled after the current Injection.

Batch preflighting can also be used to cross-check the volume specified in the Injection List (where the user's input is only validated against the range of property `Sampler.Volume`) against settings in the Instrument Method that might result in additional constraints. For example, consider two instrument methods for a sampler that has a 10 µL loop and a Volume property with range [0...10 µL]:

FullLoop.meth

```
Sampler.InjectionType = FullLoop
Sampler.Inject
```

PartialLoop.meth

```
Sampler.InjectionType = PartialLoop
Sampler.Inject
```

and the following sequence:

Injection Name	Position	Volume	Instrument Method
Injection 1	A1	10 µL	FullLoop
Injection 2	A2	10 µL	PartialLoop
Injection 3	A3	5 µL	FullLoop
Injection 4	A4	5 µL	PartialLoop

Injection 3 needs to be rejected, because a FullLoop injection can't inject 5 µL. You'll implement this by:

- During preflight of the two instrument methods, remember (in `IRunContext.CustomData`) information such as:
 - The script requests `InjectionType FullLoop`
 - There is an `Inject` command
 - The method does not override the Volume property, so the value from the Injection List is relevant
- During batch preflight, for each injection (`OnBatchPreflightSample`), look at the remembered information for the relevant instrument method and validate against that

injection's volume as specified in the Injection List. Emit an audit message with level [Error] during batch preflight if there is an injection that violates the constraint.

Batch preflighting can be also used to track "queue global" aspects, such as eluent consumption, disk space usage, overall queue run time estimates and the like.

Batch Preflight Example

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

15.6 Blank Runs

A blank run is an Injection run where no analytes are present. Thus, the chromatogram should not result in any peaks. Such runs can be used by post-processing to determine the baseline chromatogram and to provide the (0, 0) data point for calibration curves.

Typically, this is achieved by either not injecting anything at all, or (preferably) by injecting from a vial that has been filled with pure eluent only.

For the purposes of post-processing and calibration, these runs are identified via the Injection Type "Blank", which the user selects in the Injection List when setting up a Sequence.

The script engine evaluates the Injection Type as specified in the Injection List when it is about to process the `Sampler.Inject` command and, at the same time, inspects the `Blank` parameter that the command might have. Eight scenarios are possible:

Injection Type	Command and Parameter	Action
Blank	(absent)	No Inject
Blank	<code>Sampler.Inject</code>	Skipping Inject
Blank	<code>Sampler.Inject Blank=Skip</code>	Skipping Inject
Blank	<code>Sampler.Inject Blank=Inject</code>	Injecting
Any other (Unknown, Standard, ...)	(absent)	No Inject
Any other (Unknown, Standard, ...)	<code>Sampler.Inject</code>	Injecting
Any other (Unknown, Standard, ...)	<code>Sampler.Inject Blank=Skip</code>	Injecting
Any other (Unknown, Standard, ...)	<code>Sampler.Inject Blank=Inject</code>	Injecting

When the resulting action is "Skipping Inject", the corresponding `OnCommand` event is not fired for your driver, and the system will not wait for an inject response to be coming in from the driver. Processing is almost as if the script would not have a "`Sampler.Inject`" line at all.

If your driver requires that an injection takes place for all runs (e.g., because your driver performs method download and the firmware doesn't all to run a method that runs time tables without an injection), you need to enforce that all used instrument methods contain "`Sampler.Inject Blank=Inject`" by suitable preflight checks. In `OnPreflightCommand` for your `Inject` command, access the parameters, find the parameter named "Blank" and test for the expected value. If the parameter is absent, assume "Blank=Skip" behavior.

Make sure that your IME plug-in generates the expected script to make this easier for end users.

You might need to use batch preflighting to access the Injection Type in case you need to figure out up-front (e.g. for drivers performing method download, and the method contains a setting whether an inject should happen or not) whether an inject will be skipped. In instrument method preflighting, determine whether "Blank=Skip" behavior has been requested, and combine this with the Injection Type of the current injection to determine the desired action.

A typical pattern is:

```
m_MyDevice.OnBatchPreflightSample += new
SamplePreflightEventHandler(m_MyDevice.OnBatchPreflightSample);

void m_MyDevice_OnBatchPreflightSample(SamplePreflightEventArgs args)
{
    Boolean doesntInject = true;
    foreach (IProgramStep step in args.SamplePreflight.RunContext.ProgramSteps)
    {
        IBroadcastStep bsStep = step as IBroadcastStep;
        if (bsStep != null)
        {
            if (bsStep.Broadcast == Broadcast.Inject)
                doesntInject = false; // found Inject command
            if (bsStep.Broadcast == Broadcast.InjectBlankRun)
            {
                if (args.SamplePreflight.SampleType == SampleType.Blankrun)
                    doesntInject = true; // will however skip Inject for blank samples
            }
        }
    }
    if (doesntInject)
        ... audit message with error level.
}
```

Be aware that the broadcast `RetentionZeroCrossed` is always generated, whereas the broadcast `InjectResponse` will be missing if a blank run skips the injection.

15.7 Manually triggered Injections

Some samplers might not be able to support injections triggered manually (either from an ePanel Script button, or the F8 command box). This must be rejected in the ready check. A typical code pattern is:

```
m_injectHandler.InjectCommand.OnPreflightCommand += new
CommandEventHandler(InjectCommand_OnPreflightCommand);

void InjectCommand_OnPreflightCommand(CommandEventArgs args)
{
    if ((args.RunContext.IsSemanticCheck == false) && (args.RunContext.IsManual == true))
    {
        m_device.AuditMessage(AuditLevel.Error, Properties.Resources.ERR_NoManualInject);
    }
    <data name="ERR_NoManualInject" xml:space="preserve">
        <value>The instrument is download controlled. Manual inject is not possible.</value>
    </data>
}
```

15.8 Autosamplers and dynamic Tray Changes

- Be aware that the system allows three ways for sample position and volume to be specified. Users can enter these values in the **injection list** ([CM6]: sample table); empty fields are not possible). These values are validated if, and only if, at the time of input an active instrument ([CM6]: Timebase) connection was available to the browser/console and the configuration was the same as are currently used. Thus, you should be prepared to validate each of these values prior to using them, e.g. during batch preflight, pre-exec and at execution time. Also, these values can be **assigned** to the corresponding properties in the method or as explicit **parameters** to the Inject command (overriding any previous values). If this is the case, you should skip checking the values from the sample table and validate only those values that are actually used by the Inject command. (Or you prevent those overrides via a suitable semantic check constraint)
- When the valid injection volume range depends on the selected position, make sure that you perform the validation at the end of the respective time step (i.e. in **OnPreflightSync**) only, to be sure to have seen both values if both are set.
- The system handles the change in position and volume prior to the start of the instrument method of an injection. To inform drivers about this implicit assignment, the `ProgramSteps` for a method contain an additional `PropertyAssignmentStep` for position and volume that does not correspond to a line in the method. These additional items are flagged with

```
(args.RunContext.IsSample && !args.RetentionTime.Initial)
```

- Non-numerical position handling is somewhat awkward. CM6 data sources store the position users have entered in the Position column in the following way:

<optional string part>:<encoded part> Stack1_01:H10

The <optional string part> can be any text [A-Za-z0-9_], and is meant to identify a tray or rack, for example Stack1_01 may refer to tray 01 in stack 1. This part is stored verbatim in the data source.

The <encoded part> is structured as follows

```
0987654321 (digit no)
0TTPSSNNN
| | | | |
| | | | +-- NNN: Numeric part (0-999)
| | | +----- SS: Alpha2 (0 = none, 1-26 = A-Z, 27-99 reserved)
| | +----- PP: Alpha1 (0 = none, 1-26 = A-Z, 27-99 reserved)
| +----- TT: Numeric part, e.g. tray selection (1-99)
+----- 0: reserved, must be 0
```

and is stored as a number in the data source. H10, for example would be stored as 8010. This mapping between data source and cell contents is independent of what autosampler is used for the sequence (the browser doesn't even need a timebase connection for this mapping)
- When the browser has a timebase connection available, the sampler device is identified in the symbol table; input into the Volume column is validated against the range of the Volume symbol of the driver. Input into the Position column is validated against the list of valid positions (see below) as the rack view/sequence wizard provide.

When no timebase connection is available, no input validation takes place; any position that matches the syntax stated above can be entered.
- Rack view/sequence wizard determine the list of valid positions (and their order) in the following way: If the sampler device has a **_TrayDescription** property, this will be evaluated first. Otherwise, the sampler device will be tested for a symbol attribute (only used by native drivers) which might provide a list of valid positions. If this is absent, the range of the Position symbol is used and only numerical positions within that range can be entered (or, if specified, their "named value" aliases)
- For batch ready checking and at execution time, the batch engine retrieves the textual position ("Stack1_01:H10") from the data source layer, and attempts to find this text in the list of the "named value" aliases associated with the sampler's Position symbol. If found, the corresponding numerical position is used for all further processing. For our example, the alias table could look like

```
Stack1_01:A1 <-> 1
Stack1_01:A2 <-> 2
...
Stack1_01:H10 <-> 94
Stack1_01:H11 <-> 95
Stack1_01:H12 <-> 96
```

so an assignment of position 94 would be communicated to the driver. As the driver has created the alias table (via AddNamedValue), it knows the correct semantic of position 94 for further processing.

(If the data source layer returns a pure numerical value, that number will be passed on as is to the driver.)
- If the autosampler supports dynamic tray changes (common for well plate samplers), it therefore needs to set up a large alias table, containing an item for each conceivable position, even if not all positions are currently available on the plate actually present. (For example

```
A1 <-> 1 to
P24 <-> 384
```

) as the alias table is part of the symbol table and cannot be changed dynamically. When the user changes the plate type (which the device either detects automatically or the driver/firmware provides a dedicated plate type property), the driver replaces the **_TrayDescription** value (something that can be changed dynamically) to describe the currently

available positions. Rack view, sequence wizard and input validation will then allow the user to use valid positions only (which are a subset of all positions contained in the alias table)

15.9 Overlapped Sampler Preparation

Some autosamplers support overlapped sampler preparation (e.g. Headspace samplers). Even if the autosampler firmware does not support such a feature explicitly it might be possible to start parts of the sampling process for the next pending injection while the previous analysis is still running (e.g. pre-injection washes, drawing the solvent from the vial).

To support such features, the driver must cover the following features

- **Injection Lock:** Before the preparation of an injection is started the driver must lock the corresponding entry in the Injection List to ensure that the user can't change the relevant parameters (Position, Volume and Instrument Method name) of this injection anymore.
- **Look-ahead:** The driver must be able to retrieve the related information (Position, Volume, and Instrument Method), for the injection to be prepared, which is not the currently running injection.

This is possible by using the `ISequencePreflight` object, which is provided by the `IDevice.OnSequenceStart`, `IDevice.OnSequenceChanged` and `IDevice.OnSequenceEnd` events. With the `ISequencePreflight` object, one can:

- Lock one or several injections by setting `LastPreparingIndex`
- Retrieve information about upcoming injections by using the `Samples` list

To get informed about changes of the running sequence while it is performed, one has to set `IDevice.UpdatesWanted` to `true`.

Note: The following feature is only supported in CM7.2 or newer: If a driver needs to report audit trail information for injections in Preparing state (for instance, the specified vial is not present, and this was determined during preparation), it needs to switch the audit trail context accordingly before it calls `IDevice.AuditMessage`.

Use `IDevice.SetAuditContextSample` to specify the injection that the message is recorded for. The first message issued within the context of a preparing injection will start the injection audit trail for that injection, and those messages will be highlighted with a different background in all audit trail views.

Remember to set the context back to 0 (i.e., current injection) after the message has been delivered to the DDK. Be careful when issuing messages from more than one thread; the context is device-global and not thread-specific.

15.10 Other considerations

A method might not specify an Inject command.

This is rare for injection methods (but common for scripts on script buttons, for emergency methods, SmartX methods), but if your driver cannot process this, preflighting should reject such scripts. In `OnPreflightEnd`, count the number of `IBroadcastSteps` with value `Broadcast.Inject`

A method might specify more than one Inject command.

This is rare, but if your driver cannot process this, preflighting should reject such scripts. In `OnPreflightEnd`, count the number of `IBroadcastSteps` with value `Broadcast.Inject`

A script might specify an Inject command at time <> 0.

This is rare, but if your (sampler) driver cannot process this, preflighting should reject such scripts. In `OnPreflightCommand` for your Inject command, inspect `args.RetentionTime`.

Autosampler procedure exceeds analysis time.

Autosamplers performing lengthy wash operations after injection should support the `DelayTermination` feature (see above) to ensure that a sample run doesn't end prior to the wash

cycle's completion. This will ensure that the sampler is idle and clean before the system attempts to run the next injection. Make sure that any abort scenarios while such wash operations are in progress are handled correctly.

16 Pump Drivers

16.1 Setting up the symbol table

IFlowHandler : Interface for a pump

- Creates `FlowNominalProperty`, `FlowValueProperty` and
- `n ComponentProperties`
- Doesn't care about `%A+%B+%C+%D == 100`
- Doesn't care about ramps and interpolation
- Doesn't support simulation mode automatically

Better interface in planning.

Creating a minimum pump device

- Create an `IDevice` instance
- Create `IFlowHandler` instance with `IDevice.CreateFlowHandler`
- Implement `FlowNominalProperty.OnSetProperty` and `ComponentProperties[].OnSetProperty` for isocratic case
- Implement `FlowNominalProperty.OnSetRamp` and `ComponentProperties[].OnSetRamp` for gradient ramps
- For example code see
- `\CmDDKExamples\ExampleLCSystem\Pump.cs`

Typically, a pump driver also needs to provide a way to define pressure limits:

```
// Create pressure structure
IntPtr tPressure = cmDDK.CreateDouble(0, m_MyDriver.m_maxPressUI, 0);
IntPtr tPressureMax = cmDDK.CreateDouble(0, m_MyDriver.m_maxPressUI, 0);
IntPtr tPressureMin = cmDDK.CreateDouble(0, m_MyDriver.m_maxPressUI, 0);
tPressure.Unit = m_MyDriver.m_PressureUnit;
tPressureMax.Unit = m_MyDriver.m_PressureUnit;
tPressureMin.Unit = m_MyDriver.m_PressureUnit;
m_PressureStruct = m_MyDevice.CreateStruct("Pressure", Resources.HID_Pressure);

m_PressureValue = m_PressureStruct.CreateStandardProperty(StandardPropertyID.Value,
tPressure);
m_PressureValue.HelpText = Resources.HID_Pressure;

m_PressureLowerLimit = m_PressureStruct.CreateStandardProperty(StandardPropertyID.LowerLimit,
tPressureMin);

m_PressureUpperLimit = m_PressureStruct.CreateStandardProperty(StandardPropertyID.UpperLimit,
tPressureMax);

m_PressureStruct.DefaultGetProperty = m_PressureValue;
```

By using a structure like this one, the Pressure properties can be linked to a Slider control on an ePanel, which visualizes all three settings nicely.

The expectation is that the firmware stops the pump automatically when the upper pressure limit is reached. When the pressure is below the lower pressure limit, continue pumping for an appropriate time, then stop the pump. In both cases, report an [Abort] error and set the `FlowValueProperty` to 0.

Restart the flow upon the next change to `FlowNominalProperty` that the driver receives (this might be one that sets `Flow=0` explicitly, e.g., from an Emergency Method). If the firmware doesn't support pressure limit monitoring, emulate it in the driver.

Pump drivers should support pressure units bar, MPa and psi. Allow the user to select the desired pressure unit in the driver's configuration dialog. Create all pressure related properties with the selected unit (make sure that the property ranges are also adapted), and interpret all values that `args.NewValue` delivers in that selected unit. You may convert to a different value if the communication class/firmware operates in a particular unit only. Make sure to convert values received from the communication class/firmware into the selected unit, e.g. for `m_PressureValue.Update()`. Watch out for rounding errors, especially at the borders of the property ranges.

16.2 Broadcast handling

Pump drivers need to react to (at least) three broadcasts:

```
public enum Broadcast
{
    Hold,
    Continue,
    Stopflow,
    ...
    RetentionZeroCrossed,
}
```

A Hold can be caused by various method commands:

- Wait <boolean condition>
- Message "some string to be displayed to the user"
- Sampler.Inject
- System.Hold

In any case, the clock that controls script execution is paused and no more script lines are processed for the time being. Pump drivers need to inform their module that the firmware should also temporarily pause gradient execution, and flow/composition should remain unchanged for the time being (if there is a change in progress).

If your driver uses method download or your pump's firmware doesn't support temporarily pausing gradient time table execution, preflighting must reject such scripts. Iterate over all `ProgramSteps` and check for any `IBroadcastSteps` with argument `Broadcast.Hold` that appear after `Broadcast.RetentionZeroCrossed` in your device's `OnPreflightEnd()` handler.

A Continue can be caused by:

- Wait resolves when <boolean condition> becomes true
- User has confirmed message that has been displayed
- Sampler driver has notified system about the inject response
- System.Continue

In any case, the clock that controls method execution is resumed and the next line is processed immediately. Pump drivers need to inform their module that the firmware should also resume changing flow/composition where it left off (if there is a change in progress).

A StopFlow can be caused by:

- System.StopFlow

The driver needs to do whatever it does for Hold, and additionally set the flow to 0.

`FlowNominalProperty` should remain unchanged; `FlowValueProperty` should be set to 0. Upon Continue, the driver needs to restore the previous flow, update `FlowValueProperty` accordingly and do whatever it does for Continue.

16.3 Special interfaces

Factory functions for special interfaces:

- `void CreatePumpRippleCalculator (double thresholdSignal, double defaultRippleLimit, double rippleMin, double rippleMax, long rippleDigits):`
This is needed for drivers supporting SmartStartup. It instructs the DDK to add properties and commands to the symbol table that support the semantics that the SmartStartup wizard expects.

17 Delivering Data

17.1 2D Signal Processing

IChannel : Interface for channels

- **IChannel** is basically a device with a predefined set of properties and commands:
 - **AcqOn** - command to start data acquisition for this channel.
 - **AcqOff** - command to stop data acquisition for this channel.
 - **Signal** - property with value of most recent data point processed. Automatically updated by the DDK.
 - **Retention** - property with time of most recent data point processed. The value of this property is usually somewhat smaller than the current clock used by the script execution engine (which is `System.Retention`).
- Functions for data handling:
 - `void UpdateData (int ignored, int[] data):`
Send data points at a fixed data rate. The `int` parameter is kept for backward compatibility and is ignored. We recommend setting it to 0.
 - `void UpdateData (DataPoint[] data):`
Maintained for backward compatibility only; **do not use for new drivers**. Use `UpdateDataEx` instead.
 - `void UpdateDataEx (DataPointEx[] data):`
Send data points (double values) that were acquired at random times. For high data rates, we recommend using the corresponding integer function below.
 - `void UpdateDataEx (DataPointInt64[] data):`
Send data points (integer values) that were acquired at random times. **CM7 only**.
- Events used for data handling:
 - **OnDataFinished:**
The DDK fires this event when the driver has forwarded enough data (by using a series of `UpdateData()` calls) to the DDK. This event is typically fired at some point shortly after the `AcqOff` command has been processed (when `Signal.Retention >= nominal AcqOff time`). Get/set the event handler to be called when the data interface has finished to save data and the driver may stop data acquisition.
 - **NoMoreData():**
A driver must call `NoMoreData()` when the hardware has stopped sending data while acquisition is on (i.e., if the data stream stops, but `OnDataFinished` hasn't been fired yet). Call this only if your driver detected a communication failure, a fatal error has occurred, or when the firmware signals that the data stream has ended (e.g., for download controlled modules where the run length also determines the length of data delivery).
- Factory functions for special interfaces:
 - `void CreateNoiseDriftCalculator (double defaultNoiseLimit, double defaultDriftLimit, double noiseMinimum, double noiseMaximum, long noiseDigits, double driftMinimum, double`

```
driftMaximum, long driftDigits):
```

This is needed for drivers supporting SmartStartup. It instructs the DDK to add properties and commands to the symbol table that support the semantics that the SmartStartup wizard expects.

- `void AddPropertyToChannelInfo (IProperty property):`
Add a channel property to the channel info of the data file. Chromeleon stores basic information on a signal as part of the signal's meta data. (You can look at the meta data by exporting a 2D signal to text from the Studio's chromatogram plot.) A driver might want to add some extra information to the meta data if the information that is generated automatically by the DDK is not sufficient.

Creating a minimum channel device:

- Create an `IChannel` instance with `IDDK.CreateChannel()`
- Implement a handler for `IChannel.AcquisitionOnCommand`
- Implement a handler for `IChannel.OnDataFinished`
- Update `SignalFactorProperty`
- Update `TimeStepFactorProperty` and `TimeStepDivisorProperty`
- Send data with `UpdateData/UpdateDataEx`
- For example code see
 - `CmDDKExamples\ChannelTest\`

General implementation approach for direct control:

1. On the first `AcqOn` command (in case your driver supports more than one channel device), use the communication class to instruct the firmware to start delivering data. Note that the first `AcqOn` may not be at script time 0.000, it can be scripted at any later time, too. In this case, make sure to correct the time stamp information accordingly in case you use one of the `UpdateData()` functions that use time stamps.
2. On subsequent `AcqOn` commands for the other channels, you usually don't have anything to do because the firmware already delivers data. (If you need to know how many channels might receive an `AcqOn` during script execution, determine this as part of your script preflighting, then start data delivery for all channels in step 1 already.)
3. Call `UpdateData` repeatedly as soon as the communication class has incoming data available. Make sure that you call the function on the correct `IChannel` device in case the driver supports more than one channel.)
4. Usually, you can ignore all `AcqOff` commands.
5. In `OnDataFinished` (there will be one event fired for each channel), check whether this channel is the last one to finish. (You can iterate over all channels and peek at the `AcquisitionState` property to figure this out or count `AcqOn/OnDataFinished` pairs yourself.) If another channel still requires data, do nothing. Otherwise, use the communication class to instruct the firmware to stop delivering data.
6. Call `NoMoreData` at any time if there is an error or you don't get any more data from the module.

General implementation approach for drivers using method download:

1. During preflight, collect information on which channels receive an `AcqOn`, and at which script time the latest `AcqOff` has been scripted. Use this time (or the time of the final command in the script) as the run time for the method.
2. Download the method in `OnTransferPreflightToRun` as usual.
3. Ignore all `AcqOn` commands. Data delivery should start automatically due to the execution of the downloaded method (note that method execution is started either as part of the `Sampler.Inject` command handling or on `Broadcast.RetentionZeroCrossed`). However, be prepared that some data may already be delivered to the driver prior to the `AcqOn` command being seen. You may not call `UpdateData` prior to the `AcqOn` command, so you might need to buffer the data somewhat.

4. Call `UpdateData` repeatedly as soon as the communication class has incoming data available. Make sure that you call the function on the correct `IChannel` device in case the driver supports more than one channel.)
5. Ignore all `AcqOff` commands.
6. Ignore all `OnDataFinished` events (most of the time, these won't get fired anyway as you call `NoMoreData` first). The firmware will stop data delivery anyway based on the downloaded method's run time or other method settings.
7. Call `NoMoreData` for each channel once you don't get any more data from the module (and have not yet received an `OnDataFinished` call), or at any time if there is an error.

`AcqOn` may result from scripts used for Instrument Methods (i.e., acquisition during Injections) as well as from scripts triggered by the user clicking on "Monitor Baseline" explicitly. For the latter, the script will not specify a time at which acquisition should stop (there are no `AcqOff` commands in that script.) If you can't do that (e.g., because your module delivers data only as part of an injection run), add additional preflighting to reject the "Monitor Baseline"-`AcqOn` for your channels. Alternatively, program an infinite (or very long) run on the module.

CM7 expects that drivers store the data as the module has delivered it. If the firmware allows specifying at which data collection rate (DCR) data should be measured/delivered (note that for some detector types the DCR has a significant effect on signal-to-noise-ratio or detector performance), provide a Chromeleon property ("Data_Collection_Rate") that allows the user to define this parameter. If your module always delivers data at a high data rate, and you think that it makes sense for users to store data at a lower rate, provide a Chromeleon property for this, too, and implement the downsampling in your driver.

If the DCR setting affects all channels alike, provide the property at the main device, if you can specify it on a per-channel basis; provide a property for each channel device individually.

CM7 distinguishes between signals that are used for chromatography purposes (i.e., for determining a baseline, finding peaks, and integrating the peak area) and signals of diagnostic/monitoring nature (e.g., pump pressure, column compartment temperature). Set `IChannel.NeedsIntegration` to true for the first type of signals, leave it at the default (false) for the latter type of signals.

`IChannel.UpdateData (Ex)` interfaces

- Data flow

1. Module → RS232/USB/TCP/IP/GPIB-Communication → Driver
2. Driver → `IChannel.UpdateData/UpdateDataEx` → `CmDDKHost.exe`
3. `CmDDKHost.exe` → Windows Named Pipes → RTK
4. RTK → Windows Named Pipes → ICT
5. ICT → Data Vault Service → Data Vault/File System
6. ICT → Windows Communication Foundation → Client's Online Signal Plot

[CM6]: The file format that we use for CM6 raw data storage stores time and value for the first data point as an absolute value, all subsequent data points are stored using deltas to the previous point. Each time delta is an integer, in 1/100 second units. Thus, if all data points are stored with a delta of 1, the signal looks like a 100 Hz signal. If all data points are stored with a delta of 2, the signal looks like a 50 Hz signal.

What delta is actually used for storage is something the user can define via setting of the channel device's `Step` property. If `Step` = 0.01 [s], we store using `delta` = 1, if `Step` = 0.02 [s], we store using `delta` = 2 etc., if `Step` = Auto, we dynamically change `delta` - where the curvature of the signal is high, `delta` is small and vice versa. This allows reducing disk consumption while still maintaining curve shape.

[CM7]: The raw data storage format has been changed in CM7. The `Step` property is no longer available for channel devices.

There are still some consequences for DDK driver which are designed to run in CM6 and CM7:

- If you use the `UpdateDataEx(double, double)` interface in CM6, data is not passed through 1:1. Instead the DDK will first interpolate the data to 100 Hz and default to Step = Auto.
- CM6 is not able to handle data rates above 100 Hz because of its internal data formats. If the detector module provides data rates above 100 Hz you might need to restrict the DCR range in the driver when running in CM6.

In CM7 we recommend using the `UpdateDataEx(DataPointInt64[])` interface. Data is passed through 1:1 – CM7 uses a (int64, int64) raw data storage format in that case. `DataPointInt64` consists of two int64 values for timestamp and data value. Make sure that you set the signal scaling factor as well as the time scaling correctly.

If you use the `UpdateDataEx(DataPointEx[])` interface, data is passed through 1:1. CM7 uses a (double, double) raw data storage format in that case (somewhat disk-wasting compared to the int64 format). `DataPointEx` consists of two double values for retention time and data value. We don't recommend this when using higher data rates.

You can still use the `UpdateData(int, int[])` interface with the following restrictions:

- The detector signal resolution should not exceed 32 bit.
- The data collection rate does not change during data acquisition.

Make sure that you set the signal scaling factor as well as the time scaling correctly before starting the acquisition.

Signal scaling

All DDK 2D Channels implement the standard property "SignalFactor". By default this property is not visible in Chromeleon release versions.

The `IChannel.SignalFactorProperty` interface provides access to this property. Set the value of this property to the signal factor which is applied to convert the data delivered by the driver into the "real" value.

Examples for signal scaling:

- Detector sends equidistant data, as integer, with 'Data' values between 0 and 1.0E6, representing floating point data 0.000 - 1000.000 mV, i.e. 3 digits precision
Choose 0.001 as signal factor. Write the data values coming from the detector 1:1 into the `UpdateData` function.
- Detector sends equidistant 24 bit integer data (e.g. 'counts'), to be displayed 'as is'.
Choose 1.000 as signal factor
- Detector sends 36 bit integer data
Similar to example 2, but use the `UpdateDataEx(DataPointInt64[] data)` function and create appropriate timestamps even if the detector sends equidistant data.

When creating the channel the signal factor is initialized to 1.000.

Time scaling

All DDK 2D Channels implement the standard properties "TimeStepFactor" and "TimeStepDivisor". By default these properties are not visible in Chromeleon release versions.

The `IChannel.TimeStepFactorProperty` and `IChannel.TimeStepDivisorProperty` interfaces give access to them. All timestamp values provided to any of the `UpdateData` functions are scaled in the following manner:

$\text{RetentionTime} = 10\text{ms} * \text{timestamp} * \text{TimeStepFactor} / \text{TimeStepDivisor}$

When creating the channel both values are initialized to 1.

If your channel records data with a fixed rate you should set the value of the `TimeStepFactor` and `TimeStepDivisor` properties as shown in the following examples. Then count the values received from the firmware and set the counter as timestamp.

Data Rate	TimeStepFactor	TimeStepDivisor
1 Hz	100	1
60 Hz	10	6
100 Hz	1	1
300 Hz [CM7 only]	1	3

TimeStepFactor and TimeStepDivisor must be set to 1 (default value) if the `UpdateDataEx(DataPointEx[])` interface is used. The Retention property of the `DataPointEx` parameter contains the retention time directly. The same applies to the corresponding `Int64` interface.

17.2 3D Signal Processing

3D data can be delivered to the DDK in two different ways:

17.2.1 IPDACHannel

This interface is intended for PDA (photo diode array) detectors that use an array of photo diodes to measure light intensity at a large number of wavelengths in parallel (an intensity spectrum). Typically, UV transmission is calculated by either the firmware or the driver by dividing two intensity spectra recorded at acquisition start and at some time into the run. UV absorption is $-\log(\text{transmission})$.

The resulting 3D data set is UV absorption (shown in mAU in client) as a function of time and wavelength.

Write Mode

In CM7 the `IPDACHannel` interface allows storing photo diode array data which will be always evaluated and displayed as absorption spectra. Therefore the channel should be created using a `WriteMode` of `PDAWriteMode.AbsorbanceDirect`. For backward compatibility the mode `PDAWriteMode.Raw` is supported, too. However, the raw data file of the channel will also contain absorbance values in this case. The conversion is done in Chromeleon Real Time Kernel process already.

Signal scaling

Spectra are expected as an array of integers.

- If the driver submits intensity counts (`PDAWriteMode.Raw`), these need to be scaled with a suitable factor so that the highest conceivable count value can still be represented as an int. The factor will cancel out when absorbance values are calculated upon reading the data.
- If the driver submits absorbance values (`PDAWriteMode.AbsorbanceDirect`), these need to be scaled with a fixed factor. To represent 1 AU, you need to submit a value of `IPDACHannel.OneAU`. This is a constant value currently implemented as $1024 * 1024 * 16 - 1$.

Wavelength scaling

It is also mandatory that the spectra contain data points at equidistant wavelength intervals. If the firmware delivers the spectra in some other representation, a driver might need to use interpolation in the wavelength direction to make the data points equidistant.

Chromeleon needs to know the Minimum and Maximum Wavelength and the Bunch Width (the wavelength interval between two consecutive spectra points) prior to `AcqOn`. The `IPDACHannel` implementation creates and manages the three properties `MinWavelength`, `MaxWavelength` and `BunchWidth` for you, you only need to provide the valid ranges for these properties.

Be aware that the user might request a smaller 3D data set to be acquired, e.g. by specifying

```
3DFIELD.MinWavelength = 190
3DFIELD.MaxWavelength = 350
3DFIELD.BunchWidth = 4
```

All spectra placed into `UpdateData()` must have the same number of data points, as calculated from $(\text{MaxWvl} - \text{MinWvl}) / \text{BunchWidth} + 1$, in this case 41 data points.

In case the detector firmware also supports these range and interval parameters the correct number of data points can be requested from the module already. Only signal data scaling is necessary to deliver the spectrum points with correct representation as mentioned above.

Often the detector modules can only deliver the full spectrum (e.g. from 190 ... 800 nm) with a fixed wavelength interval (e.g. 1.2 nm) or support a few discrete range and interval values only. Then the driver needs to clip, bunch and scale accordingly. Spectrum interpolation will be necessary, if:

- The firmware delivers the spectra with non-equidistant wavelength intervals.
- The firmware delivers the spectra with one fixed equidistant interval. But the BunchWidth values available in Chromeleon should not be limited to a multiple of this value.

Time scaling

Channels created with `IDDK.CreatePDChannel` or `IDDK.CreatePDChannelOnInstrument` implement the standard properties "TimeStepFactor" and "TimeStepDivisor" with the same behavior as described for 2D channels.

The timestamp parameter of the `UpdateData()` function is converted into a retention time:
 $\text{RetentionTime} = 10\text{ms} * \text{timestamp} * \text{TimeStepFactor} / \text{TimeStepDivisor}$
When creating the channel both values are initialized to 1.

Data Flow

1. Module → RS232/USB/TCP/IP/GPIB-Communication → Driver
2. Driver interpolates all spectra to constant wavelength interval representation
3. Driver → `UpdateData` → `CmDDKHost.exe`
4. `CmDDKHost.exe` → Windows Named Pipes → RTK
5. RTK → Windows Named Pipes → ICT
6. ICT → Data Vault Service → Data Vault/File System

3D Data Acquisition license

The DDK will refuse `AcqOn` for `IPDChannel` devices if the instrument controller lacks the 3D Data Acquisition license. A corresponding error message will be issued during ready check and at execution time.

Limitations

- Currently, there is no online plot available in the CM7 client for 3D data. This will be available in a later Chromeleon 7.2 service release.
- `IPDChannel` can store absorption only.

17.2.2 ISpectrumWriter

This general interface can be used to store single spectra (e.g. fluorescence spectra) rather than continuous 3D absorption spectra. It is also able to store other types of continuous 3D spectra that cannot be stored by the `IPDChannel` interface.

To create a spectrum writer, call `IDevice.CreateSpectrumWriter()`, typically on your main device. The device also needs to provide suitable commands (e.g. `TakeScan` or even `AcqOn/AcqOff`) and related properties that allows the user to control when and how individual spectra should be recorded.

The general workflow for this object is:

- Create the `ISpectrumWriter` object and set `SpectralFieldName` and `DetectionPrinciple`. You may use more than one `ISpectrumWriter` object for different types of spectra.
- Use `UpdateSerialNo` to provide the serial number of your module (if available). This meta information will be used by all spectra that will subsequently be stored by the spectrum writer. Users can later report the value for documentation purposes.
- Forward any instrument method properties and commands that determine spectra acquisition to your module in the usual way.
- As you receive a single spectrum (or "scan") from the instrument,
 - set all `ISpectrumWriter` properties that describe the spectrum as appropriate.
 - set the array of `DataPoints`. Note that the data points need to be equidistant in the wavelength direction; and that the wavelength range is determined by `WavelengthMinimum` and `WavelengthMaximum`.
 - Save the spectrum.

Contrary to the `IPDAChannel` interface, the `ISpectrumWriter` interface allows for greater flexibility. For instance, you can use it to store a few individual UV absorptions scans from a scanning VWD to a spectrum library associated with the Injection during which the scans were produced; each scan might have a different size or resolution, and all the meta data can be different for each individual scan.

You can also use the `ISpectrumWriter` interface to store non-PDA 3D data, such as a series of 3D fluorescence spectra generated at a high rate. These should probably be visualized as a 3D plot rather than as n individual scans. To request this type of behavior, set the `ISpectrumWriter` into continuous mode using `IsContinuousAcq`. The workflow is:

- Ensure that the driver requests a 3D Data Acquisition license by calling `IDevice::RegisterLicenseFeature`. during `IDriver.Init`. (Without requesting the license, you may risk run time license failure on the first attempt to store in continuous mode if users forget to manually assign the 3D Data Acquisition license to this ICT in the Administration Console.)
- Set all meta data as stated above and set `IsContinuousAcq` to true.
- As spectra are received, store them as stated above. Note that all spectra need to cover the same wavelength range, otherwise, 3D visualization might fail.
- Once the final spectrum has been received, call `NoMoreData()` to allow the framework to perform any necessary cleanup.

17.3 General Questions

- **Q:** When can I call `UpdateData()` / `UpdateDataEx()` ?
A: Call `UpdateData()` / `UpdateDataEx()` only AFTER you have received the 'AcqOn' callback function. Data points being sent before `AcqOn` are discarded and are lost. If necessary, you have to buffer your data until the `AcqOn` callback call has arrived in your driver (this can happen, for example, with detectors that are controlled with a download concept and where the acquisition is not started by Chromeleon but with a hardware signal).
- **Q:** We have (at most) 8 detectors, which do not all acquire useful data all the time (this depends on the state of the instrument). Is it possible to define reduced `AcqOn/AcqOff` intervals for selected detectors? How is this data handled by Cm – is it still synchronized on

the same time base?

A: In CM methods it is possible to define individual acquisition intervals for single channels. It is also not necessary that an acquisition interval matches the program run time (example: After injection there can be a pump ramp program that lasts 20 minutes. It is possible to acquire data between five and 15 minutes for channel one and from 0 to 10 minutes for channel 2.)

If your detector does not allow individual run times for single channels, a quick and simple approach would be to reject such methods by the ready check. A more sophisticated approach would be to let the driver acquire all data and to filter them in the driver according to the specified acquisition intervals.

18 Smart Startup/Shutdown Support

- Factory functions for special interfaces (on `IChannel`):
 - `void CreateNoiseDriftCalculator (double defaultNoiseLimit, double defaultDriftLimit, double noiseMinimum, double noiseMaximum, long noiseDigits, double driftMinimum, double driftMaximum, long driftDigits):`
This is needed for drivers supporting SmartStartup. It instructs the DDK to add properties and commands to the symbol table that support the semantics that the SmartStartup wizard expects.
 - `void CreatePumpRippleCalculator (double thresholdSignal, double defaultRippleLimit, double rippleMin, double rippleMax, long rippleDigits):`
This is needed for pump drivers with a column pressure channel that support SmartStartup. It instructs the DDK to add properties and commands to the symbol table that support the semantics that the SmartStartup wizard expects.

19 Localization

- Any UI that you provide (e.g. for the configuration plug-in) needs to be fully localizable. Move all dialog templates, UI validation error message strings, etc. to **resources**.
- Do **not** localize anything that ends up in the symbol table or in instrument methods (names of symbols, properties, commands, parameters, and string property values). The reason for this is to allow users with different locale settings to be able to exchange instrument methods. Make sure that all these strings are **not** placed in resources, but are hardcoded. This will prevent accidental translation. The only exception to this rule is the unit string of a numeric property.
- When formatting/parsing numbers for/from UI text boxes, use **CurrentCulture**, not **CurrentUICulture** (the latter is used to determine what languages the resources should be taken from). Ensure that all numbers you format for persistent storage (e.g. as part of the configuration XML or in PGM/METH files) or for transmission between client and server are using **InvariantCulture** (and not implicitly EN-us or EN-gb).
- Also, when formatting/parsing numbers for/from text based device communication, use **InvariantCulture**.
- Putting translated resources into satellite assemblies (e.g. in a JP-jp subdirectory) is OK; this will not invalidate the certificate, even when those translations are added after certification.

Configuration UI: Should be localizable, including error messages issued by validation. Make sure that numbers that user can read or enter are formatted according to the current locale. Use `CurrentCulture`, not `CurrentUICulture`.

Configuration Report: Use English! (required for support scenarios)

IME UI: Should be localizable, including error messages issued by validation. Make sure that numbers that user can read or enter are formatted according to the current locale. Use `CurrentCulture`, not `CurrentUICulture`.

Symbol names: Use English! Make sure the strings are hardcoded and do not come from resources (to prevent accidental translation).

Named values: Use English! Make sure the strings are hardcoded and do not come from resources (to prevent accidental translation).

Help texts for symbols: You may take them from resources to allow localization. However, remember that user's client locale is not known by the instrument controller, so it probably does not make sense to translate these help texts.

Audit messages: We recommend preparing the driver for localization of audit messages. By default, an English version of the audit messages should be always available.

We also recommend using a separate resource file only for the audit messages and ready check messages. A growing number of customers are requesting detailed lists of error and warning messages coming from the driver and such lists can be easily extracted from resources files. For UI implemented using Windows Forms, set the Localizable property of the form to True. Take strings from resources. Do not concatenate strings, but use `String.Format()` and `{n}` inserts instead.

PS: We expect that all UI works for both 96 DPI and 120 DPI screen output. No text clipping should occur and all dialog layouts should be as intended.

20 Online Help Support

We expect that all UI supports context sensitive help.

For internal developers, we expect that all help is integrated with the CHM help files that are shipped with Chromeleon.

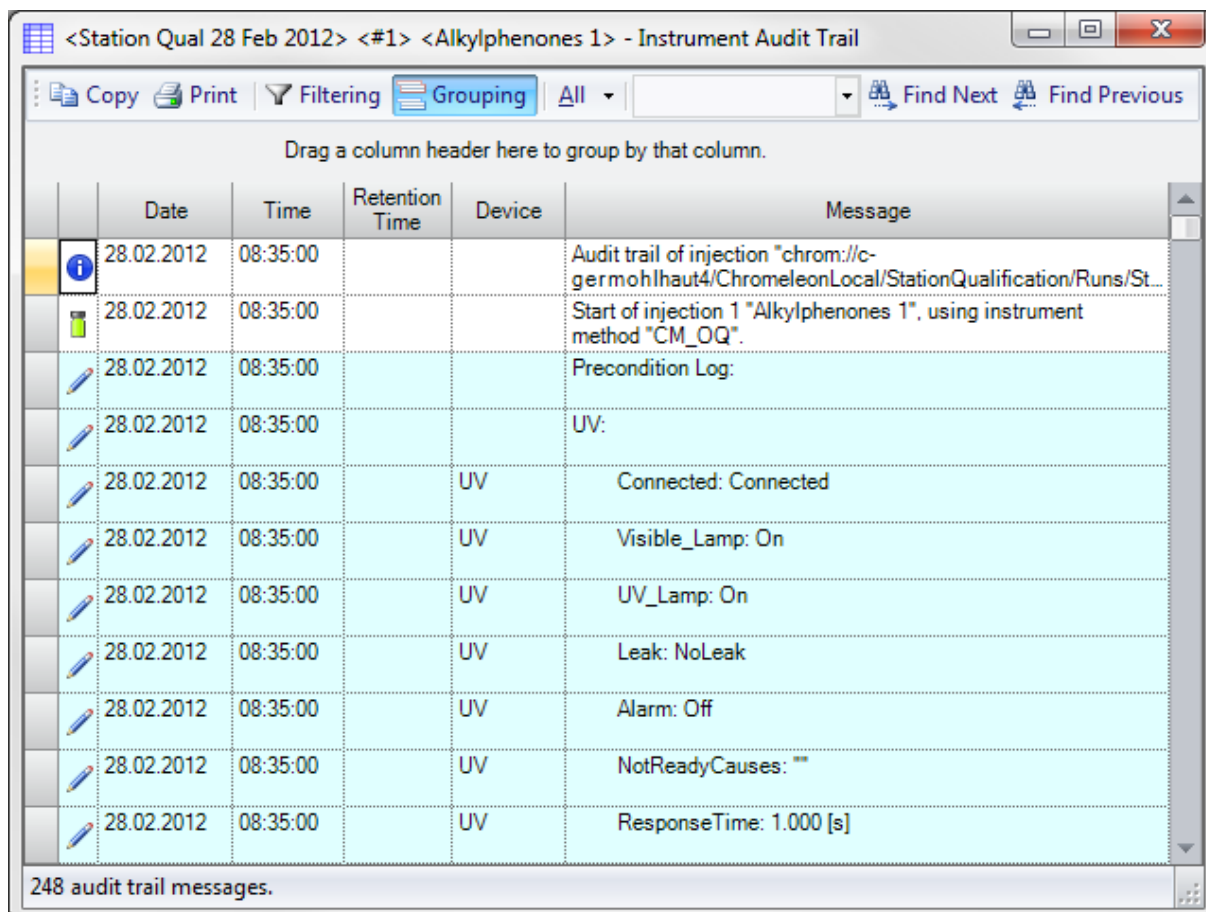
External developers may ship their own help file, and they must ensure that clicking the Help button or pressing F1 while their UI has focus opens up the correct help file at a suitable topic.

WinHelp (HLP) is not supported; the recommended help format is HTMLHelp (CHM).

21 Reporting Support

21.1 Precondition Log entries

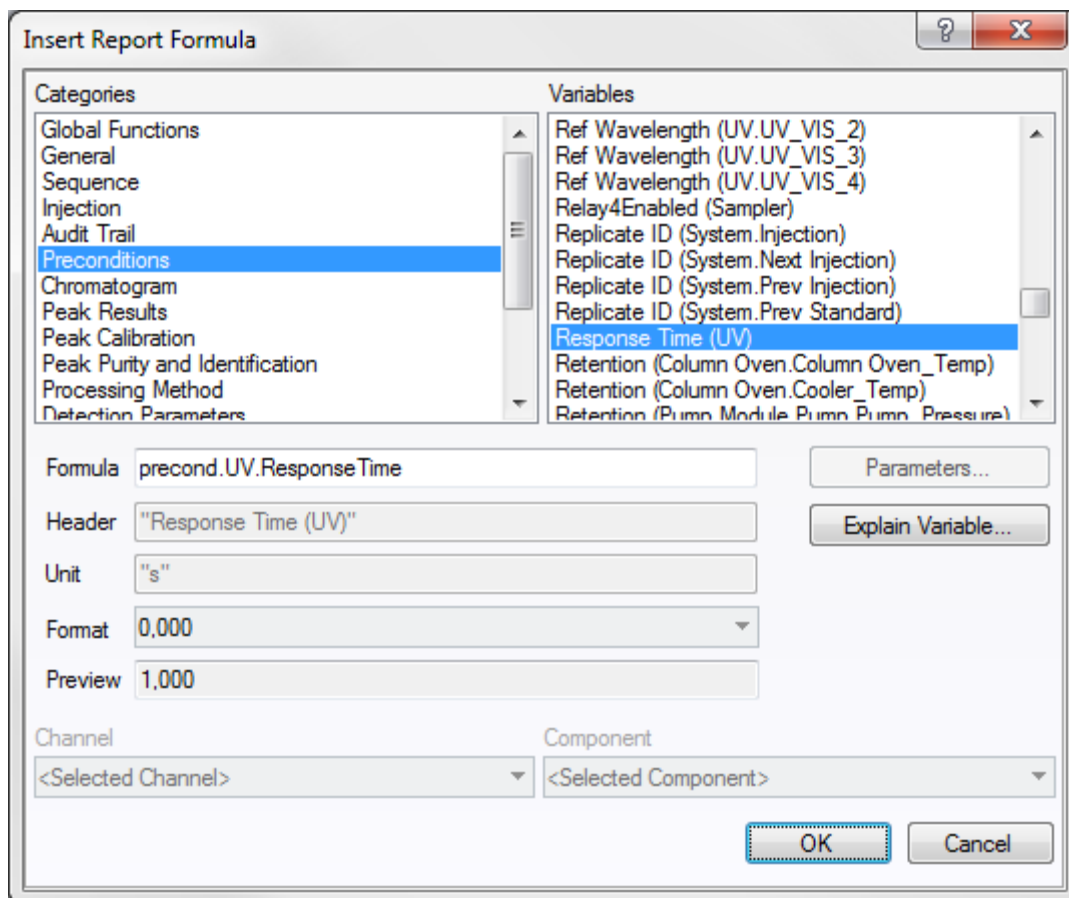
At the start of each Injection, the Instrument Controller automatically generates a Precondition Log (PL). The PL is stored as part of the Injection Audit Trail and provides a dump of all properties that the Instrument offers, along with their current values:



	Date	Time	Retention Time	Device	Message
	28.02.2012	08:35:00			Audit trail of injection "chrom://c-germohlhaut4/ChromeleonLocal/StationQualification/Runs/St...
	28.02.2012	08:35:00			Start of injection 1 "Alkylphenones 1", using instrument method "CM_OQ".
	28.02.2012	08:35:00			Precondition Log:
	28.02.2012	08:35:00			UV:
	28.02.2012	08:35:00		UV	Connected: Connected
	28.02.2012	08:35:00		UV	Visible_Lamp: On
	28.02.2012	08:35:00		UV	UV_Lamp: On
	28.02.2012	08:35:00		UV	Leak: NoLeak
	28.02.2012	08:35:00		UV	Alarm: Off
	28.02.2012	08:35:00		UV	NotReadyCauses: ""
	28.02.2012	08:35:00		UV	ResponseTime: 1.000 [s]

248 audit trail messages.

Users can access individual items of the PL via Report Variables when designing reports:

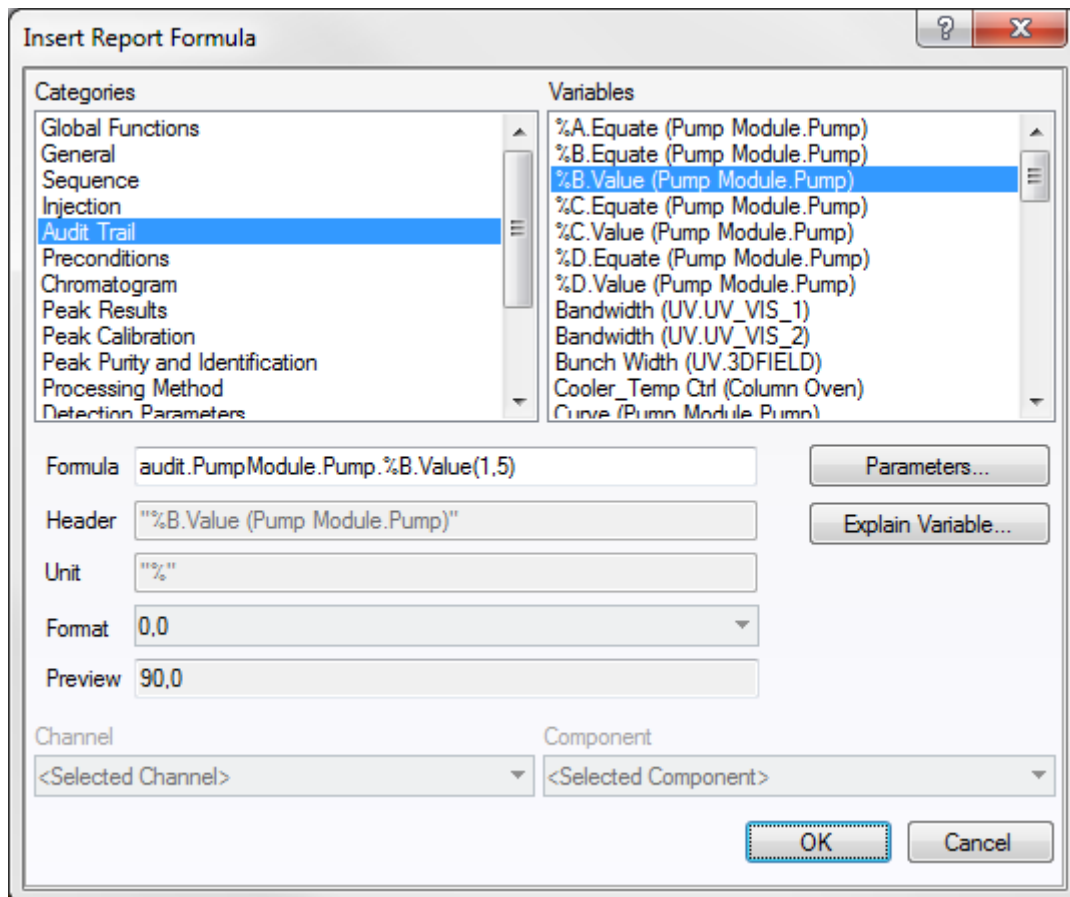


It is assumed that all drivers keep the values of all the properties that the drivers offer in sync with the state of the hardware. Only then can the PL provide a reliable snapshot of the hardware's state at the beginning of an Injection. To ensure that, do not change any injection related method parameters before the `IDevice.OnPreflightToRun` call, except for the Volume and Position setting updates of the `IInjectHandler` interface instance.

21.2 Audit Trail report variables

During an injection, the script execution engine writes all property assignments and commands to the Injection Audit Trail (IAT) before it dispatches the request to the corresponding driver. In other words, the IAT documents what the drivers **should** do (and not, what they did).

If an Injection has completed successfully (Status has changed to Finished), users can access individual items of the IAT via Report Variables when designing reports:



Note that this evaluates what the script engine has written to the IAT, and is capable of interpolation should a value be requested for a time where no property assignment is present in the script.

It is assumed that - if a driver cannot fulfill any of the requests made by the script engine - an abort message is generated by the driver at run time. This will set the Injection's Status to Interrupted, which means that Report Formula values in category Audit Trail may not be accurate.

21.3 Signal metadata

When 2D or 3D data is shown in Studio > Data Processing category, users can export the data to file or clipboard via the context menu of the respective plots.

For 2D signals, this looks like:

File Path	chrom://c-germohlhaut4/ChromeleonLocal/StationQualification/Runs/Station Qual 28 Feb 2012.seq/515.smp/UV_VIS_1.channel/UV_VIS_1.chm	
Channel	UV_VIS_1	
Injection Information:		
Data Vault	ChromeleonLocal	
Injection	Alkylphenones 1	
Injection Number	1	
Position	RA1	

Comment		
Processing Method	CM_OQ	
Instrument Method	CM_OQ	
Type	Calibration Standard	
Status	Finished	
Injection Date	28.02.2012	
Injection Time	08:35:15	
Injection Volume (µl)	2	
Dilution Factor	1	
Weight	1	
Chromatogram Data Information:		
Time Min. (min)	0	
Time Max. (min)	0,45	
Data Points	2701	
Detector	UV	
Generating Data System	Chromeleon 7.2.0.2250	
Exporting Data System	Chromeleon 7.2.0.2250	
Operator	Instrument Controller	
Signal Quantity	Absorbance	
Signal Unit	mAU	
Signal Min.	-0,243922	
Signal Max.	44,269648	
Channel	UV_VIS_1	
Driver Name	DAD3000.dll	
Channel Type	Evaluation	
Min. Step (s)	0,01	
Max. Step (s)	0,01	
Average Step (s)	0,01	
Signal Parameter Information:		
Signal Info	WVL:244 nm	
Chromatogram Data:		
Time (min)	Step (s)	Value (mAU)
0	n.a.	0
0,000167	0,01	0,0001
...
0,45	0,01	0,301066

Most of this meta information is generated automatically.
 Channel Type (Evaluation/Diagnostic) is based on the value of `IChannel.NeedsIntegration`.
 Signal Info (WVL:244 nm) is based on the initial value of the standard property Wavelength, if the channel device has one.

For 3D Signals, this looks like:

File Path	chrom://c-germohlhaut4/ChromeleonLocal/StationQualification/Runs/Station Qual 28 Feb 2012.seq/515.smp/3DFIELD.field				
Spectral Field	3DFIELD				
Injection Information:					
Data Vault	ChromeleonLocal				
Injection	Alkylphenones 1				
Injection Number	1				
Position	RA1				
Comment					
Processing Method	CM_OQ				
Instrument Method	CM_OQ				
Type	Calibration Standard				
Status	Finished				
Injection Date	28.02.2012				
Injection Time	08:35:15				
Injection Volume (µl)	2				
Dilution Factor	1				
Weight	1				
Raw Data Information:					
Time Min. (min)	0,000167				
Time Max. (min)	0,45				
Scan Min. (nm)	190				
Scan Max. (nm)	350				
Signal Min. (mAU)	-1,209498				
Signal Max. (mAU)	85,432296				
Spectra	2700				
Detector	UV				
Detector Type	UV				
Generating Data System	Chromeleon 7.2 Build 2250 (Alpha) (191301)				
Exporting Data System	Chromeleon 7.2.0.2250				
Spectral Field	3DFIELD				
Driver Name	DAD3000.dll				
Raw Data:					
Time (min)	Integr.Time (s)	190	191	...	350
0,000167	0,01	-0,019789	-0,008345	...	-0,002444
0,000333	0,01	-0,095725	-0,047505	...	-0,008285
...
0,45	0,01	13,178051	9,82982		0,187397

22 Data Audit Trail Support

For objects stored in a Data Vault ([CM6] Data Source), Chromeleon supports versioning. Every time an object has been modified and is saved, a new version is stored. At any later time, users can use the Data Audit Trail (available from the context menu of each of the objects) to look at the version history. Previous versions can be retrieved if desired.

Users can also compare two arbitrary versions for differences to see what changes have been made. The difference report is created on the fly upon request.

For instrument methods that consist of script only, difference report generation is automatically handled by the framework. Instrument methods that also use 3rd party data blobs need to provide diffable data as well, according to the schema defined in CmFormattedData.xsd.

23 Simulation Mode

Drivers are not strictly required to implement Simulation Mode. However, having Simulation Mode available has a lot of benefits:

- Developers can debug driver logic and data processing easily and are not hindered by disconnects by hardware/firmware due to communication timeouts.
- ePanel designers can load and configure the driver, and therefore access the symbol table created by the driver) even when they don't have hardware available.
- Testers can load and configure the driver, and therefore run the instrument method wizard/editor even when they don't have hardware available.
- Technical writers can also see all functionality even when they don't have hardware available.

Thus, Simulation Mode will allow reducing time-to-market as some work can be parallelized.

If a driver offers simulation mode, it should:

- Allow to complete the initial configuration dialog/wizard with suitable defaults. Ideally, the UI allows defining which module options should be present in the simulation.
- Allow the driver to load with such a configuration. The driver should allow Connect and Disconnect commands to execute fine.
- All properties should be initialized to suitable values which are inside the valid ranges.
- Methods created by the Instrument Method Wizard should pass Check Method.
- Setting any property to any valid value should work. After setting, the property should be updated to the new value immediately.
- Executing any command should work. For complex commands (e.g. UV.Autozero), simulating side effects (such as being NotReady for a while, or resetting signal property values to 0) is nice to have.
- Detector drivers should either reject AcqOn commands, or implement means of generating simulated data. This can be as simply as emitting a constant value, a sine signal, simulated peaks, or even replaying a simulation data file that has been recorded with a real detector.

We recommend implementing simulation mode by defining an interface that abstracts the communication with the hardware, and by implementing two classes that implement this interface. One class can really communicate with the hardware, and the other class can simulate hardware responses. Turning on Simulation Mode is then simply using the second implementation, and all driver code using the abstract interface can remain unchanged and ignorant of the simulation mode.

Simulation Mode can be as smart as you want and your resources allow.

24 Update of a Released Driver

It is a quite common scenario that a released driver is to be updated after a while; either just to fix bugs or to extend its functionality, e.g. to support a new hardware version or new hardware options.

Especially in the latter case changes to the driver configuration XML file and the device/ property / command set are necessary quite often.

However, when considering such an update, one has to take into account that customers are already using the current version of a driver. This means that:

- The customer uses a server configuration in which an older version of the driver configuration is stored.
- The user uses instrument methods, which were created with an older version of the driver and its symbol table.

Chromeleon standards require that driver updates are always set up in a manner that the customer can proceed working with his driver after an upgrade without manual actions, which means:

- The customer can start the server with an older driver configuration. The driver detects and interprets the old configuration xml correctly and can work with it. The driver behavior should be compatible to the origin driver functionality.
- The configuration plug-in can re-configure a driver instance which was added to the server configuration with an older version of the driver. It is acceptable that features that are introduced with the new driver version are not available in such cases until one creates a new instance of the driver with the new configuration. But the original driver options should be available and configurable in any case.
- An instrument method being created with the original driver version should work seamlessly with the new driver version (and original driver configuration). Especially take care that the symbols being present in the original driver are not renamed or removed and keep their original meaning. The same applies to named values of numeric properties.

In seldom cases (major design flaws or bug fixes) I may be reasonable to break these rules. However, it should be clear that this is not what a customer would expect. In such a case, these breaking changes must be clearly mentioned in the driver update release notes and clarified with Thermo Fisher Scientific DDK support before the certification process takes place. In cases in which the extended functionality breaks the original functionality significantly, one should consider a 'backward compatibility' mode or even a new driver that can be installed in parallel to the original one. Then the customer can decide individually what to use.

25 Debugging and Troubleshooting

25.1 Debugging with Visual Studio

Debugging a DDK driver:

Debugging a DDK driver instance is not that comfortable because the driver process is running under a local service account. If a problem occurs during special actions of the driver but not during startup, the easiest way is to attach your VS debugger to the corresponding CmDDKHost process before triggering these actions (usually this requires admin privileges). If your instrument controller configuration contains more than one DDK driver instance use Sysinternals' ProcessExplorer (or other free tools that list the CmDDKHost processes with the corresponding command lines). The command line indicates which driver is encapsulated in this process. Attach to the correct CmDDKHost with the VS debugger by comparing the PID.

Process Name	PID	Private Bytes	Company Name
CmDriver.exe	5028	0.91 Chromeleon Real Time Kernel	Thermo Fisher Scientific Inc.
CmDDKHost.exe	6028	0.01 Chromeleon Device Driver Host	Thermo Fisher Scientific Inc.
CmDDKHost.exe	7880	0.07 Chromeleon Device Driver Host	Thermo Fisher Scientific Inc.
CmDDKHost.exe			Thermo Fisher Scientific Inc.
CmDDKHost.exe			Thermo Fisher Scientific Inc.
CmDDKHost.exe			Thermo Fisher Scientific Inc.
CmDDKHost.exe			Thermo Fisher Scientific Inc.
ServiceHost.exe			Microsoft Corporation
lsass.exe			Microsoft Corporation

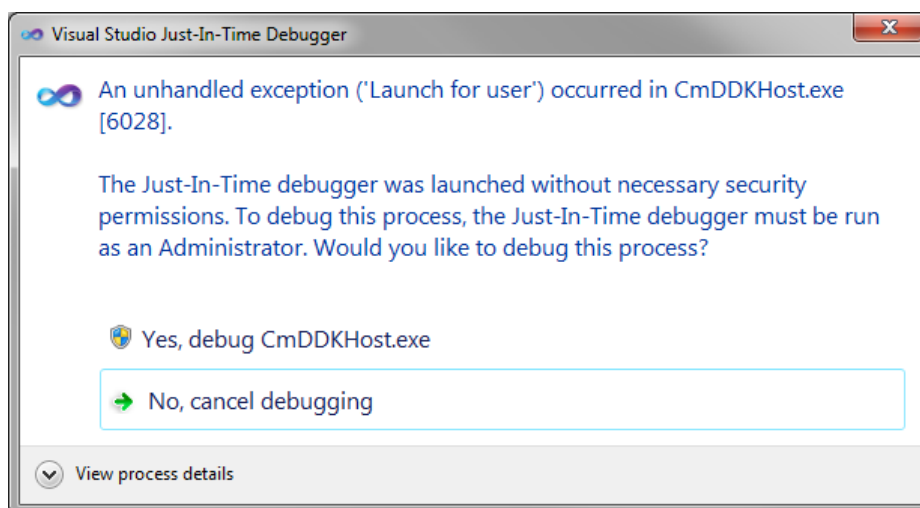
It can happen that a problem occurs during the startup phase (Driver constructor, Init()). In these cases, the driver startup must be halted as early as possible so that one has time to attach to the driver process before the error occurs.

With older operating systems like Windows XP, the following procedure will work:

- Allow the Chromeleon 7 Realtime Kernel Service to interact with the desktop (otherwise CmDDKHost.exe can't show dialog boxes for assertions)
- Start Instrument Controller using Chromeleon Services Manager
- Start Instrument Configuration Manager
- Add an instrument
- Add the driver
- Use Assertions (`Debug.Fail()`) to break into the debugger or attach the debugger to CmDDKHost.exe after loading the driver

It turned out that in newer operating systems like Vista and Windows 7 the `Debug.Fail()` construct does not longer work if the process runs under a kernel service account, even if one has allowed desktop interaction. The process seems to stop at the 'Fail' position but no message box is popping up.

In Win7 and Vista use `System.Diagnostics.Debugger.Launch()`; instead. It will trigger such a pop up message. Clicking on 'Yes, debug CmDDKHost' will get to the debugger at the line where this code was added.



Debugging a DDK ICM plug-in:

- Use Assertions (`Debug.Fail()`) to break into the debugger or attach the debugger to InstrumentConfiguration.exe after adding the driver

Debugging a DDK IME plug-in:

- DDK IME plug-ins are running in the Chromeleon client (console) process Chromeleon.exe. To debug a plug-in, open its project settings, go to the 'Debug' tab, check option 'Start external program' with the path to Chromeleon.exe as external program. Alternatively you can start debugging by attaching to the Chromeleon.exe process (in Visual Studio, select

<Debug>-<Attach to process>). The debugger will stop at any breakpoint you have set in the source code. Note, that plug-ins aren't loaded until you start the IMW or open the IME with an instrument that contains a module with a DriverId corresponding to your IME plug-in.

25.2 Typical Problems with a DDK Driver

- The driver doesn't show up in Instrument Configuration
 - Start Sysinternals' DebugView and restart Instrument Configuration, watch enumerator's output
 - Check that your driver files are present in a sub folder of <Chromeleon>\bin\DDK\V1\Drivers. Check that an appropriate DriverCert.xml file is also present!
 - Verify that within the bin\DDK tree, no two assemblies having the same name are present.
 - Verify that within each of the bin\DDK\V1 and bin\DDK\V2 trees, no two DriverCert.xml files referring to the same DriverID are present; behavior will be undetermined.
- The driver doesn't load after restart of Instrument Controller
 - Start DebugView and restart the Instrument Controller, watch output from CmDriver.exe (RTK) and CmDDKHost.exe (DDK)
- The driver is enumerated properly but the assembly can't be loaded
 - Check whether the DDK Developer license has been assigned to your instrument controller. See 26.4 for details.
 - Start FusLogVw.exe, retry and check binding results (your driver could be using the wrong references)
- I built a new driver, and handed it to the test department, but it doesn't load on their PC
 - You cannot generate a proper DriverCert.xml on your local PC, this is only done by Thermo Fisher Scientific
 - Ask the tester to use the DDK Developer license. See 26.4 for details.
- Start-up times for InstrumentConfiguration and CmDriver.exe are very long when running these processes in the developer. Can I avoid this?
 - Both processes enumerate and validate all available plug-ins. You can reduce the time needed for this by simply deleting those sub-directories of bin\DDK\V1\Drivers that contain drivers that you don't need for your current task. Run a repair installation to get the drivers back.
- I have just added ISendReceive, but it doesn't work. What is going on?
 - Try removing and re-adding the driver. The DDK caches information on whether a driver uses ISendReceive, and this might not have picked up the change.

25.3 Debugging tools

ProcessExplorer

- Replaces Windows' Task Manager
- Use this to explore processes, determine process details (such as Process ID, Command Line, User Name) and kill any hanging processes
- <http://www.sysinternals.com/>

Reflector or ILSpy

- To have a look into the binaries, references etc.
- <http://www.reflector.net>
- <http://wiki.sharpdevelop.net/ILSpy.ashx>

DebugView

- Use this to watch the DDK driver enumeration and loading (even in release builds)
- <http://www.sysinternals.com/>

Assembly Binding Log Viewer

- Use this to find load errors due to missing assemblies, binding redirects, etc.
- [%PROGRAMFILES%\Microsoft Visual Studio 8\SDK\v2.0\Bin\FusLogVW.exe](#)
- [%PROGRAMFILES%\Microsoft SDKs\Windows\v6.0A\Bin\x64\FUSLOGVW.exe](#)

26 Deployment

26.1 Build

Developing drivers using the CM 7.1 DDK requires Visual Studio 2008 or 2010 (2005 may also work, but are not tested by us, we don't recommend it). CM 7.2 DDK requires Visual Studio 2010. We suggest that a driver package consists of (at least) three distinct assemblies, each one providing an implementation of `IDriver`, `IConfigurationPlugin` and `IEditorPlugin`, respectively. See the example projects that come with the DDK for how to set up suitable projects. Either start with the example projects or use the templates to create new projects.

We recommend that you place your binaries into one directory

V1\manufacturer name\model name

for the driver and configuration plug-ins and into one directory

V2\manufacturer name\model name

for the IME plug-in.

Have a look at the drivers that ship with Chromeleon for naming conventions.

Note: In CM6, there is no V2 for the method plug-ins. The CM6 editor plug-in can be placed into the same folder as the driver and configuration files.

Caveat: Chromeleon will enumerate all assemblies in folders where a `DriverCert.xml` is present. When multiple assemblies for the same `DriverID` are found, it is undetermined which ones will be loaded. Thus, watch out for stale copies that might end up on your disk when you rename output folders and rebuild your projects. The best way to make a driver package invisible to the system is to rename the `DriverCert.xml` file to `_DriverCert.xml_`.

26.2 Source tree layout and projects

The DDK is divided into two parts. `DDK_V1` is for driver and instrument configuration plug-in development and the `DDK_V2` is for instrument method editor and SmartX plug-in development.

In general, drivers should only reference assemblies located in the driver output folder or the GAC. The same is true for method editor and SmartX plug-ins.

This can lead to code duplication, e.g. a driver creates a helper assembly for accessing XML or a 3rd party component for `Windows.Forms.Controls` is used. Code duplication is not desired since it increases the hard disk space requirements by Chromeleon and the setup file list handling becomes more complicated.

In order to address these issues and support code sharing a new folder for shared assemblies was introduced with the release of the DDK V1 version 1.20. Drivers, configuration-, method editor- and SmartX plug-ins can load assemblies from this location. The folder for code sharing is called "*Shared*" and is located directly in the DDK folder ("*C:\Program Files\Thermo\Chromeleon\bin\DDK*").

Assemblies that are provided by Thermo Fisher and can be reused by all drivers and plug-ins are located in “*Shared\ThermoFisher*”.

The following guidelines should be considered when implementing new drivers:

- Create a new project in “*DDK\V1_x\Drivers\<Manufacturer>\<DeviceName>\Driver*” with the name “*<DeviceName>Driver.csproj*”. This project should contain at least two C# code files; one for the driver and one for the device logic. Make sure that the project output is build to:

„Output\\$(ConfigurationName)\DDK\V1\Drivers\<Manufacturer>\<DeviceName>\“

- Build assemblies that get reused by the Chromeleon client to:

“Output\\$(ConfigurationName)\DDK\Shared”

Select a unique name for such binaries to ensure that no other drivers come with a shared binary having the same name.

- Create a new project in “*DDK\V1_x\Drivers\<Manufacturer>\<DeviceName>\Config*” with the name “*<DeviceName>Config.csproj*”. Use the same output path as defined below for the driver.
- Instrument method editor plug-ins should be located in “*DDK\V2\ Drivers\<Manufacturer>\<DeviceName>\<DeviceName>.EditorPlugIn*”. The project name should be “*<DeviceName>.EditorPlugIn.csproj*”. Make sure that the project output is built to:

„Output\\$(ConfigurationName)\DDK\V2\Drivers\<Manufacturer>\<DeviceName>“

by using the assembly name:

“<Manufacturer>.DDK.V2.<DeviceName>.EditorPlugIn.dll”

26.3 Signature

After all your assemblies have been built, DriverSignature.exe needs to be called as the final build step. (See example drivers for how to call the tool.) The tool will inspect all the assemblies in the specified directory and generate a DriverCert.xml file. This file provides loading information for plug-in enumeration, license information and prevents code tampering.

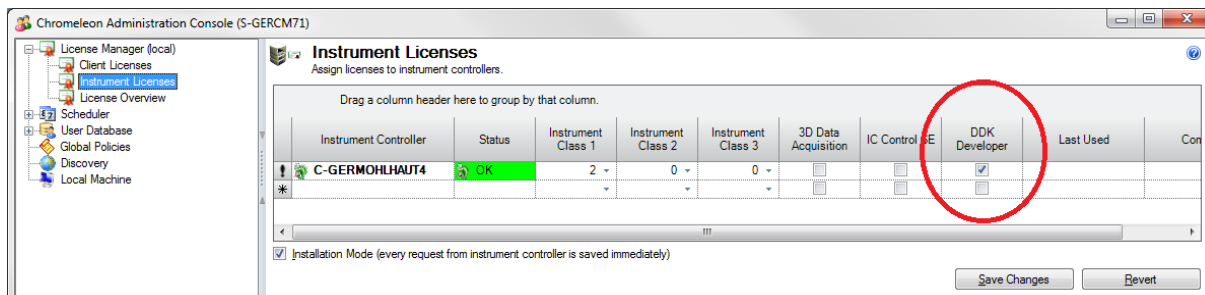
Caveat: Make sure that your driver package is built completely before calling the tool. Placing additional assemblies into the directory after the tool has run will invalidate the DriverCert.xml file. You must ensure that when you finally deploy your package to an end user's system the list of files in DriverCert.xml file matches the list of files found in the directory.

Your code may reference other assemblies if these are placed in the GAC. You may not load any other assemblies by navigating into parent folders or sibling folders of your package's folder.

26.4 Certification

Driver packages containing a DriverCert.xml file generated by you can only be used on Instrument Controllers that have a DDK Developer License.

You can use Administration Console > License Manager > Instrument Licenses to check this:



Caveat: Even with Installation Mode checked, this license bit is not requested or granted automatically. You need to manually assign it to your Instrument Controller.

When your driver package has passed our quality checking (see reference 3 for requirements), we provide you with a DriverCert.xml that has been digitally signed by Thermo Fisher Scientific. Such driver packages can be run on any Instrument Controller (irrespective of whether the DDK Developer License is present) provided the required Instrument Class X license is available.

26.5 Setup and Distribution

- If you supply your own setup, provide a way to install script-based (e.g. via a standard MSI installation package) and make all options accessible via command line switches. Document the switches in a suitable way. This allows system administrators to build their own setup scenarios that can run unattended without requiring user input.
- Make sure that any file system paths that are used during installation are aware of the specific Windows platform that is used as well as to any system/user locale settings. For instance, use %ProgramFiles% rather than "C:\Program Files". Use the `ShGetSpecialFolder()` API if needed.
- Problems with drivers not loading, not being listed in Add Devices, or being terminated immediately after startup typically indicate an issue with the signature/DriverCert.xml or unresolved assembly references. Use `DebugView.exe` (from Sysinternals) and `FusLogVw.exe` (from Microsoft) to track these down. Also look into the Windows Event logs.
- Favor so-called Side-By-Side (SxS, e.g. see <http://msdn.microsoft.com/en-us/library/ms973913.aspx>) COM activation over registry-based COM activation if your code uses COM servers that you ship with your driver package. This reduces the potential for interfering with other applications or other Chromeleon versions on the same PC.
- If you want to distribute your own panel/ePanel, it needs to be installed in a specific place to be available for users. Contact DDK support for details.
- If you want your driver to appear under different names/manufacturers, it is best to simply create a common base class with the implementation and derive several thin wrapper classes from the base, each class using their own set of DeviceID attributes and DriverInfo resources.
- File names as stored in DriverCert.xml are case sensitive. Make sure that you install files on the target system using the same case as used during driver signature.
- Do not install files in the Chromeleon installation directory, or in DDK\V1. All driver files must either be in `DDK\V1\<manufacturer>\<device name>\...` or placed in the GAC.
- If your communication layer needs Windows firewall exemptions, configure them as part of your setup process. Also make sure that these requirements are properly documented for users who use a different firewall product.
- We recommend targeting the .NET framework that ships with the Chromeleon product that the driver should be compatible with (see Chromeleon product documentation for details). If you

rely on a different .NET framework, your setup must ensure that the required framework is present, ideally, provide a suitable bootstrapper setup package.

- The setup distribution package should be able to detect whether and where Chromeleon is installed and whether it's the correct version before trying to install the drivers.

Here are some hints:

In Chromeleon 6, check for the 'Path' variable of the registry key HKLM\Software\Dionex\Chromeleon. If it exists, it contains the path of the Chromeleon installation. To determine the Chromeleon version check for the version of the file Messages.dll in the 'bin' subfolder of your Chromeleon installation; it is updated even with driver updates; so its version and build number clearly identifies the Chromeleon version.

In Chromeleon 7 check for the 'InstDir' variable of the registry key HKLM\Software\Dionex\Chromeleon\7. If it exists, it contains the path of the Chromeleon installation. The 'ProductVersion' variable in the same key identifies the Chromeleon version.

27 Miscellaneous

27.1 Do's and Don'ts

- Compare instrument configuration vs. persisted configuration XML during connect. See chapter 7.
- Detect problems as early as possible:
 - Pages in the IMW/IME should validate all input for range. Leaving the page should not be possible while an input is invalid.
 - Pages in the IMW/IME should check for all constraints related to input on that page. Leaving the page should not be possible while a constraint is violated.
 - Ideally, plug-ins in the IMW/IME should check for all constraints related to input across all pages. Be aware that users might need to be able to jump between pages to resolve the constraint violation.
 - During semantic check, identify inconsistencies in the script solely based on what is stated by the script. Ignore the current state of the system, the driver and/or the module.
 - During ready check, also take current state of the system, the driver and/or the module into account. (You could imagine this as a semantic check of a script that assigns all properties to their current value at the very top of the script).
 - If batch preflighting can identify an issue that will appear at injection x in sequence y, let the user know before she starts the queue and walks away. Keep track of consumables, waste levels, disk space, wear parts and associated limits if you want your driver to be really sophisticated.
- Support scripts such as
UV.Autozero
Wait UV.Ready
or
TCC.Temperature.Nominal = 20 [°C]
Wait TCC.Ready
via ImmediateNotReady. See 10.2.
- Handle dependent properties in a smart way. E.g.,
TCC.Temperature.Nominal = 20 [°C]
should implicitly set
TCC.TemperatureControl = On
If you do this, Update() the dependent property, then call LogValue() to explicitly

document the change in the audit trail.

- For properties that change slowly in response to a change in set point (e.g., a column compartment temperature, or a pump flow for a pump that limits the rate of changes to flow rate), provide a structure with `X.Nominal` and `X.Value`.
- During semantic check (and all subsequent preflights), check for constraints such as
`3DFIELD.MinWavelength <= 3DFIELD.MaxWavelength`
`Pressure.LowerLimit <= Pressure.UpperLimit`
`Temperature.LowerLimit <= Temperature.Nominal <= Temperature.UpperLimit`
 even if the script doesn't specify all these properties within one time step. Use the previously set value (or the current value) for any missing value.
- Ensure that any configuration file (*.CMIC) can be imported into ICM and the driver's configuration property dialog can be opened and looked at even without hardware present. This is needed for support scenarios where a CMIC is sent to us and we want to inspect the configuration, but don't have matching hardware available.
- While acquisition is on (or your module executes a downloaded method), explicitly disconnecting should be rejected. Add preflight checks in `OnPreflightSetProperty` for your main device's Connect property and in `OnPreflightCommand` for your main device's Disconnect command.
- While acquisition is on (or your module executes a downloaded method), manual commands (`runContext.IsManual && !runContext.IsSemanticCheck`) should be rejected. Add preflight checks in `OnPreflightSetProperty` and `OnPreflightCommand` if property changes/command executions interfere with acquisition or method execution. Otherwise, make sure that these operations work as intended.

27.2 Frequently Asked Questions

Q: We need to do some post-run calculations, both on the raw data and on analysis results (quantities, etc). Is there a (or what is the best) way to handle this?

A: This would be something that needs to be handled outside the DDK. Write a Chromeleon SDK application that performs this analysis offline, either triggered by the user or as part of a post-acquisition processing (not sure when this will be available/extensible in CM7)

Q: Is it possible to store a set of diagnostics values at the moment of injection (e.g. module temperature, ambient pressure, etc. – not time dependent, but just one value), and (if yes) how is this represented in the stored data set for a run?

A: If values at the moment of injection start are sufficient, the precondition log (see 21.1) will do the job automatically. Alternatively, you can call `IProperty.LogValue()` to explicitly write a property to the Injection Audit Trail and/or Daily Audit Trail at any time.

Q: We would like in the end to use Chromeleon also in our production/test environment, where we would use Chromeleon as a front-end to run designated sequences, etc. To test our instruments, we need to do specific additional data analysis, which requires access to the raw data by another (test) application. Is there a way to access the stored data without using the Chromeleon client?

A: Again, this can be implemented using a Chromeleon SDK application, which can access the stored data. Plus, if any additional drivers are needed for this use case, writing small DDK-based drivers should be easy. We use similar technology as part of automated burn-in and factory operation qualification at our production floor, too.

Q: What are the expected developments for CM7.2 and higher – are there important changes for driver implementation and/or development?

A: Our intention is that any extensions to the DDK for CM7.2 are fully backward compatible with the DDK that ships with CM7.1 SR1. If there are any breaking changes, release notes for the DDK will document them.

Q: What script validation checks are automatically performed by the DDK?

A: At execution time, the DDK automatically checks whether your main device is connected (property Connected has value Connected) and, for property sets, whether the value that should be set is within the range specified for this property. Moreover, Disconnect for a detector device is rejected as long as it is acquiring data.

Q: What are Sharable interfaces (Instrument configuration) and are these relevant for driver development?

A: Shareable Interfaces (such as the UCI-100 or PCI cards with TTL inputs or outputs) provide hardware services (such as relays, TTL inputs, A/D converters and D/A converters). These objects are not associated with a particular instrument, as they can be operated independent of each other. Individual drivers (which are associated with a particular instrument) can use these objects via the `IPortUser` interfaces. For example, a driver for a detector supporting analog data acquisition only could implement a channel device that records the data that is physically delivered to one of the UCI-100's A/D converter inputs.

DDK based-drivers can not appear under Shareable Interfaces at this time, but it is possible to provide objects for use by other drivers via the `IPortProvider` interfaces.

Q: How can I access a property value that is maintained by another driver?

A: Provided you know the full symbol path (remember that the instrument and device names are something users can change in ICM), you can use the `IDDK.RequestPropertyValue()` function. However, this is polling based and should not be used for accessing foreign properties at high rates. Also, it is not guaranteed that you see all values if the property is changed quickly by the foreign driver. An extended interface with callbacks is in preparation.

Q: How does the data acquisition of diagnostics channels work?

A: It works the same as for any detector data, using AcqOn and AcqOff. The only difference is whether `IChannel.NeedsIntegration` is set or not.

Q: What is `CmHelpManager` and do I need it?

A: You need this if, and only if, you want to integrate context sensitive help topics with the common help file that ships with Chromeleon. If you provide your own small help file for your driver, you don't need to use this class.

Q: What does a typical temperature control look like?

A: Regulating the temperature is a firmware duty. The CM interface should be similar to:

1. When temp control (TempCtrl = On|Off) is set to Off, firmware should simply stop regulation. TempReady (NotReady|Ready) should be true. Safety limits (if any -> Temperature.UpperLimit, Temperature.LowerLimit) need still be monitored and result in an abort error if temperature is outside these limits to allow for emergency shutdown.
2. When temp control is set to on, start regulation towards the most recently set Temperature.Nominal. (see 4. below)
3. When temp nominal is set while temp control is off, turn on temp control implicitly. As this leads to a change in temp control property, log it explicitly.
4. When temp nominal changes to a new value, set temp ready to false and start heating/cooling.
5. When new nominal temp is reached (i.e., $\text{abs}(\text{value} - \text{nominal}) < \text{delta}$), start equilibration timer.
6. When equilibration timer has elapsed, set temp ready to true.
7. During equilibration time, if value is outside delta interval, or a new nominal is set, reset equilibration timer.

27.3 Miscellaneous CM6 hints

- In PGM Editor plug-ins, use `IEditPGM.SetDocModified()` rather than `IPageProperties.SetModified()`.
- For autosamplers, the PGM Editor plug-in in CM6 needs to generate the Inject command line explicitly. A typical pattern is

```
// Page Write Handler
public void OnPageWrite()
{
    if (m_Container.Edit.Mode == EditMode.Online)
        return;
    if ((m_Container.Edit.Mode == EditMode.Wizard) || m_isDirty)
    {
        // Store/update method in PGM
        AddInjectToPgmScript();
    }
}

// Write Inject command to PGM
public void AddInjectToPgmScript()
{
    try
    {
        // Insert Inject for injector
        ISymbolList subDeviceList = m_DeviceSymbol.ChildrenOfType(SymbolType.Device);

        foreach (ISymbol subDevice in subDeviceList)
        {
            // Create a PGM step Inject at retention 0.0
            ISymbol symbol = subDevice.Child("Inject");
            if (symbol != null)
            {
                IPgmStepCommand step = m_Container.Edit.PgmSteps.GetStep(symbol, new RetentionTime(0),
                StepPosition.Inject, true) as IPgmStepCommand;
                if (step != null)
                {
                    step.Write();
                }
            }
        }
    }
}
```

- For 3rd party method data blob access, you need to reference an additional assembly:


```
<Reference Include="SharedFifo, Version=1.9.0.0, Culture=neutral,
processorArchitecture=MSIL">
    <SpecificVersion>False</SpecificVersion>
    <HintPath>..\..\..\Win$(ConfigurationName)\DDK\V1\SharedFifo.dll</HintPath>
    <Private>False</Private>
</Reference>
```

 and a


```
using Dionex.Chromeleon.DDK.Implementation;
```

 statement in the code.
- IAAttributeDefinition is in Dionex::Chromeleon::Symbols::Extensions provided by C:\Chromel\Bin\DDK\V1\CmSymbolsExtensions.dll
- For conveying configuration information from driver to client, use this:

```
Driver:
const short cat_xml3rdPartyData = 293;
IAAttributeDefinition attributeDefinition = device as IAAttributeDefinition;
attributeDefinition.AddString(cat_xml3rdPartyData, configurationInfo);

PGM Editor plug-in:
const short cat_xml3rdPartyData = 293;
ISymbolExtension symbolExtension = device as ISymbolExtension;
if (symbolExtension == null)
    return String.Empty;
IAAttribute attribute = symbolExtension.Attributes[cat_xml3rdPartyData];
if (attribute == null)
    return String.Empty;
String configurationInfo = attribute.Value as String;
```

- Chromeleon installation path can be retrieved from registry key


```
HKEY_LOCAL_MACHINE\SOFTWARE\Dionex\Chromeleon\InstDir\Path
```

 If this is e.g. c:\Chromel_680_SR10,
 DDK would then be in c:\Chromel_680_SR10\bin\DDK\V1\CmDDK.dll,
 driver installs to c:\Chromel_680_SR10\bin\DDK\V1\<manufacturer>\<some subdir>
- To add more than one parameter to a command when generating script, use

```
e.g. UdpAdd      Volume=0.000, From=SampleVial, To=ReagentAVial
```

```
newStep.Parameters["Volume"].SymbolValue.StringValue = "0.0";  
newStep.Parameters["From"].SymbolValue.StringValue = "SampleVial";  
newStep.Parameters["To"].SymbolValue.StringValue = "ReagentAVial";  
newStep.Write();
```

- Some samplers might not be able to support injections triggered via the Inject dialog in the CM6 toolbar. A typical pattern is

```
void IEditorPlugin.GetOnlineDlgPages(IPageList pages, IEditPGM edit, ISymbol deviceSymbol,  
OnlineDlgType dlgType)  
{  
    switch (dlgType)  
    {  
        default:  
            MessageBox.Show(Resources.ERR_ModeOnlineNotSupported, Resources.UI_PGM_TITLE);  
            break;  
    }  
}  
<data name="ERR_ModeOnlineNotSupported" xml:space="preserve">  
    <value>The Thermo instrument is download controlled. Direct control is not supported by  
the driver.</value>  
</data>
```