Reading Files using the C# file reader

Rev: Dec 7 2015

# Introduction

This document defines how the .net (C#) file reader can be used to access data from raw files, and various other Xcalibur files. The file reader is included in the common core project (version 3.0 onwards).

It is expected that this reader can be used as a complete replacement to the C++ version, from all .Net products.

The code is designed for use by 64 bit code only.

We have only tested this from projects compiled using visual studio 2013 (.net 4.5.1), on windows 7 64 bit.

We have done limited tests of:

* Projects which are compiled with VS 2015 (.net 4.6)
  + The tool will remain at 4.5.1 minimum framework.
* Applications running on windows 10.
  + No known issues.

# Data Reading Interface

Raw Data is provided using the IRawDataPlus interface.

This interface and the types used by the interface are defined in the dll:

**ThermoFisher.CommonCore.data.dll**

## Brief History

The raw file format was created about 1994, for the “Finnigan LCQ” mass spectrometer data system.

The same history applies to certain other files used by the Xcalibur system, including “pmd” files (processing method).

Some data within these files may be “legacy data” and may be currently not displayed on screens of Xcalibur.

### Original Technology

This system was written in C++, with file objects defined by header files.

This was designed for “single threaded 32 bit processes”. (At the time, there were no multi-core processors).

Since there were a large number of headers, and a specific compiler was needed, this software was not practical for customer use.

For customer access, COM objects (wrappers) were created. (including XDK, XRawfile, MSFileReader).

If you have experience with one of these prior toolkits, please check for available migration guides. In particular the MS file reader team is developing such a guide.

Early systems collected raw files which were typically 200kb in length. Large files were considered as >5MB., with very few files of size 30MB or above. The “new” (at that time) 32 bit Windows NT could address all files as a “memory map”.

Over the next decade: files had grown above 1GB. Full file mapping (in 32 bit) was no longer practical and either “seek and read” or “partial file maps” were needed for larger areas. Internal addressing was updated (to 64 bit) to support files >4GB. Software was updates to run “as 32 bit on 64 bit windows” extending the technology life.

From about 2010 we have supported 64 bit apps on 64 bit “processing workstations” for large proteomics data. We have since migrated this to shipping 64 bit OS with all MS systems. By recompiling our C++ base in 64 bit (with no code changes) “memory pressure” on applications was significantly improved.

### New technology

In review of our future file reading architecture, .Net came across as a better option that C++.

* A key factor in a file reader was “stability”. Net object lifetime and heap management offers higher stability against memory fragmentation and related issues in C++ code. With newer files above 10GB, memory management (even in 64 bit C++) is problematic.
* With most code already in .net, we no longer needed to keep both “.net” and “C++” heaps, leading to better memory performance.
* The C++ system was a “single thread reader”, using “seek and read”. The .Net framework offers very good parallel tools. It was determined that recoding in “parallel C#” would be more beneficial to stakeholders than “resigning in parallel C++”.
* Customer applications went through a COM layer (which we rarely used internally).
* Microsoft has committed to making .Net open source and cross platform (Linux and OS X).

## Interfaces

In order to plan for this migration, we began to design “C# interfaces” to our data several years ago.

If we had “the correct interface”, and designed software to use it (rather than linking the C++/CLI code directly), we could then “plug and play” new file reading technology into exiting applications.

A data reading interface “IRawData” was defined about 2010, for the “Watson LIMS” integration project.

This interface was designed to return all valuable data in a raw file, such that the data could be:

* Transmitted over a network.
* Written and read from a XML format.
* Read from existing raw files (using the C++ layer).

This interface was initially implemented against the existing (32 bit) C++ code, and also against XML data.

The interface was later implemented against the 64 bit C++ code. Several of our tools were coded to use this interface, where possible.

Over the next 4 years, limitations in this interface were analyzed. Especially: any data reading limitation which led to code continuing to use the older “C++ DLL” directly, for access to raw data.

In 2014, the IRawDataPlus interface was completed, and supported by the CommonCore 2.0 project, offering more extensive data reading features. For backwards compatibility, this inherits from IRawData.

This interface was initially implemented using IoAdapater64, which was linked to the 64 bit “ThermoFisher.Foundtion.Io.dll”, which in turn linked directly to the 64 bit compilation of the C++ project “fileio.dll”.

### Pure .Net reader

Having completed a better interface to our data, we were able to create a new 64 bit file reader in pure .net. By knowing we were “always 64 bit”, no more “complex 32 bit memory model management” was needed. By being in .Net, we gained the significant advantages of Microsoft’s memory management scheme.

This project was added to common core 3.0.

CommonCore 3.0 includes an improved definition of the IRawDataPlus interface, which remains compile time compatible with the earlier versions.

Interface definitions to all data are defined in the “ThermoFisher.CommonCore.data.dll”

The C# file reading code is in “ThermoFisher.CommonCore.RawFileReader.dll”

Although projects may link directly to the factories in “RawFileReader”, it is recommended that factories in “data” are used, as we may provide alternative technologies in future. There is no significant performance difference between these choices, and applications may use either set of factories.

# Raw File structure

## Overview

Raw files are designed with “meta data” at the start, such as “sample information” followed by a number of “Instruments”.

The term “Instrument” in a raw file refers to a single device (one detector, or one auto-sampler). Instrument (in this context) does **not** refer to “the collection of devices in the used to acquire data”. In the terms of the Xcalibur data system, that is referred to as “the configured set of instruments”.

So: Raw file collected from the following devices:

* A mass spectrometer.
* An LC pump.
* A UV detector
* An auto sampler

May potentially have 4 “instruments” in the file.

Two of the instruments (mass spectrometer, UV detector) are detectors, and can return chromatography data.

Only one of the instruments (the mass spectrometer) has scanning data.

The other instruments (the LC pump, the auto sampler) can (at most) record:

* The name of the instrument (instrument name, model, serial number etc.)
* Logs from the instrument (such as status log, giving “pump pressures over time”).

The raw file system defines only two types of scanning detectors: Mass spectrometer and PDA (UV photo diode array detector).

The raw file has a “flat model” in that each instrument has the same set of available entry points (all instruments may have a status stream, etc.).

When using data from a raw file, applications can inspect what instruments are available, and should only display to the operator the data which is relevant to that detector type.

Methods which are not defined for a specific detector type may either return diagnostic data, not typically presented to a user, or may throw exceptions. Application layers should validate what data is available, and what interfaces are called, depending on the selected instrument type.

The “instruments which log data” in a raw file, do not have a “one to one mapping” with an “instrument method”.

For example:

A “mass spectrometer detector” may also have analog inputs on the same mechanical device.

This device may therefore record three instrument streams:

* One MS detector
* Two analog detectors.

If the analog channels are sampled at the same rate, there may be two instruments:

* One MS detector
* One analog detector, with 2 channels.

Each “instrument” has only one “time series” and may record one or more channels per time point.

A file may only contain one “MS detector” and may have multiple instances of all other instrument types.

Note that instruments may only have data pertinent to that instrument type, so:

An MS instrument in a raw file may not have “analog data” in the same “data stream” as the MS scans. An MS detector would (if needed) create a separate “virtual instrument” within the file, to record that analog data.

Because data for each instrument is separately recorded in its own section of the file, and because methods called on one instrument cannot get data recorded for another instrument, one possible application model is to use separate threads to process data for separate instruments.

The IRawDataPlus interface does not have separate “classes” or “interfaces” for each instrument type, as all instruments have the same entry points.

This permits, for example a “status log display” to be based on “IRawDataPlus” only, such that it display the log for “whatever instrument the application selected”, rather than having to be separately coded to “Get a status log from a MS detector” or “get a status log from a UV detector”.

This also permits future flexibility:

Although “get a scan” is not currently defined for a “non-scanning detector”, research code can log data for a device of type “other” and can inspect that data “scan by scan”, as all devices in fact have the same entry points, and the same data logging features.

Instruments in the raw file are separated into two basic base classes internally:

* MS Detectors
* All other devices.

Methods relating to “scan filters” and “scan events” are specific to MS detectors, and should never be called after selecting any other instrument type. Such calls will always throw exceptions.

A very simplified view of the file is:

The raw file begins with (fixed) header information, such as time stamp, sample name, vial number etc.

Other blocks (not listed) include autosampler tray data, and other information known before any data is acquired. A copy of the “instrument method” is included in the file. As noted above, this is not directly connected to the “instrument list” which is an index to the list of “detector data streams” contained in the file.

For each instrument in the list, there is a data area (an instrument), which can be viewed by calling the “SelectInstrument” method on the IRawDataPlus interface.

In this example, there are two instruments.

* MS
* UV

Each instrument contains an index table to the various objects and streams available for an instrument. For example Within the MS data, the next level of indexing includes:

After calling “SelectInstrument”, methods such as “GetStatusLogForRetentionTime” can inspect these data streams.

Note that the device data steams do not contain “chromatograms”. As in the term XIC (eXtracted Ion Chromatogram), chromatograms are created from other data.

## About logs

Much of the (non-scan) data in raw files is in log format.

This includes:

* Tune method
* Status Log
* Trailer extra log

Logs have a “header” which defines the record format of the log, and fixed size records.

For example:

A certain devices status log may record two integers and a float, recorded as:

|  |  |  |
| --- | --- | --- |
| Field name | Data Type | Bytes |
| Temperature 1 | Short int | 2 |
| Temperature 2 | Short int | 2 |
| Pressure | 32 bit float | 4 |

Each log record would then be exactly 8 bytes long.

The header would then return the filed labels “Temperature 1”, “Temperature 2”, “Pressure”.

For float items, there is a suggested display precision.

Most logs are returned as formatted strings, ready for display.

By using the header information, applications can reformat the log data as needed.

One of the logs is intended to contain additional numeric data about MS scans (the “trailer extra” log). This log may be accessed directly in numeric form, as it is used for certain calculations.

The name “trailer extra” comes from the following design:

* Scan data is logged first by instruments.
* A “scan trailer” is then written, including some information about that scan (such as retention time, or base peak mass).

That trailer is in a fixed format, so that all applications know that “retention time” is a valid field.

All other (device specific) data about a scan follows as a “trailer extra” record.

# Object lifetime

It is important to consider that this interface is connected to a data source. In the case of the C# file reader that is “an open handle to a file”. In the case of other implementations of IRawDataPlus, this may be a “network connection”, “a database” etc.

In general:

Avoid “disposing” of the returned interface until all data has been extracted.

This is especially important for objects that return an interface or an enumeration. This interface may need a connection to the “active data”.

An example is “FilteredScanEnumerator” which does not hold “the collection of scans” in memory. Scans are read from the data source as they are enumerated.

There are also uses of “Lazy evaluation” in returned objects and interfaces, which require a connection to the data to remain.

It is highly recommended that any information needed from IRawDataPlus is “used for processing” or otherwise archived before disposing of the original interface.

This is a variation from the earlier IRawData interface, which returned mostly “objects” which fully evaluated data from the file, and had an “independent lifetime”. We found that using interfaces and lazy evaluation improved performance.

Note:

Lazy evaluations can typically be forced by using “deep clone” methods.

More specific lifetime notes are added for specific methods.

# Inspecting files

If you wish to open a file temporarily, to read the file header only, you can use this factory in CommonCore.data.Business

FileHeaderReaderFactory.ReadFile

The read file method has this signature:

public static IFileHeader ReadFile(string fileName)

This can get information from (at least) .raw, .pmd and .sld files, without keeping the file open.

Xcalibur uses this technique to show preview information about a file (such as operator who created the file) in its file open dialog boxes. This same file header is available from additional files within the Xcalibur product family. This document only describes use for the above file extensions.

This same data can be obtained by using the property “IFileHeader FileHeader { get; }”, on an open raw file.

|  |  |
| --- | --- |
| Property | Purpose |
| string WhoCreatedId | The creator Id is the full text user name of the user when the file is created. |
| string WhoCreatedLogon | The creator login name is the user name of the user when the file is created, as entered at the "user name, password" screen in windows. |
| int Revision | The file format revision |
| DateTime CreationDate | The file creation date. |

# Opening Files

This chapter deals with the fundamentals of opening raw data (for reading).

All algorithms which process raw data should accept the IRawDataPlus interfaces (or any members, base class etc. of that interface).

To fully isolate code from specific implementation, the factory class in CommonCore.data.Business namespace may be used:

RawFileReaderFactory

Code which needs to open a raw file needs to use the Static methods in this class.

Note that the class “RawFileReaderAdapter” of the raw file reader dll remains public, and can be used to specify that DLL as the reader.

The remainder of this document only discusses opening raw files via “RawFileReaderFactory”

When opening a file:

Decide first if your application will be using a single thread or multiple threads to access the data.

In most cases there is no performance penalty in supporting multiple threads (parallel access to the data) so it is reasonable to plan ahead, and open for multiple thread access.

The sections below describe the calls to make for these alternative approaches.

The IRawDataPlus interface is structured as a set of method calls and properties.

Methods or properties may return objects or interfaces.

The IRawDataPlus interface is “disposable”.

All calling code should extract information from (process) the returned objects and interfaces before disposing of the original object, as disposing of an object which implements IRawDataPlus will close the data connection.

A single instance of IRawDataPlus provides data to one thread.

The raw file reader has two methods to open a file which have a path (string) parameter.

The sections below describe these methods.

An example exists in common core “demos” which shows how to delegate these methods such that business logic does not have to reference the raw file reader DLL.

See the code in “RawFileReaderDemo” file “program.cs” for this example.

## Single threaded code

The factory method RawFileReaderFactory.ReadFile can be used to open a raw file.

For example:

Using ( myFile= RawFileReaderFactory.ReadFile (path))

{

DoStuffWith(myFile);

}

If a using statement is not feasible:

It is important to dispose of the file after all operations are completed, because the active object keeps a file open on disk.

Do not use a using statement if any part of the IRawData interface is passed to an object with a lifetime beyond the using statement.

The returned interface is an instance object. It has various properties and methods which can change the object’s state. The most significant is “RefreshViewOfFile”. When a file is being acquired, this method changes the number of “available scans” seen in the raw file, for all detectors.

The property “bool IncludeReferenceAndExceptionData { get; set; }” will change the operation of many calls which return MS data.

Fortunately, the file reader has a mode to fully support multiple threads, including lockless parallel access to files. If this is required refer to the next section on multi-threaded code.

## Multi-threaded code

As noted in the section above, the IRawDataPlus interface is an instance object, and should be used by only one thread. However, the raw file reader can generate any number of such objects, so that any number of threads can access the same raw file in parallel.

For multi-threaded code, the pattern is:

A manager object is created (from a file name), which cannot itself read any raw data.

This manager is then used to create IRawDataPlus interfaces for each thread, as detailed below:

The C# raw file reader includes an implementation of IRawFileThreadManager

The factory method RawFileReaderFactory.CreateThreadManager can be used to open a raw file, for use by multiple threads.

For example:

var myThreadManager = RawFileReaderFactory.CreateThreadManager (path);

This can be used for multi-thread access to the same raw file.

All business logic still accesses the information using the IRawDataPlus interface. *Application teams do not have to write any thread synchronization or locking code.*

The usage pattern is as follows.

Open a file returning a thread data manager:

|  |
| --- |
| Try  {  myThreadManager = RawFileReaderFactory.CreateThreadManager (filename)  }  Catch exceptions |

Exceptions may occur if the required raw file reader dll is not found, or a null string is sent for the raw file name.

The action of opening a file executes a small number of “one time” single threaded actions, such as:

* Opening the file on disk.
* Reading the file header (time stamps, operator name etc.)
* Reading sample information.
* Obtaining the list of detectors.

Important: This “thread manager” cannot itself read any raw data.

Assuming no exceptions, for each thread which needs access to data (including the current thread, if needed):

IRawDataPlus myThreadDataReader = myThreadManager. CreateThreadAccessor();

To test for errors, create a thread accessor.

This property:

/// <summary>

/// Gets the file error state.

/// </summary>

IFileError FileError { get; }

Can then be used to check for any errors (such as invalid file name).

Performance Note: Within the C# reader, there is no significant performance overhead in “CreateThreadAccessor”. In evaluation of this, we have tested that in can be used within a “Parallel.For” pattern, to make an accessor for each scan in a raw file.

Note: The method “CreateThreadAccessor();” is actually a member of the interface “IRawFileThreadAccessor” which has other implementations noted later.

After all created threads have exited, call:

myThreadManager.Dispose();

*Using this pattern all business logic for multi-threaded code is exactly the same as for single threaded code.*

All interface members of IRawDataPlus can be used, with no concerns for locking or thread safety. Do not add any additional locks in calling code.

Data for each thread is read in parallel (lockless), wherever possible. Some larger objects in the raw file use (thread safe) lazy loading, such that, for example: The first thread which opens the MS data will incur a small overhead for opening the MS data stream, then all other threads will share parallel access to that same stream.

Note: Locking may occur when a file is opened in real time mode (during data acquisition), as a real time data is continually changing state. This locking is internal to the file reader, and calling code need not take any special action for real time files.

## Alternate approaches to threading

Even though the C# reader natively supports parallel access to data, not all file readers may support this, but “There’s an interface for that”.

The interface “IRawFileThreadManager”

Is defined as follows:

public interface : IRawFileThreadAccessor, IDisposable

That is: When you open a raw file, and obtain this interface, as described above, you “dispose” to close the file.

The mechanism for allocating data to threads is descried in “IRawFileThreadAccessor”, so if the application layer code opens a file, it may pass “IRawFileThreadAccessor” to the next layer of code.

As above, that code can call “CreateThreadAccessor”:

MyMethod(IRawFileThreadAccessor myThreadDataReader)

IRawDataPlus myThreadDataReader = myThreadDataReader. CreateThreadAccessor();

An advantage of this scheme is that an application can be designed to use other readers which do not support the thread manager interface, by using another implementation of “IRawFileThreadAccessor”.

CommonCore already includes an implementation of this:

public class ThreadSafeRawFileAccess : IRawCache, IRawFileThreadAccessor

This class permits multi thread support (by IRawFileThreadAccessor) to be generated from any instance of IRawDataPlus

Using this public constructor:

public ThreadSafeRawFileAccess(IRawDataPlus file).

Via this pattern, library code need not be aware of how the thread management is done. If the library code (business logic) receives “IRawFileThreadAccessor” it will be able support multi-threaded access to any raw file, from any file reader.

The difference is one of performance.

When using the C# raw file reader’s direct implementation (ThreadedFileFactory) the business logic will have lockless parallel access to data. When the interface passed in is created by the class “ThreadSafeRawFileAccess” then calls into the file are serialized via locks.

Note: In all cases

* The business logic need no add any additional locking.
* The business logic need not reference any specific file reading DLL.
* All required interfaces are contained in ThermoFisher.CommonCore.Data.dll

# Reading Raw File Headers (sample information)

This topic details how to read the “sample information” and other headers from a raw file.

That is:

All information which is logged to the file, before data acquisition begins.

Because these are not “instrument data stream specific”, it is not necessary to select any instrument data stream before reading any of these fields.

All of these items can also be read after selecting an instrument.

A raw file has the following table of contents:

|  |  |
| --- | --- |
| Item | Purpose |
| File header | General information about the file, such as “who created” and “creation date” |
| Sample information | Information from the “sequence row” or batch, such as “vial number” or “sample ID” |
| AutoSampler tray configuration | Details about the tray shape used. |
| Instrument method | Data (text format) about the instrument settings. Note: The raw file reader does not show “binary method information” which is private to the instrument. |
| Detector list | Lists the various detectors which are configured to store data and logs in the raw file. |

The following methods or properties can be used to inspect this information:

## File Header

IFileHeader FileHeader { get; }

See the chapter above on “Inspecting files” for details about the IFileHeader interface.

## Autosampler Tray information

IAutoSamplerInformation AutoSamplerInformation { get; }

This interface has the following properties:

|  |  |
| --- | --- |
| Property | Purpose |
| int TrayIndex | For an autosampler with multiple trays, the tray number. -1 when “not applicable” |
| int VialIndex | Vial index within the tray. -1 when “not applicable” |
| int VialsPerTray | The number of vials or wells in the tray. -1 when “not applicable”. For a rectangular tray or plate, this will be “VialsPerTrayX” x “VialsPerTrayY”. |
| int VialsPerTrayX | The number of vials or wells across the tray. -1 when “not applicable” |
| int VialsPerTrayY | The number of vials or wells down the tray. -1 when “not applicable” |
| TrayShape TrayShape | The shape of the tray:  Rectangular, Circular, StaggeredOdd, StaggeredEven, Unknown, Invalid |
| string TrayShapeAsString | Descriptive name of the tray shape. |
| string TrayName | Name (model) of the tray |

## Sample information

The data which is entered on the acquisition grid (for example, home page sequence editor) is called “Sample information”

Use this property to obtain that information.

SampleInformation SampleInformation { get; }

This object has the following properties:

|  |  |
| --- | --- |
| Property | Purpose |
| string Comment | Descriptive comment about sample |
| string SampleId | Customer’s ID for the sample |
| string SampleName | Customer’s name for the sample |
| public string Vial | Vial or well used to acquire this sample. |
| double InjectionVolume | Amount of sample injected ( micro liter) |
| string Barcode | Barcode read by scanner. |
| BarcodeStatusType BarcodeStatus | Determines if a barcode was read. |
| string CalibrationLevel | Calibration or Qc Level |
| double DilutionFactor | Bulk dilution factor (volume correction) of this sample. |
| string InstrumentMethodFile | The instrument method filename used to acquire this sample. |
| string RawFileName | Name of acquired file (excluding path). |
| string CalibrationFile | Name of calibration file. |
| double IstdAmount | The internal standard amount of this sample. |
| int RowNumber | The row number of the sample, in the sequence. |
| string Path | Path to original raw data (at time sequence was acquired). |
| string ProcessingMethodFile | processing method filename |
| SampleType SampleType | Type of sample (for example, Blank, Unknown etc.) |
| double SampleVolume | sample volume (micro liters) |
| double SampleWeight | sample weight |
| string[] UserText | Collection of user text. These values are for columns which have configurable titles either in a sequence editor, or set by an application. See also “UserLabels” |

On the IRawDataPlus interface:

string[] UserLabel { get; }

Returns labels (titles) for the “user text”, present in “SampleInformation”.

Note that this is not part of the sample information, due to the way sequences are structured.

There is one set of column labels for the entire sequence, so “UserLabel” is not part of the data for one sample, permitting the SampleInformation object to be reused for data from the raw file, or from a row in a sequence.

Opening SLD files (Xcalibur sequence) is a separate feature, and is separately documented (see “Opening other files” chapter).

## Instrument method

### Introduction

For raw files which were acquired using an instrument method, it is possible to read:

* The names of the instruments used
* The text version of the instrument method, for each instrument.

Not all raw files have an instrument method. First test to see if a method is present:

bool HasInstrumentMethod { get; }

If this returns false, then the returned information from the following methods is undefined by the interface (they should not be called).

The property “InstrumentMethodsCount” returns the number of instruments which saved data in the instrument method.

### Instrument methods within a raw file

Raw files can contain a complete copy of the instrument method used to acquire the file.

This is an optional feature: Sometimes data created by an application window, or as part of a tuning or calibration workflow, would not have any method.

When capturing a method, a small amount of additional data is saved.

Xcalibur instrument methods are keyed to the “internal device names”. These names are plain text, which is suitable for use as a file name, or a registry key name etc. “Device internal names” are not the same as the “product names” which appear on windows such as “instrument configuration” in Xcalibur. The product names are referred to as “descriptive names” or “friendly names” of devices, rather than just “device name”.

There is also a distinction between “devices which save an instrument method” and “detectors which log data”.

This section is about “devices which save an instrument method” only. Devices which have data within an instrument method do not necessarily log any other data in the file. A common example is an autosampler, which typically has an injection method, but would have no time series data logged in a raw file. So: You cannot call “SelectInstrument” with a detector of type “Autosampler”.

The activity of “Exporting” an instrument method needs special attention, as described in the table below.

Note: The “InstrumentMethodsCount” is not related to “the number of detectors which logged data in the raw file”. Some instruments (such as most autosamplers) log no data, and some instruments (such as multi-channel UV) may have multiple detectors.

All other properties of the instrument method have one entry per instrument (as in “InstrumentMethodsCount”.

Instruments have:

* Instrument names: Which are not normally displayed (and may be the “registry key names” for the instruments). These names are also “stream names” in the compound document for an instrument method file.
* Friendly names: Which are longer (descriptive) display names.
* Method text: Which is multi-line text describing the method.

Here’s a summary of the methods and properties related to embedded methods.

|  |  |
| --- | --- |
| Method/Proerty | Meaning |
| int InstrumentMethodsCount { get; } | Gets the number of instruments which have saved method data, within the instrument method embedded in this file. |
| string[] GetAllInstrumentFriendlyNamesFromInstrumentMethod(); | Gets all instrument “friendly names” from the instrument method. These are the "display names" or product names for the instruments. For example: suppose you wanted to display instrument method data as “one tab per instrument”, then these names may be used as “tab names”. |
| string[] GetAllInstrumentNamesFromInstrumentMethod(); | Gets names of all instruments, which have a method stored in the raw file's copy of the instrument method file. These names are "Device internal names" which map to storage names within an instrument method, and other instrument data (such as registry keys). Use "GetAllInstrumentFriendlyNamesFromInstrumentMethod” to get display names for instruments. |
| bool ExportInstrumentMethod(string methodFilePath, bool forceOverwrite); | Export the instrument method to a file.  Because of the many potential issues with this, use with care, especially if adding to a customer workflow. Try catch should be used with this method. .Net exceptions may be thrown, for example if the path is not valid. Not all instrument methods can be exported, depending on raw file version, and how the file was acquired.  If the "instrument method file name" is not present in the sample information, then the exported data may not be a complete method file.  Not all exported files can be read by an instrument method editor. Instrument method editors may only be able to open methods when the exact same list of instruments is configured.  Code using this feature should handle all cases.  When the “forceOverwrite” parameter is true, then this call is permitted to save over existing files of the same name. If false: UnauthorizedAccessException will occur if there is an existing read only file. |
| string GetInstrumentMethod(int index); | Gets a text form of an instrument method, for a specific instrument.  “index” is the zero based index into the count of available instruments. The property "InstrumentMethodsCount", determines the valid range of "index" for this call. Some instruments do not log this data. Always test "string.IsNullOrEmpty" on the returned value. Multiple lines should be split using "\n". |

This code example adds “instrument names” and multi-line method text to a grid control “dataGrid”:

var names=\_raw.GetAllInstrumentFriendlyNamesFromInstrumentMethod();

int row = 0;

for (int index = 0; index < instMethodCount; index++)

{

dataGrid.Rows.Add();

dataGrid.Rows[row].DefaultCellStyle= new DataGridViewCellStyle(){BackColor = Color.Yellow};

dataGrid.Rows[row++].Cells[0].Value = names[index];

string methodText = \_raw.GetInstrumentMethod(index);

string[] splitMethod = methodText.Split(new string[] {"\n"}, StringSplitOptions.None);

foreach (string s in splitMethod)

{

dataGrid.Rows.Add();

dataGrid.Rows[row++].Cells[0].Value = s;

}

}

## Instrument (Detector) List

The interface has the following methods to interrogate which instruments logged data in to then file.

This information is commonly shown as “detector” in applications.

|  |  |
| --- | --- |
| Method/Property | Notes |
| int GetInstrumentCountOfType (Device type) | Gets the number of instruments (data streams) of a certain classification. For example: the number of UV devices which logged data into this file. |
| Device GetInstrumentType(int index); | Gets the device type for an instrument data stream at a (zero based) index. |
| int InstrumentCount { get; } | Gets the number of instruments (data streams) in this file. |

Example:

This code makes lists of detectors, to present on the UI:

\_instTypes = new List<int>();

\_instCount = new List<int>();

int adCardCount = 0;

int statusCount = 0;

int msAnalogCount = 0;

int pdaCount = 0;

int uvCount = 0;

int count = 0;

for (int instIndex = 0; instIndex < \_raw.InstrumentCount; instIndex++)

{

Device instType = \_raw.GetInstrumentType(instIndex);

string instName = string.Empty;

switch(instType)

{

case Device.Analog:

adCardCount++;

instName = "A/D card";

if (adCardCount > 1)

{

instName += " " + adCardCount.ToString();

}

count = adCardCount;

break;

case Device.MS:

instName = "MS";

count = 1;

break;

case Device.MSAnalog:

instName = "MS Analog";

msAnalogCount++;

if (msAnalogCount > 1)

{

instName += " " + msAnalogCount.ToString();

}

count = msAnalogCount;

break;

case Device.Other:

// other detectors only have status

statusCount++;

instName = "Status Device";

if (statusCount > 1)

{

instName += " " + statusCount.ToString();

}

count = statusCount;

break;

case Device.Pda:

pdaCount++;

instName = "PDA";

if (pdaCount > 1)

{

instName += " " + pdaCount.ToString();

}

count = pdaCount;

break;

case Device.UV:

uvCount++;

instName = "UV";

if (uvCount > 1)

{

instName += " " + uvCount.ToString();

}

count = uvCount;

break;

}

cmbInstrument.Items.Add(instName);

\_instTypes.Add((int)instType);

\_instCount.Add(count);

}

Note: The algorithm above adds instruments to a list, based on the order of detectors.

An alternative way is to use “GetInstrumentCountOfType” for each device type then show “All MS detectors” the “All PDA” etc.

You can also infer “HasPdaData” etc. by tests like “GetInstrumentCountOfType(Device.Pda)>0”

# Reading data from detectors

Before reading any data, a particular instrument (detector) must be selected by calling

SelectInstrument(Device instrumentType, int instrumentIndex);

Note that the instrument index in this call is “1 based”, so:

SelectInstrument(Device.MS, 1);

Selects the MS data.

## Getting detector details

The method “InstrumentData GetInstrumentData();” returns specific details about a detector. (Note: this is not related to instrument methods).

This table is mostly text, logged by the device driver. Exact meaning of this may depend on the specific device team. The file system can only report the logged text, and does not reformat or interpret it.

For example: Different driver teams may have different approaches to record an instrument name and Model.

|  |  |
| --- | --- |
| Property | Meaning |
| string Name | The name of this instrument (detector name). For example “TSQ 8000”. This must identify the equipment logging data (and not a data system). For example: this name should not be “Xcalibur”. |
| string Model | Model of the instrument. Typically a postfix to Name. |
| string SerialNumber | The detector’s serial number. |
| string SoftwareVersion | Version of the instrument (driver) software. |
| string HardwareVersion | Version of the instrument hardware. |
| string[] ChannelLabels | For analog or UV (channel format detectors), this gave a name for each channel. |
| string Flags | Any other additional information from the detector. |
| string AxisLabelX | Suggested label for X axis of this instrument (rarely used in software). Most software will use standard terms such as “m/z” for mass spec detectors, regardless of this value. |
| string AxisLabelY | This string may be used to record units for analog data. |

## Reading Scans

This topic is incomplete. See the definition of IRawDataPlus for available methods.

The IRawDataPlus interface includes several methods related to reading scan data, particularly from MS.

Before reading scan data, an application writer needs to know what family of instruments are supported by the algorithm. Data form different MS detectors can vary significantly in format.

Scan data can come from either MS or PDA detectors.

Scans (generically) have values for “**position**” (x) and “**intensity**” (y).

For an MS detector, **position** is the mass to charge ratio (m/z) and **intensity** is the absolute abundance.

For a PDA detector, **position** is a wavelength, and **intensity** is absorbance.

Data which is specific to MS may use the term “mass” implying m/z.

## MS scan overview

### Simple format

Simple instruments (such as a single quad MS) have a scan format which returns one data type, either “profile” or “centroid”.

The scans type code (scan event) indicates which of these formats is used.

Profiles represent a set of raw samples from the instrument, with no analysis. These are generally shown as a continuous line of a plot.

Centroids are internally greeted by running a “center of mass” algorithm on the profile (within the instrument firmware).

Some instruments will log only one or the other of these data types, while others can store both.

Instruments can scan continuously across a mass range (called a full scan) or over smaller SIM or SRM windows. These windows are referred to as “segments”.

So: An MS scan may have one or more segments (of the mass range), with a full scan having only one segment.

Segments may not overlap.

This data is red by the interface “GetSegmentedScan”

SegmentedScan GetSegmentedScanFromScanNumber(int scanNumber, ScanStatistics stats);

In order to read scan data, a scan number is provided.

This scan number is used to access an index record for the scan, which includes various statistics for the scan.

If the parameter “stats” is passed as NULL, then no statistics data is returned.

If the parameter is not null, then the object is polluted with the scan’s statistics.

This can reduce the number of interface calls needed, where the application needs a scan, and meta data about the scan.

The class SegmentedScan implements this interface to the data:

/// <summary>

/// Interface for Access to the data in a segmented scan

/// </summary>

public interface ISegmentedScanAccess

{

/// <summary>

/// Gets the number of segments

/// </summary>

int SegmentCount { get; }

/// <summary>

/// Gets the number of data points in each segment

/// </summary>

ReadOnlyCollection<int> SegmentLengths { get; }

/// <summary>

/// Gets Intensities for each peak

/// </summary>

double[] Intensities { get; }

/// <summary>

/// Gets Masses or wavelengths for each peak

/// </summary>

double[] Positions { get; }

/// <summary>

/// Gets Flagging information (such as saturated) for each peak

/// </summary>

PeakOptions[] Flags { get; }

/// <summary>

/// Gets the Mass ranges for each scan segment

/// </summary>

ReadOnlyCollection<IRangeAccess> MassRanges { get; }

}

Note that the positions and intensities in this object are not subdivided by segments.

The arrays contain the data for all segments in order.

Because of this, applications can choose to view the scan as just a set of mass and intensity values.

Segments are useful for high resolution SIM or SRM data, as it is possible to see the peak groups (centroid) or profiles within each mass window, using the common core plotting tools.

An application could also choose to do peak integration based on “the data in a particular segment”.

### Complex formats

Some instruments have a much more complex format of scan data, including two representations of the data (profile and centroid).

For these instruments, additional data is returned by:

GetCentroidsStream

A means of identifying what data may have this extended information, is to examine the packet type form the scan header.

Types 18, 19, 20 and 20 may have this data.

## Generating Chromatograms

The IRawDataPlus interface defines several overloads of methods for generating chromatograms.

One fact is the same for all methods:

The interface is only designed to read chromatogram data from one detector at a time.

However, you can use the thread manager to make new readers for separate detectors if you wish, and can requires chromatograms from multiple detectors in parallel.

Chromatograms for UV, PDA, and Analog detectors are most commonly generated using the simplest call:

/// <summary>

/// Create a chromatogram from the data stream

/// </summary>

/// <param name="settings">

/// Definition of how the chromatogram is read

/// </param>

/// <param name="startScan">

/// First scan to read from. -1 for "all data"

/// </param>

/// <param name="endScan">

/// Last scan to read from. -1 for "all data"

/// </param>

/// <returns>

/// Chromatogram points

/// </returns>

IChromatogramData GetChromatogramData(IChromatogramSettings[] settings, int startScan, int endScan);

The following class: (in CommonCore.Data.Buisiness)

/// <summary>

/// Settings to define a chromatogram Trace.

/// </summary>

[Serializable]

[DataContract]

public class ChromatogramTraceSettings : CommonCoreDataObject,

IChromatogramSettingsEx,

IChromatogramTraceSettingsAccess, ICloneable

May be used to create parameters for each chromatogram trace.

This method determines if the trace settings are compatible with the selected detector, then calculates the required chromatograms.

Note that: All chromatograms share the same “scan number” range, which may be set to “-1, -1” for full file.

There is no limit to the number of chromatograms which may be requested in one call.

For MS detectors, these overloads may be of interest:

IChromatogramData GetChromatogramData(

IChromatogramSettings[] settings, int startScan, int endScan, MassOptions toleranceOptions);

IChromatogramDataPlus GetChromatogramDataEx(

IChromatogramSettingsEx[] settings, int startScan, int endScan);

IChromatogramDataPlus GetChromatogramDataEx(

IChromatogramSettingsEx[] settings, int startScan, int endScan, MassOptions toleranceOptions);

Tolerance options are applied to mass ranges (for XICs) and to values in scan filters.

The “massPrecision” value in filters (after converting from a text string) is set based on the tolerance options, when supplied.

If no tolerance options are supplied, the default value of precision is taken form the mass spectrometer’s run header.

The “Ex” settings permit an application to supply a “compound name”, for use when the MS data has compound names embedded in the scan events.

The “IChromatogramDataPlus” extends the returned data, by adding “base peak” values.

See IRawDataPlus definition for details on the specific parameters.

MS chromatograms are read using parallel code (using the class ChromatogramBatchGenerator).

Applications may use this class to configure custom chromatogram generation, such as “reading many chromatograms in parallel, with separate time limits for each chromatogram”.

Assuming “manager” has been created using “var myThreadManager = RawFileReaderFactory.CreateThreadManager (path);”

The following method would return an unfiltered TIC from the MS detector.

private static ChromatogramSignal[] GetUnfilteredTic(IRawFileThreadManager manager)

{

ChromatogramSignal[] chroTrace;

using (IRawDataPlus file = manager.CreateThreadAccessor())

{

// open MS data

file.SelectInstrument(Device.MS, 1);

// Define settings for Tic

var settingsTic = new ChromatogramTraceSettings(TraceType.TIC);

// read the chromatogram

var data = file.GetChromatogramData(new IChromatogramSettings[] {settingsTic}, -1, -1);

// split the data into chromatograms

chroTrace = ChromatogramSignal.FromChromatogramData(data);

}

return chroTrace;

}

The class “chromatogram trace settings” can be configured to make various other chromatograms.

For example, this makes an XIC, out of all mass from 0 to 1000:

string filterAll = string.Empty;

// make a request for the chromatogram.

// Define settings for Mass

ChromatogramTraceSettings settingsMass0To1000 =

new ChromatogramTraceSettings(TraceType.MassRange)

{

Filter = filterAll,

MassRanges = new[] { Range.Create(0,1000) }

};

This block of code makes an XIC of a single ion plus tolerance, from scans of type “ms”, assuming “file” is created as above

IRawDataPlus file;

// Other code to open this file…

// read the ms data

string filterMs = "ms";

// next make a request for the filtered chromatogram.

// Define settings for Tic

ChromatogramTraceSettings traceSettings =

new ChromatogramTraceSettings(TraceType.MassRange)

{

Filter = filterMs,

MassRanges = new[] { new Range(1422.05, 1422.05) }

};

// open MS data

file.SelectInstrument(Device.MS, 1);

// create the array of chromatogram settings

IChromatogramSettings[] allSettings = { traceSettings };

// set tolerance of +/- 0.05 amu

MassOptions tolerance=new MassOptions(){Tolerance = 0.05,ToleranceUnits = ToleranceUnits.amu};

// read the chromatogram (1422 to 1422.1)

var data = file.GetChromatogramData(allSettings, -1, -1, tolerance);

The examples above create a single chromatogram. Note that the settings are passed as an array.

Any number of chromatograms can be read from a detector. It is usually more efficient to request multiple chromatogram per call.

Having read multiple chromatograms, the following line (from the examples above) will still work to extract the data into objects, where the data can be manipulated further:

// split the data into chromatograms

chroTrace = ChromatogramSignal.FromChromatogramData(data);

### Creating custom chromatograms

Chromatograms may be made in a custom manner, just by writing your own code to read and analyze scans. However, the class “ParallelChromatogramFactory” can be used to generate a wide variety of chromatograms, from MS data.

For example: Having a separate set of retention time limits for each chromatogram.

This factory is used by the .Net raw file reader internally, with specific rules based on the passed in chromatogram trace settings.

You could use a pattern like this to configure the generator:

public class Limit

{

public double Low { get; set; }

public double High { get; set; }

}

public class Component

{

public Limit RtRange { get; set; }

public Limit MassRange { get; set; }

public string Filter { get; set; }

public string Name { get; set; }

}

private ChromatogramDelivery[] CreateMassChromatograms(IRawDataPlus rawData, List<Component> components)

{

rawData.SelectInstrument(Device.MS, 1);

var generator = new ChromatogramBatchGenerator();

// configure this tool to use raw data

ParallelChromatogramFactory.FromRawData(generator, \_rawData);

int totalChros = components.Count;

var deliveries = CreateMassChromatogramJobs(components, totalChros);

var tasks = generator.GenerateChromatograms(deliveries);

Task.WaitAll(tasks);

return deliveries;

}

private ChromatogramDelivery[] CreateMassChromatogramJobs

(List<Component> components, int totalChros)

{

ChromatogramDelivery[] deliveries = new ChromatogramDelivery[totalChros];

for (int trace = 0; trace < totalChros; trace++)

{

var imported = components[trace];

Limit massrange = imported.MassRange;

deliveries[trace] = new ChromatogramDelivery()

{

Request =

ChromatogramPointBuilderFactory.CreatePointBuilder(

pointRequests:

new List<IChromatogramPointRequest> { ChromatogramPointRequest.MassRangeRequest(massrange.Low, massrange.High) },

retentionTimeRange: Range.Create(imported.RtRange.Low, imported.RtRange.High),

scanSelector: ScanSelect.SelectByFilter(\_rawData.GetFilterFromString(imported.Filter))

)

};

}

return deliveries;

}

In simple terms:

The chromatogram generator relies on just 2 things: “scanSelector:” determines if a scan should be included in a chromatogram, and “pointRequests:” determine how a numeric value is calculated from the data for an included scan.

See the object help for more details.

## Scan types, events and filters

This topic is specific to MS data.

There are three terms which are used to describe “how a mass spectrometer is programmed to scan”

|  |  |  |
| --- | --- | --- |
| Term | Meaning |  |
| Scan Event | The “event” is a programmed (planned) activity for a mass spectrometer scan. It can be used to describe either the planned scan, or an actual scan which occurred using these settings.  For example: An MS method may be defined to “scan ms data in centroid mode, with alternative positive and negative scans”. This would translate to 2 (planned) events “positive centroid ms” and “negative centroid ms”.  If there are 1000 scans in the file, then 500 may be codes as “using the positive centroid ms” event, and 500 using the “negative centroid ms” event. When displaying such events as a text string, a shorthand notation is used. In this case “+ c ms” for the positive scans and “- c ms” for the negative scans.  Some events may be defines as “custom” or “data dependent”. With these event types, the way in which the MS scans is not known before a scan occurs. Scanning information needs to be separately saved with each scan.  Custom implies “some other information has been used to determine how the scan is performed”. For example: individual scans saved from a tuning window may be flagged as “custom”, because there is no predefined method.  Data Dependent implies that the MS may trigger a scan, and analyzing previous scans. Again: the scanning rules are not known before this decision is made, and must be saved with a scan. The table recording this (per scan) data is referred to as the “trailer scan event” table.  The pre-programmed events are also described as “method scan events”, as they are the events known at the time an instrument method is written. |  |
| Scan Filter | Filters are a selection mechanism for scans. For example, given the pos/neg switching experiment described above, a filter of “-” would return only the negative scans. A filter of “ms” would return all of the scans.  Scan filters are often used in chromatograms, to select a particular ms/ms transition.  Enumerators are available to find the set of scans which match given rules.  Filters can be considered as a set of “and” conditions.  So, a filter of “+ ms” means, “scan is positive AND scan is ms”.  Filters can have “AND NOT” operators for many of the fields. The symbol “!” is used for NOT.  A common one is “data dependent” (code “d”).  The filter “ms !d” will return “all MS scans which are not data dependent”.  This code will not return any MS/MS data. |  |
| Scan Type | Scan type is description of how a particular scan was performed, as a text string, in the scan filter format.  This can be useful just to display how a scan was performed, or to search a file for like scans, using filtering. |  |

An important difference between scan event and scan filters is:

Events are immutable. They are a record of what occurred.

Filters can be constructed and modified, as they are intended to perform searches on the data.

The precise list of possible codes in a filter will vary by MS detector. In general, filtering logic will be tied to a specific detector.

Because this raw file format goes back over 20 years, some of the available codes in filter syntax refer to technologies which are no longer used (such as old source types).

It is suggested that to design filters, data is acquired with a specific ms method, and the returned filter codes are examined for that scan. There may also be data about scan formats in the manual for a specific mass spectrometer model.

There are a number of methods in IRawDataPlus relating to scan types, events and filters.

There are also some extension methods.

Because many applications save filter as “text strings”, filter parsing needs to be performed, to determine the rule set. There are also classes and interfaces for this, which can:

Validate that a text string is a well formed filter.

Avoid duplicating the parsing, when processing many scans.

Assist in efficient filtering. That is “test if a particular scan’s event would pass a given filter”.

Here is a summary of filter and event specific calls:

Filter text appears in fields or parameters for various other calls (especially chromatogram generation). All of these methods throw an exception, when any device other than MS is selected.

|  |  |  |
| --- | --- | --- |
| Property/Method | Meaning |  |
| string[] GetAutoFilters(); | Gets the filter strings for this file. This analyses all scan types in the file. It may take some time, especially with data dependent files. Filters are grouped, within tolerance (as defined by the MS detector). |  |
| string[][] GetSegmentEventTable(); | Gets the segment event table for the current instrument. This table indicates planned scan types for the MS detector. It is usually created from an instrument method, by the detector. With data dependent or custom scan types, this will not be a complete list of scan types used within the file.  If this object implements the derived IRawDataPlus interface, then this same data can be obtained in object format (instead of string) with the IRawDataPlus property "ScanEvents" |  |
| ReadOnlyCollection<IScanFilter> GetFilters(); | Calculate the filters for this raw file, and return as an array. See also “GetAutoFilters”. This is the same information, in interface form instead of string. |  |
| IScanFilter GetFilterForScanNumber(int scan); | Get the filter (scanning method) for a scan number. |  |
| IScanFilter GetFilterFromString(string filter); | Get a filter interface from a string. Parses the supplied string. If the string is not a valid format, this may return null. |  |
| IScanFilter GetFilterFromString(string filter, int precision); | Get a filter interface from a string, with a given mass precision. If the string is not a valid format, this may return null. |  |
| IFilteredScanIterator GetFilteredScanIterator(IScanFilter filter); | Obtain an interface to iterate over a scans which match a specified filter. The iterator is initialized at "scan 0" such that "GetNext" will return the first matching scan in the file. This is a low level version of GetFilteredScanEnumerator" |  |
| IEnumerable<int> GetFilteredScanEnumerator(IScanFilter filter); | Get a filtered scan enumerator, to obtain the collection of scans matching given filter rules.    “filter” is the filter, which all enumerated scans match. This filter may be created from a string using "GetFilterFromString (string, int)"    This returns an enumerator which can be used to "foreach" over all scans in a file, which match a given filter. Note that each "step" through the enumerator will access further data from the file. To get a complete list of matching scans in one call, the "ToArray()" extension can be called, but this will result in a delay as all scans in the file are analyzed to return this array.  Note that, since this only return the scan numbers (and not the actual scans), collecting this entire list (for example with ToArray) will not consume a significant amount of memory.  For fine grained iterator control, including "back stepping" consider using “GetFilteredScanIterator(IScanFilter)" |  |
| IEnumerable<int> GetFilteredScanEnumeratorOverTime(IScanFilter filter, double startTime, double endTime); | Get a filtered scan enumerator, to obtain the collection of scans matching given filter rules, over a given time range.  See the “**GetFilteredScanEnumerator(IScanFilter filter);**” for details. |  |
| IScanEvents ScanEvents { get; } | Gets the scan events.  This is the set of events which have been programmed in advance of collecting data (based on the MS method). This does not analyze any scan data. |  |
| IScanEvent GetScanEventForScanNumber(int scan); | Gets the scan event details for a scan. Determines how this scan was programmed. |  |
| string GetScanEventStringForScanNumber(int scan); | Gets the scan event as a string for a scan. |  |
| bool TestScan(int scan, string filter); | Test if a scan passes a filter. If all matching scans in a file are required, consider using "GetFilteredScanEnumerator" or "GetFilteredScanEnumeratorOverTime" |  |

#### Extension methods file filtering

Many applications hold “filter” as a text string.

Internally, features like “**GetFilteredScanEnumerator**” use IScanFilter, as this removes the need to “parse the filter from text at each scan”.

Extensions are provided for scenarios which involve filter testing in custom code, to reduce parsing and other overheads.

For example “TestScan” takes a string form of a filter. You would not want to call this for a large number of scans.

The following extension methods are available for filters:

|  |  |  |
| --- | --- | --- |
| Property/Method | Meaning |  |
| public static bool TestScan(this IRawDataPlus data, int scan, IScanFilter filter) | Test if a scan passes a filter. This extension is provided for improved efficiency where the same filter string needs to be used to test multiple scans, without repeating the parsing.  GetFilterFromString(string filter).  Also consider using "GetFilteredScanEnumerator" when processing all scans in a file. |  |
| public static bool TestScan(this IRawDataPlus data, int scan, ScanFilterHelper filterHelper) | Test if a scan passes a filter. This extension is provided for improved efficiency where the same filter string needs to be used to test multiple scans, without repeating the parsing. Consider using one of the overloads of BuildFilterHelper()  Parsing can be done using: GetFilterFromString(string filter.  Also consider using "GetFilteredScanEnumerator" when processing all scans in a file. |  |
| public static ScanFilterHelper BuildFilterHelper(this IRawDataPlus data, string filter) | Constructs an object which has an analysis of the selections being made by a scan filter. Improves efficiency when validating many scans against a filter. |  |
| public static ScanFilterHelper BuildFilterHelper(this IRawDataPlus data, IScanFilter filter) | Constructs an object which has an analysis of the selections being made by a scan filter. Improves efficiency when validating many scans against a filter. |  |

Note that several of these methods use or create “ScanFilterHelper”.

This helper class analyzes rules, which are set in IScanFilter, making a list of filter conditions (by checking the state of each interface member).

For example, for a filter which was parsed from the text “ms”, there is only one rule: All scans must be ms.

This makes the operation of the TestScan extension faster.

Conceptually: If “IScanFilter” is a “list of check boxes” then ScanFilterHelper is contains “A list of which boxes are checked”.

The scan filter helper contains the business logic of “testing if a scan would pass the filter”.

#### Filter codes

Most filter codes consist of one or more letters, with an optional prefix of “!”.

When a filter string is parsed to IScanFilter, most of the properties in the interface will have a value of “Any” (or a member of an enum, which ends in the text “Any”).

“Any” implies that a scan will pass the filter, regardless of whether or not this feature is used.

The enumerated type TriState is used to represent this for many of the filter features, an example being Dependent.

For example:

“ms d” returns scans which are ms, and also dependent. The parsed value of Dependent is TriState.On

“ms !d” returns scans which are ms, but not dependent. . The parsed value of Dependent is TriState.Off

“ms” returns scans which are ms (regardless of the dependent state). The parsed value of Dependent is TriState.Any

To recap:

|  |  |
| --- | --- |
| TriState value | Meaning |
| On | The feature must be used in the scan. |
| Off | The feature must not be used in the scan |
| Any | All scans pass filtering (of this feature), regardless of whether this feature is used or not to generate this scan. |

A number of the codes are “reserved” and may have an instrument specific definition. These are name like “Parameter A”. Refer to MS manual for details.

Some codes may be followed by a numeric value,

For example “sid=<value>” represents strings like “sid=34.6”

Here are the possible codes.

|  |  |
| --- | --- |
| Feature | Values (meaning) |
| Meta Filter | hcd, etd, cid  Note: These filters test if “any MS/MS stage uses these features”. No other filter codes can be entered with a meta filter code. |
| Ionization Mode | EI (electron impact)  CI (chemical ionization)  FAB (fast atom bombardment)  ESI (electrospray)  APCI (atmospheric pressure chemical ionization)  NSI (nanospray)  TSP (thermospray)  FD (field desorption)  MALDI (matrix assisted laser desorption ionization.)  GD (glow discharge) |
| Mass Analyzer | ITMS (ion trap)  TQMS (triple quad)  SQMS (single quad)  TOFM (time of flight)  FTMS (Fourier transform)  Sector (magnetic sector) |
| Sector Scan | BSCAN (magnet scan)  ESCAN (electrostatic scan) |
| Lock | lock, !lock |
| Field Free Region | ffr1, ffr2 |
| Ultra | u, !u |
| Enhanced | E, !E |
| Parameter A | a, !a |
| Parameter B | b, !b |
| Parameter F | f, !f |
| Sps Multi Notch | sps, !sps |
| Parameter R | r, !r |
| Parameter V | v, !v |
| Multi Photon Dissociation | mpd, !mpd |
| Electron Capture Dissociation | ecd, !ecd |
| Photo Ionization | pi, !pi |
| Polarity | +, - |
| Scan Data Type | c (centroid)  p (profile) |
| Corona | corona, !corona |
| Source Fragmentation | sid=<value>, sid, !sid |
| Compensation Voltage | cv=<value>, cv, !cv |
| Data Dependent | d, !d |
| Wideband | w, !w |
| Supplemental Activation | sa, !sa |
| Multi stage Activation | msa, !msa |
| Accurate Mass | AMI (internal)  AME (external)  !AM |
| Turbo Scan | t, !t |
| Scan Mode | full  z  sim  srm  crm  q1ms  q3m |
| multiplex | msx, !msx |
| Detector Values | det, det=<value>, !det |

In addition to these file codes, filter contain “MS/MS reaction data” and “scanned mass ranges”

For MS/MS data, the MS order is specified as ms<n> (omitting “1” for first order) For example, basic ms scans: “ms”, MS/MS data “ms2”, MS/MS/MS data “ms3” etc.

After the MS order, there is a list of precursor masses and reactions.

The general format is <precursor mass>{[@<reaction code>[<reaction Value>]]}

Where “[]” implies optional data, and “{}” implies repeated data.

When reactions are repeated, a space separator is used.

The following shows a scan filter for an ion trap experiment (ITMS), with MS/MS/MS data (ms3).

Precursor mass “262.6000” is fragmented multiple times, using “etd”, “hcd” and “cid” techniques.

See your instrument manual for activation techniques which apply to a particular model of detector.

ITMS + c ESI r d sa Full ms3 262.6000@etd104.31@hcd25.00 377.1985@cid30.00 [98.0000-388.0000]

Note that after the final reaction, a mass range is supplied. For SRM data this may be a list of masses or ranges, for the fragment ions.

When performing filtering, it is valid to supply an MS/MS order only (with no precursor data), but: You cannot supply precursor data without first supplying the MS/MS order.

##### msx filters

MSX is a special case of MS/MS, where multiple precursor masses are activated.

For example, this is a complete filter code for an MS/MS experiment using msx:

FTMS + p ESI Full msx ms2 262.64@hcd35.00 524.27@hcd35.00 1422.00@hcd35.00 [50.00-1470.00]

## Reading logs (status, errors etc.)

### General format of logs

Almost all detectors support Logs.

MS detectors support more logs than others.

Use this table to determine what logs may be requested, per detector type.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Log\Detector | Status | UV | Analog | PDA | MS |  |
| Error | Yes | Yes | Yes | Yes | Yes |  |
| Status | Yes | Yes | Yes | Yes | Yes |  |
| Tune | No | No | No | No | Yes |  |
| Trailer Extra | No | No | No | No | Yes |  |

It is necessary to select an instrument before requesting log data. There are no logs which are “file scope”

Status, Tune, And Trailer logs have the same general format.

See the section “About logs” for some additional information on logs.

These logs contain a header, describing the format of each field. Logs then have a series of fixed size records. Logs format data is returned as an array of “HeaderItem”.

Headers have the following properties.

|  |  |
| --- | --- |
| Property | Meaning |
| string Label | The display label for the field.  For example: If this a temperature, this label may be "Temperature" and the DataType may be "GenericDataTypes.FLOAT" |
| GenericDataTypes DataType | The data type for the field |
| int StringLengthOrPrecision | The precision, if the data type is float or double, or string length of string fields. |
| bool IsScientificNotation | Indicated whether a number should be displayed in scientific notation. |

Strings may be shorter than the indicated length. Length here indicates the maximum possible length of the strings, saved in the file, as the records for logs have a constant size.

Information about logs can be found in the run header, as detailed below

### Error Logs

Error logs have no special formatting, and just consist of time stamped error messages.

|  |  |
| --- | --- |
| Method | Meaning |
| IErrorLogEntry GetErrorLogItem  (int index); | Gets an entry from the instrument error log, using a zero based index. The number of error log entries can be determined by RunHeaderEx.ErrorLogCount |

IErrorLogEntry is defined as follows:

|  |  |
| --- | --- |
| Property | Meaning |
| double RetentionTime | The retention time when the error occurred |
| string Message | The error message |

### Status Logs

Status logs are recorded as time series data. Each record has a retention time, and a log at that time. Because of this, it is possible to plot trends of status log data.

The following properties and methods can be used with status logs.

|  |  |
| --- | --- |
| Method | Meaning |
| HeaderItem[] GetStatusLogHeaderInformation(); | Returns the header information for the current instrument's status log. This defines the format of the log entries. See the section above on “general format of logs” |
| GetStatusLogEntriesCount(); | Returns the number of entries in the current instrument's status log. |
| LogEntry GetStatusLogForRetentionTime(double retentionTime); | Gets the status log record nearest to a retention time. The returned “LogEntry” includes the label/value pairs for this record. |
| StatusLogValues GetStatusLogValues(int statusLogIndex, bool ifFormatted); | Returns the Status log values for the current instrument, for the given status record.  This is most likely for diagnostics or archiving. Applications which need logged data near a scan should use “GetStatusLogForRetentionTime”.  "statusLogIndex" is the (zero based) Index into table of status logs “ifFormatted" is true if data should be formatted as per the data definition (Header Item) for this field (recommended for display). Unformatted values may be returned with default precision (for float or double) Which may be better for graphing or archiving.  Note that this does not return the “labels” for the fields. |
| KeyValuePair<string, int>[] StatusLogPlottableData { get; } | Gets the labels and index positions of the status log items which may be plotted. That is, the numeric items. Labels names are returned by "Key" and the index into the log record is "Value". |
| ISingleValueStatusLog GetStatusLogAtPosition(int position); | Gets the status log data, from all log entries, based on a specific (zero based) position in the log. For example: "position" may be selected from one of the key value pairs returned from StatusLogPlottableData”, in order to create a trend plot of a particular value.  The interface returned has an array of retention times and strings. If the position was selected by using StatusLogPlottableData", then the strings may be converted "ToDouble" to get the set of numeric values to plot. |

### Trailer extra Logs

Mass spectrometers often have custom data logged with each scan.

The format of this data is a generic record, similar to a status log record, where each detector can specify any number of custom fields to be logged.

The raw file format does not determine how many fields are logged, or the format.

This permits new devices to log any additional data they need about a scan.

Fixed format data about a scan (such as retention time) is saved in the “Scan Header”. To distinguish, we called this extra block or variable format data “Scan trailer”

The following methods relate to reading trailer extra data

|  |  |
| --- | --- |
| Method | Meaning |
| HeaderItem[] GetTrailerExtraHeaderInformation(); | Gets the trailer extra header information. This is common across all scan numbers. This defines the format of additional data logged by an MS detector, at each scan. For example, a particular detector may wish to record "analyzer 3 temperature" at each scan, for diagnostic purposes. Since this is not a defined field in "ScanHeader" it would be created as a custom "trailer" field for a given instrument. The field definitions occur only once, and apply to all trailer extra records in the file. In the example given, only the numeric value of "analyzer 3 temperature" would be logged with each scan, without repeating the label.  This defines the format of the log entries. See the section above on “general format of logs” |
| int RunHeaderEx.TrailerExtraCount { get; } | Returns the number of entries in the current instrument's trailer extra log. This will be either 0, or equal to the number of MS scans. |
| LogEntry GetTrailerExtraInformation(int scanNumber); | Gets the array of labels and values for this scan number. The values are formatted as per the header settings. |
| string[] GetTrailerExtraValues(int scanNumber, bool ifFormatted); | Gets the Trailer Extra values for the specified scan number.  “ifFormatted" is true if data should be formatted as per the data definition (Header Item) for this field (recommended for display). Unformatted values may be returned with default precision (for float or double) Which may be better for graphing or archiving.  Note that this does not return the “labels” for the fields. |
| object GetTrailerExtraValue(int scanNumber, int field); | Returns the (unformatted) Trailer Extra value for a specific (zero based) field in the specified scan number.  This offers higher performance, where numeric values are needed, as it avoids translation to and from strings.  The object type depends on the field type, as returned by GetTrailerExtraHeaderInformation.   * Numeric values (where the header for this field returns "True" for IsNumeric) can always be cast up to double. * The integer numeric types SHORT and USHORT are returned as short and ushort. * The integer numeric types LONG and ULONG are returned as int and uint. * All logical values (Yes/No, True/false, On/Off) are returned as "bool", where "true" implies "yes", "true" or "on". * CHAR and UCHAR types are returned as "byte". * String types WCHAR\_STRING and CHAR\_STRING types are returned as "string". |
|  |  |

### Tune Logs

Instruments may log one or more sets of tuning conditions used to collect this data. These follow the same general log format as status logs.

Tune data is currently only supported for MS detectors.

These methods will throw exceptions, if the selected instrument is not “MS”.

|  |  |
| --- | --- |
| Method | Meaning |
| HeaderItem[] GetTuneDataHeaderInformation(); | Return the header information for the current instrument's tune data. This defines the fields used for a record which defines how the instrument was tuned. These items can be paired with the "TuneDataValues" to correctly display each tune record in the file. |
| int GetTuneDataCount(); | Return the number of tune data entries. Each entry describes MS tuning conditions, used to acquire this file. |
| LogEntry GetTuneData(int tuneDataIndex); | Gets a text form of the instrument tuning method, at a given index. The number of available tune methods can be obtained from GetTuneDataCount.  This contains headers and formatted data values. |
| TuneDataValues GetTuneDataValues(int tuneDataIndex, bool ifFormatted); | Return tune data values for the specified index. This contains only the data values, and not the headers. Normally you would set “ifFormatted” to true, to format based on the precision defined in the header. Setting this to false uses default number formatting. This may be better for diagnostic charting, as numbers may have higher precision than the default format. |

## Analysis of data dependent scans

The following method analyses dependent scans.

It steps forwards through subsequent scans in a raw file, looking for scans which have been triggered, based on data found in the selected scan.

This may be used to annotate peaks in a spectrum plot, show which ones have been fragmented, to generate a dependent scan.

See the interface documentation for “IScanDependents” to see the format of the returned information.

/// <summary>

/// Get scan dependents.

/// Returns a list of scans, for which this scan was the parent.

/// </summary>

/// <param name="scanNumber">

/// The scan number.

/// </param>

/// <param name="filterPrecisionDecimals">

/// The filter precision decimals.

/// </param>

/// <returns>

/// Information about how data dependent scanning was performed.

/// </returns>

/// <exception cref="NoSelectedMsDeviceException">Thrown if the selected device is not of type MS</exception>

IScanDependents GetScanDependents(int scanNumber, int filterPrecisionDecimals);

## Averaging Scans

The raw file reader does not perform scan averaging or subtraction directly.

The DLL “ThermoFisher.CommonCore.BackgroundSubtration.dll” provides algorithms for averaging and subtracting scans.

The simplest way to access this is by this table of extension methods:

|  |  |
| --- | --- |
| Name | Description |
| AverageScans(List<Int32>, MassOptions) | Overloaded.  Calculates the average spectra based upon the list supplied. The application should filter the data before making this code, to ensure that the scans are of equivalent format. The result, when the list contains scans of different formats (such as linear trap MS centroid data added to orbitrap MS/MS profile data) is undefined. If the first scan in the list contains "FT Profile", then the FT data profile is averaged for each scan in the list. The combined profile is then centroided. If the first scan is profile data, but not orbitrap data: All scans are summed, starting from the final scan in this list, moving back to the first scan in the list, and the average is then computed. For simple centroid data formats: The scan stats "TIC" value is used to find the "most abundant scan". This scan is then used as the "first scan of the average". Scans are then added to this average, taking scans alternatively before and after the apex, merging data within tolerance.  (Defined by Extensions.) |
| AverageScans(List<ScanStatistics>, MassOptions) | Overloaded.  Calculates the average spectra based upon the list supplied. The application should filter the data before making this code, to ensure that the scans are of equivalent format. The result, when the list contains scans of different formats (such as linear trap MS centroid data added to orbitrap MS/MS profile data) is undefined. If the first scan in the list contains "FT Profile", then the FT data profile is averaged for each scan in the list. The combined profile is then centroided. If the first scan is profile data, but not orbitrap data: All scans are summed, starting from the final scan in this list, moving back to the first scan in the list, and the average is then computed. For simple centroid data formats: The scan stats "TIC" value is used to find the "most abundant scan". This scan is then used as the "first scan of the average". Scans are then added to this average, taking scans alternatively before and after the apex, merging data within tolerance.  (Defined by Extensions.) |
| AverageScansInScanRange(Int32, Int32, String, MassOptions) | Overloaded.  Gets the average scan between the given times.  (Defined by Extensions.) |
| AverageScansInScanRange(Int32, Int32, IScanFilter, MassOptions) | Overloaded.  Gets the average scan between the given times.  (Defined by Extensions.) |
| AverageScansInTimeRange(Double, Double, String, MassOptions) | Overloaded.  Gets the average scan between the given times.  (Defined by Extensions.) |
| AverageScansInTimeRange(Double, Double, IScanFilter, MassOptions) | Overloaded.  Gets the average scan between the given times.  (Defined by Extensions.) |
| SubtractScans | Subtracts the background scan from the foreground scan  (Defined by Extensions.) |

The averaging methods which take filters will eventually call “AverageScans(List<ScanStatistics>, MassOptions)”, or an internal version of the same, which will average the selected set of scan.

See the description of that method for details.

By offering these as extensions, programmers using IRawDataPlus immediately see the available methods (with typical “auto complete” or “auto show member” features).

These methods are based on the interface IScanAveragePlus

As an alternative to using these extensions, you may use a factory to create IScanAveagePlus

IRawDataPlus data ; // obtained form opening a file…

IScanAveragePlus average=ScanAveragerFactory.GetScanAverager(data);

Note that CommonCore also defines an (older) interface IScanAvaerge. This was designed based in IRawData, and will still operate correctly on IRawDataPlus. The “Plus” interface is preferred when using IRawDataPlus.

One difference is: the older “IScanAvaerge” was designed offer averaging code within the legacy C++ file reading technology. IScanAveragePlus no longer offers that feature.

# Opening other files

The file reader an also read data from home page sequence (sld), Xcalibur processing method (pmd) and Xcalibur instrument method (meth) files.

There are no current plans to offer writing to these files formats.

These readers are provided to permit import of data from Xcalibur.

## Reading sequence data ( sld files)

Sequence file are read using the ISequenceFileAccess interface.

This can be obtained from the factory:

ThermoFisher.CommonCore.Data.Business. SequenceFileReaderFactory

Using the “public static ISequenceFileAccess ReadFile(string fileName)” method.

This interface has similar properties as the raw files to access the file header, and any error information (see the interface documentation for details)

There are two properties to obtain sequence data:

|  |  |
| --- | --- |
| Property | Meaning |
| ISequenceInfo Info { get; } | Gets additional information about a sequence |
| List<SampleInformation> Samples { get; } | Gets the set of samples in the sequence |

The “ISequenceInfo” interface includes the following data.

Note: some of the data is display configuration for home page, and is not used for data acquisition, or to perform calculations.

|  |  |
| --- | --- |
| Property | Meaning |
| short[] ColumnWidth | Gets the display width of each sequence column. *Display configuration for home page* |
| short[] TypeToColumnPosition | Gets the column order.  *Display configuration for home page* |
| BracketType Bracket | Gets the sequence bracket type. This determines which groups of samples use the same calibration curve. |
| string[] UserPrivateLabel { get; } | Gets the set of column names for application specific columns |
| string TrayConfiguration { get; } | Gets a description of the autosampler tray |
| string[] UserLabel { get; } | Gets the user configurable column names |

See object help for details of “SampleInformation”. This is mostly text fields, matching the data shown on one row of the sample grid in home page.

## Reading processing methods (pmd files)

### Introduction

The following factory class can be used to read processing method files:

ThermoFisher.CommonCore.Data.Business.ProcessingMethodReaderFactory

The method “IProcessingMethodFileAccess ReadFile(string fileName)” will read all data from a processing method, returning an interface to the objects in memory. The file is not kept open, and so this interface is not disposable.

The interface “IProcessingMethodFileAccess” permits all method settings to be inspected. It is a read-only interface, as this tool is not able to create or modify pmd files.

Like raw files, pmd files have been available for many years, and contain features which may no longer be used in the latest versions of Xcalibur. The organization of the data within the files, and the returned objects may not map exactly to parameters, as seen in any particular application. (This is in contrast to the toolkit XDK, which attempted to map settings to values displayed on certain Xcalibur screens).

This data is designed to easily connect to other classes within common core.

For example: The returned interface to “genesis peak integration settings” from within this object hierarchy can be used to initialize a “Genesis peak integrator” in the common core PeakDetect dll, with no need to translate or scale any of the settings.

For example: There may be values saves as “ratio 0 to 1” in the binary file, and used in that manner with the algorithm code. These are passed through, with no intervening scaling. However, the UI of Xcalibur may scale some stings, preferring to show a “0 to 100%” scale for the “0 to 1” parameter.

CommonCore includes worked examples of how these method parameters can be used to replicate many Xcalibur calculations.

### Interface overview

|  |  |
| --- | --- |
| Property | Meaning |
| General file properties | |
| IFileHeader FileHeader { get; } | Get the file header for the method. Same format as for raw files, see raw file chapter for details. |
| IFileError FileError { get; } | Gets the file error state. Same format as for raw files, see raw file chapter for details. |
| bool IsError { get; } | Gets a value indicating whether the last file operation caused an error. Same format as for raw files, see raw file chapter for details. |
| bool IsOpen { get; } | Gets a value indicating whether a file was successfully opened. Inspect "FileError" when false |
| Method general data | |
| IProcessingMethodOptionsAccess MethodOptions { get; } | These settings apply to all components in the quan section. Some settings affect qual processing. |
| IPeakDisplayOptions PeakDisplayOptions { get; } | Gets additional options about the peak display (peak labels etc). |
| string RawFileName { get; } | Gets the raw file name, which was used to design this method |
| IMassOptionsAccess MassOptions { get; } | Gets the (global) mass tolerance and precision settings for the method |
| ProcessingMethodViewType ViewType { get; } | Gets the "View type" saved in a pmd file. This value is not used in calculations. It is used to configure the display in Xcalibur applications only. Returned for completeness. |
| Reports | |
| IProcessingMethodStandardReportAccess StandardReport { get; } | Gets the "Standard report" settings from a processing method. Many of these settings are “Legacy data.” And not used by Xcalibur. New software will most likely have its own reporting mechanisms, so these settings are simply provided for completeness. |
| ReadOnlyCollection<IXcaliburSampleReportAccess> SampleReports { get; } | Gets the list of reports |
| ReadOnlyCollection<IXcaliburProgramAccess> Programs { get; } | Gets the list of programs |
| ReadOnlyCollection<IXcaliburReportAccess> SummaryReports { get; } | Gets the list of reports |
| Qualitative settings | |
| IQualitativePeakDetectionAccess PeakDetection { get; } | Gets peak detection settings (Qual processing) |
| ISpectrumEnhancementAccess SpectrumEnhancement { get; } | Gets Spectrum Enhancement settings (Qual processing) |
| ILibrarySearchOptionsAccess LibrarySearch { get; } | Gets options for NIST library search |
| ILibrarySearchConstraintsAccess LibrarySearchConstraints { get; } | Gets constraints for NIST library search |
| IPeakPuritySettingsAccess PeakPuritySettings { get; } | Get setting for PDA peak purity |
|  |  |
| Quan Settings | |
|  |  |
| ReadOnlyCollection<IXcaliburComponentAccess> Components { get; } | Gets the list of compounds. This includes all integration, calibration and other settings which are specific to each component. |

For details of each specific interface, refer to the interface help.

## Reading data from instrument methods (meth files)

The instrument method file reader can read Xcalibur instrument methods, including Instrument methods exported from a raw file.

See notes about exporting methods from raw files, as sometimes the export may not be possible.

Unlike the Xcalibur “instrument setup” window, the instrument method reader does not examine which instruments are configured. You can view data from all instruments in the file.

The following factory can open instrument methods

ThermoFisher.CommonCore.Data.Business.InstrumentMethodReaderFactory

Open a file using “public static IInstrumentMethodFileAccess ReadFile(string fileName)”.

Note that this is not “disposable’ as the file is immediately closed after reading the information, into objects in memory.

The returned interface cannot be used to modify method data.

### Interface Overview

|  |  |
| --- | --- |
| Property | Meaning |
| General file properties | |
| IFileHeader FileHeader { get; } | Get the file header for the method. Same format as for raw files, see raw file chapter for details. |
| IFileError FileError { get; } | Gets the file error state. Same format as for raw files, see raw file chapter for details. |
| bool IsError { get; } | Gets a value indicating whether the last file operation caused an error. Same format as for raw files, see raw file chapter for details. |
| bool IsOpen { get; } | Gets a value indicating whether a file was successfully opened. Inspect "FileError" when false |
| Method data | |
| ReadOnlyDictionary<string, IInstumentMethodDataAccess> Devices { get; } | Gets the data for of all devices in this method. Keys are the registered device names. A method contains only the "registered device name" which may not be the same as the "device display name" (product name). Instrument methods do not contain device product names. |

Data for each device is returned using the interface IInstumentMethodDataAccess

|  |  |
| --- | --- |
| Property | Meaning |
| string MethodText { get; } | Gets the plain text form of an instrument method. |
| IReadOnlyDictionary<string, byte[]> StreamBytes { get; } | Gets all streams for this instrument, apart from the "Text" stream. Typically an instrument has a stream called "Data" containing the method in binary or XML. Other streams (private to the instrument) may also be created. |

Each device contains a text steam which can be used to display the methods. This is a single string, which can be split into multiple likes using return.

For example, this algorithm could be used to display the method text in a windows forms data grid:

string methodText = deviceData.MethodText;

if (!string.IsNullOrEmpty(methodText))

{

string[] splitMethod = methodText.Split(new[] { "\n" }, StringSplitOptions.None);

foreach (string s in splitMethod)

{

instrumentDataGridView.Rows.Add();

instrumentDataGridView.Rows[row++].Cells[0].Value = s;

}

}

The “Stream bytes” are unknown contents, private to the specific device driver. Some may be XML text. However, this interface does not attempt to decode the information.

For example: Suppose a device had used an XmlWriter to encode a stream “data”. Code similar to this could decode it.

// look for data stream

var streams = deviceData.StreamBytes;

byte[] dataStream;

bool foundData = streams.TryGetValue("data", out dataStream);

if (foundData)

{

// Turn the byte array into a stream

using (MemoryStream ms = new MemoryStream(dataStream))

{

// Create an XML reader for the stream

XmlReader reader;

reader = XmlReader.Create(ms);

// Decode the XML

XmlDocument doc = new XmlDocument();

doc.Load(reader);

// Process the document…

}

}