DewesoftX DCOM

Reference manual for the DewesoftX COM/DCOM automation interface

VERSION 2026.3 · 23 SEPTEMBER 2026

Contents

DewesoftX DCOM Interface

General

DewesoftX exposes its functionality through a **COM/DCOM automation interface**. A program that connects to it can load a setup, start and stop a measurement, read channel data, and export results — on the local machine or over the network.

`IApp` is the entry point. Everything else — channels, data, storing, exports, events — is reached from there.

Guides

Connecting — obtaining `IApp`, launching versus attaching, and the initialisation DewesoftX requires before any other call.

Receiving Events — being called back when storing starts or a trigger fires, instead of polling.

Running a Measurement — modes, starting acquisition versus starting storing, and reporting the current state.

Reading Channel Data — finding a channel and getting samples out of it.

Files and Export — loading a stored measurement, scoping a region, and exporting it.

Concepts

How DewesoftX stores and times its data. These pages explain behaviour that the individual interfaces only hint at.

Data Buffers — the direct and intermediate buffers, and how to address a sample in a ring that wraps.

Channel Kinds — synchronous, asynchronous and single-value channels, arrays, text and control channels.

Timing and Rates — the three sample rates, deriving timestamps, and why a live math channel can disagree with its input.

Reference

Every interface of the COM library is listed alphabetically in the sidebar, with `IApp` as the entry point.

Signatures, parameter names, types and enumeration values are taken from the DewesoftX type library, so they describe exactly what the installed build exposes. Interfaces that an automation client cannot call say so at the top of their page.

Languages

Every example is given in both **Python** and **C#**. Any language with COM support can drive the interface — the sequence of calls is the same, only the binding syntax differs.

The C# examples bind late through `dynamic`, so they need no interop assembly. Adding a COM reference to the **DEWESoft Library** instead gives you typed interfaces and completion.

Support

For additional support:

Visit our Support Portal

Contact our Technical Support

Connecting to DewesoftX

A client connects to the COM automation interface and obtains `IApp` — the entry point to channels, data, storing, exports and events — then drives the application from there.

Connecting

PYTHON

Install pywin32:

```
pip install pywin32
```

```
import win32com.client

# Launches DewesoftX, or connects to it if already running.
app = win32com.client.Dispatch("DEWESoft.App")

app.Init()          # required before any other member
app.Visible = True  # a client-launched instance starts hidden

print(app.Version)                # 2026.4 (DEV-260819.0840) (64-bit)
print(app.Data.AllChannels.Count)  # channels in the loaded setup
```

C#

No interop assembly is required if you bind late through `dynamic`:

```
using System;

var type = Type.GetTypeFromProgID("DEWESoft.App")
           ?? throw new InvalidOperationException("DewesoftX is not registered.");

dynamic app = Activator.CreateInstance(type)!;

app.Init();          // required before any other member
app.Visible = true;  // a client-launched instance starts hidden

Console.WriteLine(app.Version);                // 2026.4 (DEV-260819.0840) (64-bit)
Console.WriteLine(app.Data.AllChannels.Count);  // channels in the loaded setup
```

For compile-time checking and IntelliSense, add a COM reference to the **DEWESoft Library** type library instead and use the `IApp` interface directly.

> **Call `Init()` first**
>
> Call `Init()` before any other member. Reads that come before it fail, and can leave the instance unusable.
>
> It loads the hardware configuration and takes several seconds — 6 to 8 s on a typical setup. Allow for that in any connection timeout.

`Visible` controls whether the DewesoftX window is shown, and `Enabled` whether its user interface accepts input. `ActualRunMode` reports `0` before `Init()` and `1` afterwards, which confirms that initialisation succeeded.

An unattended script should also set `SuppressMessages` immediately after `Init()`. Without it, the first dialog DewesoftX shows stalls the script — see unattended operation.

Attaching to a running instance

The example above **launches** DewesoftX if it is not already running. To work only with an instance the user already has open, attach instead:

PYTHON

```python
import pywintypes
import win32com.client

try:
    app = win32com.client.GetActiveObject("DEWESoft.App")
except pywintypes.com_error:
    raise SystemExit("DewesoftX is not running.")
```

C#

```csharp
using System.Runtime.InteropServices;

dynamic app = Marshal.GetActiveObject("DEWESoft.App");
```

	Launch or reuse	Attach only
Python	`Dispatch`	`GetActiveObject`
C#	`Activator.CreateInstance`	`Marshal.GetActiveObject`
Not running	Starts DewesoftX	Fails
Already running	Connects to it	Connects to it

Attach where the user is expected to have DewesoftX open and their session must not be disturbed. Launch for unattended scripts that own the whole measurement.

A client-launched DewesoftX starts up differently from one the user opened:

	Started by the user	Started by a client
Main window	Shown	**Hidden**
Window size	Restored from settings	640 × 480
Main toolbar	Shown	Not initialised

Releasing the connection

`IApp` has **no method to close DewesoftX**. `Stop` and `ManualStop` end the measurement and `StopStoring` ends storing, but none of them shuts the application down. Releasing the reference does:

PYTHON

```python
del app
```

C#

```csharp
Marshal.FinalReleaseComObject(app);
```

Started by	On release
your client	**The process exits**, a second or two later
the user	Stays open, with their session untouched

So a client that launched DewesoftX needs no shutdown call, and one holding a reference it no longer uses holds a DewesoftX process with it.

Receiving events

DewesoftX can call back into a client when storing starts, a trigger fires or an alarm changes state. See Receiving Events for the full list.

One detail affects how you connect: **in Python the event sink must be supplied when the object is created.** It cannot be added to an object already obtained with `Dispatch`.

PYTHON

```python
app = win32com.client.DispatchWithEvents("DEWESoft.App", MyEventHandler)
```

C#

```csharp
var app = new DEWESoft.App();   // COM reference to the DEWESoft Library
app.OnTrigger += OnTrigger;     // events can be subscribed at any time
```

Connecting to a remote computer

DewesoftX can also be created on another machine over DCOM by passing that machine's name:

PYTHON

```python
import win32com.client

app = win32com.client.DispatchEx("DEWESoft.App", machine="MEASUREMENT-PC")
```

C#

```csharp
var type = Type.GetTypeFromProgID("DEWESoft.App", "MEASUREMENT-PC");
dynamic app = Activator.CreateInstance(type!)!;
```

Remote activation requires DCOM to be configured on the target machine — launch and access permissions for the calling account, and the relevant firewall rules. Configure this with `dcomcnfg.exe` under **Component Services → Computers → My Computer → DCOM Config → App Object**.

Server reference

DewesoftX registers its COM server automatically when it starts, so a normal installation needs no extra step.

Property	Value
ProgID	`DEWESoft.App`
CLSID	`{58A7D77D-64F1-4DB8-A6CB-F2B4CAB84056}`
Server type	`LocalServer32` — `DEWEsoft.exe`
Type library	`{96D9B72C-93F3-4D0C-90BB-1466845156EF}` version `630.7`
Default interface	`IApp`
Event interface	`IAppEvents`

Registration records the full path of the `DEWEsoft.exe` that performed it, so if several builds are installed, the last one started is the one a client will launch. To check which:

```powershell
Get-ItemProperty 'Registry::HKEY_CLASSES_ROOT\CLSID\{58A7D77D-64F1-4DB8-A6CB-F2B4CAB84056}\LocalServer32'
```

Three consequences of DewesoftX running as its own application:

Each call to DewesoftX carries overhead. Reading 1000 samples one value at a time is far slower than one call that returns all 1000. Use the methods that return a block.

A 32-bit client can drive 64-bit DewesoftX, and the reverse.

A second client connects to the running DewesoftX rather than starting another copy.

Registering the server writes to `HKEY_CLASSES_ROOT` and may require elevation depending on the machine's User Account Control settings.

Receiving Events

General

`IAppEvents` is the interface DewesoftX raises events on. Instead of polling `IApp` in a loop, a client can be called back when storing starts, a trigger fires or an alarm changes state.

The handler is attached as part of creating the connection — see Connecting, since in Python the sink must be supplied when the object is created.

Events

Event	Parameters	Raised when
`OnGetData`	—	A new block of data is available. **Not delivered — see below.**
`OnStartStoring`	—	Storing has started.
`OnStopStoring`	—	Storing has stopped.
`OnTrigger`	`Mid: Integer`, `Dir: Integer`, `Time: Double`	A trigger condition fired.
`OnTriggerStop`	`Mid: Integer`, `Dir: Integer`, `Time: Double`	A stop-trigger condition fired.
`OnAlarm`	`CondIndex: Integer`, `Status: Boolean`	An alarm condition changed state.
`OnDataLost`	—	Acquired data could not be processed in time. **Not delivered — see below.**
`OnException`	`Str: String`	An error occurred inside DewesoftX.
`OnProgress`	`PercentDone: Integer`	A long-running operation reports progress.
`OnMessageBox`	`Tex: String`, `Caption: String`, `MsgType: Integer`, `Result: Integer` (out)	DewesoftX would show a message box.
`OnGetTime`	`TimeLo: Integer`, `TimeHi: Integer` (both in/out)	DewesoftX asks the client to supply the current time. **Not delivered — see below.**
`OnEvent`	`Reason: EventReason`, `Param: Variant`	Generic notification, multiplexed over `EventReason`.
`OnExit`	—	DewesoftX is shutting down.

`Mid` and `Dir` together identify a sample position: `Mid` counts complete data blocks and `Dir` counts remaining samples within the current block. The same pair appears throughout the data interface, for example on `IData.CurrentPos` and `IData.EndStamp`.

The first parameter of `OnMessageBox` is named `Tex`, not `Text`.

Three events are not delivered

`OnGetData`, `OnGetTime` **and** `OnDataLost` never reach a client. Delivery fails with *"Interface not supported"*, and DewesoftX may show a trace to that effect. Attaching a sink is enough to produce it; the handler bodies make no difference. Their `OnEvent` equivalents — `erGetData` and `erDataLost` — do not arrive either.

All other events are delivered normally.

Instead of `OnGetData`, poll `IChannelConnection.NumValues` on a timer of your own, and drive the run from `OnStartStoring`, `OnTrigger` and `OnStopStoring`. Instead of `OnDataLost`, read `DataLost` at the end of a run.

Handling events

PYTHON

```python
import pythoncom
import win32com.client


class DewesoftEvents:
    def OnStartStoring(self):
        print("storing started")

    def OnTrigger(self, mid, dir_, time):
        print(f"trigger at {time:.3f} s")

    def OnStopStoring(self):
        print("storing stopped")

    def OnExit(self):
        print("DewesoftX is closing")


app = win32com.client.DispatchWithEvents("DEWESoft.App", DewesoftEvents)
app.Init()

while True:
    pythoncom.PumpWaitingMessages()
```

Only the handlers you define are called; there is no need to implement all thirteen.

C#

Add a COM reference to the **DEWESoft Library**, then subscribe to the generated event members:

```csharp
var app = new DEWESoft.App();

app.OnStartStoring += () => Console.WriteLine("storing started");
app.OnTrigger += (mid, dir, time) => Console.WriteLine($"trigger at {time:F3} s");
app.OnStopStoring += () => Console.WriteLine("storing stopped");
app.OnExit += () => Console.WriteLine("DewesoftX is closing");

app.Init();
```

The client must pump Windows messages

Events are delivered on the thread that created the connection, and that thread has to pump Windows messages for them to arrive. A client that blocks in a tight loop without pumping receives nothing, and nothing reports it.

In Python, call `pythoncom.PumpWaitingMessages()`. A WinForms or WPF application already has a message loop and needs nothing extra.

`OnEvent` duplicates the individual events

`OnEvent` reports the same notifications a second time, selected by its `Reason` parameter:

EventReason	Value	Equivalent event
`erGetData`	`0`	`OnGetData` — *not delivered*
`erStartStoring`	`1`	`OnStartStoring`
`erStopStoring`	`2`	`OnStopStoring`
`erTrigger`	`3`	`OnTrigger`
`erException`	`4`	`OnException`
`erTriggerStop`	`5`	`OnTriggerStop`
`erAlarm`	`6`	`OnAlarm`
`erExit`	`7`	`OnExit`
`erDataLost`	`8`	`OnDataLost` — *not delivered*
`erMessageBox`	`9`	`OnMessageBox`
`erProgress`	`10`	`OnProgress`
`erStopAcquisition`	`11`	*(no dedicated event)*

Handle either the individual events or `OnEvent`, not both: both fire for the same notification, so handling both processes it twice. `erStopAcquisition` has no dedicated event, so a client that needs it must use `OnEvent`.

Unattended operation

DewesoftX shows modal dialogs in a number of ordinary situations — an export called in the wrong mode, a setup warning, a missing file. Each one waits for a click that an unattended client will never make, and the script hangs behind it.

Set `SuppressMessages` to withhold them:

PYTHON

```python
app.Init()
app.SuppressMessages = True     # dialogs are withheld from here on
```

C#

```csharp
app.Init();
app.SuppressMessages = true;    // dialogs are withheld from here on
```

> **`OnMessageBox` does not suppress the dialog**
>
> The event reports that a dialog is about to appear, and it is useful for logging what DewesoftX was going to ask. It does not answer the dialog: the value a handler writes into `Result` is overwritten by the real dialog's return value, and the dialog is still shown unless `SuppressMessages` is set.
>
> Handle `OnMessageBox` to record what was asked; set `SuppressMessages` to stop the asking.

PYTHON

```python
class DewesoftEvents:
    def OnMessageBox(self, tex, caption, msg_type, result):
        print(f"DewesoftX asked: {caption} — {tex}")

    def OnException(self, str_):
        print(f"DewesoftX error: {str_}")
```

C#

```csharp
app.OnMessageBox += (string tex, string caption, int msgType, ref int result) =>
    Console.WriteLine($"DewesoftX asked: {caption} — {tex}");

app.OnException += (string s) => Console.WriteLine($"DewesoftX error: {s}");
```

An unattended script should set `SuppressMessages` and handle `OnException`. Between them they cover the dialogs DewesoftX raises deliberately and the errors it does not expect.

Running a Measurement

General

Driving a measurement means moving DewesoftX through modes and then starting two separate things: acquisition and storing.

Acquiring means samples are flowing and displays are live.

Storing means those samples are being written to a data file.

DewesoftX can acquire without storing. It cannot store without acquiring. A script that starts acquisition but never calls `StartStoring` runs to completion and writes no file.

Modes

`ActualRunMode` reports which mode the application is in: `0` none, `1` measure, `2` analysis. It reads `0` until `Init` has finished.

PYTHON

```python
app.Measure()      # measure mode
app.SetupScreen()  # ...and into Config
app.Start()        # ...and out to Live view, acquiring
app.Analyze()      # analysis mode, for stored files
```

C#

```csharp
app.Measure();      // measure mode
app.SetupScreen();  // ...and into Config
app.Start();        // ...and out to Live view, acquiring
app.Analyze();      // analysis mode, for stored files
```

`Start` leaves Config as a side effect of acquiring, which is how a client normally gets to Live view. There is also `GoToInstruments`, which moves there without starting anything.

Many members require a particular mode and say so on their reference page. `SetupScreen` needs measure mode already; the export methods need analysis mode with a file open. A call made in the wrong mode raises rather than being ignored.

Configuring before you start

Load a setup rather than configuring channels call by call.

PYTHON

```python
app.LoadSetup(r"C:\setups\vibration.dxs")
app.MeasureSampleRate = 5000
```

C#

```
app.LoadSetup(@"C:\setups\vibration.dxs");
app.MeasureSampleRate = 5000;
```

`MeasureSampleRate` is one of three sample rates, and not the one `IData.SampleRate` reports while DewesoftX is on the Config tab.

Where the file the measurement writes to matters, set it through `IFileNameSettings` before starting.

Starting and stopping

PYTHON

```python
if not app.Start():                     # switches to measure mode and acquires
    raise RuntimeError("acquisition did not start")

app.StartStoring("run1.dxd")            # begin writing to this file
# ... let it run ...
app.StopStoring()
app.Stop()                              # stops storing and acquisition
```

C#

```csharp
if (!(bool)app.Start())                 // switches to measure mode and acquires
    throw new Exception("acquisition did not start");

app.StartStoring("run1.dxd");           // begin writing to this file
// ... let it run ...
app.StopStoring();
app.Stop();                             // stops storing and acquisition
```

`Start` returns a `Boolean`. Check it: it is the only one of these calls that reports whether it succeeded.

`StartStoring` requires the file name. Calling it with no argument fails with *"Invalid number of parameters"*. A bare name lands in the data directory, and `UsedDatafile` then reports the full path being written.

`Stop` stops both, so `StopStoring` before it is optional. Call it where the file should be closed at a known point rather than as a side effect.

Acquisition state

Member	`True` when
`Acquiring`	Data is being acquired.
`IsAcqRunning`	Acquisition is running, including in Config and Design.

Use `Acquiring` to test whether a measurement is running. `IsAcqRunning` is also `True` on the Config tab, where live values are shown but nothing is being measured.

Neither reports whether data is being *written*. Track that from your own `StartStoring` and `StopStoring` calls, or from `OnStartStoring` and `OnStopStoring`.

`DataLost` is `True` if samples were dropped. Read it at the end of a run: a file with a gap in it should be noticed before the measurement is dismantled.

Triggered storing

With a trigger configured, `StartStoring` *arms* it rather than starting to write. Storing begins when the trigger fires, which `OnTrigger` reports.

`ManualStart` fires the trigger directly. It requires measure mode with the trigger already armed.

Configure triggers through `ITrigger` and `ITriggerCondition`. The setup dialog is modal and hangs an unattended script.

Pausing rather than stopping

`PauseStoring` and `ResumeStoring` suspend writing without ending the file, so a run produces one file with a gap rather than two files. Acquisition continues throughout.

`SetFreezeMode` freezes the *display* only. Acquisition and storing carry on, which makes it useful for letting an operator read a screen without interrupting the measurement, and unsuitable as a way to pause either.

A complete run

PYTHON

```python
import win32com.client, time

app = win32com.client.dynamic.Dispatch("DEWESoft.App")
app.Init()
app.SuppressMessages = True

app.LoadSetup(r"C:\setups\vibration.dxs")
app.Measure()

if not app.Start():
    raise RuntimeError("acquisition did not start")
app.StartStoring("run1.dxd")

time.sleep(30)

app.StopStoring()
app.Stop()

if app.DataLost:
    print("warning: samples were dropped")
print("wrote", app.UsedDatafile)
```

C#

```
dynamic app = Activator.CreateInstance(Type.GetTypeFromProgID("DEWESoft.App"));
app.Init();
app.SuppressMessages = true;

app.LoadSetup(@"C:\setups\vibration.dxs");
app.Measure();

if (!(bool)app.Start())
    throw new Exception("acquisition did not start");
app.StartStoring("run1.dxd");

Thread.Sleep(30000);

app.StopStoring();
app.Stop();

if ((bool)app.DataLost)
    Console.WriteLine("warning: samples were dropped");
Console.WriteLine($"wrote {app.UsedDatafile}");
```

A fixed sleep suits a run of known length. For anything else, stop on the condition that matters — a trigger, a channel value, a sample count — using events rather than a timer.

Reading Channel Data

General

There is more than one way to get samples out of a channel. They differ in how much they fetch per call, whether reading consumes what it read, and which channels they work on.

What you want	Use	Notes
Stream every sample exactly once	a connection	Tracks a read position, so nothing is read twice or missed.
Everything in the buffer now	`GetScaledData`	One call for the whole buffer. The fastest way to move a lot of samples.
A window out of the buffer	`GetScaledDataEx`	The same, bounded by start and count.
Control of exactly which samples you take	indexed access	You address the ring yourself, one sample per call.
The current value	`SingleValue`	One property, no setup.
The value from a moment ago	`GetValueAtRelTimeDouble`	Counts back from now, and can interpolate. Returns unscaled values.
An overview of a long measurement	reduced blocks	Minimum, maximum, average and RMS per block, without fetching every sample.

Each of these is a call into DewesoftX, so **fetch in blocks**. One call for a thousand samples is far faster than a thousand calls.

For the buffer geometry behind these paths, see Data Buffers — the ring, what happens when it wraps, and how to address a sample by age. Channel Kinds covers which paths a given channel supports.

Finding the channel

PYTHON

```python
data = app.Data

ch = data.FindChannel("AI 1")                    # by name
ch = data.FindChannelByIndexEx([0, 1, 0])        # by index

for i in range(data.UsedChannels.Count):         # only enabled channels
    print(data.UsedChannels.Item(i).Name)
```

C#

```
dynamic data = app.Data;

dynamic ch = data.FindChannel("AI 1");                    // by name
ch = data.FindChannelByIndexEx(new int[] { 0, 1, 0 });    // by index

for (int i = 0; i < (int)data.UsedChannels.Count; i++)    // only enabled channels
    Console.WriteLine(data.UsedChannels.Item[i].Name);
```

Both lookups return nothing when nothing matches, so test the result.

Channel names are not unique. Several channels can share a name, and `FindChannel` returns the first match. The comparison is case-**insensitive**, and it matches either the short name or the long `Group/Name` form, so `"ai 1"` and `"AI 1/AI 1"` both find `"AI 1"`. Where the exact channel matters, use `FindChannelByIndexEx`: an index is unique.

Looking a channel up by index

The array is laid out like this:

Element	Value
`0`	always `0`
`1`	the group ID — `1` for analog input
`2` onwards	the channel's position, one element per remaining level

So `[0, 1, 0]` is the first analog input and `[0, 1, 1]` the second. An array of fewer than three elements never matches anything.

`IChannel.IndexEx` returns exactly the array `FindChannelByIndexEx` takes, so an index already in hand round-trips without being built by hand:

PYTHON

```
ix = ch.IndexEx                          # e.g. (0, 1, 0)
same = data.FindChannelByIndexEx(ix)     # the same channel again
```

C#

```
dynamic ix = ch.IndexEx;                 // e.g. (0, 1, 0)
dynamic same = data.FindChannelByIndexEx(ix);
```

Store `IndexEx` to identify a channel across runs. It survives a rename, where a name does not.

Use `IndexEx`, not `Index`

The older `Index` property returns a record rather than an array, and its `Index1` field is the **second** array element. The leading `0` is missing, so passing those fields to `FindChannelByIndexEx` matches nothing.

The record-based `FindChannelByIndex` is not an alternative: a late-bound client can receive a record but not pass one back in, and the call fails with *"Bad variable type"*.

`UsedChannels` holds the enabled channels; `AllChannels` holds every channel the hardware and mathematics define, enabled or not. A channel that is not enabled produces no data.

Streaming with a connection

A connection keeps its own read position, so nothing is read twice and several readers on one channel do not interfere.

PYTHON

```python
conn = ch.CreateConnection()
conn.AType = 3          # only data that has arrived since the last read
conn.Start()

n = conn.NumValues
if n > 0:
    values = conn.GetDataValues(n)
    for v in values:
        print(v)
```

C#

```csharp
dynamic conn = ch.CreateConnection();
conn.AType = 3;         // only data that has arrived since the last read
conn.Start();

int n = (int)conn.NumValues;
if (n > 0)
{
    dynamic values = conn.GetDataValues(n);
    for (int i = 0; i < n; i++)
        Console.WriteLine(values[i]);
}
```

`GetTSValues(n)` reads the matching timestamps, and works **only on an asynchronous channel**. A synchronous channel stores no timestamps, and asking one for them faults inside DewesoftX rather than returning an error. Derive the time from the sample rate instead, and check `Async` where the channel kind is not known in advance.

`AType` selects what a read returns:

AType	Reads	Use it when
`0`	the most recent samples, afresh each time	You want a snapshot. Repeating the read returns the same samples again.
`1`	overlapping blocks, stepping by `Overlap`	You process in windows that need to overlap.
`2`	data around triggers	You care about what happened near a trigger.
`3`	only what has arrived since the last read	You are logging every sample exactly once.

Nothing accumulates until `Start` has been called.

Poll `NumValues` on a timer of your own to decide when to read. There is no event for it: `OnGetData` is not delivered, so it cannot drive a read loop.

`GetDataBlocks` and `GetTSBlocks` do the same in units of `BlockSize` samples. Use values *or* blocks on one connection, not both.

Reading the buffer in bulk

Straight off the channel, with no connection to set up. This is the quickest way to move a lot of samples.

PYTHON

```python
values = ch.GetScaledData()             # everything available
window = ch.GetScaledDataEx(0, 1000)    # 1000 samples from ring index 0
raw = ch.GetUnscaledData()              # as acquired, before scaling
```

C#

```csharp
dynamic values = ch.GetScaledData();            // everything available
dynamic window = ch.GetScaledDataEx(0, 1000);   // 1000 samples from index 0
dynamic raw = ch.GetUnscaledData();             // as acquired, before scaling
```

Scaled values are in the channel's physical unit; unscaled values are the numbers as acquired from the hardware. `GetScaledData` returns `Single`; use `GetScaledDataDoubleEx` where that is not enough precision.

Use the scaled forms unless the raw numbers are what you want. `Scale` and `Offset` do not generally convert one to the other: on a channel scaled by its amplifier range they read `1.0` and `0.0` while the scaled and unscaled values differ by a factor of thousands.

The unbounded read fails on array channels

On a channel whose samples are arrays — an FFT, a matrix — `GetScaledData` with no arguments fails. Ask for a window instead: `GetScaledDataEx(start, count)` works on those channels, as do indexed access and a connection.

`Count` is a number of **samples**, so one spectrum is `GetScaledDataEx(0, 1)`. The result is flat and `Count` × `ArraySize` elements long — a 100-element matrix channel asked for 5 samples returns 500 values, in sample order. Slice it in `ArraySize` chunks to recover the individual arrays.

Asking for more samples than `DBDataSize` reports does not fail. The elements past the end are whatever was in memory. Read `DBDataSize` first and clamp the count to it.

Indexed access to the ring

The direct buffer is a ring. Three properties describe it, and reading it correctly takes all three:

Property	Meaning
`DBBufSize`	total capacity
`DBDataSize`	how many samples are valid
`DBPos`	where the next sample will be written

Once `DBDataSize` reaches `DBBufSize` the ring has wrapped and the oldest samples are being overwritten.

PYTHON

```python
size, pos, cap = ch.DBDataSize, ch.DBPos, ch.DBBufSize

newest = ch.DBValues((pos - 1) % cap)          # most recent sample
precise = ch.DBValuesDouble((pos - 1) % cap)   # ...in double precision
```

C#

```csharp
int size = ch.DBDataSize, pos = ch.DBPos, cap = ch.DBBufSize;

float newest = ch.DBValues[(pos - 1 + cap) % cap];           // most recent
double precise = ch.DBValuesDouble[(pos - 1 + cap) % cap];   // double precision
```

This path costs a call per sample. Use it where exactly which samples you take matters, and read in bulk otherwise.

`DBTimeStamp(index)` gives the matching timestamp, on asynchronous channels only. On a synchronous channel it fails, since the sample time is derived from the rate rather than stored.

The current value

The simplest read available, and it works on any channel, not only ones flagged as single-value.

PYTHON

```python
print(ch.SingleValue)     # current numeric value
print(ch.Text)            # current value as text, for a string channel
```

C#

```csharp
Console.WriteLine(ch.SingleValue);   // current numeric value
Console.WriteLine(ch.Text);          // as text, for a string channel
```

Use it for a dashboard or a limit check, where the current reading is all that matters.

A value from a moment ago

These reach backwards from the present rather than addressing a buffer index:

PYTHON

```python
value, seek = ch.GetValueAtRelTimeDouble(
    1.25,      # seconds BACK from now; 0.0 is the present
    0,         # in/out seek position — pass it back on the next call
    True,      # interpolate between samples
    0,         # array index, for an array channel
    0)         # complex presentation

value, seek = ch.GetValueAtAbsPos(-1, 0, False)   # newest sample
```

C#

```csharp
int seek = 0;
double value = ch.GetValueAtRelTimeDouble(
    1.25,      // seconds BACK from now; 0.0 is the present
    ref seek,  // in/out seek position — pass it back on the next call
    true,      // interpolate between samples
    0,         // array index, for an array channel
    0);        // complex presentation

int seek2 = 0;
float v2 = ch.GetValueAtAbsPos(-1, ref seek2, false);   // newest sample
```

These hide the difference between synchronous and asynchronous channels: you name a moment and get a value. `SeekPos` is in/out — keep the value it returns and pass it back on the next call, and repeated reads walking through the data avoid re-seeking from the start each time.

Two things differ from every other read path

The time and the position are relative to now, counting backwards. `GetValueAtRelTimeDouble(0.0, …)` is the present moment and `1.25` is a second and a quarter ago, not an absolute time from the start of the measurement. `GetValueAtAbsPos` counts samples back the same way, so the newest sample is `-1`; `0` and any positive number are in the future and return `0.0`.

They return unscaled values. Every other path on this page returns the physical value; these return the raw buffer contents, so a channel reading `1.16 V` returns something like `3804`. Use `SingleValue`, `DBValues` or `GetScaledDataEx` for the value in the channel's unit.

An overview without every sample

Above the direct buffer sit up to six levels of *reduced* blocks, each holding the minimum, maximum, average and RMS over a block of the level below. Reading these gives the shape of a long measurement without fetching every sample.

PYTHON

```python
print(ch.FirstIBLevel, ch.IBDataSize, ch.IBPos)

mn, mx, ave, rms = ch.GetIBValues(0)     # as four values
rec = ch.IBValues(0)                     # ...or as one structure
print(rec.Min, rec.Max, rec.Ave, rec.Rms)
```

C#

```csharp
Console.WriteLine($"{ch.FirstIBLevel} {ch.IBDataSize} {ch.IBPos}");

float mn, mx, ave, rms;
ch.GetIBValues(0, out mn, out mx, out ave, out rms);   // as four values
dynamic rec = ch.IBValues[0];                          // ...or as one structure
Console.WriteLine($"{rec.Min} {rec.Max} {rec.Ave} {rec.Rms}");
```

These hold no individual samples and are always synchronous, so they carry no timestamps. `FirstIBLevel` may be above `0` — a slow asynchronous channel starts higher — so do not assume level `0` exists.

Pitfalls

`NumValues` **is** `-1` **when there is nothing to read**, not `0`. That is the state before acquisition has produced anything. Test `> 0`: a bare truthiness test treats `-1` as "there is data", and the read that follows asks for a negative count.

Data and timestamps advance separately on a connection. It keeps two positions, which is what allows a run of samples and then its times to line up. Ask for 100 values and 50 timestamps, though, and every later pair is mismatched, with nothing reporting it. Read both with the same count, every time.

Timestamps exist only on asynchronous channels. On a synchronous channel `GetTSData`, `DBTimeStamp` and a connection's `GetTSValues` all fault inside DewesoftX rather than returning an error. Derive the time from the sample rate instead, or check `Async` before asking.

An empty buffer is not handled for you. Check `DBDataSize` before reading anything. With no samples in it, `GetTSData` fails outright, while `DBValues` and `DBTimeStamp` read past the end of the buffer and return whatever was in memory — zeroes, uninitialised values, or an access violation. Nothing distinguishes that from a real reading.

Channels that are not a stream of numbers

Test	Shape	How to read it
`ch.ArrayInfo` is nothing	one value per sample	Any of the paths above.
`ch.ArrayInfo` is an object	an array per sample	Anything except the unbounded `GetScaledData()`. Axes come from `IArrayInfo`, and `ArraySize` gives the array length.
`ch.Async` is `True`	samples carry their own timestamps	Read the timestamps; they are not derived from the rate.
`ch.Text` is meaningful	a string channel	Read `Text` rather than a numeric path.

`ChannelType` reports which subsystem produced a channel, and `DataType` how its samples are stored.

Reading from a stored file

Everything above works on a loaded file. Load it, scope the region, then read:

PYTHON

```python
app.LoadFile(r"C:\data\run1.dxd")
app.Data.SelectDataRegion(10.0, 20.0)      # seconds

ch = app.Data.FindChannel("AI 1")
values = ch.GetScaledData()
```

C#

```csharp
app.LoadFile(@"C:\data\run1.dxd");
app.Data.SelectDataRegion(10.0, 20.0);     // seconds

dynamic ch = app.Data.FindChannel("AI 1");
dynamic values = ch.GetScaledData();
```

See Files and Export for loading and region selection in full.

Files and Export

General

Working with a stored measurement is three steps: load the file, scope the part of it you want, then export or read. All of it requires **analysis mode** — the export methods do not run in measure mode.

Loading a file

PYTHON

```python
app.Analyze()
app.LoadFile(r"C:\data\run1.dxd")
print("loaded", app.LoadEngine.FileName)
```

C#

```csharp
app.Analyze();
app.LoadFile(@"C:\data\run1.dxd");
Console.WriteLine($"loaded {app.LoadEngine.FileName}");
```

`LoadEngine.FileName` is the file open for analysis. `UsedDatafile` is not an alternative: that is the file a *measurement* writes to, and it does not change when a file is loaded. `LoadEngine.FileOpened` reports whether anything is open, since the name is not cleared on close.

Scoping the region

By default an export covers the whole file. Narrow it with `SelectDataRegion`, in seconds:

PYTHON

```python
app.Data.SelectDataRegion(10.0, 20.0)
```

C#

```csharp
app.Data.SelectDataRegion(10.0, 20.0);
```

Both arguments are shifted by `RelativeTimeOffset` and then clamped to the data actually present, so a range wider than the file is tolerated rather than rejected. A `StopPos` below `StartPos` is raised to match, which produces an **empty region** rather than an error, and an export of an empty region produces an empty file with no warning.

Exporting

Two methods do the same job, differing only in how the export format is named.

Select the export by name, not by number

`GetExportList` returns the available exports one per line, but **a line's position is not its export number**. The list omits UNV, which is number `4`, so everything after it sits one place below its numeric index. Passing a line number to `ExportDataEx` selects the wrong format.

Use `ExportDataName` and pass the line text. A name also survives a change in which exports are installed, where a number does not.

The lines are separated by `\r\n`. Splitting on `\n` alone leaves a trailing carriage return on every name, and `ExportDataName` matches the name exactly, so it rejects the name with *"Export name … does not exist"* and exports nothing.

PYTHON

```python
for line in app.GetExportList().splitlines():
    print(line)                       # e.g. "Excel (*.xlsx)"

err, message = app.ExportDataName(
    "Excel (*.xlsx)",                 # export, by name
    0,                                # time axis: relative
    0,                                # data type: full speed
    0,                                # options
    r"C:\export\run1.xlsx",
    "")                               # in/out: receives the result text
if err != 0:
    raise RuntimeError(f"export failed ({err}): {message}")
```

C#

```csharp
foreach (var line in ((string)app.GetExportList())
             .Split(new[] { "\r\n" }, StringSplitOptions.RemoveEmptyEntries))
    Console.WriteLine(line);           // e.g. "Excel (*.xlsx)"

string message = "";
int err = app.ExportDataName(
    "Excel (*.xlsx)",                 // export, by name
    0,                                // time axis: relative
    0,                                // data type: full speed
    0,                                // options
    @"C:\export\run1.xlsx",
    ref message);                     // in/out: receives the result text
if (err != 0)
    throw new Exception($"export failed ({err}): {message}");
```

The list covers the built-in exports only. Formats added by add-ons do not appear, and those are numbered from `-2` downwards for `ExportDataEx`.

The three numeric arguments

Argument	Values
`TimeAxis`	`0` relative, `1` absolute, `2` trigger, `3` pre-trigger time. For CEA data: `0` degrees by cycle, `1` degrees continuous, `2` cycles, `3` samples.
`ExportDataTyp e`	`0` full speed, `1` reduced, `3` CEA.
`ExportOption`	A bit mask whose meaning depends on the data type.

`ExportOption` combines flags by addition:

Data type	Flags
Reduced	`1` min, `2` max, `4` average, `8` RMS
Complex	`1` real, `2` imaginary, `4` amplitude, `8` phase
CEA	`1` cycle, `2` once per cycle, `4` averaged cycle

So minimum and maximum together is `3`. An `ExportOption` of `0` on reduced data asks for none of the four statistics.

> **Both export methods block on a dialog rather than returning an error**
>
> Two conditions put a message box on the DewesoftX window and wait for someone to dismiss it, which hangs an unattended client:
>
> - Calling either method **outside analysis mode**.
> - Asking for `ExportDataType` `0` (full speed) from a file that holds **only reduced data**.
>
> Call `Analyze` first, and use `ExportDataType` `1` for a reduced-only file. `ExportDataEx` returns nothing at all, so it cannot report either case even in principle.
>
> Set `SuppressMessages` as well. With it on, `ExportDataName` returns its error code instead of waiting behind a dialog — see unattended operation.

Exporting to DewesoftX's own format

`ExportToDewesoft` writes a `.dxd`, which cuts a long measurement down to a region of interest without leaving the native format.

How it treats the name it is given:

`FileName`	Result
a full path	used as given, and any missing directories are created
a bare name, no path	placed in the configured export directory
empty	named after the loaded file, without its extension, in the export directory

Called outside analysis mode it puts an *Export data issue* message box on the DewesoftX window and waits for someone to dismiss it, rather than returning an error. Call `Analyze` first, and set `SuppressMessages` so a mistake cannot stall the script.

Recalculating before exporting

Mathematics configured after a measurement was recorded does not exist in the file until it is recalculated. `IOfflineCalc` does that:

PYTHON

```
oc = app.OfflineCalc
oc.Calculate()                 # blocks until finished
oc.StoreCalculatedChannels()   # ...and closes the file
```

C#

```
dynamic oc = app.OfflineCalc;
oc.Calculate();                // blocks until finished
oc.StoreCalculatedChannels();  // ...and closes the file
```

`Calculate` reports nothing and blocks until the recalculation finishes, which is not quick on a large file.

Recalculating also corrects mathematics that was computed live against data which arrived too late to be included. The raw inputs in the file are correct in that case even when the calculated channel is not — see calculation delay.

`StoreCalculatedChannels` **closes the loaded file** as part of storing. Read anything still needed from the calculated channels first, or load the file again afterwards.

Reading values instead of exporting

To get the numbers into your own program rather than into a file, read the channels directly — see Reading Channel Data. The selected region applies there too.

A complete job

PYTHON

```
app.Analyze()
app.LoadFile(r"C:\data\run1.dxd")

app.Data.SelectDataRegion(10.0, 20.0)

app.OfflineCalc.Calculate()

err, message = app.ExportDataName(
    "Excel (*.xlsx)", 0, 0, 0, r"C:\export\run1.xlsx", "")
```

C#

```
app.Analyze();
app.LoadFile(@"C:\data\run1.dxd");

app.Data.SelectDataRegion(10.0, 20.0);

app.OfflineCalc.Calculate();

string message = "";
app.ExportDataName("Excel (*.xlsx)", 0, 0, 0, @"C:\export\run1.xlsx", ref message);
```

Recalculate **before** exporting, or the export writes the channels as they were when the file was recorded.

Data Buffers

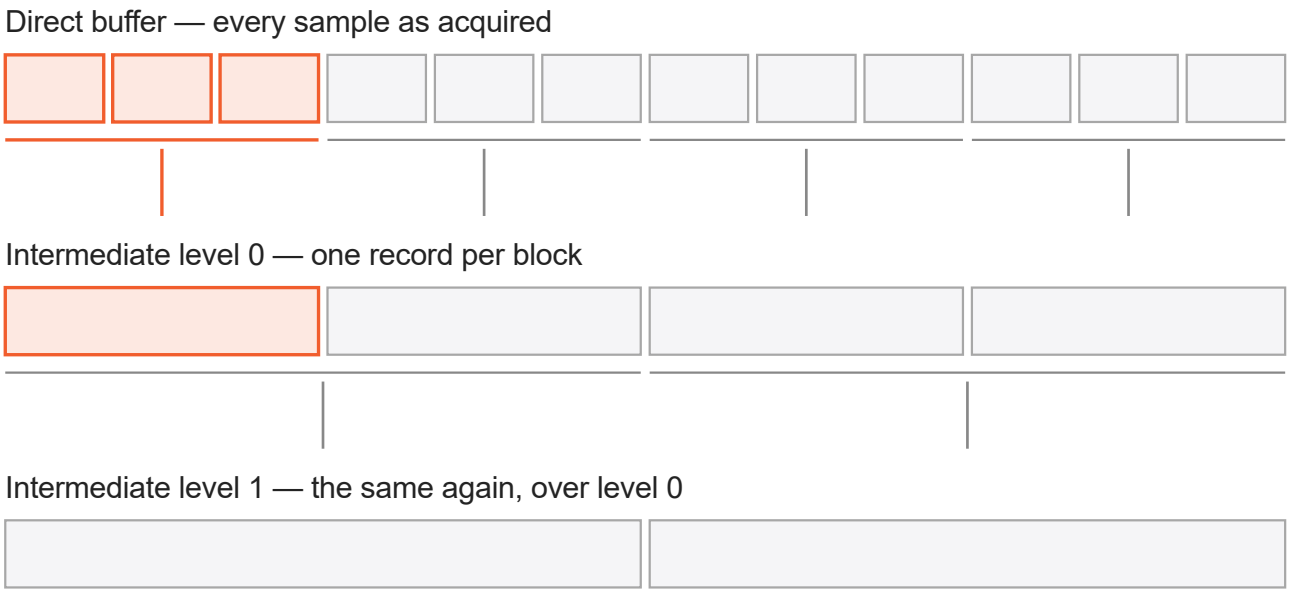
General

Every channel holds its samples in memory in the same shape, and reading a channel means reading that shape correctly. There are two kinds of buffer, stacked:

The **direct buffer** holds the samples as acquired.

Above it sit up to a handful of **intermediate buffers**. Each one stores the minimum, maximum, average and RMS of a block of the level below it.

So level 0 summarises the direct buffer, level 1 summarises level 0, and so on. A display uses them to draw the shape of an hour-long measurement without touching every sample, and they are what to read for an overview rather than the data itself.



The highlighted block shows the correspondence: those samples become the single minimum / maximum / average / RMS record above them. Every level does the same to the level below it, so a coarser level covers a longer span of the measurement in the same number of reads — for as many levels as `IData.IBLevels` reports.

`IData.IBLevels` reports how many intermediate levels exist. It is `0` until acquisition starts.

The ring

Each buffer is a ring: a fixed-size array that wraps around and overwrites its oldest samples. Three properties describe the direct buffer, and reading it correctly takes all three.

Property	Meaning
`DBBufSize`	How many samples the buffer can hold.
`DBDataSize`	How many samples in it are valid.
`DBPos`	The slot the *next* sample will be written to.

`DBBufSize` is worked out from the sample rate and other settings when the channel is mounted. It stays constant for the whole measurement, but a different measurement may give a different size.

> **The sizes are `0` until acquisition starts**
>
> Before `Start`, `DBBufSize` reads `0` and there are no intermediate levels. Read buffer geometry after acquisition is running, not while setting up.

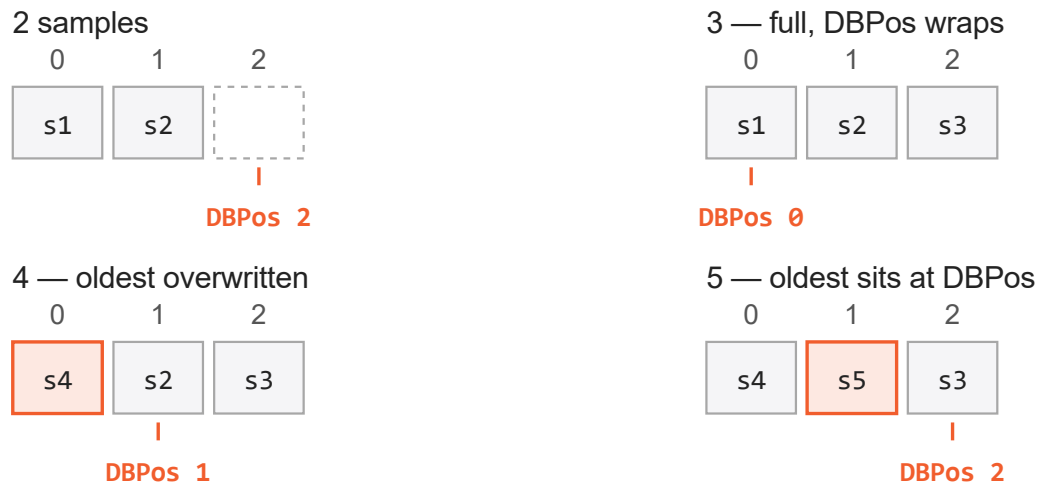
`DBDataSize` counts up from `0` and stops at `DBBufSize`. Once it gets there the ring is full and the oldest samples are being overwritten — but `DBDataSize` **does not change again**, so it is not a count of everything acquired and nothing signals the wrap. `DBPos` is what keeps moving.

How the ring fills

With a buffer size of `3` — real buffers are far larger, but three is enough to see the behaviour:

Samples acquired	`DBDataSize`	`DBPos`	Newest at	Oldest at
0	0	0	—	—
1	1	1	0	0
2	2	2	1	0
3	3	0	2	0
4	3	1	0	1
5	3	2	1	2

At sample 3 the buffer is full and `DBPos` has wrapped back to `0`. From sample 4 on, each write destroys the oldest sample, `DBDataSize` sits at `3` for the rest of the measurement, and the oldest sample is no longer at index `0`.



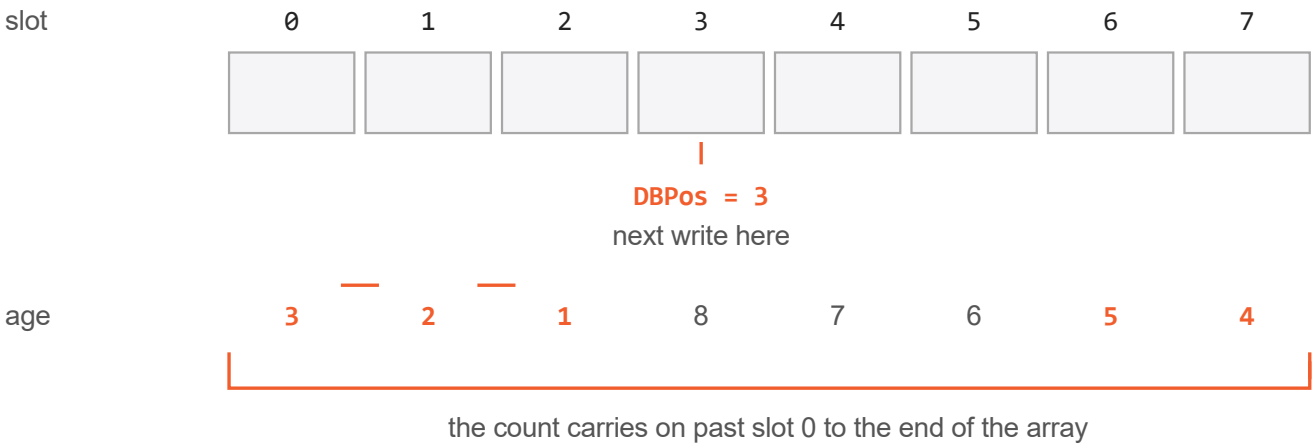
Once the ring is full the write position keeps moving but `DBDataSize` stays at 3 — so it cannot tell you the buffer has wrapped, and index `0` is no longer the oldest sample. The highlighted slot is the one just overwritten.

Addressing a sample by age

Because `DBPos` moves and the ring wraps, an index into the buffer is only meaningful relative to the write position. Count backwards from it:

```
index = (DBPos - age + DBBufSize) mod DBBufSize
```

where `age` is `1` for the newest sample, `2` for the one before it, and at most `DBDataSize` for the oldest.



Age `1` is the slot just behind `DBPos`, and adding `DBBufSize` before the modulo carries the count around the end of the array instead of producing a negative index. On a full ring the oldest sample — age `8` here — sits at `DBPos` itself, which is why it is the next one overwritten.

PYTHON

```python
def sample_by_age(ch, age):
    """age 1 = newest, up to ch.DBDataSize = oldest."""
    if age < 1 or age > ch.DBDataSize:
        raise IndexError(f"only {ch.DBDataSize} samples available")
    return ch.DBValues((ch.DBPos - age + ch.DBBufSize) % ch.DBBufSize)

print(sample_by_age(ch, 1))    # newest
```

C#

```csharp
static float SampleByAge(dynamic ch, int age)
{
    int size = (int)ch.DBDataSize, cap = (int)ch.DBBufSize, pos = (int)ch.DBPos;
    if (age < 1 || age > size)
        throw new IndexOutOfRangeException($"only {size} samples available");
    return (float)ch.DBValues[(pos - age + cap) % cap];
}

Console.WriteLine(SampleByAge(ch, 1));   // newest
```

Adding `DBBufSize` before the modulo is what keeps the index valid when `DBPos` has just wrapped to `0` and `DBPos - age` would otherwise be negative.

The **oldest** sample is a special case, because it depends on whether the ring has wrapped yet:

PYTHON

```python
oldest = 0 if ch.DBDataSize < ch.DBBufSize else ch.DBPos
print(ch.DBValues(oldest))
```

C#

```csharp
int oldest = (int)ch.DBDataSize < (int)ch.DBBufSize ? 0 : (int)ch.DBPos;
Console.WriteLine(ch.DBValues[oldest]);
```

Nothing is range-checked

Reading outside the valid data does not fail. It returns whatever is in memory — zeroes, values from a previous measurement, or uninitialised bytes — and none of it is distinguishable from a real sample. On some paths it faults instead.

Check `DBDataSize` before every read and clamp the count to it. Reading ten samples means confirming `DBDataSize >= 10` first.

Reading in bulk

One call per sample means one call into DewesoftX per sample. Reading a block is a single call for the whole block, and the difference is large.

What you want	Use
A block of samples	`GetScaledDataEx`
A stream, with the position tracked for you	`IChannelConnection`
One specific sample	`DBValues`
An overview of a long measurement	the intermediate buffers, below

Prefer the first two. See Reading Channel Data for what each path does and which channels it works on.

The intermediate buffers

These hold no individual samples — only the minimum, maximum, average and RMS of each block. Two sets of members reach them:

For	Members
Level 0 only	`IBBufSize`, `IBDataSize`, `IBPos`
Any level	`IBDataSizeEx`, `IBPosEx`, `FirstIBLevel`

The same ring rules apply — same wrap, same lack of range checking.

`FirstIBLevel` **may not be** `0`: a slow asynchronous channel starts higher, so do not assume level `0` exists. The valid levels run from `0` to `IData.IBLevels - 1`; asking for `IBLevels` itself, or anything above it, raises a range-check error rather than returning nothing.

PYTHON

```python
print(app.Data.IBLevels)                # how many levels exist
print(ch.FirstIBLevel)                  # the lowest this channel has

rec = ch.IBValues(0)
print(rec.Min, rec.Max, rec.Ave, rec.Rms)
```

C#

```csharp
Console.WriteLine(app.Data.IBLevels);    // how many levels exist
Console.WriteLine(ch.FirstIBLevel);      // the lowest this channel has

dynamic rec = ch.IBValues[0];
Console.WriteLine($"{rec.Min} {rec.Max} {rec.Ave} {rec.Rms}");
```

Because these are summaries, they are always synchronous — there are no timestamps to read from them.

Channel Kinds

General

Everything DewesoftX measures or calculates is a channel, but channels are not all the same shape. Four things vary independently, and each one changes how you read the channel:

Varies in	Values
Timing	synchronous, asynchronous, or single-value
Sample shape	one value per sample, or an array
Data	numeric, or text
Direction	read-only, or a control channel you write to

These combine freely: an asynchronous array channel is ordinary. Test each property separately rather than inferring one from another.

Where channels live

Channels of one kind are collected into a **channel group** — one for analog inputs, one for CAN data, one for each add-on that supplies channels, and so on. The groups together hang off `IData`.

Most of the time you do not walk that structure. Two flat lists are easier:

PYTHON

```python
data = app.Data
print(data.UsedChannels.Count)     # enabled channels
print(data.AllChannels.Count)      # every channel defined, enabled or not
```

C#

```csharp
dynamic data = app.Data;
Console.WriteLine(data.UsedChannels.Count);   // enabled channels
Console.WriteLine(data.AllChannels.Count);    // every channel defined
```

A channel that is not enabled produces no data, so `UsedChannels` is usually the list to read. See Reading Channel Data for looking a single channel up.

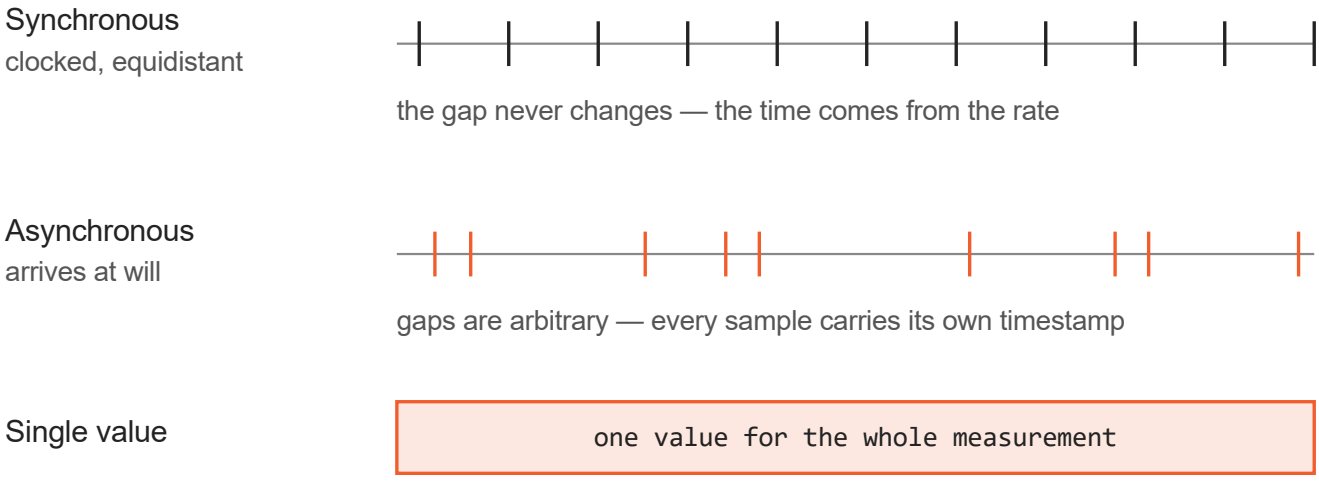
Timing: synchronous, asynchronous, single-value

This is the distinction that most affects reading.

Synchronous channels are clocked by DewesoftX's master clock, so their samples are equidistant. They carry **no timestamps** — the time of a sample is derived from the sample rate, as described in Timing and Rates.

Asynchronous channels arrive when they arrive. The gap between samples is arbitrary, so each sample carries its own timestamp and you read values and timestamps together.

Single-value channels hold one value for the whole measurement — a statistic computed over the entire run, say. There is no history to read.



The gap between samples is what separates the three: fixed, arbitrary, or absent. It decides whether you read timestamps, derive them, or have nothing to read.

PYTHON

```python
if ch.IsSingleValue:
    print(ch.SingleValue)
elif ch.Async:
    values = ch.GetScaledDataEx(0, n)
    times = ch.GetTSDataEx(0, n)
else:
    values = ch.GetScaledDataEx(0, n)   # times come from the sample rate
```

C#

```csharp
if ((bool)ch.IsSingleValue)
    Console.WriteLine(ch.SingleValue);
else if ((bool)ch.Async)
{
    dynamic values = ch.GetScaledDataEx(0, n);
    dynamic times = ch.GetTSDataEx(0, n);
}
else
{
    dynamic values = ch.GetScaledDataEx(0, n);   // times from the sample rate
}
```

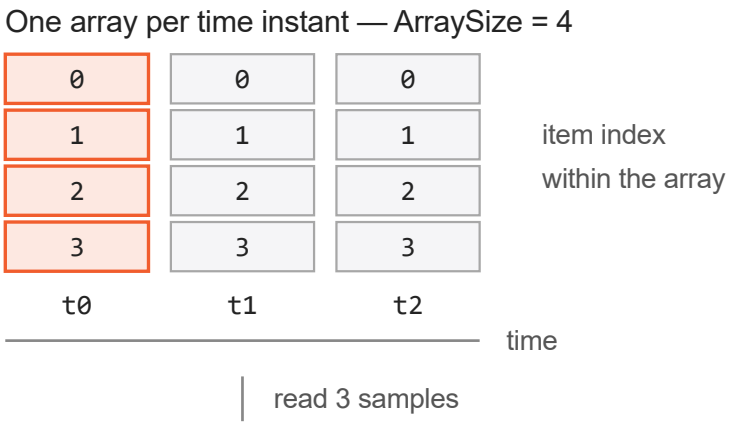
Asking a synchronous channel for timestamps faults

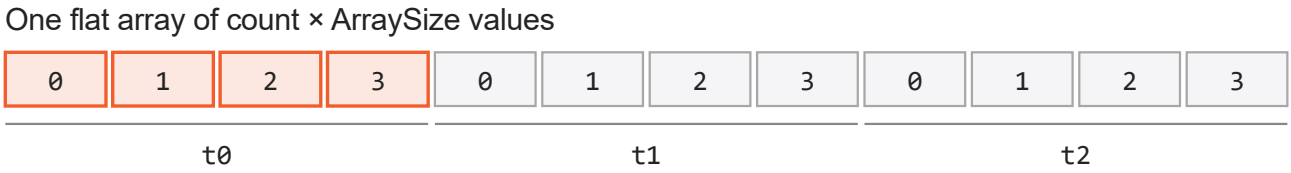
`GetTSData`, `DBTimeStamp` and a connection's `GetTSValues` all fault inside DewesoftX on a synchronous channel rather than returning an error — there is no timestamp array to read. Check `Async` first.

Array channels

An array channel stores a whole array at each time instant rather than a single value — an FFT spectrum, a matrix. `ArraySize` gives the array length, and `ArrayInfo` describes its shape and axes: `DimCount` is `1` for a vector and `2` for a matrix.

Reading one is mostly like reading any other channel, with two differences. The unbounded `GetScaledData()` **fails** on an array channel, so ask for a window. And the flat result of `GetScaledDataEx(start, count)` is `count × ArraySize` values, in sample order — `count` is still a number of samples, not of elements.





The number in each cell is the item's index within its own array, not a value. Items stay together and samples follow one another, so element `i × ArraySize + j` is item `j` of sample `i`. `count` is a number of samples — only the length of the result is multiplied by `ArraySize`.

PYTHON

```python
if ch.ArraySize > 1:
    n = ch.ArraySize
    flat = ch.GetScaledDataEx(0, 3)          # 3 samples -> 3 * n values
    arrays = [flat[i:i + n] for i in range(0, len(flat), n)]
    print(ch.ArrayInfo.DimCount, ch.ArrayInfo.DimSizes(0))
```

C#

```csharp
if ((int)ch.ArraySize > 1)
{
    int n = (int)ch.ArraySize;
    dynamic flat = ch.GetScaledDataEx(0, 3);  // 3 samples -> 3 * n values
    Console.WriteLine($"{ch.ArrayInfo.DimCount} {ch.ArrayInfo.DimSizes[0]}");
}
```

An array channel may be synchronous or asynchronous — do not infer one from the other.

Text channels

Some channels carry text rather than numbers. Read `Text` instead of a numeric path; `SingleValue` on a text channel returns nothing useful. `Text` returns an empty string on an ordinary numeric channel, so it is not by itself a reliable test of kind.

Control channels

A control channel goes the other way: it is written to during a measurement, usually by an operator working a button or a slider on a display screen. A client reads it to find out what the operator did — resetting a counter when a button is pressed, for instance.

`IsControlChannel` identifies them.

PYTHON

```python
for i in range(data.UsedChannels.Count):
    c = data.UsedChannels.Item(i)
    if c.IsControlChannel:
        print(c.Name, c.SingleValue)
```

C#

```csharp
for (int i = 0; i < (int)data.UsedChannels.Count; i++)
{
    dynamic c = data.UsedChannels.Item[i];
    if ((bool)c.IsControlChannel)
        Console.WriteLine($"{c.Name} {c.SingleValue}");
}
```

Telling them apart

Test	Means
`ch.IsSingleValue`	one value for the whole measurement; read `SingleValue`
`ch.Async`	samples carry their own timestamps
`ch.ArraySize > 1`	an array per sample
`ch.IsControlChannel`	written by the operator, not measured

`ChannelType` additionally names the subsystem a channel came from, and `DataType` how its samples are stored.

Timing and Rates

General

DewesoftX has more than one sample rate, and it does not use the same one all the time. Reading the wrong one is an easy way to compute timestamps that are wrong by a factor of several.

Three rates

Rate	Member	What it is
Dynamic acquisition rate	`IApp.MeasureSampleRate`	The rate analog data is acquired at in measure mode. This is the one people mean by "the sample rate".
Setup sample rate	`IApp.SetupSampleRate`	A separate, usually different rate used for the live preview in Config.
Reduced rate	`IApp.ReducedRate`	In Hz, used only by the slow storing modes.

Channels from add-on devices may have their own rates, unrelated to any of these: a serial device can be far slower than the analog channels beside it.

`IData.SampleRate` is not the acquisition rate

`IData.SampleRate` reports the rate **currently in force**, which is not the same thing:

Where DewesoftX is	What it returns
Config	the setup sample rate
Acquiring or storing	the dynamic acquisition rate

Derive timestamps from it only while acquisition is running, or read `MeasureSampleRate` to be explicit about which rate is meant.

Use `SampleRateEx` rather than `SampleRate`: it keeps fractional rates, where `SampleRate` truncates them.

Timestamps on a synchronous channel

A synchronous channel stores no timestamps, because it does not need to: its samples are equidistant, so the time of a sample is its position divided by the rate.

Getting the absolute position takes two members. `GetSamplesAcquired` returns a block count and an offset within the current block, and `Samples` is the block size:

```
absolute sample = blocks × IData.Samples + offset
time in seconds = absolute sample ÷ IData.SampleRateEx
```

PYTHON

```python
mid, dir_ = app.Data.GetSamplesAcquired()      # both are out parameters
absolute = mid * app.Data.Samples + dir_
seconds = absolute / app.Data.SampleRateEx
print(f"{absolute} samples, {seconds:.3f} s into the measurement")
```

C#

```csharp
int mid, dir;
app.Data.GetSamplesAcquired(out mid, out dir);
long absolute = mid * (int)app.Data.Samples + dir;
double seconds = absolute / (double)app.Data.SampleRateEx;
Console.WriteLine($"{absolute} samples, {seconds:F3} s into the measurement");
```

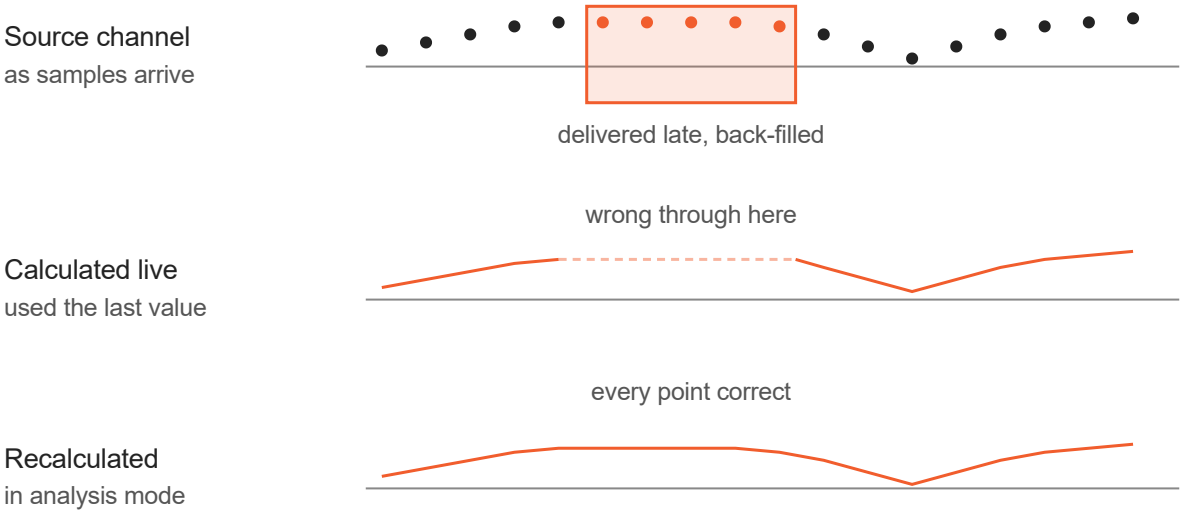
`Mid` is the block count and `Dir` the offset within the current block. The same pair appears throughout the interface — it is what `OnTrigger` reports, and what `IData.CurrentPos` and `IData.EndStamp` carry.

Calculation delay

DewesoftX does not calculate mathematics on data the moment it arrives. It waits a little, because data from outside the analog hardware does not arrive on time — an Ethernet device may deliver a block every hundred milliseconds, and an add-on reading a slow instrument may be later still.

That wait is the **calculation delay**. Within it, late data still gets included. Past it, the calculation goes ahead with **the last value the channel had**, and the result is wrong.

When the late data does arrive, it lands in the channel's buffer correctly. The raw data in the file is right; only the *calculated* channels are wrong, because they were computed before the input existed.



The raw samples in the file are the same in both cases. Only the calculated channel differs, because live it ran before its input arrived.

PYTHON

```python
print(app.Data.MaxCalcDelay)   # highest delay across all channels, in samples
print(ch.CalcDelay)            # this channel's own delay, in samples
```

C#

```csharp
Console.WriteLine(app.Data.MaxCalcDelay);  // highest across all channels, samples
Console.WriteLine(ch.CalcDelay);           // this channel's own delay, samples
```

Both are in **samples**, not seconds — divide by the acquisition rate to compare them against a device's latency. They read `0` when nothing in the setup has asked for a delay, which is the normal case for a purely analog measurement.

> **A math channel that looks wrong live may be right after recalculating**
>
> Because the raw inputs are stored correctly, recalculating in analysis mode computes the mathematics again with all the data present, and the result is correct.
>
> A live math channel disagreeing with the raw signal feeding it is therefore not necessarily a setup fault: it can be data that arrived after its calculation ran. Recalculate before drawing conclusions, and before exporting. See Files and Export.

The acquisition loop

`IApp.TimerInterval` is how often, in milliseconds, DewesoftX runs the loop that fetches data and drives calculations. It sets the granularity of everything above: how often a block appears, and therefore how often anything downstream can react.

Ordinary automation has no reason to change it. It matters when diagnosing why data appears in the chunks it does.

IAISetupScreen

Two operations on the analog input page in **Config** — which channel it shows, and which of its columns are visible.

Get it from `IApp.AISetupScreen`.

Both act on the interface a person is looking at, and both need DewesoftX to be in **Config**.

PYTHON

```python
s = app.AISetupScreen
s.SetColumnVisible(5, False)   # hide the description column
s.ShowChannelSetup(0)
```

C#

```csharp
dynamic s = app.AISetupScreen;
s.SetColumnVisible(5, false);   // hide the description column
s.ShowChannelSetup(0);
```

Members

ShowChannelSetup

Method

Opens the setup for one channel.

Parameter	Type	Direction	Description
`ChNo`	`Integer`	in	The channel, by position.

> **Shows a dialog when not in Config**
>
> Outside Config this puts a *Not in setup mode* message box on the DewesoftX window and waits for someone to dismiss it, rather than returning an error. Switch to Config first.

SetColumnVisible

Method

Shows or hides one column of the channel grid.

Parameter	Type	Direction	Description
`ColNo`	`Integer`	in	Which column — see below.
`Visible`	`Boolean`	in	`True` to show it, `False` to hide it.

Value	Column
`1`	Slot
`2`	On/off
`3`	Colour
`4`	Name
`5`	Description
`6`	Values
`7`	Zero
`8`	Setup

Column `0` is the expander and cannot be hidden — passing it does nothing. Hiding restores each column to its own standard width when shown again, so a width a user had dragged is not preserved.

IAOChannel

One analog output channel of the function generator — its waveform and the shape of it.

Every analog output channel carries this interface as well as `IChannel`, so one object serves both. Get one from `IAOGroup.AOChannels`.

Frequency and sweep behaviour are set for the whole generator on `IAOGroup`, not per channel. What lives here is the shape, level and phase of this one output.

PYTHON

```
ao = app.AOGroup.AOChannels.Item(0)
ao.WaveForm = 0        # sine
ao.Ampl = 2.0
ao.Offset = 0.0
ao.Phase = 90.0
```

C#

```
dynamic ao = app.AOGroup.AOChannels.Item[0];
ao.WaveForm = 0;       // sine
ao.Ampl = 2.0;
ao.Offset = 0.0;
ao.Phase = 90.0;
```

Waveform

WaveForm

Property — read/write — `Integer`

The shape this channel puts out:

Value	Waveform
`0`	sine
`1`	triangle
`2`	rectangle
`3`	sawtooth
`4`	white noise
`5`	arbitrary — a shape loaded into the channel

Ampl

Property — read/write — `Single`

Amplitude, in the channel's output unit.

Offset

Property — read/write — `Single`

DC offset added to the waveform, in the same unit as `Ampl`.

Phase

Property — read/write — `Single`

Phase, in degrees, relative to the generator's own phase reference. This is what puts two outputs at a fixed angle to each other.

Range

Property — read/write — `Integer`

The output range of the module driving this channel.

Only meaningful on hardware with a range-switching output module. Everything else reads `1` and ignores writes, so a successful write is not evidence the range changed.

Noise filter

These shape the white-noise output and **do nothing for any other waveform** — setting them on a sine channel is accepted and has no effect.

FilterType

Property — read/write — `Integer`

Which filter the noise passes through:

Value	Filter
`0`	none
`1`	pink — the usual choice for pink noise
`2`	low-pass
`3`	band-pass
`4`	band-stop

There is no high-pass option.

FilterProtoType

Property — read/write — `Integer`

The filter's design: `0` Butterworth, `1` Chebyshev, `2` Bessel. Not used when `FilterType` is `0` or `1`.

FilterOrder

Property — read/write — `Integer`

The filter order — how steeply it rolls off.

FilterFreq1

Property — read/write — `Single`

The filter's first corner frequency in **Hz**: the cut-off for a low-pass, or the lower edge for a band-pass or band-stop.

FilterFreq2

Property — read/write — `Single`

The upper edge in **Hz**, for a band-pass or band-stop. Unused by the other filters.

The corner frequencies are held relative to `IAOGroup.SampleRate`, so changing the generator's sample rate moves the filter unless you set these again afterwards. A pair of frequencies the filter cannot be made stable at is ignored, leaving the previous filter in place.

IAOGroup

The function generator — what all the analog outputs do together.

Get it from `IApp.AOGroup`. The individual outputs are `IAOChannel`, reached through `AOChannels`.

The division of labour is worth knowing before you start: the **frequency, the sweep and the timing belong to the group**, and every channel follows them. Amplitude, offset, phase and waveform belong to each channel. So a two-channel setup at the same frequency but different phase is one group frequency and two channel phases.

PYTHON

```python
g = app.AOGroup
g.OperationMode = 1        # sweep
g.StartFreq = 10.0
g.StopFreq = 1000.0
g.StopTime = 30.0
g.LogSweep = True
g.SweepMode = 1            # loop
```

C#

```csharp
dynamic g = app.AOGroup;
g.OperationMode = 1;       // sweep
g.StartFreq = 10.0;
g.StopFreq = 1000.0;
g.StopTime = 30.0;
g.LogSweep = true;
g.SweepMode = 1;           // loop
```

Mode and output

OperationMode

Property — read/write — `Integer`

What the generator does:

Value	Mode	Frequency comes from
`0`	fixed	`Freq`
`1`	sweep	`StartFreq` to `StopFreq`, continuously
`2`	step sweep	the same range, in `DeltaFreq` steps
`3`	burst	`Freq`, gated by `StartTime` and `StopTime`
`4`	chirp	`StartFreq` to `StopFreq`, once per period

The mode decides which of the members below matter. The others keep their values and are ignored.

Freq

Property — read/write — `Single`

The output frequency in **Hz**, for fixed and burst modes.

While the generator is running this reports the frequency of the first waveform channel — the frequency actually being produced, which in a sweep is not the value you last wrote. Stopped, it reports the stored setting.

SampleRate

Property — read/write — `Integer`

The rate the generator produces samples at, in samples per second. It bounds the highest frequency you can usefully ask for, and the noise filter corners are held relative to it.

AOChannels

Property — read-only — `IChannelList`

The analog output channels. Each also carries `IAOChannel`, so a channel from this list answers to both.

ControlsClock

Property — read-only — `Boolean`

`True` when the analog output is driving the acquisition clock rather than following it. Set by the hardware configuration, not from here.

ShowInfoChannels

Property — read/write — `Boolean`

Whether the generator publishes its own state — frequency, amplitude and so on — as extra channels you can store and display alongside the measurement.

Sweep

StartFreq

Property — read/write — `Single`

The frequency a sweep or chirp begins at, in **Hz**.

StopFreq

Property — read/write — `Single`

The frequency it ends at, in **Hz**.

DeltaFreq

Property — read/write — `Single`

The step size in **Hz** for a step sweep. Unused by the other modes.

LogSweep

Property — read/write — `Boolean`

`True` to sweep logarithmically — equal time per octave rather than per hertz, which is what you usually want across a wide range. `False` sweeps linearly.

SweepMode

Property — read/write — `Integer`

`0` to sweep once and stop, `1` to loop back to the start and keep going.

StartTime

Property — read/write — `Single`

When the output begins, in **seconds** from the start of acquisition.

StopTime

Property — read/write — `Single`

When it ends, in **seconds**. For a sweep this is also what sets the sweep's duration, and so its rate.

Change rates

These four limit how fast the generator may move a setting that is changed while it is running, so the change ramps instead of stepping. Each is a rate per second in the unit of the thing it governs.

> **Zero freezes the setting rather than removing the limit**
>
> The rate is applied by adding it to the current value each sample, with no special case for `0` — so a factor of `0` means the setting never reaches what you asked for and stays where it is. To make a change take effect at once, set a rate high enough to cover the distance, not `0`.
>
> The per-sample step is worked out from `SampleRate` at the moment you write the factor, and only for outputs that are switched on. Change the sample rate, or switch an output on, after setting these and the ramp rate is stale until you write the factor again.

FreqChangeFactor

Property — read/write — `Single`

How fast the frequency may change, on the generator's internal frequency scale per second — **not** hertz per second. Defaults to `10`.

AmplChangeFactor

Property — read/write — `Single`

How fast the amplitude may change, in volts per second. Defaults to `1`.

DCChangeFactor

Property — read/write — `Single`

How fast the offset may change, in volts per second. Defaults to `1`.

PhaseChangeFactor

Property — read/write — `Single`

How fast the phase may change, in degrees per second. Defaults to `30`.

IAcceptChannel

A callback asking whether a channel may be dropped into a slot.

Derives from `IUnknown`, so it is vtable-only. Paired with `IAcceptGroup`.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`Call`	Return whether the channel is acceptable.

IAcceptGroup

A callback asking whether a channel group may be dropped into a slot.

Derives from `IUnknown`, so it is vtable-only. Paired with `IAcceptChannel`.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`Call`	Return whether the group is acceptable.

IAcqLoop

The acquisition loop — how hard DewesoftX drives its display updates.

Get it from `IApp.AcqLoop`.

MaxUpdateRateMode

Property — read/write — `Boolean`

`True` to update the displays as fast as the data allows rather than at the regular interval.

Worth turning on when a display needs to keep up with fast signals, and worth leaving off otherwise — it spends processor time on drawing that acquisition and mathematics would otherwise have.

IAlarmCond

One alarm — the conditions that raise it, and how it is cleared.

Get one from `IAlarms`, either by index or from `Add`.

An alarm is raised by the conditions in `Trigger`, and cleared according to `StopOption`. While raised it appears in `IAlarms.ActiveItem` and its state changes are recorded as events.

PYTHON

```python
a = app.Alarms.Add()
a.CustomName = "Over temperature"
cond = a.Trigger.OrList.Add()
cond.AddChannel(app.Data.FindChannel("Temp"))
cond.Level1 = 85.0
a.StopOption = 3           # clear after a time
a.StopTime = 10.0          # seconds
```

C#

```csharp
dynamic a = app.Alarms.Add();
a.CustomName = "Over temperature";
dynamic cond = a.Trigger.OrList.Add();
cond.AddChannel(app.Data.FindChannel("Temp"));
cond.Level1 = 85.0;
a.StopOption = 3;          // clear after a time
a.StopTime = 10.0;         // seconds
```

Identity and state

Name

Property — read-only — `String`

The alarm's name — `CustomName` if one was given, otherwise a name DewesoftX generates.

CustomName

Property — read/write — `String`

A name of your choosing. Setting it makes `Name` return the same value.

Status

Property — read-only — `Boolean`

`True` while the alarm is raised.

Raising and clearing

Trigger

Property — read-only — `ITrig`

The conditions that raise the alarm.

StopOption

Property — read/write — `Integer`

How the alarm is cleared:

Value	Cleared
`0`	never — it stays raised for the rest of the measurement
`1`	by hand, through `EndAlarm`
`2`	when the `StopTrigger` conditions are met
`3`	after `StopTime` has elapsed

StopTime

Property — read/write — `Single`

How long the alarm stays raised, in **seconds**. Only used when `StopOption` is `3`.

StopTrigger

Property — read-only — `ITrig`

The conditions that clear the alarm. Only used when `StopOption` is `2`.

EndAlarm

Method

Clears the alarm. Only has an effect when `StopOption` is `1`.

Not implemented

Avail

Property — read-only — `Boolean`

Reserved. Not implemented.

Index

Property — read-only — `Integer`

Reserved. Not implemented — use the position in `IAlarms.Item` instead.

IAlarms

The configured alarms, and which of them are currently active.

Get it from `IApp.Alarms`. Each alarm is an `IAlarmCond`.

Two views of the same alarms:

	Contents	Indexed by
`Item` / `Count`	every alarm configured	`0` to `Count - 1`
`ActiveItem` / `ActiveCount`	only those currently raised	`0` to `ActiveCount - 1`

PYTHON

```python
alarms = app.Alarms
print(f"{alarms.ActiveCount} of {alarms.Count} active")
for i in range(alarms.ActiveCount):
    print(alarms.ActiveItem(i).Name)
```

C#

```csharp
dynamic alarms = app.Alarms;
Console.WriteLine($"{alarms.ActiveCount} of {alarms.Count} active");
for (int i = 0; i < (int)alarms.ActiveCount; i++)
    Console.WriteLine(alarms.ActiveItem[i].Name);
```

Members

Count

Property — read-only — `Integer`

Number of alarms configured.

Item

Property — read-only — `IAlarmCond`

The alarm at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `Count - 1`.

ActiveCount

Property — read-only — `Integer`

How many alarms are currently raised — those whose `Status` is `True`.

ActiveItem

Property — read-only — `IAlarmCond`

The raised alarm at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `ActiveCount - 1`.

This list is only dependable during a measurement, and its membership changes as alarms come and go — so an index is valid only for as long as you hold it. Read `ActiveCount` and the items you want in one pass.

Add

Method — returns `IAlarmCond`

Creates an alarm and returns it, ready to configure.

Remove

Method

Deletes an alarm.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position in the `Item` list, not the active list.

IAmplChain

One amplifier chain — a run of modules on a connection, and the sensor settings that apply to the chain as a whole.

Get it from `IAmplChainList.Item`. Each module in it is an `IAmplifier`.

Members

Count

Property — read-only — `Integer`

Number of amplifier modules in the chain.

Item

Property — read-only — `IAmplifier`

The module at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the modules can be iterated directly in languages that support it.

IOControl

Method — returns `Integer`

Reads or writes one chain-level setting — mostly the custom sensor scaling applied across the chain.

Parameter	Type	Direction	Description
`IOCode`	`Integer`	in	Which setting — see below.
`InParam`	`Variant`	in	The value to write. Ignored by `Get` codes.
`OutParam`	`Variant`	out	The value read. Not set by `Set` codes.

Returns `0` on success, `-1` for an unrecognised code, `-2` for the wrong `InParam` type, `-3` for an index out of range.

These codes are not the amplifier codes

Chain codes start at `0` and so do amplifier codes — `0` is the found index here but the module name on an `IAmplifier`. Send each code to the level it belongs to.

Chain properties

Code	Constant	Value type
`0`	`chainGetFoundIdx`	`Integer`
`1`	`chainGetAmplTypeAt`	`String`
`2`	`chainGetSensorName`	`String`
`3`	`chainSetSensorName`	`String`
`4`	`chainGetSensorCustomGainUsed`	`Boolean`
`5`	`chainGetSensorCustomGain`	`Double`
`6`	`chainSetSensorCustomGain`	`Double`
`7`	`chainGetSensorCustomOffsetUsed`	`Boolean`
`8`	`chainGetSensorCustomOffset`	`Double`
`9`	`chainSetSensorCustomOffset`	`Double`

Chain commands

Code	Constant	Effect
`10000`	`chainPrepareScaling`	Recomputes the chain's scaling from its current settings.
`10001`	`chainTedsForcedFullRead`	Forces a full re-read of the TEDS data on the chain.

IAmplChainList

The amplifier chains on one connection.

Get it from `IAmplInterface.ChainList`. Each chain is an `IAmplChain`.

Members

Count

Property — read-only — `Integer`

Number of chains.

Item

Property — read-only — `IAmplChain`

The chain at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the chains can be iterated directly in languages that support it.

IAmplInterface

A connection to amplifier hardware — the chains hanging off it, and commands aimed at the connection itself.

Get it from `IAmplInterfaces.MainInterface`, or from `SubInterface` for a nested one.

Members

ChainList

Property — read-only — `IAmplChainList`

The amplifier chains reachable through this connection.

SubInterface

Property — read-only — `IAmplInterface`

A further connection nested below this one, or nothing when there is none — which is the usual case.

IOControl

Method — returns `Integer`

Reads a property of the connection, or sends a command to it.

Parameter	Type	Direction	Description
`IOCode`	`Integer`	in	Which property or command.
`InParam`	`Variant`	in	The argument.
`OutParam`	`Variant`	out	The result.

Code	Meaning	Value type
`0`	Maximum channels the acquisition hardware supports	`Integer`
`1`	Number of additional modules found	`Integer`
`2`	The interface type	`String`
`3`	Send a raw command to the hardware	array of arguments
`4`	A description of the interface	`String`
`5`	Whether the main interface is offline	`Boolean`
`6`	Whether the sub-interface is offline	`Boolean`
`7`	Whether the first hardware scan has finished	`Boolean`

Returns `0` on success, `-1` for a code this connection does not recognise, `-2` for the wrong `InParam` type, `-3` for an index out of range.

Three separate code sets

These codes are not the ones `IAmplifier` and `IAmplChain` take. All three levels have an `IOControl` with its own numbering, and the low numbers collide — code `0` means the channel maximum here, the module name on an amplifier, and the found index on a chain. Send each code to the level it belongs to.

IAmplInterfaces

The way in to the amplifier hardware.

Get it from `IApp.AmplInterfaces`.

The amplifiers form a four-level tree: this object leads to an `IAmplInterface` — the connection to the hardware — which holds an `IAmplChainList` of chains, each holding the amplifier modules plugged into it.

PYTHON

```python
chains = app.AmplInterfaces.MainInterface.ChainList
for i in range(chains.Count):
    chain = chains.Item(i)
    for j in range(chain.Count):
        ok, name = chain.Item(j).IOControl(0, None)   # amplGetModuleName
        print(i, j, name if ok == 0 else "-")
```

C#

```csharp
dynamic chains = app.AmplInterfaces.MainInterface.ChainList;
for (int i = 0; i < (int)chains.Count; i++)
{
    dynamic chain = chains.Item[i];
    for (int j = 0; j < (int)chain.Count; j++)
    {
        object name = null;
        int ok = chain.Item[j].IOControl(0, null, out name);   // amplGetModuleName
        Console.WriteLine($"{i} {j} {(ok == 0 ? name : "-")}");
    }
}
```

Members

MainInterface

Property — read-only — `IAmplInterface`

The primary connection to the amplifier hardware.

IAmplifier

One amplifier module — the conditioning stage behind a physical input.

Get it from `IAmplChain.Item`.

Everything about a module is read and written through a single method, `IOControl`, with a numeric code saying which setting you mean.

IOControl

Method — returns `Integer`

Reads or writes one amplifier setting.

Parameter	Type	Direction	Description
`IOCode`	`Integer`	in	Which setting — see the codes below.
`InParam`	`Variant`	in	The value to write. Ignored by `Get` codes.
`OutParam`	`Variant`	out	The value read. Not set by `Set` codes.

Returns `0` on success, or a negative status:

Result	Meaning
`0`	Success. `OutParam` holds the value for a `Get` code.
`-1`	The code is not one this module recognises.
`-2`	`InParam` was the wrong type for this code.
`-3`	The index in `InParam` is out of range.
`-4`	The module recognises the code but does not support the setting.

A failed call leaves `OutParam` untouched, and `-1` or `-4` is the ordinary answer for a setting the hardware simply does not have — so check the result before trusting the output.

PYTHON

```
ampl = app.AmplInterfaces.MainInterface.ChainList.Item(0).Item(0)

ok, name = ampl.IOControl(0, None)      # amplGetModuleName
if ok == 0:
    print("module:", name)

ok, _ = ampl.IOControl(113, 2)          # amplSetRangeIndex -> third range
ampl.IOControl(10000, None)             # amplUpdateAmpl - apply it
```

C#

```
dynamic ampl = app.AmplInterfaces.MainInterface.ChainList.Item[0].Item[0];

object outParam = null;
int ok = ampl.IOControl(0, null, out outParam);   // amplGetModuleName
if (ok == 0)
    Console.WriteLine($"module: {outParam}");

ampl.IOControl(113, 2, out outParam);             // amplSetRangeIndex
ampl.IOControl(10000, null, out outParam);        // amplUpdateAmpl - apply it
```

`OutParam` is an out parameter, so Python receives it as a second return value and C# must declare it. A setting is not on the hardware until you follow the writes with `amplUpdateAmpl` (`10000`).

Codes

Most settings that offer a choice follow one pattern: a `…Count` code for how many options there are, a `…Index` code to read or write which one is selected, and `…StringAt` or `…ValueAt` to ask what an option at a given position is called or is worth. Read the count, walk the positions to find the one you want, then write the index.

Module identity and calibration

Code	Constant	Value type
0	amplGetModuleName	String
1	amplGetModuleUnit	String
2	amplGetShortInfoString	String
3	amplGetModuleType	Integer
4	amplGetModuleSN	String
5	amplGetModuleRev	String
6	amplGetModuleFirmware	String
7	amplGetModuleBaseType	Integer
8	amplGetModuleBaseSN	String
9	amplGetModuleBaseRev	String
10	amplGetModuleBaseFirmware	String
11	amplCalDate	String
12	amplAdjustDate	String
13	amplFilterName	String
14	amplFilterSN	String
15	amplFilterRev	String
16	amplGetOptionalGain	Double
17	amplModuleBaseCorrGain	Double
18	amplModuleBaseCorrOffset	Double
19	amplModuleSubCorrGain	Double

Code	Constant	Value type
20	amplModuleSubCorrOffset	Double
21	amplSetOptionalGain	Double
22	amplOptionDigOut	Byte
23	amplGetLongInfoString	String
24	amplGetModuleDispName	String
25	amplSetCalDate	Boolean
26	amplSetAdjustDate	Boolean

Measurement mode

Code	Constant	Value type
100	amplGetMeasurementCount	Integer
101	amplGetMeasurementIndex	Integer
102	amplGetMeasurementStringAt	String
103	amplGetMeasurementValueAt	String
104	amplSetMeasurementIndex	Integer
105	amplSetMeasurementString	String
106	amplSetMeasurementValue	String

Range

Code	Constant	Value type
110	amplGetRangesCount	Integer
111	amplGetRangeIndex	Integer
112	amplGetRangeStringAt	String
113	amplSetRangeIndex	Integer
114	amplSetRangeString	String
115	amplGetRangeGainAt	Double (uncorrected)
116	amplGetRangeOffsetAt	Double (uncorrected)
117	amplGetAutoRange	Integer
118	amplSetAutoRange	Integer
119	amplGetAutoRangeCount	Integer

Low-pass filter

Code	Constant	Value type
120	amplGetFiltersCount	Integer
121	amplGetFilterIndex	Integer
122	amplGetFilterStringAt	String
123	amplGetFilterValueAt	Double
124	amplSetFilterIndex	Integer
125	amplSetFilterString	String
126	amplSetFilterValue	Double

High-pass filter

Code	Constant	Value type
130	amplGetHPFiltersCount	Integer
131	amplGetHPFilterIndex	Integer
132	amplGetHPFilterStringAt	String
133	amplGetHPFilterValueAt	Double
134	amplSetHPFilterIndex	Integer
135	amplSetHPFilterString	String
136	amplSetHPFilterValue	Double

Input type

Code	Constant	Value type
140	amplGetInputTypesCount	Integer
141	amplGetInputTypeIndex	Integer
142	amplGetInputTypeStringAt	String
143	amplGetInputTypeValueAt	String
144	amplSetInputTypeIndex	Integer
145	amplSetInputTypeString	String
146	amplSetInputTypeValue	String

External shunt

Code	Constant	Value type
150	amplGetExtShuntsCount	Integer
151	amplGetExtShuntIndex	Integer
152	amplGetExtShuntStringAt	String
153	amplGetExtShuntValueResAt	Double
154	amplGetExtShuntValuePowAt	Double
155	amplGetExtShuntValueImaxAt	Double
156	amplSetExtShuntIndex	Integer
157	amplSetExtShuntString	String
158	amplSetExtShuntCustRes	Double
159	amplSetExtShuntCustPow	Double

PAD range

Code	Constant	Value type
160	amplGetPADRangeCode	Integer
161	amplGetPADRangeIndex	Integer
162	amplSetPADRangeCode	Integer

Excitation

Code	Constant	Value type
170	amplGetExcCount	Integer
171	amplGetExcIndex	Integer
172	amplGetExcStringAt	String
173	amplGetExcValueAt	Double
174	amplSetExcIndex	Integer
175	amplSetExcString	String
176	amplSetExcValue	Double
177	amplGetExcInfo	String
178	amplGetExc	Double

Excitation unit and type

Code	Constant	Value type
180	amplGetExcUnitCount	Integer
181	amplGetExcUnitIndex	Integer
182	amplGetExcUnitAt	String
183	amplSetExcUnitIndex	Integer
184	amplSetExcUnit	String
185	amplGetExcTypeCount	Integer
186	amplGetExcTypeAt	String
187	amplSetExcType	String
188	amplSetExcSense	String

Output maths and sensor offset

Code	Constant	Value type
190	amplSetOutputMathEnabled	Boolean
191	amplSetOutputMathOffset	Double
192	amplSetOutputMathDisableAllCh	—
193	amplGetSupportsSensorOffset	Boolean
194	amplGetSensorOffsetValue_Percent	Double
195	amplGetSensorOffsetValue_mVperV	Double
196	amplGetOutputMathEnabled	—
197	amplGetOutputMathChannels	—
198	amplGetOutputMathOffset	—
199	amplGetOutputMathAutoscale	—

Bridge type and shunt

Code	Constant	Value type
200	amplGetBridgeTypesCount	—
201	amplGetBridgeTypeIndex	—
202	amplSetBridgeTypeIndex	—
203	amplGetBridgeShuntsCount	Integer
204	amplGetBridgeShuntIndex	Integer
205	amplSetBridgeShuntIndex	Integer
206	amplSetBridgeShuntOnOff	Boolean
207	amplSetBridgeOhm	Double

Short, range limits and disabling

Code	Constant	Value type
210	amplSetShort	—
211	amplForceSetShort	—
212	amplGetRangeMax	Double
213	amplGetRangeMin	Double
214	amplGetRangeUnit	Double
215	amplGetSupportsDisabled	Boolean
216	amplGetDisabled	Boolean
217	amplSetDisabled	Boolean

Bridge readback

Code	Constant	Value type
220	amplSupportsBridgeResistance	—
221	amplGetBridgeResistance	—
222	amplGetBridgeVoltage	—
223	amplGetBridgeCurrent	—
224	amplGetSensorStatus	Integer
225	amplGetExcSenseCurrent	—
226	amplGetExcSenseVoltage	—

Bridge wiring

Code	Constant	Value type
230	amplSetRawBrDACOffset	raw calibration value
231	amplGetBridgeWiresCount	Integer
232	amplGetBridgeWireIndex	Integer
233	amplSetBridgeWireIndex	Integer
234	amplSetBridgeLegPosIndex	Integer
235	amplGetBridgeLegPosIndex	—
236	amplGetBridgeLegPosCount	—

Overload

Code	Constant	Value type
240	amplGetOverloadStatus	—

Charge generator

Code	Constant	Value type
250	amplDoChargeGenerator	—
251	amplDoChargeReset	—

Module status

Code	Constant	Value type
260	amplGetMCTSStatusShown	—
261	amplGetMCTSStatus	—

Connection type and TEDS

Code	Constant	Value type
`270`	`amplGetConnectionTypesCount`	—
`271`	`amplGetConnectionTypeIndex`	—
`272`	`amplSetConnectionTypeIndex`	—
`273`	`amplTedsCustomSendCmdRead`	—

Commands

Code	Constant	Effect
`10000`	`amplUpdateAmpl`	Applies pending settings to the amplifier. Includes `amplPrepareScaling`.
`10001`	`amplPrepareScaling`	Recomputes the scaling from the current settings.
`10002`	`amplReadDAQPSensorOffset`	Re-reads the sensor offset from the module.

IApp

The DewesoftX application itself, and the entry point to every other interface.

Obtain it as described in Connecting, then reach everything else from its properties — `Data` for channels and measurement data, `StoreEngine` for storing, `LoadEngine` for data files, and so on.

PYTHON

```python
app = win32com.client.Dispatch("DEWESoft.App")
app.Init()
print(app.Version, app.Data.UsedChannels.Count)
```

C#

```csharp
var type = Type.GetTypeFromProgID("DEWESoft.App")!;
dynamic app = Activator.CreateInstance(type)!;
app.Init();
Console.WriteLine($"{app.Version} {app.Data.UsedChannels.Count}");
```

Starting up

Init

Method

Initialises DewesoftX. **Call this before any other member.**

> **`Init` is a hard precondition**
>
> Call it before you touch anything else. Reads that come first fail, and can leave the instance unusable.
>
> It takes several seconds — 6 to 8 s on a typical setup — because it loads the hardware configuration, so allow for that in any timeout. `ActualRunMode` changes from `0` to `1` once it succeeds.

`IniFileDir` may be set before calling `Init`; almost nothing else may be touched.

Enabled

Property — read/write — `Boolean`

Whether the DewesoftX user interface accepts input. Setting it to `False` leaves the window visible but ignores mouse and keyboard, which is what you want while a script is driving the application.

Visible

Property — read/write — `Boolean`

Whether the DewesoftX window is shown. An instance launched by a client starts hidden, so set this to `True` if you want to see it.

Version

Property — read-only — `String`

The version as displayed to users, for example:

```
2026.4 (DEV-260819.0840) (64-bit)
```

For values you can compare against, use `GetDewesoftVersion`.

GetDewesoftVersion

Method

The application version as separate numbers.

Parameter	Type	Direction	Description
`Super`	`Integer`	out	Super version number.
`Major`	`Integer`	out	Major version number.
`Minor`	`Integer`	out	Minor version number.
`Build`	`Integer`	out	Build number.
`State`	`Byte`	out	`0` release, `1` alpha, `2` beta, `3` release candidate.

PYTHON

```
super_, major, minor, build, state = app.GetDewesoftVersion()
```

C#

```
app.GetDewesoftVersion(out int super, out int major,
                       out int minor, out int build, out byte state);
```

GetInterfaceVersion

Method

The version of the COM interface itself, which is what to test against when a member may not exist in every build you support.

Parameter	Type	Direction	Description
`Major`	`Integer`	out	Major version number.
`Minor`	`Integer`	out	Minor version number.
`Revision`	`Integer`	out	Revision of the type library.

`Revision` increases whenever the interface gains members, so it is the value to gate on:

PYTHON

```python
major, minor, revision = app.GetInterfaceVersion()
if revision >= 469:
    ...   # safe to call a member introduced at revision 469
```

C#

```csharp
app.GetInterfaceVersion(out int major, out int minor, out int revision);
if (revision >= 469)
{
    // safe to call a member introduced at revision 469
}
```

IsSimulationMode

Property — read-only — `Boolean`

`True` when the operation mode is *simulation* — the setting under **Settings** → **Devices** → **operation mode**, stored with the project. DewesoftX then runs against a simulation device instead of real hardware. Simulated channels still exist and still produce data, so a client cannot tell from the data alone.

This is the application's operation mode, not a driver-level simulated-channel option. The three operation modes are real measurement, simulation and analysis, and this property is `True` only for simulation — so it is `False` while the operation mode is analysis.

RegType

Property — read-only — `Integer`

The licence in use. Negative values report a licensing problem rather than an edition:

Value	Meaning	Value	Meaning
`-9`	licence expired	`0`	evaluation
`-8`	PCG missing	`1`	lite
`-7`	option missing	`2`	standard
`-6`	PCG changed	`3`	professional
`-5`	options changed	`4`	DSA
`-4`	not found	`5`	enterprise
`-3`	not supported	`6`	automotive
`-2`	trial expired	`9`	base
`-1`	demo	`10`	bundle
		`11`	simulation

Treat any negative value as "not licensed to run" rather than testing for a specific one.

GetSelectedLanguageIso639_1

Method — returns `String`

The selected interface language as a two-letter ISO 639-1 code, or `default` when no explicit language has been chosen — so a caller must handle a value that is not a language code.

Modes and navigation

DewesoftX is in one of two modes, and many members require a particular one.

ActualRunMode

Property — read-only — `Integer`

Which mode the application is in: `0` none, `1` measure, `2` analysis. It reads `0` before `Init` completes, and `Stop` returns it to `0`.

Measure

Method

Switches to measure mode — the equivalent of pressing **Acquisition**.

Analyze

Method

Switches to analysis mode — the equivalent of pressing **Analysis**.

SetupScreen

Method

Switches to the **Config** tab. DewesoftX must already be in measure mode.

GoToInstruments

Method

Switches to the **Live view** tab.

IsSetupMode

Property — read-only — `Boolean`

`True` while the **Config** tab is showing.

SetMainToolBar

Method

Selects a main toolbar tab and a button on it — the general way to reach any part of the interface.

Parameter	Type	Direction	Description
`TabName`	`String`	in	Toolbar tab name, for example `Ch. Setup` or `Measure`.
`ButtonName`	`String`	in	Button on that tab.

Which tabs and buttons exist depends on the current mode. Use the **English** names regardless of the selected interface language.

SetInstrument

Method

Shows a display screen by index, counting from `0`. Valid in measure mode.

Parameter	Type	Direction	Description
`Id`	`Integer`	in	Screen index.

Screens are user-defined and can be added, removed or renamed, so an index is only meaningful for a setup you control. Enumerate `Screens` otherwise.

SetScreenIndex

Method

Selects a sub-screen of the current screen. Call `SetInstrument` first.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Sub-screen index, counting from `0`.

ActiveScreen

Property — read-only — `Integer`

The index of the screen currently shown.

MenuClick

Method

Activates a main menu item.

Parameter	Type	Direction	Description
`Item`	`MenuItems`	in	The menu item to click.

UpdateSetupScreen

Method

Redraws the active **Config** page, after changes made through the interface that it would not otherwise notice.

SetFullScreen

Method

Parameter	Type	Direction	Description
`Full`	`Boolean`	in	`True` for full screen, `False` for a normal window.

ShowSensorEditor

Method

Opens the analog sensor editor — **Editors** → **Analog sensors**.

MeasureMenuIndex

Property — read-only — `Integer`

Which page of **Measure** is showing: `0` none, `1` Config, `2` Live view while measuring, `3` Live view stopped, `4` Setup files.

AnalyseMenuIndex

Property — read-only — `Integer`

Which page of **Analyze** is showing: `0` none, `1` Setup, `2` Review, `3` Export, `4` Print, `5` Data files.

Both indexes are reported independently of which section is actually active, so a non-zero `AnalyseMenuIndex` does not by itself mean DewesoftX is in analysis mode. Check `ActualRunMode` as well.

Acquisition and storing

Acquisition and storing are separate concerns: DewesoftX can acquire without storing, and `Acquiring` being `True` does not mean data is being written to a file.

Start

Method — returns `Boolean`

Switches to measure mode and starts acquisition. Returns `True` if acquisition started.

Stop

Method

Stops storing and acquisition, returning `ActualRunMode` to `0`.

Acquiring

Property — read-only — `Boolean`

`True` while data is being acquired. Says nothing about whether it is being stored.

IsAcqRunning

Property — read-only — `Boolean`

`True` while acquisition is running, including in **Config** and **Design**, where live values are shown.

StartStoring

Method

Starts storing, or arms the trigger if one is configured. Requires measure mode.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Target file. Without a path, the data directory is used.

`FileName` is **not** optional — omitting it fails with *"Invalid number of parameters"*, in late-bound and early-bound clients alike. Pass a name and read it back from `UsedDatafile`.

StopStoring

Method

Stops storing but leaves acquisition running — unlike `Stop`, which ends both.

PauseStoring

Method

Pauses storing. Requires measure mode with storing active.

ResumeStoring

Method

Resumes storing after `PauseStoring`.

DisableStoring

Property — read/write — `Boolean`

Hides the **Store** button, so a user cannot start storing by hand while a client is in control.

ManualStart

Method

Fires a manual start trigger. Requires measure mode with an armed trigger.

ManualStop

Method

Fires a manual stop trigger. Requires measure mode with a trigger currently active.

AlwaysEnableTrigger

Property — read/write — `Boolean`

Evaluates triggers even when storing is not armed, for cases where something other than the store engine consumes trigger events.

SetStoreMode

Method

Parameter	Type	Direction	Description
`Mode`	`Integer`	in	`0` always fast, `1` always slow, `2` fast on trigger, `3` fast on trigger and slow otherwise.

Modes `1` and `3` use the `ReducedRate`.

SetFreezeMode

Method

Freezes or unfreezes the display. Acquisition continues either way.

Parameter	Type	Direction	Description
`Enabled`	`Boolean`	in	`True` freezes the display.

DataLost

Property — read-only — `Boolean`

`True` if data was lost during the measurement — acquired faster than it could be processed. Worth checking at the end of an unattended run.

Sample rates and timing

MeasureSampleRateEx

Property — read/write — `Double`

The acquisition rate used in measure mode, in Hz.

> **Only writable from the Config tab**
>
> DewesoftX must be in measure mode **and** on the **Config** tab, or writing this is silently ignored — no error is raised.

MeasureSampleRate

Property — read/write — `Integer`

As `MeasureSampleRateEx` but truncated to a whole number. Prefer the `Ex` form.

SetupSampleRate

Property — read/write — `Integer`

The slower rate used for live values while in **Config**. DewesoftX must not be acquiring when this is set.

ReducedRate

Property — read/write — `Single`

The reduced storing rate, in Hz, used by the slow store modes.

TimerInterval

Property — read/write — `Integer`

The interval in milliseconds at which DewesoftX runs its acquisition loop, and therefore how often new data becomes available. Also configurable in **Settings**.

This is a requested interval, not a guarantee. Never rely on calls arriving exactly this far apart, nor on the first call arriving after exactly one interval.

FixedExternalClock

Property — read/write — `Boolean`

`False` treats the external clock as variable and related to revolutions per second; `True` treats it as a fixed rate, such as from a function generator.

GetAcquisitionLoopRate

Method — returns `Double`

The acquisition loop rate in Hz — the same setting as `TimerInterval` expressed as a frequency rather than a period, clamped to 1–1000 Hz.

Rounded to whole hertz, so it is not an exact reciprocal of the interval: an interval of 3 ms reports 333 Hz.

GetSamplesEx

Method — returns `Integer`

Calculates the block size, in samples, that DewesoftX *would* use for a channel running at a given rate, without changing anything. Use it to size buffers before committing to a sample rate.

Parameter	Type	Direction	Description
`SampleRate`	`Double`	in	Rate to calculate for, in Hz.
`SetupMode`	`Boolean`	in	`True` for the setup-mode block size, `False` for measure mode.
`Level`	`Integer`	in	Multiplies the result by 10 to this power. `0` returns the base block size.

The measure-mode figure accounts for the sample-rate-divider LCM, the calculation delay and any pre-trigger time, and is capped by the buffer size. Contrast `IData.Samples`, which is the block size actually in force.

GetTimingType

Method — returns `TimingType`

Where the absolute clock comes from: `0` none, `1` a timing device, `2` a plug-in, `3` the DAQ device.

The value can also be `4` (a master-clock camera) or `5` (an openDAQ master clock), neither of which is named in the enumeration. Treat anything above `3` as "some other timing source" rather than an error.

Setups and projects

LoadSetup

Method

Loads a setup file.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Setup file. Without a path the setup directory is used; without an extension `d7s` is assumed.

SaveSetup

Method

Saves the setup.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Target file. **Empty overwrites the currently loaded setup.**

NewSetup

Method

Starts a new, empty setup.

LoadSetupFromXML

Method

Loads a channel setup from XML held in memory rather than from a file.

Parameter	Type	Direction	Description
`XML`	`String`	in	The channel setup as XML.

SaveSetupToXML

Method

Returns the current channel setup as XML. With `LoadSetupFromXML` this lets a client keep setups somewhere other than the filesystem.

Parameter	Type	Direction	Description
`XML`	`String`	out	The channel setup as XML.

LoadModuleSetup

Method

Loads only the analog and PAD channel groups from a setup file, leaving screens and triggers alone. `LoadSetup` is normally what you want.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Setup file including path.

LoadDisplaySetup

Method

Loads only the display setup.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Display setup file including path.

LoadProject

Method

Opens a project.

Parameter	Type	Direction	Description
`Name`	`String`	in	Project name, without path or extension.

LoadSequence

Method

Loads a sequence file.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Sequence file. Without a path the setup directory is used; without an extension `d7t` is assumed.

UsedSetupfile

Property — read-only — `String`

Full path of the setup currently loaded, or empty if none is.

Data files

LoadFile

Method

Loads a measurement file for analysis.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Data file. Without a path the project's data folder is used.

If the name is wrong or empty, DewesoftX opens a file dialog and waits — which hangs an unattended script. Verify the path before calling.

UsedDatafile

Property — read/write — `String`

Full path of the data file that acquired data is written to.

ImportDataFile

Method

Imports another file into the currently open data file.

Parameter	Type	Direction	Description
`Filename`	`String`	in	File to import.
`ReferenceType`	`Integer`	in	Time base: `0` relative, `1` absolute, `2` first trigger event, `3` channel value, `4` channel value, destination only.
`Params`	`Variant`	in	Additional parameters for the chosen time base.
`ChannelFilter`	`String`	in	Restricts which channels are imported.

SetHeaderData

Method

Sets one field of the measurement file header. `GlobalHeader` gives fuller access to the same data.

Parameter	Type	Direction	Description
`Caption`	`String`	in	Which header field to set.
`Header`	`String`	in	Value to set it to.

SaveChangesToDataFile

Method

Saves the loaded data file in place with no prompt, merging the data of offline and calculated channels into it along with the setup, events and last screenshot.

> **Raises if no file is loaded**
>
> Calling this with no data file open raises `File not opened`. Check first.
>
> It also fails quietly in the other direction: if the file is locked or write-protected, DewesoftX shows a message box and nothing is saved, but no error reaches the caller. On a large file, merging calculated channels can take a long time.

AskSaveDataFile

Method — returns `Integer`

Shows the "save changes to the data file?" prompt and, if the user agrees, writes the setup and calculated-channel data back into the loaded file.

Parameter	Type	Direction	Description
`AskWithoutAnySettingChanged`	`Boolean`	in	`True` prompts even when nothing has changed. `False` prompts only when there are pending changes.

The return value is the dialog result:

Value	Meaning
`0`	no prompt was shown
`6`	user chose Yes; the file was saved
`7`	user chose No
`2`	user cancelled

Requires analysis mode with a file open. A result of `6` does not guarantee the save succeeded — a locked or write-protected file reports the failure in a message box and still returns `6`. Being modal, this is not suitable for unattended use; prefer `SaveChangesToDataFile`.

Exporting and printing

Exports are identified by name or by position in a list that depends on which export add-ons are installed. Read the list at runtime rather than hard-coding a number.

GetExportList

Method — returns `String`

The available exports, one per line, separated by carriage-return/line-feed:

```
Flexpro (*.fpd)
Excel (*.xlsx)
DIAdem (*.dat)
Matlab (*.mat)
...
```

Each line is a name accepted by `ExportDataName`.

> **A line's position is not the `ExportType` index**
>
> The list omits the UNV format, which is `ExportType` `4`, so every entry after it sits one place below its numeric index. Using a line number as the `ExportType` for `ExportDataEx` selects the wrong export.
>
> Pass the name to `ExportDataName` instead. The list also covers only the built-in exports — custom exports supplied by add-ons do not appear.

ExportDataName

Method — returns `Integer`

Exports the loaded measurement file, selecting the export **by name**. Preferable to `ExportDataEx`, because a name does not shift when the set of installed exports changes.

Parameter	Type	Direction	Description
`ExportName`	`String`	in	Export name, as returned by `GetExportList`.
`TimeAxis`	`Integer`	in	Reduced or complex data: `0` relative, `1` absolute, `2` trigger, `3` pre-trigger time. For CEA: `0` degrees by cycle, `1` degrees continuous, `2` cycles, `3` samples.
`ExportDataType`	`Integer`	in	`0` full speed, `1` reduced, `3` CEA.
`ExportOption`	`Integer`	in	Bit mask, meaning depends on `ExportDataType` — see below.
`FileName`	`String`	in	Target file.
`OutputString`	`String`	in/out	Result text from the export. Declared in/out, so pass a placeholder in — an empty string will do.

Returns `0` on success; a non-zero result is an error code, with `OutputString` carrying the explanation. `ExportName` is matched exactly, so trim the line from `GetExportList` — its lines end `\r\n`, and a trailing carriage return is rejected as an unknown name.

Called outside analysis mode, or asked for `ExportDataType` `0` on a file holding only reduced data, it puts a message box on the DewesoftX window and waits for someone to dismiss it. Call `Analyze` first and match `ExportDataType` to what the file actually contains.

`ExportOption` combines flags by addition:

Data type	Flags
Reduced	`1` min, `2` max, `4` average, `8` RMS
Complex	`1` real, `2` imaginary, `4` amplitude, `8` phase
CEA	`1` cycle, `2` once per cycle, `4` averaged cycle

So minimum and maximum together is `1 + 2 = 3`.

ExportDataEx

Method

Exports the loaded measurement file, selecting the export by number. Parameters match `ExportDataName` except for `ExportType`.

Parameter	Type	Direction	Description
`ExportType`	`Integer`	in	Export number. **Not** a line number from `GetExportList`. Custom exports are numbered from `-2` downwards.
`TimeAxis`	`Integer`	in	As for `ExportDataName`.
`ExportDataTyp e`	`Integer`	in	As for `ExportDataName`.
`ExportOption s`	`Integer`	in	As for `ExportDataName`.
`FileName`	`String`	in	Target file. Missing extension, path or name each fall back to a default.

The file must be open in analysis mode.

This returns nothing, so it cannot report failure. Called outside analysis mode, or asked for `ExportDataType` `0` on a file holding only reduced data, it puts a message box on the DewesoftX window and waits for someone to dismiss it. Prefer `ExportDataName`, which at least returns an error code.

ExportData

Method

A shorter form of `ExportDataEx` taking only the export type, time axis and file name.

ExportToDewesoft

Method

Exports the selected data region to a new DewesoftX data file — `.dxd`, or `.dxz` if the name ends in `dxz`. Only channels marked for export are written, and the region exported is the current selection rather than the whole file. An existing target is overwritten without asking. Requires analysis mode.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Target file **including a three-letter extension**. A bare name goes to the export directory; a relative or absolute path is used as given, and missing directories are created.

Always supply an extension

The last three characters of the name are replaced with `dxd`, so a name without an extension is silently truncated — `Test` becomes `Tdxd`. Passing an empty string suffers the same truncation on the derived name.

PrintScreenEx

Method

Prints or saves the current screen.

Parameter	Type	Direction	Description
`ShowDialog`	`Boolean`	in	`True` shows printer settings; `FileName` must then be empty.
`FileName`	`String`	in	Output file. Empty opens a file dialog, but only if `ShowDialog` is `True`. A relative path resolves under the export directory.
`PrinterName`	`String`	in	Printer to use. Empty uses the default. For a physical printer, `FileName` is ignored.

PrintScreen

Method

Prints the current screen to the default printer. Requires analysis mode.

Parameter	Type	Direction	Description
`ShowDialog`	`Boolean`	in	Whether to show the print dialog.

Directories

GetSpecDir

Method — returns `String`

The full path of one of the directories DewesoftX uses.

Parameter	Type	Direction	Description
`DirType`	`SpecDirectory`	in	Which directory — see below.

Value	Directory	Value	Directory
`0`	system	`7`	ini file
`1`	logs	`8`	add-ons
`2`	scripts	`9`	application
`3`	temp	`10`	sequence base
`4`	data	`11`	translations
`5`	setups	`12`	automatic exports
`6`	exports	`13`	sequence files

Paths depend on the installation and the selected project, so always ask rather than assuming a layout.

MainDataDir

Property — read-only — `String`

The measurement data directory — the same as `GetSpecDir(4)`.

SetMainDataDir

Method

Sets the measurement data directory.

Parameter	Type	Direction	Description
`DataDir`	`String`	in	Path where data files are stored.

ChangeDataFolder

Method

Changes the data folder.

Parameter	Type	Direction	Description
`FolderName`	`String`	in	New data folder.

ChangeSetupFolder

Method

Changes the setup folder.

Parameter	Type	Direction	Description
`FolderName`	`String`	in	New setup folder.

IniFileDir

Property — read/write — `String`

Directory holding the DewesoftX ini file. This is the one property that may be set before `Init`, so a client can point DewesoftX at its own configuration.

Window and appearance

Top

Property — read/write — `Integer`

Distance in pixels from the top of the screen to the top of the window.

Left

Property — read/write — `Integer`

Distance in pixels from the left of the screen to the left of the window.

Width

Property — read/write — `Integer`

Window width in pixels.

Height

Property — read/write — `Integer`

Window height in pixels.

StayOnTop

Property — read/write — `Boolean`

Keeps the DewesoftX window above other applications even when it does not have focus.

ShowStyle

Property — read/write — `Integer`

How much of the menu bar is shown: `0` hides it entirely, `1` shows the buttons only, `2` shows the menu and the buttons.

ShowPropertyFrame

Property — read/write — `Boolean`

Shows or hides the property panel and channel list while measuring.

ShowSROptions

Property — read/write — `Boolean`

Shows or hides the sample rate panel in **Config**.

ShowStoreOptions

Property — read/write — `Boolean`

Shows or hides the **Storing** options in **Config**. Hiding them lets a user edit header fields without being able to change the file name.

ShowInstrumentsInFullScreen

Property — read/write — `Boolean`

Whether the displays toolbar remains visible in full screen.

MainWindowHandle

Property — read-only — `Integer`

Window handle of the DewesoftX main form, for a client that needs to position or parent windows around it.

Parent

Property — read/write — `Integer`

Window handle of the parent form, so DewesoftX can be embedded in another application's window.

DisableKeyboardShortcuts

Property — read/write — `Boolean`

Disables keyboard shortcuts, so a user cannot change state behind a script's back with, for example, Ctrl+F for full screen.

SendKey

Method

Sends a key stroke to DewesoftX.

Parameter	Type	Direction	Description
`Key`	`Cardinal`	in	Key code.

LastKey

Property — read-only — `Integer`

Key code of the last shortcut key pressed.

DialogHandle

Property — read/write — `Cardinal`

The window handle of a dialog DewesoftX does not own, so that keyboard messages are routed to it. Set it while showing your own modal window and Tab, arrow keys, Escape and mnemonics work inside that window.

> **Reset it to zero when the dialog closes**
>
> While this is non-zero, every application message is offered to that handle first — so a stale handle breaks keyboard input across the whole of DewesoftX.

Hardware

HardwareSetup

Method

Opens the **Settings** dialog.

Parameter	Type	Direction	Description
`Plugins`	`Boolean`	in	Ignored.

> **Opens a modal dialog**
>
> Settings is modal, so an unattended script that calls this stops until someone closes the dialog. The `Plugins` parameter is not read at all — rescanning for extensions is not triggered by this call.

UpdateHardwareSetup

Method

Re-applies the device configuration, as after a change in **Settings** → **Devices**.

FirstScanDonePercent

Method — returns `Single`

Progress of the amplifier scan that runs when a setup is loaded, as a percentage. Poll it to wait for the scan rather than sleeping for a fixed time.

StartModuleScan

Method

Resumes scanning of hardware modules.

StopModuleScan

Method

Stops scanning of hardware modules.

ZeroAllAutoChannels

Method

Zeroes every channel that has auto-zero enabled.

Parameter	Type	Direction	Description
`Zero`	`Boolean`	in	`True` applies the zero offset; `False` clears it again.

In measure mode this is only available while not storing.

ZeroCntChannels

Method

Zeroes the counter channels that are configured to allow reset during a measurement.

LoadDBC

Method

Loads a DBC file onto a CAN port.

Parameter	Type	Direction	Description
`PortNo`	`Integer`	in	CAN port, counting from `0`.
`FileName`	`String`	in	DBC file. Without a path the setup directory is used.

ZeroAutoChannelsByGroup

Method

Applies or clears the zero offset on analog input channels, optionally limited to one action group.

Parameter	Type	Direction	Description
`Zero`	`Boolean`	in	`True` captures the zero offset; `False` **clears** previously captured offsets.
`GroupID`	`Byte`	in	`0` for all channels, otherwise the action group a channel is assigned to in **Config**.

`Zero = False` is not a no-op — it is the "reset zero" action.

Only effective in measure mode on the **Config** tab, or while stopped; it exits silently in any other state, including while measuring. The action is forwarded to connected measurement units.

GetMultiDevice

Method — returns `IMultiDevice`

A device-family container. The list is fixed at start-up with two entries: index `0` is the real hardware, index `1` the simulation device.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	`0` or `1`.

An out-of-range index returns nothing rather than raising, and there is no count property — check the result.

GetSystemDevicesWithParentIds

Method — returns `Variant`

The USB parent identifiers of detected system-box devices, as a zero-based array of unsigned integers, for reconstructing which hub each device hangs off.

Only devices that support USB topology reporting appear, so the indexes do not line up with any other device enumeration. Returns an empty array when nothing matches.

FillModuleChannelList

Method

Clears the supplied list and fills it with the channels of one channel group belonging to one hardware module.

Parameter	Type	Direction	Description
`SerialNumber`	`String`	in	Module, motherboard or system serial number, or `*` to match any module.
`GroupID`	`Integer`	in	Channel group. In practice only counter (`6`) and DAQ-out (`17`) are implemented.
`List`	`IChannelListEx`	in/out	List to fill — obtain it from `CreateChannelList`.

Any other `GroupID`, or non-Dewesoft DAQ hardware, leaves the list empty. For counters, only output channels with `Used` set are added. The list is cleared on entry, so its previous contents are lost.

SetVariableCntLevels

Method

Sets the custom low and high trigger levels on one variable-trigger input pin of one counter channel, and pushes the setting to the device immediately.

Parameter	Type	Direction	Description
`CombinedIdx`	`Integer`	in	`counterIndex * 3 + pinIndex`, where pin `0` is Source, `1` Gate, `2` Aux.
`LevelLow`	`Double`	in	Low level, **in millivolts**.
`LevelHigh`	`Double`	in	High level, **in millivolts**.

Levels are millivolts, not volts. If `LevelHigh` is below `LevelLow` the levels are ignored, silently. As a side effect the pin's trigger type is forced to the first preset and its coupling to DC — and the first preset is not guaranteed to honour custom levels on every device. The call does nothing if there are no counter channels or the index is out of range.

BeginOpenDaqDeviceUpdate

Method

Opens a property-update transaction on an openDAQ device, so that changes made until `EndOpenDaqDeviceUpdate` are applied as one batch instead of one property at a time.

Parameter	Type	Direction	Description
`SerialNumber`	`String`	in	Serial number of the connected device.

EndOpenDaqDeviceUpdate

Method

Closes the transaction opened by `BeginOpenDaqDeviceUpdate` and applies the batch.

Parameter	Type	Direction	Description
`SerialNumber`	`String`	in	Serial number of the connected device.

> **The calls must balance**
>
> They are counted, so they nest — but only returning to zero applies the batch. An unmatched `Begin` leaves the device in update state indefinitely: the queued changes never apply, and code that checks whether any device is mid-update behaves as though one still is. An extra `End` drives the count negative, and the next balanced pair then fails to apply.
>
> Both calls do nothing if no connected device has that serial number, or if the device is the local system device or is locked — so a typo in the serial number is silent.

UpdateTopology

Method

Asks DewesoftX to rescan for devices and refresh the device topology shown in the settings dialog. It posts a message rather than doing the work inline, so it returns immediately and is safe to call from a background thread.

Only has an effect while the Settings window is open; otherwise it is a silent no-op. It refreshes that dialog's view rather than rescanning hardware generally.

Physical units

GetPhysicalUnits

Method — returns `Variant`

The unit labels available for a physical quantity, as a zero-based array of strings — the same list, in the same order, that the unit drop-downs show.

Parameter	Type	Direction	Description
`PhysicalQuantity`	`String`	in	Internal quantity identifier, such as `TEMPERATURE` or `VOLTAGE` — not a translated display name.

The list is filtered to the active unit system (metric, imperial or custom), so it depends on a global setting. An unknown quantity yields an empty array rather than an error.

GetPhysicalUnitScaleAndOffset

Method

The scale factor and offset of one unit within a physical quantity, relative to that quantity's reference unit.

Parameter	Type	Direction	Description
`PhysicalQuantity`	`String`	in	Internal quantity identifier.
`UnitLabel`	`String`	in	Unit label, as returned by `GetPhysicalUnits`.
`Scale`	`Double`	out	Scale factor.
`Offset`	`Double`	out	Offset. **Always** `0` — see below.

Not sufficient for affine units

`Offset` is always `0`. For units that need an additive term — temperatures above all — that term is held separately and is not exposed here, so scale alone cannot convert °C or °F correctly.

An unknown quantity or label is not an error either: you get `Scale = 1` and `Offset = 0`, which is indistinguishable from a genuine identity conversion.

The scope trigger

Used in analysis mode to find a trigger in a loaded file. Call them in order: `SetScopeParams`, `SetScopeUsed`, `InitScopeTrig`, then `CalcScopeTrig`.

SetScopeParams

Method — returns `Integer`

Sets the trigger channel, level and pre/post-trigger times.

Parameter	Type	Direction	Description
`PreTime`	`Double`	in	Pre-trigger time in milliseconds.
`PostTime`	`Double`	in	Post-trigger time in milliseconds.
`Channel`	`IChannel`	in	Channel to trigger on.
`Level`	`Single`	in	Trigger level.

SetScopeUsed

Method

Enables or disables the scope trigger.

Parameter	Type	Direction	Description
`Value`	`Boolean`	in	`True` enables it.

InitScopeTrig

Method

Sets the region to search for a trigger.

Parameter	Type	Direction	Description
`Start`	`T_RecordPosition`	in	Start position.
`Stop`	`T_RecordPosition`	in	End position.

CalcScopeTrig

Method — returns `Boolean`

Searches for the trigger. Returns `True` if one was found, in which case the triggered data is loaded into the channel buffers.

Messages, logging and plug-ins

SuppressMessages

Property — read/write — `Boolean`

Suppresses error and warning dialogs, so they do not pop up and block an unattended run. The messages are still recorded; only the dialog is withheld. Set this on any client that runs without someone at the keyboard — it is what keeps a misplaced call from stalling the script behind a message box.

> **It is a counter, not a flag**
>
> Writing `True` increments an internal count and `False` decrements it; reading reports whether the count is above zero. So the writes must balance:
>
> - `True` twice then `False` once leaves suppression **on**.
> - One `False` more than you wrote `True` drives the count negative, and the next `True` only brings it back to zero — suppression stays **off**.
>
> Set it once after `Init` and leave it alone, or pair every `True` with exactly one `False`.

WriteErrorLog

Method

Writes a line to the DewesoftX error log, in the log directory reported by `GetSpecDir(1)`.

Parameter	Type	Direction	Description
`Str`	`String`	in	Message to log.

ReportMessage

Method

Adds a message to the DewesoftX report.

Parameter	Type	Direction	Description
`MsgType`	`ReportMessageType`	in	`0` error, `1` warning, `2` information, `3` developer.
`MsgSource`	`String`	in	Where the message came from.
`Msg`	`String`	in	Message text.

MainWndMessage

Method

Posts a message to the DewesoftX main thread, which is then delivered to every active plug-in through `IPlugin2.OnOleMsg`. Use it when a plug-in must signal DewesoftX from a thread other than the main one.

Parameter	Type	Direction	Description
`Msg`	`Integer`	in	Message identifier, meaningful to the receiving plug-in.
`WParam`	`Integer`	in	Message parameter.
`Wait`	`Boolean`	in	`True` blocks the calling thread until the message is handled.

NotifyTrackingChanged

Method

Reports a change in clock tracking. Only meaningful for a plug-in that provides the master clock.

Parameter	Type	Direction	Description
`Tracking`	`Boolean`	in	`False` means clock information was lost.
`TimeDiff`	`Double`	in	Difference between real time and device time. Only meaningful when `Tracking` is `True`.

WriteErrorMessage

Method

Writes a message into the DewesoftX title bar. Intended for debugging.

Parameter	Type	Direction	Description
`ErrorMsg`	`String`	in	Message text.

SendPluginCommand

Method — returns `Integer`

Sends a client-defined command to a loaded, active plug-in. The plug-in receives it through `IPlugin4.OnEvent`, with `CommandID` as the event id and `Input` as the input parameter; whatever it writes back comes out in `Output`.

Parameter	Type	Direction	Description
`PluginGUID`	`String`	in	GUID of the target plug-in.
`CommandID`	`Integer`	in	Command identifier. **Must be 1000 or above.**
`Input`	`Variant`	in	Payload for the plug-in.
`Output`	`Variant`	in/out	Reply from the plug-in. Pass an initialised variant.

Result	Meaning
`0`	delivered without error
`-1`	`PluginGUID` is not a valid GUID
`-2`	no active plug-in has that GUID
`-3`	the plug-in does not implement `IPlugin4`
`-4`	`CommandID` is below 1000
`-5`	the plug-in raised an exception

Identifiers below 1000 are reserved for the events DewesoftX itself dispatches through `OnEvent`, which is why they are rejected. The plug-in must be active, not merely installed.

SendMathCommand

Method — returns `Integer`

Sends a text command with a text payload to one math module, addressed by the name shown in the setup, and returns its reply.

Parameter	Type	Direction	Description
`MathName`	`String`	in	Name of the math module. Must be unique.
`Command`	`String`	in	Command to send.
`Input`	`String`	in	Payload for the command.
`Output`	`String`	in/out	Reply from the module.

Result	Meaning
`0`	handled successfully
`-1`	more than one module has that name
`-2`	no module has that name
`-3`	the module does not support that command
`-4`	the module raised an exception

Most math modules implement no commands at all, so `-3` is the ordinary answer rather than a sign of a mistake.

CreateChannelList

Method — returns `IChannelListEx`

Creates an empty, **writable** channel list for you to fill and pass to members that take one, such as `FillModuleChannelList`. It is reference-counted, so releasing it is the only cleanup needed.

The lists on `IData` are application-owned and read-only — adding to or clearing one raises `Channel list is read only`. This gives you a private list instead.

AskUserToDeleteDisplays

Method

Finds the display frames created from a given display template and, if any exist, asks the user whether to delete them. Intended for a plug-in whose channels have just changed in a way that leaves displays invalid.

Parameter	Type	Direction	Description
`PluginGUID`	`String`	in	GUID of the **display template**, despite the parameter name.

Does nothing unless DewesoftX is in **Config**. A malformed GUID raises an exception rather than being ignored.

The analysis cursors

LockCursors

Method

Positions and locks the two recorder cursors, so they stay pinned while the display scrolls or you change screens. Unlock them through `LockableCursors`.

Parameter	Type	Direction	Description
`StartPos`	`Double`	in	Position of the first cursor, in seconds from the start of the data. `-1` locks it where it already is.
`EndPos`	`Double`	in	Position of the second cursor, in seconds. `-1` locks it where it already is.

Because `-1` means "leave in place", −1 s is not a usable position. Positions are clamped to the loaded data range, and the current screen must contain a recorder display or nothing useful is locked.

Distributed measurement

NETMode

Property — read-only — `Integer`

The role in a distributed setup: `0` standalone, `1` slave measurement unit, `2` master measurement unit, `3` view client, `4` master client, `5` auto-detect client.

RemoteControlled

Property — read-only — `Boolean`

`True` when this instance is being controlled by another DewesoftX instance.

TimeFormat

Property — read/write — `TimeFormat`

How times are interpreted and displayed: `0` local, `1` UTC, `2` telemetry, `3` TAI.

Other interfaces

Each of these returns another interface. They are the route to everything DewesoftX exposes beyond the application itself.

Property	Returns	Covers
`Data`	`IData`	channels, sample rates, the analysis cursor
`StoreEngine`	`IStoreEngine`	storing
`LoadEngine`	`ILoadEngine`	data files
`Math`	`IMath`	mathematics
`OfflineCalc`	`IOfflineCalc`	offline recalculation
`Screens`	`IScreens`	display screens
`Trigger`	`ITrigger`	start and stop trigger conditions
`Alarms`	`IAlarms`	alarm conditions
`EventList`	`IEventList`	events recorded in the measurement
`GlobalHeader`	`IGlobalHeader`	the data file header
`MasterClock`	`IMasterClock`	the acquisition clock
`Timing`	`ITiming`	timing configuration
`CAN`	`ICAN`	the CAN bus
`Daq`	`IDaq`	the analog device
`DaqGroup`	`IDaqGroup`	analog channel group
`AmplInterfaces`	`IAmplInterfaces`	amplifiers
`AISetupScreen`	`IAISetupScreen`	the analog input page in Config
`AOGroup`	`IAOGroup`	analog output and the function generator
`PowerModules`	`IPowerModules`	power modules
`Video`	`IVideo`	video
`Sequencer`	`ISequencer`	test sequences

Property	Returns	Covers
`Report`	`IReport`	reporting
`ProjectManager`	`IProjectManager`	projects
`FileNameSettings`	`IFileNameSettings`	automatic file naming
`UserInterface`	`IUserInterface`	the user interface
`XMLHelper`	`IXMLHelper`	XML handling
`AcqLoop`	`IAcqLoop`	the acquisition loop
`LockableCursors`	`ILockableCursors`	lockable analysis cursors
`RemoteManager`	`IRemoteManager`	remote measurement units
`NTPClient`	`INTPClient`	NTP time synchronisation
`RTC`	`IRTC`	the real-time controller
`RTCore`	`IRTCore`	the real-time core

Data

Property — read-only — `IData`

The measurement data — channel lists, sample rates, acquisition progress and the analysis cursor. The most used member of this interface.

StoreEngine

Property — read-only — `IStoreEngine`

Storing configuration and control.

LoadEngine

Property — read-only — `ILoadEngine`

Loading and manipulating measurement files.

Analog output

AOGetManualAvail

Method — returns `Boolean`

Whether the function generator's manual start and stop is available in the current setup.

AOSetManual

Method

Starts the function generator output, when it is set to manual — the equivalent of pressing **Manual**.

Legacy members

These remain for compatibility and should not be used in new code.

Member	Instead use
`Modules`	`AmplInterfaces`
`ChangeComPort`	`AmplInterfaces`
`SendCommand`	`AmplInterfaces`
`ExecuteModulesFunction`	`AmplInterfaces`
`AveragedFFT`	an array channel
`AveragedCPB`	an array channel
`ConfigMode`	—
`HasFRF`	—
`SetRemoteMode`	—
`ShowCaptionBar`	—
`HideCaptionBar`	—

ChangeDaqType

Method

Not functional. Calling it always raises `ChangeDaqType not supported.` Change the analog device in **Settings → Devices** instead.

Parameter	Type	Direction	Description
`DaqType`	`Integer`	in	Ignored.

IAppEvents

The events DewesoftX raises. This is the App object's outgoing event sink: rather than obtaining it and calling it, you supply an object with these methods and DewesoftX invokes them as things happen.

See Receiving Events for how to attach a handler in Python and C#, which events fire when, and the overlap between `OnEvent` and the individual events.

Members

Member	Parameters	Raised when
`OnGetData`	—	New data is available. **Not delivered — see below.**
`OnStartStoring`	—	Storing has started.
`OnStopStoring`	—	Storing has stopped.
`OnTrigger`	`Mid`, `Dir`, `Time`	A trigger has fired.
`OnTriggerStop`	`Mid`, `Dir`, `Time`	A stop trigger has fired.
`OnAlarm`	`CondIndex`, `Status`	An alarm has changed state.
`OnDataLost`	—	Samples have been dropped. **Not delivered — see below.**
`OnException`	`Str`	Something went wrong inside DewesoftX.
`OnExit`	—	DewesoftX is closing.
`OnEvent`	`Reason`, `Param`	Any of the above, selected by `Reason` — plus stopping acquisition, which has no dedicated event.
`OnGetTime`	`TimeLo`, `TimeHi`	DewesoftX is asking for the current time. **Not delivered — see below.**
`OnMessageBox`	`Tex`, `Caption`, `MsgType`, `Result`	DewesoftX would otherwise show a message box.
`OnProgress`	`PercentDone`	A long operation is reporting progress.

`Mid` and `Dir` locate the trigger in the data; `Time` is when it fired.

Three of these are not delivered

`OnGetData`, `OnGetTime` and `OnDataLost` never reach a client: delivery fails with *"Interface not supported"*. Poll `IChannelConnection.NumValues` instead of `OnGetData`, and read `IApp.DataLost` instead of `OnDataLost`. See Receiving Events for the full picture.

`OnMessageBox` does not suppress the dialog

Handling it reports what DewesoftX was about to ask, but the dialog is still shown and the `Result` a handler writes is overwritten. Set `IApp.SuppressMessages` to stop dialogs appearing.

The parameter name `Tex` is a misspelling of `Text` in the type library, and is the name a handler must use.

IArrayInfo

The shape of an array channel — how many dimensions it has, how big each is, and what each axis means.

Get it from `IChannel.ArrayInfo`. It is **nothing** on a single-value channel, so test the result before using it — that is also how you tell an array channel from a scalar one.

Each axis is an `IAxisDef`.

PYTHON

```python
ai = channel.ArrayInfo
if ai is not None:
    print(ai.DimCount, "dimension(s)")
    for d in range(ai.DimCount):
        print(" ", ai.AxisDef(d).Name, ai.DimSizes(d))
```

C#

```csharp
dynamic ai = channel.ArrayInfo;
if (ai != null)
{
    Console.WriteLine($"{ai.DimCount} dimension(s)");
    for (int d = 0; d < (int)ai.DimCount; d++)
        Console.WriteLine($"  {ai.AxisDef[d].Name} {ai.DimSizes[d]}");
}
```

Shape

DimCount

Property — read/write — `Integer`

How many dimensions the array has: `1` for a spectrum, `2` for a matrix.

DimSizes

Property — read/write — `Integer`

The number of positions along one dimension.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Dimension, `0` to `DimCount - 1`.

AxisDef

Property — read-only — `IAxisDef`

The axis definition for one dimension.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Dimension, `0` to `DimCount - 1`.

Init

Method

Applies the shape you have set, allocating what the new `DimCount` and `DimSizes` call for. Call it after changing them.

Contents

ItemChannels

Property — read/write — `Boolean`

`True` when each position of the array is also published as a channel of its own, so a single element can be displayed or stored on its own.

NameArr

Property — read/write — `String`

The name of one element.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	The element's position.

ColorArr

Property — read/write — `Integer`

The colour of one element, as a BGR integer.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	The element's position.

SyncSource

Property — read-only — `ISyncSource`

The sample rate the array's axis is generated against.

IAveragedFFT

An averaged FFT or constant-percentage-bandwidth analysis, and the spectra it has accumulated.

No way to obtain one

Both accessors that return this interface — `IApp.AveragedFFT` and `IApp.AveragedCPB` — are stubs that always return nothing, whatever is configured. There is no other member anywhere that returns one, so nothing below is reachable from any client.

The implementation behind it is complete and correctly registered; only the two ways in are missing.

What it would give you

Recorded here so the interface is accounted for, and against the two accessors being implemented.

Member		Description
`AverageType`	property, read/write	How spectra are combined: `1` linear (running RMS), `2` exponential (weighted 9:1 towards history), `3` peak hold. Defaults to `1`.
`Window`	property, read/write	The window function — see the table below. Defaults to `0`, rectangular.
`Lines`	property, read/write	Number of FFT lines.
`AveCount`	property, read/write	How many spectra go into one average.
`Overlap`	property, read/write	Block overlap as a percentage. Reported truncated to a whole number.
`GetFFTData`	method	The averaged spectrum for one channel, with optional frequency weighting and a low-frequency cutoff.
`GetCPBData`	method	The same data as constant-percentage-bandwidth octave bands.
`GetCPBXData`	method	The band centre frequencies matching `GetCPBData`.
`GetData`	method	An alias — it calls `GetCPBData` with the same arguments.
`GetChannels`	method	The source channels, as an array.
`CalculateFromPos`	method	Whether enough data exists from a position to fill one average.

Window functions

Value	Window	Value	Window
`0`	rectangular	`5`	Blackman
`1`	Hann	`6`	exponential down
`2`	Hamming	`7`	transient
`3`	flat top	`8`	Blackman-Harris
`4`	triangle	`9`	Kaiser-Bessel

Frequency weighting

`GetFFTData`, `GetCPBData` and `GetCPBXData` all take a `Weighting`: `0` none, `1` A, `2` B, `3` C. D-weighting is not supported, and an unrecognised value attenuates the data to near zero rather than reporting an error.

`GetCPBData` divides octaves by its `OctaveDivider` — `3` for third-octave bands — and applies a Hann window internally whatever `Window` is set to.

`GetFFTData`'s cutoff argument is divided by 1000 before being matched against the line spacing, so it is in **millihertz**: pass `10000` to zero everything below 10 Hz.

IAxisDef

One axis of an array channel — what its positions mean and what they are called.

Get it from `IArrayInfo.AxisDef`.

An axis defines its positions in one of three ways, chosen by `AxisType`: as a start plus a step, as an explicit list of numbers, or as a list of labels. Which of the members below matter follows from that.

PYTHON

```python
ai = channel.ArrayInfo
axis = ai.AxisDef(0)
print(axis.Name, axis.Unit, axis.Size)
if axis.AxisType == 2:
    print("from", axis.StartValue, "step", axis.StepValue)
```

C#

```csharp
dynamic ai = channel.ArrayInfo;
dynamic axis = ai.AxisDef[0];
Console.WriteLine($"{axis.Name} {axis.Unit} {axis.Size}");
if ((int)axis.AxisType == 2)
    Console.WriteLine($"from {axis.StartValue} step {axis.StepValue}");
```

`_Unit` is spelled with a leading underscore in the type library because `Unit` is a reserved word in Object Pascal. Most languages, Python and C# included, accept it as `Unit`.

How the axis is defined

AxisType

Property — read/write — `Integer`

Value	Positions come from	Members that matter
`0`	a list of labels	`StringValues`
`1`	a list of numbers	`FloatValues`
`2`	a start and a step	`StartValue`, `StepValue`

An FFT's frequency axis is `2` — evenly spaced, so a start and a bin width describe it completely.

StartValue

Property — read/write — `Double`

The first position's value. Used when `AxisType` is `2`.

StepValue

Property — read/write — `Double`

The distance between positions. Used when `AxisType` is `2`.

FloatValues

Property — read/write — `Double`

The value at one position, for an axis whose spacing is not even.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Size` `- 1`.

StringValues

Property — read/write — `String`

The label at one position, for an axis whose positions are named rather than measured.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Size` `- 1`.

Size

Property — read-only — `Integer`

How many positions the axis has. Set by `IArrayInfo.DimSizes`, not from here.

Presentation

Name

Property — read/write — `String`

The axis's name, as displayed — `Freq` on an FFT, for instance.

_Unit

Property — read/write — `String`

The axis's unit.

Precision

Property — read/write — `Integer`

How many decimal places to show axis values to.

CursorChannel

Property — read/write — `IChannel`

A channel whose value positions a cursor along this axis.

AxisViewInfo

Property — read-only — `IAxisViewInfo`

How the axis should be drawn.

> **Not usable from an automation client**
>
> The object this returns resolves none of its own member names. See `IAxisViewInfo`.

IAxisViewInfo

How one axis of an array channel should be drawn — its scaling, its direction and its default range.

Get it from `IAxisDef.AxisViewInfo`.

> **Not usable from an automation client**
>
> **None of this object's member names resolve** — every read and every write fails with an unknown-name error. `CursorChannel` is the one exception: it resolves and then fails with a server exception.
>
> There is no workaround from a client.

PYTHON

```python
avi = channel.ArrayInfo.AxisDef(0).AxisViewInfo
avi.ReverseAxis     # fails - name not found
avi.CursorChannel   # fails - the server throws an exception
```

C#

```csharp
dynamic avi = channel.ArrayInfo.AxisDef[0].AxisViewInfo;
var r = avi.ReverseAxis;     // fails - name not found
var c = avi.CursorChannel;   // fails - the server throws an exception
```

Members

AxisViewType

Property — read/write — `Integer`

How the axis is scaled:

Value	Scaling
`0`	linear
`1`	logarithmic
`2`	dB relative to 0 dB
`3`	dB relative to the noise floor
`4`	dB relative to a reference value

ReverseAxis

Property — read/write — `Boolean`

`True` to draw the axis from high to low.

DisplayDefaultMin

Property — read/write — `Double`

The lower end of the range a display should open at, in the axis's own units. Unset is negative infinity, meaning "no preference — scale to the data".

DisplayDefaultMax

Property — read/write — `Double`

The upper end of the same range. Unset is positive infinity.

CursorChannel

Property — read/write — `IChannel`

The channel that drives a cursor along this axis.

Unlike the other members this name does resolve, but the call then fails. See the note at the top of this page.

ICAN

The CAN interfaces present on the system.

Get it from `IApp.CAN`. Each port is an `ICANPort`.

PYTHON

```python
can = app.CAN
print(can.Count, "port(s), output:", can.SupportsOutput)
for i in range(can.Count):
    print(can.Item(i).InterfaceName)
```

C#

```csharp
dynamic can = app.CAN;
Console.WriteLine($"{can.Count} port(s), output: {can.SupportsOutput}");
for (int i = 0; i < (int)can.Count; i++)
    Console.WriteLine(can.Item[i].InterfaceName);
```

Members

Count

Property — read-only — `Integer`

Number of CAN ports.

Item

Property — read-only — `ICANPort`

The port at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

SupportsOutput

Property — read-only — `Boolean`

Whether the hardware can transmit as well as receive. `False` on a receive-only interface, which makes `ICANPort.SendFrame` pointless — check this before building a transmit path.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the ports can be iterated directly in languages that support it.

ICANContext

A custom CAN plug-in's handle on DewesoftX.

Its ports are `ICANPortContext`.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member	Description
`PortCount`	How many ports the plug-in provides.
`Ports`	The port at a position.
`GetClock`	The acquisition clock.
`GetClockOffset`	Its offset.

ICANMsg

One CAN frame being handed from a custom CAN plug-in to DewesoftX.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member	Description
`AddData`	Append data to the frame.

ICANPort

One CAN port — its baud rate, the frames it has received, and the frames you send on it.

Get it from `ICAN.Item`.

PYTHON

```python
port = app.CAN.Item(0)
print(port.InterfaceName, port.GetBaudRate(), port.SupportsCANFD)

port.Capture(True)
port.StartRead()
try:
    while True:
        ok, ts, arb_id, lo, hi = port.ReadMessage(0.0, 0, 0, 0)
        if not ok:
            break
        print(f"{ts:.6f} {arb_id:X} {lo:08X}{hi:08X}")
finally:
    port.EndRead()
```

C#

```csharp
dynamic port = app.CAN.Item[0];
Console.WriteLine($"{port.InterfaceName} {port.GetBaudRate()} {port.SupportsCANFD}");

port.Capture(true);
port.StartRead();
try
{
    double ts = 0; int arbId = 0, lo = 0, hi = 0;
    while (port.ReadMessage(ref ts, ref arbId, ref lo, ref hi))
        Console.WriteLine($"{ts:F6} {arbId:X} {lo:X8}{hi:X8}");
}
finally
{
    port.EndRead();
}
```

Reading frames

Reading is bracketed: `Capture` to start collecting, `StartRead` to open a read pass, `ReadMessage` until it returns `False`, then `EndRead`. The bracket matters — the driver uses it to hold the receive buffer still while you drain it.

Capture

Method

Starts or stops collecting frames into the port's buffer.

Parameter	Type	Direction	Description
`Status`	`Boolean`	in	`True` to collect, `False` to stop.

Nothing accumulates until this is on, so a read pass on an uncaptured port always comes back empty.

StartRead

Method

Opens a read pass.

ReadMessage

Method — returns `Boolean`

Takes the next frame from the buffer. Returns `False` when there are none left, which is how you end the loop.

Parameter	Type	Direction	Description
`TimeStamp`	`Double`	in/out	The frame's timestamp.
`ArbId`	`Integer`	in/out	The arbitration id.
`DataLo`	`Integer`	in/out	Data bytes 0–3, packed into one integer.
`DataHi`	`Integer`	in/out	Data bytes 4–7, packed into one integer.

All four are in/out, so Python returns them alongside the result — the call yields five values — while C# passes each by reference. Pass placeholders for them on the way in.

> **The data length is lost**
>
> The eight data bytes come back as two integers with no length alongside them, so a frame carrying three bytes is indistinguishable from one carrying eight — the unused bytes are whatever the driver left there. Nor is the extended-id flag reported.
>
> Use `ReadCANFDMessage` instead where you need either. It returns the data as a properly sized array and reports `Extended`, and it works for ordinary CAN frames as well as CAN FD.

EndRead

Method

Closes the read pass.

MessageCount

Property — read-only — `Integer`

Frames waiting in the buffer right now.

TotalMsgCount

Property — read-only — `Integer`

Frames received since capturing began.

TotalErrMsgCount

Property — read-only — `Integer`

Error frames received since capturing began. Worth watching alongside `TotalMsgCount` — a rising error count with a static message count usually means the baud rate is wrong.

ReadCANFDMessage

Method — returns `Boolean`

Takes the next frame, reporting its data as an array of the right length. Returns `False` when the buffer is empty.

Parameter	Type	Direction	Description
`Timestamp`	`Double`	in/out	The frame's timestamp.
`Extended`	`Boolean`	in/out	`True` for a 29-bit extended id.
`ArbId`	`Cardinal`	in/out	The arbitration id.
`Data`	`Variant`	in/out	The data bytes, as an array.

Sending frames

Check `ICAN.SupportsOutput` first, and enable output with `EnableOutput` — a receive-only interface cannot transmit however the call is made.

EnableOutput

Method

Turns transmission on or off.

Parameter	Type	Direction	Description
`Enable`	`Boolean`	in	`True` to allow sending.

SendFrame

Method — returns `Boolean`

Sends one ordinary CAN frame. `True` when it went out.

Parameter	Type	Direction	Description
`Extended`	`Boolean`	in	`True` for a 29-bit extended id.
`ArbId`	`Integer`	in	The arbitration id.
`Data`	`T_CANFrame`	in	The eight data bytes, as a structure of `Byte0` to `Byte7`.
`Size`	`Integer`	in	How many of those bytes to send, `0` to `8`.

The structure always has eight byte fields; `Size` is what decides how many are actually transmitted, so fill only what you need and set `Size` to match.

SendCANFDFrame

Method — returns `Boolean`

Sends a CAN FD frame, with the data as an array rather than a fixed structure — which also makes it the easier of the two for ordinary frames.

Parameter	Type	Direction	Description
`Extended`	`Boolean`	in	`True` for a 29-bit extended id.
`ArbId`	`Cardinal`	in	The arbitration id.
`Data`	`Variant`	in	The data bytes, as an array. Its length is the frame's length.

Configuration

InterfaceName

Property — read-only — `String`

The port's name. When more than one CAN interface is present the vendor is included, so the same port can report a different name depending on what else is installed — do not match on it.

GetBaudRate

Method — returns `Integer`

The current baud rate.

SetBaudRate

Method

Sets the baud rate.

Parameter	Type	Direction	Description
`BaudRate`	`Integer`	in	The rate, in bits per second.

GetBaudRateList

Method — returns `Variant`

The baud rates this hardware accepts, as an array. Pick from these rather than assuming a value is supported.

SupportsCANFD

Property — read-only — `Boolean`

Whether the hardware can do CAN FD.

IsCANFD

Property — read/write — `Boolean`

Whether the port is running in CAN FD mode. Only meaningful when `SupportsCANFD` is `True`.

SetCANFDBaudRate

Method

Sets both CAN FD baud rates and their sample points — the arbitration phase and the faster data phase.

Parameter	Type	Direction	Description
`BaudRate`	`Integer`	in	Arbitration baud rate, in bits per second.
`BaudRateSamplePoint`	`Double`	in	Its sample point, as a fraction of the bit time.
`DataBaudRate`	`Integer`	in	Data-phase baud rate.
`DataBaudRateSamplePoint`	`Double`	in	Its sample point.

UseListenOnly

Property — read/write — `Boolean`

`True` to put the port in listen-only mode, where it acknowledges nothing and cannot disturb the bus. The right setting for observing a live vehicle bus.

ICANPortContext

One port of a custom CAN plug-in, as DewesoftX sees it.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member	Description
`Used`	Whether the port takes part in the measurement.
`BaudRate`	The port's baud rate.
`ListenOnly`	Whether the port only listens.
`Termination`	Whether bus termination is on.
`Captured`	Whether frames are being collected.
`GetMsg`	Hand a received frame to DewesoftX, as an `ICANMsg`.
`SetTotalMsgCount`	Report the total frame count.
`SetErrMsgCount`	Report the error frame count.

ICNTGroup

The counter channels of the acquisition hardware.

Not reachable from an automation client

Channels of this kind carry the interface, but unlike `IDaqChannel` it is not registered for late binding — so none of its member names resolve from Python or C#. Only the `IChannel` members of such a channel are usable.

ICamera

One camera — its identity, its frame buffer, and the frames in it.

Get it from `IVideo.Cameras`. Each frame is an `IVideoFrame`.

Frames live in a ring buffer of `FrameBufSize` entries, with `FramePos` saying where the newest one is. To read the latest frame, read `FramePos` and index `FrameList` with it.

PYTHON

```python
cam = app.Video.Cameras(0)
print(cam.Name, cam.DisplayName, cam.FrameBufSize)
frame = cam.FrameList(cam.FramePos)
print(frame.GetTS(), frame.BufSize)
```

C#

```csharp
dynamic cam = app.Video.Cameras[0];
Console.WriteLine($"{cam.Name} {cam.DisplayName} {cam.FrameBufSize}");
dynamic frame = cam.FrameList[cam.FramePos];
Console.WriteLine($"{frame.GetTS()} {frame.BufSize}");
```

Identity

Name

Property — read-only — `String`

The camera's name.

DisplayName

Property — read-only — `String`

The name shown to the user, which is the one to put in a report or a log.

Used

Property — read-only — `Boolean`

Whether the camera takes part in the measurement.

Frames

FrameBufSize

Property — read-only — `Integer`

How many frames the ring buffer holds.

FramePos

Property — read-only — `Integer`

Where the newest frame sits in the buffer. It wraps, so it is a position and not a count — do not treat it as "how many frames so far".

FrameList

Property — read-only — `IVideoFrame`

The frame at a buffer position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Buffer position, `0` to `FrameBufSize - 1`.

The buffer keeps filling while you work through it, so a frame you fetched can be overwritten before you have finished with it. Copy the data out promptly rather than holding a frame across other calls.

FrameDataSize

Property — read-only — `Integer`

The size of one frame's image data.

FrameSizeInBytes

Property — read-only — `Integer`

The size of one frame in the buffer, in bytes.

GetBitmapInfoHeader

Method — returns `Variant`

The Windows bitmap header describing the frame format — dimensions, bit depth and compression. This is what tells you how to interpret the bytes `IVideoFrame.GetData` hands back.

TakeSnapshot

Method — returns `Variant`

Grabs the current image as a single value, without going through the buffer. The straightforward choice when you want one picture rather than a stream.

Control

Start

Method

Starts the camera.

Stop

Method

Stops it.

ICanvas

The drawing surface a visual control paints on.

Carries the usual primitives — lines, rectangles, ellipses, arcs, pies, polygons and polylines — along with text output and measurement, clipping, and block transfers between surfaces. Its `Brush`, `Font` and `Pen` hold the current style, and it creates the two kinds of offscreen bitmap: `IGDIBitmap` and `IDirectXBitmap`.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

ICanvasBrush

The fill style of an `ICanvas` — its colour and pattern.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

ICanvasFont

The text style of an `ICanvas` — typeface, size, colour, rotation and rendering quality.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

ICanvasPen

The line style of an `ICanvas` — colour, width, pattern and drawing mode.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IChannel

A single data channel — its identity, configuration, scaling, and the buffers its samples live in.

Get one from `IData.UsedChannels`, `IData.FindChannelByIndexEx`, or a channel group.

PYTHON

```python
ch = app.Data.FindChannel("AI 0")
print(ch.Name, ch.Unit_, ch.DBDataSize)
```

C#

```csharp
dynamic ch = app.Data.FindChannel("AI 0");
Console.WriteLine($"{ch.Name} {ch.Unit_} {ch.DBDataSize}");
```

Identity and naming

Name

Property — read/write — `String`

The channel name, as shown and editable in **Config**. **Names are not unique** — several channels may share one.

ChNo

Property — read-only — `String`

The channel's built-in identifier, assigned by DewesoftX and not user-editable — `AI 0`, `Math 0 (Formula)` and so on.

DisplayName

Property — read-only — `String`

The name to label the channel with, resolved the way DewesoftX itself does it: `LongName` when `ShowLongName` is set, otherwise plain `Name`. Prefer this over `Name` for anything user-facing.

LongName

Property — read-only — `String`

The fully qualified name — the channel's description path with `/` and the name appended, for example `AI 1/Temperature`.

ShowLongName

Property — read-only — `Boolean`

Whether the long form should be used. Read-only here: it is set internally, for instance by an export that has "add description to channel name" enabled.

Description

Property — read/write — `String`

A free-text comment on the channel. The same underlying value as `Measurement`.

Measurement

Property — read/write — `String`

The same value as `Description`. Two names for one property.

Unit_

Property — read/write — `String`

The unit of measurement — `V`, `m/s2`, or `-` for none.

The trailing underscore is there because `Unit` is a reserved word in the language the interface was defined in.

Writing this marks the unit as user-overridden, so it will no longer be re-derived from the sensor or amplifier. Writing `UnitDescription` does not have that effect.

UnitDescription

Property — read/write — `String`

A short qualifier appended after the unit when the full unit string is composed — an acoustic weighting suffix such as `(A)`, for example. The composed form is `Unit_` followed by a space and this.

The qualifier is dropped for phase presentations, where it would be wrong.

IndexEx

Property — read-only — `Variant`

The channel index, as an array of `Integer` — the value `IData.FindChannelByIndexEx` takes. Unlike a name, an index is unique.

GetIndexString

Method — returns `String`

The channel index rendered as text: elements joined with `;`, prefixed with `unit:` when the channel comes from a remote measurement unit — `300;5`, or `2:300;5`.

Returns an empty string for a channel whose index level is `0`. That is not an error.

Group

Property — read-only — `IChannelGroup`

The channel group this channel belongs to.

LogicalName

Property — read/write — `String`

Free for a client or plug-in to use. Stored in the setup; DewesoftX itself never reads it.

LogicalIndex

Property — read/write — `Int64`

Free for a client or plug-in to use. Stored in the setup; DewesoftX itself never reads it.

Tag

Property — read/write — `Integer`

An arbitrary integer for a client or plug-in to use.

Settings

Property — read-only — `String`

A channel-type-specific summary string shown in the channel list — an analog channel's measurement range, a math channel's formula.

SelectorIndex

Property — read/write — `Variant`

Overrides the index used to place the channel in the channel-tree view, without changing its real index.

SelectorIndexLevel

Property — read/write — `Integer`

Index level for `SelectorIndex`.

SelectorIndexStartLevel

Property — read/write — `Integer`

Start level for `SelectorIndex`.

Enabling and visibility

These four flags are independent, and they gate each other in one direction: only a used channel can be stored.

Property	Effect
`Used`	the channel exists during a measurement at all
`Stored`	its data is written to the data file
`Shown`	it is offered to the user in channel lists
`Exported`	it is included in exports

Used

Property — read/write — `Boolean`

Whether the channel takes part in the measurement. Only used channels appear during measurement or can feed a math channel.

After changing this, call `IData.BuildChannelList` before reading any channel list, or the list still reflects the old state.

Stored

Property — read/write — `Boolean`

Whether the channel's data is written to the data file. Setting it `False` on a fast raw channel while keeping a slower calculated channel stored is the usual way to keep file sizes down without losing the derived result.

An unused channel cannot be stored.

Shown

Property — read/write — `Boolean`

Whether the channel is offered to the user — a way to hide channels that exist only to feed something else.

Exported

Property — read/write — `Boolean`

Whether the channel is included when data is exported.

ExportOrder

Property — read/write — `Integer`

Position of the channel in the export order. The default `-1` means unordered, and all unordered channels come after every channel that has an explicit position.

What kind of channel this is

Async

Property — read-only — `Boolean`

`True` for an asynchronous channel — one whose samples carry their own timestamps rather than arriving on the master clock. Set it with `SetAsync`.

SetAsync

Method

Parameter	Type	Direction	Description
`Async`	`Boolean`	in	`True` makes the channel asynchronous.

IsSingleValue

Property — read-only — `Boolean`

`True` for a channel that holds one value for the whole measurement rather than a stream of samples.

SetIsSingleValue

Method

Makes the channel a single-value channel.

Parameter	Type	Direction	Description
`Value`	`Boolean`	in	`True` for single value.

A single-value channel must be synchronous — do not also call `SetAsync(True)`.

CalcSingleValue

Property — read-only — `Boolean`

`True` when a single-value channel is recomputed on demand from its inputs rather than read from a stored sample. Such a channel works at any cursor position in analysis without having been recorded.

Only meaningful together with `IsSingleValue`. Because reading the value can trigger a calculation, reads are neither free nor guaranteed side-effect-free.

ArrayChannel

Property — read/write — `Boolean`

`True` when each time instant carries a whole array of values rather than one — an FFT spectrum, for instance.

ArrayInfo

Property — read-only — `IArrayInfo`

Dimensions and axis definitions of an array channel. Only meaningful when `ArrayChannel` is `True`.

ArraySize

Property — read-only — `Integer`

Total number of array items per sample — the product of all dimension sizes.

IsControlChannel

Property — read/write — `Boolean`

`True` for a channel a user can write to during a measurement through a visual control.

ControlChannelFlags

Property — read/write — `Integer`

Flags describing a control channel. Bits `0` – `2` form a masked numeric-type field and bit `3` is a separate flag, so read it with a mask rather than comparing to whole values.

ControlChannelState

Property — read/write — `Integer`

`0` the control channel is enabled, `1` it is disabled and the user cannot change it.

SetAsStringChannel

Method

Makes the channel carry text. Internally it becomes an array of bytes.

Parameter	Type	Direction	Description
`Size`	`Integer`	in	Maximum length in characters. Longer strings are truncated.

Choose the smallest size that fits — it is allocated per sample, so it drives the data file size.

Write to it with `AddAsyncString`.

SetAsCANFDMessage

Method

Configures the channel to carry raw CAN FD frames: data type `14`, asynchronous, and with the intermediate buffer switched off since min/max statistics are meaningless for frames.

> **No interface method writes a CAN FD frame**
>
> `AddData` and `AddAsyncData` reject this data type. A client can create such a channel but cannot fill it through any `Add…` method — the only mechanical route is writing frames through `GetDBAddress64` and `GetTSAddress64` and then calling `IncDBSamples`.
>
> Because the intermediate buffer is off, displays and mathematics that rely on reduced data cannot consume the channel. Call this before the buffer is allocated.

ChannelType

Property — read-only — `Integer`

Which subsystem produced the channel. The values are sparse group bases, not a dense enumeration:

Value	Kind	Value	Kind
`0`	general or unclassified	`6000`	power
`1`	analog input	`6001`	power output
`100`	digital input channel	`100000`	plug-in
`101`	digital input port	`102000`	remote
`201`	additional counter	`200000`	analog output
`300`	DAQ output	`600000`	variable
`1000`	PAD or module	`700000`	video
`2000`	CAN channel	`900000`	visual control
`2001`	CAN message	`1000000`	system monitor
`2002`	CAN port	`1100000`	control
`3000`	mathematics	`1200000`	event log
`3001`	filter	`1300000`	real-time control
`3002`	reference curve	`1400000`	real time
		`1500000`	openDAQ

Treat an unrecognised value as "some other kind of channel" rather than an error. `3000` does not distinguish the two mathematics implementations, and GPS channels report `0`.

MType

Property — read/write — `Integer`

How a value should be formatted as text, and parsed back from it:

Value	Format	Value	Format
`0`	plain numeric	`6`	relative time
`1`	GPS X absolute	`7`	IP address
`2`	GPS Y absolute	`8`	binary data
`3`	GPS direction	`9`	MAC address
`4`	absolute time	`10`	discrete
`5`	character or string	`11`	LIN data

This is presentation only — it is unrelated to `DataType`, which is how a sample is stored, and to `ChannelType`, which is what produced the channel.

Data type

DataType

Property — read-only — `Integer`

How one sample is stored. This determines which `Add…Sample` method is valid and what `Bytes` reports.

Value	Type	Bytes	Value	Type	Bytes
`0`	Byte	1	`10`	ComplexDouble	16
`1`	ShortInt	1	`11`	Text	4
`2`	SmallInt	2	`12`	Binary	16
`3`	Word	2	`13`	CAN message	12
`4`	Integer	4	`14`	CAN FD message	72
`5`	Single	4	`15`	Bytes8	8
`6`	Int64	8	`16`	Bytes16	16
`7`	Double	8	`17`	Bytes32	32
`8`	Longword	4	`18`	Bytes64	64
`9`	ComplexSingle	8			

SetDataType

Method

Parameter	Type	Direction	Description
`ADataType`	`Integer`	in	One of the values above.

Bytes

Property — read-only — `Integer`

Bytes used per sample, following from `DataType`.

BitCount

Property — read/write — `Integer`

Significant bits per sample, which can be fewer than `Bytes` allows — a 24-bit converter stores 24 significant bits in 4 bytes.

IsDoublePrecision

Property — read-only — `Boolean`

Whether the channel is processed in double precision throughout the calculation and reduction path. Derived rather than stored: true when the significant bit count exceeds 20.

This is about processing precision, not storage — a 24-bit integer channel reports `True`. It also decides which of two parallel intermediate-buffer arrays holds this channel's reduced values, which matters if you use `FastCalcInt16`.

Scaling

A raw buffer value becomes a physical value through `physical = scale * raw + offset`. Two different pairs are exposed, and they are **not** interchangeable.

	What it is
`Scale` / `Offset`	the user-entered pair stored on the channel
`CalcDataScale` / `CalcDataOffset`	the **effective** pair actually applied

For an analog input the effective pair also folds in the amplifier range and gain, so the two differ by orders of magnitude. For a plain math or plug-in channel they coincide.

Scale

Property — read/write — `Double`

The channel's scale factor.

Offset

Property — read/write — `Double`

The channel's offset.

CalcDataScale

Method — returns `Double`

The effective scale factor actually applied to raw values. Use this — with `CalcDataOffset` — when you read raw samples through `GetUnscaledData…` or `GetDBAddress64` and scale them yourself.

CalcDataOffset

Method — returns `Double`

The effective offset actually applied to raw values.

Both describe a *linear* transform, so they are wrong for a channel where `GetHasNonLinearScale` is `True`.

ScaleValue

Method — returns `Single`

Applies the channel's scaling to one value.

Parameter	Type	Direction	Description
`Value`	`Single`	in	Unscaled value.

ScaleValueDouble

Method — returns `Double`

As `ScaleValue` in double precision.

Parameter	Type	Direction	Description
`Value`	`Double`	in	Unscaled value.

IsScaleEnabled

Property — read-only — `Boolean`

Whether the channel's scaling may be edited at all. `False` for channels where scaling is meaningless or locked — GPS and video channels, and sensor channels whose scaling is disabled while measuring.

GetHasNonLinearScale

Method — returns `Boolean`

`True` when the channel applies a non-linear conversion — a thermocouple or RTD linearisation, or an interpolated table — rather than scale and offset. When this is `True`, none of the four scale/offset values fully describes the conversion; use `ScaleValueDouble` instead.

Display

MainDisplayColor

Property — read/write — `Integer`

The channel's colour, as a BGR-ordered integer — red is `0x0000FF`.

TypicalMinValue

Property — read-only — `Single`

The **minimum** of the y-axis a graph should use for this channel: `UserScaleMin` when it has been set, otherwise `-5`.

TypicalMaxValue

Property — read-only — `Single`

The maximum of the y-axis a graph should use: `UserScaleMax` when it has been set, otherwise `5`.

UserScaleMin

Property — read/write — `Single`

Overrides the default lower bound of the display scale.

UserScaleMax

Property — read/write — `Single`

Overrides the default upper bound of the display scale.

ChangeThreshold

Property — read/write — `Single`

For tabular displays set to print only on value change: the value must move by more than this before a new row is written.

GetDigitsAfterDecPoint

Method — returns `Integer`

How many decimal places to show. Resolved from an explicit per-channel override, then a per-unit setting, then an estimate from the channel's resolution.

A display hint only — it does not affect stored or returned values, and it can change when the range or sensor changes.

DiscreteList

Property — read-only — `IDiscreteList`

Named values for a channel whose numbers stand for discrete states.

ViewInfo

Property — read-only — `IViewInfo`

Default display preferences — axis mode, which visual control to create, default 2-D graph style — so a derived channel lands on screen already configured.

Advisory only: a user's later change to a display overrides it.

Not readable from an automation client

The object this returns resolves no member names, so there is nothing you can do with it from Python or C#. See `IViewInfo`.

Alarm and warning levels

Two-stage thresholds in the channel's **scaled physical units**. The measurement engine compares each block against them, with hysteresis and a hold time, and can log crossings into the data file.

Property	Threshold
`WarningLevelLow`	lower warning
`WarningLevelHigh`	upper warning
`CriticalLevelLow`	lower critical
`CriticalLevelHigh`	upper critical

All four are read/write `Double`.

`NaN` means no limit

A channel with no alarms configured returns `NaN` for all four. Writing a non-`NaN` value creates the alarm structure — there is no separate enable call — and writing `NaN` to the last remaining limit destroys it again.

These four address only the first limit set. A channel can carry several sets selected at runtime by a condition channel, and the others are not reachable here.

Writing synchronous samples

A synchronous channel takes one value per master-clock sample and no timestamp. Pick the method matching the channel's `DataType`.

Method	Value type
`AddByteSample`	`Byte`
`AddShortintSample`	`ShortInt`
`AddSmallintSample`	`SmallInt`
`AddWordSample`	`Word`
`AddIntegerSample`	`Integer`
`AddLongwordSample`	`Cardinal`
`AddIn64Sample`	`Int64`
`AddSingleSample`	`Single`
`AddDoubleSample`	`Double`
`AddData`	`Variant`, any type

Each takes a single `Value` parameter of the type shown.

`AddIn64Sample` is spelled without the `t` — that is the actual member name, unlike the asynchronous `AddAsyncInt64Sample`.

Add exactly the expected number of samples

In each `OnGetData` cycle a synchronous channel must receive exactly the number of samples reported by `IData.GetSamplesAcquired`. If you cannot supply them all yet, set `CalcDelay` to the number still missing.

Account for `SRDiv` as well: with a divider of 2, only every second sample is expected.

AddSingleSamples

Method

Adds a whole block of `Single` samples in one call — far cheaper than one call per sample.

Parameter	Type	Direction	Description
`Count`	`Integer`	in	Number of elements in both arrays.
`Data`	`Variant`	in	Array of `Single`, exactly `Count` elements.
`Timestamps`	`Variant`	in	Array of `Double`, or empty for a synchronous channel.

Writing asynchronous samples

An asynchronous channel takes a value and its timestamp, in seconds of relative measurement time.

Method	Value type
`AddAsyncByteSample`	`Byte`
`AddAsyncShortintSample`	`ShortInt`
`AddAsyncSmallintSample`	`SmallInt`
`AddAsyncWordSample`	`Word`
`AddAsyncIntegerSample`	`Integer`
`AddAsyncLongwordSample`	`Cardinal`
`AddAsyncInt64Sample`	`Int64`
`AddAsyncSingleSample`	`Single`
`AddAsyncDoubleSample`	`Double`
`AddAsyncData`	`Variant`, any type

Each takes `Value` and `TimeStamp: Double`.

AddAsyncString

Method

Adds one text sample. Only valid after `SetAsStringChannel`.

Parameter	Type	Direction	Description
`Value`	`String`	in	Text. Truncated to the configured size.
`TimeStamp`	`Double`	in	Timestamp in seconds.

AddAsyncComplexSingleSampleValue

Method

Adds one complex sample to a `ComplexSingle` channel.

Parameter	Type	Direction	Description
`RealValue`	`Single`	in	Real part.
`ImagValue`	`Single`	in	Imaginary part.
`Timestamp`	`Double`	in	Timestamp in seconds.

AddAsyncComplexDoubleSampleValue

Method

As above for a `ComplexDouble` channel, with `Double` parts.

AddAsyncBinarySample

Method

Adds one variable-length blob to a binary channel — see Binary channels.

Parameter	Type	Direction	Description
`Timestamp`	`Double`	in	Timestamp in seconds.
`Data`	`Variant`	in	One-dimensional array of `Byte`.

Pass a **byte** array specifically. The type check is a bitmask test rather than an equality test, so an array of another numeric type is accepted and then misread — its element count is taken as a byte count.

The block-add family

Nine methods — `AddAsyncByteSamples`, `AddAsyncShortIntSamples`, `AddAsyncSmallIntSamples`, `AddAsyncWordSamples`, `AddAsyncIntegerSamples`, `AddAsyncLongWordSamples`, `AddAsyncInt64Samples`, `AddAsyncSingleSamples` and `AddAsyncDoubleSamples` — add a block of samples with their timestamps in one call.

> **Not callable from an automation client**
>
> Unlike `AddSingleSamples`, which takes `Variant` arrays, these take **raw memory pointers** to the value and timestamp arrays, which a client cannot supply.
>
> For a block write from an automation client, use `AddSingleSamples`.

Reading the direct buffer

The direct buffer is a ring of raw acquired samples, with a parallel ring of timestamps for asynchronous channels. Read whole blocks where you can: one call for a thousand samples is far faster than a thousand calls.

Three positions describe the ring, and you need all three to read it correctly:

Property	Meaning
`DBBufSize`	total capacity
`DBDataSize`	how many samples are valid
`DBPos`	where the next sample will be written

Once `DBDataSize` reaches `DBBufSize` the ring has wrapped and the oldest samples are being overwritten.

DBBufSize

Property — read-only — `Integer`

Capacity of the direct buffer, in samples. Fixed for the duration of a measurement.

DBDataSize

Property — read-only — `Integer`

Number of valid samples in the direct buffer. It grows to `DBBufSize` and then stays there.

DBPos

Property — read-only — `Integer`

Index the next sample will be written to. The newest sample is therefore at `DBPos - 1`, wrapping to `DBBufSize - 1` when `DBPos` is `0`.

DBValues

Property — read-only — `Single`

One scaled sample at a raw ring index.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Ring index, `0` to `DBBufSize - 1`.

DBValuesDouble

Property — read-only — `Double`

As `DBValues` in double precision.

DBValuesEx

Method — returns `Double`

One scaled sample, choosing how a complex value is collapsed to a real number.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Ring index.
`ComplexPresent`	`ComplexPresentation`	in	See below.

Value	Presentation	Value	Presentation
`0`	channel default	`4`	real part
`1`	magnitude	`5`	imaginary part
`2`	phase in degrees, ±180	`6`	phase in degrees, 0–360
`3`	phase in radians		

Complex values here are not scaled

For a complex channel this method returns the converted value **without** applying `Scale` and `Offset`, while `DBValues` does apply them. Do not mix the two and expect the same units.

For a complex channel, `0` yields magnitude rather than the channel's configured default.

DBTimeStamp

Property — read-only — `Double`

Timestamp of one sample, for an asynchronous channel.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Ring index.

The index is not checked. On a synchronous channel, or an asynchronous one whose `DBDataSize` is `0`, this reads memory that holds no timestamp and returns an uninitialised value or faults. Check `DBDataSize` and `Async` before calling. The same applies to `DBValues`, which quietly returns `0` past the end of the data.

GetScaledData

Method — returns `Variant`

All available scaled samples from the direct buffer, as an array of `Single`. Values outside the `Single` range are clamped; for a complex channel only the real part is returned.

GetScaledDataEx

Method — returns `Variant`

Scaled samples from a chosen span.

Parameter	Type	Direction	Description
`Start`	`Integer`	in	Start position — see the note below.
`Count`	`Integer`	in	Number of samples.

`Start` changes meaning once the ring wraps

While `DBDataSize` is below `DBBufSize`, `Start` is an absolute index from the beginning of the buffer. Once the ring has wrapped, reading begins at `(DBPos + Start) mod DBBufSize` — so `Start` becomes an offset relative to the write position, and you pass a negative value to look back.

Nothing is range-checked. An oversized `Count` reads past valid data and returns whatever was in memory, indistinguishable from real samples — clamp it to `DBDataSize`.

On an array channel `Count` is still a number of samples, and the flat result is `Count` × `ArraySize` elements long, in sample order. One array is `GetScaledDataEx(start, 1)`.

GetScaledDataEx1

Method

As `GetScaledDataEx` but returning through an out parameter instead of a return value.

GetScaledDataDoubleEx

Method — returns `Variant`

As `GetScaledDataEx` in double precision. The array holds `Count * ArraySize` values, or twice that for a complex channel where each sample contributes real part then imaginary part.

GetUnscaledData

Method — returns `Variant`

All available raw samples, in the channel's own `DataType`. Scale them with `CalcDataScale` and `CalcDataOffset`.

GetUnscaledDataEx

Method — returns `Variant`

Raw samples from a chosen span. `Start` behaves as for `GetScaledDataEx`.

GetUnscaledDataEx1

Method

As above, returning through an out parameter.

GetTSData

Method — returns `Variant`

Timestamps for an asynchronous channel, as an array of `Double`, the same length as `GetScaledData` returns.

GetTSDataEx

Method — returns `Variant`

Timestamps from a chosen span, to pair with `GetScaledDataEx`.

GetTSDataEx1

Method

As above, returning through an out parameter.

CreateConnection

Method — returns `IChannelConnection`

Creates a connection for reading this channel's data with block and overlap handling managed for you — usually easier than driving the ring positions directly.

Reading a value at a position

These hide the difference between synchronous and asynchronous channels: you ask for a position or a time and get a value.

Method	Returns	Position given as
`GetValueAtAbsPos`	`Single`	sample offset
`GetValueAtAbsPosDouble`	`Double`	sample offset
`GetValueAtAbsPosEx`	`Single`	sample offset, with array item and complex presentation
`GetValueAtAbsPosDoubleEx`	`Double`	sample offset, with array item and complex presentation
`GetValueAtAbsPosInt64`	`Int64`	sample offset, raw
`GetValueAtRelTimeDouble`	`Double`	seconds back from now

The shared parameters:

Parameter	Type	Direction	Description
`Pos`	`Integer`	in	Offset in master-clock samples **back** from the current position. The newest sample is `-1`; `0` and positive values lie in the future and return `0`.
`SeekPos`	`Integer`	in/out	Search cursor for asynchronous channels — see below.
`Interpolate`	`Boolean`	in	`True` interpolates between the bracketing samples; `False` returns the last sample at or before the position.
`Ind`	`Integer`	in	Array item within the sample, `0` for a scalar channel.
`ComplexPresent`	`ComplexPresentation`	in	How to collapse a complex value.

`SeekPos` is worth understanding: pass a variable and hand it back unchanged on the next call, and the search resumes from the previous hit instead of scanning from scratch — which turns a sequential sweep from quadratic to linear. **Pass a negative value to opt out** and let the channel use its own cursor. It is ignored for synchronous and single-value channels.

`GetValueAtRelTimeDouble` takes a number of **seconds back from the present** instead, converting it with the channel's own rate: `0.0` is now and `1.25` is a second and a quarter ago. It is not an absolute time from the start of the measurement.

These return unscaled values

Every method in this family reads the sample buffer directly, so you get the raw number as acquired — a channel reading `1.16 V` returns something like `3804`. That is unlike `SingleValue`, `DBValues` and `GetScaledData`, which all give the value in the channel's unit.

`Scale` and `Offset` will not generally convert one to the other — on a channel scaled by its amplifier range they are `1.0` and `0.0`. Where you want a physical value, read one of the scaled paths instead.

No value means zero, not an error

None of these report that there is no data at the requested position. They return `0` or the last known value, and the internal flag that distinguishes the two cases is not exposed.

`GetValueAtAbsPosInt64` differs further: it takes no array item or complex presentation, truncates interpolated results rather than rounding, and **raises** for `Single`, `Double` and complex channels.

Intermediate buffers

Above the direct buffer sit up to six levels of *reduced* blocks, each holding minimum, maximum, average and RMS over a block of the level below. They hold no individual samples, and they are always synchronous — there are no timestamps to read.

FirstIBLevel

Property — read-only — `Integer`

The first level that exists for this channel, which may be above `0` — a slow asynchronous channel starts higher.

IBBufSize

Property — read-only — `Integer`

Capacity of the intermediate buffer, the same for every level.

IBDataSize

Property — read-only — `Integer`

Valid entries at the first level. The same as `IBDataSizeEx[FirstIBLevel]`.

IBDataSizeEx

Property — read-only — `Integer`

Valid entries at a given level.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level.

IBPos

Property — read-only — `Integer`

Next write position at the first level.

IBPosEx

Property — read-only — `Integer`

Next write position at a given level.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level.

IBValues

Property — read-only — `T_ReducedRec`

The reduced record at the first level — minimum, maximum, average and RMS.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position in the buffer.

IBValuesEx

Property — read-only — `T_ReducedRec`

The reduced record at a given level. `IBValuesEx[FirstIBLevel, Index]` is the same as `IBValues[Index]`.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level.
`Index`	`Integer`	in	Position in the buffer.

GetIBValues

Method

The same four values as separate out parameters, for callers that cannot receive a record.

Parameter	Type	Direction	Description
`Pos`	`Integer`	in	Position in the buffer.
`Min`	`Single`	out	Minimum.
`Max`	`Single`	out	Maximum.
`Ave`	`Single`	out	Average.
`Rms`	`Single`	out	RMS.

Direct memory access

These return the base address of a buffer inside DewesoftX, to be interpreted with `DBPos`, `DBDataSize`, `DBBufSize` and `Bytes`.

Not usable from an automation client

The call succeeds and returns a number, but the address belongs to DewesoftX. A client cannot dereference it. Read samples through `GetScaledDataEx` or `DBValues` instead.

GetDBAddress64

Method — returns `Int64`

Base address of the direct sample buffer.

GetTSAddress64

Method — returns `Int64`

Base address of the timestamp array. Allocated only for asynchronous channels.

GetBinAddress64

Method — returns `Int64`

Base address of the binary payload ring. Allocated only for binary channels.

IncDBSamples

Method

Advances `DBPos`, and possibly `DBDataSize`, after you have written samples directly into buffer memory.

Parameter	Type	Direction	Description
`Count`	`Integer`	in	Number of samples written.

Addresses go stale, and there is no safety net

`GetDBAddress`, `GetTSAddress` and `GetBinAddress` — the versions without `64` — are obsolete and return `0` in a 64-bit build. Always use the `…64` variants.

Entering or leaving freeze mode swaps the buffers for freeze copies, and growing the binary ring reallocates it, so re-fetch the address after any such transition and never cache it across acquisition state changes.

Nothing checks that a buffer exists. Calling these on a channel that is not `Used`, or in the wrong mode, dereferences a null pointer.

Binary channels

A binary channel has `DataType` `12` and carries a variable-length blob per sample rather than a number — raw receiver output, for instance. The direct buffer holds only position-and-size descriptors; the payload bytes live in a separate byte ring.

BinAvgSampleSize

Property — read/write — `Integer`

Expected average payload size in bytes, used to size the byte ring as `DBBufSize * BinAvgSampleSize`. Default `100`.

Takes effect only when the buffer is next allocated — entering or leaving setup, or starting a measurement — and only for asynchronous channels. Setting it on a live channel changes nothing.

BinBufSize

Property — read-only — `Integer`

Total size of the byte ring.

BinAbsPos

Property — read-only — `Int64`

Running write cursor into the byte ring, as a monotonically increasing **byte** count. Its position in the ring is `BinAbsPos mod BinBufSize`. This is not a sample index.

GetBinaryDataAtPos

Method

Reads back the blob at a direct-buffer index, reassembled across the ring's wrap point.

Parameter	Type	Direction	Description
`Pos`	`Integer`	in	Direct-buffer index.
`Data`	`Variant`	out	Array of `Byte`, or empty for a zero-length sample.

If the payload added within one intermediate-buffer block reaches `BinBufSize`, further samples are dropped and an overrun is logged. Raise `BinAvgSampleSize` before allocation if blobs are larger than expected.

Data available in analysis

In analysis mode only part of a stored file is held in memory, and only that part is reachable.

DStartDataAvail

Property — read-only — `Integer`

Start of the loaded window, in blocks. The absolute sample number is `DStartDataAvail * IData.Samples`.

DStopDataAvail

Property — read-only — `Integer`

End of the loaded window, in blocks. The absolute sample number is `DStopDataAvail * IData.Samples + DStopDataAvailDir`.

DStopDataAvailDir

Property — read-only — `Integer`

Remaining samples past the last complete block of the loaded window.

LoadFullBuffer

Property — read/write — `Boolean`

Requests that the channel's entire stored history be loaded at once rather than a window around the cursor — for a channel that must be examined as a whole.

Only affects asynchronous channels that are `Used` and `Stored`, and only in analysis mode. On a long recording it allocates a buffer proportional to the whole file.

KeepValues

Property — read/write — `Boolean`

Makes the channel's buffer and contents survive events that would normally discard them, and embeds the buffer contents in the setup file so sample values travel with the setup.

> **Clearing becomes a no-op**
>
> With this set, the channel is never cleared between measurements — its sample positions carry over, and managing that state becomes the caller's problem.
>
> The setup also grows by roughly `DBDataSize * Bytes * ArraySize` bytes for each such channel.

Sample rate and timing

SRDiv

Property — read-only — `Integer`

The channel's divider from the master sample rate, so its nominal rate is `IData.SampleRateEx / SRDiv`.

SetSRDiv

Method

Parameter	Type	Direction	Description
`SRDiv`	`Integer`	in	Divider.

SRDivType

Property — read-only — `TSRDivType`

How skipped samples are handled: `0` skip, `1` average, `2` 4^{th}-order filter, `3` 6^{th}-order filter, `4` 8^{th}-order filter. Only relevant when `SRDiv` is above `1`.

SetSRDivType

Method

Parameter	Type	Direction	Description
`AType`	`TSRDivType`	in	One of the values above.

CalcSRDiv

Property — read-only — `Integer`

The divider actually in use, which can differ from `SRDiv` — it is `1` in **Config**.

GetSampleRate

Method — returns `Double`

The channel's **measured** rate in Hz: for a synchronous channel the real acquisition rate divided by the divider, for an asynchronous channel the rate observed from the data.

Returns `0` — not the nominal rate — for a single-value channel, an offline channel, or an asynchronous channel in analysis whose sample count is unknown. For a configured rather than observed rate, compute `IData.SampleRateEx / SRDiv`.

ExpectedAsyncRate

Property — read/write — `Single`

Estimated rate in Hz for an asynchronous channel. **The buffer is sized from this** — set it too low and samples are lost.

ExpectedAsyncRateType

Property — read/write — `Integer`

The timing character of an asynchronous channel, which guides buffer sizing and resampling: `0` semi-fixed (nearly constant, the default), `1` variable (genuinely irregular, such as CAN or GPS), `2` equidistant and synchronised to the DewesoftX clock, `3` equidistant on its own independent clock.

ResamplerType

Property — read/write — `Integer`

Algorithm used when the channel must be resampled onto a common time base, mainly for export: `-2` automatic, `-1` disabled, `0` hold, `1` linear, `2` alias-free.

Not every algorithm suits every channel. An asynchronous channel accepts only hold or linear; an asynchronous array channel, a single-value channel and a discrete channel accept only hold. A channel with `CustExportRate` of `0` cannot be resampled at all.

CustExportRate

Property — read/write — `Double`

Target rate in Hz for export, for exporters that support one. `0` means export at the native rate.

The stored value is single precision, so very precise values are rounded.

AsyncValidUpTo

Property — read/write — `Double`

Lets the source of an asynchronous channel declare that it has delivered everything up to a point in time, in seconds of relative measurement time, even though no sample carries that timestamp. Without it, a channel that legitimately produces nothing for a while looks like one that is lagging, and the calculation delay grows to accommodate it.

Asynchronous channels only. It must be at least the timestamp of the last delivered sample, and it resets to `0` whenever the channel is cleared — so republish it each measurement.

CalcDelay

Property — read/write — `Integer`

How many samples this channel is behind.

A synchronous channel must add exactly the expected number of samples each `OnGetData` cycle; if it cannot yet, set this to the number still missing. For an asynchronous channel DewesoftX derives the delay from the timestamps, so it need not be set.

GetOfflineStatus

Method — returns `Integer`

`0` online, `1` offline, `2` offline calculated but not stored, `3` calculated and stored, `11` currently calculating in analysis.

Statistics

AbsMin

Property — read-only — `Double`

Lowest value seen on this channel during the current measurement.

AbsMax

Property — read-only — `Double`

Highest value seen on this channel during the current measurement.

StartFastCalc

Method

Resets the block accumulators, at the start of a block, before supplying statistics with `FastCalcInt16`.

FastCalcInt16

Method

Supplies block statistics directly instead of letting DewesoftX derive them, for a source that already has them. It writes one reduced record and finalises the block.

Parameter	Type	Direction	Description
`Min`	`SmallInt`	in	Raw, unscaled minimum.
`Max`	`SmallInt`	in	Raw, unscaled maximum.
`Ave`	`Single`	in	**Sum** of the block's values, not their average.
`Rms`	`Single`	in	**Sum of squares** of the block's values, not the RMS.

`Ave` and `Rms` are sums, and the channel must be 16-bit

DewesoftX divides `Ave` by the block's sample count and takes the square root of `Rms` itself. Passing an already-averaged figure gives a result too small by a factor of the sample count.

It writes the 16-bit and single-precision accumulators, so it is only correct for a two-byte channel where `IsDoublePrecision` is `False`. Check that first.

Call `StartFastCalc` once per block and this exactly once at the end of each block — it advances the buffer position, so an extra call inserts a phantom block.

Single value and text

SingleValue

Property — read/write — `Double`

The value of a numeric single-value channel.

Text

Property — read/write — `String`

The value of a text single-value channel — one with `DataType` `11`.

Setup and metadata

GetChannelSetup

Method

The channel's setup as XML, returned as an array of bytes.

Parameter	Type	Direction	Description
`Data`	`Variant`	out	Array of `Byte` holding XML.

PYTHON

```python
raw = ch.GetChannelSetup()
xml = bytes(raw).decode("utf-8")
```

C#

```csharp
ch.GetChannelSetup(out object raw);
string xml = Encoding.UTF8.GetString((byte[])raw);
```

SetChannelSetup

Method

Applies a channel setup.

Parameter	Type	Direction	Description
`Data`	`Variant`	in	Array of `Byte` holding XML.

UpdateXML

Method

Reads or writes the channel's properties from or to a node of an XML setup document — the hook a plug-in uses to persist its own channel settings.

Parameter	Type	Direction	Description
`DOMDocument`	`Variant`	in	The XML setup document.
`DOMNode`	`Variant`	in	The node to read from or write to.
`Write`	`Boolean`	in	`True` writes, `False` reads.

ShowChannelSetup

Method

Opens the setup dialog for this channel.

Properties

Property — read-only — `IProperties`

A freely named metadata collection on the channel — enumerable, with scalar entries and whole XML documents. Drivers use it to publish facts about a channel. Entries persist in the setup.

Entries added as public are shown to the user; others are not. Values are untyped, with no schema — the reader must know what was stored.

SetCustomProp

Method

Stores a named value on the channel, for structured measurement metadata such as modal and transfer-function descriptors.

Parameter	Type	Direction	Description
`Name`	`String`	in	A **registered** property name.
`Value`	`Variant`	in	The value, of the type that name declares.

Unregistered names are silently lost

On save, each entry is looked up in a fixed catalogue of known property names. An entry whose name is not in that catalogue is **skipped** with a setup warning — so a name you invent survives the session and then disappears.

Registered names are also typed, so a mismatched value is coerced or dropped. For arbitrary user-defined names use `Properties` instead.

GetCustomProp

Method — returns `Variant`

Reads a named value back, or an empty variant if the name is not present.

Parameter	Type	Direction	Description
`Name`	`String`	in	Property name.

Ownership and messaging

OnlyOwnerCanWrite

Property — read/write — `Boolean`

Restricts writing to the channel's owner — the component that created it. With this set, every `Add…Sample` call is discarded unless it falls between `BeforeOwnerWriteSample` and `AfterOwnerWriteSample`.

BeforeOwnerWriteSample

Method

Opens an owner write. Also takes the channel's write lock, so the pair is both an authorisation marker and mutual exclusion against other writers.

AfterOwnerWriteSample

Method

Closes an owner write.

The pair must balance

A missing `After` call leaves the write lock held and **deadlocks every other writer** to that channel.

Rejected writes are silent — no error, the sample simply never appears. Both calls are harmless when `OnlyOwnerCanWrite` is `False`, so they can be left in unconditionally.

SendMessageEvent

Method — returns `Boolean`

Sends a free-form text message to the channel, and if it declines, onward to the channels it is computed from — stopping at the first handler that consumes it.

Parameter	Type	Direction	Description
`MsgHeader`	`String`	in	Names the command.
`MsgData`	`String`	in	Payload.
`MsgOut`	`String`	out	The handler's reply.

Returns `True` if some handler consumed the message. When `False`, `MsgOut` is meaningless.

SendMessageToOwnerEvent

Method — returns `Boolean`

As above but offered **only** to the channel's own producer, with no propagation upstream. Same parameters and result.

Ordinary acquisition channels — analog input, CAN, counter — handle nothing, so both methods return `False` on them. Only mathematics modules respond, and the message vocabulary is defined entirely by the receiving module.

Freeze mode

SetFreezeMode

Method

Switches this channel between reading from the live buffer and reading from the frozen copy, for a visual control that must keep showing frozen data.

Parameter	Type	Direction	Description
`Freeze`	`Boolean`	in	`True` reads the freeze buffer.

Legacy members

These remain for compatibility and should not be used in new code.

Member	Instead use
`Index`	`IndexEx`
`GetIndex1`	`IndexEx`
`Scale_`	`Scale`
`RBBufSize`	`IBBufSize`
`RBDataSize`	`IBDataSizeEx`
`RBPos`	`IBPosEx`
`RBValues`	`IBValuesEx`
`GetRBValues`	`IBValuesEx`
`GetDBAddress`	`GetDBAddress64`
`GetTSAddress`	`GetTSAddress64`
`GetBinAddress`	`GetBinAddress64`

The `RB…` family addressed a reduced buffer that no longer exists. `Index` and `GetIndex1` use a record type that cannot express more than five index levels or carry the measurement-unit element.

ValueChanged

Method

Does nothing. The notification it once raised is now issued automatically when a sample is added to a control channel. Existing calls are harmless; new code should simply add the sample.

FastCalc

Method

Incomplete — not usable. Use `FastCalcInt16`.

FastCalcInt32

Method

Incomplete — not usable. Use `FastCalcInt16`.

Internal members

`AddIntegerSampleWithCalc`, `AddShortintSampleWithCalc` and `AddSmallintSampleWithCalc` are internal to DewesoftX. `FirstX` and `SecondX` serve one specific project and have no general meaning.

IChannelConnection

A reader on one channel — how you get samples out of DewesoftX.

Create one with `IChannel.CreateConnection`. Each connection keeps its own read position, so several readers on one channel do not interfere.

A connection is used in one of two shapes. Either you take **values** — a flat run of samples — or **blocks**, where the channel's data is cut into fixed-size pieces and you take whole pieces. Which you want decides whether you call the `…Values` or the `…Blocks` methods, and whether `BlockSize` matters.

PYTHON

```python
ch = app.Data.FindChannel("AI 1")
conn = ch.CreateConnection()
conn.AType = 0            # take the newest samples
conn.Start()

n = conn.NumValues
if n > 0:
    values = conn.GetDataValues(n)
    for v in values:
        print(v)
```

C#

```csharp
dynamic ch = app.Data.FindChannel("AI 1");
dynamic conn = ch.CreateConnection();
conn.AType = 0;           // take the newest samples
conn.Start();

int n = (int)conn.NumValues;
if (n > 0)
{
    dynamic values = conn.GetDataValues(n);
    for (int i = 0; i < n; i++)
        Console.WriteLine(values[i]);
}
```

Read the timestamps alongside the values with `GetTSValues`, but only on an **asynchronous** channel — a synchronous one has none.

Setting the connection up

AType

Property — read/write — `Integer`

What the connection reads:

Value	Reads
`0`	the most recent data
`1`	overlapping blocks, stepping by `Overlap`
`2`	data around triggers
`3`	only data that has arrived since the last read

BlockSize

Property — read/write — `Integer`

How many samples make up one block, for the `…Blocks` methods. Leave it alone if you are reading values rather than blocks.

Overlap

Property — read/write — `Integer`

How far consecutive blocks overlap, in samples. Only used when `AType` is `1`.

Start

Method

Opens the connection so it begins tracking data. Nothing accumulates before this is called.

Reset

Method

Puts the read position back to the beginning.

Channel

Property — read-only — `IChannel`

The channel this connection reads — the way back if you are holding connections without their channels.

Finding out what is available

NumValues

Property — read-only — `Integer`

How many samples are waiting.

This is `-1`, not `0`, when there is nothing to read — which is the state before acquisition has produced anything. Test for `> 0` rather than truthiness, or a `-1` will read as "there is data" and the read that follows will ask for a negative count.

NumBlocks

Property — read-only — `Integer`

How many whole blocks are waiting, given the current `BlockSize`.

Reading

The four reads pair up: data and timestamps, values and blocks. Ask for no more than `NumValues` or `NumBlocks` reports.

GetDataValues

Method — returns `Variant`

The next samples, as an array.

Parameter	Type	Direction	Description
`NumValues`	`Integer`	in	How many samples to take.

GetTSValues

Method — returns `Variant`

The timestamps of those samples, as an array. Call it alongside `GetDataValues` with the same count to get value and time together.

Parameter	Type	Direction	Description
`NumValues`	`Integer`	in	How many timestamps to take.

Asynchronous channels only. A synchronous channel stores no timestamps, and calling this on one faults inside DewesoftX instead of returning an error. Check `IChannel.Async` first, and derive the time from the sample rate for synchronous channels.

GetDataBlocks

Method — returns `Variant`

The next whole blocks of samples, as an array.

Parameter	Type	Direction	Description
`NumBlocks`	`Integer`	in	How many blocks to take.

GetTSBlocks

Method — returns `Variant`

The timestamps of those blocks.

Parameter	Type	Direction	Description
`NumBlocks`	`Integer`	in	How many blocks to take.

The `...1` variants

`GetDataValues1`, `GetTSValues1`, `GetDataBlocks1` and `GetTSBlocks1` do exactly what their unsuffixed counterparts do. The difference is only in how the array comes back: the plain forms return it, the `1` forms hand it back through an out parameter.

Use the plain forms from Python and C#. The `1` forms exist for callers whose language cannot take a variant array as a return value.

> **Data and timestamps advance separately**
>
> A connection keeps **two** read positions, one for data and one for timestamps, so `GetDataValues` and `GetTSValues` do not disturb each other. That is what lets you read a run of samples and then its times and have them line up.
>
> It also means the two can drift apart for good. Ask for 100 values and 50 timestamps and the timestamp position is now 50 behind — every later pair is mismatched, with nothing reporting it. Read both with the same count, every time, or read only one of them.
>
> This applies where the connection consumes as it reads — `AType` `3`. With `AType` `0` each read returns the newest data afresh, so repeating a read gives the same samples again rather than the next ones.

IChannelGroup

One channel group — a branch of the channel tree, holding the channels from one source.

Get it from `IChannelGroups.Item` or `IChannel.Group`.

Extends `IChannelList`, so a group is also a list of its channels: `Count` and `Item` work here as they do on any channel list.

PYTHON

```python
g = app.Data.Groups.Item(0)
print(g.Name, g.Count, "channel(s), export rate", g.ExportRate)
```

C#

```csharp
dynamic g = app.Data.Groups.Item[0];
Console.WriteLine($"{g.Name} {g.Count} channel(s), export rate {g.ExportRate}");
```

Members

Name

Property — read-only — `String`

The group's name — `AI` for the analog inputs, and so on.

ExportRate

Property — read/write — `Integer`

The rate the group's channels are written at on export, as a divider of the acquisition rate. Set per group rather than per channel, so changing it moves every channel in the group.

GetIndexName

Method — returns `String`

The display name for a position in the channel tree.

Parameter	Type	Direction	Description
`Index`	`T_ChIndex`	in	The tree position.

Groups whose channels are laid out in more than one dimension — a matrix of inputs, say — use this to turn a position into something readable. `IChannelGroup2` offers a shorter form of the same thing.

IChannelGroup2

A shorter form of a channel tree position's name.

Extends `IChannelGroup`, and groups carry both — but see below.

> **Not usable from an automation client**
>
> Groups register `IChannelGroup` for late binding and not this interface, so `GetIndexNameShort` does not resolve by name from Python or C#. Everything on `IChannelGroup` works normally.

GetIndexNameShort

Method — returns `String`

The short display name for a position in the channel tree — the abbreviated counterpart of `GetIndexName`.

Parameter	Type	Direction	Description
`Index`	`T_ChIndex`	in	The tree position.

IChannelGroups

The channel groups in the setup — the top level of the channel tree.

Get it from `IData.Groups`. Each group is an `IChannelGroup`.

Groups are how DewesoftX divides channels by origin: analog inputs, counters, mathematics, each plug-in. A channel's own group is `IChannel.Group`.

PYTHON

```python
groups = app.Data.Groups
for i in range(groups.Count):
    print(groups.Item(i).Name)
```

C#

```csharp
dynamic groups = app.Data.Groups;
for (int i = 0; i < (int)groups.Count; i++)
    Console.WriteLine(groups.Item[i].Name);
```

Members

Count

Property — read-only — `Integer`

Number of groups.

Item

Property — read-only — `IChannelGroup`

The group at a position. The list's default member, so it can be subscripted directly in languages that support that.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the groups can be iterated directly in languages that support it.

IChannelList

A read-only list of channels.

This is the plain channel-list shape, returned all over the interface — by `IData`, by `IMathItem.InputChannels` and `OutputChannels`, by `IAOGroup.AOChannels`, by `ITriggerCondition`, and by the input slots of mathematics and widgets.

Where a list can also be modified it carries `IChannelListEx` as well, and both sets of names resolve on the same object.

PYTHON

```python
chans = app.Data.UsedChannels
for i in range(chans.Count):
    print(chans.Item(i).Name)
```

C#

```csharp
dynamic chans = app.Data.UsedChannels;
for (int i = 0; i < (int)chans.Count; i++)
    Console.WriteLine(chans.Item[i].Name);
```

Members

Count

Property — read-only — `Integer`

Number of channels in the list.

Item

Property — read-only — `IChannel`

The channel at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

This is the list's default member, so a language that supports default indexing can subscript the list directly.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the channels can be iterated directly in languages that support it.

IChannelListEx

A channel list you can change — the same object as `IChannelList`, with adding and removing on top.

Make one with `IApp.CreateChannelList`. Lists returned from elsewhere carry this interface too — the object is the same one, so `IChannelList`'s members resolve on it as well — but see the warning below before trying to modify one.

Build a list this way to hand to something that takes one, such as `IMathObject.AddModules_SingleInput`.

PYTHON

```
lst = app.CreateChannelList()
lst.Clear()
lst.AddCh(app.Data.FindChannel("AI 1"))
lst.AddCh(app.Data.FindChannel("AI 2"))
print(lst.Count)
```

C#

```
dynamic lst = app.CreateChannelList();
lst.Clear();
lst.AddCh(app.Data.FindChannel("AI 1"));
lst.AddCh(app.Data.FindChannel("AI 2"));
Console.WriteLine(lst.Count);
```

Most lists refuse to be modified

`AddCh`, `SetCh`, `UnsetCh`, `DeleteCh` and `Clear` all check a flag first and raise *Channel list is read only* when it is not set. A list handed to you by DewesoftX — the channels of a group, a math module's inputs — is generally not modifiable, whatever the interface suggests.

Build your own with `IApp.CreateChannelList` when you need a list you can fill.

Members

Count

Property — read-only — `Integer`

Number of channels in the list.

Item

Property — read-only — `IChannel`

The channel at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

AddCh

Method

Appends a channel.

Parameter	Type	Direction	Description
`Ch`	`IChannel`	in	The channel to add.

SetCh

Method

Puts a channel at a position, replacing whatever was there.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	The position to write.
`Ch`	`IChannel`	in	The channel.

UnsetCh

Method

Empties a position without removing it, so the positions after it do not move.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	The position to empty.

DeleteCh

Method

Removes a position entirely, closing the gap.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	The position to remove.

`UnsetCh` and `DeleteCh` are not the same. Unsetting leaves a hole and keeps `Count` and every later index as they were; deleting shifts everything after it down by one. Use unset where positions carry meaning — a slot per input — and delete

where the list is just a list.

Clear

Method

Empties the list.

ICntChannel

Counter channel settings — counter mode, filtering, the channels making up a counter pair, and up/down and update behaviour.

Not implemented

This interface is declared in the type library, but nothing implements it and the only member that returns one is its own `CntPair`. There is no way to obtain one and nothing to call.

ICppResamplerEngine

The resampling engine a C++ script module uses to bring its inputs onto a common time base.

Reports the acquisition and current sample rates, the block size, and how many past and future samples a calculation needs, and nominates the channel whose rate the others follow. Its channels are `IResamplerChannel`.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

ICustomDAQ

The older interface a custom DAQ device plug-in implements.

Superseded by `ICustomDAQ2`, which is what current plug-ins implement.

> **Implemented by a custom DAQ plug-in, not by DewesoftX**
>
> This is an inbound interface: a custom DAQ plug-in implements it and DewesoftX calls it. It does not derive from `IDispatch`, so a late-bound client could not use it in any case.

ICustomDAQ2

The interface a custom DAQ device plug-in implements.

A single `OnMessage` entry point: DewesoftX passes every request through it and the plug-in decides what to do, rather than implementing a method per operation.

> **Implemented by a custom DAQ plug-in, not by DewesoftX**
>
> This is an inbound interface: a custom DAQ plug-in implements it and DewesoftX calls it. It does not derive from `IDispatch`, so a late-bound client could not use it in any case.

ICustomExport

The interface a custom export format implements.

DewesoftX drives an export by calling these in order: `StartExport`, then per channel and per value the `Start…` / `Write…` / `End…` calls, then `EndExport`. The plug-in writes whatever file format it likes as the calls arrive.

`ICustomExport2` extends this with double precision, events and channel metadata; `ICustomExport3` adds a generic event hook.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`ExportType`	Which export this plug-in provides.
`Extension`	The file extension it writes.
`FileName`	The file being written.
`StartExport`	Begin an export.
`EndExport`	Finish it.
`StartInfo`	Begin the file's information section.
`EndInfo`	End it.
`WriteInfoString`	Write a text entry into the information section.
`WriteInfoInteger`	Write a whole number.
`WriteInfoSingle`	Write a real number.
`WriteInfoDate`	Write a date.
`EndHeader`	End the column header.
`StartChannel`	Begin one channel.
`EndChannel`	End it.
`DataCount`	How many values the channel has.
`StartDataFolder`	Begin a group of data.
`EndDataFolder`	End it.
`StartDataField`	Begin one field.
`StartTimeField`	Begin the time column.
`StartValue`	Begin one value.

Member	Description
`EndValue`	End it.
`WriteValue`	Write a value.
`StartAbsValue`	Begin a value with an absolute timestamp.
`AbsoluteTime`	Whether times are absolute rather than relative.
`TimeIncrease`	The time step between values.
`SupportsAsync`	Whether the format can carry asynchronous channels.
`WriteAsyncValue`	Write an asynchronous value.
`SupportsSRDiv`	Whether the format can carry rate-divided channels.

ICustomExport2

The extended custom export interface.

Extends `ICustomExport`, so a plug-in implementing this one has all of that interface's members too. What it adds is double precision, events, channel metadata, and a way in to `IApp`.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`SetApp`	Receives `IApp`.
`SetChannel`	Receives the channel being exported.
`SetChannelColor`	Its colour.
`SetRangeMin`	Its range minimum.
`SetRangeMax`	Its range maximum.
`SetAbsMin`	Its absolute minimum.
`SetAbsMax`	Its absolute maximum.
`SetTrigOffset`	The trigger offset.
`SupportsDouble`	Whether the format can carry double precision.
`SetDoubleFloat`	Switch to double precision.
`WriteDoubleValue`	Write a double-precision value.
`WriteAsyncDoubleValue`	Write an asynchronous double-precision value.
`StartEvents`	Begin the events section.
`StopEvents`	End it.
`WriteEvent`	Write one event.
`GetDWTypeLibVersion`	Report which type library version the plug-in was built against.

ICustomExport3

A generic event entry point for a custom export.

Implemented alongside `ICustomExport` and `ICustomExport2`.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`OnEvent`	An event has occurred; the code says which.

IDIChannel

A digital input channel's own settings: its filter, whether the input is inverted, and its hardware trigger levels.

The trigger levels are an `IDigitalTrigLevel`. The same thresholds are reachable from the automation surface through `IDaq.GetDITrgLevel`.

> **Not reachable from an automation client**
>
> Channels of this kind carry the interface, but unlike `IDaqChannel` it is not registered for late binding — so none of its member names resolve from Python or C#. Only the `IChannel` members of such a channel are usable.

IDIGroup

The digital input channels of the acquisition hardware.

Not reachable from an automation client

Channels of this kind carry the interface, but unlike `IDaqChannel` it is not registered for late binding — so none of its member names resolve from Python or C#. Only the `IChannel` members of such a channel are usable.

IDIPort

One digital input port — a group of digital lines read as a unit.

Not reachable from an automation client

Channels of this kind carry the interface, but unlike `IDaqChannel` it is not registered for late binding — so none of its member names resolve from Python or C#. Only the `IChannel` members of such a channel are usable.

IDaq

The acquisition hardware as a whole — the devices present, their trigger levels, and the clock they run on.

Get it from `IApp.Daq`. It is nothing until `Init` has run.

Per-channel hardware settings live on `IDaqChannel` instead.

PYTHON

```python
daq = app.Daq
print(daq.CardCount, "device(s)")
for i in range(daq.CardCount):
    info = daq.GetDeviceInfo(i)
    print(info.Name, info.SerialNumber, info.AICount, "AI")
```

C#

```csharp
dynamic daq = app.Daq;
Console.WriteLine($"{daq.CardCount} device(s)");
for (int i = 0; i < (int)daq.CardCount; i++)
{
    dynamic info = daq.GetDeviceInfo(i);
    Console.WriteLine($"{info.Name} {info.SerialNumber} {info.AICount} AI");
}
```

Devices

CardCount

Property — read-only — `Integer`

How many acquisition devices are present. Indexes for `GetDeviceInfo` and `GetDeviceCode` run from `0` to `CardCount - 1`.

DaqType

Property — read-only — `Integer`

Which driver is in use, as an index into the drivers registered in the running program. Values from `255` upwards identify a multi-device controller rather than a single card.

This is an internal index, not a stable identifier — the same number can mean a different driver in another build, or with a different set of drivers present. Use `GetDeviceInfo` to find out what the hardware actually is.

GetDeviceInfo

Method — returns a `DaqDeviceInfo` structure

Everything DewesoftX knows about one device.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Device position, `0` to `CardCount - 1`.

The structure's fields:

Field	Type	Description
`Name`	`String`	The device's display name.
`SerialNumber`	`String`	Device serial number.
`Firmware`, `FirmwareSub`	`String`	Firmware versions of the device and its sub-board.
`CalDate`	`String`	Calibration date.
`DriverVersion`	`String`	Version of the driver handling it.
`Used`	`String`	`YES` or `NO` — whether the device is in use.
`OptionCount`, `OptionTypes`	`String`	Installed hardware options.
`AICount`, `DICount`, `CNTCount`	`String`	Analog, digital and counter inputs the device has.
`AIUsedCount`, `DIUsedCount`, `CNTUsedCount`	`String`	How many of each are enabled.
`ConnectedInterface`	`Integer`	How it is attached: `0` unknown, `1` USB, `2` EtherCAT, `3` openDAQ.
`ProductCode`	`Integer`	Hardware product code.
`RevisionNumber`	`String`	Hardware revision.
`PrevId`, `PrevPort`, `ParentId`	`Integer`	Position in the device chain.
`SystemSerialNumber`, `BoxSerialNumber`, `BoxSystemSerialNumber`	`String`	Serial numbers of the enclosing system and box.
`Flags`	`Integer`	Device flag bits.

Most of these are strings even where they hold a number, the channel counts included — convert before doing arithmetic on them. Fields a driver does not fill in come back empty rather than absent.

GetDeviceCode

Method — returns `String`

The device's identifying code.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Device position, `0` to `CardCount - 1`.

SetDeviceCalDate

Method — returns `Integer`

Writes a new calibration date into the device.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Device position.
`CalDate`	`String`	in	The new date.
`Pwd`	`String`	in	Calibration password.

Returns `0` on success, `-1` when the driver does not support writing the date.

Trigger levels

Digital and counter inputs decide what counts as an edge by voltage, and these four members read and write those thresholds on the hardware. Levels are in **millivolts**.

The same values are readable per channel through `IDigitalTrigLevel`.

Every one of them returns `0` on success and `-1` when the driver has not implemented it — which is the default for drivers that have not, so check the result rather than assuming a write landed.

GetDITrgLevel

Method — returns `Integer`

Reads the thresholds of one digital input pin.

Parameter	Type	Direction	Description
`PinNr`	`Integer`	in	The pin.
`TrgLevel`	`Integer`	out	Threshold an edge must cross, in mV.
`ReTrgLevel`	`Integer`	out	Level the signal must return past before the next edge, in mV.
`Coupling`	`Byte`	out	`0` DC, `1` AC.
`SupportLevel`	`Byte`	out	How much of this the hardware supports.

SetDITrgLevel

Method — returns `Integer`

Writes the thresholds of one digital input pin.

Parameter	Type	Direction	Description
`PinNr`	`Integer`	in	The pin.
`TrgLevel`	`Integer`	in	Threshold in mV.
`ReTrgLevel`	`Integer`	in	Return level in mV.
`Coupling`	`Byte`	in	`0` DC, `1` AC.

GetCNTTrgLevel

Method — returns `Integer`

The same for one pin of a counter input.

Parameter	Type	Direction	Description
`CNTNr`	`Integer`	in	The counter channel.
`PinType`	`Byte`	in	Which pin of that counter.
`TrgLevel`	`Integer`	out	Threshold in mV.
`ReTrgLevel`	`Integer`	out	Return level in mV.
`Coupling`	`Byte`	out	`0` DC, `1` AC.
`SupportLevel`	`Byte`	out	How much the hardware supports.

SetCNTTrgLevel

Method — returns `Integer`

Writes a counter input pin's thresholds.

Parameter	Type	Direction	Description
`CNTNr`	`Integer`	in	The counter channel.
`PinType`	`Byte`	in	Which pin of that counter.
`TrgLevel`	`Integer`	in	Threshold in mV.
`ReTrgLevel`	`Integer`	in	Return level in mV.
`Coupling`	`Byte`	in	`0` DC, `1` AC.

Clock and state

GetAbsTime

Method — returns `Double`

The current absolute time from the timing source, as a date-and-time value.

TimingTracking

Property — read-only — `Boolean`

`True` while the timing source is locked and tracking — a GPS or IRIG input that has acquired sync, for instance.

CreateSynchronization

Method — returns `ISynchronization`

Creates an object for configuring how this hardware locks to an external clock. A fresh one is returned on every call.

DataLost

Property — read/write — `Boolean`

`True` when samples have been dropped. Writable so you can clear it back to `False` once you have noticed and recorded the fact.

CanAutoCalculate

Property — read-only — `Boolean`

Whether the driver can work out its own acquisition rates rather than being told them.

IOControl

Method — returns `Integer`

Sends a driver-specific command to the hardware.

Parameter	Type	Direction	Description
`Msg`	`Integer`	in	The command code.
`InParam`	`Variant`	in	The command's argument.
`OutParam`	`Variant`	out	The command's result.

Returns `0` on success. The codes belong to whichever driver is loaded and are not a published set. For amplifier and sensor settings use `IAmplifier.IOControl` instead, whose codes are listed.

IDaqChannel

The hardware side of an analog input channel — the card's own gain and offset, the amplifier behind it, and the sensor assigned to it.

Every analog input channel carries this interface **as well as** `IChannel`, so one object serves both and you can read members of either from the same reference. Get one from `IDaqGroup.Item`, or from any channel you already hold.

PYTHON

```python
ch = app.Data.UsedChannels.Item(0)
print(ch.Name, ch.CardGain, ch.CardOffset, ch.CardBitResolution)
print("sensor:", ch.GetSensor(), ch.GetSensorType())
```

C#

```csharp
dynamic ch = app.Data.UsedChannels.Item[0];
Console.WriteLine($"{ch.Name} {ch.CardGain} {ch.CardOffset} {ch.CardBitResolution}");
Console.WriteLine($"sensor: {ch.GetSensor()} {ch.GetSensorType()}");
```

Where a member name exists on both interfaces, prefix it to say which you mean — `ch.IDaqChannel_ModuleType`. Without a prefix the plain name is resolved on whichever interface declares it, which is what you want almost always.

The card

These describe the acquisition card's own conversion, before the amplifier.

CardGain

Property — read-only — `Double`

The gain the card applies. Write it with `SetCardGain` — there is no setter on the property itself.

SetCardGain

Method

Sets the card gain.

Parameter	Type	Direction	Description
`Gain`	`Single`	in	The new gain.

The value is stored as a single but read back through `CardGain` as a double, so a value you write may not compare equal to the one you read.

CardOffset

Property — read-only — `Double`

The offset the card applies.

CardBitResolution

Property — read-only — `Integer`

Bits per sample the card converts at — `16` or `24` on current hardware.

GetBoardOpt

Method — returns `Word`

Reads one hardware option of the card this channel sits on — input type, coupling or input mode, depending on the option.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Which option, by position in the card's option list.

Returns the position of the setting currently selected within that option.

The option is identified by matching its **translated** name, so which index means what depends on the interface language, and an option the card does not have reads as `0` rather than reporting an error. Treat the result as advisory.

SetBoardOpt

Method

Selects one hardware option of the card.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Which option, by position.
`Used`	`Word`	in	The setting to select within that option.

The amplifier

Amplifier

Property — read-only — `IAmplifier`

The amplifier module conditioning this channel. Everything adjustable about the module is reached through its `IOControl`.

ModuleGain

Property — read-only — `Double`

The gain the amplifier module contributes, separate from `CardGain`.

ModuleOffset

Property — read-only — `Double`

The offset the amplifier module contributes.

ModuleType

Property — read-only — `Integer`

The module's type code, or `-1` when no module is fitted — which is what a simulated or plain voltage input reports.

AmplifierAnalyseInfo

Property — read-only — `String`

A summary of the amplifier's settings as one line of text, for display beside the channel.

The sensor

GetSensor

Method — returns `String`

The name of the sensor assigned to the channel, or empty when none is.

SetSensor

Method — returns `Boolean`

Assigns a sensor by name.

Parameter	Type	Direction	Description
`SensorName`	`String`	in	The sensor's name, as it appears in the sensor database.

The result does not mean what it looks like

`True` means assigning the sensor **changed the amplifier's settings**, not that the sensor was found and assigned. A sensor that does not exist, and a sensor that assigned cleanly without disturbing the amplifier, both return `False`.

Confirm the assignment by reading `GetSensor` back.

GetSensorType

Method — returns `String`

The assigned sensor's type, or empty when no sensor is assigned.

SensorDescription

Property — read-only — `String`

The assigned sensor's description, or empty when no sensor is assigned.

SensorCalDate

Property — read-only — `String`

When the sensor was last calibrated, as text. Empty when no sensor is assigned, and also when the sensor carries no calibration period — so an empty value does not by itself mean the sensor is uncalibrated.

CustomSensorScale

Property — read/write — `Single`

A scale factor applied on top of the sensor's own scaling. Reads as `1` when no sensor is assigned, and writing it then does nothing.

CustomSensorOffset

Property — read/write — `Single`

An offset applied on top of the sensor's own scaling. Reads as `0` when no sensor is assigned, and writing it then does nothing.

Zeroing

AutoZero

Property — read/write — `Boolean`

Whether the channel takes part in auto-zeroing.

This is not a flag of its own — it reflects whether the channel is in an auto-zero group. Setting it to `True` puts the channel in the main group, and `False` removes it from any group. A channel placed in some other group by the setup reads as `True`, and setting it to `True` again moves it to the main group.

IDaqData

What a hardware module reports about one analog input.

Covers the module's identity and address, its amplitude and offset, the ranges and filters it offers, its input coupling and highpass type, thermocouple linearisation, IEPE excitation, and error and overflow state. The frequency-input members configure a `FREQ` module's coupling, output filter and trigger level.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IDaqGroup

The analog input channels of the acquisition hardware.

Get it from `IApp.DaqGroup`. Each channel is an `IDaqChannel`, which also carries `IChannel` — so a channel from here answers to both.

PYTHON

```python
g = app.DaqGroup
for i in range(g.Count):
    ch = g.Item(i)
    print(ch.Name, ch.CardGain)
```

C#

```csharp
dynamic g = app.DaqGroup;
for (int i = 0; i < (int)g.Count; i++)
{
    dynamic ch = g.Item[i];
    Console.WriteLine($"{ch.Name} {ch.CardGain}");
}
```

Members

Count

Property — read-only — `Integer`

Number of analog input channels the hardware provides — every one it has, not only those switched on.

Item

Property — read-only — `IDaqChannel`

The channel at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the channels can be iterated directly in languages that support it.

IData

The measurement data: channel lists, sample rates, acquisition progress, the analysis cursor, and the time at which storing began.

Obtain it from `IApp.Data`. It stays valid for the lifetime of the connection — hold the reference rather than fetching it per call.

PYTHON

```python
data = app.Data
print(data.SampleRate, data.UsedChannels.Count)
```

C#

```csharp
dynamic data = app.Data;
Console.WriteLine($"{data.SampleRate} {data.UsedChannels.Count}");
```

Mode and state

MeasureMode

Property — read-only — `Boolean`

`True` while DewesoftX is in measure (acquisition) mode.

AnalyseMode

Property — read-only — `Boolean`

`True` while DewesoftX is in analysis mode.

Both can be `False` at once — neither mode is active before initialisation completes.

FreezeMode

Property — read-only — `Boolean`

`True` while the display is frozen. Acquisition continues; only the on-screen update is paused.

ExternalClock

Property — read-only — `Integer`

The external clock rate in Hz, or `0` when no external clock is configured. Write it with `SetExternalClock`.

ExternalTrigger

Property — read/write — `Boolean`

`True` once an external trigger event has occurred. Writing it reflects the *external trigger* checkbox in **Config**.

SetExternalClock

Method

Sets the external clock rate.

Parameter	Type	Direction	Description
`Value`	`Integer`	in	Rate in Hz. `0` disables the external clock.

Sample rates and block size

SampleRate

Property — read-only — `Integer`

The rate at which data is being acquired, in Hz. Which rate that is depends on where DewesoftX currently is:

Situation	Value returned
Config	the setup sample rate
Measure mode	the dynamic acquisition rate

It never returns the reduced storing rate.

SampleRateEx

Property — read-only — `Double`

As `SampleRate`, but keeps fractional rates such as `10.5` Hz. Prefer this one; `SampleRate` truncates.

Samples

Property — read-only — `Integer`

The number of samples in one data block, for a synchronous channel with no sample-rate divider. Together with `GetSamplesAcquired` this converts a block count into an absolute sample position.

SRDivLCM

Property — read-only — `Integer`

Internal. The lowest common multiple of the configured sample-rate dividers.

Channel lists

Four lists are exposed, and only two of them should be used.

Property	Contents
`AllChannels`	every channel the configured devices and mathematics define
`UsedChannels`	those set to used in Config
`ActiveChannels`	internal — do not use
`ShownChannels`	not recommended — see below

All four are stale until `BuildChannelList` is called after any channel's `Used` flag changes.

AllChannels

Property — read-only — `IChannelList`

Every channel the configured devices and mathematics define, whether used or not.

UsedChannels

Property — read-only — `IChannelList`

The channels set to used in **Config**. This is the list most automation clients want.

ActiveChannels

Property — read-only — `IChannelList`

Internal to DewesoftX. Use `AllChannels` or `UsedChannels` instead.

ShownChannels

Property — read-only — `IChannelList`

The channels that can be displayed — CAN messages, for instance, cannot be. The list is only accurate in measure mode *after* Config has been left, so it is not dependable. Use `AllChannels` or `UsedChannels`.

BuildChannelList

Method

Rebuilds `AllChannels`, `UsedChannels` and `ShownChannels`. Call it after changing any channel's `Used` flag and before reading those lists, or you will read the previous contents.

ApplyChannels

Method

Applies channels a plug-in has mounted or unmounted. DewesoftX calls this itself after `IPlugin2.NewSetup`, `IPlugin2.LoadSetup`, and `IPlugin4.OnEvent` for `evOnUpdateXML`, so a plug-in only needs it when it mounts channels outside those paths.

InputGroups

Property — read-only — `IInputGroups`

The input groups. There is no lookup by identifier — iterate and compare each group's GUID.

Finding channels

FindChannelByIndexEx

Method — returns `IChannel`

Finds a channel by its channel index. This is the recommended lookup: an index is unique, whereas names are not.

Parameter	Type	Direction	Description
`Index`	`Variant`	in	An array of `Integer`. Element `0` must be `0`, element `1` is the group ID, and the rest give the channel's position within it.

The array holds `IndexLevel + 1` elements, and fewer than three never matches anything. `[0, 1, 0]` is the first analog input, `[0, 1, 1]` the second.

`IChannel.IndexEx` returns exactly this array for a channel you already hold, so an index round-trips without being rebuilt. Do not use the older `IChannel.Index` record: its `Index1` field is the **second** array element, so passing those fields back matches nothing.

Returns nothing if no channel matches.

FindChannel

Method — returns `IChannel`

Finds a channel by name. Names are **not unique** — several channels may share one, and the first match wins — so prefer `FindChannelByIndexEx`.

Parameter	Type	Direction	Description
`Name`	`String`	in	Channel name. Matched case-insensitively against either the short name or the long `Group/Name` form.

Returns nothing if no channel matches.

Channel index names

GetIndexNameEx

Method — returns `String`

The display name for a channel index.

Parameter	Type	Direction	Description
`Index`	`Variant`	in	An array of `Integer` holding the index, one element per level.

IndexStringToIndex

Method — returns `T_ChIndex`

Parses the textual form of a channel index — the inverse of `GetIndexNameEx`. An optional `unit:` prefix selects a remote measurement unit; without it the index is taken as local.

Parameter	Type	Direction	Description
`Value`	`String`	in	The textual channel index.

Returns the deprecated `T_ChIndex` record, which is limited to five levels and cannot carry the measurement-unit element. Prefer the array form used by `FindChannelByIndexEx` and `IChannel.IndexEx`.

GetTimestring

Method — returns `String`

Formats a time value the way DewesoftX itself displays it, so exported or logged timestamps match what the user sees on screen.

Parameter	Type	Direction	Description
`Time`	`Double`	in	Time in seconds.
`TimeDisplay`	`TimeDisplay`	in	`0` relative, `1` absolute, `2` absolute time, `3` absolute day and time.
`ForceZeroPrec`	`Boolean`	in	`True` renders no decimal places.

Channel groups

Groups

Property — read-only — `IChannelGroups`

The channel groups. Every channel belongs to exactly one.

`Groups` is indexed **by position, not by group identifier**, and positions are not a stable contract — groups are appended over time, so an ordinal that is correct today may not be in a later release. Match on `IChannelGroup.Name` rather than hard-coding one:

PYTHON

```python
plugins = next(g for g in data.Groups if g.Name == "Plugins")
```

C#

```csharp
dynamic plugins = Enumerable.Range(0, (int)data.Groups.Count)
    .Select(i => data.Groups[i])
    .First(g => g.Name == "Plugins");
```

The current positions are:

#	Name	#	Name
0	`Control`	13	`Math`
1	`AI`	14	`Variables`
2	`PAD`	15	`Video`
3	`MathOld`	16	`Import`
4	`CAN`	17	`DAQ Out`
5	`GPS`	18	`VC Info`
6	`CNT`	19	`System Monitor`
7	`DI`	20	`Control channels`
8	`Plugins`	21	`Event log`
9	`Power`	22	`RTC`
10	`COM`	23	`Daq/Additional`
11	`AO`	24	`RT`
12	`Remote`	25	`openDAQ`

There are two similarly named groups: `MathOld` at position 3 and `Math` at position 13. Mathematics channels are in `Math`.

Acquisition progress

GetSamplesAcquired

Method

How much data has been acquired since the measurement started.

Parameter	Type	Direction	Description
`Mid`	`Integer`	out	Number of complete blocks of `Samples` each.
`Dir`	`Integer`	out	Remaining samples not yet forming a complete block.

Both values are `out` parameters, which the two languages surface differently:

PYTHON

```
mid, dir_ = data.GetSamplesAcquired()
total = mid * data.Samples + dir_
elapsed = total / data.SampleRateEx     # seconds since acquisition started
```

C#

```
data.GetSamplesAcquired(out int mid, out int dir);
long total = (long)mid * data.Samples + dir;
double elapsed = total / data.SampleRateEx;   // seconds since acquisition started
```

Only meaningful inside the data callbacks

Read this from `OnGetData`, `OnStopAcq` or `OnStopStoring` only.

In `OnStartStoring` it returns the count from the *previous* acquisition, not zero. In `OnStartAcq` it always returns zero. Both are easy to mistake for real values.

MaxCalcDelay

Property — read-only — `Integer`

The highest calculation delay, in samples, across all channels.

It is only set once every device, math module and plug-in has contributed its data — that is, at the end of an `OnGetData` cycle. On the first `OnGetData` call it is therefore `0`. For the delay across only your own input channels, iterate them and take the maximum of `IChannel.CalcDelay` instead.

CalcDelayLimit

Property — read-only — `Integer`

The lower bound applied to the calculation delay, in samples. It derives from a default of 0.2 s converted at the current sample rate, and the delay actually used is the greater of this and the channel's own. Reads `0` until acquisition configures it.

The analysis cursor and region

These are analysis-mode properties. Each exists in two forms: a `T_RecordPosition` pair, and a `…D` variant in seconds. The `…D` forms are far easier to work with, and are what you want unless you already hold a record position.

Position form	Seconds form	Meaning
`StartStamp`	`StartStampD`	first position the cursor may take
`CurrentPos`	`CurrentPosD`	where the cursor is now
`EndStamp`	`EndStampD`	last position the cursor may take

A `T_RecordPosition` holds `Mid` (complete blocks) and `Dir` (remaining samples); the absolute sample number is `Mid * Samples + Dir`.

CurrentPos

Property — read/write — `T_RecordPosition`

The cursor position, as a record position.

CurrentPosD

Property — read/write — `Double`

The cursor position, in seconds.

StartStamp

Property — read/write — `T_RecordPosition`

The earliest position the cursor may take, as a record position.

StartStampD

Property — read/write — `Double`

The earliest cursor position, in seconds.

EndStamp

Property — read/write — `T_RecordPosition`

The latest position the cursor may take, as a record position.

EndStampD

Property — read/write — `Double`

The latest cursor position, in seconds.

SelectDataRegion

Method

Selects the region between two times, in seconds — the equivalent of setting `StartStampD` and `EndStampD` together, and the simplest way to scope an export or a recalculation.

Parameter	Type	Direction	Description
`StartPos`	`Double`	in	Region start, in seconds.
`StopPos`	`Double`	in	Region end, in seconds.

Both arguments are shifted by `RelativeTimeOffset` and then clamped to the data actually present, so values outside the file are tolerated rather than rejected. If `StopPos` is below `StartPos` it is raised to match, giving an empty region rather than an inverted one.

FirstTimeStamp

Property — read-only — `T_RecordPosition`

The first position for which data exists.

MRealTimeStamp

Property — read-only — `T_RecordPosition`

The most recent position for which data exists. With `FirstTimeStamp` this is the range `SelectDataRegion` clamps to.

RelativeTimeOffset

Property — read/write — `Double`

An offset in seconds added to relative times, so displayed and selected times can be shifted away from the start of the file. `SelectDataRegion` applies it.

Storing time

StartStoreTime

Property — read-only — `DateTime`

Local time at which storing began.

StartStoreTimeUTC

Property — read-only — `DateTime`

UTC time at which storing began. Read it from `OnGetData` or `OnStopAcq` only — elsewhere it may not yet be set, and it returns `-1` before storing starts.

SetStartStoreTimeUTC

Method

Overrides the UTC start-of-storing time, for a client that owns the timing.

Parameter	Type	Direction	Description
`Time`	`DateTime`	in	The UTC start time.

Intermediate buffers

DewesoftX reduces acquired data into successive levels of intermediate buffer, each holding minimum, maximum, average and RMS values over a block of the level below. These properties describe that hierarchy.

IBLevels

Property — read-only — `Integer`

The number of intermediate buffer levels. `0` means none are in use.

IBRate

Property — read-only — `Integer`

The sample rate of one intermediate buffer level.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level, `0` to `IBLevels - 1`.

IBAbsRate

Property — read-only — `Integer`

The absolute sample rate accumulated up to and including the given level.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level, `0` to `IBLevels - 1`.

IBAbsMidRate

Property — read-only — `Integer`

Project-specific. Not part of the general interface.

Parameter	Type	Direction	Description
`Level`	`Integer`	in	Buffer level.

Coordinated access

StartDataSync

Method

Begins a block in which several channels can be read or written consistently, for use with `IChannelConnection`.

Always pair it with `EndDataSync` in a construct that runs on failure, or the data lock is never released.

PYTHON

```python
data.StartDataSync()
try:
    ...
finally:
    data.EndDataSync()
```

C#

```csharp
data.StartDataSync();
try { /* ... */ }
finally { data.EndDataSync(); }
```

EndDataSync

Method

Ends the block opened by `StartDataSync`.

ManyChannelsDestroying

Property — read/write — `Boolean`

Suppresses the per-channel bookkeeping that normally runs each time a channel is removed. Set it while deleting many channels and clear it afterwards, so the rebuild happens once instead of once per channel.

While it is set DewesoftX never rebuilds its active-channel list, so clear it in a `finally` block.

Other

MetaDataParser

Property — read-only — `IMetaDataParser`

The parser for metadata expressions, used to resolve placeholders such as the current cursor position into text.

ProcessingMarkers

Property — read-only — `IProcessingMarkerList`

The processing markers defined for the measurement.

Legacy members

These remain for compatibility and should not be used in new code.

Member	Instead use
`FindChannelByIndex`	`FindChannelByIndexEx`
`FindChannelByIndex1`	`FindChannelByIndexEx`
`GetIndexName`	`GetIndexNameEx`
`GetIndexName1`	`GetIndexNameEx`
`GetIndexNameShort`	`GetIndexNameEx`
`GetIndexNameShort1`	`GetIndexNameEx`

`FindChannelByIndex`, `GetIndexName` and `GetIndexNameShort` take the deprecated `T_ChIndex` record, which cannot represent the measurement-unit element of an index and is limited to five levels.

A late-bound client cannot call these at all. A record can be *returned* to Python or C# but not passed back **in**, so each of these fails with *"Bad variable type"* however the record was obtained. Use the `Ex` forms, which take an array — and `IChannel.IndexEx` to obtain one.

The variants ending in `1` take the same index as six separate integers instead of a record. They exist because LabVIEW cannot pass a record, and are of no use from Python or C#.

IDataSection

One contiguous stretch of stored data — the span between a start and a stop event. Sections are what a triggered measurement produces: each time the store trigger fires, the resulting data becomes a section.

Sections are independent of data blocks, so a section boundary does not line up with a block boundary.

Get them from `ILoadEngine.DataSections`.

PYTHON

```python
sections = app.LoadEngine.DataSections
for i in range(sections.Count):
    s = sections[i]
    print(s.Time, s.DataCount, s.TrigPos)
```

C#

```csharp
dynamic sections = app.LoadEngine.DataSections;
for (int i = 0; i < (int)sections.Count; i++)
{
    dynamic s = sections[i];
    Console.WriteLine($"{s.Time} {s.DataCount} {s.TrigPos}");
}
```

Members

Time

Property — read-only — `DateTime`

Absolute start time of the section.

DataCount

Property — read-only — `Integer`

Number of samples in the section.

TrigPos

Property — read-only — `Integer`

Sample position of the trigger event within the section, counted from the section's start. With a pre-trigger configured this is greater than zero, and the samples before it are the pre-trigger data.

ReadData

Method — returns `Variant`

Reads one channel's data for the whole section, values and timestamps together.

Parameter	Type	Direction	Description
`Channel`	`IChannel`	in	Channel to read.
`Timestamps`	`Variant`	out	Array of `Double` timestamps.

Returns an array of `Single` values, the same length as the timestamp array.

PYTHON

```
values, timestamps = section.ReadData(channel)
```

C#

```
dynamic values = section.ReadData(channel, out object timestamps);
```

ReadData1

Method

The same as `ReadData` but returning the values through an out parameter rather than as a return value.

Parameter	Type	Direction	Description
`Channel`	`IChannel`	in	Channel to read.
`Data`	`Variant`	out	Array of `Single` values.
`Timestamps`	`Variant`	out	Array of `Double` timestamps.

IDataSections

The data sections of the loaded file — one per span between a start and a stop event, so a triggered measurement produces one section per trigger.

Get it from `ILoadEngine.DataSections`.

PYTHON

```python
sections = app.LoadEngine.DataSections
print(sections.Count)
for s in sections:
    print(s.Time, s.DataCount)
```

C#

```csharp
dynamic sections = app.LoadEngine.DataSections;
Console.WriteLine(sections.Count);
for (int i = 0; i < (int)sections.Count; i++)
{
    dynamic s = sections[i];
    Console.WriteLine($"{s.Time} {s.DataCount}");
}
```

Members

Count

Property — read-only — `Integer`

Number of sections in the loaded file.

Item

Property — read-only — `IDataSection`

The section at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard COM enumerator, which is what lets the collection be iterated directly rather than by index. You never call it yourself — the language does.

IDeviceNode

One device in the hardware tree, with its name, its sub-devices, and its synchronisation.

Its synchronisation is an `ISynchronization`.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IDewePlugin

A single generic message entry point.

Derives from `IUnknown` rather than `IDispatch`, so it is a vtable-only interface — usable from compiled code, not from a scripting language.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`OnMessage`	A message has arrived for the plug-in.

IDigitalTrigLevel

The hardware trigger thresholds of a digital or counter input — what voltage the input treats as a transition.

Every member is read-only: these report how the input is configured, they do not set it.

> **Not callable from an automation client**
>
> The two properties that return one — `TrigLevels` on `IDIChannel` and on `ICntChannel` — are on interfaces a client cannot use: the first is not available to late binding and the second is not implemented at all.
>
> From Python or C#, read the same thresholds through `IDaq.GetDITrgLevel` and `IDaq.GetCNTTrgLevel` instead.

Members

TrigType

Property — read-only — `Integer`

The input standard:

Value	Standard
`0`	TTL / CMOS
`1`	24 V (PLC)
`2`	inductive
`255`	custom levels

Coupling

Property — read-only — `Integer`

`0` DC coupled, `1` AC coupled.

TrigLevel

Property — read-only — `Integer`

The threshold a transition must cross, in **millivolts**.

ReTrigLevel

Property — read-only — `Integer`

The level the signal must return past before another transition is recognised, in **millivolts** — the hysteresis that stops a noisy signal producing repeated edges.

On an input whose levels are fixed rather than programmable, both level properties report the fixed value rather than anything you configured. Programmable inputs accept 0 to 40000 mV.

IDirectXBitmap

A DirectX-backed offscreen bitmap, exposing its texture data and a shareable handle.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IDiscreteItem

One named state of a discrete channel — a value, what to call it, and what colour to draw it.

Get it from `IDiscreteList`, by position, by value with `Find`, or from `Add`.

Members

Value

Property — read/write — `Integer`

The channel reading this state stands for.

Nothing stops two states carrying the same value. When they do, `Find` returns the first and the other is unreachable.

Caption

Property — read/write — `String`

The text shown in place of the number.

Color

Property — read/write — `Integer`

The colour used for this state, as a BGR integer — so `0x0000FF` is red and `0xFF0000` is blue, the reverse of the more familiar RGB order.

IDiscreteList

The named states of a channel whose numbers stand for conditions rather than measurements — `0` closed, `1` open, and so on.

Get it from `IChannel.DiscreteList`. Each state is an `IDiscreteItem`.

An empty list means the channel's values are shown as plain numbers.

PYTHON

```
dl = channel.DiscreteList
item = dl.Add()
item.Value = 1
item.Caption = "Open"
item.Color = 0x00FF00
```

C#

```
dynamic dl = channel.DiscreteList;
dynamic item = dl.Add();
item.Value = 1;
item.Caption = "Open";
item.Color = 0x00FF00;
```

Members

Count

Property — read-only — `Integer`

Number of named states.

Item

Property — read-only — `IDiscreteItem`

The state at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

Find

Method — returns `IDiscreteItem`

The state carrying a given value, or nothing when no state does.

Parameter	Type	Direction	Description
`Val`	`Integer`	in	The value to look for.

This is the one to use for turning a reading into a caption — the position in `Item` has nothing to do with the value a state stands for.

Add

Method — returns `IDiscreteItem`

Appends a state and returns it, ready to fill in. A new state starts with a value of `0`, so set `Value` before adding another.

Remove

Method

Deletes a state.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	Position in the list, not the state's value.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the states can be iterated directly in languages that support it.

IDisplayFrameTemplate

One frame in a display template — the widgets it holds and the channels they show.

Creates widgets and widget groups, assigns channels to them, sets widget properties, and carries the frame's name, caption and icon. It is also the unit that is written to and read back from a template's XML.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

IDisplayFrameTemplates

The frames belonging to an `IDisplayTemplate`.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

IDisplayTemplate

A saved screen layout — a design canvas size and the frames laid out on it.

The frames are `IDisplayFrameTemplates`.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

IDwXMLDocument

An XML document, with typed readers and writers for the values inside it.

Get one from `IXMLHelper.GetCustomIDwXMLDocument`. Its nodes are `IDwXMLNode`.

This is the document format DewesoftX stores setups in, so the same interface serves for reading a setup, writing one, and keeping your own settings alongside.

PYTHON

```python
doc = app.XMLHelper.GetCustomIDwXMLDocument(True)   # for writing
root = doc.CreateNode("MySettings", "")
doc.SetStartNode(root)
doc.WriteString(root, "Operator", "A. Smith", "")
doc.WriteInteger(root, "Runs", 12, 0)
```

C#

```csharp
dynamic doc = app.XMLHelper.GetCustomIDwXMLDocument(true);   // for writing
dynamic root = doc.CreateNode("MySettings", "");
doc.SetStartNode(root);
doc.WriteString(root, "Operator", "A. Smith", "");
doc.WriteInteger(root, "Runs", 12, 0);
```

Reading and writing values

Every value lives in a child node of the node you pass, named by `Name`. There is a `Read`, a `Write` and an `Update` method for each type, all with the same shape:

Method	Parameters
`Read…`	`Node`, `Name`, `Value` (in/out), `Default`
`Write…`	`Node`, `Name`, `Value`, `Default`
`Update…`	`Node`, `Name`, `Value` (in/out), `Default`

Parameter	Type	Direction	Description
`Node`	`IDwXMLNode`	in	The node to look under.
`Name`	`String`	in	The child node's name.
`Value`	*the method's type*	varies	The value read, or the value to write.
`Default`	*the method's type*	in	Used when the entry is missing, and compared against on writing.

The types available:

Type	Methods
`Integer`	`ReadInteger`, `WriteInteger`, `UpdateInteger`
`String`	`ReadString`, `WriteString`, `UpdateString`
`Boolean`	`ReadBoolean`, `WriteBoolean`, `UpdateBoolean`
`Byte`	`ReadByte`, `WriteByte`, `UpdateByte`
`Word`	`ReadWord`, `WriteWord`, `UpdateWord`
`SmallInt`	`ReadSmallInt`, `WriteSmallInt`, `UpdateSmallInt`
`LongWord`	`ReadLongWord`, `WriteLongWord`, `UpdateLongWord`
`Single`	`ReadSingle`, `WriteSingle`, `UpdateSingle`
`Double`	`ReadDouble`, `WriteDouble`, `UpdateDouble`
byte array	`ReadByteArray`, `WriteByteArray` — no `Update`, and no `Default`

Everything is held as text in the file; the typed methods only convert. So a value written as a `Double` reads back happily as a `String`.

What Default does

On reading, `Default` is what you get when the entry is not there — a missing node, a missing document, or text that will not convert all give you the default rather than an error. **There is no way to tell a missing entry from one that holds the default value**, so choose a default you can live with rather than a sentinel you plan to test for.

On writing, `Default` decides whether the entry is written at all — see `WriteDefault`.

Update and UpdateOperation

`Update…` is `Read…` or `Write…` depending on `UpdateOperation`. It exists so one piece of code can both save and load: write the sequence of `Update` calls once, then set the operation and run it in either direction.

PYTHON

```
def transfer(doc, node, settings):
    settings["Operator"] = doc.UpdateString(node, "Operator", settings["Operator"], "")
    settings["Runs"] = doc.UpdateInteger(node, "Runs", settings["Runs"], 0)

doc.UpdateOperation = 1     # write
transfer(doc, root, settings)
```

C#

```
void Transfer(dynamic doc, dynamic node, Settings s)
{
    string op = s.Operator; doc.UpdateString(node, "Operator", ref op, ""); s.Operator = op;
    int runs = s.Runs;      doc.UpdateInteger(node, "Runs", ref runs, 0);   s.Runs = runs;
}

doc.UpdateOperation = 1;    // write
Transfer(doc, root, settings);
```

`Value` is in/out on the `Read` and `Update` methods, so Python gets the result as a return value while C# must pass a variable by reference.

Nodes

CreateNode

Method — returns `IDwXMLNode`

Creates a node. It is not part of the document until you attach it with `SetStartNode` or `IDwXMLNode.AddChild`.

Parameter	Type	Direction	Description
`NodeName`	`String`	in	The node's name.
`NodeValue`	`String`	in	Its value. Pass empty for a node that only holds children.

GetStartNode

Method — returns `IDwXMLNode`

The document's root node.

SetStartNode

Method

Makes a node the document's root.

Parameter	Type	Direction	Description
`Node`	`IDwXMLNode`	in	The node to use as root.

Settings

WriteDefault

Property — read/write — `Boolean`

Whether values equal to their `Default` are written.

`False`, the initial setting, leaves them out — and **removes an existing entry** that holds the default. That is what keeps setup files down to the settings a user actually changed. Set it `True` to write every value whatever it is, which is what you want if something else will read the file and expects every entry present.

SearchForExistingNodes

Property — read/write — `Boolean`

How writing treats an entry that is already there.

`True`, the initial setting, looks for a child of that name and overwrites it. `False` appends without looking, which is quicker but leaves two nodes of the same name if one already existed — and reading finds only the first.

UpdateOperation

Property — read/write — `Integer`

Which direction the `Update…` methods go: `0` read, `1` write.

Set from the `Write` argument you passed to `GetCustomIDwXMLDocument`, so a document asked for as writable starts at `1`.

Behaviour worth knowing

A new document has **no root node** — `GetStartNode` returns nothing until you create one and set it.

A nil `Node` is accepted everywhere. Reading from one gives you the default; writing to one does nothing at all. Neither reports an error, so a mistake in building the node tree shows up as values that silently fail to persist.

IDwXMLNode

One node of an `IDwXMLDocument` — its name, its value, its children and its attributes.

Create nodes with `IDwXMLDocument.CreateNode`; reach existing ones from `GetStartNode` and the members below.

A node holds two separate lists — child nodes and attribute nodes — with a parallel set of members for each. Attributes are themselves nodes, so the same interface serves both.

For reading and writing *values* under a node, use the typed methods on the document rather than walking children by hand.

PYTHON

```python
root = doc.GetStartNode()
for i in range(root.GetChildCount()):
    child = root.GetChild(i)
    print(child.GetName(), "=", child.GetValue())
```

C#

```csharp
dynamic root = doc.GetStartNode();
for (int i = 0; i < (int)root.GetChildCount(); i++)
{
    dynamic child = root.GetChild(i);
    Console.WriteLine($"{child.GetName()} = {child.GetValue()}");
}
```

Name and value

GetName

Method — returns `String`

The node's name. There is no setter — a node is named when it is created.

GetValue

Method — returns `String`

The node's own value, as text. Empty for a node that only holds children.

SetValue

Method

Sets the node's value.

Parameter	Type	Direction	Description
`Value`	`String`	in	The new value.

Children

GetChildCount

Method — returns `Integer`

Number of child nodes.

GetChild

Method — returns `IDwXMLNode`

The child at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `GetChildCount - 1`.

FindChildNode

Method — returns `IDwXMLNode`

The first child with a given name, or nothing when there is none.

Parameter	Type	Direction	Description
`NodeName`	`String`	in	The name to look for.

Only the **first** match is returned. A document written with `SearchForExistingNodes` off can hold several children of one name, and the later ones are unreachable this way — walk `GetChild` if you need them all.

AddChild

Method

Appends a node as a child.

Parameter	Type	Direction	Description
`Node`	`IDwXMLNode`	in	The node to add.

RemoveChildNode

Method

Removes one child.

Parameter	Type	Direction	Description
`Node`	`IDwXMLNode`	in	The child to remove.

ClearChildNodes

Method

Removes every child.

Attributes

These mirror the child members exactly, for the node's attribute list.

GetAttrCount

Method — returns `Integer`

Number of attributes.

GetAttr

Method — returns `IDwXMLNode`

The attribute at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `GetAttrCount - 1`.

FindAttrNode

Method — returns `IDwXMLNode`

The attribute with a given name, or nothing when there is none.

Parameter	Type	Direction	Description
`AttrNodeName`	`String`	in	The name to look for.

AddAttrNode

Method

Adds an attribute.

Parameter	Type	Direction	Description
`AttrNode`	`IDwXMLNode`	in	The attribute node to add.

RemoveAttrNode

Method

Removes one attribute.

Parameter	Type	Direction	Description
`AttrNode`	`IDwXMLNode`	in	The attribute to remove.

ClearAttrNodes

Method

Removes every attribute.

Navigating and copying

GetNextSibling

Method — returns `IDwXMLNode`

The next node alongside this one under the same parent, or nothing when this is the last.

Clone

Method — returns `IDwXMLNode`

A copy of the node. The copy is not attached anywhere — add it with `AddChild` to put it in a document.

IEvent

One event recorded during a measurement — a start, a stop, a trigger, a note the operator typed, an alarm changing state.

Get events from `IEventList`.

PYTHON

```python
for ev in app.EventList:
    print(ev.TimeStamp, ev.Type_, ev.Data)
```

C#

```csharp
dynamic events = app.EventList;
for (int i = 0; i < (int)events.Count; i++)
{
    dynamic ev = events[i];
    Console.WriteLine($"{ev.TimeStamp} {ev.Type_} {ev.Data}");
}
```

Members

Type_

Property — read-only — `Integer`

What kind of event this is:

Value	Event	Value	Event
`1`	start	`20`	keyboard
`2`	stop	`21`	text note
`3`	trigger	`22`	voice
`11`	video start	`23`	picture
`12`	video stop	`24`	module

The trailing underscore is there because `Type` is a reserved word in the language the interface was defined in.

Only text (`21`) and keyboard (`20`) events can be *added* through `IStoreEngine.AddNewEvent`. The rest are generated by DewesoftX itself.

TimeStamp

Property — read-only — `Double`

When the event occurred, in seconds.

Data

Property — read-only — `Variant`

Extra information, depending on the type — the text of a note, or `True` / `False` for an alarm going on or off. Empty for events that carry nothing.

TrigInfo

Property — read-only — `ITrigInfo`

For a trigger event, what the trigger was set to when it fired. Empty for other event types.

PosMid

Property — read-only — `Integer`

The data block the event fell in.

PosDir

Property — read-only — `Integer`

The sample position within that block. As elsewhere, the absolute sample number is `PosMid * IData.Samples + PosDir`.

IEventList

The events recorded during the measurement, in the order they occurred.

Get it from `IApp.EventList`. Add to it with `IStoreEngine.AddNewEvent`.

PYTHON

```python
events = app.EventList
print(events.Count)
for ev in events:
    print(ev.TimeStamp, ev.Type_)
```

C#

```csharp
dynamic events = app.EventList;
Console.WriteLine(events.Count);
for (int i = 0; i < (int)events.Count; i++)
{
    dynamic ev = events[i];
    Console.WriteLine($"{ev.TimeStamp} {ev.Type_}");
}
```

Members

Count

Property — read-only — `Integer`

Number of events.

Item

Property — read-only — `IEvent`

The event at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard COM enumerator, which is what lets the list be iterated directly rather than by index. You never call it yourself — the language does.

IExportFrame

The setup frame a custom export shows for its own options.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`SetExpApp`	Receives the export application object.
`ShowFrame`	Put the frame on screen.
`HideFrame`	Take it off.
`Apply`	Commit what the user changed.
`UpdateXML`	Read the frame's settings from, or write them to, the setup.

IExpression

One metadata expression — its display name, its description, and the snippet that inserts it.

Not usable from an automation client

The only way in is `IMetaDataParser.ExpressionGroupList`, and neither that object nor this one is registered for late binding — so no member name on either resolves from Python or C#.

IExpressionGroup

One named group of metadata expressions.

Each expression is an `IExpression`.

> **Not usable from an automation client**
>
> The only way in is `IMetaDataParser.ExpressionGroupList`, and neither that object nor this one is registered for late binding — so no member name on either resolves from Python or C#.

IExpressionGroupList

The groups of metadata expressions available for building text out of live values.

Each group is an `IExpressionGroup`.

> **Not usable from an automation client**
>
> The only way in is `IMetaDataParser.ExpressionGroupList`, and neither that object nor this one is registered for late binding — so no member name on either resolves from Python or C#.

IFileNameSettings

How DewesoftX names the files it stores, and when it starts a new one.

Get it from `IApp.FileNameSettings`.

PYTHON

```
fn = app.FileNameSettings
fn.BaseFileName = "Run"
fn.UseDate = True
fn.UseTime = True
```

C#

```
dynamic fn = app.FileNameSettings;
fn.BaseFileName = "Run";
fn.UseDate = true;
fn.UseTime = true;
```

Automatic naming

AutoCreate

Property — read-only — `Boolean`

Whether file names are generated rather than given explicitly — `True` when any of `UseDate`, `UseTime` or `UseMultiFile` is set. The remaining properties in this section only matter when it is `True`.

> **Declared read/write, but writing raises**
>
> The type library declares a setter, and it raises `AutoCreate property is read only`. Turn automatic naming on by setting one of the three properties above instead — this one follows from them.

BaseFileName

Property — read/write — `String`

The prefix a generated name is built from.

UseDate

Property — read/write — `Boolean`

Include the date in the generated name.

UseTime

Property — read/write — `Boolean`

Include the time in the generated name.

UseMultiFile

Property — read/write — `Boolean`

Include a sequential index in the generated name, giving `BaseFileName_NNNN`. The counter starts at `MultiFileStartIndex` and skips names already on disk.

MultiFileStartIndex

Property — read/write — `Integer`

The number the sequential index starts from.

Starting a new file automatically

AutoFlipFile

Property — read/write — `Smallint`

What happens when the criterion below is met:

Value	Behaviour
`0`	nothing — keep storing to the same file
`1`	**stop storing**
`2`	close the file and start a new one

This is not a boolean. Value `1` stops the measurement rather than continuing into a new file, which is a different outcome entirely — set it deliberately.

Flipping files is useful to keep individual files small enough to transfer, and to make finished files available for analysis while the measurement continues — DewesoftX cannot open a file it is still writing.

AutoFlipSize

Property — read/write — `Single`

The threshold, in the unit given by `AutoFlipUnit`.

AutoFlipUnit

Property — read/write — `Integer`

What `AutoFlipSize` counts:

Value	Unit
`0`	megabytes
`1`	hours
`2`	minutes
`3`	seconds
`4`	trigger occurrences

Values of `5` and above select a custom condition contributed by a plug-in — see `AutoFlipUnits`.

AutoFlipAbsTime

Property — read/write — `Boolean`

For the time units, whether `AutoFlipSize` is relative or absolute. With `False` it counts from the start of storing; with `True` it is a clock time — `AutoFlipUnit` of hours and `AutoFlipSize` of `14` starts a new file every day at 14:00.

Only relevant for units `1`, `2` and `3`.

AutoFlipUnits

Method

The custom flip conditions contributed by one plug-in, as an array of the `AutoFlipUnit` values that select them. Plug-ins and math modules can register their own conditions for ending a file.

Parameter	Type	Direction	Description
`PluginGuid`	`String`	in	The plug-in's GUID, braced and upper case: `{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`.
`Units`	`Variant`	out	Array of `Integer` unit values.

Empty means unassigned, not an empty array

If no registered condition matches the GUID, `Units` is left **unassigned** rather than set to an empty array — test for that before indexing it.

The numbers are positions in a freshly collected list, so they are only valid until the plug-in or math configuration changes. Re-query after any setup change.

Writing a custom value to `AutoFlipUnit` and reading it back does not return the same number — the read and write paths number custom units differently. Keep the value you wrote rather than reading it back.

IGDIBitmap

A GDI-backed offscreen bitmap, exposing the device context to draw through.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IGHObject

One field of the data header — a caption, a value, and how the user edits it.

Get one from `IGlobalHeader`, either by index or from `Add`.

PYTHON

```python
f = app.GlobalHeader.Add()
f.Caption = "Test stand"
f.SetObjType(2)                       # drop-down
f.SubList = ["Bench A", "Bench B"]
```

C#

```csharp
dynamic f = app.GlobalHeader.Add();
f.Caption = "Test stand";
f.SetObjType(2);                      // drop-down
f.SubList = new object[] { "Bench A", "Bench B" };
```

What kind of field

ObjType

Property — read-only — `Integer`

How the field is presented and edited:

Value	Field
`0`	static text the user cannot change
`1`	single-line input box
`2`	drop-down selection

SetObjType

Method

Sets the field type. `ObjType` is read-only, so this method is its counterpart.

Parameter	Type	Direction	Description
`Value`	`Integer`	in	`0`, `1` or `2`.

Only three values work

Values `3` to `7` are **silently ignored** — the call succeeds and the field is unchanged. Only static text, input box and drop-down are implemented.

Setting the type bypasses the coercion that would force the value back to text, so switching a numeric field to static text or drop-down can leave it with a numeric `DataType`. Set `DataType` explicitly afterwards.

Caption and value

Caption

Property — read/write — `String`

The label shown next to the field.

For a static-text field the caption *is* the content, so writing it also overwrites the value.

Data

Property — read/write — `String`

The field's current value. Writing it takes effect immediately.

DefaultData

Property — read/write — `String`

The value the field falls back to when variables are reset, normally at the start of a measurement.

Writing `DefaultData` does not change the current value, and writing `Data` does not change the default. If the field is set to keep its last value, a reset restores that instead of the default.

Both are exchanged as strings and parsed according to `DataType`. A string that does not parse is **silently rejected** and the old value stays.

DataType

Property — read/write — `Integer`

The value type: `11` text, `7` double, `4` integer. A new field is text.

The setter only takes effect on an **input-box** field (`ObjType` `1`) and only for those three values; anything else is ignored without error. Static-text and drop-down fields are always text. Reading it returns `-1` if the field has no backing variable. Changing it remounts the field's channel.

SubList

Property — read/write — `Variant`

The choices for a drop-down field, as an array of strings. Writing the list also sets both the default and the current value to its first item.

Only meaningful for a drop-down (`ObjType` `2`). On any other type writing does nothing and reading returns empty — as it also does for a drop-down whose list is empty.

UniqueID

Property — read/write — `String`

The field's programmatic name, as opposed to its human-readable `Caption`. This is what identifies it in setups and formulas.

DewesoftX assigns one automatically when the field is created, of the form `VARIABLE0`, `VARIABLE1` and so on.

Any string is accepted, but a write is **silently ignored** if another field already uses that name. Read the property back to confirm a rename took effect. Names are matched case-insensitively.

IGenericBitmap

An offscreen bitmap a visual control draws into.

Exposes its dimensions, colour depth and raw scan lines, and is reference counted by hand — a caller that takes a reference must release it. The two backends extend it: `IGDIBitmap` and `IDirectXBitmap`.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IGlobalHeader

The data header — the named fields such as operator, test number or comment that a user fills in for a measurement. They are defined in the project setup, stored in the data file, and also available as channels.

Get it from `IApp.GlobalHeader`. Each field is an `IGHObject`.

PYTHON

```python
gh = app.GlobalHeader
for i in range(gh.Count):
    f = gh[i]
    print(f.Caption, "=", f.Data)
```

C#

```csharp
dynamic gh = app.GlobalHeader;
for (int i = 0; i < (int)gh.Count; i++)
{
    dynamic f = gh[i];
    Console.WriteLine($"{f.Caption} = {f.Data}");
}
```

Members

Count

Property — read-only — `Integer`

Number of fields.

Item

Property — read-only — `IGHObject`

The field at a position.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	Position, `0` to `Count - 1`.

Add

Method — returns `IGHObject`

Creates a field and appends it, returning the new object — which is also `Item[Count - 1]`.

The field is usable immediately but its defaults are rarely what you want: a single-line input box holding text, with a generated identifier and the caption `Channel <n>`. Set at least `Caption`, and `SubList` if you make it a selection field.

PYTHON

```python
f = app.GlobalHeader.Add()
f.Caption = "Operator"
f.Data = "unknown"
```

C#

```csharp
dynamic f = app.GlobalHeader.Add();
f.Caption = "Operator";
f.Data = "unknown";
```

Remove

Method

Deletes the field at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position to delete.

> **The collection re-packs**
>
> Every field after the removed one shifts down by one and `Count` drops, so cached indices and previously fetched `IGHObject` references go stale.
>
> When removing several fields, iterate **downwards** — or re-read indices after each removal. An out-of-range index raises.

_NewEnum

Property — read-only — `IUnknown`

The standard COM enumerator, which is what lets the collection be iterated directly rather than by index. You never call it yourself — the language does.

IImportChannel

A channel created by a custom import.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`DefaultRes`	The suggested number of decimal places for displaying values.

IImportGroup

A channel group created by a custom import.

Extends `IChannelGroup`.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`MountChannel`	Attach a channel to the group.

IIndexChanger

Renumbers a plug-in's channels when their order changes.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

IInputCh

A channel as seen from an input slot — the channel itself plus where to read it from.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IInputGroup

One channel selection — a named, identified group of channels that can be handed to mathematics or a widget as a single input.

Get it from `IInputGroups.Item`.

PYTHON

```python
g = app.Data.InputGroups.Item(0)
print(g.Name, g.Guid, g.IconName)
g.AddChannel(app.Data.FindChannel("AI 1"), "Signal")
```

C#

```csharp
dynamic g = app.Data.InputGroups.Item[0];
Console.WriteLine($"{g.Name} {g.Guid} {g.IconName}");
g.AddChannel(app.Data.FindChannel("AI 1"), "Signal");
```

> **The channels in a group cannot be listed**
>
> The object is also a channel list internally, but it is registered only for `IInputGroup` — so the channel-list members do not resolve by name and there is no way to enumerate or count what a group contains. You can add to a group and read its identity; you cannot read back its contents.

Members

Name

Property — read/write — `String`

The group's name, as shown wherever it is offered as an input.

Guid

Property — read/write — `String`

The group's identifier. This is what a plug-in or a piece of mathematics matches on to find the group it wants, so it is stable in a way the position is not.

Index

Property — read-only — `Variant`

The group's position in the channel tree. A variant rather than a number because the position is a path through the tree, not a single index.

IconName

Property — read/write — `String`

The name of the icon shown beside the group.

Properties

Property — read-only — `IProperties`

Named values carried alongside the group, for anything the group needs to remember that it has no field for.

AddChannel

Method

Adds a channel to the group under a connection name.

Parameter	Type	Direction	Description
`Ch`	`IChannel`	in	The channel to add.
`ConnectionName`	`String`	in	The name this channel takes within the group — the role it plays.

Raises if `Ch` is not a real channel. The connection name is how a consumer of the group tells one member from another, so give each a distinct one.

IInputGroups

The channel selections in the setup — named groups of channels that mathematics and widgets take as a unit.

Get it from `IData.InputGroups`. Each group is an `IInputGroup`.

There is no member here that creates one: groups are made by DewesoftX and by plug-ins, and an automation client reads what already exists. A setup with no channel selections reports `Count` `0`, which is the normal state of a fresh setup.

PYTHON

```python
groups = app.Data.InputGroups
for i in range(groups.Count):
    g = groups.Item(i)
    print(g.Name, g.Guid)
```

C#

```csharp
dynamic groups = app.Data.InputGroups;
for (int i = 0; i < (int)groups.Count; i++)
{
    dynamic g = groups.Item[i];
    Console.WriteLine($"{g.Name} {g.Guid}");
}
```

Members

Count

Property — read-only — `Integer`

Number of channel selections.

Item

Property — read-only — `IInputGroup`

The group at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the groups can be iterated directly in languages that support it.

IInputSlot

One input connection point on a visual control or mathematics module — the socket a channel is dropped into.

Carries the slot's name and description, its tag and group, whether it is assigned, optional or static, and the channel currently in it.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IInputSlots

The input slots of a visual control or mathematics module.

Adds, finds, removes and reorders slots, reports which are in use, and saves and restores the whole set.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

ILoadEngine

Loading and navigating stored data files in analysis mode.

Get it from `IApp.LoadEngine`. Open a file with `IApp.LoadFile`; this interface then reports on it and reads from it.

PYTHON

```python
app.LoadFile(r"C:\Data\Run_0001.dxd")
le = app.LoadEngine
if le.FileOpened:
    print(le.FileName, le.NumBlocks, le.Duration)
```

C#

```csharp
app.LoadFile(@"C:\Data\Run_0001.dxd");
dynamic le = app.LoadEngine;
if (le.FileOpened)
    Console.WriteLine($"{le.FileName} {le.NumBlocks} {le.Duration}");
```

The loaded file

FileOpened

Property — read-only — `Boolean`

Whether a file is currently open. **Check this before trusting** `FileName`.

FileName

Property — read-only — `String`

Full path of the file the engine is working with.

It is **not cleared when a file is closed**, so after `CloseFile` it still reports the last file loaded. Use `FileOpened` to tell whether anything is actually open.

For a multi-file set it returns the single file's path, not the combined label the interface displays.

Duration

Property — read-only — `Double`

Time span covered by the file, in seconds.

Returns `-1` when the duration cannot be determined — a triggered file with no usable reduced-data stream, or a file with no start/stop event pair. Handle `-1` rather than treating it as a length.

NumBlocks

Property — read-only — `Integer`

Number of data blocks in the file.

ReducedOnly

Property — read-only — `Boolean`

`True` when the file holds only reduced-rate data, so full-rate samples are not available from it.

DataSections

Property — read-only — `IDataSections`

The file's data sections — the spans between start and stop events.

CloseFile

Method

Closes the open file.

Reloading data

Reading a channel's samples means first loading the span you want into its buffers, then reading the buffers through `IChannel`.

Reload

Method

Loads a span of data into the channel buffers.

Parameter	Type	Direction	Description
`Start`	`T_RecordPosition`	in	First position to load.
`Stop`	`T_RecordPosition`	in	Last position to load.

ReloadBlock

Method

Loads one block.

Parameter	Type	Direction	Description
`Num`	`Integer`	in	Block number, `0` to `NumBlocks - 1`.

After this, a channel's `GetScaledData` returns exactly `IData.Samples` values.

ReloadEx

Method

Loads a span, optionally for one channel only and at a chosen buffer level — cheaper than `Reload` when you need one channel, or only reduced data.

Parameter	Type	Direction	Description
`StartBlock`	`Integer`	in	First block, counting from `0`.
`EndBlock`	`Integer`	in	Last block.
`Channel`	`IChannel`	in	One channel, or nothing for all channels.
`MinLevel`	`Integer`	in	`0` fills the direct buffer, `1` the first intermediate buffer, and so on.

StartDBLoad

Method

Begins a sequential sweep through the file in fixed-size blocks — the efficient way to process a whole recording without loading it all at once. Advance with `NextDBLoad`.

Parameter	Type	Direction	Description
`Start`	`T_RecordPosition`	in	Where to begin.
`Stop`	`T_RecordPosition`	in	Where to end.
`BlockSize`	`Integer`	in	Samples per step.

NextDBLoad

Method — returns `Boolean`

Advances the sweep to the next block. Returns `False` when there are no more.

PYTHON

```python
le.StartDBLoad(start, stop, 10000)
while le.NextDBLoad():
    values = channel.GetScaledData()
    ...
```

C#

```
le.StartDBLoad(start, stop, 10000);
while (le.NextDBLoad())
{
    dynamic values = channel.GetScaledData();
    // ...
}
```

ShrinkFile

Method

Writes the currently selected region out to a new DewesoftX file — the way to cut a long recording down to the part that matters.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Target file, including path.

Set the region first, with `IData.SelectDataRegion` or by writing `IData.StartStamp` and `IData.EndStamp`. Without that the exported span is whatever happens to be selected.

Coordinating access

EnterLoadSync

Method

Takes the lock that serialises reading from the loaded file. Everything that pulls samples out of a data file takes it — offline recalculation, display refresh, batch processing, sequencer steps — so holding it stops those running underneath you while you read a consistent set of values.

LeaveLoadSync

Method

Releases that lock.

> **Must balance, and holding it freezes the interface**
>
> One `Leave` per `Enter`. Nesting within the same thread is safe, but the count must return to zero — leaving it held blocks offline calculation and display refresh, so DewesoftX appears frozen. Releasing more times than acquired corrupts the lock.
>
> This is **a different lock** from `IData.StartDataSync`, which coordinates with the live acquisition loop instead. Holding one does not give you the other, and neither substitutes for the other.

Video

VideoLoadEngines

Property — read-only — `IVideoLoadEngines`

The video streams in the loaded file.

StartVideoCompress

Method

Starts compressing the file's video.

GetVideoCompressDone

Method — returns `Integer`

Compression progress as a percentage.

IsVideoCompressDone

Method — returns `Boolean`

`True` once compression has finished.

StopVideoCompress

Method — returns `Boolean`

Ends compression, returning `True` if it completed successfully. Call it after `IsVideoCompressDone` reports `True`.

ILockableCursor

One of the two locked cursors — a position that every recorder graph in the setup is held at.

Get it from `ILockableCursors.Item`.

Setting the position and locking are independent: a position can be set while the cursor is unlocked, and takes effect when you lock it.

PYTHON

```python
c = app.LockableCursors.Item(0)
print(c.Locked, c.Position)
c.Position = 2.0
c.Lock()
```

C#

```csharp
dynamic c = app.LockableCursors.Item[0];
Console.WriteLine($"{c.Locked} {c.Position}");
c.Position = 2.0;
c.Lock();
```

Members

Locked

Property — read-only — `Boolean`

`True` while the cursor is pinned. Change it with `Lock` and `Unlock`.

Position

Property — read/write — `Double`

Where the cursor sits, in seconds relative to the start of the data.

Writing this does not redraw anything on its own. Call `IScreen.UpdateLockableCursors` on the screens you want refreshed.

Lock

Method

Pins the cursor at its current `Position`.

Unlock

Method

Releases the cursor, letting the recorders move it again.

ILockableCursors

The two locked cursors shared across every recorder graph in the setup.

Get it from `IApp.LockableCursors`. Each cursor is an `ILockableCursor`.

Locking a cursor pins it to one position and holds every recorder on every screen at that position, so the same instant can be read across displays.

PYTHON

```python
cursors = app.LockableCursors
c = cursors.Item(0)
c.Position = 1.5
c.Lock()
app.Screens.Item(0).UpdateLockableCursors()
```

C#

```csharp
dynamic cursors = app.LockableCursors;
dynamic c = cursors.Item[0];
c.Position = 1.5;
c.Lock();
app.Screens.Item[0].UpdateLockableCursors();
```

Members

Count

Property — read-only — `Integer`

Always `2`. There are exactly two locked cursors and they cannot be added to or removed.

Item

Property — read-only — `ILockableCursor`

One of the two cursors.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	`0` for the first cursor, `1` for the second.

Any other index raises.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the pair can be iterated directly in languages that support it.

IMarkerInput

A draggable position inside a marker — where the marker sits on its axis.

Extends `IMarkerObject`, so it also has a value type, unit, name, colour and visibility. See `IProcessingMarker` for the whole model.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`ValueEx`	property, read/write	The position on the axis. Reading tells you where the marker sits; writing moves it. Units are the owner channel's raw axis units — seconds for a time axis, hertz for an FFT — not display-scaled.
`UsingChannelInput`	property, read/write	`True` when the position is driven by a channel rather than the user. **Setting it makes the marker undraggable.**
`InitialPositionSource`	property, read/write	Where the position comes from when the marker is first placed: `0` the click position (the default), `1` the recorder's yellow cursor. Only the recorder honours `1`.

Writing `ValueEx` does not by itself recalculate anything — the plug-in must notify DewesoftX that the input changed.

IMarkerMath

The mathematics module behind a marker — the bridge from the math side back to the marker it owns.

See `IProcessingMarker` for the whole model.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`ProcessingMarker`	property, read-only → `IProcessingMarker`	The marker this module owns. Created and destroyed with the module.
`ViewOnly`	property, read/write	`True` when the marker reports only its current value instead of keeping a history.

`ViewOnly` is not a cheap display toggle — changing it alters the module's channel shapes and forces a recalculation.

IMarkerObject

One value slot inside a marker — a single number with a value type, a unit, a colour and a visibility flag.

Two kinds extend it: `IMarkerInput`, a position you can drag, and `IMarkerOutput`, a computed result bound to an output channel. See `IProcessingMarker` for how markers fit together.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`ValueType`	property, read/write	Which quantity this slot holds — see the value types below. Defaults to a value on the amplitude axis.
`ValueUnit`	property, read/write	The unit shown beside the value, such as `Hz` or `%`.
`IsReadOnly`	property, read-only	`True` for an output, `False` for an input. A type test, not a permission — it cannot be changed.
`Name`	property, read/write	The slot's caption.
`Color`	property, read/write	The slot's colour.
`Visible`	property, read/write	Whether the slot is drawn.
`AddAxisType`	method	Adds a value type to the set of axes this slot relates to, which tells the widgets which axis to plot against. Additive only — there is no removal and the set cannot be read back.

Name, Color and Visible live on the draw groups

They are not object state. All three read from and write to the draw groups the object belongs to, so:

- Before the object has been added to a draw group, reading `Name` returns empty, `Color` returns none, and `Visible` returns `False` — which is indistinguishable from being deliberately hidden. Writing any of them does nothing.
- `Name` also does nothing until `AddGroupName` has been called for that group.
- `Color` has the same asymmetry as `GroupColor`: reading gives the effective colour, writing sets only the lowest-priority slot.

Value types

Value	Meaning
`0`	a position on the time axis — rejected for single-value channels
`1`	a value on the amplitude axis (the default)
`2`	a position on the array channel's first axis — frequency, for an FFT
`3`	a position on the second axis; only meaningful on a matrix channel
`4`	a position on the third axis; carried but not displayed anywhere
`5`	a point in the plane of both first and second axes; matrix channels only
`6`	a computed result — not a position, rendered as caption text

IMarkerObjectsList

A list of marker value slots — returned by `IProcessingMarker.MarkerInputs`, `MarkerOutputs`, and the per-draw-group equivalents.

See `IProcessingMarker` for the model.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`Count`	property, read-only	Number of slots.
`Items`	property, read/write → `IMarkerObject`	The slot at a position.
`Add`	method	Appends a slot.
`Insert`	method	Inserts at a position.
`Remove`	method → `Integer`	Removes a slot and returns the index it held, or `-1`.
`Delete`	method	Removes the entry at a position.
`Clear`	method	Empties the list.

Add slots with `ConnectInputEx` and `ConnectOutputEx` rather than through this list — they also link the slot back to its marker, which `Add` does not.

IMarkerOutput

A computed result inside a marker, bound to an output channel.

Extends `IMarkerObject`. See `IProcessingMarker` for the whole model.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`Channel`	property, read/write → `IChannel`	The output channel the result is written to. Binding it is what turns a marker result into a real channel that can be stored, displayed and consumed by other mathematics.
`CanUseScaling`	property, **write-only**	Declares that the value may be transformed by the display's decibel or logarithmic scaling. There is no getter. Only takes effect for amplitude and result value types.

An output whose channel is never bound, or which is never added to a draw group, produces no visible result — it is quietly collected into the marker's not-drawn bookkeeping.

IMarkerOwner

Obsolete. The channel a marker is attached to.

The owner is now derived from the marker's mathematics module — its first input connection. Read it from there instead. See `IProcessingMarker`.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member		Description
`Owner`	property, read/write → `IChannel`	The owner channel. The getter still works; **the setter is an empty body** and assigning it does nothing.
`ChannelType`	property, read/write	The owner channel's shape: `0` scalar, `1` vector, `2` matrix, `3` none. Stored and returned, but **nothing in DewesoftX ever reads it** — channel-shape checks use the marker type's own declaration instead.

IMasterClock

The acquisition clock — what time it is, by two different reckonings.

Get it from `IApp.MasterClock`.

Members

GetCurrentTime

Method — returns `Double`

The current time, as a date-and-time value.

GetAcqTime

Method — returns `Double`

The time within the acquisition, in seconds from its start.

These answer different questions. `GetAcqTime` is what channel timestamps are measured against, so it is the one to compare sample times to. `GetCurrentTime` is wall-clock time, for stamping a result with when it happened.

IMath

The mathematics setup — every calculation configured on the measurement.

Get it from `IApp.Math`. Each calculation is an `IMathObject`.

> **The same object also serves `IPowerModules`**
>
> `IApp.PowerModules` returns this very object, not a different one, so members of `IPowerModules` resolve on it too — `app.Math.Item(0)` works, as does `app.PowerModules.MathObject(0)`. They are two views of one thing. Use whichever reads better and do not expect the counts to agree: `Count` means all mathematics on `IMath` and only power modules on `IPowerModules`.

PYTHON

```python
m = app.Math
print(m.Count, "calculation(s)")
for i in range(m.Count):
    obj = m.MathObject(i)
    print(obj.Name, obj.MathType)
```

C#

```csharp
dynamic m = app.Math;
Console.WriteLine($"{m.Count} calculation(s)");
for (int i = 0; i < (int)m.Count; i++)
{
    dynamic obj = m.MathObject[i];
    Console.WriteLine($"{obj.Name} {obj.MathType}");
}
```

Members

Count

Property — read-only — `Integer`

Number of calculations configured.

MathObject

Property — read-only — `IMathObject`

The calculation at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

FindObjByID

Method — returns `IMathObject`

The calculation with a given id, rather than position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	The calculation's `Id`.

AddObj

Method — returns `IMathObject`

Adds a calculation of a given type and returns it, ready to configure.

Parameter	Type	Direction	Description
`Guid`	`String`	in	The mathematics type's GUID, braces included.

The GUID identifies which kind of mathematics to create — an FFT, a filter, a statistics module. They are the ids the mathematics modules register themselves under; read one off an existing calculation with `MathGUID`.

PYTHON

```python
fft = app.Math.AddObj("{0FB43E15-DA25-4278-85D4-94A8AF6B8E62}")
print(fft.Name, fft.MathType)     # 'FFT analysis 1' 'FFT_Application'
```

C#

```csharp
dynamic fft = app.Math.AddObj("{0FB43E15-DA25-4278-85D4-94A8AF6B8E62}");
Console.WriteLine($"{fft.Name} {fft.MathType}");
```

RemoveObj

Method

Deletes a calculation.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the calculations can be iterated directly in languages that support it.

IMathChannel

The display defaults a calculation gives the channels it creates.

Math channels carry this interface as well as `IChannel`, so one object serves both — read these members on any channel a calculation produced.

These are the values the calculation *suggests* when the channel first appears. A user's later change to the channel or its display overrides them.

PYTHON

```python
out = app.Math.MathObject(0).MathModule(0).OutputChannels.Item(0)
print(out.DefaultName, out.DefaultMin, out.DefaultMax, out.DefaultRes)
```

C#

```csharp
dynamic outCh = app.Math.MathObject[0].MathModule[0].OutputChannels.Item[0];
Console.WriteLine($"{outCh.DefaultName} {outCh.DefaultMin} {outCh.DefaultMax} {outCh.DefaultRes}");
```

Members

DefaultName

Property — read/write — `String`

The name the calculation gives the channel.

DefaultMin

Property — read/write — `Double`

The lower end of the channel's suggested display scale.

DefaultMax

Property — read/write — `Double`

The upper end of the suggested display scale.

DefaultRes

Property — read/write — `Integer`

The suggested number of decimal places for displaying values.

MathObject

Property — read-only — `IMathObject`

The calculation that created this channel — the way back from an output channel to the mathematics behind it.

IMathContext

A custom mathematics module's handle on DewesoftX.

Through it a module reads its input and output channels, mounts and unmounts channels and input groups, sets default channel properties, reports errors, applies its configuration, walks its sub-items, marks itself offline and reports offline progress, and drives control-channel calibration.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IMathFrameContext

A custom mathematics setup frame's handle on the module it is editing.

Adds and removes modules, adds sub-calculations, reports the module currently being edited, and applies the edits.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

IMathItem

What every piece of mathematics has in common: the channels going in, the channels coming out, and whether it is the last step of its calculation.

You do not obtain an `IMathItem` on its own. Both `IMathModule` and `IMathObjContext` extend it, so these three members are available on either.

Members

InputChannels

Property — read-only — `IChannelList`

The channels feeding the calculation.

OutputChannels

Property — read-only — `IChannelList`

The channels it produces. These are ordinary channels — storable, displayable, and usable as the input of further mathematics.

LastStep

Property — read-only — `Boolean`

`True` when this is the final step of its calculation, so its outputs are the calculation's results rather than an intermediate stage.

IMathItem2

A second, later interface on mathematics items: a generic command entry point and access to the underlying implementation object.

`IMathModule` and `IMathObjContext` both carry it, but they register only `IMathItem`, so `IOControl` and `GetImplObject` cannot be reached by name. The `IMathItem` members on those objects work normally.

> **Not usable from an automation client**
>
> Objects carrying this interface are not registered for it, so **its member names do not resolve** from Python or C# even on an object you already hold.

IMathModule

One module of a calculation — the calculation applied to one input.

Get it from `IMathObject.MathModule` or `IMathObject.FindModuleByID`.

Extends `IMathItem`, so `InputChannels`, `OutputChannels` and `LastStep` are available here too — and those are what you will use most.

Members

Id

Property — read-only — `Integer`

The module's id, as taken by `IMathObject.FindModuleByID`.

MathObject

Property — read-only — `IMathObject`

The calculation this module belongs to.

IMathObjContext

A calculation's context object.

Get it from `IMathObject.MathObjContext`.

Extends `IMathItem`, so `InputChannels`, `OutputChannels` and `LastStep` are available here as well — which is the only reason to hold one from an automation client, since its own single member just points back where you came from.

MathObject

Property — read-only — `IMathObject`

The calculation this is the context for.

IMathObject

One calculation in the mathematics setup — an FFT, a filter, a statistics block.

Get it from `IMath.MathObject`, `IMath.FindObjByID` or `IMath.AddObj`.

A calculation is made of one or more modules, one per input it was applied to: an FFT across four channels is one `IMathObject` holding four `IMathModule`s. Some kinds also hold *sub-calculations* — a nested list of further `IMathObject`s — which is how compound analyses are built.

PYTHON

```python
obj = app.Math.AddObj("{0FB43E15-DA25-4278-85D4-94A8AF6B8E62}")
obj.Name = "Spectrum"
print(obj.MathType, obj.Count, "module(s)")
for i in range(obj.Count):
    mod = obj.MathModule(i)
    print(" ", mod.Id, mod.OutputChannels.Count, "output(s)")
```

C#

```csharp
dynamic obj = app.Math.AddObj("{0FB43E15-DA25-4278-85D4-94A8AF6B8E62}");
obj.Name = "Spectrum";
Console.WriteLine($"{obj.MathType} {obj.Count} module(s)");
for (int i = 0; i < (int)obj.Count; i++)
{
    dynamic mod = obj.MathModule[i];
    Console.WriteLine($"  {mod.Id} {mod.OutputChannels.Count} output(s)");
}
```

Identity

Name

Property — read/write — `String`

The calculation's name, as it appears in the setup. A new one is named after its type with a number appended — `FFT analysis 1`.

MathType

Property — read-only — `String`

The kind of mathematics, as a short identifier — `FFT_Application` for an FFT. Fixed by the type; use it to recognise what a calculation is.

MathGUID

Property — read-only — `String`

The GUID of the mathematics *type*, the same value you would pass to `IMath.AddObj` to create another of the same kind. Shared by every calculation of this type.

InstanceGUID

Property — read-only — `String`

The GUID of *this particular* calculation, unique to it. This is the one to keep if you need to find the same calculation again later — `Id` and the position both shift as calculations are added and removed.

Id

Property — read-only — `Integer`

The calculation's numeric id, as taken by `IMath.FindObjByID`.

Modules

Count

Property — read-only — `Integer`

Number of modules — one per input the calculation was applied to.

MathModule

Property — read-only — `IMathModule`

The module at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

FindModuleByID

Method — returns `IMathModule`

The module with a given id, rather than position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	The module's `Id`.

AddModules_SingleInput

Method

Adds one module per channel in a list, each taking a single input — the usual way to apply a calculation across many channels at once.

Parameter	Type	Direction	Description
`InputChannelList`	`IChannelListEx`	in	The channels to apply it to.

MathObjContext

Property — read-only — `IMathObjContext`

The calculation's context object. Its only member points back here, so it is of little use from an automation client.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the modules can be iterated directly in languages that support it.

Sub-calculations

A calculation of the right kind can hold a nested list of further calculations. Ordinary mathematics has none and `SubMathCount` reads `0`.

SubMathCount

Property — read-only — `Integer`

Number of sub-calculations.

SubMath

Property — read-only — `IMathObject`

The sub-calculation at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `SubMathCount - 1`.

SubMathWithGuid

Property — read-only — `IMathObject`

The sub-calculation with a given instance GUID — stable across additions and removals, unlike the position.

Parameter	Type	Direction	Description
`InstanceGuid`	`String`	in	The sub-calculation's `InstanceGUID`.

AddToSubList

Method — returns `IMathObject`

Adds a sub-calculation of a given type.

Parameter	Type	Direction	Description
`Guid`	`String`	in	The mathematics type's GUID.

DeleteSubListItem

Method

Deletes a sub-calculation by position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `SubMathCount - 1`.

DeleteSubListItemWithGuid

Method

Deletes a sub-calculation by instance GUID. Preferable to the positional form when you are removing several, since positions shift as you go.

Parameter	Type	Direction	Description
`InstanceGuid`	`String`	in	The sub-calculation's `InstanceGUID`.

DeleteSubListItems

Method

Deletes every sub-calculation.

State

Locked

Property — read/write — `Boolean`

`True` to stop the calculation being changed.

Offline

Property — read-only — `Byte`

Whether the calculation runs offline — on stored data — rather than live. `0` means it does not.

MarkAsOffline

Method

Marks the calculation as offline, so it is recalculated from stored data instead of running during acquisition. This is the setter for `Offline`, which is otherwise read-only, and there is no member that reverses it.

IMeasUnit

One measurement unit in a distributed setup — a remote DewesoftX the local one talks to over the network.

Get it from `IRemoteManager.Item`.

PYTHON

```python
mu = app.RemoteManager.Item(0)
print("in use:", mu.Used)
cmd_out, data_out = mu.SendCustomCommand("GETSTATE", "")
```

C#

```csharp
dynamic mu = app.RemoteManager.Item[0];
Console.WriteLine($"in use: {mu.Used}");
string cmdOut, dataOut;
mu.SendCustomCommand("GETSTATE", "", out cmdOut, out dataOut);
```

Members

Used

Property — read/write — `Boolean`

Whether this unit takes part in the measurement.

SendCustomCommand

Method

Sends a command over the network connection and waits for the reply.

Parameter	Type	Direction	Description
`CmdIn`	`String`	in	The command to send.
`DataIn`	`String`	in	Its payload.
`CmdOut`	`String`	out	The reply's command.
`DataOut`	`String`	out	The reply's payload.

This blocks until the remote answers or the connection times out. Both out parameters come back as return values in Python and must be declared in C#.

EnterSync

Method

Takes the lock on the connection, so a sequence of commands to this unit is not interleaved with anyone else's.

LeaveSync

Method

Releases the lock.

> **Every EnterSync needs a LeaveSync**
>
> The lock is held until released. Miss the release — because an exception escaped in between, say — and every later call touching this unit blocks, including `SendCustomCommand`, which takes the same lock. Release it in a `finally`.

IMetaDataParser

Expands metadata expressions — the placeholders a setup uses to build text out of live values, such as a filename or a report caption.

Parses text containing expressions, evaluates a single expression, and lists the expression groups available.

> **Not usable from an automation client**
>
> The object is reachable — `IData.MetaDataParser` resolves — but it is not available to late binding, so **none of its member names resolve** from Python or C#.

IModule

One hardware module — detecting it, filling it in, and reading what it reports.

Its per-channel data is `IDaqData` and its per-pad data `IPadData`, both of which a module plug-in implements.

No way to obtain one

The only accessor that returns this — `IApp.Modules` — is a stub that always returns nothing. There is no other route to it, so nothing here is reachable from any client.

Members

Member	Description
`Index`	The module's position.
`ModuleType`	What kind of module it is.
`GetSerialNumber`	Its serial number.
`DetectModule`	Probe for the module.
`FillModule`	Populate it from the hardware.
`SetModule`	Write settings to it.
`ClearModule`	Reset it.
`SetDaq`	Attach the acquisition device.
`SetDaqAddress`	Set its address on that device.
`SetPad`	Set one of its pads.
`GetDataPad`	Read one pad's data.
`DaqData`	Its channel data, as `IDaqData`.
`PadData`	Its pad data, as `IPadData`.
`FREQAFindTriggerLevel`	Work out a trigger level for a frequency input.

IModules

The hardware modules DewesoftX has found.

> **No way to obtain one**
>
> The only accessor that returns this — `IApp.Modules` — is a stub that always returns nothing. There is no other route to it, so nothing here is reachable from any client.

Members

Member	Description
`Count`	Number of modules.
`Item`	The module at a position, as an `IModule`.
`_NewEnum`	The standard enumeration hook.

IMultiDevice

A multi-device controller — several acquisition devices presented as one.

Get it from `IApp.GetMultiDevice(Index)`, which is a method rather than a property.

Where `IDaq` describes the acquisition hardware as configured, this describes one controller within a multi-device system.

PYTHON

```python
md = app.GetMultiDevice(0)
print(md.GetCardCount(), "device(s)")
ok, info = md.GetDeviceInfo(0)
if ok:
    print(info.Name, info.SerialNumber)
```

C#

```csharp
dynamic md = app.GetMultiDevice(0);
Console.WriteLine($"{md.GetCardCount()} device(s)");
object info;
if (md.GetDeviceInfo(0, out info))
    Console.WriteLine($"{((dynamic)info).Name}");
```

Members

GetCardCount

Method — returns `Integer`

How many devices this controller has.

GetDeviceInfo

Method — returns `Boolean`

Details of one device. `True` when the information was filled in.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Device position, `0` to `GetCardCount - 1`.
`OutDevice`	`DaqDeviceInfo`	out	The device details.

The structure is the same one `IDaq.GetDeviceInfo` returns, and its fields are listed there. Note the difference in shape: this one reports success separately and hands the structure back as an out parameter, rather than returning it.

INTPClient

The NTP time client — whether it is running and which server it is using.

Get it from `IApp.NTPClient`.

Read-only apart from resolving a name: this reports the client's state, it does not configure it.

PYTHON

```python
ntp = app.NTPClient
print(ntp.Started, ntp.CurrentHost)
print(ntp.ResolveHost("pool.ntp.org"))
```

C#

```csharp
dynamic ntp = app.NTPClient;
Console.WriteLine($"{ntp.Started} {ntp.CurrentHost}");
Console.WriteLine(ntp.ResolveHost("pool.ntp.org"));
```

Members

Started

Property — read-only — `Boolean`

`True` while the NTP client is running.

CurrentHost

Property — read-only — `String`

The server in use, or empty when the client is not running.

ResolveHost

Method — returns `String`

Resolves a host name to an address.

Parameter	Type	Direction	Description
`AHost`	`String`	in	The name to resolve.

This goes out to the network, so it can take as long as a DNS lookup takes and returns an empty string rather than raising when the name will not resolve.

INothing

A placeholder interface with no members.

A placeholder, not an interface to call

This has no members. It exists so a class with nothing to expose can still name an interface where the type library requires one. There is nothing to obtain and nothing to call.

IOfflineCalc

Recalculates mathematics over stored data.

Get it from `IApp.OfflineCalc`.

The usual sequence is to load a data file, then `Calculate`, then `StoreCalculatedChannels` if the results are to be kept in the file rather than just read out.

PYTHON

```
app.LoadFile(r"C:\data\run1.dxd")
oc = app.OfflineCalc
oc.Calculate()
oc.StoreCalculatedChannels()
```

C#

```
app.LoadFile(@"C:\data\run1.dxd");
dynamic oc = app.OfflineCalc;
oc.Calculate();
oc.StoreCalculatedChannels();
```

Members

Calculate

Method

Runs the mathematics over the loaded data.

This returns nothing and reports nothing. It is also not instant on a large file — the call blocks until the recalculation finishes, so allow for that rather than treating it as a trigger.

StoreCalculatedChannels

Method

Writes the calculated channels into the data file, so they are present next time it is loaded without recalculating.

This **closes the loaded file** as part of storing. Read anything you still want from the calculated channels before calling it, or load the file again afterwards.

IPadData

What a hardware module reports about one pad — a sensor slot on a module.

Covers the pad's identity and address, its amplitude and offset, the ranges it offers, and its configuration and speed codes.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IPairedValueMarkerInput

Obsolete and non-functional. Every member asserts and returns nothing usable.

This was the input of a marker occupying a set of coordinate pairs — a polyline or a region. That role is now served by one `IMarkerInput` per scalar position: a region is two inputs, a radial marker three, an XY region four.

Members

Member		Description
`GetValues`	method	Asserts. Returns an uninitialised value.
`SetValues`	method	Asserts. Does nothing.
`NumberOfPairs`	property, read/write	Asserts on both read and write.

IPermission

One node of the permission tree — what a user may do with a part of the setup.

A permission has a unique id, a name, a type — `0` modify, `1` read-only, `2` hidden — and child permissions of its own, so the tree mirrors the structure it protects. The only member that returns one is its own `GetOrCreatePermission`, which is why there is no way in from outside.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IPlugin

The original plug-in interface.

DewesoftX calls these as a measurement proceeds. `IPlugin2` supersedes it and is what current plug-ins implement — it re-declares the same lifecycle callbacks and adds setup and frame handling, so the two are alternatives rather than layers.

Not callable from an automation client

DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`SetApp`	Receives `IApp` — the plug-in's way in to everything else.
`Initiate`	Called once to set the plug-in up.
`Configure`	Opens the plug-in's own configuration.
`SetData`	Receives the data object.
`OnStartAcq`	Acquisition has started.
`OnStopAcq`	Acquisition has stopped.
`OnStartStoring`	Storing has started.
`OnStopStoring`	Storing has stopped.
`OnGetData`	New data is available — where a plug-in does its work.
`OnTrigger`	A trigger has fired.

IPlugin2

The plug-in interface current plug-ins implement.

Covers what `IPlugin` offered and adds setup persistence, a setup frame, and an identity. `IPlugin3` builds on this one with finer lifecycle hooks.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`Id`	The plug-in's identifier.
`Initiate`	Called once to set the plug-in up.
`Configure`	Opens the plug-in's own configuration.
`Used`	Read/write — whether the plug-in takes part in the measurement.
`NewSetup`	Start from a blank setup.
`LoadSetup`	Read the plug-in's settings back.
`SaveSetup`	Write the plug-in's settings out.
`ClearChannelsInstance`	Drop the channels the plug-in created.
`ShowFrame`	Put the plug-in's setup frame on screen.
`HideFrame`	Take it off screen.
`UpdateFrame`	Refresh it.
`OnStartAcq`	Acquisition has started.
`OnStopAcq`	Acquisition has stopped.
`OnStartStoring`	Storing has started.
`OnStopStoring`	Storing has stopped.
`OnGetData`	New data is available.
`OnTrigger`	A trigger has fired.
`OnOleMsg`	A message has arrived over automation.

IPlugin3

The extended plug-in interface — the finer-grained hooks.

Extends `IPlugin2`, so a plug-in implementing this one has all of that interface's members as well. What it adds is timing: hooks either side of starting and stopping, setup entry and exit, frame painting and resizing, and the ability to provide the clock.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`OnStartSetup`	The user has entered setup.
`OnStopSetup`	The user has left setup.
`OnBeforeStartAcq`	Before acquisition starts.
`OnAfterStartAcq`	After acquisition has started.
`OnBeforeStopAcq`	Before acquisition stops.
`OnAfterStopAcq`	After acquisition has stopped.
`OnTriggerStop`	A stop trigger has fired.
`OnAfterCalcMath`	Mathematics has finished for this block, so results are available.
`OnAlarm`	An alarm has changed state.
`OnRepaintFrame`	The setup frame needs repainting.
`OnResizeFrame`	The setup frame has been resized.
`OnShowHWFrame`	Show the extension's page in **Settings** → **Extensions**.
`OnHideHWFrame`	Hide it.
`OnGetSetupData`	Supply data for the setup display.
`ProvidesClock`	Whether the plug-in supplies the acquisition clock.
`OnGetClock`	Supply the clock, when it does.
`SetCANPort`	Receive the CAN port to use.
`OnBigListLoad`	A large channel list is being loaded.
`OnExit`	DewesoftX is closing.
`GetDWTypeLibVersion`	Report which type library version the plug-in was built against.

IPlugin4

A single generic event entry point, for plug-ins that would rather switch on an event code than implement a method per hook.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`OnEvent`	An event has occurred; the code says which.

IPluginChannel

A channel created by a plug-in, and the settings only its creator sets.

These are the parts of a channel that belong to whoever made it — its display defaults, its buffer, and how its values are produced.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

Members

Member	Description
`PluginGUID`	The plug-in that owns the channel.
`LongName`	The channel's full name.
`SetChNo`	Set the channel's number.
`SetIndex`	Set its position in the channel tree.
`SetSettings`	Set its settings from a string.
`DefaultMin`	Suggested lower end of the display scale.
`DefaultMax`	Suggested upper end.
`DefaultRes`	Suggested number of decimal places.
`SetCalcSingleValue`	Mark the channel as producing one value per block rather than a stream.
`SetCanOverload`	Declare that the channel can report overload.
`AsyncBufSize`	Size of the asynchronous buffer.
`ReserveMemory`	Reserve the channel's buffer.
`FreeMemory`	Release it.
`AlwaysReserveMemoryInSetup`	Keep the buffer reserved while in setup, not only while acquiring.
`MarkAsOffline`	Mark the channel as calculated offline.

IPluginChannelXMLHelper

Walks the channel definitions in a plug-in's saved setup.

The pattern is to call `StartExtractChannels`, then `ExtractNextChannel` repeatedly.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

Members

Member	Description
`FindNode`	Locate a node in the setup XML.
`StartExtractChannels`	Begin walking the channel definitions.
`ExtractNextChannel`	Take the next one.
`MountAllChannels`	Mount every channel the setup defines, in one call.

IPluginGroup

A plug-in's channel group, and the mounting of channels into it.

Extends `IChannelGroup`. *Mounting* is the act of attaching a channel or an input group to the plug-in so it receives data for it.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

Members

Member	Description
`MountChannel`	Attach a channel.
`MountChannelEx`	Attach a channel, with more control.
`UnmountChannel`	Detach one channel.
`UnmountChannels`	Detach all of them.
`MountInputGroup`	Attach a whole input group.
`FindChannel`	Look a channel up in the group.
`FindInputGroup`	Look an input group up.
`ClearAllChannels`	Remove every channel from the group.
`AddIndexName`	Name a level of the channel tree.
`AddIndexNameEx`	The same, with more control.

IPluginLicense

How a plug-in reports what licence it needs, and accepts one.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`GetHardwareCode`	The hardware code the licence is tied to.
`GetTrustedCode`	The plug-in's trusted code.
`GetRegTypeWanted`	Which kind of registration the plug-in is asking for.
`SetLicenseCode`	Receives the licence code.

IPluginLicense2

Reports a plug-in's licence code back to DewesoftX.

Extends `IPluginLicense`, so its four members are present too.

> **Not callable from an automation client**
>
> DewesoftX calls this interface on code that implements it. Nothing on the automation surface returns one.

Members

Member	Description
`GetLicenseCode`	The plug-in's current licence code.

IPowerModule

One power analysis module — its configuration as XML, and the spectra it produces.

Get it from `IPowerModules.Item` or `IPowerModules.Add`.

A power module has far more settings than a COM interface would sensibly expose one by one, so the whole configuration moves as XML through `SaveToXML` and `LoadFromXML`. The remaining members read out calculated results.

PYTHON

```python
pm = app.PowerModules.Item(0)
xml = pm.SaveToXML(1)                  # 1 = as a string
pm.LoadFromXML(1, xml)                 # ...and put it back
print(pm.FFTSampleRate, pm.FFTBlockSize)
```

C#

```csharp
dynamic pm = app.PowerModules.Item[0];
object xml = null;
pm.SaveToXML(1, out xml);              // 1 = as a string
pm.LoadFromXML(1, xml);
Console.WriteLine($"{pm.FFTSampleRate} {pm.FFTBlockSize}");
```

Configuration

SaveToXML

Method

Writes the module's whole configuration out as XML.

Parameter	Type	Direction	Description
`AType`	`Integer`	in	Where the XML goes — see below.
`XML`	`Variant`	in/out	A filename when `AType` is `0`; otherwise receives the XML.

`AType`	Meaning
`0`	write to a file, named by `XML`
`1`	return the XML as a string in `XML`
`2`	return an MSXML document object in `XML`

LoadFromXML

Method

Reads a configuration back in.

Parameter	Type	Direction	Description
`AType`	`Integer`	in	Where the XML comes from — the same three values.
`XML`	`Variant`	in	A filename, a string, or an MSXML document.

SaveToXML1

Method

The same as `SaveToXML` with the value strictly out rather than in/out. Use this one when your language will not pass a variant both ways.

Parameter	Type	Direction	Description
`AType`	`Integer`	in	As above.
`XML`	`Variant`	out	Receives the XML.

LoadFromXML1

Method

The counterpart of `SaveToXML1`.

Parameter	Type	Direction	Description
`AType`	`Integer`	in	As above.
`XML`	`Variant`	in	The XML.

Results

ModuleIndex

Property — read-only — `Integer`

The module's position among the power modules.

FFTSampleRate

Property — read-only — `Single`

The sample rate the module's FFT runs at, in samples per second.

FFTBlockSize

Property — read-only — `Integer`

The number of samples in one FFT block.

GetFFT

Method

Reads one of the module's spectra.

Parameter	Type	Direction	Description
`ValueType`	`Integer`	in	Which quantity's spectrum — voltage, current, power.
`Phase`	`Integer`	in	Which phase.
`Data`	`Variant`	out	The spectrum, as an array.

GetVectorScopeData

Method — returns `Variant`

The vector scope data — the phasor magnitudes and angles the module has calculated, as an array.

IPowerModules

The power analysis modules configured on the measurement.

Get it from `IApp.PowerModules`. Each module is an `IPowerModule`.

> **The same object also serves `IMath`**
>
> This is the very object `IApp.Math` returns, so `IMath`'s members resolve on it as well. `Count` and `Item` here describe the power modules; `IMath`'s `Count` and `MathObject` describe all mathematics. The two counts are unrelated — do not index one with the other.

PYTHON

```
pm = app.PowerModules
for i in range(pm.Count):
    print(pm.Item(i).ModuleIndex)
```

C#

```
dynamic pm = app.PowerModules;
for (int i = 0; i < (int)pm.Count; i++)
    Console.WriteLine(pm.Item[i].ModuleIndex);
```

Members

Count

Property — read-only — `Integer`

Number of power modules.

Item

Property — read-only — `IPowerModule`

The module at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

Add

Method — returns `IPowerModule`

Adds a power module and returns it.

Remove

Method

Deletes a power module.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the modules can be iterated directly in languages that support it.

IProcessingMarker

A marker — a user-placed annotation on a display that *computes something*.

> **Not callable from an automation client**
>
> Marker objects are not available to late binding, so **no member name on one resolves** from Python or C#. This includes `app.Data.ProcessingMarkers`, which is reachable from the automation surface but yields nothing usable.

What a marker is

You drop a marker on a recorder trace, a 2-D graph or a 3-D graph. It appears as a cursor line, a shaded band, a point, or a pair of points joined by a line, with a caption showing its position and one or more computed values. The same markers appear as rows in the marker table.

What distinguishes a DewesoftX marker from a drawn annotation is that **a marker is a mathematics module**. Placing one creates a real entry in the math setup, and that module produces genuine output channels which are stored, exported and usable by other mathematics. Dragging the marker changes the module's inputs, the module recalculates, and its output channels update. So a damping marker on an FFT is not a picture — it is a live calculation whose position you set with the mouse.

Marker *types* are not built in. Each is a registered mathematics extension identified by a GUID, which declares which axes it understands and which channel shapes it accepts.

How the pieces fit

```
IChannel                    the channel the marker sits on ("the owner")
  ▲
IMarkerMath                 the mathematics module, one per marker
  └── IProcessingMarker     the marker itself
        ├── MarkerInputs    draggable positions   (IMarkerInput)
        ├── MarkerOutputs   computed results      (IMarkerOutput)
        │     └── each bound to an IChannel
        └── draw groups[0..n]  how it is drawn — no interface, addressed by index
```

An `IMarkerObject` is **one value slot** — a single number with a type, unit, colour and visibility. `IMarkerInput` is a position you can drag; `IMarkerOutput` is a computed result bound to an output channel. A marker with two cursors and three results is one `IProcessingMarker` holding five marker objects.

A **draw group** is the unit of drawing: a style, a colour, a name, and the subset of the marker's inputs and outputs that take part. One marker can have several — that is how a peak marker draws a cursor, a shaded search region and a point on the curve as one logical thing. Draw groups have no interface of their own and are addressed by integer index through the `Group…` members below.

The owner channel is derived from the mathematics module's first input connection. `IMarkerOwner` is the obsolete way to read it.

Building a marker

A plug-in creates one through its context object, then builds its shape when DewesoftX calls back:

Create the marker from the marker type's GUID.

Add one input per draggable position and one output per result, with `ConnectInputEx` and `ConnectOutputEx`.

Declare capabilities with `SetMarkerProps`.

Build the drawing: `ClearGroups`, then per group `CreateGroup`, `AddGroupStyle`, `AddGroupName`, and `AddGroupMarkerInput` or `AddGroupMarkerOutput` for each participant.

Bind the marker to its owner channel.

All of these are described under Building the marker.

Thereafter, each time the user drags an input the plug-in notifies DewesoftX so the mathematics recalculates.

Building the marker

Member		Description
`ConnectInputEx`	method → `IMarkerObject`	Creates a draggable input of a given value type and appends it. The way to add an input.
`ConnectOutputEx`	method → `IMarkerObject`	Creates a result output of a given value type and appends it.
`SetMarkerProps`	method	Sets the capability flags from a `Variant` array — positional, at least 4 elements. These decide which options the marker's setup dialog offers.
`ClearGroups`	method	Resets the drawing before rebuilding it. Logical only: the next `CreateGroup` **reuses the slot and inherits the previous group's colour, visibility and user-assigned name**, which is what preserves a user's customisation across a reconfigure.
`CreateGroup`	method	Appends a draw group. Takes no index and returns nothing — the new group's index is the previous count, which you must track yourself.
`AddGroupStyle`	method	Adds a style to a group: `0` point, `1` result, `2` XY line, `3` region X, `4` line X, `5` line Y, `6` region Y, `7` radial, `8` region XY, `9` inverse region X, `10` inverse region Y, `11` polyline.
`AddGroupName`	method	Appends a name to a group. **Must be called at least once per group** or `IMarkerObject.Name` silently does nothing.
`AddGroupMarkerInput`	method	Adds an existing input to a group — this is what makes it visible.
`AddGroupMarkerOutput`	method	Adds an existing output to a group.
`AddGroupDisplayOption`	method	Adds a display option to a group: `0` hide in the marker table, `1` hide on the graph.

A group's style dictates how many inputs it needs before the marker can be dragged: one for a line, two for a region or point, three for radial, four for an XY line or region. Get the count wrong and the marker draws but cannot be moved, with no error.

Reading the model

Member		Description
`MarkerInputs`	property, read-only → `IMarkerObjectsList`	The marker's inputs, in connection order.
`MarkerOutputs`	property, read-only → `IMarkerObjectsList`	The marker's outputs, in connection order.
`GroupMarkerInputs`	property, read-only → `IMarkerObjectsList`	The inputs in one draw group, by index.
`GroupMarkerOutputs`	property, read-only → `IMarkerObjectsList`	The outputs in one draw group, by index.
`HasStyle`	method → `Boolean`	Whether a group carries a given style.

None of the `Group…` members bound-check their index. An out-of-range index either faults or, for the flag properties below, returns an uninitialised value.

Appearance and interaction

Member		Description
`GroupColor`	property, read/write	A group's colour. **Asymmetric**: reading returns the effective colour, resolved template → user → processing; writing sets only the *processing* colour, the lowest priority. Setting it on a marker with a template or user colour appears to do nothing.
`GroupVisible`	property, read/write	Whether a group is drawn.
`GroupHasOwnCol or`	property, read/write	Marks a group as holding its own colour, so a marker-level colour change does not overwrite it.
`GroupAlwaysDra wInputs`	property, read/write	Forces a group to draw its cursors even where they would normally be suppressed.
`GroupWaitForMo useInput`	property, read/write	Marks a group as not yet positioned; the marker cannot be dragged until it has received a click.

Obsolete members

These predate the current model. Use the replacements.

Member	Instead use	Why
`ConnectInput`	`ConnectInputEx`	Writes into a slot the count setter never allocated, so on a new marker it raises or silently drops the input.
`ConnectOutput`	`ConnectOutputEx`	Same defect.
`InputCount` (write)	`ConnectInputEx`	The setter only moves a counter — it allocates nothing, so a larger value makes the list report slots that do not exist.
`OutputCount` (write)	`ConnectOutputEx`	Same.
`GetMarkerInput`	`MarkerInputs.Items`	Raises on an out-of-range index instead of returning nothing.
`GetMarkerOutput`	`MarkerOutputs.Items`	Same.
`SetMarkerInput`	—	Copies only the value type; the object passed is otherwise discarded. Nothing is replaced.
`SetMarkerOutput`	—	Copies only the channel and value type.
`MarkerObjects`	`MarkerInputs`	Returns the **inputs** list, despite the name.
`MarkerOwner`	the mathematics module	The setter is an empty body. See `IMarkerOwner`.
`DisconnectInputs`	—	Logical only: the next `ConnectInputEx` reuses the slot and inherits the old position and unit, so it is not a clean slate.
`DisconnectOutputs`	—	Same.

`InputCount` and `OutputCount` are safe to *read*.

IProcessingMarkerList

Every marker in the setup. Markers register themselves here when they are created.

Nominally reachable as `IData.ProcessingMarkers`. See `IProcessingMarker` for the model.

> **Not usable from an automation client**
>
> This list is on the automation path — `app.Data.ProcessingMarkers` resolves — but the object is not available to late binding, so **not even `Count` resolves**.

Members

Member		Description
`Count`	property, read-only	Number of markers.
`Items`	property, read/write → `IProcessingMarker`	The marker at a position. Silently returns nothing for an index at or beyond `Count`; a **negative** index raises.
`Add`	method	Appends a marker, unless it is already present — a duplicate is a silent no-op.
`Insert`	method	Inserts at a position. Silently does nothing past the end, and unlike `Add` does **not** check for duplicates.
`Remove`	method → `Integer`	Removes a marker and returns the index it held, or `-1` if it was not in the list.
`Delete`	method	Removes the entry at a position, silently doing nothing if out of range.
`Clear`	method	Empties the list.

The list does not own its markers, so `Remove`, `Delete` and `Clear` do not delete a marker or its mathematics module — they only forget it. Deleting a marker properly is done through the plug-in's context object.

IProjectManager

The projects configured in DewesoftX. A project bundles the setups, data directory and settings for one measurement task, so switching project switches all of them together.

Get it from `IApp.ProjectManager`.

PYTHON

```python
pm = app.ProjectManager
print(pm.GetCurrentProject())
for name in pm.GetProjects():
    print(name)
```

C#

```csharp
dynamic pm = app.ProjectManager;
Console.WriteLine(pm.GetCurrentProject());
foreach (string name in (object[])pm.GetProjects())
    Console.WriteLine(name);
```

Members

GetProjects

Method — returns `Variant`

The file names of all available projects, as an array of strings.

GetCurrentProject

Method — returns `String`

The file name of the project currently active.

ChangeProject

Method — returns `Boolean`

Switches to another project. Returns `True` if the switch happened.

Parameter	Type	Direction	Description
`ProjectName`	`String`	in	Project file name **including its extension**, for example `default.d7p`.

Switching project changes the data and setup directories, so paths obtained earlier from `IApp.GetSpecDir` or `IApp.MainDataDir` may no longer be current.

IProperties

A bag of named values attached to a channel — anything you or a setup wants to carry alongside it that the channel itself has no field for.

Get it from `IChannel.Properties`, or from `IInputGroup.Properties`.

Values are variants, so a property holds whatever fits. For something more structured than a single value, a property can hold a whole XML document instead.

PYTHON

```python
props = channel.Properties
props.Add("CalibratedBy", "A. Smith")
print(props.Item("CalibratedBy"), "of", props.Count)
```

C#

```csharp
dynamic props = channel.Properties;
props.Add("CalibratedBy", "A. Smith");
Console.WriteLine($"{props.Item["CalibratedBy"]} of {props.Count}");
```

Members

Count

Property — read-only — `Integer`

Number of properties.

Item

Property — read/write — `Variant`

The value of one property.

Parameter	Type	Direction	Description
`Index`	`Variant`	in	Either the property's name, or its position as a number.

The index is a variant so it takes either form. A name that does not exist is not the same as a position out of range — check `Count` when indexing by number.

Add

Method

Adds a property.

Parameter	Type	Direction	Description
`Name`	`String`	in	The property's name.
`Value`	`Variant`	in	Its value.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the properties can be iterated directly in languages that support it.

XML properties

AddXmlItem

Method — returns `IDwXMLDocument`

Adds a property whose value is an XML document, and returns the empty document for you to fill in.

Parameter	Type	Direction	Description
`Name`	`String`	in	The property's name.

XmlItem

Property — read-only — `IDwXMLDocument`

The XML document held by a property.

Parameter	Type	Direction	Description
`Name`	`String`	in	The property's name.

Public properties

These are the subset of properties published beyond the channel — visible to exports and to other parts of DewesoftX rather than kept privately on the channel.

AddPublicProperty

Method

Adds a property and marks it public.

Parameter	Type	Direction	Description
`Name`	`String`	in	The property's name.
`Value`	`Variant`	in	Its value.

ClearPublicProperties

Method

Removes every public property. The private ones are left alone.

IRTC

The real-time control system.

Its one member leads to the controllers.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Controllers`	The controllers, as `IRTCControllers`.

IRTCController

One real-time controller — its modules, its scan rate, and its state.

Its modules are `IRTCModules`.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Modules`	The controller's modules.
`ScanCycleFrequency`	How often the controller runs its cycle.
`GetStatus`	The controller's current state.
`Start`	Start this controller.
`Stop`	Stop it.
`ApplyChanges`	Commit pending configuration changes.
`IsConfigChanged`	Whether there are uncommitted changes.
`CheckConfiguration`	Validate the configuration.
`ReadDefaultValues`	Reload default values.

IRTCControllers

The real-time controllers, and starting and stopping them together.

Each controller is an `IRTCController`.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Count`	Number of controllers.
`Items`	The controller at a position.
`Start`	Start every controller.
`Stop`	Stop them.
`ApplyChanges`	Commit pending configuration changes.
`IsConfigChanged`	Whether there are uncommitted changes.
`CheckConfiguration`	Validate the configuration.
`ReadDefaultValues`	Reload default values.

IRTCModule

One real-time module, and its properties.

Properties come in two families: the module's own, and the device's behind it. Each family has a get, a set, an execute and a metadata call, the metadata arriving as `IRTCPropertiesInfo`.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Name`	The module's name.
`Enabled`	Whether the module runs.
`GetProperty`	Read a module property.
`SetProperty`	Write one.
`ExecuteProperty`	Invoke one as an action.
`GetPropertiesInfo`	What module properties exist, and their types.
`GetDeviceProperty`	Read a property of the device behind the module.
`SetDeviceProperty`	Write one.
`ExecuteDeviceProperty`	Invoke one as an action.
`GetDevicePropertiesInfo`	What device properties exist, and their types.

IRTCModules

The modules of one real-time controller.

Each is an `IRTCModule`.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Count`	Number of modules.
`Items`	The module at a position.

IRTCPropertiesInfo

What properties a real-time module has.

Each entry is an `IRTCPropertyInfo`.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Count`	Number of properties.
`Items`	The property at a position.

IRTCPropertyInfo

The name and type of one real-time module property.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Name`	The property's name, as passed to `GetProperty` and `SetProperty`.
`Type`	Its type.

IRTCore

The real-time core.

Its one member hands back an untyped object, so even registered it would tell you little.

> **Not usable from an automation client**
>
> The real-time control objects are not registered for late binding, so **no member name on one resolves** from Python or C#. `app.RTC` hands you an object and every access on it then fails.
>
> The whole chain below it is affected, since the only way in is through that object.

Members

Member	Description
`Data`	The core's data object, as a bare `IUnknown`.

IRTModule

A real-time module as the loader sees it.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member	Description
`ModuleId`	The module's identifier.
`DataModule`	Its data object.

IRTModuleLoader

Loads real-time modules and their drivers.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

Members

Member	Description
`AddDriver`	Register a driver.
`LoadModule`	Load a module, as an `IRTModule`.

IRegistrationHelper

Lets a plug-in ask what the active licence covers.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

Members

Member	Description
`CheckRegistration`	Whether the plug-in is registered.
`ActiveLicenseIncludes`	Whether the active licence includes a given item.

IRemoteManager

The measurement units in a distributed setup.

Get it from `IApp.RemoteManager`. Each unit is an `IMeasUnit`.

PYTHON

```python
rm = app.RemoteManager
print("connected:", rm.Connected, "units:", rm.Count)
for i in range(rm.Count):
    print(rm.Item(i).Used)
```

C#

```csharp
dynamic rm = app.RemoteManager;
Console.WriteLine($"connected: {rm.Connected} units: {rm.Count}");
for (int i = 0; i < (int)rm.Count; i++)
    Console.WriteLine(rm.Item[i].Used);
```

Members

Connected

Property — read-only — `Boolean`

`True` when the local instance is connected to its remote units.

Count

Property — read-only — `Integer`

Number of measurement units configured. `0` on a standalone instance.

Item

Property — read-only — `IMeasUnit`

The unit at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the units can be iterated directly in languages that support it.

IReport

The live link between DewesoftX analysis mode and a Microsoft Word document.

While a link is active, a display widget — a recorder, scope, table or any other grouped visual control — can be pushed into the document as a picture that remembers which data file and which widget it came from. Reopening the data file later lets the embedded widget be refreshed in place, so a report re-renders against current data instead of holding a dead screenshot.

The link is served by a separate Word add-in, so reporting needs that add-in installed.

Get it from `IApp.Report`.

A normal session

Enter analysis mode and load a data file.

`LinkDewesoftToWord` with the document's full path.

Select a widget on a display, then `OpenWidgetInCurrentWordDocument` to insert it.

`UpdateSelectedWidget` to refresh one already inserted.

PYTHON

```python
r = app.Report
r.LinkDewesoftToWord(r"C:\Reports\Run.docx")
print(r.IsInReportingMode, r.GetActiveDocument())
```

C#

```csharp
dynamic r = app.Report;
r.LinkDewesoftToWord(@"C:\Reports\Run.docx");
Console.WriteLine($"{r.IsInReportingMode} {r.GetActiveDocument()}");
```

Members

IsInReportingMode

Property — read-only — `Boolean`

Whether a reporting session is active.

Every widget call below is gated on this. While it is `False` they do nothing at all rather than reporting a problem. While it is `True`, DewesoftX also refuses to exit.

LinkDewesoftToWord

Method

Starts a reporting session against a document.

Parameter	Type	Direction	Description
`DocumentName`	`String`	in	**Full path** to an existing `.docx` or `.doc`.

Nothing is validated

Unlike the interactive *Link with Word* command, this method checks nothing — not that the add-in is installed, not that the file exists, not that Word and DewesoftX match in bitness.

The path is stored as given and its directory is used later to write the temporary widget image, so a bare file name or a wrong path makes every subsequent widget push fail. Reporting is an analysis-mode feature.

GetActiveDocument

Method — returns `String`

Full path of the linked document, or an empty string when no session is active.

SetActiveDocument

Method

Not implemented. Its body is empty: the call succeeds and changes nothing. Use `LinkDewesoftToWord` to set the document.

Parameter	Type	Direction	Description
`DocumentFilePath`	`String`	in	Ignored.

OpenWidgetInCurrentWordDocument

Method

Inserts a widget into the linked document. DewesoftX renders it to an image and hands Word that image together with the data file name and the widget's definition, so it can be refreshed later.

Parameter	Type	Direction	Description
`WidgetID`	`String`	in	**Ignored** — see below.

The argument is ignored; selection decides

Whatever you pass, the method acts on the **currently selected** widget on the current display. Passing a different identifier will not select or insert that widget.

So select the widget in DewesoftX first. The call does nothing if no display is active, if the selection is not a widget group, or if `IsInReportingMode` is `False`.

UpdateSelectedWidget

Method

Re-renders the selected widget and replaces its existing copy in the document, rather than inserting another. This is how a report is refreshed after changing the data file, the selected region, or the widget's own settings.

Operates on the current selection only — there is no way to name a target. It does not check that the widget is actually in the document, so calling it for one never inserted has no useful effect.

OnWordShutdown

Method

Tells DewesoftX that Word has ended the session: it reports that reporting stopped, then tears the link down.

For the Word add-in to call, not for clients

This is the inbound half of the link. It shows a modal message box in DewesoftX, so calling it from a script blocks until somebody acknowledges it.

Because it treats the shutdown as Word-initiated, DewesoftX deliberately does not tell Word to unlink — so a client calling it leaves Word holding a link DewesoftX has forgotten. To end a session from the DewesoftX side, use the *Unlink from Word* command.

IResamplerChannel

One input channel of the resampling engine, and the buffers its resampled data is read from.

Hands out the sample, time and size buffers directly as pointers.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IScreen

One display screen — a page of widgets, and the operations that act on it.

Get it from `IScreens.Item` or `IScreens.Current`.

PYTHON

```python
s = app.Screens.Item(0)
print(s.Name, s.TemplateName)
s.Show()
s.CopyImageToFile(1920, 1080, 0xFFFFFF, r"C:\temp\screen.png")
```

C#

```csharp
dynamic s = app.Screens.Item[0];
Console.WriteLine($"{s.Name} {s.TemplateName}");
s.Show();
s.CopyImageToFile(1920, 1080, 0xFFFFFF, @"C:\temp\screen.png");
```

Identity

Name

Property — read-only — `String`

The screen's caption, as it appears on its tab.

TemplateName

Property — read-only — `String`

The display template the screen was built from, or empty when it was not built from one.

Id

Property — read-only — `Integer`

The screen's **type**, not a unique identifier — screens of the same kind share a value, so several screens in one setup normally report the same `Id`. Use the position in `IScreens.Item` to address a particular screen.

IsCurrent

Property — read-only — `Boolean`

`True` when this is the screen on display. All screens report `False` when none is current.

Showing the screen

Show

Method

Brings the screen up, as clicking its tab would.

ShowOnMonitor

Method

Moves the screen onto another monitor.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Zero-based monitor index.

A negative index does nothing, as does asking for the monitor the main window is already on. An index above the number of monitors raises.

CopyImageToFile

Method

Renders the screen to an image file at a size you choose, independent of how large it is on screen.

Parameter	Type	Direction	Description
`Width`	`Integer`	in	Image width in pixels.
`Height`	`Integer`	in	Image height in pixels.
`Background`	`Integer`	in	Background colour, as a BGR integer.
`Filename`	`String`	in	Destination path.

The file is always written as PNG whatever extension you give it, so name it `.png` to avoid a file that lies about its format.

Cursors and zoom

These act on the recorder graphs on the screen. A screen with no recorder on it accepts them and does nothing.

GetCursor

Method — returns `Double`

The position of a cursor on the screen's first recorder graph.

Parameter	Type	Direction	Description
`ACursor`	`Integer`	in	Which cursor: `0` or `1`.
`AbsTime`	`Boolean`	in	`True` for absolute time, `False` for time relative to the start of the data.

Returns **not-a-number** when the screen has no recorder graph — test for that rather than treating it as a position.

SetCursor

Method

Moves a cursor and repaints.

Parameter	Type	Direction	Description
`ACursor`	`Integer`	in	Which cursor: `0` or `1`.
`Time`	`Double`	in	The new position, in the units `AbsTime` selects.
`AbsTime`	`Boolean`	in	`True` for absolute time, `False` for relative.

Unlike `GetCursor`, this sets the cursor on **every** recorder graph on the screen, not just the first.

ZoomIn

Method

Zooms the first recorder graph in one step.

ZoomOut

Method

Zooms the first recorder graph out one step.

UpdateLockableCursors

Method

Re-reads the locked cursor positions into the screen's recorder graphs and repaints them. Call it after setting `ILockableCursor.Position` so the change shows.

Widgets and channels

A widget is addressed by its position on the screen, in the order widgets were added.

> **These act on the screen currently on display**
>
> All five members below operate on whichever screen is up at the time, not on the screen you called them on. Call `Show` first and check `IsCurrent`, or you will edit the wrong screen.
>
> With no screen on display they do nothing — except `AddControl`, which fails.

AddControl

Method

Adds a widget to the screen.

Parameter	Type	Direction	Description
`Id`	`Integer`	in	The kind of widget to create, as a widget id.

The ids come from the widgets registered in the running program rather than a published set, so a value only means the same thing on the same build.

RemoveControl

Method

Removes a widget.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	The widget's position on the screen.

AssignChannel

Method

Puts a channel into a widget, appending it to whatever the widget already shows.

Parameter	Type	Direction	Description
`ControlIndex`	`Integer`	in	The widget's position on the screen.
`ChannelName`	`String`	in	The channel's name.

UnassignChannel

Method

Takes one channel back out of a widget.

Parameter	Type	Direction	Description
`ControlIndex`	`Integer`	in	The widget's position on the screen.
`ControlChannelIndex`	`Integer`	in	Which of the widget's channels, by position.

IsChannelAssigned

Method — returns `Boolean`

Whether a widget has a channel in the given slot.

Parameter	Type	Direction	Description
`ControlIndex`	`Integer`	in	The widget's position on the screen.
`ControlChannelIndex`	`Integer`	in	The slot to test, by position.

A position that does not exist is ignored rather than reported, and `AssignChannel` also does nothing when the widget will not take the channel you offered. None of the four returns anything, so confirm with `IsChannelAssigned` rather than assuming the edit landed.

IScreens

The display screens — the pages of widgets a user switches between, shown in **Live view**.

Get it from `IApp.Screens`. Each screen is an `IScreen`.

The list is flat. Screens that a setup nests under a parent still appear here as individual entries, in display order.

PYTHON

```python
screens = app.Screens
for i in range(screens.Count):
    s = screens.Item(i)
    print(s.Name, s.IsCurrent)
```

C#

```csharp
dynamic screens = app.Screens;
for (int i = 0; i < (int)screens.Count; i++)
{
    dynamic s = screens.Item[i];
    Console.WriteLine($"{s.Name} {s.IsCurrent}");
}
```

Members

Count

Property — read-only — `Integer`

Number of screens.

Item

Property — read-only — `IScreen`

The screen at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

An index outside that range raises rather than returning nothing, so check `Count` first.

Current

Property — read-only — `IScreen`

The screen on display, or **nothing** when none is — which is the case before a measurement or analysis has put a screen up. Test the result before using it.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the collection can be iterated directly in languages that support it. Iterating yields the same screens as `Item`.

FindWidgetByID

Method

Brings the widget with the given id into view: switches to the screen holding it, selects it, and turns on design mode.

Parameter	Type	Direction	Description
`WidgetUniqueID`	`String`	in	The widget's unique id, as a decimal string.

> **Opens a dialog when the widget is not found**
>
> A miss puts a *Widget not found* message box on the DewesoftX window and waits for someone to dismiss it. Your call does not return until they do. Only pass ids you know exist.

There is no return value and no way to ask whether the search succeeded.

IsReportGroupOwner

Method — returns `Boolean`

Reports whether the screen currently on display is a report.

Parameter	Type	Direction	Description
`Datafile`	`String`	in	Ignored.
`GroupName`	`String`	in	Ignored.

Both arguments are accepted and discarded — the answer describes the current screen, whatever you pass. It is `False` when no screen is current.

ISequencer

The test automation sequencer — its name, its variables, and stopping it.

Get it from `IApp.Sequencer`.

This is a narrow surface: it will tell you which sequence is loaded, move its variables in and out as files, and stop it. There is nothing here that loads or starts a sequence.

PYTHON

```python
seq = app.Sequencer
print("sequence:", seq.Name or "(none loaded)")
seq.ExportSeqVariables(r"C:\temp\vars.xml")
```

C#

```csharp
dynamic seq = app.Sequencer;
Console.WriteLine($"sequence: {seq.Name}");
seq.ExportSeqVariables(@"C:\temp\vars.xml");
```

Members

Name

Property — read-only — `String`

The name of the loaded sequence, or empty when none is loaded.

ExportSeqVariables

Method

Writes the sequence's variables out to a file.

Parameter	Type	Direction	Description
`FileName`	`String`	in	Where to write them.

ImportSeqVariables

Method

Reads variables back in from a file.

Parameter	Type	Direction	Description
`FileName`	`String`	in	The file to read.

This is the way to drive a sequence from outside: write the inputs to a file, import them, and let the sequence read its variables.

StopSequence

Method

Stops the running sequence. Does nothing when none is running.

ISetupMessages

Collects the warnings and errors shown in **Config**.

A plug-in adds a message here to have it appear alongside the setup rather than in a dialog.

> **Not callable from an automation client**
>
> Nothing on the automation surface returns this interface, so a client cannot obtain one.

Members

Member	Description
`Add`	Add a message.

ISingleValueMarkerInput

Obsolete and non-functional. Every member asserts and returns nothing usable.

This was the input of a marker occupying a single point on an axis. That role is now served by `IMarkerInput`, one per position — use `ValueEx`.

Members

Member		Description
`Value`	property, read/write	Asserts on both read and write. Reading returns an uninitialised number.

IStoreEngine

Storing: whether it is running, where it is writing, how large the file has grown, and the events recorded alongside the data.

Get it from `IApp.StoreEngine`. Storing is *started* and *stopped* through `IApp` — `StartStoring`, `StopStoring`, `PauseStoring` — while this interface reports on it.

PYTHON

```python
se = app.StoreEngine
print(se.Storing, se.FileName, se.FileSize)
```

C#

```csharp
dynamic se = app.StoreEngine;
Console.WriteLine($"{se.Storing} {se.FileName} {se.FileSize}");
```

State

Storing

Property — read-only — `Boolean`

`True` while data is being written to a file.

Paused

Property — read-only — `Boolean`

`True` while storing is paused — either because `IApp.PauseStoring` was called, or because a trigger is armed but not currently active.

IsTriggering

Property — read-only — `Boolean`

`True` between a start trigger and its stop trigger. With a *fast on trigger* store mode, this is `True` exactly while fast data is being written.

StoreMode

Property — read-only — `Integer`

Value	Mode	Behaviour
`0`	always fast	Everything at the dynamic acquisition rate.
`1`	always slow	Everything at the reduced rate.
`2`	fast on trigger	Fast data only while a trigger is active; nothing otherwise.
`3`	fast on trigger, slow otherwise	Fast while a trigger is active, reduced data the rest of the time.

Set it with `IApp.SetStoreMode`. Modes `1` and `3` use `IApp.ReducedRate`.

FileName

Property — read-only — `String`

Name of the file being written.

FileSize

Property — read-only — `UInt64`

Current size of the file in bytes. It starts at `0` when storing begins and grows from there, so it is a convenient progress indicator for an unattended run.

Multiple files

StartNewMultifile

Method

Closes the file being written and continues storing into a new one with the next generated name, without interrupting acquisition. This is the manual equivalent of the automatic flip driven by `IFileNameSettings.AutoFlipFile`.

> **Raises unless storing is running and automatic naming is on**
>
> It raises `Not storing` if storing is not active, and `Multifile not enabled` unless at least one of `UseMultiFile`, `UseTime` or `UseDate` is set. So call it only after storing has begun.

The flip is not immediate — it is requested, and happens on the next acquisition cycle.

Events

An event is a marker stored alongside the data, visible on the recorder and in the event list.

AddNewEvent

Method

Adds an event at the current position.

Parameter	Type	Direction	Description
`Type_`	`EventType`	in	`21` text, `20` keyboard — see below.
`Data`	`Variant`	in	Event text, for a text event. Ignored otherwise.

PYTHON

```python
ET_TEXT = 21
app.StoreEngine.AddNewEvent(ET_TEXT, "valve opened")
```

C#

```csharp
const int ET_TEXT = 21;
app.StoreEngine.AddNewEvent(ET_TEXT, "valve opened");
```

Only two event types do anything

The `EventType` enumeration declares ten members, but only these two can be added through this interface:

Value	Member	Effect
`21`	`etText`	A text event. `Data` becomes its text.
`20`	`etKeyboard`	A keyboard event. `Data` is ignored.

The other eight — `etStart` (1), `etStop` (2), `etTrigger` (3), `etVStart` (11), `etVStop` (12), `etVoice` (22), `etPicture` (23), `etModule` (24) — are **silently discarded**. No event is created and no error is reported. Start, stop, trigger and video events are generated by DewesoftX itself and cannot be injected.

A `Data` value that cannot be converted to a string yields the literal text `Error text` rather than an error.

AddNewEventWithTime

Method

Adds an event at a time you choose, rather than at the current position. `AddNewEvent` is this method called with the current acquisition time.

Parameter	Type	Direction	Description
`Type_`	`EventType`	in	As for `AddNewEvent`.
`Data`	`Variant`	in	As for `AddNewEvent`.
`Time`	`Double`	in	Relative time in seconds from the start of acquisition.

A time ahead of the current position is queued and committed when acquisition reaches it. A time in the past is inserted there, which may be in data already written. Negative values are clamped to `0`. Events only reach the file while storing is active.

Internal

AllowIBSkipping

Property — read/write — `Boolean`

Internal to DewesoftX.

StartStoreTimeChanged

Method

Internal to DewesoftX.

TrackingOffset

Property — read-only — `Double`

Internal to DewesoftX.

ISyncSource

The sample rate an array channel's axis is generated against, and whether that channel is the one defining it.

Get it from `IArrayInfo.SyncSource` on an array channel.

PYTHON

```python
ss = channel.ArrayInfo.SyncSource
print(ss.SampleRate, ss.IsSyncSource)
```

C#

```csharp
dynamic ss = channel.ArrayInfo.SyncSource;
Console.WriteLine($"{ss.SampleRate} {ss.IsSyncSource}");
```

Members

SampleRate

Property — read/write — `Double`

The rate in samples per second that the channel's axis is referred to.

IsSyncSource

Property — read/write — `Boolean`

`True` when this channel is the one that defines the rate for others rather than following it.

Both are stored as given, with no validation and no effect on other channels at the moment of writing — marking two channels as the source leaves two marked.

ISynchronization

How a device locks its clock to something else — which synchronisation modes it can provide, which it can follow, and which one is in use.

Get one from `IDaq.CreateSynchronization`. Every call returns a **new, empty** object, so it describes a device's synchronisation rather than reporting the running system's — fill it in and hand it back.

A device can be either end of a link: a **source** hands its clock to others, a **slave** follows someone else's. Most devices support several of each, so both are sets of modes rather than single values.

PYTHON

```python
s = app.Daq.CreateSynchronization()
s.AddSourceMode(12)       # can provide GPS PPS
s.AddSlaveMode(4)         # can follow a clock/trigger input
s.AddSlaveMode(11)        # ...and NTP
s.CurrentMode = 12
s.IsClockSource = True
```

C#

```csharp
dynamic s = app.Daq.CreateSynchronization();
s.AddSourceMode(12);      // can provide GPS PPS
s.AddSlaveMode(4);        // can follow a clock/trigger input
s.AddSlaveMode(11);       // ...and NTP
s.CurrentMode = 12;
s.IsClockSource = true;
```

Modes

Value	Mode
`0`	not defined
`1`	custom — named separately, see custom modes
`2`	software sync
`3`	stand-alone, no synchronisation
`4`	clock and trigger input
`5`	IRIG-A, DC coupled
`6`	IRIG-B, DC coupled
`7`	IRIG-G, DC coupled
`8`	IRIG-A, AC coupled
`9`	IRIG-B, AC coupled
`10`	IRIG-G, AC coupled
`11`	NTP
`12`	GPS PPS
`13`	EtherCAT PPS
`14`	PTP over UDP
`15`	PTP over IEEE 802.3
`16`	NET

Declaring what a device supports

AddSourceMode

Method

Records that the device can act as a source for a mode.

Parameter	Type	Direction	Description
`SyncMode`	`Integer`	in	A mode from the table above.

RemoveSourceMode

Method

Takes a mode back out of the source set.

Parameter	Type	Direction	Description
`SyncMode`	`Integer`	in	A mode from the table above.

AddSlaveMode

Method

Records that the device can follow a mode as a slave.

Parameter	Type	Direction	Description
`SyncMode`	`Integer`	in	A mode from the table above.

RemoveSlaveMode

Method

Takes a mode back out of the slave set.

Parameter	Type	Direction	Description
`SyncMode`	`Integer`	in	A mode from the table above.

Both sets behave as sets: adding a mode twice is the same as adding it once, and removing one that was never added does nothing. Neither set can be read back or cleared — the four methods above are the whole of the interface to them, so keep your own record of what you put in.

Custom modes

A device with a synchronisation scheme not in the table declares `CurrentMode` `1` and names its own modes here.

CustomSourceModeAdd

Method

Appends a named custom source mode.

Parameter	Type	Direction	Description
`CustomMode`	`String`	in	The mode's name.

CustomSourceModeCount

Method — returns `Integer`

How many custom source modes have been added.

CustomSourceModeItem

Method — returns `String`

The custom source mode at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `CustomSourceModeCount - 1`.

CustomSlaveModeAdd

Method

Appends a named custom slave mode.

Parameter	Type	Direction	Description
`SyncMode`	`String`	in	The mode's name.

CustomSlaveModeCount

Method — returns `Integer`

How many custom slave modes have been added.

CustomSlaveModeItem

Method — returns `String`

The custom slave mode at a position.

Parameter	Type	Direction	Description
`I`	`Integer`	in	Position, `0` to `CustomSlaveModeCount - 1`.

These are plain lists, not sets: the same name added twice appears twice, and there is no way to remove one.

Current state

CurrentMode

Property — read/write — `Integer`

The mode in use, from the table above.

IsClockSource

Property — read/write — `Boolean`

`True` when the device is providing the clock, `False` when it is following one.

CurrentCustomModeIdx

Property — read/write — `Integer`

Which custom mode is in use, as a position in the custom list. Only meaningful while `CurrentMode` is `1`.

CurrentCustomModeValue

Property — read/write — `Integer`

A value belonging to the selected custom mode, for a scheme that needs one number — a divider or a channel, say. Its meaning is the device's own.

SupportsAutomaticMode

Property — read/write — `Boolean`

`True` when the device can work out its own synchronisation rather than being told. Writable because the device declares it.

AutomaticMode

Property — read/write — `Boolean`

`True` to let the device choose its synchronisation. Only useful when `SupportsAutomaticMode` is `True`.

ITiming

The state of the timing source — whether it is locked, and what time it says.

Get it from `IApp.Timing`. It is **nothing** unless a timing device is configured, which is the usual case on hardware without one, so test the result before using it.

Everything here is read-only: this reports how timing is behaving, it does not configure it. Which modes a device can synchronise with is `ISynchronization`.

PYTHON

```python
t = app.Timing
if t is not None:
    print(t.Tracking, t.DeviceSyncError, t.GetAbsTime())
```

C#

```csharp
dynamic t = app.Timing;
if (t != null)
    Console.WriteLine($"{t.Tracking} {t.DeviceSyncError} {t.GetAbsTime()}");
```

Members

Tracking

Property — read-only — `Boolean`

`True` while the timing source is locked and following its reference — a GPS receiver that has acquired satellites, or an IRIG input reading a good signal.

`False` covers both "not locked yet" and "lost lock", which are not distinguished.

DeviceSyncError

Property — read-only — `Boolean`

`True` when a device has fallen out of synchronisation with the timing source. Worth checking alongside `Tracking`: the source can be locked while an individual device has still drifted.

GetAbsTime

Method — returns `Double`

The current absolute time from the timing source, as a date-and-time value.

ITrig

One half of a trigger — the conditions that fire it, and for a start trigger the conditions that suppress it.

Reached from `ITrigger.StartTrig` and `ITrigger.StopTrig`, and also from `IAlarmCond.Trigger` and `IAlarmCond.StopTrigger`.

PYTHON

```
start = app.Trigger.StartTrig
print(start.OrList.Count, start.NotOrList.Count)
```

C#

```
dynamic start = app.Trigger.StartTrig;
Console.WriteLine($"{start.OrList.Count} {start.NotOrList.Count}");
```

Members

OrList

Property — read-only — `ITriggerCondList`

The conditions that fire the trigger. They are OR-combined — any one of them is enough.

NotOrList

Property — read-only — `ITriggerCondList`

Conditions that suppress the trigger: while any of them is met, the trigger does not fire even if a condition in `OrList` is.

Only meaningful on a start trigger. A stop trigger uses `OrList` alone.

TrigIndex

Property — read-only — `Integer`

Position in `OrList` of the condition that actually caused the current trigger — which is how you tell which of several conditions fired.

GetTrigIndexEx

Method

The condition that fired, together with which of its channels was responsible.

Parameter	Type	Direction	Description
`CondIndex`	`Integer`	out	Position in `OrList`.
`ChIndex`	`Integer`	out	Position within that condition's channel list.
`Ch`	`IChannel`	out	The channel itself.

PYTHON

```
cond_index, ch_index, ch = app.Trigger.StartTrig.GetTrigIndexEx()
```

C#

```
app.Trigger.StartTrig.GetTrigIndexEx(out int condIndex,
                                     out int chIndex, out dynamic ch);
```

ITrigInfo

What a trigger was set to at the moment it fired. It is a snapshot of the `ITriggerCondition` that caused a trigger event, recorded with the event and stored in the data file — so you can tell after the fact why a trigger happened, even if the setup has since changed.

Get it from `IEvent.TrigInfo` on an event whose `Type_` is `3` (trigger).

Every member is read-only, and the values mean exactly what they mean on `ITriggerCondition` — the same `Mode` table applies.

PYTHON

```python
for ev in app.EventList:
    if ev.Type_ == 3 and ev.TrigInfo:
        ti = ev.TrigInfo
        print(ti.Channel.Name, ti.Mode, ti.Level1, ti.Manual)
```

C#

```csharp
dynamic events = app.EventList;
for (int i = 0; i < (int)events.Count; i++)
{
    dynamic ev = events[i];
    if ((int)ev.Type_ == 3 && ev.TrigInfo != null)
    {
        dynamic ti = ev.TrigInfo;
        Console.WriteLine($"{ti.Channel.Name} {ti.Mode} {ti.Level1} {ti.Manual}");
    }
}
```

Members

Channel

Property — read-only — `IChannel`

The channel that fired the trigger.

Resolved from a stored index the first time it is read, so it is only available while that channel still exists in the setup.

Manual

Property — read-only — `Boolean`

`True` when the trigger was fired by hand — through `IApp.ManualStart` or the trigger button — rather than by a condition being met. The other members are not meaningful for a manual trigger.

Mode

Property — read-only — `Integer`

The condition's mode. See the table under `ITriggerCondition.Mode`.

TrigValue

Property — read-only — `Integer`

Which value was tested: `0` real data, `1` average, `2` RMS, `3` maximum, `4` minimum.

Level1

Property — read-only — `Double`

The condition's first level, as interpreted by its `Mode`.

Level2

Property — read-only — `Double`

The condition's second level.

Direction

Property — read-only — `Integer`

The condition's direction.

Direction1

Property — read-only — `Integer`

The condition's second direction.

DeltaTime

Property — read-only — `Double`

The time the condition compared against, for the pulse-width and slope modes.

ITrigger

The store trigger — when storing starts, when it stops, and how much data around the trigger is kept.

Get it from `IApp.Trigger`.

The structure has three levels:

```
ITrigger
├── StartTrig  (ITrig)
│   ├── OrList     – start storing when any of these is met
│   └── NotOrList  – but not while any of these is met
└── StopTrig   (ITrig)
    └── OrList     – stop storing when any of these is met
```

Each list holds `ITriggerCondition` objects. Conditions within a list are OR-combined.

PYTHON

```python
trg = app.Trigger
cond = trg.StartTrig.OrList.Add()
cond.AddChannel(app.Data.FindChannel("AI 0"))
cond.Level1 = 2.5
trg.PreTimeUsed = True
trg.PreTime = 500          # ms
```

C#

```csharp
dynamic trg = app.Trigger;
dynamic cond = trg.StartTrig.OrList.Add();
cond.AddChannel(app.Data.FindChannel("AI 0"));
cond.Level1 = 2.5;
trg.PreTimeUsed = true;
trg.PreTime = 500;         // ms
```

Conditions

StartTrig

Property — read-only — `ITrig`

The conditions that start storing, and the conditions that suppress it.

StopTrig

Property — read-only — `ITrig`

The conditions that stop storing.

Data around the trigger

Each of these has a matching `…Used` flag, and the time value is ignored unless its flag is set.

PreTime

Property — read/write — `Single`

How much data before the trigger to keep, in **milliseconds**. DewesoftX holds this much in the buffer and writes it once the trigger fires.

PreTimeUsed

Property — read/write — `Boolean`

Whether `PreTime` applies. When `False`, storing begins at the trigger itself.

PostTime

Property — read/write — `Single`

How much data after the trigger to keep, in **milliseconds**.

When `PostTimeUsed` is `False` there is no post-trigger limit: storing continues until it is stopped by hand or a stop condition is met.

PostTimeUsed

Property — read/write — `Boolean`

Whether `PostTime` applies.

PostTimeExtensionUsed

Property — read/write — `Boolean`

Whether a further trigger arriving while the previous one is still being recorded extends the recording rather than being ignored.

Only meaningful together with `PostTime` — with no post-trigger limit there is nothing to extend.

HoldoffTime

Property — read/write — `Single`

How long to ignore further triggers after one has fired, in **milliseconds**. Useful when a signal produces bursts of triggers and you want one record per burst.

HoldoffTimeUsed

Property — read/write — `Boolean`

Whether `HoldoffTime` applies.

ITriggerCondList

A list of trigger conditions, OR-combined — any one of them firing fires the trigger.

Reached from `ITrig.OrList` and `ITrig.NotOrList`.

PYTHON

```python
lst = app.Trigger.StartTrig.OrList
cond = lst.Add()
print(lst.Count)
```

C#

```csharp
dynamic lst = app.Trigger.StartTrig.OrList;
dynamic cond = lst.Add();
Console.WriteLine(lst.Count);
```

Members

Count

Property — read-only — `Integer`

Number of conditions in the list.

Item

Property — read-only — `ITriggerCondition`

The condition at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

Add

Method — returns `ITriggerCondition`

Appends a condition and returns it, ready to configure.

Remove

Method

Deletes the condition at a position.

Parameter	Type	Direction	Description
`Ind`	`Integer`	in	Position, `0` to `Count - 1`.

Removing re-packs the list, so positions after the removed one shift down. When removing several, iterate downwards.

_NewEnum

Property — read-only — `IUnknown`

The standard COM enumerator, which is what lets the list be iterated directly rather than by index. You never call it yourself — the language does.

ITriggerCondition

One trigger condition — what to watch, and what counts as an event.

Get one from `ITriggerCondList.Add`, on a list reached through `ITrigger`.

PYTHON

```
cond = app.Trigger.StartTrig.OrList.Add()
cond.AddChannel(app.Data.FindChannel("AI 0"))
cond.TrigType = 0          # on data
cond.Mode = 0              # simple edge
cond.Direction = 0         # rising
cond.Level1 = 2.5
```

C#

```
dynamic cond = app.Trigger.StartTrig.OrList.Add();
cond.AddChannel(app.Data.FindChannel("AI 0"));
cond.TrigType = 0;         // on data
cond.Mode = 0;             // simple edge
cond.Direction = 0;        // rising
cond.Level1 = 2.5;
```

What to watch

TrigType

Property — read/write — `Integer`

What kind of condition this is. **Almost every other property depends on this.**

Value	Type	Relevant properties
`0`	on data	`Mode`, `TrigValue`, `Level1`, `Level2`, `Direction`, `Direction1`, `DeltaTime`
`1`	on time	`TimeCond`, `TimeFormat`, `TimeValue`, `TimeUnit`
`2`	on FFT	—

Channels

Property — read-only — `IChannelList`

The channels this condition watches.

AddChannel

Method

Adds a channel to watch.

Parameter	Type	Direction	Description
`Ch`	`IChannel`	in	Channel to add.

DeleteChannel

Method

Removes a watched channel.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Channels.Count - 1`.

ClearChannels

Method

Internal to DewesoftX. Remove channels with `DeleteChannel`.

Conditions on data

These apply when `TrigType` is `0`.

TrigValue

Property — read/write — `Integer`

Which value the condition tests: `0` real data, `1` average, `2` RMS, `3` maximum, `4` minimum.

This restricts which `Mode` values are available — with an averaged value, only simple edge, filtered edge, window and slope apply.

Mode

Property — read/write — `Integer`

How the value is tested, and therefore what `Level1`, `Level2`, `Direction`, `Direction1` and `DeltaTime` each mean:

Mode	Test	Level1	Level2	Direction	Direction1	DeltaTime
0	simple edge	trigger level	—	0 rising, 1 falling	—	—
1	filtered edge	trigger level	re-arm level	0 rising, 1 falling	—	—
2	window	upper level	lower level	0 entering, 1 leaving	—	—
3	pulse width	trigger level	—	0 positive, 1 negative pulse	0 longer than, 1 shorter than	pulse time
4	window and pulse width	upper level	lower level	0 in range, 1 out of range	0 longer than, 1 shorter than	pulse time
5	slope	delta level	delta level	0 positive, 1 negative, 2 any	0 smoother than, 1 steeper than	delta time
6	delta amplitude	delta level	delta level	0 positive, 1 negative, 2 any	—	—

Level1

Property — read/write — `Single`

First level. Its meaning depends on `Mode` — see the table above.

Level2

Property — read/write — `Single`

Second level. Its meaning depends on `Mode` — see the table above.

Direction

Property — read/write — `Integer`

Direction of the event. Its meaning depends on `Mode` — see the table above.

Direction1

Property — read/write — `Integer`

Second direction, used by the pulse-width and slope modes. See the table above.

DeltaTime

Property — read/write — `Double`

The time the pulse-width and slope modes compare against. See the table above.

Conditions on time

These apply when `TrigType` is `1`.

TimeCond

Property — read/write — `Integer`

`0` triggers once at the given time, `1` triggers repeatedly at that interval.

TimeFormat

Property — read/write — `Integer`

`0` the time is relative to the start of the measurement, `1` it is a clock time.

TimeValue

Property — read/write — `Double`

The time, in the unit given by `TimeUnit`.

TimeUnit

Property — read/write — `Integer`

`0` seconds, `1` minutes, `2` hours.

IUIHelper

Declared but unimplemented.

> **Not implemented**
>
> This interface is declared in the type library, but nothing implements it and nothing returns it. There is nothing to obtain and nothing to call.

IUserInterface

Two operations that drive the DewesoftX window itself.

Get it from `IApp.UserInterface`.

Both members act on the interface a person is looking at rather than on measurement state, and one of them waits for that person. Neither is suitable for an unattended client.

PYTHON

```
app.UserInterface.ChangeSetupScreen("Analog in")
```

C#

```
app.UserInterface.ChangeSetupScreen("Analog in");
```

Members

ChangeSetupScreen

Method

Switches the setup area to one of its pages, as clicking that page's tab would.

Parameter	Type	Direction	Description
`ScreenName`	`String`	in	The page's caption, matched without regard to case.

Raises when DewesoftX is not in **Config**. A name that matches no page is accepted and does nothing, so a silent return does not mean the page changed.

ShowTrigCondSetup

Method

Opens the trigger condition setup dialog for one condition, and applies the channels chosen there if the user accepts it.

Parameter	Type	Direction	Description
`Cond`	`ITriggerCondition`	in	The condition to edit.

Waits for a person

The dialog is modal. This call does not return until someone closes it, and while it is open DewesoftX answers nothing else. Do not call it from an unattended script.

Build the condition through `ITrig` and `ITriggerCondition` instead if no one is at the machine.

IVCCheckBoxProperty

A property-panel row holding a single on-or-off setting.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCColorProperty

A property-panel row holding a colour.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCContext

A visual-control plug-in's handle on DewesoftX.

Through it a control reads the channels and input slots assigned to it, asks for the current sample position and value, requests a repaint, learns whether it is in design mode, opens colour and input dialogs, and creates and edits processing markers.

Not callable from an automation client

Nothing on the automation surface returns this interface, so a client cannot obtain one.

IVCFloatProperty

A property-panel row holding a real number, with bounds and a displayed precision.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCIntegerProperty

A property-panel row holding a whole number, with bounds.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCProperties

The property panel a visual control publishes — the settings a user edits when the control is selected.

Properties are organised into groups; this interface finds and creates them.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCPropertiesGroup

One collapsible group of settings in a visual control's property panel.

A group creates its own rows, one method per row type — check box, colour, float, integer, label, search, select and text — each returning the correspondingly typed property.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCProperty

One row in a visual control's property panel — the behaviour every row type shares.

Carries the row's identity, caption, description, indentation, enabled and visible state, and any buttons attached to it. The typed row interfaces extend it.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCSearchProperty

A property-panel row offering a searchable list of choices.

The same shape as `IVCSelectProperty`, presented with a search box for longer lists.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCSelectProperty

A property-panel row offering a fixed list of choices.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVCTextProperty

A property-panel row holding free text.

> **Not callable from an automation client**
>
> This interface is not available to late binding, so a client cannot use it.

IVariableChannel

The control side of a variable channel — a channel whose value you set rather than measure.

Variable channels carry this interface as well as `IChannel`, so one object serves both and you can call this member on any channel that is one.

SetControlValue

Method

Sets the channel's value.

Parameter	Type	Direction	Description
`Value`	`Double`	in	The new value.

PYTHON

```python
ch = app.Data.FindChannel("MyVariable")
ch.SetControlValue(42.0)
```

C#

```csharp
dynamic ch = app.Data.FindChannel("MyVariable");
ch.SetControlValue(42.0);
```

A channel that is not a variable channel does not carry this member, so the call fails as an unknown name rather than being ignored.

IVideo

The cameras configured on the measurement.

Get it from `IApp.Video`. Each camera is an `ICamera`.

PYTHON

```python
v = app.Video
for i in range(v.CameraCount):
    cam = v.Cameras(i)
    print(cam.DisplayName, cam.Used)
```

C#

```csharp
dynamic v = app.Video;
for (int i = 0; i < (int)v.CameraCount; i++)
{
    dynamic cam = v.Cameras[i];
    Console.WriteLine($"{cam.DisplayName} {cam.Used}");
}
```

Members

CameraCount

Property — read-only — `Integer`

Number of cameras.

Cameras

Property — read-only — `ICamera`

The camera at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `CameraCount - 1`.

IVideoFrame

One frame from a camera's buffer.

Get it from `ICamera.FrameList`.

Interpret the bytes with `ICamera.GetBitmapInfoHeader` — the frame itself carries no format information.

Members

GetData

Method — returns `Variant`

The frame's image data, as an array of bytes.

BufSize

Property — read-only — `Integer`

How many bytes the frame holds.

GetTS

Method — returns `Double`

The frame's timestamp, for lining the image up against measured channels.

IVideoLoadEngine

One recorded video stream in a loaded data file.

Get it from `IVideoLoadEngines.Item`.

GetFramesInfo

Method

Describes the frames in the stream.

Parameter	Type	Direction	Description
`Frames`	`Variant`	out	The frame information, as an array.

This is the whole of the interface — enough to find out what is in the stream and where, but not to fetch image data. For that, display the video or export it.

IVideoLoadEngines

The video streams in a loaded data file.

Get it from `ILoadEngine.VideoLoadEngines`. Each stream is an `IVideoLoadEngine`.

This is the stored-video counterpart of `IVideo`: `IVideo` is the cameras on a live measurement, this is the video already recorded in a file.

Members

Count

Property — read-only — `Integer`

Number of video streams in the file.

Item

Property — read-only — `IVideoLoadEngine`

The stream at a position.

Parameter	Type	Direction	Description
`Index`	`Integer`	in	Position, `0` to `Count - 1`.

_NewEnum

Property — read-only — `IUnknown`

The standard enumeration hook, so the streams can be iterated directly in languages that support it.

IViewInfo

A channel's default display preferences — how it should be drawn the first time it lands on a screen.

Get it from `IChannel.ViewInfo`.

> **Not usable from an automation client**
>
> **No member name on this object resolves.** Every read and every write fails with an unknown-name error, however you spell it.
>
> There is no workaround from a client.

Advisory only in any case: these are the settings DewesoftX starts from, and a user's later change to a display overrides them.

PYTHON

```
vi = channel.ViewInfo
vi.AxisViewType     # fails - name not found
```

C#

```
dynamic vi = channel.ViewInfo;
var t = vi.AxisViewType;   // name not found
```

Members

AxisViewType

Property — read/write — `Integer`

How the amplitude axis is scaled:

Value	Scaling
`0`	linear
`1`	logarithmic
`2`	dB relative to 0 dB
`3`	dB relative to the noise floor
`4`	dB relative to a reference value

Graph2DType

Property — read/write — `Integer`

Which 2-D graph style to use: `0` chosen automatically from the channel, `1` histogram, `2` line.

DefaultVC

Property — read/write — `Integer`

Which widget to create for this channel, as an internal widget id, or `-1` for no preference. The ids are not a published set — they are assigned by the widgets registered in the running program, so a value only means anything to the build that produced it.

IXMLHelper

Creates XML documents.

Get it from `IApp.XMLHelper`.

GetCustomIDwXMLDocument

Method — returns `IDwXMLDocument`

Creates an empty XML document.

Parameter	Type	Direction	Description
`Write`	`Boolean`	in	`True` for a document you intend to write, `False` for one you intend to read.

`Write` sets the document's initial `UpdateOperation` — `1` for `True`, `0` for `False` — so the `Update…` methods go the right way without further setting up. Nothing else about the document depends on it: a document asked for as readable can still be written to.

The document comes back empty, with no root node. Create one and attach it before writing anything into it.

PYTHON

```
doc = app.XMLHelper.GetCustomIDwXMLDocument(True)
root = doc.CreateNode("MySettings", "")
doc.SetStartNode(root)
```

C#

```
dynamic doc = app.XMLHelper.GetCustomIDwXMLDocument(true);
dynamic root = doc.CreateNode("MySettings", "");
doc.SetStartNode(root);
```