OpenTAP



Developer Guide

Table of Contents

Introduction **3**
- Audience 3
- Suggested Resources 3

Migrating to .NET 9 **4**
- .NET 9 Runtime Prerequisite 4

Resources **10**
- DUT 10
- Instrument 10
- Resource Management 12

Annotations **14**
- Custom Controls and Annotations 16

Attributes **17**
- Attributes Used by OpenTAP 17
- Attribute Details 18

Macro Strings **25**
- Test Steps 25
- Result Listeners 25
- Other Uses 26

Threading and Parallel Processing **27**
- Parallel Steps 27
- Deferred Processing 27
- TapThreads 29
- .NET Threads and Tasks 31

Environment Variables **32**

Introduction

This document describes the programmatic interface to OpenTAP and shows how to get started using OpenTAP for implementing test steps, instrument plugins, DUT plugins and result listeners.

Audience

This document is written for somebody who wants to develop OpenTAP plugins or integrate OpenTAP into their own applications using the C# programming language. It is not a reference manual, but rather a document that describes the principles behind OpenTAP and how to use its most important features from a programmer's perspective. If you are looking for Python developer documentation, go here .

For development, we recommend using the following software:

- Visual Studio, JetBrains Rider or Visual Studio Code.
- .NET 9 SDK (Included in the installer.)
- Windows 11, Linux (Ubuntu >22.04) or MacOS (>26.6)

  Note: If you are upgrading from an earlier version (e.g. 9.28), .NET 9 is **not** automatically installed during the upgrade. You must install .NET 9 before running `tap`. See Migrating to .NET 9 for details.

Suggested Resources

VISA driver e.g. Keysight I/O libraries for instrument communication

PathWave Test Automation

Together with OpenTAP it is recommended to use a Graphical User Interface. Keysight Technologies offers both enterprise and community licensed versions of PathWave Test Automation Developer's System that provides a highly flexible graphical user interface and code examples.

Migrating to .NET 9

With the release of **OpenTAP 9.29**, the runtime has been upgraded from **.NET Framework 4.7.2** to **.NET 9**.
This change brings significant performance improvements, better cross-platform support, and many new APIs. Unfortunately, it also introduces subtle behavior changes, and the removal of legacy APIs.

.NET 9 Runtime Prerequisite

OpenTAP 9.29 and later requires .NET 9 to be installed on the system. This issue only affects upgrades performed via `tap package install`, which does not install system-level prerequisites. If you use an installer (e.g. `.exe` or `.msi`), the .NET 9 runtime is included automatically.

If .NET 9 is missing, you will see errors indicating missing runtime libraries, for example:

```
A fatal error was encountered. The library 'hostpolicy.dll' required to execute the application was not found in
'C:\Program Files\dotnet\'.
Failed to run as a self-contained app.
```

To resolve this, download and install the .NET 9 SDK ### How the Runtime Is Selected

The .NET runtime used is determined by the **host executable** that starts the process. Plugins built against .NET Framework continue to work regardless of which runtime the host targets. For example, if you upgrade OpenTAP to 9.32 but keep Editor 9.28 (which targets .NET Framework), the Editor process still runs on .NET Framework — even though OpenTAP 9.32 is installed. Running `tap` from OpenTAP 9.29+ launches a .NET 9 process, and newer host applications (such as PathWave Test Automation) may also target .NET 9. You only need the .NET 9 runtime installed if you are using a host executable that targets it.

This guide highlights common problems and workarounds specifically related to OpenTAP, and is not meant to be comprehensive. For a comprehensive list of breaking changes, see the official documentation .

To know which version of .NET OpenTAP is using, you can check the log file. It might say:

```
; Microsoft Windows 10.0.19045X64
; .NET 9.0.7
; OpenTAP Engine 9.29.0+f68daa39 X64
```

Here it says, OpenTAP 9.29.0 running on .NET 9(.0.7) on Microsoft Windows 10, 64-bit.

Using OpenTAP 9.28, it might have said:

```
; Microsoft Windows 10.0.19045 X64
; .NET Framework 4.8.4790.0
; OpenTAP Engine 9.28.2+504225fd X64
```

Here it says, OpenTAP 9.28.2, running on .NET Framework 4.8, on Microsoft Windows 10, 64-bit.

Even though OpenTAP depends on a given .NET version, it might actually be running on a newer, but compatible runtime. For example, in 9.28, it specifies 4.7.2, but can run on 4.8. And in the same way, it could be running on .NET 10, even though it requires .NET 9.

ProcessStartInfo Defaults Changed

Symptoms

ProcessStartInfo behaves differently when starting processes, causing unexpected behavior in some applications.

Cause
Several default properties of `ProcessStartInfo` have changed in .NET 9 compared to .NET Framework 4.7.2. For example, `UseShellExecute` now defaults to `false`, which affects how processes are launched.

Solution
Explicitly set important properties like `UseShellExecute`, `RedirectStandardOutput`, or `CreateNoWindow` to ensure

consistent behavior across .NET versions.

Example

```
var startInfo = new ProcessStartInfo
{
    UseShellExecute = true,  // Explicitly set to match .NET Framework behavior
};
Process.Start(startInfo);
```

Missing System.IO.Ports API

Symptoms

```
The type or namespace name 'IO' does not exist in the namespace 'System' (are you missing an assembly reference?)
```

or

```
Could not load file or assembly 'System.IO.Ports, Version=0.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51'.
The system cannot find the file specified.
```

Cause
The `System.IO.Ports` namespace has been removed from .NET 9 and is no longer included by default.

Solution
Install the OpenTAP Serial Ports package, which provides serial port functionality compatible with .NET 9:

```
<ItemGroup>
  <OpenTapPackageReference Include="Serial Ports" Version="9.0.7" />
</ItemGroup>
```

> This package restores the `System.IO.Ports` API functionality for OpenTAP users on .NET 9.

Other Missing .NET Framework APIs

Symptoms

```
The type or namespace name 'Drawing' does not exist in the namespace 'System' (are you missing an assembly
reference?)
```

or

```
Could not load for or assembly 'name, Version=x.y.z.w, ...
```

Cause
Many APIs that were part of .NET Framework are not included by default in .NET 9.

Solution
Check if Microsoft provides a compatibility package. This is the case for most removed APIs. There is a meta package called Microsoft.Windows.Compatibility which restores many removed APIs, but there are also more targetted compatibility packages such as System.Drawing.Common which restores specific APIs. The specific packages should be preferred if possible.

> Hint: Check the Dependencies tab of the meta package for a list of all compatibility packages provided by Microsoft.

```
<ItemGroup>
  <PackageReference Include="Microsoft.Windows.Compatibility" Version="9.0.7" />
</ItemGroup>
```

> This enables use of legacy APIs such as `System.Drawing`, `System.ServiceModel`, and others.

Assembly Side-Loading Not Supported

Symptoms

Unexpected behavior of certain assemblies.

Cause

.NET Framework supports automatic side-by-side execution of strong named assemblies. This is a powerful feature, but can lead to subtle bugs and dependency issues, and has been removed in .NET 9.

In .NET Framework, attempting to load two assemblies with the same name but different versions results in two different instances of the same assembly. In .NET 9, the second load call will return a handle to the first assembly.

Solution

Side-loading is still possible, but must be manually managed using `AssemblyLoadContext`.

AppDomain Changes

Symptoms

Some `AppDomain` APIs are missing or behave differently in .NET 9, causing compatibility issues.

Cause
Several `AppDomain` APIs related to assembly isolation and unloading have been removed or altered in .NET 9. The runtime now uses `AssemblyLoadContext` for dynamic assembly loading and unloading.

Solution
Migrate code depending on `AppDomain` to use `AssemblyLoadContext` instead. `AssemblyLoadContext` is too big a topic to cover in this document. Please refer to Microsoft's documentation about AssemblyLoadContext

Global Assembly Cache (GAC) Not Supported

Symptoms

FileNotFoundException or TypeLoadException exceptions raised from code previously working in .NET Framework.

In some cases, this is because the assembly is located in the GAC (Global Assembly Cache) and that is no longer supported in .NET 9.

Cause
.NET 9 does not support GAC.

Solution

Load the assembly directly or create a GAC assembly resolver.

Usually those assemblies are located inside a folder in `C:\Windows\assembly\GAC_MSIL\[assembly-name]\[version]\[assembly-name].dll`.

The simplest solution is to add a custom assembly resolver for those assemblies. This can be done via `AppDomain.Current.ResolveAssembly`. That event will be called if it was otherwise not possible to resolve the assembly.

If that solves the issue, instruct the users to install relevant software for the assembly to be located there. For example, the assembly may follow an IVI driver.

Alternatively, you can use the DLL as part of your solution by simply copying it into the OpenTAP installation folder. Check if the license permits that before distributing it.

CET Enabled by Default

Symptoms

Application crashes when calling functions from shared libraries.

Cause

.NET 9 enables CET by default. This is a security feature which sets some limitations to how a low level

library can manipulate return addresses on the stack, and can cause issues in low level libraries using custom exception handling. For more information, see CET supported by default and Shadow Stack .

Solution

For compatibility reasons, OpenTAP disables CET, so if you are starting the process with `tap.exe` or `Editor.exe`, no steps are needed.

If you are in control of the shared library, you can update it to ensure it does not violate shadow stack protection rules. This is the preferred solution since it is the most secure solution, and it solves the problem regardless of how the library is used.

If you are in control of the executable with the problem, you can disable CET by setting a build property:

```
<PropertyGroup>
  <!-- Disable Control-flow Enforcement Technology -->
  <CETCompat>false</CETCompat>
</PropertyGroup>
```

If you are not in control of either the library or the binary, you can disable stack protections for your system or for a particular process.

BinaryFormatter Removed

Symptoms

Code using `BinaryFormatter` for serialization fails to compile or throws runtime errors.

Cause
`BinaryFormatter` has been fully removed in .NET 9 due to long-standing security concerns.

Solution
Migrate to safer serialization alternatives such as `System.Text.Json`.

Assembly.LoadWithPartialName Removed

Symptoms

Code using `Assembly.LoadWithPartialName` fails to compile or throws runtime errors.

Cause
`Assembly.LoadWithPartialName` was deprecated and is removed in .NET 9 due to its unreliable behavior.

Solution
Use `Assembly.Load` or `AssemblyLoadContext` with the full assembly name or path instead.

Thread.Suspend and Thread.Resume Removed

Symptoms

Calls to `Thread.Suspend` or `Thread.Resume` result in compilation errors or runtime exceptions in .NET 9.

Cause
These APIs have been removed because there is no way to use them safely.

Solution
Refactor code to use safer synchronization primitives such as the `lock` keyword, `Monitor`, `Mutex`, `Semaphore` or `Event` for synchronization.

Thread.Abort Removed

Symptoms

Calls to `Thread.Abort` cause compilation errors or runtime failures in .NET 9.

Cause

`Thread.Abort` has been removed due to its unsafe nature and potential to leave application state inconsistent.

Solution
Use cooperative cancellation patterns with `CancellationToken` to gracefully stop threads.

Registry API Changes

Symptoms

Code using certain Windows-specific Registry APIs fails to compile or run in .NET 9.

Cause
Some Registry APIs have been removed or now require additional NuGet packages (like `Microsoft.Win32.Registry`) to be referenced explicitly.

Solution
Add the `Microsoft.Win32.Registry` package to your project and update your code to use the supported APIs.

```
<ItemGroup>
  <PackageReference Include="Microsoft.Win32.Registry" Version="5.0.0" />
</ItemGroup>
```

System.IO.FileInfo.Length Exception on Missing File

Symptoms

Accessing `FileInfo.Length` throws a `FileNotFoundException` if the file does not exist.

Cause
In .NET 9, `FileInfo.Length` no longer returns 0 for nonexistent files and instead throws an exception.

Solution
Check `FileInfo.Exists` before accessing `FileInfo.Length`.

TLS 1.0/1.1 Disabled by Default

Symptoms

Connections using TLS 1.0 or 1.1 fail to establish, causing communication errors.

Cause
.NET 9 disables TLS 1.0 and 1.1 by default for improved security. Only TLS 1.2 and higher are enabled.

Solution
Update your applications and servers to use TLS 1.2 or higher. Explicitly configure `SslProtocols` if needed.

Example

```
var handler = new HttpClientHandler
{
    SslProtocols = SslProtocols.Tls12 | SslProtocols.Tls13
};

using var client = new HttpClient(handler);
```

HttpWebRequest & WebClient Deprecated

Symptoms

Code using `HttpWebRequest` or `WebClient` issues warnings or may not behave optimally in .NET 9.

Cause
`HttpWebRequest` and `WebClient` are deprecated in favor of the newer, more flexible `HttpClient` API.

Solution

Migrate to using `HttpClient` for HTTP operations.

CodeDomProvider Removed for Dynamic Compilation

Symptoms

Code using `CodeDomProvider` or `CSharpCodeProvider` to compile code at runtime fails to compile or run in .NET 9.

Cause
`CodeDomProvider` and related runtime compilation APIs have been removed in .NET 9.

Solution
Migrate to the Roslyn compiler APIs (`Microsoft.CodeAnalysis.CSharp`) for runtime code compilation.

Remoting Removed

Symptoms

Code using .NET Remoting APIs fails to compile or run after upgrading to .NET 9. This affects the following APIs: * MarshalByRefObject.GetLifetimeService * MarshalByRefObject.InitializeLifetimeService

Cause
.NET Remoting has been completely removed in .NET 9. These APIs are no longer supported or available.

Solution
There is no direct replacement. You must migrate to a different communication technology.

Resources

DUT

To develop a *device under test* (DUT) plugin, extend (or inherit from) the **DUT** class, which itself extends the **Resource** class. The *Open* and *Close* methods MUST be implemented:

- The **Open** method is called before the test plan starts, and must execute successfully. The Open method should include any code necessary to configure the DUT prior to testing. All open methods on all classes that extend Resource are called in parallel, and prior to any use of the DUT in a test step.
- The **Close** method is called after the test plan is done. The Close method should include any code necessary to configure the DUT to a safe condition after testing. The Close method will also be called if testing is halted early. All close methods are called in parallel, and after any use of the DUT in a test step.

The DUT template generated by the Visual Studio class wizard includes minimal implementations of these calls.

Developers should add appropriate properties and methods to the plugin code to allow:

- Configuration of the DUT during setup. The DUT base class already has defined string properties for **ID** and **Comment**.
- Control of the DUT during the execution of test steps.

For examples of DUT plugin development, see:

- `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\InstrumentsAndDuts`

Instrument

Developing an instrument plugin is done by extending either the:

- **Instrument class** (which extends *Resource*), or
- **ScpiInstrument** base class (which extends *Instrument*)

It is recommended to use ScpiInstrument over the Instrument class when possible.

Instrument plugins must implement the **Open** and **Close** methods:

- The **Open** method is called before the test plan starts, and must execute successfully. The Open method should include any code necessary to configure the instrument prior to testing. All open methods on all classes that extend Resource are called in parallel, and prior to any use of the instrument in a test step.
- The **Close** method is called after the test plan is done. The Close method should include any code necessary to configure the instrument to a safe condition after testing. The Close method will also be called if testing is halted early. All close methods are called in parallel, and after any use of the instrument in a test step.

Developers should add appropriate properties to the plugin code to allow:

- Configuration of the instrument during setup. The Instrument base class has no predefined properties. The ScpiInstrument base class has a string property that represents the **VisaAddress** (see SCPI Instruments below).
- Control of the instrument during the execution of test steps.

Similar to DUTs, instruments must be preconfigured via the **Bench** menu choice, and tests will use the first instrument found that matches the type they need. For instrument plugin development examples, see the files in:

- `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\InstrumentsAndDuts`

SCPI Instruments

OpenTAP provides a number of utilities for using SCPI instruments and SCPI in general. The **ScpiInstrument** base class:

- Has properties and methods useful for controlling SCPI based instruments
- Includes a predefined VisaAddress property
- Requires Open and Close logic

Important methods and properties here include:

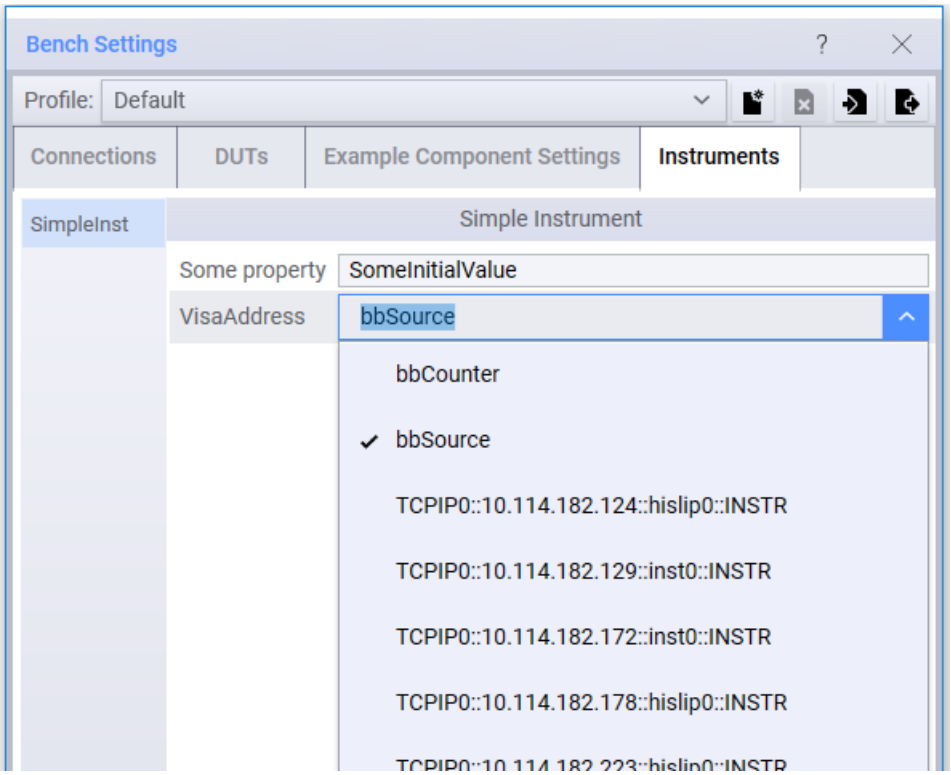
- `ScpiCommand`, which sends a command.
- `ScpiQuery`, which sends the query and returns the results.
- `VisaAddress`, which specifies the Visa address of the instrument.
- `ScpiQueryBlock<T>`, which sends the block query, and parses the binary block as an array of type T. All numeric types except Decimal are supported.

The SCPI *attribute* is used to identify a method or enumeration value that can be handled by the SCPI class.

For an example, see:

- `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\ScpiAttributeExample.cs`

The example below shows how the VisaAddress property for a SCPI instrument is automatically populated with values retrieved from VISA:



In some cases, VISA device discovery can cause issues. This mostly occurs with VISA vendors who do not support it. In that case, the behavior can be disabled by setting the environment variable OPENTAP_NO_VISA_DISCOVERY to “true”.

Raw IO

In rare cases, it may be necessary to resort to raw I/O reads and writes. This should generally be avoided because an incomplete read might interfere with the results of future queries. However, in situations such as streaming indeterminate-length data from the instrument, the use of raw I/O may become necessary. A low-level API for this is available. The raw IO is located inside an explicit interface implementation, to access it do the following:

```
ScpiInstrument instrument = /* ... */;
IScpiIO io = ((IScpiInstrument) instrument).IO;
```

This provides access to raw reads and writes: - **Read**, which reads data from the instrument into a user-provided byte buffer. The number of bytes read is placed in an **out** parameter. - **Write**, which writes data from a user-provided byte buffer to the instrument.

Resource Management

OpenTAP comes with the **ResourceOpen** attribute that is used to control how and if referenced resources are opened. This attribute is attached to a resource property and it has three modes:

- **Resource Open Before** - This mode indicates that the resources pointed to by this property will be opened in sequence, so any referenced resources are open before `Open()` and until after `Close()`. This is the default behavior.
- **Resource Open Parallel** - This mode indicates that a resource property on a resource can be opened in parallel with the resource itself.
- **Resource Open Ignore** - This mode indicates that a resource referenced by this property will not be opened or closed.

For an examples of Resource Management, see:

- `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\ResourceOpenAttributeExample`

Both examples have a base instrument that depends on a sub instrument , each having its own respective `resource open` attribute.

Resource Open

Upon running *ResourceOpenBeforeAttributeExample* test step in a test plan, the sub instrument will invoke its `Open()` method first, before the base instrument's `Open()` method. A delay is added to demonstrate the connection status of sub instrument connecting first.

When the test plan stops, the base instrument's `Close()` method will be invoked and disconnected. A delay is added to demonstrate that the sub instrument will invoke its `Close()` method after the base instrument has been disconnected.

Resource Parallel

Upon running *ResourceOpenParallelAttributeExample* test step in a test plan, the base instrument will open in parallel with the sub instrument. To demonstrate that the sub instrument is being invoked to open, a delay is added to delay the opening of the connection of the base instrument.

When the test plan stops, the sub instrument will disconnect and close its connection in parallel with the base instrument. Subsequently, a delay is added to demonstrate that the base instrument is being disconnected.

Resource Ignore

Resources with the `resource ignore` attribute are ignored by the test plan during execution. Hence, these resources will not invoke their `Open()` nor their `Close()` methods.

Resource Strategy

There is a setting that can affect the open and close sequence of resources. Under **Engine settings**, change resource strategy from **Default Resource Manager** to **Short Lived Connections**.

```
`09:14:45.593  TestPlan     ----------------------------------------------------------------`
`09:14:45.594  TestPlan     Starting TestPlan 'Untitled' on 10/13/2020 09:14:45, 1 of 1 TestSteps enabled.`
`09:14:45.606  TwoPortInst  Opening TwoPortInstrument.`
`09:14:45.606  TwoPortInst  Resource "TwoPortInst" opened. [65.4 us]`
`09:14:47.606  INST         Opening Prior Instrument`
`09:14:47.606  INST         PriorSubInstr connected: True`
`09:14:47.606  INST         IgnoreSubInstr connected: False`
`09:14:47.606  INST         Resource "INST" opened. [2.00 s]`
`09:14:47.606  TestPlan     "Resource Open Before Example" started.`
`09:14:48.607  INST         Closing Prior Instrument`
`09:14:50.607  INST         PriorSubInstr connected: True`
```

```
`09:14:50.607  INST         IgnoreSubInstr connected: False`
`09:14:50.607  INST         Resource "INST" closed. [3.00 s]`
`09:14:50.607  TwoPortInst  Closing TwoPortInstrument.`
`09:14:50.607  TwoPortInst  Resource "TwoPortInst" closed. [50.1 us]`
`09:14:50.607  TestPlan     "Resource Open Before Example" completed. [5.00 s]`
`09:14:50.613  Summary      ----- Summary of test plan started 10/13/2020 09:14:45 -----`
`09:14:50.613  Summary       Resource Open Before Example                    5.00 s`
`09:14:50.613  Summary      ------------------------------------------------------------`
`09:14:50.613  Summary      -------- Test plan completed successfully in 5.01 s --------`
```

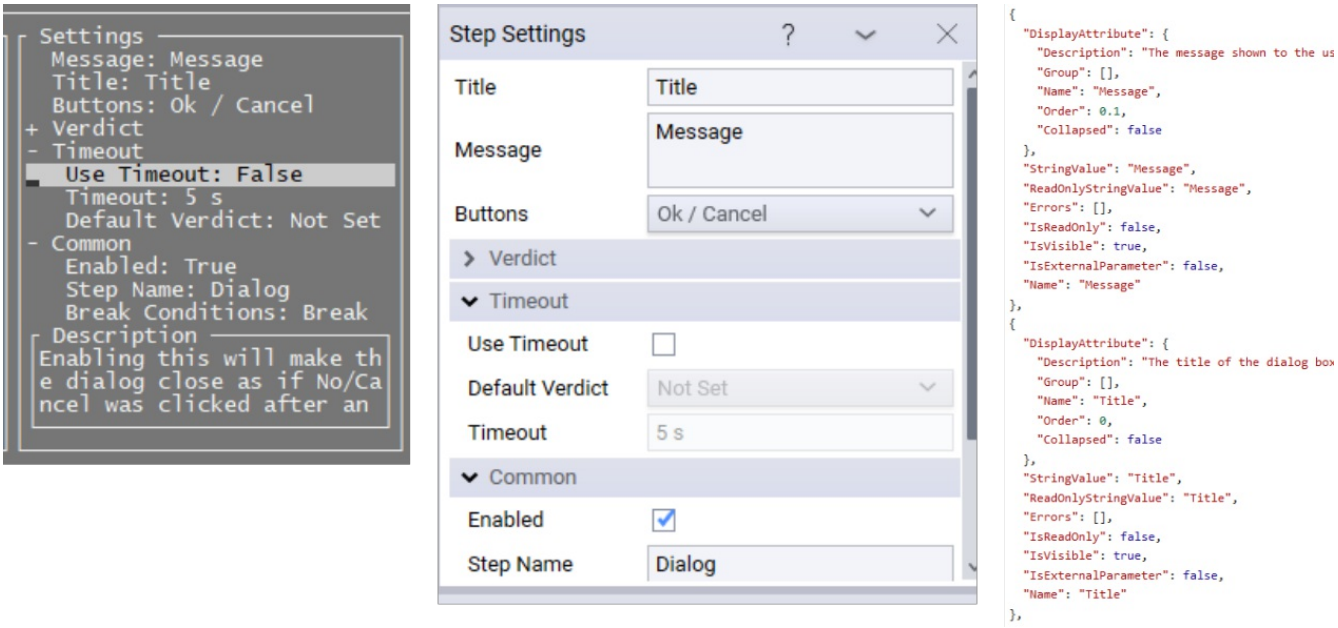
The above log demonstrates that **Short Lived Connections** always close before the test plan execution ends. However, using the **Default Resource Manager**, the resource connections will be closed only after the test plan execution has ended. Using short lived connections result in more efficient resource management since connections are closed when no longer needed by the test plan.

Annotations

The annotations system is a flexible way to describe object interactions between users and OpenTAP. It provides a way to dynamically generate a UI model, that can be used in multiple different GUI application to create a unified user experience.

Common elements of user interactions are for example text boxes, check-boxes and drop-downs. These elements are abstracted in the annotation system, without explicitly stating which kinds should be used. This is not the easiest way to create user interfaces, but it ensures the same interactivity across User Interfaces like Graphics User Interfaces, Remoting APIs and text based user interfaces. It is a powerful abstraction because it allows defining new user interactions without resorting to writing any specific GUI code.

An 'annotation' describes an aspect of an object in relation to known user interactions. One such aspect could be e.g the name of a property or the fact that its value should be selected from a list of available values. Objects are described by multiple of such annotations. A model of an object or property is abstracted into the `AnnotationCollection` class. This class is a collection of annotations for the object, describing it in all currently available ways. Additional classes can be added to the annotation by plugins called 'annotators'.



Annotated test step settings for a text/terminal based, a WPF based GUI and a JSON WEB API. The views are based on data from the same type of Test Step. Each view has the same properties and the same annotations, but different technologies are being used to show them.

The annotation system has two sides to it, the User Interface and the Plugin side. This section will mostly focus on the plugin side of this. For a complete example of how to implement a User Interface using the annotation system the open-source text based user interface can be used as a reference. It is located at https://github.com/StefanHolst/opentap-tui .

For plugin developers the annotation system can be used to extend the number of types that the user can interact with in the GUI.

For example, for a given type, let's say an IPAddress, a "String Value Annotation" can be implemented to enable editing an IP in a text box. If it is also wanted to have a drop-down of pre-selected values, the "Suggested Values Annotation" can be used. For the concrete example see `IPAnnotation.cs` in the SDK Examples.

GUIs are looking for annotation interface implementations to know if they can display a given annotation to the user. For example, if a GUI application can find String Value Annotation in the list of annotations for given property, then it can display that property as a text box.

The following are the most commonly implemented types of annotations:

Interface	Name	Description
IStringValueAnnotation	String Value Annotation	Used for properties that can be edited as text. Most commonly a single line of text. This is used for numbers, text, dates, time spans, URLs.
IStringReadOnlyValueAnnotation	Read-only String Value Annotation	Used for items that maybe cannot be edited as a string, but the current value can be read as a string. This is for example useful for things that might appear in a drop-down.
IAvailableValuesAnnotation	Available Values Annotation	Used for editing properties based on a selection of available values. This is commonly used for DUTs, instruments and enums.
ISuggestedValuesAnnotation	Suggested Values Annotation	Used along with the IStringValueAnnotation. Allows the user to select from a list of suggested values as well as inserting a custom value.
IMultiSelect	Multi Select Annotation	Usage is very similar to Available values annotation, but multiple values can be selected simultaneously. This annotation must also appear with an Available Values Annotation as it only specfies how to select multiple items and not how to list them.
IMemberAnnotation	Member Annotation	Provides reflection information for a given member this is often used by the code annotating properties, but not implemented directly.
IDisplayAnnotation	Display Annotation	Attribute used to describe user input properties and generic classes and properties. This is commonly used for describing user input interfaces and classes and is also an IAnnotation.
DisplayAttribute	Display Attribute Annotation	Attribute used to describe classes and properties. This is commonly used in plugin code and is also an IDisplayAnnotation.
HelpLinkAttribute	Help Link Attribute Annotation	Provides a help link for a given class or property.
Less Used Annotations		
IStringExampleValueAnnotation	String Example Value Annotation	Used to provide extra information about a string. This is often used along with macros to provide the expanded macro string to the user.
IErrorAnnotation	Error Annotation	Used to provide data errors to the user interface. Validation errors are implemented this way. Custom new types of errors can be added by extending it.
IAccessAnnotation	Access Annotation	Used to define whether the user has write or view access to a property.
IEnabledAnnotation	Enabled Annotation	Used to specify whether a property should be enabled
IBasicCollectionAnnotation	Basic Collection Annotation	Used to specify that something should be edited as a collection of items. This normally means it should be shown as a data grid.
IFixedSizeCollectionAnnotation	Fixed Size Collection Annotation	Can be used to specify if a collection has fixed size. Meaning if elements can be added or removed.
IValueDescriptionAnnotation	Value Description Annotation	Can be used to provide a more thorough description of the value of an object. These values usually show up in tooltips.
IMembersAnnotation	Members Annotation	Used to specify that an object has members that can also be edited. Each of the members are annotated themselves. This interface is very complicated to

Interface	Name	Description
		implement. If dynamic members for a given type need to be implemented, it would normally be better to specify a custom ITypeData/ITypeDataProvider.

Some of these annotation types have a corresponding ‘proxy’ interface. The proxy interface is there to wrap the simpler non-proxy variants in a manner that is usable from the user interface. They don’t need to be considered unless developing a user interface.

The static method `AnnotationCollection.Annotate` is used to build the AnnotationCollection object. A plugin type called an “annotator” (`IAnnotator`) is the definition of classes that can be used to provide annotations for a given object. To use custom annotations an `IAnnotator` must be implemented to specify where the custom annotations should be inserted. A number of annotators exist and they need to be applied in a specific order, therefore the `IAnnotator.Priority` Property is used to control the order in which annotations are being applied. Normally, this value should just be set to ‘1’.

Custom Controls and Annotations

Various GUI implementations support writing new custom controls. To support this it might also be necessary to create custom annotation types. Both things are possible, but it is generally discouraged, since existing GUIs cannot know about these new annotations. If new annotation types or new types of controls are needed we encourage starting a discussion on the OpenTAP repository at https://github.com/opentap/opentap . This way, we can ensure that plugins are broadly supported across multiple user interfaces.

Attributes

Attributes are standard parts of C# and are used extensively throughout .NET. They have constructors (just like classes) with different signatures, each with required and optional parameters. For more information on attributes, refer to the MSDN C# documentation.

For OpenTAP, *type* information is not enough to fully describe what is needed from a property or class. For this reason, attributes are a convenient way to specify additional information. OpenTAP, the GUI Editor and CLI use reflection (which allows interrogation of attributes) extensively. Some attributes have already been shown in code samples in this document.

Attributes Used by OpenTAP

OpenTAP uses the following attributes:

Attribute name	Description
AllowAnyChild	Used on *step class* to allow children of any type to be added.
AllowAsChildIn	Used on *step* to allow step to be inserted into a specific step type.
AllowChildrenOfType	Used on *step* to allow any children of a specific type to be added.
AvailableValues	Allows the user to select from items in a list. The list can be dynamically changed at run-time.
ColumnDisplayName	Indicates a property could be displayed as a column in the test plan grid.
CommandLineArgument	Used on a property in a class that implements the *ICliAction* interface, to add a command line argument/switch to the action (e.g. “–verbose”).
DeserializeOrder	Can be used to control the order in which properties are deserialized.
DirectoryPath	Indicates a string property is a folder path.
Display	Expresses how a property is shown and sorted. Can also be used to group properties.
EnabledIf	Disables some controls under certain conditions.
ExternalParameter	Indicates that a property on a TestStep (a step setting) should be a External Parameter by default when added to a test plan.
FilePath	Indicates a string property is a file path.
Flags	Indicates the values of an enumeration represents a bitmask.
HandlesType	Indicates a IPropGridControlProvider can handle a certain type. Used by advanced programmers who are modifying the GUI editor internals.
HelpLink	Defines the help link for a class or property.
Layout	Used to specify the desired layout of the element in the user interface.
MacroPath	Indicates a setting should use MacroPath values, such as <Name> and %Temp%.
MetaData	A *property* marked by this attribute becomes metadata and will be provided to all result listeners. If a resource or a component setting is used with this attribute (and *Allow Metadata Dialog* is enabled), a dialog prompts the user. This works for both the GUI Editor and the OpenTAP CLI.
Output	Indicates a test step property is an output variable.

Attribute name	Description
ResourceOpen	Used to control how and if referenced resources are opened. This attribute is attached to a resource property. Three modes are available, Before, InParallel and Ignore. Refer to the API documentation for more information.
ResultListenerIgnore	Indicates a property that should not be published to ResultListeners.
Scpi	Identifies a method or enumeration value that can be handled by the SCPI class.
SettingsGroup	Indicates that component settings belong to a settings group (e.g. “Bench” for bench settings).
SuggestedValues	Marks the property value can be selected from a list in the OpenTAP Editor. Points to another property that contains the list of suggested values.
TimeSpanFormat	Attribute applicable to a property of type ‘TimeSpan’ to display the property value in a human readable format in the user interface.
UnnamedCommandLineArgument	When used on a property in a class that implements the *ICliAction* interface, the property becomes an unnamed parameter to the command line argument.
Unit	Indicates a unit displayed with the setting values. Multiple options exist.
VisaAddress	Indicates a property that represents a VISA address. The editor will be populated with addresses from all available instruments.
XmlIgnore	Indicates that a property should not be serialized.

For attribute usage examples, see the files in:

- `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes`

Some of the commonly used attributes are described in the following sections. For more details on the attributes see OpenTapApiReference.chm.

Attribute Details

Display

The **Display** attribute is the most commonly used OpenTAP attribute. This attribute:

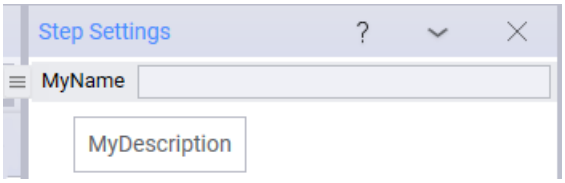
- Can be applied to class names (impacting the appearance in dialogs, such as the Add New Step dialog), or to properties (impacting appearance in the Step Settings Panel).
- Has the following signature in its constructor:

```
(string Name, string Description = "", string Group = null, double Order = 0D, bool Collapsed = false, string[]
Groups = null)
```

- Requires the **Name** parameter. All the other parameters are optional.
- Supports a **Group** or **Groups** of parameters to enable you to organize the presentation of the items in the Test Automation Editor.

The parameters are ordered starting with the most frequently used parameters first. The following examples show example code and the resulting Editor appearance:

```
// Defining the name and description.
[Display("MyName", "MyDescription")]
public string NameAndDescription { get; set; }
```



See the examples in `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes` for different uses of the Display attribute.

Display has the following parameters:

Attribute	Required	Description
Name	Required	The name displayed in the Editor. If the Display attribute is not used, the **property name** is used in the Editor.
Description	Optional	Text displayed in tools tips, dialogs and editors in the Editor.
Group/Groups	Optional	Specifies an item's group. Use **Group** if the item is in a one-level hierarchy or **Groups** if the item is in a hierarchy with two or more levels. The hierarchy is specified by the left-to-right order of the string array. Use either Group or Groups; do not use both. Groups is preferred. Groups are ordered according to the average order value of their child items. For test steps, the top-level group is always ordered alphabetically. Syntax: `Groups: new[] { "Group" , "Subgroup" }`
Order	Optional	Specifies the display order for an item. Note that **Order** is supported for settings and properties, such as test step settings, DUT settings, and instrument settings. It does not support types: test steps, DUTs, instruments. These items are ordered alphabetically, with groups appearing before ungrouped items. Order is of type double, and can be negative. Order's behavior matches the Microsoft behavior of the *Display.Order* attribute. If order is not specified, a default value of -10,000 is assumed. Items (ungrouped or within a group) are ranked so that items with lower order values precede those with higher values; alphabetically if order values are equal or not specified. To avoid confusion, we recommend that you set the order value for ungrouped items to negative values so that they appear at the top and Grouped items to a small range of values to avoid conflicts with other items (potentially specified in base classes). For example, if *Item A* has order = 100, and *Item B* has order = 50, *Item B* is ranked first.

Embed Properties Attribute

The EmbedPropertiesAttribute can be used to embed the members of one object into the owner object. This hides the embedded object from reflection, but shows the embedded objects members instead. This can be used to let objects share common settings and code without using inheritance.

When properties are embedded, the reflection engine will use the static type data on the test step to figure out which new properties will be added to the test step. This means that if the actual object value is a subclass of the type, then additional properties on the sub class will not show up.

When using this attribute, validation rules can be added to the embedded object by inheriting from ValidatingObject.

See the example EmbedPropertiesAttributeExample for more information about how it can be used.

EnabledIf Attribute

The **EnabledIf** attribute disables or enables settings properties based on other settings (or other

properties) of the same object. The decorated settings reference another property of an object by name, and its value is compared to the value specified in an argument. Properties that are not settings can also be specified, which allows the implementation of more complex behaviors.

For test steps, if instrument, DUTs or other resource properties are disabled, the resources will not be opened when the test plan starts, However, if another step needs them they will still be opened.

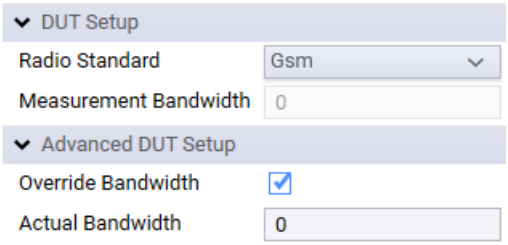
The **HideIfDisabled** optional parameter of EnabledIf makes it possible to hide settings when they are disabled. This is useful to hide irrelevant information from the user.

Multiple EnabledIf statements can be used at the same time. In this case all of them must be enabled (following the logical *AND* behavior) to make the setting enabled. If another behavior is wanted, an extra property (hidden to the user) can be created and referenced to implement another logic. In interaction with HideIfDisabled, the enabling property of that specific EnabledIf attribute must return false for the property to be hidden.

In the following code, BandwidthOverride is enabled when **Radio Standard** = GSM.

```
public class EnabledIfExample : TestStep
{
    #region Settings

    // Radio Standard to set DUT to transmit.
    [Display("Radio Standard", Group: "DUT Setup", Order: 1)]
    public RadioStandard Standard { get; set; }

    // This setting is only used when Standard == LTE || Standard == HCDHA.
    [Display("Measurement Bandwidth", Group: "DUT Setup", Order: 2.1)]
    [EnabledIf("Standard", RadioStandard.Lte, RadioStandard.Wcdma)]
    public double Bandwidth { get; set; }

    // Only enabled when the Standard is set to GSM.
    [Display("Override Bandwidth", Group: "Advanced DUT Setup", Order: 3.1)]
    [EnabledIf("Standard", RadioStandard.Gsm, HideIfDisabled = true)]
    public bool BandwidthOverride { get; set; }

    // Only enabled when both Standard = GSM, and BandwidthOverride property is enabled.
    [Display("Override Bandwidth", Group: "Advanced DUT Setup", Order: 3.1)]
    [EnabledIf("Standard", RadioStandard.Gsm, HideIfDisabled = true)]
    [EnabledIf("BandwidthOverride", true, HideIfDisabled = true)]
    public double ActualBandwidth { get; set; }

    #endregion Settings
}
```

When **Radio Standard** is set to GSM in the step settings, both **Override Bandwidth** options are then displayed:



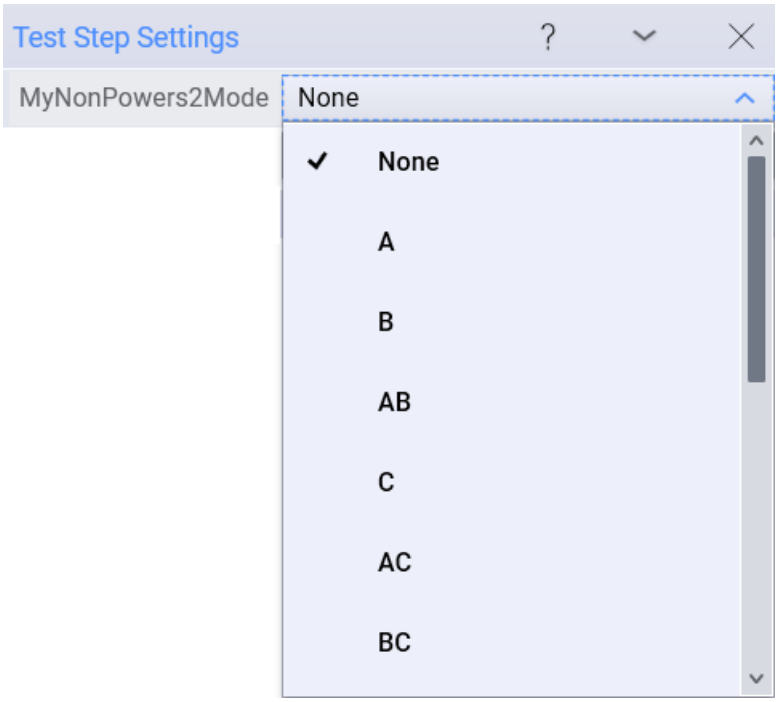
For an example, see
`TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\EnabledIfAttributeExample.cs`.

Flags Attribute

The **Flags** attribute is a C# attribute used with enumerations. This attribute indicates that an enumeration can be treated as a *bit field* (meaning, elements can be combined by bitwise OR operation). The enumeration constants can be defined in powers of two (for example 1, 2, 4, ...).

Using the Flags attribute results in a multiple select in the Editor, as shown below: These enumeration

constants are defined in ascending values from zero.



Enumeration constant with value zero is handled specially by unselecting all enumeration constants except that with zero. Enumeration constant with value not of powers of two will be selected accordingly to its bitmask representative. (eg. enumeration constant with value 3 will be selected togther with value 1 and 2).

FilePath and DirectoryPath Attributes

The FilePath and DirectoryPath attributes can be used on a string-type property to indicate the string is a file or a folder system path. When this attribute is present, the Editor displays a browse button allowing the user to choose a file or folder. These attributes can be used as follows:

```
[FilePath]
public string MyFilePath { get; set; }
```

This results in the following user control in the Editor:



The DirectoryPath attribute works the same as the FilePath attribute, but in the place of a file browse dialog, a directory browse dialog opens when the browse (‘…’) button is clicked.

The FilePath attribute supports specifying file type as well.

It can be done by writing the file extension as such:

```
[FilePath(FilePathAttribute.BehaviorChoice.Open, "csv");
```

Or it can be done by specifying a more advanced filter expression as shown below.

```
[FilePath(FilePathAttribute.BehaviorChoice.Open, "Comma Separated Files (*.csv)|*.csv| Tab Separated Files (*.tsv) |
*.tsv| All Files | *.*")]
```

The syntax works as follows: `[Name_1] | [file extensions 1] | [Name_2] | [file extensions 2] ...`

Each filter comes in pairs of two, a name and a list of extensions. The name of a filter can be anything, excluding the ‘|’ character. It normally contains the name of all the included file extensions, for example “Image Files (*.png, *.jpg)”. The file extensions is normally not seen by the user, but should contain all

the supported file extensions as a semi-colon separated list. Lastly, it is common practice to include the 'AllFiles | *.*' part, which makes it possible for the user to override the known filters and manually select any kind of file.

In addition to string properties, the FilePath attribute can also be applied to `List<string>` types. For example:

```
[FilePath]
public List<string> FilePaths {get;set;} = new List<string>();
```

When used this way, multiple files can be selected simultaneously, simplifying the process of choosing several files from the same directory. However, the user experience may vary across different platforms.

Submit Attribute

This attribute is used only for objects used together with UserInput.Request. It is used to mark the property that finalizes the input. For example this could be used with an enum to add an OK/Cancel button, that closes the dialog when clicked. See the example in UserInputExample.cs for an example of how to use it.

HelpLink Attribute

The HelpLink attribute can be used to provide help documentation for properties or classes. In user interfaces, this will often result in a `?` button showing up near the item.

The content of the HelpLink attribute can be a documentation reference, a link to a website, a PDF file or a CHM file.

Here are some examples:

```
// Refer to a topic in the package documentation.
[HelpLink("MyPackage/My Topic")]
public class MyStepWithHelpLinks : TestStep
{
    // Refer to a section in a topic in the package documentation.
    [HelpLink("MyPackage/My Topic#Configuring a Setting")]
    public string ASetting { get; set; }

    // Refer to an external documentation source.
    [HelpLink("https://mydoc.example.com/another_topic.html")]
    public string AnotherSetting { get; set; }
}
```

For more information about package documentation, see Package Documentation .

Layout Attribute

LayoutAttribute is used to control how settings are arranged in graphical user interfaces. It can be used to control the height, width and positioning of settings elements. Use this with the Submit attribute to create a dialog with options like OK/Cancel on the bottom. See UserInputExample.cs for an example.

MetaData Attribute

Metadata is a set of data that describes and gives information about other data. The Metadata attribute marks a property as metadata.

OpenTAP can prompt the user for metadata. Two requirements must be met:

- The MetaData attribute is used and the promptUser parameter is set to *true*
- The *Allow Metadata Dialog* property in **Settings > Engine**, is set to *true*

If both requirements are met, a dialog (in the Editor) or prompt(in OpenTAP CLI) will appear on each test plan run to ask the user for the appropriate values. This works for both the Editor and the OpenTAP CLI. An example of where metadata might be useful is when testing multiple DUTs in a row and the serial number must be typed in manually.

Values captured as metadata are provided to all the result listeners, and can be used in the macro system. See SimpleDut.cs for an example of the use of the MetaData attribute.

Unit Attribute

The `Unit` attribute specifies the unit of measurement for a setting. In the OpenTAP Editor, the unit is displayed after the value, separated by a space (e.g., `10 V`). Compound units (e.g., watt-hours) should be hyphenated: `Watt-hours`, `Newton-meters`, etc.

When enabled, engineering prefixes are automatically applied to improve readability. For example:

- `0.001 s` is displayed as `1 ms`
- `1,000,000 Hz` is displayed as `1 MHz`

Example:

```
[Unit("Hz", UseEngineeringPrefix: true)]
public double Frequency{get;set;} = 2.1e9 // displayed as "2.1 GHz"
```

String Formatting with StringFormat

You can optionally specify how values are formatted using the `StringFormat` parameter. This is particularly useful for displaying integers in hexadecimal format.

When working with hexadecimal values, type 0x to enter a hexadecimal value and omit 0x to enter a regular decimal value.

It is generally recommended to use the unsigned integer types when you want to enter values as hexadecimal.

Hexadecimal Formatting Options:

Format	Description	Example Output
`"X"`	Uppercase hexadecimal (use `"x"` for lowercase).	`10AF`
`"0x"`	Hexadecimal with `0x` prefix.	`0x10AF`
`"0x8"`	Hexadecimal with `0x` prefix and zero-padded to 8 characters.	`0x000010AF`
`"0x16"`	Hexadecimal with `0x` prefix and zero-padded to 16 characters. Useful for 64-bit integers.	`0x00000000000010AF`

Example:

```
[Unit("", StringFormat: "0x8")]
public uint IntegerValue { get; set; } = 0xA; // Displayed as: "0x0000000A"
```

Opting out of range simplification with UseRanges

This only affects sequences of elements such as Lists or Arrays. Range simplification is normally applied when you write a range of values, e.g “1,2,3,4,5”. This will get simplified to “1 : 5”.

You can opt out of that behavior using the UseRanges property as in the below example

Example:

```
[Unit("Hz", true, UseRanges: false)]
public double[] DoubleArray { get; set; } = [1,2,3,4,5]; // displayed as "1, 2, 3, 4, 5".
```

Additional Notes

- If no unit is required, use an empty string (`""`) for the `Unit`.

- The `StringFormat` works similarly to standard .NET format strings.

For a complete working example, refer to:

`TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\UnitAttributeExample.cs`

XmlIgnore Attribute

The XmlIgnore attribute indicates that a setting should not be serialized. If XmlIgnore is set for a property, the property will not show up in the Editor. If you want to NOT serialize the setting AND show it in the Editor, then use the Browsable(true) attribute, as shown below:

```
// Editable property not serialized to XML
[Browsable(true)]
[XmlIgnore]
public double NotSerializedVisible { get; set; }
```

Properties that represent instrument settings (like the one below) should not be serialized as they will result in run-time errors:

```
[XmlIgnore]
public double Power
{
    set; { ScpiCommand(":SOURce:POWer:LEVel:IMMediate:AMPLitude {0}", value) }
    get; { return ScpiQuery<double>(":SOURce:POWer:LEVel:IMMediate:AMPLitude?"); }
}
```

Macro Strings

Sometimes certain elements need customizable text that can dynamically change depending on circumstances. These are known as macros and are identifiable by the use of the < and > symbols in text. Macros can be expanded by the plugin developer to use other macros with different values depending on the context. The class used for macro string properties is called `OpenTAP.MacroString`.

One example is the `<Date>` macro that is available to use in many Result Listeners, like the log or the CSV result listeners. Another example is the `<Verdict>` macro. These are both examples of macros that can be inserted into the file name of a log or CSV file like so: `Results/<Date>-<Verdict>.txt`. If you insert `<Date>` in the file name the macro will be replaced by the start date and time of the test plan execution.

When used with the MetaData attribute, a property of the ComponentSettings can be used to define a new macro. For example, all DUTs have an ID property that has been marked with the attribute `[MetaData("DUT ID")]`. This means that you can put `<DUT ID>` into the file path of a Text Log Listener to include the DUT ID in the log file's name.

In addition to macros using `<>`, environment variables such as `%USERPROFILE%` will also be expanded.

There are a few different contexts in which macro strings can be used.

Test Steps

MacroStrings can be used in test steps. In this context the following macros are available:

- `<Date>`: The start date of the test plan execution
- `<TestPlanDir>`: The directory of the currently executing test plan
- MetaData attribute: Defines macro properties on parent test steps

Verdict is not available as a macro in the case of test steps, because at the time of execution the step does not yet have a verdict. However, it can be manually added by the developer if needed. In this case it is up to the plugin developer to provide documentation.

Below is an example of MacroString used with the `[FilePath]` attribute in a test step. This attribute provides the information that the text represents: a file path. In the GUI Editor this results in the `"..."` browse button being shown next to the text box.

```
public class MyTestStep: TestStep
{

  [FilePath] // A MacroString that is also a file path.
  public MacroString Filename { get; set; }

  public MyTestStep()
  {
    // 'this' useful for TestStep instances.
    // otherwise a MacroString can be created without constructor arguments.
    Filename = new MacroString(this) { Text = "MyDefaultPath" };
  }
  public override void Run(){
     Log.Info("The full path was '{0}'.", Path.GetFullPath(Filename.Expand(PlanRun)));
  }
}
```

Result Listeners

Result listeners have access to TestStepRun and TestPlanRun objects which contain variables that can be used as macros. An example is the previously mentioned DUT ID property, which is available if a DUT is used in the test plan. The following macros are available in the case of result listeners:

- `<Date>`: The start date and time of the executing test plan.
- `<Verdict>`: The verdict of the executing test plan. Only available in OnTestPlanRunCompleted.
- `<DUT ID>`: The ID (or ID's) of the DUT (or DUTs) used in the test plan.
- `<OperatorName>`: Normally the name of the user on the test station.

- `<StationName>`: The name of the test station.
- `<TestPlanName>`: The name of the executing test plan.
- `<ResultType>` (CSV only): The type of result being registered. This macro can be used if it is required to create multiple files, one for each type of results per test plan run.

Other Uses

Macro strings can also be used in custom contexts defined by a plugin developer. In this case it is up to the plugin developer to provide documentation of the available macros.

One example is the session log. It can be configured in the **Engine** pane in the **Settings** panel. The session log only supports the `<Date>` macro, which is defined as the start date and time of the OpenTAP instance and not the test plan run. This is because the session is active for multiple test plan runs and needs to be loaded when OpenTAP starts, therefore, most macros are not applicable.

Threading and Parallel Processing

Threading and parallelism are essential tools for enhancing the performance of test plan execution. By default, a test plan executes in a single thread, but it can branch off into multiple parallel threads as it progresses. Utilizing logging or Result Listeners can cause certain actions to execute in separate threads.

However, parallelism comes with some limitations due to the inherent complexity of managing multiple threads.

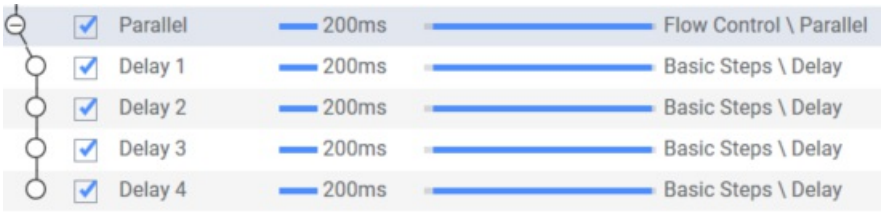
In OpenTAP and C#, parallelism can be implemented in several ways:

- **Parallel Steps**: The simplest form of parallelism, allowing test steps to be executed concurrently.
- **Deferred Processing**: Enables the processing of results in a separate thread.
- **TapThreads**: OpenTAP's own thread pool, which manages parent and child threads.
- **.NET Threads**: Basic threading provided by the .NET framework.
- **.NET Tasks**: Lightweight threads, also provided by the .NET framework.

Parallel Steps

The parallel step is a test step from the OpenTAP basic plugins. It runs all the child steps in parallel threads.

In the below screen shot, you can see four delay steps running in parallel inside a Parallel step:



Four Delays

It is best to use this step in a configuration where the resources used are not interferring with each other.

For example, you can have one branch setting up an instrument and one configuring a DUT. If threads needs to access the same instrument at the same time you might get unexpected behavior due to race conditions.

In order to control this, you can use the Lock step, which locks a named local or system-wide mutex.

In the below screenshot you can see how parallel steps with locks are evaluated. Notice that the total time was 3x the delay because none of the steps were executed in parallel.



Locked Delays

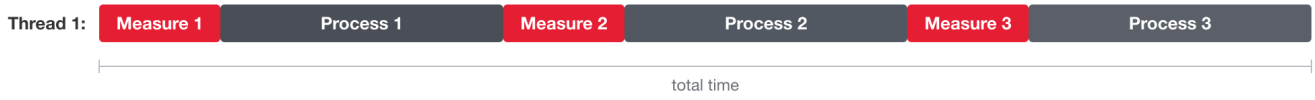
Deferred Processing

Deferred results processing enables post-processing of results while the test plan execution continues. This approach is a form of limited parallelism, beneficial when performance is constrained by sequential

data acquisition and processing, and there are spare computational resources available. Deferred processing is most effective when processing time significantly exceeds measurement time, but it can also be useful for shorter processing tasks.

Sequential Processing

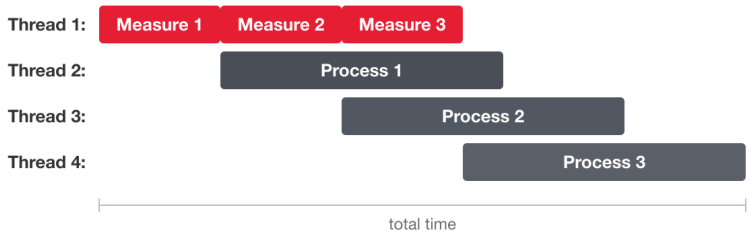
In traditional sequential processing, the order of operations is as follows:



Sequential Processing

Deferred Processing

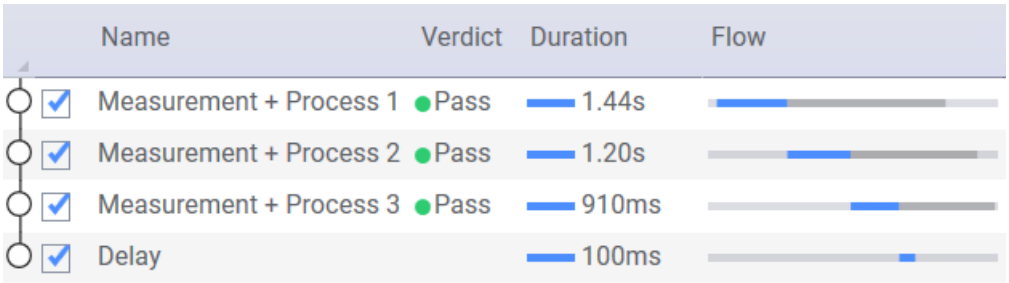
When deferred processing is used, operations are handled in parallel, as shown in the diagram below:



Parallel Processing

Visualization in KS8400

In KS8400, you can visualize the parallelism to gain insights into the performance improvements. The image below shows three Measurement + Process steps with a blocking measurement part and a non-blocking processing part. The Flow column's bars indicate the blocking part of the execution in blue and the non-blocking part in dark gray.



Deferred Parallelism

Implementing Deferred Processing

To incorporate deferred processing within a test step's `Run` method, use the `Results.Defer` method. The example below demonstrates this:

```
// This goes inside a Test Step implementation
public override void Run()
{
    // Execute the blocking part of the test step
    double[] data = instrument.DoMeasurement();

    Results.Defer(() => {
      // The non-blocking part of the execution is handled inside this anonymous function
      var processedData = ProcessData(data);
      Results.Publish(processedData);
      var limitsPassed = CheckLimits(processedData);
```

```
      if(limitsPassed)
        UpgradeVerdict(Verdict.Pass);
      else
        UpgradeVerdict(Verdict.Fail);
    });
}
```

In this example: 1. **Blocking Measurement**: The test step performs a measurement that blocks further execution. 2. **Deferred Processing**: The `Results.Defer` method queues the non-blocking processing operations to be executed concurrently. 3. **Processing**: Inside the deferred anonymous function, data is processed, results are published, and limits are checked. 4. **Verdict Upgrading**: Based on the processed data, the test verdict is upgraded to `Pass` or `Fail`.

By using deferred processing, you can optimize test execution, reducing the overall test plan duration and improving resource utilization.

TapThreads

TapThreads function similarly to .NET Threads but include additional features tailored for OpenTAP plugins, enhancing their efficiency and manageability within the OpenTAP environment.

- **Thread Pools**: When a function is requested for execution, a thread is either retrieved from the pool or a new one is started. Once the function completes, the thread is returned to the pool. Unlike .NET Thread Pools, TapThreads are more proactive in starting new threads and are optimized for IO-bound applications, ensuring minimal latency and high performance in handling asynchronous tasks.
- **Hierarchical Structure**: TapThreads utilize thread-local storage to track the initiating thread, enabling data sharing across thread hierarchies. The `ThreadHierarchyLocal` class facilitates this data sharing. If an OpenTAP thread is aborted, its child threads also receive the abort signal. This mechanism is crucial for managing Sessions and maintaining data integrity across different layers of the thread hierarchy.

Starting a TapThread

To start a TapThread, use `TapThread.Start()`, which provides an easy-to-use interface for running tasks asynchronously.

```
TapThread.Start(() =>
{
    // Perform a time-consuming task
    TapThread.Sleep(100);
    // Thread finishes and is returned to the pool
});
```

Threads from the pool start almost instantly due to the pre-allocation and management of live threads, ensuring efficient task execution.

Obtaining Results

To obtain results from your thread, use the `TaskCompletionSource` object. This approach provides a robust and flexible way to handle asynchronous operations and their outcomes.

```
var promise = new TaskCompletionSource<double>();
TapThread.Start(() =>
{
    try
    {
        TapThread.Sleep(100); // Throws an exception if the thread is aborted.
        promise.SetResult(9000.0);
    }
    catch (Exception e)
    {
        promise.SetException(e);
    }
});
```

```
double resultValue = promise.Task.Result;
```

Using `TaskCompletionSource` ensures that your asynchronous code can handle both successful completion and exceptions in a structured manner.

Note, you can also use other synchronization mechanisms for getting the result, but this method is very robust and performant.

Sleeping

You can make the current thread sleep with `TapThread.Sleep(TimeSpan duration)` or `TapThread.Sleep(int milliseconds)`. This pauses the thread for at least the specified time but may take a few extra milliseconds to wake up, making it unsuitable for highly precise waits.

```
TapThread.Sleep(100); // Sleep for 100 milliseconds
```

If the thread is aborted while sleeping, an `OperationCancelledException` will be thrown. To ensure the thread sleeps for the specified duration regardless of abort signals, use `System.Threading.Thread.Sleep()`:

```
System.Threading.Thread.Sleep(100); // Uninterruptible sleep
```

Aborting

TapThreads can be aborted using the `TapThread.Abort()` method. This event propagates to all child threads, which also receive the abort notification. This is a 'soft' abort, meaning the threads must cooperate to be aborted. To detect if a thread has been aborted, you have several options:

1. **Throw an Exception**: Use `TapThread.ThrowIfAborted()` to throw an exception if the current TapThread has been aborted.

   ```
   TapThread.ThrowIfAborted();
   ```

2. **Check Abort Status**: Check if the thread is aborted via `TapThread.Current.AbortToken.IsCancellationRequested`.

   ```
   if (TapThread.Current.AbortToken.IsCancellationRequested)
   {
       // Handle abort
   }
   ```

3. **Use AbortToken**: Use the `TapThread.Current.AbortToken` with calls that support it. Various .NET APIs accept a `CancellationToken` to enable canceling long-running operations.

   ```
   var token = TapThread.Current.AbortToken;
   // Example with Task.Delay
   await Task.Delay(1000, token);
   ```

4. **Register an Event**: Register an event to occur when the thread is aborted:

   ```
   using (TapThread.Current.AbortToken.Register(() =>
   {
       Log.Info("The thread was aborted!");
   }))
   {
       // Do something that takes time.
   }
   ```

These options provide flexibility in managing thread termination and ensuring resources are cleaned up properly.

Creating New Thread Contexts

In rare cases, you might need to run code in a context that cannot be aborted or can be aborted separately. For this, use `TapThread.WithNewContext`. It runs inside the same physical thread but creates a new temporary context where some code can be executed. You can also control which parent thread the new context has.

```
var firstThread = TapThread.Current;
```

```
TapThread.WithNewContext(() =>
{
    var secondThread = TapThread.Current;
    // firstThread != secondThread
},
// Specify the parent thread. Null means the root thread of the application.
null);
```

Creating new thread contexts allows for isolated execution environments within the same physical thread, providing greater control over task execution and abort behavior.

.NET Threads and Tasks

Generally, using the default .NET Threads and Tasks is not recommended. Threads are expensive to start, and tasks can exhibit unexpected behaviors that make them unsuitable for many use cases.

For OpenTAP plugins, it is recommended to use other parallelism techniques unless .NET Threads or Tasks are strictly necessary.

Environment Variables

Below is table of environment variables defined and/or used by OpenTAP. Some are set by OpenTAP itself at various times, some can be set by the user.

Name	Description	Values
OPENTAP_NO_VISA_DISCOVER	Disables VISA address discovery. On some software configurations this has been known to cause a hard crash.	true
OPENTAP_COLOR	This can be used to control if OpenTAP should use colors when printing to the console.	auto/always/never
OPENTAP_NONINTERACTIVE	Set by OpenTAP to signal to subprocesses that it’s running without the option for the user to respond	true
OPENTAP_ANSI_COLORS	Set if OpenTAP should use ANSI secape codes for log colors.	true
OPENTAP_NO_UPDATE_CHECK	Disables update checks.	true
OPENTAP_LIBVISA_LOCATION	Used for locating a custom libvisa resource. e.g libvisa32.dll	filepath
OPENTAP_INIT_DIRECTORY	Automatically set during startup. This is the directory in which OpenTAP is located.	
OPENTAP_DEBUG_INSTALL	Set while building plugins	true
OPENTAP_WARNINGS_TO_STDERR	If true, OpenTAP prints warnings to stderr. Otherwise warnings are printed to stdout.	true