Developer Guide

Table of Contents

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 **C# programmers** who are developing OpenTAP plugins or integrating OpenTAP into their own applications. 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 .

Development requires the following software:

- Visual Studio 2022 or above
- OpenTAP

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 an enterprise and community version of PathWave Test Automation Developer's System that provides a highly flexible graphical user interface and code examples.

OpenTAP Overview

OpenTAP is a software solution for fast and easy development and execution of automated test and calibration algorithms. These algorithms control measurement instruments and *devices under test* (DUTs). By leveraging the features of C#/.NET and providing an extensible architecture, OpenTAP minimizes the amount of code needed to be written by the programmer.

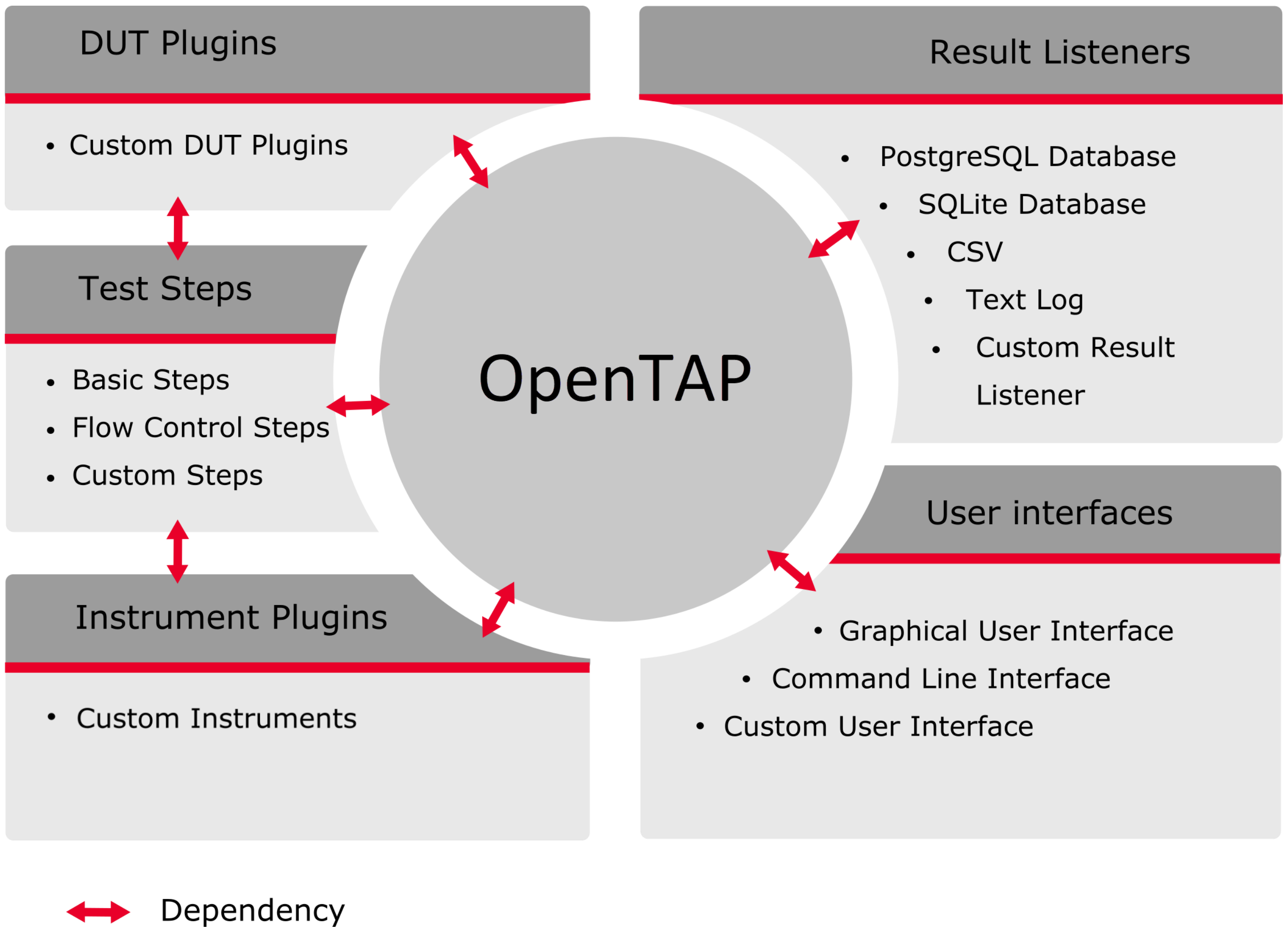
OpenTAP offers a range of functionality and infrastructure for configuring, controlling, and executing test algorithms. OpenTAP provides an API for implementing plugins in the form of test steps, instruments, DUTs and more.

OpenTAP consists of multiple executables, including: - OpenTAP (as a dll) - Command Line Interface (CLI) - Package Manager

Steps frequently depend on DUT and Instrument plugins. The development of different plugins is discussed later in this document.

Architecture

The illustration below shows how OpenTAP is central to the architecture, and how plugins (all the surrounding items) integrate with it.



OpenTAP Assembly

The OpenTAP assembly is the core and is required for any OpenTAP plugin. The most important classes in OpenTAP are: TestPlan, TestStep, Resource, DUT, Instrument, PluginManager and ComponentSettings. OpenTAP also provides an API, which is used by the CLI, and other programs like the editor GUI.

Graphical User Interface

If a graphical user interface is needed you can download the Keysight Test Automation Developer's System (Community or Enterprise Edition). It provides you with both a Software Development Kit (SDK) as well as an Editor GUI

- The graphical user interface consists of multiple dockable panels. It is possible to extend it with custom dockable panels. For an example, see `TAP_PATH\Packages\SDK\Examples\PluginDevelopment.Gui\GUI\DockablePanel.cs`
- Users can specify one or more of the following command line arguments when starting the editor GUI:

Command	**Description**	**Example**
Open	Opens the specified test plan file	tap editor –open testplan.tapplan
Add	Adds a specific step to the end of the test plan.	tap editor –add VerifyWcdmaMultiRx
Search	Allows PluginManager to search the specified folder for installed plugins. Multiple paths can be searched.	tap editor –search `C:\myPlugins`

OpenTAP Command Line Interface

The OpenTAP CLI is a console program that executes a test plan and allows easy integration with other programs. The CLI has options to configure the test plan execution, such as setting external parameters, configuring resources and setting meta data.

OpenTAP API

The OpenTAP API allows a software to configure and run a test plan. A help file for the C# API (OpenTAPApiReference.chm) is available in the OpenTAP installation folder. Examples on how to use the OpenTAP API can be found in the **TAP_PATH** folder.

OpenTAP Plugins

An essential feature of OpenTAP is its flexible architecture that lets users create plugins. **OpenTAP plugins** can be any combination of TestStep, Instrument, and DUT implementations. Other OpenTAP components such as Result Listeners and Component Settings are also plugins. By default, OpenTAP comes with plugins covering basic operations, such as flow control and result listeners.

Plugins are managed by the **PluginManager**, which by default searches for assemblies in the same directory as the running executable (the GUI, CLI or API). Additional directories to be searched can be specified for the GUI, the CLI and the API. When using the API, use the PluginManager.DirectoriesToSearch method to retrieve the list of directories, and add any new directories to the list.

OpenTAP Packages

OpenTAP plugins and non-OpenTAP files (such as data files or README files) can be distributed as **OpenTAP Packages**. OpenTAP Packages can be managed (installed, uninstalled, etc.) using the Package Manager (described in more details later in this guide).

Test Plans

A *test plan* is a sequence of test steps with some additional data attached. Test plans are created via the Editor GUI. Creating test plans is described in the *Editor Help* (EditorHelp.chm), accessible within the Editor GUI. Test plan files have the *.TapPlan* suffix, and are stored as xml files.

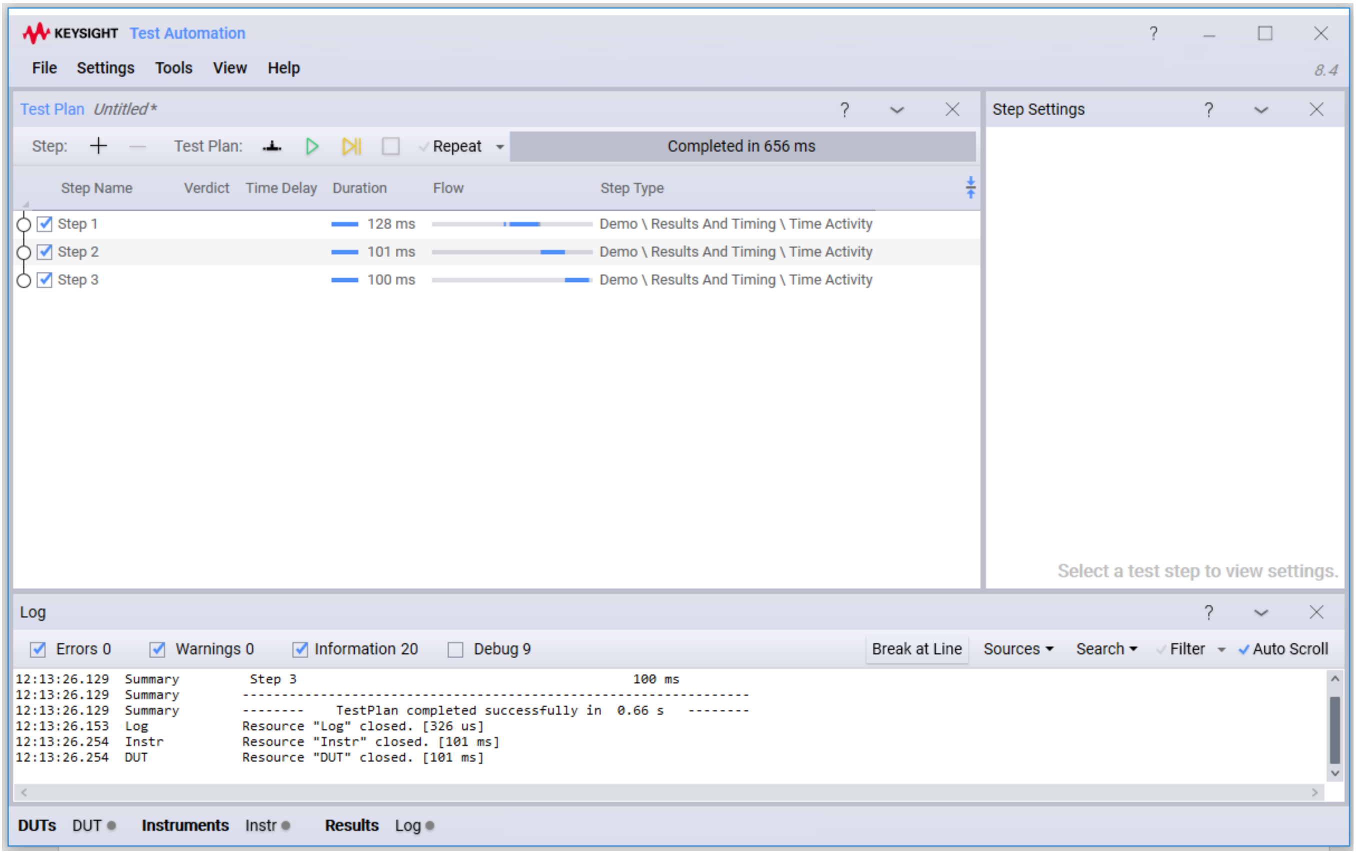
Test Plan Control Flow

To use OpenTAP to its full potential, developers must understand the control flow of a running test plan. Several aspects of OpenTAP can influence the control flow. Important aspects include:

- Test plan hierarchy
- TestStep.PrePlanRun, TestStep.Run, TestStep.PostPlanRun methods

- Result Listeners
- Instruments and DUTs
- Test steps modifying control flow

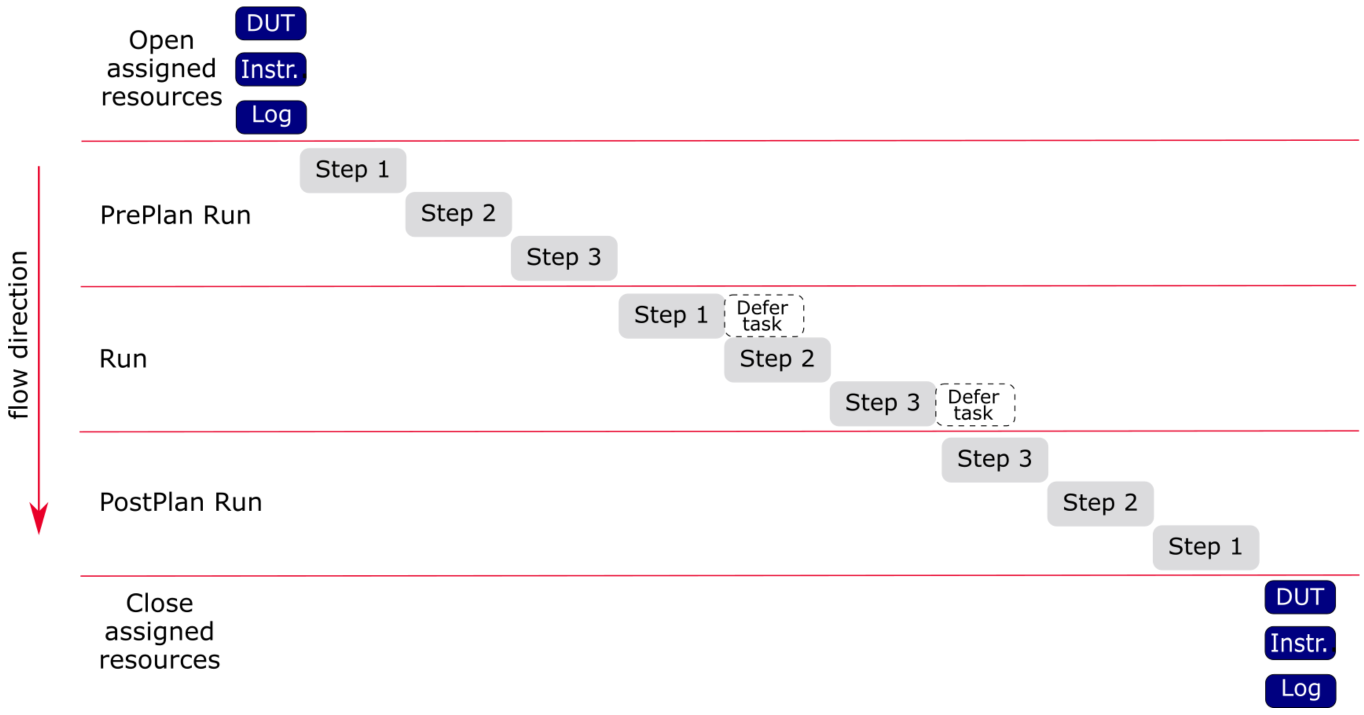
The following test plan uses test steps, DUTs and instruments defined in the **Demonstration** plugin:


KEYSIGHT Test Automation
File Settings Tools View Help
8.4
Test Plan Untitled*
Step: Test Plan: Repeat
Completed in 656 ms
Step Settings
Select a test step to view settings.
Log
Errors 0 Warnings 0 Information 20 Debug 9
Break at Line Sources Search Filter Auto Scroll
DUTs DUT Instruments Instr Results Log

Step Name	Verdict	Time Delay	Duration	Flow	Step Type
Step 1			128 ms		Demo \ Results And Timing \ Time Activity
Step 2			101 ms		Demo \ Results And Timing \ Time Activity
Step 3			100 ms		Demo \ Results And Timing \ Time Activity

```
12:13:26.129  Summary       Step 3                                                 100 ms
12:13:26.129  Summary      ------------------------------------------------------------------
12:13:26.129  Summary      --------     TestPlan completed successfully in  0.66 s   --------
12:13:26.153  Log          Resource "Log" closed. [326 us]
12:13:26.254  Instr        Resource "Instr" closed. [101 ms]
12:13:26.254  DUT          Resource "DUT" closed. [101 ms]
```

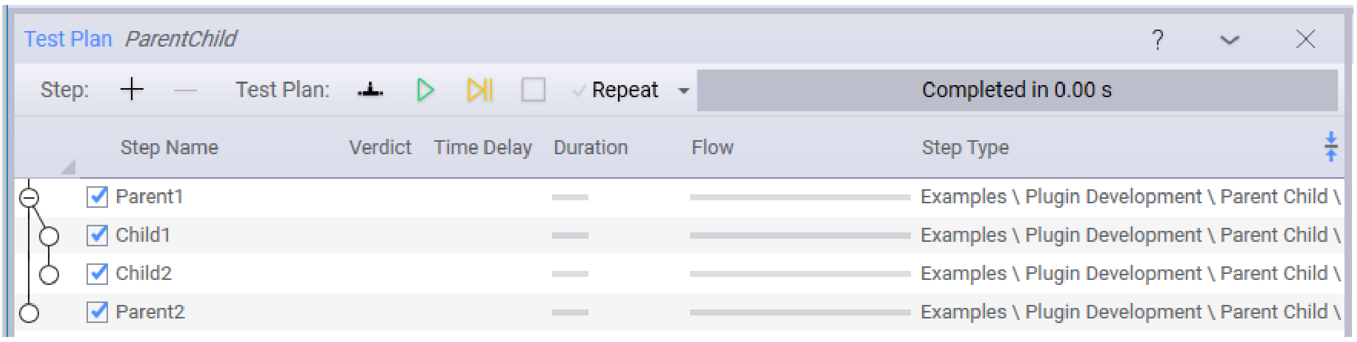
The test plan has three test steps, in succession. In this test plan, none of the steps have child steps. A more complex example, with child steps, is presented later in this section. The test plan relies on the resources DUT, Instr. and Log to be available and configured appropriately. The following figure illustrates what happens when this test plan is run:



In the **Open assigned resources** phase all DUTs, instruments and configured result listeners are opened in parallel. As soon as all resources are open the **PrePlanRun** methods of the test steps execute in succession. This is followed by the execution of the **Run** methods where all test steps are run one at a time. It is possible to allow a test step to run code after its run is completed. This is done by defining a **defer task** for the test step. To learn more about defer task see the *Plugin Development* folder under *Packages/SDK/Examples*, located in the OpenTAP installation folder.

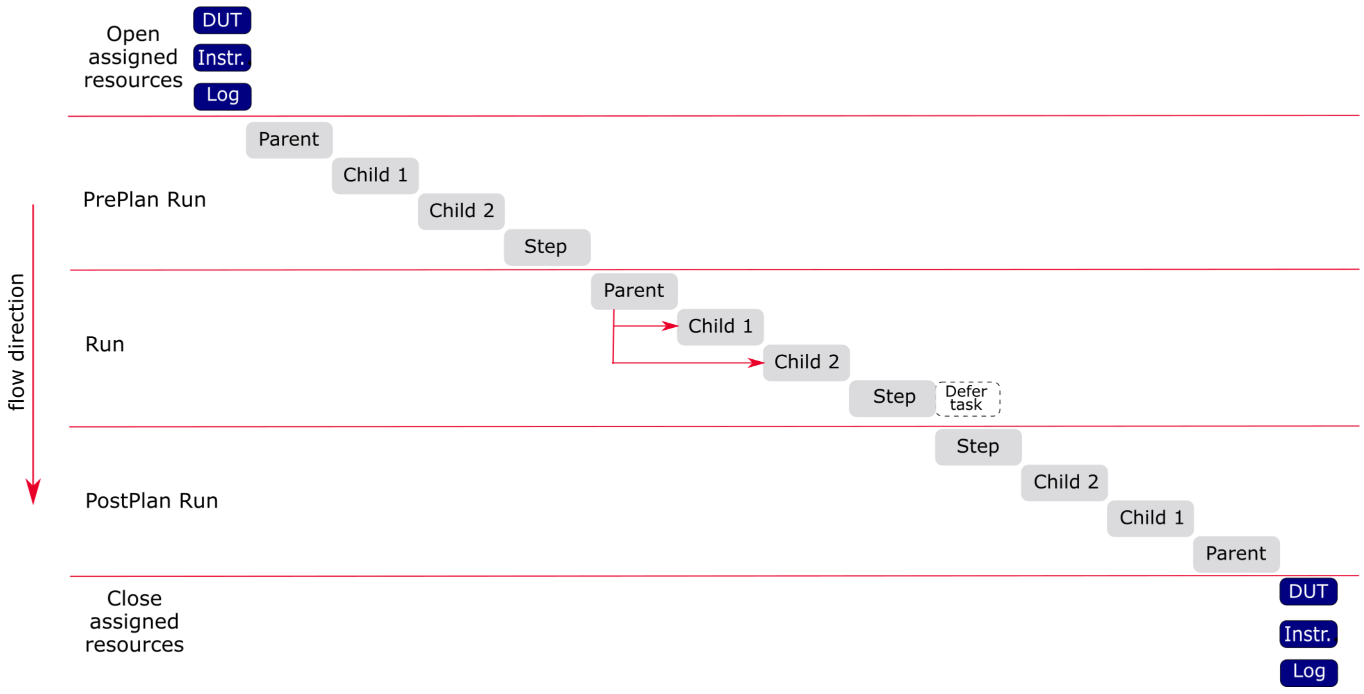
After the test step run is completed for each test step, **PostPlanRun** is executed, *in reverse order*, for each test step. The final step is **Closing assigned resources** which happens in parallel for all previously opened resources.

The test plan below illustrates how child test steps are handled:



Step Name	Verdict	Time Delay	Duration	Flow	Step Type
✓ Parent1					Examples \ Plugin Development \ Parent Child \
✓ Child1					Examples \ Plugin Development \ Parent Child \
✓ Child2					Examples \ Plugin Development \ Parent Child \
✓ Parent2					Examples \ Plugin Development \ Parent Child \

The methods in the test steps execute in the following order:



Similar to the previous example, test plan execution starts with the **Open assigned resources** phase, followed by the execution of the **PrePlanRun** methods. The PreplanRun methods are executed in the order of the steps in the test plan. Next, the Run method of the Parent step is executed. The Parent step controls the execution order of the Run methods of its child steps. The example above shows the case, where *Parent* calls its child steps sequentially. Following this, the run method of *Step* is executed. The **PostPlanRun** methods are executed in reverse order of placement in the test plan, starting with *Step 2* followed by *Child2*, *Child1* and finally *Parent*. In the last step all assigned resources are closed.

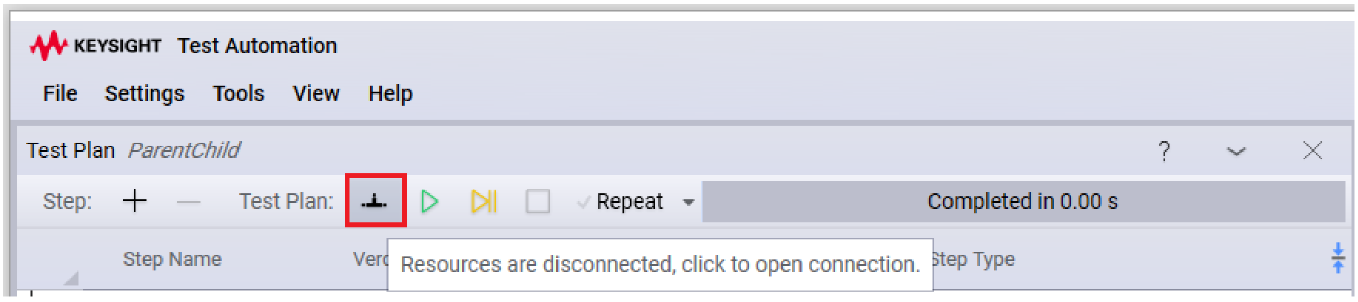
Note that the above examples are very simple. A more advanced test plan may incorporate flow control statements to change execution flow. For example, adding a *Parallel* test step as a parent to child test steps will make the test steps run in parallel. This only affects the run stage, the other stages remain unchanged.

External Parameters

Editable OpenTAP step settings can be marked as *External*. The value of such settings can be set through the Editor GUI, through an external program (such as OpenTAP CLI), or with an external file. This gives the user the ability to set key parameters at run time, and (potentially) from outside the Editor GUI. You can also use the API to set external parameter values with your own program.

Manual Resource Connection

You may want to avoid the time required to open resources at each test plan start. To do so, open the resources by using the **Connection** button:



Resources are disconnected, click to open connection.

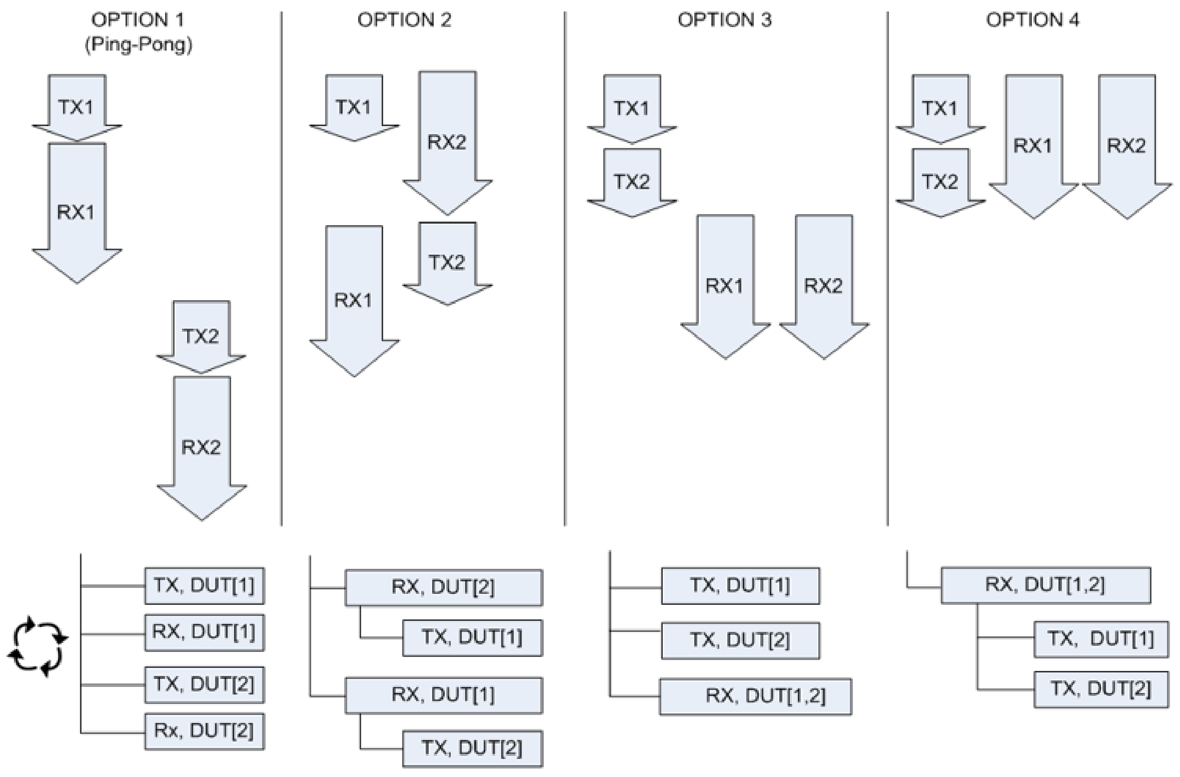
Resources opened manually remain open between test plan runs. This eliminates the time required to open and close them for each test plan run.

Note: You must ensure that the resources can be safely used in this manner. For example, if a Dut.Open configures the DUT for testing, you may need to keep the default behavior of opening the resource on every run.

Testing Multiple DUTs

Test step hierarchies can be built, and attributes set, to allow certain steps to have child steps. This hierarchical approach, and the possibility of communicating with one or multiple DUTs from a single test step, allows for a variety of test flows.

The following figure illustrates four different approaches where both sequential and parallel execution is used. The upper part of the illustration is the flow; the lower part is the test plan execution showing the corresponding TX and RX test steps.



- Flow Option 1 is a simple sequential test plan execution where TX (transmit) and RX (receive) test steps are repeated once for each DUT. In a production environment, this is a simple way to reduce the test/calibration time, because it lets the operator switch in a DUT while the other DUT is being tested.
- Options 2 to 4 all consist of the same two test steps: a TX step that controls a *single* DUT and an RX step that allow control of *multiple* DUTs simultaneously. Usually several DUTs can listen to an instrument transmitting a signal at the same time, but calibration instruments (which receive) usually can’t analyze more than one TX signal at a time.
- In Option 2, the TX test step is a child step of the RX step (via the use of the AllowChildrenOfType attribute). Being a child means that the TX step will be executed as a part of the RX step execution. The flow in Option 2 starts by executing the RX test step that brings DUT2 into RX test operation. Then the TX child step runs, bringing DUT1 into TX mode. When the TX child step has finished, the RX test step continues for DUT1.
- Options 3 and 4 reuse the functionality, but form even more optimized test plan flows.

Note: All the above is highly DUT and instrument dependent.

Getting Started

An OpenTAP **plugin** refers to a type which satisfies an interface recognized by OpenTAP. The majority of plugins are C# classes. In this case, more concrelety, you can think of a plugin as a C# class which inherits from a base class from OpenTAP, or implements an OpenTAP interface. A plugin is distributed as a compiled `.dll` file in a **TapPackage**. A TapPackage is a collection of one or more plugin DLLs, their dependencies, and additional metadata, such as a description and versioning information. TapPackages can be shared on the Package Repository . For in-depth information on packaging, see Plugin Packaging and Versioning . A TapPackage is usually based on a C# solution. This document is details how to create a new solution which automates the process of creating and versioning an OpenTAP plugin package.

Besides C#, it is also possible to develop plugins in Python by using the Python plugin. Python development will not be covered here. See the Python Documentation for more information.

To get started, we recommend installing the OpenTAP NuGet Templates . They can be installed by using the dotnet CLI: `dotnet new install OpenTap.Templates::9.23.2`, or if your IDE supports it, you can search for online templates and look for `OpenTap.Templates`.

Using the NuGet Package

With the templates installed, you can create a new OpenTAP project through the `New Solution` option in your IDE, or you can use the dotnet CLI:

```
dotnet new sln --name MySolution
dotnet new opentap --name MyFirstPlugin
dotnet sln add MyFirstPlugin
```

To convert an existing project to an OpenTAP plugin, add a reference to the OpenTAP NuGet package . You can do this by using the dotnet CLI: `dotnet add package OpenTAP --version 9.23.2`, or by searching for "OpenTAP" in the NuGet package manager in your IDE. > NOTE: On Windows, only .NET Framework and netstandard2.0 is supported. If you are using .NET 6 or later, OpenTAP may not be able to correctly load your plugin.

NuGet Features

The OpenTAP NuGet package provides the following build-time features:

Installation

When your project references the NuGet package, OpenTAP will automatically be installed in your project's output directory (e.g. bin/Debug/). The installation will have the same version as your NuGet reference. By using a unique installation for each plugin, it is easy to manage different plugins which may depend on different versions of OpenTAP. The version of OpenTAP is recorded in the *.csproj file, which should be managed by version control (e.g. git), so all developers use the same version. This also automates the process of installing OpenTAP on a new machine, such as a Continuous Integration environment.

Package Creation

The NuGet package provides the option to automate packaging your plugins as a *.TapPackage as part of the build process. To take advantage of this feature, your project needs a package definition file . If you created you project with the NuGet Template, a package definition file called `package.xml` was already created for you. If you did not use the NuGet template, you can use the OpenTAP CLI Action `tap sdk new packagexml` to generate a skeleton file. If your project contains a file named package.xml, a TapPackage will be generated when it is built in Release mode. You can customize this behavior using these MSBuild properties in your csproj file:

```
<OpenTapPackageDefinitionPath>package.xml</OpenTapPackageDefinitionPath>
<CreateOpenTapPackage>true</CreateOpenTapPackage>
<InstallCreatedOpenTapPackage>true</InstallCreatedOpenTapPackage>
```

Reference Other OpenTAP Packages

When using the OpenTAP NuGet package, you can reference other TapPackages similar to how NuGet packages are referenced. Referenced packages are installed into your project's output directory (usually ./bin/Debug/) along with OpenTAP.

Reference a package by specifying it in your .csproj file:

```
<ItemGroup>
  <!-- NuGet reference to OpenTAP -->
  <PackageReference Include="OpenTAP" Version="9.23.2" />
  <!-- OpenTAP package sources -->
  <OpenTapPackageRepository Include="packages.opentap.io"/>
  <OpenTapPackageRepository Include="$HOME/Downloads;$HOME/Documents"/>

  <!-- Packages to reference  -->
  <OpenTapPackageReference Include="DMM API" Version="2.1.2" UnpackOnly="false" IncludeAssemblies="pattern1;pattern2"
ExcludeAssemblies="pattern3;pattern4" />
</ItemGroup>
```

> Notice the similarity between `<PackageReference .../>` (a NuGet reference), and `<OpenTapPackageReference ../>`.

The `<OpenTapPackageRepository/>` element is similar to the concept of NuGet Sources . It specifies from where OpenTAP packages should be resolved. The element can be specified multiple times, or the repository sources can be separated by a semicolon (`;`). All instances of this element will be joined during compilation. Http repositories and directory names are both valid sources. `packages.opentap.io` is always included by default, and cannot be excluded. It is included here only as an example.

When referencing a package in this way, assemblies belonging to that package are automatically referenced in your project.

All attributes except `Include` are optional. `Version` defaults to the latest release if omitted. `UnpackOnly` defaults to false, and can be used to suppress any install actions from running as part of the package being installed in the output dir.

The `IncludeAssemblies` and `ExcludeAssemblies` attributes control which assemblies are referenced. The supplied value is interpreted as one or more glob patterns, separated by semicolons (`;`). If not specified, they use the default values `IncludeAssemblies="**"` and `ExcludeAssemblies="Dependencies/**"`, meaning that all DLLs from the package are referenced if they are not in a subdirectory of the `Dependencies` folder. By leveraging glob patterns, your project can target specific dependencies of other packages.

Glob patterns are tested in order of specificity, meaning that the evaluation order of patterns in IncludeAssemblies and ExcludeAssemblies can be interleaved. Specificity is measured by the number of tokens used in the expression. By tokens, we mean the *components* of the expression, and not the characters. The components of `"Dependencies/**"` are `["Dependencies", "/", "**"]`, for instance. Because `Dependencies/**` is more specific than `**`, no DLLs in the Dependencies folder are referenced by default.

Building on this, the pattern `Dependencies/*AspNet*/**` will match all ASP.NET dependencies that the referenced package contains, and will take precedence over the default exclude pattern because it contains more components, and is thus more specific. In case of ties, Include patterns take precedence over Exclude patterns.

There is one exception to this rule: any pattern ending with `.dll`, except patterns ending with `*.dll`, are considered literal expressions, and will always take precedence regardless of the number of tokens, thus allowing for easily targeting a specific dll by using a pattern like `**NameOfDll.dll`. Note however that specifying an empty pattern, e.g. ExcludeAssemblies="", will not override the default value of this attribute, and will still exclude `Dependencies/**`. In order to not include or exclude anything, you must provide a placeholder value instead.

Note that OpenTAP uses the DotNet.Glob library to generate matches, which uses unix-like globbing syntax.

You can also specify a package that you just want installed (in e.g. bin/Debug/) but don't want your project to reference. This can be useful for defining a larger context in which to debug. It is done as follows:

```
<ItemGroup>
  <AdditionalOpenTapPackage Include="DMM Instruments" Version="2.1.2"/>
</ItemGroup>
```

SDK Package

The Software Development Kit (SDK) package demonstrates the core capabilities of OpenTAP and makes it faster and simpler to develop your solutions. It also contains the Developer Guide which contains documentation relevant for developers.

The package is available on packages.opentap.io . You can use the CLI install the package with the following command:

```
tap package install SDK
```

If you are using the Keysight Developer's System (Community or Enterprise Edition) you already have the SDK package.

SDK Templates

The OpenTAP SDK makes it easy to create plugin templates using the `tap sdk new` group of subcommands. From the command line you can call the following subcommands:

Commands	Description
tap sdk gitversion	Calculates a semantic version number for a specific git commit.
tap sdk new cliaction	Create a C# template for a CliAction plugin. Requires a project.
tap sdk new dut	Create a C# template for a DUT plugin. Requires a project.
tap sdk new instrument	Create C# template for an Instrument plugin. Requires a project.
tap sdk new packagexml	Create a package definition file (package.xml).
tap sdk new resultlistener	Create a C# template for a ResultListener plugin. Requires a project.
tap sdk new settings	Create a C# template for a ComponentSetting plugin. Requires a project.
tap sdk new testplan	**OBSOLETED:** Use an editor to create a TestPlan.
tap sdk new teststep	Create a C# template for a TestStep plugin. Requires a project.
tap sdk new project	Create a C# Project (.csproj). Including a new TestStep, solution file (.sln) and package.xml.
tap sdk new integration gitlab-ci	Create a GitLab CI build script. For building and publishing the .TapPackage in the given project.
tap sdk new integration gitversion	Configure automatic version of the package using version numbers generated from git history.
tap sdk new integration vs	Create files that enable building and debugging with Visual Studio.
tap sdk new integration vscode	Create files to enable building and debugging with vscode.

The following example shows how to create a new project using the SDK:

```
tap sdk new project MyAwesomePlugin
```

This command creates a new project called `MyAwesomePlugin` with a .csproj file, a test step class, a solution file, and a package.xml file.

Once you created a project you can easily add other templates. To add a DUT for example simply call the following command:

```
tap sdk new dut MyNewDut
```

Use an editor to create a testplan to use your new teststep and DUT! See https://doc.opentap.io/User%20Guide/Editors/ for more info on the different editors!

SDK Examples

Before you start to create your own project, look at the projects and files in **`TAP_PATH\Packages\SDK\Examples`**. This folder provides code for example DUT, instrument and test step plugins. First-time OpenTAP developers should browse and build the projects, then use e.g. the Editor GUI to view the example DUTs, instruments and test steps.

SDK Examples contains the following projects:

Folder	Description
`ExamplePlugin\ExamplePlugin.csproj`	Creates a plugin package that contains one DUT resource, one instrument resource, and one test step.
`PluginDevelopment\PluginDevelopment.csproj`	Creates a plugin package that contains several test steps, DUT resources, instrument resources, and result listeners.
`TestPlanExecution\BuildTestPlan.Api\BuildTestPlan.Api.csproj`	Shows how to build, save and execute a test plan using the OpenTAP API.
`TestPlanExecution\RunTestPlan.Api\RunTestPlan.Api.csproj`	Shows how to load and run a test plan using the OpenTAP API.

Offline Development

It is possible to develop plugins in an offline development. If you compile your project once, all online resources will be cached locally so subsequent builds will not require internet access. If you cannot bring your development machine online even once, you can perform the following steps:

1. Install Dotnet 6 SDK
2. Create a directory for local NuGet packages. Let’s call it `\path\to\nuget\source`
3. Download required NuGet packages and put them in the local source directory: > OpenTAP

   NETStandard.Library 2.0.3

   Microsoft.NETCore.Platforms 1.1.0 4. Add the local source to the list of nuget sources: `dotnet nuget add source \path\to\nuget\source` 5. Build your project with e.g. dotnet build

With these steps, you should be able to build an OpenTAP plugin if it does not have any dependencies. If you have additional NuGet dependencies, they can of course be added to the source you just created.

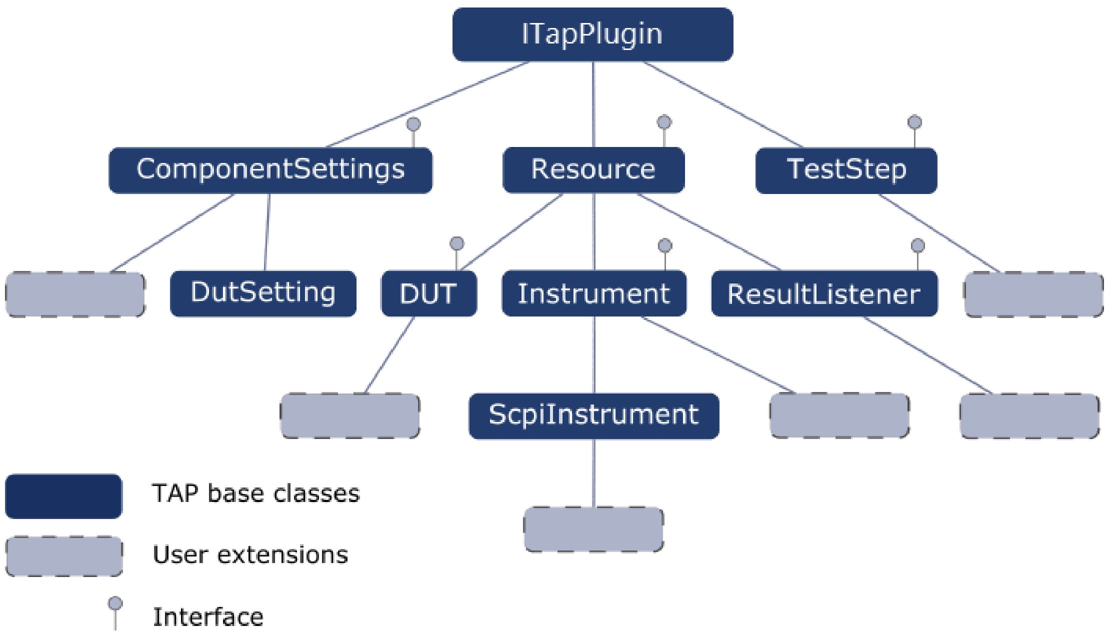
OpenTAP plugin dependencies consumed with the `AdditionalOpenTapPackage` feature must be added to the OpenTAP package cache: - Windows: `C:\Users\<username>\Appdata\Local\OpenTap\PackageCache\`. - Linux: `/home/<username>/.local/share/OpenTap/PackageCache/`.

If the directory does not exist you must create it.

Development Essentials

OpenTAP Plugin Object Hierarchy

At the very root of the OpenTAP plugin class hierarchy is the **ITapPlugin** class. Items in blue are OpenTAP base classes. The green, dotted rectangles are user extensions to those base classes.



Extending **ITapPlugin** are the **ComponentSettings**, **Resource**, and **TestStep** base classes. These are abstract classes and hence cannot be instantiated. They are used as base classes for other classes.

- **ComponentSettings** is used for storing settings throughout the system. It contains properties to be set or lists of instruments, DUTs, and so on. ComponentSettings are available via the various Settings menus in the GUIs.
- **Resource** is a base class for the DUT, Instrument and ResultListener classes. Resources are often specified in the settings for a test step. Referencing a 'Resource' from a test step will cause it to automatically be opened upon test plan execution.
- **TestStep** does not inherit from Resource but inherits directly from ITapPlugin.

Developing Using Interfaces

Developers may choose to NOT extend the OpenTAP base classes, but instead to implement the required interfaces. You can inherit from as many interfaces as you want, but from only one class. The TestStepInterface.cs file provides an example of implementing ITestStep.

Developers implementing IResource (required by virtually all other OpenTAP plugin classes), must ensure that the implementation of:

- IResource.Open sets IsConnected to true, and
- IResource.Close sets IsConnected to false.

Working with User Input

It is possible to request information from the user via the `UserInput` class. You can use this in both the CLI and a GUI. In a GUI, this allows you to create a pop-up dialog the user can interact with.

The `UserInput.Request` method takes three parameters:

- `dataObject` - the object the user needs to provide. Can be a test step, an instrument or a DUT among others.

- `Timeout` - specify how long to wait for the user input. After the specified amount of time an exception will be thrown.
- `modal` - a Boolean that specifies if the user can interact with background objects if the dialog is open. If it is set to true the user will have to answer before doing anything else.

To finalize the input you can use the `SubmitAttribute` . If this is used in a GUI with an enum, buttons appear that can submit or cancel the user input. In this case we recommend using the `SubmitAttribute` together with the `LayoutAttribute` . This helps to create a natural look-and-feel for your UI because it handles how the buttons are displayed. You can use this attribute to control the height, width and placement of the buttons. To give a title to your dialog box you can use the `Create Name` property.

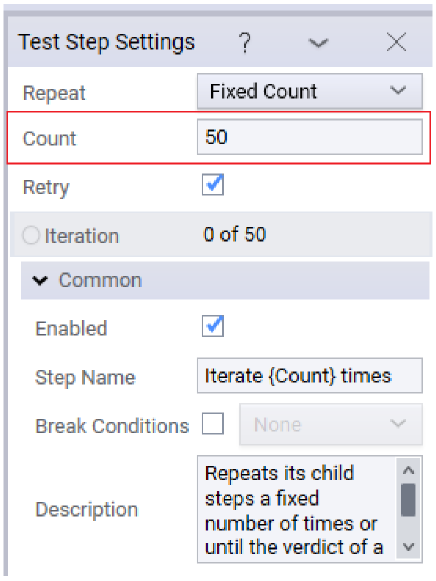
Plugin Settings

Adding user settings to a *TapPlugin*, such as a step setting for a test step, can be done with properties. These properties are visible and editable in the GUI Editor. Properties typically include instrument and DUT references, instrument and DUT settings, timing and limit information, etc. Defining these properties is a major part of plugin development.

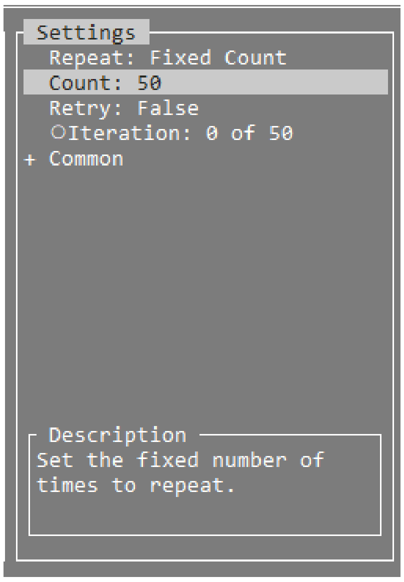
An example from the repeat step:

```
[EnabledIf("Action",RepeatStepAction.Fixed_Count, HideIfDisabled = true)]
[Display("Count", Order: 1, Description: "Set the fixed number of times to repeat.")]
public uint Count { get; set; }
```

Turns into:


Test Step Settings

Repeat	Fixed Count
Count	50
Retry	✓
Iteration	0 of 50
Common	
Enabled	✓
Step Name	Iterate {Count} times
Break Conditions	None
Description	Repeats its child steps a fixed number of times or until the verdict of a


```
Settings
  Repeat: Fixed Count
  Count: 50
  Retry: False
  ○Iteration: 0 of 50
+ Common

Description
Set the fixed number of
times to repeat.
```

The SDK provides many examples of properties for test step development in the **`TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps`** folder.

A C# property has a type, a name and a visibility specifier. The most basic setting has just these three components. By default the property must have public visibility to be configured by the user - otherwise it is not considered a setting.
A very typical example could look like this:

```
public double TxFrequency { get; set;} = 1e9; // A transmission frequency of 1 GHz.
```

Let’s break this down. - `public`: This means that this property can be seen from outside the scope of the class. This can also be set to private, protected or internal, but in these cases they are not considered ‘settings’ and won’t show up in UIs and won’t be saved in the test plan file. - `double`: The type of the property. In this case this is a decimal value with 64 bits of precision. Here anything can be specified and it will show up in the user interface, but displaying it or configuring it may not always be possible - it has to be a type supported by the installed plugins. By default the following types are supported - float, double, decimal - Respectively 32-, 64- and 128-bit precision decimal values. - integer types - short, ushort, int, uint, long, ulong, byte - various precision integer types. - string - a string of text. - bool - A true / false value. This is often shown as a checkbox. - enums - A list of selectable values. This is often shown as a drop down. - Resources - A selectable resource type, e.g Instrument, DUT, … This will be restricted by the selected subtype of resource, e.g ScpiInstrument is a specialization of Instrument and hence only ScpiInstrument types will be available in the dropdown. - Lists - A editable list of objects. Often shown as a data grid or spreadsheet. - TimeSpan - A span of time. e.g 1 hour. This is shown as a text box. - Enabled`<T>` - An enable setting encapsulating another type. This is shown as a checkbox followed by something else. - `TxFrequency`: The name of the property. By default this is what will be presented to the user. -`{get;set;}`: Specifies that this is a C# property. 'set' can be omitted to make it read-only. - `= 1e9`: Specifies the default value of this property.

Setting Attributes

Attributes can be used to further improve the behavior of the property. For a full list of default supported attributes, see Attributes Used by OpenTAP .

In the example seem above, we could further improve the usability by adding some attributes.

```
// Specify that the user-friendly name is "Transmission Frequency" and provide a tooltip.
// furthermore, the group is set to "DUT", which means the setting will be added to a group called "DUT".
[Display("Transmission Frequency", "Transmission frequency configured on the DUT. This should be between 500MHz and
2GHz.", "DUT")]
// Specify that the unit is hertz, Hz.
[Unit("Hz")]
public double TxFrequency { get; set;} = 1e9;
```

Best Practices For Settings

Any code can be evaluated in property getters and setters, but this can often be the source of bugs and performance issues.

1. Keep it simple: Don't use complex behavior in property setters as this can cause subtle bugs and performance issues. Avoid setting other properties' backing value in a property setter. If you want to be able to configure a preset of values, consider adding a button.
2. Use validation rules for signaling invalid configurations.

3. Use a guard clause to check the value before calling expensive update functions.

    ```
    public double TxFrequency
    {
       get => txFrequency;
       set
       {
          if (value == txFrequency) return; // early return.
          txFrequency = value;
          OnPropertyChange(nameof(TxFrequency)); // Expensive operation.
       }
    }
    ```

Best Practices for Plugin Development

The following recommendations will help you get your project off to a good start and help ensure a smooth development process:

- You can develop one or many plugins in one C# project. The organization is up to the developer. The following is recommended:
    - Encapsulate your logic. Keeping all instrument logic inside the instrument class makes it possible to swap out instruments without changing TestSteps. For example, a TestStep plugin knows to call **MeasureVoltage**, and the instrument plugin knows how to get that measurement from its specific instrument.
    - You can put Instruments, DUTs, and TestSteps all in separate packages and create a "plug-and-play" type of interaction for test developers. For example, you can create test steps that make a measurement and plot a result. If done properly, the steps work regardless of which instrument gets the data or what type of device is being tested.

- Don't introduce general settings unless absolutely necessary. Instead try to move general settings to test steps (such as a parent step holding settings for a group of child steps) or to DUT or Instrument settings.

- For DUTs, Instruments, and Result Listeners, set the Name property in the constructor, so that this Name appears in the Resource Bar.

- Use the **Display** attribute (with a minimum of name and description) on properties and classes. This ensures good naming and tooltips in the GUI Editor.
- Use **Rules** for input validation to ensure valid data.

Test Step

A test step plugin is developed by extending the **TestStep** base class. As you develop test steps, the following is recommended:

- Isolate each test step into a single .cs file, with a name similar to the display name. This makes the code easy to find, and focused on a single topic.
- Use **TapThread.Sleep()** for sleep statements. This makes it possible for the user to abort the test plan during the sleep state.

Default Implementation

The default implementation of a TestStep (as generated when using the Visual Studio Item Template for TestSteps) includes:

- A region for **Settings**, which are configurable inputs displayed in **Step Settings** panel. While initially empty, most test steps require the user to specify settings, which are very likely referenced in the Run method.
- A **PrePlanRun** method that may be overridden. The PrePlanRun method:
  - Is called *after* any required resources have been opened and *prior* to any calls to TestStep.Run. (It is NOT called immediately before the test step runs.)
  - Should perform setup that is required for each test plan run, such as configuring resources that are needed for test plan execution.

  PrePlanRun methods are called sequentially in a flattened, top-to-bottom order of the steps placement in the test plan.
- A **Run** method that must be included. In the absence of any flow control statements, the required Run method is called in the order of the placement of test steps in the test plan. The Run method:
  - Implements the primary functionality of the test step.
  - Typically leverages the test step's settings.
  - Often includes logic to control the DUTs and instruments, determine verdicts, publish results, log messages, etc. Separate sections deal with many of these topics.
- A **PostPlanRun** method that may be overridden. PostPlanRun methods are called sequentially in a flattened, bottom-to-top order of their placement in the test plan (the reverse of the PrePlanRun order.) The PostPlanRun method:
  - Is used for "one-time" cleanup or shutdown.
  - Is called after the test plan has completed execution, and *prior* to any calls to close the resources used in the test. (It is NOT called immediately after the test step runs).
  - Is always called if PrePlanRun for the test step was called (also in case of errors/abort). This method can be used to clean up PrePlanRun actions.

The following code shows the template for a test step:

```
namespace MyOpenTAPProject
{
    [Display("MyTestStep1", Group: "MyPlugin2", Description: "Insert a description here")]
    public class MyTestStep1 : TestStep
    {
        #region Settings
        // ToDo: Add property here for each parameter the end user is able to change
        #endregion
        public MyTestStep1()
        {
            // ToDo: Set default values for properties / settings.
        }

        public override void PrePlanRun()
        {
            base.PrePlanRun();
            // ToDo: Optionally add any setup code this step must run before testplan start.
        }

        public override void Run()
        {
            // ToDo: Add test case code here.
            RunChildSteps(); // If step has child steps, run them.
            UpgradeVerdict(Verdict.Pass);
        }
        public override void PostPlanRun()
        {
            // ToDo: Optionally add any cleanup code this step needs to run after the
            // entire testplan has finished.
            base.PostPlanRun();
        }
    }
}
```

Test Step Hierarchy

Each TestStep object contains a list of TestSteps called *Child Steps*. A hierarchy of test steps can be built with an endless number of levels. The child steps:

- Are run in sequence by calling the **RunChildSteps** method from the parent step's Run method.
- Can also be run individually using the **RunChildStep** method to get even more fine-grained control. This is an advanced topic. Running child steps should only be done when the control flow is inside the parent step's Run method.

Defining Relationships

Parent/Child relationships are defined by attributes associated with the Test Step class definition:

- From a *Parent* perspective, the **AllowAnyChild** and **AllowChildrenOfType** attributes define the parent/child relationship.
- From a *Child* perspective, the **AllowAsChildIn** attribute defines the parent/child relationship.

If multiple attributes apply (such as AllowChildrenOf Type and AllowAsChildIn), then both must evaluate to **true** for the GUI to allow a child to be inserted under a parent.

It is common practice for a child to use properties from a parent. For example, a child might need to reference a DUT or instrument defined in a parent. The **GetParent** method allows a child to search for a parent of a particular type, and retrieve a setting. For example:

```
public override void PrePlanRun()
{
    base.PrePlanRun();

    // Find parents of a certain type, and get a resource reference from them.
    // Resources include things like Instruments and DUTs.
    _parentsDut = GetParent<ParentWithResources>().SomeDut;
    _parentsInstrument = GetParent<ParentWithResources>().SomeInstrument;
}
```

It is valuable to use *interfaces* instead of *types* in the AllowChildrenOfType and AllowAsChildIn attributes. This more general approach allows any test step child that implements the appropriate interface. For example:

```
[Display(Groups: new[] { "Examples", "Feature Library", "ParentChild" }, Name: "Child Only Parents With InterfaceB",
Description: "Only allowed in parents with interface B")]
// This will only allow children that implement this interface.
[AllowAsChildIn(typeof(IInterfaceB))]
public class ChildOnlyParentsWithInterfaceB : TestStep {


}
```

It is possible to programmatically assign children in the parent's constructor, as shown below:

```
public ExampleParentTestStep()
{
    Name = "Parent Step";
    ChildTestSteps.Add(new ExampleChildTestStep { Name = "Child Step 1" });
    ChildTestSteps.Add(new ExampleChildTestStep { Name = "Child Step 2" });
}
```

For examples of parent/child implementations, see:
TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\ParentChild.

Verdict

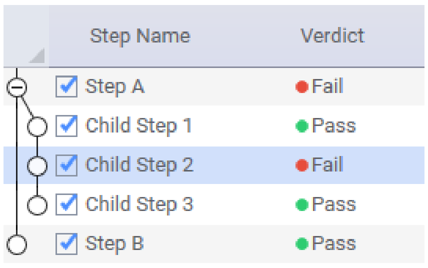
OpenTAP allows steps to be structured in parent/child hierarchy. The OpenTap.Verdict enumeration defines a ‘verdict’ indicating the state and progress of each test step and test plan run. The following table shows the available values for OpenTap.Verdict in increasing order of severity.

Verdict Severity (lowest to highest)	Description
NotSet	No verdict was set (the initial value)
Pass	Step or plan passed
Inconclusive	More information is needed to make a verdict or the results were close to the limits
Fail	Results fail the limits
Aborted	Test plan is aborted by the user
Error	An error occurred; this could be instrument, DUT, software errors, etc.

Each TestStep has its own verdict property. The verdict can be set using the UpgradeVerdict function as shown below (from SetVerdicts.cs):

```
[Display("Set Verdict", Groups: new[] { "Examples", "Plugin Development", "Step Execution" }, Description: "Shows how
the verdict of a step is set using UpgradeVerdict")]
public class SetVerdict : TestStep
{
    #region Settings

    public Verdict MyVerdict { get; set; }
    public double LowerLimit { get; set; }
    public double UpperLimit { get; set; }
    #endregion

    public SetVerdict()
    {
        MyVerdict = Verdict.NotSet;
        LowerLimit = 0;
        UpperLimit = 5;
    }

    public override void Run()
    {
        UpgradeVerdict(MyVerdict);

        var result = 2.5;
        if (result > LowerLimit && result < UpperLimit)
        {
            UpgradeVerdict(Verdict.Pass);
        }
        else UpgradeVerdict(Verdict.Fail);

    }
}
```

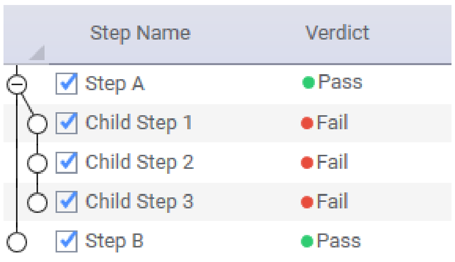
If possible, a test step changes its verdict (often from **NotSet** to one of the other values) during execution. The test step verdict is set to the most severe verdict of its direct child steps, and it is not affected by child steps further down the hierarchy. In the example below, you can see the default behavior, according to which the parent step reflects the most severe verdict of its children.



Step Name	Verdict
✓ Step A	● Fail
✓ Child Step 1	● Pass
✓ Child Step 2	● Fail
✓ Child Step 3	● Pass
✓ Step B	● Pass

The verdict of Step A and Step B affect the test plan verdict. The verdict of Step A is based on the most severe verdict of its child steps. Since *Child Step 2* failed, the verdict of Step A is also *Fail* even though the other two child steps passed. Therefore, the verdict of Step A, and thus the verdict of the test plan, is also fail.

This behavior is expected if the child steps are executed by calling the RunChildSteps/RunChildStep methods. In the case when a different verdict is desired than the one from the child steps, there is a possibility to override the verdict in the parent step. This is useful in cases, where, for example, a recovering strategy like DUT/instrument reboot is handled.



Step Name	Verdict
✓ Step A	● Pass
✓ Child Step 1	● Fail
✓ Child Step 2	● Fail
✓ Child Step 3	● Fail
✓ Step B	● Pass

In the example above, Step A is implemented such that it sets its verdict based on different criteria from the verdict of its child steps.

Log Messages

Log messages provide useful insight to the process of writing and debugging the test step code (as well as other plugin code). The TestStep base class has a predefined Log source, called **Log**. Log messages are displayed in the GUI Editor **Log** panel and saved in the log file.

When creating log messages, the following is recommended:

- Ensure that your logged messages are using the correct log levels. Make use of debug level for less relevant messages.
- Ensure that time-consuming operations write a descriptive message to the log that includes *duration* (to ensure that the operation will be clearly visible in the Timing Analyzer.)

Note: Logs are NOT typically used for RESULTS, which are covered in a different section.

Four levels of log messages — **Error**, **Warning**, **Information**, and **Debug** — allow messages to be grouped in order of importance and relevance. Log messages are shown in the GUI and CLI, and are stored in the session's log file, named `SessionLogs\SessionLog [DateTime].txt` (debug messages are enabled by the –*verbose* command line argument).

By default, log messages for each:

- *Run* are stored in `TAP_PATH\Results` (configurable in the Results settings).
- *Session* are stored in `TAP_PATH\SessionLogs` (configurable in the Engine settings).

Log messages for each run are also available to the ResultListener plugins, as the second parameter on the ResultListeners' OnTestPlanRunCompleted method which looks like it's shown below:

```
void OnTestPlanRunCompleted(TestPlanRun planRun, System.IO.Stream logStream);
```

Note: Users can create their own logs by creating an instance of **TraceSource** as shown in the code below. The *name* used to create the source is shown in the log:

```
Log.Debug("Info from Run");
private TraceSource MyLog = OpenTAP.Log.CreateSource("MyLog");
MyLog.Info("Info from Run");
```

Timestamps and Timing Analysis

The log file contains a timestamp for all entries. This time reflects the time at which the logging method was called. Additionally, it is possible to log time spans/durations of specific actions, such as the time it takes to measure, set up, or send a group of commands.

To log duration, overloads of the `Debug()`, `Info()`, `Warning()` and `Error()` methods are provided. These accept a

`TimeSpan` or a `Stopwatch` instance as the first parameter, as shown in the following code:

```
// The Log can accept a Stopwatch Object to be used for timing analysis
Stopwatch sw1 = Stopwatch.StartNew();
TapThread.Sleep(100);
Log.Info(sw1, "Info from Run");

Stopwatch sw2 = Stopwatch.StartNew();
TapThread.Sleep(200);
Log.Error(sw2, "Error from step");
```

This will result in a log message containing the event text and a time duration tag enclosed in square brackets.

```
12:27:32.883 MyLog     Info from Run [ 100 ms ]
12:27:33.083 MyLog     Error from step [ 201 ms ]
```

The time duration tags make it possible to do more advanced post timing analysis. The Timing Analyzer tool visualizes the timing of all log messages with time stamps.

Exception Logging

Errors are generally expressed as exceptions. Exceptions thrown during test step execution prevent the step from finishing. If an exception is thrown in a test step run method, it will abort the execution of the test step. The exception will be caught by the TestPlan and it will gracefully stop the plan, unless configured to continue in the Engine Settings.

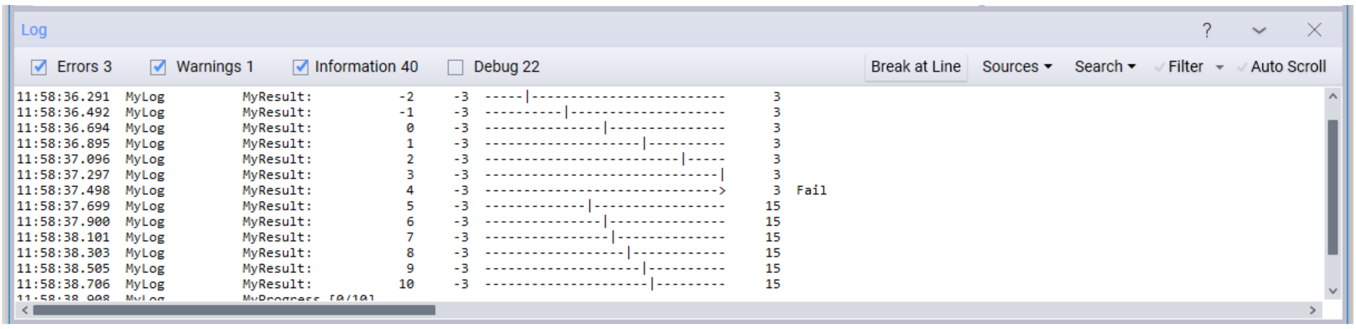
A step can abort the test plan run by calling the `PlanRun.MainThread.Abort()` method. If you have multiple steps in a plan, the engine will check if abort is requested between the `Run()` methods of successive steps. If you have a step which takes a long time to execute, you can call the `OpenTap.TapThread.ThrowIfAborted()` method to check if abort is requested during the execution of the step.

A message is written to the log when a step throws an exception. The log message contains information on source file and line number if debugging symbols (.pdb files) are available, and **Settings > GUI > Show Source Code Links** is enabled.

If an unexpected exception is caught by plugin code, its stacktrace can be logged by calling `Log.Debug(exception)` to provide useful debugging information. The exception message should generally be logged using `Log.Error`, to show the user that something has gone wrong.

TraceBar

The **TraceBar** is a utility class used to display log results and verdicts in the **Log** panel. If an upper and lower limit is available, the TraceBar visually displays the one-dimensional high-low limit sets in a log-friendly graphic:



```
11:58:36.291  MyLog       MyResult:         -2     -3  -----|-------------------------     3
11:58:36.492  MyLog       MyResult:         -1     -3  ----------|--------------------     3
11:58:36.694  MyLog       MyResult:          0     -3  ---------------|---------------     3
11:58:36.895  MyLog       MyResult:          1     -3  --------------------|----------     3
11:58:37.096  MyLog       MyResult:          2     -3  -------------------------|-----     3
11:58:37.297  MyLog       MyResult:          3     -3  ------------------------------|     3
11:58:37.498  MyLog       MyResult:          4     -3  ------------------------------>     3  Fail
11:58:37.699  MyLog       MyResult:          5     -3  -------------|-----------------    15
11:58:37.900  MyLog       MyResult:          6     -3  ---------------|---------------    15
11:58:38.101  MyLog       MyResult:          7     -3  ----------------|--------------    15
11:58:38.303  MyLog       MyResult:          8     -3  ------------------|------------    15
11:58:38.505  MyLog       MyResult:          9     -3  --------------------|----------    15
11:58:38.706  MyLog       MyResult:         10     -3  ---------------------|---------    15
11:58:38.908  MyLog       MyProgress [0/10]
```

Additionally, it handles the verdict of the results. If all the limits passed, the TraceBar.CombinedVerdict is *Pass*; otherwise it is *Fail*. If the result passed to TraceBar is NaN, the verdict will upgrade to Inconclusive. For an example, see the code sample in LogMessages.cs.

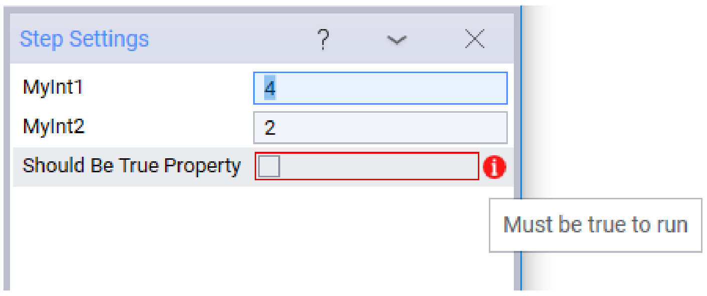
Validation

Validation is customized by adding one or more *Rules* to the constructor of their object. A rule has three parameters: - A delegate to a function that contains the validation logic (may be an anonymous function or a lambda expression) - The message shown to the user when validation fails - The list of properties to which this rule applies

See an example of the use of validation in RuleValidation.cs, as shown below:

```
[Display("RuleValidation Example", Groups: new[] { "Examples", "Feature Library", " Commonly Used" }, Description: "An
example of how Validation works.")]
// Also works for instruments, result listeners, DUTs...., since they all extend
// ValidatingObject
public class RuleValidation : TestStep
{
    #region Settings

    [Display("Should Be True Property", Description: "Value should be true to pass validation.")]
    public bool ShouldBeTrueProp { get; set; }

    public int MyInt1 { get; set; }
    public int MyInt2 { get; set; }

    #endregion

    public RuleValidation()
    {
        // Validation occurs during the constructor.
        // When using the GUI, validation will occur upon editing. When using the engine
        // without the GUI, validation occurs upon loading the test plan.

        // Calls a function that returns a boolean
        Rules.Add(CheckShouldBeTrueFunc, "Must be true to run", "ShouldBeTrueProp");

        // Calls an anonymous function that returns a boolean
        Rules.Add(() => MyInt1 + MyInt2 == 6, "MyInt1 + MyInt2 must == 6", "MyInt1", "MyInt2");

        //Ensure all rules fail.
        ShouldBeTrueProp = false;
        MyInt1 = 2;
        MyInt2 = 2;
    }

    private bool CheckShouldBeTrueFunc()
    {
        return ShouldBeTrueProp;
    }
}
```

The setting as displayed in the GUI looks like this:



MyInt1: 4
MyInt2: 2
Should Be True Property: ☐
Must be true to run

Publishing Results

Publishing results from a test step is a fundamental part of test step execution. The following section discusses publishing results in detail. At a high level, publishing results usually involves a single call, as shown in the following code snippet from
`TAP_PATH\Packages\SDK\Examples\ExamplePlugin\MeasurePeakAmplituteTestStep.cs.`

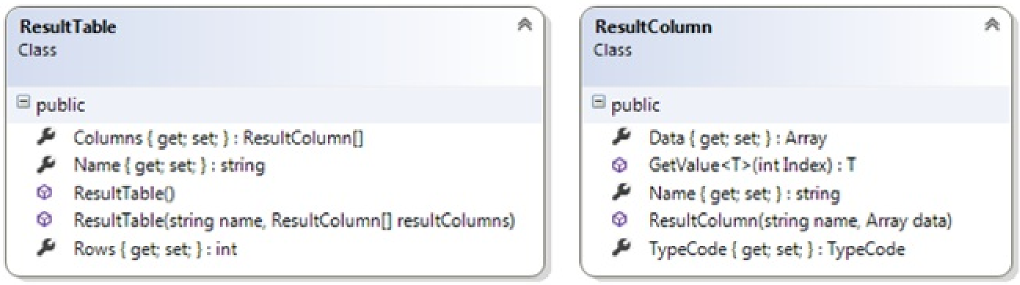
```
InputData = new double[] {0, 0, 5, 5, 5, 50};
ReadOnlyOutputData = new double[]{10, 10, 15, 15, 15, 150};
Results.PublishTable("Inputs vs. Moving Average", new List<string>() {"Input Values", "Output Values"},
     InputData, ReadOnlyOutputData);
```

Result Tables

Test step results are represented in a ResultTable object. A ResultTable consists of a name, one to N columns, and one to M rows. Each test step typically publishes one uniquely named table. Less frequently (but possible), a test step will publish K tables with different names, row/column definitions and values. Each ResultTable is passed to the configured ResultListeners for individual handling.

Result Table Details

This graphic shows the ResultTable definition.



ResultTable
Class

public
- Columns { get; set; } : ResultColumn[]
- Name { get; set; } : string
- ResultTable()
- ResultTable(string name, ResultColumn[] resultColumns)
- Rows { get; set; } : int

ResultColumn
Class

public
- Data { get; set; } : Array
- GetValue<T>(int Index) : T
- Name { get; set; } : string
- ResultColumn(string name, Array data)
- TypeCode { get; set; } : TypeCode

An example of a result table could be a measurement of Power over Frequency. In this case the Result Table name could be “Power over Frequency” and it could contain two Columns, one “Power [W]” column and one “Frequency [Hz]” column. Each column would have the same number of elements, for example 1000, and this would also be the value of the ‘Rows’ property.

ResultSource Object

A ResultSource object (named Results in the test step base class) and its publish methods push result tables to the configured ResultListeners.

There are three major considerations for publishing results:

- What is the “shape” of your results? Is it a single name/value pair, a single “row” or a set of name/value pairs of data and N rows?
- How fast do you want to store the results?
- What table and column names do you wish to use?

The following **ResultSource.Publish** methods are available:

Method Name	General Use	Scope
`Publish<T>(T result)`	For a type T, publishes all the public scalar properties as a single row with N columns. The names of the properties become the column names. The values become the row values. The table name will be the name of the type T, unless overridden by the Display attribute.	Single Row
`Publish<T>(string name,  T result)`	Similar to the previous method, but assigns a unique name to the table name.	Single Row
`Publish(string name, List<string> columnNames, params IConvertible[] results)`	Publishes a row of data with N column names, and N values. The number of columnNames must match the size of the Results array.	Single Row
`PublishTable(string name, List<string> columnNames, params Array[] results)`	Publishes N columns of data, each with M rows. The columnNames parameter defines the ResultTable.ColumnNames property. The results parameter (an array), with N columns, and M rows, is used to populate the N ResultColumn objects, each with an array of data. The size of the columnNames property must match the results array column count. PublishTable: Can be called repeatedly to fill up a table; Is the **fastest** way to store data and should be used when results are large	N Rows

For different approaches to publishing results, see the examples in:

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

Artifacts

Artifacts are another kind of results, originating from test steps or result listeners. Artifacts are files or named streams that can be processed by other result listeners (artifact listeners). This presents an opportunity to do more high-level things with the artifacts, such as telling the user about them, uploading them to the cloud or combining them into more high-level artifacts.

Artifacts can for example be: - A log file. - A CSV file containing measurements. - A screenshot from an instrument. - A waveform file.

Publishing Artifacts

To publish an artifact, call `TestStepRun.PublishArtifact` or `TestPlanRun.PublishArtifact`. If the artifact is associated with a specific test step run, it is strongly recommended to use `TestStepRun.PublishArtifact`.

Most result listeners should publish artifacts. For example, the `CsvResultListener` publishes each CSV file as such:

```
planRun.PublishArtifact(csvFileName);
```

This ensures that it is communicated to OpenTAP which files are associated with the test plan run.

Processing Artifacts

When artifacts are published, events occur in the results processing thread for each result listener that support listening to them.

This way, aggregate artifacts can be created which themselves contains other artifacts. For example, imagine you want to create a HTML report containing tables of results, but also screenshots from the instruments. This can be done by implementing the IArtifactListener interface.

Another example of this is the ZipArtifactsResultListener, which is included in the examples. It is capable of creating a zip file containing all other artifacts from the test plan run and then finally publishing the zip file itself as an artifact.

The lifetime of an artifact varies depending on the artifacts and the environment in which the test plan runs. If it is not wanted for an artifact to stay on the hard drive after the test plan has been run, they can be published as a stream of bytes(MemoryStream) and then deleted. If they are published by name, they will not be automatically deleted.

Implementing An Artifact Listener

To create a result listener that can listen to artifacts, implement the IArtifactListener interface. When implementing IArtifactListener, the following method needs to be defined:

```
void OnArtifactPublished(TestRun run, Stream artifactStream, string artifactName);
```

A few notes about the arguments:

- `run` is the run object to which the artifact is associated. It is either a TestPlanRun or TestStepRun.
- The `artifactStream` object will be disposed after the call to OnArtifactPublished, so it should not be handed over to a different thread for processing.
- `artifactName` is the name of the artifact, including eventual file extensions, such as “.csv” or “.png”. It might not be a file that actually exists.

Child Test Steps

Test step can have any number of child test steps. Exactly which can be controlled by using `AllowAnyChildAttribute`, and `AllowChildOfType`.

To execute child test steps within a test step run, the RunChildStep method can be used to run a single child step or RunChildSteps can be used to run all of them.

Depending on the verdict of the child steps, the break condition setting of the parent step, and the way RunChildSteps is called, there are a number of different ways that the control flow will be affected.

The default implementation should look something like this:

```
public override void Run()
{
    // Insert here things to do before running child steps
    RunChildSteps();
    // Insert here things to do after running child steps.
}
```

`RunChildSteps` will take care of setting the verdict of the calling test step based on the verdict of the child test steps. The default algorithm here is to take the most severe verdict and use that for the parent test step. For example, if the child test steps have verdicts Pass, Inconclusive and Fail, the parent test step will get the Fail verdict.

`RunChildSteps` may throw an exception. This happens if the break conditions for the parent test step are satisfied. If the break conditions are satisfied due to an exception being thrown, resulting in an Error verdict, the same exception will be thrown. To avoid this behavior, the overload `RunChildSteps(throwOnBreak: false)` can be used, but the exceptions are still available from the returned list of test step runs.

To override a verdict set by `RunChildSteps`, the exception must be caught, or `throwOnBreak` must be set. After this, the `Verdict` property can be set directly.

```
public override void Run()
{
    RunChildSteps(throwOnBreak: false);
    Verdict = Verdict.Pass; // force the verdict to be 'Pass'
}
```

Child test steps can be run in a separate thread. This should be done with the TapThread.Start method. Before returning from the parent step, the child step thread should be waited for, or alternatively this should be done in a Defer operation. A great example of using parallelism can be found in the Parallel Step

.

Serialization

Default values of properties should be defined in the constructor. Upon saving a test plan, the test plan's **OpenTAP.Serializer** adds each step's public property to the test plan's XML file. Upon loading a test plan from a file, the OpenTAP.Serializer first instantiates the class with values from the constructor and then fills the property values from the values found in the test plan file.

Because the resource references are declared as properties:

- Their value can be saved and loaded from XML files
- The GUI will support setting the references in a user-friendly way

This convention applies for many different types.

Inputs and Outputs

Inputs and outputs are test step settings that transfer data between test steps during test plan execution. This is useful in situations where one test step depends on a result from another test step, but it can also be used for flow control.

There are two ways to use outputs: - Connecting the output to any fitting setting of a different step. - Using the Input`<T>` class to require an input from another test step.

In both cases, the output setting needs to be marked with the `[Output]` attribute.

Connecting Outputs to Settings

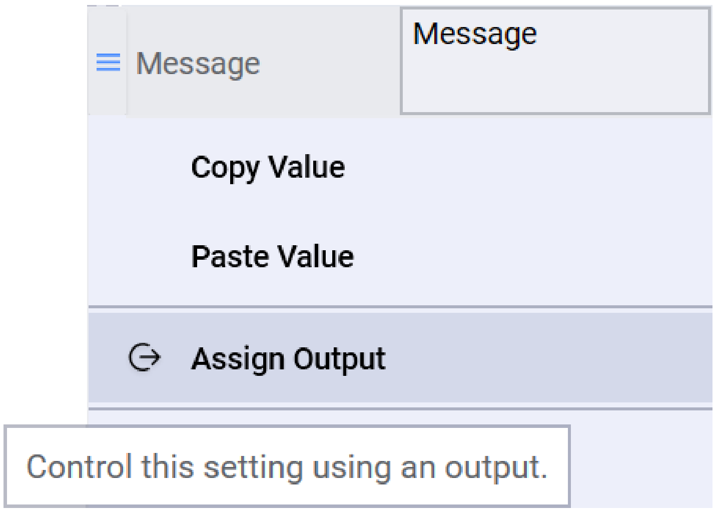
Outputs can be connected to settings of other test steps. For example if one test step has an output property containing an error message, that can be connected to the Message of a dialog test step. That way it is possible to make the user extra aware of the error message.

When connecting outputs to settings, the output property needs to have a type that is compatible with the setting it is connected to. The compatibility of types is both checked at test plan build time and test plan run time. For example, a number output can be connected to a number setting, but it cannot be connected to an Instrument setting. This compatibility is checked at design time so that it is not possible to create the wrong connection in this case. However, a string can be connected to a number setting. In this case, if the string can be converted to a number, the connection will work. If not, an exception will be thrown during runtime and the test step will return an Error verdict.

Below is a matrix of compatible types:

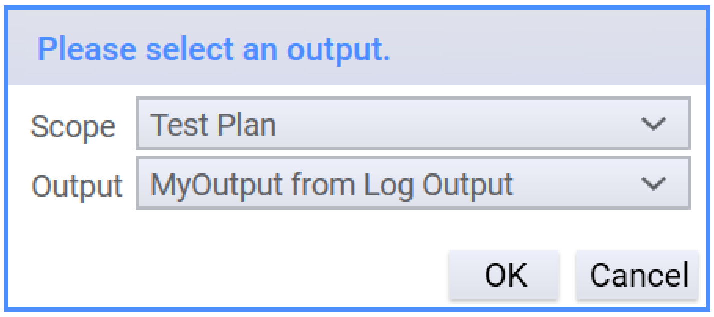
Input / Output	**Number (int,float,…)**	**String**	**Boolean**	**Instrument**	**Subclass of Instrument**	**DUT**
Number	✔	✔	✔	□	□	□
String	✔	✔	✔	□	□	□
Boolean	✔	✔	✔	□	□	□
Instrument	□	□	□	✔	✔	□
Subclass of Instrument	□	□	□	□	✔	□
DUT	□	□	□	□	□	✔

Connecting two settings is generally done through a user interface. Right-click the setting you want to assign an output to and then click "Assign Output".



Assigning an output

Then you will be prompted to select an output.



Selecting an output

You will be able to select the scope of the output and the specific output. The scope refers to some scope in the test plan. Each parent of the test step owning the input is a scope. So the Test Plan scope refers to all the outputs in the test plan at the test plan level. Hence it is not possible to select an output from a child test step. If that is wanted, the output from that child test step should be parameterized to the nearest accessible scope.

Using the `Input<T>` object

The Input class specifies a required input from another test step. The Input class is generic, and the type argument specifies the type of the input. The Input class has a Value property that contains the value of the input. If the input is not connected to an output, the Value property will throw an exception when accessed.

For an example of this, see: - `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\InputOutput`

The generic **Input** class takes one type argument. The Input property references an *Output* of a different step. If no Output is assigned to the Input, the value of the Input is null, and will result in an error.

The **Output** attribute indicates a property that is an output variable. Outputs can be connected to Inputs. Every step has a Verdict property, which is automatically an output property. The Verdict output can be connected to the *If* step, which has an `Input<Verdict>` property.

The following code (from GenerateOutput.cs) shows how to generate **Output** properties.

```
[Output]
[Display("Output Value")]
public double OutputValue { get; private set; }
```

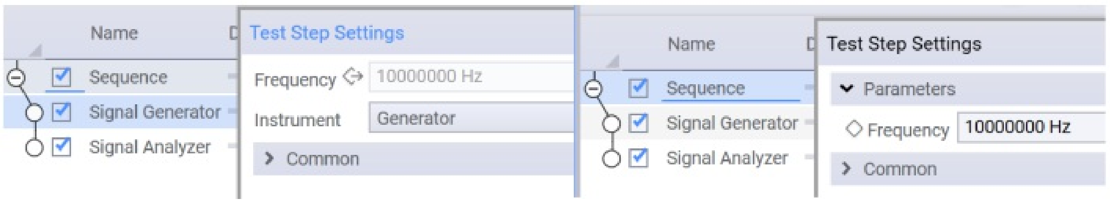
The following code (from HandleInput.cs) shows how to use Input properties. You should use the InputValue.**Value** to access the value contained in the input variable. See the red box below:

```
[Display("Handle Input", Groups: new[] { "Examples", "Feature Library", "InputOutput" },
    Description: "Handles a double input value.")]
public class HandleInput : TestStep
{
    #region Settings
    [Display("Input Value")]
    // Properties defined using the Input generic class will accept values
    // from other (typically prior) test steps with properties that have been
    // marked with the Output attribute.
    public Input<double> InputValue { get; set; }
    #endregion

    public HandleInput()
    {
        InputValue = new Input<double>();
    }

    public override void Run()
    {
        if (InputValue == null) throw new ArgumentException();

        Log.Info("Input Value: " + InputValue.Value);
        UpgradeVerdict(Verdict.Pass);
    }
}
```

Parameterized Settings

In some cases multiple test steps requires using the same setting values. This can be a bit cumbersome to configure and error prone under normal circumstances. To better consolidate settings across many test steps, parameterizations can be used. A setting can be parameterized on any parent test step or the test plan itself. The purpose of the parameterization is to group together test step settings, and make it easier to manage setting values across many different test steps.

For example, let's say a `Signal Generator Test Step` and a `Signal Analyzer Test Step` both has a `Frequency` setting, and it is wanted to make sure that the values of the frequencies are always the same. Each `Frequency` setting can then be Parameterized to the parent test step under the same name, and the parent test step will get a single property representing `Frequency` for both test steps.



Parameterized frequency setting

Notice the parameterized `Frequency` property in the `Signal Generator` step that is marked as read-only. The setting can be modified on the parent `Sequence` step.

In addition, some test steps have special behavior with regards to parameters that are useful to know about.

Sweeping Parameters

The Parameter Sweep test steps utilizes parameterized settings to make it possible to sweep specific test step settings on specific test steps. When a child test steps setting is parameterized onto the Sweep Parameter test step, that parameter can be swept.

- Sweep Parameter Test Step: Sweeps a set of parameters across a table of values defined by the user.
- Sweep Parameter Range Test Step: Sweeps a set of parameters across a range of numeric values. For example, an exponential sweep can be made of values between 1MHz to 1GHz. Or a linear sweep can be done across a number of channels.

Test Plan Parameters

If the setting is parameterized on a test plan, it is sometimes referred to as an "External Parameter" or

"Test Plan Parameter". A test plan parameter can be configured from the command line when "tap run" is being used, for example by specifying `-e "Frequency=10MHz"`. It can also be configured from a test plan reference, as the referenced test plan parameters are promoted onto the Test Plan Reference test step.

ExternalParameterAttribute

The ExternalParameterAttribute marks a property that should be an external parameter (Test Plan Parameter) by default. When the test step is inserted into the test plan, these properties will be automatically assigned to test plan parameters. Note that the setting can be unparameterized manually in case the default behavior is not wanted.

Exceptions

Exceptions from user code are caught by OpenTAP. This could break the control flow, but all resources will always be closed. If the exception gets caught before PostPlanRun, the steps that had PrePlanRun called will also get PostPlanRun called. When a step fails (by setting the Verdict to *Fail* or *Abort*) or throws an exception, execution can be configured to continue (ignoring the error), or to abort execution, by changing the "Abort Run If" property in Engine settings

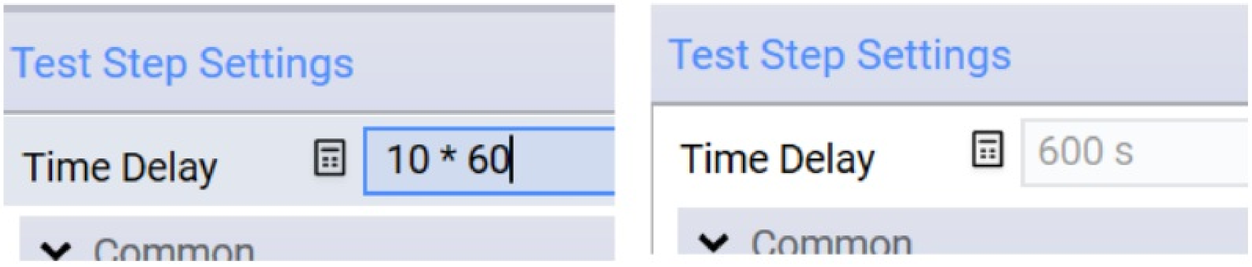
Expressions

Expressions provides users with the ability to use mathematical and functional expressions directly within the test step settings. This offers a more dynamic and flexible approach to configuring properties. The feature supports a range of mathematical functions, constants, and operators, as well as string manipulation.

To use expressions you need the `Expressions` plugin which is normally included in the installation or can otherwise be downloaded from https://packages.opentap.io.

It uses basic mathematical syntax to express relationships between settings or to calculate values or manipulate strings.

For example, let's say we want to use time delay for exactly 10 minutes, we can configure Time Delay as the expression: `10 * 60`, resulting the Time Delay having a value of 600 seconds.



Test Step Settings

Time Delay: 10 * 60

Common

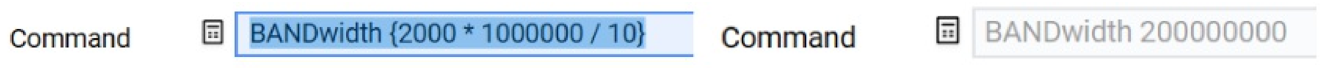
Test Step Settings

Time Delay: 600 s

Common

Setting a number setting with an expression.

For text based settings, expressions can be used to manipulate the text using a `{}` syntax. Take for example a SCPI step and set the expression to: `BANDwidth {2000 * 1000000 / 10}`.



Command: BANDwidth {2000 * 1000000 / 10}

Command: BANDwidth 200000000

Assigning a string expression to a SCPI command.

Basic Syntax

When using the Expressions feature, it's essential to adhere to a specific syntax to ensure accurate evaluation of the expressions. Here's a breakdown of the basic syntax rules:

1. **Numerical Values**:
   - Expressions can contain floating-point numbers like `3.14` or integers like `7`.
   - Avoid using commas or other delimiters for large numbers.

2. **Operators**:
   - Use standard arithmetic operators: `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division.
   - The power operation is represented by `^`.
   - Parentheses `(` and `)` can be used to group expressions or change the precedence of evaluation.
     - Example: `(1 + 2) * 3` evaluates to `9`.
3. **Functions**:
   - Function names are always followed by parentheses `()`.
   - When a function requires multiple arguments, separate each argument with a comma `,`.
     - Example: `max(1, 2, 3)` or `log(8, 2)`.
4. **Identifiers**:
   - Settings on test steps like `Time Delay` is a valid identifier and will get the current value of that setting.
   - Constants like `π`, `pi` or `e` don't require any additional symbols. Use them as you would use a number.
     - Example: `2 * π`.
5. **String Interpolation**:
   - To use expressions with strings, enclose the expression within curly braces `{ }`.
     - Example: `"The radius is {2 * π * r}."`.
6. **Whitespace**:
   - Spaces between numbers, operators, and functions are optional but can make your expression more readable.
   - Avoid adding spaces inside function names or immediately after a function name and before its opening parenthesis.
     - Correct: `max(2, 3)`.
     - Incorrect: `max (2, 3)` or `max( 2, 3 )`.
7. **Case Sensitivity**:
   - While some functions or constants may be case-insensitive, names a generally lower-case. And plugins should try to follow that rule.
8. **Names Escaping**:
   - In cases where your expression includes names that might be confused with built-in functions or constants, you can escape these names using single quotes `' '`.
     - Example: If there's a property named `A/B`, you can distinguish it from the `A / B` expression by writing it as `'A/B'`.

Remember, adhering to the correct syntax is crucial for the expressions to evaluate your input accurately. Ensure that your expressions are well-formed to avoid unexpected results or errors.

Using Expressions

1. **Basic Arithmetic Operations**: You can perform simple mathematical calculations using standard operators such as `+`, `-`, `*`, and `/`.

   Example: `1 * 2 * 3 * 4 * 5` evaluates to `120`.

2. **Functions**:

   - **String Functions**:
     - `empty(str)`: Checks if the provided string `str` is empty or null. Returns `true` or `false`.
       - Example: `empty("")` returns `true`.
   - **Trigonometric Functions**:
     - `sin(v)`: Returns the sine of value `v`.
     - `asin(v)`: Returns the arcsine (inverse of sine) of value `v`.
     - `cos(v)`: Returns the cosine of value `v`.
     - `acos(v)`: Returns the arccosine (inverse of cosine) of value `v`.
     - `tan(v)`: Returns the tangent of value `v`.
     - `atan(v)`: Returns the arctangent of value `v`.
   - **Arithmetic and Rounding Functions**:
     - `abs(v)`: Returns the absolute value of `v`.
     - `floor(v)`: Rounds the value `v` down to the nearest integer.
     - `ceiling(v)`: Rounds the value `v` up to the nearest integer.
     - `round(v)`: Rounds the value `v` to the nearest integer.
     - `round(v, decimals)`: Rounds the value `v` to the specified number of `decimals`.

    - **Sign Function**:
        - `sign(v)`: Returns the sign of `v`. It gives `1` for positive values, `-1` for negative values, and `0` for zero.
    - **MinMax Functions**:
        - `max(a, b)`: Returns the maximum between values `a` and `b`.
        - `max(a, b, c)`: Returns the maximum among values `a`, `b`, and `c`.
        - `max(a, b, c, d)`: Returns the maximum among values `a`, `b`, `c`, and `d`.
        - `min(a, b)`: Returns the minimum between values `a` and `b`.
        - `min(a, b, c)`: Returns the minimum among values `a`, `b`, and `c`.
        - `min(a, b, c, d)`: Returns the minimum among values `a`, `b`, `c`, and `d`.
    - **Logarithmic and Exponential Functions**:
        - `log2(x)`: Returns the base-2 logarithm of `x`.
        - `log10(x)`: Returns the base-10 logarithm of `x`.
        - `log(x, base)`: Returns the logarithm of `x` using the specified `base`.
        - `exp(x)`: Returns the exponential function of `x`, i.e., e raised to the power `x`.

3. **Constants**:

    - `π` or `Pi`: Represents the mathematical constant Pi (π).
        - Example: `π` or `Pi` returns the value of `Math.PI`.

4. **Power Operation**: You can raise a number to the power of another using the `^` operator.

    Example: `2 ^ 3` evaluates to `8`.

5. **String Interpolation**:

    If the target type is a string, you can embed expressions within a string using curly braces `{ }`. The embedded expression will be evaluated, and the result will be placed in the string.

    Example: `The number is {1 + 2}.` becomes “The number is 3.”

Mixins

Mixins are small units of functionality that can be integrated or ‘mixed in’ with an object. The term MixIn comes from object-oriented programming where a mix-in is a class that contains extensions for use by other classes. In the context of OpenTAP, they are be used for extending the capabilities of e.g. a test step.

Getting Started

By default, OpenTAP contains no mixins, but they can be added with packages. Notable packages includes: - Expressions: Adds Number and Text mixins. - Basic Mixins: Contains a number of basic mixins. For example the Repeat mixins, which enables a test step to repeat itself. Limit checks which allows checking a value against a limit. Or Artifact which allows adding a file path pointing to a file that should be considered an artifact of the step.

Adding Mixins as an End User

The mixins are available through the settings context menu. Right-click a setting or in the settings area and click “Add Mixin”. Then a window will pop up guiding you through adding the mixin. Proceeding through this menu will cause the test step to have one or more new settings. To remove the mixin, use the context for any of the new settings and click “remove mixin”.

Mixin Types

Mixins can add functionality in several ways. 1. Adding new settings to the test step. This is the behavior for all mixins. 2. ITestStepPostRunMixin: This type of mixin does something after the step has been executed. 3. ITestStepPreRunMixin: This type of mixin does something before the step has been executed. 4. ITestPlanPreRunMixin: This adds functionality that is executed once during test plan execution.

The way Mixins works is that they add a new dynamic property to the object. This is similar to the way parameters work. The added property can use the EmbedProperties attribute to add more than one additional setting. If the embedding object implements one of the interfaces mentioned above, those will get invoked at the appropriate time.

See for example the following implementation if a ITestStepPostRunMixin.

```
// this defines the mixin. It implements ITestStepPostRunMixin and adds "SomeSetting" to the embedder class.
public class EmbeddingClassMixin : ITestStepPostRunMixin {
    [Display("Some Setting")]
    public string SomeSetting{get;set;} = "123";
    static readonly TraceSource log = Log.CreateSource("test");
    public void OnPostRun(TestStepPostRunEventArgs eventArgs){
        log.Info($"OnPostRun executed. SomeSetting was {SomeSetting}");
    }
}

public class EmbedderTestStep : TestStep {
    // this adds the functionality of embeddingClassMixin to EmbedderTestStep.
    [EmbedProperties]
    public EmbeddingClassMixin Embed {get;} = new EmbeddingClass();
}
```

When you think of mixins you normally think of something added manually, but by using EmbedProperties directly in the test step, you can take advantage of the feature in the implementation as well. This could be used for making your code more composable and for sharing functionality between different test steps without resorting to inheritance.

Adding new Mixin Plugins

To make a new Mixin available to the user, the IMixinBuilder interface needs to be implemented. This is done once for each new type of mixin. The implementation for doing this is a bit advanced because it is reflection-heavy.

```
// This builds a bit on the example from before.
[Display("My Mixin Name", "This is an example of a mixin.")]
// the MixinBuilderAttribute must be used to specify which subclasses the mixin supports. ITestStepParent, covers both
test steps and test plans.
[MixinBuilder(typeof(ITestStepParent))]
public class MyMixinBuilder : IMixinBuilder {
    // add more settings here.
    public string MemberName{get;set;} = "";

    public MyMixinBuilder()
    {
        // if MyMixinBuilder implemented ValidatingObject, rules should be added here.
    }


    public void Initialize(ITypeData type){
        // here you can initialize the settings
        // and get the names of other types to avoid name collision with other mixins or settings.

    }

    public MixinMemberData ToDynamicMember(ITypeData targetType){
        // create a new mixin memberdata nad define how the initial value is created.
        return new MixinMemberData(this, () => new EmbeddingClassMixin()){
            // set the name of the member (this is hidden from the user)
            Name = "MyMixinBuider:" + MemberName,
            // Set the type to the target type of the property.
            TypeDescriptor = TypeData.FromType(typeof (EmbeddingClassMixin)),
            // Add the DisplayAttribute to form the group
            Attributes = new object[]{new DisplayAttribute(MemberName),
            // Add EmbedPropertiesAttribute
            new EmbedPropertiesAttribute()},
            DeclaringType = TypeData.FromType(typeof(ITestStep))
        };
    }
}
```

After adding this class, the user should have access to the mixin type in the context menu of the test step.

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:

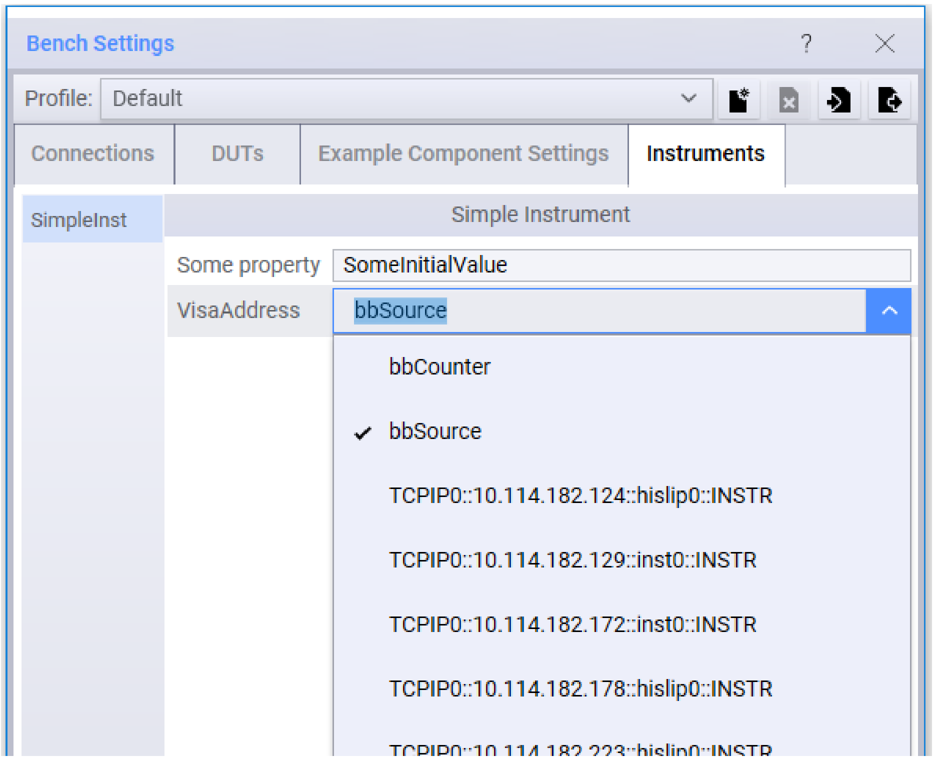
- `ScpiCommand`, which sends a command.
- `ScpiQuery`, which sends the query and returns the results.
- `VisaAddress`, which specifies the Visa address of the instrument.
- `ScpiQueryBlock<T>`, which sends the block query, and parses the binary block as an array of type T. All numeric types except Decimal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



Some property: SomeInitialValue

VisaAddress: bbSource

bbCounter
✓ bbSource
TCPIP0::10.114.182.124::hislip0::INSTR
TCPIP0::10.114.182.129::inst0::INSTR
TCPIP0::10.114.182.172::inst0::INSTR
TCPIP0::10.114.182.178::hislip0::INSTR
TCPIP0::10.114.182.223::hislip0::INSTR

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.

Result Listener

OpenTAP comes with a number of predefined result listeners, summarized in the following table.

Group	Name	Description
Action	**Notifier**	Runs a program or plays a sound based on the verdict of a test plan start and end.
Database	**PostgreSQL**	Stores results into a PostgreSQL database.
Database	**SqLite**	Stores results into an SqLite database.
Text	**CSV**	Stores results into a CSV file. Supported delimiters are semicolon, comma and tab.
Text	**Log**	Stores log messages (NOT results) into a log file. One file is created for each test plan run.

OpenTAP also supports custom results listeners.

Custom Result Listeners

A custom result listener stores the data in a “custom” way. For example, if OpenTAP is to be deployed in a manufacturing shop that has a preexisting data storage system, you can create a custom result listener to interface with that system.

To create a custom result listener, make a new public class which extends the OpenTAP.ResultListener class. The ResultListener class has virtual methods that are called during test plan execution. Implement only those that are needed for the specific ResultListener implementation. The SDK includes an ExampleResultListener that places a summary of any ResultTables into the log.

For examples, see:

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

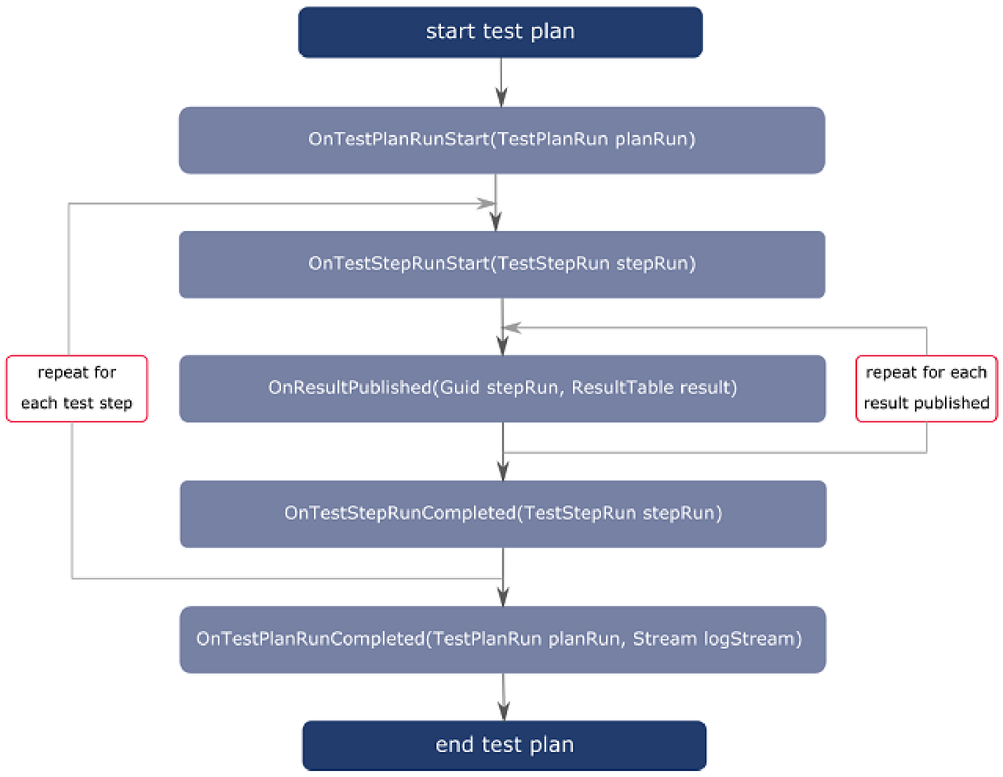
When extending the ResultListener class, the following methods can be overwritten:

- `public override void OnTestPlanRunStart(TestPlanRun planRun)`
- `public override void OnTestStepRunStart(TestStepRun stepRun)`
- `public override void OnResultPublished(Guid stepRun, ResultTable result)`
- `public override void OnTestStepRunCompleted(TestStepRun stepRun)`
- `public override void OnTestPlanRunCompleted(TestPlanRun planRun, Stream logStream)`

These methods are called by OpenTAP and are guaranteed to be called in a certain order:

1. **OnTestPlanRunStart** - Called when the test plan starts and all resources have been opened (including Result Listeners). It takes a TestPlanRun argument, containing parameters for the plan run.
2. **OnTestStepRunStart** - Called for each step when the step execution starts.
3. **OnResultPublished** - Called for each published result from the test step. This method is not called at all for test steps which do not publish any results.
4. **OnTestStepRunCompleted** - Called for each step when the step execution stops. At this point it is guaranteed that the test step does not publish more results.
5. **OnTestPlanRunCompleted** - Called when the test plan run finishes. This is called once during a test plan execution. The TestPlanRun object contains information regarding the run, including the duration and final verdict. The LogStream contains the log file produced by the plan run.

To illustrate the sequence of the above-mentioned methods, consider the following figure:


start test plan
OnTestPlanRunStart(TestPlanRun planRun)
OnTestStepRunStart(TestStepRun stepRun)
repeat for each test step
OnResultPublished(Guid stepRun, ResultTable result)
repeat for each result published
OnTestStepRunCompleted(TestStepRun stepRun)
OnTestPlanRunCompleted(TestPlanRun planRun, Stream logStream)
end test plan

Note that if a step publishes multiple results the OnResultPublished method is called multiple times after the OnTestStepRunStart method. If there are multiple test steps the OnTestStepRunStart and OnTestStepRunCompleted methods are called as many times as the number of test steps.

The five methods described here run on a separate thread, and do NOT run synchronously with the test plan. That is important because the design of this class is to handle results, but NOT specifically to use the above methods to control external operations. The non-synchronous behavior was designed to allow faster throughput.

It is possible to abort the test plan execution in case the result listener fails or encounters an error. This is done by calling the `PlanRun.MainThread.Abort()` method.

OpenTAP SQL Database

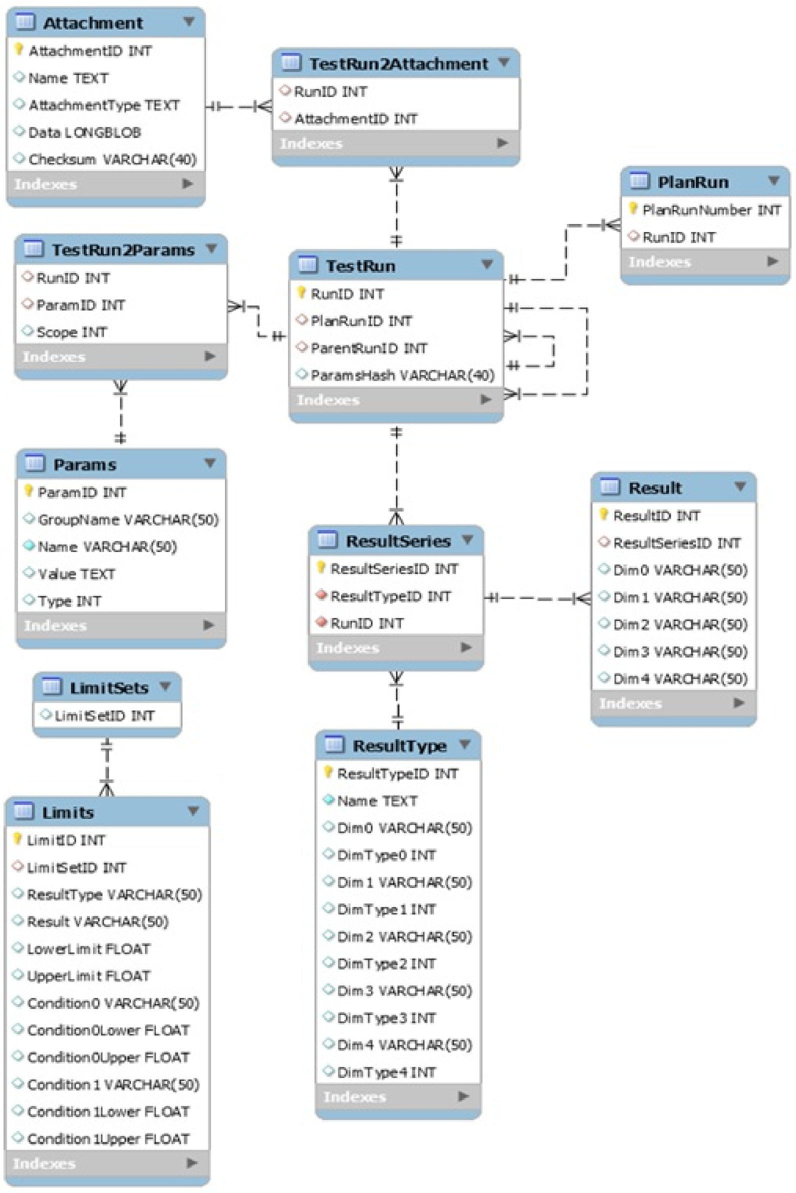
Interacting with the predefined databases or interfacing to an unsupported database requires DB knowledge.

When the user runs a test plan for the first time, the entire test plan file (XML) is automatically compressed and saved together with the test plan name in the Attachment database table, seen in the figure below. At the same time, the given test plan run is registered in the TestRun table, with time of execution and a reference to the test plan saved in the Attachment table. The log file of the test plan run is stored in the same table.

The TestRun table gets a new entry for each executed step in the test plan. The PlanRunID table points at the TestRun entry of the test plan run. The ParentRunID points to the entry of the parent step’s run (if there is any). Associated with each TestRun, you can store multiple result series in the ResultSeries table. This points to the ResultType table, which contains additional information about the result data such as titles for the data columns/axis (x,y,z). The TestRun table points via the TestRun2Params to a range of different parameters in the Params table. These parameters become the public properties of the test step when the result series are generated.

Lastly, ResultSeries is pointed to by the Results table, so that each row in ResultSeries can have many results belonging to it.

The OpenTAP database schema consists of 11 tables, shown in the following diagram. You should explore the schema of the sample databases before attempting to write a new result listener targeting a new database.

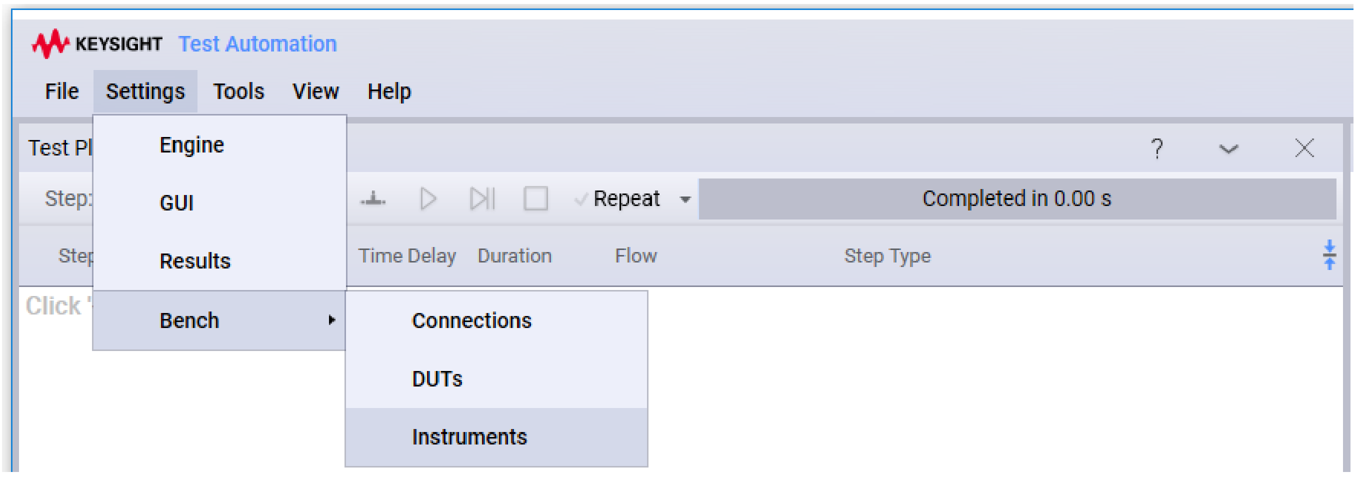

Attachment
AttachmentID INT
Name TEXT
AttachmentType TEXT
Data LONGBLOB
Checksum VARCHAR(40)
Indexes
TestRun2Attachment
RunID INT
AttachmentID INT
Indexes
PlanRun
PlanRunNumber INT
RunID INT
Indexes
TestRun2Params
RunID INT
ParamID INT
Scope INT
Indexes
TestRun
RunID INT
PlanRunID INT
ParentRunID INT
ParamsHash VARCHAR(40)
Indexes
Params
ParamID INT
GroupName VARCHAR(50)
Name VARCHAR(50)
Value TEXT
Type INT
Indexes
Result
ResultID INT
ResultSeriesID INT
Dim0 VARCHAR(50)
Dim1 VARCHAR(50)
Dim2 VARCHAR(50)
Dim3 VARCHAR(50)
Dim4 VARCHAR(50)
Indexes
ResultSeries
ResultSeriesID INT
ResultTypeID INT
RunID INT
Indexes
LimitSets
LimitSetID INT
ResultType
ResultTypeID INT
Name TEXT
Dim0 VARCHAR(50)
DimType0 INT
Dim1 VARCHAR(50)
DimType1 INT
Dim2 VARCHAR(50)
DimType2 INT
Dim3 VARCHAR(50)
DimType3 INT
Dim4 VARCHAR(50)
DimType4 INT
Indexes
Limits
LimitID INT
LimitSetID INT
ResultType VARCHAR(50)
Result VARCHAR(50)
LowerLimit FLOAT
UpperLimit FLOAT
Condition0 VARCHAR(50)
Condition0Lower FLOAT
Condition0Upper FLOAT
Condition1 VARCHAR(50)
Condition1Lower FLOAT
Condition1Upper FLOAT
Indexes

Component Setting

Global and Grouped Settings

In addition to the settings defined for test steps, DUTs, and instruments, there are several “built in” settings collections. These can be divided into two categories: *Global* and *Grouped*.

- **Global** settings are shown under the **Settings** menu. At a minimum, you will find settings for the **Engine**, **Editor**, and **Results**.
- **Grouped** settings are applicable to a particular configuration profile and are shown under the **Settings > GroupName** menu. The Bench settings (shown under the **Settings > Bench** menu) is an example of grouped settings.

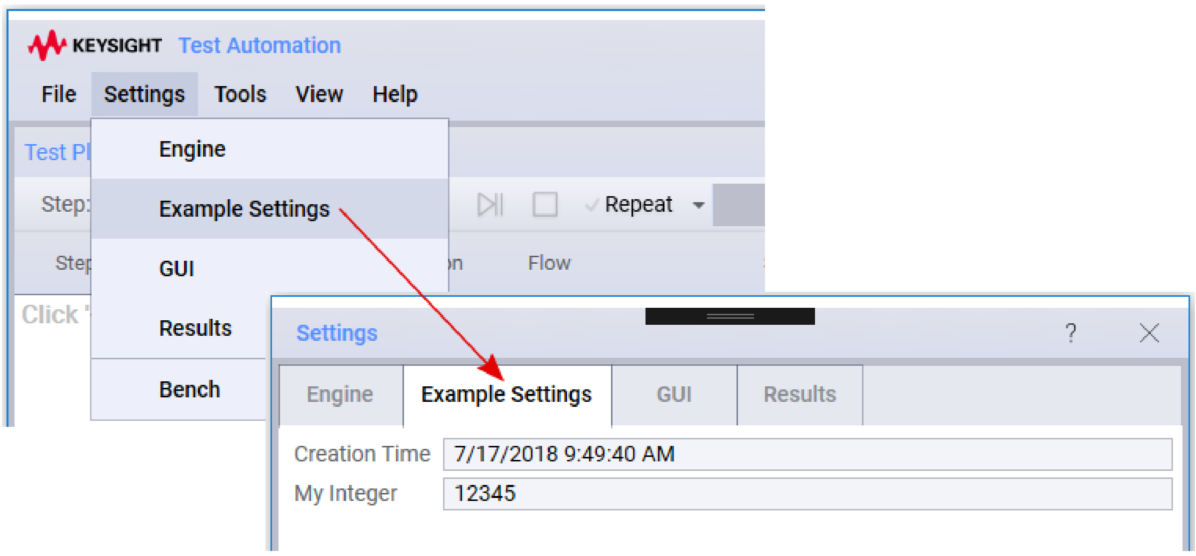


Creating a New Global/Grouped Settings Dialog

OpenTAP developers can create their own settings dialogs under the **Settings** or **Settings > GroupName** menus. By default, the dialog appears under the **Settings** menu. If the class is decorated with the `[SettingsGroup("GroupName")]` attribute, the dialog will appear under the **Settings > GroupName** menu. It is also possible to extend the bench settings by decorating a class with the `[SettingsGroup("Bench")]` attribute.

Single Instance of Multiple Different Settings

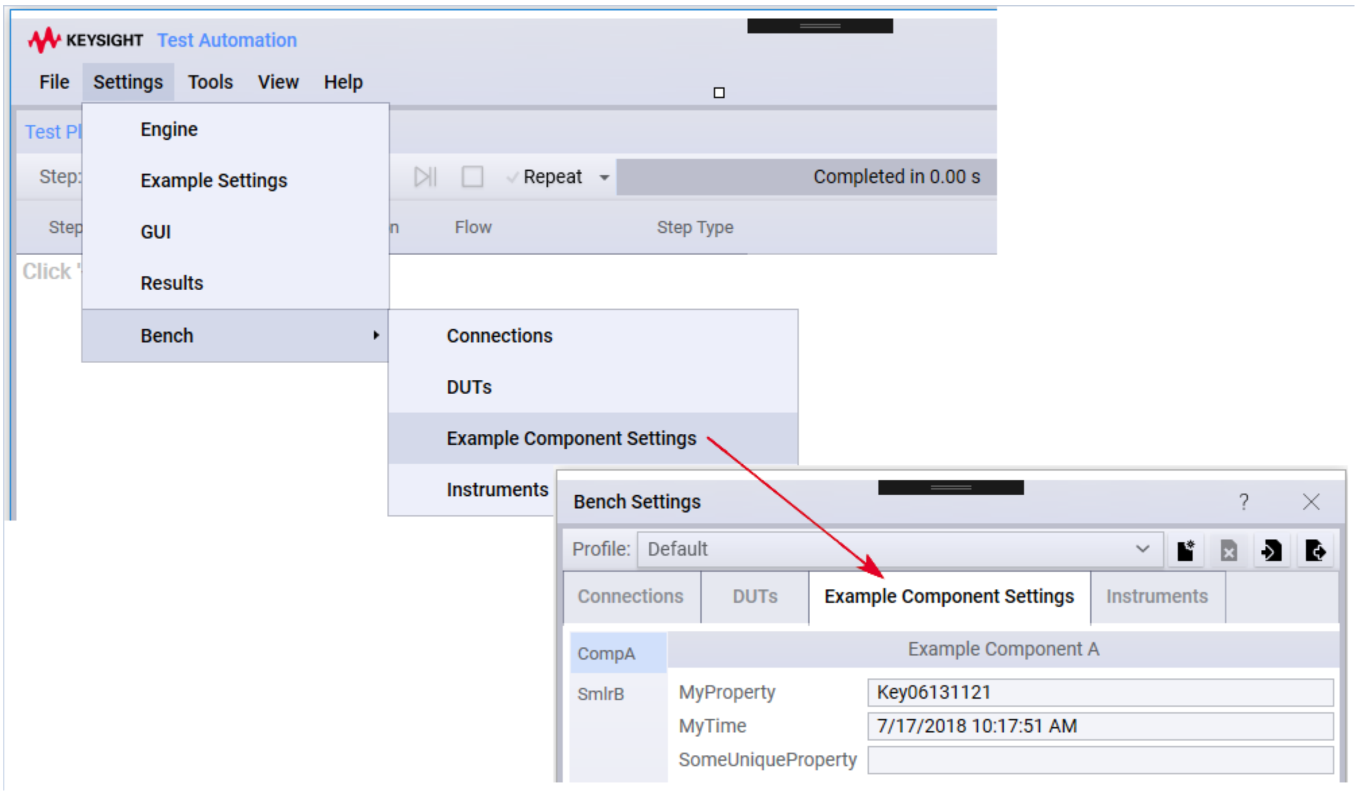
If you want to create a dialog consisting of multiple settings, you should inherit from the ComponentSettings class. See the `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\GUI\ExampleSettings.cs` file. The result looks like this:



Creation Time 7/17/2018 9:49:40 AM
My Integer 12345

List of Similar Settings

Suppose you are trying to list several objects that are slightly different but share a common base class. This is similar to what is used in the DUT or Instrument settings dialog. To do so, you should inherit from the ComponentSettingsList class. See the CustomBenchSettings.cs file. The results (with several instances created), are shown below:


[redacted]
[redacted]

MyProperty	Key06131121
MyTime	7/17/2018 10:17:51 AM
SomeUniqueProperty	

Reading and Writing Component Settings

The SettingsRetrieval.cs file demonstrates different approaches for reading component settings, as shown in the code below:

```
[Display("Settings Retrieval", Groups: new[] { "Examples", "Feature Library", "Step Execution" }, Order: 10000,
Description: "Shows how to retrieve settings.")]
public class SettingsRetrieval : TestStep
{
    public override void Run()
    {
        // These settings always exist
        Log.Info("Component Settings directory={0}", ComponentSettings.SettingsDirectoryRoot);
        Log.Info("Session log Path={0}", EngineSettings.Current.SessionLogPath);
        Log.Info("Result Listener Count={0}", ResultSettings.Current.Count);

        if (DutSettings.Current.Count > 0)
        {
            string s = DutSettings.GetDefaultOf<Dut>().Name;
            Log.Info("The first dut found has a name of {0}", s);
        }

        if (InstrumentSettings.Current.Count > 0)
        {
            string s = InstrumentSettings.GetDefaultOf<Instrument>().Name;
            Log.Info("The first instrument found has a name of {0}", s);
        }

        // An example of user defined settings, which show up as individual tabs
        // Default values will be used, if none exist.
        Log.Info("DifferentSettings as string={0}", ExampleSettings.Current.ToString());

        // An example of custom Bench settings.
        // This is similar to the DUT or Instrument editors.
        // Only use the values if something exists.
        if (CustomBenchSettingsList.Current.Count > 0)
        {
            Log.Info("Custom Bench Settings List Count={0}", CustomBenchSettingsList.Current.Count);
            Log.Info("First instance of Custom Bench setting as string={0}",
                    CustomBenchSettingsList.GetDefaultOf<CustomBenchSettings>());
            foreach (var customBenchSetting in CustomBenchSettingsList.Current)
            {
                Log.Info("Type={0} Time={1} MyProperty={2}", customBenchSetting.GetType(),
                customBenchSetting.MyTime, customBenchSetting.MyProperty);
            }
        }
    }
}
```

Connection Management

An OpenTAP *connection* represents a physical connection between Instrument and/or DUT ports. A *physical connection* is modeled in **software** by creating a class that extends the **OpenTap.Connection** abstract base class.

A *port* is the endpoint of a connection. Ports are often defined on Instruments or DUTs to represent physical connectors. An instrument or DUT can have an arbitrary number of ports.

OpenTAP comes with a number of predefined connections. These include:

- **RfConnection**, which extends the OpenTAP.Connection class, consists of two ports (inherited from the OpenTAP.Connection), and adds the concept of CableLoss, which is a list of LossPoints. An RfConnection represents a physical RF cable with known loss characteristics by using the list of loss points to represent the cable loss at different frequencies.
- **DirectionalRfConnection**, which extends the RfConnection class to include direction.

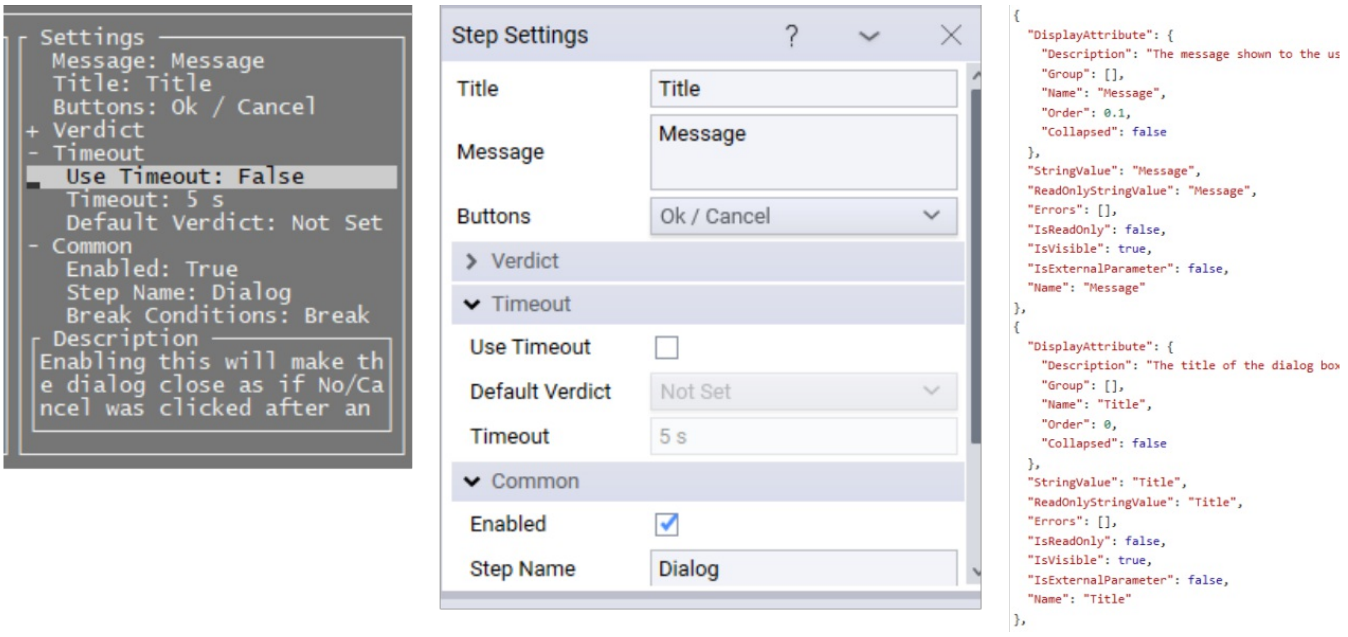
For more information, see the *Bench Settings - Connections* topic in the GUI Editor help, or contact support.

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’.



```
┌ Settings ─────────────────────────────
   Message: Message
   Title: Title
   Buttons: Ok / Cancel
 + Verdict
 - Timeout
     Use Timeout: False
     Timeout: 5 s
     Default Verdict: Not Set
 - Common
     Enabled: True
     Step Name: Dialog
     Break Conditions: Break
┌ Description ──────────────────
Enabling this will make th
e dialog close as if No/Ca
ncel was clicked after an
```

Step Settings

Title	Title
Message	Message
Buttons	Ok / Cancel
Verdict	
Timeout	
Use Timeout	☐
Default Verdict	Not Set
Timeout	5 s
Common	
Enabled	✓
Step Name	Dialog

```
{
  "DisplayAttribute": {
    "Description": "The message shown to the us
    "Group": [],
    "Name": "Message",
    "Order": 0.1,
    "Collapsed": false
  },
  "StringValue": "Message",
  "ReadOnlyStringValue": "Message",
  "Errors": [],
  "IsReadOnly": false,
  "IsVisible": true,
  "IsExternalParameter": false,
  "Name": "Message"
},
{
  "DisplayAttribute": {
    "Description": "The title of the dialog box
    "Group": [],
    "Name": "Title",
    "Order": 0,
    "Collapsed": false
  },
  "StringValue": "Title",
  "ReadOnlyStringValue": "Title",
  "Errors": [],
  "IsReadOnly": false,
  "IsVisible": true,
  "IsExternalParameter": false,
  "Name": "Title"
},
```

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 implement, if dynamic members for a given type need

Interface	Name	Description
		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.

Plugin Packaging and Versioning

Packaging

A OpenTAP Package is a file that contains plugin DLLs and supporting files. Packages are used to distribute OpenTAP plugins, while providing support for correct versioning and dependency checking. This section deals with the construction and use of OpenTAP packages. The different programs and processes involved are described below:

- The OpenTAP installation includes the **Package Manager**, accessible by the `tap package` command. This can be used to create, install or uninstall packages, list installed packages, and run tests on one or more packages.
- The GUI Editor installation also includes the **PackageManager.exe** program which is a GUI for the Package Manager. It permits package downloading, displays an inventory of the packages, and ultimately installs package files found into the OpenTAP install directory.
- The default OpenTAP plugin project (release builds only) includes an *AfterBuild* task for creating an OpenTAP Package based on package declarations in the **package.xml** file. The resulting OpenTAP package has the **.TapPackage** suffix. Files with this suffix are renamed zip files, and as such, can be examined with a file compressor and archiver software, such as WinZip.

When run from Visual Studio, most of the processes of the packaging system are automatic and invisible to the operation. However, the developer may wish to modify the content and/or properties of the package by editing the package.xml file. The following package.xml is found in `TAP_PATH\Packages\SDK\Examples\ExamplePlugin`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
InfoLink: Specifies a location where additional information about the package can be found.
Version: The version of the package. Must be in a semver 2.0 compatible format. This can be automatically updated from
GIT.

For Version the following macro is available (Only works if the project directory is under Git source control):
$(GitVersion) - Gets the version from Git in the recommended format Major.Minor.Build-PreRelease+CommitHash.BranchName.
-->
<Package Name="Example Plugin"
         xmlns="http://opentap.io/schemas/package"
         InfoLink="http://www.keysight.com/"
         Version="0.1.0-alpha"
         Tags="sdk example">
  <Description>Example plugin containing Instrument, DUT and TestStep.</Description>
  <Files>
    <File Path="Packages/Example Plugin/OpenTap.Plugins.ExamplePlugin.dll">
      <!--SetAssemblyInfo updates assembly info according to package version.-->
      <SetAssemblyInfo Attributes="Version"/>
    </File>
    <File Path="Packages/Example Plugin/SomeSampleData.txt"></File>
  </Files>
</Package>
```

A package that references an OpenTAP assembly version 9 is compatible with any OpenTAP version 9.y, but not compatible with version 8 or earlier, or a future version 10. The Package Manager checks version compatibility before installing packages.

Packaging Configuration File

When creating a package the configuration is specified using an XML file (typically called package.xml).

Attributes in the Configuration File

The root element of the configuration file is `Package` and it supports the following optional attributes:

Attribute	Description
Name	We advise to make the names human readable, i.e. not Camel Case, no underscores, etc. No need to add “plugin” in the package name, as OpenTAP packages typically contain OpenTAP plugins. The package name determines where the package definition will be installed, relative to the Packages directory in the OpenTAP installation. E.g. the package definition of a package named `My Dut Driver` will be installed to `./Packages/My Dut Driver/package.xml`. Forward slashes are strongly discouraged because they can’t be published to the package repository. The following characters are not allowed: `"`, `<`, `>`, `\\|`, `\0`, `\u0001`, `\u0002`, `\u0003`, `\u0004`, `\u0005`, `\u0006`, `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\u000e`, `\u000f`, `\u0010`, `\u0011`, `\u0012`, `\u0013`, `\u0014`, `\u0015`, `\u0016`, `\u0017`, `\u0018`, `\u0019`, `\u001a`, `\u001b`, `\u001c`, `\u001d`, `\u001e`, `\u001f`, `:`, `*`, `?`, `\`.
InfoLink	Specifies a location where additional information about the package can be found. It is visible in the Package Manager as the **More Information** link.
Version	The version of the package. This field supports the $(GitVersion) macro. The version is displayed in the Package Manager. See Versioning for more details.
OS	Which operating systems the package is compatible with. This is a comma separated list. It is used to filter packages which are compatible with the operating system the Package Manager is running on. If the attribute is not specified, the default, Windows, is used. Example: `OS="Windows,Linux"`. The following OS values are currently supported by the package manager for automatic detection: Windows, Linux and OSX. Using one of these is recommended.
Architecture	Used to filter packages which are compatible with a certain CPU architecture. If the attribute is not specified it is assumed that the plugin works on all architectures. The available values are AnyCPU, x86, x64 (use for AMD64 or x86-64), arm and arm64.
Class	This attribute is used to classify a package. It can be set to **package**, **bundle** or **system-wide** (default value: **package**). A package of class **bundle** references a collection of OpenTAP packages, but does not contain the referenced packages. Packages in a bundle do not need to depend on each other to be referenced. For example, Keysight Developer’s System is a bundle that reference the Editor (GUI), Timing Analyzer, Results Viewer, and SDK packages. A package of class **system-wide** is installed in a global system folder so these packages can affect other installations of OpenTAP and cannot be uninstalled with the Package Manager. System-wide packages should not be OpenTAP plugins, but rather drivers and libraries. The system folders are located differently depending on operating system and drive specifications: Windows (normally) - `C:\ProgramData\Keysight\OpenTAP`, Linux - `/usr/share/Keysight/OpenTAP`
Group	Name of the group that this package belongs to. Groups can be nested in other groups, in which case this string will have several entries separated with ‘/’ or ‘\’. The attribute may be empty. UIs may use this information to show a list of packages as a tree structure.
Tags	A list of keywords that describe the package. Tags are separated by space or comma.
LicenseRequired	License key(s) required to use this package. During package create all `LicenseRequired` attributes from the `File` Elements will be concatenated into this property. Bundle packages (`Class` is ‘bundle’) can use this property to show license keys that are required by the bundle dependencies.

OpenTAP does not validate any `LicenseRequired` attributes. This attribute is only used by UIs to inform the user of a license key. The license key check should be implemented by the plugin assembly.

Elements in the Configuration File

The configuration files can also contain certain elements. These elements add additional information about the package and can have their own attributes.

Description Element

The **Description** element can be used to write a short description about the plugin. Custom elements like `Organization` or `Status` can be added the provide additional highlighted information. For example: For example:

```
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
  <Description>
    This is an example of a "package.xml" file.
    <Status>Released</Status>
    <Organisation>Keysight Technologies</Organisation>
  </Description>
  ...
</Package>
```

Owner Element

The **Owner** element inside the configuration file is the name of the package owner. There can be multiple owners of a package, in which case this string will have several entries separated by ','. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
  ...
  <Owner>OpenTAP</Owner>
  ...
</Package>
```

SourceUrl Element

The **SourceUrl** element in the configuration file is a link to the package source code. This is intended for open sourced projects.

SourceLicense Element

The license of the open source project. Must be a SPDX identifier .

Dependency Element

OpenTAP will automatically add dependencies to other packages if they are referenced in the plugin code. In some cases, it is necessary to add dependencies to packages that are not referenced in this way. The **Dependency** element can be used to manually specify such dependencies:

```
<Package Name="MyPackage">
  <Description>My Plugin Package.</Description>
  <Dependencies>
    <PackageDependency Package="CSV" Version="^9.1" />
    <PackageDependency Package="Demonstration" Version="^9.3" />
  </Dependencies>
</Package>
```

File Element

The File element inside the Files element denotes files that are included inside the package file. Any type of file can be added to be inserted anywhere in the deployment folder.

If the file is a .NET DLL, DLL reference dependencies will automatically get included in the package, for example if your project references System.Text.Json.dll, the DLL will be added automatically to the

TapPackage file. So this is how the package.xml file looks after creating the package.

```
<Files>
    <File Path="Packages/MyPlugin/OpenTAP.Plugins.MyPlugin.dll">
        <!-- This plugin file 'needs' System.Text.Json.dll -->
    </File>
    <File Path="Dependencies/System.Text.Json.4.0.1.2/System.Text.Json.dll">
        <!-- This dependency is automatically added when the package is created.-->
    </File>
</Files>
```

If it is not wanted to include a .NET DLL dependency, the `IgnoreDependency` element can be added. See the example below.

The **File** element inside the configuration file supports the following attributes:

Attribute	Description
Path	The path to the file. This is relative to the root of the OpenTAP installation directory. This serves as both source (from where the packaging tool should get the file when creating the package) and target (where the file should be located when installed). Unless there are special requirements, the convention is to put all payload files in a `Packages/<PackageName>` subfolder. Wildcards are supported - see later section.
SourcePath	Optional. If present the packaging tool will get the file from this path when creating the package.
LicenseRequired	Indicates if a license key is required by the package file. This is for information only and is not enforced by OpenTAP. The license key check should be implemented by the plugin assembly.

The **File** element can optionally contain custom elements supported by OpenTAP packages. The example below includes the `SetAssemblyInfo` element, which is supported by the OpenTAP package. When `SetAssemblyInfo` is set to `Version`, AssemblyVersion, AssemblyFileVersion and AssemblyInformationalVersion attributes of the file are set according to the package's version.

```
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
      ...
 <Files>
    <File Path="Packages/MyPlugin/OpenTAP.Plugins.MyPlugin.dll">

      <!-- Set the Assembly Version to the same version as this package (GitVersion). -->
      <SetAssemblyInfo Attributes="Version"/>

      <!-- Ignore the System.Text.Json.dll DLL dependency. -->
      <IgnoreDependency>System.Text.Json</IgnoreDependency>
    </File>
    <File Path="Packages/MyPlugin/waveform1.wfm"/>
    <File Path="Packages/MyPlugin/waveform2.wfm"/>
  </Files>
  ...
</Package>
```

Hidden Folders and Files

To make folders and files hidden, add a dot (.) in front of their names. In Linux, files and folders starting with `.` are implicitly hidden. In Windows, `tap package install` will unpack assemblies to the destination folder, those files and folders starting with a `.` will have their file attribute set to hidden.

Package Icon

A package can also include a package icon. The **File** element inside the configuration file supports adding a package icon by using the `Path` attribute to point to an image and using the `PackageIcon` element inside the `File` element. For example:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
     ...
    <File Path="Packages/MyPlugin/Example Icon.ico">
      <PackageIcon/>
    </File>
  ...
</Package>
```

Wildcards

It is possible to include multiple files using only a single **File** element using *wildcards* (file globbing). When using a wildcard in a **File** element's **Path** attribute, the element is replaced with new **File** elements representing all the files that match the pattern when the packaging tool is run. The following wildcards are supported:

Wildcard	Description	Example	Matches
*	Matches any number of any characters including none.	Law*	Law, Laws, or Lawyer
?	Matches any single character.	?at	Cat, cat, Bat or bat
**	Matches any number of path / directory segments. When used must be the only contents of a segment.	/**/some.*	/foo/bar/bah/some.txt, /some.txt, or /foo/some.txt.

When using wildcards in the **Path** attribute, the **SourcePath** attribute has no effect. All matching **File** elements will have all the same child elements as the original wildcard element. So this feature could be applied to the XML as such:

```
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
...
  <Files>
    <File Path="Packages/MyPlugin/*.dll">
      <!-- SetAssemblyInfo Applied to all '.dll' files matching the wildcard. -->
      <SetAssemblyInfo Attributes="Version"/>
    </File>
   <!-- All '.wfm' files from the directory are included. -->
   <File Path="Packages/MyPlugin/*.wfm"/>
   <File Path="Packages/MyPlugin/Example Icon.ico">
    <!-- Only one package icon - no wildcard is used. -->
     <PackageIcon/>
   </File>
  </Files>
 ...
 </Package>
```

ActionStep Element

A package can define **ActionStep** elements, which are commands that OpenTAP will run at predefined stages. The **ActionStep** element supports the following attributes:

Attribute	Description
ExeFile	The name of the program to execute. This is always relative to the directory containing the OpenTAP executable.
Arguments	The arguments with which to invoke the ExeFile.
ActionName	The stage at which to run the action.

OpenTAP runs actions at four predefined stages:

1. **ActionName == "install"** is executed *after* a package has finished installing.
2. **ActionName == "prepareUninstall"** is executed before a package is uninstalled *before* OpenTAP has verified that no files are in use.
3. **ActionName == "uninstall"** is executed before a package is uninstalled *after* a package OpenTAP has verified that no files are in use.
4. **ActionName == "test"** is executed when `tap package test MyPlugin` is invoked on the command line.

The difference between **prepareUninstall** and **uninstall** is subtle. **prepareUninstall** can be used to release any resources held by the installation that would otherwise cause the package uninstall to fail. If **prepareUninstall** fails for some reason, the uninstall will be stopped before any files are removed. **uninstall**, on the other hand, can be used to clean up files created by the plugin, but which are not part of the plugin package. This is necessary because OpenTAP cannot track loose files, and will only remove files which are part of the package definition.

A package can contain any number of **ActionStep** elements, but they must be contained in a **PackageActionExtensions** element:

```
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
...
  <Files>
    <File Path="Packages/MyPlugin/OpenTAP.Plugins.MyPlugin.dll">
      <SetAssemblyInfo Attributes="Version"/>
    </File>
     <File Path="Packages/MyPlugin/waveform1.wfm"/>
    <File Path="Packages/MyPlugin/waveform2.wfm"/>
    <File Path="Packages/MyPlugin/waveform-test.TapPlan"/>
    <File Path="./Packages/MyPlugin/WaveformGenerator.exe"/>
    <File Path="Packages/MyPlugin/Example Icon.ico">
  <PackageActionExtensions>
    <ActionStep ExeFile="tap" Arguments="MyPlugin install" ActionName="install" />
    <ActionStep Exefile="tap" Arguments="MyPlugin prepare-uninstall" ActionName="prepareUninstall" />
    <ActionStep ExeFile="tap" Arguments="MyPlugin uninstall" ActionName="uninstall" />
    <ActionStep ExeFile="./Packages/MyPlugin/WaveformGenerator.exe" Arguments='--generate-waveforms --debug'
ActionName="test" />
    <ActionStep ExeFile="tap" Arguments='run -v ./Packages/MyPlugin/waveform-test.TapPlan' ActionName="test" />
  </PackageActionExtensions>
 ...
</Package>
```

The above example plugin definition makes use of the following features:

- A CLI action to be run when it is installed
- A CLI action to be run when OpenTAP is preparing to uninstall it
- A CLI action to be run when it is uninstalled.
- A binary executable which generates waveforms, used in preparation for testing
- A testplan which verifies the plugin steps are working correctly with `tap package test MyPlugin`

The **ActionStep** elements are executed in the order that they appear in the package file. When `MyPlugin` is installed, OpenTAP will run the CLI action:

```
tap MyPlugin install
```

When it is uninstalled, OpenTAP will run the CLI actions:

```
tap MyPlugin prepare-uninstall
tap MyPlugin uninstall
```

When `tap package test MyPlugin` is invoked, OpenTAP will first execute the binary executable:

```
./Packages/MyPlugin/WaveformGenerator.exe --generate-waveforms --debug
```

And then run the bundled test plan:

```
tap run -v ./Packages/MyPlugin/waveform-test.TapPlan
```

> Using **ExeFile="tap.exe"** will work on Windows, but not on Linux and MacOS. Omitting the `.exe` extension will work on all platforms.

MetaData elements

Any unknown xml elements will be treated as metadata. These elements will be mapped to the `PackageDef.MetaData` dictionary on the C# package object model .

A Complete Example

The below configuration file results in `MyPlugin.{version}.TapPackage` file,containing `OpenTap.Plugins.MyPlugin.dll`, `waveform1.wfm` and `waveform2.wfm`.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Package Name="MyPlugin" xmlns="http://opentap.io/schemas/package" InfoLink="http://myplugin.com"
         Version="$(GitVersion)" OS="Windows,Linux" Architecture="x64" Group="Example" Tags="Example DUT Instrument">
  <Description>
    This is an example of a "package.xml" file.
    <Status>Released</Status>
    <Organisation>Keysight Technologies</Organisation>
    <Contacts>
      <Contact Email="tap.support@keysight.com" Name="TAP Support"/>
    </Contacts>
    <Prerequisites>None</Prerequisites>
    <Hardware>Emulated PSU</Hardware>
    <Links>
      <Link Description="Description of the MyPlugin" Name="MyPlugin" Url="http://www.keysight.com/find/TAP"/>
    </Links>
  </Description>
  <Owner>OpenTAP</Owner>
  <Files>
    <File Path="Packages/MyPlugin/OpenTAP.Plugins.MyPlugin.dll">
      <SetAssemblyInfo Attributes="Version"/>
    </File>
     <File Path="Packages/MyPlugin/waveform1.wfm"/>
    <File Path="Packages/MyPlugin/waveform2.wfm"/>
    <File Path="Packages/MyPlugin/waveform-test.TapPlan"/>
    <File Path="./Packages/MyPlugin/WaveformGenerator.exe"/>
    <File Path="Packages/MyPlugin/Example Icon.ico">
      <PackageIcon/>
    </File>
  </Files>
      <PackageIcon/>
    </File>
  </Files>
  <PackageActionExtensions>
    <ActionStep ExeFile="tap" Arguments="MyPlugin install" ActionName="install" />
    <ActionStep ExeFile="tap" Arguments="MyPlugin uninstall" ActionName="uninstall" />
    <ActionStep ExeFile="./Packages/MyPlugin/WaveformGenerator.exe" Arguments='--generate-waveforms --debug'
ActionName="test" />
    <ActionStep ExeFile="tap" Arguments='run -v ./Packages/MyPlugin/waveform-test.TapPlan' ActionName="test" />
  </PackageActionExtensions>
</Package>
```

In this example the package version is set according to the Git tag and branch, since `GitVersion` is expanded based on Git (described later in this section). The resulting filename would be something like `MyPlugin.9.0.103+d58122db.TapPackage`. Additionally, the `OpenTAP.Plugins.MyPlugin.dll` file would have the same version as the package, according to the `SetAssemblyInfo` element.

This `package.xml` file is preserved inside the TapPackage as metadata. The Package Manager will add some additional information to the file. The metadata file for the above configuration could look like the following:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Package Version="9.0.103+d58122db" Name="MyPlugin" InfoLink="http://myplugin.com" Date="03/14/2019 21:20:31"
OS="Windows,Linux" Architecture="x64" Tags="Example DUT Instrument" xmlns="http://opentap.io/schemas/package">
  <Description>
    This is an example of an "package.xml" file.
    <Status>Released</Status>
    <Organisation>Keysight Technologies</Organisation>
    <Contacts>
      <Contact Email="tap.support@keysight.com" Name="TAP Support"/>
    </Contacts>
    <Prerequisites>None</Prerequisites>
    <Hardware>Emulated PSU</Hardware>
    <Links>
      <Link Description="Description of the MyPlugin" Name="MyPlugin" Url="http://www.keysight.com/find/TAP"/>
    </Links>
  </Description>
  <Dependencies>
    <PackageDependency Package="OpenTAP" Version="^9.0" />
  </Dependencies>
  <Files>
    <File Path="Packages/MyPlugin/OpenTAP.Plugins.MyPlugin.dll">
      <Plugins>
        <Plugin Type="OpenTAP.Plugins.MyPlugin.Step" BaseType="Test Step"/>
        <Plugin Type="OpenTAP.Plugins.MyPlugin.MyDut" BaseType="Dut"/>
      </Plugins>
    </File>
    <File Path="Packages/MyPlugin/waveform1.wfm"/>
    <File Path="Packages/MyPlugin/waveform2.wfm"/>
    <File Path="Packages/MyPlugin/waveform-test.TapPlan"/>
    <File Path="./Packages/MyPlugin/WaveformGenerator.exe"/>
    <File Path="Packages/MyPlugin/Example Icon.ico">
      <PackageIcon/>
    </File>
  </Files>
      <PackageIcon/>
    </File>
  </Files>
  <PackageActionExtensions>
    <ActionStep ExeFile="tap" Arguments="MyPlugin install" ActionName="install" />
    <ActionStep ExeFile="tap" Arguments="MyPlugin uninstall" ActionName="uninstall" />
    <ActionStep ExeFile="./Packages/MyPlugin/WaveformGenerator.exe" Arguments='--generate-waveforms --debug'
ActionName="test" />
    <ActionStep ExeFile="tap" Arguments='run -v ./Packages/MyPlugin/waveform-test.TapPlan' ActionName="test" />
  </PackageActionExtensions>
</Package>
```

The dependency and version information added by the Package Manager allows it to determine whether all prerequisites have been met when trying to install the package on the client.

If the package has dependencies on other packages it is possible to create a file with the `.TapPackages` extension. This is essentially a zip file that contains the created package and all the other packages it depends on. This allows the installation of all necessary packages at the same time, thus making package distribution easier.

Folder Conventions

Package authors are able to put payload files anywhere in the installation folder structure for increased flexibility. However, some conventions are defined to encourage an organized folder structure. In this context two subfolders of the OpenTAP installation folder are significant:

Packages Folder

The **Packages** folder contains one folder for every package installed. The name of each of these package folders correspond to the package name. The folder contains at least the package.xml file for that package. By convention other files of the package should also be located here or in subsequent subfolders.

Dependencies Folder

The **Dependencies** folder contains managed dependency assemblies (.NET DLL) that can be shared between several packages. Each assembly has its own subfolder named with the assembly name and version. This allows several versions of the same assembly to be present. `tap package create` will automatically detect any managed assemblies referenced by the assemblies specified in the package.xml, and add them to this folder following this scheme. Files in this folder will not be searched during plugins discovery.

Excluding Folders From Search

OpenTAP will search assemblies in the installation directory on startup for two purposes:

- Discovering OpenTAP plugins
- Resolving dll dependencies

Package authors can exclude sub folders from being searched by adding a marker file to the sub folder. This file must be named `.OpenTapIgnore`. The content of the file is not important (can be empty, or document why this folder should be ignored). The presence of this file will cause the folder and all its subfolders to be excluded from search for both of the above purposes.

Any folder named exactly “Dependencies” will be excluded from plugin discovery only. See above section on folder conventions.

Command Line Use

You can create an OpenTAP package from the command line or from MSBUILD (directly in Visual Studio). If you create an OpenTAP project in Visual Studio using the SDK, the resulting project is set up to generate a .TapPackage using the Keysight.OpenTAP.Sdk.MSBuild.dll (only when building in “Release” configuration).

tap.exe is the OpenTAP command line tool. It can be used for different package related operations using the “package” group of subcommands. The following subcommands are supported:

Command	Description
tap package create	Creates a package based on an XML description file.
tap package list	List installed packages.
tap package uninstall	Uninstall one or more packages.
tap package test	Runs tests on one or more packages.
tap package download	Downloads one or more packages.
tap package install	Install one or more packages.

The following example shows how to create TAP packages based on package.xml:

```
tap.exe package create -v package.xml
```

The behavior of the `tap package create` command when packaging, can be customized using arguments. To list these arguments, from a terminal call the following:

```
$ tap.exe package create --help
Options:
Usage: create [-h] [-v] [-c] [--project-directory <arg>] [-o <arg>] [-p <arg>] [--fake-install] <PackageXmlFile>
  -h, --help             Write help information.
  -v, --verbose          Also show vebose/debug level messages.
  -c, --color            Color messages according to their level.
  --project-directory    The directory containing the GIT repo.
                         Used to get values for version/branch macros.
  -o, --out              Path to the output file.
  -p, --prerelease       Set type of prerelease
  --fake-install         Fake installs the created package by only extracting files not already in your installation
```

Versioning

The OpenTAP executables and OpenTAP packages are versioned independently and should use semantic versioning (see definition here). Versions are of the form **X.Y.Z-A+B**, where:

- X is the major version number, incremented upon changes that **break** backwards-compatibility.
- Y is the minor version number, incremented upon backwards-compatible changes.
- Z is the patch version, incremented upon every set of code changes. The patch version can include pre-release labels.
- A is an optional pre-release label.
- B is optional metadata (e.g. Git short commit hash and/or branch name).

It is possible to set the version of the *.TapPackage using one of the following methods:*

- Git assisted versioning
- Manual versioning

Git Assisted Versioning

The **$(GitVersion) Macro** can be used in the Version attribute of the Package and File element in package.xml. It follows semantic versioning with the **X.Y.Z-A+B** format (as described earlier). Git assisted versioning uses the Git repository history to automatically determine/increment prerelease versions. Git commits marked with annotated tags will be interpreted as **release versions**, and will not have any prerelease information added to their version numbers. Note that Git assisted versioning only recognizes annotated tags, not lightweight tags. To determine the first three values of the version number, Git assisted versioning reads a `.gitversion` file in from the root of the repository (see example later in this section). To determine the prerelease label the Git branch name is considered like this:

- **beta**: The code is on a branch named “integration”, “develop”, “dev” or “master” (name configurable in `.gitversion` file). The version is marked with a “beta” pre-release identifier. A number **N** is also added denoting the commit count from the last change to the X, Y or Z parts of the version number (in the `.gitversion` file). The format of the resulting package version is X.Y.Z-**beta.N**+W, where W is set to Git short commit hash.

- **rc**: The code is on a branch named “release” (optionally followed by a “release series” number - e.g. “release8x”) (name configurable in `.gitversion` file). When there is no tag on the current commit, this is just considered a release candidate, and is marked with an “rc” pre-release identifier. A number **M** is also added denoting the commit count from when this branch was last branched out from the default branch (e.g. rc.3). The format of the resulting package version is X.Y.Z-**rc.M**+W, where W is set to Git short commit hash.

- **alpha**: Code is on an alpha/feature branch. All branches, which do not meet the above criteria, are considered as alpha/feature branches. On these branches, an “alpha” pre-release identifier is added along with both N and M as defined above. The format of the resulting package version is X.Y.Z-**alpha.N.M**+W.**BRANCH_NAME**. For example: 1.0.0-alpha+c5317128.456-FeatureBranch, where the branch name is appended to the metadata.

To add and push annotated tag to the latest commit (and create a release version), run the following command in your project folder:

```
git tag -a v1.0.0 -m "version 1.0.0"
git push --tags
```

Annotated tags can also be created in Visual Studio. This is done by including a tag message during tag creation. A lightweight tag, which Git assisted versioning will not consider, is created if the tag message is left out.

The example above marks the latest commit with the “v1.0.0” annotated tag, i.e. a release version. When the package is created, the version (major, minor and patch) of the package is set to the value from `.gitversion`.

Example `.gitversion` file including options, their descriptions and default values:

```
# This file specifies the (first part of the) version number and some options used by the
# "OpenTAP sdk gitversion" command and the $(gitversion) macro in package.xml

# This is the version number that will be used. Prerelease numbers are calculated by
# counting git commits since the last change in this value.
version = 1.0.1

# A version is determined to be a "beta" prerelease if it originates from the default branch
# The default branch is the first branch that matches the following regular expession.
# Uncomment to change the default.
#beta branch = integration

# When specified multiple times later specifications of "beta branch" will only be tried
# if earlier ones did not match any branches in the git repository
#beta branch = develop
#beta branch = dev
#beta branch = master

# A version is determined to be a "rc" prerelease if it originates from a branch that matches
# the following regular expression.
# Uncomment to change the default.
#release branch = release[0-9x]*

# A version is determined to be a release (no prerelease identifiers, just the version number
# specified in this file), if it originates from a commit that has an annotated tag that matches
# the following regular expression. (Note that the actual value of the tag is not used).
# Uncomment to change the default.
#release tag = v\d+\.\d+.\d+
```

To preview the version number that Git assisted versioning generates, you can use the command:

```
tap sdk gitversion
```

This command can also be useful if you need the same version number elsewhere in your build script.

Manual Versioning

The version can be set manually, e.g. `Version="1.0.2"`. The version **must** follow the semantic versioning format.

Advanced Packaging

As a plugin grows in complexity, special care is needed when targeting multiple platforms and architectures.

> For example, when shipping native binaries, different binaries must be shipped for different platforms.

Although these package definitions are often nearly identical for each platform or architecture, these subtle differences nonetheless require different *package definitions* for each target (or build-time modification).

Both of these solutions hurt the maintainability of the package definition. This process is made easy with **Variables** and **Conditions**.

The Variables Element

The **Variables** element is placed as a child of the **Package** element, and can be used to define file-scoped variables. A variable can be referenced inside an element or an attribute using a `$(VaribleName)` syntax.

> <SomeElement Attr1="$(abc)">abc $(def) ghi</SomeElement> will expand $(abc) and $(def) &lt;$(XmlElement)></$(XmlElement)> will not expand, and is invalid XML.
>
> A variable will be expanded exactly once. E.g. if `$(abc)` expands to the string `"$(def)"`, then $(def) will not be expanded.
>
> Tip: Although **Variables** appears a a child of the **Package** element, variables can still be used in **Package** attributes. The following `package.xml` example will correctly set the package architecture and OS.

```
<Package OS="$(Platform)" Architecture="$(Architecture)">
    <Variables>
        <Architecture>x64</Architecture>
        <Platform>Windows</Platform>
    </Variables>
</Package>
```

If an *Environment* variable is defined with the same name as a *file-local* variable, then the *file-local* variable will take precedence. Default values can be specified using *Conditions*.

> **Note** the $(GitVersion) variable has a special meaning, and cannot be overriden.

The Condition Attribute

The **Condition** attribute can be placed on any XML element except the root **Package** element. A condition can take two forms; an equality comparison, or a literal value:

> Condition examples:
> <SomeElement Condition="$(abc)" /&gt;<br> &lt;SomeElement Condition="$(abc) == 123" />
> <SomeElement Condition="$(abc) != $(def)" />

If the condition evaluates to false, the **Element** containing the condition is removed.

When a literal value is used, it is considered *true* if the value is a non-empty string, and *false* if the value is an empty string or contains only whitespace characters.

Condition Examples

This table demonstrates the general behavior of conditions. Assume `$(a) = 1` and `$(b) = 2`

Condition	**Value**	
“”	**false**	
" “	**false**	
1	**true**	
0	**true**	
true	**true**	
false	**true**	
false == “false”	**true**	
false == “false”	**false**	
(*a*)= = 1 * *|* * *true* * *|| * *(a) == (*b*)* * | * **false* * *|| * *(a) != $(b)	**true**	

Variables and Conditions Example

Consider this example (which is an excerpt from the OpenTAP package definition) for a real world use case. Some elements have been omitted for brevity

```xml
<Package Version="$(GitVersion)" OS="$(Platform)" Architecture="$(Architecture)" Name="OpenTAP" >
    <Variables>
        <!-- We include some native dependencies based on the platform and architecture, notably libgit2sharp -->
        <Architecture Condition="$(Architecture) == ''">x64</Architecture>
        <Platform Condition="$(Platform) == ''">Windows</Platform>
        <!--Set Sign=false to disable Signing elements. This is useful for local debug builds -->
        <Sign Condition="$(Sign) != false">true</Sign>
        <!-- Set Debug=true to exclude documentation files and include debugging symbols -->
        <Debug Condition="$(Debug) != true">false</Debug>
    </Variables>
    <!-- Common files  -->
    <Files>
        <File Path="tap.runtimeconfig.json"/>
        <File Path="tap.dll">
            <SetAssemblyInfo Attributes="Version"/>
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
        <File Path="OpenTap.dll">
            <SetAssemblyInfo Attributes="Version"/>
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
        <File Path="Packages/OpenTAP/OpenTap.Cli.dll">
            <SetAssemblyInfo Attributes="Version"/>
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
        <File Path="OpenTap.Package.dll">
            <SetAssemblyInfo Attributes="Version"/>
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
        <File Path="Packages/OpenTAP/OpenTap.Plugins.BasicSteps.dll" SourcePath="OpenTap.Plugins.BasicSteps.dll">
            <SetAssemblyInfo Attributes="Version"/>
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
    </Files>
    <!-- Windows only files -->
    <Files Condition="$(Platform) == Windows">
        <File Path="tap.exe">
            <Sign Certificate="Keysight Technologies, Inc" Condition="$(Sign)==true"/>
        </File>
        <File Path="Dependencies/LibGit2Sharp.0.25.0.0/git2-4aecb64.dll"
              SourcePath="runtimes/win-$(Architecture)/native/git2-4aecb64.dll"/>
        <File Path="OpenTapApiReference.chm"
              SourcePath="../../Help/OpenTapApiReference.chm"
              Condition="$(Debug) == false"/>
    </Files>
    <!-- Linux only files -->
    <Files Condition="$(Platform) != Windows">
        <File Path="tap"/>
        <File Path="Dependencies/LibGit2Sharp.0.25.0.0/libgit2-4aecb64.so.linux-x64"/>
    </Files>
    <!-- PDB files -->
    <Files Condition="$(Debug) == true">
      <File Path="tap.pdb"/>
      <File Path="OpenTap.pdb"/>
      <File Path="Packages/OpenTAP/OpenTap.Cli.pdb"/>
      <File Path="OpenTap.Package.pdb"/>
      <File Path="Packages/OpenTAP/OpenTap.Plugins.BasicSteps.pdb" SourcePath="OpenTap.Plugins.BasicSteps.pdb"/>
    </Files>
    <PackageActionExtensions Condition="$(Platform) != Windows">
        <ActionStep ActionName="install" ExeFile="chmod" Arguments="+x tap"  />
    </PackageActionExtensions>
</Package>
```

Here we use several properties for targeting different architectures and platforms with a single package definition. Assume for now that all the *Condition* attributes evaluate to 'true'.

1. We bundle different versions of the native binary `LibGit2Sharp` depending on platform and architecture.
2. We use the `Sign` property in order to easily disable signing when building in debug environments without signing capabilities.
3. We use the `Debug` property in order to include .pdb files with debugging symbols.
4. We run `chmod +x tap` after the package has been installed on non-windows platforms

After preprocessing this package definition, we get the following definition:

> This preprocessing is performed automatically when `tap package create` is used, and is only shown here for illustrative purposes

```xml
<Package Version="$(GitVersion)" OS="Windows" Architecture="x64" Name="OpenTAP">
  <!-- Common files  -->
  <Files>
    <File Path="OpenTap.dll">
      <Sign Certificate="Keysight Technologies, Inc" />
    </File>
    <File Path="Packages/OpenTAP/OpenTap.Cli.dll">
      <Sign Certificate="Keysight Technologies, Inc" />
    </File>
    <File Path="OpenTap.Package.dll">
      <Sign Certificate="Keysight Technologies, Inc" />
    </File>
    <File Path="Packages/OpenTAP/OpenTap.Plugins.BasicSteps.dll" SourcePath="OpenTap.Plugins.BasicSteps.dll">
      <Sign Certificate="Keysight Technologies, Inc" />
    </File>
    <File
Path="Dependencies/System.Runtime.InteropServices.RuntimeInformation.4.0.2.0/System.Runtime.InteropServices.RuntimeInformation.dll"
SourcePath="System.Runtime.InteropServices.RuntimeInformation.dll" />
    <File Path="tap.exe">
      <Sign Certificate="Keysight Technologies, Inc" />
    </File>
    <File Path="Dependencies/LibGit2Sharp.0.25.0.0/git2-4aecb64.dll" SourcePath="lib/win32/x64/git2-4aecb64.dll" />
  </Files>
  <!-- Windows only files -->
  <!-- Linux only files -->
  <!-- PDB files -->
</Package>
```

1. Notice that the **Variables** element, and all the **Condition** attributes have been removed.
2. Notice that all the **Files** elements have been merged into a single element (though the comments are still there).

As mentioned earlier, **Variables** variables take precedence over **Environment** variables, but leveraging the **Condition** attribute allows us to reverse this behavior. Take another look at the **Variables** from earlier:

```xml
<Variables>
    <!-- We include some native dependencies based on the platform and architecture, notably libgit2sharp -->
    <Architecture Condition="$(Architecture) == ''">x64</Architecture>
    <Platform Condition="$(Platform) == ''">Windows</Platform>
    <!--Set Sign=false to disable Signing elements. This is useful for local debug builds -->
    <Sign Condition="$(Sign) != false">true</Sign>
    <!-- Set Debug=true to exclude documentation files and include debugging symbols -->
    <Debug Condition="$(Debug) != true">false</Debug>
</Variables>
```

When using the **Condition** attribute in this way, the value specified in a **Variables** element is used *only* if the value is not defined in the environment. Leveraging this, we can now create an OpenTAP package for Windows-x86, Windows-x64, and Linux-x64 in a few easy steps:

```
$env:Platform="Windows"
$env:Architecture="x86"
tap package create package.xml -o OpenTAP.Windows.x86.TapPackage
$env:Architecture = "x64"
tap package create package.xml -o OpenTAP.Windows.x64.TapPackage
$env:Platform = "Linux"
tap package create package.xml -o OpenTAP.Linux.x64.TapPackage
```

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.
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.

Attribute name	Description
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:

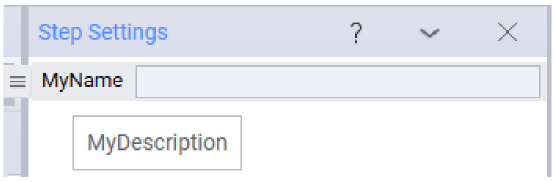
- Can be applied to class names (impacting the appearance in dialogs, such as the Add New Step dialog), or to properties (impacting appearance in the Step Settings Panel).
- Has the following signature in its constructor:

```
(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.

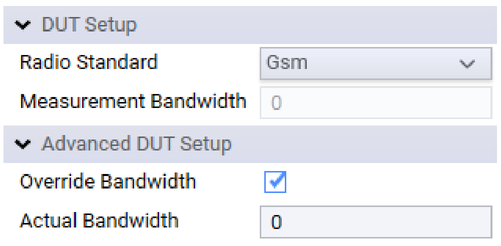
The **HideIfDisabled** optional parameter of EnabledIf makes it possible to hide settings when they are disabled. This is useful to hide irrelevant information from the user.

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:


DUT Setup

Radio Standard	Gsm
Measurement Bandwidth	0

Advanced DUT Setup

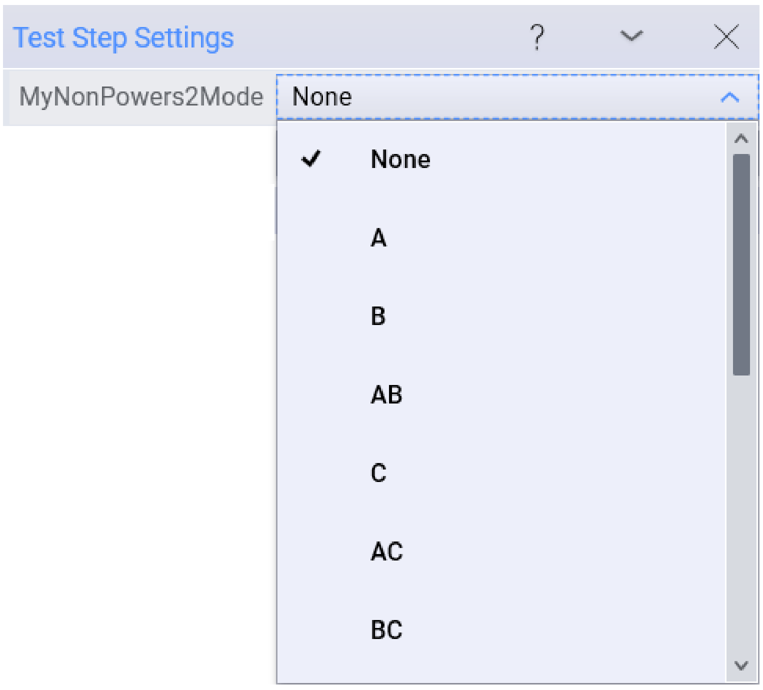
Override Bandwidth	✓
Actual Bandwidth	0

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.



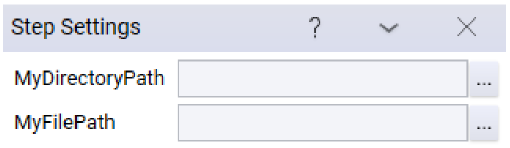
Enumeration constant with value zero is handled specially by unselecting all enumeration constants except that with zero. Enumeration constant with value not of powers of two will be selected accordingly to its bitmask representative. (eg. enumeration constant with value 3 will be selected togther with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

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.

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 units for a setting. The Editor displays the units after the value (with a space separator). Compound units (watt-hours) should be hyphenated. Optionally, displayed units can insert engineering prefixes.

See the `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\UnitAttributeExample.cs` file for an extensive example.

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.

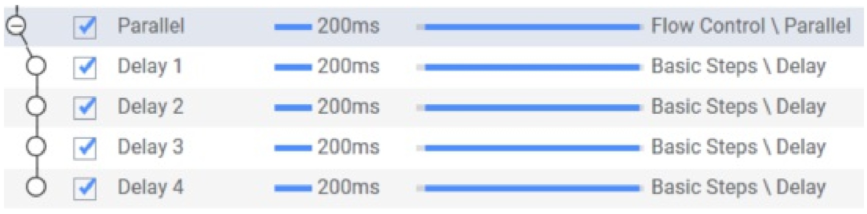
In OpenTAP and C#, parallelism can be implemented in several ways:

- **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:



Parallel	200ms	Flow Control \ Parallel
Delay 1	200ms	Basic Steps \ Delay
Delay 2	200ms	Basic Steps \ Delay
Delay 3	200ms	Basic Steps \ Delay
Delay 4	200ms	Basic Steps \ Delay

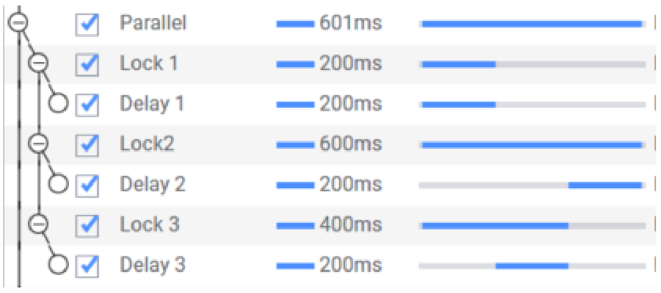
Four Delays

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.



Parallel	601ms
Lock 1	200ms
Delay 1	200ms
Lock2	600ms
Delay 2	200ms
Lock 3	400ms
Delay 3	200ms

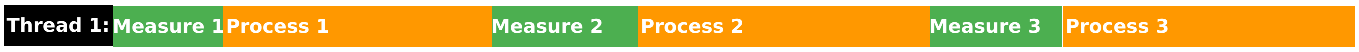
Locked Delays

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 tasks.

Sequential Processing

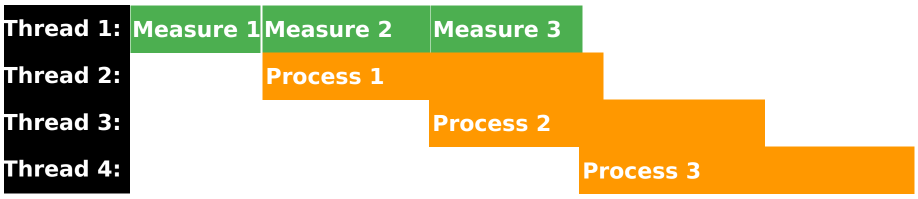
In traditional sequential processing, the order of operations is as follows:



Sequential Processing

Deferred Processing

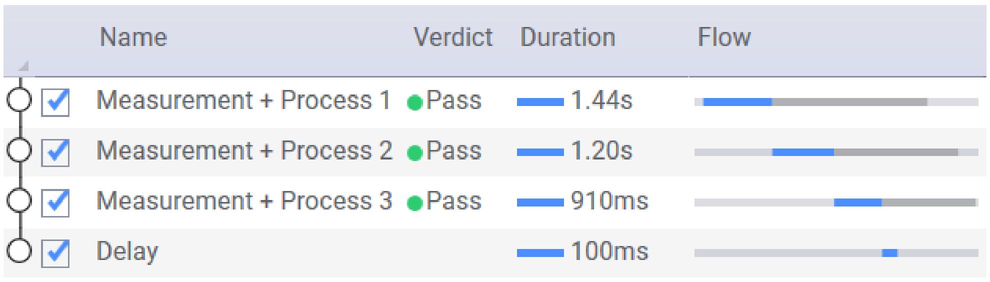
When deferred processing is used, operations are handled in parallel, as shown in the diagram below:



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.



Name	Verdict	Duration	Flow
Measurement + Process 1	Pass	1.44s	
Measurement + Process 2	Pass	1.20s	
Measurement + Process 3	Pass	910ms	
Delay		100ms	

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.