OpenTAP

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.

Keysight Test Automation

Together with OpenTAP it is recommended to use a Graphical User Interface. Keysight Technologies offers both an enterprise and community version of Keysight Test Automation Developer's System that provides a highly flexible graphical user interface and some code examples. The offering is available for download using the OpenTAP package repository (packages.opentap.io).

Development requires the following software:

- Visual Studio 2017 or 2019
- OpenTAP
- Keysight Test Automation Developer's System (Community or Enterprise Edition)

Suggested Resources

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

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.

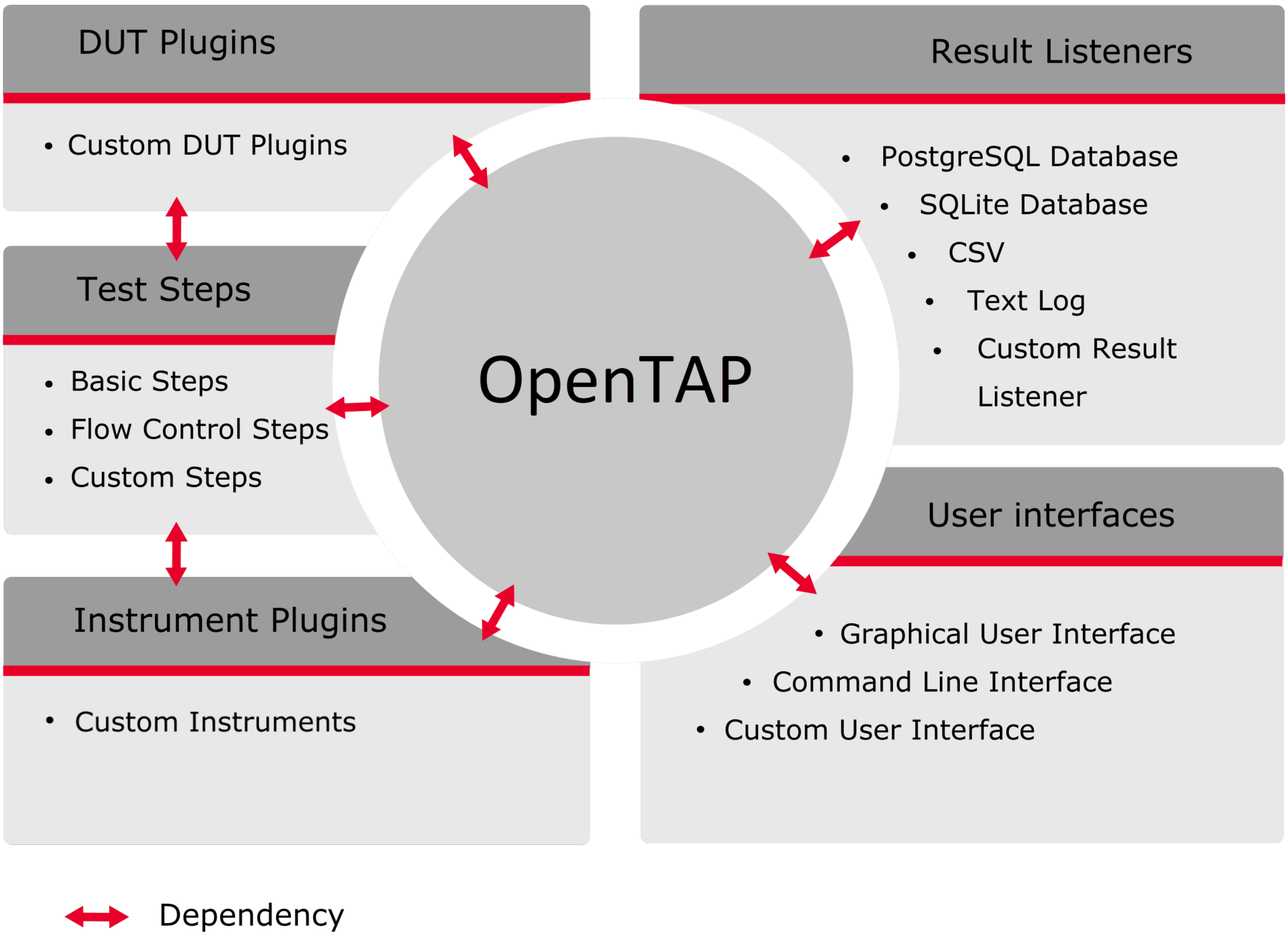
OpenTAP offers a range of functionality and infrastructure for configuring, controlling, and executing test algorithms. OpenTAP provides an API for implementing plugins in the form of test steps, instruments, DUTs and more.

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

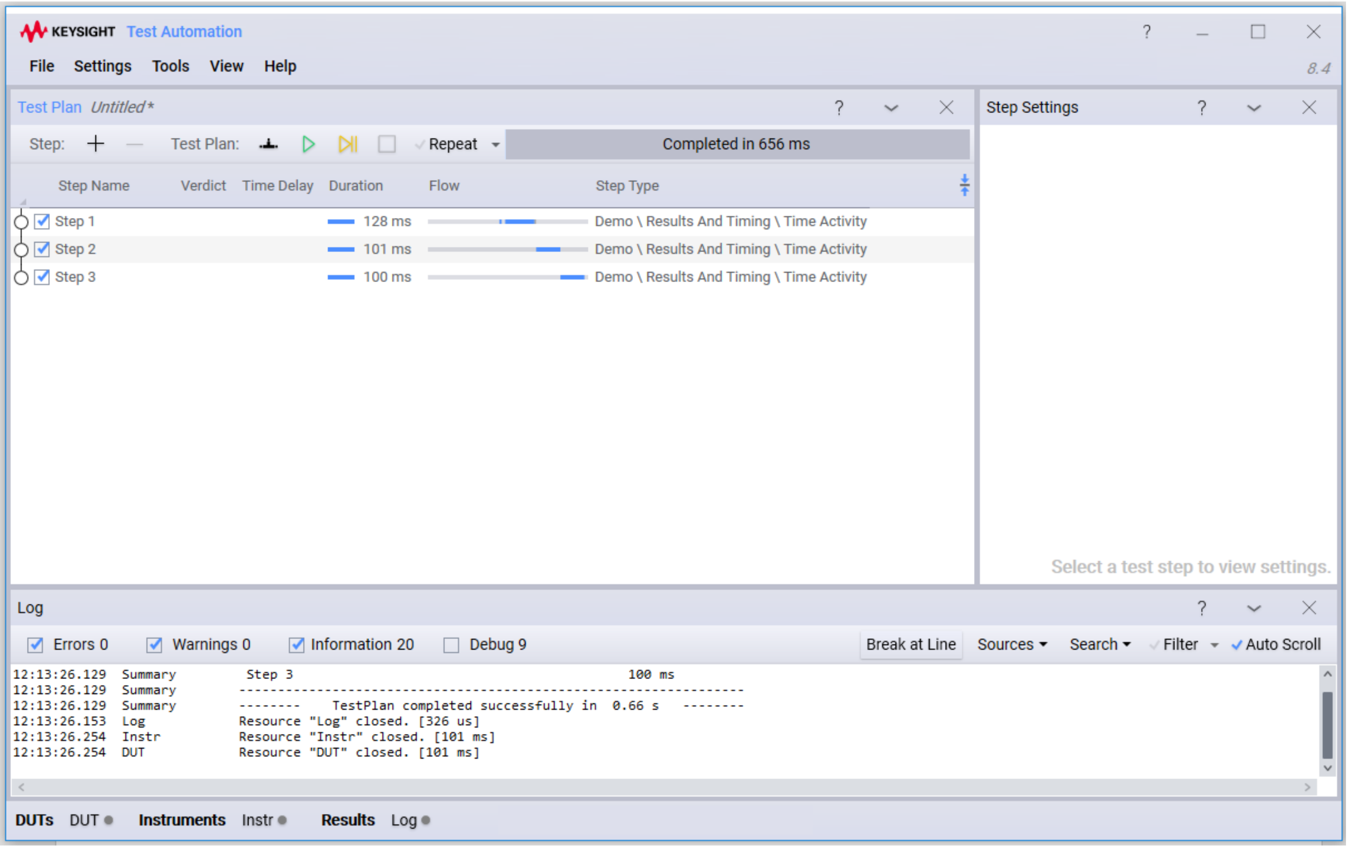
A *test plan* is a sequence of test steps with some additional data attached. Test plans are created via the Editor GUI. Creating test plans is described in the *Editor Help* (EditorHelp.chm), accessible within the Editor GUI. Test plan files have the *.TapPlan* suffix, and are stored as xml files.

Test Plan Control Flow

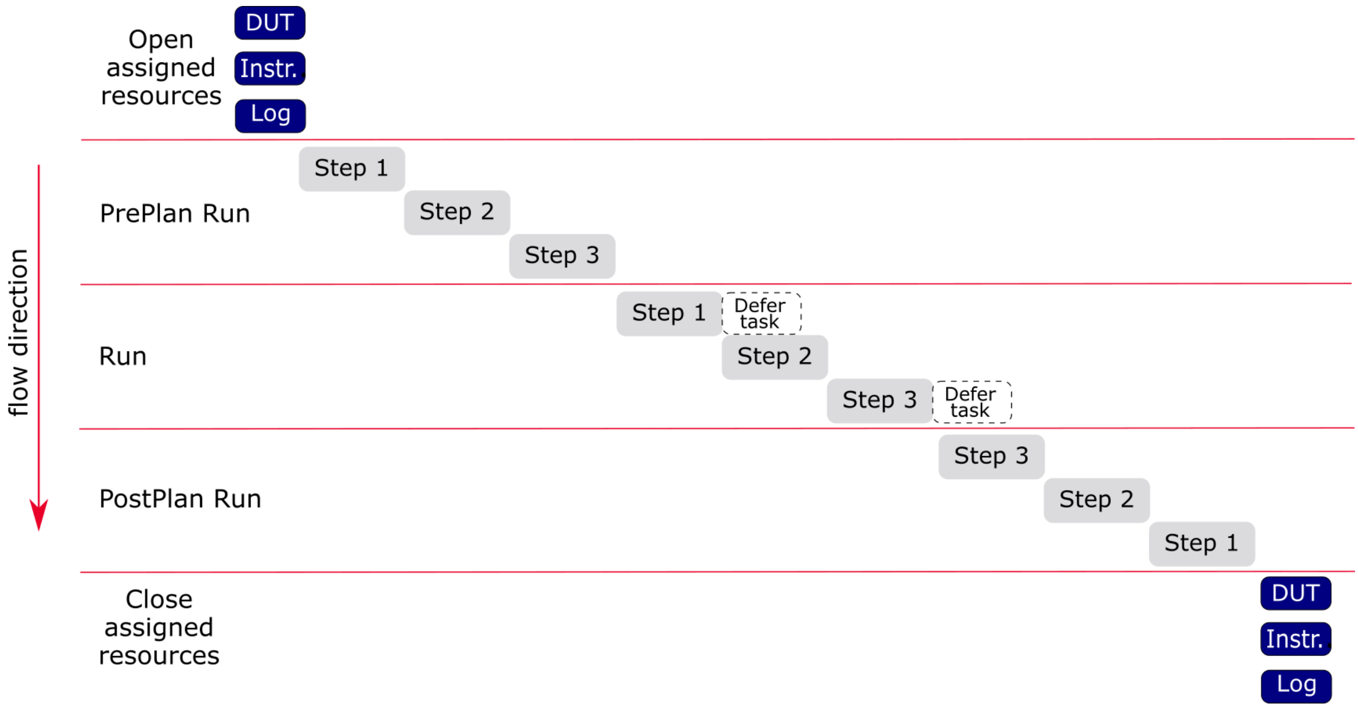
To use OpenTAP to its full potential, developers must understand the control flow of a running test plan. Several aspects of OpenTAP can influence the control flow. Important aspects include:

- 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:



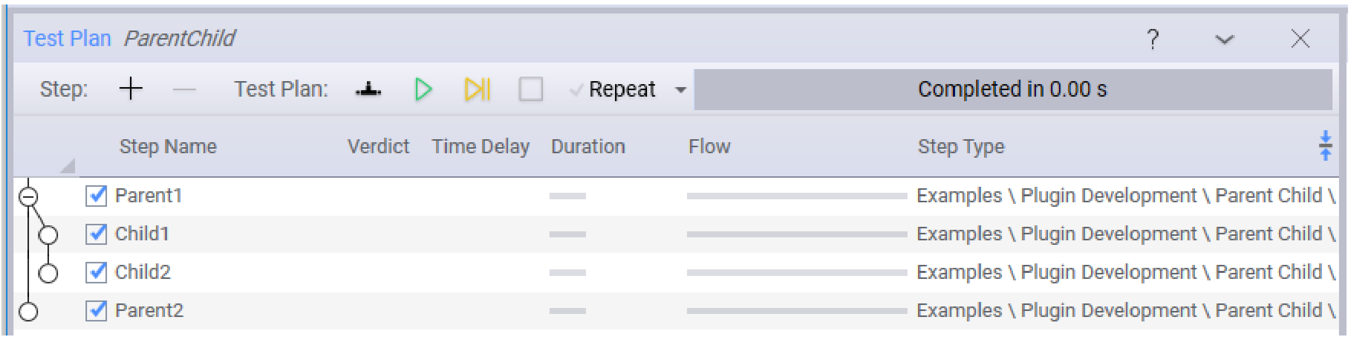
The test plan has three test steps, in succession. In this test plan, none of the steps have child steps. A more complex example, with child steps, is presented later in this section. The test plan relies on the resources DUT, Instr. and Log to be available and configured appropriately. The following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



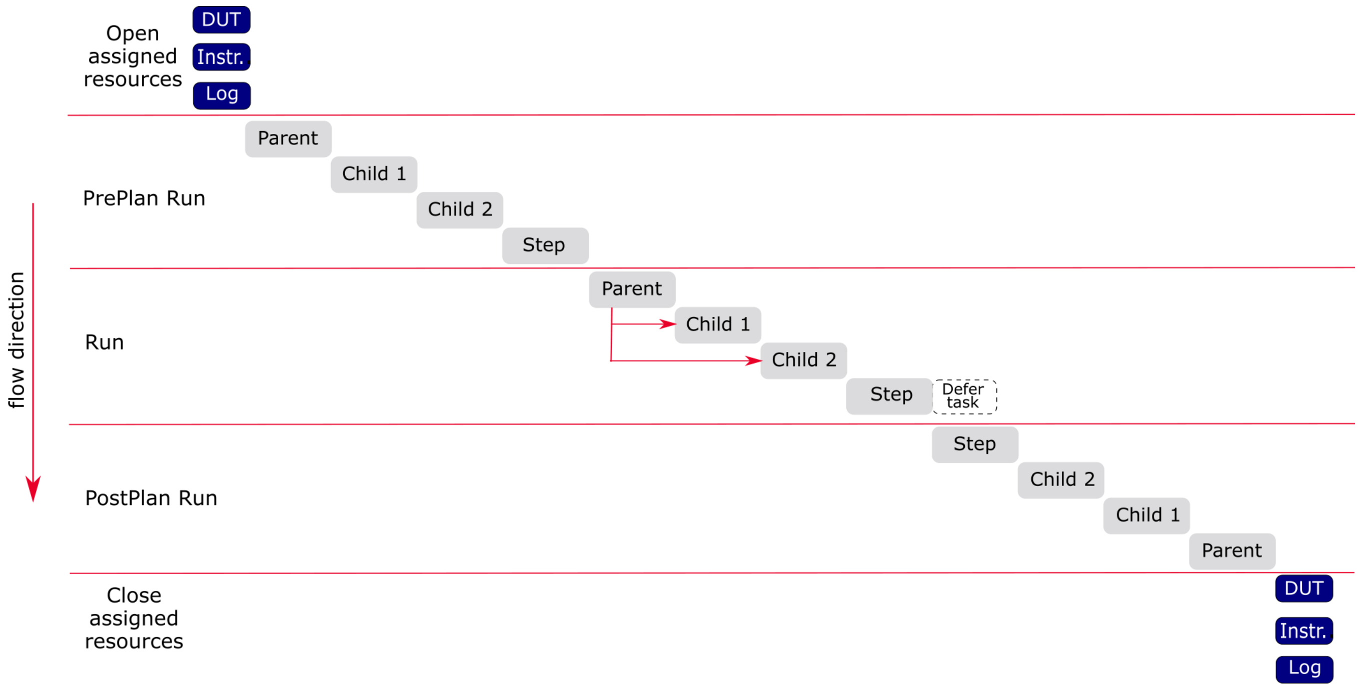
In the **Open assigned resources** phase all DUTs, instruments and configured result listeners are opened in parallel. As soon as all resources are open the **PrePlanRun** methods of the test steps execute in succession. This is followed by the execution of the **Run** methods where all test steps are 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:



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.

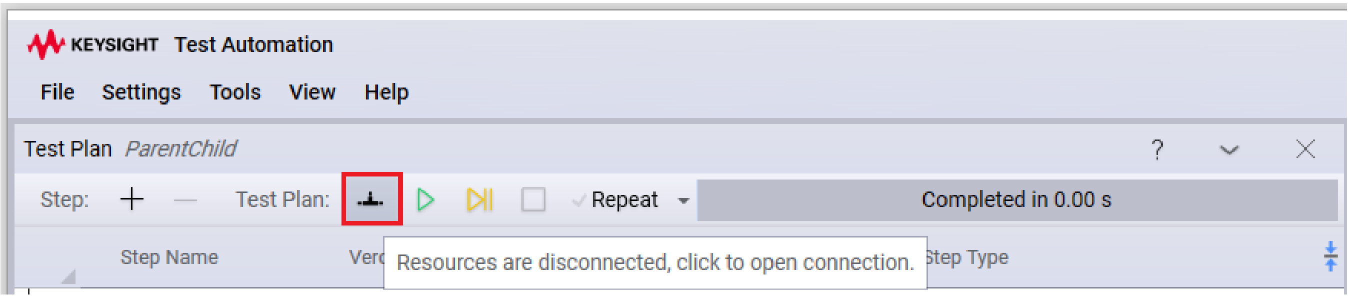
Note that the above examples are very simple. A more advanced test plan may incorporate flow control statements to change execution flow. For example, adding a *Parallel* test step as a parent to child test steps will make the test steps run in parallel. This only affects the run stage, the other stages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



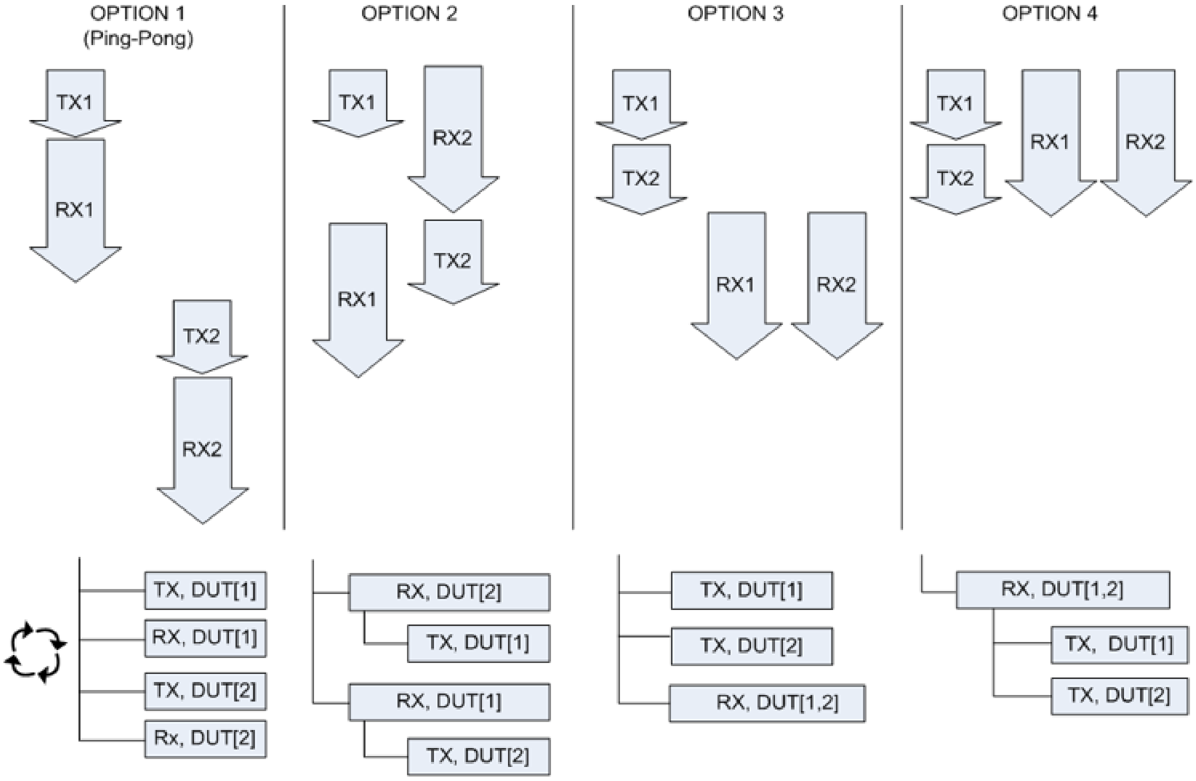
Resources opened manually remain open between test plan runs. This eliminates the time required to open and close them for each test plan run.

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 with your own project

This section describes the helpful features of the OpenTAP NuGet package for developing plugins. Any C# project (*.csproj) can be turned into a plugin project by adding a reference to the OpenTAP NuGet package.

Create project

First you will need a C# project to hold your plugin classes. This can be an existing project you have already started, or you can start a new one. To start a new one we recommend choosing “Class Library (.NET Standard)” in the Visual Studio “Create a new project” wizard.

Reference OpenTAP

The OpenTAP NuGet package is the recommended way to get OpenTAP for plugin developers/projects. The NuGet package is available on nuget.org , so Visual Studio will list it in the build in the NuGet package manager. You can get to that by right clicking your project in the Solution Explorer and selecting “Manage NuGet Packages”. In the NuGet package manager, search for OpenTAP, and click install.

OpenTAP NuGet features

This section describes the additional msbuild features that come with the OpenTAP NuGet package.

OpenTAP 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 be the version of OpenTAP specified in your project file (e.g. using the VS NuGet package manager as described above). This feature makes it easier to manage several plugins that might not target the same version 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.

OpenTAP Package creation

The NuGet package also adds build features to help packaging your plugins as a *.TapPackage for distribution through OpenTAP’s package management system (e.g. by publishing it on packages.opentap.io). To take advantage of these features, your project needs a package definition file (normally named package.xml). You can use the command line `tap sdk new packagexml` to generate a skeleton file. As soon as a file named package.xml is in your project, 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 you need directly. TapPackages referenced like this will be installed into your projects output directory (e.g. bin/Debug/) along with OpenTAP itself.

You can specify an OpenTAP package that your project should reference. When doing this any .NET assemblies in that packages are added as references in your project. You do this by adding the following to your csproj file:

```
<ItemGroup>
  <OpenTapPackageReference Include="DMM API" Version="2.1.2" Repository="packages.opentap.io"/>
</ItemGroup>
```

This should be very similar to the way you add a NuGet package using `<PackageReference>`. `Version` and `Repository` are optional attributes, and default to latest release, and packages.opentap.io if omitted.

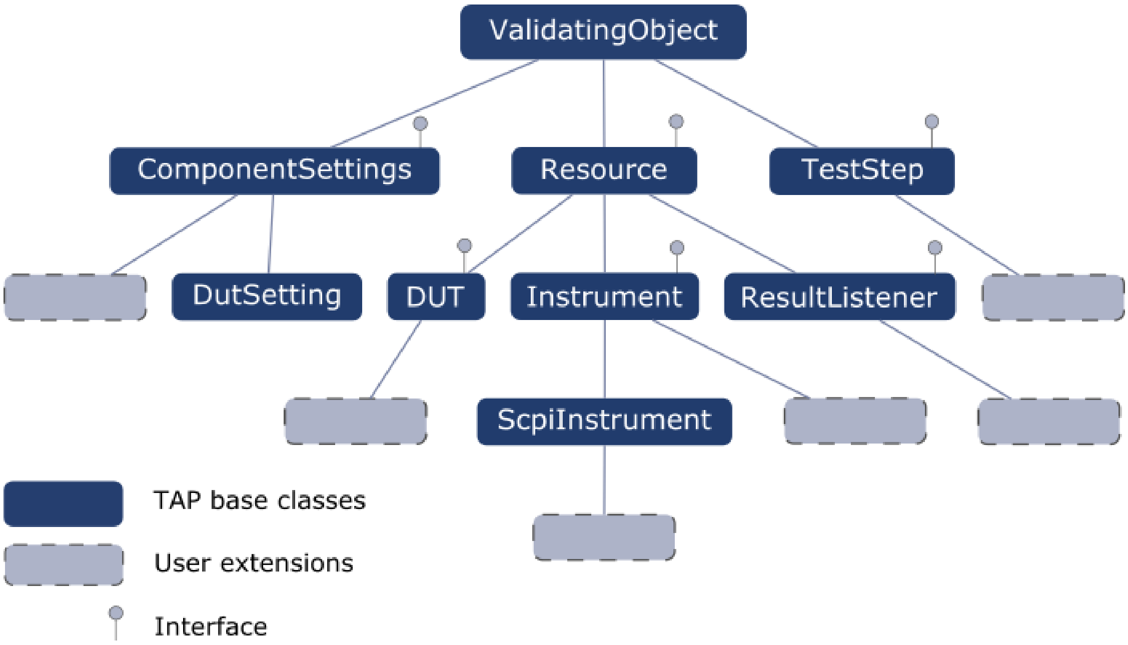
You can also specify a package that you just want installed (in e.g. bin/Debug/) but don’t want your project to reference. This can be useful for defining a larger context in which to debug. It is done as follows:

```xml
<ItemGroup>
  <AdditionalOpenTapPackage Include="DMM Instruments" Version="2.1.2" Repository="packages.opentap.io"/>
</ItemGroup>
```

Plugin Development

OpenTAP Plugin Object Hierarchy

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



Extending **ValidatingObject** 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 ValidatingObject.

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.

Using Attributes in Plugins

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.
IgnoreSerializer	Used on classes to ignore serialization. Useful for cases where a plugin implementation contains non-serializable members or types.
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 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.
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).

Attribute name	Description
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.

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 Visual Studio 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.
            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();
        }
    }
}
```

To allow user configuration of a test step, developers must add appropriate properties to the plugin code. These properties will very likely be 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.

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

TestStep 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.

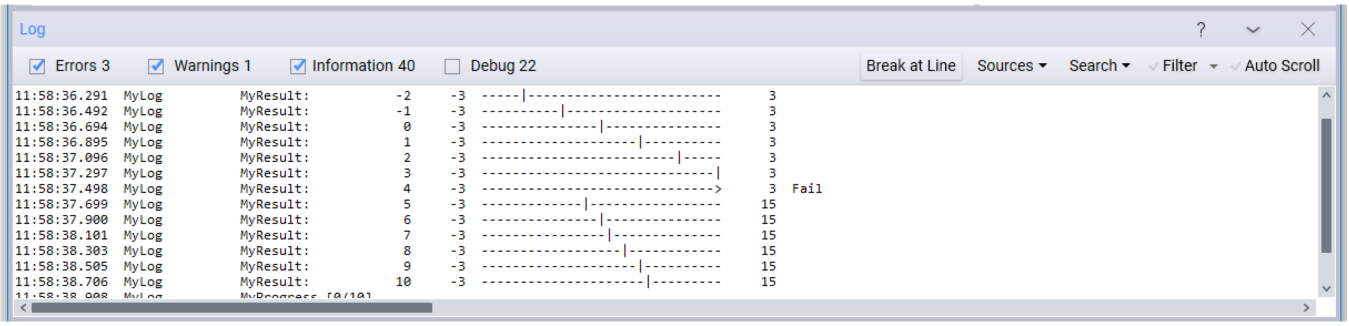
A step can abort the test plan run by calling the `PlanRun.MainThread.Abort()` method. If you have multiple steps in a plan, the engine will check if abort is requested between the `Run()` methods of successive steps. If you have a step which takes a long time to execute, you can call the `OpenTap.TapThread.ThrowIfAborted()`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



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.

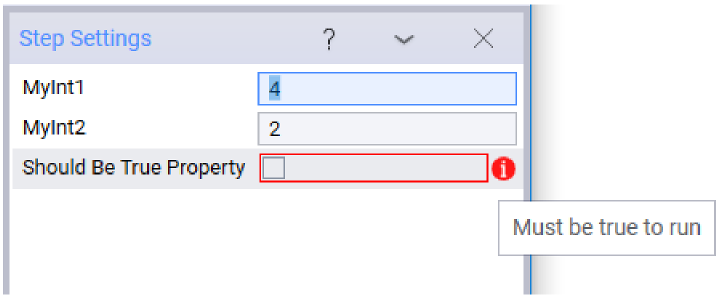
Validation

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:



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`.

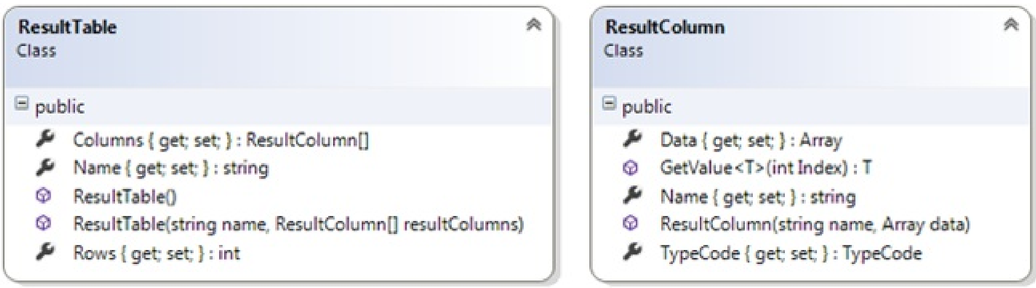
```
InputData = new double[] {0, 0, 5, 5, 5, 50};
ReadOnlyOutputData = new double[]{10, 10, 15, 15, 15, 150};
Results.PublishTable("Inputs vs. Moving Average", new List<string>() {"Input Values", "Output Values"},
     InputData, ReadOnlyOutputData);
```

Basic Theory

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.

ResultTable Details

This graphic shows the ResultTable definition.



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`

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 a test plan run. This is useful in situations where one step depends on a result from another step, but could also be used for flow control.

For examples, 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);
    }
}
```

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

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 Plugin Development

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:

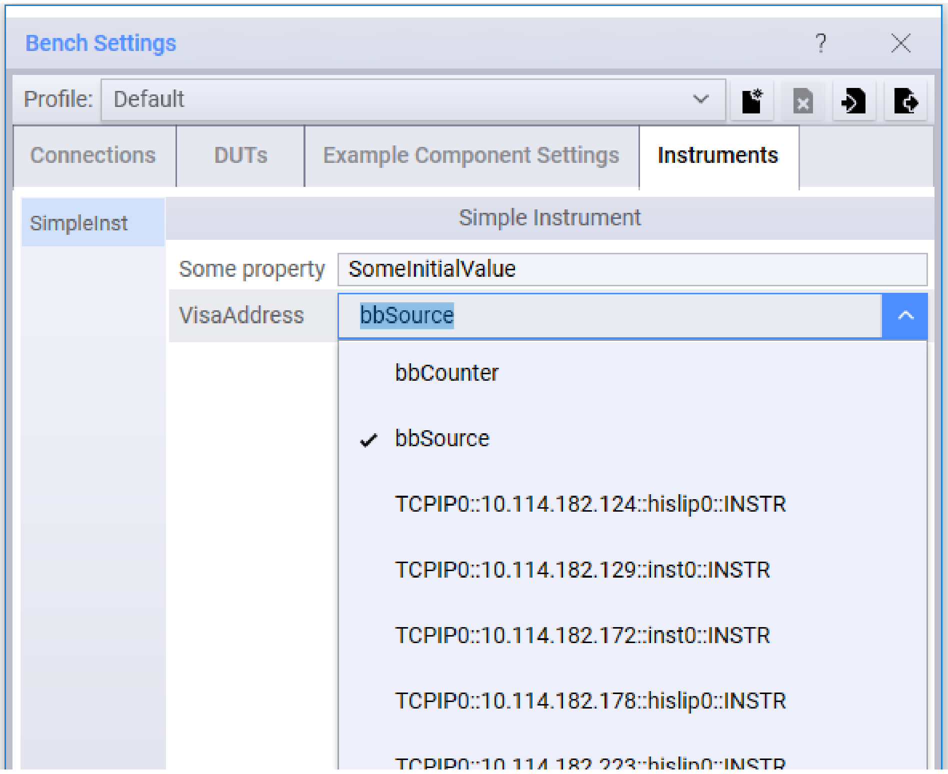
- **ScpiCommand**, which sends a command
- **ScpiQuery**, which sends the query and returns the results
- **VisaAddress**, which specifies the Visa address of the instrument

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:


Bench Settings
Profile: Default
Connections
DUTs
Example Component Settings
Instruments
SimpleInst
Simple Instrument
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

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 (to avoid conflicts with commas in VISA addresses), 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`

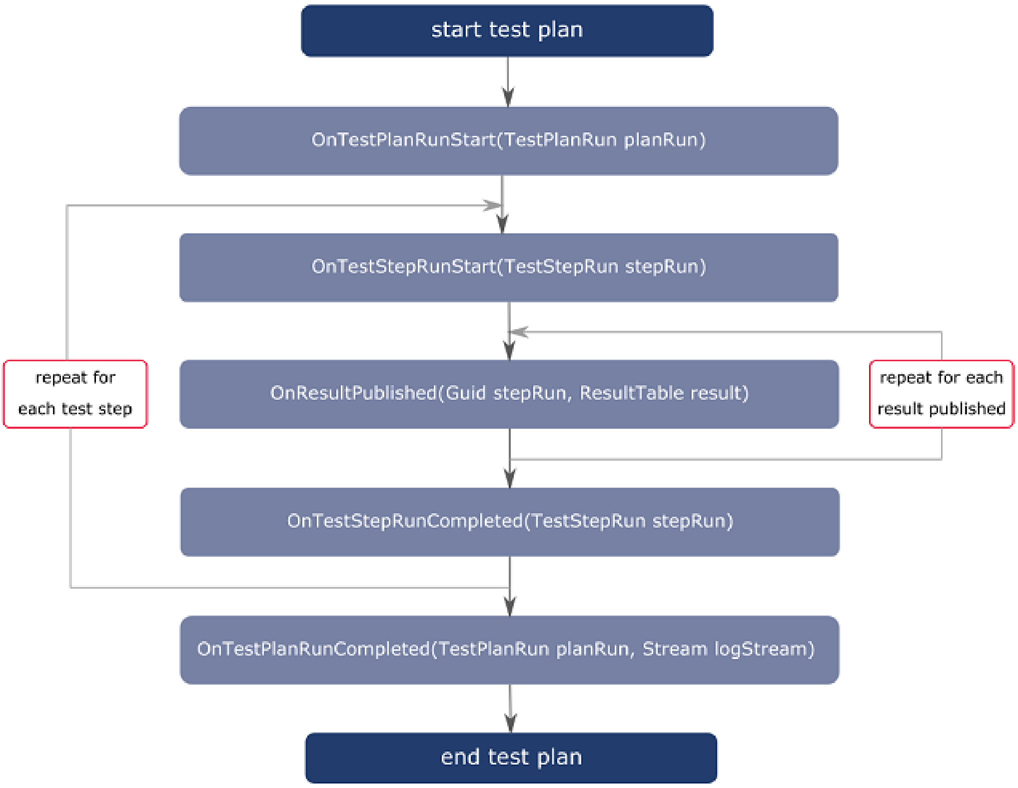
When extending the ResultListener class, the following methods can be overwritten:

- `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:



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

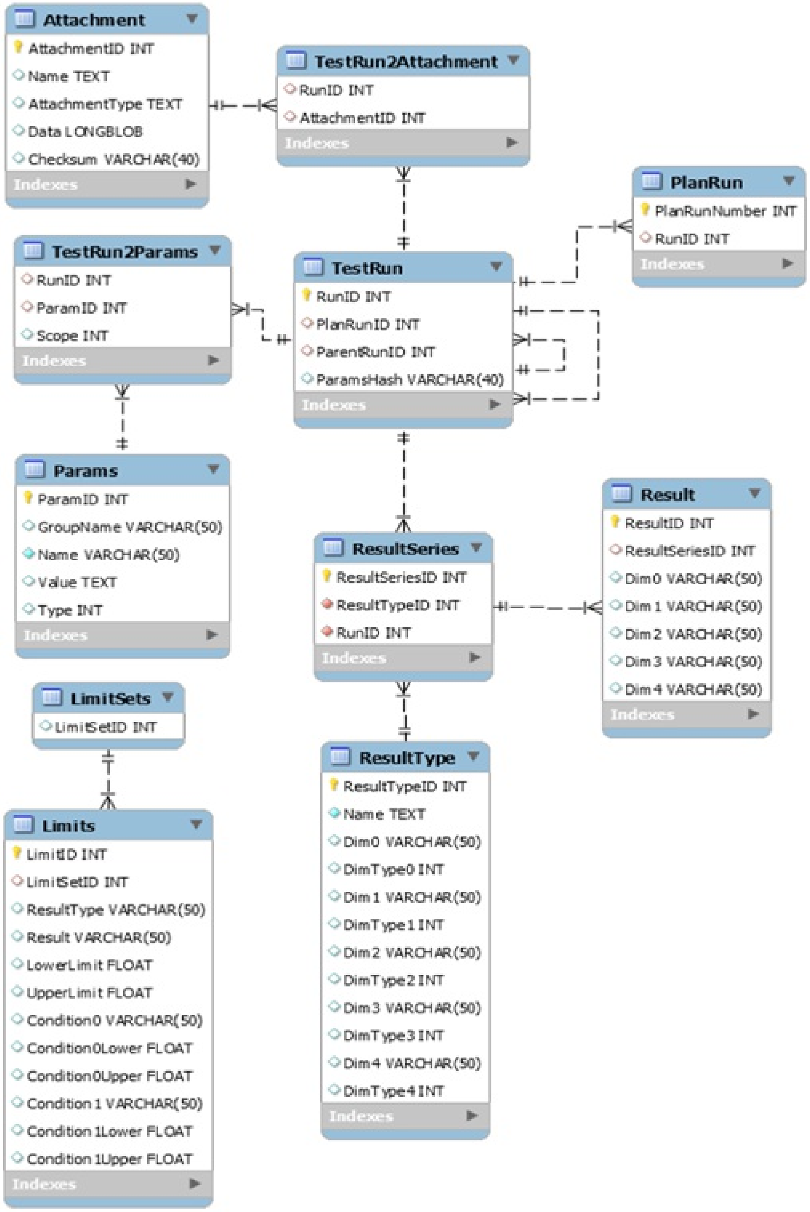
Interacting with the predefined databases or interfacing to an unsupported database requires DB knowledge.

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.



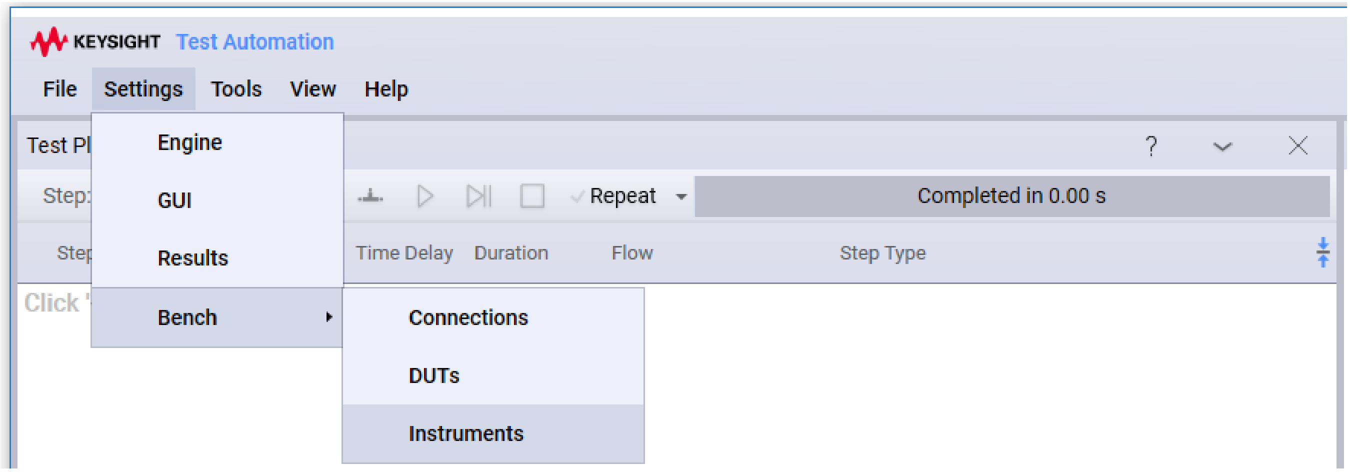
Attachment
AttachmentID INT
Name TEXT
AttachmentType TEXT
Data LONGBLOB
Checksum VARCHAR(40)
Indexes
TestRun2Attachment
RunID INT
AttachmentID INT
Indexes
PlanRun
PlanRunNumber INT
RunID INT
Indexes
TestRun2Params
RunID INT
ParamID INT
Scope INT
Indexes
TestRun
RunID INT
PlanRunID INT
ParentRunID INT
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.

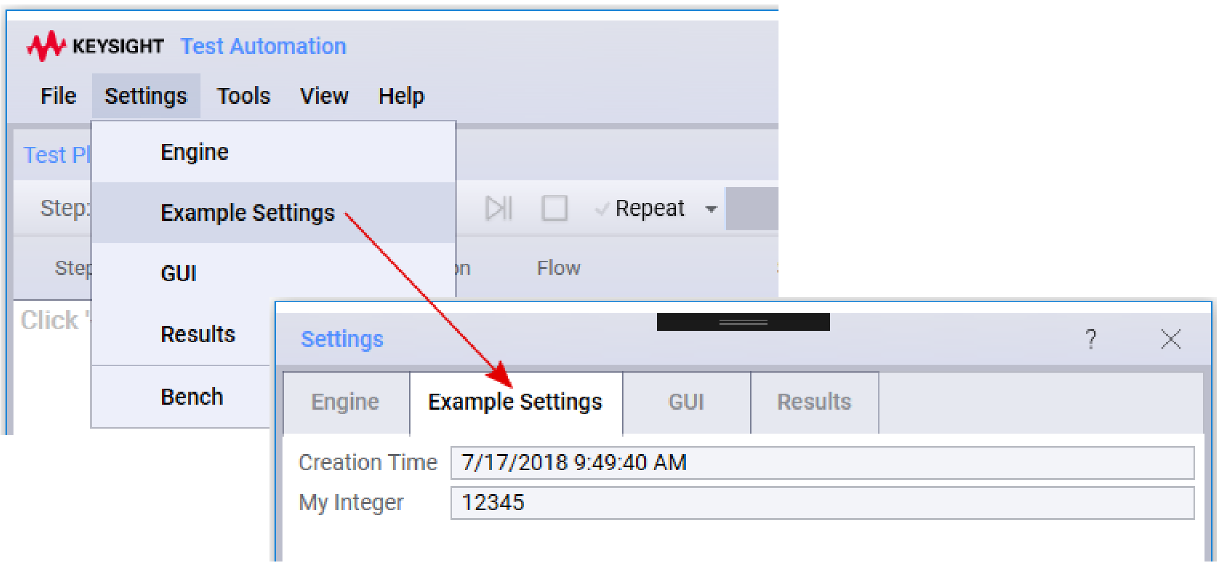


Creating a New Global/Grouped Settings Dialog

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.

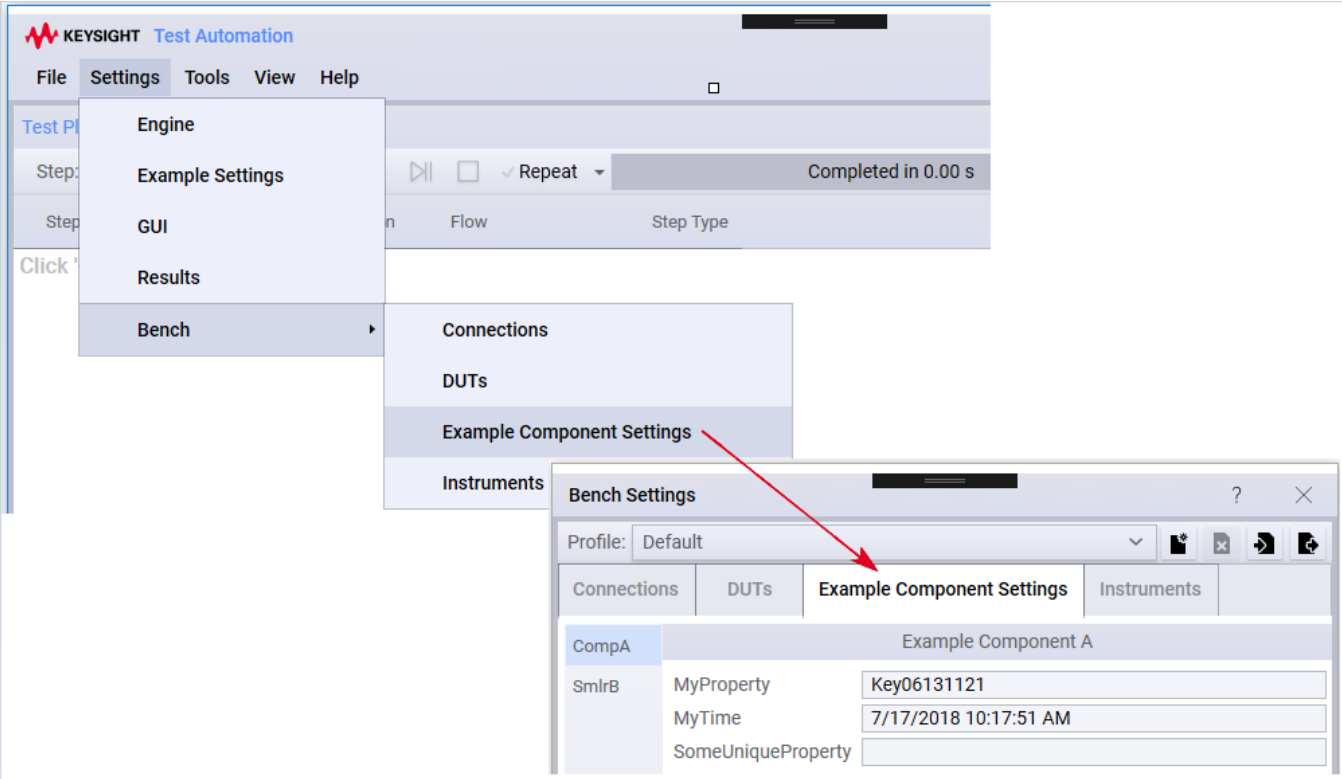
Single Instance of Multiple Different Settings

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:



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:



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.

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 PackageManager. 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 a 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 number 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"
         Group="Example">
  <Description>Example plugin containing Instrument, DUT and TestStep.</Description>
  <Owner>OpenTAP</Owner>
    <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 Path="Packages/Example Plugin/Example Icon.ico">
        <PackageIcon/>
      </File>
  </Files>
</Package>
```

Note: 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 PackageManager 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).

The configuration file supports optional attributes:

Attribute	Description
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 operative 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 PackageManager is running on. If the attribute is not specified, the default "Windows" is used. Example: `OS="Windows,Linux"`. Note, only the following OS values are currently supported by the package manager for automatic detection: Windows, Linux and OSX. So 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 PackageManager. 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 ''. May be empty. UIs may use this information to show a list of packages as a tree structure. See the example below.
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.

> **Note:** 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.

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.

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 with ','. An example of this can be seen in the example below.

SourceUrl Element

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

File Element

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 the OpenTAP installation directory. This serves as both source (where the packaging tool should get the file when creating the package) and target (where the file sould 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	License key 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. In the above example it 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.

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. See the example above.

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 from the previous section as such:

```
...
 <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>
 ...
```

Folder Conventions

In a the package definition XML file, package authors are able to put payload files anywhere in the installation folder structure for increased flexibility. However, some conventions are defined to encurage 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 folders contains at least the package.xml file for that package. By convention other files of the package should also be located here or in 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 dir 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 subfolder 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.

Example

The below configuration file results in `MyPlugin.{version}.TapPackage` file,containing `OpenTap.Plugins.MyPlugin.dll`, `waveform1.wfm` and `waveform2.wfm`. `OpenTap.Plugins.MyPlugin.dll` is obfuscated but none of the waveform files are.

```
<?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">
  <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>
```

```xml
  <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/Example Icon.ico">
      <PackageIcon/>
    </File>
  </Files>
</Package>
```

In this example the package version is set according to 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:

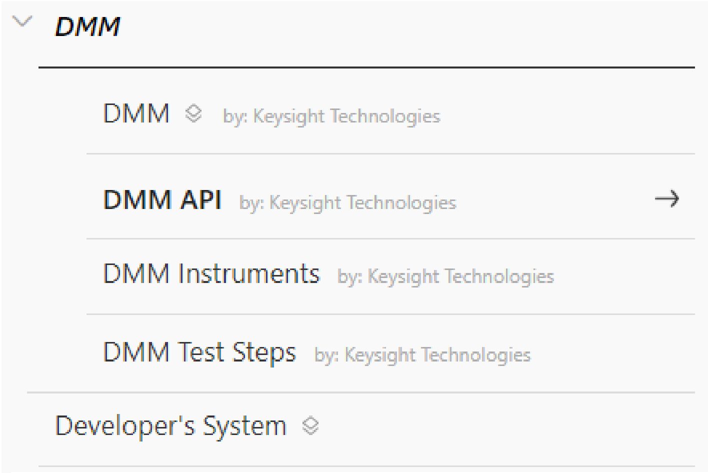
```xml
<?xml version="1.0" encoding="utf-8" ?>
<Package Version="9.0.103+d58122db" Name="MyPlugin" InfoLink="http://myplugin.com" Date="03/14/2019 21:20:31"
OS="Windows,Linux" Architecture="x64" xmlns="http://opentap.io/schemas/package">
  <Description>
    This is an example of an "package.xml" file.
    <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/Example Icon.ico">
      <PackageIcon/>
    </File>
  </Files>
</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.

Example of UIs using Owner, Group and Class

Following the image below: - The first `DMM` entry represent the group specified by the four following packages. - The small square icons signifies the package is a bundle (class attribute). - The gray text `by: Keysight Technologies` signifies the owner of the package.



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
```

Obfuscation

OpenTAP supports obfuscation provided by Keysight OpenTAP plugins. Two different obfuscators are supported: Preemptive Software Dotfuscator and Obfuscar v2.2.9.

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 sprcifications 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.

Appendix A: Attribute Details

Display

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

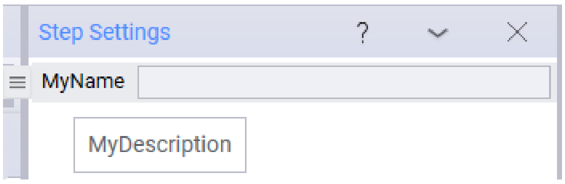
- Can be applied to class names (impacting the appearance in dialogs, such as the Add New Step dialog), or to properties (impacting appearance in the Step Settings Panel).
- Has the following signature in its constructor:

```
(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.

Embedded Attribute

The EmbeddedAttribute 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.

EnabledIf Attribute

The **EnabledIf** attribute disables or enables settings properties based on other settings (or other properties) of the same object. The decorated settings reference another property of an object by name, and its value is compared to the value specified in an argument. Properties that are not settings can also be specified, which allows the implementation of more complex behaviors.

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

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

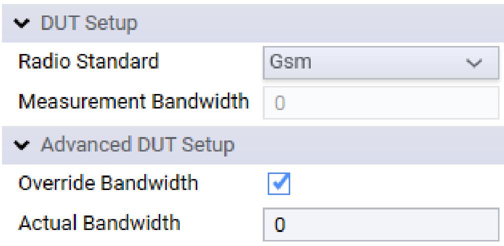
Multiple EnabledIf statements can be used at the same time. In this case all of them must be enabled (following the logical *AND* behavior) to make the setting enabled. If another behavior is wanted, an extra property (hidden to the user) can be created and referenced to implement another logic. In interaction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

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

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

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

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

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

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

    #endregion Settings
}
```

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

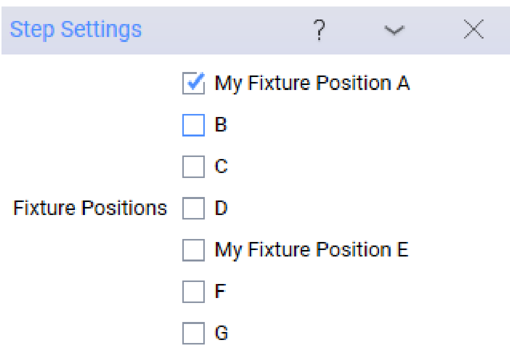


For an example, see `TAP_PATH\Packages\SDK\Examples\PluginDevelopment\TestSteps\Attributes\EnabledIfAttributeExample.cs`.

Flags Attribute

The **Flags** attribute is a C# attribute used with enumerations. This attribute indicates that an enumeration can be treated as a *bit field* (meaning, elements can be combined by bitwise OR operation). The enumeration constants must 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:

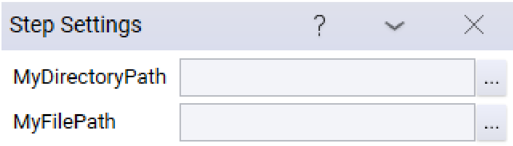


FilePath and DirectoryPath Attributes

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.

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.

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.

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?"); }
}
```

Appendix B: 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.