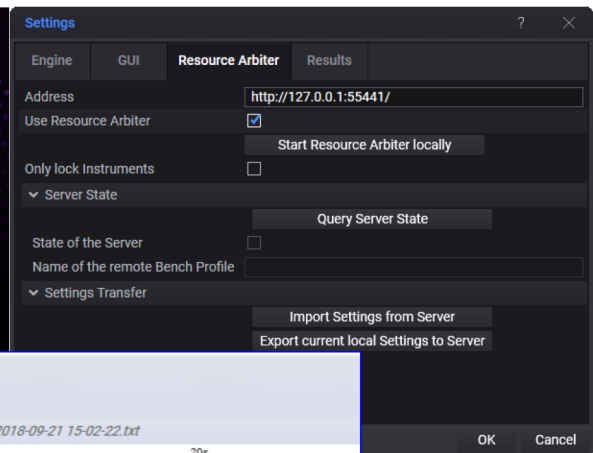


Keysight Resource Arbiter

User Guide

The image shows a screenshot of the Keysight Timing Analyzer. The title bar says "KEYSIGHT Timing Analyzer". The menu bar has "File" and "View". The main window shows a "Time View" for a file named "Keysight.Tap.Plugins.ResourceManager 2018-09-21 15-02-22.txt". The table below shows the timing data for various sources.

Source	Time Active, %	Inst 1 [4.09 s]	Inst 2 [9.01 s]	Inst 1 [4.01 s]
DUT A	15.0 s (28.3%)			
DUT B	15.1 s (28.5%)	Inst 1 [4.09 s]	Inst 2 [9.01 s]	
DUT C	15.0 s (28.3%)			Inst 1 [4.01 s]
DUT D	15.0 s (28.3%)		Inst 1 [4.01 s]	
Inst 1	24.2 s (45.6%)	DUT B [4.09 s]	DUT D [4.01 s]	DUT C [4.00 s]
Inst 2	36.0 s (68%)		DUT B [9.01 s]	

Table of Contents

The Resource Arbiter in a Nutshell	4
Introduction and Motivation	5
Setting up the Resource Arbiter Server	7
Create your own PathWave Test Automation Plugin	7
Model your Instruments and DUTs	7
Setting up the Testbench	10
Setting up the Connections	12
Consuming the Resource Arbiter Service with PathWave Test Automation as a client	14
Setting up the PathWave Test Automation Client	14
Advanced Features	16
Example Programs	18
Using the Resource Arbiter Service without PathWave Test Automation as a client	23
Sample API Client	23
REST API Quick Reference	25
REST API Documentation	26
Visualizing Resource Utilization with the PathWave Test Automation Timing Analyzer	40
Using Resource Arbiter GUI	41
Installing and Accessing the Resource Arbiter GUI	41
Adding Resources	42
Updating Resources	44
Deleting Resources	44
Viewing Resource Utilization	44
Disabling Resources	44
Unlocking Resources	45
Resetting Utilization Counts	45

Notices

DFARS/Restricted Rights Notice

If software is for use in the performance of a U.S. Government prime contract or subcontract, software is delivered and licensed as “Commercial computer software” as defined in DFAR 252.227-7014 (June 1995), or as a “commercial item” as defined in FAR 2.101(a) or as “Restricted computer software” as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Keysight Technologies’ standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Warranty

THE MATERIAL CONTAINED IN THIS DOCUMENT IS PROVIDED “AS IS,” AND IS SUBJECT TO BEING CHANGED, WITHOUT NOTICE, IN FUTURE EDITIONS. FURTHER, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, KEYSIGHT DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED WITH REGARD TO THIS MANUAL AND ANY INFORMATION CONTAINED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. KEYSIGHT SHALL NOT BE LIABLE FOR ERRORS OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, USE, OR PERFORMANCE OF THIS DOCUMENT OR ANY INFORMATION

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

© Keysight Technologies, Inc.

For support, go to the [PathWave Test Automation Web page](#) and click Contact an Expert.

The Resource Arbiter in a Nutshell

The Resource Arbiter is a service that enables the sharing of instruments between participating DUTs without forcing the DUTs to know about each other, while also guaranteeing that they do not interfere with each other in their use of the instruments.

The Resource Arbiter Server has the following functions and characteristics:

- Lock/Allocate an instrument based on:
 - Instrument class (e.g. "Any DCA") or a specific instrument
 - Available routes
 - Cost of switching
- REST API for use from any client. An example that explains the API and its use has been provided.
- Switch Operation:
 - Easily integrate any Switch by implementing a simple `ISwitch` interface for the Switch Instrument driver
 - Routes for DUT <-> locked instruments automatically set
- Visualize resource usage history with PathWave Test Automation Timing Analyzer
- Statistical data via API:
 - Insight into the current usage state of the instruments as well as the pending requests to lock them
 - Enables analysis of the overall utilization of your instruments
- Robust in times of Client failures (crashes)
- Special support for Instrument Service and Cloud Computing
- Internally uses PathWave Test Automation and PathWave Test Automation related data structures

Resource Arbiter client integration for Test plans running on PathWave Test Automation ("Client") offers the following capabilities:

- configure, enable/disable interaction with Resource Arbiter
- automatically inject communication to Resource Arbiter
- push/pull instrument/DUT configuration changes

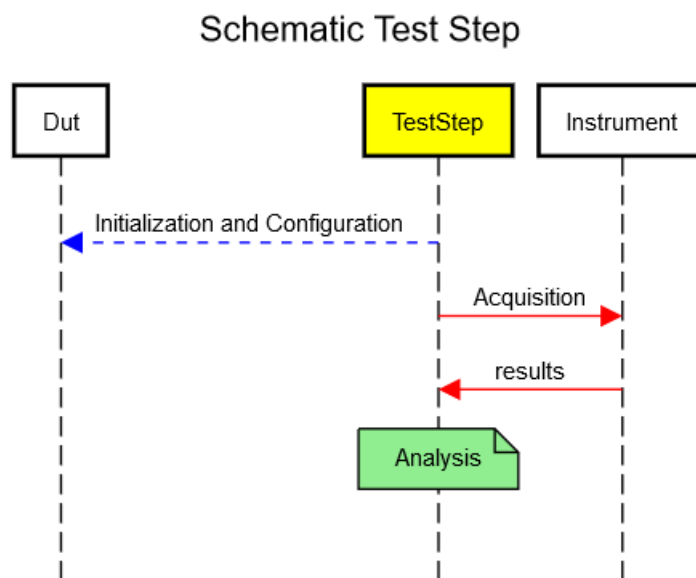
Introduction and Motivation

When faced with the challenge to test as many devices as possible in parallel, the obvious approach is to buy one piece of measurement equipment per DUT that shall be tested in parallel. However, in cases where one DUT must undergo a series of tests with different instruments, one quickly finds out that the instruments are only utilized for fractions of the time.

The Resource Arbiter service enables you to test multiple devices in parallel, without the need to have one instrument of each type per DUT. Instead the pool of instruments is shared among the devices under test.

This sharing of devices has to happen in a manner that the different measurements do not interfere with each other to avoid corruption of the results. The Resource Arbiter service provides such hardware arbitration.

To better understand the underlying concept of the Resource Arbiter, let us first take a look at the basic sequence of events that happen during the acquisition of measurement data.



Schema of a Test Step

There are three participants in this abstracted scenario. The central one (yellow) is the test step. This test step can either be performed by a person or be completely automated. The entity to the left is the device under test and the one to the right is the measurement equipment that is needed to perform the required measurements.

The test step consists of three phases:

1. The DUT is set up in a state that enables it to send or receive the required data.
2. The Data is acquired with the measurement equipment. This includes setting up the instruments as well as the reception of the results.
3. The returned results are processed, analyzed, or stored.

Note that this is a very abstracted view on a test step. A test step might consist of multiple cycles of communication with the DUT and then acquisition with the measurement equipment but the overall procedure is more or less identical.

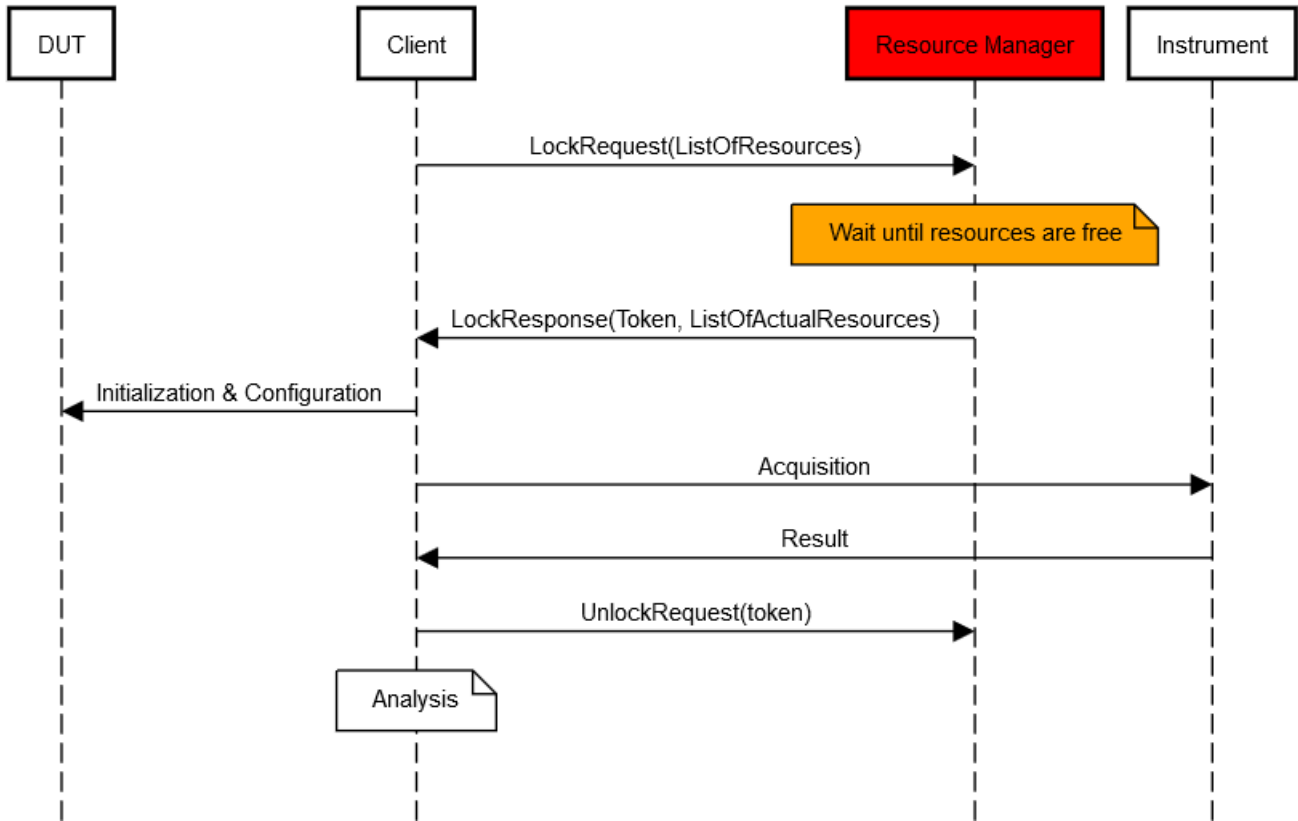
As mentioned above, to be able to test more complex devices, one usually needs a number of test steps, each using potentially different instruments. This leads to the concept of a test plan, which is then executed by a test sequencer. The test plan consists of a number of test steps and is responsible for the testing of one DUT. The test plan is also responsible to keep the order of the test steps, in case of dependencies among test steps but is not aware of any other test plans running concurrently.

The Resource Arbiter is a central component, using which a test plan does not need to know about all the other test plans in order to avoid the concurrent usage of one single instrument. The Resource

Arbiter is configured to know about all DUTs and instruments that are present in the currently active profile in the test station. Whenever a test step needs to use one or more instruments, it requests permission to use them from the Resource Arbiter, instead of using them in an uncontrolled manner. When the test step has finished its acquisition, it releases the lock of the instrument resources either explicitly or based on the expiry of a predefined lock period. Note that it is not necessary to hold the lock for the instruments for the Analysis phase (shown in green).

This procedure leads to the following sequence of events:

Test Step with Resource Manager



Test Step Sequence with locking

In red, you can see the Resource Arbiter. Before the test step executes, it makes a call to the Resource Arbiter to request the desired instruments. This call happens via REST, which makes it possible to be sent from almost every programming language and platform. The call only returns a response once the requested instrument resources are available (as indicated by the orange note). This means that upon receiving the response, the test step can execute its measurements without interfering with other test steps. Once the test step has completed its acquisition, it releases the lock of the instruments (again via a REST call) to free them up for use by other test steps that possibly operate on other DUTs. For a detailed references on the REST API calls, refer to [REST API Documentation](#).

Two use models are supported:

1. Calling the REST API from any client. This is described in [Using the Resource Arbiter Service without PathWave Test Automation as a client](#).
2. Using PathWave Test Automation as Test Sequencer, the communication to the Resource Arbiter can be injected automatically. This is described in [Consuming the Resource Arbiter Service with PathWave Test Automation as a client](#).

Setting up the Resource Arbiter Server

Perform the following steps to set up the Resource Arbiter Server:

1. Create your own PathWave Test Automation Plugin
2. Model your DUTs and Instruments
3. Setup the Testbench

Create your own PathWave Test Automation Plugin

As specified in the Installation instructions, it is highly recommended to use the PathWave Test Automation SDK in order to create a custom plugin that contains all the components you need for the use of the Resource Arbiter.

Perform the following steps to create a PathWave Test Automation plugin:

1. Open Visual Studio (version 2015 or later).
2. Select File -> New -> Project and select **OpenTAP Plugin Project (.NET Framework)**.
Note: Refer to the *Installing PathWave Test Automation* section in the Installation Guide.

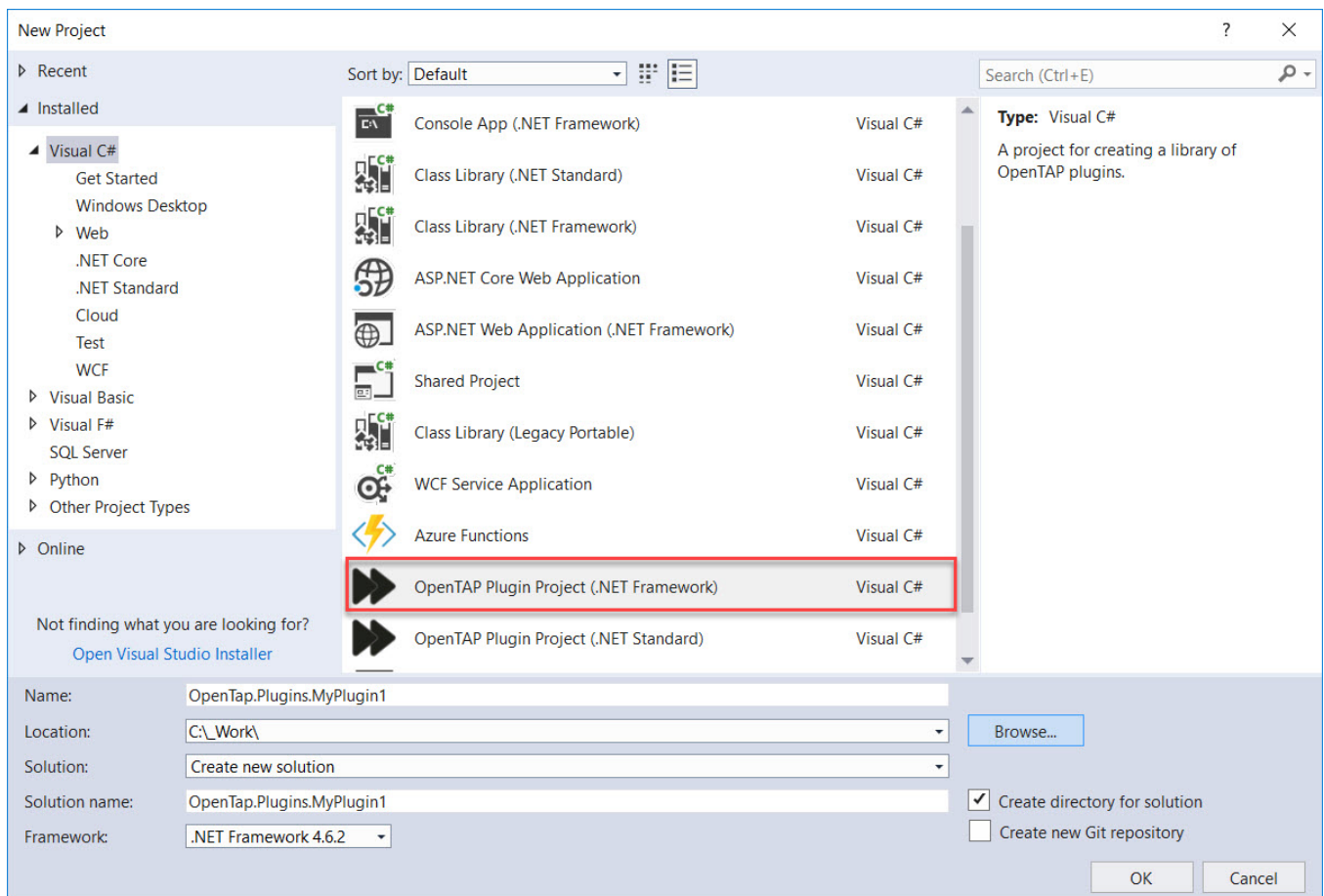


Figure: PathWave Test Automation Project

Ensure that you have selected the correct .NET Framework (version 4.6.2). Click **OK**. You will get a new PathWave Test Automation Plugin project with an automatically generated test step, called Step. You will need this boilerplate step later on, but first create the Resources to be used within this test step.

Model your Instruments and DUTs

Model your DUT

Right-click your project in the Solution Explorer in Microsoft Visual Studio and select **Add > New Item**.

In the following dialog, select OpenTAP DUT. For a thorough description of the offered possibilities, please refer to the Keysight PathWave Test Automation Documentation.

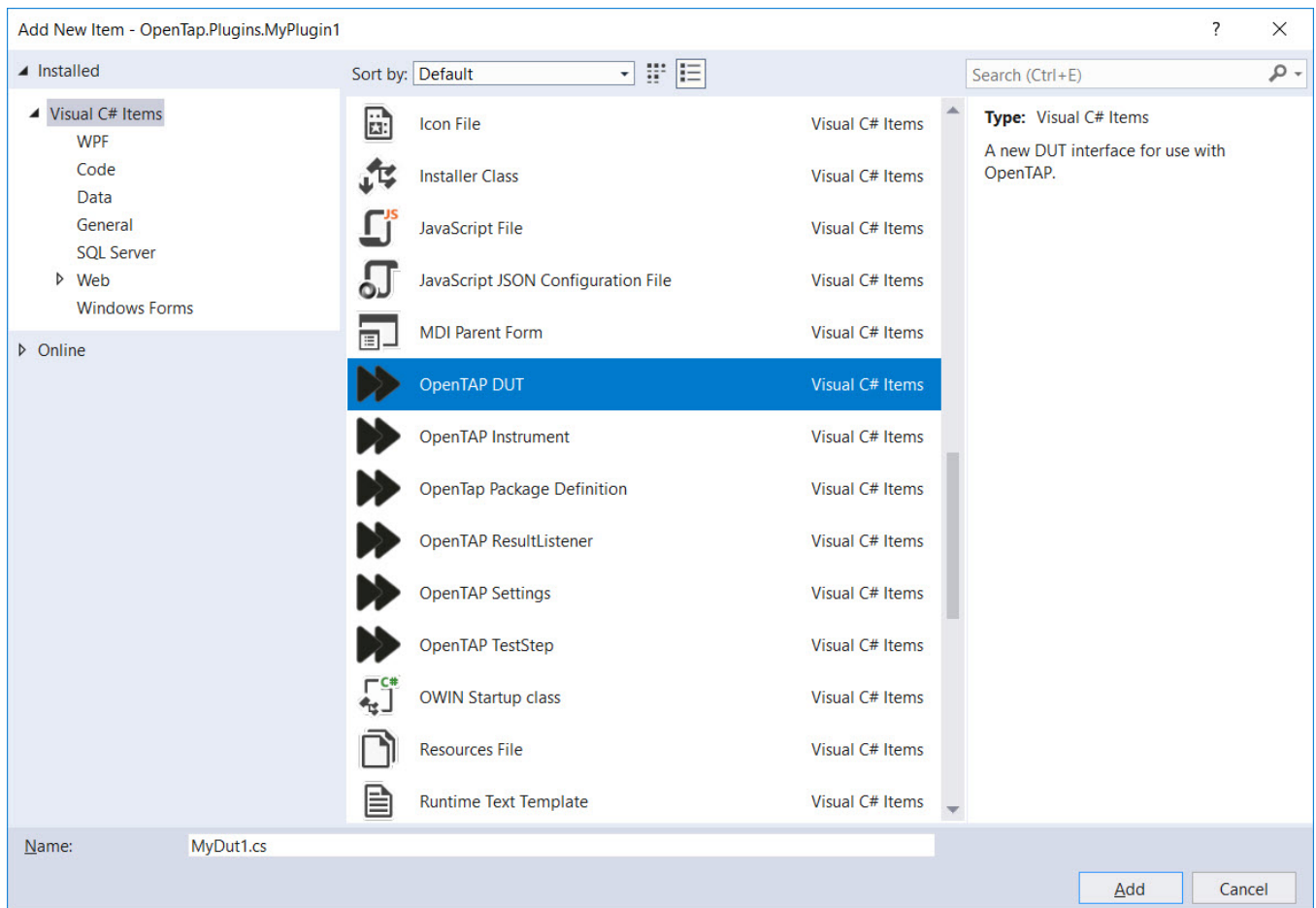


Figure: PathWave Test Automation DUT

For the purpose of using the Resource Arbiter it is sufficient to specify the ports, that are present at the DUT.

```
public InputPort Input1 { get; private set; }
public OutputPort Output1 { get; private set; }
```

will add an input and an output port to the DUT. In order to generate a complete model of the DUT, please add as many InputPorts and OutputPorts as you will need to use on your device. Note that the getter of the Ports has to be public, so that the test step will actually have access to these structures.

Do not forget to initialize the Ports in the constructor of the DUT if you want to model connections between your DUTs and instruments.

```
Input1 = new InputPort(this, "The name that you want to be displayed");
```

This constructor will be automatically called when you later add a DUT of this type to your virtual work bench inside PathWave Test Automation.

A sample DUT class that can be used by the Resource Arbiter could look like this:

```
namespace OpenTap.Plugins.YourPluginName
{
    [Display("ExampleDut", Group: "ExampleGroup", Description: "Add a description here")]
    public class ExampleDut : Dut
    {
        #region Ports
        // ToDo: Add property here for each parameter the end user should be able to change.
        public InputPort Input1 { get; private set; }
        public OutputPort Output1 { get; private set; }
        #endregion
    }
}
```



```

/// <summary>
/// Initializes a new instance of this DUT class.
/// </summary>
public ExampleDut()
{
    // ToDo: Set default values for properties / settings.
    Name = "MyDUT";
    Input1 = new InputPort(this, "Input1PortName");
    Output1 = new OutputPort(this, "Output1PortName");
}

/// <summary>
/// Opens a connection to the DUT represented by this class
/// </summary>
public override void Open()
{
    base.Open();
    // TODO: establish connection to DUT here
}

/// <summary>
/// Closes the connection made to the DUT represented by this class
/// </summary>
public override void Close()
{
    // TODO: close connection to DUT
    base.Close();
}
}
}
}

```

Please specify all the classes of DUTs that you want to use in this manner, so that you will be able to access them inside PathWave Test Automation.

Model your Instruments

In addition to defining the DUTs that you want to model in your test station, you need to model your instruments in a similar way. Right-click your project in the Solution Explorer in Microsoft Visual Studio and select **Add > New Item**. In the following dialog select OpenTAP Instrument.

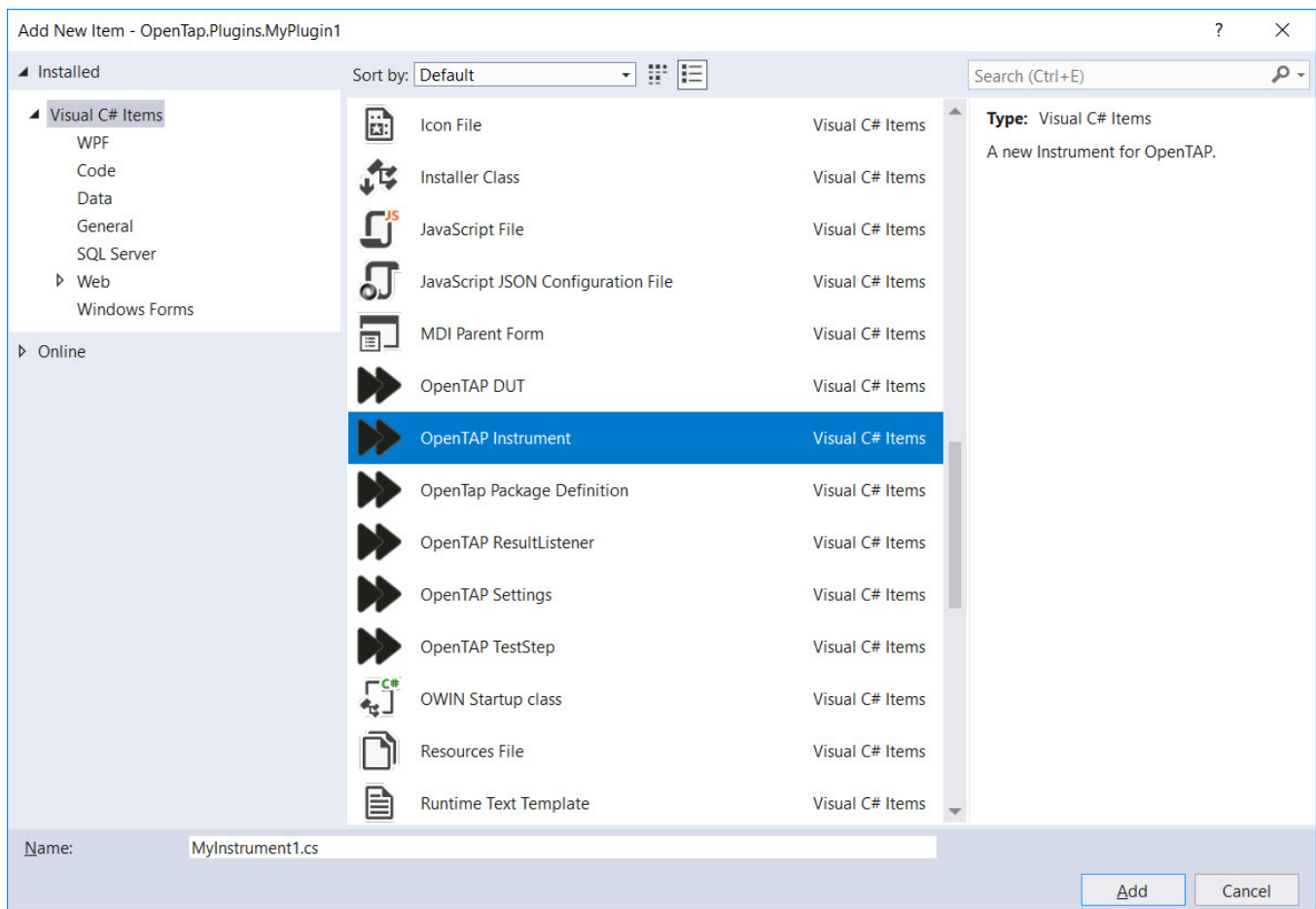


Figure: PathWave Test Automation Instrument

You will get a class with boilerplate code for an instrument. If you want to automatically get SCPI capabilities for this instrument, you should consider inheriting from the `ScpiInstrument` class instead of the `Instrument` class. Next, define the Ports on the instruments that you want to use in a similar manner as you did with the DUTs.

Additional Considerations for a switch

Resource Arbiter can interact with a switch and set the switch positions according to the DUT to instrument connection that it assigns. In this case the Resource Arbiter needs to know how to set the switch positions.

Implement `ISwitch` interface

Additionally derive your Instrument Driver from `Keysight.Tap.Plugins.ResourceArbiter.Api.ISwitch` and add a reference to `%TAP_PATH%\Packages\Resource Arbiter\Keysight.Tap.Plugins.ResourceArbiter.Api.dll`. As an example see `%TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Example Classes\ExampleSwitch.cs`. More information about the needed methods and their description can be found there.

After performing the above steps, compile your code by building the solution. Once you successfully build your solution, you will observe that your plugin gets added to the `%TAP_PATH%\Packages` directory location.

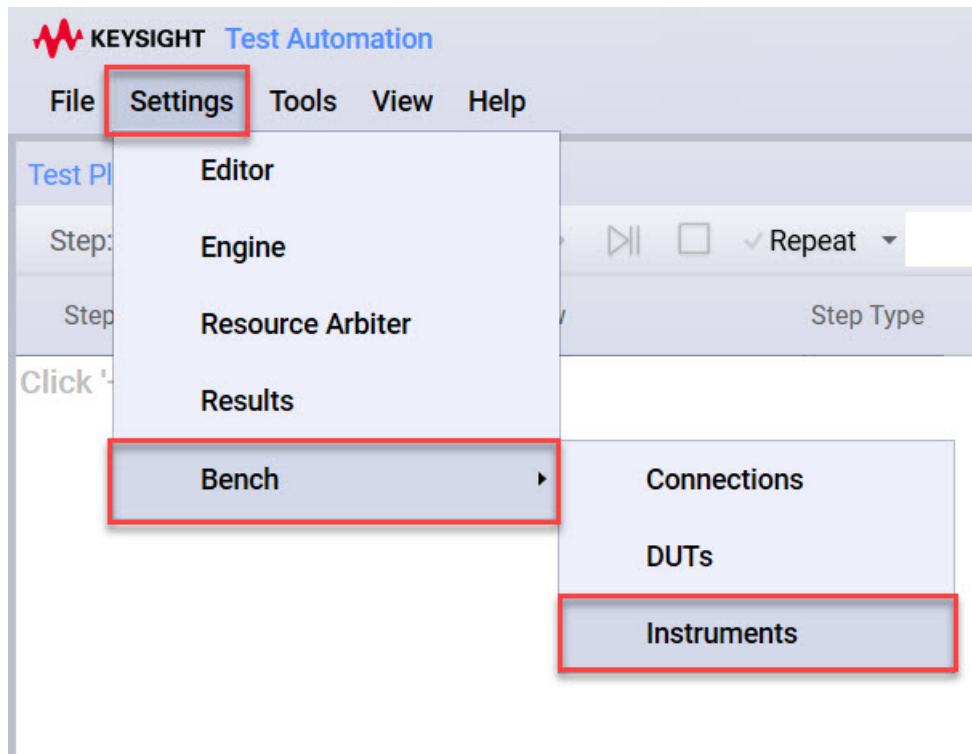
Setting up the Testbench

Add Instruments and DUTs

In the previous steps you set up the classes of instruments and DUTs that you want to use. The following section instructs you on how to instantiate the actual instruments that are physically present on your test station.

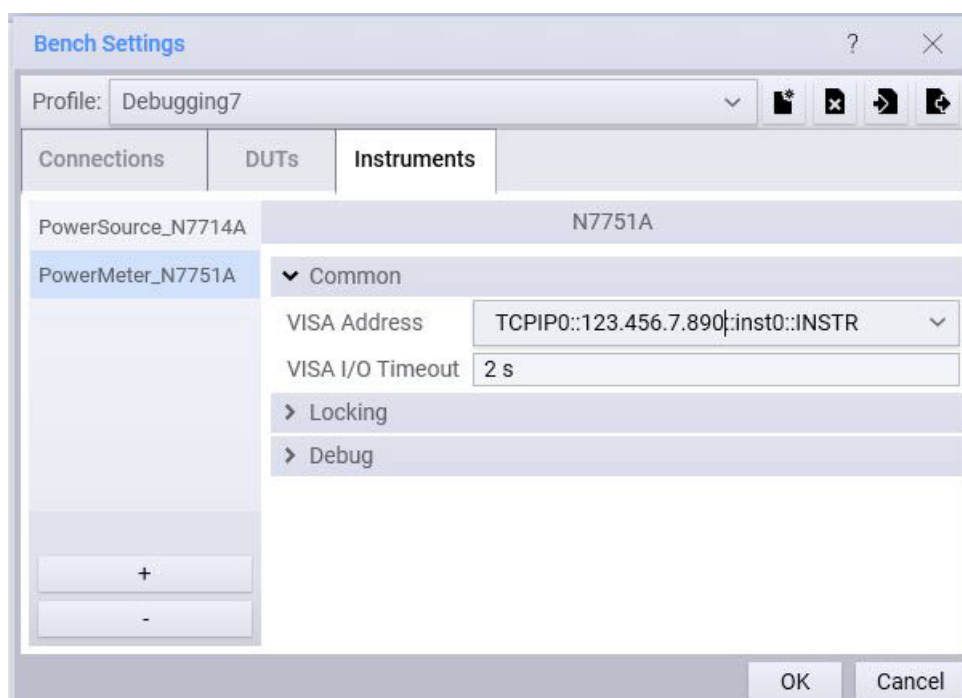
Perform the following steps in the PathWave Test Automation GUI:

1. Start the PathWave Test Automation program.
2. Select **Settings > Bench > Instruments**.



Select Bench Settings

3. In the following screen, add new instruments by clicking the “+” button to add instruments. In the list that appears, you will find all the instrument types that are known in your PathWave Test Automation environment. These should include the instruments that you set up previously.
4. Once you added an instrument, you are able to set all the properties contained in the instrument class that have a public set method. As an example, you see the following screenshot of the instrument settings screen, with an added simple SCPI-Instrument.



Properties of an instrument

5. Add and customize all the instruments in the way you want to use them. **Do not forget to add all the switches that you want to use in your setup as instruments as well.**
6. Proceed to the “DUTs” tab and do the same with the DUTs on you test station.

Setting up the Connections

The connection settings can be accessed by selecting **Settings > Bench > Connections**. For a profile with no configured connections, the following window is displayed.

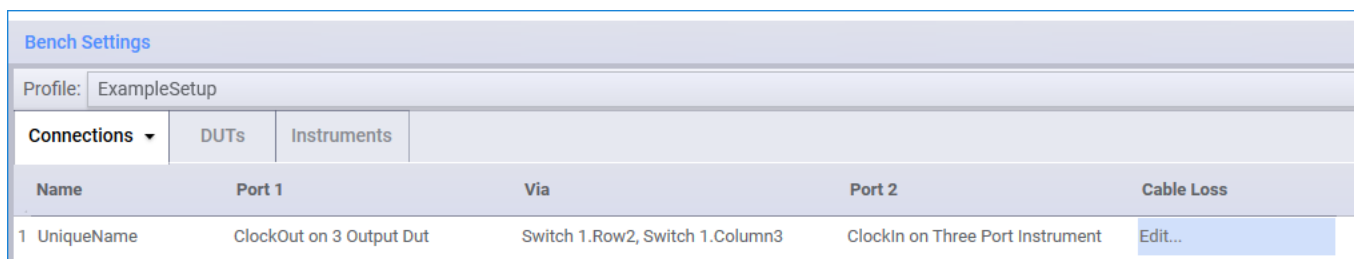


Empty Connection Settings

After double-clicking in any of the cells of the first row, choose the connection type that you want to add. The new connection will appear in the list and you can then set all the properties that it exposes. Please adhere to the following restrictions:

1. The Connection names must be unique.
2. To make the connections useful at all, both ports must be set. To set a port, click on the corresponding field in the connection row and choose the port from the drop down. If you are not offered any ports ensure that your testbench contains DUTs and instruments that own ports.
3. Add all the *ViaPoints* to the connection for all the devices that lie on the connection path to allow the Resource Arbiter to ensure that no mutually exclusive connections are activated in parallel.

Here is an example for a sufficiently specified connection:



Example of a single Connection

In addition to the mandatory settings, you can also specify the cable loss. You can also inherit from the connection class and define your custom connections class to specify all the properties that you need to associate with a connection.

Consuming the Resource Arbiter Service with PathWave Test Automation as a client

While you can consume the Resource Arbiter Service with the environment of your preference, as long as it is capable of sending HTTP requests and receiving responses, it is very convenient to do so with PathWave Test Automation as a client.

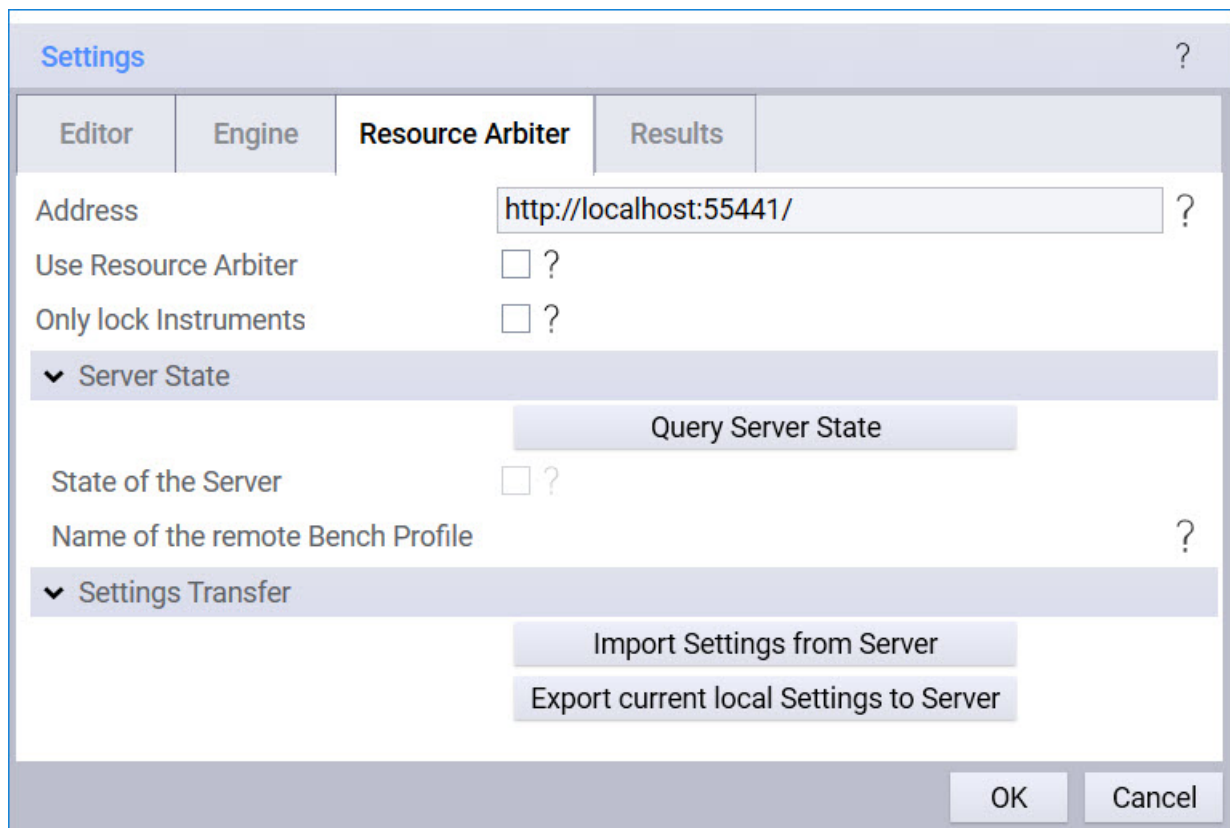
This section documents the setup and use of this scenario and concludes with information on some advanced capabilities of this approach.

Setting up the PathWave Test Automation Client

The Resource Arbiter plugin seamlessly integrates with the PathWave Test Automation Platform (TAP). In fact, the platform hides all the necessary communication from you in most cases.

The only requirement for a seamless use of the Resource Arbiter is that each instrument that a test step requires must be declared as a public property.

When the Resource Arbiter client package is installed on your system, it automatically integrates its settings with the other platform-related settings. To view the Resource Arbiter settings, click **Settings > Resource Arbiter** from the menu bar. The Settings dialog showing the Resource Arbiter tab is displayed.



Resource Arbiter Settings

Enrich Instruments and DUTs with Functionality

During the setup of the Resource Arbiter server you already specified the instruments and DUTs that you want to use on your testbench. In addition to just defining them, you can implement all the functions that you need for the resources in the DUT and instrument classes on the client side. Just copy the dll from the server and reference it in your plugin on the client side. Please refer to the examples or the PathWave Test Automation documentation regarding how to best leverage the instrument and DUT constructs.

Define your own Test Steps as a PathWave Test Automation Plugin

When you created the PathWave Test Automation plugin in the server setup, there was a test step automatically generated. In these test steps you can define the behavior of your test cases, as well as the resources that they need to perform their measurements. For additional examples please refer to those given in the examples folder or take a look into the PathWave Test Automation documentation.

Specifying the Resource Arbiter Location

To use the Resource Arbiter service, you must ensure that it is enabled in the **Settings** window. Also, the Settings window enables you to specify the server address where the Resource Arbiter service is running. From the PathWave Test Automation GUI, click **Settings > Resource Arbiter** to access the Settings window. Within this window, the **Resource Arbiter** tab is displayed. - Enter the Resource Arbiter server address in the format shown in the Address box. - Ensure that the **Use Resource Arbiter** check box is selected. - If you do not want to consider the connections between DUTs and instruments, but are only interested in locking the required instruments, you can enable the “Only lock Instruments” option.

Deciding for a Resource Strategy

When PathWave Test Automation is used as the client for the Resource Arbiter, the environment takes care of the locking and unlocking of all the resources that are needed by the test steps. By default all resources are locked for the entire *Test Plan*. To optimize the resource utilization, we recommend changing this behavior to only lock the instruments for the duration of the *Test Steps* in which they are actually used. The Resource Strategy setting is located at **Settings > Engine**. If you want the instruments to be locked only for the duration of the test step that needs them, please set this property to “Short Lived Connections”. Note that the Short lived Connections strategy will lead to a call of the *Open()* method for all involved resources of each test step before the test step executes, as well as a call to the *Close()* method of all involved resources after each test step. This means that you should design your test steps in a way that they do not rely on any initial state of the instruments when using this strategy. The assumption of the Resource Arbiter is that each test plan corresponds to one DUT. Please note that the “default” option of the Resource Strategy can lead to errors when multiple DUTs are used within one test plan. This can be mitigated by choosing the “Only lock Instruments” setting. As all the needed resources are requested for the complete test plan, the connections will not be automatically changed when different instruments need to be connected to the same DUT in different test steps. If you want to use different instruments that need mutually exclusive connections in the scope of a test plan, please select the “Only lock Instruments” option in the Resource Arbiter settings to avoid errors.

Locking “ANY” Instrument, that matches a type

In many cases the test plan does not rely on a certain physical instrument, but will take anything, that is able to perform the desired measurement. So in addition to requesting a set of specific instruments, the Resource Arbiter allows you to request any instrument, that matches a specific type as well. To do that with PathWave Test Automation as a client, you have to specify which of the instrument properties in your test step are not relying on a specific piece of hardware. The concept here is called a placeholder, which can be declared in the following two ways. First, you can declare it via the GUI. As soon as there are multiple instruments on the test bench that match the instrument type that your test step uses, you will be offered the “Any ...” option in the drop-down menu for the respective instrument in the Step Settings window. If the usage of the GUI is not an option, one can also declare an instrument property as a placeholder via code as shown in the below code snippet.

```
TestStep step = new TestStep();
step.MyInstrument = null;
step.GetResourcePlaceholders().Add(new ResourcePlaceholder{step,
TypeData.GetTypeData(step).GetMember("MyInstrument")});
```

If you specified any instruments as placeholders, these instruments will get substituted automatically by the instruments that were chosen by the Resource Arbiter before the test step executes. This enables you to treat a pool of instruments just like one instrument of that class. Once the test step finished, all placeholders are reset and the instruments are released.

Transfer of the test bench topology

The Resource Arbiter service runs an instance of PathWave Test Automation from which it pulls the bench settings such as instruments, DUTs and connections. If you run PathWave Test Automation on the client side as well it is paramount that these settings are identical on the server and client side. In order to achieve that, you can transfer the settings between machines by using the export and import function of the PathWave Test Automation bench settings, which produces/consumes `.TapSettings` files. This way, you do not have to take care of physically copying the settings files from your server to the clients and vice versa. The Resource Arbiter Settings offer the exact same functionality via HTTP. The "Import Settings from Server" function downloads the current test bench profile from the Resource Arbiter server and changes the local bench settings to this profile. Note that the local settings will get **overwritten** if they share the same name with the ones present on the server. This method also saves the settings as a `.TapSettings` file at `%TAP_PATH%\Settings\Temp\PROFILE_NAME.TapSettings`. It is also possible to push the local settings profile to the Resource Arbiter Server using the "Export Current Settings" button. Please note that doing this might heavily impact other clients of the same Resource Arbiter instance, as their settings will be out of sync with the server.

Advanced Features

For the majority of use cases the Resource Arbiter Client (PathWave Test Automation) Plugin takes care of all the necessary communication with the server without further configuration. However there are a couple of additional features that enable some advanced use cases.

Deferred Actions

As mentioned in the introduction, the instruments only have to be locked for the time it takes to acquire the data. If this data has to be post processed, the instruments can and should be released before this post processing starts. This post processing can be realized in a PathWave Test Automation test step by using the `Results.Defer()` method. The use of this method will lead to an automatic release of all the resources that the test step used, as soon as all code in the `Run()` method of the test step outside of this deferred action has been executed. In the below code snippet taken from the examples, the test step will release its resources before the Sleep and the multiplication.

```
Results.Defer(() =>
{
    TestPlan.Sleep(1000);
    ResultAfterCalculation = RawResult * RawResult;
});
```

Connections Placeholders

In some scenarios it is important to know what connections were actually made. Some examples where this might be interesting:

- accessing the calibration data attached to the connections to compensate for connection specific losses.
- specific initializations to the device or instrument are required to be performed, depending on what switches were changed, e.g. to lock to a changed clock.
- some code needs to be placed around the actual switching, e.g. powering down the device before switching and then powering it up after switching.

This information is provided by the `ConnectionsPlaceholder` CLASS in `%TAP_PATH%\Packages\Resource Arbiter\Keysight.Tap.Plugins.ResourceArbiter.Api.dll` to get the information about the chosen connections between the DUT and the instrument.

If you declare a property of this type with public setter and getter in your test step, the Resource Arbiter Client will fill the list of names of the connections that were chosen by the Resource Arbiter Server along with the list of actually changed VIAs before the test step executes. This way the test step can assume this data to be contained in the list. To understand the usage of the `ConnectionsPlaceholder` class via an example, refer to the code in `ThreePortStep.cs` available in `%TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Example Classes`. Here are a few snippets from this example:

```
public ConnectionsPlaceholder MyConnections { get; set; }
....
....
foreach (var connectionNameAndViaIdxs in MyConnections.ConnectionNamesAndViaIdxs)
{
```



```

// locate the Connection data structure
Connection connection = ConnectionSettings.Current.FirstOrDefault(conn => conn.Name ==
connectionNameAndViaIdxs.Item1);
if (connection is RfConnection)
{
// This is an RF connection. We can get the cable loss per frequency to compensate in our setup
// Note: RfConnection and DirectionalRfConnection are PathWave Test Automation built-in types.
// For other loss characteristics, derive your own class from Connection base class.
RfConnection rfConnection = (RfConnection)connection;
if (rfConnection.CableLoss.Count > 0)
{
var loss1e6 = rfConnection.GetInterpolatedCableLoss(1e6);
// do something in the setup to compensate the loss
Log.Info(Invariant($"Cable loss @1e6 = {loss1e6}"));
}
else
Log.Info(Invariant($"Cable loss - no table given."));
}
}
....
....

```

Locking Resources for multiple test steps

The two basic resource strategies were introduced in the previous section. In addition to locking the resources for the whole test plan or separately for each test step, which is both configurable via the engine settings, it is also possible to lock them for several test steps combined. The basic idea is to define the needed resources in one test step and then to make use of these resources in its child test steps. Thus, the lock will be held for the execution of all child test steps. This feature basically allows for arbitrary scopes for the resource locks and makes the use of PathWave Test Automation as a client exceedingly flexible. For a more detailed example, please refer to the [ResourceArbiterExampleWithoutSwitchManagerAdvanced](#) example. Please note that this mechanism depends on the “Short Lived Connections” resource strategy and that you should not define the resources as properties of the child steps again, because this might lead to a deadlock, since it would generate a blocking request for an already locked resource.

Modeling multiport DUTs

By default, the Resource Arbiter will connect the DUT that is used in a test step with all the instruments that are used in the same test step, as long as the connections between the DUT and the instruments are unambiguous. This means that we recommend only adding those connections to the connections list of PathWave Test Automation that you really consider activating over the course of your test plans. For example if you have a data port and a clock port on both the DUT and the instrument, you should only add a connection between the data port of the DUT and the clock port of the instrument (and vice versa) if you really consider this connection to be valid in a test case.

While the above case can be mitigated by just connecting the data ports and the clock ports to avoid ambiguous matchings, this mechanism fails in case of a DUT with multiple data ports, that have to be connected to the same port of an instrument.

In cases like this we recommend modeling the multiport DUT as a set of DUTs that only contain one port of each flavor and thus eliminating the ambiguity.

Complex locking scenarios

If the automatically injected communication with Resource Arbiter is not fulfilling all needs, the Rest API can be used directly within a Test Step. More information about the Rest API can be found in [Using the Resource Arbiter Service without PathWave Test Automation as a client](#)

Locking Compute Nodes

For doing longer lasting computations, it can be beneficial to offload the actual work to a compute node. To allocate such a compute node, the Resource Arbiter can also be used. The `ICompute` interface in `%TAP_PATH%\Packages\Resource Arbiter\Keysight.Tap.Plugins.ResourceArbiter.Api.dll` is intended to model the available compute nodes, similar to how the instruments are modeled. The Lock / Unlock of such a compute node is usually done in a [Deferred Action](#), and needs to be done using the REST API directly. See [The Lock / Unlock of such a compute node is usually done in a Deferred Action](#), and needs to be done using the REST API directly. See [Complex locking scenarios](#).

Infinite Locking and Counted Locking

If an Instrument can be locked by multiple clients in parallel, the ICountedLock interface enables this behavior. When an Instrument implements this interface it will expose two properties, one is a boolean that can be set to true to allow this resource to be locked for an infinite amount of times simultaneously. The other one is a number indicating, how often the resource can be locked simultaneously. This latter number will only be taken into account, if the instrument is not marked as infinitely lockable.

Capabilities based Locking

In addition to locking instruments by name or by the Class/Interface that they inherit from, it is possible to assign your instruments capabilities to later request an instrument that has a desired capability. This can be achieved by implementing the IResourceCapabilities Interface, which will expose a Capabilities string that will be interpreted as a comma separated list of Capability-identifiers. If, for example, an instrument has the Capabilities "53GHz,CDR", a request specifying either "53GHz" or "CDR" as an instrumentIdentifier, will receive a lock for this instrument or any other instrument matching the requested capability, once it becomes available.

Autostarting the Resource Arbiter

The Resource Arbiter can be configured to start automatically at system startup. This behavior can be enabled by running the "Resource Arbiter Autostart.bat" at the Test Automation location (%TAP_PATH%) as an administrator. This will configure the Resource Arbiter as a scheduled task. With this scheduled task the Resource Arbiter will automatically start with the system, i.e. even before a user is logged in. However, the Resource Arbiter will only start to work once the Keysight License Service is running and a K58108A License is found. To unregister this task please execute the "Resource Arbiter Autostart Unregister.bat" as an administrator from the same location.

Example Programs

The Resource Arbiter comes with some example programs to demonstrate its resource arbitration capabilities. There are two sets of example programs, those that come with the server-side package and those that come with client-side package. The following sets of example programs ship with the client-side package and are described in this section.

- Resource Arbiter Example Without Switch Manager
- Resource Arbiter Example Without Switch Manager2
- Resource Arbiter Example Without Switch Manager Advanced

The Resource Arbiter Client API program ships with the server-side package and is described in the [Example REST API Client](#) section.

These example programs are based on a few predefined DUTs, instruments, and test steps. These predefined entities can be used to try out the REST API commands or for modeling your own DUTs, instruments, and test steps. The .cs files for these example entities are available from

%TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Example Classes.

The following table briefly describes these .cs files.

Example file	Description
OneOutputDut.cs	Declares <code>OneOutputDut</code> class with one output port <code>Output1</code> . Also declares a sample <code>Configure</code> method that introduces a time lag by making the test plan sleep for 1000 milliseconds.
ShortMeasuringInstrument.cs	Declares <code>ShortMeasuringInstrument</code> with one input port <code>Input1</code> . Also declares a sample <code>MakeMeasurement()</code> method that introduces some processing time by making the test plan sleep for 2500 milliseconds and generates a random double as the measurement result.
	Declares one DUT of <code>OneOutputDut</code> class and one

Example file	Description
ShortMeasurementStep.cs	Instrument of ShortMeasuringInstrument class. The Short Measurement Step calls the Configure method on DUT and MakeMeasurement method on the Instrument. This class also demonstrates the use of Defer method that is primarily used to release the resources and perform post-processing tasks.
LongMeasuringInstrument.cs	Declares LongMeasuringInstrument with one input port Input1. Also declares a sample MakeMeasurement() method that introduces some processing time by making the test plan sleep for 8000 milliseconds and generates a random double as the measurement result.
LongMeasurementStep.cs	Declares one DUT of OneOutputDut class and one instrument of LongMeasuringInstrument class. The Long Measurement Step calls the Configure method on DUT and MakeMeasurement method on the Instrument. This class also demonstrates the use of Defer method that is primarily used to release the resources and perform post-processing tasks.
ThreePortResources.cs	Declares ThreeOutputDut with three output ports - ClockOut, TriggerOut, and DataOut. Declares a sample Initialize method that introduces a time lag of 1000 milliseconds. Declares an example IThreePortInst interface to show that it is possible to create extensive instrument inheritance hierarchies and then let test steps decide which kind of specialization they need in order to perform their measurements. The IThreePortInst interface declares three input ports ClockInput, TriggerInput, and DataInput and a MakeMeasurement function. Further, an example implementation of the interface is available that declares the three instrument ports and adds functionality to the method by introducing processing time of 1000 milliseconds.
ThreePortStep.cs	Declares ThreeOutputDut, IThreePortInst, and ConnectionsPlaceholder objects. In this test step, the Instrument is already assigned to the Test Step by the Resource Arbiter. However, the Resource Arbiter does not take care of switching the needed connections (as in this case, Switch Manager is not being utilized). The Resource Arbiter populates the connections placeholder with the names of the chosen Connections. These are then manually switched on the respective instruments to guarantee the correct signal flow.
ExampleSwitch.cs	This is an abstract base class for a matrix switch. This instrument enables setting a switching cost value that will be used by the Resource Arbiter to prioritize those request that have a comparably low total switching cost.
LockInstrumentsMultipleSteps.cs	Declares multiple Test Step classes. LockInstrumentsMultipleSteps demonstrates how two instruments of type ShortMeasuringInstrument and LongMeasuringInstrument can be locked, and used by child steps UseLockedShortInstrumentStep, UseLockedLongInstrumentStep and UseLockedShortAndLongInstrumentStep. Additionally two test steps LongMeasurementOnTwoOutputDutStep and

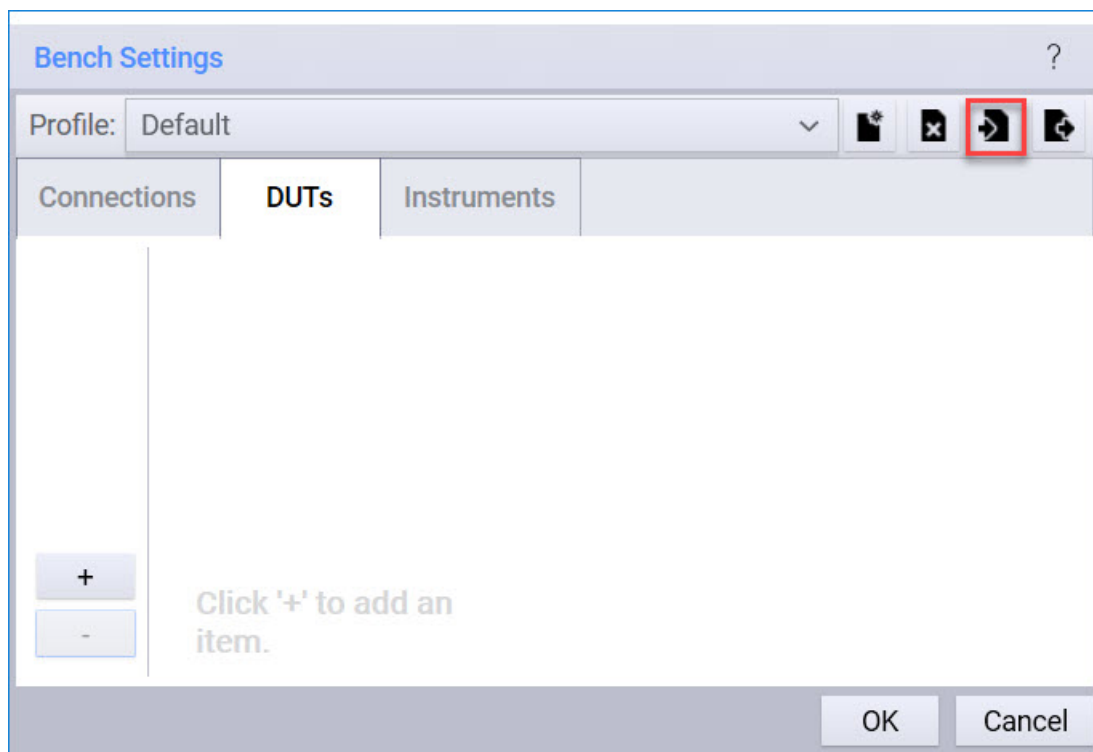
Example file	Description
	ShortMeasurementOnTwoOutputDutStep that show how a single instrument LongMeasuringInstrument and ShortMeasurementOnTwoOutputDutStep can be used.
TwoOutputDut.cs	Declares TwoOutputDut class with two output ports Output1 and Output2. Also declares a sample Configure method that introduces a time lag by making the test plan sleep for 1000 milliseconds.

The following tables display the relevant details of the example programs.

Parameter	Resource Arbiter Example Without Switch Manager	Resource Arbiter Example Without Switch Manager2
Location	%TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Without Switch Manager	%TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Without Switch Manager2
Profile	ResourceManagerExampleWithoutSwitchManager.TapSettings	ResourceManagerExampleWithoutSwitchManager2.TapSettings
Test Plan	ThreePort_Dut1.TapPlan, ThreePort_Dut2.TapPlan, ThreePort_Dut3.TapPlan, ThreePort_Dut4.TapPlan	ExampleDut1.TapPlan, ExampleDut2.TapPlan, ExampleDut3.TapPlan, ExampleDut4.TapPlan
DUTs	3 Output Dut1, 3 Output Dut2, 3 Output Dut3, 3 Output Dut4	OneOutputDut, OneOutputDut1, OneOutputDut2, OneOutputDut3
Instruments	Three Port Instrument 1, Three Port Instrument 2, Clock Switch, Trigger Switch, Data Switch	Long Measuring Instrument, Long Measuring Instrument1, Short Measuring Instrument, Matrix

Each of these example folders contains a .TapSettings file. The .TapSettings file is a profile file that contains predefined entities – DUTs and Instruments. All you need to do is to import the profile file by performing the following steps illustrated using “Resource Arbiter Example Without Switch Manager2”:

1. Click **Settings > Bench > DUT**. The Bench Settings dialog is displayed.



2. Click **Import settings profile** icon.
3. Navigate to %TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Without Switch Manager2.
4. Open the .TapSettings profile.
5. Specify an alternative name for the profile (if required) and click **Add** to add the new settings profile.
6. Notice the addition of new DUTs and new Instruments in the Bench Settings window. Also notice

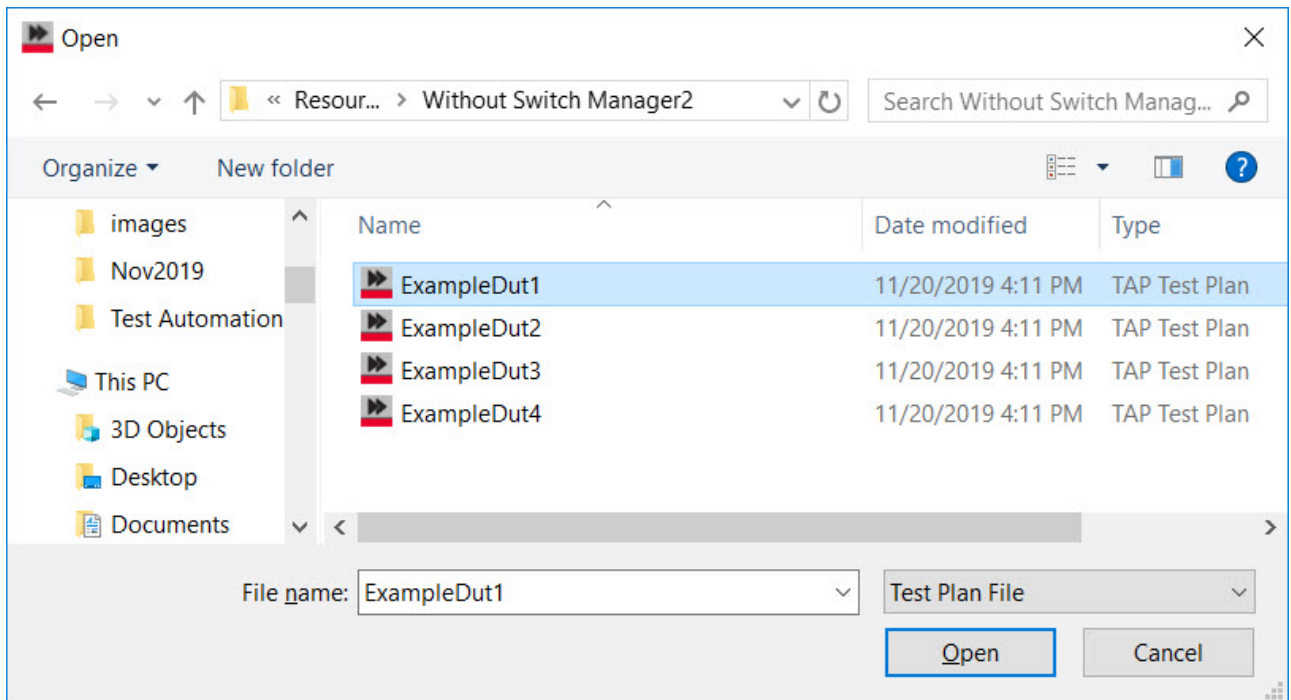
the Connections tab to view the connections between the DUTs, switch matrix, and Instruments.

7. Close the **Bench Settings** window.

Running Sample Test Plans

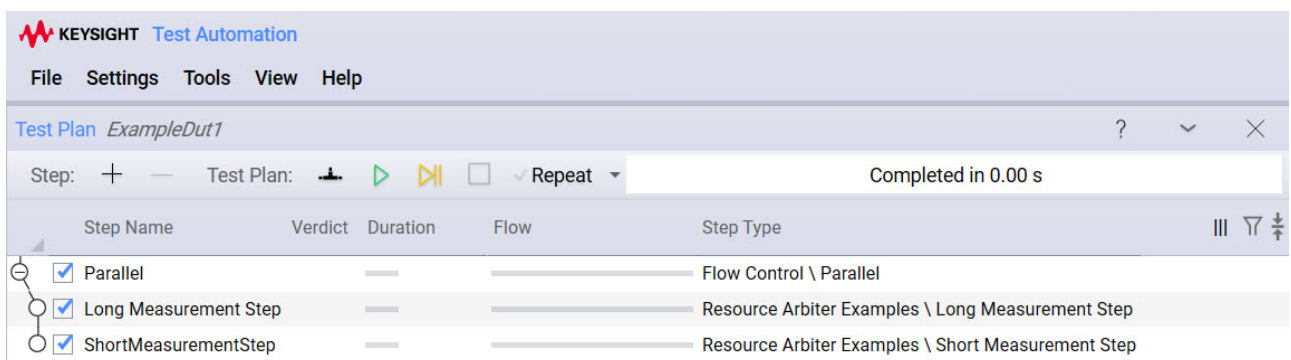
Once the profile is loaded, you can run the sample test plans provided with the examples. Perform the following steps:

1. From the PathWave Test Automation GUI, click **File > Open**.
2. Navigate to %TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Without Switch Manager2.



Opening Test Plans

3. Select the appropriate PathWave Test Automation Test Plan and click Open. The Test Plan displays the associated test steps.



Test Plan Details

4. Click the Run icon to run the test plan. The Test Plan runs and the logs display the relevant messages.

The following tables further describe the respective test plans.

Resource Arbiter Example Without Switch Manager

Test Plan	Description
ThreePort_Dut1.TapPlan	Consists of single test step ThreePortSetp running on 3 Output Dut1. Any instrument Three Port Instrument 1 OR Three Port Instrument 2 is used.
ThreePort_Dut2.TapPlan	Consists of single test step ThreePortSetp running on 3 Output Dut2. Any instrument Three Port Instrument 1 OR Three Port Instrument 2 is used.
ThreePort_Dut3.TapPlan	Consists of single test step ThreePortSetp running on 3 Output Dut3. Any instrument Three Port Instrument 1 OR Three Port Instrument 2 is used.
ThreePort_Dut4.TapPlan	Consists of single test step ThreePortSetp running on 3 Output Dut4. Any instrument Three Port Instrument 1 OR Three Port Instrument 2 is used.

Resource Arbiter Example Without Switch Manager2

Test Plan	Description
ExampleDut1.TapPlan	Consists of two steps that run in parallel threads. The Long Measurement Step and Short Measurement Step. Both the steps operate on the same DUT (OneOutputDut) but using Long Measuring Instrument and Short Measuring Instrument, respectively.
ExampleDut2.TapPlan	Consists of two steps that run in parallel threads. The Long Measurement Step and Short Measurement Step. Both the steps operate on the same DUT (OneOutputDut1) but using Long Measuring Instrument and Short Measuring Instrument, respectively.
ExampleDut3.TapPlan	Consists of two steps that run in parallel threads. The Long Measurement Step and Short Measurement Step. Both the steps operate on the same DUT (OneOutputDut2) but using Long Measuring Instrument and Short Measuring Instrument, respectively.
ExampleDut4.TapPlan	Consists of two steps that run in parallel threads. The Long Measurement Step and Short Measurement Step. Both the steps operate on the same DUT (OneOutputDut3) but using Long Measuring Instrument and Short Measuring Instrument, respectively.

Resource Arbiter Example Without Switch Manager Advanced

Test Plan	Description
Example_Dut1.TapPlan	Consists of multiple steps that run in parallel threads, testing TwoOutputDut 1. It is demonstrated how Instruments can be locked for multiple test steps, by locking them in the parent test step LockInstrumentMultipleSteps
Example_Dut2.TapPlan	Consists of multiple steps that run in parallel threads, testing TwoOutputDut 2. It is demonstrated how Instruments can be locked for multiple test steps, by locking them in the parent test step LockInstrumentMultipleSteps

Using the Resource Arbiter Service without PathWave Test Automation as a client

Since the Resource Arbiter is a REST service, it is able to interact with every platform that is able to send and receive HTTP requests and responses. To integrate the Resource Management into your existing system make sure to implement a blocking **POST call** to the LockRequests resource that contains the identifiers of the instruments you want to lock. Alternatively, if it suits your application better or if your environment does not support this mechanism, you can send a POST request that specifies a 0 timeout. This will return a LockResponse object that only contains the token that was assigned to your request. You can then use this token to periodically poll the availability of the requested instruments, using this **GET call** . This will only return empty LockResponse objects until the resources have been successfully locked for the request identified by the token. Once you received a non empty LockResponse from the server, your application can start using the instruments. If you do not require the locked instruments to be one specific physical instance of that type, you can instead not use the identifier of the resource, but the names of the classes of the instruments that you want to use. If you want to make use of the connection concept of PathWave Test Automation, or even use the resource arbiter, you should specify the DUT identifier of the device that you want to test as well. If there is no unambiguous way to connect the DUT with the instruments, you must specify the pairs of ports that shall be connected with each other.

Please make sure to refresh the timeout of your resource usage, in cases where you need the resource longer than initially stated, by making this **PUT call**.

For a thorough introduction to the REST API, please refer to the dedicated part of the **REST API documentation** .

Additionally the Resource Arbiter package contains an example program that shows how the REST API can be consumed with a client application. This example will by default be installed to %TAP_PATH%\Packages\Resource Arbiter\Resource Arbiter Examples\Non-Tap Client and is introduced in the following section.

Sample API Client

The Sample API Client is an auto-generated C# client library generated using the NSwag tool chain.

Along with the client library, there is a sample Program.cs file that demonstrates the use of various methods from the C# client library. These methods correspond to the REST API calls.

In order to run this sample program, ensure that the ResourceArbiterExampleWithoutSwitchManager profile is the active profile. You can use the GET API call to determine the current profile.

The Sample API Client program performs the following steps:

1. Prompt the user to type the address of the machine on which the Resource Arbiter service is running. If the Resource Arbiter service is to be accessed locally (on the same machine as client), the user must simply press Enter.
2. Create a new client object.

```
client = new Client(address);
```

Lock and unlock instruments

Note: Getting the snapshot commands have been inserted here multiple times to verify the behavior of the lock and unlock calls. Otherwise, these are not mandatory.

1. Get the snapshot before making a request for locking an instrument.

```
var snapshot = client.Snapshot_Get();
```

This snapshot contains the lock state of all the instruments as well as the number of currently queued requests for instruments.

2. Create a LockRequest object and place a request for locking Three Port Instrument 1 as well as all connections between 3 Output Dut1 and Three Port Instrument 1. The response is captured using a LockResponse object.

```
ObservableCollection<SourceAndDestination> entries = new ObservableCollection<SourceAndDestination>();
entries.Add(new SourceAndDestination { DutIdentifier = "3 Output Dut1", InstrumentIdentifier = "Three Port Instrument 1" });
LockRequest lockRequest = new LockRequest { Entries = entries, MaxLockDurationSeconds = 10.0 };
LockResponse response = client.LockRequests_Post(lockRequest, 7);
```

3. Get the snapshot after making request for locking the instrument.

```
snapshot = client.Snapshot_Get();
```

The snapshot now shows that the Instrument “Three Port Instrument 1” is no longer free, but locked.

4. Wait until the lock time of the instrument has passed to demonstrate that the instrument will be unlocked after that time.
5. View snapshot after the instrument lock time expires.

```
snapshot = client.Snapshot_Get();
```

The snapshot shows that the instrument has been released after the lock time has elapsed.

View the utilization statistics

1. Place a request to view the utilization statistics of the instruments. The utilization statistics shows the total time elapsed and utilization time per instrument.

```
var utilization = client.Utilization_Get();
```

It will be observed that the usage of the instrument locked in the previous steps has been registered.

Use the refresh timeout feature

1. Place the same request as in a previous step but for a maximum lock duration of 3 seconds.

```
lockRequest.MaxLockDurationSeconds = 3.0;
```

```
response = client.LockRequests_Post(lockRequest, 7);
```

2. Just before the maximum lock duration expires, refresh the instrument timeout by 60 sec. Use the LockToken received from the response from Resource Arbiter to identify the instrument whose timeout period is to be refreshed.

```
client.LockRequests_ResetTimeout(response.LockToken, 60.0);
```

3. Wait for at least 5 seconds. But the initial lock on the instrument was for only 3 seconds (see step 1)
4. Get the snapshot to see if timeout refresh helped.

```
snapshot = client.Snapshot_Get();
```

The snapshot shows that the instrument is locked.

Explicitly unlock the instrument

1. Unlock the instrument. Otherwise, the instrument gets unlocked by itself after the timeout.

```
client.UnlockRequests_Post(response.LockToken);
```

2. Get the snapshot after the instrument is released.

```
snapshot = client.Snapshot_Get();
```


The snapshot shows that the instrument has been released.

Poll for availability of the instrument status

1. Try locking the same instrument twice.

First call:

```
lockRequest.MaxLockDurationSeconds = 15;  
response = client.LockRequests_Post(lockRequest, 7);
```

Second call:

```
lockRequest.MaxLockDurationSeconds = 2;  
var delayedResponse = client.LockRequests_Post(lockRequest, 5);
```

Note that the second parameter in the call represents the time to wait in the blocking call. After this time period has elapsed, the Resource Arbiter will send a response regardless of the availability of the instrument. The LockResponse that we received for the second call should only contain a LockToken and not much else.

2. Poll the state of the LockRequest.

```
while (delayedResponse.AssignedInstrumentIdentifiers == null)  
{  
    Console.WriteLine("Polled the state of the LockRequest");  
    delayedResponse = client.LockRequests_PollToken(delayedResponse.LockToken, 1);  
}  
  
Console.WriteLine("The polling succeeded in acquiring the LockRequest");
```

3. Unlock the instrument.

```
client.UnlockRequests_Post(delayedResponse.LockToken);
```

4. Get the snapshot after the instrument is released.

```
snapshot = client.Snapshot_Get();
```

Requesting for instruments that match a category

1. If you do not depend on a specific instance of an instrument type but any instrument of a given type will serve the purpose, you can specify that by not providing the name of the instrument in the LockRequest object, rather providing the name of the class of the instrument. In our example, there are two instruments of type Three Port Instrument present on the Test Bench, and this implements the interface IThreePortInst. Simply change the requested instrument identifier to the name of the class or interface that you want to request.

```
lockRequest.Entries[0].InstrumentIdentifier = "Keysight.Tap.Plugins.ResourceArbiter.Examples.IThreePortInst";  
lockRequest.MaxLockDurationSeconds = 5;  
Console.WriteLine("Requesting two Instruments that implement the interface IThreePortInst. We do not care which  
one we get with which request.");
```

```
var firstResponse = client.LockRequests_Post(lockRequest, 10);
```

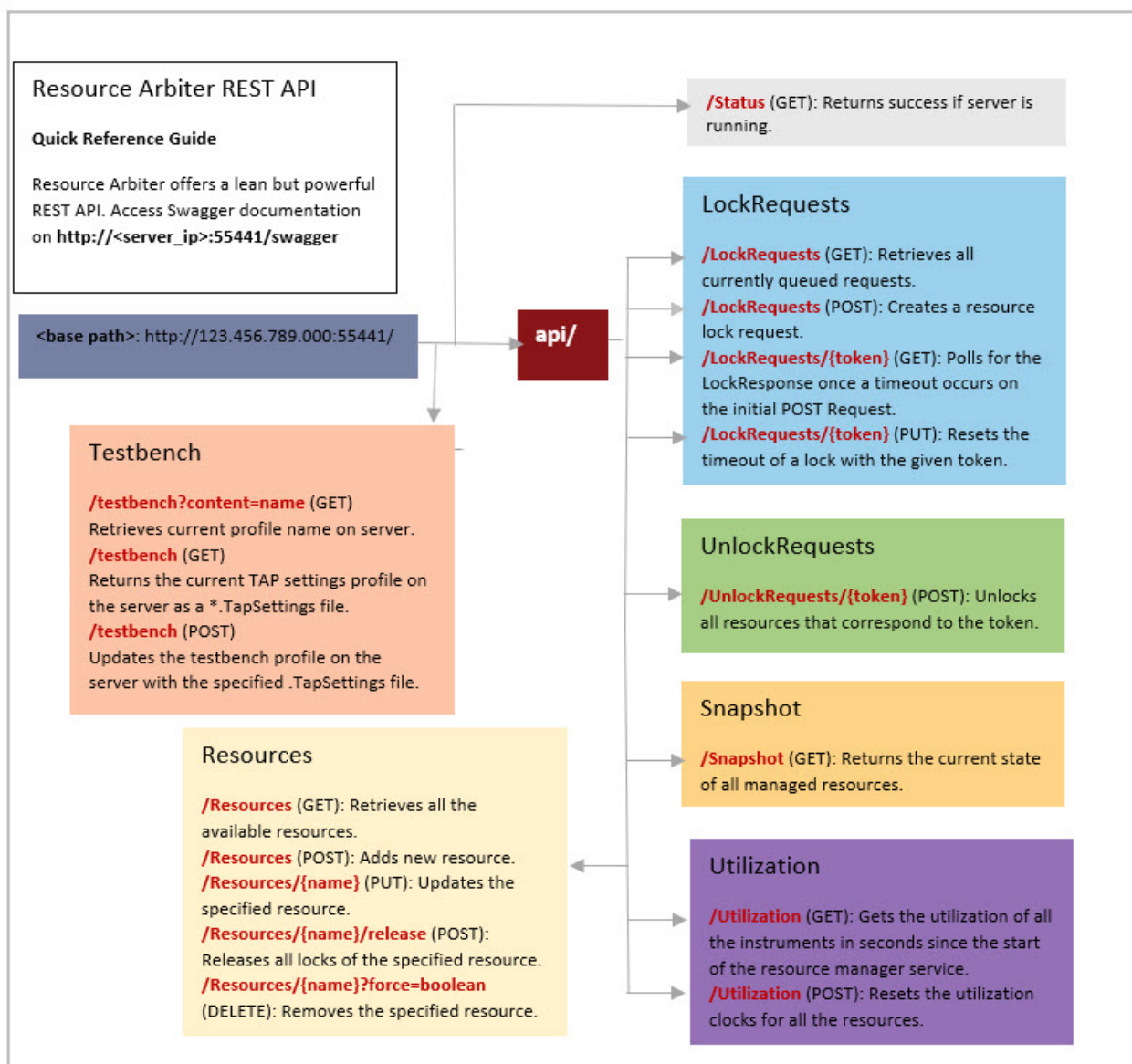
2. To avoid using the already locked connection between the DUT from the first call and the instrument from the first call, change the DutIdentifier and send another request to the IThreePortInst interface.

```
lockRequest.Entries[0].DutIdentifier = "3 Output Dut2";  
var secondResponse = client.LockRequests_Post(lockRequest, 10);
```

3. Get the snapshot.

```
snapshot = client.Snapshot_Get();
```

REST API Quick Reference



REST API Quick Reference

REST API Documentation

The Resource Arbiter offers a lean but powerful REST API.

Overview

Status

[GET] **/Status** : Returns success if the Resource Arbiter service is running.

Testbench

[GET] **/Testbench?content=name** : Returns the name of the current testbench profile on the server-side as a string.

[GET] **/Testbench** : Returns the current PathWave Test Automation settings profile on the server-side as a .TapSettings file.

[POST] **/Testbench** : Takes a .TapSettings file in multipart/formdata format and updates the testbench profile on the Resource Arbiter server based on the profile contained in the file.

LockRequests

[GET] /api/LockRequests : Returns all the LockRequests that are currently in the queue.

[POST] /api/LockRequests : Enqueues the LockRequest that is in the body of the HTTP message.

[GET] /api/LockRequests/{token} : This method is used to poll the state of an ongoing request.

[PUT] /api/LockRequests/{token} : Extends the timeout of the already granted request with the given token.

UnlockRequests

[POST] /api/UnlockRequest/{token} : Unlocks all the resources that are associated with the given token.

Resources

[GET] /api/Resources : Returns a list of all the available resources with their public properties.

[POST] /api/Resources : Adds a new resource to the setup. Please note that this is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

[DELETE] /api/Resources/{name}?force=boolean : Removes the resource with the given name from the setup.

[PUT] /api/Resources/{name} : Updates the specified resource. Please note that this is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

[POST] /api/Resources/{name}/release : Releases all the current locks of the specified resource.

Snapshot

[GET] /api/Snapshot : Returns the current state of all managed resources.

Utilization

[GET] /api/Utilization : Returns for each instrument, the time in seconds that it has been utilized since the start of the Resource Arbiter service as well as the time that expired since the service was started. This enables the calculation of a utilization ratio per instrument.

[POST] /api/Utilization : Resets the utilization clocks for all the resources.

Status (/Status)

[GET] Method:

Returns success if the Resource Arbiter service is running.

Parameters:

No parameters

Request Example:

http://127.0.0.1:55441/Status

Response Example:

Response Code	Description
200	Success

Testbench (/testbench?content=name)

[GET] Method:

Returns the name of the current testbench profile on the server-side as a string. **Tip:** You can use this to verify that your client has the same local settings profile as the server.

Parameters:

- content

Example Request:

http://127.0.0.1:55441/testbench?content=name

Example Response:

ResourceArbiterExampleWithoutSwitchManager

Response Code	Description
200	Success

Testbench (/Testbench)

[GET] Method:

Returns the current PathWave Test Automation settings profile on the server-side as a .TapSettings file. You can then import this file in your local PathWave Test Automation instance.

Request Example:

http://127.0.0.1:55441/Testbench

Response:

Response Code	Description
200	Success

Testbench (/testbench)

[POST] Method:

Takes a .TapSettings file in multipart/formdata format and updates the testbench profile on the Resource Arbiter server based on the profile contained in the file. This leads to a “soft update” of the DUT and instruments on the server, which means that the locking state of all the DUTs, Instruments and Connections sharing the same names in the previous settings profile and are still present (by Name) on the new settings profile is preserved. However, all instruments and DUTs, as well as connections that are not contained in the new settings profile will no longer be accessible via the Resource Arbiter server. This will lead to the failure of requests that demand for the no longer available resources. So, ensure that all the clients are aware of the change in settings. All DUTs, instruments and connections that are present on the new settings profile will be available on the Resource Arbiter server after the request is successfully processed. To sum up, all new resources will be free and all already existing resources (if any) will maintain their lock state.

LockRequests (/api/LockRequests)

[GET] Method:

Retrieves a list of all the currently queued lock requests. Requests under process are not returned. This call can be used for debugging purposes.

Parameters:

No parameters

Request example:

http://127.0.0.1:55441/api/LockRequests

Response example:

The list is formatted as a JSON list of LockRequest objects, as specified [here](#) .

An example is shown below.

```
[
  {
    "entries": [
      {
        "dutIdentifier": "OneOutputDut",
        "instrumentIdentifier": "Long Measuring Instrument",
        "dutPortName": "Output1",
        "instrumentPortName": "Input 1"
      }
    ],
    "maxLockDurationSeconds": 300,
    "token": "1b9b4316-0b85-4e09-8ba9-e5477cc4dcb0"
  }
]
```

Response Code	Description
200	Success. Returns a response in JSON format.
202	Success
400	Bad Request
404	Not Found

LockRequests (/api/LockRequests)

[POST] Method:

Creates a resource lock request. Upon posting a LockRequest object in the body, the client will receive a LockResponse object. These objects have to be formatted in JSON and look like follows:

LockRequest:

```
{
  "entries": [
    {
      "dutIdentifier": "string",
      "instrumentIdentifier": "string",
      "dutPortName": "string",
      "instrumentPortName": "string"
    }
  ],
  "maxLockDurationSeconds": double,
  "token": "string"
}
```

An example:

```
{
  "entries": [
    {
      "dutIdentifier": "OneOutputDut",
      "instrumentIdentifier": "Short Measuring Instrument",
      "dutPortName": "Output1",
      "instrumentPortName": "Input 1"
    }
  ]
}
```

```

    }
  ],
  "maxLockDurationSeconds": 300,
  "token": "firstRequest"
}

```

Semantics of the parameters:

parameter	meaning
entries	A list of 4-tuples that describes which resources are requested
dutIdentifier(optional)	The unique name of the DUT that is to be tested with the requested resources.
instrumentIdentifier	The unique name of the instrument that should be locked by this request. This may also be a unique name of a class of instruments.
dutPortName(optional)	The name of the Port that should be connected on the DUT side. (must be unique per DUT)
instrumentPortName(optional)	The name of the Port that should be connected on the instrument side. (must be unique per instrument)
MaxLockDurationSeconds(optional)	Specifies the maximum time duration for which the instruments will be required. After expiration of this time period, the instruments are automatically released. If no such duration is specified, the resources will be locked infinitely or until an UnlockRequest is received for these resources. The maximum value for this parameter is 922337203685.4775.

LockResponse:

```

{
  "lockToken": "string",
  "unlockUrl": "string",
  "maxLockDurationSeconds": double,
  "assignedInstrumentIdentifiers": [ "string" ],
  "assignedConnectionsAndSwitchedViaIdxs": [
    {
      "item1": "string",
      "item2": [ 0 ]
    }
  ]
  "Resources": [
    {
      "propertyName1" : "value1", "propertyName2" : "value2"
    }
  ]
}

```

An example:

```

{
  "lockToken": "d9b5fcf9-be00-447d-b494-c221fdbd9b13",
  "unlockUrl": "http://156.141.0.191:55441/api/UnlockRequests/d9b5fcf9-be00-447d-b494-c221fdbd9b13",
  "maxLockDurationSeconds": 300,
  "assignedInstrumentIdentifiers": [
    "Three Port Instrument 2",
    "Three Port Instrument 2",
    "Three Port Instrument 2"
  ],
  "assignedConnectionsAndSwitchedViaIdxs": [
    {
      "item1": "Clock_1_2",
      "item2": [
        1
      ]
    }
  ]
}

```

```

    },
    {
      "item1": "Trigger_1_2",
      "item2": [
        1
      ]
    },
    {
      "item1": "Data_1_2",
      "item2": [
        1
      ]
    }
  ]
  "Resources": [
    { "Name" : "Three Port Instrument 2", "Address" : "123.456.789" }
  ]
}

```

Semantics of the parameters:

parameter	meaning
lockToken	A unique string that identifies the enqueued request. This string will be used for the unlock operation as well as for polling and the refreshing of timeouts.
unlockUrl	The complete URL where an empty post message will lead to the release of the resources.
MaxLockDurationSeconds(optional)	Specifies the maximum time duration for which the instruments will be required. After expiration of this time period, the instruments are automatically released. If no such duration is specified, the resources will be locked infinitely or until an UnlockRequest is received for these resources. The maximum value for this parameter is 922337203685.4775.
assignedInstrumentIdentifiers	The list of names of the assigned instruments that were successfully locked.
assignedConnectionsAndSwitchedViaIdxs	The list of names of the connections and for each connection the activated VIAs that were locked, or in the presence of a Switch Instrument implementing ISwitch interface activated. This can be used in order to retrieve calibration data (i.e. S-parameter files) for the chosen connections.
Resources	A List that contains key-value pairs for all public properties of each instrument that was successfully locked. This is especially useful when TAP is not used as a client to consume the REST API. This way any client can access all the information that is present on the TAP test bench without replicating the Test Bench itself on the client machine.

Depending on which of the optional parameters were specified in the lock request, the request has different meanings:

1. Only instrumentIdentifiers are specified: In this case only the instruments will be locked for use.
2. instrumentIdentifiers and dutIdentifiers are specified: In this case the instruments will get locked and all connections between corresponding DUT and instrument will be switched. For this case, all connections have to be unambiguous. This means that for each port on the instrument side there is only one port on the DUT side, that can be connected and vice versa.
3. All four parameters are specified: In this case the instruments get locked and connections between all pairs of ports, that were specified are switched.

In addition to the described data structures that are transferred in the bodies of the HTTP messages, one can specify another parameter, which is provided as a query parameter. This parameter is called `timeout` and specifies the time, in seconds, for which the HTTP reply will be awaited. If this is not provided, the result is awaited infinitely. This ensures that the request really waits until the locks are granted. A shorter timeout might be useful for reacting earlier, for example when the user wants to stop test plan execution earlier. In this case, use a very short timeout here (e.g. 0). This guarantees a fast response giving the `lockToken`. With this `lockToken` you can then **Poll** to actually get a lock or **Cancel** to take back the lock request. The maximum value for this parameter is 922337203685.4775.

Due to the `timeout` parameter, the HTTP endpoint looks like the following:

```
/api/LockRequests?timeout=""
```

Example Request

```
http://127.0.0.1:55441/api/LockRequests?timeout=60
```

Example Response

```
{
  "lockToken": "d9b5fcf9-be00-447d-b494-c221fdbd9b13",
  "unlockUrl": "http://156.141.0.191:55441/api/UnlockRequests/d9b5fcf9-be00-447d-b494-c221fdbd9b13",
  "maxLockDurationSeconds": 300,
  "assignedInstrumentIdentifiers": [
    "Three Port Instrument 2",
    "Three Port Instrument 2",
    "Three Port Instrument 2"
  ],
  "assignedConnectionsAndSwitchedViaIdxs": [
    {
      "item1": "Clock_1_2",
      "item2": [
        1
      ]
    },
    {
      "item1": "Trigger_1_2",
      "item2": [
        1
      ]
    },
    {
      "item1": "Data_1_2",
      "item2": [
        1
      ]
    }
  ]
}
"Resources": [
  { "Name" : "Three Port Instrument 2", "Address" : "123.456.789" }
]
```

Response Code	Description
200	Success
201	Returns the newly created item
400	If the Body of the request does not contain a valid LockRequest object
404	If an Instrument or DUT of the request is not known
408	If the lock was not granted before the given timeout

LockRequests (/api/LockRequests/{token})

GET{token}

Retrieves the LockResponse for the LockRequest that was assigned the given token. As this token is only known after the response of the Resource Arbiter on the POST of said LockRequest was retrieved, this method should be used in polling scenarios only.

Parameters:

- token
- timeout

Example Request:

http://127.0.0.1:55441/api/LockRequests/ba866042-8927-4367-b222-8f7b698233d0?timeout=300

Example Response:

```
{
  "lockToken": "ba866042-8927-4367-b222-8f7b698233d0",
  "unlockUrl": null,
  "maxLockDurationSeconds": 10,
  "assignedInstrumentIdentifiers": null,
  "assignedConnectionsAndSwitchedViaIdxs": null
}
```

Response Code	Description
200	Success
404	Not Found
408	Request Timeout

LockRequests (/api/LockRequests{token}?timeout='doubleInSeconds')

[PUT] Method:

Refreshes the timeout for the request with the given token and sets it to the new timeout that is specified as a query parameter. This query parameter is of type double and represents the timespan in seconds.

This interplays with the overall procedure in the following way: In the POST method, one can specify the timespan for which the resources should be locked. Since an unlock request can be sent at any time, that will release the lock. It is a good practice to be rather generous with the timespan that is specified in the POST method. However, should something unforeseen happen, the PUT method enables you to keep the lock for resources longer than initially demanded.

Parameters

- token
- timeout

Example Request:

http://127.0.0.1:55441/api/LockRequests/e361a939-a40e-493c-bfdb-8099a4d47464?timeout=720

Example Response:

Response Code	Description
200	Success
400	Bad Request

Snapshot (/api/Snapshot)

[GET] Method:

Gathers the current locking state of all the managed instruments as well as the size of the queue of LockRequests. It returns the desired data in the body of the HTTP response in the following format(JSON):

```
{
  "lockedInstruments": [],
  "freeInstruments": [],
  "sizeOfQueue": 0
}
```

Semantics of the parameters:

parameter	meaning
lockedInstruments	A list of all the unique instrument identifiers of instruments that are currently locked.
freeInstruments	A list of all the unique instrument identifiers of instruments that are currently not locked and thus available for use.
sizeOfQueue	An integer parameter that represents the number of LockRequests that are currently in the queue.

Request Example:

http://127.0.0.1:55441/api/Snapshot

Response Example:

```
{
  "lockedInstruments": [
    "Short Measuring Instrument"
  ],
  "freeInstruments": [
    "Long Measuring Instrument",
    "Long Measuring Instrument1"
  ],
  "sizeOfQueue": 0
}
```

Response Code	Description
200	Success

Utilization (/api/Utilization)

[GET] Method:

Returns for each instrument, the time in seconds for which it has been utilized since the start of the Resource Arbiter service as well as the time that expired since the service was started. This enables the calculation of a utilization ratio per instrument.

Parameters

No parameters

Request Example:

http://127.0.0.1:55441/api/Utilization

Response Example:

The data is provided in the body of the HTTP response and is in the following JSON format:

```
{
  "totalElapsedTime": "double",
  "utilizationPerInstrument": {
    "instrumentIdentifier": "double"
  }
}
```

Semantics:

parameter	meaning
totalElapsedTime	Elapsed time in seconds since the Resource Arbiter service was started.
utilizationPerInstrument	A dictionary containing a double value that represents the time in seconds for each managed instrument that the instrument was locked (and thus hopefully used).

The following is an example of a response to this request.

```
{
  "totalElapsedTime": 947.2101341,
  "utilizationPerInstrument": {
    "Long Measuring Instrument": 0,
    "Long Measuring Instrument1": 0,
    "Short Measuring Instrument": 300.52365510000004
  }
}
```

Response Code	Description
200	Success

Utilization (/api/Utilization)

[POST] Method:

Resets the utilization clocks for all the resources.

Parameters

No parameters

Request Example:

http://127.0.0.1:55441/api/Utilization

Response Example:

Response Code	Description
200	Success

UnlockRequests (/api/UnlockRequests/{token})

[POST] Method:

Releases the lock for all the resources assigned with the LockResponse that contained the given token.

After sending this request to the Resource Arbiter, the client should not use the instruments any further, since the instruments might be assigned to the next user. To use the instruments again, the client must

place a fresh Lock request.

Parameters

No parameters

Request Example:

http://127.0.0.1:55441/api/UnlockRequests/58561477-aeab-48a9-8f53-3cac64a13942

Response Example:

Response Code	Description
200	Success. "Instruments unlocked"
404	Not Found

Resources (/api/Resources)

[GET] Method:

Returns a list of all the available resources with their public properties.

Parameters

No parameters

Request Example:

http://127.0.0.1:55441/api/Resources

Response Example:

The data is provided in the body of the HTTP response and is in the following JSON format:

```
[
  {
    "Address": "123.456.789",
    "Capabilities": "PCI",
    "IsInfinitelyLockable": false,
    "MaxLockCount": 2,
    "IsEnabled": true,
    "Name": "PCI2",
    "CurrentLockCount": 0,
    "Utilization": 0
  },
  {
    "Address": "http://somethinginteresting:2500",
```

```

    "Capabilities": "PCI",
    "IsInfinitelyLockable": false,
    "MaxLockCount": 3,
    "IsEnabled": true,
    "Name": "PCI3",
    "CurrentLockCount": 0,
    "Utilization": 0
  }
]

```

Response Code	Description
200	Success

Resources (/api/Resources)

[POST] Method:

Adds a new resource to the setup. Please note that this is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

The POST HTTP message sent to the server must contain a JSON body with this content:

```

{
  "Address": "string",
  "Capabilities": "string",
  "IsInfinitelyLockable": true,
  "MaxLockCount": 0,
  "IsEnabled": true,
  "Name": "string",
}

```

Semantics of the parameters:

parameter	meaning
Name	The name of the resource that gets added to the test bench. You must use unique resource names. This name must also not be the same as any capability name that is already present on the test bench.
Address	The VISA address, IP Address, or the Host name of the resource.
Capabilities	A string of comma-separated capabilities. Once resources are present on the Setup, you can lock them by using these capabilities strings.
IsInfinitelyLockable	If this is set to true, the resource can be locked by an arbitrary number of clients in parallel. If set to false, the resource will be available for "MaxLockCount" clients, in parallel.
MaxLockCount	The number of clients that should be able to lock this resource in parallel. This parameter will be taken into account only if IsInfinitelyLockable is set to false. A Count of 0 will lead to this resource never being locked. All negative counts will be trimmed to 0 automatically.
IsEnabled	If set to true, this resource can be locked by clients. If set to false, this resource cannot be locked by clients. This function is supposed to be used for maintenance or downtimes.

Response Example:

Response Code	Description
200	Success
400	Bad Request

Resources (/api/Resources/{name})

[PUT] Method:

Updates the specified resource. The expected body of this method is identical to the POST method. However, properties that should not change can be omitted from the body and will remain unchanged.

Please also note that this is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

Parameters

- name

Response Example:

Response Code	Description
200	Success
400	Bad Request
404	Error: Not Found

Resources (/api/Resources/{name}?force=boolean)

[DELETE] Method:

Removes the resource with the given name from the setup. This request will produce a response with HTTP status code 200, if the resource was successfully deleted and produce a 204 status code, if the resource was already deleted or not found in the setup. If the resource is currently locked, it will not be deleted to avoid interference with measurement execution. To override this behavior, an additional query parameter to force the deletion can be set. Be absolutely sure that you want to delete the resource immediately, regardless of its lock state, when using this force parameter.

Parameters

- name

Request Example:

http://127.0.0.1:55441/api/Resources/PCI_Worker1?force=true

Response Example:

Response Code	Description
200	Success. The resource was successfully deleted.
204	The resource has already been deleted or was not found in the setup.

Resources (/api/Resources/{name}/release

[POST] Method:

Releases all the current locks of the specified resource.

CAVEAT: This call may severely impact measurement execution, as another client will then be able to lock the resource and might interfere with the process that held the lock before. This method should only be used when you are certain that the client holding the lock has crashed.

Example Request

http://127.0.0.1:55441/api/Resources/PCI_Worker1/release

Response Code	Description
200	Success
404	Error: Not Found

Visualizing Resource Utilization with the PathWave Test Automation Timing Analyzer

In addition to using the **REST call** to retrieve information about the resource utilization, one can also use the PathWave Test Automation Timing Analyzer tool. It can be found in the PathWave Test Automation GUI via *Tools > Timing Analyzer*.

Once the tool has started you can select the log file that shall be used to analyze the timing behavior via *File > Open Log File(s)*. You can find the logs of the Resource Arbiter in the directory:

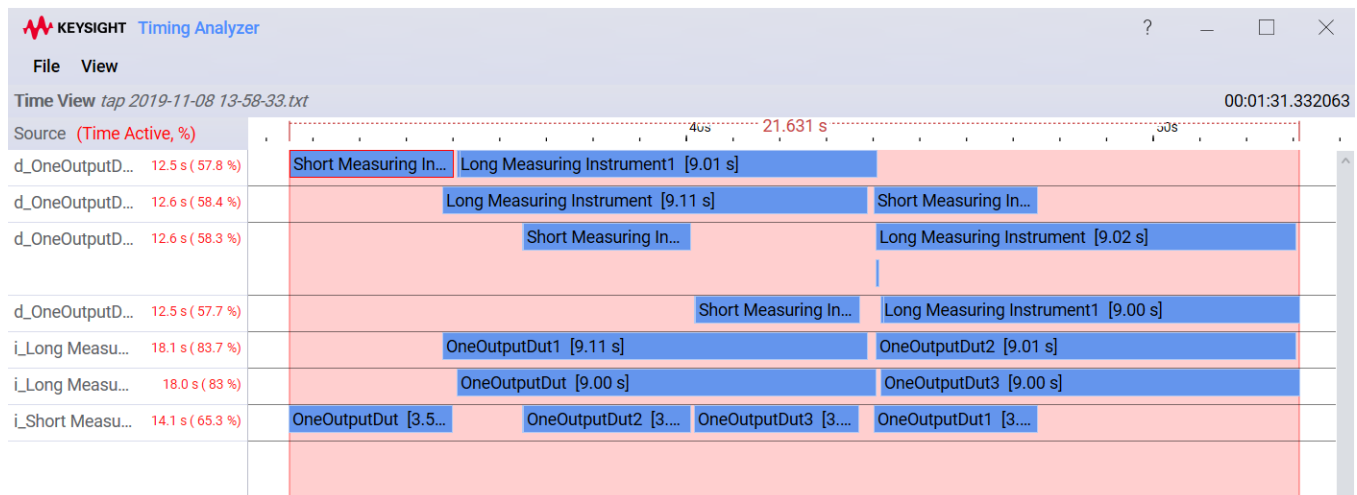
%TAP_PATH%\SessionLogs\tap.

When you load a log file into the Timing Analyzer, you can select which sources of logs you want to be displayed by clicking *View > Sources*. By default, every possible source of logs will be selected. The Resource Arbiter creates one log source for every DUT and non switch instrument on the testbench. Whenever an instrument is logged you will see a blue bar that indicates the duration of the lock in the swim lane that corresponds to the respective instrument.

Additionally, you will also see one such blue bar in the swim lane of the DUT that was able to acquire the lock for the instruments. The text in the blue bars indicates the instrument that was assigned to the DUT and vice versa.

In the example of one such Timing Analyzer view below you can see the bars that indicate the resource usage. Please note that you can also see the utilization for a region indicated as percent directly behind the name of each resource by dragging the mouse over the interested region.

If you only want to track instrument usage, you can of course simply disable all logs, except those that correspond to instruments.



Example Screenshot of the Timing Analyzer

Note that you can only access the logs on the machine that runs the Resource Arbiter server.

Using Resource Arbiter GUI

The Resource Arbiter GUI enables you to add and configure your resources (including acquisition engine or oscilloscope and processing and results collection PCs) for use in measurement disaggregation. You can define your resource by specifying the following:

- Name
- Domain Name or IP address or VISA address
- Capabilities (strings that define the application names and whether the resources will be used for waveform acquisition, measurement processing, or results collection)
- The number of times a resource can be locked. You can also specify unlimited locking capability for a resource.

Installing and Accessing the Resource Arbiter GUI

You can perform the following functions from the Resource Arbiter GUI:

- [Adding Resources](#)
- [Updating Resources](#)
- [Deleting Resources](#)
- [Viewing Resource Utilization](#)
- [Disabling Resources](#)
- [Unlocking Resources](#)
- [Resetting Utilization Counts](#)

Installing and Accessing the Resource Arbiter GUI

The Resource Arbiter GUI can be installed by running the Resource Arbiter server installer (a .exe file) available on keysight.com. There is no prerequisite for installing the PathWave Test Automation, in this case.

When installing the Resource Arbiter GUI from the .exe file, it is strongly recommended to install the application on a machine with no prior PathWave Test Automation installation. Existing PathWave Test Automation users intending to make use of the Resource Arbiter web GUI are strongly recommended to install Resource Arbiter server version 1.1 from the .TapPackages file and follow the steps for accessing the web GUI.

Before you can access the Resource Arbiter GUI, install the KS8108A license using Keysight License Manager, on the machine that has the Resource Arbiter installed.

Accessing the Resource Arbiter GUI

You can access the Resource Arbiter GUI by performing the following steps:

1. Start the Resource Arbiter server using ANY of the following techniques:
 - Use the `ResourceArbiter.bat` shortcut in the Start Menu: Start > Keysight Test Automation > Keysight Resource Arbiter
 - Start the `ResourceArbiter.bat` batch file from your Keysight Test Automation installation folder. The default location for this batch file is `C:/Program Files/Keysight/Test Automation`
 - Start the Resource Arbiter service by entering the following commands on a command line:

```
cd %TAP_PATH%
tap resourcearbiter
```

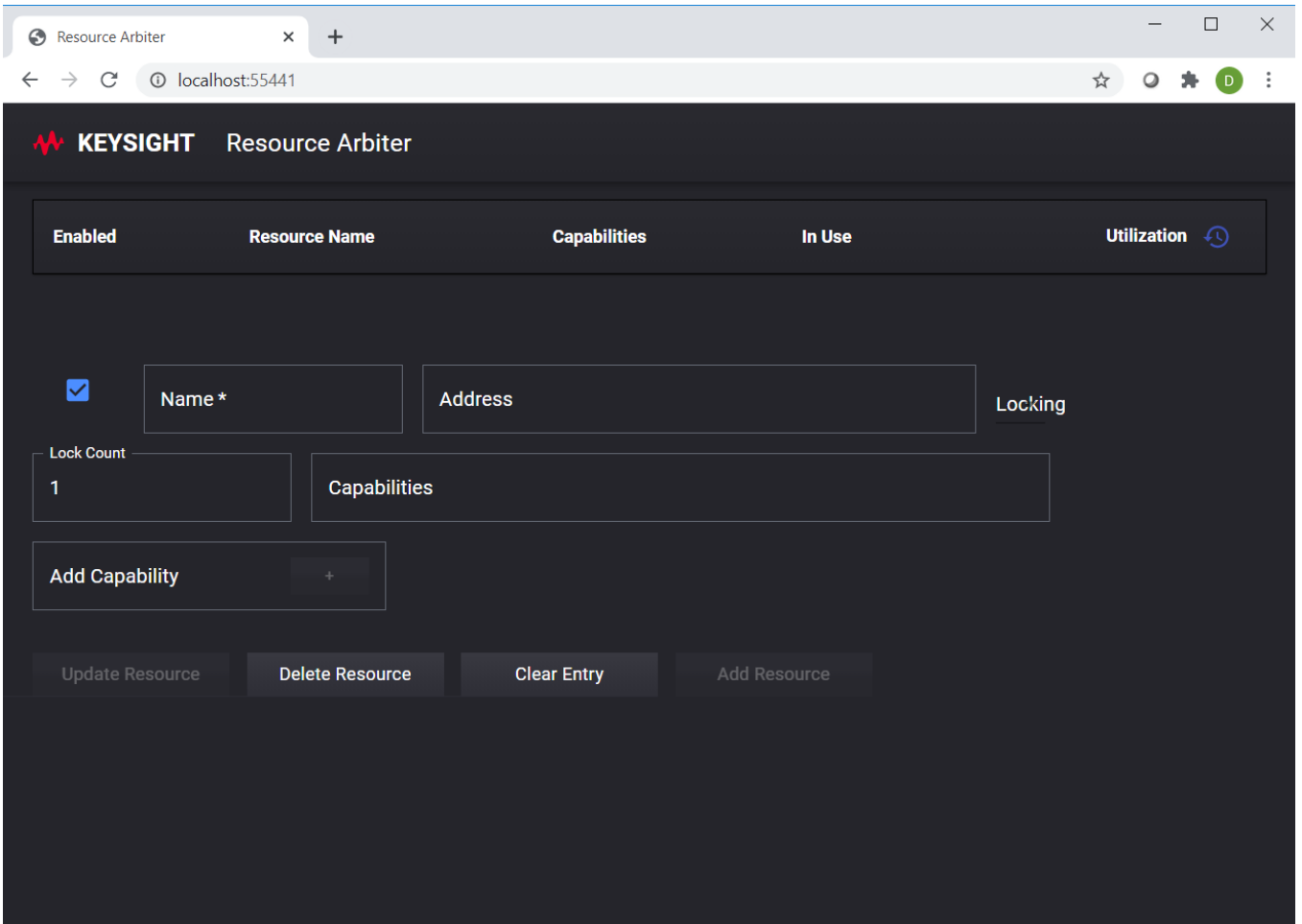
2. Access the following url from your web browser:

`http://<serverIP_or_hostname>:55441/`

Note: The Internet Explorer web browser is not recommended for accessing the Resource Arbiter GUI.

If you are accessing the Resource Arbiter GUI from the same machine on which you started the Resource Arbiter service, access the GUI from the following url: `http://localhost:55441/`

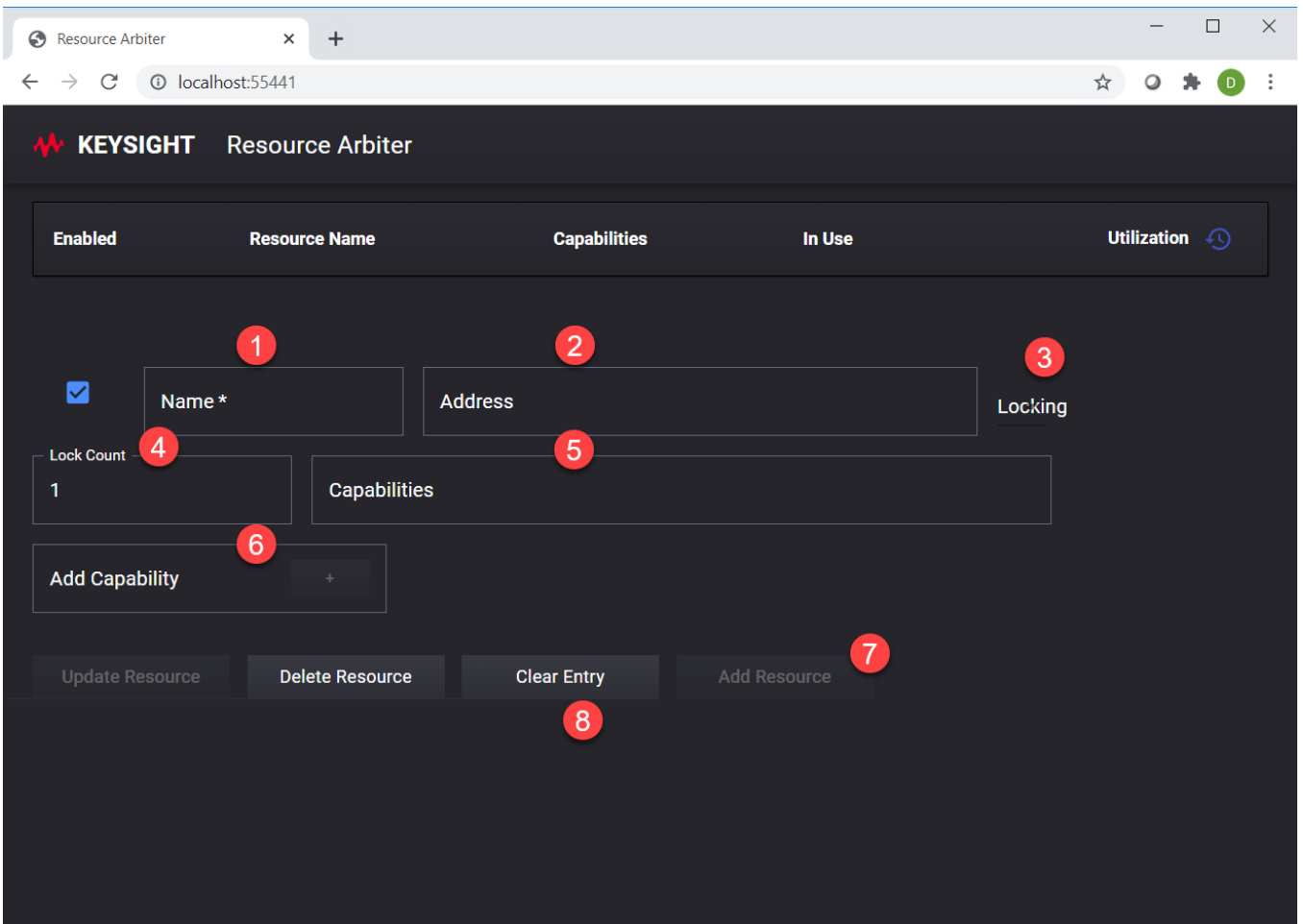
The Resource Arbiter GUI is displayed:



Resource Arbiter GUI

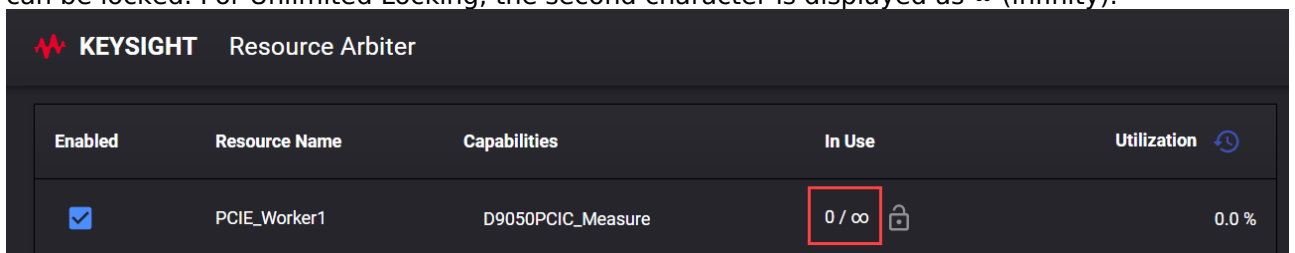
Adding Resources

Perform the following steps to add resources to your system using the Resource Arbiter GUI.



Adding Resources

1. Click inside the **Name** field and specify a name for the resource. Note that the name must be unique amongst all the resources and must not be identical to any capabilities that exist on the setup. Furthermore, the Name field must not be empty.
2. Click inside the **Address** field and enter the domain name or IP address or VISA address of the resource.
3. Select **Unlimited** from the **Locking** drop-down list if the resource can be locked for an unlimited number of times. You can unlock a locked resource anytime subject to some conditions.
4. If you do not select Unlimited from the **Locking** drop-down list, in the **Lock Count** field, specify the maximum number of times that the resource can be locked. If you specify a number smaller than 1, the Resource Arbiter will never assign this resource to an incoming request.
5. Click inside the **Capabilities** field to display the available capabilities of the resource. These include strings for acquisition, processing, and report collection capabilities.
6. To add a new capability, click inside the **Add Capability** field and enter a name for the capability. Click the **+** button. Note that a new capability cannot be an empty string and must not be identical to an already existing resource name. In case of invalid inputs, the **+** button will be disabled.
7. Click **Add Resource**. The newly added resource appears in the list above. The **Add Resource** button is disabled in case of any invalid input. Note that the **In Use** field displays two characters separated by a "/". The first character shows the number of times that the specified resource is currently locked and the second character shows the maximum number of times that the resource can be locked. For Unlimited Locking, the second character is displayed as ∞ (infinity).



8. Click **Clear Entry** to clear the editable fields below the list of resources.

Please note that this procedure is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

Updating Resources

1. Click the row of the resource that needs to be updated. The values for the resource attributes are displayed under the list of resources.
2. Make appropriate changes in the editable fields.
3. Click **Update Resource**.



Updating Resources

Please note that this procedure is currently working with resources of type ResourceNode only, as used by disaggregated compliance applications.

Deleting Resources

1. Click the row of the resource that needs to be deleted. The values for the resource properties are displayed under the list of resources.
2. Click **Delete Resource**. You are prompted to confirm whether you want to permanently delete the resource.
3. Click **Yes** to confirm.



Deleting Resources

Viewing Resource Utilization



The Resource Arbiter GUI displays the utilization percentage of the resource. The utilization is calculated as a ratio of the time the instrument was locked and the elapsed time since the start of the Resource Arbiter service. If a resource can be locked more than once at the same time, this percentage will show the utilization in contrast to the full utilization of all lock counts. This means that if a resource that is configured to be lockable by two clients in parallel is constantly locked once, this percentage will show as 50%. For infinitely lockable resources, this percentage will be calculated as if the resource was lockable only once at a time. The Utilization percentage of the resource is displayed in the Utilization (last column) of the list of instruments.

Disabling Resources

Sometimes, you might not need a resource. Rather than deleting it from the list of resources and adding it again when you require it, you can temporarily disable it. Note that disabling will keep all current locks intact, but prevent any other clients from acquiring new locks for this resource.

- Uncheck the **Enabled** check box on the left of the resource name that needs to be disabled.

Disabling resources is an efficient way for performing maintenance on a system. You might disable a resource, wait until all locks are gone, shut it down, repair it, bring it up again, and then enable it by selecting the **Enabled** check box.

Enabled	Resource Name	Capabilities	In Use	Utilization 
<input checked="" type="checkbox"/>	PCIE_Worker1	D9050PCIC_Measure	0 / ∞ 	0.0 %



Disabling Resources

Unlocking Resources

In situations like a machine crash, you might want to forcefully unlock a resource irrespective of its lock count or current locking state.

- Click the Lock icon in the In Use field corresponding to the resource that must to be unlocked and confirm that you really want to unlock this resource in the ensuing prompt.



CAVEAT: This forceful unlocking might severely interfere with measurement execution. If the process that acquired the lock before was not indeed faulty, the measurement or computation of this process will most likely be interfered with by the next client that acquires the lock. Moreover, the execution of the next client's task might as well suffer from interference from the previous process that held the lock if it had not crashed but continues execution.

Enabled	Resource Name	Capabilities	In Use	Utilization 
<input checked="" type="checkbox"/>	PCIE_Worker1	D9050PCIC_Measure	0 / ∞ 	0.0 %

Unlocking Resources

Resetting Utilization Counts

To reset the utilization data, click the **Reset** icon next to the column name.

Enabled	Resource Name	Capabilities	In Use	Utilization 
<input checked="" type="checkbox"/>	PCIE_Worker1	D9050PCIC_Measure	0 / ∞ 	0.0 %

Resetting Utilization Counts