Writing Test Code

Complete the following steps to use Semiconductor Test Library in your test code:

1. Create a new TSMSessionManager object and any other local variables that are necessary for your test.
2. Use the TSMSessionManager object to query the session for the target pins.
3. Configure the instrumentation connected to the target pins and configure the relay modules.
4. Source and/or measure the signals.
5. Burst the patterns required to configure the DUT.
6. Calculate and/or publish the required test results.
7. Repeat step 4 and 6 as necessary for your test.
8. Clean up and restore the instrument state(s) after finishing the test.

Example C# Code Snippet of Workflow

```
public static void WorkFlowExample(
    ISemiconductorModuleContext semiconductorModuleContext,
    string[] sumPinNames,
    string[] digitalPinNames,
    string relayConfigBeforeTest,
    string relayConfigAfterTest,
    string patternName)
{
    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    double voltageLevel = 3.3;
    double currentLimit = 0.01;
    double settlingTime = 0.001;
    var measureSettings = new DCPowerMeasureSettings()
    {
        ApertureTime = 0.001,
        Sense = DCPowerMeasurementSense.Remote
    };

    var smuPins = sessionManager.DCPower(sumPinNames);
    var digitalPins = sessionManager.Digital(digitalPinNames);

    smuPins.ConfigureMeasureSettings(measureSettings);
    semiconductorModuleContext.ApplyRelayConfiguration(relayConfigBeforeTest,
waitSeconds: settlingTime);

    smuPins.ForceVoltage(voltageLevel, currentLimit);
    PreciseWait(timeInSeconds: settlingTime);
    var currentBefore = smuPins.MeasureAndPublishCurrent(publishedDataId: "CurrentBefore");
```

```
    digitalPins.BurstPatternAndPublishResults(patternName, publishedDataId:
"PatternResults");
    PreciseWait(timeInSeconds: settlingTime);
    var currentAfter = smuPins.MeasureAndPublishCurrent(publishedDataId: "CurrentAfter");

    var currentDifference = currentBefore.Subtract(currentAfter).Abs();
    semiconductorModuleContext.PublishResults(currentDifference,
publishedDataId: "CurrentDifference");

    smuPins.ForceVoltage(voltageLevel: 0, currentLimit: 0.001);
    PreciseWait(timeInSeconds: settlingTime);
    semiconductorModuleContext.ApplyRelayConfiguration(relayConfigAfterTest,
waitSeconds: settlingTime);
}
```

Semiconductor Test Library

Welcome to the documentation site for the Semiconductor Test Library!

The Semiconductor Test Library simplifies programming on the NI Semiconductor Test System (STS) and enables users to develop test programs efficiently using C#/.NET.

The Semiconductor Test Library includes the following high-level features:

- Interfaces and classes—Abstract instrument sessions and encapsulate the necessary pin and site awareness.
- Pin- and site-aware data types— Simplify instrument configuration and measurement results processing.
- Extension methods—Abstract common, high-level instrument operations.
- Parallelization methods—Abstract parallel for loops required to iterate over multiple instrument sessions regardless of how sessions map to pins or sites.
- Publishing methods—Simplify results publishing and add support for the SiteData and PinSiteData types.
- Utilities methods—Provide utility methods commonly required for writing test code.
- TestStand step types—Perform common operations, such as setting up and closing instruments, powering up a DUT, or executing common tests.

Software Requirements

You must have the following software to use the Semiconductor Test Library:

- STS Software 24.5 or later
- .NET Framework 4.8 or later

Visual Studio 2022 is highly recommended.

Adding Using Directives

Depending on how your Visual Studio IDE is configured, the required using directives for accessing the Semiconductor Test Library may or may not automatically populate for as you write code. Therefore, you should always ensure that you have added the appropriate namespaces as using directives to the top of your code. This is required for the code to compile and for certain IDE features to work properly, such as Visual Studio's IntelliSense.

Example 1: When working with the NI DCPower instruments, make sure you are using the following using directives in your code:

```
using NationalInstruments.SemiconductorTestLibrary.InstrumentAbstraction;
using NationalInstruments.SemiconductorTestLibrary.InstrumentAbstraction.DCPower;
```

Example 2: When working with the `PinSiteData` or `SiteData` objects, make sure you are using the following using directives in your code:

```
using NationalInstruments.SemiconductorTestLibrary.DataAbstraction;
```

> **TIP**
>
> Consider a static using directive when working methods in the `Utilities` class, such as `PreciseWait`. This will allow you to write `PreciseWait()` in your code, instead of `Utilities.PreciseWait()`.
>
> ```
> using static NationalInstruments.SemiconductorTestLibrary.Common.Utilities;
> ```

Related information:

- Microsoft Learn: Using Directive

Instrument Setup and Cleanup

The Semiconductor Test Library expects instrument sessions to be created for and stored via a separate session manager. The library currently leverages the TestStand Semiconductor Module to act its session manager, where the initialization and cleanup of instrument sessions is intended to be invoked from TestStand's ProcessSetup and ProcessCleanup sequences, respectively.

The Semiconductor Test Library provides instrument type specific initialization and cleanup code in the `Initialization` class.

> **NOTE**
>
> Class: `Initialization`
> Namespace: `NationalInstruments.SemiconductorTestLibrary.InstrumentAbstraction.<instrument type`
> Assembly: `NationalInstruments.SemiconductorTestLibrary.Abstractions.dll`

Additionally, the TestStandSteps provides the high-level,

> **NOTE**
>
> Namespace: `NationalInstruments.SemiconductorTestLibrary.TestStandSteps`
> Assembly: `NationalInstruments.SemiconductorTestLibrary.TestStandSteps.dll`

Using the Map Method

As of STS Software 24.5, the NationalInstruments.SemiconductorTestLibrary.TestStandSteps assembly provides method that are available as step types for you to drag-and-drop from the TestStand Insertion Pallet. These provided methods aid quick commonly suers...

The source code for these steps are available on github for reference.

It is expect that users can leverage

It also uses abstractions, they may not be the same memory space. This is not a problem except for init code

nuget inted

Use the `MapInitializedInstrumentSessions` method from the `Initialization` class to ensure the cached instrument session information is consistent between the

NationalInstruments.SemiconductorTestLibrary.TestStandSteps assembly and client assembly.

When using the Steps provided by the the NationalInstruments.SemiconductorTestLibrary.TestStandSteps assembly in your test sequence, and writing although it uses the NationalInstruments.SemiconductorTestLibrary.Abstractions assembly is dedent on the the NationalInstruments.SemiconductorTestLibrary.Abstractions. they may not share the same memory space if the client assembly is using a newer version of the NationalInstruments.SemiconductorTestLibrary.Abstractions.

If the client assembly is using a newer version of the NationalInstruments.SemiconductorTestLibrary.Abstractions, then will not share the same memory space which can cause problems.

NI DCPower Session Groups

The instrument abstraction for the NI DCPower drive expects that the sessions are configured in one or more groups within the loaded pin map. If the load pin map does not use session groups, the `InitializeAndClose.Initialize` method will throw and exception.

Applicable namespace: `SemiconductorTestLibrary.InstrumentAbstraction.DCPower`

Related information:

- LINK

Writing Test Code

Complete the following steps to use Semiconductor Test Library in your test code:

1. Create a new TSMSessionManager object and any other local variables that are necessary for your test.
2. Use the TSMSessionManager object to query the session for the target pins.
3. Configure the instrumentation connected to the target pins and configure the relay modules.
4. Source and/or measure the signals.
5. Burst the patterns required to configure the DUT.
6. Calculate and/or publish the required test results.
7. Repeat step 4 and 6 as necessary for your test.
8. Clean up and restore the instrument state(s) after finishing the test.

Example C# Code Snippet of Workflow

```
public static void WorkFlowExample(
    ISemiconductorModuleContext semiconductorModuleContext,
    string[] sumPinNames,
    string[] digitalPinNames,
    string relayConfigBeforeTest,
    string relayConfigAfterTest,
    string patternName)
{
    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    double voltageLevel = 3.3;
    double currentLimit = 0.01;
    double settlingTime = 0.001;
    var measureSettings = new DCPowerMeasureSettings()
    {
        ApertureTime = 0.001,
        Sense = DCPowerMeasurementSense.Remote
    };

    var smuPins = sessionManager.DCPower(sumPinNames);
    var digitalPins = sessionManager.Digital(digitalPinNames);

    smuPins.ConfigureMeasureSettings(measureSettings);
    semiconductorModuleContext.ApplyRelayConfiguration(relayConfigBeforeTest,
waitSeconds: settlingTime);

    smuPins.ForceVoltage(voltageLevel, currentLimit);
    PreciseWait(timeInSeconds: settlingTime);
    var currentBefore = smuPins.MeasureAndPublishCurrent(publishedDataId: "CurrentBefore");
```

```
    digitalPins.BurstPatternAndPublishResults(patternName, publishedDataId:
"PatternResults");
    PreciseWait(timeInSeconds: settlingTime);
    var currentAfter = smuPins.MeasureAndPublishCurrent(publishedDataId: "CurrentAfter");

    var currentDifference = currentBefore.Subtract(currentAfter).Abs();
    semiconductorModuleContext.PublishResults(currentDifference,
publishedDataId: "CurrentDifference");

    smuPins.ForceVoltage(voltageLevel: 0, currentLimit: 0.001);
    PreciseWait(timeInSeconds: settlingTime);
    semiconductorModuleContext.ApplyRelayConfiguration(relayConfigAfterTest,
waitSeconds: settlingTime);
}
```

Configuring Instrument Sessions

The Semiconductor Test Library makes an attempt to configure instruments in the most efficient way possible by consolidating the most common drive properties into one a single class. An instance of this class can be created and configured with only the properties intended to be updated, and then operate on that object to update the driver within in one go. This minimizes test time as it minimizes the use of parallel for loops that get called under-the-hood.

How it works

It always aborts the session. Does not re-initiate or commit the sessions (this is expected to happen in proceeding code).'

Sharing SiteData and PinSiteData Between Code Modules

During test development, it may become necessary to store some results measured in one code module with another code module later in the test sequence execution, within the same run. This can be achieved by storing the data within in the SemiconductorModuleContext in one code module and retrieving later in another code module using an ID string. For more information, refer the Sharing Data between Code Modules (TSM) topic in the TSM help documentation.

The `SetSiteData` and `GetSiteData` .NET methods provided by TSM, do not currently support being passed SiteData or PinSiteData objects directly. Therefore, SiteData/PinSiteData must first be converted into a 1D array of per-site values. Where, each element in the array represents a given site values. In the case of PinSiteData, this will be each element in the array a dictionary of per-pin values that represents a given site.

> **NOTE**
>
> The data must be ordered to match the order of sites in the Semiconductor Module context. This order might not be sequential. Use the `SiteNumbers` property on the `ISemiconductorModuleContext` .NET object to determine the order of the sites in the Semiconductor Module context and arrange the data manually.

Sharing SiteData Example

The following example shows how to store a per-site measurement data for comparison in a later test step:

```
public static void FirstCodeModule(
    ISemiconductorModuleContext semiconductorModuleContext,
    string pinName,
    string patternName,
    string waveformName,
    int samplesToRead)
{
    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    var digitalPins = sessionManager.Digital(pinName);
    digitalPins.BurstPattern(patternName);
    SiteData<uint[]> measurement = digitalPins.FetchCaptureWaveform(waveformName,
samplesToRead);

    var perSiteDataArray = new uint[semiconductorModuleContext.SiteNumbers.Count][];
```

```
    for (int i = 0; i < perSiteDataArray.Length; i++)
    {
        perSiteDataArray[i] =
measurement.GetValue(semiconductorModuleContext.SiteNumbers.ElementAt(i));
    }
    semiconductorModuleContext.SetSiteData("ComparisonData", perSiteDataArray);
}

public static void SecondCodeModule(
    ISemiconductorModuleContext semiconductorModuleContext,
    string pinName,
    string patternName,
    string waveformName,
    int samplesToRead)
{
    var perSiteComparisonDataArray = semiconductorModuleContext.GetSiteData<uint[]>
("ComparisonData");
    var comparisonData = new SiteData<uint[]>(perSiteComparisonDataArray);

    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    var digitalPins = sessionManager.Digital(pinName);
    digitalPins.BurstPattern(patternName);
    SiteData<uint[]> measurement = digitalPins.FetchCaptureWaveform(waveformName, 1);

    var comparisonResults = measurement.Compare(ComparisonType.EqualTo, comparisonData);
    semiconductorModuleContext.PublishResults(comparisonResults, "ComparisonResults");
}
```

Sharing PinSiteData Example

The following example shows how to store a per-pin per-site measurement data for comparison in a later test step:

```
public static void FirstCodeModule(ISemiconductorModuleContext semiconductorModuleContext,
string pinName)
{
    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    var dcPowerPin = sessionManager.DCPower(pinName);
    PinSiteData<double> measurement = dcPowerPin.MeasureVoltage();

    var perSiteDataArray = new IDictionary<string, double>
[semiconductorModuleContext.SiteNumbers.Count];
    for (int i = 0; i < perSiteDataArray.Length; i++)
    {
        perSiteDataArray[i] =
```

```
measurement.ExtractSite(semiconductorModuleContext.SiteNumbers.ElementAt(i));
    }
    semiconductorModuleContext.SetSiteData("ComparisonData", perSiteDataArray);
}

public static void SecondCodeModule(ISemiconductorModuleContext semiconductorModuleContext,
string pinName)
{
    var perSitePinDict = semiconductorModuleContext.GetSiteData<IDictionary<string,
double>>("ComparisonData");
    var pinSiteDictionary = new Dictionary<string, IDictionary<int, double>>();
    for (int i = 0; i < semiconductorModuleContext.SiteNumbers.Count; i++)
    {
        var siteNumber = semiconductorModuleContext.SiteNumbers.ElementAt(i);
        foreach (var pin in perSitePinDict[i].Keys)
        {
            var singlePinSiteValue = perSitePinDict[i][pin];
            if (!pinSiteDictionary.TryGetValue(pin, out IDictionary<int,
double> perSitePinValues))
            {
                perSitePinValues = new Dictionary<int, double>() { [siteNumber] =
singlePinSiteValue };
                pinSiteDictionary.Add(pin, perSitePinValues);
                continue;
            }
            if (!perSitePinValues.ContainsKey(siteNumber))
            {
                perSitePinValues.Add(siteNumber, singlePinSiteValue);
                continue;
            }
            perSitePinValues[siteNumber] = singlePinSiteValue;
        }
    }
    var comparisonData = new PinSiteData<double>(pinSiteDictionary);

    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    var dcPowerPin = sessionManager.DCPower(pinName);
    PinSiteData<double> measurement = dcPowerPin.MeasureVoltage();

    var comparisonResults = measurement.Subtract(comparisonData);
    semiconductorModuleContext.PublishResults(comparisonResults, "ComparisonResults");
}
```

Publishing Results

The Semiconductor Test Library does not directly process or evaluate test results. It instead passes published test results to be separate results handler. The library currently leverages the TestStand Semiconductor Module (TSM) to act its results handler, where the test results are evaluated against limits at the TestStand level. Refer to the TSM documentation linked below for more information.

Related information:

- NI TestStand Semiconductor Module: Publishing Results (TSM)

Concurrent Code Execution

Code statements that are independent of each other can be written to execute in concurrently. You can use the `InvokeInParallel` method from the `Utilities` class to allow separate lines of code to execute concurrently. For example, to perform operations on pins of different instrument types at the same time.

> **NOTE**
>
> This method uses the `Parallel.Invoke` method to execute multiple methods in parallel, and can be invoked in the exact same way. However, note that it also wraps the Parallel.Invoke method in a try-catch statement such that if an expectation occurs, only the first exception that is encountered will be returned to the call. This allows the exception to bubble-up and display properly by the TestStand runtime error dialog.
>
> Class: `Utilities`
> Namespace: `NationalInstruments.SemiconductorTestLibrary.Common`
> Assembly: `NationalInstruments.SemiconductorTestLibrary.Abstractions.dll`

The following example demonstrates how to use the `InvokeInParallel` method from the `Utilities` class:

```
public static void ConcurrentCodeExample(ISemiconductorModuleContext
semiconductorModuleContext, string pinNames)
{
    var sessionManager = new TSMSessionManager(semiconductorModuleContext);
    var publishDataID = "Measurement";
    var filteredPinNamesDmm =
semiconductorModuleContext.FilterPinsByInstrumentType(pinNames,
InstrumentTypeIdConstants.NIDmm);
    var filteredPinNamesPpmu =
semiconductorModuleContext.FilterPinsByInstrumentType(pinNames,
InstrumentTypeIdConstants.NIDigitalPattern);
    var filteredPinNamesSmu =
semiconductorModuleContext.FilterPinsByInstrumentType(pinNames,
InstrumentTypeIdConstants.NIDCPower);
    var ppmuPins = sessionManager.DCPower(filteredPinNamesSmu);
    var smuPins = sessionManager.Digital(filteredPinNamesPpmu);
    var dmmPins = sessionManager.DMM(filteredPinNamesPpmu);

    // Assumes that the instrumentation is already configured.
    Utilities.InvokeInParallel(
        () => ppmuPins.MeasureAndPublishCurrent(publishDataID),
        () => smuPins.MeasureAndPublishCurrent(publishDataID),
        () =>
```

```
        {
            var measurements = dmmPins.Read(maximumTimeInMilliseconds: 2000);
            semiconductorModuleContext.PublishResults(measurements, publishDataID);
        });
}
```

Parallelization Methods

The Semiconductor Test Library provides extension methods for abstracting parallel for loops that are required to iterate of each of the various instrument sessions within a given pin sessions bundle.

- Note that the Parallelization methods can be used with any class that inherits from the `ISessionsBundle` interface, such as the `DCPowerSessionsBundle` class.
- These methods can be used to perform one or more driver operation(s) across each instrument session associated with the pin(s) queried.
- There are overloads to allow you to specify if an operation is to be performed across each session or across each pin and site.
- These methods should only be used when needing to write low-level driver calls to implement instrument capabilities not yet exposed by the Semiconductor Test Library.

> **NOTE**
>
> Class: `ParallelExecution`
> Namespace: `NationalInstruments.SemiconductorTestLibrary.Common`
> Assembly: `NationalInstruments.SemiconductorTestLibrary.Abstractions.dll`

Related concepts:

- Making Low-level Driver Calls

Making Low-level Driver Calls

During test program development, you may be a low-level driver function or capability that is not directly exposed as a high-level extension method by the Semiconductor Test Library. In such cases, it may be necessary to use the low-level driver API to implement the desired test code. The `ParallelExecution` methods provided by the Semiconductor Test Library can be leveraged to directly access to the low-level driver sessions enabling you to implement ad-hoc driver calls inline with other high-level methods in your test code.

How-to Make Low-level Driver API Calls

To invoke a low-level driver API call, use the `Do` methods provided within the `ParallelExecution` class within the `NationalInstruments.SemiconductorTestLibrary.Common` namespace.

The following example highlights how use the `Do` method with an existing `DigitalSessionsBundle` object to start clock generation by invoking the `ClockGenerator.GenerateClock` method from the niDigital driver API on the underlying PinSet of the niDigital driver session.

> Note that if you need to invoke a method on the driver session object, rather than simply access a property, you must also include a using declaration for the namespace of the driver API at the top of your code file. In the example below requires the following using directive: `using NationalInstruments.ModularInstruments.NIDigital;`

```
public static void GenerateClock(DigitalSessionsBundle sessionsBundle, double frequency)
{
    sessionsBundle.Do((DigitalSessionInformation sessionInfo) =>
    {
        sessionInfo.PinSet.ClockGenerator.GenerateClock(frequency,
selectDigitalFunction: true);
    });
};
```

Similarly, you may want to return values from low-level driver properties.

The following example demonstrates how to get the configured the voltage level for the target DCPower instruments by accessing the `VoltageLevel` property of the low-level niDCPower driver session by using the `DoAndReturnPerPinPerSiteData` method.

```
public static PinSiteData<double> GetVoltageLevel(DCPowerSessionsBundle sessionsBundle)
{
    return sessionsBundle.DoAndReturnPerSitePerPinResults(
        (DCPowerSessionInformation sessionInfo, SitePinInfo sitePinInfo) =>
```

```
        {
            return
sessionInfo.Session.Outputs[sitePinInfo.IndividualChannelString].Source.Voltage.VoltageLevel
;
        });
}
```

Related concepts:

- Extending The Semiconductor Test Library

Extending The Semiconductor Test Library

The Semiconductor Test Library provides extension methods for abstracting common, high-level instrument operations.

If you are unfamiliar Extension Methods, they are static methods that can be called as if they were instance methods for the object type they extend. For tests code being written, there's no apparent difference between calling an extension method and calling the methods defined in a type.

- Extension methods enable the developer to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- An extension method will never be called if it has the same signature as a method defined in the type. So extension method can never be used to impersonate existing methods on a type, because all name collisions are resolved in favor of the instance or static method defined by the type itself.
- Extension methods cannot access any private data in the extended class.
- Learn more at: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods.

Writing Your Own Extension Method

It is recommended that you start by creating a separate .cs file to write your project-specific extensions in, with it's own unique namespace. It must contain a static class, which should use the same name as the file name. You then write out a new static method, with the first parameter being the target class you wish to extend proceeded by the `this` keyword. Finally, can add other parameters to method signature and your desired logic within the method body. Refer to the example below, which extends `PinSiteData` to have a `MaxByPin()` method that calculates the per-site maximum value across each pin.

```
// FileName: MyProjectExtensions.cs
using System.Collections.Generic;
using System.Linq;
using NationalInstruments.SemiconductorTestLibrary.DataAbstraction;

namespace NationalInstruments.MyProject.Extensions
{
    /// <summary>
    /// Class containing custom extension methods for MyProject.
    /// </summary>
    public static class MyProjectExtensions
    {
        /// <summary>
        /// Calculates the per-site maximum value across each pin.
        /// </summary>
        /// <remarks>
```

```
        /// <example>
        /// Example usage:
        /// <code>
        /// var measurements = dcPins.MeasureVoltage();
        /// var maxValueAcrossPins = measurements.MaxByPin();
        /// </code>
        /// </example>
        /// </remarks>
        /// <typeparam name="T">The base type for the per-site per-pin data</typeparam>
        /// <param name="data">The <see cref="PinSiteData{T}"/> object</param>
        /// <returns>
        /// Returns a new <see cref="SiteData{T}"/> object,
        /// containing the per-site maximum values across all pins.
        /// </returns>
        public static SiteData<T> MaxByPin<T>(this PinSiteData<T> data)
        {
            var perSiteMax = new Dictionary<int, T>();
            for (int siteIndex = 0; siteIndex < data.SiteNumbers.Length; siteIndex++)
            {
                var perPinSingleSiteValues = new T[data.PinNames.Length];
                for (int pinIndex = 0; pinIndex < data.PinNames.Length; pinIndex++)
                {
                    perPinSingleSiteValues[pinIndex] =
                    data.GetValue(data.SiteNumbers[siteIndex], data.PinNames[pinIndex]);
                }
                perSiteMax.Add(data.SiteNumbers[siteIndex], perPinSingleSiteValues.Max());
            }
            return new SiteData<T>(perSiteMax);
        }
    }
}
```

Published Data IDs of TestStandSteps

The following table shows the published data ids for the TestStandSteps provided by the Semiconductor Test Library that publish test results.

Method Name	PublishedDataId(s)	Data Type	Units
`AcquireAnalogInputWaveforms`	Minimum	Numeric	*Depends on task type
	Minimum	Numeric	*Depends on task type
`BurstPattern`	Pattern Pass/Fail Result	Boolean	N/A
	Pattern Fail Count	Numeric	N/A
`ContinuityTest`	Continuity	Numeric	Volts
`ForceCurrentMeasureVoltage`	Voltage	Numeric	Volts
`ForceVoltageMeasureCurrent`	Current	Numeric	Amperes
`LeakageTest`	Leakage	Numeric	Amperes

NOTE

When dragging and dropping a step from the TestStand Insertion Pallet into a sequence, the resulting step be linked to the version of the TestStandSteps assembly that ships with the version of STS Software you are using. However, Test tab in the Step Settings pane will not automatically be populated, as this is dependent on the number of pins used by the test. You must populate the Test tab yourself using the PublishedDataIds listed above.

Namespace NationalInstruments.SemiconductorTestLibrary.Common

Classes

NISemiconductorTestException
: Defines a specific exception for NI Semiconductor Test Library.

ParallelExecution
: Defines utility methods for handling parallel execution.

Publish
: Defines measurement results publish methods.

SitePinInfo
: Defines an object that contains the information for an individual site-pin pair. Such as, the site number, pin name, instrument channel string used by the driver, instrument model, etc.

SystemSettings
: Defines system settings.

Utilities
: Provides general helper methods.

Enums

MeasurementType
: Defines whether to measure voltage or current.

Namespace NationalInstruments.SemiconductorTestLibrary.TestStandSteps

Classes

CommonSteps

s

SetupAndCleanupSteps

Defines entry points for semiconductor setup and cleanup steps.

Structs

DMMMeasurementSettings

Defines DMM measurement settings.

Enums

SetupAndCleanupSteps.NIInstrumentType

Defines NI instrument types the NI Semiconductor Test Library supports.