SunSpec System Validation Platform

SunSpec Alliance Users Guide

Version 1.0

ABSTRACT

This document provides an overview of the SunSpec System Validation Platform and information on how to use the tools for running test scripts.

About the SunSpec Alliance

The SunSpec Alliance is a trade alliance of developers, manufacturers, operators and service providers, together pursuing open information standards for the distributed energy industry. SunSpec standards address most operational aspects of PV, storage and other distributed energy power plants on the smart grid—including residential, commercial, and utility-scale systems—thus reducing cost, promoting innovation, and accelerating industry growth.

Over 70 organizations are members of the SunSpec Alliance, including global leaders from Asia, Europe, and North America. Membership is open to corporations, non-profits, and individuals. For more information about the SunSpec Alliance, or to download SunSpec specifications at no charge, please visit www.sunspec.org.

Change History

1.0: Initial version

Table of Contents

Chapter 1
SunSpec System Validation Platform

Introduction

This document describes the concepts and operational details of the SunSpec System Validation Platform (SVP).

Overview

The objective of the SunSpec SVP is to provide a framework for testing and validating SunSpec compliant devices. Two principal types of testing have been targeted: SunSpec protocol conformance testing, and Equipment functional testing.

SunSpec Protocol Conformance Testing

SunSpec protocol conformance tests evaluate the correctness of the implementation of SunSpec information models used by the device. These tests verify that the device can provide and accept data point values as specified in the relevant SunSpec specifications. No functional results of the specific data point settings are evaluated.

Equipment Functional Testing

Equipment functional testing consists of verifying the behavior of the device with specified settings under specific electrical conditions. These tests are comprised of test cases specified in test protocol documents such as the Sandia Inverter Test Protocols[1] and UL 1741 Supplement A.

One of the major objectives of the SunSpec SVP for equipment functional testing is automation of the test cases. Due to the permutations created by multiple device settings under multiple electrical conditions, it is impractical to run a comprehensive set of tests without a high level of automation.

A common use case for device functional testing is inverter control functionality. Functional testing for a device implementing inverter control functionality would typically have the following components, shown in Figure 1: equipment under test, grid simulation, PV simulation, data acquisition, and post-test analysis.

[1] J. Johnson S. Gonzalez, M.E. Ralph, A. Ellis, and R. Broderick, "Test Protocols for Advanced Inverter Interoperability Functions – Main Document," Sandia Technical Report SAND2013- 9880, Nov 2013. J. Johnson S. Gonzalez, M.E. Ralph, A. Ellis, and R. Broderick, "Test Protocols for Advanced Inverter Interoperability Functions–Appendices," Sandia Technical Report SAND2013-9875, Nov 2013.

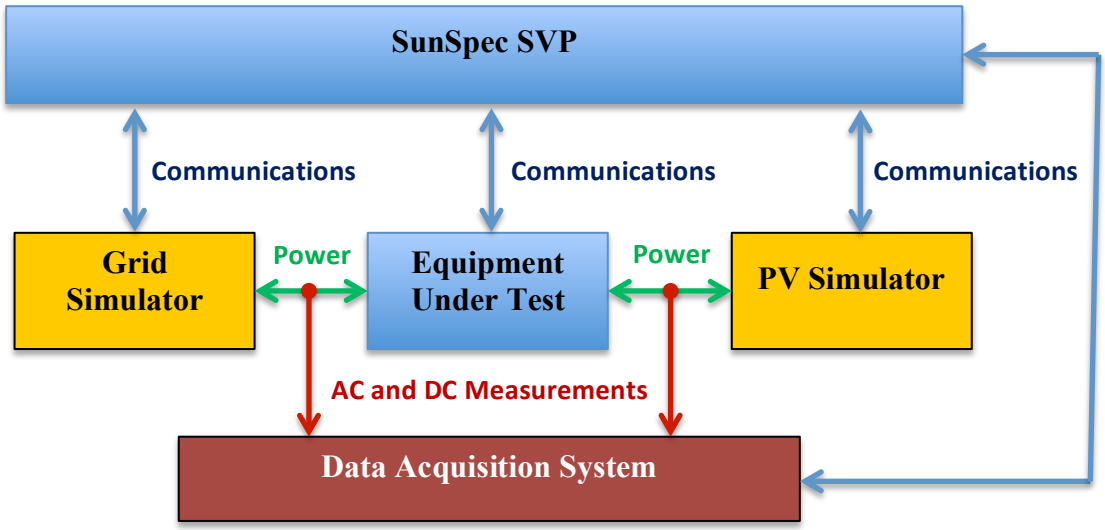


Figure 1: Example SunSpec SVP implementation for a Distributed Energy Resource EUT.

SunSpec SVP Approach

The general approach in the SunSpec SVP is to provide an environment that can manage and execute test scripts that utilize libraries that provide access to all the necessary components in the system. This approach allows for the same test logic to be applied in testing scenarios that may be using different physical components to implement any particular functional block in the test system.

Both the test scripts and support libraries are implemented in Python. Python is a rich language that lends itself to both procedural and object oriented styles of programming.

Script Management, Execution, and Reporting

SunSpec SVP provides management, execution, and report capabilities for the test scripts.

Roles

The following roles are identified in using the system: script runner, script creator/updater, and library creator. A graphical representation of the roles is shown in Figure 2.

Script Runner

The script runner does not require any proficiency in python. The SVP graphical user interface (GUI) can be used to start and stop scripts, set parameters for running scripts, and collect results.

Script Creator/Updater

The script creator requires basic Python proficiency. Scripts and tests can be defined and built using the modules available in the SVP library. The script creator would need to have knowledge of how to interpret the test results for measuring device conformance and compliance.

Library Creator

More advanced Python proficiency is usually required to create and modify support libraries. Objects in this context are sets of functions that can possibly be reused in different scripts.

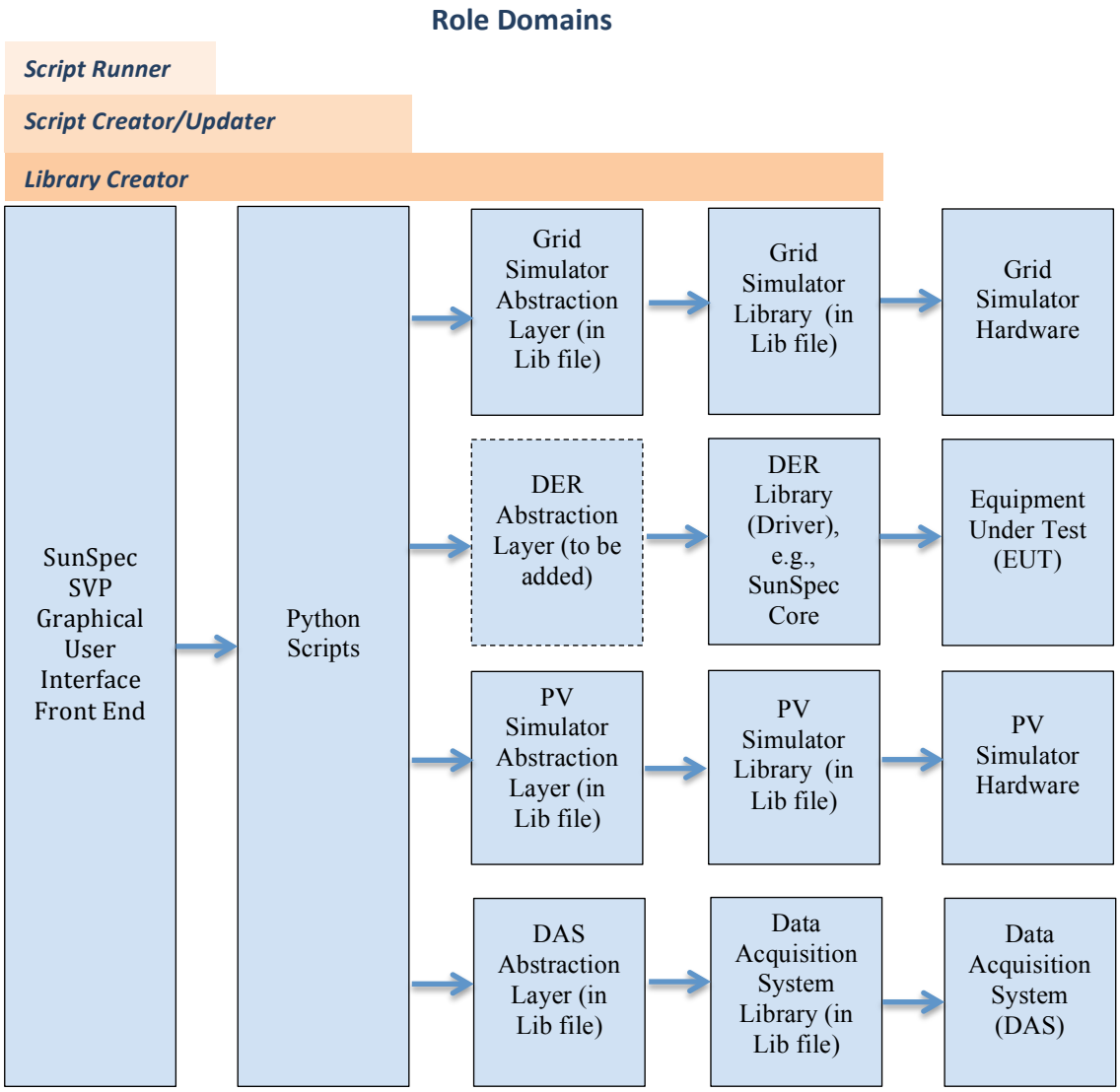


Figure 2: Domains of the SVP roles and overview of possible support libraries.

Simple Procedural Test Scripts

A key objective of the system is to keep the logic in the test scripts as simple as possible. Ideally, test script should be written in a procedural style with the logic being tied as directly as possible to the test protocol documentation.

This allows scripts to be created, updated, and understood by a larger group of users of the system.

Higher complexity interactions with system components should be built into to support libraries.

Support Libraries

Support libraries provide blocks of functionality required in the system. In general, support libraries would be written in a modular, object-oriented style providing objects with rich functionality to be used by test scripts.

Python

The Python language was chosen for its robustness, ease of use, and multi-platform support. Currently the SunSpec SVP is only supported on Windows 7, but support is planned for MacOS and Linux.

The Windows SunSpec SVP installation executable contains a full Python distribution so it is not necessary to have Python installed in the system to use the SVP or even create and run new Python scripts within the SVP environment.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and testing.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

Chapter 2
Installing SunSpec SVP

The SunSpec System Validation Platform is distributed as a single Windows setup executable. The SunSpec SVP installation Windows executable is self-contained and does not require Python to be installed on the system.

Installing the Software

The SunSpec System Validation Platform software is available as a setup .exe file from the SunSpec website, http://www.sunspec.org. Run the executable to unpack and install the software.

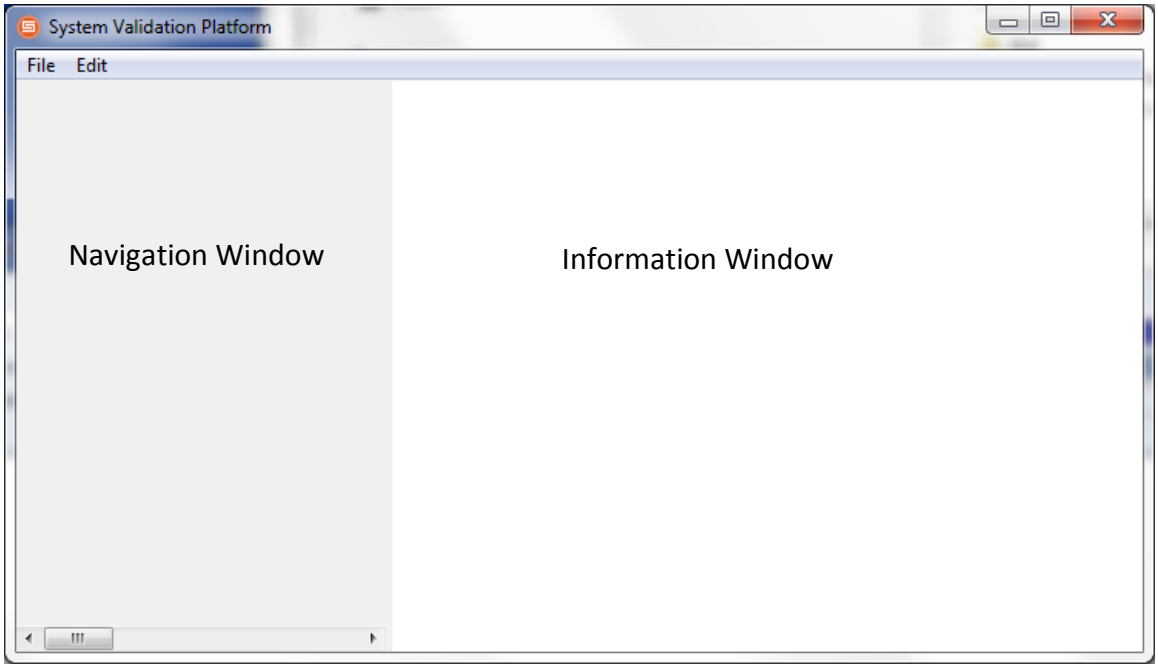
SunSpec SVP Updates

New versions install over old versions without loss of saved context information.

Chapter 3
Using SunSpec SVP

The SunSpec System Validation Platform provides a GUI interface for managing and running SunSpec SVP based test functionality. The main SVP interface consists of an SVP directory navigation window on the left and an information window on the right.

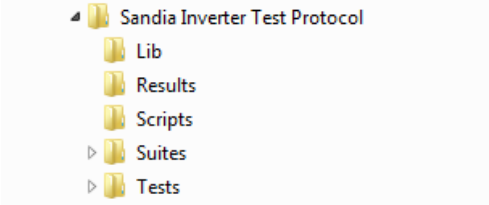
Below is the main SVP interface without any SVP directories loaded:



SunSpec SVP Directory

The SunSpec SVP application allows a user to load and run test functionality bundled as an SVP directory. The SVP application can have multiple SVP directories available for use simultaneously.

An SVP directory is a normal file system directory that conforms to a specific structure. An SVP directory contains at least the following sub-directories: **Lib**, **Results**, **Scripts**, **Suites**, and **Tests**.

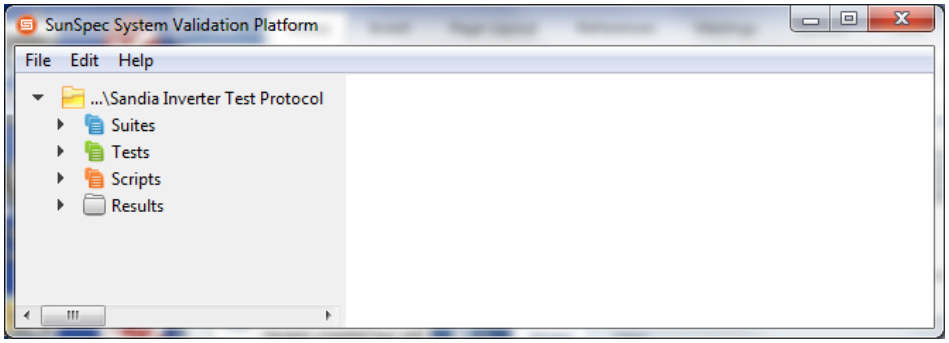


Each of these sub-directories contains the components associated with an SVP functional test instance described below. All the contents of the SVP Directory are referenced relative to the directory location in the system. This allows SVP Directories to be published and easily passed between SVP instances running on different systems.

To load an SVP directory into SVP, use **File->Add SVP Directory**.

Once the directory is loaded in SVP, all interaction with the directory elements should be performed through SVP and usually the directory elements should not be modified outside of SVP.

SVP directory after loading:



SVP remembers the directories that are loaded in the application when SVP is restarted.

To remove an SVP directory, select the directory and use either **Edit->Remove** in the main menu or right click to bring up a context menu with the **Remove** option. The directory reference is removed from SVP but the directory contents are left unchanged in the file system. An SVP directory can be re-added at any time.

SunSpec SVP Terminology and Function

The SunSpec SVP platform is built around three main elements: Scripts, Tests, and Suites. Each element performs a specific function.

Script

The script element has two main attributes: functional logic and an input parameter set.

Functional Logic

A script is written in Python and provides the functional decision making associated with a specific test scenario. An SVP script uses both general purpose and task specific Python libraries to accomplish its task. The SVP platform itself also supplies several Python objects that allow the script to use the resources provided by the platform.

Input Parameter Set

The SunSpec SVP provides a standard mechanism for scripts to define a set of input parameters to be used by a script. This mechanism allows scripts to be written for a particular testing scenario that may want to apply the same test logic to a number of different parameter sets. By using the standard SVP parameter definition technique, individual parameter sets can be created though the SVP interface. A single set of parameter values associated with a script is called a Test.

Test

A Test is a single set of parameter values associated with a Script. Tests can be created based on any Script in the Scripts directory.

Once a Test is created, it can be run in the SVP environment either individually or grouped with other Tests in a specific sequence.

A group of Tests is called a Suite. Suites specify the execution order of a set of Tests or other Suites and allow for global script parameters to be set at the Suite level.

Suite

A Suite specifies the execution order of a group of Tests or other Suites and also provides the ability to set global parameter values.

Test and Suite Execution Order

Suites may contain both Tests and Suites in any order. Suites may not contain other Suites that would create a circular reference.

Set Global Parameters

Script input parameters that are defined as global in the parameter definition in the Script can be set at the Suite level. This allows Tests to be reused and the environment specific parameter settings to be applied at the top level Suite rather than in all the Suites and Tests contained in the Suite.

When a test or suite is referenced in a suite, it is treated as a pointer to the suite or test being referenced and is not a copy. Any change to a suite or test is reflected in any suite that is referencing that suite or test.

SunSpec SVP Directory Structure

An SVP directory contains at least the following sub-directories: **Lib**, **Results**, **Scripts**, **Suites**, and **Tests**.

The **Scripts**, **Suites**, and **Tests** directories contain the scripts, suites, and tests for the functionality contained in the SVP directory.

The **Results** directory contains the results log of all test or suites that are run.

The **Lib** directory is not shown in the SVP directory visualization and contains additional Python files that are used by the scripts in the SVP directory. In general these additional python files will be abstraction layers and Python libraries for specific hardware.

SunSpec SVP Interface

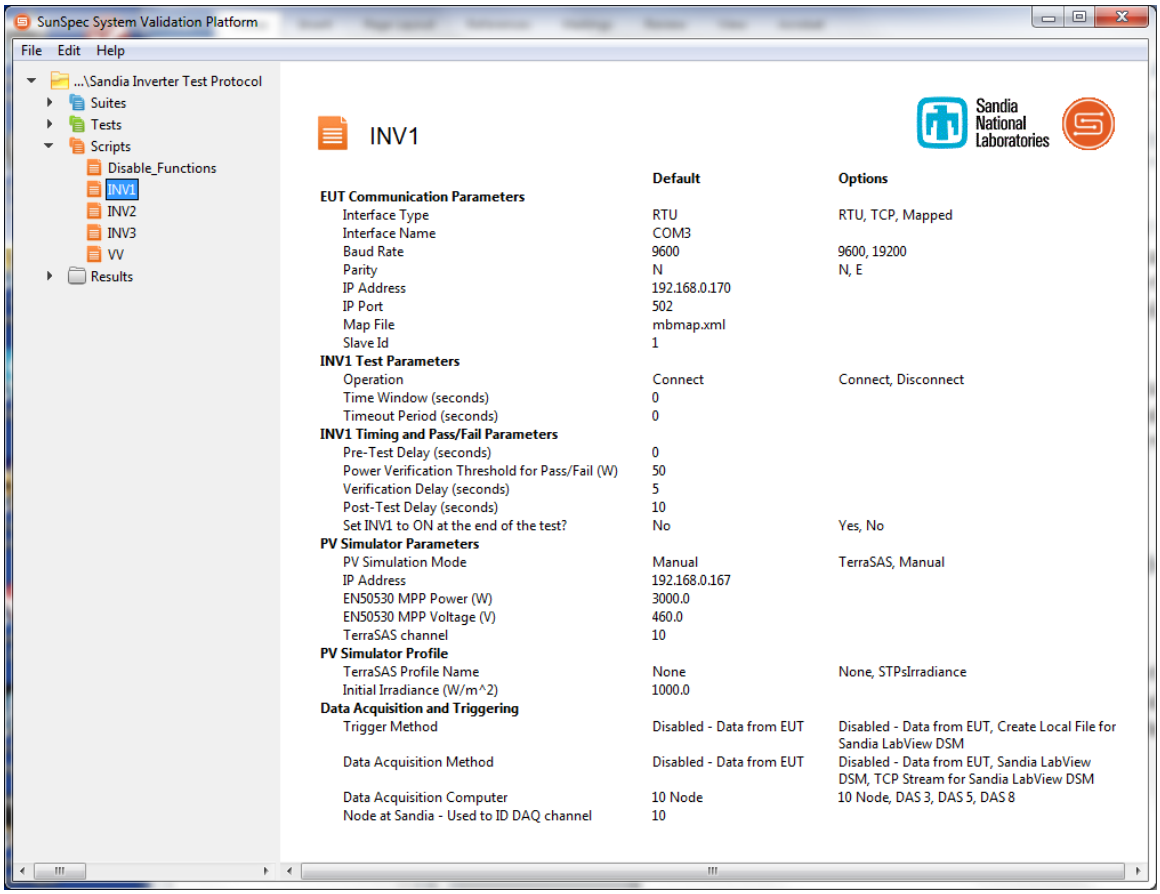
All actions are performed on SunSpec SVP components using the menus in the menu bar or through a context menu invoked by a right click. Almost all operations are available in both the main menu and context menu. The available operations change based on the item currently selected.

In general, all actions on SVP components should be performed through the SVP interface. Altering the SVP components directly in the file system could create inconsistencies that would prevent SVP from managing them properly.

Scripts

Scripts are Python programs that are created in accordance with the SVP structure outlined in the script creation chapter in this document. **Scripts are created outside of SunSpec SVP and added to the Scripts directory.** This allows scripts to be created and even tested in a Python development tool of choice. Script related icons are orange.

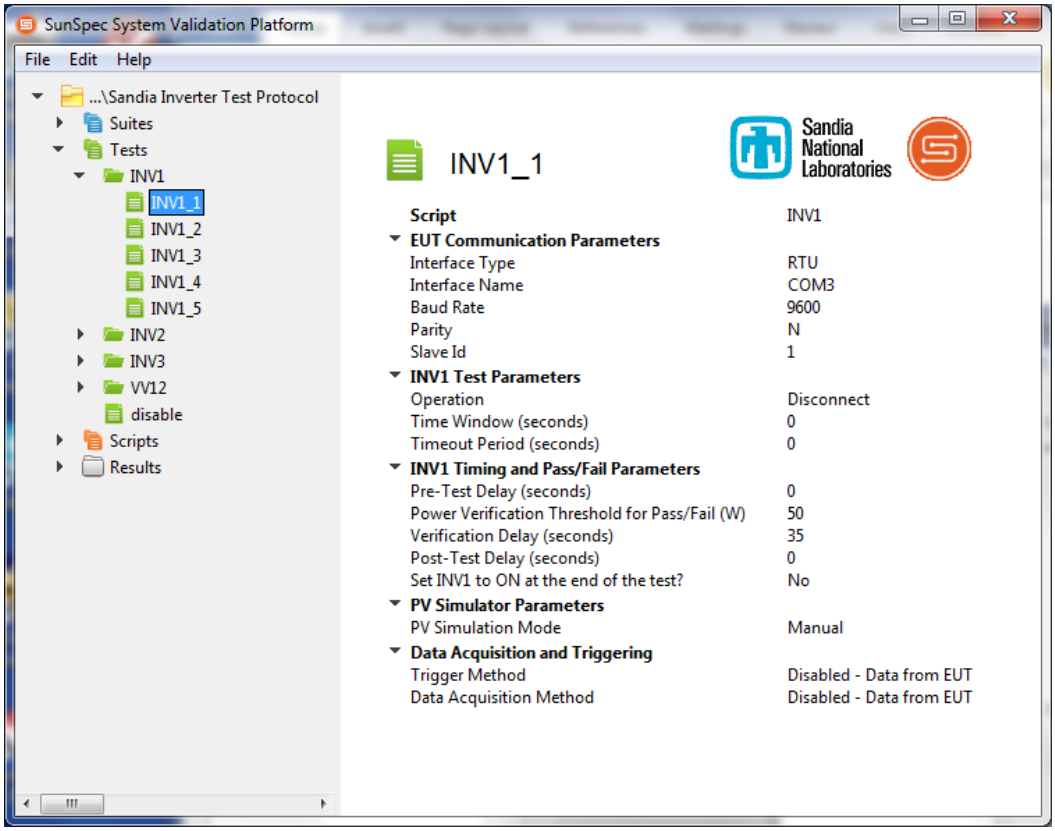
When a script is selected in the navigation window, detailed script information is displayed in the information window. The script information shows all the parameters that are available for the script. The default value and possible values are shown for each parameter.



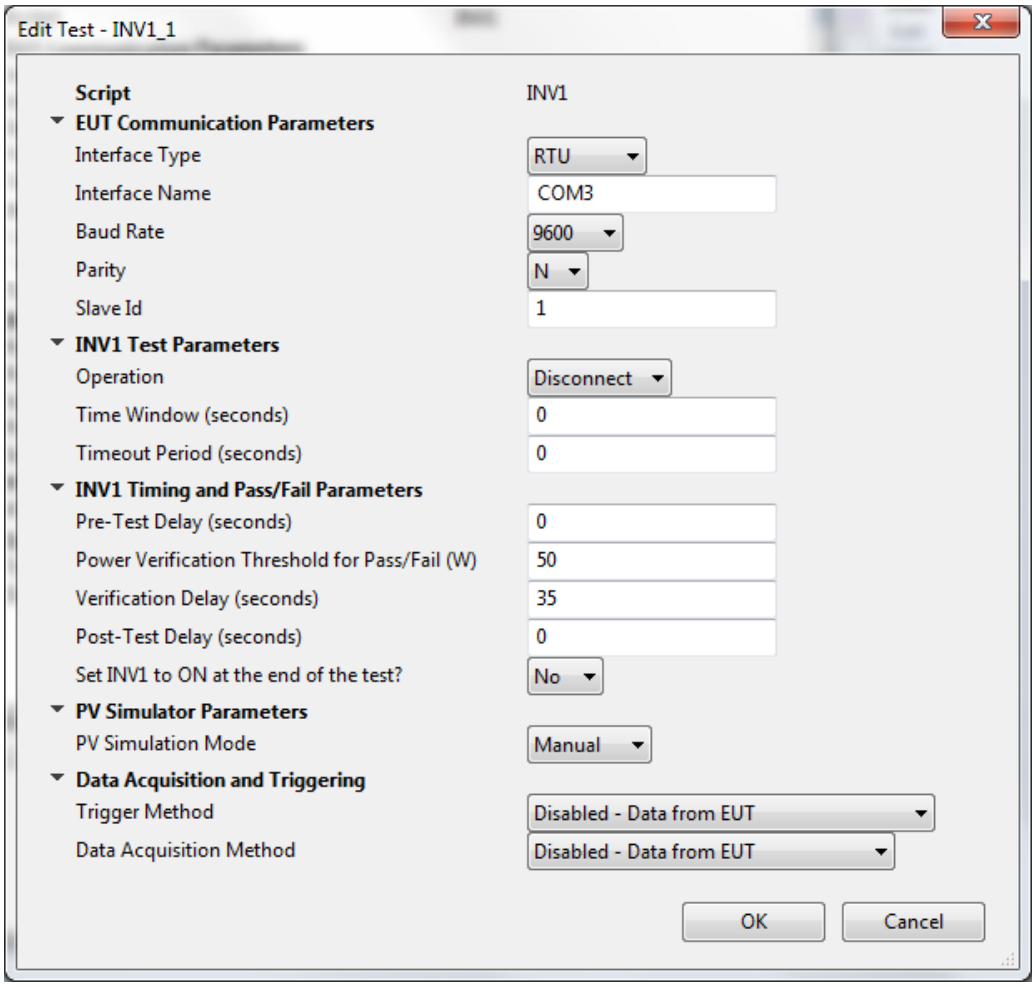
A script that is currently selected can be run with its default values by selecting the **Run** option from the menu.

Tests

When a test is selected in the navigation window, detailed test information is displayed in the information window. The test information shows the script associated with the test and the test parameter values. Test related icons are green.

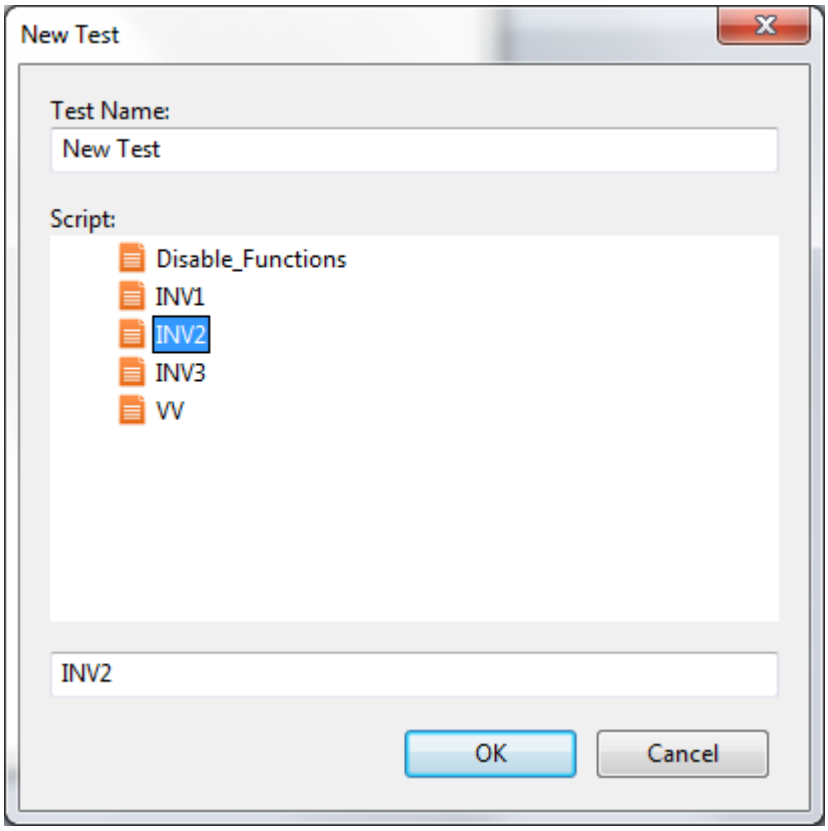


A test that is currently selected can be edited by selecting the edit option from the menu. The edit test dialog allows any test parameter value to be set. The parameters that are available for a test are parameters defined by the script associated with the test.



A new test can be created by first selecting the test directory in which the new test will reside. Use **Select File->New->Test** from the main menu or **New->Test** from the context menu to create the new test.

The new test dialog asks for a test name and the script the test is associated with. Once that information is provided, a standard test edit dialog populated with the default values is used to finish the test creation.



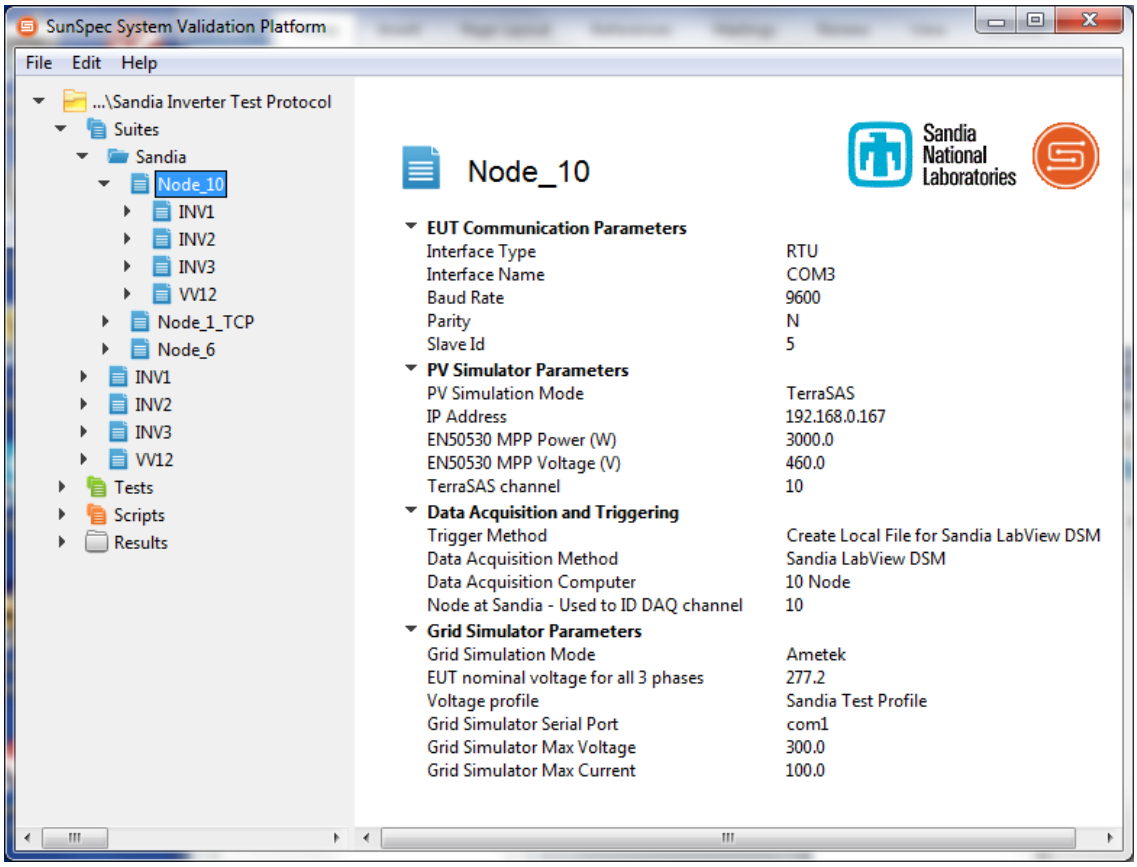
A test that is currently selected can be run by selecting the **Run** option from the menu.

Tests can be moved or renamed using the **Move/Rename** menu option. Any references to the test in suites are updated with the new name.

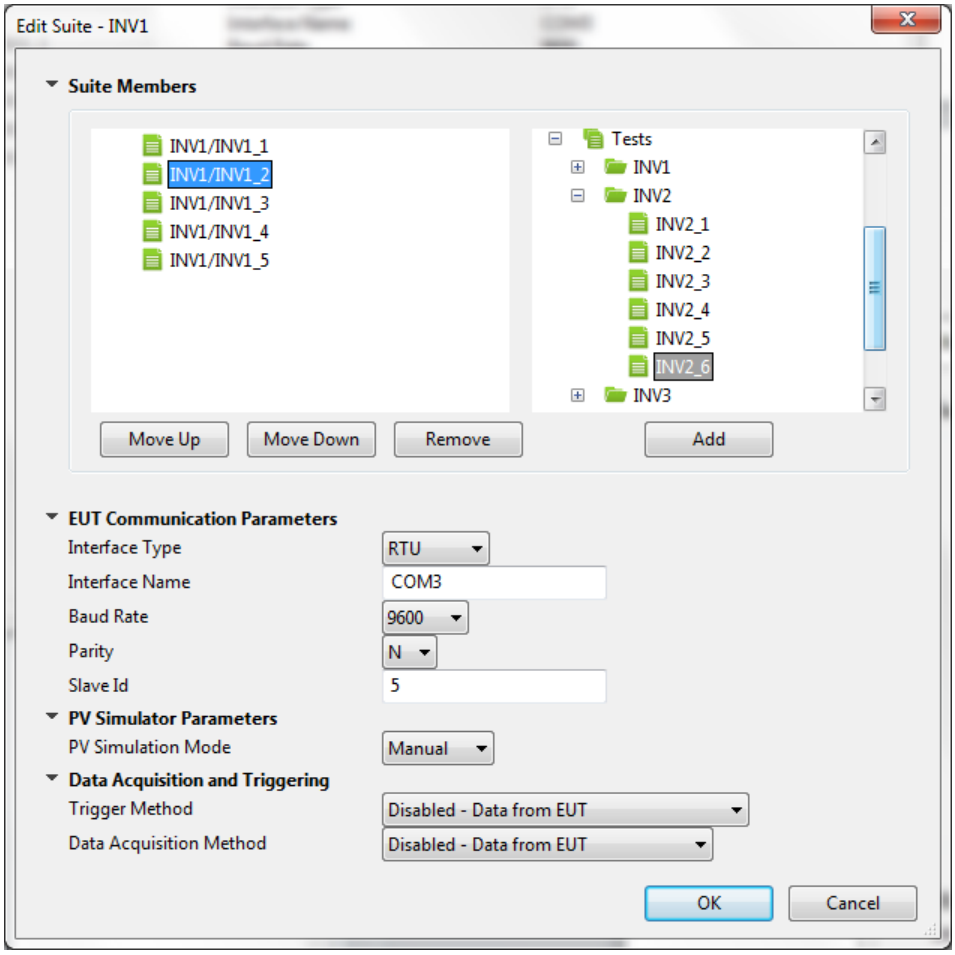
Tests can be deleted using the **Delete** menu option. The test is deleted and any references to that test in suites are removed.

Suites

When a suite is selected in the navigation window, detailed suite information is displayed in the information window. The suite information shows global parameter values associated with the tests and suites contained in the suite. Suite related icons are blue.

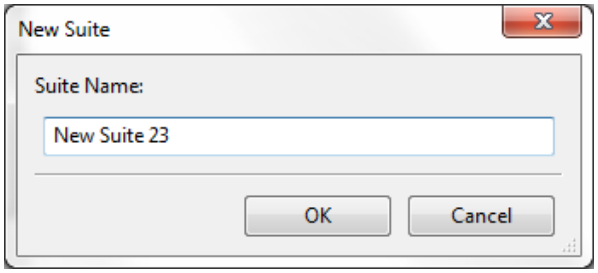


A suite that is currently selected can be edited by selecting the **Edit** option from the menu. The edit suite dialog allows any global parameter value to be set. The parameters that are available for a suite are an aggregation of all of the global parameters associated with the tests and suites contained in the suite.



A new suite can be created by first selecting the test directory in which the new suite will reside. **Select File->New->Suite** from the main menu or **New->Suite** from the context menu to create the new suite.

The new suite dialog asks for the new suite name. Once that information is provided, a standard suite edit dialog populated with the default values is used to finish the suite creation.



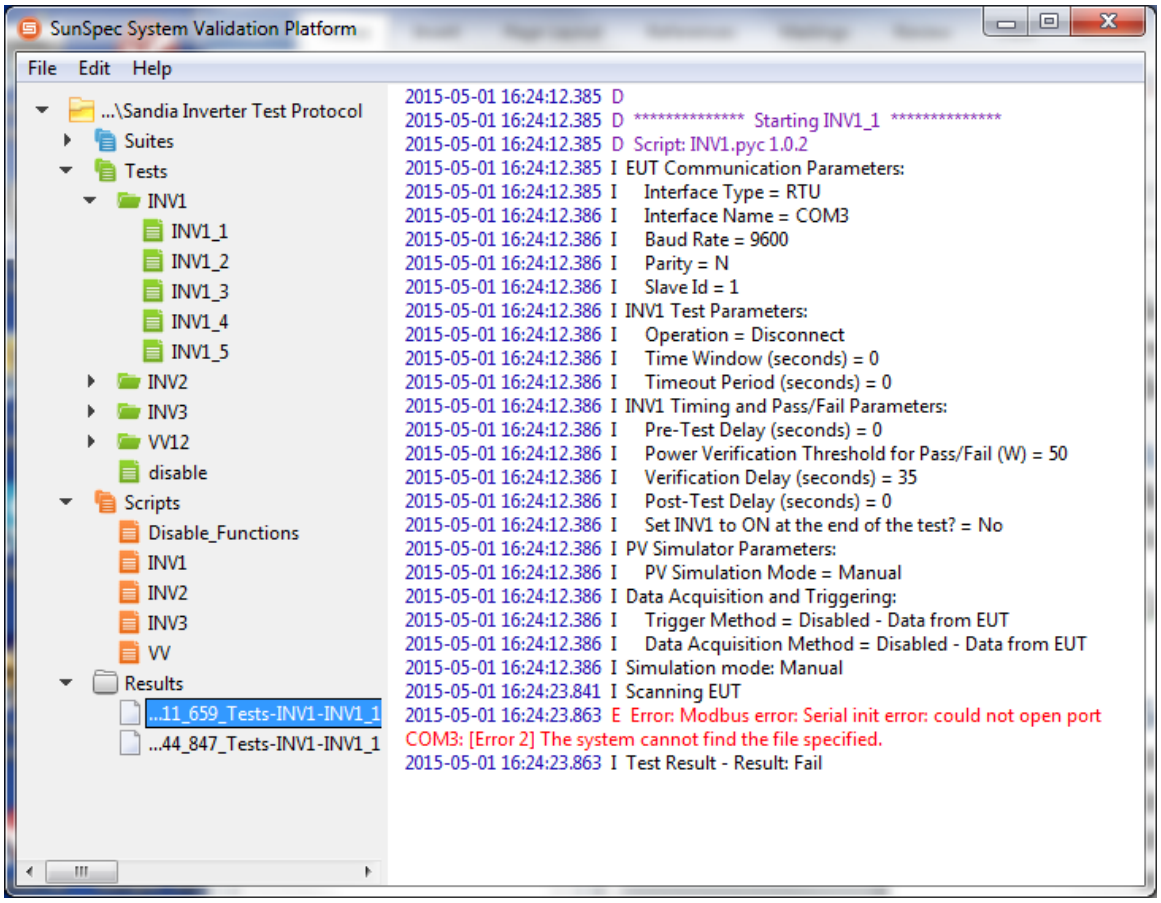
A suite that is currently selected can be run by selecting the **Run** option from the menu. All members of the suite are run in the order they are specified in the suite.

Suites can be moved or renamed using the **Move/Rename** menu option. Any references to the suite in other suites are updated with the new name.

Suites can be deleted using the **Delete** menu option. The suite is deleted and any references to that suite in other suites are removed. The members of the suite being deleted are not deleted.

Results

Results are created by running test or suites. When a result is selected in the navigation window, the result log is displayed in the information window.



SunSpec SVP Execution Interface

When a test or suite is executed in SunSpec SVP using the **Run** option, execution information is shown in the SVP execution interface window. The execution interface window contains a progress and status window on the left and a log window on the right.

Progress Information

The progress and status window on the left provides an entry for each of the members of the suite or test being run along with a result icon indicating success, failure, or completion.

Log Information

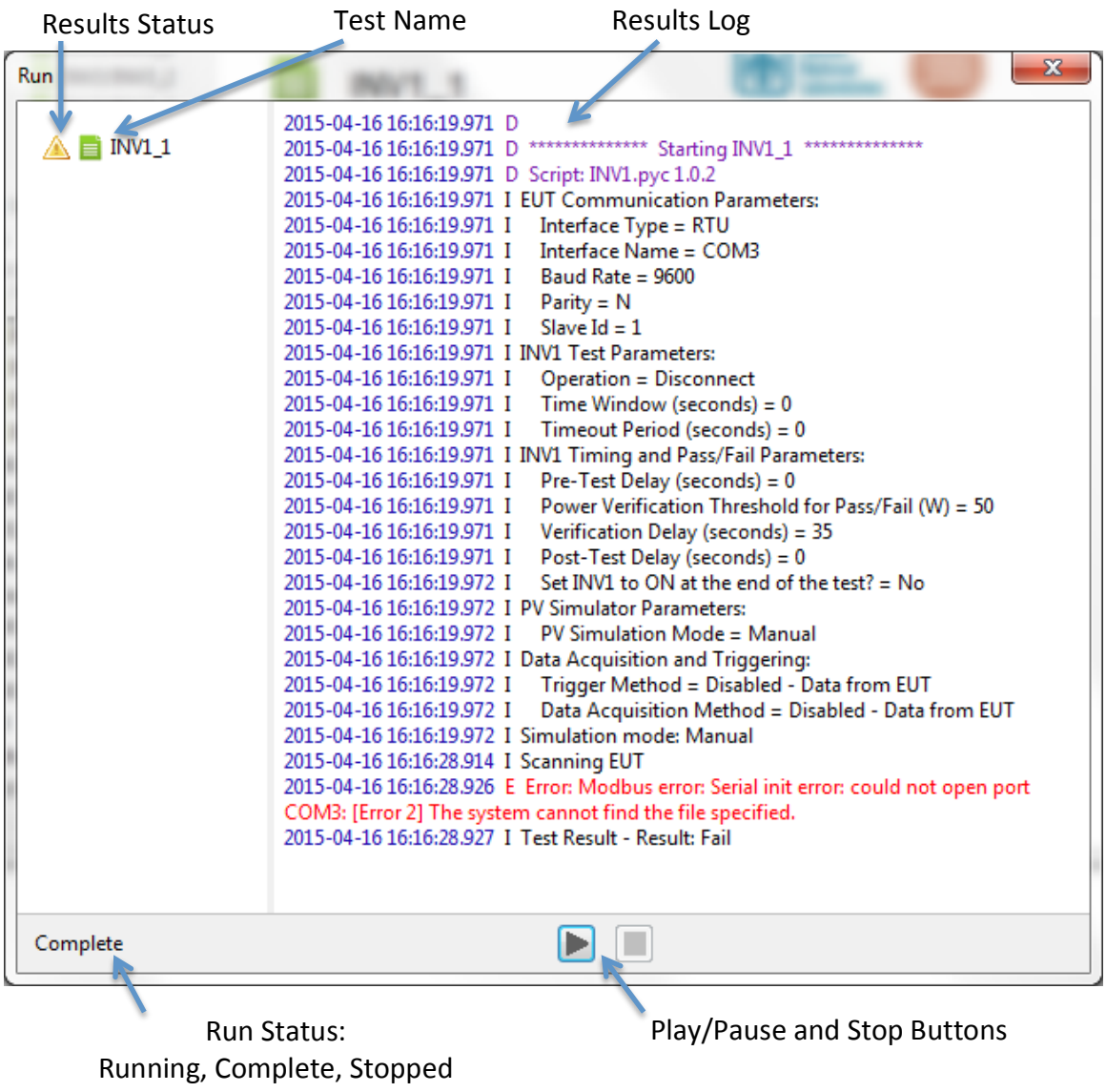
The log window displays the log generated by the script that is running. When there are multiple members being run in sequence, the logs are concatenated in the window.

Each log entry consists of a timestamp, a severity, and a message. Severity is represented by a single letter: D – debug, E – error, I – informational, W – warning.

The contents of the log window are also stored in a result file in the **Results** directory.

Status and Execution Control

The bottom of the execution interface window contains the current status of the running member on the left and **Play/Pause** and **Stop** execution control buttons in the middle.


Results Status
Test Name
Results Log
Run
INV1_1
2015-04-16 16:16:19.971 D
2015-04-16 16:16:19.971 D ********** Starting INV1_1 **********
2015-04-16 16:16:19.971 D Script: INV1.pyc 1.0.2
2015-04-16 16:16:19.971 I EUT Communication Parameters:
2015-04-16 16:16:19.971 I Interface Type = RTU
2015-04-16 16:16:19.971 I Interface Name = COM3
2015-04-16 16:16:19.971 I Baud Rate = 9600
2015-04-16 16:16:19.971 I Parity = N
2015-04-16 16:16:19.971 I Slave Id = 1
2015-04-16 16:16:19.971 I INV1 Test Parameters:
2015-04-16 16:16:19.971 I Operation = Disconnect
2015-04-16 16:16:19.971 I Time Window (seconds) = 0
2015-04-16 16:16:19.971 I Timeout Period (seconds) = 0
2015-04-16 16:16:19.971 I INV1 Timing and Pass/Fail Parameters:
2015-04-16 16:16:19.971 I Pre-Test Delay (seconds) = 0
2015-04-16 16:16:19.971 I Power Verification Threshold for Pass/Fail (W) = 50
2015-04-16 16:16:19.971 I Verification Delay (seconds) = 35
2015-04-16 16:16:19.971 I Post-Test Delay (seconds) = 0
2015-04-16 16:16:19.972 I Set INV1 to ON at the end of the test? = No
2015-04-16 16:16:19.972 I PV Simulator Parameters:
2015-04-16 16:16:19.972 I PV Simulation Mode = Manual
2015-04-16 16:16:19.972 I Data Acquisition and Triggering:
2015-04-16 16:16:19.972 I Trigger Method = Disabled - Data from EUT
2015-04-16 16:16:19.972 I Data Acquisition Method = Disabled - Data from EUT
2015-04-16 16:16:19.972 I Simulation mode: Manual
2015-04-16 16:16:28.914 I Scanning EUT
2015-04-16 16:16:28.926 E Error: Modbus error: Serial init error: could not open port COM3: [Error 2] The system cannot find the file specified.
2015-04-16 16:16:28.927 I Test Result - Result: Fail
Complete
Run Status:
Running, Complete, Stopped
Play/Pause and Stop Buttons

Chapter 4
Creating Scripts

This section describes the SunSpec SVP environment for scripts. Scripts provide the functional logic associated with a particular testing use case. Ideally, the unique testing logic is contained in the script and the more general device support logic is contained in support libraries.

The SunSpec SVP provides support for a number of operations a script might want to perform such as define and acquire input parameters, send messages to the log, prompt for confirmation, alert for exceptional conditions, and supply a final script result.

A script is created and edited in a Python editor of choice and placed in the **Scripts** directory of the SVP directory.

Scripts can be edited and debugged easily while residing in the destination **Scripts** directory.

SunSpec SVP is distributed as a self-contained executable and contains all the functionality of a standard Python 2.7.x release. In addition it contains the following additional libraries that can be imported in any script: pysunspec, pyserial, and numpy.

SunSpec SVP automatically puts the **Lib** directory of the SVP directory in the Python system path (sys.path) to provide access to any libraries in that directory. Additional libraries from an external Python distribution can be used but that solution may break if the SVP directory is moved to system that does not have that library installed.

Script Structure

A script that is structured to run in the SunSpec SVP environment must include the following:

- A function named ***run()*** that has an **svp.script.Script** object as an argument
- A function named ***script_info()*** that returns an **svp.script.ScriptInfo** object
- The script must import **svp.script** to be able to reference the **Script** and **ScriptInfo** class definitions

This is all that is strictly required for a script to run in the SVP environment but there is a little additional script structure that is useful to repeat in each script. A *template.pyx* file is usually distributed in the Script directory as an example starting point. The extension is changed so it will not be visible as a script in the SVP directory.

The **svp.script.ScriptInfo** object is used by the script to define the script input parameters and any logo files that should be associated with the script. See the reference information for **ScriptInfo** below.

When a script is run in SVP, the run function is called with an **svp.script.Script** object as an argument. As mentioned above, the methods of the **Script** object are used to acquire input parameters, send messages to a progress/result log, prompt for confirmation, alert for exceptional conditions, and supply a final script result. See the reference information for **Script** below.

ScriptInfo

A ScriptInfo object is used by a script to publish information about the script. The two types of information that are currently supported are input parameters and any logos that are associated with the script.

class **ScriptInfo**(*Object*)

Methods

logo(*self, filename*)

Associate a logo with the script for display purposes.

filename – The file name of the logo file in the Scripts directory.

param_group(*self, name, label=None, desc=None, active=None, active_value=None, glob=False, index_count=None, index_start=None*)

Define a script parameter group.

name – The name of the parameter group. The name is the identifier that is used to reference it in other parameters or functions associated with parameters.

label – The display label for the parameter group.

desc – The description of the parameter group. The description is displayed in a mouse over of the display label.

active – The active and active value settings are used to indicate that this parameter is only active is any of the values in the active value list match the current value of the parameter specified by active.

active_value – A list of values that are used to determine if the current script parameter is active as outline in the description of active above.

glob – This is a True or False value that indicates if the parameter is global. If a parameter is specified as global, it can be overridden in Suite configurations.

index_count – If index count is specified, the script parameter is implemented as a dictionary with a key created for each possible integer value from *index_start* to *index_count*.

index_start – The starting integer key value for the script parameter dictionary created if *index_count* is specified.

param(*self, name, label=None, default=None, desc=None, values=None, active=None, glob=False, active_value=None, ptype=None, width=None, index_count=None, index_start=None*)

Define a script parameter.

name – The name of the parameter group. The name is the identifier that is used to reference it in other parameters or functions associated with parameters.

label – The display label for the parameter group.

desc – The description of the parameter group. The description is displayed in a mouse over of the display label.

active – The active and active value settings are used to indicate that this parameter is only active is any of the values in the active value list match the current value of the parameter specified by active.

active_value – A list of values that are used to determine if the current script parameter is active as outline in the description of active above.

glob – This is a True or False value that indicates if the parameter is global. If a parameter is specified as global, it can be overridden in Suite configurations. Parameters inherit the *glob* settings of any parents.

ptype – Allows the specification of a specific parameter type for display purposes. The valid values are: PTYPE_DIR, PTYPE_FILE. PTYPE_DIR specifies the parameter is a string representing a directory path in the file system. PTYPE_FILE specifies the parameter is a string representing a file path in the file system.

width – Allows the specification of an alternate display width for the parameter.

index_count – If index count is specified, the script parameter is implemented as a dictionary with a key created for each possible integer value from *index_start* to *index_count*.

index_start – The starting integer key value for the script parameter dictionary created if *index_count* is specified.

Script

An instance of a Script object is passed to the Run function in the script.

class **Script***(Object)*

Methods

sleep(*self, seconds*)

Sleep for the time specified. The time may be a fractional float.

seconds – Time in seconds to sleep.

log(*self, message, level=INFO*)

Create log message.

message – Log message string.

level – Severity level of the log message. Valid severity levels are: DEBUG, ERROR, INFO, and WARNING.

log_debug(*self, message*)

Create debug log message. Same as log(message, level=DEBUG).

message – Log message string.

log_error(self, message)

Create debug log message. Same as log(message, level=ERROR).

message – Log message string.

log_warning(*self, message*)

Create debug log message. Same as log(message, level=WARNING).

message – Log message string.

alert(*self, message*)

Generate an alert message than requires acknowledgment.

message – Alert message string.

param_get(*self, name*)

Get the current value of the script parameter specified by *name*.

Returns the current value.

name – Parameter name.

log_active_params(*self, param_group=None, config=None, level=0*)

Log the current active parameters. Typically used at script startup to log the script configuration.

param_group – Parameter group object.

config - Script configuration object.

level – Indentation level.

confirm(*self, message*)

Generate an confirmation message than requires acknowledgment.

Returns True if confirmed, other False.

message – Confirm message string.

result(*self, result=None, params=None, detail=N***one**)

Set the script result. Should be one of the following: RESULT_COMPLETE, RESULT_PASS, RESULT_FAIL.

result – Result

config_name(*self*)

Returns the name of the script configuration.