niveristand-python Documentation

Release 3.0.0

National Instruments

Jun 22, 2023

USER DOCUMENTATION:

Info	VeriStand Python Support
Author	National Instruments

CHAPTER

ONE

ABOUT

The **niveristand** package contains an API (Application Programming Interface) that interacts with VeriStand systems. The package is implemented in Python. NI created and supports this package.

CHAPTER

TWO

REQUIREMENTS

niveristand requires the following to be installed:

- VeriStand 2020 or later
- CPython 3.8 or later (the standard Python, available on python.org and elsewhere)

2.1 Recommended

NI recommends you use an editor with code completion, such as Visual Studio Code, to make it easier to browse and use this code.

CHAPTER

THREE

INSTALLATION

To install **niveristand**, use one of the following methods:

1. pip:

```
$ python -m pip install niveristand
```

2. **easy_install** from setuptools:

```
$ python -m easy_install niveristand
```

3. Download the project source and run the following script:

```
$ python setup.py install
```

CHAPTER

FOUR

USAGE

Refer to the System Definition Examples section for detailed examples of how to script a system definition file.

Refer to the Basic Real-time Sequence Examples section for detailed information on how to write a Python real-time sequence.

CHAPTER

FIVE

SUPPORT / FEEDBACK

The **niveristand** package is supported by NI. For support for **niveristand**, open a request through the NI support portal at ni.com.

CHAPTER

SIX

BUGS / FEATURE REQUESTS

To report a bug or submit a feature request, please use the GitHub issues page.

CHAPTER

SEVEN

DOCUMENTATION

To view the documentation, visit the VeriStand Python Documentation Page.

CHAPTER

EIGHT

ADDITIONAL DOCUMENTATION

Refer to the VeriStand Help for detailed information on setting up a system and running real-time test scenarios.

VeriStand Help installs only with the full version of VeriStand.

CHAPTER

NINE

LICENSE

niveristand is licensed under an MIT-style license (see LICENSE). Other incorporated projects may be licensed under different licenses. All licenses allow for non-commercial and commercial use.

9.1 Getting Started

9.1.1 Features

niveristand has two major capabilities: system definition scripting, and real-time sequence scripting and deployment. NI recommends you use an editor with code completion, such as Visual Studio Code, to make it easier to browse and use this code.

Scripting system definition files

You can script system definition (.nivssdf) files for use in the NI VeriStand editor and deploy them to the NI VeriStand engine.

Real-time sequences

You can create and run NI VeriStand real-time (RT) sequences from Python that work in both the NI VeriStand engine and Stimulus Profile Editor: - Convert Python functions into real-time sequences and save the converted Python functions to a file. - Run test sequences in two different modes:

- **Deterministic Mode** Deploys a real-time sequence to a running NI VeriStand system and executes the sequence in real-time.
- **Python Mode** Runs a test sequence from a host machine that communicates with an NI VeriStand system through the Gateway. Python mode emulates the behavior of Deterministic mode. Python mode is useful in the following cases:
  - You need to debug your real-time sequence.
  - You want to take full advantage of the Python ecosystem.

Recommended

NI recommends you use an editor with code completion, such as Visual Studio Code, to make it easier to browse and use this code.

9.1.2 Installation

To install **niveristand**, use one of the following methods:

1. pip:

```
$ python -m pip install niveristand
```

2. **easy_install** from setuptools:

```
$ python -m easy_install niveristand
```

3. Download the project source and run the following script:

```
$ python setup.py install
```

9.1.3 Usage

Refer to *System Definition Examples* for detailed examples of how to script a system definition file.

Refer to *Basic Real-time Sequence Examples* for detailed examples of how to write a Python real-time sequence.

9.2 Examples

9.2.1 System Definition Examples

Contents

* *DAQ*
* *CAN*
* *LIN*

Basic Examples

Creating a basic system definition file

A system definition can be created with a target operating system of Windows or Linux_x64.

```
    is_local = ip_address == "localhost" or ip_address == "127.0.0.1"
    target_type = "Windows" if is_local else "Linux_x64"
    system_definition = SystemDefinition(
        filename,
        "This is an example System Definition file created using the System␣
↪Definition API",
        "System Definition API",
        "1.0.0.0",
        "Controller",
        target_type,
        filepath,
    )

    target = system_definition.root.get_targets().get_target_list()[0]
    if not is_local:
        target.ip_address = ip_address
```

Be sure to save the system definition when you have made all necessary modifications.

```
    filepath = system_definition.document_type.document_file_path
    saved, error = system_definition.save_system_definition_file()
    if saved:
        print(f'System Definition saved to "{filepath}"')
    else:
        raise FileNotFoundError(f'Unable to save System Definition to "{filepath}
↪": {error}')
```

Adding User Channels, Calculated Channels, and Aliases

User channels can be added to a target.

```
    target.get_user_channels().add_new_user_channel("MyUserChannel", "", "", 1.0)
    user_channel = target.get_user_channels().get_user_channel_list()[1]

```

Calculated channels can likewise be added to a target.

```
    lpf_calculated_channel = LowpassFilter("MyLowpassFilterChannel", "", user_
↪channel, 50, 1)
    target.get_calculated_channels().add_calculated_channel(lpf_calculated_
↪channel)

```

Aliases get added to the system definition root instead of the target.

```
    alias = Alias("MyUserChannelAlias", "", user_channel)
    alias_folder = AliasFolder("MyAliasFolder", "")
    system_definition.root.get_aliases().add_alias_folder(alias_folder)
    system_definition.root.get_aliases().get_alias_folder_list()[0].add_
↪alias(alias)

```

Adding Alarms and Procedures

Procedures are added to a target.

```
    procedure = Procedure("MyProcedure", "")
    procedure.add_new_dwell("Dwell Step", "Dwell for 3 seconds.", 3)
    procedure.add_new_set_variable(
        "Counting Step", "", user_channel, SetVariableStepFunction.ADD, user_
↪channel, 1
    )
    target.get_procedures().add_procedure(procedure)

```

Alarms are also added to a target.

```
    target.get_alarms().add_new_alarm(
        "MyAlarm",
        "",
        user_channel,
        2,
        1,
```

(continues on next page)

(continued from previous page)

```
        procedure,
        AlarmMode.NORMAL,
        AlarmState.ENABLED,
        AlarmPriority.LOW,
        0,
        "",
    )

```

Adding Models

Models are added to a target.

```
    target = system_definition.root.get_targets().get_target_list()[0]
    simulation_models = target.get_simulation_models()

    random_model = Model("RandomFMU", "", get_asset("RandomFMU.fmu"), 0, 1, 0,␣
↪True, True, True)
    simulation_models.get_models().add_model(random_model)

```

More Detailed Examples

DAQ

There are many options available when adding DAQ devices.

```
    daq_device = DAQDevice(
        "Dev1",
        "This is a DAQ Device created using the System Definition Offline API.",
        DAQDeviceInputConfiguration.DEFAULT,
    )
    target = system_definition.root.get_targets().get_target_list()[0]
    chassis = target.get_hardware().get_chassis_list()[0]
    chassis.get_daq().add_device(daq_device)

    # Analog Input Channels
    analog_inputs = daq_device.create_analog_inputs()
    analog_inputs.add_analog_input(
        DAQAnalogInput("AI0", 1, DAQMeasurementType.ANALOG_INPUT_TEMPERATURE_
↪THERMOCOUPLE)
    )
    analog_inputs.add_analog_input(
```

(continues on next page)

(continued from previous page)

```
        DAQAnalogInput("AI1", 0, DAQMeasurementType.ANALOG_INPUT_VOLTAGE)
    )

    # Analog Output Channels
    analog_outputs = daq_device.create_analog_outputs()
    analog_outputs.add_analog_output(
        DAQAnalogOutput("AO0", 0, DAQMeasurementType.ANALOG_OUTPUT_VOLTAGE)
    )

    # Digital Input Channels
    digital_inputs = daq_device.create_digital_inputs()
    daq_input_port = DAQDIOPort(0, False)
    digital_inputs.add_dio_port(daq_input_port)
    daq_input_port.add_digital_input(DAQDigitalInput("DI0", False, 0, 0))
    daq_input_port.add_digital_input(DAQDigitalInput("DI1", False, 1, 0))

    # Digital Output Channels
    digital_outputs = daq_device.create_digital_outputs()
    daq_output_port = DAQDIOPort(1, False)
    digital_outputs.add_dio_port(daq_output_port)
    daq_output_port.add_digital_output(DAQDigitalOutput("DO4", False, 4, 1))

    # Counter Channels
    counters = daq_device.create_counters()
    counters.add_counter(
        DAQFrequencyMeasurement("FreqIn", "", 0, 0.0, 1.0, 0.0, DAQCounterEdge.
↪FALLING)
    )
    counters.add_counter_output(DAQPulseGeneration("PWMOut", "", 1))

    # Internal Channels
    internal_channels = daq_device.create_internal_channels()
    internal_channels.add_internal_channel(DAQInternalChannel("Channel 0", 0.0))

    # Waveform Tasks
    daq_device_waveform = DAQDevice("Dev2", "", DAQDeviceInputConfiguration.
↪DEFAULT)
    chassis.get_daq().add_device(daq_device_waveform)
    daq_tasks = chassis.get_daq().get_tasks()
    waveform_task = DAQTaskAI("Task1", 1000, AcquisitionMode.CONTINUOUS)
    waveform_task.get_triggers().start_trigger = DAQTriggerDigitalEdge(
        "PFI0", DirectionType.FALLING
    )
    daq_tasks.add_task(waveform_task)
    analog_waveform_input = DAQWaveformAnalogInput(
        "AI2", 0, DAQMeasurementType.ANALOG_INPUT_CURRENT
```

(continues on next page)

(continued from previous page)

```
    )
    waveform_analog_inputs = daq_device_waveform.create_analog_inputs()
    waveform_analog_inputs.sample_mode = SampleMode.WAVEFORM
    waveform_analog_inputs.add_waveform_analog_input(analog_waveform_input)
    waveform_analog_inputs.waveform_analog_input_task = waveform_task

    # Polynomial Scale
    coefficients = [1.0]
    reverse_coefficients = []
    scale = PolynomialScale("MyScale", coefficients, reverse_coefficients, "")
    system_definition.root.get_scales().add_scale(scale)
    daq_device.get_analog_input_section().get_analog_input_list()[0].scale =␣
↪scale
```

CAN

An NI-XNET CAN interface can be added.

```
    target = system_definition.root.get_targets().get_target_list()[0]
    chassis = target.get_hardware().get_chassis_list()[0]
    chassis.get_xnet().enable_xnet()  # enable XNET if we haven't already

    target.get_user_channels().add_new_user_channel("MyXnetUserChannel", "", "",␣
↪1.0)
    user_channel = target.get_user_channels().get_user_channel_list()[0]

    # CAN Database
    can = chassis.get_xnet().get_can()
    can_database = Database("NIXNET_example")
    target.get_xnet_databases().add_database(can_database)

    # CAN Cluster
    can_cluster = "CAN_Cluster"
    can_port = CANPort("CAN 1", 1, can_database, can_cluster, 125000)
    can_port.termination = XNETTermination.ON
    can.add_can_port(can_port)

    # Frame Variables
    cyclic_frame = "CANCyclicFrame1"
    cyclic_frame_signals = ["CANCyclicSignal1", "CANCyclicSignal2"]
    event_frame = "CANEventFrame1"
    event_frame_signals = ["CANEventSignal1", "CANEventSignal2"]

    # CAN Incoming Frames
```

(continues on next page)

(continued from previous page)

```
    incoming_cyclic_frame = SignalBasedFrame(
        cyclic_frame, 64, can_database, can_cluster, 8, 0.1, False, cyclic_frame_
↪signals
    )
    can_port.get_incoming().get_single_point().add_signal_based_frame(incoming_
↪cyclic_frame)
    for signal in cyclic_frame_signals:
        incoming_cyclic_frame.create_signal_based_signal(signal, "", "volts", 0.
↪0)

    incoming_event_frame = SignalBasedFrame(
        event_frame, 66, can_database, can_cluster, 8, 0.1, False, event_frame_
↪signals
    )
    can_port.get_incoming().get_single_point().add_signal_based_frame(incoming_
↪event_frame)
    for signal in event_frame_signals:
        incoming_event_frame.create_signal_based_signal(signal, "", "volts", 0.0)

    incoming_cyclic_frame.create_frame_information()

    incoming_data_logging = DataLoggingFile(
        "log", "file", os.path.dirname(system_definition.document_type.document_
↪file_path)
    )
    incoming_data_logging.data_logging_file_type = FileType.TDMS
    can_port.get_incoming().get_raw_frame_data_logging().add_data_logging_file(
        incoming_data_logging
    )
    can_port.get_incoming().get_raw_frame_data_logging().get_data_logging_file_
↪list()[
        0
    ].trigger_channel = user_channel

    # CAN Outgoing Frames
    outgoing_cyclic_frame = SignalBasedFrame(
        cyclic_frame, 64, can_database, can_cluster, 8, 0.1, False, cyclic_frame_
↪signals
    )
    can_port.get_outgoing().get_cyclic().add_signal_based_frame(outgoing_cyclic_
↪frame)
    for signal in cyclic_frame_signals:
        outgoing_cyclic_frame.create_signal_based_signal(signal, "", "volts", 0.
↪0)

    outgoing_event_frame = SignalBasedFrame(
```

(continues on next page)

(continued from previous page)

```
        event_frame, 64, can_database, can_cluster, 8, 0.1, False, event_frame_
↪signals
    )
    can_port.get_outgoing().get_event_triggered().add_signal_based_
↪frame(outgoing_event_frame)
    for signal in event_frame_signals:
        outgoing_event_frame.create_signal_based_signal(signal, "", "volts", 0.0)

    outgoing_cyclic_frame.create_frame_faulting(True, True)
    outgoing_cyclic_frame.get_frame_faulting().get_skip_cyclic_frames().trigger_
↪channel = (
        user_channel
    )
    outgoing_cyclic_frame.get_frame_faulting().get_transmit_time().set_trigger_
↪channel(user_channel)

    outgoing_data_replay = DataFileReplay("replay", get_asset("fake.tdms"))
    can_port.get_outgoing().get_data_replay().add_data_file_replay(outgoing_data_
↪replay)
    can_port.get_outgoing().get_data_replay().get_data_file_replay_list()[
        0
    ].trigger_channel = user_channel
```

LIN

An NI-XNET LIN interface can be added.

```
    target = system_definition.root.get_targets().get_target_list()[0]
    chassis = target.get_hardware().get_chassis_list()[0]
    chassis.get_xnet().enable_xnet()  # enable XNET if we haven't already

    # LIN Database
    lin = chassis.get_xnet().get_lin()
    lin_database = Database("NIXNET_exampleLDF")
    target.get_xnet_databases().add_database(lin_database)

    # LIN Cluster
    lin_cluster = "Cluster"
    lin_port = LINPort("LIN 1", 1, lin_database, lin_cluster, 125000,
↪"SlowSchedule")
    lin.add_lin_port(lin_port)

    # LIN Incoming Frame
    incoming_signals = ["SlaveSignal3_U8", "SlaveSignal4_U8"]
```

(continues on next page)

(continued from previous page)

```
    incoming_frame = SignalBasedFrame(
        "Slave1Frame2", 5, lin_database, lin_cluster, 8, 0.1, False, incoming_
↪signals
    )
    lin_port.get_incoming().get_single_point().add_signal_based_frame(incoming_
↪frame)
    for signal in incoming_signals:
        incoming_frame.create_signal_based_signal(signal, "", "volts", 0.0)

    # LIN Outgoing Frame
    outgoing_signals = ["MasterSignal3_U8", "MasterSignal4_U8"]
    outgoing_frame = SignalBasedFrame(
        "MasterFrame2", 3, lin_database, lin_cluster, 8, 0.1, False, outgoing_
↪signals
    )
    lin_port.get_outgoing().get_unconditional().add_signal_based_frame(outgoing_
↪frame)
    for signal in outgoing_signals:
        outgoing_frame.create_signal_based_signal(signal, "", "volts", 0.0)
```

9.2.2 Basic Real-time Sequence Examples

Writing a real-time sequence

A Python real-time sequence is a Python function decorated with the *niveristand.nivs_rt_sequence* decorator. For example, the following sequence calls a function and checks the result.

```
@nivs_rt_sequence
def call_add_two_numbers_test():
    result = DoubleValue(0)
    result.value = add_two_numbers(1, 2)
    if result.value != 3:
        generate_error(-100, "Unexpected result", ErrorAction.
↪ContinueSequenceExecution)
```

The function also takes in some parameters. You must define parameters using the *niveristand.NivsParam* decorator.

```
@NivsParam("x", DoubleValue(0), NivsParam.BY_VALUE)
@NivsParam("y", DoubleValue(0), NivsParam.BY_VALUE)
@nivs_rt_sequence
def add_two_numbers(x, y):
    result = DoubleValue(0)
    # There is an intentional mistake here. It multiplies when the function␣
↪implies it adds.
```

(continues on next page)

(continued from previous page)

```
    result.value = x.value * y.value
    return result.value
```

You can now run the test just like any other Python function. You can run it non-deterministically, as in the following example:

```
    try:
        call_add_two_numbers_test()
    except RunError as run_error:
        print("Something Non-deterministic went wrong:" + str(run_error))

```

Or, you can run the test deterministically on the VeriStand engine connected to your system.

```
    try:
        realtimesequencetools.run_py_as_rtseq(call_add_two_numbers_test)
    except RunError as run_error:
        print("Something Deterministic went wrong:" + str(run_error))
```

Combining the legacy API with real-time sequences

To create a fully-automated test environment, you can mix the *Legacy API* with Python real-time sequences.

```
import os
from examples.engine_demo.engine_demo_basic import run_engine_demo
from niveristand import run_py_as_rtseq
from niveristand.errors import RunError
from niveristand.legacy import NIVeriStand


def mix_legacy_and_rtseq_run():
    """Combines the legacy API with Python real-time sequences to run a␣
↪deterministic test."""
    # Ensures NI VeriStand is running.
    NIVeriStand.LaunchNIVeriStand()
    NIVeriStand.WaitForNIVeriStandReady()
    # Uses the ClientAPI interface to get a reference to Workspace2
    workspace = NIVeriStand.Workspace2("localhost")
    engine_demo_path = os.path.join(
        os.path.expanduser("~public"),
        "Documents",
        "National Instruments",
        "NI VeriStand 2019",
        "Examples",
        "Stimulus Profile",
```

(continues on next page)

(continued from previous page)

```
        "Engine Demo",
        "Engine Demo.nivssdf",
    )
    # Deploys the system definition.
    workspace.ConnectToSystem(engine_demo_path, True, 120000)
    try:
        # Uses Python real-time sequences to run a test.
        run_py_as_rtseq(run_engine_demo)
        print("Test Success")
    except RunError as e:
        print("Test Failed: %d -  %s" % (int(e.error.error_code), e.error.
↪message))
    finally:
        # You can now disconnect from the system, so the next test can run.
        workspace.DisconnectFromSystem("", True)


if __name__ == "__main__":
    mix_legacy_and_rtseq_run()
```

Array operations example

```
@nivs_rt_sequence
def array_operations():
    """
    Shows operations you can perform with array data types in a real-time␣
↪sequence.

    An array can hold multiple values of the same data type. You cannot have␣
↪arrays of arrays.
    Use arrays to pass buffers of data for playback or storage.

    Returns:
        float: sum of all values in the array.

    """
    var = DoubleValue(0)
    arr_size = I64Value(0)
    array = DoubleValueArray([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

    # Indexes a value out of an array.
    var.value = array[4].value + 100
    # Updates a value in an array.
    array[2].value = 6.0
```

(continues on next page)

(continued from previous page)

```
    # Gets the size of an array.
    arr_size.value = arraysize(array)
    # Loops over each element of an array. Each time the loop iterates a value from the array is copied into x.
    var.value = 0.0
    for x in array:
        var.value += x
    return var.value
```

Measuring elapsed time example

```
@nivs_rt_sequence
def measure_elapsed_time():
    """
    Shows different ways to measure elapsed time in a sequence.

    You can measure time in milliseconds, microseconds, or seconds.

    Returns:
        int: time, in milliseconds, it took to run this sequence.

    """
    seqtime_timer = DoubleValue(0)
    seqtime_us_timer = I64Value(0)
    tick_ms_timer = I64Value(0)
    tick_us_timer = I64Value(0)

    # The following steps demonstrate different ways you can capture an initial timestamp:
    seqtime_timer.value = seqtime()
    seqtime_us_timer.value = seqtimeus()
    tick_ms_timer.value = tickcountms()
    tick_us_timer.value = tickcountus()

    # Simulates work to time.
    while iteration() < 1000:
        nivs_yield()

    # Measures the elapsed time by subtracting the initial timestamp from the current time.
    seqtime_timer.value = seqtime() - seqtime_timer.value
    seqtime_us_timer.value = seqtimeus() - seqtime_us_timer.value
    tick_ms_timer.value = tickcountms() - tick_ms_timer.value
    tick_us_timer.value = tickcountus() - tick_us_timer.value
```

(continues on next page)

(continued from previous page)

```

    return tick_ms_timer.value
```

State machine

```
@nivs_rt_sequence
def state_machine_example():
    state = I32Value(0)
    iters = I32Value(0)
    amplitude = DoubleValue(1000)
    stop = BooleanValue(False)
    output = ChannelReference("Aliases/DesiredRPM")

    while (
        stop.value != True  # noqa: E712 NI recommends you use comparison␣
↪instead of identity.
        and iters.value < 10
    ):
        state.value = rand(7)
        if state.value == 0:
            wait(2)
        elif state.value == 1:
            sine_wave(output, amplitude, 1, 0, 0, 2)
        elif state.value == 2:
            square_wave(output, amplitude, 5, 0, 0, 50, 2)
        elif state.value == 3:
            triangle_wave(output, amplitude, 1, 0, 0, 2)
        elif state.value == 4:
            uniform_white_noise_wave(output, amplitude, tickcountus(), 2)
        elif state.value == 5:
            ramp(output, -amplitude.value, amplitude, 2)
        elif state.value == 6:
            sawtooth_wave(output, amplitude, 1, 0, 0, 2)
        else:
            stop.value = True
        iters.value += 1
        state.value = rand(7)
```

9.2.3 Engine Demo Examples

Engine Demo Basic

```python
from niveristand import nivs_rt_sequence, NivsParam, realtimesequencetools
from niveristand.clientapi import BooleanValue, ChannelReference, DoubleValue
from niveristand.library import wait

""" This module contains a basic example of how to create an RT sequence in Python.

This example mirrors the 'Engine Demo Basic' example that installs with VeriStand.
Open the 'Engine Demo Basic' stimulus profile to help you understand the following example.
"""


# You must mark RT sequences with the following decorator:
@nivs_rt_sequence
# You must also specify parameter data types, default values, and whether to pass parameters by value or by reference.
@NivsParam("engine_power", BooleanValue(0), NivsParam.BY_REF)
@NivsParam("desired_rpm", DoubleValue(0), NivsParam.BY_REF)
def engine_demo_basic(engine_power, desired_rpm):
    """Turn on the engine, set the desired_rpm to the passed value for 20 seconds, and shut down the engine.

    You must access parameters through their ".value" property.
    """
    # You can access a channel with a ChannelReference
    engine_power_chan = ChannelReference("Aliases/EnginePower")
    desired_rpm_chan = ChannelReference("Aliases/DesiredRPM")
    engine_power_chan.value = engine_power.value
    desired_rpm_chan.value = desired_rpm.value
    wait(DoubleValue(20))
    engine_power_chan.value = False
    desired_rpm_chan.value = 0


@nivs_rt_sequence
def run_engine_demo():
    """Sets up channel references and calls the actual test."""
    # You can call an RT sequence the same way you call a normal Python function.
    # However, if you pass functions by reference, you must create strongly-typed objects.
    engine_demo_basic(BooleanValue(True), DoubleValue(2500))
```

(continues on next page)

(continued from previous page)

```
38
39
40  def run_non_deterministic():
41      """Runs the sequence non-deterministically.
42
43      This function executes the RT Sequence on the host using the public ClientAPI
44      that installs with VeriStand. This function communicates with the gateway to␣
    ↪set and get channel values.
45
46      If you use a Python integrated developer environment (IDE),
47      you can debug this function like a normal Python function.
48      """
49      run_engine_demo()
50
51
52  def run_deterministic():
53      """Compiles the sequence and runs it deterministically inside the VeriStand␣
    ↪engine.
54
55      You cannot use debugging tools at this stage because VeriStand executes the␣
    ↪sequence, not Python.
56      If you do not mark the functions as @nivs_rt_sequence, Python will raise a␣
    ↪:any:`niveristand.errors.VeristandError`.
57      """
58      # The run_py_as_rtseq function accepts, as a parameter, the Python function␣
    ↪you want to call as an RT sequence.
59      realtimesequencetools.run_py_as_rtseq(run_engine_demo)
60
61
62  if __name__ == "__main__":
63      realtimesequencetools.save_py_as_rtseq(run_engine_demo, "d:\\share\\temp\\
    ↪demo")
64      run_non_deterministic()
65      print("Finished non-deterministic")
66      run_deterministic()
67      print("Finished deterministic")
```

Engine Demo Advanced

```
from niveristand import nivs_rt_sequence, NivsParam, run_py_as_rtseq
from niveristand.clientapi import BooleanValue, ChannelReference, DoubleValue
from niveristand.library import multitask, nivs_yield, stop_task, task, wait_
↪until_settled

""" This module adds multitasking, return values, and cleanup tasks to engine_
↪demo_basic.

This example mirrors the 'Engine Demo Advanced and Return Value' example that␣
↪installs with VeriStand.
Open the 'Engine Demo Advanced and Return Value' stimulus profile to help you␣
↪understand the following example.
"""


@NivsParam("desired_rpm", DoubleValue(0), NivsParam.BY_REF)
@NivsParam("actual_rpm", DoubleValue(0), NivsParam.BY_REF)
@NivsParam("engine_temp", DoubleValue(0), NivsParam.BY_REF)
@nivs_rt_sequence
def engine_demo_advanced(desired_rpm, actual_rpm, engine_temp):
    """Turns on the engine, sets it to the desired rpm, and monitors the engine␣
↪temperature."""
    # Use the following local variable declarations to keep track of the test's␣
↪status:
    warmup_complete = BooleanValue(False)
    warmup_succeeded = BooleanValue(False)

    # Create a multitask with two tasks: one for setting rpm values and another␣
↪for monitoring.
    # In general, a multitask can contain as many tasks as desired. The tasks␣
↪will all execute asynchronously,
    # but not in parallel. For more information on multitask behavior, refer to␣
↪the VeriStand help.
    with multitask() as mt:
        # You must decorate tasks using the following notation.
        # The following code shows example of a task.
        @task(mt)
        def engine_warmup():
            """Spawns a task to wait for the actual rpm signal to settle."""
            desired_rpm.value = 2500
            # Waits for up to 120 seconds for the actual RPM to be between␣
↪999999 and 2450 for 25 seconds.
            wait_until_settled(actual_rpm, 9999999, 2450, 25, 120)
            desired_rpm.value = 8000
```

(continues on next page)

(continued from previous page)

```
            wait_until_settled(actual_rpm, 9999999, 7800, 25, 120)
            warmup_complete.value = True

        @task(mt)
        def monitor_temp():
            """Spawns a task to monitor engine temperature.

            If the temperature rises above 110 degrees (C), the previous task will stop.
            """
            while warmup_complete.value is False:
                if engine_temp.value > 110:
                    stop_task(engine_warmup)
                    warmup_complete.value = True
                    warmup_succeeded.value = False
                nivs_yield()

    # You can use a return value, but some restrictions will apply.
    # For example, the function may only return previously declared variables.
    return warmup_succeeded.value


@nivs_rt_sequence
def run_engine_demo_advanced():
    """Run the engine_demo_advanced example.

    To handle a condition that stops a task (such as, the engine temperature rising above a safe value),
    use a try/finally block.

    Regardless of the result of the execution, the finally block can be used to safely shut down the engine.
    """
    try:
        warmup_succeeded = BooleanValue(False)
        engine_power = ChannelReference("Aliases/EnginePower")
        desired_rpm = ChannelReference("Aliases/DesiredRPM")
        actual_rpm = ChannelReference("Aliases/ActualRPM")
        engine_temp = ChannelReference("Aliases/EngineTemp")
        engine_power.value = True
        warmup_succeeded.value = engine_demo_advanced(desired_rpm, actual_rpm, engine_temp)
    finally:
        engine_power.value = False
        desired_rpm.value = 0
    return warmup_succeeded.value
```

(continues on next page)

(continued from previous page)

```


def run_deterministic():
    return run_py_as_rtseq(run_engine_demo_advanced)


def run_non_deterministic():
    return run_engine_demo_advanced()


if __name__ == "__main__":
    # Run the tests.
    # Note:  We expect the tests to fail because the engine temperature rises above 110 degrees (C),
    # but the cleanup code at the end turns the engine off.
    print("Non-Deterministic test:")
    print("Test Passed!" if run_non_deterministic() else "Test Failed (expected)!")
    print("Deterministic test:")
    print("Test Passed!" if run_deterministic() else "Test Failed (expected)!")
```

Test Engine Set Points

```
from niveristand import nivs_rt_sequence, NivsParam, run_py_as_rtseq
from niveristand.clientapi import BooleanValue, ChannelReference, DoubleValue, DoubleValueArray
from niveristand.library import localhost_wait, seqtime, wait_until_settled


""" This module contains a complex example for running multiple tests in sequence.

This example mirrors the 'Test Engine Setpoints' stimulus profile found in the examples that install with VeriStand.

Instead of using a stimulus profile to report results, this example uses the py.test
unit-testing framework that is commonly used for running Python tests.
"""


@nivs_rt_sequence
@NivsParam("on_off", BooleanValue(False), NivsParam.BY_VALUE)
def set_engine_power(on_off):
```

(continues on next page)

(continued from previous page)

```
    """Turns the engine on or off."""
    engine_power = ChannelReference("Aliases/EnginePower")
    engine_power.value = on_off.value


# If you do not specify a parameter decorator, the parameter defaults to the following:
# Type=DoubleValue
# Default Value = 0
# Passed by reference.
# In this case, the default is adequate, so you do not need to specify the decorator.
@nivs_rt_sequence
def measure_set_point_response(setpoint, timeout, tolerance):
    """Sets the desired rpm to the specified setpoint and wait until the signal settles.

    The tolerance is used to create upper and lower boundaries for the signal.
    Returns the amount of time it takes the signal to settle or timeout.
    """
    actual_rpm = ChannelReference("Aliases/ActualRPM")
    desired_rpm = ChannelReference("Aliases/DesiredRPM")
    start_time = DoubleValue(0)
    settle_time = DoubleValue(0)

    desired_rpm.value = setpoint.value
    # Waits .5 seconds, so the gateway has time to update.
    localhost_wait(0.5)

    start_time.value = seqtime()
    wait_until_settled(
        actual_rpm,
        desired_rpm.value + tolerance.value,
        desired_rpm.value - tolerance.value,
        DoubleValue(2.0),
        timeout.value,
    )
    settle_time.value = seqtime() - start_time.value
    return settle_time.value


@nivs_rt_sequence
def inbounds_check(test_value, upper, lower):
    """Returns True if lower <= value <= upper.

    Performs an inbounds check.
```

(continues on next page)

(continued from previous page)

```
    """
    result = BooleanValue(False)
    # Typically, you could write this instruction as lower.value <= test_value.value <= upper.value
    # because Python supports cascading operators. However, for real-time sequences,
    # you must write all comparisons using only two operands.
    result.value = test_value.value >= lower.value and test_value.value <= upper.value
    return result.value


# The following function runs the profile above deterministically.
# A unit test framework, such as py.test, can find the function.
def test_run_engine_set_points_profile_deterministic():
    set_engine_power(True)
    setpoints = DoubleValueArray([2500, 6000, 3000])
    try:
        for setpoint in setpoints:
            test_passed = run_py_as_rtseq(
                measure_set_point_response,
                {"setpoint": setpoint, "timeout": DoubleValue(60), "tolerance": DoubleValue(100)},
            )
            assert 0 < test_passed <= 60, "Setpoint %d failed" % setpoint
    finally:
        set_engine_power(False)


# If you do not need to run the profile deterministically, you can run this function as part of a py.test run.
def test_run_engine_set_points_python():
    set_engine_power(True)
    setpoints = DoubleValueArray([2500, 6000, 3000])
    try:
        for setpoint in setpoints:
            assert (
                0 < measure_set_point_response(setpoint, DoubleValue(60), DoubleValue(100)) <= 60
            ), ("Setpoint %d failed" % setpoint)
    finally:
        set_engine_power(False)
```

9.3 API Reference

9.3.1 Decorators

@niveristand.**nivs_rt_sequence**
Marks a function as an RT sequence. You must mark a function as *nivs_rt_sequence* to run it deterministically with *run_py_as_rtseq*.

class niveristand.**NivsParam**(*param_name*, *default_elem*, *by_value*)
Describes a parameter passed down to a function.

Parameters

- **param_name** (*str*) – Name of the parameter as it is found in the function definition.
- **default_elem** – Default value and type. Refer to *NI VeriStand Data types* for valid values.
- **by_value** (*bool*) – Specifies whether to pass a parameter by value or by reference. Set to True to pass by value. Set to False to pass by reference. Refer to NivsParam.BY_REF or *NivsParam.BY_VALUE* for details.

BY_VALUE = True
Passes a parameter by value. Creates a copy of the caller's value for use inside the function.

9.3.2 Client API

NI VeriStand Data types

niveristand.clientapi.**BooleanValue**
alias of <MagicMock id='140137930752720'>

niveristand.clientapi.**BooleanValueArray**
alias of <MagicMock id='140137930826384'>

niveristand.clientapi.**ChannelReference**
alias of <MagicMock id='140137930891024'>

niveristand.clientapi.**DoubleValue**
alias of <MagicMock id='140137930378128'>

niveristand.clientapi.**DoubleValueArray**
alias of <MagicMock id='140137930856976'>

niveristand.clientapi.**I32Value**
alias of <MagicMock id='140137930707472'>

niveristand.clientapi.**I32ValueArray**
alias of <MagicMock id='140137932439120'>

niveristand.clientapi.**I64Value**
alias of <MagicMock id='140137930995856'>

niveristand.clientapi.**I64ValueArray**
alias of <MagicMock id='140137931827728'>

niveristand.clientapi.**U32Value**
alias of <MagicMock id='140137930773776'>

niveristand.clientapi.**U32ValueArray**
alias of <MagicMock id='140137930477840'>

niveristand.clientapi.**U64Value**
alias of <MagicMock id='140137930534096'>

niveristand.clientapi.**U64ValueArray**
alias of <MagicMock id='140137930589648'>

niveristand.clientapi.**VectorChannelReference**
alias of <MagicMock id='140137930775696'>

niveristand.clientapi._datatypes.rtprimitives.**DataType**
alias of <MagicMock id='140137930582224'>

Real-Time Sequence APIs

class niveristand.clientapi.**RealTimeSequence**(*top_level_func*, *rtseq_pkg=None*)
A real-time sequence that can run on the VeriStand Engine.

Parameters

- **top_level_func** – the function to transform.
- **rtseq_pkg** (RealTimeSequencePackage) – the containing package in case you want to add this sequence to a library.

Raises

- *niveristand.errors.TranslateError* – if translation fails.
- *niveristand.errors.VeristandError* – if compilation fails.

run(*rtseq_params={}*)
Runs the sequence on the globally configured VeriStand Engine.

Parameters **rtseq_params** (*Dict[str, niveristand.clientapi._datatypes.rtprimitives.DoubleValue]*) – the parameters to be passed to the RT sequence.

Returns Stimulus profile session state.

Return type *niveristand.clientapi.stimulusprofileapi.StimulusProfileState*

Deploys and runs the sequence without waiting for the sequence to finish. Use the returned *StimulusProfileState* to wait for the sequence to complete and obtain the return value.

For a simpler use case, refer to *niveristand.realtimesequencetools.run_py_as_rtseq()*

save(*path=None*)
Saves this sequence to disk.

Parameters **path** (*Optional[str]*) – path to the location you want to save the sequence file.

Returns path you specify in **path**.

All dependencies required for deployment of this sequence save to the same path. If you do not specify a path in **path**, this sequence saves to the location where you last saved the object. If you did not previously save the object, it saves to a temporary folder.

For a simpler use case, refer to *niveristand.realtimesequencetools.save_py_as_rtseq()*

class niveristand.clientapi.**ErrorAction**(*value*)
Actions you can take when calling *niveristand.library.generate_error()*.

AbortSequence = 2
Stops execution and avoid calling try/finally blocks.

ContinueSequenceExecution = 0
Continues execution but still fails the test run.

StopSequence = 1
Stops execution and calls all try/finally blocks.

Stimulus Profile APIs

class niveristand.clientapi.**StimulusProfileState**(*session*)
Contains the execution state of a real-time sequence.

class CompletionState(*value*)
Enum used for possible completion states.

Aborted = 1
operation stopped forcefully.

Failed = 2
operation ran to completion, but an error occurred.

Success = 0
operation ran to completion successfully. No errors occurred.

property completion_state
Returns the state after running.

Returns state after the operation runs to completion. *None* if unfinished.

Return type *CompletionState*

property last_error
Returns the last error generated by the sequence.

Returns final error the sequence generated.

Return type *niveristand.errors.SequenceError*

property ret_val
Returns the return value of the sequence.

Returns the return value of the sequence.

Return type (bool, int, float)

property session
Returns the session you created to execute this sequence.

Returns a session connected to the VeriStand Engine.

wait_for_result()
Waits for the sequence to finish running.

Returns the value returned by the VeriStand Engine after this sequence runs.

9.3.3 Errors

class niveristand.errors.**VeristandError**
The base class for all VeriStandErrors.

Note: This class generates a *VeristandError* if a more specific error cannot be determined.

class niveristand.errors.**TranslateError**
Bases: *niveristand.errors.VeristandError*

Raised if a Python function fails to translate to a VeriStand real-time sequence.

class niveristand.errors.**SequenceError**(*error_code*, *message*, *error_action*)
Raised by *generate_error* to report a sequence failure.

property inner_error
Returns the error generated before the most recent error, if any, or *None*.

Returns the previous error generated by this sequence.

Return type *SequenceError*

Real-time sequences report only the last error the sequence generates. If you want to see a list of all the inner errors, use *RunError.get_all_errors*.

property is_fatal
Returns whether or not any error causes the sequence to stop.

Returns True if the error is *ErrorAction.AbortSequence* or *ErrorAction.StopSequence*, false if the error is *ErrorAction.ContinueSequenceExecution*.

Return type bool

property should_raise
Determines whether or not this error raises an exception.

Returns False if the error is *ErrorAction.ContinueSequenceExecution* with an error code of 0. Otherwise, this function returns True.

Return type bool

class niveristand.errors.**RunError**(*error*)
Bases: *niveristand.errors.VeristandError*

Raised at the end of execution if an RT sequence called *generate_error*.

get_all_errors()
Generates a list of all errors reported during execution.

Returns all errors generated during execution.

Return type List(*SequenceError*)

class niveristand.errors.**RunFailedError**(*error*)
Bases: *niveristand.errors.RunError*

Raised by *run_py_as_rtseq* to report that the sequence failed.

This error is raised when a real-time sequence executes successfully, but *generate_error* was called with *ErrorAction.ContinueSequenceExecution*.

class niveristand.errors.**RunAbortedError**(*error*)
Bases: *niveristand.errors.RunError*

Raised by *run_py_as_rtseq* to report that the sequence failed.

This error is raised when a real-time sequence executes successfully, but *generate_error* was called with *ErrorAction.StopSequence* or *ErrorAction.AbortSequence*.

9.3.4 Library

niveristand.library.**abstime**()
Returns the current date and time, in seconds, relative to the operating system's epoc.

Note: Only available for RT sequences.

niveristand.library.**arraysize**(*x*)
Returns the number of elements in x, where x is an array.

Parameters **x** – the array for which you want to get the number of elements.

Returns the size of the array. If **x** is not an array, this function returns 0.

Return type int

niveristand.library.**clearfault**(*x*)
Clears all faults set on channel x.

Parameters **x** – the channel you want to clear faults on.

Channel *x* must be a reference to a channel and should not be a reference to a local variable. If *channel* references a local variable, *clearfault()* performs no operation.

Note: Only available for RT sequences.

niveristand.library.**clearlasterror**()
Clears the last error set by *generate_error()*.

Note: Only available for RT sequences.

niveristand.library.**deltat**()
Returns the duration, in seconds, of the current system timestep.

To perform equality or comparison operations, use *deltatus*.

Note: In Python Mode, this function always returns *0.01* for a rate of 100Hz.

niveristand.library.**deltatus**()
Returns the duration, in microseconds, of the current system timestep.

Note: In Python Mode, this function always returns *10,000* for a rate of 100Hz.

niveristand.library.**fault**(*channel*, *value*)
Faults *channel* with *value*.

Parameters

- **channel** – channel to fault.
- **value** (*float*) – value to fault the channel.

channel must be a reference to a channel and should not be a reference to a local variable. If *channel* references a local variable, *fault()* performs no operation.

Note: Only available for RT sequences.

niveristand.library.**fix**(*x*)
Rounds x to the nearest integer between x and zero.

Parameters **x** (*float*) – value you want to round.

Returns floating-point representation of the rounded value.

Return type (float)

Note: Only available for RT sequences.

niveristand.library.**generate_error**(*code*, *message*, *action*)
Generates an error to report test failure.

Parameters

- **code** (*int*) – error code to display.
- **message** (*str*) – error string to display.
- **action** (*niveristand.clientapi.ErrorAction*) – action to perform.

Returns If action is Continue, returns the generated error.

niveristand.library.**getlasterror**()
Returns the numeric error code of the last error set by *generate_error()*.

Note: Only available for RT sequences.

niveristand.library.**iteration**()
Returns the number of iterations since the current top-level sequence started.

Returns iteration count.

Return type int

niveristand.library.**localhost_wait**(*amount=0.1*)
Waits for channel values to update.

Parameters **amount** (*float*) – time, in seconds, this function waits for channel values to update.

When running in the VeriStand Engine, this function is ignored as channels are always up to date.

niveristand.library.**multitask**()
Creates a multitask context for branching execution.

Refer to *niveristand.library.multitask()* for more details on branching execution.

niveristand.library.**nivs_yield**()
Yields execution from this task or block to the next.

Refer to *niveristand.library.multitask()* for more details on yielding to other tasks.

niveristand.library.**quotient**(*x*, *y*)
Returns floor(x/y), the number of times y evenly divides into x.

Parameters

- **x** – dividend.
- **y** – divisor.

Returns integer quotient of x/y

Return type int

niveristand.library.**recip**(*x*)
Returns 1/x.

Parameters **x** – divisor.

Note: Only available for RT sequences.

niveristand.library.**rand**(*x*)
Returns a random floating-point number between 0 and the maximum value.

Parameters **x** (*float*) – maximum value.

Returns random number between 0 and *x*

Return type float

niveristand.library.**rem**(*x*, *y*)
Returns the remainder of x/y, when the quotient is rounded to the nearest integer.

Parameters

- **x** (*float*) – dividend.

- **y** (*float*) – divisor.

niveristand.library.**seqtime**()
Returns the number of elapsed seconds since the epoch.

Returns time, in seconds, since the epoch.

Return type float

To perform equality or comparison operations, use *seqtimeus* instead.

niveristand.library.**seqtimeus**()
Returns the elapsed time, in microseconds, since the epoch.

Returns elapsed time, in microseconds, as reported by the system clock.

Return type int

niveristand.library.**stop_task**(*task_function*)
Stops the task you specify.

Parameters **task_function** – task function you want to stop. You must have previously declared the task function inside a *multitask()* context.

Refer to *niveristand.library.multitask()* for more details on stopping tasks.

niveristand.library.**task**(*mt*)
Marks a nested function-definition as a task inside a *niveristand.library.multitask()*.

Parameters **mt** – the parent *niveristand.library.multitask()*

Use this function as a decorator. Refer to *niveristand.library.multitask()* for more details on using tasks.

niveristand.library.**tickcountms**()
Returns the current value of the milliseconds counter.

Returns time, in milliseconds, as reported by the high-precision counter (if available).

Return type int

niveristand.library.**tickcountus**()
Returns the current value of the microseconds counter.

Returns time, in microseconds, as reported by the high-precision counter (if available).

Return type int

niveristand.library.**wait**(*duration*)
Waits the duration, in seconds, you specify.

Parameters **duration** (*DoubleValue*) – time, in seconds, this function waits. You may specify fractions of seconds.

Returns actual seconds waited.

Return type float

This wait is non-blocking, so other tasks will run while this wait executes.

niveristand.library.**wait_until_next_ms_multiple**(*ms_multiple*)
Waits until the next millisecond multiple of the number you specify in *ms_multiple*.

Parameters **ms_multiple** (*I64Value*) – the millisecond multiple to wait until.

Returns actual milliseconds waited.

Return type int

This wait is non-blocking, so other tasks will run while this wait executes.

niveristand.library.**wait_until_next_us_multiple**(*us_multiple*)
Waits until the next microsecond multiple of the number you specify in *us_multiple*.

Parameters **us_multiple** (*I64Value*) – the microsecond multiple to wait until.

Returns actual microseconds waited.

Return type int

This wait is non-blocking, so other tasks will run while this wait executes.

niveristand.library.**wait_until_settled**(*signal*, *upper_limit*, *lower_limit*, *settle_time*, *timeout*)
Waits until *signal* settles for the amount of time you specify in *settle_time*.

Parameters

- **signal** (*DoubleValue*) – value to monitor.
- **upper_limit** (*DoubleValue*) – maximum value of the settle range.
- **lower_limit** (*DoubleValue*) – minimum value of the settle range.
- **settle_time** (*DoubleValue*) – time, in seconds, *signal* must stay inside the settle range.
- **timeout** (*DoubleValue*) – seconds to wait before the function times out.

Returns True: The signal failed to settle before the operation timed out. False: The signal settled before the operation timed out.

Return type bool

This wait is non-blocking, so other tasks will run while this wait executes.

niveristand.library.**ramp**(*ramp_out*, *init_value*, *final_value*, *duration*)
Ramps a variable from an initial value to an ending value over the duration you specify.

Parameters

- **ramp_out** (*DoubleValue*) – variable you want to ramp.
- **init_value** (*DoubleValue*) – starting value.
- **final_value** (*DoubleValue*) – ending value.
- **duration** (*DoubleValue*) – time, in seconds, you want the ramp to take.

niveristand.library.**sine_wave**(*wave_out*, *amplitude*, *freq*, *phase*, *bias*, *duration*)
Plays a sine wave with the parameters you specify.

Parameters

- **wave_out** (*DoubleValue*) – variable onto which the sine wave plays.
- **amplitude** (*DoubleValue*) – amplitude of the sine wave.
- **freq** (*DoubleValue*) – frequency, in Hz, of the sine wave.
- **phase** (*DoubleValue*) – phase, in degrees, of the sine wave.
- **bias** (*DoubleValue*) – offset to add to the sine wave.
- **duration** (*DoubleValue*) – duration, in seconds, to play the sine wave.

niveristand.library.**square_wave**(*wave_out*, *amplitude*, *freq*, *phase*, *bias*, *duty_cycle*, *duration*)

Plays a square wave with the parameters you specify.

Parameters

- **wave_out** (*DoubleValue*) – variable onto which the square wave plays.
- **amplitude** (*DoubleValue*) – amplitude of the square wave.
- **freq** (*DoubleValue*) – frequency, in Hz, of the square wave.
- **phase** (*DoubleValue*) – phase, in degrees, of the square wave.
- **bias** (*DoubleValue*) – offset to add to the square wave.
- **duty_cycle** (*DoubleValue*) – percentage of time the square wave remains high versus low over one period.
- **duration** (*DoubleValue*) – time, in seconds, to play the square wave.

niveristand.library.**triangle_wave**(*wave_out*, *amplitude*, *freq*, *phase*, *bias*, *duration*)

Plays a triangle wave with the parameters you specify.

Parameters

- **wave_out** (*DoubleValue*) – variable onto which the triangle wave plays.
- **amplitude** (*DoubleValue*) – amplitude of the triangle wave.
- **freq** (*DoubleValue*) – frequency, in Hz, of the triangle wave.
- **phase** (*DoubleValue*) – phase, in degrees, of the triangle wave.
- **bias** (*DoubleValue*) – offset to add to the triangle wave.
- **duration** (*DoubleValue*) – duration, in seconds, to play the triangle wave.

niveristand.library.**uniform_white_noise_wave**(*wave_out*, *amplitude*, *seed*, *duration*)

Plays a uniform white noise wave with the parameters you specify.

Parameters

- **wave_out** (*DoubleValue*) – variable onto which the white noise wave plays.
- **amplitude** (*DoubleValue*) – amplitude of the white noise wave.
- **seed** (*I32Value*) – seed for random number generator.

- **duration** (*DoubleValue*) – duration, in seconds, to play the white noise wave.

9.3.5 Real-Time Sequence Tools

niveristand.realtimesequencetools.**run_py_as_rtseq**(*toplevelfunc*, *rtseq_params={}*)

Runs a Python function as an RT sequence in the VeriStand Engine.

Parameters

- **toplevelfunc** – the Python function to run.
- **rtseq_params** (*Dict[str, niveristand.clientapi._datatypes.rtprimitives.DoubleValue]*) – the parameters to be passed to the RT sequence.

Returns The numeric value returned by the real-time sequence execution.

Return type Union[float, None]

Raises

- ***TranslateError*** – if the function is not successfully translated.
- ***RunAbortedError*** – if this function calls *generate_error* with an action of Abort or Stop.
- ***RunFailedError*** – if this function calls *generate_error* with a Continue action.

niveristand.realtimesequencetools.**save_py_as_rtseq**(*toplevelfunc*, *dest_folder*)

Saves a Python function as an RT sequence that is compatible with the Stimulus Profile Editor.

Parameters

- **toplevelfunc** – the Python function you want to save.
- **dest_folder[str]** – the folder you want to save the sequence and all its dependencies in.

Returns The full path to the main sequence file.

Raises ***niveristand.errors.TranslateError*** – if the function is not successfully translated.

9.3.6 Legacy API

This module describes the functionality equivalent to the deprecated IronPython API.

class niveristand.legacy.NIVeriStand.**Workspace**

Interface that controls the running state of the system and accesses the channels in the system.

GetAliasList()

Acquires all the aliases of a system.

GetChannelVectorValues(*name*)
Acquires the vector value of the channel you specify.

GetEngineState()
Returns the current state of the system.

GetMultipleChannelValues(*names*)
Acquires values from the channels you specify.

GetMultipleSystemNodesData(*names*)
Acquires data from the nodes you specify.

GetSingleChannelValue(*name*)
Acquires the value of the channel you specify.

GetSystemNodeChannelList(*name*)
Acquires all channels of the node you specify.

If you want to acquire all the channels in the system, enter "" as the node name.

GetSystemNodeChildren(*name*)
Acquires a list of all the child nodes nested under the node you specify.

LockWorkspaceFile(*old_password*, *new_password*)
Locks the configuration that is currently running.

This function will only succeed if a configuration is currently running. If this configuration was locked previously, you must enter the previous password in *old_password*.

RunWorkspaceFile(*file*, *launchworkspacewindow*, *deploysystemdefinition*, *timeout*, *username*, *password*)
Runs the workspace configuration file you specify.

Raises an error if you call this function while a configuration is already running. You must stop all running configurations before you call this function. If this function times out, check to see if the deployment process took longer than expected and caused the operation to timeout. Use the GetEngineState function to check the status of the system.

SetChannelVectorValues(*name*, *values*)
Sets the starting parameter vector value for the channel you specify.

The value you specify in *values* must be a matrix data type.

SetMultipleChannelValues(*names*, *values*)
Sets the value(s) for the channels you specify.

SetSingleChannelValue(*name*, *value*)
Sets the value for the channel you specify.

StopWorkspaceFile(*password*)
Stops the execution of the currently running configuration.

UnlockWorkspaceFile(*password*)
Unlocks the currently running configuration.

class niveristand.legacy.NIVeriStand.**Workspace2**(*gatewayIPAddress=None*)
Interface that controls the running state of the system and accesses the channels in the system.

ConnectToSystem(*systemdefinition_file*, *deploy*, *timeout*)
Connects the VeriStand Gateway to one or more targets running on the System Definition file you specify.

DisconnectFromSystem(*password*, *undeploy_system_definition*)
Disconnects the VeriStand Gateway from the targets.

GetSystemState()
Returns the current state of the system.

LockConnection(*old_password*, *new_password*)
Locks the current VeriStand Gateway connection.

If the connection was locked previously, you must enter the previous password in *old_password*.

ReconnectToSystem(*target*, *deploy*, *calibration_file*, *timeout*)
Reconnects the VeriStand Gateway to a target within the system definition file used by the Gateway. You can also redeploy the system definition file.

SetChannelValues(*channels*, *newValues*)
Sets the value for the channels you specify.

The *newValues* parameter accepts scalar, vector, and matrix data types.

StartDataLogging(*configuration_name*, *logInfo*)
Starts logging data to the configuration you specify.

StopDataLogging(*configuration_name*)
Terminates data logging for the configuration you specify.

UnlockConnection(*password*)
Unlocks the current VeriStand Gateway connection.

class niveristand.legacy.NIVeriStand.**PyAlarmPriority**
Priority of an alarm.

class niveristand.legacy.NIVeriStand.**PyAlarmState**
State of an alarm in the engine.

class niveristand.legacy.NIVeriStand.**PyAlarmMode**
Specifies the mode of an alarm when triggered.

Normal mode - triggers the alarm and runs the associated script. Indicate mode - only triggers the alarm.

class niveristand.legacy.NIVeriStand.**Alarm**(*name*, *target=None*, *gatewayIPAddress=None*)
Interface that queries information on a configured alarm.

GetAlarmData(*timeout*)
Acquires the alarm data.

SetAlarmData(*alarmDict*)
Sets the alarm data.

DEPRECATED function. This function does not support the Priority Number field. Use SetAlarmData2() instead.

SetAlarmData2(*alarmDict*)
Modifies an alarm in the system.

SetAlarmMode(*mode*)
Changes the mode of this alarm. See PyAlarmMode for possible values.

SetEnabledState(*enabled*)
Enables or disables the current alarm.

class niveristand.legacy.NIVeriStand.**AlarmManager**
Interface that acquires information on the state of the server alarm.

GetAlarmList()
Acquires a list of names of all the alarms configured in the system.

GetAlarmsStatus()
Acquires a list of alarms organized by status (high, medium, and low).

GetMultipleAlarmsData(*alarms*, *timeout*)
Acquires information about a list of alarms.

class niveristand.legacy.NIVeriStand.**AlarmManager2**(*gateway_ip_address=None*)
Interface that acquires information on the state of the server alarm.

GetAlarmList(*target*)
Acquires a list of names of all the alarms configured in the system.

GetAlarmsStatus(*target*)
Acquires a list of alarms organized by status (high, medium, and low).

GetMultipleAlarmsData(*target*, *alarms*, *timeout*)
Acquires information about a list of alarms.

class niveristand.legacy.NIVeriStand.**PyModelState**
Represents the state of a model.

class niveristand.legacy.NIVeriStand.**PyModelCommand**
Changes the state of the model.

class niveristand.legacy.NIVeriStand.**Model**(*name*, *target=None*, *gatewayIPAddress=None*)
Interface that acquires information on a specific model running on the system.

GetModelExecutionState()
Acquires the execution time and state of the model.

RestoreModelState(*filepath*)
Restores the state of a model running on the target. Specify the path to the model file in *filepath*.

SaveModelState(*filepath*)
Saves the current model state to the path on the target you specify in *filepath*.

SetModelExecutionState(*command*)
Changes the current state of the model on the server.

This is a request operation on the server. Even if this function executes successfully, the model state may remain unchanged in some cases. See PyModelState for command values.

class niveristand.legacy.NIVeriStand.**ModelManager**

Interface that queries information on the models configured in the system.

GetModelList()

Returns a list of all models configured in the system.

GetMultipleParameterValues(*names*)

Acquires the value(s) of the parameters you specify.

GetParameterVectorValues(*name*)

Acquires the vector values of the parameter you specify.

GetParametersList()

Returns a list of all parameters in the system.

GetSingleParameterValue(*name*)

Acquires the value of the parameter you specify.

SetMultipleParameterValues(*names*, *values*)

Sets the value(s) of the parameters you specify.

SetParameterVectorValues(*name*, *values*)

Sets a vector value for a parameter.

The value you specify in *value* must be a matrix data type.

SetSingleParameterValue(*name*, *value*)

Sets the value of the parameter you specify.

class niveristand.legacy.NIVeriStand.**ModelManager2**(*gateway_ip_address=None*)

Interface that queries information on the models configured in the system.

GetModelList(*target*)

Returns a list of models on the target you specify.

GetMultipleParameterValues(*target*, *names*)

Acquires the value(s) of the parameters you specify.

GetParameterVectorValues(*target*, *name*)

Acquires the vector values of the parameter you specify.

GetParametersList(*target*)

Returns a list of all parameters in the target you specify.

GetSingleParameterValue(*target*, *name*)

Acquires the value of the parameter you specify.

SetMultipleParameterValues(*target*, *names*, *values*)

Sets the value of the parameter(s) you specify.

SetParameterValues(*target*, *names*, *matrixArr*)

Sets the vector value of the parameter(s) you specify.

The value you specify in *matrixArr* must be a matrix data type. Sample usage ModelManager2.SetParameterValues("target1",["1By3Param","2By3Param"],[[[1,2,3]],[[1,2,3],[4,5,6]]])

SetParameterVectorValues(*target*, *name*, *values*)
Set a parameter vector values.

Values are expected to be a matrix type.

SetSingleParameterValue(*target*, *name*, *value*)
Sets the value of the parameter you specify.

UpdateParametersFromFile(*target*, *parameterfiles*)
Update a set of parameters specified in the parameter files.

class niveristand.legacy.NIVeriStand.**ChannelFaultManager**(*gatewayIPAddress=None*)
Interface that institutes software value forcing on the system.

ClearAllFaults()
Clears all faults.

ClearFault(*name*)
Removes the channel you specify from the fault list.

GetFaultList()
Acquires a list of all currently faulted channels.

GetFaultValue(*name*)
Acquires the fault value of a channel you specify.

SetFaultValue(*name*, *value*)
Sets the fault value of the channel you specify.

class niveristand.legacy.NIVeriStand.**PyStimulusState**
Represents the state of the stimulus generation server.

class niveristand.legacy.NIVeriStand.**PyStimulusResult**
Represents the result of the stimulus generation.

class niveristand.legacy.NIVeriStand.**Stimulus**

GetStimulusProfileFile()
Acquires the current stimulus definition file.

GetStimulusProfileManagerState()
Returns the state of the stimulus generation component.

GetStimulusProfileResult()
Acquires the result of stimulus generation test.

Only the table test produces a test file of the result.

ReserveStimulusProfileManager()
Creates a task that reserves the stimulus generation server.

This task prevents other clients from interrupting the stimulus generation process.

RunStimulusProfile(*testfile*, *baselogpath*, *timeout*, *autostart*, *stopondisconnect*)
Starts the stimulus generation you defined in the test file.

StopStimulusProfile()
Stops the stimulus generation.

UnreserveStimulusProfileManager()
Destroys the task that reserves the stimulus generation server. Frees the server for other clients to use.

class niveristand.legacy.NIVeriStand.**Stimulus2**(*gatewayIPAddress=None*)
Automates the execution of stimulus profiles.

RunStimulusProfile(*testfile*, *baselogpath*, *timeout*, *autostart*, *stopondisconnect*, *parameterfiles=()*)
Starts the stimulus generation you defined in the test file.

class niveristand.legacy.NIVeriStand.**MacroRecorder**

class niveristand.legacy.NIVeriStand.**PyMacroPlayerState**
Represents the state of the macro player.

class niveristand.legacy.NIVeriStand.**PyMacroPlayerMode**
Represents the replay mode of the macro player.

class niveristand.legacy.NIVeriStand.**MacroPlayer**(*gatewayIPAddress=None*)

LoadMacro(*file*)
Loads a workspace macro.

PlayMacro(*mode*)
Replays the loaded macro.

PlayState()
Acquires the current play state.

9.4 Restrictions

The following section contains a list of all restrictions inside a function using the *nivs_rt_sequence* decorator. If you violate any of the following rules, a *TranslateError* occurs.

9.4.1 Assignment

- To assign values to an existing variable, you must use the *value* property of the object.

```
int_var = I32Value(0)
int_array_var = I32ValueArray([1, 2, 3])
# Invalid value assignments
int_var = 5
int_array_var = [2, 3, 4]
int_array_var[2] = 5
# Valid value assignments
```

(continues on next page)

(continued from previous page)

```
int_var.value = 5
int_array_var.value = [2, 3, 4]
int_array_var[2].value = 5
```

- Redefining a variable is not allowed.

```
int_var = I32Value(0)
int_var = DoubleValue(1.0) # The variable is already defined.
```

9.4.2 Conditional

- *If* statements only allow for boolean checks. You cannot use numbers, numeric data type declarations, or numeric variables inside *If* statements.

```
# Invalid conditions
int_var = I32Value(0)
if 1:
if I32Value(0):
if int_var.value:
# Valid conditions
bool_var = BooleanValue(True)
if True:
if False:
if BooleanValue(True):
if bool_var.value
```

- You cannot use numeric constants or data type declarations inside *If* expressions.

```
# Invalid conditions
int_var = I32Value(0)
int_var.value = 1 if 1 else 2
int_var.value = 1 if DoubleValue(1) else 2
int_var.value = 1 if int_var.value else 2
# Valid conditions
bool_var = BooleanValue(True)
int_var.value = 1 if True else 2
int_var.value = 1 if BooleanValue(True) else 2
int_var.value = 1 if bool_var.value else 2
```

9.4.3 Data Types

- Vector channel references will only work when you run sequences deterministically.
- **Channel references are the only data type declarations you can initialize with strings. All other data type decla**
  - Note: The BooleanValue data type is an exception to this rule. You can initialize BooleanValue with 'true' 'false' 'True' and 'False'.

```
# Invalid variable declarations
bool_var = BooleanValue("string")
double_var = DoubleValue("3.0")
int32_var = I32Value("1")
int64_var = I64Value("1")
uint32_var = U32Value("1")
uint64_var = U64Value("1")
bool_array_var = BooleanValueArray([True, "False"])
double_array_var = DoubleValueArray([3.0, 5.0, "6.0"])
int32_array_var = I32ValueArray([1, 2, "3"])
int64_array_var = I64ValueArray([1, 2, "3"])
uint32_array_var = U32ValueArray([1, 2, "3"])
uint64_array_var = U64ValueArray([1, 2, "3"])
```

- Signed integers cannot use the full range of values.

```
int32_invalid_var = I32Value(0xFFFFFFFF)
int32_last_valid_var = I32Value(0x7FFFFFFF)
int64_invalid_var = I64Value(0xFFFFFFFFFFFFFFFF)
int64_last_valid_var = I64Value(0x7FFFFFFFFFFFFFFF)
```

9.4.4 Error Generation

- When you generate an error, you can only use integer constants for the error code parameter, strings for the error message parameter, and ErrorAction members as the error action parameter.

```
# Valid statement
generate_error(-1, "My error", ErrorAction.AbortSequence)
# Invalid statements
int_var = I32Value(-1)
generate_error(int_var.value, "My error", ErrorAction.AbortSequence)
generate_error(-1, 2, ErrorAction.AbortSequence)
generate_error(-1, "My error", 3)
```

9.4.5 Functions

Built-in Math Functions

- You cannot pass down an NI VeriStand data type directly as a parameter of the built-in math functions. As an alternative, you can pass a variable or data type declaration to these functions using the *value* property.

```
int_var = I32Value(-1)
# Invalid usage
int_var.value = abs(I32Value(-1))
# Valid usages
int_var.value = abs(I32Value(-1).value)
int_var.value = abs(int_var.value)
```

- BooleanValue for *abs* behaves differently between Python and Stimulus Profile Editor.

```
bool_var = BooleanValue(-5)
bool_var.value = abs(bool_var.value)
return bool_var.value # This returns False in the Stimulus Profile␣
↪Editor but returns True in Python.
```

Built-in VeriStand Functions

- Some of these functions are not implemented in Python. Please refer to *Library* for more information.

9.4.6 Function Definitions

- You cannot define new functions inside an *if* block, a loop, or a task.
- The **args* and *kwargs* variables are not supported.

9.4.7 Loops

- ***For Loops* do not support:**
  - *else* blocks
  - ranges with a start value
  - ranges with a step value
  - ranges that use a channel reference
  - ranges that use array constants

```
# The following statements are invalid:
for x in range(5):
    pass
else:
    pass
for x in range(2, 5):
for x in range(2, 5, 2):
channel_ref = ChannelReference('Aliases/DesiredRPM')
for x in range(channel_ref.value):
for x in [1, 2, 3]:
```

- ***While Loops* do not support:**
  - using *else* blocks
  - using a numeric constant as the condition
  - using *break* statements

```
# The following statements are invalid:
int_var = I32Value(5)
while 1:
while int_var:
while int_var.value:
while True:
    pass
else:
    pass
while True:
    break
```

9.4.8 Operators

Add

- You cannot use several pluses one after another. Always use one plus sign. If you violate this rule, a *TranslateError* occurs.

```
int_var = I32Value(0)
int_var.value = 1 +++ 2 # This is not supported.
int_var.value = 1 + 2 # Always use a single plus.
```

Arithmetic Shift

- You cannot use double data types to the left of an arithmetic shift operation in Python.

```
double_var = DoubleValue(5.0)
# The following statements only work when the code is run␣
↪deterministically.
double_var.value = DoubleValue(3.0) << 5
double_var.value = 3.0 >> 5
double_var.value = double_var.value >> 5
```

- You cannot use double or boolean data types to the right of an arithmetic shift operation.

```
bool_var = BooleanValue(True)
bool_var.value = 5 >> BooleanValue(True)
bool_var.value = 5 << True
bool_var.value = 5 << bool_var.value
double_var = DoubleValue(5.0)
double_var.value = 5 >> DoubleValue(3.0)
double_var.value = 5 << 3.0
double_var.value = 5 << double_var.value
```

- You cannot use a negative number to the right of an arithmetic shift operation. As an alternative, use the opposite operation with a positive value.

```
int_var = I32Value(1)
int_var.value = int_var.value >> -2 # This is not allowed.
int_var.value = int_var.value << 2 # Use this instead.
```

Bitwise Operators

- You cannot use bitwise operations on float or boolean values in Python.

```
bool_var = BooleanValue(False)
double_var = DoubleValue(1.0)
# The following statements only work when the code is run␣
↪deterministically.
bool_var.value = BooleanValue(True) & BooleanValue(True)
double_var.value = 3.5 | 2.5
double_var.value = DoubleValue(3.5) ^ DoubleValue(2.5)
```

Comparison Operators

- You cannot use cascading comparison operators. Only use one comparison operator at a time.

```
int_var = I32Value(0)
int_var.value = 1 == 2 == 3 == 4 # This is not allowed.
```

Logical Operators

- Logical operators only accept boolean values.
- You cannot use cascading logical operators. Only use one logical operator at a time.

Unary Invert

- The unary inversion operator (~) only accepts integer values.

9.4.9 Parameters

- If you need to pass an immutable object (such as the *value* property of an NI VeriStand data type) by reference, you must run your code deterministically. Otherwise, the parameter will not actually pass by reference when you run the code in Python.

```
@NivsParam('param', DoubleValue(0), NivsParam.BY_REF)
@nivs_rt_sequence
def _increment_by_ref(param):
    param.value += 1
    return param.value


@nivs_rt_sequence
def call_increment_by_ref():
    int_var = I32Value(1)
    _increment_by_ref(int_var.value)
    return int_var.value # This will return 1 in Python and 2 in␣
↪the Stimulus Profile Editor.
```

9.4.10 Return Statements

- A function can only have a single return statement and it has to be the last line of the function.
- You cannot use return statements inside an *if* block, a *try* block, a *finally* block, a loop, a multitask, or a task.
- Return statements can only return scalar values through the *value* property.

```
int_var = I32Value(1)
int_array_var = I32ValueArray([1, 2, 3])
# Invalid return statements
return int_var
return int_array_var
return DoubleValueArray[1.0, 2.0]
# Valid return statements
return int_var.value
return int_array_var[0].value
```

9.4.11 Tasks

- You cannot create more than one task with the same name.

```
with multitask() as mt:
    @task(mt)
    def f1():
        pass
    @task(mt)
    def f1(): # Task with the same name already exists.
        pass
```

- You cannot create parameters for tasks or multitasks.

```
with multitask(param) as mt: # Parameter not allowed.
    @task(mt)
    def f1(param_task): # Parameter not allowed.
```

9.4.12 Try

- *Try* is only allowed to be the first statement of a function.
- **You cannot use a *try* statement within:**
  - another *try* statement
  - an *if* block
  - an *else* block

  - a loop
  - a task
  - a multitask

- You cannot use a *try* statement with *except* or *orelse*.

9.4.13 Yield

- You cannot use *yield* as an operator or parameter.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

I

L

M

N

P

Q

R

S

T

U

V

W