



# spacely

Spacely Docs v1.1 – 2/20/2024

Adam Quinn

# Table of Contents

|  |    |
|--|----|
| Table of Contents .....  | 2  |
| Using this Document .....  | 4  |
| Installing Spacely .....   | 5  |
| Step 1: Install Python 3.11 .....                                | 5  |
| Step 2: Install Git .....  | 5  |
| Step 3: Clone Spacely Repos .....                                | 5  |
| Step 4: Library Installation .....                               | 7  |
| Optional: Create a Taskbar Shortcut to Run Spacely .....         | 8  |
| Spacely Fundamentals .....                                       | 9  |
| Setting up a new ASIC .....                                      | 9  |
| Master_Config.py .....   | 9  |
| Configuring Test Equipment: ASIC_Config.py .....                 | 10 |
| Writing Your Tests: ASIC_Routines.py .....                       | 14 |
| Writing Routines that can be Run from Spacely Command Line ..... | 14 |
| An Example Spacely Routine .....                                 | 14 |
| Iospec .....   | 17 |
| Spacely Quick Reference .....                                    | 18 |
| Command Line Options .....                                       | 18 |
| Spacely Commands .....   | 18 |
| Spacely Idioms .....   | 18 |
| Using NI-FPGAs with Spacely (Glue) .....                         | 19 |
| The Glue Wave Format .....                                       | 20 |
| Current Hardware Configurations Supported by Glue .....          | 21 |
| Working with Glue Waves .....                                    | 22 |
| Read A Glue Wave into Spacely from File .....                    | 22 |
| Plot a Glue Wave (Uses Matplotlib) .....                         | 22 |
| Write a Glue to File .....                                       | 22 |
| Run a Glue Wave on the ASIC .....                                | 23 |
| Extract One Particular Signal from a Glue Wave .....             | 23 |
| Glue Waves Which Run Concurrently on Multiple FIFOs .....        | 23 |
| Python2FPGA Flow .....   | 24 |

|  |    |
|--|----|
| ASCII2FPGA Flow.....                                       | 25 |
| RTL2FPGA Flow.....   | 26 |
| Exporting a VCD file from the Verification testbench ..... | 26 |
| Converting VCD → Glue Wave and Running the Test.....       | 27 |
| Spacely Instrument Libraries.....                          | 29 |
| AWG Library.....   | 29 |
| Oscilloscope Library.....                                  | 30 |
| Source_Port Library (Voltage & Current Bias).....          | 32 |
| Spacely Experiments.....                                   | 34 |
| Spacely + Caribou.....                                     | 36 |

# Using this Document

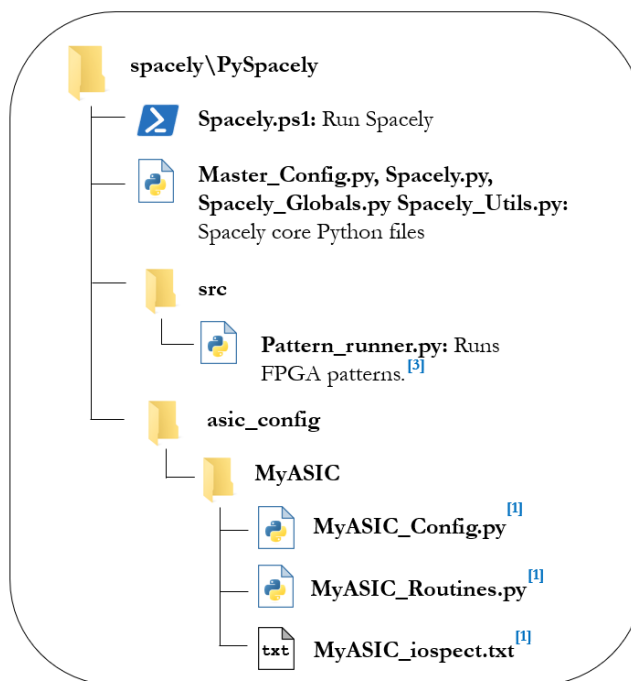
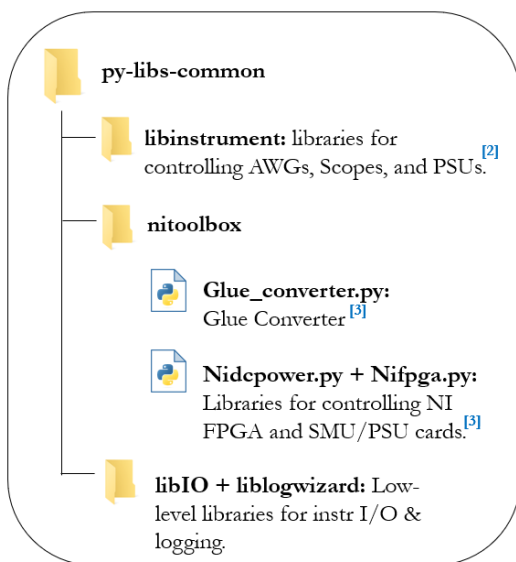
Welcome to Spacely! This document is intended to help you test ASICs with Spacely by creating and running Python test routines.

You do not need to read this document from start to finish.

For most users, the following flow is recommended:

1. Review the **Map of Spacely** below.
2. Read and understand the **Spacely Fundamentals** chapter.
3. Stop reading. Boot up Spacely and experiment with setting up the basic voltage and current rails you need for your chip. Use the **Spacely Quick Reference**.
4. When you're ready to run FPGA patterns on your ASIC, decide whether you're going to generate those patterns from an RTL testbench (RTL2FPGA Flow) or from a Python script (Python2FPGA Flow). Read through that section in the **Using NI-FPGAs with Spacely (Glue)** chapter.
5. Refer to other chapters only as needed.

## Map of spacely



Documentation Chapter References:

<sup>[1]</sup> Spacely Fundamentals

<sup>[2]</sup> Spacely Instrument Libraries

<sup>[3]</sup> Using NI-FPGAs with Spacely (Glue)

# Installing Spacely

## Step 1: Install Python 3.11

[Python Release Python 3.11.4 | Python.org](#)

WARNING: Make sure that you are using 64-bit Python (not 32-bit),

If necessary, **edit the system PATH** to point to the correct version of Python.

## Step 2: Install Git

| WINDOWS  | LINUX  |
|--|--|
| <p>Download Github Desktop <a href="#">GitHub Desktop   Simple collaboration from your desktop</a></p> | <p>Git is already installed. You just need to authenticate to Github. Recommend to use an SSH key for authentication.</p> <p><a href="https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent">https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent</a></p> <p>If you are connecting to a new repo using “git remote add”, make sure to always add the SSH version of the repo name, which should start with “git@github.com” rather than the HTTPS version of the name, which starts with “https”. This will allow you to authenticate using SSH key only rather than having to enter username + pwd every time.</p> |

## Step 3: Clone Spacely Repos

Clone the “spacely” repo from <https://github.com/Fermilab-Microelectronics>. Doing so will also clone the “spacely-asic-config” repo which is a submodule of Spacely.

| WINDOWS   | LINUX   |
|---|---|
| <p>Add spacely-asic-config to your Github Desktop by selecting File &gt; Add Local Repository... and navigating to spacely-asic-config.</p> | <p>Use <i>git submodule add</i> to add spacely-asic-config as a submodule of the spacely repo.</p> <p>The “spacely-asic-config” folder/repo will appear in “/PySpacely” but might be empty. If this is the case, navigate into that folder and run the command <i>git submodule update --init --recursive</i></p> |

|  |   |
|--|---|
|  | to update this submodule and pull the latest version on the server. |
|--|---|

**If you are a developer:** Also clone “py-libs-common” repo.

## Step 4: Library Installation

### NI Driver Installation

Using NI Package Manager, ensure that up-to-date versions of the following drivers are installed:

- FlexRIO
- NI-DCPower

### Python Library Installation

| WINDOWS  | LINUX  |
|--|--|
| <ol style="list-style-type: none"><li>1. Install Windows Terminal from the Microsoft App store for the best experience using Spacely.</li><li>2. Navigate to /spacely/PySpacely</li><li>3. Run <code>.\SetupWindows.ps1</code></li><li>4. If this results in an error w/ permissions, retry after running the command:<ol style="list-style-type: none"><li>a. <code>Set-ExecutionPolicy - ExecutionPolicy RemoteSigned -Scope CurrentUser</code></li></ol></li><li>5. If there are any libraries which are still not installed after running SetupWindows, report to developers. As a temporary measure, you can use “<code>pip install &lt;library&gt;</code>” inside the venv</li></ol> | <p>Use pip to install all the libraries listed in requirements.txt</p> |

**If you are a developer:** Create an editable installation of py-libs-common by running “`pip install -e <directory>`” for each of the following subdirectories of py-libs-common:

- /py-libs-common/libinstrument/
- /py-libs-common/liblogwizard/
- /py-libs-common/libIO/
- /py-libs-common/nitoolbox/

If you ever need to reinstall a particular python library from requirements.txt, use this command:

**`pip install --force-reinstall --no-deps <library>`**

## Optional: Create a Taskbar Shortcut to Run Spacely

Windows:

- (1) RMB > Create New Shortcut on your desktop.
- (2) For the shortcut path, enter the following, replacing the path to Spacely with the path to your install.

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -ExecutionPolicy Bypass -  
NoExit -File "C:\Users\aquinn\OnpieDrive - Fermi National Accelerator  
Laboratory\Resources\Software\spacely\PySpacely\Spacely.ps1"
```

- (3) From Windows Terminal settings, set Windows Terminal as the default terminal application.
- (4) pwd(Just for fun) change the icon of your shortcut to *spacely\_icon.ico* from PROJECTS/CAD/Spacely.
- (5) RMB on your new shortcut > Pin to task bar.



# Spacely Fundamentals

## Setting up a new ASIC

- (1) Create a new subdirectory for your ASIC in the /PySpacely/asic\_config/ folder:
  - a. Ex: /PySpacely/asic\_config/MyASIC/
- (2) Create ASIC-specific configuration files (see following sections):
  - a. **MyASIC\_Config.py**: Holds ASIC-specific settings like voltage levels, current limits, etc.
  - b. **MyASIC\_Routines.py**: Script for various test routines specific to this ASIC.
  - c. **MyASIC\_iospec.txt**: Defines the I/O specifications for the ASIC.
- (3) Edit **Master\_Config.py** to target MyASIC (see following section)

**NOTE:** Following the naming scheme above is required, as this enables Spacely to automatically load your configuration files based on your ASIC name.

**REMINDER: Whenever you update the ASIC-specific configuration files (or any Spacely source code), you must restart Spacely for the changes to take effect.**

## Master\_Config.py

- This script serves as the **primary configuration script** for the PySpacely environment. It's responsible for setting **global software settings** and dynamic loading of ASIC-specific configurations.
- It imports ASIC-specific modules (like routines and configurations) at runtime, allowing flexible and modular control over various ASICs.
- It allows easily switching between different ASIC targets by changing the TARGET variable.

In this file, ONLY modify the following variables:

- TARGET (= "MyASIC" to target MyASIC)
- VERBOSE (= True or False)

# Configuring Test Equipment: ASIC\_Config.py

This script provides ASIC-specific configuration information, detailing the setup of test equipment, voltage and current rails.

- **USE\_NI Flag:**  
This flag instructs Spacely that an NI FPGA chassis is used for this ASIC, which should be initialized when you start Spacely.

```
USE_NI = True
```

- **Instrument Initialization Sequence and Configuration:**

INSTR: This dictionary defines which test instruments (other than the FPGA) are used by Spacely, and gives key properties for each instrument (type, slot, IO, resource, etc.).

```
INSTR = {"SMU_A" : {"type" : "NIDCPower",
                    "slot" : "PXI1Slot2"},
         "SMU_B" : {"type" : "NIDCPower",
                    "slot" : "PXI1Slot3"},
         "PSU_A" : {"type" : "NIDCPower",
                    "slot" : "PXI1Slot7"},
         "PSU_B" : {"type" : "NIDCPower",
                    "slot" : "PXI1Slot8"},
         "Scope" : {"type" : "Oscilloscope",
                    "io"   : "VISA",
                    "resource" : "USB0::0x0957::0x1745::MY48080042::INSTR"},
         "AWG"   : {"type" : "AWG",
                    "io"   : "VISA",
                    "resource" : "GPIB1::10::INSTR"},
         "PSU_C" : {"type" : "Supply",
                    "io"   : "Prologix",
                    "ipaddr" : "192.168.1.15",
                    "gpibaddr" : 5}
        }
```

The following properties are supported for instruments. On startup, a lint checker will examine the INSTR dictionary and (in most cases) flag inappropriate values of these settings.

- "type"

**Description:** Specifies the kind of instrument.

**Acceptable Values:** Must match one of the instrument types in the system. "NIDCPower", "Oscilloscope", "AWG", and "Supply".

The exact list of acceptable types can be found or edit in `instr_type_required_fields.keys()` within the `Spacely_Utils.py`.

- "slot"

**Description:** Indicates the physical slot on the PXI where the instrument is installed.

**Acceptable Values:** A string indicating the chassis and slot number, e.g., "PXI1Slot2".

- "io"

**Description:** Defines the method of communication with the instrument.

**Acceptable Values:** Must be one of the recognized I/O types. Possible options are

"VISA" for instruments that communicate over the VISA interface, "Prologix" for instruments that communicate using Prologix GPIB to Ethernet converters. The valid io types are listed in `io_required_fields.keys()`.

- "resource"

**Description:** For instruments using the VISA I/O type, specifies the VISA resource string.

**Acceptable Values:** A string that uniquely identifies the VISA instrument, e.g., "USB0::0x0957::0x1745::MY48080042::INSTR".

- "ipaddr" and "gpibaddr"

**Description:** For instruments using the Prologix adapter, ipaddr specifies the IP address of the Prologix adapter, and gpibaddr specifies the GPIB address of the instrument.

**Acceptable Values:** ipaddr is a string representing the IP address, e.g., "192.168.1.15". gpibaddr is an integer representing the GPIB address, e.g., 5.

- **FPGA Configuration:** The script allows choosing specific FPGA bitfiles for different NI slots. The bitfile is linked to the FPGA's I/O hardware and clock speed. To support new combinations of I/O hardware and clock speed, new bitfiles must be generated in LabView. See the section **Current Hardware Configurations Supported by Glue** on page 21

```
DEFAULT_FPGA_BITFILE_MAP = {"PXI1Slot4": "NI7972_NI6583_40MHz"}
```

**WARNING:** The slot name is an arbitrary string which may not always follow the "PXI{x}Slot{y}" convention. Use NI MAX to double-check the slot name of the FPGA you are targeting.

- I/O Specification:

DEFAULT\_IOSPEC: Path to the I/O specification file. Typically, this should point to the iospec file in the same directory as the Config file.

```
DEFAULT_IOSPEC = ".\\asic_config\\SPROCKET2\\sprocket2_iospec.txt"
```

- **Voltage and Current Configuration:** These variables configure voltage and current bias rails, including their initialization sequence, levels, and limits.

#### Voltage Bias Configuration:

V\_SEQUENCE: This is a list that defines the order in which voltage supplies are initialized and applied. Each voltage rail should have a unique string name; for example: "VDDIO\_LT", "VCC\_LT", "VDD\_ASIC", "VCC\_ASIC", "VDDA", and "Vref\_adc"

```
V_SEQUENCE = [ "VCC_LT", "VDDIO_LT", "Vref_adc", "Vdd12", "VDD_ASIC", "vdda", "VCC_ASIC", "Vref_fe", "Ibdig"]
```

V\_INSTR: This dictionary maps voltage names to their corresponding instrument identifiers. Each key-value pair should specify the voltage name and the instrument assigned to control it, for example, "vdda": "SMU\_A". The instrument identifiers should match those defined in the INSTR dictionary.

V\_CHAN: It specifies the channel numbers on the instruments used for each voltage source. For example, if vdda is supplied by channel 0 on SMU\_A, the entry might be "vdda": 0.

V\_LEVEL: This dictionary defines the target voltage levels for each channel. Values should be set according to the operational voltage levels required by the ASIC. For example, "vdda": 2.5 indicates that the vdda line should be set to 2.5 volts.

V\_WARN\_VOLTAGE: Specifies the warning voltage ranges for each supply to monitor for levels outside expected ranges, helping identify potential issues early. "vdda": [2.0, 3.0] indicates that Spacely should print a warning if the vdda rail falls below 2.0 Volts or above 3.0 Volts, perhaps due to abnormal loading.

V\_CURR\_LIMIT: Defines the current limits for each voltage source in amperes to prevent damage to the ASIC or test setup.

V\_PORT: Used to hold references to the instrument and channel objects for each voltage source, allowing for easy control and adjustments during tests. As a default, it is initialized to None and set during the test initialization phase.

#### Current Bias Configuration:

**I\_SEQUENCE:** Similar to **V\_SEQUENCE**, it defines the order in which current sources are initialized and applied. The acceptable values would be the names of the current channels.

**I\_INSTR, I\_CHAN, I\_LEVEL, I\_PORT:** These fields are similar to the voltage configuration fields but are specific to current sources. They define the mapping to instruments, channel numbers, target current levels, and hold instrument object references for current sources, respectively.

**I\_VOLT\_LIMIT:** Sets the voltage limit across the current source to protect the ASIC.

**I\_WARN\_VOLTAGE:** Specifies warning voltage ranges for current monitoring, ensuring that current sources operate within safe parameters.

# Writing Your Tests: ASIC\_Routines.py

When testing an ASIC, we generally have a set of specific tasks or measurements that we want to perform. Tasks can be simple, such as writing a few bits to a scan chain, or complex and multi-step, such as sweeping an input voltage across a range of values and plotting a histogram of ASIC responses for each value.

In Spacely, all of these ASIC-specific tasks are written as Python functions in **ASIC\_Routines.py**

## Writing Routines that can be Run from Spacely Command Line

To tell Spacely that a routine should be runnable from the command line, start the function name with “ROUTINE\_”.

The next time you run Spacely, a comment will be automatically added to your function:

```
#<<Registered w/ Spacely as ROUTINE 8, call as ~r8>>
def ROUTINE_Full_Conversion_vs_ArbParam():
```

You can now use the idiom “~r8” to call this function from the Spacely command line.

ROUTINEs are intended to be top-level test functions called by the users: they should be callable without arguments.

## An Example Spacely Routine

This section provides a line-by-line breakdown of an example Spacely routine, which may be helpful for those who are newer to Python.

- 1) **Define the Test Function:** Each test routine is a Python function. Start by defining a function with a descriptive name that reflects the test's purpose.

Example: The function `ROUTINE_Full_Conversion_Histogram()` is designed to create a histogram of the results from a Full Conversion test, the distribution of ADC conversion results over a large number of samples.

**Note:** This routine takes two arguments, which may be passed to it when it is called by a different, larger-scope routine. However, note that each argument has a default value (“= None”) which allows this routine to be called independently from the Spacely command line.

```
def ROUTINE_Full_Conversion_Histogram(experiment = None, data_file=None):
    """Create a histogram of the results from a Full Conversion """
```

- 2) **Routine-specific Instrument Configuration:** When Spacely starts, it will perform configuration specified in MyASIC\_Config.py. However, you may want to change elements of the configuration for a specific test.

Example: The initialization step for this function configures the Arbitrary Waveform Generator (AWG) to output a pulse waveform with a specific amplitude, pulse width, and pulse period.

config\_AWG\_as\_Pulse() is a method from the AWG library from py-libs-common.

```
# When tsf=1, pulses are 250 nanoseconds (0.25 us) wide because mclk's frequency is 2 MHz (T=500ns).
sg.INSTR["AWG"].config_AWG_as_Pulse(df.get("VIN_mV"), pulse_width_us=0.25*df.get("tsf_sample_phase"), pulse_period_us=period_us)
#time.sleep(3)
```

- 3) **Define Test Patterns or Test Logic:** Write the algorithm for the test. This could include generating test patterns, sending commands to the ASIC, and controlling the timing of operations.

Example: This function generates two different Glue waveforms at runtime. SC\_PATTERN is a Glue waveform used to program the ASIC's scan chain. Fc\_glue is a Glue waveform used to execute a "full conversion" ADC operation.

```
#Set CapTrim value and ensure TestEn=0
SC_PATTERN = SC_CFG(override=0,TestEn=0,Range2=df.get("Range2"), CapTrim=df.get("CapTrim"))
sg.pr.run_pattern( genpattern_SC_write(SC_PATTERN),outfile_tag="sc_cfg")

fc_glue = genpattern_Full_Conversion(time_scale_factor=df.get("time_scale_factor"),tsf_sample_phase=df.get("tsf_sample_phase"), n_samp=df.get("n_skip"))
```

- 4) **Run a Test:** Direct the FPGA to run your test pattern and save the result to file.

Example: This function uses the global pattern-runner (sg.pr) to run a pattern defined above on the FPGA. The resulting waveforms are stored to file, and we save the filename as "fc\_result"

```
fc_result = sg.pr.run_pattern(fc_glue,outfile_tag="fc_result")[0]
```

- 5) **Post-Process Data:** Extract meaningful information from the saved Glue waves and format the result for long-term storage.

Example: Helper function `gc.read_glue()` is used to read in the waves generated in the last step, and `gc.get_bitstream()` is used to extract a specific wave from the file in a Python list format.

```
#Get DACclr and capLo
result_glue = sg.gc.read_glue(fc_result)
dacclr_wave = sg.gc.get_bitstream(result_glue,"DACclr")
caplo_wave = sg.gc.get_bitstream(result_glue,"capLo_ext")

results.append(interpret_CDAC_pattern_edges(caplo_wave, dacclr_wave))
```



# Iospec

The iospec file contains a mapping of digital communication pins from the FPGA to the ASIC.

**Signal Specification:** Each line within the IOSPEC file describes to a specific signal, formatted as {name}, {I/O}, {position}, {optional default value}

Where:

{name} represents the signal's identifier in the testbench.

{I/O} indicates whether it is an input to or an output from the ASIC.

{position} refers to its order in the bit vector for the FPGA.

{optional default value} is used when no signal change occurs.

**Hardware Resource Grouping:** Signals are grouped by their associated hardware resource (e.g., a specific slot, module, or FIFO). These groupings are encapsulated by the following construct:

**{SLOT\_NAME}/{IO MODULE}/{FIFO NAME} BEGIN**

...

**END**

For details on what hardware resource combinations are allowed, see **Current Hardware Configurations Supported by Glue on page 21**

```
// ExampleASIC iospec
//
// The iospec defines which pins on the NI (typically 0~31) are used for which ASIC signals.

// All I/O definitions need to be inside a "HARDWARE XYZ BEGIN ... END" block to tell which NI hardware we're talking about.

// PXI1Slot4 = The NI slot we are using.
// NI6583 = The type of IO card attached to the FPGA in this slot.
// se_io = single-ended I/O (the NI6583 also supports lvds)

HARDWARE PXI1Slot4/NI6583/se_io BEGIN

//mclk and data_in are ASIC inputs on NI pins 0 and 1
mclk,I,0
data_in,I,1

//data_out is an ASIC output on NI pin 2.
data_out,O,2

END
```

# Spacely Quick Reference

## Command Line Options

|                         |  |
|-------------------------|--|
| <code>--defaults</code> | Initialize all instruments with their default addresses from <code>ASIC_Config.py</code><br>(If this is not specified, Spacely will ask for permission to initialize each instrument.) |
| <code>-r</code>         | After initializing, automatically run routine with number <code>&lt;routine #&gt;</code>   |

## Spacely Commands

These commands will work only in the Spacely command shell.

```
# Spacely ready to take commands (see "help" for details)
> |
```

|                                       |   |
|---------------------------------------|---|
| <code>&gt; lr</code>                  | This shows all the possible routines with their numbers   |
| <code>&gt; ~r&lt;routine #&gt;</code> | This instructs Spacely to run   |
| <code>&gt; ni_mon</code>              | One can see all the current rails in the opened window.   |
| <code>&gt; help</code>                | Lists all the possible Spacely commands and Command Line Options  |
| <code>&gt; gcshell</code>             | Launches the Glue Converter Shell.<br>See the chapter <b>Using NI-FGPAs with Spacely (Glue)</b> for more details. |

## Spacely Idioms

These are useful snippets of code to perform common tasks in Spacely. They will work both in Spacely routines, as well as in the Spacely command shell.

|  |  |
|--|--|
| <code>sg.pr.run_pattern(&lt;glue file&gt;)</code>  | Use the global Pattern Runner to run a pattern on the FPGA.  |
| <code>sg.gc.read_glue(&lt;glue file&gt;)</code><br><code>sg.gc.get_bitstream(...)</code><br>etc... | Use the global Glue Converter to read and manipulate Glue files. For a full list of methods, see <i>py-libs-common/nitoolbox/src/fnal_ni_toolbox/glue_converter.py</i> |
| <code>Sg.INSTR["Instr_name"].method()</code>   | Run methods of a specific instrument, such as setting a Scope's resolution or setting an AWG's output voltage.<br>See <b>Spacely Instrument Libraries</b> chapter.     |
| <code>V_PORT["VDD"].set_voltage()</code><br><code>I_PORT["Ibias"].set_current()</code><br>Etc...   | Modify the voltage or current of rails specified in <code>MyASIC_Config.py</code>  |
| <code>Genpattern_from_waves_dict(&lt;waves dict&gt;)</code>  | Generate a glue waveform from a dictionary that contains waves specified as lists of integers for each signal.   |

# Using NI-FPGAs with Spacely (Glue)

One of the most powerful capabilities of Spacely is applying arbitrary digital patterns to an ASIC using NI-FPGA cards. This capability can be used to program digital scan chains, send commands using SPI or other communication protocols, or generate control patterns for analog / mixed-signal blocks.

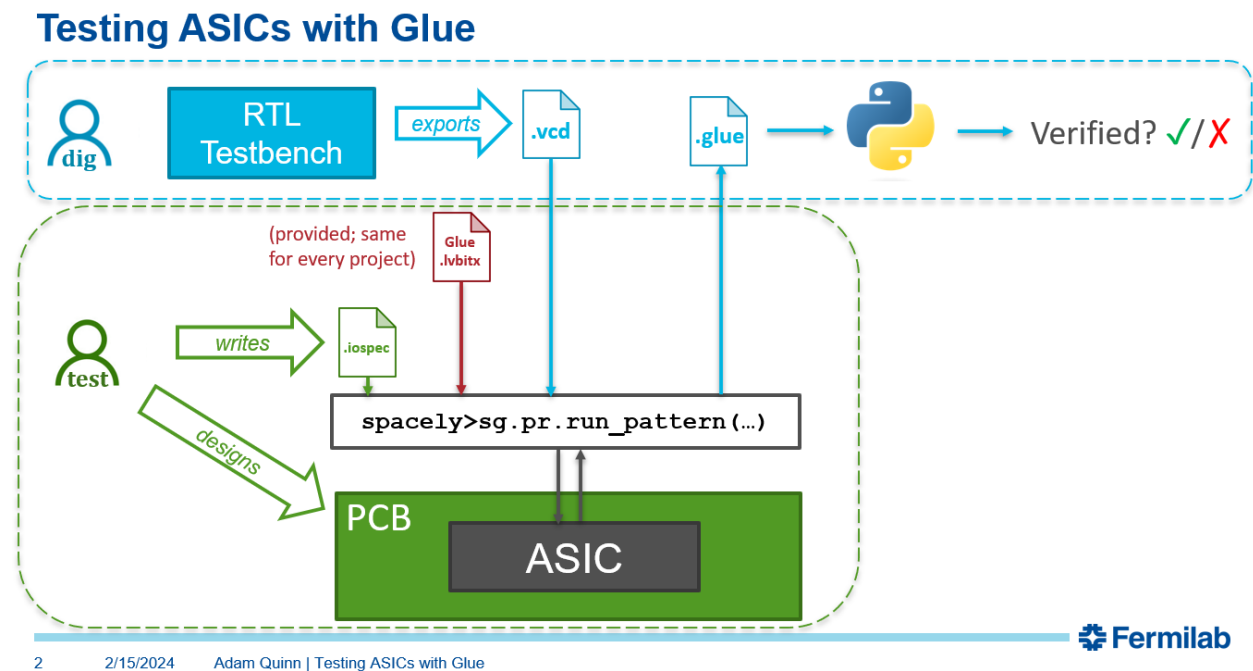
The basic workflow is:

1. Specify the intended pattern for ASIC inputs in a Glue Wave.
2. Use the *pattern\_runner* class to run that pattern on your ASIC.
3. A new Glue Wave will be written to file containing your inputs and the ASIC's response on its outputs.
4. Analyze that Glue Wave.

There are generally three ways to create a Glue Wave, which are described in later sections in this chapter:

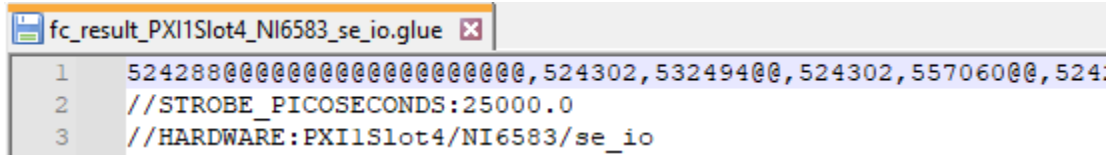
1. Create a Glue Wave natively in Python (**Python2FPGA Flow**)
2. Manually create a wave in a TXT file and convert it into a Glue Wave (**ASCII2FPGA Flow**)
3. Convert a VCD file from a digital testbench into a Glue Wave (**RTL2FPGA Flow**)

The last option notably allows direct re-use of RTL testbenches created by digital designers:



# The Glue Wave Format

In order to be understood by the NI-FPGA, a test pattern must be specified in “Glue Wave” format. A Glue Wave is essentially a list of 32-bit integers, where each integer corresponds to the state of all 32 FPGA I/O bits at a single FPGA timestep.



```
1 52428800000000000000000000000000,524302,53249400,524302,55706000,524:
2 //STROBE_PICOSECONDS:25000.0
3 //HARDWARE:PXI1Slot4/NI6583/se_io
```

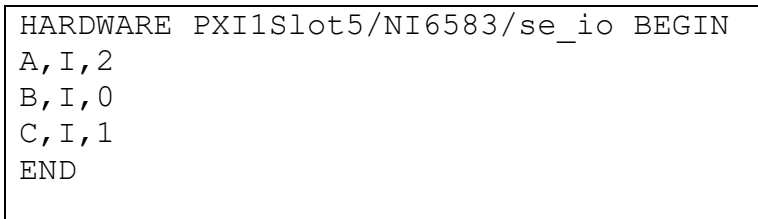
When all I/Os remain constant for more than one timestep, the repeated integer is replaced with an “@” symbol to make the resulting data file smaller.

## Basic Example Scenario

To illustrate, consider a test bench with signals A, B, and C with the pattern below:

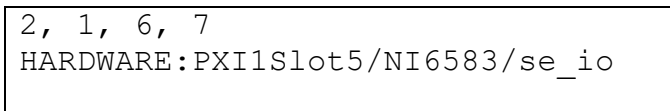
|   |   |   |   |   |
|---|---|---|---|---|
| A | 0 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 1 |
| C | 1 | 0 | 1 | 1 |

And the iospec file is the following:



```
HARDWARE PXI1Slot5/NI6583/se_io BEGIN
A, I, 2
B, I, 0
C, I, 1
END
```

The expected content of output.glue is:



```
2, 1, 6, 7
HARDWARE:PXI1Slot5/NI6583/se_io
```

Expressed in binary, the numbers 2,1,6,7 are as follows. These correspond to the I/O bits which are asserted in each timestep.

3'b010, 3'b001, 3'b110, 3'b111

## Current Hardware Configurations Supported by Glue

| FPGA I/O Module | FPGA Card Version | Bitfile Clock Speed                         | FIFOs                     | Spacely Bitfile Name  |
|-----------------|-------------------|---|---------------------------|-----------------------|
| NI6583          | NI7976            | 40 MHz                                      | se_io (32b)<br>lvds (32b) | “NI7976_NI6583_40MHz” |
|                 | NI7972            | 40 MHz                                      | se_io (32b)<br>lvds (32b) | “NI7972_NI6583_40MHz” |
| NI6581          | NI7962            | 40 MHz                                      | <i>iba</i>                | “NI7962_NI6581_40MHz” |
|                 | NI7972            | N/A – This configuration is not compatible. |                           |                       |

### Choosing a FIFO

Glue uses internal FIFOs to pass data from computer memory to the ASIC and back again, ensuring that your waveforms can run at a sufficient rate. Each FIFO corresponds to a specific group of digital I/O signals.

#### NI6583

- The **se\_io** FIFO is used to control single-ended I/Os. There are 32 controllable signals, and read/write direction can be set for each signal independently.
- The **lvds** FIFO is used to control LVDS I/Os. There are 32 controllable signals, and read/write direction can be set for each signal independently.

# Working with Glue Waves

This section gives examples for how to perform various operations with Glue Waves.

Generally, any Glue Wave operation can be done in two ways:

- **Manually, using gcshell (“Glue Converter shell”)**, which can be launched from the Spacely command window by typing “gcshell”
- **With Python code**, calling methods from the global Glue Converter instance `sg.gc`

Creating Glue Waves is covered in the next three sections of this chapter, Python2FPGA Flow, ASCII2FPGA Flow, and RTL2FPGA Flow.

## Read A Glue Wave into Spacely from File

| Command Line  | Python Script  |
|---|--|
| <code>gcshell&gt;getglue</code><br>(You will be prompted to choose a Glue file) | <code>glue_wave = sg.gc.read_glue("filename")</code> |

## Plot a Glue Wave (Uses Matplotlib)

| Command Line  | Python Script                                   |
|---|---|
| <code>gcshell&gt;plotglue</code><br>(If you have not already read a Glue file you will be prompted to do so.) | <code>sg.gc.plot_glue(&lt;glue_wave&gt;)</code> |

## Write a Glue to File

| Command Line   | Python Script   |
|--|---|
| <code>Gcshell&gt;writeglue</code><br>File name? MyOutputFile | <code>sg.gc.writeglue(&lt;glue_wave&gt;, "MyOutputFile")</code> |

## Run a Glue Wave on the ASIC

| Command Line   | Python Script   |
|--|---|
| <pre>&gt;run_pattern</pre> <p>(You will be prompted to choose a Glue File to run.)</p> <p><b>Note:</b> This command is run in the regular Spacely command window, not gcshell.</p> | <pre>x = sg.pr.run_pattern&lt;glue_wave&gt;,<br/>outfile_tag="result") [0]</pre> <p><b>Note:</b> run_pattern returns a <i>list</i> of results. If we are only running a single glue file, we add the "[0]" to get the first entry in that list.</p> |

## Extract One Particular Signal from a Glue Wave

| Command Line  | Python Script   |
|---|---|
| <pre>Gcshell&gt;bits</pre> <p>(You will be prompted to choose a glue file)</p> <p>Data name? MySignal</p> <p>Output file name? MyOutputFile</p> | <pre>x = sg.gc.get_bitstream(&lt;glue_wave&gt;,<br/>"MySignal")</pre> |

## Glue Waves Which Run Concurrently on Multiple FIFOs

This capability is still in development. (Not needed for 2/22 Spacely Workshop.)

# Python2FPGA Flow

Oftentimes, the simplest way to generate a GlueWave is to just write a short Python script that will generate it.

First, create a waveform dictionary, where the keys of the dictionary are signal names, and the values are lists of integers that tell the value of that signal at each timestep. For example:

```
my_waves_dict = { "Signal1": [0, 1, 1, 0],
                  "Signal2": [0, 1, 0, 1]}
```

Next, you can call the built-in Spacely routine **genpattern\_from\_waves\_dict(<waves\_dict>)** which will convert your waveform dictionary to a glue wave and write it to file. The function will return the name of the Glue File that was written so that you can run it later.

Here's an example of a simple routine that writes user data to an ASIC scan chain using this flow:

```
#EWISOTT
def genpattern_scan_chain_write(sc_data, time_scale_factor):

    waves = {}

    waves["S_CLK"] = []
    waves["S_DIN"] = []

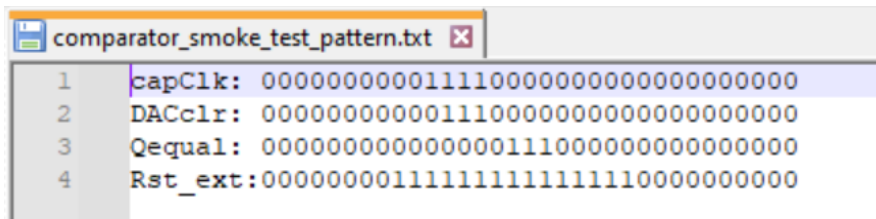
    for i in sc_data:
        waves["S_DIN"] = waves["S_DIN"] + [i]*2*time_scale_factor
        waves["S_CLK"] = waves["S_CLK"] + [0]*time_scale_factor + [1]*time_scale_factor

    #Use the Spacely built-in routine genpattern_from_waves_dict()
    #to turn your dictionary into a Glue wave
    return genpattern_from_waves_dict(waves)
```



## ASCII2FPGA Flow

If your pattern is short and always the same, it may make sense to just specify it by hand in a text file. Here's an example:



```
1 capClk: 00000000001111000000000000000000
2 DACclr: 00000000000111000000000000000000
3 Qequal: 00000000000000011100000000000000
4 Rst_ext:00000000111111111111110000000000
```

Once this ASCII file is created, you can convert it to a glue wave with:

```
sg.gc.ascii2Glue("ascii_file_name.txt", ticks_per_bit=1, output_file_tag= "my_glue_wave")
```

or

```
gcshell>ascii2input
```

By default, the Glue Converter will only capture the signals that correspond to ASIC inputs, because these are the only ones that you would logically want to drive. To override this behavior and convert all signals, you can pass the argument **inputs\_only=False** to **sg.gc.ascii2Glue()** or use


```
gcshell>ascii2golden
```

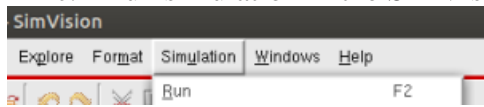
# RTL2FPGA Flow

If an RTL testbench already exists for verifying or communicating with an ASIC, it may make sense to re-use that testbench for testing the ASIC in the lab. This can be done by capturing the output signals of the testbench in VCD format and converting the VCD to a Glue Wave.

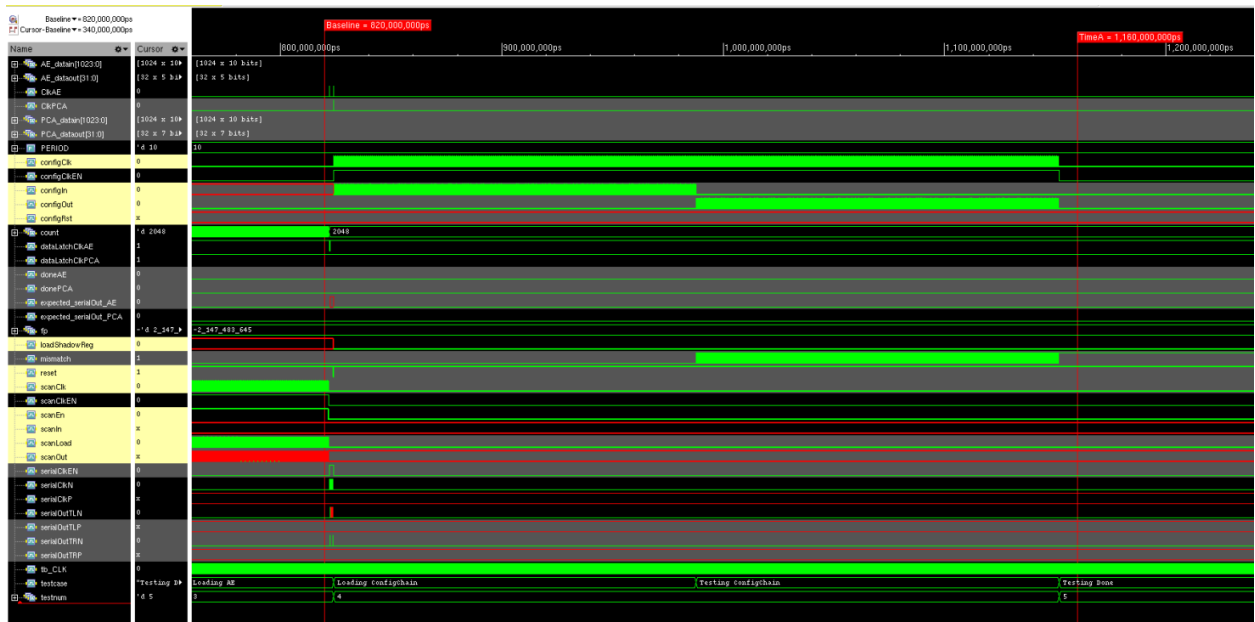
## Exporting a VCD file from the Verification testbench

From the verification testbench, a vcd file containing all essential signals is exported, serving as the test pattern.

- (1) Run the top-level testbench in SimVision.
  - a. Use the 'make sim' command to run the testbench.
  - b. Select objects in the top-level testbench and send them to the target waveform window .
  - c. Run simulation in the SimVision window.

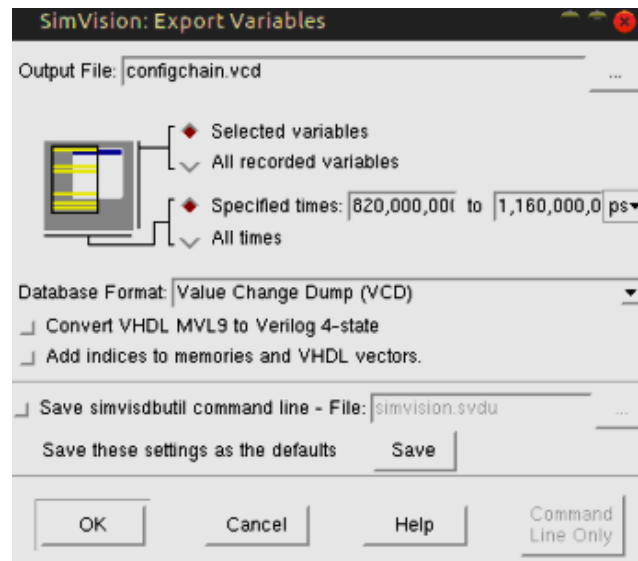


- (2) Choose an appropriate time interval and signals to export depending on the test goals. Exporting too many signals or too long of a time interval may make the Glue Wave needlessly large, slowing down the test.



config chain example

- (3) Export to a VCD file.
  - a. Confirm selected variables.
  - b. Set specified times.
  - c. Database Format: Value Change Dump



## Converting VCD → Glue Wave and Running the Test

This can be done in the command line or in Python:

| Command Line  | Python Script  |
|---|--|
| <pre>Gcshell&gt;vcd2input (You will be prompted to choose a VCD file.) Tb name? (Hit enter to auto identify) Strobe ps? (Hit enter for default = 25e3) Out file tag? MyGlueFile</pre> | <pre>sg.gc.VCD2Glue("MyVCD.vcd", strobe_ps=25e3, "MyGlueFile")</pre> |

By default, the Glue Converter will only capture the signals that correspond to ASIC inputs, because these are the only ones that you would logically want to drive. To override this behavior and convert all signals, you can pass the argument **inputs\_only=False** to **sg.gc.VCD2Glue()** or use **gcshell>vcd2golden**

### What is “strobe\_ps”?

The FPGA operates with a fixed, finite clock speed. By default in the current version of Spacely, that clock speed is 40 MHz. Your RTL testbench may operate much faster or slower than 40 MHz, so the Glue Converter will choose a strobe period at which to sample your VCD waveform. Once FPGA tick will be generated for each strobe period. The strobe period is specified by *strobe\_ps*.

Note that  $1 / 40 \text{ MHz} = 25 \text{ ns} = 25,000 \text{ ps}$ . So if you specify *strobe\_ps=25e3* (the default), Spacely will run your testbench in “real time”: one microsecond of simulated time will correspond to one

microsecond of real time. This may not be what you want if the timebase in your simulation is set to be very slow or very fast.

When Glue Converter runs, it will print out the mathematical conversions from VCD time to real time to allow you to double-check what will actually be run.

Example:

```
----- <<< VCD Timebase Conversion >>> -----  
Your VCD file uses time units of 1000000.0 ps.  
The length of your VCD is 10200 time units or 0.0102 seconds.  
Default FPGA clock is 40 MHz (1 / 25ns).  
You chose strobe_ps=25000 so 1 fpga tick (25 ns) = 25000 ps of sim time.  
Your resulting FPGA pattern will be 408000 ticks or 0.010199999999999999 seconds.  
(FPGA waveform is 1.0x the speed of the sim waveform.)  
-----
```

Now that you've converted your VCD to a Glue Wave, see the section **Run a Glue Wave on the ASIC on page 23** to run it on the ASIC.

# Spacely Instrument Libraries

## AWG Library

AWGs have the following methods:

1. ``connect(max_attempts=10, tcp_timeout=0.5) -> bool``:
  - Description: Connects to the Agilent AWG.
  - Parameters:
    - ``max_attempts`` (int, optional): Maximum number of connection attempts. Default is 10.
    - ``tcp_timeout`` (float, optional): TCP timeout value. Default is 0.5 seconds.
  - Returns:
    - ``bool``: True if connected successfully, False otherwise.
2. ``set_limit(voltage_volts: float) -> None``:
  - Description: Sets the voltage limit.
  - Parameters:
    - ``voltage_volts`` (float): Voltage limit in volts.
3. ``set_amplitude(voltage_mv: float) -> bool``:
  - Description: Sets the amplitude.
  - Parameters:
    - ``voltage_mv`` (float): Voltage amplitude in millivolts.
  - Returns:
    - ``bool``: True if successful, False otherwise.
4. ``set_offset(voltage_mv: float) -> bool``:
  - Description: Sets the offset.
  - Parameters:
    - ``voltage_mv`` (float): Voltage offset in millivolts.
  - Returns:
    - ``bool``: True if successful, False otherwise.
5. ``set_output(enabled: bool) -> None``:
  - Description: Sets the output state.
  - Parameters:
    - ``enabled`` (bool): Output state (True for ON, False for OFF).
6. ``restart() -> None``:
  - Description: Restarts the instrument, restoring volatile memory to defaults.
7. ``config_AWG_as_DC(val_mV: float) -> None``:
  - Description: Configures the Agilent AWG a DC output source.

- Parameters:
  - ``val_mV`` (float): Voltage in millivolts.
- 8. ``set_Vin_mV(val_mV: float) -> None``:
  - Description: Sets the AWG DC voltage.
  - Parameters:
    - ``val_mV`` (float): Voltage in millivolts.
- 9. ``set_pulse_mag(val_mV: float) -> None``:
  - Description: Sets the pulse magnitude.
  - Parameters:
    - ``val_mV`` (float): Pulse magnitude in millivolts.
- 10. ``config_AWG_as_Pulse(pulse_mag_mV, pulse_width_us=0.28, pulse_period_us=9) -> None``:
  - Description: Configures the Agilent AWG as a pulse generator.
  - Parameters:
    - ``pulse_mag_mV`` (float): Pulse magnitude in millivolts.
    - ``pulse_width_us`` (float, optional): Pulse width in microseconds. Default is 0.28.
    - ``pulse_period_us`` (float, optional): Pulse period in microseconds. Default is 9.

## Oscilloscope Library

The following methods are available for Oscilloscope instruments:

1. ``__init__(logger, io)``:
  - Description: Initializes the Oscilloscope object.
  - Parameters:
    - ``logger``: Object implementing ``debug()``, ``notice()``, and ``error()`` methods for logging.
    - ``io``: Spacely Interface (VISA or Prologix).
  - Returns: None.
2. ``get_id()``:
  - Description: Gets the identification string of the oscilloscope.
  - Parameters: None.
  - Returns:
    - ``str``: Identification string.
3. ``reset()``:
  - Description: Resets the oscilloscope.
  - Parameters: None.
  - Returns: None.
4. ``query(query_text)``:
  - Description: Sends a query command to the oscilloscope.

- Parameters:
    - ``query_text` (str)`: Query command.
  - Returns:
    - ``str``: Response from the oscilloscope.
5. ``write(write_text)``:
    - Description: Writes a command to the oscilloscope.
    - Parameters:
      - ``write_text` (str)`: Command to be written.
    - Returns: None.
  6. ``set_scale(scale_V, chan_num=None)``:
    - Description: Sets the scale (vertical sensitivity) of the specified channel or all channels.
    - Parameters:
      - ``scale_V` (float)`: Vertical scale in volts per division.
      - ``chan_num` (int, optional)`: Channel number. If None, sets scale for all channels.
    - Returns: None.
  7. ``set_timebase(timebase_s)``:
    - Description: Sets the timebase (horizontal scale) of the oscilloscope.
    - Parameters:
      - ``timebase_s` (float)`: Timebase in seconds per division.
    - Returns: None.
  8. ``set_offset(offset_V, chan_num=None)``:
    - Description: Sets the vertical offset of the specified channel or all channels.
    - Parameters:
      - ``offset_V` (float)`: Vertical offset in volts.
      - ``chan_num` (int, optional)`: Channel number. If None, sets offset for all channels.
    - Returns: None.
  9. ``get_wave(chan_num=1, convert_to_volts=True)``:
    - Description: Gets the waveform data from the specified channel.
    - Parameters:
      - ``chan_num` (int, optional)`: Channel number. Default is 1.
      - ``convert_to_volts` (bool, optional)`: Whether to convert the waveform data to volts. Default is True.
    - Returns:
      - ``list[float]``: Waveform data.
  10. ``enable_channels(enable_nums)``:
    - Description: Enables/disables the specified channels.
    - Parameters:
      - ``enable_nums` (list[int])`: List of channel numbers to be enabled.

- Returns: None.

11. ``setup_trigger(trigger_channel, threshold_V)``:

- Description: Sets up the trigger for the oscilloscope.

- Parameters:

- ``trigger_channel`` (int): Trigger channel.

- ``threshold_V`` (float): Trigger threshold in volts.

- Returns: None.

12. ``onscreen(channels=[1,2,3,4])``:

- Description: Displays the current contents of all oscilloscope channels on-screen.

- Parameters:

- ``channels`` (list[int], optional): List of channel numbers to be displayed. Default is [1,2,3,4].

- Returns: None.

## Source\_Port Library (Voltage & Current Bias)

Voltage and Current rails have the following methods.

These methods are available irrespective of whether the voltage/current rail is implemented using NIDCPower or an independent benchtop supply.

1. ``__init__(instrument, channel, default_current_limit=0.001, default_voltage_limit=0.1)``:

- Description: Initializes a Source\_Port object.

- Parameters:

- ``instrument``: Handle to the instrument responsible for this channel (could be ``nidcpower`` or ``supply``).

- ``channel``: Channel number.

- ``default_current_limit`` (optional): Default current limit for the channel. Default is 0.001 A.

- ``default_voltage_limit`` (optional): Default voltage limit for the channel. Default is 0.1 V.

- Returns: None.

2. ``set_voltage(voltage, current_limit=None)``:

- Description: Sets the voltage of the channel.

- Parameters:

- ``voltage``: Voltage value to be set.

- ``current_limit`` (optional): Current limit for the channel. If not provided, the default current limit will be used.

- Returns: None.

3. ``set_current(current, voltage_limit=None)``:

- Description: Sets the current of the channel.

- Parameters:



- ``current``: Current value to be set.
  - ``voltage_limit`` (optional): Voltage limit for the channel. If not provided, the default voltage limit will be used.
  - Returns: None.
4. ``report()``:
- Description: Reports the current settings of the channel.
  - Parameters: None.
  - Returns: None.
5. ``get_voltage()``:
- Description: Retrieves the voltage of the channel.
  - Parameters: None.
  - Returns:
    - ``float``: Voltage value.
6. ``get_current()``:
- Description: Retrieves the current of the channel.
  - Parameters: None.
  - Returns:
    - ``float``: Current value.
7. ``update_voltage_limit(voltage_limit)``:
- Description: Updates the voltage limit for the channel.
  - Parameters:
    - ``voltage_limit``: New voltage limit value.
  - Returns: None.
8. ``set_output_on()``:
- Description: Turns on the output of the channel.
  - Parameters: None.
  - Returns: None.
9. ``set_output_off()``:
- Description: Turns off the output of the channel.
  - Parameters: None.
  - Returns: None.

# Spacely Experiments

**Experiments** in Spacely are a data structure you can use to organize the data you take from experiments and keep track of metadata.

- An **Experiment** object is a collection of **DataFiles** and **metadata** represented as key-value pairs.
- Experiment metadata can be accessed with **Experiment.set(key,value)** and **Experiment.get(key)**
- A **DataFile** object represents an ordinary file (for example a CSV), and you can write lines of data to it with **DataFile.write()**
- DataFiles can also have their own metadata, which can be accessed with **DataFile.set(key,value)** and **DataFile.get(key)**.
- Experiment metadata is considered to be the “default” values, while DataFile metadata contains anything that varies from data file to data file. If DataFile does not have metadata for “x”, then DataFile.get(x) will return Experiment.get(x)

```
DataFile.get(x):  
    if DataFile.metadata[x] exists:  
        return DataFile.metadata[x]  
    else:  
        return Experiment.get(x)
```

Pseudo-code demonstrates how DataFile metadata overrides Experiment metadata.

See next page for an example routine using Experiments and DataFiles.

```

def ROUTINE_Sweep_vs_Vreffe_Experiment():
    """Run a Full Conversion Sweep for different values of Vref_fe"""

    e = Experiment("Sweep_vs_Vreffe_Experiment")

    #Defaults (same for all data files)
    e.set("time_scale_factor",10)
    e.set("tsf_sample_phase",2)
    e.set("CapTrim",25)
    e.set("SINGLE_PULSE_MODE",True)
    e.set("n_skip",1)

    e.set("VIN_STEP_uV",5000)
    e.set("VIN_STEP_MIN_mV",5)
    e.set("VIN_STEP_MAX_mV",1000)

    Vref_fe_range = [x/1000 for x in range(400,1000,25)]

    for Range2 in [0,1]:
        for Vref_fe in Vref_fe_range:

            df = e.new_data_file(f'data,r2={Range2},Vref_fe={Vref_fe*1000}mV')
            df.set("Range2",Range2)

            if Range2 == 0:
                df.set("VIN_STEP_uV",5000)
                df.set("VIN_STEP_MIN_mV",5)
                df.set("VIN_STEP_MAX_mV",1000)
            else:
                df.set("VIN_STEP_uV",1000)
                df.set("VIN_STEP_MIN_mV",5)
                df.set("VIN_STEP_MAX_mV",100)

            #Set Vref_fe voltage to a different value for each data file.
            sg.log.notice(f"Setting Vref_fe={Vref_fe}")
            V_PORT["Vref_fe"].set_voltage(Vref_fe)
            time.sleep(0.1)

            #Run full conversion sweep
            result = ROUTINE_Full_Conversion_Sweep(e,df)

            if result == -1: #Abort on abnormal return status
                return

```

# Spacely-Caribou

See Spacely-Caribou Reference.