# NiFpga Example Documentation

***Release 17.0.0***

**National Instruments**

**Jun 05, 2017**

# User Documentation


**1 About** **3**

**2 Bugs / Feature Requests** **5**
2.1 Information to Include When Asking For Help . . . . . 5

**3 Additional Documentation** **7**

**4 License** **9**
4.1 Getting Started . . . . . 9
4.1.1 Installation . . . . . 9
4.1.2 Basic Examples . . . . . 10
4.2 API References . . . . . 14
4.2.1 Sessions . . . . . 14
4.2.2 Registers . . . . . 15
4.2.3 Array Registers . . . . . 16
4.2.4 FIFOs . . . . . 17
4.2.5 Status . . . . . 18
4.3 Examples . . . . . 19

**5 Indices and Tables** **21**

The National Instruments FPGA Interface Python API is used for communication between processor and FPGA within NI reconfigurable I/O (RIO) hardware such as NI CompactRIO, NI Single-Board RIO, NI FlexRIO, and NI R Series multifunction RIO.

With the FPGA Interface Python API, developers can use LabVIEW FPGA to program the FPGA within NI hardware and communicate to it from Python running on a host computer. This gives engineers and scientists with Python expertise the ability to take advantage of compiled LabVIEW FPGA bitfiles, also the option to reuse existing Python code.

| Info | Python API for interacting with LabVIEW FPGA Devices. See our GitHub. |
|---|---|
| Author | National Instruments |
| Maintainers | Michael Strain <Michael.Strain@ni.com>, Mose Gumble <mose.gumble@ni.com> |

# CHAPTER 1

## About

The `nifpga` package contains an API for interacting with National Instrument's LabVIEW FPGA Devices - from Python. This package was created and is officially supported by National Instruments.

**nifpga** supports versions 16.0 and later of the RIO driver.

Some functions in the **nifpga** package may be unavailable with earlier versions of your RIO driver. Visit the National Instruments downloads page to upgrade the appropriate RIO device driver for your hardware.

**nifpga** supports Windows and Linux operating systems.

**nifpga** supports Python 2.7, 3.4+ . **nifpga** will likely work on other Python implementations. Feel free to open a issue on github for supporting a new implementation.

CHAPTER 2

# Bugs / Feature Requests

To report a bug or submit a feature request, please use our GitHub issues page to open a new issue.

## Information to Include When Asking For Help

Please include **all** of the following information when opening an issue:

- Detailed steps on how to reproduce the problem, and full traceback (if applicable).
- The exact python version used:

```
$ python -c "import sys; print(sys.version)"
```

- The exact versions of packages used:

```
$ python -m pip list
```

- The exact version of the RIO driver used. Follow this KB article to determine the RIO driver you have installed.
- The operating system and version (e.g. Windows 7, CentOS 7.2, ...)

# CHAPTER 3

## Additional Documentation

If you are unfamiliar with LabVIEW FPGA module, perusing the LabVIEW FPGA Module resource is a great way to get started. This documentation is API-agnostic.

CHAPTER 4

# License

**nifpga** 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.

## Getting Started

This document will show you how to get up and running with the NI FPGA Interface Python API.

### Installation

The NI FPGA Interface Python API can be installed through pip, see below for more detailed instructions.

#### Windows

1. Install the correct driver for your RIO device
    - You can find drivers at http://www.ni.com/downloads/ni-drivers/
2. Install Python https://www.python.org/downloads/
3. Install nifpga using pip (pip will be installed under "Scripts" in your python installation location.

```
pip install nifpga
```

#### Desktop Linux

1. Install the correct driver for your RIO device
    - You can find drivers at http://www.ni.com/downloads/ni-drivers/
2. Use your package manager to install the "python-pip" package

3. Install nifpga using pip

```
pip install nifpga
```

### NI Linux RT

1. Install the driver for your device using NI MAX
2. Enable SSH or the serial console from NI MAX
3. Connect to SSH or the serial console and login as admin
4. Run the following commands

```
opkg update
opkg install python-pip
pip install nifpga
```

## Basic Examples

### Opening a Session

The FPGA Interface Python API is session based. LabVIEW FPGA will generate bitfiles (.lvbitx) that can be used to program the hardware. For additional information on sessions view the API Page *Sessions*.

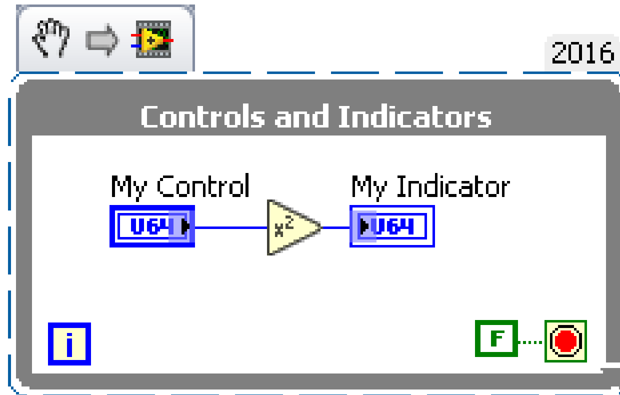
Recommended usage is to open a Session as follows:

```
from nifpga import Session

with Session(bitfile="MyBitfile.lvbitx", resource="RIO0") as session:
    # Reset stops the logic on the FPGA and puts it in the default state.
    # May substitute reset with download if your bitfile doesn't support it.
    session.reset()

    # Add Initialization code here!
    # Write initial values to controls while the FPGA logic is stopped.

    # Start the logic on the FPGA
    session.run()

    # Add code that interacts with the FPGA while it is running here!
```

### Using Controls and Indicators

Controls and indicators are used to transmit small amounts of data to and from the FPGA. The controls and indicators accessible by the FPGA Interface Python API are from the front panel of the top level VI from the LabVIEW FPGA code that was built into the bitfile. Accessing a control or indicator is done via its unique name from *Sessions*'s register property. For additional information on controls and indicators view the API page *Registers*.

The following example uses this FPGA VI:




This VI will take in a value from MyControl, square it, and output it to MyIndicator.

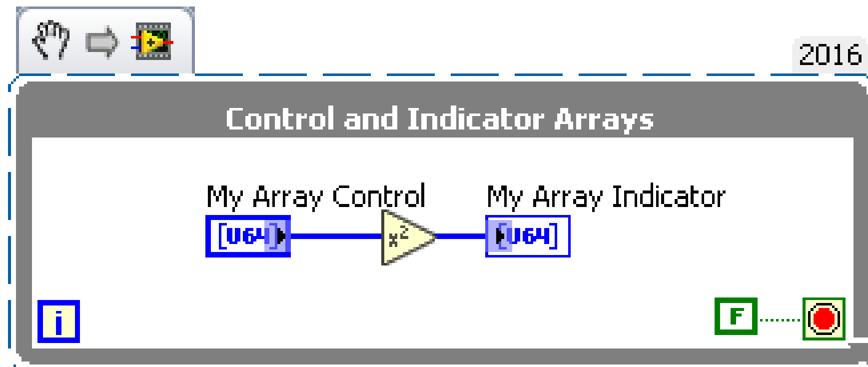
Example Usage:

```
from nifpga import Session

with Session("MyBitfile.lvbitx", "RIO0") as session:
    my_control = session.registers['My Control']
    my_indicator = session.registers['My Indicator']
    my_control.write(4)
    data = my_indicator.read()
    print(data)   # prints 16
```

### Using Array Controls and Indicators

Controls and indicators can also be an array type. They work like the a non-array registers, except use a python list for reading and writing. Accessing an array control or indicator is done via its unique name from *Sessions*'s register property, all controls and indicators exist in this dictionary. For additional information on array controls and indicators view the API page *Array Registers*.

For the following example, we have added two arrays to our FPGA VI:




Example Usage:

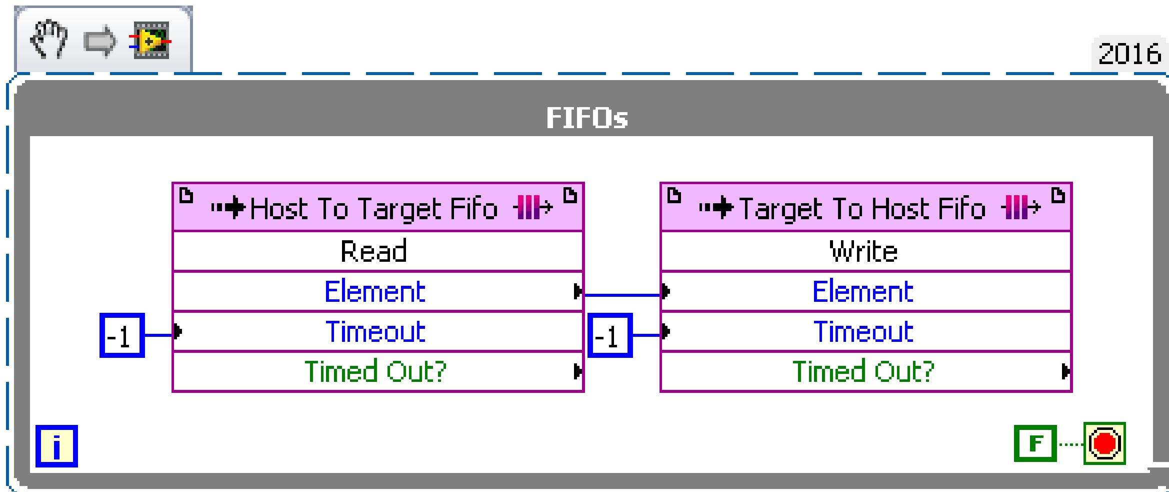
```
from nifpga import Session

with Session("MyBitfile.lvbitx", "RIO0") as session:
    my_array_control = session.registers['My Array Control']
    my_array_indicator = session.registers['My Array Indicator']
```

```
    data = [0, 1, 2, 3, 4]
    my_array_control.write(data)
    print(my_array_indicator.read())  # prints [0, 1, 4, 9, 16]
```

### Using FIFOs

FIFOs are used for streaming data to and from the FPGA. A FIFO is accessible by the FPGA Interface Python API via the top level VI from LabVIEW FPGA code. For additional information on FIFOs view the API page *FIFOs*.

For the following example, we have made a VI with two FIFOs. One FIFO is a host to target FIFO and the other is target to host FIFO. This VI uses the FIFOs to stream data from the processor, to the FPGA and then back to the processor.



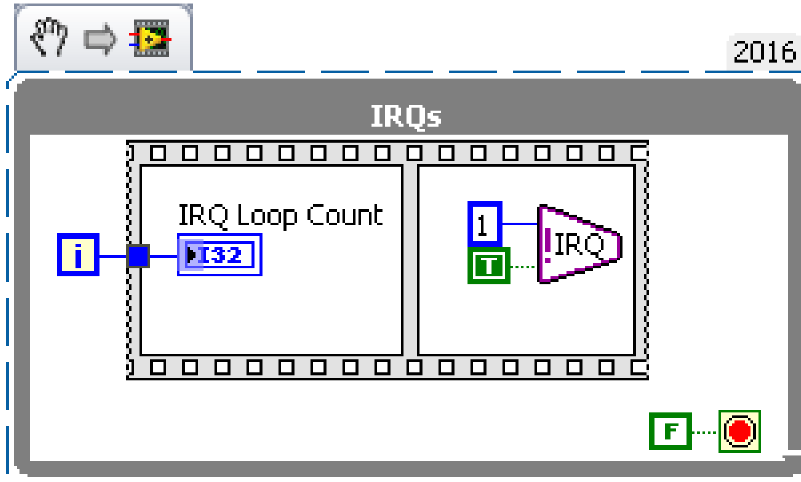

Example Usage:

```
from nifpga import Session

# create a list of 100 incrementing values
data = list(range(0, 100))

with Session("MyBitfile.lvbitx", "RIO0") as session:
    host_to_target = session.fifos['Host To Target Fifo']
    target_to_host = session.fifos['Target To Host Fifo']
    host_to_target.start()
    target_to_host.start()

    # stream the data to the FPGA
    host_to_target.write(data, timeout_ms=100)
    # steam the data back to the processor
    read_value = target_to_host.read(100, timeout_ms=100)
    # read_value is a tuple containing the data and elements remaining
    print(read_value.elements_remaining)  # prints 0

    # loop over both lists and print if the data doesn't match
    for input_data, output_data in zip(data, read_value.data):
        if input_data != output_data:
            print("data error")
```

### Using IRQs

IRQs are used to generate and handle user interrupts occurring on the FPGA. IRQs are accessible through the *Sessions* class. IRQs have two methods `Session.wait_on_irqs(irqs, timeout_ms)()` and `Session.acknowledge_irqs(irqs)()`.

For the following example, we have made a VI with an IRQ in a loop. This will fire IRQ 1 continuously and block the loop until the user acknowledges the IRQ.




Example Usage:

```
from nifpga import Session

timeout_ms = 300
irq_1 = 1

with Session("MyBitfile.lvbitx", "RIO0") as session:
    loop_count = session.registers["IRQ Loop Count"]

    # Wait on irq_1
    irq_status = session.wait_on_irqs(irq_1, timeout_ms)
    if irq_status.timed_out is True:
        print("timeout out while waiting for the interrupt")

    # Check to see if irq 1 asserted
    if irq_1 in irq_status.irqs_asserted:
        print("1 was asserted")
    else:
        print("1 was not asserted")

    # Print the loop count before and after acknowledging the irq
    print("Initial loop count:")
    print(loop_count.read())
    # Acknowledge the IRQ(s) when we're done
    session.acknowledge_irqs(irq_status.irqs_asserted)

    # Wait for the IRQ to fire again
    session.wait_on_irqs(irq_1, timeout_ms)
    # Print the loop count again to see its been incremented once
    print("Loop count after acknowledge:")
    print(loop_count.read())
```

# API References

## Sessions

**class** `nifpga.session.`**`Session`**(*bitfile*, *resource*, *no_run=False*, *reset_if_last_session_on_exit=False*, ***kwargs*)

Session, a convenient wrapper around the low-level _NiFpga class.

The Session class uses regular python types, provides convenient default arguments to C API functions, and makes controls, indicators, and FIFOs available by name. If any NiFpga function return status is non-zero, the appropriate exception derived from either WarningStatus or ErrorStatus is raised. Example usage of FPGA configuration functions:

```
with Session(bitfile="myBitfilePath.lvbitx", resource="RIO0") as session:
    session.run()
    session.download()
    session.abort()
    session.reset()
```

**Note:** It is always recommended that you use a Session with a context manager (with). Opening a Session without a context manager could cause you to leak the session if `Session.close()` is not called.

Controls and indicators are accessed directly via a _Register object obtained from the session:

```
my_control = session.registers["MyControl"]
my_control.write(data=4)
data = my_control.read()
```

FIFOs are accessed directly via a _FIFO object obtained from the session:

```
myHostToFpgaFifo = session.fifos["MyHostToFpgaFifo"]
myHostToFpgaFifo.stop()
actual_depth = myHostToFpgaFifo.configure(requested_depth=4096)
myHostToFpgaFifo.start()
empty_elements_remaining = myHostToFpgaFifo.write(data=[1, 2, 3, 4],
                                                  timeout_ms=2)

myFpgaToHostFifo = session.fifos["MyHostToFpgaFifo"]
read_values = myFpgaToHostFifo.read(number_of_elements=4,
                                    timeout_ms=0)
print(read_values.data)
```

**`__init__`**(*bitfile*, *resource*, *no_run=False*, *reset_if_last_session_on_exit=False*, ***kwargs*)

Creates a session to the specified resource with the specified bitfile.

**Parameters**

- **`bitfile`** (*`str) (Bitfile`*) – A bitfile.Bitfile() instance or a string filepath to a bitfile.
- **`resource`** (*`str`*) – e.g. “RIO0”, “PXI1Slot2”, or “rio://hostname/RIO0”
- **`no_run`** (*`bool`*) – If true, don’t run the bitfile, just open the session.
- **`reset_if_last_session_on_exit`** (*`bool`*) – Passed into Close on exit. Unused if not using this session as a context guard.
- **`**kwargs`** – Additional arguments that edit the session.

**abort**()
Aborts the FPGA VI.

**acknowledge_irqs**(*irqs*)
Acknowledges an IRQ or set of IRQs.

**Parameters** **irqs** (*list*) – A list of irq ordinals 0-31, e.g. [0, 6, 31].

**close**(*reset_if_last_session=False*)
Closes the FPGA Session.

**Parameters** **reset_if_last_session** (*bool*) – If True, resets the FPGA on the last close. If true, does not reset the FPGA on the last session close.

**download**()
Re-downloads the FPGA bitstream to the target.

**fifos**
This property returns a dictionary containing all FIFOs that are associated with the bitfile opened with the session. A FIFO can be accessed by its unique name.

**registers**
This property returns a dictionary containing all registers that are associated with the bitfile opened with the session. A register can be accessed by its unique name.

**reset**()
Resets the FPGA VI.

**run**(*wait_until_done=False*)
Runs the FPGA VI on the target.

**Parameters** **wait_until_done** (*bool*) – If true, this functions blocks until the FPGA VI stops running

**wait_on_irqs**(*irqs*, *timeout_ms*)
Stops the calling thread until the FPGA asserts any IRQ in the irqs parameter or until the function call times out.

**Parameters**

- **irqs** – A list of irq ordinals 0-31, e.g. [0, 6, 31].
- **timeout_ms** – The timeout to wait in milliseconds.

**Returns**

session_wait_on_irqs (namedtuple):

```
session_wait_on_irqs.irqs_asserted (list): is a list of the
    asserted IRQs.
session_wait_on_irqs.timed_out (bool): Outputs whether or not
    the time out expired before all irqs were asserted.
```

## Registers

**class** nifpga.session.**_Register**(*session*, *nifpga*, *bitfile_register*, *base_address_on_device*)
Bases: object

_Register is a private class that is a wrapper of logic that is associated with controls and indicators.

All Registers will exists in a sessions session.registers property. This means that all possible registers for a given session are created during session initialization; a user should never need to create a new instance of this class.

**__len__**()
A single register will always have one and only one element.

**Returns** Always a constant 1.

**Return type** (int)

**datatype**
Property of a register that returns the datatype of the control or indicator.

**name**
Property of a register that returns the name of the control or indicator.

**read**()
Reads a single element from the control or indicator

**Returns** The data inside the register.

**Return type** data (DataType.value)

**write**(*data*)
Writes the specified data to the control or indicator

**Parameters** **data** (*DataType.value*) – The data to be written into the register

## Array Registers

**class** nifpga.session.**_ArrayRegister**(*session*, *nifpga*, *bitfile_register*, *base_address_on_device*)
Bases: *nifpga.session._Register*

_ArryRegister is a private class that inherits from _Register with additional interfaces unique to the logic of array controls and indicators.

**__len__**()
Returns the length of the array.

**Returns** The number of elements in the array.

**Return type** (int)

**datatype**
Property of a register that returns the datatype of the control or indicator.

**name**
Property of a register that returns the name of the control or indicator.

**read**()
Reads the entire array from the control or indicator.

**Returns** The data in the register in a python list.

**Return type** (list)

**write**(*data*)
Writes the specified array of data to the control or indicator

**Parameters**

- **data** (*list*) – The data "array" to be written into the registers
- **into a python list.** (*wrapped*) –

## FIFOs

**class** `nifpga.session.`**`_FIFO`**(*session*, *nifpga*, *bitfile_fifo*)

Bases: `object`

_FIFO is a private class that is a wrapper for the logic that associated with a FIFO.

All FIFOs will exists in a sessions session.fifos property. This means that all possible FIFOs for a given session are created during session initialization; a user should never need to create a new instance of this class.

**`configure`**(*requested_depth*)

Specifies the depth of the host memory part of the DMA FIFO.

**Parameters** **`requested_depth`** (*`int`*) – The depth of the host memory part of the DMA FIFO in number of elements.

**Returns** The actual number of elements in the host memory part of the DMA FIFO, which may be more than the requested number.

**Return type** actual_depth (int)

**`datatype`**

Property of a Fifo that contains its datatype.

**`get_peer_to_peer_endpoint`**()

Gets an endpoint reference to a peer-to-peer FIFO.

**`name`**

Property of a Fifo that contains its name.

**`read`**(*number_of_elements*, *timeout_ms=0*)

Read the specified number of elements from the FIFO.

**Note:** If the FIFO has not been started before calling `_FIFO.read()`, then it will automatically start and continue to work as expected.

**Parameters**

- **`number_of_elements`** (*`int`*) – The number of elements to read from the FIFO.
- **`timeout_ms`** (*`int`*) – The timeout to wait in milliseconds.

**Returns**

ReadValues (namedtuple):

```
ReadValues.data (list): containing the data from
    the FIFO.
ReadValues.elements_remaining (int): The amount of elements
    remaining in the FIFO.
```

**`start`**()

Starts the FIFO.

**`stop`**()

Stops the FIFO.

**`write`**(*data*, *timeout_ms=0*)

Writes the specified data to the FIFO.

**Note:** If the FIFO has not been started before calling `_FIFO.write()`, then it will automatically start and continue to work as expected.

**Parameters**

- **data** (*list*) – Data to be written to the FIFO.
- **timeout_ms** (*int*) – The timeout to wait in milliseconds.

**Returns** The number of elements remaining in the host memory part of the DMA FIFO.

**Return type** elements_remaining (int)

## Status

**class** `nifpga.status.`**`Status`**(*code*, *code_string*, *function_name*, *argument_names*, *function_args*)

Bases: `exceptions.BaseException`

**`__init__`**(*code*, *code_string*, *function_name*, *argument_names*, *function_args*)

Base exception class for when an NiFpga function returns a non-zero status.

**Parameters**

- **code** (*int*) – e.g. -52000
- **code_string** (*str*) – e.g. ‘MemoryFull’
- **function_name** (*string*) – the function that returned the error or warning status. e.g. ‘NiFpga_ConfigureFifo’
- **argument_names** (*list*) – a list of the names of the arguments to the function. e.g. [”session”, “fifo”, “requested depth”]
- **function_args** (*tuple*) – a tuple of the arguments passed to the function. The order of argument_names should correspond to the order of function_args. e.g. ‘(session, fifo, depth)’

**`__str__`**()

Returns the function name, status code, and arguments used. Example:

```
Error: FifoTimeout (-50400) when calling 'Dummy Function Name' with
arguments:
    session: 0xbeef
    fifo: 0xf1f0L
    data: 0xda7aL
    number of elements: 0x100L
    timeout ms: 0x200L
    elements remaining: 0x300L
    a bogus string argument: 'I am a string'
```

**`get_args`**()

Returns a dictionary of argument names to argument values of the function that caused the exception to be raised.

Returns: arg_dict (dictionary): Converts ctypes args to their actual values instead of the ctypes instance. e.g.

```
{
"session":0x10000L,
"fifo" : 0x0,
...}
```

**get_function_name**()

Returns a string for the functions name,

**class** nifpga.status.**WarningStatus**(*code*, *code_string*, *function_name*, *argument_names*, *function_args*)

Bases: *nifpga.status.Status*, exceptions.RuntimeWarning

Base warning class for when an NiFpga function returns a warning (> 0) status.

Useful if trying to catch warning and error status exceptions separately

**class** nifpga.status.**ErrorStatus**(*code*, *code_string*, *function_name*, *argument_names*, *function_args*)

Bases: *nifpga.status.Status*, exceptions.RuntimeError

Base Error class for when an NiFpga function returns an error (< 0) status.

Useful if trying to catch warning and error status exceptions separately

## Examples

This Section will go different snippets of example code using the FPGA Interface python API to accomplish different tasks. Use the following links to navigate to the examples.

CHAPTER 5

# Indices and Tables

- genindex
- modindex
- search

# Index

## Symbols

_ArrayRegister (class in nifpga.session), 16
_FIFO (class in nifpga.session), 17
_Register (class in nifpga.session), 15
__init__() (nifpga.session.Session method), 14
__init__() (nifpga.status.Status method), 18
__len__() (nifpga.session._ArrayRegister method), 16
__len__() (nifpga.session._Register method), 15
__str__() (nifpga.status.Status method), 18

## A

abort() (nifpga.session.Session method), 14
acknowledge_irqs() (nifpga.session.Session method), 15

## C

close() (nifpga.session.Session method), 15
configure() (nifpga.session._FIFO method), 17

## D

datatype (nifpga.session._ArrayRegister attribute), 16
datatype (nifpga.session._FIFO attribute), 17
datatype (nifpga.session._Register attribute), 16
download() (nifpga.session.Session method), 15

## E

ErrorStatus (class in nifpga.status), 19

## F

fifos (nifpga.session.Session attribute), 15

## G

get_args() (nifpga.status.Status method), 18
get_function_name() (nifpga.status.Status method), 19
get_peer_to_peer_endpoint() (nifpga.session._FIFO method), 17

## N

name (nifpga.session._ArrayRegister attribute), 16
name (nifpga.session._FIFO attribute), 17
name (nifpga.session._Register attribute), 16

## R

read() (nifpga.session._ArrayRegister method), 16
read() (nifpga.session._FIFO method), 17
read() (nifpga.session._Register method), 16
registers (nifpga.session.Session attribute), 15
reset() (nifpga.session.Session method), 15
run() (nifpga.session.Session method), 15

## S

Session (class in nifpga.session), 14
start() (nifpga.session._FIFO method), 17
Status (class in nifpga.status), 18
stop() (nifpga.session._FIFO method), 17

## W

wait_on_irqs() (nifpga.session.Session method), 15
WarningStatus (class in nifpga.status), 19
write() (nifpga.session._ArrayRegister method), 16
write() (nifpga.session._FIFO method), 17
write() (nifpga.session._Register method), 16