Analog Devices Hardware Python Interfaces

Release 0.0.3

Nov 05, 2020

Contents

1 Requirements **3**

2 Sections **5**

2.1 Quick Start 5
2.2 Attributes 6
2.3 Examples 6
2.4 Connectivity 8
2.5 Supported Devices 10
2.6 Buffers 10
2.7 FPGA Features 12
2.8 Developers 13
2.9 Support 14

3 Indices and tables **15**

This module provides a convenient way to access and control ADI hardware from Python through existing IIO drivers.

CHAPTER 1

Requirements

- libiio

CHAPTER 2

Sections

2.1 Quick Start

Before installing pyadi-iio make sure you have libiio and its python bindings installed.

Note: libiio does not currently have a pip installer, but releases are available on GitHub along with the source. For releases v0.19+ of libiio, when building from source the -DPYTHON_BINDINGS=ON flag is required

pyadi-iio can by installed from pip

```
(sudo) pip install pyadi-iio
```

or by grabbing the source directly

```
git clone https://github.com/analogdevicesinc/pyadi-iio.git
cd pyadi-iio
(sudo) python3 setup.py install
```

Note: On Linux the libiio python bindings are sometimes installed in locations not on path. On Ubuntu this is a common fix

```
export PYTHONPATH=$PYTHONPATH:/usr/lib/python{PYTHON VERSION}/site-packages
```

2.1.1 Install Checks

For check for libiio with the following from a command prompt or terminal:

```
dave@hal:~$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import iio
>>> iio.version
(0, 18, 'eec5616')
```

If that worked, try the follow to see if pyadi-iio is there:

```
dave@hal:~$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import adi
>>> adi.__version__
'0.0.5'
>>> adi.name
'Analog Devices Hardware Interfaces'
```

2.2 Attributes

To simplify hardware configuration through different IIO drivers, basic class properties are exposed at the top-level of each device specific class. These properties abstract away the need to know a specific channel name, attribute type, source device name, and other details required in the libIIO API. Instead properties have easy to understand names, documentation, and error handling to help manage interfacing with different hardware. Property data can be read and written as follows from a given device interface class:

```
import adi

lidar = adi.fmclidar1()
# Read current pulse width
print(lidar.laser_pulse_width)
# Change laser frequency to 1 MHz
lidar.laser_frequency = 1000000
```

If more detail is required about a specific property it can be directly inspected in the class definitions documnentation or in python itself through the help methods:

```
python3
>>> import adi
>>> help(adi.Pluto.gain_control_mode_chan0)
Help on property:
    gain_control_mode_chan0: Mode of receive path AGC. Options are:
    slow_attack, fast_attack, manual
```

For complete documentation about class properties reference the *supported devices* classes.

2.3 Examples

Here is a collection of small examples which demonstrate interfacings with different devices in different ways.

Configuring hardware properties and reading back settings

```
# Import the library
import adi

# Create a device interface
sdr = adi.ad9361()
# Configure properties
sdr.rx_rf_bandwidth = 4000000
sdr.rx_lo = 2000000000
sdr.tx_lo = 2000000000
sdr.tx_cyclic_buffer = True
sdr.tx_hardwaregain = -30
sdr.gain_control_mode = "slow_attack"
# Read back properties from hardware
print(sdr.rx_hardwaregain)
```

Send data to a device and receiving data from a device

```
import adi
import numpy as np

sdr = adi.ad9361()
data = np.arange(1, 10, 3)
# Send
sdr.tx(data)
# Receive
data_rx = sdr.rx()
```

Configure the DDS of a transmit capable FPGA based device

```
import adi

sdr = adi.DAQ2()
# Configure DDS
tone_freq_hz = 1000  # In Hz
tone_scale = 0.9  # Range: 0-1.0
tx_channel = 1  # Starts at 0
sdr.dds_single_tone(tone_freq_hz, tone_scale, tx_channel)
```

Using URIs to access specific devices over the network

```
import adi

# Create device from specific uri address
sdr = adi.ad9361(uri="ip:192.168.2.1")
data = sdr.rx()
```

Using URIs to access specific devices over USB

```
import adi

# Create device from specific uri address
sdr = adi.Pluto(uri="usb:1.24.5")
data = sdr.rx()
```

Other complex examples are available in the source repository

2.4 Connectivity

Since pyadi-iio is built on top of libiio, it can use the different backends which allow device control and data transfer to and from devices remotely. These backends include serial, Ethernet, PCIe, USB, and of course locally connected devices can be controlled through the local backend. Connecting to a board remotely over a specific backend is done by defining a specific universal resource indicator (URI) and passing it the class constructors for a specific device. Here is a simple example that uses the Ethernet backend with a target board with IP address 192.168.2.1:

```
# Import the library
import adi

# Create a device interface
sdr = adi.ad9361(uri="ip:192.168.2.1")
# Read back properties from hardware
print(sdr.rx_hardwaregain)
```

Devices that are connected over USB or are on a system with IIO devices like a ZC706 or Zedboard, should be able to automatically connect without defining a URI like:

```
# Import the library
import adi

# Create a device interface
sdr = adi.Pluto()
# Read back properties from hardware
print(sdr.tx_rf_bandwidth)
```

Whoever if you have multiple USB device connected an want to pick one specifically, the set the USB URI similar to IP:

```
# Import the library
import adi

# Create a device interface
sdr = adi.Pluto(uri="usb:1.24.5")
# Read back properties from hardware
print(sdr.tx_rf_bandwidth)
```

If you are not sure of the device URI you can utilize libiio commandline tools like iio_info and iio_attr.

2.5 Supported Devices

2.5.1 ad5627

2.5.2 ad5686

2.5.3 ad7124

2.5.4 ad9094

2.5.5 ad9144

2.5.6 ad9152

2.5.7 ad936x

2.5.8 ad9371

2.5.9 ad9680

2.5.10 adar1000

2.5.11 adis16460

2.5.12 adis16507

2.5.13 adrv9002

2.5.14 adrv9009

2.5.15 adrv9009_zu11eg

2.5.16 adrv9009_zu11eg_fmcomms8

2.5.17 adxl345

2.5.18 daq2

2.5.19 daq3

2.5.20 fmclidar1

2.5.21 fmcomms5

2.5.22 ltc2983

2.6 Buffers

Using buffers or transmitting and receiving data is done through interacting with two methods.

For receivers this is the **rx** method. How data is captured and therefore produced by this method is dependent on two main properties:

- **rx_enabled_channels**: This is an array of integers and the number of elements in the array will determine the number of list items returned by **rx**. For devices with complex data types these are the indexes of the complex channels, not the individual I or Q channels.
- **rx_buffer_size**: This is the number of samples returned in each column. If the device produces complex data, like a transceiver, it will return complex data. This is defined by the author of each device specific class.

For transmitters this is the **tx** method. How data is sent and therefore must be passed by this method is dependent on one main property:

- **tx_enabled_channels**: This is an array of integers and the number of elements in the array will determine the number of items in list to be submitted to **tx**. Like for **rx_enabled_channels**, devices with complex data types these are the indexes of the complex channels, not the individual I or Q channels.

2.6.1 Cyclic Mode

In many cases, it can be useful to continuously transmit a signal over and over, even for just debugging and testing. This can be especially handy when the hardware you are using has very high transmit or receive rates, and therefore impossible to keep providing data to. To complement these use cases it is possible to create transmit buffer which repeats, which we call **cylic buffers**. Cyclic buffers are identical or normal or non-cylic buffers, except when they reach hardware they will continuously repeat or be transmitted. Here is a small example on how to create a cyclic buffer:

```
import adi

sdr = adi.ad9361()
# Create a complex sinusoid
fc = 3000000
N = 1024
ts = 1 / 30000000.0
t = np.arange(0, N * ts, ts)
i = np.cos(2 * np.pi * t * fc) * 2 ** 14
q = np.sin(2 * np.pi * t * fc) * 2 ** 14
iq = i + 1j * q
# Enable cyclic buffers
sdr.tx_cyclic_buffer = True
# Send data cyclically
sdr.tx(iq)
```

At this point, the transmitter will keep transmitting the create sinusoid indefinitely until the buffer is destroyed or the *sdr* object destructor is called. Once data is pushed to hardware with a cyclic buffer the buffer must be manually destroyed or an error will occur if more data push. To update the buffer use the **tx_destroy_buffer** method before passing a new vector to the **tx** method.

2.6.2 Members

2.6.3 Buffer Examples

Collect data from one channel

```
import adi
```

(continues on next page)

(continued from previous page)

```
sdr = adi.ad9361()
# Get complex data back
sdr.rx_enabled_channels = [0]
chan1 = sdr.rx()
```

Collect data from two channels

```
import adi

sdr = adi.ad9361()
# Get both complex channel back
sdr.rx_enabled_channels = [0, 1]
data = sdr.rx()
chan1 = data[0]
chan2 = data[1]
```

Send data on two channels

```
import adi
import numpy as np

# Create radio
sdr = adi.ad9371()
sdr.tx_enabled_channels = [0, 1]
# Create a sinewave waveform
N = 1024
fs = int(sdr.tx_sample_rate)
fc = 40000000
ts = 1 / float(fs)
t = np.arange(0, N * ts, ts)
i = np.cos(2 * np.pi * t * fc) * 2 ** 14
q = np.sin(2 * np.pi * t * fc) * 2 ** 14
iq = i + 1j * q
fc = -30000000
i = np.cos(2 * np.pi * t * fc) * 2 ** 14
q = np.sin(2 * np.pi * t * fc) * 2 ** 14
iq2 = i + 1j * q
# Send data to both channels
sdr.tx([iq, iq2])
```

2.7 FPGA Features

2.7.1 Direct Digital Synthesizers

For FPGA based systems ADI reference designs include direct digital synthesizer (DDS) which can generate tones with arbitrary phase, frequency, and amplitude. For each individual DAC channel there are two DDSs which can have a unique phase, frequency, and phase. To configure the DDSs there are a number of methods and properties available depending on the complexity of the configuration.

For the most basic or easiest configuration options use the methods **dds_single_tone** and **dds_dual_tone** which generate a one tone or two tones respectively on a specific channel.

```
import adi

sdr = adi.ad9361()
# Generate a single complex tone
dds_freq_hz = 10000
dds_scale = 0.9
# Enable all DDSs
sdr.dds_single_tone(dds_freq_hz, dds_scale)
```

To configure DDSs individually a list of scales can be passed to the properties **dds_scales**, **dds_frequencies**, and **dds_phases**.

```
import adi

sdr = adi.ad9361()
n = len(sdr.dds_scales)
# Enable all DDSs
sdr.dds_enabled = [True] * n
# Set all DDSs to same frequency, scale, and phase
dds_freq_hz = 10000
sdr.dds_phase = [0] * n
sdr.dds_frequency = [dds_freq_hz] * n
sdr.dds_scale = [0.9] * n
```

2.7.2 Methods

2.8 Developers

Warning: This section is only for developers and advanced users.

When submitting code or running tests, there are a few ways things are done in pyadi-iio.

2.8.1 Invoke

To make repetitve tasks easier, pyadi-iio utilizes pyinvoke. To see the available options (once pyinvoke is installed) run:

```
invoke --list
Available tasks:

  build           Build python package
  builddoc        Build sphinx doc
  changelog       Print changelog from last release
  checkparts      Check for missing parts in supported_parts.md
  createrelease   Create GitHub release
  libiiopath      Search for libiio python bindings
  precommit       Run precommit checks
  setup           Install required python packages for development through pip
  test            Run pytest tests
```

2.8.2 Precommit

pre-commit is heavily relied on for keeping code in order and for eliminating certain bugs. Be sure to run these checks before submitting code. This can be run through pyinvoke or directly from the repo root as:

```
invoke precommit
```

```
pre-commit run --all-files
```

2.8.3 Testing

Testing pyadi-iio requires hardware, but fortunately by default it assumes no hardware is connected unless found. It will only load specific tests for hardware it can find and skip all other tests. **pytest**, which is the framework pyadi-iio uses, can be call as following:

```
invoke test
```

```
python3 -m pytest <add more arguments as needed>
```

Test Configuration

When running tests a single URI can be provided to the command line. Devices can be dynamically scanned for on the network, and they can be provided through a configuration file. URIs for hardware are descripted in the **uri-map** section of the pyadi_test.yaml file with the convention "<uri>: hardware1, hardware2,..". Here is an example where the URI ip:192.168.2.1 applied to tests looking for the hardware adrv9361 or fmcomms2.

```
uri-map:
  "ip:192.168.86.35": adrv9361, fmcomms2
```

This file will automatically be loaded when it is in the location **/etc/default/pyadi_test.yaml** on Linux machines. Otherwise, it can be provided to pytest through the **–test-configfilename** argument.

2.9 Support

Question and general support related to pyadi-iio should be ask in the Software Interface Tools forum at ADI's EngineerZone. Code bugs or enhancement requests should be submitted through GitHub issues for the repository itself.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search