pydwf

Release 1.1.19

Sidney Cadot

Oct 18, 2023

INTRODUCTION

Warning: Important Notice

Following Digilent's decision in October 2023 to require creation of a login account to download the Waveforms software and the accompanying library that is needed to use their measurement devices, I decided to suspend my work on pydwf indefinitely. There will be no support for new features, no bug fixes, and no more user support from me either via Digilent's user forum or via other channels.

The primary reason for this is that I consider it unethical to sell a hardware device that requires accompanying software downloads if those software downloads require a mandatory login account, given that account creation (1) requires mandatory acceptance of Terms and Conditions that were not part of the original hardware sale; and (2) requires the user to register personally identifiable data.

I realize that these kinds of unsavory practices are pervasive in the brave new world of modern technology, but my motivation to add value for such devices in my free time is precisely zero.

So for the time being (and probably forever, since I don't expect Digilent to reverse course), version 1.1.19 will be the last pydwf release. It is essentially identical to version 1.1.18, with this notice prominently added.

Digilent used to be a pretty cool engineering-first company, providing very nice FPGA development boards and their excellent-value Analog Discovery devices. Unfortunately, their original customer oriented reputation is rapidly deteriorating by adopting customer-unfriendly practices, probably as a result of their acquisition by National Instruments back in 2013. It is all very unfortunate.

Winding down pydwf

- I have taken the Github repository private on 18 October 2023 to prevent uncoordinated forks.
- I am considering what to do with the existing pydwf packages on PyPI and the documentation on ReadTheDocs. For now they will remain, but at some point in the future I may remove them.

Given all this, if you are considering using pydwf for new projects: I'd advice against it.

If you have a dependency on pydwf, this may be a good time to consider ways to cut that dependency.

I apologise for the inconvenience.

CHAPTER

ONE

WELCOME TO *PYDWF* !

Warning: Important Notice

Following Digilent's decision in October 2023 to require creation of a login account to download the Waveforms software and the accompanying library that is needed to use their measurement devices, I decided to suspend my work on pydwf indefinitely. There will be no support for new features, no bug fixes, and no more user support from me either via Digilent's user forum or via other channels.

The primary reason for this is that I consider it unethical to sell a hardware device that requires accompanying software downloads if those software downloads require a mandatory login account, given that account creation (1) requires mandatory acceptance of Terms and Conditions that were not part of the original hardware sale; and (2) requires the user to register personally identifiable data.

I realize that these kinds of unsavory practices are pervasive in the brave new world of modern technology, but my motivation to add value for such devices in my free time is precisely zero.

So for the time being (and probably forever, since I don't expect Digilent to reverse course), version 1.1.19 will be the last pydwf release. It is essentially identical to version 1.1.18, with this notice prominently added.

Digilent used to be a pretty cool engineering-first company, providing very nice FPGA development boards and their excellent-value Analog Discovery devices. Unfortunately, their original customer oriented reputation is rapidly deteriorating by adopting customer-unfriendly practices, probably as a result of their acquisition by National Instruments back in 2013. It is all very unfortunate.

Winding down pydwf

- I have taken the Github repository private on 18 October 2023 to prevent uncoordinated forks.
- I am considering what to do with the existing pydwf packages on PyPI and the documentation on ReadTheDocs. For now they will remain, but at some point in the future I may remove them.

Given all this, if you are considering using pydwf for new projects: I'd advice against it.

If you have a dependency on pydwf, this may be a good time to consider ways to cut that dependency.

I apologise for the inconvenience.

This is the documentation of *pydwf*, a Python package to control the Digilent Waveforms lineup of electronic test and measurement devices made by Digilent.

It wraps all functions of *libdwf*, the *low-level C library* provided by Digilent, in an easy-to-use, class-based Python API. Like the C library, the *pydwf* package supports Windows, Linux (Intel and ARM), and macOS.

The DWF library can be downloaded and installed from Digilent's website.

The current release of *pydwf* is version 1.1.19. It is based on version 3.20.1 of *libdwf*, but it should also work with other versions.

The *pydwf* package comes with documentation and a number of ready-to-run *examples* that demonstrate how *pydwf* can be used to perform common and not-so-common tasks.

A *command-line tool* is provided that can be used, among other things, to list the available Digilent Waveforms devices and their configurations.

This section contains information about the project. Readers who want to learn how to use *pydwf* are referred to the *API documentation*.

1.1 Supported devices

The following devices can be controlled using *pydwf*:

- Electronics Explorer (legacy)
- Analog Discovery (legacy)
- Analog Discovery 2
- Analog Discovery 3
- Digital Discovery
- Analog Discovery Studio
- DPS3340 Discovery USB power supply
- Analog Discovery Pro 3x50 (3250 and 3450 models)
- Analog Discovery Pro 5250 (a National Instruments VB-8012 rebranded as a Digilent device; Windows only)

The *pydwf* package has been extensively tested with the Analog Discovery 2, Digital Discovery, and ADP3450 devices. It should also work with the other devices listed, but these haven't been tested. If you have such a device and encounter problems, please report an issue on the GitHub issue tracker.

1.2 Dependencies

The *pydwf* package requires Python 3.6 or higher.

In order for *pydwf* to work, recent versions of the Digilent Adept and Digilent Waveforms packages must be installed. These provide the C libraries that *pydwf* uses to interact with devices. Generally speaking, if the Waveforms GUI application provided by Digilent works on your system, you're good to go.

pydwf depends on the numpy package to handle the considerable amount of data transferred between the PC and Digilent Waveforms devices when performing high-speed signal generation or capture operations.

Some of the *examples* depend on the matplotlib package, but *pydwf* itself will work without it.

1.3 Project hosting

The project repository and issue tracker are hosted on GitHub:

https://github.com/sidneycadot/pydwf/

1.4 Installation using *pip*

The installable package is hosted on PyPI:

https://pypi.org/project/pydwf/

This allows installation using the standard *pip* (or *pip3*) tool:

```
pip install pydwf
```

After installing *pydwf*, the following command will show the version of *pydwf* and the underlying DWF library:

```
python -m pydwf version
```

The following command will list all Digilent Waveforms devices connected to the system and, for each of them, list the supported configurations:

```
python -m pydwf list -c
```

1.5 Documentation

The project documentation is hosted on Read The Docs. The latest version can be reached via the following link:

https://pydwf.readthedocs.io/en/latest/

If desired, the documentation can also be installed locally after installing the package by executing the following command:

```
python -m pydwf extract-html-docs
```

This will create a local directory called *pydwf-docs-html* containing the project documentation in HTML format.

Alternatively, a PDF version of the manual can be extracted as well:

```
python -m pydwf extract-pdf-manual
```

1.6 Examples

The Python *examples* can be installed locally after installing the *pydwf* package by executing the following command:

```
python -m pydwf extract-examples
```

This will create a local directory called *pydwf-examples* containing the Python examples that demonstrate many of the capabilities of the Digilent Waveforms devices and *pydwf*.

These examples are intended as a useful starting point for your own Python scripts. See the *examples overview* for more information.

1.7 Acknowledgements

Many thanks to Digilent for making the awesome Waveforms devices, and to provide not only the very capable *Waveforms* GUI software, but also the cross-platform SDK on which *pydwf* is based. Great work!

My company Jigsaw B.V. supported the effort to make *pydwf*. If you need any kind of high-tech software (with or without Digilent Waveforms devices), and you're somewhat in the vicinity of Delft, The Netherlands, give us a call.

Thanks to my longtime friend Pepijn for proof-reading the documentation and providing his perspective on several issues that came up while implementing *pydwf*. The package is a lot better because of your help.

Lastly, thanks to Petra for your patience with having all kinds of electronics equipment in the living room while developing this package (and before, and after, ...). You may not share my enthusiasm for this particular hobby, but I am very fortunate that you are at least enthusiastic about my enthusiasm, if that makes sense.

— *SC*

CHAPTER

TWO

OVERVIEW OF *PYDWF*

Warning: Important Notice

Following Digilent's decision in October 2023 to require creation of a login account to download the Waveforms software and the accompanying library that is needed to use their measurement devices, I decided to suspend my work on pydwf indefinitely. There will be no support for new features, no bug fixes, and no more user support from me either via Digilent's user forum or via other channels.

The primary reason for this is that I consider it unethical to sell a hardware device that requires accompanying software downloads if those software downloads require a mandatory login account, given that account creation (1) requires mandatory acceptance of Terms and Conditions that were not part of the original hardware sale; and (2) requires the user to register personally identifiable data.

I realize that these kinds of unsavory practices are pervasive in the brave new world of modern technology, but my motivation to add value for such devices in my free time is precisely zero.

So for the time being (and probably forever, since I don't expect Digilent to reverse course), version 1.1.19 will be the last pydwf release. It is essentially identical to version 1.1.18, with this notice prominently added.

Digilent used to be a pretty cool engineering-first company, providing very nice FPGA development boards and their excellent-value Analog Discovery devices. Unfortunately, their original customer oriented reputation is rapidly deteriorating by adopting customer-unfriendly practices, probably as a result of their acquisition by National Instruments back in 2013. It is all very unfortunate.

Winding down pydwf

- I have taken the Github repository private on 18 October 2023 to prevent uncoordinated forks.
- I am considering what to do with the existing pydwf packages on PyPI and the documentation on ReadTheDocs. For now they will remain, but at some point in the future I may remove them.

Given all this, if you are considering using pydwf for new projects: I'd advice against it.

If you have a dependency on pydwf, this may be a good time to consider ways to cut that dependency.

I apologise for the inconvenience.

All core *pydwf* functionality is made available for import from the top-level *pydwf* package:

- the `DwfLibrary` class, which is the starting point for all *pydwf* functionality;
- the `PyDwfError` and `DwfLibraryError` exceptions;
- the *enumeration types* that are used for parameters and result values of *pydwf* methods.

A small number of convenience functions and types have been implemented on top of the core *pydwf* package to simplify often-recurring tasks. These can be found in the *pydwf.utilities* package.

2.1 A minimal example of *pydwf* usage

In practice, Python scripts that use *pydwf* will deal almost exclusively with just two classes: *DwfLibrary* and *DwfDevice*.

The following is a minimal example of using *pydwf* that uses both of these classes to produce a 1 kHz tone on the first analog output channel:

```
"""A minimal, self-contained example of using pydwf."""

from pydwf import DwfLibrary, DwfAnalogOutNode, DwfAnalogOutFunction
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    CH1 = 0  # Analog-out channel numbering starts at zero.
    node = DwfAnalogOutNode.Carrier

    device.analogOut.reset(CH1)

    device.analogOut.nodeEnableSet(CH1, node, True)
    device.analogOut.nodeFunctionSet(CH1, node, DwfAnalogOutFunction.Sine)
    device.analogOut.nodeFrequencySet(CH1, node, 1000.0)

    # Start the channel.
    device.analogOut.configure(CH1, True)

    input("Producing a 1 kHz tone on CH1. Press Enter to quit ...")
```

With this example in mind, we can introduce the all-important *DwfLibrary* and *DwfDevice* classes.

2.2 The two main *pydwf* classes

As a *pydwf* user, you will interact directly with two classes: *DwfLibrary* and *DwfDevice*. We shortly summarize what they do here. They each have their own more comprehensive sections later on.

The *DwfLibrary* class

The *DwfLibrary* class represents the loaded Digilent Waveforms shared library itself, and provides methods that are not specific to a particular previously opened device. Examples include querying the library version, enumeration of devices, and opening a specific device for use.

Typically, a script will instantiate a single *DwfLibrary* and use that instance to open a specific Digilent Waveforms device, yielding a *DwfDevice* instance that can be used for the task at hand. This is also what happens in the example shown above.

A *DwfLibrary* instance provides a small number of top-level methods. It also provides some attributes that provide access to further functionality:

- *deviceEnum* provides device enumeration functionality;
- *deviceControl* provides functionality to open a single device and to close all previously opened devices;
- *spectrum* provides functionality for signal processing.

In most programs, the *DwfLibrary* class is only used to open a device for use, optionally selecting a specific *device configuration*. Since this is such an often-occurring operation, *pydwf* provides the `pydwf.utilities.openDwfDevice()` convenience function that handles several practical use-cases, such as opening a specific device by its serial number, and/or selecting a device configuration that maximizes the buffer size for a certain instrument.

A comprehensive description of the *DwfLibrary* class and its attributes can be found *here*.

The *DwfDevice* class

The `DwfDevice` class represents a specific Digilent Waveforms device, such as an Analog Discovery 2 or a Digital Discovery, connected to the computer.

Instances of *DwfDevice* are obtained either by calling on of the low-level `DeviceControl.open()` or `DeviceControl.openEx()` methods, or by calling the higher-level, more powerful `pydwf.utilities.openDwfDevice()` convenience function.

The *DwfDevice* class provides several miscellaneous methods, but the bulk of its functionality is accessible via one of the attributes listed below:

- `analogIn` provides a multi-channel oscilloscope;
- `analogOut` provides a multi-channel analog signal generator;
- `analogIO` provides voltage, current, and temperature monitoring and control;
- `analogImpedance` provides measurement of impedance and other quantities;
- `digitalIn` provides a multi-channel digital logic analyzer;
- `digitalOut` provides a multi-channel digital pattern generator;
- `digitalIO` provides static digital I/O functionality;
- `protocol.uart` provides UART protocol configuration, send, and receive functionality;
- `protocol.spi` provides SPI protocol configuration, send, and receive functionality;
- `protocol.i2c` provides I^2C protocol configuration, send, and receive functionality;
- `protocol.can` provides CAN protocol configuration, send, and receive functionality;
- `protocol.swd` provides SWD protocol configuration, send, and receive functionality.

After use, a Python script should `close()` the *DwfDevice*. Alternatively, the *DwfDevice* can act as a *context manager* for itself, to make sure it is closed whenever the containing *with* statement ends.

A comprehensive description of the *DwfDevice* class and its attributes can be found *here*.

CHAPTER

THREE

THE *DWFLIBRARY* CLASS AND ITS ATTRIBUTES

This section discusses the *DwfLibrary* class and the three attributes that provide access to most of its functionality.

3.1 The *DwfLibrary* class

The *DwfLibrary* class is the entry point to all *pydwf* functionality. Most importantly, it is needed to obtain *DwfDevice* instances.

3.1.1 Using the *DwfLibrary* class

The *DwfLibrary* class is defined in the *pydwf.core.dwf_library* module. The top-level *pydwf* package imports it from that module to make it available to user scripts. To use the `DwfLibrary` class, you should import it from the top-level *pydwf* package and create an instance:

```
from pydwf import DwfLibrary

dwf = DwfLibrary()

print("DWF library version:", dwf.getVersion())
```

After instantiating a *DwfLibrary*, you can use the handful of methods the instance provides. These methods are documented as part of the `DwfLibrary` class in the next section.

Three attributes are provided to access particular sub-APIs of a *DwfLibrary* instance:

- `deviceEnum` provides *device enumeration* functionality;
- `deviceControl` provides *device control* functionality;
- `spectrum` provides *signal processing* functionality.

In most programs, the *DwfLibrary* instance is only used for opening a *DwfDevice*, using either one of the `DeviceControl.open()` or `DeviceControl.openEx()` methods, or the `pydwf.utilities.openDwfDevice()` convenience function; the latter takes a *DwfLibrary* instance as a parameter.

3.1.2 *DwfLibrary* reference

class DwfLibrary

The *DwfLibrary* class provides access to miscellaneous library functionality through the handful of methods it provides, and to *device enumeration*, *device control*, and *signal processing* functionality via its `deviceEnum`, `deviceControl`, and `spectrum` attributes.

DwfLibrary attributes

deviceEnum

Provides access to the *device enumeration* functionality.

Type
DeviceEnumeration

deviceControl

Provides access to the *device control* functionality.

Type
DeviceControl

spectrum

Provides access to the *signal processing* functionality.

Type
Spectrum

DwfLibrary methods

__init__(*check_library_version: bool = False*) → None

Initialize a *DwfLibrary* instance.

A single *DwfLibrary* instance should be created by a user of *pydwf* to serve as an entry point to all *pydwf* functionality.

When initializing a *DwfLibrary*, the shared library *libdwf* on top of which *pydwf* is built is loaded into memory using Python's standard `ctypes` module.

A version check can be enabled to make sure that the shared library version corresponds exactly to the version that was used while developing and testing the current *pydwf* version (3.20.1), and an exception is raised if a mismatch is detected. Enabling this flag is recommended for critical applications.

After passing the version check (if enabled), the functions provided by the shared library are type-annotated. This means that calls into the shared library with incompatible parameter types will raise an exception. This mechanism helps to catch many bugs while using *pydwf*.

As a last initialization step, the `deviceEnum`, `deviceControl`, and `spectrum` attributes are initialized. They can be used by a user program to access the *device enumeration*, *device control* functionality, and *signal processing* functionality.

Parameters
check_library_version (*bool*) – If True, the version number of the C library will be checked against the version of the C library from which the type information used by *pydwf* was derived. In case of a mismatch, an exception will be raised.

Raises
PyDwfError – The version check could not be performed due to an unexpected low-level error while querying the shared library version, or a version mismatch was detected.

getLastError() → *DwfErrorCode*

Retrieve the last error code in the calling process.

The error code is cleared when other API functions are called and is only set when an API function fails during execution.

Note: When using *pydwf* there is no need to call this method directly, since low-level errors reported by the C library are automatically converted to a `DwfLibraryError` exception, which includes both the error code and the corresponding message.

Returns
The DWF error code of last API call.

Return type
DwfErrorCode

Raises
DwfLibraryError – the last error code cannot be retrieved.

getLastErrorMsg() → str

Retrieve the last error message.

The error message is cleared when other API functions are called and is only set when an API function fails during execution.

Note: When using *pydwf* there is no need to call this method directly, since low-level errors reported by the C library are automatically converted to a *DwfLibraryError* exception, which includes both the error code and the corresponding message.

Returns

The error message of the last API call.

The string may consist of multiple messages, separated by a newline character, that describe the events leading to the error.

Return type
str

Raises
DwfLibraryError – The last error message cannot be retrieved.

getVersion() → str

Retrieve the library version string.

Returns
The version of the DWF C library, composed of major, minor, and build numbers (e.g., "3.20.1").

Return type
str

Raises
DwfLibraryError – The library version string cannot be retrieved.

paramSet(*device_parameter:* DwfDeviceParameter, *value: int*) → None

Configure a default device parameter value.

Device parameters are settings of a specific *DwfDevice*. Refer to the *device parameters* section for more information.

This method sets a default device parameter value to be used for devices that are opened subsequently.

See also:

To set the parameter value of a specific *DwfDevice*, use the `DwfDevice.paramSet` method.

Warning: The device parameter values are not checked to make sure they correspond to a valid value for the specific device parameter.

Parameters

- **device_parameter** (DwfDeviceParameter) – The device parameter for which to set the default value.
- **value** (*int*) – The default device parameter value.

Raises
DwfLibraryError – The device parameter value cannot be set.

paramGet(*device_parameter:* DwfDeviceParameter) → int

Return a default device parameter value.

Device parameters are settings of a specific *DwfDevice*. Refer to the *device parameters* section for more information.

This method retrieves device parameter values at the library level (i.e., not tied to a specific device). They are used as default device parameter values for devices that are opened subsequently.

See also:

To get the parameter value of a specific *DwfDevice*, use the `DwfDevice.paramGet` method.

Parameters
device_parameter (*DwfParameter*) – The device parameter for which to get the value.

Returns
The retrieved device parameter value.

Return type
int

Raises
DwfLibraryError – The device parameter value cannot be retrieved.

3.2 Device enumeration functionality

The device enumeration functionality provides a way to enumerate accessible Digilent Waveforms devices, i.e., probe the USB bus and the network to find all Digilent Waveforms devices that can potentially be used.

The enumeration functionality also provides functionality to get basic information for all devices found, such as the device type, hardware revision, and serial number.

Lastly, the enumeration functionality allows user programs to examine *device configurations*, which provide a way to optimize the Digilent Waveforms device for a certain task by allocating resources such as buffer memory to certain instruments.

3.2.1 Using the device enumeration functionality

To use the device enumeration functionality you first need to initialize a *DwfLibrary* instance. The device enumeration functionality can then be accessed via its *deviceEnum* attribute, which is an instance of the *DeviceEnumeration* class:

```
from pydwf import DwfLibrary

dwf = DwfLibrary()

# Enumerate all Digilent Waveforms devices and return the count.
device_count = dwf.deviceEnum.enumerateDevices()

print("Number of Digilent Waveforms devices found:", device_count)
```

3.2.2 Alternatives to the device enumeration functionality

For most users, there is little reason to use the device enumeration API directly. Consider the following alternatives:

- From the command line, it is easy to obtain a list of all Digilent Waveforms devices and their configurations like this:

```
python -m pydwf list -c
```

- For Python scripts that want to select a specific device by serial number, or want to select a specific *device configuration* while opening the device to maximize the capabilities of a certain instrument (for example, to open the device with the largest possible *AnalogIn* or *AnalogOut* sample buffer sizes), consider using the `pydwf.utilities.openDwfDevice()` convenience function.

3.2.3 *DeviceEnumeration* reference

class DeviceEnumeration

The *DeviceEnumeration* class provides access to the device enumeration functionality of a `DwfLibrary`.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfLibrary* and subsequently assigned to its public `deviceEnum` attribute for access by the user.

enumerateDevices(*enum_filter:* DwfEnumFilter | *None* = *None*) → int

Build an internal list of available Digilent Waveforms devices and return the count of devices found.

This method must be called before using other *DeviceEnumeration* methods described below, because they obtain information about the enumerated devices from the internal device list that is built by this method.

> **Note:** This method can take several seconds to complete.

Parameters
enum_filter (*Optional[DwfEnumFilter]*) – Specify which devices to enumerate. If None, enumerate all devices.

Returns
The number of Digilent Waveforms devices detected.

Return type
int

Raises
DwfLibraryError – The Digilent Waveforms devices cannot be enumerated.

enumerateStart(*enum_filter:* DwfEnumFilter | *None* = *None*) → None

Start device enumeration.

> **Note:** This method is non-blocking (i.e., fast).

> **Note:** This method was added in DWF version 3.17 to provide an alternative to the blocking behavior of the `enumerateDevices()` method.

Parameters
enum_filter (*Optional[DwfEnumFilter]*) – Specify which devices to enumerate. If None, enumerate all devices.

Returns
The number of Digilent Waveforms devices detected.

Return type
int

Raises
DwfLibraryError – The Digilent Waveforms devices cannot be enumerated.

enumerateStop() → int

Stop device enumeration.

This should be called after a preceding *enumerateStart()* invocation.

A *enumerateStart()* call followed by a *enumerateStop()* call essentially performs the same function as a single *enumerateDevices()* call, and takes approximately the same amount of time.

The advantage of using the Start/Stop methods is that the application can do useful work while the devices are being enumerated in a background thread.

Note: This method can take several seconds to complete.

Note: This method was added in DWF version 3.17.

Returns
The number of Digilent Waveforms devices detected.

Return type
int

Raises
DwfLibraryError – The Digilent Waveforms devices cannot be enumerated.

enumerateInfo(*device_index: int*, *options: str*) → None

Get info of the current device.

It is not clear what this method does; the underlying DWF *FDwfEnumInfo* function is missing from the documentation.

An inquiry about this was made on the Digilent forum but the reply did not go into sufficient detail on the functionality provided.

Parameters

- **device_index** (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).
- **options** (*str*) – The function or format of this parameter is not known.

Todo: Figure out what this method does.

Note: This method was added in DWF version 3.17.

Raises
DwfLibraryError – The operation could not be performed.

deviceType(*device_index: int*) → Tuple[int, int]

Return the device ID and version (hardware revision) of the selected Digilent Waveforms device.

Note: This method returns the integer values as reported by the 'FDwfEnumDeviceType()' function and does not cast them to the *DwfDeviceID* and *DwfDeviceVersion* enumeration types.

This is done to prevent unknown devices from raising an exception.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
A tuple of the *DwfDeviceID* and *DwfDeviceVersion* integer values of the selected Digilent Waveforms device.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – The device type and version cannot be retrieved.

deviceIsOpened(*device_index: int*) → bool

Check if the specified Digilent Waveforms device is already opened by this or any other process.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
True if the Digilent Waveforms device is already opened, False otherwise.

Return type
bool

Raises
DwfLibraryError – The open state of the Digilent Waveforms device cannot be determined.

userName(*device_index: int*) → str

Retrieve the username of the selected Digilent Waveforms device.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The username of the Digilent Waveforms device, which is a short name indicating the device type (e.g., "Discovery2", "DDiscovery").

Return type
str

Raises
DwfLibraryError – The username of the Digilent Waveforms device cannot be retrieved.

deviceName(*device_index: int*) → str

Retrieve the device name of the selected Digilent Waveforms device.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The device name of the Digilent Waveforms device, which is a long name denoting the device type (e.g., "Analog Discovery 2", "Digital Discovery").

Return type
str

Raises
DwfLibraryError – The device name of the Digilent Waveforms device cannot be retrieved.

serialNumber(*device_index: int*) → str

Retrieve the 12-digit, unique serial number of the enumerated Digilent Waveforms device.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns

The 12 hex-digit unique serial number of the Digilent Waveforms device.

The 'SN:' prefix returned by the underlying C API function (for most devices) is discarded.

Return type
str

Raises

- ***DwfLibraryError*** – The serial number of the Digilent Waveforms device cannot be retrieved.
- ***PyDwfError*** – The serial number of the device is not 12 characters long.

enumerateConfigurations(*device_index: int*) → int

Build an internal list of detected configurations for the specified Digilent Waveforms device.

This method must be called before using the *configInfo()* method described below, because that method obtains information from the internal device configuration list that is built by this method.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The count of configurations of the Digilent Waveforms device.

Return type
int

Raises
DwfLibraryError – The configuration list of the Digilent Waveforms device cannot be retrieved.

configInfo(*config_index: int*, *info:* DwfEnumConfigInfo) → int | str

Return information about a Digilent Waveforms device configuration.

Parameters

- **config_index** (*int*) – Zero-based index of the previously enumerated configuration (see the *enumerateConfigurations()* method described above).
- **info** (DwfEnumConfigInfo) – Selects which configuration parameter to retrieve.

Note: For most values of the *info* parameter, this method returns an integer, but for some values it returns a string. Refer to the *DwfEnumConfigInfo* documentation for details.

This explains the somewhat unusual *Union[int, str]* return type of this method.

Returns
The value of the selected configuration parameter, of the selected configuration.

Return type
Union[*int*, *str*]

Raises
DwfLibraryError – The requested configuration information of the Digilent Waveforms device cannot be retrieved.

analogInChannels(*device_index: int*) → int

Return the analog input channel count of the selected Digilent Waveforms device.

Warning: This method is obsolete.

Use either of the following instead:

- method *configInfo()* to obtain the *DwfEnumConfigInfo.AnalogInChannelCount* configuration value;
- *AnalogIn.channelCount()*

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The number of analog input channels of the Digilent Waveforms device.

Return type
int

Raises
DwfLibraryError – The analog-in channel count of the Digilent Waveforms device cannot be retrieved.

analogInBufferSize(*device_index: int*) → int

Retrieve the analog input buffer size of the selected Digilent Waveforms device.

Warning: This method is obsolete.

Use either of the following instead:

- method *configInfo()* to obtain the *DwfEnumConfigInfo.AnalogInBufferSize* configuration value;
- *AnalogIn.bufferSizeGet()*

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The analog input buffer size of the selected Digilent Waveforms device.

Return type
int

Raises
DwfLibraryError – The analog-in buffer size of the Digilent Waveforms device cannot be retrieved.

analogInBits(*device_index: int*) → int

Retrieve the analog input bit resolution of the selected Digilent Waveforms device.

> **Warning: This method is obsolete.**
>
> Use *AnalogIn.bitsInfo()* instead.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The analog input bit resolution of the selected Digilent Waveforms device.

Return type
int

Raises
DwfLibraryError – The analog-in bit resolution of the Digilent Waveforms device cannot be retrieved.

analogInFrequency(*device_index: int*) → float

Retrieve the analog input sample frequency of the selected Digilent Waveforms device.

> **Warning: This method is obsolete.**
>
> Use *AnalogIn.frequencyInfo()* instead.

Parameters
device_index (*int*) – Zero-based index of the previously enumerated Digilent Waveforms device (see the *enumerateDevices()* method).

Returns
The analog input sample frequency of the selected Digilent Waveforms device, in samples per second.

Return type
float

Raises
DwfLibraryError – The analog input sample frequency of the Digilent Waveforms device cannot be retrieved.

property dwf

Return the *DwfLibrary* instance of which we are an attribute.

Returns
The *DwfLibrary* instance.

Return type
DwfLibrary

3.3 Device control functionality

The device control functionality provides a way to open specific Digilent Waveforms devices, and to close all previously opened devices. It is complemented by the methods provided be the *DwfDevice* class.

3.3.1 Using the device control functionality

To use the device control functionality you first need to initialize a *DwfLibrary* instance. The device control functionality can then be accessed via its *deviceControl* attribute, which is an instance of the *DeviceControl* class:

```
from pydwf import DwfLibrary

dwf = DwfLibrary()

# Open the first available Digilent Waveforms device, and close it immediately.
device = dwf.deviceControl.open(-1)
device.close()
```

3.3.2 Alternatives to the device control functionality

For most users, there is little reason to use the device control API directly. Consider the following alternatives:

- The *DeviceControl.open()* method is occasionally useful, but the *pydwf.utilities.openDwfDevice()* convenience function provides a more powerful alternative.
- The *DeviceControl.closeAll()* method is not recommended for general use. Devices can and should be closed individually, either by calling their *DwfDevice.close()* method explicitly, or by using their *context manager* feature.
- The device control API of the underlying C library supports several more functions that work on a previously opened Digilent Waveforms device. In *pydwf*, these functions are available as methods of the *DwfDevice* class.

3.3.3 *DeviceControl* reference

class DeviceControl

The *DeviceControl* class provides access to the device control functionality of a *DwfLibrary*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfLibrary* and subsequently assigned to its public *deviceControl* attribute for access by the user.

open(*device_index: int*, *config_index: int | None = None*) → *DwfDevice*

Open a Digilent Waveforms device identified by the device index, using a specific device configuration index if specified.

> **Note:** This method combines the functionality of the C API functions 'FDwfDeviceOpen()' and 'FDwfDeviceConfigOpen()' into a single method. The call that is actually made depends on the value of the *config_index* parameter.

Note: This method can take several seconds to complete.

Parameters

- **device_index** (*int*) – The zero-based index of the previously enumerated device (see the *DeviceEnum.enumerateDevices()* method).

  To automatically enumerate all connected devices and open the first discovered device, use the value -1 for this parameter.

- **config_index** (*Optional[int]*) – The zero-based index of the device configuration to use (see the *DeviceEnum.enumerateConfigurations()* method). If None, open the default (first) device configuration.

See also:

The *pydwf.utilities.openDwfDevice()* convenience function provides a more powerful way to select and open a device and, if desired, specify its device configuration.

Returns
The *DwfDevice* instance created as a result of this call.

Return type
DwfDevice

Raises
DwfLibraryError – The specified device or configuration cannot be opened.

openEx(*options: str*, *separator: str = ','*) → *DwfDevice*

Open a device using options given as a string.

This provides an extended (hence the 'Ex' prefix) version of the *:py:meth:open* method.

The following table lists the options that can be specified in the *options* string, as listed in the DWF documentation. Not all of these have been verified to work.

Table 1: Options for the openEx method

option	description
index:#	Connect to device by enumeration 0 based index.
sn:##########	Open by serial number. (The devices will be enumerated).
name:*device-name*	Open by device name number. (The devices will be enumerated).
config:#	Use configuration (0-based index).
ip:#.#.#.#/host	Connect to network device identified by IP address or hostname.
ip:*user*:*pass**@#.#.#.#/**host*	Connect to network device (with username and password).
user:*username*	Provide username.
pass:*password*	Provide password.
secure:#	Enable TLS communication encryption (0/1).

Todo: Figure out the precise syntax and semantics of options.

Note: This method was added in DWF version 3.17.

Parameters

- **options** (*str*) – A string listing options, separated by a 'separator' string (see below).

- **separator** (*str*) – The separator string that separating options. Default: a single comma (‘,’).

Returns
The *DwfDevice* instance created as a result of this call.

Return type
DwfDevice

Raises
DwfLibraryError – The operation could not be performed.

closeAll() → None

Close all Digilent Waveforms devices opened by the calling process.

This method does not close all Digilent Waveforms devices across all processes.

Raises
DwfLibraryError – The *close all* operation failed.

property dwf

Return the *DwfLibrary* instance of which we are an attribute.

Returns
The *DwfLibrary* instance.

Return type
DwfLibrary

3.4 Signal processing functionality

The signal processing functionality provides several standard functions for processing digitized analog signals.

3.4.1 Using the signal processing functionality

To use the signal processing functionality you first need to initialize a *DwfLibrary* instance. The signal processing functionality can then be accessed via its *spectrum* attribute, which is an instance of the *Spectrum* class:

```
from pydwf import DwfLibrary, DwfWindow

dwf = DwfLibrary()

# Make an 11-element Hamming window.
(hamming_window, noise_equivalent_bandwidth) = dwf.spectrum.window(11, DwfWindow.
↪Hamming)
```

3.4.2 *Spectrum* reference

class Spectrum

The *Spectrum* class provides access to the signal processing functionality of a *DwfLibrary*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfLibrary* and subsequently assigned to its public *spectrum* attribute for access by the user.

window(*n: int*, *winfunc:* DwfWindow, *beta: float* | *None* = *None*) → Tuple[ndarray, float]

Return the window coefficients for the specified window, and the window's noise-equivalent bandwidth.

Parameters

- **n** (*int*) – The length of the window to be generated.
- **winfunc** (DwfWindow) – The type of window to be generated.
- **beta** (*Optional[float]*) – Parameter for the Kaiser window; unused for other windows. If not specified, a value of 0.5 is used.

Returns

A two-element tuple. The first element of the tuple is a numpy array of length *n*, containing the window coefficients. All windows supported are symmetric, and they are scaled to have the sum of their values equal to *n*.

The second element of the tuple is the noise equivalent bandwidth of the signal. For a coefficient array *w*, this value is equal to *len(w) * np.sum(w**2) / np.sum(w)**2*.

Return type

Tuple[np.ndarray, *float*]

Note: When working in Python, it may be better to use the functionality provided in the scipy.signal.windows package instead.

---

fft(*data: ndarray*) → Tuple[ndarray, ndarray]

Perform a Fast Fourier Transform.

Parameters

data – The real-valued data to be transformed. This must be a 1-dimensional array with a length that is a power of 2.

Returns

A tuple (magnitude, phase); both the magnitude and phase arrays are real-valued 1D numpy arrays with the same length as the input array *data*. The first array contains the non-negative magnitude of the signal for this particular frequency; the second array contains its phase.

Return type

Tuple[np.ndarray, np.ndarray]

Note: The scaling of the FFT calculated is unusual. Compared to Matlab, numpy, and most other implementations, it is scaled by a factor (2 / n), with n the number of points in the input array.

To get from the (magnitude, phase) representation as returned by this function to a vector of complex number that is comparable to what e.g. the *numpy.fft.rfft* function returns, you can do:

z = (len(magnitude) / 2.0) * magnitude * np.exp(phase * 1j)

When working in Python, it may be better to simply use the numpy.fft.rfft() function directly. It is faster and provides support for data vectors with a length that is not a power of two.

---

transform(*data: ndarray*, *num_bins: int*, *first_freq: float*, *last_freq: float*) → Tuple[ndarray, ndarray]

Perform a Chirp-Z transform.

Parameters

- **data** – The real-valued data to be transformed.
- **num_bins** (*int*) – The number of bins to be calculated.

- **first_freq** (*float*) – The frequency of the first bin, scaled to 0.5 * sample_frequency.
- **1** (*Should be between 0 and*) –
- **inclusive.** –
- **last_freq** (*float*) – The frequency of the last bin, scaled to 0.5 * sample_frequency.
- **1** –
- **inclusive.** –

Returns
A tuple (magnitude, phase); both the magnitude and phase arrays are real-valued 1D numpy arrays with the same length as the input array *data*. The first array contains the non-negative magnitude of the signal for this particular frequency; the second array contains its phase.

Return type
Tuple[np.ndarray, np.ndarray]

Note: The scaling of the FFT calculated is unusual. Compared to Matlab, numpy, and most other implementations, it is scaled by a factor (2 / n), with n the number of points in the input array.

To get from the (magnitude, phase) representation as returned by this function to a vector of complex number that is comparable to what e.g. the *scipy.signal.czt* function returns, you can do:

z = (len(data) / 2.0) * magnitude * np.exp(phase * 1j)

When working in Python, it may be better to simply use the scipy.signal.czt() function directly.

---

property dwf

Return the *DwfLibrary* instance of which we are an attribute.

Returns
The *DwfLibrary* instance.

Return type
DwfLibrary

CHAPTER

FOUR

THE *DWFDEVICE* CLASS AND ITS ATTRIBUTES

This section discusses the *DwfDevice* class and the twelve attributes that provide access to most of its functionality.

4.1 The *DwfDevice* class

The *DwfDevice* class represents a previously opened Digilent Waveforms device. It is the entry point to all useful functionality of the Digilent Waveforms device.

4.1.1 Using the *DwfDevice* class

To obtain a *DwfDevice* instance you first need to initialize a *DwfLibrary* instance. The *DwfLibrary* can then be used to obtain a *DwfDevice*, either by using the *DeviceControl.open()* method or by using the *pydwf.utilities.openDwfDevice()* convenience function.

After the program is done using a device, it should be closed. This can be done explicitly, via the *DwfDevice.close()* method, or implicitly, by using the *DwfDevice* as a so-called *context manager* for itself. The latter method is often preferable, since it guarantees that the device will be closed even when an exception occurs while using it:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    # When we leave the 'with' statement, the device is guaranteed to be closed.
    print("Trigger sources supported by this device:", device.triggerInfo())

    # This is true even if an exception is raised inside the 'with' statement's body:
    raise RuntimeError("yikes!")
```

After obtaining a *DwfDevice*, you can use the dozen or so methods it provides. These methods are documented as part of the *DwfDevice* class in the next section.

Twelve attributes are provided to access particular sub-APIs of a *DwfDevice* instance. Depending on the type of task that you are using your Digilent Waveforms device for, one or several of these attributes will be your main handle to configure instruments and to send or receive data:

- *analogIn* provides a multi-channel oscilloscope;
- *analogOut* provides a multi-channel analog signal generator;
- *analogIO* provides voltage, current, and temperature monitoring and control;
- *analogImpedance* provides measurement of impedance and other quantities;
- *digitalIn* provides a multi-channel digital logic analyzer;
- *digitalOut* provides a multi-channel digital pattern generator;

- *digitalIO* provides static digital I/O functionality;
- *protocol.uart* provides UART protocol configuration, send, and receive functionality;
- *protocol.spi* provides SPI protocol configuration, send, and receive functionality;
- *protocol.i2c* provides I^2C protocol configuration, send, and receive functionality;
- *protocol.can* provides CAN protocol configuration, send, and receive functionality;
- *protocol.swd* provides SWD protocol configuration, send, and receive functionality.

4.1.2 *DwfDevice* reference

class DwfDevice

The *DwfDevice* represents a single Digilent Waveforms test and measurement device.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> Use *DeviceControl.open()* or *pydwf.utilities.openDwfDevice()* to obtain a valid *DwfDevice* instance.

The main test and measurement functionality of a Digilent Waveforms device is provided as multiple sub-interfaces (instruments, protocols, and measurements). To access those, use one of the twelve attributes described below.

DwfDevice attributes

analogIn

Provides access analog input (oscilloscope) functionality.

Type
AnalogIn

analogOut

Provides access to the analog output (waveform generator) functionality.

Type
AnalogOut

analogIO

Provides access to the analog I/O (voltage source, monitoring) functionality.

Type
AnalogIO

analogImpedance

Provides access to the analog impedance measurement functionality.

Type
AnalogImpedance

digitalIn

Provides access to the dynamic digital input (logic analyzer) functionality.

Type
DigitalIn

digitalOut

Provides access to the dynamic digital output (pattern generator) functionality.

Type
DigitalOut

digitalIO

Provides access to the static digital I/O functionality.

Type
DigitalIO

protocol.uart

Provides access to the UART protocol functionality.

Type
ProtocolUART

protocol.can

Provides access to the CAN protocol functionality.

Type
ProtocolCAN

protocol.spi

Provides access to the SPI protocol functionality.

Type
ProtocolSPI

protocol.i2c

Provides access to the I^2C protocol functionality.

Type
ProtocolI2C

protocol.swd

Provides access to the SWD protocol functionality.

Type
ProtocolSWD

DwfDevice properties and methods

property dwf: *DwfLibrary*

Return the *DwfLibrary* instance that was used to create (open) this *DwfDevice* instance.

This is useful if we have a *DwfDevice*, but we need its *DwfLibrary*.

Returns
The *DwfLibrary* that was used to create (open) this *DwfDevice* instance.

Return type
DwfLibrary

property digitalUart: *ProtocolUART*

Old attribute-style access to the device's UART functionality.

> **Warning:** This attribute is obsolete. Use *protocol.uart* instead.

property digitalSpi: *ProtocolSPI*

Old attribute-style access to the device's SPI functionality.

> **Warning:** This attribute is obsolete. Use *protocol.spi* instead.

property digitalI2c: *ProtocolI2C*

Old attribute-style access to the device's I^2C functionality.

> **Warning:** This attribute is obsolete. Use *protocol.i2c* instead.

property digitalCan: *ProtocolCAN*

Old attribute-style access to the device's CAN functionality.

> **Warning:** This attribute is obsolete. Use *protocol.can* instead.

property digitalSwd: *ProtocolSWD*

Old attribute-style access to the device's SWD functionality.

> **Warning:** This attribute is obsolete. Use *DwfDevice.protocol.swd* instead.

close() → None

Close the device.

This method should be called when access to the device is no longer needed.

Once this method returns, the *DwfDevice* can no longer be used.

Raises
DwfLibraryError – The device cannot be closed.

autoConfigureSet(*auto_configure: int*) → None

Enable or disable the autoconfiguration setting of the device.

When this setting is enabled (the default), any change to an instrument setting is automatically transmitted to the Digilent Waveforms hardware device, without the need for an explicit call to the instrument's *configure()* method.

This adds a small amount of latency to every *Set()* method; just as much latency as calling the corresponding *configure()* method explicitly after the *Set()* method would add.

Autoconfiguration is enabled by default, and there is little reason to turn it off unless the user program wants to make frequent changes to many settings at once between measurements.

With value 3, configuration will be applied dynamically, without stopping the instrument.

Parameters
auto_configure (*int*) – The new autoconfiguration setting.

Possible values for this option:

- 0 — disable
- 1 — enable
- 3 — dynamic

Raises
DwfLibraryError – The value cannot be set.

autoConfigureGet() → int

Return the autoconfiguration setting of the device.

Returns

The current autoconfiguration setting.

Possible values for this option:

- 0 — disable
- 1 — enable
- 3 — dynamic

Return type
int

Raises
DwfLibraryError – The value cannot be retrieved.

reset() → None

Reset all device and instrument settings to default values.

The new settings are applied immediately if autoconfiguration is enabled.

Raises
DwfLibraryError – The device cannot be reset.

enableSet(*enable: bool*) → None

Enable or disable the device.

Parameters
enable (*bool*) – True for enable, False for disable.

Raises
DwfLibraryError – The device's enabled state cannot be set.

triggerInfo() → list[*DwfTriggerSource*]

Return the available trigger source options for the global trigger bus.

Refer to the section on *triggering* for more information.

The four main instruments (*AnalogIn*, *AnalogOut*, *DigitalIn*, and *DigitalOut*) can be configured to start their operation (data acquisition for the *In* instruments; signal generation for the *Out* instruments) immediately after some event happens. This is called *triggering*.

Each of the instruments can be configured independently to use any of the trigger sources available inside the device. This method returns a list of all trigger sources that are available to each of the instruments.

Returns
A list of available trigger sources.

Return type
list[DwfTriggerSource]

Raises
DwfLibraryError – The list of supported trigger sources cannot be retrieved.

triggerSet(*pin_index: int*, *trigger_source:* DwfTriggerSource) → None

Configure the trigger I/O pin with a specific trigger source option.

Digilent Waveforms devices have dedicated digital I/O pins that can be used either as trigger inputs or trigger outputs. Use this method to select which line of the global triggering bus is driven on those pins, e.g. to trigger some external device or to monitor the Digilent Waveforms device's internal trigger behavior.

Pass `DwfTriggerSource.None_` to disable trigger output on the pin. This is the default setting, and the appropriate value to use when the intention is to have some external trigger signal drive the pin.

Refer to the section on *triggering* for more information.

Parameters

- **pin_index** (*int*) – The trigger pin to configure.
- **trigger_source** (DwfTriggerSource) – The trigger source to select.

Raises

DwfLibraryError – The trigger source cannot be set.

triggerGet(*pin_index: int*) → *DwfTriggerSource*

Return the selected trigger source for a trigger I/O pin.

Refer to the section on *triggering* for more information.

Parameters

pin_index (*int*) – The pin for which to obtain the selected trigger source.

Returns

The trigger source setting for the selected pin.

Return type

DwfTriggerSource

Raises

DwfLibraryError – The trigger source cannot be retrieved.

triggerPC() → None

Generate a trigger pulse on the PC trigger line.

The generated pulse will trigger any instrument that is configured with trigger source *DwfTriggerSource.PC*, and currently armed (i.e., waiting for a trigger).

Raises

DwfLibraryError – The PC trigger line cannot be pulsed.

triggerSlopeInfo() → list[*DwfTriggerSlope*]

Return the supported trigger slope options.

Returns

A list of supported trigger slope values.

Return type

list[DwfTriggerSlope]

Raises

DwfLibraryError – The trigger slope options cannot be retrieved.

paramSet(*parameter:* DwfDeviceParameter, *value: int*) → None

Set a device parameter value.

Device parameters are settings of a specific *DwfDevice*. Refer to the *device parameters* section for more information.

This method sets a device parameter value of a currently opened *DwfDevice*.

> **Warning:** The device parameter values are not checked to make sure they correspond to a valid value for the current device.

Parameters

- **parameter** (DwfDeviceParameter) – The device parameter to set.
- **value** (*int*) – The value to assign to the parameter.

Raises

DwfLibraryError – The specified device parameter cannot be set.

paramGet(*parameter:* DwfDeviceParameter) → int

Get a device parameter value.

Device parameters are settings of a specific *DwfDevice*. Refer to the *device parameters* section for more information.

Parameters
parameter (DwfDeviceParameter) – The device parameter to get.

Returns
The integer value of the parameter.

Return type
int

Raises
DwfLibraryError – The value of the specified device parameter cannot be retrieved.

4.2 Analog input instrument

The *AnalogIn* instrument provides multiple channels of analog input on devices that support it, such as the Analog Discovery and the Analog Discovery 2. It provides the functionality normally associated with a stand-alone oscilloscope.

Todo: This section is missing some important information:

- A discussion about the different acquisition modes;
- A description of how the status variables behave in the different acquisition modes;
- A discussion of the precise meaning of all settings.

4.2.1 Using the analog input instrument

To use the *AnalogIn* instrument you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *AnalogIn* instrument can now be accessed via its *analogIn* attribute, which is an instance of the *AnalogIn* class.

For example:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's AnalogIn instrument.
    analogIn = device.analogIn

    # Use the AnalogIn instrument.
    analogIn.reset()
```

4.2.2 The *AnalogIn* state machine

The *AnalogIn* instrument is controlled by a state machine. As a measurement is prepared and executed, the instrument goes through its various states.

The current state of the instrument is returned by the `analogIn.status()` method, and is of type `DwfState`.

The figure below shows the states used by the *AnalogIn* instrument and the transitions between them:

Fig. 1: States of the *AnalogIn* instrument

The *AnalogIn* states are used as follows:

1. `Ready`

   In this preparatory state, instrument settings can be changed that specify the behavior of the instrument in the coming measurement. If the auto-configure setting of the device is enabled (the default), setting changes will automatically be transferred to the device. If not, an explicit call to the `analogIn.configure()` method with the *reconfigure* parameter set to True is needed to transfer updated settings to the device.

   Once the instrument is properly configured, an acquisition can be started by calling the `analogIn.configure()` with the *start* parameter set to True. This will start the first stage of the acquisition by entering the `Prefill` state.

2. `Configure`

   This state is entered momentarily when a setting is being pushed to the device, either by changing the setting while auto-configure is enabled, or by an explicit call to `analogIn.configure()` with the *reconfigure* parameter set to True. The settings inside the device will be updated, and the device will immediately thereafter go back to the `Ready` state, unless the *start* parameter to `analogIn.configure()` was set to True.

3. `Prefill`

   This state marks the beginning of an acquisition sequence. During the `Prefill` state, input samples will be acquired until enough samples are buffered for the instrument to be ready to react to a trigger.

   This state is only relevant if the trigger position has been configured in such a way that the measurement must also yield sample values prior to the moment of triggering.

   Once enough samples are received for the instrument to be able to react to a trigger, it proceeds to the `Armed` state.

4. `Armed`

   In this state the instrument continuously captures samples and monitors the configured trigger input. As soon as a trigger event is detected, the instrument proceeds to the `Running` state.

5. `Running`

   In this state the instrument continues capturing samples until the acquisition is complete. Completion is reached when the acquisition buffer has filled up in `Single` mode, or when the recording length has been reached in `Record` mode. When completion is reached, the instrument proceeds to the `Done` state.

6. `Done`

   This state indicates that a measurement has finished.

   From this state, it is possible to go back to the `Ready` state by performing any kind of configuration, or to start a new acquisition with the same settings.

4.2.3 *AnalogIn* instrument API overview

With 101 methods, the *AnalogIn* instrument is the most complicated instrument supported by the Digilent Waveforms API. Below, we categorize all its methods and shortly introduce them. Detailed information on all methods can be found in the `AnalogIn` class reference that follows.

Instrument control

Like all instruments supported by the Digilent Waveforms library, the *AnalogIn* instrument provides `reset()`, `configure()`, and `status()` methods.

The `reset()` method resets the instrument.

The `configure()` method is used to explicitly transfer settings to the instrument, and/or to start a configured operation.

The `status()` method retrieves status information from the instrument. Optionally, it can also retrieve bulk data, i.e., analog signal and noise samples. The method returns the current `DwfState` of the *AnalogIn* instrument; to obtain more elaborate status information, one of the methods in the next two sections must be used.

Table 1: Instrument control (3 methods)

control operation	type/unit	methods
reset instrument	*n/a*	`reset()`
configure instrument	*n/a*	`configure()`
request instrument status	`DwfState`	`status()`

Status variables

When executing the `status()` method, status information is transferred from the *AnalogIn* instrument to the PC. Several status variables can then be retrieved by using the methods listed below.

Table 2: Status variables (7 methods)

status value	type/unit	method
timestamp	tuple [s]	`statusTime()`
most recent sample value	float [V]	`statusSample()`
auto-triggered flag	bool	`statusAutoTriggered()`
samples left in acquisition	int [samples]	`statusSamplesLeft()`
samples valid count	int [samples]	`statusSamplesValid()`
buffer write index	int [samples]	`statusIndexWrite()`
recording status	tuple [samples]	`statusRecord()`

Status data retrieval

Executing the `status()` method with the *read_data* parameter set to True transfers captured samples from the instrument to the PC. The samples can then be retrieved using the methods listed here.

Table 3: Status data retrieval (5 methods)

status data	type/unit	methods
get sample data (without buffer offset)	float [V]	`statusData()`
get sample data (with buffer offset)	float [V]	`statusData2()`
get sample data (raw samples, with buffer offset)	int [-]	`statusData16()`
get sample noise (without buffer offset)	float [V]	`statusNoise()`
get sample noise (with offset)	float [V]	`statusNoise2()`

Acquisition settings

The following methods are used to get and set channel-independent configuration values related to acquisition, and to obtain information about their possible values.

Table 4: Acquisition settings (15 methods)

setting	type/unit	methods
ADC sample resolution	int [bits]	*bitsInfo()*
record length	float [s]	*recordLengthSet()* , *-Get()*
sample frequency	float [Hz]	*frequencyInfo()* , *-Set()* , *-Get()*
sample buffer size	int [samples]	*bufferSizeInfo()* , *-Set()* , *-Get()*
noise buffer size	int [samples]	*noiseSizeInfo()* , *-Set()* , *-Get()*
acquisition mode	*DwfAcquisitionMode*	*acquisitionModeInfo()* , *-Set()* , *-Get()*

Channel count

This method returns the number of analog input channels.

Table 5: Channel count (2 methods)

operation	type/unit	method
channel count channel count, distinguish real/filter channels	int int	*channelCount()* *channelCount()*

Channel configuration

The following methods are used to get and set channel-dependent configuration values, and to obtain information about their possible values.

Table 6: Channel configuration (21 methods)

setting	type/unit	methods
channel enable	bool	*channelEnableSet()* , *-Get()*
channel filter	*DwfAnalogInFilter*	*channelFilterInfo()* , *-Set()* , *-Get()*
channel range	float [V]	*channelRangeInfo()* , *-Set()* , *-Get()* , *-Steps()*
channel offset	float [V]	*channelOffsetInfo()* , *-Set()* , *-Get()*
channel attenuation	float [-]	*channelAttenuationSet()* , *-Get()*
channel bandwidth	float [Hz]	*channelBandwidthSet()* , *-Get()*
channel impedance	float [Ohms]	*channelImpedanceSet()* , *-Get()*
channel coupling	*DwfAnalogCoupling*	*channelCouplingInfo()* , *-Get()* , *-Get()*

Instrument trigger configuration

The following methods are used to configure the trigger of the *AnalogIn* instrument. The trigger source is fully configurable; the *AnalogIn* instrument can use its own trigger detector for triggering, but it is also possible to use a different trigger source. For that reason, we distinguish between the methods that configure the instrument trigger, and the methods that configure the *AnalogIn* trigger detector that are discussed below.

Table 7: Instrument trigger configuration (10 methods)

setting	type/unit	methods
trigger source	*DwfTriggerSource*	*triggerSourceInfo()* , *-Set()* , *-Get()*
trigger position	float [s]	*triggerPositionInfo()* , *-Set()* , *-Get()* , *-Status()*
trigger auto-timeout	float [s]	*triggerAutoTimeoutInfo()* , *-Set()* , *-Get()*

Note: The *triggerSourceInfo()* method is obsolete. Use the generic *DwfDevice.triggerInfo()* method instead.

Force instrument trigger

The *triggerForce()* method can be used to force the *AnalogIn* instrument to start acquiring.

Table 8: Force instrument trigger (1 method)

operation	type/unit	method
force trigger	*n/a*	*triggerForce()*

Trigger detector configuration

The *AnalogIn* trigger detector is highly configurable. It has nine different settings that can be queried and set using the methods below.

Table 9: Trigger detector configuration (27 methods)

setting	type/unit	methods
trigger hold-off	float [s]	*triggerHoldOffInfo()* , *-Set()* , *-Get()*
trigger type	*DwfAnalogInTriggerType*	*triggerTypeInfo()* , *-Set()* , *-Get()*
trigger channel	int [-]	*triggerChannelInfo()* , *-Set()* , *-Get()*
trigger filter	*DwfAnalogInFilter*	*triggerFilterInfo()* , *-Set()* , *-Get()*
trigger level	float [V]	*triggerLevelInfo()* , *-Set()* , *-Get()*
trigger hysteresis	float [V]	*triggerHysteresisInfo()* , *-Set()* , *-Get()*
trigger slope	*DwfTriggerSlope*	*triggerConditionInfo()* , *-Set()* , *-Get()*
trigger length	float [s]	*triggerLengthInfo()* , *-Set()* , *-Get()*
trigger length condition	*DwfAnalogInTriggerLengthCondi*	*triggerLengthConditionInfo()* , *-Set()* , *-Get()*

Counter functionality

Table 10: Counter configuration (4 methods)

setting	type/unit	methods
counter configuration	float [s], int [-] float [s], float [Hz], int [-]	*counterInfo()* , *-Set()* , *-Get()*
counter status		*counterStatus()*

Sampling clock configuration

The *AnalogIn* instrument can use a sampling clock that is different from the internally generated clock that it would normally use. Three settings determine its behavior.

Table 11: Sampling clock configuration (6 methods)

setting	type/unit	methods
sampling source	*DwfTriggerSource*	*samplingSourceSet()* , *-Get()*
sampling slope	*DwfTriggerSlope*	*samplingSlopeSet()* , *-Get()*
sampling delay	float [s]	*samplingDelaySet()* , *-Get()*

4.2.4 *AnalogIn* reference

class AnalogIn

The *AnalogIn* class provides access to the analog input (oscilloscope) instrument of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *analogIn* attribute for access by the user.

reset() → None

Reset all *AnalogIn* instrument parameters to default values.

If autoconfiguration is enabled at the device level, the reset operation is performed immediately; otherwise, an explicit call to the *configure()* method is required.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

configure(*reconfigure: bool*, *start: bool*) → None

Configure the instrument and start or stop the acquisition operation.

Parameters

- **reconfigure** (*bool*) – If True, the instrument settings are sent to the instrument. In addition, the auto-trigger timeout is reset.
- **start** (*bool*) – If True, an acquisition is started. If False, an ongoing acquisition is stopped.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status(*read_data_flag: bool*) → *DwfState*

Get the *AnalogIn* instrument state.

This method performs a status request to the *AnalogIn* instrument and receives its response.

The following methods can be used to retrieve *AnalogIn* instrument status information as a result of this call, regardless of the value of the *read_data_flag* parameter:

- *statusTime()*
- *statusSample()*
- *statusAutoTriggered()*
- *statusSamplesLeft()*
- *statusSamplesValid()*
- *statusIndexWrite()*
- *statusRecord()*

The following methods can be used to retrieve bulk data obtained from the *AnalogIn* instrument as a result of this call, but only if the *read_data_flag* parameter is True:

- *statusData()*
- *statusData2()*
- *statusData16()*
- *statusNoise()*
- *statusNoise2()*

Parameters
read_data_flag (*bool*) – If True, read sample data from the instrument.

In *Single* acquisition mode, the data will be read only when the acquisition is finished.

Returns
The status of the *AnalogIn* instrument.

Return type
DwfState

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

statusTime() → Tuple[int, int, int]

Retrieve the timestamp of the current status information.

Returns

A three-element tuple, indicating the POSIX timestamp of the status request. The first element is the POSIX second, the second and third element are the numerator and denominator, respectively, of the fractional part of the second.

In case *status()* hasn't been called yet, this method will return zeroes for all three tuple elements.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while retrieving the status time.

statusSample(*channel_index: int*) → float

Get the last ADC conversion sample from the specified *AnalogIn* instrument channel, in Volts.

Note: This value is updated even if the *status()* method is called with argument False.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
The most recent ADC value of this channel, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while retrieving the sample value.

statusAutoTriggered() → bool

Check if the current acquisition is auto-triggered.

Returns
True if the current acquisition is auto-triggered, False otherwise.

Return type
bool

Raises
DwfLibraryError – An error occurred while retrieving the auto-triggered status.

statusSamplesLeft() → int

Retrieve the number of samples left in the acquisition, in samples.

Returns
In case a finite-duration acquisition is active, the number of samples remaining to be acquired in the acquisition.

Return type
int

Raises
DwfLibraryError – An error occurred while retrieving the number of samples left.

statusSamplesValid() → int

Retrieve the number of valid acquired data samples.

In *Single* acquisition mode, valid samples are returned when *status()* reports a result of *Done*.

The actual number of samples transferred and reported back here is equal to max(16, *bufferSizeGet()*).

Returns
The number of valid samples.

Return type
int

Raises
DwfLibraryError – An error occurred while retrieving the number of valid samples.

statusIndexWrite() → int

Retrieve the buffer write index.

This is needed in *ScanScreen* acquisition mode to display the scan bar.

Returns
The buffer write index.

Return type
int

Raises
DwfLibraryError – An error occurred while retrieving the write-index.

statusRecord() → Tuple[int, int, int]

Retrieve information about the recording process.

Data loss occurs when the device acquisition is faster than the read process to the PC.

If this happens, the device recording buffer is filled and data samples are overwritten.

Corrupt samples indicate that the samples have been overwritten by the acquisition process during the previous read.

In this case, try optimizing the loop process for faster execution or reduce the acquisition frequency or record length to be less than or equal to the device buffer size (i.e., record_length is less than or equal to buffer_size / sample_frequency).

Returns
A three-element tuple containing the counts for *available*, *lost*, and *corrupt* data samples, in that order.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while retrieving the record status.

statusData(*channel_index: int*, *count: int*) → ndarray

Retrieve the acquired data samples from the specified *AnalogIn* instrument channel.

This method returns samples as voltages, calculated from the raw, binary sample values as follows:

```
voltages = analogIn.channelOffsetGet(channel_index) + \
           analogIn.channelRangeGet(channel_index) * (raw_samples / 65536.0)
```

Note that the applied calibration is channel-dependent.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **count** (*int*) – The number of samples to retrieve.

Returns
A 1D numpy array of floats, in Volts.

Return type
nd.array

Raises
DwfLibraryError – An error occurred while retrieving the sample data.

statusData2(*channel_index: int*, *offset: int*, *count: int*) → ndarray

Retrieve the acquired data samples from the specified *AnalogIn* instrument channel.

This method returns samples as voltages, calculated from the raw, binary sample values as follows:

```
voltages = analogIn.channelOffsetGet(channel_index) + \
           analogIn.channelRangeGet(channel_index) * (raw_samples / 65536.0)
```

Note: The applied calibration is channel-dependent.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **offset** (*int*) – Sample offset.
- **count** (*int*) – Sample count.

Returns
A 1D numpy array of floats, in Volts.

Return type
nd.array

Raises
DwfLibraryError – An error occurred while retrieving the sample data.

statusData16(*channel_index: int*, *offset: int*, *count: int*) → ndarray

Retrieve the acquired data samples from the specified *AnalogIn* instrument channel.

This method returns raw, signed 16-bit samples.

In case the ADC has less than 16 bits of raw resolution, least significant zero-bits are added to stretch the range to 16 bits.

To convert these raw samples to voltages, use the following:

```
voltages = analogIn.channelOffsetGet(channel_index) + \
           analogIn.channelRangeGet(channel_index) * (raw_samples / 65536.0)
```

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.

- **offset** (*int*) – The sample offset to start copying from.
- **count** (*int*) – The number of samples to retrieve.

Returns
A 1D numpy array of 16-bit signed integers.

Return type
nd.array

Raises
DwfLibraryError – An error occurred while retrieving the sample data.

statusNoise(*channel_index: int*, *count: int*) → Tuple[ndarray, ndarray]

Retrieve the acquired noise samples from the specified *AnalogIn* instrument channel.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **count** (*int*) – Sample count.

Returns

A two-element tuple; each element is a 1D numpy array of floats, in Volts.

The first array contains the minimum values, the second array contains the maximum values.

Raises
DwfLibraryError – An error occurred while retrieving the noise data.

statusNoise2(*channel_index: int*, *offset: int*, *count: int*) → Tuple[ndarray, ndarray]

Retrieve the acquired data samples from the specified *AnalogIn* instrument channel.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **offset** (*int*) – Sample offset.
- **count** (*int*) – Sample count.

Returns

A two-element tuple; each element is a 1D numpy array of floats, in Volts.

The first array contains the minimum values, the second array contains the maximum values.

Raises
DwfLibraryError – An error occurred while retrieving the noise data.

bitsInfo() → int

Get the fixed the number of bits used by the *AnalogIn* ADC.

The number of bits can only be queried; it cannot be changed.

Note: The Analog Discovery 2 uses an Analog Devices AD9648 two-channel ADC. It converts 14-bit samples at a rate of up to 125 MHz. So for the Analog Discovery 2, this method always returns 14.

Returns
The number of bits per sample for each of the *AnalogIn* channels.

Return type
int

Raises
DwfLibraryError – An error occurred while getting the number of bits.

recordLengthSet(*record_duration: float*) → None

Set the *AnalogIn* record length, in seconds.

Note: This value is only used when the acquisition mode is configured as *Record*.

Parameters
record_duration (*float*) – The record duration to be configured, in seconds.

A record duration of 0.0 (zero) seconds indicates a request for an arbitrary-length record acquisition.

Raises
DwfLibraryError – An error occurred while setting the record duration.

recordLengthGet() → float

Get the *AnalogIn* record length, in seconds.

Note: This value is only used when the acquisition mode is configured as *Record*.

Returns

The currently configured record length, in seconds.

A record length of 0.0 (zero) seconds indicates a request for an arbitrary-length record acquisition.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the record length.

frequencyInfo() → Tuple[float, float]

Retrieve the minimum and maximum configurable ADC sample frequency of the *AnalogIn* instrument, in samples/second.

Returns
The valid sample frequency range (min, max), in samples/second.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while getting the allowed sample frequency range.

frequencySet(*sample_frequency: float*) → None

Set the ADC sample frequency of the *AnalogIn* instrument, in samples/second.

Parameters
sample_frequency (*float*) – Sample frequency, in samples/second.

Raises
DwfLibraryError – An error occurred while setting sample frequency.

frequencyGet() → float

Get the ADC sample frequency of the *AnalogIn* instrument, in samples/second.

The ADC always runs at maximum frequency, but the method in which the samples are stored and transferred can be configured individually for each channel with the *channelFilterSet* method.

Returns
The configured sample frequency, in samples/second.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the sample frequency.

bufferSizeInfo() → Tuple[int, int]

Returns the minimum and maximum allowable buffer size for the *AnalogIn* instrument, in samples.

When using the *Record* acquisition mode, the buffer size should be left at the default value, which is equal to the maximum value. In other modes (e.g. *Single*), the buffer size determines the size of the acquisition window.

Note: The maximum buffer size depends on the device configuration that was selected while opening the device.

For example, on the Analog Discovery 2, the maximum *AnalogIn* buffer size can be 512, 2048, 8192, or 16384, depending on the device configuration.

Returns
A two-element tuple. The first element is the minimum buffer size, the second element is the maximum buffer size.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while getting the buffer size info.

bufferSizeSet(*buffer_size: int*) → None

Adjust the *AnalogIn* instrument buffer size, expressed in samples.

The actual buffer size configured will be clipped by the *bufferSizeInfo()* values.

The actual value configured can be read back by calling *bufferSizeGet()*.

Parameters
buffer_size (*int*) – The requested buffer size, in samples.

Raises
DwfLibraryError – An error occurred while setting the buffer size.

bufferSizeGet() → int

Return the used *AnalogIn* instrument buffer size, in samples.

Returns
The currently configured buffer size, in samples.

Return type
int

Raises
DwfLibraryError – An error occurred while getting the buffer size.

noiseSizeInfo() → int

Return the maximum noise buffer size for the *AnalogIn* instrument, in samples.

Returns
The maximum noise buffer size, in samples.

Raises
DwfLibraryError – An error occurred while getting the noise buffer size info.

noiseSizeSet(*noise_buffer_size: int*) → None

Enable or disable the noise buffer for the *AnalogIn* instrument.

This method determines if the noise buffer is enabled or disabled.

Note: The name of this method and the type of its parameter (int) suggest that it can be used to specify the size of the noise buffer, but that is not the case.

Any non-zero value enables the noise buffer; a zero value disables it.

If enabled, the noise buffer size reported by *noiseSizeGet()* is always equal to the size of the sample buffer reported by *bufferSizeGet()*, divided by 8.

Parameters
noise_buffer_size (*int*) – Whether to enable (non-zero) or disable (zero) the noise buffer.

Raises
DwfLibraryError – An error occurred while setting the noise buffer enabled/disabled state.

noiseSizeGet() → int

Return the currently configured noise buffer size for the *AnalogIn* instrument, in samples.

This value is automatically adjusted according to the sample buffer size, divided by 8. For instance, setting the sample buffer size of 8192 implies a noise buffer size of 1024; setting the sample buffer size to 4096 implies noise buffer size will be 512.

Returns
The currently configured noise buffer size. Zero indicates that the noise buffer is disabled.

Return type
int

Raises
DwfLibraryError – An error occurred while getting the noise buffer size.

acquisitionModeInfo() → List[*DwfAcquisitionMode*]

Get a list of valid *AnalogIn* instrument acquisition modes.

Returns
A list of valid acquisition modes for the *AnalogIn* instrument.

Return type
List[DwfAcquisitionMode]

Raises
DwfLibraryError – An error occurred while getting the acquisition mode info.

acquisitionModeSet(*acquisition_mode:* DwfAcquisitionMode) → None

Set the *AnalogIn* acquisition mode.

Parameters
acquisition_mode (DwfAcquisitionMode) – The acquisition mode to be configured.

Raises
DwfLibraryError – An error occurred while setting the acquisition mode.

acquisitionModeGet() → *DwfAcquisitionMode*

Get the currently configured *AnalogIn* acquisition mode.

Returns
The acquisition mode currently configured.

Return type
DwfAcquisitionMode

Raises
DwfLibraryError – An error occurred while getting the acquisition mode.

channelCount() → int

Read the number of *AnalogIn* input channels.

Returns
The number of analog input channels.

Return type
int

Raises
DwfLibraryError – An error occurred while retrieving the number of analog input channels.

channelCounts() → Tuple[int, int, int]

Read the number of *AnalogIn* input channels, distinguishing between real and filter channels.

Returns
The number of real, fiiltered, and total analog input channels.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while retrieving the number of analog input channels.

channelEnableSet(*channel_index: int*, *channel_enable: bool*) → None

Enable or disable the specified *AnalogIn* input channel.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_enable** (*bool*) – Whether to enable (True) or disable (False) the specified channel.

Raises
DwfLibraryError – An error occurred while enabling or disabling the channel.

channelEnableGet(*channel_index: int*) → bool

Get the current enable/disable status of the specified *AnalogIn* input channel.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
Channel is enabled (True) or disabled (False).

Return type
bool

Raises
DwfLibraryError – An error occurred while getting the enabled/disabled state of the channel.

channelFilterInfo() → List[*DwfAnalogInFilter*]

Get a list of valid *AnalogIn* channel filter settings.

Returns
A list of valid channel filter settings.

Return type
List[DwfAnalogInFilter]

Raises
DwfLibraryError – An error occurred while getting the channel filter info.

channelFilterSet(*channel_index: int*, *channel_filter:* DwfAnalogInFilter) → None

Set the filter for a specified *AnalogIn* input channel.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_filter** (DwfAnalogInFilter) – The channel filter mode to be selected.

Raises
DwfLibraryError – An error occurred while setting the channel filter.

channelFilterGet(*channel_index: int*) → *DwfAnalogInFilter*

Get the *AnalogIn* input channel filter setting.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
The currently selected channel filter mode.

Return type
DwfAnalogInFilter

Raises
DwfLibraryError – An error occurred while getting the current channel filter setting.

channelRangeInfo() → Tuple[float, float, int]

Report the possible voltage range of the *AnalogIn* input channels, in Volts.

The values returned represent ideal values. The actual calibrated ranges are channel-dependent.

See also:

The *channelRangeSteps()* method returns essentially the same information in a different representation.

Returns
The minimum range (Volts), maximum range (Volts), and number of different discrete channel range settings of the *AnalogIn* instrument.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the analog input range setting info.

channelRangeSet(*channel_index: int*, *channel_range: float*) → None

Set the range setting of the specified *AnalogIn* input channel, in Volts.

Note: The actual range set will generally be different from the requested range.

Note: Changing the channel range may also change the channel offset.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_range** (*float*) – The requested channel range, in Volts.

Raises

DwfLibraryError – An error occurred while setting the channel voltage range.

channelRangeGet(*channel_index: int*) → float

Get the range setting of the specified *AnalogIn* input channel, in Volts.

Together with the channel offset, this value can be used to transform raw binary ADC values into Volts.

Parameters

channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns

The actual channel range, in Volts.

Return type

float

Raises

DwfLibraryError – An error occurred while setting the channel voltage range.

channelRangeSteps() → List[float]

Report the possible voltage ranges of the *AnalogIn* input channels, in Volts, as a list.

The values returned represent ideal values. The actual calibrated ranges are channel-dependent.

See also:

The *channelRangeInfo()* method returns essentially the same information in a different representation.

Returns

A list of ranges, in Volts, representing the discrete channel range settings of the *AnalogIn* instrument.

Return type

List[*float*]

Raises

DwfLibraryError – An error occurred while getting the list of analog input range settings.

channelOffsetInfo() → Tuple[float, float, int]

Get the possible *AnalogIn* input channel offset settings, in Volts.

Returns

The minimum channel offset (Volts), maximum channel offset (Volts), and number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the channel offset info.

channelOffsetSet(*channel_index: int*, *channel_offset: float*) → None

Set the *AnalogIn* input channel offset, in Volts.

Note: The actual offset will generally be different from the requested offset.

Note: Changing the channel offset may also change the channel range.

Parameters
- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_offset** (*float*) – The channel offset, in Volts.

Raises
DwfLibraryError – An error occurred while setting the channel offset.

channelOffsetGet(*channel_index: int*) → float

Get the *AnalogIn* input channel offset, in Volts.

Together with the channel range, this value can be used to transform raw binary ADC values into Volts.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
The currently configured channel offset, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the current channel offset setting.

channelAttenuationSet(*channel_index: int*, *channel_attenuation: float*) → None

Set the *AnalogIn* input channel attenuation setting.

The channel attenuation is a dimensionless factor.

This setting is used to compensate for probe attenuation. Many probes have two attenuation settings (e.g., ×1 and ×10). The value of this setting should correspond to the value of the probe, or 1 (the default) if a direct connection without attenuation is used.

Note: Changing the channel attenuation will also change the channel offset and range.

Parameters
- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_attenuation** (*float*) – The requested channel attenuation setting. If it is 0.0, the attenuation is set to 1.0 (the default) instead.

Raises
DwfLibraryError – An error occurred while setting the current channel attenuation.

channelAttenuationGet(*channel_index: int*) → float

Get the *AnalogIn* input channel attenuation setting.

The channel attenuation is a dimensionless factor.

This setting is used to compensate for probe attenuation. Many probes have two attenuation settings (e.g., ×1 and ×10). The value of this setting should correspond to the value of the probe, or 1 (the default) if a direct connection without attenuation is used.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
The channel attenuation setting.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the current channel attenuation.

channelBandwidthSet(*channel_index: int*, *channel_bandwidth: float*) → None

Set the *AnalogIn* input channel bandwidth setting.

Note: On the Analog Discovery 2, the channel bandwidth setting exists and can be set and retrieved, but the value has no effect.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_bandwidth** (*float*) – The channel bandwidth setting, in Hz.

Raises
DwfLibraryError – An error occurred while setting the channel bandwidth.

channelBandwidthGet(*channel_index: int*) → float

Get the *AnalogIn* input channel bandwidth setting.

Note: On the Analog Discovery 2, the channel bandwidth setting exists and can be set and retrieved, but the value has no effect.

Parameters
channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns
The channel bandwidth setting, in Hz.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the current channel bandwidth.

channelImpedanceSet(*channel_index: int*, *channel_impedance: float*) → None

Set the *AnalogIn* input channel impedance setting, in Ohms.

Note: On the Analog Discovery 2, the channel impedance setting exists and can be set and retrieved, but the value has no effect.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_impedance** (*float*) – channel impedance setting, in Ohms.

Raises

DwfLibraryError – An error occurred while setting the current channel impedance.

channelImpedanceGet(*channel_index: int*) → float

Get the *AnalogIn* input channel impedance setting, in Ohms.

Note: On the Analog Discovery 2, the channel impedance setting exists and can be set and retrieved, but the value has no effect.

Parameters

channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns

The channel impedance setting, in Ohms.

Return type

float

Raises

DwfLibraryError – An error occurred while getting the current channel impedance.

channelCouplingInfo() → List[*DwfAnalogCoupling*]

Get the *AnalogIn* channel coupling info.

Raises

DwfLibraryError – An error occurred while getting the channel coupling info.

channelCouplingSet(*channel_index: int*, *channel_coupling:* DwfAnalogCoupling) → None

Set the *AnalogIn* input channel coupling.

Parameters

- **channel_index** (*int*) – The channel index, in the range 0 to *channelCount()*-1.
- **channel_coupling** (*AnalogCoupling*) – channel coupling to be set.

Raises

DwfLibraryError – An error occurred while setting the current channel coupling.

channelCouplingGet(*channel_index: int*) → *DwfAnalogCoupling*

Get the *AnalogIn* input channel impedance setting, in Ohms.

Parameters

channel_index (*int*) – The channel index, in the range 0 to *channelCount()*-1.

Returns

The channel coupling (DC or AC).

Return type

DwfAnalogCoupling

Raises

DwfLibraryError – An error occurred while getting the current channel impedance.

triggerSourceInfo() → List[*DwfTriggerSource*]

Get the *AnalogIn* instrument trigger source info.

Warning: This method is obsolete.

Use the generic *DwfDevice.triggerInfo()* method instead.

Returns
A list of trigger sources that can be selected.

Return type
List[DwfTriggerSource]

Raises
DwfLibraryError – An error occurred while retrieving the trigger source information.

triggerSourceSet(*trigger_source:* DwfTriggerSource) → None

Set the *AnalogIn* instrument trigger source.

Parameters
trigger_source (DwfTriggerSource) – The trigger source to be selected.

Raises
DwfLibraryError – An error occurred while setting the trigger source.

triggerSourceGet() → *DwfTriggerSource*

Get the currently selected instrument trigger source.

Returns
The currently selected trigger source.

Return type
DwfTriggerSource

Raises
DwfLibraryError – An error occurred while retrieving the selected trigger source.

triggerPositionInfo() → Tuple[float, float, int]

Get the *AnalogIn* instrument trigger position range.

Returns
The valid range of trigger positions that can be configured. The values returned are the *minimum* and *maximum* valid position settings, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while retrieving the trigger position info.

triggerPositionSet(*trigger_position: float*) → None

Set the *AnalogIn* instrument trigger position, in seconds.

The meaning of the trigger position depends on the currently selected acquisition mode:

- In *Record* acquisition mode, the trigger position is the time of the first valid sample acquired relative to the position of the trigger event. Negative values indicates times before the trigger time.

  To place the trigger in the middle of the recording, this value should be set to -0.5 times the duration of the recording.

- In *Single* acquisition mode, the trigger position is the trigger event time relative to the center of the acquisition window.

  To place the trigger in the middle of the acquisition buffer, the value should be 0.

Parameters
trigger_position (*float*) – The trigger position to be configured, in seconds.

Raises
DwfLibraryError – An error occurred while setting the trigger position.

triggerPositionGet() → float

Get the *AnalogIn* instrument trigger position, in seconds.

Returns
The currently configured trigger position, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger position.

triggerPositionStatus() → float

Get the current *AnalogIn* instrument trigger position status.

Returns
The current trigger position, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger position status.

triggerAutoTimeoutInfo() → Tuple[float, float, int]

Get the *AnalogIn* instrument trigger auto-timeout range, in seconds.

Returns
The valid range of trigger auto-timeout values that can be configured. The values returned are the *minimum* and *maximum* valid auto-timeout settings, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger auto-timeout info.

triggerAutoTimeoutSet(*trigger_auto_timeout: float*) → None

Set the *AnalogIn* instrument trigger auto-timeout value, in seconds.

When set to 0, the trigger auto-timeout feature is disabled, corresponding to *Normal* acquisition mode on desktop oscilloscopes.

Parameters
trigger_auto_timeout (*float*) – The auto timeout setting, in seconds.

Raises
DwfLibraryError – An error occurred while setting the trigger auto-timeout value.

triggerAutoTimeoutGet() → float

Get the *AnalogIn* instrument trigger auto-timeout value, in seconds.

Returns
The currently configured auto-timeout value, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger auto-timeout value.

triggerForce() → None

Force assertion of the *AnalogIn* instrument trigger.

Important: This method forces the *AnalogIn* device to act as if it was triggered, independent of the currently active trigger source. It does not generate an artificial trigger event on the *AnalogIn* trigger detector.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerHoldOffInfo() → Tuple[float, float, int]

Get the *AnalogIn* trigger detector holdoff range, in seconds.

The trigger holdoff setting is the minimum time (in seconds) that should pass for a trigger to be recognized by the trigger detector after a previous trigger event.

Returns
The valid range of trigger detector holdoff values that can be configured. The values returned are the *minimum* and *maximum* valid holdoff settings, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger holdoff info.

triggerHoldOffSet(*trigger_detector_holdoff: float*) → None

Set the *AnalogIn* trigger detector holdoff value, in seconds.

Parameters
trigger_detector_holdoff (*float*) – The trigger holdoff setting, in seconds.

The value 0 disables the trigger detector holdoff feature.

Raises
DwfLibraryError – An error occurred while setting the trigger detector holdoff value.

triggerHoldOffGet() → float

Get the current *AnalogIn* trigger holdoff value, in seconds.

Returns
The currently configured trigger holdoff value, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger holdoff value.

triggerTypeInfo() → List[*DwfAnalogInTriggerType*]

Get the valid *AnalogIn* trigger detector trigger-type values.

This setting determines the type of event recognized by the *AnalogIn* trigger detector as a trigger. Possible types includes *Edge*, *Pulse*, *Transition*, and *Window*.

Returns
A list of trigger detector trigger-types that can be configured.

Return type
List[DwfAnalogInTriggerType]

Raises
DwfLibraryError – An error occurred while getting the trigger detector trigger-type info.

triggerTypeSet(*trigger_detector_type:* DwfAnalogInTriggerType) → None

Set the *AnalogIn* trigger detector trigger-type.

Parameters
trigger_detector_type (DwfAnalogInTriggerType) – The trigger detector trigger-type to be configured.

Raises
DwfLibraryError – An error occurred while setting the trigger detector trigger-type.

triggerTypeGet() → *DwfAnalogInTriggerType*

Get the currently configured *AnalogIn* trigger detector trigger-type.

Returns
The currently configured trigger detector trigger-type.

Return type
DwfAnalogInTriggerType

Raises
DwfLibraryError – An error occurred while getting the trigger detector trigger-type.

triggerChannelInfo() → Tuple[int, int]

Get the *AnalogIn* trigger detector channel range.

The *AnalogIn* trigger detector monitors a specific analog in channel for trigger events. This method returns the range of valid analog input channels that can be configured as the *AnalogIn* trigger detector channel.

Returns
The first and last channel that can be used as trigger detector channels.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger detector channel range.

triggerChannelSet(*trigger_detector_channel_index: int*) → None

Set the *AnalogIn* trigger detector channel.

This is the analog input channel that the *AnalogIn* trigger detector monitors for trigger events.

Parameters
trigger_detector_channel_index (*int*) – The trigger detector channel to be selected.

Raises
DwfLibraryError – An error occurred while setting the trigger detector channel.

triggerChannelGet() → int

Get the *AnalogIn* trigger detector channel.

This is the analog input channel that the *AnalogIn* trigger detector monitors for trigger events.

Returns
The currently configured trigger detector channel.

Return type
int

Raises
DwfLibraryError – An error occurred while getting the trigger detector channel.

triggerFilterInfo() → List[*DwfAnalogInFilter*]

Get a list of valid *AnalogIn* trigger detector filter values.

Returns
A list of filters that can be configured for the trigger detector channel.

Return type
List[DwfAnalogInFilter]

Raises
DwfLibraryError – An error occurred while getting the valid trigger detector channel filter values.

triggerFilterSet(*trigger_detector_filter:* DwfAnalogInFilter) → None

Set the *AnalogIn* trigger detector channel filter.

Parameters
trigger_detector_filter (DwfAnalogInFilter) – The trigger detector channel filter to be selected.

Raises
DwfLibraryError – An error occurred while setting the trigger detector channel filter.

triggerFilterGet() → *DwfAnalogInFilter*

Get the *AnalogIn* trigger detector channel filter.

Returns
The currently configured trigger detector channel filter.

Return type
DwfAnalogInFilter

Raises
DwfLibraryError – An error occurred while getting the trigger detector channel filter.

triggerLevelInfo() → Tuple[float, float, int]

Get the *AnalogIn* trigger detector valid trigger level range, in Volts.

Returns
The range of valid trigger levels that can be configured. The values returned are the *minimum* and *maximum* trigger levels in Volts, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger level range.

triggerLevelSet(*trigger_detector_level: float*) → None

Set the *AnalogIn* trigger detector trigger-level, in Volts.

Parameters
trigger_detector_level (*float*) – The trigger level to be configured, in Volts.

Raises
DwfLibraryError – An error occurred while setting the trigger level.

triggerLevelGet() → float

Get the *AnalogIn* trigger detector trigger-level, in Volts.

Returns
The currently configured trigger level, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger level.

triggerHysteresisInfo() → Tuple[float, float, int]

Get the *AnalogIn* trigger detector valid hysteresis range, in Volts.

Returns
The valid range of trigger hysteresis values that can be configured. The values returned are the *minimum* and *maximum* trigger hysteresis levels in Volts, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger hysteresis info.

triggerHysteresisSet(*trigger_detector_hysteresis: float*) → None

Set the *AnalogIn* trigger detector hysteresis, in Volts.

Parameters
trigger_detector_hysteresis (*float*) – The trigger hysteresis to be configured, in Volts.

Raises
DwfLibraryError – An error occurred while setting the trigger hysteresis.

triggerHysteresisGet() → float

Get the *AnalogIn* trigger detector trigger hysteresis, in Volts.

Returns
The currently configured trigger hysteresis, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger hysteresis.

triggerConditionInfo() → List[*DwfTriggerSlope*]

Get the valid *AnalogIn* trigger detector condition (slope) options.

Returns
A list of valid trigger detector condition (slope) values.

Return type
List[DwfTriggerSlope]

Raises
DwfLibraryError – An error occurred while getting the valid trigger detector condition values.

triggerConditionSet(*trigger_detector_condition:* DwfTriggerSlope) → None

Set the *AnalogIn* trigger detector condition (slope).

Parameters
trigger_detector_condition (DwfTriggerSlope) – The trigger detector condition (slope) to be configured.

Raises
DwfLibraryError – An error occurred while setting the trigger condition.

triggerConditionGet() → *DwfTriggerSlope*

Get the *AnalogIn* trigger detector condition (slope).

Returns
The currently configured trigger condition (slope).

Return type
DwfTriggerSlope

Raises
DwfLibraryError – An error occurred while setting the trigger condition (slope).

triggerLengthInfo() → Tuple[float, float, int]

Get the valid *AnalogIn* trigger detector length range, in seconds.

Returns
The valid range of trigger detector length values that can be configured. The values returned are the *minimum* and *maximum* trigger detector length values in seconds, and the number of steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while getting the trigger length range.

triggerLengthSet(*trigger_detector_length: float*) → None

Set the *AnalogIn* trigger detector length, in seconds.

Parameters
trigger_detector_length (*float*) – The trigger detector trigger length to be configured, in seconds.

Raises
DwfLibraryError – An error occurred while setting the trigger detector length.

triggerLengthGet() → float

Get the *AnalogIn* trigger detector length, in seconds.

Returns
The currently configured trigger detector trigger length, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the trigger detector length.

triggerLengthConditionInfo() → List[*DwfAnalogInTriggerLengthCondition*]

Get a list of valid *AnalogIn* trigger detector length condition values.

Trigger length condition values include *Less*, *Timeout*, and *More*.

Returns
A list of valid trigger detector length condition values.

Return type
List[DwfAnalogInTriggerLengthCondition]

Raises
DwfLibraryError – An error occurred while getting the trigger length condition info.

triggerLengthConditionSet(*trigger_detector_length_condition:* DwfAnalogInTriggerLengthCondition) → None

Set the *AnalogIn* trigger detector length condition.

Parameters
trigger_detector_length_condition (DwfAnalogInTriggerLengthCondition) – The trigger detector length condition to be configured.

Raises
DwfLibraryError – An error occurred while setting the trigger detector length condition.

triggerLengthConditionGet() → *DwfAnalogInTriggerLengthCondition*

Get the *AnalogIn* trigger detector length condition.

Returns
The currently configured trigger detector length condition.

Return type
DwfAnalogInTriggerLengthCondition

Raises
DwfLibraryError – An error occurred while getting the trigger detector length condition.

counterInfo() → Tuple[int, float]

Get *AnalogIn* counter info.

counterSet(*duration: float*) → None

Set *AnalogIn* counter duration.

counterGet() → float

Get *AnalogIn* counter duration.

counterStatus() → Tuple[float, float, int]

Get *AnalogIn* counter status.

samplingSourceSet(*sampling_source:* DwfTriggerSource) → None

Set the *AnalogIn* sampling source.

Parameters
sampling_source (DwfTriggerSource) – The sampling source to be configured.

Raises
DwfLibraryError – An error occurred while setting the sampling source.

samplingSourceGet() → *DwfTriggerSource*

Get the *AnalogIn* sampling source.

Returns
The currently configured sampling source.

Return type
DwfTriggerSource

Raises
DwfLibraryError – An error occurred while getting the sampling source.

samplingSlopeSet(*sampling_slope:* DwfTriggerSlope) → None

Set the *AnalogIn* sampling slope.

Parameters
sampling_slope (DwfTriggerSlope) – The sampling slope to be configured.

Raises
DwfLibraryError – An error occurred while setting the sampling slope.

samplingSlopeGet() → *DwfTriggerSlope*

Get the *AnalogIn* sampling slope.

Returns
The currently configured sampling slope.

Return type
DwfTriggerSlope

Raises
DwfLibraryError – An error occurred while getting the sampling slope.

samplingDelaySet(*sampling_delay: float*) → None

Set the *AnalogIn* sampling delay, in seconds.

Parameters
sampling_delay (*float*) – The sampling delay to be configured, in seconds.

Raises
DwfLibraryError – An error occurred while setting the sampling delay.

samplingDelayGet() → float

Get the *AnalogIn* sampling delay, in seconds.

Returns
The currently configured sampling delay, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while getting the sampling delay.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.3 Analog output instrument

The *AnalogOut* instrument provides multiple channels of analog output on devices that support it, such as the Analog Discovery and the Analog Discovery 2. It provides the functionality normally associated with a stand-alone arbitrary waveform generator.

Todo: This section is currently incomplete.

It lacks a detailed discussion of how all the settings work.

Important: The *AnalogOut* channels are designed to be capable of operating independently.

To that end, each *AnalogOut* channel has its own settings and state, and its behavior is fully independent from the behavior of the other analog output channels, unless explicitly commanded using the *AnalogOut.masterSet()* method.

Because of this, each *AnalogOut* channel can be considered a fully independent instrument.

4.3.1 Using the analog output instrument

To use the *AnalogOut* instrument you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *AnalogOut* instrument can now be accessed via its `analogOut` attribute, which is an instance of the `AnalogOut` class.

For example:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's AnalogOut instrument.
    analogOut = device.analogOut

    # Use the AnalogOut instrument: reset all output channels.
    analogOut.reset(-1)
```

4.3.2 The *AnalogOut* channel state machine

Each *AnalogOut* channel is controlled by a state machine. As an output sequence is prepared and executed, the channel goes through its various states.

The current state of the channel is returned by the `analogOut.status()` method, and is of type `DwfState`.

The figure below shows the states used by the *AnalogOut* instrument and the transitions between them:

Fig. 2: States of the *AnalogOut* instrument

The *AnalogOut* states are used as follows:

1. `Ready`

   In this preparatory state, channel settings can be changed that specify the behavior of the channel in the coming output sequence. If the auto-configure setting of the device is enabled (the default), setting changes will automatically be transferred to the device. If not, an explicit call to the `analogOut.configure()` method is needed to transfer updated settings to the device.

   Once the channel is properly configured, an output sequence can be started by calling the `analogOut.configure()` with the *start* parameter set to True. This will start the first stage of the output sequence by entering the `Armed` state.

2. `Armed`

   In this state the channel continuously monitors the configured trigger input. As soon as a trigger event is detected, the instrument proceeds to the `Wait` state.

3. `Wait`

   In this state, the analog output is driven according to the channel's *Idle* setting. The duration of the wait state is configurable. Once this duration has passed, the channel proceeds to the `Running` state.

4. `Running`

   In this state the channel drives its output according to its node settings. This continues until the run duration has been reached. The channel then proceeds to the **Repeat** state.

5. **Repeat**

Note: This is not a true state, in that there is no `DwfState` value that represents it. It is included here to explain the control flow of the *AnalogOut* channel state machine.

When an output run is finished, the *repeat count* is decremented.

If, after decrementing, the *repeat count* is unequal to zero, more output must be produced. If the *repeat trigger* setting is True, the channel proceeds to the `Armed` state; in that case, a trigger is needed to start each of the output runs. If the *repeat trigger* setting is False, the channel proceeds immediately to the `Wait` state to start another output sequence; a trigger is only required before the very first output run.

If, after decrementing, the *repeat count* did reach zero, the channel becomes idle and proceeds to the `Done` state.

6. `Done`

   This state indicates that an output sequence has finished. In this state, the analog output is driven according to the channel's *Idle* setting.

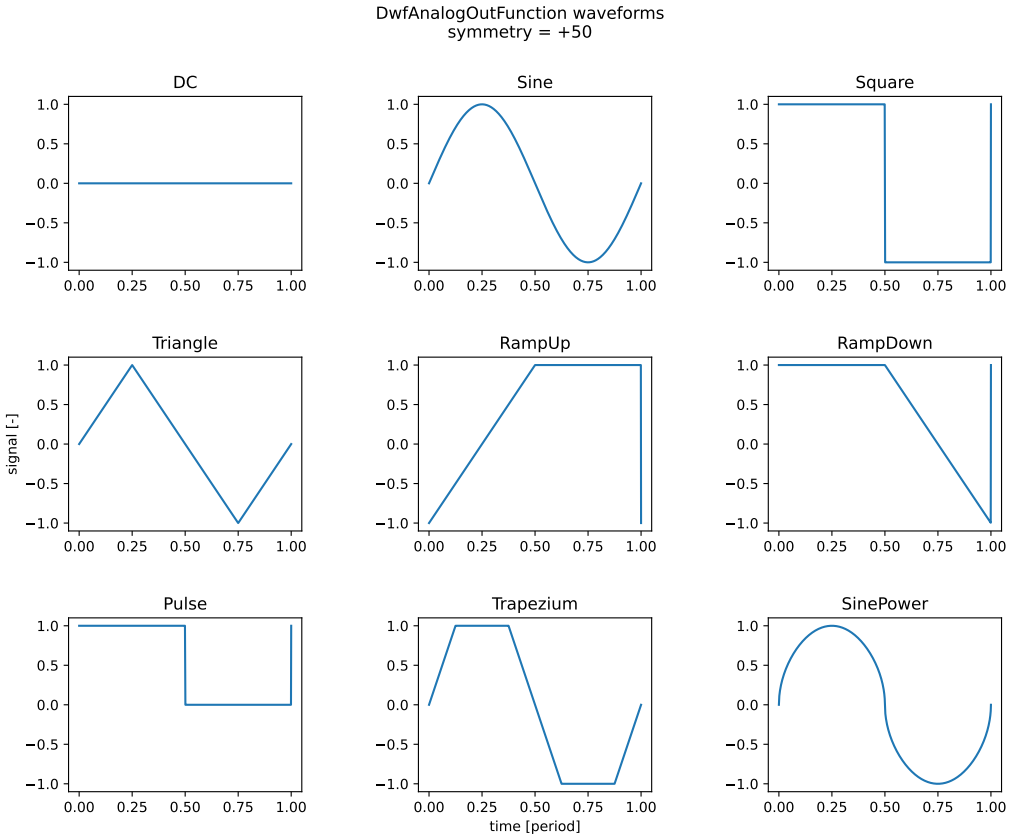
   From this state, it is possible to go back to the `Ready` state by performing any kind of configuration, or to start a new output sequence.

4.3.3 *AnalogOut* channel nodes

AnalogOut channels are organized in *nodes*, which can be independently configured. A node represent either the primary non-modulated signal (the *Carrier*), or some form of modulation, like *Amplitude Modulation* (AM) or *Frequency Modulation* (FM). The output of each node varies over time according to its settings. The node outputs are combined to synthesize the signal that is driven onto the analog output channel via a DAC.

Note: Early versions of the library only implemented the *Carrier* signal and lacked modulation support. With the introduction of AM and FM modulation, the *node* concept was introduced. Because of this, there are 24 methods that configure the carrier signal directly, but also 24 methods that configure a selectable node. In new user programs, only the latter should be used.

The nodes of an analog output channel can be configured independently. The contribution of nodes can be individually enabled or disabled, which is most useful for the AM and FM nodes. For nodes that are enabled, a number of standard waveform shape functions are available, such as sine, block, triangle, and ramp. These can be modified by controlling their offset, amplitude, frequency, phase, and symmetry; the latter alters the waveform from its regular, symmetrical shape.



It is also possible to upload an arbitrary wave-shape to the instrument as a sequence of samples to be played. This can be used for short wave-shapes, but it is also possible to perform continuous playback by uploading blocks of samples in a loop.

4.3.4 *AnalogOut* instrument API overview

With 83 methods, the *AnalogOut* instrument is the second most complicated instrument supported by the Digilent Waveforms API, after the *AnalogIn* instrument. Below, we categorize all its methods and shortly introduce them. Detailed information on all methods can be found in the `AnalogOut` class reference that follows.

Instrument control

Like all instruments supported by the Digilent Waveforms library, the *AnalogOut* instrument provides `reset()`, `configure()`, and `status()` methods.

The *AnalogOut* instrument is unusual in that these methods operate on individual *AnalogOut* channels, meaning that each *AnalogOut* channel acts as as a separate, independent instrument.

The `reset()` method resets a specific analog output channel (or all channels).

The `configure()` method is used to explicitly transfer settings to the instrument, and/or to start a configured operation.

The `status()` method retrieves status information from the instrument. It returns the current `DwfState` of the *AnalogOut* instrument.

Table 12: Instrument control (3 methods)

control operation	type/unit	methods
reset instrument	*n/a*	*reset()*
configure instrument	*n/a*	*configure()*
request instrument status	*DwfState*	*status()*

Channel count

This method returns the number of analog output channels.

Table 13: Channel count (1 method)

property	type/unit	method
channel count	int	*count()*

Per-channel state machine settings

These settings determine the duration of the *Wait* and *Running* states, how many times the Wait/Running cycle should be repeated, and whether a trigger must precede each Wait/Running cycle.

The *channel master* setting allows an analog output channel to be controlled by another channel, synchronizing their behavior.

Table 14: State machine settings (15 methods)

setting	type/unit	methods
wait duration	float [s]	*waitInfo()* , *-Set()* , *-Get()*
run duration	float [s]	*runInfo()* , *-Set()* , *-Get()* , *-Status()*
repeat count	int [-]	*repeatInfo()* , *-Set()* , *-Get()* , *-Status()*
repeat trigger	bool	*repeatTriggerSet()* , *-Get()*
channel master	int	*masterSet()* , *-Get()*

Per-channel trigger configuration

These settings configure the channel trigger.

Table 15: Per-channel trigger configuration (5 methods)

setting	type/unit	methods
trigger source	*DwfTriggerSource*	*triggerSourceInfo()* , *-Set()* , *-Get()*
trigger slope	*DwfTriggerSlope*	*triggerSlopeSet()* , *-Get()*

Note: The *triggerSourceInfo()* method is obsolete. Use the generic *DwfDevice.triggerInfo()* method instead.

Per-channel output settings

These settings determine the channel output behavior.

Table 16: Per-channel output settings (8 methods)

setting	type/unit	methods
channel mode	*DwfAnalogOutMode*	*modeSet()* , *-Get()*
channel idle	*DwfAnalogOutIdle*	*idleInfo()* , *-Set()* , *-Get()*
channel limitation	float [V] or [A]	*limitationInfo()* , *-Set()* , *-Get()*

Per-channel miscellaneous settings

The function of the *custom AM/FM enable* setting is currently not understood. It is only applicable to Electronics Explorer devices, as stated in a message on the Digilent forum.

Todo: Figure out what the *custom AM/FM enable* setting does.

Table 17: Per-channel configuration (2 methods)

setting	type/unit	methods
custom AM/FM enable	bool	*customAMFMEnableSet()* , *-Get()*

Node enumeration

This method enumerates all nodes of an *AnalogOut* channel.

Table 18: Node enumeration (1 method)

property	type/unit	methods
node enumeration	*DwfAnalogOutNode* list	*nodeInfo()*

Node configuration

These methods configure the output signal of an *AnalogOut* channel node.

Table 19: Node configuration (20 methods)

setting	type/unit	methods
node enable	bool	*nodeEnableSet()* , *-Get()*
node function	*DwfAnalogOutFunction*	*nodeFunctionInfo()* , *-Set()* , *-Get()*
node frequency	float [Hz]	*nodeFrequencyInfo()* , *-Set()* , *-Get()*
node amplitude	float [V]	*nodeAmplitudeInfo()* , *-Set()* , *-Get()*
node offset	float [V] | *nodeOffsetInfo()* , *-Set()* , *-Get()*	
node symmetry	float [%] | *nodeSymmetryInfo()* , *-Set()* , *-Get()*	
node phase	float [deg]	*nodePhaseInfo()* , *-Set()* , *-Get()*

Node data management

These methods transfer arbitrary waveform data to an *AnalogOut* channel node.

Table 20: Node data management (4 methods)

operation	type/unit	methods
node data upload	*n/a*	*nodeDataInfo()* , *-Set()*
node play status	*n/a*	*nodePlayStatus()*
node play data upload	*n/a*	*nodePlayData()*

Carrier configuration (obsolete)

Note: These methods have been replaced by equivalent *node* methods.

Table 21: Carrier configuration (20 methods)

setting	type/unit	methods
carrier enable	bool	*enableSet()* , *-Get()*
carrier function	*DwfTriggerSource*	*functionInfo()* , *-Set()* , *-Get()*
carrier frequency	float [Hz]	*frequencyInfo()* , *-Set()* , *-Get()*
carrier amplitude	float [V]	*amplitudeInfo()* , *-Set()* , *-Get()*
carrier offset	float [V]	*offsetInfo()* , *-Set()* , *-Get()*
carrier symmetry	float [%]	*symmetryInfo()* , *-Set()* , *-Get()*
carrier phase	float [deg]	*phaseInfo()* , *-Set()* , *-Get()*

Carrier node data management (obsolete)

Note: These methods have been replaced by equivalent *node* methods.

Table 22: Carrier data management (4 methods)

operation	type/unit	methods
carrier data upload	*n/a*	*dataInfo()* , *-Set()*
carrier play status	*n/a*	*playStatus()*
carrier play data upload	*n/a*	*playData()*

4.3.5 *AnalogOut* reference

class AnalogOut

The *AnalogOut* class provides access to the analog output (signal generator) instrument of a *DwfDevice*.

Attention: Users of *pydwf* should not create instances of this class directly.

It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *analogOut* attribute for access by the user.

reset(*channel_index: int*) → None

Reset the *AnalogOut* instrument.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

configure(*channel_index: int*, *start: int*) → None

Configure the *AnalogOut* instrument.

Parameters

- **channel_index** (*int*) – The output channel to configure. Specify -1 to configure all channels.
- **start** (*int*) – Whether to start/stop the instrument:
  - 0 — Stop instrument
  - 1 — Start instrument
  - 3 — Apply settings; do not change instrument state

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status(*channel_index: int*) → *DwfState*

Get the *AnalogOut* instrument channel state.

This method performs a status request to the *AnalogOut* instrument and receives its response.

Parameters
channel_index (*int*) – The output channel for which to get the status.

Returns
The status of the *AnalogOut* instrument channel.

Return type
DwfState

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

count() → int

Count the number of analog output channels.

Returns
The number of analog output channels.

Return type
int

Raises
DwfLibraryError – An error occurred while retrieving the number of analog output channels.

waitInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel valid *Wait* state duration range, in seconds.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of configurable *Wait* state durations, in seconds.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

waitSet(*channel_index: int*, *wait_duration: float*) → None

Set the *AnalogOut* channel *Wait* state duration, in seconds.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **wait_duration** (*float*) – The *Wait* state duration, in seconds.

Raises
DwfLibraryError – An error occurred while executing the operation.

waitGet(*channel_index: int*) → float

Get the *AnalogOut* channel *Wait* state duration, in seconds.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured *Wait* state duration.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

runInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel valid *Running* state duration range, in seconds.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of allowed *Running* state durations, in seconds.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

runSet(*channel_index: int*, *run_duration: float*) → None

Set the *AnalogOut* channel *Running* state duration, in seconds.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **run_duration** (*float*) – The *Running* state duration, in seconds. Specify 0 for a run of indefinite length.

Raises
DwfLibraryError – An error occurred while executing the operation.

runGet(*channel_index: int*) → float

Get the *AnalogOut* channel *Running* state duration, in seconds.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured *Running* state duration, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

runStatus(*channel_index: int*) → float

Get *Running* state duration time left.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The current time remaining in the *Running* state, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatTriggerSet(*channel_index: int*, *repeat_trigger_flag: bool*) → None

Set the *AnalogOut* channel *repeat trigger* setting.

This setting determines if a new trigger must precede all Wait/Running sequences, or only the first one.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **repeat_trigger_flag** (*bool*) – True if each Wait/Running sequence needs its own trigger, False if only the first Wait/Running sequence needs a trigger.

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatTriggerGet(*channel_index: int*) → bool

Get the *AnalogOut* channel *repeat trigger* setting.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured *repeat trigger* setting.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatInfo(*channel_index: int*) → Tuple[int, int]

Get *AnalogOut* repeat count range.

The *repeat count* is the number of times the *AnalogOut* channel will go through the Wait/Running or Armed/Wait/Running state cycles during the output sequence.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of configurable repeat values.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatSet(*channel_index: int*, *repeat: int*) → None

Set the *AnalogOut* repeat count.

The *repeat count* is the number of times the *AnalogOut* channel will go through the Wait/Running or Armed/Wait/Running state cycles during the output sequence.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **repeat** (*int*) – The repeat count. If 0, repeat indefinitely.

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatGet(*channel_index: int*) → int

Get the *AnalogOut* repeat count.

The *repeat count* is the number of times the *AnalogOut* channel will go through the Wait/Running or Armed/Wait/Running state cycles during the output sequence.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured repeat value. 0 means: repeat indefinitely.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatStatus(*channel_index: int*) → int

Get the *AnalogOut* current repeat count, which decreases to 0 while going through Running/Wait state cycles.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The current repeat count value.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

masterSet(*channel_index: int*, *master_channel_index: int*) → None

Set the *AnalogOut* channel master.

Sets the state machine master channel of the analog output channel.

Parameters

- **channel_index** (*int*) – The output channel for which to set the master setting. Specify -1 to set all channels.
- **master_channel_index** (*int*) – The master channel.

Raises
DwfLibraryError – An error occurred while setting the value.

masterGet(*channel_index: int*) → int

Get the *AnalogOut* channel master.

Parameters
channel_index (*int*) – The analog output channel for which to get the master channel.

Returns
The index of the master channel which the channel is configured to follow.

Return type
int

Raises
DwfLibraryError – An error occurred while getting the value.

triggerSourceInfo() → List[*DwfTriggerSource*]

Get a list of valid *AnalogOut* instrument trigger sources.

> **Warning: This method is obsolete.**
>
> Use the generic *DwfDevice.triggerInfo()* method instead.

Returns
The list of DwfTriggerSource values that can be configured.

Return type
List[DwfTriggerSource]

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSourceSet(*channel_index: int*, *trigger_source:* DwfTriggerSource) → None

Set the *AnalogOut* channel trigger source.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **trigger_source** (DwfTriggerSource) – The trigger source to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSourceGet(*channel_index: int*) → *DwfTriggerSource*

Get the currently selected channel trigger source.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently selected channel trigger source.

Return type
DwfTriggerSource

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSlopeSet(*channel_index: int*, *trigger_slope:* DwfTriggerSlope) → None

Select the *AnalogOut* channel trigger slope.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **trigger_slope** (DwfTriggerSlope) – The trigger slope to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSlopeGet(*channel_index: int*) → *DwfTriggerSlope*

Get the currently selected *AnalogOut* channel trigger slope.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently selected *AnalogOut* channel trigger slope.

Return type
DwfTriggerSlope

Raises
DwfLibraryError – An error occurred while executing the operation.

modeSet(*channel_index: int*, *mode:* DwfAnalogOutMode) → None

Set the *AnalogOut* channel mode (voltage or current).

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **mode** (DwfAnalogOutMode) – The analog output mode to configure.

Raises
DwfLibraryError – An error occurred while executing the operation.

modeGet(*channel_index: int*) → *DwfAnalogOutMode*

Get the *AnalogOut* channel mode (voltage or current).

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured analog output mode.

Return type
DwfAnalogOutMode

Raises
DwfLibraryError – An error occurred while executing the operation.

idleInfo(*channel_index: int*) → List[*DwfAnalogOutIdle*]

Get the valid *AnalogOut* channel idle settings.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
A list of options for the channel behavior when idle.

Return type
List[DwfAnalogOutIdle]

Raises
DwfLibraryError – An error occurred while executing the operation.

idleSet(*channel_index: int*, *idle:* DwfAnalogOutIdle) → None

Set the *AnalogOut* channel idle behavior.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **idle** (DwfAnalogOutIdle) – The idle behavior setting to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

idleGet(*channel_index: int*) → *DwfAnalogOutIdle*

Get the *AnalogOut* channel idle behavior.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The *AnalogOut* channel idle behavior setting.

Return type
DwfAnalogOutIdle

Raises
DwfLibraryError – An error occurred while executing the operation.

limitationInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel limitation range, in Volts or Amps.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of limitation values that can be configured.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

limitationSet(*channel_index: int*, *limitation: float*) → None

Set the *AnalogOut* channel limitation value, in Volts or Amps.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **limitation** (*float*) – The limitation value, in Volts or Amps.

Raises
DwfLibraryError – An error occurred while executing the operation.

limitationGet(*channel_index: int*) → float

Get the *AnalogOut* channel limitation value, in Volts or Amps.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured limitation value, in Volts or Amps.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

customAMFMEnableSet(*channel_index: int*, *enable: bool*) → None

Set the *AnalogOut* channel custom AM/FM enable status.

Todo: Understand and document what this setting does.

Note: This setting is only applicable to Electronics Explorer devices, as stated in a message on the Digilent forum.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **enable** (*bool*) – The custom AM/FM enable setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

customAMFMEnableGet(*channel_index: int*) → bool

Get the *AnalogOut* channel custom AM/FM enable status.

Todo: Understand and document what this setting does.

Note: This setting is only applicable to Electronics Explorer devices, as stated in a message on the Digilent forum.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The custom AM/FM enable state.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeInfo(*channel_index: int*) → List[*DwfAnalogOutNode*]

Get a list of valid *AnalogOut* channel nodes.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The valid nodes for this channel.

Return type
List[DwfAnalogOutNode]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeEnableSet(*channel_index: int*, *node:* DwfAnalogOutNode, *mode: int*) → None

Enabled or disable an *AnalogOut* channel node.

The carrier node enables or disables the channel or selects the modulation. With channel_index -1, each analog-out channel enable mode will be configured to the same, new option.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel. Specify -1 to configure all *AnalogOut* channels.
- **node** (DwfAnalogOutNode) – The channel node.
- **mode** (*int*) – The enable mode.

Note: The precise meaning of the mode parameter is not clear from the documentation.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeEnableGet(*channel_index: int*, *node:* DwfAnalogOutNode) → int

Get the enabled state of an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured enable mode setting.

Return type
int

Note: The precise meaning of the mode parameter is not clear from the documentation.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFunctionInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → List[*DwfAnalogOutFunction*]

Get the valid waveform shape function options of an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The available node waveform shape functions.

Return type
List[DwfAnalogOutFunction]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFunctionSet(*channel_index: int*, *node:* DwfAnalogOutNode, *func:* DwfAnalogOutFunction) → None

Set the waveform shape function for an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **func** (DwfAnalogOutFunction) – The waveform shape function to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFunctionGet(*channel_index: int*, *node:* DwfAnalogOutNode) → *DwfAnalogOutFunction*

Get the waveform shape function for an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured waveform shape function.

Return type
DwfAnalogOutNode

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFrequencyInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get the channel node valid frequency range for an *AnalogOut* channel node, in Hz.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The range of valid frequencies, in Hz.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFrequencySet(*channel_index: int*, *node:* DwfAnalogOutNode, *frequency: float*) → None

Set the channel node frequency for an *AnalogOut* channel node, in Hz.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **frequency** (*float*) – The frequency, in Hz.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeFrequencyGet(*channel_index: int*, *node:* DwfAnalogOutNode) → float

Get the frequency for an *AnalogOut* channel node, in Hz.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured frequency, in Hz.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeAmplitudeInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get the amplitude range for an *AnalogOut* channel node, in Volts.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The range of allowed amplitude values, in Volts.

Return type
tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeAmplitudeSet(*channel_index: int*, *node:* DwfAnalogOutNode, *amplitude: float*) → None

Set the amplitude for an *AnalogOut* channel node, in Volts.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **amplitude** (*float*) – The amplitude to be configured, in Volts.

Raises

DwfLibraryError – An error occurred while executing the operation.

nodeAmplitudeGet(*channel_index: int*, *node:* DwfAnalogOutNode) → float

Get the amplitude for an *AnalogOut* channel node, in Volts.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns

The currently configured amplitude, in Volts.

Return type

float

Raises

DwfLibraryError – An error occurred while executing the operation.

nodeOffsetInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get the valid offset range for an *AnalogOut* channel node, in Volts.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns

The range of valid node offsets, in Volts.

Return type

Tuple[*float*, *float*]

Raises

DwfLibraryError – An error occurred while executing the operation.

nodeOffsetSet(*channel_index: int*, *node:* DwfAnalogOutNode, *offset: float*) → None

Set the offset for an *AnalogOut* channel node, in Volts.

Note: Configuring the offset of the *Carrier* node takes a noticeable amount of time (100s of milliseconds).

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **offset** (*float*) – The channel offset to be configured, in Volts.

Raises

DwfLibraryError – An error occurred while executing the operation.

nodeOffsetGet(*channel_index: int*, *node:* DwfAnalogOutNode) → float

Get the offset for an *AnalogOut* channel node, in Volts.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured node offset.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeSymmetryInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get the *symmetry* range for an *AnalogOut* channel node.

The *symmetry* value alters the waveform shape function of the node.

The *symmetry* value ranges from 0 to 100 for most waveform shape functions, except for the *SinePower* waveform shape function, where it ranges from -100 to +100.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The range of valid symmetry settings.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeSymmetrySet(*channel_index: int*, *node:* DwfAnalogOutNode, *symmetry: float*) → None

Set the *symmetry* value for an *AnalogOut* channel node.

The *symmetry* value alters the waveform shape function of the node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **symmetry** (*float*) – The symmetry setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeSymmetryGet(*channel_index: int*, *node:* DwfAnalogOutNode) → float

Get the *symmetry* value for an *AnalogOut* channel node.

The *symmetry* value alters the waveform shape function of the node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured channel node symmetry value.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

nodePhaseInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get the valid phase range for an *AnalogOut* channel node, in degrees.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The range of valid channel node phase values, in degrees.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodePhaseSet(*channel_index: int*, *node:* DwfAnalogOutNode, *phase: float*) → None

Set the phase for an *AnalogOut* channel node, in degrees.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **phase** (*float*) – The phase setting, in degrees.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodePhaseGet(*channel_index: int*, *node:* DwfAnalogOutNode) → float

Get the phase for an *AnalogOut* channel node, in degrees.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The currently configured node phase value, in degrees.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeDataInfo(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[float, float]

Get data range for an *AnalogOut* channel node, in samples.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The range of valid values.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodeDataSet(*channel_index: int*, *node:* DwfAnalogOutNode, *data: ndarray*) → None

Set the data for an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **data** (*np.ndarray*) – The data.

Raises
DwfLibraryError – An error occurred while executing the operation.

nodePlayStatus(*channel_index: int*, *node:* DwfAnalogOutNode) → Tuple[int, int, int]

Get the play status for an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.

Returns
The *free*, *lost*, and *corrupted* status counts, in samples.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

nodePlayData(*channel_index: int*, *node:* DwfAnalogOutNode, *data: ndarray*) → None

Provide the playback data for an *AnalogOut* channel node.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **node** (DwfAnalogOutNode) – The channel node.
- **data** (*np.ndarray*) – The playback data.

Raises
DwfLibraryError – An error occurred while executing the operation.

enableSet(*channel_index: int*, *enable: bool*) → None

Enable or disable the specified *AnalogOut* channel.

> **Warning: This method is obsolete.**
>
> Use the *nodeEnableSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **enable** (*bool*) – The enable setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

enableGet(*channel_index: int*) → bool

Get the current enable/disable status of the specified *AnalogOut* channel.

> **Warning: This method is obsolete.**
>
> Use the *nodeEnableGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The 'enable' state of the channel.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

functionInfo(*channel_index: int*) → List[*DwfAnalogOutFunction*]

Get the *AnalogOut* channel waveform shape function info.

> **Warning: This method is obsolete.**
>
> Use the *nodeFunctionInfo()* method instead.

Returns
The valid waveform shape functions.

Return type
List[DwfAnalogOutFunction]

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Raises
DwfLibraryError – An error occurred while executing the operation.

functionSet(*channel_index: int*, *func:* DwfAnalogOutFunction) → None

Set the *AnalogOut* channel waveform shape function.

> **Warning: This method is obsolete.**
>
> Use the *nodeFunctionSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **func** (DwfAnalogOutFunction) – The waveform shape function to use.

Raises
DwfLibraryError – An error occurred while executing the operation.

functionGet(*channel_index: int*) → *DwfAnalogOutFunction*

Get the *AnalogOut* channel waveform shape function.

> **Warning: This method is obsolete.**
>
> Use the *nodeFunctionGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured waveform shape function.

Return type
DwfAnalogOutFunction

Raises
DwfLibraryError – An error occurred while executing the operation.

frequencyInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel valid frequency range, in Hz.

> **Warning: This method is obsolete.**
>
> Use the *nodeFrequencyInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The valid frequency range, in Hz.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

frequencySet(*channel_index: int*, *frequency: float*) → None

Set the *AnalogOut* channel frequency, in Hz.

> **Warning: This method is obsolete.**
>
> Use the *nodeFrequencySet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **frequency** (*float*) – The frequency to use.

Raises
DwfLibraryError – An error occurred while executing the operation.

frequencyGet(*channel_index: int*) → float

Get the *AnalogOut* channel frequency, in Hz.

> **Warning: This method is obsolete.**
>
> Use the *nodeFrequencyGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured frequency, in Hz.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

amplitudeInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel amplitude range info.

> **Warning: This method is obsolete.**
>
> Use the *nodeAmplitudeInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of valid amplitudes, in Volts.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

amplitudeSet(*channel_index: int*, *amplitude: float*) → None

Set the *AnalogOut* channel amplitude.

> **Warning: This method is obsolete.**
>
> Use the *nodeAmplitudeSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **amplitude** (*float*) – The amplitude, in Volts.

Raises
DwfLibraryError – An error occurred while executing the operation.

amplitudeGet(*channel_index: int*) → float

Get the *AnalogOut* channel amplitude.

> **Warning: This method is obsolete.**
>
> Use the *nodeAmplitudeGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured amplitude, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

offsetInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel offset range info, in Volts.

> **Warning: This method is obsolete.**
>
> Use the *nodeOffsetInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The valid range of offset values, in Volts.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

offsetSet(*channel_index: int*, *offset: float*) → None

Set the *AnalogOut* channel offset, in Volts.

> **Warning: This method is obsolete.**
>
> Use the *nodeOffsetSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **offset** (*float*) – The channel offset, in Volts.

Raises
DwfLibraryError – An error occurred while executing the operation.

offsetGet(*channel_index: int*) → float

Get the *AnalogOut* channel offset, in Volts.

> **Warning: This method is obsolete.**
>
> Use the *nodeOffsetGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The valid offset value, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

symmetryInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel symmetry setting range.

> **Warning: This method is obsolete.**
>
> Use the *nodeSymmetryInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of valid symmetry settings.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

symmetrySet(*channel_index: int*, *symmetry: float*) → None

Set the *AnalogOut* channel symmetry setting.

> **Warning: This method is obsolete.**
>
> Use the *nodeSymmetrySet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **symmetry** (*float*) – The channel symmetry setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

symmetryGet(*channel_index: int*) → float

Get the *AnalogOut* channel symmetry setting.

> **Warning: This method is obsolete.**
>
> Use the *nodeSymmetryGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured symmetry setting.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

phaseInfo(*channel_index: int*) → Tuple[float, float]

Get the *AnalogOut* channel phase range, in degrees.

> **Warning: This method is obsolete.**
>
> Use the *nodePhaseInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The range of valid phase values, in degrees.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

phaseSet(*channel_index: int*, *phase: float*) → None

Set the *AnalogOut* channel phase, in degrees.

> **Warning: This method is obsolete.**
>
> Use the *nodePhaseSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **phase** (*float*) – The phase setting, in degrees.

Raises
DwfLibraryError – An error occurred while executing the operation.

phaseGet(*channel_index: int*) → float

Get the *AnalogOut* channel phase, in degrees.

> **Warning: This method is obsolete.**
>
> Use the *nodePhaseGet()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The currently configured phase, in degrees.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

dataInfo(*channel_index: int*) → Tuple[int, int]

Get the *AnalogOut* channel data buffer range.

> **Warning: This method is obsolete.**
>
> Use the *nodeDataInfo()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The data range.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

dataSet(*channel_index: int*, *data: ndarray*) → None

Set the *AnalogOut* channel data.

> **Warning: This method is obsolete.**
>
> Use the *nodeDataSet()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **data** (*np.ndarray*) – The data.

Raises
DwfLibraryError – An error occurred while executing the operation.

playStatus(*channel_index: int*) → Tuple[int, int, int]

Get the *AnalogOut* channel playback status, in samples.

> **Warning: This method is obsolete.**
>
> Use the *nodePlayStatus()* method instead.

Parameters
channel_index (*int*) – The *AnalogOut* channel.

Returns
The playback status.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

playData(*channel_index: int*, *data: ndarray*) → None

Provide the *AnalogOut* channel playback data.

> **Warning: This method is obsolete.**
>
> Use the *nodePlayData()* method instead.

Parameters

- **channel_index** (*int*) – The *AnalogOut* channel.
- **data** (*np.ndarray*) – The playback data.

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.4 Analog I/O

The *AnalogIO* API provides two types of analog functionality:

- **Monitoring** of on-board sensors, e.g. for voltages, currents, and temperatures;
- **Control** of voltage (power) supplies, on Digilent Waveforms devices that support it.

The *AnalogIO* functionality, despite its name, does not overlap with the functionality of the *AnalogIn* and *AnalogOut* instruments. It cannot be used to control the analog signal outputs or to monitor the analog signal inputs.

An exception is that the *AnalogOut* instrument on an Analog Discovery 2 device (and perhaps some others) can be made to control the voltage sources of the device as if they were regular, low bandwidth analog output channels, by selecting an appropriate device configuration when opening the device. This feature can be useful in rare cases when a non-constant supply voltage is needed for testing. However, in most cases by far, only a static (constant) voltage is needed, and the easiest way to accomplish that is via the *AnalogIO* API.

4.4.1 Using the Analog I/O functionality

To use the *AnalogIO* functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *AnalogIO* API can now be accessed via its *analogIO* attribute, which is an instance of the *AnalogIO* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's AnalogIO functionality.
    analogIO = device.analogIO

    # Use the AnalogIO functionality.
    analogIO.reset()
```

4.4.2 *AnalogIO* channels and nodes

The quantities that can be monitored and controlled by the *AnalogIO* functionality are organized in *channels*. Each channel can have one or more *nodes*.

A typical example is the *USB Monitor* channel of the Analog Discovery 2 device. It has 3 nodes: *Voltage*, *Current*, and *Temperature*, that can be used to report on these three quantities.

The *USB Monitor* channel is fully passive; its nodes can only be monitored. Other channels have nodes that can also be controlled, for example the *Positive Supply* and *Negative Supply* channels of the Analog Discovery 2. Those channels have two nodes each that can be controlled (*Enabled* and *Voltage*), and one node that can only be monitored (*Current*).

The AnalogIO.py example program enumerates the *AnalogIO* channels and nodes of any Digilent Waveforms device; it is recommended to study that program to understand how to use the *AnalogIO* functionality.

Below, by way of example, we show the channels and nodes of three Digilent Waveforms devices: the Analog Discovery 2, the Digital Discovery, and the Analog Discovery Pro.

Table 23: *AnalogIO* channels and nodes of the Analog Discovery 2 device

channel	channel name	ch. label	node	node name	unit	node type
0	Positive Supply	V+	0	Enable	*n/a*	*Enable*
			1	Voltage	V	*Voltage*
			2	Current	A	*Current*
1	Negative Supply	V-	0	Enable	*n/a*	*Enable*
			1	Voltage	V	*Voltage*
			2	Current	A	*Current*
2	USB Monitor	USB	0	Voltage	V	*Voltage*
			1	Current	A	*Current*
			2	Temperature	C	*Temperature*
3	Auxiliary Monitor	Aux	0	Voltage	V	*Voltage*
			1	Current	A	*Current*
4	Power Supply	V+-	0	Limit	*n/a*	*Enable*

Table 24: *AnalogIO* channels and nodes of the Digital Discovery device

channel	channel name	ch. label	node	node name	unit	node type
0	Digital Voltage	VDD	0	Voltage	V	*Voltage*
			1	DINPP	*n/a*	*Enable*
			2	DIOPE	*n/a*	*Enable*
			3	DIOPP	*n/a*	*Enable*
			4	Drive	*n/a*	*Current*
			5	Slew	*n/a*	*Enable*
			6	Clock	Hz	*Frequency*
1	Voltage Output	VIO	0	Voltage	V	*Voltage*
			1	Current	A	*Current*
2	USB Monitor	USB	0	Voltage	V	*Voltage*
			1	Current	A	*Current*

Table 25: *AnalogIO* channels and nodes of the Analog Discovery Pro (ADP3450) device

channel	channel name	ch. label	node	node name	unit	node type
0	Digital Voltage	DVCC	0	Voltage		*Voltage*
			1	DIOPE	*n/a*	*Enable*
			2	DIOPP	*n/a*	*Enable*
1	Zynq	Zynq	0	Temperature	C	*Temperature*
			1	VccInt	V	*Voltage*
			2	VccAux	V	*Voltage*
			3	VccBRam	V	*Voltage*
			4	VccPInt	V	*Voltage*
			5	VccPAux	V	*Voltage*
			6	VccDDR	V	*Voltage*
2	ZynqMin	ZynqMin	0	Temperature	C	*Temperature*
			1	VccInt	V	*Voltage*
			2	VccAux	V	*Voltage*
			3	VccBRam	V	*Voltage*
			4	VccPInt	V	*Voltage*
			5	VccPAux	V	*Voltage*
			6	VccDDR	V	*Voltage*
3	ZynqMax	ZynqMax	0	Temperature	C	*Temperature*
			1	VccInt	V	*Voltage*
			2	VccAux	V	*Voltage*
			3	VccBRam	V	*Voltage*
			4	VccPInt	V	*Voltage*
			5	VccPAux	V	*Voltage*
			6	VccDDR	V	*Voltage*

Note: Channel 2 (ZynqMin), node 5 of the Analog Discovery Pro device was reported incorrectly in version 3.16.3 of the DWF library. This has been corrected in version 3.17.1.

4.4.3 *AnalogIO* reference

class AnalogIO

The *AnalogIO* class provides access to the analog I/O functionality of a *DwfDevice*.

The *AnalogIO* methods are used to control the power supplies, reference voltage supplies, voltmeters, ammeters, thermometers, and any other sensors on the device. These are organized into channels which contain a number of nodes. For instance, a power supply channel might have three nodes: an 'enable' setting, a voltage level setting/reading, and current limitation setting/reading.

Attention: Users of *pydwf* should not create instances of this class directly.

It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *analogIO* attribute for access by the user.

reset() → None

Reset and configure all *AnalogIO* settings to default values.

If autoconfiguration is enabled, the changes take effect immediately.

Raises

DwfLibraryError – An error occurred while executing the *reset* operation.

configure() → None

Configure the *AnalogIO* functionality.

This method transfers the settings to the Digilent Waveforms device. It is not needed if autoconfiguration is enabled.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status() → None

Read the status of the device and stores it internally.

The status inquiry methods that follow will return the information that was read from the device when this method was last called.

Note that the *AnalogIO* functionality is not managed by a state machine, so this method does not return a value.

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

enableInfo() → Tuple[bool, bool]

Verify if *Master Enable* and/or *Master Enable Status* are supported.

The *Master Enable* is a software switch that enable the *AnalogIO* voltage sources.

If supported, the current value of this *Master Enable* switch (Enabled/Disabled) can be set by the *enableSet()* method and queried by the *enableGet()* method.

The *Master Enable Status* that can be queried by the *enableStatus()* method may be different from the *Master Enable* value if e.g. an over-current protection circuit has been triggered.

Returns
The tuple elements indicate whether *Master Enable Set* and *Master Enable Status*, respectively, are supported by the *AnalogIO* device.

Return type
Tuple[*bool*, *bool*]

Raises
DwfLibraryError – An error occurred while executing the operation.

enableSet(*master_enable: bool*) → None

Set value of the *Master Enable* setting.

Parameters
master_enable (*bool*) – The new value of the *Master Enable* setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

enableGet() → bool

Return the current value of the *Master Enable* setting.

Returns
The value of the *Master Enable* setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

enableStatus() → bool

Return the actual *Master Enable Status* value (if the device supports it).

The *Master Enable Status* value may be different from the *Master Enable* setting if e.g. an over-current protection circuit has been triggered.

Returns
the current status of the *Master Enable* circuit.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

channelCount() → int

Return the number of *AnalogIO* channels available on the device.

Returns
The number of *AnalogIO* channels.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

channelName(*channel_index: int*) → Tuple[str, str]

Return the name (long text) and label (short text, printed on the device) for the specified *AnalogIO* channel.

Parameters
channel_index (*int*) – The channel for which we want to get the name and label.

Returns
The name and label of the channel.

Return type
Tuple[*str*, *str*]

Raises
DwfLibraryError – An error occurred while executing the operation.

channelInfo(*channel_index: int*) → int

Return the number of nodes associated with the specified *AnalogIO* channel.

Parameters
channel_index (*int*) – The channel for which we want to get the number of associated nodes.

Returns
The number of nodes associated to the channel.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeName(*channel_index: int*, *node_index: int*) → Tuple[str, str]

Return the node name (“Voltage”, “Current”, . . .) and units (“V”, “A”, . . .) for the specified *AnalogIO* node.

Parameters

- **channel_index** (*int*) – The channel for which we want to get the name and unit.
- **node_index** (*int*) – The node for which we want to get the name and unit.

Returns
The name and unit of the quantity associated with the node.

Return type
Tuple[*str*, *str*]

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeInfo(*channel_index: int*, *node_index: int*) → *DwfAnalogIO*

Return the type of physical quantity (e.g., voltage, current, or temperature) represented by the specified *AnalogIO* channel node.

Parameters

- **channel_index** (*int*) – The channel for which we want to get the type of physical quantity.
- **node_index** (*int*) – The node for which we want to get the type of physical quantity.

Returns
The type of physical quantity represented by the node.

Return type
DwfAnalogIO

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeSetInfo(*channel_index: int*, *node_index: int*) → Tuple[float, float, int]

Return the limits of the value that can be assigned to the specified *AnalogIO* channel node.

Since a node can represent many things (power supply, temperature sensor, etc.), the *minimum*, *maximum*, and *steps* parameters also represent different types of values.

The *channelNodeInfo()* method returns the type of values to expect and the *channelNodeName()* method returns the units of these values.

Parameters

- **channel_index** (*int*) – The channel for which we want to get the currently configured value.
- **node_index** (*int*) – The node for which we want to get the currently configured value.

Returns
The minimum and maximum values for the specified node's value, and the number of resolution steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeSet(*channel_index: int*, *node_index: int*, *node_value: float*) → None

Set the node value for the specified *AnalogIO* channel node.

Parameters

- **channel_index** (*int*) – The channel for which we want to set the value.
- **node_index** (*int*) – The node for which we want to set the value.
- **node_value** (*float*) – The value we want to set the node to.

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeGet(*channel_index: int*, *node_index: int*) → float

Return the current value of the specified *AnalogIO* channel node.

Parameters

- **channel_index** (*int*) – The channel for which we want to get the current value.
- **node_index** (*int*) – The node for which we want to get the current value.

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeStatusInfo(*channel_index: int*, *node_index: int*) → Tuple[float, float, int]

Return the range of status values for the specified *AnalogIO* channel node.

Parameters

- **channel_index** (*int*) – The channel for which we want to get status information.
- **node_index** (*int*) – The node for which we want to get status information.

Returns
The minimum and maximum status values for the specified node, and the number of resolution steps.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

channelNodeStatus(*channel_index: int*, *node_index: int*) → float

Return the most recent status value reading of the specified *AnalogIO* channel node.

To fetch updated values for all *AnalogIO* nodes, use the *status()* method.

Returns
The most recent value read for this channel node.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.5 Analog impedance measurements

The *AnalogImpedance* functionality supports analog measurements of signal propagation properties on devices that support it, such as the Analog Discovery and the Analog Discovery 2.

Todo: This section is currently incomplete.

It lacks information on the state machine used in the *AnalogImpedance* API, what the different settings mean, what the different methods actually do, and how they can be used to perform measurements.

There are also no *AnalogImpedance* examples yet.

4.5.1 Using the analog impedance measurements

To use the *AnalogImpedance* functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *AnalogImpedance* functionality can now be accessed via its *analogImpedance* attribute, which is an instance of the *AnalogImpedance* class.

For example:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's AnalogImpedance functionality.
    analogImpedance = device.analogImpedance

    # Use the AnalogImpedance functionality.
    analogImpedance.reset()
```

4.5.2 *AnalogImpedance* reference

class AnalogImpedance

The *AnalogImpedance* class provides access to the analog impedance measurement functionality of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *analogImpedance* attribute for access by the user.

reset() → None

Reset the *AnalogImpedance* functionality.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

configure(*start: bool*) → None

Configure the *AnalogImpedance* functionality, and optionally start a measurement.

Parameters
start (*bool*) – Whether to start the measurement.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status() → *DwfState*

Return the status of the *AnalogImpedance* functionality.

Returns
The status of the measurement.

Return type
DwfState

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

modeSet(*mode: int*) → None

Set *AnalogImpedance* measurement mode.

Parameters
mode (*int*) – The measurement mode.

The following modes are defined:

- 0 — W1-C1-DUT-C2-R-GND
- 1 — W1-C1-R-C2-DUT-GND
- 8 — Impedance analyzer for the Analog Discovery

Raises
DwfLibraryError – An error occurred while executing the operation.

modeGet() → int

Get *AnalogImpedance* measurement mode.

Returns

The measurement mode.

The following modes are defined:

- 0 — W1-C1-DUT-C2-R-GND
- 1 — W1-C1-R-C2-DUT-GND
- 8 — Impedance analyzer for the Analog Discovery

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

referenceSet(*reference: float*) → None

Set *AnalogImpedance* reference load value, in Ohms.

Parameters
reference (*float*) – The reference load, in Ohms.

Raises
DwfLibraryError – An error occurred while executing the operation.

referenceGet() → float

Get *AnalogImpedance* reference load value, in Ohms.

Returns
The reference load, in Ohms.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

frequencySet(*frequency: float*) → None

Set *AnalogImpedance* source frequency, in Hz.

Parameters
frequency (*float*) – The source frequency, in Hz.

Raises
DwfLibraryError – An error occurred while executing the operation.

frequencyGet() → float

Get *AnalogImpedance* source frequency, in Hz.

Returns
The source frequency, in Hz.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

amplitudeSet(*amplitude: float*) → None

Set *AnalogImpedance* source amplitude value, in Volts.

Parameters
amplitude (*float*) – The source amplitude, in Volts.

Raises
DwfLibraryError – An error occurred while executing the operation.

amplitudeGet() → float

Get *AnalogImpedance* source amplitude, in Volts.

Returns
The source amplitude, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

offsetSet(*offset: float*) → None

Set *AnalogImpedance* source offset, in Volts.

Parameters
offset (*float*) – The source offset, in Volts.

Raises
DwfLibraryError – An error occurred while executing the operation.

offsetGet() → float

Get *AnalogImpedance* source offset, in Volts.

Returns
The source offset, in Volts.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

probeSet(*resistance: float*, *capacitance: float*) → None

Set *AnalogImpedance* probe parameters.

Parameters

- **resistance** (*float*) – The probe resistance, in Ohms.
- **capacitance** (*float*) – The probe capacitance, in Farads.

Raises
DwfLibraryError – An error occurred while executing the operation.

probeGet() → Tuple[float, float]

Get *AnalogImpedance* probe parameters.

Returns
A two-element tuple: the probe resistance, in Ohms, and the probe capacity, in Farads.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

periodSet(*period: int*) → None

Set the *AnalogImpedance* measurement period.

Todo: Figure out what this setting is for, why it's an *int*, and what its physical unit is.

Parameters
period (*int*) – The measurement period.

Raises
DwfLibraryError – An error occurred while executing the operation.

periodGet() → int

Get the *AnalogImpedance* measurement period.

Todo: Figure out what this setting is for, why it's an *int*, and what its physical unit is.

Returns
The measurement period.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

compReset() → None

Reset the *AnalogImpedance* measurement computation.

Raises
DwfLibraryError – An error occurred while executing the operation.

compSet(*open_resistance: float*, *open_reactance: float*, *short_resistance: float*, *short_reactance: float*) → None

Set the *AnalogImpedance* measurement computation parameters.

Parameters

- **open_resistance** (*float*) – The open-circuit resistance, in Ohms.
- **open_reactance** (*float*) – The open-circuit reactance, in Ohms.
- **short_resistance** (*float*) – The short-circuit resistance, in Ohms.
- **short_reactance** (*float*) – The short-circuit reactance, in Ohms.

Raises
DwfLibraryError – An error occurred while executing the operation.

compGet() → Tuple[float, float, float, float]

Get the *AnalogImpedance* measurement computation parameters.

Returns
The open-circuit resistance, open-circuit reactance, short-circuit resistance, and short-circuit reactance (all in Ohms).

Return type
Tuple[*float*, *float*, *float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

statusInput(*channel_index: int*) → Tuple[float, float]

Get the *AnalogImpedance* input measurement status.

Parameters
channel_index (*int*) – The channel for which to get gain and phase information.

Returns
The gain and phase (in radians) of the current measurement.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

statusWarning(*channel_index: int*) → int

Get warning for scope input range exceeded.

Parameters
channel_index (*int*) – The channel for which to get scope input range warning information.

Returns
The warning, if any. 1 means *low*, 2 means *high*, 3 means *both*.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

statusMeasure(*measure: DwfAnalogImpedance*) → float

Retrieve the *AnalogImpedance* measurement status value.

Parameters
measure (DwfAnalogImpedance) – The quantity to measure.

Returns
The value measured for the requested quantity.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.6 Digital input instrument

The *DigitalIn* instrument provides multiple channels of digital input on devices that support it, such as the Analog Discovery, Analog Discovery 2, and Digital Discovery. It provides the functionality normally associated with a stand-alone logic analyzer.

Todo: This section is missing some important information:

- A discussion about the different acquisition modes;
- A description of how the status variables behave in the different acquisition modes;
- A discussion of the precise meaning of all settings.

4.6.1 Using the digital input instrument

To use the *DigitalIn* instrument you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *DigitalIn* instrument can now be accessed via its `digitalIn` attribute, which is an instance of the `DigitalIn` class.

For example:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's DigitalIn instrument.
    digitalIn = device.digitalIn

    # Use the DigitalOut instrument.
    digitalIn.reset()
```

4.6.2 The *DigitalIn* instrument state machine

The *DigitalIn* instrument is controlled by a state machine. As a measurement is prepared and executed, the instrument goes through its various states.

The current state of the instrument is returned by the `digitalIn.status()` method, and is of type `DwfState`.

The figure below shows the states used by the *DigitalIn* instrument and the transitions between them:

Fig. 3: States of the *DigitalIn* instrument

The *DigitalIn* states are used as follows:

1. `Ready`

   In this preparatory state, instrument settings can be changed that specify the behavior of the instrument in the coming measurement. If the auto-configure setting of the device is enabled (the default), setting changes will

automatically be transferred to the device. If not, an explicit call to the *digitalIn.configure()* method with the *reconfigure* parameter set to True is needed to transfer updated settings to the device.

Once the instrument is properly configured, an acquisition can be started by calling the *digitalIn.configure()* with the *start* parameter set to True. This will start the first stage of the acquisition by entering the *Prefill* state.

2. *Configure*

   This state is entered momentarily when a setting is being pushed to the device, either by changing the setting while auto-configure is enabled, or by an explicit call to *digitalIn.configure()* with the *reconfigure* parameter set to True. The settings inside the device will be updated, and the device will immediately thereafter go back to the *Ready* state, unless the *start* parameter to *digitalIn.configure()* was set to True.

3. *Prefill*

   This state marks the beginning of an acquisition sequence. During the *Prefill* state, input samples will be acquired until enough samples are buffered for the instrument to be ready to react to a trigger.

   This state is only relevant if the trigger position has been configured in such a way that the measurement must also yield sample values prior to the moment of triggering.

   Once enough samples are received for the instrument to be able to react to a trigger, it proceeds to the *Armed* state.

4. *Armed*

   In this state the instrument continuously captures samples and monitors the configured trigger input. As soon as a trigger event is detected, the instrument proceeds to the *Running* state.

5. *Running*

   In this state the instrument continues capturing samples until the acquisition is complete. Completion is reached when the acquisition buffer has filled up in *Single* mode, or when the recording length has been reached in *Record* mode. When completion is reached, the instrument proceeds to the *Done* state.

6. *Done*

   This state indicates that a measurement has finished.

   From this state, it is possible to go back to the *Ready* state by performing any kind of configuration, or to start a new acquisition with the same settings..

4.6.3 *DigitalIn* instrument API overview

The *DigitalIn* instrument is quite complicated; 62 methods are provided to control its behavior. Below, we categorize all methods and shortly introduce them. Detailed information on all methods can be found in the *DigitalIn* class reference that follows.

Instrument control

Like all instruments supported by the Digilent Waveforms library, the *DigitalIn* instrument provides *reset()*, *configure()*, and *status()* methods.

The *reset()* method resets the instrument.

The *configure()* method is used to explicitly transfer settings to the instrument, and/or to start a configured operation.

The *status()* method retrieves status information from the instrument. Optionally, it can also retrieve bulk data, i.e. digital input samples. The method returns the current *DwfState* of the *DigitalIn* instrument; to obtain more elaborate status information, one of the methods in the next two sections must be used.

Table 26: Instrument control (3 methods)

control operation	type/unit	methods
reset instrument	*n/a*	*reset()*
configure instrument	*n/a*	*configure()*
request instrument status	*DwfState*	*status()*

Status variables

When executing the *status()* method, status information is transferred from the *DigitalIn* instrument to the PC. Several status variables can then be retrieved by using the methods listed below.

Table 27: Status variables (7 methods)

status value	type/unit	method
timestamp	tuple [s]	*statusTime()*
auto-triggered flag	bool	*statusAutoTriggered()*
samples left in acquisition	int [samples]	*statusSamplesLeft()*
samples valid count	int [samples]	*statusSamplesValid()*
buffer write index	int [samples]	*statusIndexWrite()*
recording status	tuple [samples]	*statusRecord()*
compressed status	tuple [samples]	*statusCompress()*

Status data retrieval

Executing the *status()* method with the *read_data* parameter set to True transfers captured samples from the instrument to the PC. The samples can then be retrieved using the methods listed here.

Table 28: Bulk status data retrieval (5 methods)

status data	type/unit	methods
get sample data (without buffer offset)	[bytes]	*statusData()*
get sample data (with buffer offset)	[bytes]	*statusData2()*
get compressed sample data (without buffer offset)	[bytes]	*statusCompressed()*
get compressed sample data (with buffer offset)	[bytes]	*statusCompressed2()*
get sample noise (with offset)	[bytes]	*statusNoise2()*

Acquisition timing settings

The acquisition settings control the timing of the digital data acquisition process.

Table 29: Acquisition timing settings (7 methods)

setting	type/unit	methods
internal clock	float [Hz]	*internalClockInfo()*
clock source	*DwfDigitalInClockSource*	*clockSourceInfo()* , *-Set()* , *-Get()*
divider	int [-]	*dividerInfo()* , *-Set()* , *-Get()*

Acquisition settings

The acquisition settings control various aspects of the digital data acquisition process.

Table 30: Acquisition settings (17 methods)

setting	type/unit	methods
acquisition mode	*DwfAcquisitionMode*	*acquisitionModeInfo()* , *-Set()* , *-Get()*
bits	int [-]	*bitsInfo()*
sample format	int [bits]	*sampleFormatSet()* , *-Get()*
input order	bool	*inputOrderSet()*
buffer size	int [samples]	*bufferSizeInfo()* , *-Set()* , *-Get()*
sample mode	*DwfDigitalInSampleMode*	*sampleModeInfo()* , *-Set()* , *-Get()*
sample sensible	int	*sampleSensibleSet()* , *-Get()*
trigger prefill	int [samples]	*triggerPrefillSet()* , *-Get()*

Instrument trigger configuration

The following methods are used to configure the trigger of the *DigitalIn* instrument. The trigger source is fully configurable; the *DigitalIn* instrument can use its own trigger detector for triggering, but it is also possible to use a different trigger source. For that reason, we distinguish between the methods that configure the instrument trigger in this section, and the methods that configure the *DigitalIn* trigger detector that are discussed below.

Table 31: Instrument trigger configuration (11 methods)

setting	type/unit	methods
trigger source	*DwfTriggerSource*	*triggerSourceInfo()* , *-Set()* , *-Get()*
trigger slope	*DwfTriggerSlope*	*triggerSlopeSet()* , *-Get()*
trigger position	float [s]	*triggerPositionInfo()* , *-Set()* , *-Get()*
trigger auto-timeout	float [s]	*triggerAutoTimeoutInfo()* , *-Set()* , *-Get()*

Note: The *triggerSourceInfo()* method is obsolete. Use the generic *DwfDevice.triggerInfo()* method instead.

Trigger detector configuration

The *DigitalIn* trigger detector is highly configurable. Unfortunately, its documentation is sparse, so some experimentation is needed to figure out how it works.

Todo: Figure out and explain how the *DigitalIn* trigger detector works.

Table 32: Trigger detector configuration (7 methods)

setting	type/unit	methods
trigger	bit masks	*triggerInfo()* , *-Set()* , *-Get()*
trigger reset	*to be documented*	*triggerResetSet()*
trigger count	*to be documented*	*triggerCountSet()*
trigger length	*to be documented*	*triggerLengthSet()*
trigger match	*to be documented*	*triggerMatchSet()*

Counter functionality

Table 33: Counter configuration (4 methods)

setting	type/unit	methods
counter configuration counter status	float [s], int [-] float [s], float [Hz], int [-]	*counterInfo()* , -Set() , -Get() *counterStatus()*

Miscellaneous settings

The *mixed* setting is obsolete, undocumented, and not understood.

Todo: Figure out what the *mixed* setting does.

Table 34: Miscellaneous settings (1 method)

operation	type/unit	method
mixed	bool	*mixedSet()*

4.6.4 *DigitalIn* reference

class DigitalIn

The *DigitalIn* class provides access to the digital input (logic analyzer) instrument of a *DwfDevice*.

Attention: Users of *pydwf* should not create instances of this class directly.

It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *digitalIn* attribute for access by the user.

reset() → None

Reset the *DigitalIn* instrument parameters to default values.

If autoconfiguration is enabled, the reset values will be used immediately.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

configure(*reconfigure: bool*, *start: bool*) → None

Configure the instrument and start or stop the acquisition operation.

Parameters

- **reconfigure** (*bool*) – If True, the instrument settings are sent to the instrument.
- **start** (*bool*) – If True, an acquisition is started. If False, an ongoing acquisition is stopped.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status(*read_data_flag: bool*) → *DwfState*

Get the *DigitalIn* instrument state.

This method performs a status request to the *DigitalIn* instrument and receives its response.

The following methods can be used to retrieve *DigitalIn* instrument status information as a result of this call, regardless of the value of the *read_data_flag* parameter:

- *statusTime()*
- *statusAutoTriggered()*
- *statusSamplesLeft()*
- *statusSamplesValid()*
- *statusIndexWrite()*
- *statusRecord()*

The following methods can be used to retrieve bulk data obtained from the *DigitalIn* instrument as a result of this call, but only if the *read_data_flag* parameter is True:

- *statusData()*
- *statusData2()*
- *statusNoise2()*

Parameters
read_data_flag (*bool*) – Whether to read data.

Returns
The current state of the instrument.

Return type
DwfState

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

statusTime() → Tuple[int, int, int]

Get the timestamp of the current status information.

Returns

A three-element tuple, indicating the POSIX timestamp of the status request. The first element is the POSIX second, the second and third element are the numerator and denominator, respectively, of the fractional part of the second.

In case *status()* hasn't been called yet, this method will return zeroes for all three tuple elements.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

statusAutoTriggered() → bool

Check if the current acquisition is auto-triggered.

Returns
True if the current acquisition is auto-triggered, False otherwise.

Return type
bool

Raises
DwfLibraryError – An error occurred while retrieving the auto-triggered status.

statusSamplesLeft() → int

Retrieve the number of samples left in the acquisition, in samples.

Returns
In case a finite-duration acquisition is active, the number of samples remaining to be acquired in the acquisition.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

statusSamplesValid() → int

Retrieve the number of valid data samples.

Returns
The number of valid samples in the buffer.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

statusIndexWrite() → int

Retrieve the buffer write index.

This is needed in *ScanScreen* acquisition mode to display the scan bar.

Returns
The buffer write index.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

statusRecord() → Tuple[int, int, int]

Get the recording status.

Returns
A three-element tuple containing the counts for *available*, *lost*, and *corrupt* data samples, in that order.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

statusCompress() → Tuple[int, int, int]

Get the recording status (compress version).

Returns
A three-element tuple containing the counts for *available*, *lost*, and *corrupt* data samples, in that order.

Return type
Tuple[*int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

statusData(*count: int*, *sample_format: int | None = None*) → ndarray

Retrieve the acquired data samples from the *DigitalIn* instrument.

Parameters

- **count** (*int*) – Sample count.
- **sample_format** (*int*) – If not specified, the current value is queried using *sampleFormatGet()*.

Returns
a 1D numpy array of 8, 16, or 32-bit unsigned words.

Raises
DwfLibraryError – An error occurred while executing the operation.

statusData2(*offset: int*, *count: int*, *sample_format: int | None = None*) → ndarray

Retrieve the acquired data samples from the *DigitalIn* instrument.

Parameters

- **count** (*int*) – Sample count.
- **sample_format** (*int*) – Either 8, 16 or 32 bits per sample. If not specified, the current value is queried using *sampleFormatGet()*.

Returns
a 1D numpy array of 8, 16, or 32-bit unsigned words.

Raises
DwfLibraryError – An error occurred while executing the operation.

statusCompressed(*count_bytes: int*) → ndarray

Retrieve the compressed data samples from the *DigitalIn* instrument.

Todo: Figure out the data format.

Raises
DwfLibraryError – An error occurred while executing the operation.

statusCompressed2(*first_sample: int*, *count_bytes: int*) → ndarray

Retrieve the acquired data samples from the *DigitalIn* instrument.

Todo: Figure out the data format.

Raises
DwfLibraryError – An error occurred while executing the operation.

statusNoise2(*first_sample: int*, *count_bytes: int*) → ndarray

Get the noise data from the *DigitalIn* instrument.

Todo: Figure out the data format.

Raises
DwfLibraryError – An error occurred while executing the operation.

internalClockInfo() → float

Get the *DigitalIn* internal clock frequency, in Hz.

Returns
The internal clock frequency, in Hz.

Raises
DwfLibraryError – An error occurred while executing the operation.

clockSourceInfo() → List[*DwfDigitalInClockSource*]

Get a list of valid clock sources for the *DigitalIn* instrument.

Returns
A list of valid clock sources.

Return type
List[DwfDigitalInClockSource]

Raises
DwfLibraryError – An error occurred while executing the operation.

clockSourceSet(*clock_source:* DwfDigitalInClockSource) → None

Set the *DigitalIn* instrument clock source.

Parameters
clock_source (DwfDigitalInClockSource) – The clock source to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

clockSourceGet() → *DwfDigitalInClockSource*

Get the *DigitalIn* instrument clock source.

Returns
The currently configured *DigitalIn* clock source.

Return type
DwfDigitalInClockSource

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerInfo() → int

Get the *DigitalIn* instrument maximum divider value.

Returns
The maximum valid divider value that can be configured.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerSet(*divider: int*) → None

Set the *DigitalIn* instrument divider value.

Parameters
divider – The divider value to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerGet() → int

Get the current *DigitalIn* instrument divider value.

If the clock source is internal, the *DigitalIn* sample frequency will be equal to *internalClockInfo()* divided by *dividerGet()*.

Returns
The currently configured divider value.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

acquisitionModeInfo() → List[*DwfAcquisitionMode*]

Get a list of valid *DigitalIn* instrument acquisition modes.

Returns
A list of valid acquisition modes for the *DigitalIn* instrument.

Return type
List[DwfAcquisitionMode]

Raises
DwfLibraryError – An error occurred while executing the operation.

acquisitionModeSet(*acquisition_mode:* DwfAcquisitionMode) → None

Select the *DigitalIn* acquisition mode.

Parameters
acquisition_mode (DwfAcquisitionMode) – The acquisition mode to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

acquisitionModeGet() → *DwfAcquisitionMode*

Get the currently selected *DigitalIn* acquisition mode.

Returns
The currently selected acquisition mode.

Return type
DwfAcquisitionMode

Raises
DwfLibraryError – An error occurred while executing the operation.

bitsInfo() → int

Get the number of *DigitalIn* bits.

Returns
The number of digital input bits available.

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleFormatSet(*num_bits: int*) → None

Set the *DigitalIn* sample format (i.e., number of bits).

Parameters
num_bits (*int*) – The number of bits per sample (8, 16, or 32).

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleFormatGet() → int

Get the *DigitalIn* sample format (i.e., number of bits).

Returns
The currently configured number of bits per sample (8, 16, or 32).

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

inputOrderSet(*dio_first: bool*) → None

Select the *DigitalIn* order of values stored in the sampling array.

If *dio_first* is True, DIO 24…39 are placed at the beginning of the array followed by DIN 0…23.

If *dio_first* is False, DIN 0…23 are placed at the beginning followed by DIO 24…31.

This method is valid only for the Digital Discovery device.

Parameters
dio_first (*bool*) – Whether the DIO pins come before the DIN pins.

Raises
DwfLibraryError – An error occurred while executing the operation.

bufferSizeInfo() → int

Get the *DigitalIn* instrument maximum buffer size.

Returns
The maximum valid buffer size.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

bufferSizeSet(*buffer_size: int*) → None

Set the *DigitalIn* instrument buffer size.

Parameters
buffer_size (*int*) – The buffer size to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

bufferSizeGet() → int

Get the currently configured *DigitalIn* instrument buffer size.

Returns
The currently configured buffer size.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleModeInfo() → List[*DwfDigitalInSampleMode*]

Get the valid *DigitalIn* instrument sample modes.

Returns
A list of valid sample modes.

Return type
List[DwfDigitalInSampleMode]

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleModeSet(*sample_mode:* DwfDigitalInSampleMode) → None

Set the *DigitalIn* instrument sample mode.

Parameters
sample_mode (DwfDigitalInSampleMode) – The sample mode to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleModeGet() → *DwfDigitalInSampleMode*

Get the *DigitalIn* instrument sample mode.

Returns
The currently configured sample mode.

Return type
DwfDigitalInSampleMode

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleSensibleSet(*compression_bits: int*) → None

Set the *DigitalIn* instrument *sample sensible* setting.

Todo: Figure out what this setting does.

Parameters
compression_bits (*int*) – (unknown)

Raises
DwfLibraryError – An error occurred while executing the operation.

sampleSensibleGet() → int

Get the *DigitalIn* instrument *sample sensible* setting.

This setting is only used in *Record* mode.

It select the signals to be used for data compression in record acquisition mode.

Todo: Figure out what this setting does.

Returns
The sample sensible setting.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerPrefillSet(*samples_before_trigger: int*) → None

Set the *DigitalIn* instrument trigger prefill setting, in samples.

This setting is only used in *Record* mode.

It determines the number of samples to acquire before arming in *Record* acquisition mode. The prefill is used for recording with a trigger to make sure that at least the required number of samples are collected before arming.

Todo: Figure out what this setting does, precisely.

Parameters
samples_before_trigger (*int*) – The prefill count, in samples.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerPrefillGet() → int

Get the *DigitalIn* instrument trigger prefill setting, in samples.

Returns
The trigger prefill count, in samples.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSourceInfo() → List[*DwfTriggerSource*]

Get the valid *DigitalIn* instrument trigger sources.

> **Warning: This method is obsolete.**
>
> Use the generic `DeviceControl.triggerInfo()` method instead.

Returns
A list of valid trigger sources.

Return type
List[DwfTriggerSource]

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSourceSet(*trigger_source:* DwfTriggerSource) → None

Set *DigitalIn* instrument trigger source.

Parameters
trigger_source (DwfTriggerSource) – The trigger source to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSourceGet() → *DwfTriggerSource*

Get the currently selected instrument trigger source.

Returns
The currently selected instrument trigger source.

Return type
DwfTriggerSource

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSlopeSet(*trigger_slope:* DwfTriggerSlope) → None

Select the *DigitalIn* instrument trigger slope.

Parameters
trigger_slope (DwfTriggerSlope) – The trigger slope to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSlopeGet() → *DwfTriggerSlope*

Get the currently selected *DigitalIn* instrument trigger slope.

Returns
The currently selected *DigitalIn* instrument trigger slope.

Return type
DwfTriggerSlope

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerPositionInfo() → int

Get *DigitalIn* trigger position info.

Returns
The maximum number of samples after the trigger.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerPositionSet(*samples_after_trigger: int*) → None

Set *DigitalIn* instrument desired trigger position.

Parameters
samples_after_trigger (*int*) – The number of samples after the trigger.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerPositionGet() → int

Get *DigitalIn* instrument trigger position.

Returns
The currently configured number of samples after the trigger.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerAutoTimeoutInfo() → Tuple[float, float, int]

Get *DigitalIn* instrument trigger auto-timeout range, in seconds.

Returns
The range and number of steps of the auto-timeout setting.

Return type
Tuple[*float*, *float*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerAutoTimeoutSet(*auto_timeout: float*) → None

Set *DigitalIn* instrument trigger auto-timeout value, in seconds.

Parameters
auto_timeout (*float*) – The auto-timeout value to be configured, in seconds.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerAutoTimeoutGet() → float

Get *DigitalIn* instrument trigger auto-timeout value, in seconds.

Returns
The currently configured trigger auto-timeout value, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerInfo() → Tuple[int, int, int, int]

Get *DigitalIn* detector trigger info.

Return the pins that support the different types of triggering. Each integer is a bitmask, representing pins that can be used as *low level*, *high level*, *rising edge*, and *falling edge* triggers.

Returns
Digital pins that can be used as *low level*, *high level*, *rising edge*, and *falling edge* triggers, respectively.

Return type
Tuple[*int*, *int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerSet(*level_low: int*, *level_high: int*, *edge_rise: int*, *edge_fall: int*) → None

Set *DigitalIn* trigger detector conditions.

The *level_low* and *level_high* settings effectively mask trigger detection events to clock cycles where the selected pins are low or high, respectively.

The *edge_rise* and *edge_fall* indicate channels where the specific type of transition on any of the included channels will lead to a trigger event, ar least if the level conditions specified by the *level_low* and *level_high* settings are satisfied.

Parameters

- **level_low** (*int*) – The channels that are required to be low for a trigger event to occur (bitfield).
- **level_high** (*int*) – The channels that are required to be high for a trigger event to occur (bitfield).
- **edge_rise** (*int*) – The channels where a rising edge will cause a trigger event (bitfield).
- **edge_fall** (*int*) – The channels where a falling edge will cause a trigger event (bitfield).

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerGet() → Tuple[int, int, int, int]

Get the configured *DigitalIn* trigger detector settings.

Returns
The currently configured *level_low*, *level_high*, *edge_rise*, and *edge_fall* settings.

Return type
Tuple[*int*, *int*, *int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerResetSet(*level_low: int*, *level_high: int*, *edge_rise: int*, *edge_fall: int*) → None

Configure *DigitalIn* trigger detector reset condition.

Parameters

- **level_low** (*int*) – The channels that are required to be low for a reset to occur (bitfield).
- **level_high** (*int*) – The channels that are required to be high for a reset to occur (bitfield).

- **edge_rise** (*int*) – The channels where a rising edge will cause a reset (bitfield).
- **edge_fall** (*int*) – The channels where a falling edge will cause a reset (bitfield).

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerCountSet(*count: int*, *restart: int*) → None

Set the *DigitalIn* trigger detector count.

Todo: Figure out what this setting does.

Parameters

- **count** (*int*) – The event count.
- **restart** (*int*) – (to be documented)

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerLengthSet(*min_length: float*, *max_length: float*, *sync_mode: int*) → None

Set the *DigitalIn* trigger detector length.

Todo: Figure out what this setting does.

Parameters

- **min_length** (*float*) – (to be documented)
- **max_length** (*float*) – (to be documented)
- **sync_mode** (*int*) – Synchronization mode:
  - 0 — Normal.
  - 1 — Timing: the *min_length* parameter specifies the bit length and the *max_length* parameter specifies the timing length.

    Used for UART, CAN protocols.
  - 2 — PWM: the *min_length* parameter specifies the sampling time and the *max_length* parameter specifies the timing length.

    Used for 1-wire protocols.

Raises
DwfLibraryError – An error occurred while executing the operation.

triggerMatchSet(*pin: int*, *mask: int*, *value: int*, *bit_stuffing: int*) → None

Set *DigitalIn* trigger detector match.

Todo: Figure out what this setting does.

Configure the deserializer. The bits are left shifted. The mask and value should be specified accordingly, in MSB-first order.

Parameters

- **pin** (*int*) – The pin to be deserialized.
- **mask** (*int*) – The mask pattern.

- **value** (*int*) – The bit pattern.
- **bit_stuffing** (*int*) – The bit stuffing count.

Raises
DwfLibraryError – An error occurred while executing the operation.

counterInfo() → Tuple[int, float]

Get *DigitalIn* counter info.

counterSet(*duration: float*) → None

Set *DigitalIn* counter duration.

counterGet() → float

Get *DigitalIn* counter duration.

counterStatus() → Tuple[float, float, int]

Get *DigitalIn* counter status.

mixedSet(*enable: bool*) → None

Set mixed state.

Warning: This method is obsolete.

Todo: Figure out what this setting does.

Parameters
enable (*bool*) – (to be documented)

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.7 Digital output instrument

The *DigitalOut* instrument provides multiple channels of digital output on devices that support it, such as the Analog Discovery, Analog Discovery 2, Analog Discovery 3, Analog Discovery Pro, and Digital Discovery. It provides the functionality normally associated with a stand-alone digital pattern generator.

Todo: This section is currently incomplete.

It lacks a detailed discussion of how all the settings work.

4.7.1 Using the digital output instrument

To use the *DigitalOut* functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *DigitalOut* functionality can now be accessed via its `digitalOut` attribute, which is an instance of the `DigitalOut` class.

For example:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's DigitalOut instrument.
    digitalOut = device.digitalOut

    # Use the DigitalOut instrument.
    digitalOut.reset()
```

Important: Both the `DigitalIO` and `DigitalOut` instruments provide an API to drive the same digital outputs. The former provides a very simple API that can be used in cases where precise timing or realtime behavior is not relevant, while the latter provides a more powerful, but also more complicated API that provides far greater control over timing.

The rule for which device gets precedence is explained in a topic on the Digilent forum. In summary:

- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 1, the behavior of the channel is determined by the *DigitalIO* instrument.
- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 0, and the *output* is set to 1, the channel is in high-impedance ('Z') state.
- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 0, and the *output* is set to 0, the behavior of the channel is determined by the *DigitalOut* instrument.

Thus, in order to use the *DigitalOut* instrument for a specific channel, the user must ensure that the *DigitalIO* instrument sets both the *outputEnable* and *output* configuration bits to 0. In most circumstances it is not necessary to do this explicitly, since this is the default setting of the *DigitalIO* instrument for each channel.

4.7.2 The *DigitalOut* instrument state machine

The *DigitalOut* instrument is controlled by a state machine. As an output sequence is prepared and executed, the instrument goes through its various states.

The current state of the instrument is returned by the `digitalOut.status()` method, and is of type `DwfState`.

The figure below shows the states used by the *DigitalOut* instrument and the transitions between them:

Fig. 4: States of the *DigitalOut* instrument

The *DigitalOut* states are used as follows:

1. `Ready`

   In this preparatory state, instrument settings can be changed that specify the behavior of the instrument in the coming output sequence. If the auto-configure setting of the device is enabled (the default), setting changes will automatically be transferred to the device. If not, an explicit call to the `digitalOut.configure()` method is needed to transfer updated settings to the device.

Once the instrument is properly configured, an output sequence can be started by calling the *digitalOut.configure()* with the *start* parameter set to True. This will start the first stage of the output sequence by entering the *Armed* state.

2. *Armed*

   In this state the instrument continuously monitors the configured trigger input. As soon as a trigger event is detected, the instrument proceeds to the *Wait* state.

3. *Wait*

   In this state, the digital outputs are driven according to their *Idle* settings. The duration of the wait state is configurable. Once this duration has passed, the instrument proceeds to the *Running* state.

4. *Running*

   In this state, the digital outputs are driven according to their individual configurations. This continues until the run duration has been reached. The channel then proceeds to the **Repeat** state.

5. **Repeat**

   Note: This is not a true state, in that there is no *DwfState* value that represents it. It is included here to explain the control flow of the *DigitalOut* instrument state machine.

   When an output run is finished, the *repeat count* is decremented.

   If, after decrementing, the *repeat count* is unequal to zero, more output must be produced. If the *repeat trigger* setting is True, the instrument proceeds to the *Armed* state; in that case, a trigger is needed to start each of the output runs. If the *repeat trigger* setting is False, the instrument proceeds immediately to the *Wait* state to start another output sequence; a trigger is only required before the very first output run.

   If, after decrementing, the *repeat count* did reach zero, the instrument becomes idle and proceeds to the *Done* state.

6. *Done*

   This state indicates that an output sequence has finished. In this state, the outputs are driven according to their *Idle* setting.

   From this state, it is possible to go back to the *Ready* state by performing any kind of configuration, or to start a new output sequence.

4.7.3 *DigitalOut* instrument API overview

The *DigitalOut* instrument is quite complicated; 53 methods are provided to control its behavior. Below, we categorize these methods and shortly introduce them. Detailed information on all methods can be found in the *DigitalOut* class reference that follows.

Instrument control

Like all instruments supported by the Digilent Waveforms library, the *DigitalOut* instrument provides *reset()*, *configure()*, and *status()* methods.

The *reset()* method resets the instrument.

The *configure()* method is used to explicitly transfer settings to the instrument, and/or to start a configured operation.

The *status()* method returns the current *DwfState* of the *DigitalOut* instrument.

Table 35: Instrument control (4 methods)

control operation	type/unit	methods
reset instrument	*n/a*	*reset()*
configure instrument	*n/a*	*configure()*
request instrument status	*DwfState*	*status()*
request instrument output status	*unknown*	*statusOutput()*

Channel count

This method returns the number of digital output channels.

Table 36: Channel count (1 method)

property	type/unit	method
channel count	int	*count()*

Instrument-level state machine settings

These settings determine the duration of the *Wait* and *Running* states, how many times the Wait/Running cycle should be repeated, and whether a trigger must precede each Wait/Running cycle.

Table 37: State machine settings (13 methods)

setting	type/unit	methods
wait duration	float [s]	*waitInfo()* , *-Set()* , *-Get()*
run duration	float [s]	*runInfo()* , *-Set()* , *-Get()* , *-Status()*
repeat trigger	bool	*repeatTriggerSet()* , *-Get()*
repeat count	int [-]	*repeatInfo()* , *-Set()* , *-Get()* , *-Status()*

Trigger configuration

These settings configure the instrument trigger.

Table 38: Trigger configuration (5 methods)

setting	type/unit	methods
trigger source	*DwfTriggerSource*	*triggerSourceInfo()* , *-Set()* , *-Get()*
trigger slope	*DwfTriggerSlope*	*triggerSlopeSet()* , *-Get()*

Note: The *triggerSourceInfo()* method is obsolete. Use the generic *DwfDevice.triggerInfo()* method instead.

Output settings

These settings determine the output behavior.

Table 39: Output settings (11 methods)

setting	type/unit	methods
enable	bool	*enableSet()* , *-Get()*
output	*DwfDigitalOutOutput*	*outputInfo()* , *-Set()* , *-Get()*
type	*DwfDigitalOutType*	*typeInfo()* , *-Set()* , *-Get()*
idle	*DwfDigitalOutIdle*	*idleInfo()* , *-Set()* , *-Get()*

Output pattern timing definition

These settings determine the per-channel output pattern timing in the *Running* state.

Table 40: Output pattern definition (14 methods)

setting	type/unit	methods
clock info	float [Hz]	*internalClockInfo()*
divider	int [-]	*dividerInfo()* , *-Set()* , *-Get()*
divider init	int [-]	*dividerInitSet()* , *-Get()*
counter	int [-]	*counterInfo()* , *-Set()* , *-Get()*
counter init	int [-]	*counterInitSet()* , *-Get()*
repetition count	int [-]	*repetitionInfo()* , *-Set()* , *-Get()*

Data playback

These methods provide data upload playback by the *DigitalOut* instrument.

Table 41: Data playback (4 methods)

operation	type/unit	methods
data upload	*complicated*	*dataInfo()* , *-Set()*
play data upload	bit string	*playDataSet()*
play rate	float [Hz]	*playRateSet()*

4.7.4 *DigitalOut* reference

class DigitalOut

The *DigitalOut* class provides access to the digital output (pattern generator) instrument of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *digitalOut* attribute for access by the user.

reset() → None

Reset the *DigitalOut* instrument.

Raises

DwfLibraryError – An error occurred while executing the *reset* operation.

configure(*start: bool*) → None

Start or stop the *DigitalOut* instrument.

Parameters
start (*int*) – Whether to start/stop the instrument.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status() → *DwfState*

Return the status of the *DigitalOut* instrument.

This method performs a status request to the *DigitalOut* instrument and receives its response.

Returns
The status of the *DigitalOut* instrument.

Return type
DwfState

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

statusOutput() → Tuple[int, int]

Get status output.

Notice:
This function is not documented in the official documentation, but it is present in the C header file.

Returns
The first entry is labeled 'value', the second is labeled 'enable' in the C header file.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the *statusOutput* operation.

count() → int

Get the *DigitalOut* instrument channel (digital pin) count.

Returns
The number of digital output channels.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

waitInfo() → Tuple[float, float]

Get the *DigitalOut* instrument range for the *Wait* state duration, in seconds.

Returns
A tuple containing the minimal and maximal configurable *Wait* state duration, in seconds.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

waitSet(*wait_duration: float*) → None

Set the *DigitalOut* instrument *Wait* state duration, in seconds.

Parameters
wait_duration (*float*) – Digital-out *Wait* state duration, in seconds.

Raises
DwfLibraryError – An error occurred while executing the operation.

waitGet() → float

Get *DigitalOut* instrument *Wait* state duration, in seconds.

Returns
The *Wait* state duration, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

runInfo() → Tuple[float, float]

Get the *DigitalOut* instrument range for the *Running* state duration, in seconds.

Returns
A tuple containing the minimal and maximal *Running* state duration, in seconds.

Return type
Tuple[*float*, *float*]

Raises
DwfLibraryError – An error occurred while executing the operation.

runSet(*run_duration: float*) → None

Set the *DigitalOut* instrument *Running* state duration, in seconds.

Parameters
run_duration – The *Running* state duration, in seconds.

The value 0 is special; it means *forever*.

Raises
DwfLibraryError – An error occurred while executing the operation.

runGet() → float

Get the *DigitalOut* instrument *Running* state duration, in seconds.

Returns
The *Running* state duration, in seconds.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

runStatus() → int

Get the *DigitalOut* instrument *Running* state duration time left, in clock cycles.

This value is internally expressed as an integer with 48-bit resolution, and is measured in integer clock cycles. The C API returns it as a double-precision floating point number, to avoid using 64-bit integers.

Use the *internalClockInfo()* method to retrieve the clock frequency.

Returns
The number of clock cycles until the nest state transition of the *DigitalOut* instrument's state machine.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatTriggerSet(*repeat_trigger_flag: bool*) → None

Specify if each *DigitalOut* pulse sequence run should wait for its own trigger.

Parameters
repeat_trigger_flag (*bool*) – If True, not only the first, both also every successive run of the pulse output sequence will wait until it receives a trigger.

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatTriggerGet() → bool

Get if each *DigitalOut* pulse sequence run should wait for its own trigger.

Returns
If True, not only the first, both also every successive run of the pulse output sequence will wait until it receives a trigger.

Return type
bool

repeatInfo() → Tuple[int, int]

Get the *DigitalOut* minimal and maximal repeat count for pulse-sequence runs.

Returns
A tuple containing the minimal and maximal repeat count for digital-out pulse-sequence runs.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatSet(*repeat: int*) → None

Set the *DigitalOut* repeat count for pulse-sequence runs.

Parameters
repeat (*int*) – Repeat count. The value 0 is special; it means *forever*.

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatGet() → int

Set the *DigitalOut* repeat count for pulse-sequence runs.

Returns
Repeat count. The value 0 is special; it means *forever*.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

repeatStatus() → int

Get the *DigitalOut* count of repeats remaining for the currently active output sequence.

This number counts down as a digital output sequence is active.

Returns
The repeat count status.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

`triggerSourceInfo`() → List[*DwfTriggerSource*]

Get the valid *DigitalOut* trigger sources.

> **Warning: This method is obsolete.**
>
> Use the generic DeviceControl.triggerInfo() method instead.

Returns
A list of valid the trigger sources.

Return type
List[DwfTriggerSource]

Raises
DwfLibraryError – An error occurred while executing the operation.

`triggerSourceSet`(*trigger_source:* DwfTriggerSource) → None

Set the *DigitalOut* trigger source.

Parameters
trigger_source (DwfTriggerSource) – The trigger source to be configured.

Raises
DwfLibraryError – An error occurred while executing the operation.

`triggerSourceGet`() → *DwfTriggerSource*

Get the currently selected instrument trigger source.

Returns
The currently selected instrument trigger source.

Return type
DwfTriggerSource

Raises
DwfLibraryError – An error occurred while executing the operation.

`triggerSlopeSet`(*trigger_slope:* DwfTriggerSlope) → None

Select the *DigitalOut* instrument trigger slope.

Parameters
trigger_slope (DwfTriggerSlope) – The trigger slope to be selected.

Raises
DwfLibraryError – An error occurred while executing the operation.

`triggerSlopeGet`() → *DwfTriggerSlope*

Get the currently configured *DigitalOut* instrument trigger slope.

Returns
The currently selected *DigitalOut* instrument trigger slope.

Return type
DwfTriggerSlope

Raises
DwfLibraryError – An error occurred while executing the operation.

`enableSet`(*channel_index: int*, *enable_flag: bool*) → None

Enable or disable a *DigitalOut* channel (pin).

Parameters

- **channel_index** (*int*) – The digital pin to enable or disable.
- **enable_flag** (*bool*) – Whether to enable or disable the digital output.

Raises
DwfLibraryError – An error occurred while executing the operation.

enableGet(*channel_index: int*) → bool

Check if a specific *DigitalOut* channel (pin) is enabled for output.

Parameters
channel_index (*int*) – The digital pin.

Returns
Whether the digital pin is enabled as an output.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

outputInfo(*channel_index: int*) → List[*DwfDigitalOutOutput*]

Get valid *DigitalOut* output choices (e.g. Push/Pull, tristate).

Returns
A list of valid output settings.

Return type
List[DwfDigitalOutOutput]

Raises
DwfLibraryError – An error occurred while executing the operation.

outputSet(*channel_index: int*, *output_value:* DwfDigitalOutOutput) → None

Set *DigitalOut* output choice (e.g. Push/Pull, tristate).

Parameters

- **channel_index** (*int*) – The digital pin.
- **output_value** (DwfDigitalOutOutput) – The digital output setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

outputGet(*channel_index: int*) → *DwfDigitalOutOutput*

Get currently configured *DigitalOut* output (e.g. Push/Pull, tristate).

Parameters
channel_index (*int*) – The digital pin.

Returns
The digital output setting.

Return type
DwfDigitalOutOutput

Raises
DwfLibraryError – An error occurred while executing the operation.

typeInfo(*channel_index: int*) → List[*DwfDigitalOutType*]

Get a list of valid *DigitalOut* output channel types.

Returns
A list of valid digital output channel types.

Return type
List[DwfDigitalOutType]

Raises
DwfLibraryError – An error occurred while executing the operation.

typeSet(*channel_index: int*, *output_type:* DwfDigitalOutType) → None

Select the *DigitalOut* output channel type.

Parameters

- **channel_index** (*int*) – The digital pin.
- **output_type** (DwfDigitalOutType) – The digital output channel type.

Raises
DwfLibraryError – An error occurred while executing the operation.

typeGet(*channel_index: int*) → *DwfDigitalOutType*

Get the currently selected *DigitalOut* output channel type.

Parameters
channel_index (*int*) – The digital pin.

Returns
The digital output channel type.

Return type
DwfDigitalOutType

Raises
DwfLibraryError – An error occurred while executing the operation.

idleInfo(*channel_index: int*) → List[*DwfDigitalOutIdle*]

Get valid *DigitalOut* idle output values.

Returns
A list of valid idle output values.

Return type
List[DwfDigitalOutIdle]

Raises
DwfLibraryError – An error occurred while executing the operation.

idleSet(*channel_index: int*, *idle_mode:* DwfDigitalOutIdle) → None

Set the *DigitalOut* idle output value.

Parameters

- **channel_index** (*int*) – The digital pin.
- **idle_mode** (DwfDigitalOutIdle) – The idle output value.

Raises
DwfLibraryError – An error occurred while executing the operation.

idleGet(*channel_index: int*) → *DwfDigitalOutIdle*

Get the currently configured idle output value.

Parameters
channel_index (*int*) – The digital pin.

Returns
The currently configured idle output value.

Return type
DwfDigitalOutIdle

Raises
DwfLibraryError – An error occurred while executing the operation.

internalClockInfo() → float

Get the *DigitalOut* instrument clock frequency.

Returns
The digital-out clock frequency, in Hz.

Return type
float

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerInfo(*channel_index: int*) → Tuple[int, int]

Get the *DigitalOut* divider value range.

Returns
The range of valid divider settings.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerSet(*channel_index: int*, *divider: int*) → None

Set the *DigitalOut* divider value.

Parameters

- **channel_index** (*int*) – The digital pin.
- **divider** (*int*) – The divider setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerGet(*channel_index: int*) → int

Get the currently configured *DigitalOut* divider value.

Parameters
channel_index (*int*) – The digital pin.

Returns
The divider setting.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerInitSet(*channel_index: int*, *divider_init_value: int*) → None

Set the *DigitalOut* initial divider value.

Parameters

- **channel_index** (*int*) – The digital pin.
- **divider_init_value** (*int*) – The initial divider counter value.

Raises
DwfLibraryError – An error occurred while executing the operation.

dividerInitGet(*channel_index: int*) → int

Get the currently configured *DigitalOut* initial divider value.

Parameters
channel_index (*int*) – The digital pin.

Returns
The divider init setting.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

counterInfo(*channel_index: int*) → Tuple[int, int]

Get the *DigitalOut* counter value range.

Parameters
channel_index (*int*) – The digital pin.

Returns
The range of valid counter settings.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

counterSet(*channel_index: int*, *low_count: int*, *high_count: int*) → None

Set the *DigitalOut* counter durations for both the low and high signal output levels.

Parameters

- **channel_index** (*int*) – The digital pin.
- **low_count** (*int*) – The number of cycles the signal should be Low.
- **high_count** (*int*) – The number of cycles the signal should be High.

Raises
DwfLibraryError – An error occurred while executing the operation.

counterGet(*channel_index: int*) → Tuple[int, int]

Get the *DigitalOut* counter durations for both the low and high signal output levels.

Parameters
channel_index (*int*) – The digital pin.

Returns
The number of cycles the signal should be Low, High.

Return type
Tuple[*int*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

counterInitSet(*channel_index: int*, *high_flag: bool*, *counter_init_value: int*) → None

Set the *DigitalOut* initial signal value and initial counter value.

Parameters

- **channel_index** (*int*) – The digital pin.
- **high** (*bool*) – Whether to start High (True) or Low (False).
- **counter_init** (*int*) – The initial counter setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

counterInitGet(*channel_index: int*) → Tuple[bool, int]

Get the *DigitalOut* initial signal value and initial counter value.

Parameters
channel_index (*int*) – The digital pin.

Returns
Whether to start High (True) or Low (False), and the initial counter setting.

Return type
Tuple [*bool*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

repetitionInfo(*channel_index: int*) → int

Get maximum repetition count value.

The repetition count specifies how many times the counter should be reloaded. For pulse signals set twice the desired value since each pulse is generated by two counter loads, low and high. It is available with ADP3X50 and newer devices.

Parameters
channel_index (*int*) – The digital pin.

Returns
The maximum repetition value that can be configured.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

repetitionSet(*channel_index: int*, *repeat: int*) → None

Set repetition count value.

The repetition count specifies how many times the counter should be reloaded. For pulse signals set twice the desired value since each pulse is generated by two counter loads, low and high. It is available with ADP3X50 and newer devices.

Parameters
channel_index (*int*) – The digital pin.

Returns
The maximum repetition value that can be configured.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

repetitionGet(*channel_index: int*) → int

Get repetition count value.

The repetition count specifies how many times the counter should be reloaded. For pulse signals set twice the desired value since each pulse is generated by two counter loads, low and high. It is available with ADP3X50 and newer devices.

dataInfo(*channel_index: int*) → int

Return the maximum buffer size for the specified *DigitalOut* channel, i.e., the number of custom data bits.

Parameters
channel_index (*int*) – the channel for which to obtain the data bits count.

Returns
The number of custom data bits that can be specified for the channel.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

dataSet(*channel_index: int*, *bits: str*, *tristate: bool = False*) → None

Set the *DigitalOut* arbitrary output data.

This function also sets the counter initial, low and high value, according to the number of bits. The data-bits are sent out in LSB-first order.

Parameters

- **channel_index** (*int*) – the channel for which to set the output data.
- **bits** (*str*) – The bits, as a string.
- **tristate** (*bool*) – Whether to interpret the string as a tristate signal.

Raises
DwfLibraryError – An error occurred while executing the operation.

playDataSet(*bits: str*, *bits_per_sample: int*, *count_of_samples: int*) → None

Set the *DigitalOut* playback data.

The output can be PushPull, OpenDrain, or OpenSource. Tristate data is not supported.

Note: The DWF documentation explicitly states that this function is supported by the Digital Discovery. (So, by implication, it's probably not supported on anything else.)

Parameters

- **bits** (*str*) – string of '0' and '1' characters. Its length should be (bits_per_sample * count_of_samples).
- **bits_per_sample** (*int*) – Bits per sample, should be 1, 2, 4, 8, or 16.
- **count_of_samples** (*int*) – Number of samples.

Raises
DwfLibraryError – An error occurred while executing the operation.

playUpdateSet(*bits: str*, *index_of_sample: int*, *count_of_samples: int*) → None

Set the *DigitalOut* playback data.

Todo: This function is not sufficiently documented at this time.

---

playRateSet(*playback_rate: float*) → None

Set the *DigitalOut* playback rate, in Hz.

Note: The DWF documentation explicitly states that this function is supported by the Digital Discovery. (So, by implication, it's probably not supported on anything else.)

Parameters
playback_rate (*float*) – The playback rate, in Hz.

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.8 Digital I/O

The *DigitalIO* functionality provides low-speed monitoring and control of the same digital I/O pins that can also be controlled by the more powerful *DigitalIn* and *DigitalOut* instruments. The API provided here is much simpler to use, but it can only accommodate use-cases that do not require triggering, precise timing, or very fast operation.

4.8.1 Using the digital I/O functionality

To use the *DigitalIO* functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's *DigitalIO* functionality can now be accessed via its *digitalIO* attribute, which is an instance of the *DigitalIO* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:

    # Get a reference to the device's DigitalIO functionality.
    digitalIO = device.digitalIO

    # Use the DigitalIO functionality.
    digitalIO.reset()
```

Important: Both the *DigitalIO* and *DigitalOut* instruments provide an API to drive the same digital outputs. The former provides a very simple API that can be used in cases where precise timing or realtime behavior is not relevant, while the latter provides a more powerful, but also more complicated API that provides far greater control over timing.

The rule for which device gets precedence is explained in a topic on the Digilent forum. In summary:

- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 1, the behavior of the channel is determined by the *DigitalIO* instrument.
- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 0, and the *output* is set to 1, the channel is in high-impedance ('Z') state.
- For DIO channels where the *DigitalIO* instrument sets *outputEnable* to 0, and the *output* is set to 0, the behavior of the channel is determined by the *DigitalOut* instrument.

Thus, in order to use the *DigitalOut* instrument for a specific channel, the user must ensure that the *DigitalIO* instrument sets both the *outputEnable* and *output* configuration bits to 0. In most circumstances it is not necessary to do this explicitly, since this is the default setting of the *DigitalIO* instrument for each channel.

4.8.2 *DigitalIO* reference

class DigitalIO

The *DigitalIO* class provides access to the static digital I/O functionality of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently assigned to its public *digitalIO* attribute for access by the user.

The class implements 3 generic methods: *reset()*, *configure()*, and *status()*, and 8 methods that come in both 32- and 64-bits variants, where 32 and 64 refer to the maximum number of digital pins that the methods can handle.

reset() → None

Reset all *DigitalIO* settings to default values.

It sets the digital pins to tri-state (high impedance, not enabled) and output value to zero.

If autoconfiguration is enabled, the values are immediately applied.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

configure() → None

Configure the *DigitalIO* functionality.

This method transfers the settings to the Digilent Waveforms device. It is not needed if autoconfiguration is enabled.

Raises
DwfLibraryError – An error occurred while executing the *configure* operation.

status() → None

Read the *DigitalIO* status and input values from the device to the PC.

The status inquiry methods that follow will return the information that was read from the device when this method was last called.

Note that the *DigitalIO* functionality is not managed by a state machine, so this method does not return a value.

Raises
DwfLibraryError – An error occurred while executing the *status* operation.

outputEnableInfo() → int

Get the digital pins that can be enabled for output as a bitmask.

The *output enable state* of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 32-bits version of this method. For the 64-bits version, see *outputEnableInfo64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputEnableSet(*output_enable: int*) → None

Set the digital pins that are enabled for output as a bitmask.

The *output enable* state of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 32-bits version of this method. For the 64-bits version, see *outputEnableSet64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Parameters
output_enable (*int*) – A bitmask of pins that will be used as outputs.

Raises
DwfLibraryError – An error occurred while executing the operation.

outputEnableGet() → int

Get the digital pins that are enabled for output as a bitmask.

The *output enable* state of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 32-bits version of this method. For the 64-bits version, see *outputEnableGet64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently configured as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputInfo() → int

Get the digital pins that can be used as outputs, i.e., driven high or low, as a bitmask.

This is the 32-bits version of this method. For the 64-bits version, see *outputInfo64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputSet(*output: int*) → None

Set the digital pins that are currently driven high as a bitmask.

This is the 32-bits version of this method. For the 64-bits version, see *outputSet64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Parameters
output (*int*) – A bitmask of pins that will be driven high.

Raises
DwfLibraryError – An error occurred while executing the operation.

outputGet() → int

Get the digital pins that are currently driven high as a bitmask.

This is the 32-bits version of this method. For the 64-bits version, see *outputGet64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently set to high.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

pullInfo() → Tuple[int, int]

Get pull info.

Raises
DwfLibraryError – An error occurred while executing the operation.

pullSet(*pull_up_mask: int*, *pull_down_mask: int*) → None

Set pull-up and pull-down channels.

Raises
DwfLibraryError – An error occurred while executing the operation.

pullGet() → Tuple[int, int]

Get pull up/down configuration.

Raises
DwfLibraryError – An error occurred while executing the operation.

driveInfo() → Tuple[float, float, int, int]

Get drive info.

Raises
DwfLibraryError – An error occurred while executing the operation.

driveSet(*channel: int*, *amp: float*, *slew: int*) → None

Set channel drive.

Raises
DwfLibraryError – An error occurred while executing the operation.

driveGet(*channel: int*) → Tuple[float, int]

Get channel drive settings.

Raises
DwfLibraryError – An error occurred while executing the operation.

inputInfo() → int

Return the digital pins that can be used for input on the device as a bitmask.

This is the 32-bits version of this method. For the 64-bits version, see *inputInfo64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as inputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

inputStatus() → int

Return the current state of the digital input pins on the device as a bitmask.

Before calling this method, call the *status()* method to read the current digital input status from the device.

This is the 32-bits version of this method. For the 64-bits version, see *inputStatus64()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently read as high.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputEnableInfo64() → int

Get the digital pins that can be enabled for output as a bitmask.

The *output enable state* of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 64-bits version of this method. For the 32-bits version, see *outputEnableInfo()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputEnableSet64(*output_enable: int*) → None

Set the digital pins that are enabled for output as a bitmask.

The *output enable* state of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 64-bits version of this method. For the 32-bits version, see *outputEnableSet()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Parameters
output_enable (*int*) – A bitmask of pins that will be used as outputs.

Raises
DwfLibraryError – An error occurred while executing the operation.

outputEnableGet64() → int

Get the digital pins that are enabled for output as a bitmask.

The *output enable* state of a pin determines if it is driven as an output. If not, it is in high impedance (also known as high Z) mode.

Only digital pins that are used as outputs should be enabled; digital pins that are used as inputs should remain disabled (the default state after reset).

This is the 64-bits version of this method. For the 32-bits version, see *outputEnableGet()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently configured as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputInfo64() → int

Get the digital pins that can be used as outputs, i.e., driven high or low, as a bitmask.

This is the 64-bits version of this method. For the 32-bits version, see *outputInfo()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as outputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

outputSet64(*output: int*) → None

Set the digital pins that are currently driven high as a bitmask.

This is the 64-bits version of this method. For the 32-bits version, see *outputSet()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Parameters
output (*int*) – A bitmask of pins that will be driven high.

Raises
DwfLibraryError – An error occurred while executing the operation.

outputGet64() → int

Get the digital pins that are currently driven high as a bitmask.

This is the 64-bits version of this method. For the 32-bits version, see *outputGet()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently set to high.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

inputInfo64() → int

Return the digital pins that can be used for input on the device as a bitmask.

This is the 64-bits version of this method. For the 32-bits version, see *inputInfo()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that can be used as inputs.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

inputStatus64() → int

Return the current state of the digital input pins on the device as a bitmask.

Before calling this method, call the *status()* method to read the current digital input status from the device.

This is the 64-bits version of this method. For the 32-bits version, see *inputStatus()*. The 32 and 64 bits refer here to the maximum number of digital pins that the methods can handle.

Returns
A bitmask of pins that are currently read as high.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.9 UART protocol

The UART protocol support allows a Digilent Waveforms device to be used as a simple Universal Asynchronous Receiver/Transmitter (UART).

Todo: This section is currently incomplete.

Specifically, the meaning of the parity error indication as returned by the `rx()` method is unclear. It needs to be investigated and documented.

4.9.1 Using the UART protocol functionality

To use the UART protocol functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's UART protocol functionality can now be accessed via its *protocol.uart* attribute, which is an instance of the *ProtocolUART* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    uart = device.protocol.uart
    uart.reset()
```

The UART protocol as implemented supports a single digital pin to act as a transmitter (TX), and a single digital pin to act as a receiver (RX). Transmission and reception are relative to the viewpoint of the Digilent Waveforms device; so 'transmission' means that the Digilent Waveforms device sends outgoing data, and 'reception' means that the Digilent Waveforms device receives incoming data.

The UART protocol only supports the two basic serial TX and RX signals. Other signals commonly encountered on serial ports (e.g., hardware handshaking using RTS/CTS) are not supported.

Note that while the UART API provides several methods to configure the serial communication (most notably, the baudrate, number of data-bits, parity, and number of stop-bits), there is no way to read back the currently active communication parameter values.

4.9.2 *ProtocolUART* reference

class ProtocolUART

The *ProtocolUART* class provides access to the UART protocol functionality of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently accessible via its *protocol.uart* attribute.

reset() → None

Reset the UART protocol functionality.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

rateSet(*baudrate: float*) → None

Set the UART baudrate.

Parameters
baudrate (*float*) – The baud-rate used by the receiver and transmitter.

Commonly encountered values are 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200, but other values are valid as well.

Raises
DwfLibraryError – An error occurred while executing the operation.

bitsSet(*databits: int*) → None

Set the number of UART data bits.

Parameters
databits (*int*) – The number of data-bits used by the receiver and transmitter.

The most common choice is 8, but other values are possible.

Raises
DwfLibraryError – An error occurred while executing the operation.

paritySet(*parity: int*) → None

Set the UART character parity.

Parameters
parity (*int*) – The parity used by the receiver and transmitter:

- 0 — no parity
- 1 — odd parity
- 2 — even parity

The most common choice is *no parity* (i.e., 0).

Raises
DwfLibraryError – An error occurred while executing the operation.

polaritySet(*polarity: int*) → None

Set the UART signal polarity.

Parameters
polarity (*int*) – The polarity used by the receiver and transmitter:

- 0 — normal
- 1 — inverted

The most common choice and default polarity is *normal* (i.e., 0).

Raises
DwfLibraryError – An error occurred while executing the operation.

stopSet(*stopbits: float*) → None

Set the number of UART stop bits.

Parameters
stopbits (*float*) – The number of stop-bits used by the receiver and transmitter.

The most common choice is 1 stop-bit. Other values that are (rarely) encountered are 1.5 and 2 stop-bits.

Note that the actual number of stop-bits is the number specified here, rounded up to the next highest integer.

The parameter is declared as a *float* in anticipation of future support for 1.5 stop-bits.

Raises
DwfLibraryError – An error occurred while executing the operation.

txSet(*channel_index: int*) → None

Set the digital channel (pin) where the UART's outgoing (TX) signal will be transmitted.

Parameters
channel_index (*int*) – The digital channel (pin) on which to transmit data.

Raises
DwfLibraryError – An error occurred while executing the operation.

rxSet(*channel_index: int*) → None

Set the digital channel (pin) where the UART's incoming (RX) signal is received.

Parameters
channel_index (*int*) – The digital channel (pin) on which to receive data.

Raises
DwfLibraryError – An error occurred while executing the operation.

tx(*tx_data: bytes*) → None

Transmit data according to the currently active UART settings.

Parameters
tx_data (*bytes*) – The data to be transmitted.

Raises
DwfLibraryError – An error occurred while executing the operation.

rx(*rx_max: int*) → Tuple[bytes, int]

Receive UART data or prepare for reception.

Important: This method must be called with value 0 prior to receiving data, to initialize the receiver.

Parameters
rx_max (*int*) – If 0, initialize the receiver.

Otherwise, receive the specified number of characters.

Returns
Bytes received and parity error indication.

Return type
Tuple[*bytes*, *int*]

Todo: The meaning of the parity error indication is currently unclear. This needs to be investigated.

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.10 SPI protocol

The SPI protocol support allows a Digilent Waveforms device to be used as a simple SPI bus master.

Todo: This section is currently incomplete.

It does not properly explain the meaning of some of the settings that influence the behavior of the SPI functionality, and the difference between the several write/read methods.

4.10.1 Using the SPI protocol functionality

To use the SPI protocol functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's SPI protocol functionality can now be accessed via its *protocol.spi* attribute, which is an instance of the *ProtocolSPI* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    spi = device.protocol.spi
    spi.reset()
```

4.10.2 *ProtocolSPI* reference

class ProtocolSPI

The *ProtocolSPI* class provides access to the SPI protocol functionality of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently accessible via its *protocol.spi* attribute.

reset() → None

Reset the SPI protocol support.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

frequencySet(*frequency: float*) → None

Set the SPI frequency, in Hz.

Parameters
frequency (*float*) – SPI frequency, in Hz.

Raises
DwfLibraryError – An error occurred while executing the operation.

clockSet(*channel_index: int*) → None

Set the digital channel (pin) for the SPI clock signal.

Parameters
channel_index (*int*) – The digital channel (pin) where the SPI clock signal will be generated.

Raises
DwfLibraryError – An error occurred while executing the operation.

dataSet(*spi_data_bit: int*, *channel_index: int*) → None

Set the digital channel (pin) for an SPI data bit.

Parameters

- **spi_data_bit** (*int*) – The data bit to configure:
  - 0 — DQ0 / MOSI / SISO
  - 1 — DQ1 / MISO
  - 2 — DQ2
  - 3 — DQ3
- **channel_index** (*int*) – The digital channel (pin) for this data bit.

Raises
DwfLibraryError – An error occurred while executing the operation.

idleSet(*spi_data_bit: int*, *idle_mode:* DwfDigitalOutIdle) → None

Set the idle behavior for an SPI data bit.

Parameters

- **spi_data_bit** (*int*) – The data bit to configure:
  - 0 — DQ0 / MOSI / SISO
  - 1 — DQ1 / MISO
  - 2 — DQ2

- 3 — DQ3

- **idle_mode** (DwfDigitalOutIdle) – The idle behavior of this bit.

Raises
DwfLibraryError – An error occurred while executing the operation.

modeSet(*spi_mode: int*) → None

Set the SPI mode.

Parameters
spi_mode (*int*) – The values for CPOL (polarity) and CPHA (phase) to use with the attached slave device:

- 0 — CPOL = 0, CPHA = 0
- 1 — CPOL = 0, CPHA = 1
- 2 — CPOL = 1, CPHA = 0
- 3 — CPOL = 1, CPHA = 1

Refer to the slave device's datasheet to select the correct value.

Raises
DwfLibraryError – An error occurred while executing the operation.

orderSet(*bit_order: int*) → None

Set the SPI data bit order.

Parameters
bit_order (*int*) – Select the bit order of each word sent out:

- 1 — MSB first, LSB last
- 0 — LSB first, MSB last

Raises
DwfLibraryError – An error occurred while executing the operation.

delaySet(*start: int*, *cmd: int*, *word: int*, *stop: int*) → None

Set the SPI delays.

Raises
DwfLibraryError – An error occurred while executing the operation.

selectSet(*channel: int*, *level: int*) → None

Set SPI device select channel and level.

Raises
DwfLibraryError – An error occurred while executing the operation.

select(*channel_index: int*, *level: int*) → None

Set the chip select (CS) status.

Parameters

- **channel_index** (*int*) – The digital channel (pin) for the Chip Select signal.
- **level** (*int*) – The Chip Select level to configure.
  - 0 — low
  - 1 — high
  - -1 — Z (high impedance)

  The CS (chip select) is an active-low signal, from the SPI bus master to a specific SPI slave device. Before starting a bus request, the master should set CS to 0 for the chip it wants to talk to.

Each slave on an SPI bus has its own CS line. At most one of them should be selected at any time.

Raises
DwfLibraryError – An error occurred while executing the operation.

writeRead(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Write and read multiple SPI data-words, with up to 8 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…8).
- **tx** (*List[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

writeRead16(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Write and read multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **tx** (*list[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

writeRead32(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Write and read multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO

- 2 — dual
- 4 — quad

- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **tx** (*List[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

read(*transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Read multiple SPI data-words, with up to 8 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…8).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

readOne(*transfer_type: int*, *bits_per_word: int*) → int

Read a single SPI data-word, with up to 32 bits.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits of the data-word (1…32).

Returns
The data-word received.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the read operation.

read16(*transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Read multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

read32(*transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Read multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

write(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Write multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).

- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

writeOne(*transfer_type: int*, *bits_per_word: int*, *tx: int*) → None

Write a single SPI data-word, with up to 32 bits.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits of the data-word (1…32).
- **tx** (*int*) – The data-word to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

write16(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Write multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

write32(*transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Write multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

cmdWriteRead(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Send command and write and read multiple SPI data-words, with up to 8 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…8).
- **tx** (*List[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

cmdWriteRead16(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Send command and write and read multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **tx** (*list[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

cmdWriteRead32(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → List[int]

Send command and write and read multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **tx** (*List[int]*) – The data-words to write.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the write/read operation.

cmdRead(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Send command and read multiple SPI data-words, with up to 8 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…8).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

cmReadOne(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*) → int

Send command and read a single SPI data-word, with up to 32 bits.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits of the data-word (1…32).

Returns
The data-word received.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the read operation.

cmdRead16(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Send command and read multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

cmdRead32(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *number_of_words: int*) → List[int]

Send command and read multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **number_of_words** (*int*) – The number of data-words to read.

Returns
The data-words received.

Return type
List[*int*]

Raises
DwfLibraryError – An error occurred while executing the read operation.

cmdWrite(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Send command and write multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

cmdWriteOne(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: int*) → None

Send command and write a single SPI data-word, with up to 32 bits.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –

- 0 — SISO
- 1 — MOSI/MISO
- 2 — dual
- 4 — quad

- **bits_per_word** (*int*) – The number of bits of the data-word (1…32).
- **tx** (*int*) – The data-word to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

cmdWrite16(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Send command and write multiple SPI data-words, with up to 16 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…16).
- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

cmdWrite32(*command_bits: int*, *command_value: int*, *dummy_bits: int*, *transfer_type: int*, *bits_per_word: int*, *tx: List[int]*) → None

Send command and write multiple SPI data-words, with up to 32 bits per data-word.

Parameters

- **command_bits** (*int*) – The number of command bits.
- **command_value** (*int*) – The command value.
- **dummy_bits** (*int*) – The number of dummy bits before the data transfer.
- **transfer_type** (*int*) –
  - 0 — SISO
  - 1 — MOSI/MISO
  - 2 — dual
  - 4 — quad
- **bits_per_word** (*int*) – The number of bits per data-word (1…32).
- **tx** (*List[int]*) – The data-words to write.

Raises
DwfLibraryError – An error occurred while executing the write operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.11 I²C protocol

The I²C protocol support allows a Digilent Waveforms device to be used as a simple I²C bus master.

Todo: This section is currently incomplete.

It does not properly explain some of the settings that influence the behavior of the I²C functionality, and the difference between the several write/read methods.

4.11.1 Using the I²C protocol functionality

To use the I²C protocol functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's I²C protocol functionality can now be accessed via its *protocol.i2c* attribute, which is an instance of the *ProtocolI2C* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    i2c = device.protocol.i2c
    i2c.reset()
```

4.11.2 *ProtocolI2C* reference

class ProtocolI2C

The *ProtocolI2C* class provides access to the I²C protocol functionality of a *DwfDevice*.

Attention: Users of *pydwf* should not create instances of this class directly.

It is instantiated during initialization of a *DwfDevice* and subsequently accessible via its *protocol.i2c* attribute.

reset() → None

Reset the I²C protocol instrument.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

clear() → bool

Clear the I²C bus.

Detect and try to solve an I²C bus lockup condition.

Todo: The precise behavior of this method needs to be understood and documented.

Returns
True if the bus is clear after the operation, False otherwise.

Return type
bool

Raises
DwfLibraryError – An error occurred while executing the operation.

stretchSet(*stretch_enable: int*) → None

Set I²C stretch behavior.

Todo: The precise behavior of this method needs to be understood and documented.

Parameters
stretch_enable (*bool*) – True to enable stretch, False to disable.

Raises
DwfLibraryError – An error occurred while executing the operation.

rateSet(*data_rate: float*) → None

Set the I²C data rate, in Hz.

Parameters
data_rate (*float*) – I²C data rate. Often-encountered rates are 100 kHz and 400 kHz, but many modern I²C devices support higher data rates. Check the datasheet of your device.

The default value is 100 kHz.

Raises
DwfLibraryError – An error occurred while executing the operation.

timeoutSet(*timeout: float*) → None

Set the I²C timeout, in seconds.

Parameters
timeout (*float*) – I²C timeout. The default value is 1 second.

Raises
DwfLibraryError – An error occurred while executing the operation.

readNakSet(*nak_last_read_byte: int*) → None

Set read NAK state.

Todo: The precise behavior of this method needs to be understood and documented.

Parameters
nak_last_read_byte (*int*) – (undocumented)

Raises
DwfLibraryError – An error occurred while executing the operation.

sclSet(*channel_index: int*) → None

Set the digital channel (pin) where the I²C clock (SCL) signal is transmitted.

Parameters
channel_index (*int*) – The digital channel (pin) on which to generate the SCL clock.

Raises
DwfLibraryError – An error occurred while executing the operation.

sdaSet(*channel_index: int*) → None

Set the digital channel (pin) where the I²C data (SDA) signal is transmitted/received.

Parameters
channel_index (*int*) – The digital channel (pin) on which to send/receive SDA data.

Raises
DwfLibraryError – An error occurred while executing the operation.

writeRead(*address: int*, *tx: List[int]*, *number_of_rx_bytes: int*) → Tuple[int, List[int]]

Perform a combined I²C write/read operation.

Parameters

- **address** (*int*) – The I²C address of the target device.
- **tx** (*List[int]*) – The octets to send.
- **number_of_rx_bytes** (*int*) – The number of octets to receive.

Returns
The first element is the NAK indication; the second element is a list of octet values received.

Return type
Tuple[*int*, *List*[*int*]]

Raises
DwfLibraryError – An error occurred while executing the operation.

read(*address: int*, *number_of_words: int*) → Tuple[int, List[int]]

Perform an I²C read operation.

Parameters

- **address** (*int*) – The I²C address of the target device.
- **number_of_words** (*int*) – The number of octets to receive.

Returns
The first element is the NAK indication; the second element is a list of octet values received.

Return type
Tuple[*int*, *List*[*int*]]

Raises
DwfLibraryError – An error occurred while executing the operation.

write(*address: int*, *tx: List[int]*) → int

Perform an I²C write operation.

Parameters

- **address** (*int*) – The I²C address of the target device.
- **tx** (*List[int]*) – The octets to send.

Returns
The NAK indication.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

writeOne(*address: int*, *tx: int*) → int

Perform an I²C write operation of a single octet.

Parameters

- **address** (*int*) – The I²C address of the target device.
- **tx** (*int*) – The single octet to send.

Returns
The NAK indication.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

spyStart() → None

Start I²C spy.

Raises
DwfLibraryError – An error occurred while executing the operation.

spyStatus(*max_data_size: int*) → Tuple[int, int, List[int], int]

Get I²C spy status.

Returns
A tuple (start, stop, data-values, nak-indicator).

Return type
Tuple[*int*, *int*, *List*[*int*], *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.12 CAN protocol

The CAN protocol support allows a Digilent Waveforms device to be used as a simple CAN bus device.

Todo: This section is currently incomplete.

A bit of background on CAN would be helpful.

This section does not yet properly explain the *polarity* setting.

It also does not yet explain the *vID*, *extended*, and *remote* parameters / return values used in the *rx()* and *tx()* methods.

4.12.1 Using the CAN protocol functionality

To use the CAN protocol functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's CAN protocol functionality can now be accessed via its `protocol.can` attribute, which is an instance of the `ProtocolCAN` class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    can = device.protocol.can
    can.reset()
```

4.12.2 *ProtocolCAN* reference

class ProtocolCAN

The *ProtocolCAN* class provides access to the CAN bus protocol functionality of a `DwfDevice`.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently accessible via its `protocol.can` attribute.

reset() → None

Reset the CAN bus protocol functionality.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

rateSet(*data_rate: float*) → None

Set the CAN bus data rate, in Hz.

Parameters
data_rate (*float*) – The data-rate used by the receiver and transmitter.

Raises
DwfLibraryError – An error occurred while executing the operation.

polaritySet(*polarity: bool*) → None

Set the CAN bus polarity.

Parameters
polarity (*bool*) – If True, set polarity to *high*.

Raises
DwfLibraryError – An error occurred while executing the operation.

txSet(*channel_index: int*) → None

Set the digital channel (pin) where the outgoing (TX) signal will be transmitted.

Parameters
channel_index (*int*) – The digital channel (pin) on which to transmit data.

Raises
DwfLibraryError – An error occurred while executing the operation.

rxSet(*channel_index: int*) → None

Set the digital channel (pin) where the incoming (RX) signal is received.

Parameters
channel_index (*int*) – The digital channel (pin) on which to receive data.

Raises
DwfLibraryError – An error occurred while executing the operation.

tx(*v_id: int*, *extended: bool*, *remote: bool*, *data: bytes*) → None

Transmit outgoing CAN bus data.

Parameters

- **v_id** (*int*) – (to be documented).
- **extended** (*bool*) – (to be documented).
- **remote** (*bool*) – (to be documented).
- **data** (*bytes*) – The data to be transmitted. Should be at most 8 bytes.

Raises

- *PyDwfError* – A request to transmit more than 8 bytes was made.
- *DwfLibraryError* – An error occurred while executing the operation.

rx(*size: int = 8*) → Tuple[int, bool, bool, bytes, int]

Receive incoming CAN bus data.

Returns
A tuple (vID, extended, remote, data, status)

Return type
Tuple[*int*, *bool*, *bool*, *bytes*, *int*]

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

4.13 SWD protocol

The SWD protocol support allows a Digilent Waveforms device to be used as a simple SWD (Serial Wire Debug) device.

Todo: This section is currently incomplete.

A bit of background on SWD would be helpful.

4.13.1 Using the SWD protocol functionality

To use the SWD protocol functionality you first need to initialize a *DwfLibrary* instance. Next, you open a specific device. The device's SWD protocol functionality can now be accessed via its *protocol.swd* attribute, which is an instance of the *ProtocolSWD* class:

```
from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    swd = device.protocol.swd
    swd.reset()
```

4.13.2 *ProtocolSWD* reference

class ProtocolSWD

The *ProtocolSWD* class provides access to the SWD bus protocol functionality of a *DwfDevice*.

> **Attention:** Users of *pydwf* should not create instances of this class directly.
>
> It is instantiated during initialization of a *DwfDevice* and subsequently accessible via its *protocol.swd* attribute.

reset() → None

Reset the SWD protocol functionality.

Raises
DwfLibraryError – An error occurred while executing the *reset* operation.

rateSet(*data_rate: float*) → None

Set the SWD bus data rate, in Hz.

Parameters
data_rate (*float*) – The data-rate used by the receiver and transmitter.

Raises
DwfLibraryError – An error occurred while executing the operation.

clockSet(*channel: int*) → None

Set the SWD clock channel.

Parameters
channel (*int*) – The SWD clock channel.

Raises
DwfLibraryError – An error occurred while executing the operation.

ioSet(*channel: int*) → None

Set the SWD I/O channel.

Parameters
channel (*int*) – The SWD I/O channel.

Raises
DwfLibraryError – An error occurred while executing the operation.

turnSet(*turn_setting: int*) → None

Set the SWD 'turn' parameter.

Parameters
turn_setting (*int*) – The turn setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

trailSet(*trail_setting: int*) → None

Set the SWD 'trail' parameter.

Parameters
trail_setting (*int*) – The trail setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

parkSet(*park_setting: int*) → None

Set the SWD 'park_setting' parameter.

Parameters
park_setting (*int*) – The park setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

nakSet(*nak_setting: int*) → None

Set the SWD 'nak_setting' parameter.

Parameters
nak_setting (*int*) – The nak setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

ioIdleSet(*io_idle_setting: int*) → None

Set the SWD 'I/O idle' parameter.

Parameters
io_idle_setting (*int*) – The I/O idle setting.

Raises
DwfLibraryError – An error occurred while executing the operation.

clear(*reset_value: int*, *trail_value*) → None

Clear the SWD bus.

Parameters

- **reset_value** (*int*) – The reset value.
- **trail_value** (*int*) – The trail value.

Raises
DwfLibraryError – An error occurred while executing the operation.

write(*port: int*, *a32: int*, *write_data: int*) → int

Perform an SWD write.

Parameters

- **port** (*int*) –
  - 0 — DataPort
  - 1 — AccessPort
- **a32** (*int*) – Address bits 3:2.

- **write_data** – Data to write.

Returns
Acknowledgement bits: 1=OK, 2=WAIT, 4=FAILURE.

Return type
int

Raises
DwfLibraryError – An error occurred while executing the operation.

read(*port: int*, *a32: int*) → Tuple[int, int, bool]

Perform an SWD read.

Parameters

- **port** (*int*) –
  - 0 — DataPort
  - 1 — AccessPort
- **a32** (*int*) – Address bits 3:2.

Returns

The first element of the tuple is the acknowledgement bits: 1=OK, 2=WAIT, 4=FAILURE.
The second element of the tuple is the data word read. The third element of the tuple indicates if the CRC was correct (parity check).

Return type
Tuple[*int*, *int*, *bool*]

Raises
DwfLibraryError – An error occurred while executing the operation.

property device

Return the *DwfDevice* instance of which we are an attribute.

This is useful if we have a variable that contains a reference to a *DwfDevice* attribute, but we need the *DwfDevice* itself.

Returns
The *DwfDevice* instance that this attribute belongs to.

Return type
DwfDevice

CHAPTER

FIVE

PYDWF EXCEPTIONS

The *PyDwfError* and *DwfLibraryError* exceptions are used to report errors from *pydwf* to user programs.

5.1 Using the *pydwf* exceptions

The *PyDwfError* and *DwfLibraryError* exceptions are defined in the *pydwf.core.auxiliary.exceptions* module. The top-level *pydwf* package imports both of them from that module to make them available to user scripts. To use either of them, they should be imported from the top-level *pydwf* package:

```
from pydwf import DwfLibrary, PyDwfError, DwfLibraryError

dwf = DwfLibrary()

try:
    use_pydwf(dwf)
except DwfLibraryError as e:
    print("An error occurred at the C library level:", e)
except PyDwfError as e:
    print("An error occurred at the Python module level:", e)
```

5.2 Error handling in the *pydwf* package

Python provides exceptions to handle errors, which is quite different from the lower level return-value based mechanism used in the C API. Fortunately, it is possible to turn the low-level errors reported by the C API functions into Python exceptions.

To do this, the *pydwf* package inspects the return value of each call to the C API, and, in case of an error (i.e., a return value unequal to 1), it raises a *DwfLibraryError* exception that contains both the DWERC error code of the last function called and its corresponding textual description, as obtained by calling the *FDwfGetLastError* and *FDwfGetLastErrorMsg* functions in the shared library.

5.3 Exceptions raised by the *pydwf* package

Almost all exceptions raised by *pydwf* are a result of a failure reported by the underlying C library. However, there are a few circumstances where *pydwf* detects an error condition before or after such a call was made. Such errors are handled by raising a *PyDwfError* exception, which derives from Python's standard `Exception` class.

Fig. 1: Inheritance diagram of *pydwf* exceptions

As the inheritance diagram shows, the *DwfLibraryError* exception type derives from *PyDwfError*. This makes it easy to catch any *pydwf* error in code:

```
from pydwf import DwfLibrary, PyDwfError

dwf = DwfLibrary()

try:
    use_pydwf(dwf)
except PyDwfError as e:
    print("An error occurred at the C library -or- Python module level:", e)
```

5.4 *pydwf* exceptions reference

class PyDwfError

A *PyDwfError* exception represents any error in *pydwf* (caused by the underlying C API or otherwise).

It is a trivial (empty) specialization of the built-in `Exception` class.

class DwfLibraryError

A *DwfLibraryError* exception represents an error reported by one of the DWF C library functions.

This class derives from *PyDwfError*, making it easier for scripts to catch any exception originating in *pydwf*.

The following attributes are provided:

code

DWF error code as reported by the C library, if available.

Type
Optional[DwfErrorCode]

msg

DWF error message as reported by the C library, if available. It may contain multiple single-line messages, separated by a newline character.

Type
Optional[*str*]

CHAPTER

SIX

PYDWF ENUMERATION TYPES

Throughout *pydwf*, enumeration types are used as parameters or return values. They are generally used when a parameter or return value can take on a small number of specific values.

6.1 Using the *pydwf* enumeration types

The enumeration types are defined in the *pydwf.core.auxiliary.enum_types* module. They are Python equivalents of the 27 enumerations defined in the *dwf.h* header file provided by Digilent.

Note:

pydwf does not replicate the obsolete enumerations *TRIGCOND* and *STS* that are defined in the C header file. *TRIGCOND* has been replaced by `DwfTriggerSlope`; *STS* has been replaced by `DwfState`.

The definitions in the header file do not use C enum types, but rather use a *typedef* to define a type name that is an alias for either *int* or *unsigned char*, followed by a number of constant declarations. For example:

```
// instrument states:
typedef unsigned char DwfState;
const DwfState DwfStateReady        = 0;
const DwfState DwfStateConfig       = 4;
const DwfState DwfStatePrefill      = 5;
const DwfState DwfStateArmed        = 1;
const DwfState DwfStateWait         = 7;
const DwfState DwfStateTriggered    = 3;
const DwfState DwfStateRunning      = 3;
const DwfState DwfStateDone         = 2;
```

The enumeration type names in the C library are a mix between different naming styles. For reasons of consistently, we decided to rename the types in *pydwf*; the table below shows the correspondence between C and Python names.

Table 1: Correspondence between *libdwf* and *pydwf* enumeration type names

libdwf name	*pydwf* name	used by
DWFERC	*DwfErrorCode*	`DwfLibrary.getLastError()` method
ENUMFILTER	*DwfEnumFilter*	*DeviceEnum.enumerateDevices()* method
DwfEnumConfigInfo	*DwfEnumConfigInfo*	*DeviceEnum.configInfo()* method
DEVID	*DwfDeviceID*	*DeviceEnum.deviceType()* method
DEVVER	*DwfDeviceVersion*	*DeviceEnum.deviceType()* method
DwfParam	*DwfDeviceParameter*	*DwfLibrary*, *DwfDevice* methods
DwfWindow	*DwfWindow*	*DwfLibrary* methods
DwfState	*DwfState*	all 4 main instruments; *AnalogImpedance*
TRIGSRC	*DwfTriggerSource*	all 4 main instruments; *DeviceControl*
DwfTriggerSlope	*DwfTriggerSlope*	all 4 main instruments
ACQMODE	*DwfAcquisitionMode*	*AnalogIn* and *DigitalIn* instruments
FILTER	*DwfAnalogInFilter*	*AnalogIn* instrument
DwfAnalogCoupling	*DwfAnalogCoupling*	*AnalogIn* instrument
TRIGTYPE	*DwfAnalogInTriggerType*	*AnalogIn* instrument
TRIGLEN	*DwfAnalogInTriggerLengthCondit*	*AnalogIn* instrument
FUNC	*DwfAnalogOutFunction*	*AnalogOut* instrument
AnalogOutNode	*DwfAnalogOutNode*	*AnalogOut* instrument
DwfAnalogOutMode	*DwfAnalogOutMode*	*AnalogOut* instrument
DwfAnalogOutIdle	*DwfAnalogOutIdle*	*AnalogOut* instrument
DwfDigitalInClock-Source	*DwfDigitalInClockSource*	*DigitalIn* instrument
DwfDigitalInSample-Mode	*DwfDigitalInSampleMode*	*DigitalIn* instrument
DwfDigitalOutOutput	*DwfDigitalOutOutput*	*DigitalOut* instrument
DwfDigitalOutType	*DwfDigitalOutType*	*DigitalOut* instrument
DwfDigitalOutIdle	*DwfDigitalOutIdle*	*DigitalOut* instrument, *ProtocolSPI* support
ANALOGIO	*DwfAnalogIO*	*AnalogIO* functionality
DwfAnalogImpedance	*DwfAnalogImpedance*	*AnalogImpedance.statusMeasure()* method
DwfDmm	*DwfDmm*	*not currently used*

Note: The *DwfDmm* type is defined, but not yet used by any API. It appears to be geared towards support for DMM functionality offered by the ADP 5250 device.

The top-level *pydwf* package imports the enumeration types from *pydwf.core.auxiliary.enum_types* to make them available to user scripts. To use these types, you should import the ones you need from the top-level *pydwf* package:

```
# Here, we import all 27 pydwf enumeration types.
# In practical scripts, only a few of these will be imported.

from pydwf import (DwfErrorCode, DwfEnumFilter, DwfEnumConfigInfo, DwfDeviceID,
                   DwfDeviceVersion, DwfDeviceParameter, DwfWindow, DwfState,
                   DwfTriggerSource, DwfTriggerSlope, DwfAcquisitionMode,
                   DwfAnalogInFilter, DwfAnalogCoupling, DwfAnalogInTriggerType,
                   DwfAnalogInTriggerLengthCondition, DwfAnalogOutFunction,
                   DwfAnalogOutNode, DwfAnalogOutMode, DwfAnalogOutIdle,
                   DwfDigitalInClockSource, DwfDigitalInSampleMode,
                   DwfDigitalOutOutput, DwfDigitalOutType, DwfDigitalOutIdle,
                   DwfAnalogIO, DwfAnalogImpedance, DwfDmm)
```

6.2 *pydwf* enumeration classes reference

class DwfErrorCode

Enumeration type for error reporting constants of the DWF API.

This type is used by the `DwfLibrary.getLastError()` method to report the error condition of the most recent C API call.

In *pydwf*, it is only used as the type of the *code* field of *DwfLibraryError* instances.

In the C API, this type is called 'DWFERC', and it is represented as an *int*.

NoErc = 0

No error occurred.

UnknownError = 1

Call waiting on pending API time out.

ApiLockTimeout = 2

Call waiting on pending API time out.

AlreadyOpened = 3

Device already opened.

NotSupported = 4

Device not supported.

InvalidParameter0 = 16

Invalid parameter sent in API call.

InvalidParameter1 = 17

Invalid parameter sent in API call.

InvalidParameter2 = 18

Invalid parameter sent in API call.

InvalidParameter3 = 19

Invalid parameter sent in API call.

InvalidParameter4 = 20

Invalid parameter sent in API call.

Note: This value is not listed in the most recent version of the documentation.

class DwfEnumFilter

Enumeration type for device class constants, used during device enumeration.

This type is used by the *DeviceEnum.enumerateDevices()* and *DeviceEnum.enumerateStart()* methods to constrain the type of devices that will be enumerated.

In the C API, this type is called 'ENUMFILTER', and it is represented as an *int*.

All = 0

Enumerate all available devices.

DEVID = 1

Use devic to filter specific devices.

Type = 134217728

Use filters below (0x08000000).

USB = 1

Enumerate USB devices.

Network = 2

Enumerate Network devices.

AXI = 4

Enumerate embedded devices (used when running on an ADP 3x50 device).

Remote = 16777216

Enumerate remote table devices (0x01000000).

Audio = 33554432

Enumerate sound card devicess (0x02000000).

Demo = 67108864

Enumerate demo devices (0x04000000).

class DwfEnumConfigInfo

Enumeration type for device configuration parameter type constants.

This type lists the device parameters that can vary between different *device configurations* of the same device. It is used exclusively by the *DeviceEnum.configInfo()* method.

In the C API, this type is represented as an *int*.

TooltipText = -1

Tooltip text.

Maximum length: 2048 characters.

Note: This value is not officially documented. Its existence was revealed in a message on the Digilent forum.

OtherInfoText = -2

Other info text.

Maximum length: 256 characters.

Note: This value is not officially documented. Its existence was revealed in a message on the Digilent forum.

AnalogInChannelCount = 1

Number of analog input channels.

AnalogOutChannelCount = 2

Number of analog output channels.

AnalogIOChannelCount = 3

Number of analog power supply channels.

Note: This is a different number than the number of channels reported by the *AnalogIO.channelCount()* method.

DigitalInChannelCount = 4

Number of digital input channels.

DigitalOutChannelCount = 5

Number of digital output channels.

DigitalIOChannelCount = 6

Number of digital I/O channels.

AnalogInBufferSize = 7

Analog in buffer size, in samples.

AnalogOutBufferSize = 8

Analog out buffer size, in samples.

DigitalInBufferSize = 9

Digital in buffer size, in samples.

DigitalOutBufferSize = 10

Digital out buffer size, in samples.

class DwfDeviceID

Enumeration type for device ID constants.

This type is used by the *DeviceEnum.deviceType()* method to report the type of a previously enumerated device.

In the C API, this type is called 'DEVID', and it is represented as an *int*.

EExplorer = 1

Electronics Explorer devices.

Discovery = 2

Analog Discovery (1) devices.

Discovery2 = 3

Analog Discovery 2 devices.

DDiscovery = 4

Digital Discovery devices.

ADP3X50 = 6

Analog Discovery Pro devices.

Eclypse = 7

Eclypse devices.

ADP5250 = 8

ADP5250 devices.

DPS3340 = 9

DPS 3340 devices.

class DwfDeviceVersion

Enumeration type for device version (i.e., hardware revision) constants.

This type is used by the *DeviceEnum.deviceType()* method to report the hardware revision of a previously enumerated device.

Note: The device revision list given here is not complete; it does not cover all devices.

In the C API, this type is called 'DEVVER', and it is represented as an *int*.

EExplorerC = 2

Electronics Explorer devices, revision C.

EExplorerE = 4

Electronics Explorer devices, revision E.

EExplorerF = 5

Electronics Explorer devices, revision F.

DiscoveryA = 1

Discovery devices, revision A.

DiscoveryB = 2

Discovery devices, revision B.

DiscoveryC = 3

Discovery devices, revision C.

class DwfDeviceParameter

Enumeration type for device parameter constants.

Device parameters are miscellaneous integer settings that influence the behavior of a device.

The different device parameters are selected by one of the constant values defined here.

This type is used to select device parameters, either to set/get global defaults using the *DwfLibrary*, or to to set/get parameter values on a specific, previously opened device *DwfDevice*.

In the C API, this type is called 'DwfParam', and it is represented as an *int*.

KeepOnClose = 1

Keep the device running after close.

> **Warning:** This value is *obsolete*. Use *OnClose* instead.

Note: This value is not listed in the most recent version of the documentation.

UsbPower = 2

USB power behavior if AUX power is connected.

Possible values:

- 0 — Disable USB power.
- 1 — Keep USB power enabled.

This setting is implemented on the Analog Discovery 2.

Note: This value is not listed in the most recent version of the documentation.

LedBrightness = 3

Set multicolor LED brightness.

The Digital Discovery features a multi-color LED. It is normally blue in case the device is not currently controlled by software, or green if it is.

Setting this parameter from 0 to 100 changes the LED's relative brightness, in percents. This can be useful, for example, in a lab with sensitive optics that would preferably be completely dark.

On the Analog Discovery 2, this setting has no effect.

`OnClose = 4`

Define behavior on close.

Possible values:

- 0 — On close, continue.
- 1 — On close, stop the device outputs but keep the device operational to prevent temperature drifts.
- 2 — On close, shut down the device to minimize power consumption.

`AudioOut = 5`

Enable or disable audio output.

Possible values:

- 0 — Disable audio output.
- 1 — Enable audio output (default).

This setting is implemented on the Analog Discovery and the Analog Discovery 2.

`UsbLimit = 6`

USB power limit.

The value ranges from 0 to 1000, in mA. The value -1 denotes no limit. Recommended value is in the 600—1000 mA range.

This setting is implemented on the Analog Discovery and the Analog Discovery 2.

`AnalogOut = 7`

Enable or disable analog output.

Possible values:

- 0 — Disable analog output.
- 1 — Enable analog output.

This setting is implemented on the Analog Discovery Pro 3x50.

`Frequency = 8`

100 MHz.

This setting is implemented on the Digital Discovery and Analog Discovery Pro 3x50.

Type
Adjust system frequency in Hz. Default

`ExtFreq = 9`

10 MHz.

This setting is implemented on the Analog Discovery Pro 3x50.

Type
Specify for input or set reference output frequency in Hz. Default

`ClockMode = 10`

This parameter is undocumented.

Todo: The meaning of this parameter needs to be understood.

Possible values:

- 0 — Use internal oscillator (default).
- 1 — Enable reference output on trigger-1 channel.
- 2 — Use reference input from trigger-1 channel.

- 3 — Use trigger-1 as reference I/O.

This setting is implemented on the Analog Discovery Pro 3x50.

TempLimit = 11

Specify the over temperature threshold in degree Celcius on devices which support such option.

Todo: The meaning of this parameter needs to be understood.

FreqPhase = 12

Specify the system clock phase which is useful for device synchronization when a reference input clock is used.

Todo: The meaning of this parameter needs to be understood.

class DwfWindow

Enumeration type for signal processing windows.

Rectangular = 0

Rectangular window, a.k.a. no window.

Triangular = 1

Triangular window.

Hamming = 2

Hamming window.

Hann = 3

Hann window.

Cosine = 4

Cosine window.

BlackmanHarris = 5

Blackman-Harris window.

FlatTop = 6

Flat-top window.

Kaiser = 7

Kaiser window.

class DwfState

Enumeration type for instrument state constants, for instruments that are controlled by an internal state-machine.

The following instrument APIs are controlled by a state machine:

- *AnalogIn*
- *AnalogOut — independent state machine for each channel*
- *DigitalIn*
- *DigitalOut*
- *AnalogImpedance*

This type is used to return the current state from their *status()* methods: *AnalogIn.status()*, *AnalogOut.status()*, *DigitalIn.status()*, *DigitalOut.status()*, and *AnalogImpedance.status()*.

Note: The enumeration values *Triggered* and *Running* have identical integer values (3).

The state name *Triggered* is used for capture instruments (*AnalogIn*, *DigitalIn*), while *Running* is used for signal generation instruments (*AnalogOut*, *DigitalOut*).

In the C API, this type is represented as an *unsigned char*.

Ready = 0

The instrument is idle, waiting to be configured or started.

Config = 4

The instrument is being configured.

Prefill = 5

The instrument is collecting data prior to arming itself, so it can deliver pre-trigger samples.

Armed = 1

The instrument is collecting samples and waiting for the trigger.

Wait = 7

The signal generation instrument is waiting before its next run.

Triggered = 3

The capture instrument is triggered and collecting data.

Running = 3

The signal generation instrument is running (generating signals).

Done = 2

The instrument has completed a measurement or signal-generating sequence.

class DwfTriggerSource

Enumeration type for trigger source constants.

This type is used by the *DeviceControl* functionality to configure an external trigger and by the *AnalogIn*, *AnalogOut*, *DigitalIn*, and *DigitalOut* instruments to select a trigger source. The *AnalogIn* instrument can also use this type to configure a trigger as a sampling source.

In the C API, this type is called 'TRIGSRC', and it is represented as an *unsigned char*.

None_ = 0

No trigger configured (device starts immediately).

The trigger pin is high impedance (input). This is the default setting.

PC = 1

PC (software) trigger. This can be used, for example, to synchronously start multiple instruments.

DetectorAnalogIn = 2

AnalogIn trigger detector.

DetectorDigitalIn = 3

DigitalIn trigger detector.

AnalogIn = 4

AnalogIn instrument trigger. The trigger level is high when the instrument is running.

DigitalIn = 5

DigitalIn instrument trigger. The trigger level is high when the instrument is running.

DigitalOut = 6

DigitalOut instrument trigger. The trigger level is high when the instrument is running.

AnalogOut1 = 7

AnalogOut instrument trigger 1 start. The trigger level is high when the instrument is running.

AnalogOut2 = 8

AnalogOut instrument trigger 2 start. The trigger level is high when the instrument is running.

AnalogOut3 = 9

AnalogOut instrument trigger 3 start. The trigger level is high when the instrument is running.

AnalogOut4 = 10

AnalogOut instrument trigger 4 start. The trigger level is high when the instrument is running

External1 = 11

External trigger signal #1.

External2 = 12

External trigger signal #2.

External3 = 13

External trigger signal #3.

External4 = 14

External trigger signal #4.

High = 15

High (undocumented).

Low = 16

Low (undocumented).

Clock = 17

Clock (undocumented).

class DwfTriggerSlope

Enumeration type for trigger slope constants.

This type is used by the *AnalogIn*, *AnalogOut*, *DigitalIn*, and *DigitalOut* instruments to select the trigger slope.

In addition, the *AnalogIn* instrument uses it to select the slope of the sampling clock.

In the C API, this type is represented as an *int*.

Rise = 0

Rising trigger slope.

Fall = 1

Falling trigger slope.

Either = 2

Either rising or falling trigger slope.

class DwfAcquisitionMode

Enumeration type for acquisition mode constants.

This type is used by the *AnalogIn* and *DigitalIn* instruments. These instruments support multiple acquisition modes that are appropriate for different data acquisition tasks.

In the C API, this type is called 'ACQMODE', and it is represented as an *int*.

Single = 0

Perform a single buffer acquisition.

Re-arm the instrument for the next capture after the data is fetched to the host using the instrument-specific *status()* method: `AnalogIn.status()` or `DigitalIn.status()`.

Note: The difference with the `Single1` mode is unclear.

ScanShift = 1

Perform a continuous acquisition in FIFO style.

The trigger setting is ignored.

The last sample is at the end of the buffer. The instrument's *statusSamplesValid()* method gives the number of the acquired samples, which will increase until reaching the buffer size. After that, the waveform image is shifted for every new sample.

ScanScreen = 2

Perform continuous acquisition, circularly writing samples into the buffer.

This is similar to a heart-monitor display.

The trigger setting is ignored.

The instrument's *statusIndexWrite()* method gives the buffer write position.

Record = 3

Perform acquisition for the length of time set by the instrument's *recordLengthSet()* method.

Overs = 4

Overscan mode.

Note: This value is not listed in the most recent version of the documentation.

Single1 = 5

Perform a single buffer acquisition.

Note: The difference with the `Single` mode is unclear.

---

class DwfAnalogInFilter

Enumeration type for analog input filter constants.

This type is used by the *AnalogIn* instrument to select a filtering algorithm for the input and trigger channels.

The *AnalogIn* instrument's ADC always captures samples at the maximum possible rate. If data acquisition at a lower sampling rate is requested, the resampling can be handled in several different ways.

The most obvious choice is *averaging*. This will suppress high-frequency noise, which is often a good thing, but sometimes it is desirable to know that high-frequency noise is present in the signal, and the averaging may hide that fact.

For that reason, the *decimation* filter is available, which simply selects a single sample captured at high frequency when resampling to a lower frequency. The signal-to-noise ratio (SNR) will suffer, but the presence of high-frequency noise (outliers) will be more easily seen in the resampled data.

Todo: Examine the MinMax filter choice; it is not currently understood.

In the C API, this type is called 'FILTER', and it is represented as an *int*.

Decimate = 0

Decimation filter. Store every N'th ADC conversion, where N = ADC frequency / acquisition frequency.

Average = 1

Averaging filter. Store the average of N ADC conversions.

MinMax = 2

Min/max filter. Store, interleaved, the minimum and maximum values, of 2 * N concersions.

AverageFit = 3

Averaging fit filter.

Note: This value is not listed in the most recent version of the documentation.

class DwfAnalogCoupling

Enumeration type for analog coupling configuration.

DC = 0

DC coupling.

AC = 1

AC coupling.

class DwfAnalogInTriggerType

Enumeration type for analog input trigger mode constants.

This type is used by the *AnalogIn* instrument to specify the trigger type.

In the C API, this type is called 'TRIGTYPE', and it is represented as an *int*.

Edge = 0

Edge trigger type.

Pulse = 1

Pulse trigger type.

Transition = 2

Transition trigger type.

Window = 3

Window trigger type.

class DwfAnalogInTriggerLengthCondition

Enumeration type for analog input trigger length condition constants.

This type is used by the *AnalogIn* instrument to specify the trigger length condition.

In the C API, this type is called 'TRIGLEN', and it is represented as an *int*.

Less = 0

Trigger length condition 'less'.

Timeout = 1

Trigger length condition 'timeout'.

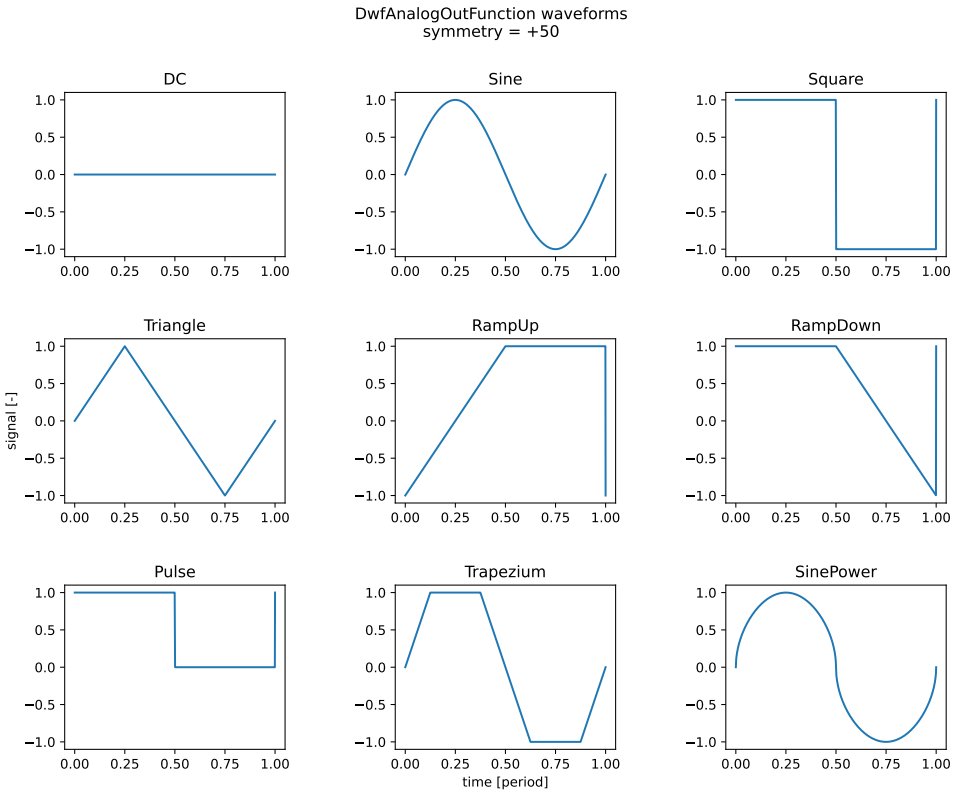
More = 2

Trigger length condition 'more'.

class DwfAnalogOutFunction

Enumeration type for analog output waveform-shape function constants.

This type is used by the *AnalogOut* instrument to represent the wave-shape produced on an analog output channel node. The nine fixed waveform shape options are shown below.



In the C API, this type is called 'FUNC', and it is represented as an *unsigned char*.

DC = 0

DC (constant signal) waveform shape. The signal level varies between -1 and 1.

Sine = 1

Sinusoid waveform shape. The signal level varies between -1 and 1.

Square = 2

Square waveform shape. The signal level varies between -1 and 1.

Triangle = 3

Triangle waveform shape. The signal level varies between -1 and 1.

RampUp = 4

Ramp Up waveform shape. The signal level varies between -1 and 1.

RampDown = 5

Ramp Down waveform shape. The signal level varies between -1 and 1.

Noise = 6

Noise waveform shape. The signal level is uniformly distributed between -1 and 1.

Pulse = 7

Pulse waveform shape. The signal level varies between 0 and 1.

Trapezium = 8

Trapezium waveform shape. The signal level varies between -1 and 1.

SinePower = 9

Sinusoid Power waveform shape. The signal level varies between -1 and 1.

CustomPattern = 28

Generate waveform from custom samples.

It provides constant sample rate, supporting integer divisions of the system frequency.

Note: This description is taken verbatim from the documentation (*dwfsdk.pdf*). It is unclear what it means.

PlayPattern = 29

Generate waveform in stream play style. It provides constant sample rate.

Note: This description is taken verbatim from the documentation (*dwfsdk.pdf*). It is unclear what it means.

Custom = 30

Generate waveform from custom samples.

Optimizes for average requested frequency, sample output lengths may vary by one system frequency period.

Note: This description is taken verbatim from the documentation (*dwfsdk.pdf*). It is unclear what it means.

Play = 31

Generate waveform in stream play style. Optimizes for average requested frequency.

Note: This description is taken verbatim from the documentation (*dwfsdk.pdf*). It is unclear what it means.

class DwfAnalogOutNode

Enumeration type for analog output node type constants.

This type is used by the *AnalogOut* instrument to represent the node types associated with each output channel.

In the C API, this type is called 'AnalogOutNode' (without the *Dwf* prefix), and it is represented as an *int*.

Carrier = 0

Carrier signal node. This node represents the base signal without modulation applied.

FM = 1

Frequency Modulation node. (Frequency or Phase modulation, according to the latest documentation).

AM = 2

Amplitude Modulation node. (Amplitude or Sum modulation, according to the latest documentation).

class DwfAnalogOutMode

Enumeration type for analog out mode constants (voltage or current).

This type is used by the *AnalogOut* instrument to set or retrieve the mode of a channel.

In the C API, this type is represented as an *int*.

Voltage = 0

Voltage mode.

Current = 1

Current mode.

class DwfAnalogOutIdle

Enumeration type for analog output idle state constants.

This type is used by the *AnalogOut* instrument to set the idle behavior of an output channel.

In the C API, this type is represented as an *int*.

Disable = 0

When idle, disable the output.

Offset = 1

When idle, drive the configured analog output offset.

Initial = 2

When idle, drive the initial value of the selected waveform shape.

class DwfDigitalInClockSource

Enumeration type for digital input clock source constants.

This type is used by the *DigitalIn* instrument to specify a clock source.

In the C API, this type is represented as an *int*.

Internal = 0

Use internal clock source.

External = 1

Use external clock source.

External2 = 2

Use alternate external clock source.

class DwfDigitalInSampleMode

Enumeration type for digital input sample mode constants.

This type is used by the *DigitalIn* instrument to specify a sample mode.

In the C API, this type is represented as an *int*.

Simple = 0

Only digital samples (no noise).

Noise = 1

Alternate samples (noise, sample, noise, sample, ...) where noise is more than one transition between two samples.

This setting is available when the sample rate is less than the maximum clock frequency (i.e., the divider is greater than one). Digital noise means more than one transition between subsequent samples was detected; this can indicate glitches or ringing.

class DwfDigitalOutOutput

Enumeration type for digital output mode constants.

This type is used by the *DigitalOut* instrument to specify the electronic behavior of a digital output channel.

In the C API, this type is represented as an *int*.

PushPull = 0

Push/Pull.

OpenDrain = 1

Open Drain.

OpenSource = 2

Open Source.

ThreeState = 3

Tristate (for custom and random).

class DwfDigitalOutType

Enumeration type for digital output type constants.

This type is used by the *DigitalOut* instrument to specify the behavior mode of a digital output channel.

In the C API, this type is represented as an *int*.

Pulse = 0

Pulse output.

Custom = 1

Custom output.

Random = 2

Random output.

ROM = 3

ROM (lookup table) output.

State = 4

State machine output.

Play = 5

Continuous playback output.

class DwfDigitalOutIdle

Enumeration type for digital output idle mode constants.

This type is used primarily by the *DigitalOut* instrument to specify the idle behavior mode of a digital output channel.

In addition to that, it is used by the *ProtocolSPI* functionality to specify the idle behavior of the pins it controls.

In the C API, this type is represented as an *int*.

Init = 0

Same as initial value of selected output pattern.

Low = 1

Low signal level.

High = 2

High signal level.

Zet = 3

High impedance.

class DwfAnalogIO

Enumeration type for Analog I/O channel node type constants.

This type is used by the *AnalogIO* functionality to report the node type.

In the C API, this type is called 'ANALOGIO', and it is represented as an *unsigned char*.

Undocumented = 0

This value was returned in Analog Discovery Pro devices when using 3.16.3 of the DWF library. This was a bug that was subsequently fixed.

Enable = 1

The node represent an on/off switch.

Voltage = 2

The node represents a voltage.

Current = 3

The node represents a current.

Power = 4

The node represents a power.

Temperature = 5

The node represents a temperature.

Dmm = 6

The node represents a DMM (digital multimeter) value.

Range = 7

The node represents a range.

Measure = 8

(unknown)

Time = 9

The node represents a time.

Frequency = 10

The node represents a frequency.

Resistance = 11

The node represents a resistance.

Slew = 12

(unknown)

class DwfAnalogImpedance

Enumeration type for analog impedance measurement types.

This type is used by the *AnalogImpedance* measurement functionality to specify a measurement quantity type.

In the C API, this type is represented as an *int*.

Impedance = 0

Measure impedance, in Ohms.

ImpedancePhase = 1

Measure impedance phase, in radians.

Resistance = 2

Measure resistance, in Ohms.

Reactance = 3

Measure reactance, in Ohms.

Admittance = 4

Measure admittance, in Siemens.

AdmittancePhase = 5

Measure admittance phase, in radians.

Conductance = 6

Measure conductance, in Siemens.

Susceptance = 7

Measure susceptance, in Siemens.

SeriesCapacitance = 8

Measure series capacitance, in Farad.

ParallelCapacitance = 9

Measure parallel capacitance, in Farad.

SeriesInductance = 10

Measure series inductance, in Henry.

ParallelInductance = 11

Measure parallel inductance, in Henry.

Dissipation = 12

Measure dissipation, as a factor.

Quality = 13

Measure quality, as a factor.

Vrms = 14

Measure Vrms, in Volts.

Vreal = 15

Measure Vreal (real part of complex voltage), in Volts.

Vimag = 16

Measure Vimag (imaginary part of complex voltage), in Volts.

Irms = 17

Measure Irms, in Amps.

Ireal = 18

Measure Ireal (real part of complex current), in Amps.

Iimag = 19

Measure Iimag (imaginary part of complex current), in Amps.

class DwfDmm

Enumeration type for DMM (digital multimeter) measurements.

Note: This type is currently unused in the API. It is intended for functionality in the new ADP5250 device.

In the C API, this type is called 'DwfDmm', and it is represented as an *int*.

Resistance = 1

Resistance measurement.

Continuity = 2

Continuity measurement.

Diode = 3

Diode measurement.

DCVoltage = 4

DC voltage measurement.

ACVoltage = 5

AC voltage measurement.

DCCurrent = 6

DC current measurement.

ACCurrent = 7

AC current measurement.

DCLowCurrent = 8

DC low current measurement.

ACLowCurrent = 9

AC low current measurement.

Temperature = 10

Temperature measurement.

CHAPTER

SEVEN

PYDWF UTILITIES

The *pydwf.utilities* package provides functionality built on top of the core functionality that is available in the *pydwf* core package. It provides high-level functions that reflect best-practice implementations for common use-cases of *pydwf*.

Currently, only a single utility function is provided by the *pydwf.utilities* module: *pydwf.utilities.openDwfDevice()*, which is documented below.

7.1 Using the *pydwf.utilities* functionality

To use functionality from the *pydwf.utilities* package, import the symbols you need from it:

```
"""Demonstrate use of the openDwfDevice function."""

from pydwf import DwfLibrary
from pydwf.utilities import openDwfDevice

dwf = DwfLibrary()

with openDwfDevice(dwf) as device:
    use_dwf_device(device)
```

The example above demonstrates the use of the *pydwf.utilities.openDwfDevice()* function. This function encapsulates a number of core *pydwf* functions to allow easy selection of devices and device configurations. It is documented below.

7.2 *pydwf.utilities.openDwfDevice* function reference

openDwfDevice(*dwf:* DwfLibrary, *enum_filter:* DwfEnumFilter | *None* = *None*, *serial_number_filter:* *str* | *None* = *None*, *device_id_filter:* *int* | *None* = *None*, *device_version_filter:* *int* | *None* = *None*, *score_func:* *Callable[[Dict[*DwfEnumConfigInfo*, *Any*]], *Any* | *None*]* | *None* = *None*) → *DwfDevice*

Open a device identified by its serial number, optionally selecting a preferred configuration.

This is a three-step process:

1. The first step this function performs is to select a device for opening.

   To do this, device enumeration is performed, resulting in a list of all reachable Digilent Waveforms devices.

   For this initial enumeration process the *enum_filter* parameter can be used to only list certain types of devices (for example, only Analog Discovery 2 devices, or Digital Discovery devices). If omitted (the default), all Digilent Waveforms devices will be listed.

Then, if the *serial_number_filter* parameter is given, the list will be filtered to exclude devices whose serial numbers do not match.

If the list that remains has a single device, this device will be used. If not, a *PyDwfError* is raised.

2. The next step is to select a *device configuration* for the selected device.

   For many use cases, the default configuration that provides a balanced tradeoff works fine. If no *score_func* is provided, this default configuration will be used.

   If a *score_func* parameter is provided, it should be a function (or lambda expression) that takes a single parameter *configuration_info*, which is a dictionary with *DwfEnumConfigInfo* keys, and parameters values for a specific device configuration.

   The *score_func* should return None if the configuration is entirely unsuitable, or otherwise a score that reflects the suitability of that particular configuration for the task at hand.

   The *openDwfDevice* method will go through all available device configurations, construct a dictionary of all parameters that describe the configuration, call the *score_func* with that dictionary as a parameter, and examine the score value it returns. If multiple suitable device configurations are found (i.e., the *score_func* does not return None), it will select the configuration with the highest score.

   This may all sounds pretty complicated, but in practice this parameter is quite easy to define for most common use-cases.

   As an example, to select a configuration that maximizes the analog input buffer size, simply use this:

```
from pydwf import DwfEnumConfigInfo
from pydwf.utilities import openDwfDevice

def maximize_analog_in_buffer_size(config_parameters):
    return config_parameters[DwfEnumConfigInfo.AnalogInBufferSize]

with openDwfDevice(dwf, score_func = maximize_analog_in_buffer_size) as␣
↪device:
    use_device_with_big_analog_in_buffer(device)
```

3. As a final step, the selected device is opened using the selected device configuration, and the newly instantiated *DwfDevice* instance is returned.

Note: This method can take several seconds to complete. This long duration is caused by the use of the *DeviceEnum.enumerateDevices()* method.

Parameters

- **dwf** (*DwfLibrary*) – The *DwfLibrary* used to open the device.
- **enum_filter** (*Optional[DwfEnumFilter]*) – An optional filter to limit the device enumeration to certain device types. If None, enumerate all devices.
- **serial_number_filter** (*str*) – The serial number filter used to select a specific device. A device is considered to match if its 12-digit serial number ends with the serial number filter string (case is ignored).
- **device_id_filter** (*int*) – The device ID filter to use.
- **device_version_filter** (*int*) – The device version filter to use.
- **score_func** (*Optional[Callable[[Dict[DwfEnumConfigInfo, Any]], Optional[Any]]]*) – A function to score a configuration of the selected device. See the description above for details.

Returns

The *DwfDevice* created as a result of this call.

Return type
DwfDevice

Raises
PyDwfError – could not select a unique candidate device, or no usable configuration detected.

CHAPTER

EIGHT

USING *PYDWF* AS A COMMAND LINE TOOL

After installation, the *pydwf* package itself can by executed as a command line tool:

```
$ python3 -m pydwf
```

This tool provides a number of sub-commands:

version
: Show the version of the *pydwf* package and the DWF C library.

list
: List all available Digilent Waveforms devices. Add the option '-c' to show the supported configurations for each device.

extract-examples
: Extract a local directory with example Python scripts.

extract-html-docs
: Extract a local directory with the HTML documentation.

extract-pdf-manual
: Extract the documentation as a PDF file.

The command line tool will output help if the '-h' command line option is provided. Below, the output of the generic help is shown.

```
$ python3 -m pydwf -h
usage: python -m pydwf [-h] {version,list,ls,extract-examples,extract-html-docs} ...

Utilities for the pydwf package.

positional arguments:
  {version,list,ls,extract-examples,extract-html-docs}
    version             show version of pydwf and the DWF library
    list (ls)           list Digilent Waveform devices
    extract-examples    extract pydwf example scripts to 'pydwf-examples' directory
    extract-html-docs   extract pydwf HTML documentation to 'pydwf-html-docs'␣
↪directory
    extract-pdf-manual  extract pydwf PDF manual in current directory

optional arguments:
  -h, --help            show this help message and exit
```

To get help for a specific sub-command, specify the sub-command in question followed by the '-h' command line option.

CHAPTER

NINE

TRIGGERING EXPLAINED

Todo: This section is currently incomplete.

The intention is to discuss here the triggering bus architecture of the Digilent Waveforms devices, and to provide a detailed explanation of the possibilities and limitations.

It would also be useful to provide some specs (latency, jitter) for common devices, which could be measured.

The Digilent Waveforms devices provide an internal triggering bus, allowing the *AnalogIn*, *AnalogOut*, *DigitalIn*, and *DigitalOut* instruments to start their operation (signal capture or signal generation) at a precisely defined time relative to some internal or external event.

The triggering infrastructure is highly flexible. The trigger detector functionality of the *AnalogIn* and *DigitalIn* instruments can be used not only by those instruments themselves, but also by any of the other instruments, and it is also possible to have instruments trigger on the start of one of the other instruments, externally supplied trigger signals, or a trigger signal sent from the controlling PC.

9.1 Trigger sources

The following trigger sources are available:

- *DwfTriggerSource.None_*
- *DwfTriggerSource.PC*
- *DwfTriggerSource.DetectorAnalogIn*
- *DwfTriggerSource.DetectorDigitalIn*
- *DwfTriggerSource.AnalogIn*
- *DwfTriggerSource.DigitalIn*
- *DwfTriggerSource.DigitalOut*
- *DwfTriggerSource.AnalogOut1*
- *DwfTriggerSource.AnalogOut2*
- *DwfTriggerSource.AnalogOut3*
- *DwfTriggerSource.AnalogOut4*
- *DwfTriggerSource.External1*
- *DwfTriggerSource.External2*
- *DwfTriggerSource.External3*
- *DwfTriggerSource.External4*
- *DwfTriggerSource.High*

- *DwfTriggerSource.Low*
- *DwfTriggerSource.Clock*

9.2 Trigger timing and precision

(to be written)

CHAPTER

TEN

DEVICE PARAMETERS

Todo: This section is currently incomplete.

It currently only lists the device parameters. The intention of this section is to provide detailed information on what the settings do for the different Digilent Waveforms devices that exist.

The following device parameters have been defined:

- *DwfDeviceParameter.UsbPower*
- *DwfDeviceParameter.LedBrightness*
- *DwfDeviceParameter.OnClose*
- *DwfDeviceParameter.AudioOut*
- *DwfDeviceParameter.UsbLimit*
- *DwfDeviceParameter.AnalogOut*
- *DwfDeviceParameter.Frequency*

CHAPTER

ELEVEN

DIGILENT WAVEFORMS DEVICES AND THEIR CONFIGURATIONS

This section introduces device configurations and lists all Digilent Waveforms devices (past and present) and the device configurations they support.

Todo: This section is currently incomplete.

We need to gather data on more devices.

11.1 About device configurations

For some Digilent Waveforms devices, multiple firmware configurations are available that make different tradeoffs for certain hard-coded parameters. For example, one configuration may support a lot of buffering memory for the *AnalogIn* instrument, while another configuration may support a lot of buffering memory for the *DigitalOut* instrument. Several examples of different devices and their configurations are shown below.

Devices have a default configuration that represent a balanced tradeoff in functionality that works well for most applications. However, for some user programs, it may be useful to select a different device configuration, that is better suited to the task at hand. The device enumeration API provides a way to enumerate the relevant properties of all available device configurations. A user application can use this information to select the best device configuration, and activate it while opening the device.

Handling device enumeration and device configurations involves quite a bit of bookkeeping. For many applications, this complexity can be avoided by using the *pydwf.utilities.openDwfDevice()* convenience function. It provides an easy-to-use alternative to dealing with device enumeration and device configurations directly.

11.2 An overview of Digilent Waveforms devices

We collected the information given below by executing the following command:

```
$ python3 -m pydwf list -c
```

11.2.1 Electronics Explorer (legacy)

An older device. This is the predecessor of the Analog Discovery Studio.

(no data)

11.2.2 Analog Discovery (legacy)

An older device, coming in a black enclosure. This is the predecessor of the Analog Discovery 2.

(no data)

11.2.3 Analog Discovery 2

A low-cost multi-function device; the successor to the original Analog Discovery and the predecessor of the Analog Discovery 3.

```
device ........... : 3
version .......... : 3
user_name ........ : 'Discovery2'
device_name ...... : 'Analog Discovery 2'

Configuration:               0       1       2       3       4       5       6       7
----------------------  ------  ------  ------  ------  ------  ------  ------  ------
AnalogInChannelCount          2       2       2       2       2       2       2       2
AnalogOutChannelCount         2       2       2       2       2       4       2       2
AnalogIOChannelCount          2       2       2       2       2       2       2       2
DigitalInChannelCount        16      16       0      16      16      16      16      16
DigitalOutChannelCount       16       0       0      16      16       8      16      16
DigitalIOChannelCount        16      16      16      16      16      16      16      16
AnalogInBufferSize         8192   16384    2048     512    8192    8192     512    8192
AnalogOutBufferSize        4096    1024   16384     256    4096    4096     256    1024
DigitalInBufferSize        4096    1024       0   16384    4096    2048   16384   16384
DigitalOutBufferSize       1024       0       0   16384    1024     256   16384     256
```

Note that we only show the parameters with integer values here. This makes it look as if device configurations 0 and 4 are identical, as well as configurations 3 and 6.

However, if we also consider the *OtherInfoText* information, we'd see that this is single period ('.') character in configurations 0 and 3, and '1V8_Digital_Input' in configurations 4 and 6. So those configurations that look identical are indeed distinct.

11.2.4 Analog Discovery 3

A current, low-cost multi-function device; the successor to the Analog Discovery 2.

```
device ........... : 10
version .......... : 3
user_name ........ : 'Discovery3'
device_name ...... : 'Analog Discovery 3'

Configuration:                0         1         2         3         4         5
----------------------  --------  --------  --------  --------  --------  --------
TooltipText (length)       (413)     (404)     (404)     (412)     (412)     (411)
OtherInfoText                  _         _         _         _         _         _
AnalogInChannelCount           2         2         2         2         2         2
AnalogOutChannelCount          4         4         4         4         4         4
AnalogIOChannelCount           2         2         2         2         2         2
DigitalInChannelCount         16        16        16        16        16        16
DigitalOutChannelCount        16        16        16        16        16        16
DigitalIOChannelCount         16        16        16        16        16        16
AnalogInBufferSize         16384     32768      8192     16384      4096      8192
AnalogOutBufferSize        16384      4096     32768      4096      4096     16384
```

(continues on next page)

(continued from previous page)

```
DigitalInBufferSize        16384      4096      2048      32768      32768      2048
DigitalOutBufferSize        2048      2048      2048       2048      32768      2048
```

11.2.5 Digital Discovery

A current, low-cost device, similar to the Analog Discovery 2, but omitting analog inputs and outputs. On the plus side, it provides high-speed differential digital inputs, and more memory for pattern generation and capture.

```
device .......... : 4
version ......... : 2
username ........ : 'DDiscovery'
devicename ...... : 'Digital Discovery'

Configuration:                0
--------------------  --------
AnalogInChannelCount          0
AnalogOutChannelCount         0
AnalogIOChannelCount          1
DigitalInChannelCount        24
DigitalOutChannelCount       16
DigitalIOChannelCount        16
AnalogInBufferSize            0
AnalogOutBufferSize           0
DigitalInBufferSize    67108864
DigitalOutBufferSize      32768
```

11.2.6 Analog Discovery Studio

This is the spiritual successor to the earlier Electronics Explorer device.

(no data)

11.2.7 DPS3340 Discovery USB power supply

(no data)

11.2.8 Analog Discovery Pro 3x50

This device is feature-wise comparable to the Analog Discovery 2, sold since 2020. Hardware-wise, it is more high-end; it provides BNC connectors for its analog inputs and outputs.

This class of devices includes two models: the Analog Discovery Pro 3250 and the Analog Discovery Pro 3450.

The data below is for an Analog Discovery Pro 3450:

```
device ........... : 6
version .......... : 4
username ......... : 'ADP3450'
devicename ....... : 'Analog Discovery Pro 3450'

Configuration:                0         1
--------------------  --------  --------
AnalogInChannelCount          4         4
AnalogOutChannelCount         2         2
```

(continues on next page)

(continued from previous page)

```
AnalogIOChannelCount            1         1
DigitalInChannelCount          16        16
DigitalOutChannelCount         16        16
DigitalIOChannelCount          16        16
AnalogInBufferSize          32768     65536
AnalogOutBufferSize         32768      4096
DigitalInBufferSize         32768      8192
DigitalOutBufferSize        16384      1024
```

11.2.9 Analog Discovery Pro 5250

A clone of the National Instruments VB-8012 device. Works only in Windows.

(no data)

CHAPTER

TWELVE

ABOUT THE DWF C LIBRARY

Digilent provides the Digilent Waveforms library to control their line of line of electronic test and measurement devices. The library is available as a DLL on Microsoft Windows, a shared object ("so") library on Linux, and a framework on Apple's macOS. The provided library is accompanied by a C header file; together with the shared library file itself, this allows access to the functionality provided from the C and C++ programming languages.

12.1 Accessing the DWF library from Python

Most popular programming languages provide a mechanism to access functions in shared libraries. In Python, such a mechanism is provided by the `ctypes` module that is part of the standard Python library.

The *pydwf* package is a binding to the functionality provided by the DWF library, using the `ctypes` module. It makes all types and functions provided by the DWF library available for use in Python programs, wrapping them in a small set of classes that makes them easy to use.

12.2 Overview of the C API

The DWF library comes with a C header file that (for version 3.20.1) defines the 27 enumeration types and 478 function calls that together make up the DWF API. Of the 478 function calls provided, 33 are labeled *obsolete*. Their functionality is usually superseded by newer, more general functions.

The API functions are organized in 16 sub-categories, each providing access to a subset of the DWF functionality — for example, a specific type of instrument, or functions to send and receive messages using a certain protocol.

The function counts for each of the sub-categories of functionality are listed below, to give some idea of the complexity of the different areas of the API.

Table 1: DWF C API function counts (by category), version 3.20.1

C API category	pydwf equivalent	active	obsolete	total
(miscellaneous)	`DwfLibrary`	5	0	5
FDwfEnum	`DwfLibrary.deviceEnum`	11	4	15
FDwfDevice	`DwfLibrary.deviceControl`, `DwfDevice`	16	0	16
FDwfSpectrum	`DwfLibrary.spectrum`	3	0	3
FDwfAnalogIn	`DwfDevice.analogIn`	100	1	101
FDwfAnalogOut	`DwfDevice.analogOut`	58	25	83
FDwfAnalogIO	`DwfDevice.analogIO`	17	0	17
FDwfAnalogImpedance	`DwfDevice.analogImpedance`	23	0	23
FDwfDigitalIn	`DwfDevice.digitalIn`	60	2	62
FDwfDigitalOut	`DwfDevice.digitalOut`	52	1	53
FDwfDigitalIO	`DwfDevice.digitalIO`	25	0	25
FDwfDigitalUart	`DwfDevice.protocol.uart`	10	0	10
FDwfDigitalSpi	`DwfDevice.protocol.spi`	32	0	32
FDwfDigitalI2c	`DwfDevice.protocol.i2c`	14	0	14
FDwfDigitalCan	`DwfDevice.protocol.can`	7	0	7
FDwfDigitalSwd	`DwfDevice.protocol.swd`	12	0	12
TOTAL		445	33	478

From this table, it is clear that the most complex parts of the API are the AnalogIn ("oscilloscope") and AnalogOut ("waveform generator") instruments, followed by the DigitalIn ("logic analyzer") and DigitalOut ("pattern generator") instruments. About 60% of all the functions provided by the DWF library are directly related to control of these four powerful instruments.

12.3 Error handling in the C API

Each function in the C API returns an integer, indicating its success or error status. A value of 0 indicates an error, while a value of 1 indicates success.

Note:

This is different from the convention used in most C libraries, where a 0 return value indicates success.

In earlier versions of *libdwf* the return value of operations was specified to be of type *bool*, with *true* (1) indicating success and *false* (0) indicating failure. The return type was changed to *int* at some point, but the values for success and failure remained the same for reasons of backward compatibility.

In case a function returns 0, indicating some kind of failure, the C API provides two functions to inquire the reason of the failure. The *FDwfGetLastError* function returns a value of C enumeration type DWFERC (represented in Python by the enumeration type `DwfErrorCode`), indicating the cause of the last error, while function *FDwfGetLastErrorMsg* returns a string describing the error.

In *pydwf*, the low-level error reporting provided by the C API is handled by checking the return value of any C API call, and raising a `DwfLibraryError` whenever a failure is detected.

CHAPTER

THIRTEEN

EXAMPLE SCRIPTS

The Python examples can be installed locally after installing the *pydwf* package by executing the following command:

```
python -m pydwf extract-examples
```

This will create a local directory called *pydwf-examples* containing the Python examples that demonstrate many of the capabilities of the Digilent Waveforms devices and *pydwf*.

Todo: This section is currently incomplete.

Some examples that are currently missing:

- We need more examples for the `DigitalIn` and `DigitalOut` instruments.
- The four main instruments need examples for all their modes.
- We need examples using inter-instrument triggering.
- We need examples for the `AnalogImpedance` functionality.
- We need examples for the `Spectrum` functionality.
- We need examples for the `deviceEnum` and `DeviceControl` APIs.

The following examples are currently provided:

DeviceControl functionality example

DigitalDiscoveryLedBrightnessParameter.py
: Modulate the brightness of the Digital Discovery power-on LED.

  This example only works with the Digital Discovery device.

DeviceEnumeration functionality example

No example available (yet).

For now, it is recommended to have a look at the *list_devices()* function in the __main.py__ top-level module.

AnalogIn instrument examples

AnalogInSimple.py
This example demonstrates the easiest way to obtain samples from the analog input channels.

The method used by the example is useful if triggering or precise timing is not important.

AnalogInShiftScanShiftScreenDemo.py
This example demonstrates recording using the *AnalogIn* instrument, in *ScanScreen* or *ScanShift* modes.

The acquisition mode (*ScanScreen* or *ScanShift*) can be selected using a command line parameter.

AnalogInRecordMode.py
This example demonstrates acquisition using the *AnalogIn* instrument.

The script emits sinusoid signals on the first two channels of the *AnalogOut* instrument, and continuously samples the first two channels of the *AnalogIn* instrument.

This example assumes that the analog output channels are connected to the analog input channels.

Availability of the matplotlib package is assumed for plotting results.

AnalogOut instrument examples

AnalogOutShowChannelAndNodeInfo.py
This program starts by selecting the device configuration that has the largest count of analog output channels.

It then enumerates all these output channels and shows their capabilities, including all sub-nodes of each channel, and *their* capabilities.

This example is mostly interesting to show the capabilities of the rarely used third and fourth analog output channels of the Analog Discovery 2, which are only available in one device configuration. These two extra channels are essentially the V+ and V- power supply outputs.

AnalogOutSimple.py
Show simple control of the analog output channels of the *AnalogOut* instrument.

This example programs an *AnalogOut* output channel for a square output waveform, but doesn't actually start it. By configuring the channel to drive the *Initial* waveform value on its output while idle, the output voltage can be controlled directly by manipulating the channel's *amplitude* setting.

This method is useful if the output needs to change only occasionally, and if triggering or precise timing is not important.

This technique is much faster than the alternative approach of changing the channel's *offset* setting, which takes a long time to stabilize after a change due to the presence of an analog low-pass filter. The demonstrated technique can change the analog output value at a rate of several hundred times per second.

AnalogOutPlayFunction.py
Play one of the built-in analog output waveforms using the *AnalogOut* instrument.

The user can select the waveform, frequency, amplitude, offset, phase, and symmetry parameters using command line options.

AnalogOutShowFunctionSymmetry.py
Show the way the *symmetry* setting changes the behavior of the built-in analog output waveforms of the *AnalogOut* instrument.

This example assumes that the first analog output channel is connected to the first analog input channel.

Availability of the matplotlib package is assumed for plotting results.

AnalogOutAmplitudeModulationDemo.py
Show the way Amplitude Modulation (AM) changes a carrier sine wave using AnalogOut instrument.

This example assumes that the first analog output channel is connected to the first analog input channel.

Availability of the matplotlib package is assumed for plotting results.

AnalogOutPlayCustomWaveform.py
Show a custom waveform on CH1 of the *AnalogOut* instrument. The custom waveform can be given as a file containing human-readable numbers. If no filename is specified, a default custom waveform is generated.

Hook up the first analog output channel of your device to an oscilloscope to see the custom waveform.

AnalogOutContinuousPlay.py
Show either a circle or a polygon on CH1 (X) and CH2 (Y) of the *AnalogOut* instrument.

To properly appreciate this example, hook up these two channels to a second oscilloscope that is configured for X-vs-Y display mode.

AnalogOutSpinningGlobe.py
Show a spinning globe on CH1 (X) and CH2 (Y) of the *AnalogOut* instrument.

To properly appreciate this example, hook up these two channels to a second oscilloscope that is configured for X-vs-Y display mode.

AnalogIO functionality example

AnalogIO.py
This example enumerates all *AnalogIO* channels and their nodes. After that, it continuously reports the quantities associated with the "USB Monitor" channel.

Analog Impedance examples

Not yet available.

DigitalIn instrument examples

Not yet available.

DigitalOut instrument examples

DigitalOutShowStatusDuringPulsePlayback.py
Demonstrate the behavior of the *DigitalOut* instrument status during Pulse playback.

DigitalIO functionality example

DigitalIO.py
Demonstrate static monitoring and control of the digital input and output pins.

Protocol examples

ProtocolUART.py
An example showing loopback transmission and reception using the UART protocol.

ProtocolCAN.py
An example showing loopback transmission and reception of messages using the CAN bus protocol.

ProtocolSPI.py
This example demonstrates continuous readout of an ADXL345 triple-axis accelerometer using the SPI protocol.

The program expects to find the accelerometer IC attached to the proper pins. See the source code for a description on how to hook up an ADXL345 to make this work.

ProtocolI2C.py

This example demonstrates continuous readout of an ADXL345 triple-axis accelerometer using the I^2C protocol.

The program expects to find the accelerometer IC attached to the proper pins. See the source code for a description on how to hook up an ADXL345 to make this work.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z