pypixxlib Documentation

Release 1.5

VPixx Technologies

March 22, 2017

pypixxlib package includes a series of classes to define our devices. For each possible device there exists a corresponding class. Each device class has a file with corresponding classes for all the devices (ex: `viewpixx.py` has the class `VIEWPixx3D`).

If you create an object that instantiates a class for a device you have connected, all possible functions attached to the object are guaranteed to work and to be applied on the correct device, assuming the operations are permitted.

If you are programming for something more permissible or if you do not know which device will be used, you can use the `_libdpx` Wrapper Module, which does not guarantee the functions will work, but does guarantee all the functions will be called on the correct devices if you provide the appropriate commands.

CHAPTER

ONE

GETTING STARTED

Before users can start using this API, they must ensure that the required material is present. The first step is to verify that Python 2.7 is installed on the computer.

Tip: Most of the pypixxlib development has been done on Python 2.7. We are working on porting our package to Python 3.4 as well.

Installing our package is user-friendly and done using `pip`. If you do not have pip installed, please follow these instructions:

Download get-pip.py and run it:

```
python get-pip.py
```

To enable the use of pip from the command line, ensure the `Scripts` subdirectory of your Python installation is available on the system variable PATH. This is not done automatically. Alternatively, navigate to the Python27/Scripts folder.

Once there, simply install our package as follows:

```
pip install -U  PATH_TO_FILE/pypixxlib.tar.gz
```

You are now ready to use Python and combine it with VPixx Technologies high performance systems.

CHAPTER TWO

BASIC DEMOS

Our examples will showcase how easy it is to use a VPixx device with our package.

Tip: We will be using an object-oriented approach to use VPixx devices. If you are more comfortable programming using a more imperative/functional way but you still want to use Python, you can use the *Libdpx Wrapper* module, which is a simple wrapper for our ANSI C SDK.

2.1 Example 1 - Setting up a device in High-bit depth grey-scale mode.

You must first consider which device you have. In this example we will use a VIEWPixx3D.

```
from pypixxlib.viewpixx import VIEWPixx3D
# viewpixx and VIEWPixx3D would need to be replaced by the appropriate devices.
my_device = VIEWPixx3D() # Opens and initiates the device
my_device.setVideoMode('M16') # Set the right video mode
my_device.updateRegisterCache() # Update the device
```

2.2 Example 2 - Setting up a PROPixx to display in 480 Hz.

This example is only meant for the PROPixx as it is the only device able to display stimuli at 480 Hz. To do so, we limit our resolution to 1/4 of the screen to display 4 times faster. Indeed, your stimulus needs to be created as follow:

Quadrant 1	Quadrant 2
Quadrant 3	Quadrant 4

One frame at 120 Hz will represent 4 frames at 480 Hz on the PROPixx, and each of those 4 frames must be put in order in the quadrants 1, 2, 3 and 4.

```
from pypixxlib.propixx import PROPixx
my_device = PROPixx()
my_device.setDlpSequencerProgram('QUAD4X')
my_device.updateRegisterCache()
# You can now send you stimulus at 120 Hz to be displayed at 480 Hz!
```

2.3 Example 3 - Setting up a PROPixx to display in 1440 Hz.

1440 Hz is very similar to 480 Hz in the set-up process. To get 1440 Hz, we use the fact that every quadrants defined above has three channels of information (RGB). Quadrant 1's red information will define the first frame we display at 1440 Hz, Quadrant 2's red will be the 2nd, and after red we use green information, then blue. In other words: Q1R, Q2R, Q3R, Q4R, Q1G, Q2G, Q3G, Q4G, Q1B, Q2B, Q3B, Q4B. This allows us to have 12 high speed frames every normal speed frame, giving 1440 Hz.

```python
from pypixxlib.propixx import PROPixx
my_device = PROPixx()
my_device.setDlpSequencerProgram('QUAD12X')
my_device.updateRegisterCache()
# You can now send you stimulus at 120 Hz to be displayed at 1440 Hz!
```

2.4 Example 4 - Setting up a PROPixx for a 3D experiment.

In this example, we will set up the PROPixx for rear projection, ceiling mode and use of a polariser. We will also set the PROPixx so that it keeps this configuration after a power down. We will also have the PROPixx Controller send out audio stimuli.

There exists four ways to set up the polariser, depending on what you wish to achieve.

1. If the video source is set to `SWTP3D`: `my_device.setVideoSource('SWTP3D')`
2. If you are using Blue Lines to sync your 3D stimulus: `my_device.setVideoBlueLine(True)`
3. If you are using the red and blue 3D sequencer mode: `my_device.setDlpSequencerProgram('RB3D')`
4. You can manually activate the polariser,(this is called VESA Free Run mode): `my_device.setVesaFreeRun(True)`

```python
from pypixxlib.propixx import PROPixx, PROPixxCTRL
from pypixxlib._libdpx import DPxSelectDevice
my_device = PROPixx()
my_device_controller = PROPixxCTRL()


my_device.setRearProjectionMode(True)# Sets the projector to read-projection mode.
my_device.setCeilingMountMode(True) # Sets the projector to ceiling-mount mode.
my_device.setVesaFreeRun(True)  # Sets the VESA port to work with the polariser.
my_device.updateRegisterCache() # Update the new modes to the device.
my_device.setCustomStartupConfig() # The projector will remember this configuration.
# The audio is done on the controller, so the next functions use the controller object.
my_device_controller.audio.initializeCodec() # Configures initial CODEC state.
my_device_controller.audio.setLeftRightMode('mono') # Set which mode the audio will be output.
my_device_controller.audio.setAudioBuffer(0, 64) # Set the audio to start at address 0, and size 64.


# Now we must create the audio stimulus.
# This is done by simply filling a list with a sound wave.
audio_stim = []
import math
for i in range(32):
    audio_stim.append(int(32767 * math.sin(2.0 * math.pi * i / 32.0)))
my_device_controller.writeRam(0, audio_stim) # Write the audio stimulus in the ram.
my_device_controller.audio.setVolume(0.25)
my_device_controller.updateRegisterCache() # Send everything to the device.
```

```python

# We must at this point create the audio schedule
# The first parameter is the schedule onset, which we set to 0,
# meaning that the first schedule tick will occur immediately once the schedule starts.
# The second and third parameters indicate that the audio samples will be played at 40 kHz.
# The forth parameter is a schedule counter, indicating the number of ticks
# that the schedule should run before stopping.
# We'll run it for 16000 samples, which will take 16000/40000 = 400 ms.

my_device_controller.audio.setAudioSchedule(0, 40000, 'hz', 16000)
my_device_controller.audio.startScheduleLeft()
my_device_controller.updateRegisterCache()
# Close the devices
my_device.close()
my_device_controller.close()
```

2.5 Example 5 - Changing the back-light intensity on a VIEWPixx

Our VIEWPixx offer a customizable back-light level, should you wish to work with specific luminance values.

```python
from pypixxlib.viewpixx import VIEWPixxEEG
my_device = VIEWPixxEEG() # Create an instance of your device.
my_device.setBacklightIntensity(35) # Value between 0-255 for the intensity.
my_device.updateRegisterCache() # Send the value to the device.
my_device.getBacklightIntensity() # Double check our value was sent and applied
35
```

2.6 Example 6 - Recording changes in digital-in

If you are running an experiment that sends triggers over a DB-25 cable, it is possible to record those changes with a DATAPixx.

```python
from pypixxlib.datapixx import DATAPixx
my_device = DATAPixx()
din_state = my_device.din.getValue()
# Start your experiment
experiement_is_running = True
while experiement_is_running:
    old_state = din_state
    my_device.updateRegisterCache()
    din_state = my_device.din.getValue()
    if old_state is not din_state: # Something triggered.
        # Now we want to check, for example, if pin 6 triggered.
        if (old_state & 2**6) is not (din_state & 2**6):
            print 'Pin 6 triggered!'
            experiement_is_running = False
        else:
            print 'Pin 6 is in the same state as before'
# Finish your experiment
```

2.7 Example 7 - How to verify the chromacity and luminance of the colors of your device using i1Device

Most of our devices come calibrated as D65, but if you want to verify the calibration or get a specific value for a colour/brightness, you can use the i1Display and i1Pro modules.

```
from pypixxlib.i1 import I1Display # We will use an i1Display here


# Initialize the device.
my_device = I1Display()
my_device.setCurrentCalibration('RGB LED') # We will be using a device with RGB LEDs
print 'Current calibration is: ', my_device.getCurrentCalibration() # Verify we have the right calib

# Set the color/intensity you wish to measure
my_device.runMeasurement()
print 'next acquisition' # Change to the next color/intensity
my_device.runMeasurement()
print 'next acquisition' # Change to the next color/intensity
my_device.runMeasurement()

results = my_device.getAllMeasurement() # Get a list of all measurements
for i in range(len(results)):
    print results[i] # Print the different measurements

my_device.printLatestMeasurement() # If you want to print your latest measurement

my_device.close()  # Close the device.
```

These basic demo should help you use your VPixx device. All possible functions are documented, and every module defines what it has access to. If you have any question or run into any problems, please do not hesitate to contact us.

CHAPTER

THREE

DATAPIXX

class datapixx.**DATAPixx**

Bases: dpxDevice.DpxDevice, vgaOut.VgaOut, videoFeatures.VideoFeatures, threeDFeatures.ThreeDFeatures

Class Definition for the DATAPixx Device.

The bases defined below are the core subsystems attached to a DATAPixx devices. A DATAPixx device also has very specific subsystems which are used through handles. See the example bellow for more details.

Note: If you have a **Lite** version of the device, some of these handles will not be available. The usage of the handles is as follows:

```
>>> from pypixxlib.datapixx import DATAPixx
>>> my_device = DATAPixx()
>>> my_device.adc.function() # adc is the subsystem in this example.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

name
Name of the device.

adc
Handle for the control of Analog In signal. See *Analog In*.

dac
Handle for the control of Analog Out signal. See *Analog Out*.

audio
Handle for the audio controls. See *Audio Out*.

micro
Handle for the microphone controls. See *Audio In*.

din
Handle for the control of Digital In signal. See *Digital Inputs*.

dout
Handle for the control of Digital Out signal. See *Digital Out*.

subsytems
A list of the available subsystems related to your physical device.

CHAPTER

FOUR

DATAPIXX2

class datapixx.**DATAPixx2**

Bases: dpxDevice.DpxDevice, dualLinkOut.DualLinkOut, videoFeatures.VideoFeatures

Class Definition for the DATAPixx2 Device.

The bases defined below are the core subsystems attached to a DATAPixx2 devices. A DATAPixx2 device also has very specific subsystems which are used through handles. See the example bellow for more details.

Note: If you have a **Lite** version of the device, some of these handles will not be available. The usage of the handles is as follows:

```
>>> from pypixxlib.datapixx import DATAPixx2
>>> my_device = DATAPixx2()
>>> my_device.adc.function() # adc is the subsystem in this example.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

name

Name of the device.

adc

Handle f or the control of Analog In signal. See *Analog In*.

dac

Handle for the control of Analog Out signal. See *Analog Out*.

audio

Handle for the audio controls. See *Audio Out*.

micro

Handle for the microphone controls. See *Audio In*.

din

Handle for the control of Digital In signal. See *Digital Inputs*.

dout

Handle for the control of Digital Out signal. See *Digital Out*.

subsytems

A list of the available subsystems related to your physical device.

CHAPTER

FIVE

PROPIXX

class propixx.**PROPixx**

Bases: dpxDevice.DpxDevice, videoFeatures.VideoFeatures

Class Definition for the PROPixx Device.

Tip: If you want to use these functions, you must connect your PROPixx device with a USB cable. If you want to use the PROPixx Controller, see PROPixxCtrl.

```
>>> from pypixxlib.propixx import PROPixx
>>> my_device = PROPixx()
>>> my_device.function() # A PROPixx has no sub-systems, only function and bases.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

get3dCrosstalk()

Get 3D crosstalk (0-1) which is being subtracted from stereoscopic stimuli

Warning: This only works with RB3D mode and requires revision 6 of the PROPixx.

Returns A double value for the 3D Crosstalk.

getDlpSequencerProgram()

Get PROPixx DLP Sequencer program.

This method allows the user to set the video mode of the PROPixx.

Returns

Any of the following predefined constants.

- **RGB**: Default RGB
- **RB3D**: R/B channels drive grayscale 3D
- **RGB240**: Only show the frame for 1/2 a 120 Hz frame duration.
- **RGB180**: Only show the frame for 2/3 of a 120 Hz frame duration.
- **QUAD4X**: Display quadrants are projected at 4x refresh rate.
- **QUAD12X**: Display quadrants are projected at 12x refresh rate with grayscales.
- **GREY3X**: Converts 640x1080@360Hz RGB to 1920x1080@720Hz Grayscale with blank frames.

Return type String

getFanPwm()

Gets the PROPixx PWM.

Returns **pwm** – PWM value.

Return type float

`getFanSpeed`(*fan_nbr*)
Gets the speed of one of the PROPixx fans.

Parameters **fan_nbr** (*int*) – number of the queried fan.

Returns **speed** – current for a given fan.

Return type float

`getLedCurrent`(*led_nbr*)
Gets the current for a given LED.

Parameters **led_nbr** (*int*) – number of the queried LED.

Returns **current** – current for a given LED.

Return type float

`getTemperature`(*component_temperature*)
Gets the temperature for a given power supply.

Parameters **component_temperature** (*int*) – power supply for which the voltage is queried. A valid argument is one of the following:

- PPX_TEMP_LED_RED,
- PPX_TEMP_LED_GRN,
- PPX_TEMP_LED_BLU,
- PPX_TEMP_LED_ALT,
- PPX_TEMP_DMD,
- PPX_TEMP_POWER_BOARD,
- PPX_TEMP_LED_POWER_BOARD,
- PPX_TEMP_RX_DVI,
- PPX_TEMP_FPGA,
- PPX_TEMP_FPGA2,
- PPX_TEMP_VOLTAGE_MONITOR

Returns **temperature** – temperature for a given power supply.

Return type float

`getVoltageMonitor`(*power_supply_type*)
Gets the voltage for a given power supply.

Parameters **power_supply_type** (*int*) – Power supply for which the voltage is queried. Valid arguments are the following:

- PPX_POWER_5V,
- PPX_POWER_2P5V,
- PPX_POWER_1P8V,
- PPX_POWER_1P5V,
- PPX_POWER_1P1V,

- PPX_POWER_1V,
- PPX_POWER_12V,
- PPX_POWER_VCC

Returns **voltage** – voltage for a given power supply.

Return type float

isCeilingMountEnabled()

Gets the ceiling mount mode state.

Returns True if enabled, False if disabled.

Return type bool

isLampLEDMode()

Returns non-0 if PROPixx lamp LED is enabled.

Warning: This requires revision 12 of the PROPixx.

See also:

setLampLED

isQuietMode()

Returns non-0 if PROPixx quiet mode is enabled.

Warning: This requires revision 19 of the PROPixx.

See also:

setQuietMode

isRearProjectionEnabled()

Gets the rear projection mode state.

Returns True if enabled, False if disabled.

Return type bool

isSequencerEnabled()

Gets the sequencer mode state.

Returns True if enabled, False if disabled.

Return type bool

isSleepMode()

Returns True if PROPixx sleep mode is enabled.

Warning: This requires revision 12 of the PROPixx.

See also:

setSleepMode

isVesaFreeRun()

Returns non-0 if PROPixx VESA 3D output is enabled.

`set3dCrosstalk`(*crosstalk*)
Set 3D crosstalk (0-1) which should be subtracted from stereoscopic stimuli.

> **Warning:** This only works with RB3D mode and requires revision 6 of the PROPixx.

Parameters **crosstalk** (*double*) – A value between 0 and 1 which represents the 3d crosstalk.

`setCeilingMountMode`(*enable*)
Sets the ceiling mount mode.

This method allows the user to turn the ceiling mount mode on or off. When enabled, the image will be displayed up side down. This allows the user to install the PROPixx on the ceiling. when this mode is disabled, the image will be displayed normally.

Parameters **enable** (*bool*) – Ceiling mount mode.

`setDlpSequencerProgram`(*program*)
Sets the PROPixx DLP Sequencer program.

Only available for PROPixx Revision 6 and higher.

Parameters **program** (*string*) – Any of the following predefined constants.

- **RGB**: Default RGB
- **RB3D**: R/B channels drive grayscale 3D
- **RGB240**: Only show the frame for 1/2 a 120 Hz frame duration.
- **RGB180**: Only show the frame for 2/3 of a 120 Hz frame duration.
- **QUAD4X**: Display quadrants are projected at 4x refresh rate.
- **QUAD12X**: Display quadrants are projected at 12x refresh rate with grayscales.
- **GREY3X**: Converts 640x1080@360Hz RGB to 1920x1080@720Hz Grayscale with blank frames.

`setLampLED`(*enable*)
Enables or disables the lamp LED.

> **Warning:** This requires revision 12 of the PROPixx.

Parameters **enable** (*bool*) – Set to true to enable the lamp LED.

See also:

`isLampLEDMode`

`setQuietMode`(*enable*)
Enables or disables the quiet mode.

Enabling this mode reduces the noise generated by the PROPixx by lowering the speed of the fans. It should be kept enabled unless reducing the noise is essential.

> **Warning:** This requires revision 19 of the PROPixx.

Parameters **enable** (*bool*) – Set to true to enable quiet mode.

See also:

`isQuietMode`

`setRearProjectionMode`(*enable*)
Sets the rear projection mode.

This method allows the user to turn the rear projection mode on or off. When enabled, the image will be displayed in mirrored view, allowing the user to install the PROPixx behind a projection screen.

Parameters **enable** (*bool*) – rear projection mode.

`setSleepMode`(*enable*)
Enables or disables the sleep mode.

Enabling this mode turns the PROPixx off. Disabling it will turn the PROPixx on.

Warning: This requires revision 12 of the PROPixx.

Parameters **enable** (*bool*) – Set to true to enable sleep mode.

See also:

`isSleepMode`

`setVesaFreeRun`(*enable*)
Enables or disables the Vesa output to work with the polariser.

Parameters **enable** (*bool*) – Set to true to enable VesaFreeRun.

CHAPTER

SIX

PROPIXX CONTROLLER

class `propixx.`**`PROPixxCTRL`**

Bases: `dpxDevice.DpxDevice`, `dualLinkOut.DualLinkOut`, `videoFeatures.VideoFeatures`

Class Definition for the PROPixx Controller Device.

Tip: If you want to use these functions, you must connect your PROPixx controller device with a USB cable. If you want to use the PROPixx, see `PROPixx`.

```
>>> from pypixxlib.propixx import PROPixxCTRL
>>> my_device = PROPixxCTRL()
>>> my_device.adc.function() # adc is the subsystem in this example.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

`name`
Name of the device.

`adc`
Handle for the control of Analog In signal. See *Analog In*.

`dac`
Handle for the control of Analog Out signal. See *Analog Out*.

`audio`
Handle for the audio controls. See *Audio Out*.

`micro`
Handle for the microphone controls. See *Audio In*.

`din`
Handle for the control of Digital In signal. See *Digital Inputs*.

`dout`
Handle for the control of Digital Out signal. See *Digital Out*.

`subsytems`
A list of the available subsystems related to your physical device.

CHAPTER SEVEN

VIEWPIXX

class `viewpixx.`**`VIEWPixx`**

Bases: `dpxDevice.DpxDevice`, `dualLinkOut.DualLinkOut`, `scanningBackLight.ScanningBackLight`, `videoFeatures.VideoFeatures`, `threeDFeatures.ThreeDFeatures`

Class Definition for the VIEWPixx Device.

The bases defined bellow are the core subsystems attached to a VIEWPixx devices. A VIEWPixx device also has very specific subsystems which are used through handles. See the example bellow for more details.

Note: If you have a **Lite** version of the device, some of these handles will not be available. The usage of the handles is as follows:

```
>>> from pypixxlib.viewpixx import VIEWPixx
>>> my_device = VIEWPixx()
>>> my_device.adc.function() # adc is the subsystem in this example.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

`name`
Name of the device.

`adc`
Handle for the control of Analog In signal. See *Analog In*.

`dac`
Handle for the control of Analog Out signal. See *Analog Out*.

`audio`
Handle for the audio controls. See *Audio Out*.

`micro`
Handle for the microphone controls. See *Audio In*.

`din`
Handle for the control of Digital In signal. See *Digital Inputs*.

`dout`
Handle for the control of Digital Out signal. See *Digital Out*.

`subsytems`
A list of the available subsystems related to your physical device.

`isPixelPolarityInversionEnabled``()`
Verifies that 3D pixel polarity inversion is enabled.

Returns True if the pixel polarity inversion is enabled, False otherwise.

Return type Bool

See also:

`setPixelPolarityInversion`

`setPixelPolarityInversion`(*enable*)

Sets the 3D pixel polarity inversion.

Liquid crystal displays can exhibit an artifact when presenting 2 static images on alternating video frames, such as with frame-sequential 3D. The origin of this artifact is related to LCD pixel polarity inversion. The optical transmission of a liquid crystal cell varies with the magnitude of the voltage applied to the cell. Liquid crystal cells are designed to be driven by an AC voltage with little or no DC component. As such, the cell drivers alternate the polarity of the cell's driving voltage on alternate video frames. The cell will see no net DC driving voltage, as long as the pixel is programmed to the same intensity on even and odd video frames. Small differences in a pixel's even and odd frame luminance tend to leave the cell unaffected, and large differences in even and odd frame luminance for short periods of time (10-20 frames?) also do not seem to affect the cell; however, large differences in luminance for a longer period of time will cause a DC buildup in the pixel's liquid crystal cell. This can result in the pixel not showing the programmed luminance correctly, and can also cause the pixel to "stick" for several seconds after the image has been removed, causing an after-image on the display. VPixx Technologies has developed a strategy for keeping the pixel cells DC balanced. Instead of alternating the cell driving voltage on every video frame, we can alternate the voltage only on every second frame. This feature is enabled by calling the routine DPxEnableVidLcd3D60Hz(). Call this routine before presenting static or slowly-moving 3D images, or when presenting 60Hz flickering stimuli. Be sure to call DPxDisableVidLcd3D60Hz() afterwards to return to normal pixel driving. Note that this feature is only supported on the VIEWPixx/3D when running with a refresh rate of 120Hz.

Parameters **enable** (*Bool*) – True to enable the pixel polarity inversion, False to disable it.

See also:

`isPixelPolarityInversionEnabled`

CHAPTER

EIGHT

VIEWPIXX3D

class viewpixx.**VIEWPixx3D**

Bases: dpxDevice.DpxDevice, dualLinkOut.DualLinkOut, scanningBackLight.ScanningBackLight, videoFeatures.VideoFeatures, threeDFeatures.ThreeDFeatures

Class Definition for the VIEWPixx3D Device.

Note: If you have a **Lite** version of the device, some of these handles will not be available. The usage of the handles is as follow:

```
>>> from pypixxlib.viewpixx import VIEWPixx3D
>>> my_device = VIEWPixx3D()
>>> my_device.adc.function() # adc is the subsystem in this example.
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

name

Name of the device.

adc

Handle for the control of Analog In signal. See *Analog In*.

dac

Handle for the control of Analog Out signal. See *Analog Out*.

audio

Handle for the audio controls. See *Audio Out*.

micro

Handle for the microphone controls. See *Audio In*.

din

Handle for the control of Digital In signal. See *Digital Inputs*.

dout

Handle for the control of Digital Out signal. See *Digital Out*.

subsytems

A list of the available subsystems related to your physical device.

isPixelPolarityInversionEnabled()

Verifies that 3D pixel polarity inversion is enabled.

Returns True if the pixel polarity inversion is enabled, False otherwise.

Return type Bool

See also:

setPixelPolarityInversion

setPixelPolarityInversion(*enable*)

Sets the 3D pixel polarity inversion.

Liquid crystal displays can exhibit an artifact when presenting 2 static images on alternating video frames, such as with frame-sequential 3D. The origin of this artifact is related to LCD pixel polarity inversion. The optical transmission of a liquid crystal cell varies with the magnitude of the voltage applied to the cell. Liquid crystal cells are designed to be driven by an AC voltage with little or no DC component. As such, the cell drivers alternate the polarity of the cell's driving voltage on alternate video frames. The cell will see no net DC driving voltage, as long as the pixel is programmed to the same intensity on even and odd video frames. Small differences in a pixel's even and odd frame luminance tend to leave the cell unaffected, and large differences in even and odd frame luminance for short periods of time (10-20 frames?) also do not seem to affect the cell; however, large differences in luminance for a longer period of time will cause a DC buildup in the pixel's liquid crystal cell. This can result in the pixel not showing the programmed luminance correctly, and can also cause the pixel to "stick" for several seconds after the image has been removed, causing an after-image on the display. VPixx Technologies has developed a strategy for keeping the pixel cells DC balanced. Instead of alternating the cell driving voltage on every video frame, we can alternate the voltage only on every second frame. This feature is enabled by calling the routine DPxEnableVidLcd3D60Hz(). Call this routine before presenting static or slowly-moving 3D images, or when presenting 60Hz flickering stimuli. Be sure to call DPxDisableVidLcd3D60Hz() afterwards to return to normal pixel driving. Note that this feature is only supported on the VIEWPixx/3D when running with a refresh rate of 120Hz.

Parameters **enable** (*Bool*) – True to enable the pixel polarity inversion, False to disable it.

See also:

`isPixelPolarityInversionEnabled`

CHAPTER

NINE

VIEWPIXXEEG

class viewpixx.**VIEWPixxEEG**

Bases: dpxDevice.DpxDevice, scanningBackLight.ScanningBackLight, videoFeatures.VideoFeatures

Class Definition for the VIEWPixxEEG Device.

```
>>> from pypixxlib.viewpixx import VIEWPixxEEG
>>> my_device = VIEWPixxEEG()
>>> my_device.BaseFunction() # A base function from one of the class' bases.
```

CHAPTER

TEN

ANALOG IN

class `analogIn.`**`AnalogIn`**

Bases: `schedule.Schedule`

Class which contains the ADC features.

These functions should only be called from your device's `adc` parameter: `my_device.adc.function()`. The functions below are used to configure and use Analog In of your VPixx Device.

`getBaseAddress``()`

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize()

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBufferSize``()`

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getChannelRange``(`*channel*`)`

Gets the range of a channel.

Returns Range of the channel. The returned value is a tuple with the minimum range followed by the maximum range. Both values are integers.

Return type tuple

`getChannelReference``(`*channel*`)`

Gets the reference associated with a channel.

Parameters **channel** (*int*) – channel to query.

Returns

one of the following predefined constants.

- gnd : Referenced to ground.
- diff : Referenced to adjacent analog input.
- ref0 : Referenced to REF0 analog input.

- ref1 : Referenced to REF1 analog input.

Return type String

See also:

`setChannelReference`

getChannelValue(*channel*)
Gets the current value of a channel.

This method allows the user to know the value of a given channel. The return value is a 16-bit 2's complement signed number. Value can be obtained for channels 0 to 17.

Returns value of the channel.

Return type int

getChannelVoltage(*channel*)
Gets the current voltage of a channel.

Returns voltage of the channel.

Return type float

getNbrOfChannel()
Gets the number of channels available.

Returns Number of channels.

Return type int

getScheduleCount()
Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleCount`, `setScheduleCountDown`

getScheduleOnset()
Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleRate()
Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleUnit()
Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleRate`, `setScheduleRate`

`getWriteAddress``()`
Gets the Ram buffer write address.

This method allows the user to get the RAM buffer write address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`isChannelRamBuffering``(`*channel*`)`
Verifies the ram buffering mode.

This method allows the user to know if a ram buffering is enabled for a given channel.

Parameters **channel** (*int*) – Channel number to query.

Returns True if the given channel is buffered, False otherwise.

Return type Bool

See also:

`setChannelRamBuffering`

`isCountDownEnabled``()`
Verifies the schedule count down mode.

Returns True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCount`, `stopSchedule`, `setScheduleCountDown`

`isFreeRunEnabled``()`
Verifies the continuous acquisition mode state.

Returns True if continuous acquisition is activated, False otherwise.

Return type Bool

See also:

`setFreeRunMode`

`isHardwareCalibration``()`
Verifies the hardware calibration mode.

Returns True if hardware calibration is enabled, False otherwise.

Return type Bool

See also:

`setHardwareCalibration`

`isLogTimetagsEnabled``()`
Verifies if the timetag mode is enabled.

Returns True if timetag mode is enabled, False otherwise.

Return type Bool

See also:

`setLogTimetags`

`isLoopbackEnabled``()`

Verifies the digital inputs and outputs loop back mode.

Returns True if transitions are being stabilized, False otherwise.

Return type Bool

See also:

`setLoopback`

`isScheduleRunning``()`

Verifies if a schedule is currently running on the subsystem.

Returns True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startSchedule`, `stopSchedule`

`setBaseAddress``(`*address*`)`

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`setBufferSize``(`*buffer_size*`)`

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use `setBaseAddress()`. The given size is in bytes and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

`setChannelRamBuffering``(`*enable*, *channel=None*`)`

Sets the ram buffering mode.

This method allows the user to enable or disable ram buffering on a given channel. When enabled, a given channel is buffered in ram. Enabling RAM buffering can only be done for channels 0 to 15.

Parameters

- **enable** (*Bool*) – True if hardware calibration is enabled, False otherwise.
- **channel** (*int*) – Channel number to enable or disable. Passing None will set all channels.

See also:

`isChannelRamBuffering`

`setChannelReference``(`*channel*, *reference*`)`

Sets a reference to a channel.

This method allows the user to enable or disable ram buffering on a given channel. When enabled, a given channel is buffered in ram. Enabling RAM buffering can only be done for channels 0 to 15.

Parameters

- **channel** (*int*) – Channel number to associate with a reference.
- **reference** (*String*) – Valid argument is one of the following predefined constants.
  - gnd : Referenced to ground.
  - diff : Referenced to adjacent analog input.
  - ref0 : Referenced to REF0 analog input.
  - ref1 : Referenced to REF1 analog input.

See also:

`getChannelReference`

`setFreeRunMode`(*enable*)

Sets the continuous acquisition mode.

This method allows the user to enable or disable the continuous acquisition (free running) mode. When enabled, the ADCs convert continuously. Doing so can add up to 4 microseconds random latency to scheduled samples. When disabled, the ADCs only convert on schedule ticks. This can be used for microsecond-precise sampling.

Parameters **enable** (*Bool*) – True if loopback is enabled, False otherwise.

See also:

`isFreeRunEnabled`

`setHardwareCalibration`(*enable*)

Sets the hardware calibration mode.

This method allows the user to enable or disable the hardware calibration mode. When enabled, the ADC data bypasses the hardware calibration. When disabled, the ADC data passes by the hardware calibration.

Parameters **enable** (*Bool*) – True if hardware calibration is enabled, False otherwise.

See also:

`isHardwareCalibration`

`setLogTimetags`(*enable*)

Sets the timetag mode on the data samples.

This method allows the user to enable or disable the timetag on acquired data. When enabled, each buffered sample is preceded with a 64-bit nanosecond timetag. When disabled, buffered data has no timetags.

Parameters **enable** (*Bool*) – True to activate the timetag mode, False otherwise.

See also:

`isLogTimetagsEnabled`

`setLoopback`(*enable*)

Sets the digital inputs and outputs loopback mode.

This method allows the user to enable or disable the loopback between digital output ports and digital inputs. When enabled, the digital outputs send their data to the digital inputs. When disabled, the digital inputs will not get the digital outputs data.

Parameters **enable** (*Bool*) – True if loopback is enabled, False otherwise.

See also:

`isLoopbackEnabled`

`setScheduleCount`(*count*)

Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of samples. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCount`, `setScheduleCountDown`

`setScheduleCountDown`(*enable*)

Sets the schedule count down mode.

This method allows the user to enable or disable the count down on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if count down is enabled, False otherwise.

See also:

`setScheduleCount`, `stopSchedule`, `isCountDownEnabled`

`setScheduleOnset`(*onset*)

Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal to or greater than 0.

`setScheduleRate`(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than 0.
- **unit** (*String*) – Any of the following predefined constants:
  - hz : samples per second, maximum 200 kHz.
  - video : samples per video frame, maximum 200 kHz.
  - nano : sample period in nanoseconds, minimum 5000 ns.

`setWriteAddress`(*address*)

Sets the Ram buffer write address.

This method allows the user to set the RAM buffer write address used in schedules. This address is used by the schedule to know where the data should be first written to. The schedule will then write the following data to the address following the RAM buffer write address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`startSchedule``()`

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

`stopSchedule``()`

Stops the active schedule for a given subsystem.

Depending on how the Schedules are configured, it may not be necessary to call this method. When a schedule is using a count down, it is not required to stop the schedule.

See also:

`startSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

CHAPTER

ELEVEN

ANALOG OUT

class `analogOut.`**`AnalogOut`**

Bases: `schedule.Schedule`

Class which implements the DAC features.

It contains all the necessary methods to use the analog outputs of a VPixx device.

`getBaseAddress``()`

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize()

Returns Any value in a range of 0 up to the RAM size.

Return type int

See also:

`getReadAddress`, `setReadAddress`, `setBaseAddress`

`getBufferSize``()`

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

See also:

`setBufferSize`

`getChannelRange``(`*channel*`)`

Gets the range of a channel.

Returns Range of the channel. The returned value is a tuple with the minimum range followed by the maximum range. Both values are integers.

Return type tuple

`getChannelValue``(`*channel*`)`

Gets the current value of a channel.

This method allows the user to know the value of a given channel. The return value is a 16-bit 2's complement signed number. Value can be obtained for channels 0 to 17.

Returns value of the channel.

Return type int

See also:

`setChannelValue`

getChannelVoltage(*channel*)
Gets the current voltage of a channel.

Returns voltage of the channel.

Return type float

See also:

`setChannelVoltage`

getNbrOfChannel()
Gets the number of channels available.

Returns Number of channels.

Return type int

getReadAddress()
Gets the Ram buffer read address.

This method allows the user to get the RAM buffer read address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

See also:

`setReadAddress`, `getBaseAddress`

getScheduleCount()
Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleCount`, `setScheduleCountDown`

getScheduleOnset()
Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleUnit`, `getScheduleRate`, `setScheduleRate`

getScheduleRate()
Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleUnit`, `setScheduleRate`, `getScheduleOnset`

`getScheduleUnit``()`

Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleRate`, `setScheduleRate`

`isChannelRamBuffering``(`*channel*`)`

Verifies the ram buffering mode.

This method allows the user to know if a ram buffering is enabled for a given channel.

Parameters **channel** (*int*) – Channel number to query.

Returns True if the given channel is buffered, False otherwise.

Return type Bool

See also:

`setChannelRamBuffering`

`isCountDownEnabled``()`

Verifies the schedule count down mode.

Returns **enable** – True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCount`, `stopSchedule`, `setScheduleCountDown`

`isHardwareCalibration``()`

Verifies the hardware calibration mode.

Returns True if hardware calibration is enabled, False otherwise.

Return type Bool

See also:

`setHardwareCalibration`

`isScheduleRunning``()`

Verifies if a schedule is currently running on the subsystem.

Returns **schedule_running** – True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startSchedule`, `stopSchedule`, `getScheduleRunningState()`

`setBaseAddress`(*address*)
Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

See also:

`getReadAddress`, `setReadAddress`, `getBaseAddress`

`setBufferSize`(*buffer_size*)
Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBaseAddress(). The given size is in bytes and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

See also:

`getBufferSize`

`setChannelRamBuffering`(*enable*, *channel=None*)
Sets the ram buffering mode.

This method allows the user to enable or disable ram buffering on a given channel. When enabled, a given channel is buffered in ram. Enabling RAM buffering can only be done for channels 0 to 15.

Parameters

- **enable** (*Bool*) – True if hardware calibration is enabled, False otherwise.
- **channel** (*int*) – Channel number to enable or disable. Passing None will set all channels.

See also:

`isChannelRamBuffering`

`setChannelValue`(*value*, *channel*)
Sets the current value of a channel.

This method allows the user to modify the output value of a given channel.

Parameters

- **value** (*int*) – Value of the channel. It is a 16-bit 2's complement signed number.
- **channel** (*int*) – Channel number.

See also:

`getChannelValue`

`setChannelVoltage`(*voltage*, *channel*)
Sets the current Voltage of a channel.

This method allows the user to modify the output voltage of a given channel. The voltage is +-10V for channels 0 and 1, and +-5V for channels 2 and 3.

Parameters

- **voltage** (*float*) – Value of the channel.
- **channel** (*int*) – Channel number.

See also:

`getChannelVoltage`

`setHardwareCalibration`(*enable*)
Sets the hardware calibration mode.

This method allows the user to enable or disable the hardware calibration mode. When enabled, the ADC data bypasses the hardware calibration. When disabled, the ADC data passes by the hardware calibration.

Parameters **enable** (*Bool*) – True if hardware calibration is enabled, False otherwise.

See also:

`isHardwareCalibration`

`setReadAddress`(*address*)
Sets the Ram buffer read address.

This method allows the user to set the RAM buffer read address used in schedules. This address is used by the schedule to know where the data should be first read from. The schedule will then read the following data to the address following the RAM buffer read address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

See also:

`getReadAddress`, `getBaseAddress`

`setScheduleCount`(*count*)
Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of samples. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCount`, `setScheduleCountDown`

`setScheduleCountDown`(*enable*)
Sets the schedule count down mode.

This method allows the user to enable or disable the count down on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if count down is enabled, False otherwise.

See also:

`setScheduleCount`, `stopSchedule`, `isCountDownEnabled`

`setScheduleOnset`(*onset*)
Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method doesn’t need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal to or greater than 0.

See also:

`getScheduleUnit`, `getScheduleRate`, `setScheduleRate`

setScheduleRate(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate. If no delay is required, this method doesn't need to be used.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than 0.
- **unit** (*str*) – hz : samples per second, maximum 1 MHz. video : samples per video frame, maximum 1 MHz. nano : sample period in nanoseconds, minimum 1000 ns.

See also:

`getScheduleUnit`, `getScheduleRate`, `getScheduleOnset`

startSchedule()

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopSchedule`, `setReadAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

stopSchedule()

Stops the active schedule for a given subsystem.

Depending on how the Schedules are configured, it may not be necessary to call this method. When a schedule is using a count down, it is not required to stop the schedule.

See also:

`startSchedule`, `setReadAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

CHAPTER

TWELVE

AUDIO IN

class `audioIn.`**`AudioIn`**

Bases: `schedule.Schedule`

Class which implements the Microphone features.

Any device which has Audio IN should instantiate this class. It contains all the necessary methods to use the audio inputs of a VPIxx device.

`getBaseAddress``()`

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize()

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBufferSize``()`

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getGroupDelay``(`*sample_rate*`)`

Gets the CODEC Audio OUT group delay in seconds.

Returns delay in seconds.

Return type float

`getMicSource``(`*dBUnits=0*`)`

Gets the source and the gain of the microphone input.

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Returns A list containing the [gain value, microphone source]

Low-level C definition `int DPxGetMicSource(int DBUnits)`

`getScheduleBufferMode``()`

Gets the microphone Left/Right configuration mode.

This method allows the user to set the microphone to one of the mono, left, right or stereo mode.

Returns

Any of the following predefined constants.

- **mono** : Mono data is written to the schedule buffer. The average of Left/Right CODEC data.
- **left** : Left data is written to the schedule buffer.
- **right** : Right data is written to the schedule buffer.
- **stereo** : Left and Right data are both written to the schedule buffer.

Return type String

See also:

`setScheduleBufferMode`

getScheduleCount()

Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleCount`, `setScheduleCountDown`

getScheduleOnset()

Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleRate()

Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleUnit()

Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleRate`, `setScheduleRate`

getValueLeft()

Gets the current value of the left channel.

This method allows the user to get the 16-bit 2's complement signed value for left MIC channel.

Returns 16-bit 2's complement signed value.

Return type int

getValueRight()

Gets the current value of the right channel.

This method allows the user to get the 16-bit 2's complement signed value for right MIC channel.

Returns 16-bit 2's complement signed value.

Return type int

getWriteAddress()

Gets the Ram buffer write address.

This method allows the user to get the RAM buffer write address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

isCountDownEnabled()

Verifies the schedule count down mode.

Returns True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCount`, `stopSchedule`, `setScheduleCountDown`

isLoopbackEnabled()

Verifies the digital inputs and outputs loopback mode.

Returns **enable** – True if transitions are being stabilized, False otherwise.

Return type Bool

See also:

`setLoopback`

isScheduleRunning()

Verifies if a schedule is currently running on the subsystem.

Returns True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startSchedule`, `stopSchedule`, `getScheduleRunningState`

setBaseAddress(*address*)

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

setBufferSize(*buffer_size*)

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBaseAddress() The given size is in byte and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

`setLoopback`(*enable*)

Sets the digital inputs and outputs loopback mode.

This method allows the user to enable or disable the loopback between digital output ports and digital inputs. When enabled, the digital outputs send their data to the digital inputs. When disabled, the digital inputs will not get the digital outputs data.

Parameters **enable** (*Bool*) – True if loopback is enabled, False otherwise.

See also:

`isLoopbackEnabled`

`setMicSource`(*source*, *gain*, *dBUnits=0*)

Sets the source for the MIC

Select the source of the microphone input. Typical gain values would be around 100 for a microphone input, and probably 1 for line-level input.

Parameters

- **source** (*int*) – One of the following:
  - `MIC`: Microphone level input.
  - `LINE`: Line level audio input.
- **gain** (*int*) – The gain can take the following values depending on the scale:
  - linear scale : [1, 1000]
  - dB scale : [0, 60] dB
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

`setScheduleBufferMode`(*mode*)

Sets the schedule buffer storing mode.

This method allows the user to configure how microphone left and right channels are stored to the schedule buffer.

Parameters **mode** (*str*) – Any of the following predefined constants.

- **mono** : Mono data is written to the schedule buffer. The average of Left/Right CODEC data.
- **left** : Left data is written to the schedule buffer.
- **right** : Right data is written to the schedule buffer.
- **stereo** : Left and Right data are both written to the schedule buffer.

See also:

`getScheduleBufferMode`

`setScheduleCount`(*count*)

Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of samples. In this case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCount`, `setScheduleCountDown`

`setScheduleCountDown`(*enable*)
Sets the schedule count down mode.

This method allows the user to enable or disable the countdown on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling `stopSchedule()`.

Parameters **enable** (*Bool*) – True if countdown is enabled, False otherwise.

See also:

`setScheduleCount`, `stopSchedule`, `isCountDownEnabled`

`setScheduleOnset`(*onset*)
Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method doesn't need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal to or greater than 0.

`setScheduleRate`(*rate*, *unit='hz'*)
Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than 0.
- **unit** (*str*) – Any of the following predefined constants.
  - **hz** : samples per second, maximum 102.4 kHz.
  - **video** : samples per video frame, maximum 102.4 kHz.
  - **nano** : sample period in nanoseconds, minimum 9750 ns.

`setWriteAddress`(*address*)
Sets the Ram buffer write address.

This method allows the user to set the RAM buffer write address used in schedules. This address is used by the schedule to know where the data should be first written to. The schedule will then write the following data to the address following the RAM buffer write address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`startSchedule`()
Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

`stopSchedule``()`

Stops the active schedule for a given subsystem.

Depending on how the Schedules are configured, it may not be necessary to call this method. When a schedule is using a countdown, it is not required to stop the schedule.

See also:

`startSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

CHAPTER

THIRTEEN

AUDIO OUT

class `audioOut.`**`AudioOut`**

Bases: `schedule.Schedule`

Class which implements the AUDIO features.

Any device which has Audio OUT should implement this class. It contains all the necessary methods to use the audio outputs of a VPIxx device.

`beep`(*volume=0.5*, *time=1*, *frequency=40000*)

Beep bop beep bop bop beep beep beep.

Parameters

- **volume** (*float*) – Value between 0 and 1 of the desired volume.
- **time** (*int*) – Time factor for the beep duration.

`getBaseAddressLeft`()

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize()

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBaseAddressRight`()

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize().

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBufferSizeLeft`()

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBufferSizeRight`()

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getCodecSpeakerLeftVolume`(*unit*)

Gets the current speaker volume of the left channel.

Parameters **unit** (*float*) – 0 for linear or 1 for dB.

Returns Range 0 to 1.

Return type float

`getCodecSpeakerRightVolume`(*unit=0*)

Gets the current speaker volume of the right channel.

Parameters **unit** (*float*) – 0 for linear or 1 for dB.

Returns Range 0 to 1.

Return type float

`getCodecSpeakerVolume`(*unit*)

Gets the current speaker volume of the right and left channel.

Parameters **unit** (*float*) – 0 for linear or 1 for dB.

Returns A tuple containing floats: [left Speaker Volume, Right speaker Volume]

`getCodecVolume`(*unit=0*)

Gets the current codec volume of the right channel.

Parameters **unit** (*int*) – Set non-zero to return the gain in dB. Defaults to 0.

Returns Range 0 to 1.

Return type float

`getCodecVolumeLeft`(*unit=0*)

Gets the current codec volume of the left channel.

Parameters **unit** (*int*) – Set non-zero to return the gain in dB. Defaults to 0.

Returns Range 0 to 1.

Return type float

`getCodecVolumeRight`(*unit=0*)

Gets the current codec volume of the right channel.

Parameters **unit** (*int*) – Set non-zero to return the gain in dB. Defaults to 0.

Returns Range 0 to 1.

Return type float

`getGroupDelay`(*sample_rate*)

Gets the CODEC Audio OUT group delay in seconds.

Returns delay in seconds.

Return type float

`getLeftRightMode`()

Gets the audio schedule update mode.

Returns

Any of the following predefined constants.

- **mono** : Left schedule data goes to left and right channels.
- **left** : Each schedule data goes to left channel only.
- **right** : Each schedule data goes to right channel only.
- **stereo1** : Pairs of Left data are copied to left/right channels.
- **stereo2** : Left data goes to left channel, Right data goes to right.

Return type String

See also:

`setScheduleBufferMode`

`getReadAddressLeft``()`

Gets the Ram buffer read address.

This method allows the user to get the RAM buffer read address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getReadAddressRight``()`

Gets the Ram buffer read address.

This method allows the user to get the RAM buffer read address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getScheduleCountLeft``()`

Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal or greater to 0.

Return type int

See also:

`setScheduleCountLeft`, `setScheduleCountDownLeft`

`getScheduleCountRight``()`

Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal or greater to 0.

Return type int

See also:

`setScheduleCountRight`, `setScheduleCountDownRight`

`getScheduleOnsetLeft``()`

Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal or greater to 0.

Return type int

getScheduleOnsetRight()

Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal or greater to 0.

Return type int

getScheduleRateLeft()

Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal or greater to 0.

Return type int

getScheduleRateRight()

Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal or greater to 0.

Return type int

getScheduleRunningStateLeft()

Gets the schedule state for the subsystem.

Returns **schedule_state** – "running" if a schedule is currently running, "stopped" otherwise.

Return type str

See also:

`startScheduleLeft`, `stopScheduleLeft`, `isScheduleRunningLeft`

getScheduleRunningStateRight()

Gets the schedule state for the subsystem.

Returns **schedule_state** – "running" if a schedule is currently running, "stopped" otherwise.

Return type str

See also:

`startScheduleRight`, `stopScheduleRight`, `isScheduleRunningRight`

getScheduleUnitLeft()

Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal or greater to 0.

Return type int

See also:

`getScheduleRateLeft`, `setScheduleRateLeft`

getScheduleUnitRight()

Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal or greater to 0.

Return type int

See also:

`getScheduleRateRight`, `setScheduleRateRight`

getVolume()

Gets volume for both Left/Right audio channels

Returns A tuple containing floats: [left Speaker Volume, Right speaker Volume]

getVolumeLeft()

Get volume for the Left audio output channel, range 0-1

getVolumeRight()

Get volume for the Right audio output channel, range 0-1

initializeCodec()

This needs to be called once to configure initial audio CODEC state.

isCountDownEnabledLeft()

Verifies the schedule countdown mode.

Returns **enable** – True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCountLeft`, `stopScheduleLeft`, `setScheduleCountDownLeft`

isCountDownEnabledRight()

Verifies the schedule countdown mode.

Returns **enable** – True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCountRight`, `stopScheduleRight`, `setScheduleCountDownRight`

isScheduleRunningLeft()

Verifies if a schedule is currently running on the subsystem.

Returns **schedule_running** – True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startScheduleLeft`, `stopScheduleLeft`, `getScheduleRunningStateLeft`

isScheduleRunningRight()

Verifies if a schedule is currently running on the subsystem.

Returns **schedule_running** – True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startScheduleRight`, `stopScheduleRight`, `getScheduleRunningStateRight`

`setAudioBuffer`(*buff_addr*, *buff_size*)

Sets base address, reads address and buffer size for AUD schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr

Parameters

- **buff_addr** (*int*) – Base addresss.
- **buff_size** (*int*) – Buffer size.

`setAudioSchedule`(*onset*, *rateValue*, *rateUnits*, *count*)

Sets AUD schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If Count > 0, enables Countdown mode.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*string*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

`setBaseAddressLeft`(*value*)

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`setBaseAddressRight`(*value*)

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`setBufferSizeLeft`(*buffer_size*)

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBaseAddress() The given size is in bytes and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

`setBufferSizeRight`(*buffer_size*)

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBaseAddress() The given size is in byte and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

`setCodecSpeakerLeftVolume`(*volume*, *unit=0*)

Sets the speaker volume of the left channel.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getCodecSpeakerRightVolume, getCodecSpeakerVolume, getCodecSpeakerRightVolume, setCodecSpeakerLeftVolume, setCodecSpeakerVolume

setCodecSpeakerRightVolume (*volume*, *unit*)
Sets the speaker volume of the right channel.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getCodecSpeakerLeftVolume, getCodecSpeakerVolume, getCodecSpeakerRightVolume, setCodecSpeakerLeftVolume, setCodecSpeakerVolume

setCodecSpeakerVolume (*volume*, *unit=0*)
Sets the speaker volume of both Left and right channels.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getCodecSpeakerLeftVolume, getCodecSpeakerRightVolume, getCodecSpeakerRightVolume, setCodecSpeakerLeftVolume, setCodecSpeakerVolume

setCodecVolume (*volume*, *unit*)
Sets the codec volume of for both left and right channels.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getVolumeRight, getVolumeLeft, setVolumeRight, setVolumeLeft, setVolume, setCodecVolume, setCodecVolumeLeft, setCodecVolumeRight, getCodecVolume, getCodecVolumeLeft, getCodecVolumeRight

setCodecVolumeLeft (*volume*, *unit*)
Sets the codec volume of the left channel.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getVolumeRight, getVolumeLeft, setVolumeRight, setVolumeLeft, setVolume, setCodecVolume, getCodecVolumeLeft, setCodecVolumeRight, getCodecVolume, getCodecVolumeLeft, getCodecVolumeRight

setCodecVolumeRight(*volume*, *unit*)

Sets the codec volume of the right channel.

Parameters

- **volume** (*float*) – Range 0-1 or dB.
- **unit** (*float*) – 0 for linear or 1 for dB.

See also:

getVolumeRight, getVolumeLeft, setVolumeRight, setVolumeLeft, setVolume, setCodecVolume, setCodecVolumeLeft, getCodecVolumeRight, getCodecVolume, getCodecVolumeLeft

setLeftRightMode(*mode*)

Sets how audio data is updated by schedules.

Parameters **mode** (*str*) – Any of the following predefined constants.

- **mono** : Left schedule data goes to left and right channels.
- **left** : Each schedule data goes to left channel only.
- **right** : Each schedule data goes to right channel only.
- **stereo1** : Pairs of Left data are copied to left/right channels.
- **stereo2** : Left data goes to left channel, Right data goes to right.

See also:

getLeftRightMode

setReadAddressLeft(*address*)

Sets the Ram buffer read address.

This method allows the user to set the RAM buffer read address used in schedules. This address is used by the schedule to know where the data should be first read from. The schedule will then read the following data to the address following the RAM buffer read address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

setReadAddressRight(*address*)

Sets the Ram buffer read address.

This method allows the user to set the RAM buffer read address used in schedules. This address is used by the schedule to know where the data should be first read from. The schedule will then read the following data to the address following the RAM buffer read address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

setScheduleCountDownLeft(*enable*)

Sets the schedule countdown mode.

This method allows the user to enable or disable the countdown on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if countdown is enabled, False otherwise.

See also:

`setScheduleCountLeft`, `stopScheduleLeft`, `isCountDownEnabledLeft`

`setScheduleCountDownRight`(*enable*)

Sets the schedule countdown mode.

This method allows the user to enable or disable the countdown on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if countdown is enabled, False otherwise.

See also:

`setScheduleCountRight`, `stopScheduleRight`, `isCountDownEnabledRight`

`setScheduleCountLeft`(*count*)

Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of samples. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleUnitLeft`, `setScheduleCountDownLeft`

`setScheduleCountRight`(*count*)

Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of samples. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCountRight`, `setScheduleCountDownRight`

`setScheduleOnsetLeft`(*onset*)

Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal or greater to 0.

`setScheduleOnsetRight`(*onset*)

Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal or greater to 0.

`setScheduleRateLeft`(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate. If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal or greater to 0.
- **unit** (*str*) – hz : samples per second, maximum 96 kHz.

  video : samples per video frame, maximum 96 kHz. nano : sample period in nanoseconds, minimum 10417 ns.

setScheduleRateRight(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate. If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal or greater to 0.
- **unit** (*str*) – hz : samples per second, maximum 96 kHz.

  video : samples per video frame, maximum 96 kHz. nano : sample period in nanoseconds, minimum 10417 ns.

setVolume(*volume*)

Sets volume for the Left and Right audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

setVolumeLeft(*volume*)

Sets volume for the Left audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

setVolumeRight(*volume*)

Sets volume for the Right audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

startScheduleLeft()

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

stopScheduleLeft, setReadAddressLeft, setBaseAddressLeft, setScheduleOnsetLeft, setScheduleRateLeft, setScheduleCountDownLeft, setScheduleCountLeft

startScheduleRight()

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopScheduleRight`, `setReadAddressRight`, `setBaseAddressRight`, `setScheduleOnsetRight`, `setScheduleRateRight`, `setScheduleCountDownRight`, `setScheduleCountRight`

`stopSchedule``()`

Stops the active schedule for a given subsystem.

Depending on how the schedules are configured, it may not be necessary to call this method. When a schedule is using a countdown, it is not required to stop the schedule.

`stopScheduleLeft``()`

Stops the active schedule for a given subsystem.

Depending on how the schedules are configured, it may not be necessary to call this method. When a schedule is using a countdown, it is not required to stop the schedule.

See also:

`startScheduleLeft`, `setReadAddressLeft`, `setBaseAddressLeft`, `setScheduleOnsetLeft`, `setScheduleRateLeft`, `setScheduleCountDownLeft`, `setScheduleCountLeft`

`stopScheduleRight``()`

Stops the active schedule for a given subsystem.

Depending on how the schedules are configured, it may not be necessary to call this method. When a schedule is using a countdown, it is not required to stop the schedule.

See also:

`startScheduleRight`, `setReadAddressRight`, `setBaseAddressRight`, `setScheduleOnsetRight`, `setScheduleRateRight`, `setScheduleCountDownRight`, `setScheduleCountRight`

CHAPTER

FOURTEEN

DIGITAL INPUTS

class `digitalIn.`**`DigitalIn`**

Bases: `schedule.Schedule`

Class which implements the DIN features.

Any device which has Digital IN should implement this Class. It contains all the necessary methods to use the digital inputs of a VPixx device.

`getBaseAddress``()`

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize().

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBitDirection``()`

Gets the port direction mask.

User can obtain the value in decimal or hexadecimal.

Returns Bit set to 1 is an enabled port. Bit set to 0 is a disabled port.

Return type int

`getBufferSize``()`

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getNbrOfBit``()`

Gets the number of bit available.

Returns Number of bits.

Return type int

`getOutputStrength``()`

Gets the strength of the driven outputs.

This method allows the user to get the strength currently driven by the outputs. The implementation uses values from `1/16` up to `16/16`. Therefore minimum strength will be `0.0625` and maximum will be `1`. The strength can be increased by `0.0625` up to `1`.

Returns Any value in a range of 0 to 1.

Return type float

getOutputValue()

Gets the data which is being driven on each output port.

This method allows the user to get the data which is currently driven on the output port.

Returns Any value in a range of 0 to 16777215.

Return type int

getScheduleCount()

Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleCount`, `setScheduleCountDown`

getScheduleOnset()

Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleOnset`

getScheduleRate()

Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleRate`, `getScheduleUnit`

getScheduleUnit()

Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleRate`, `setScheduleRate`

getValue()

Gets the current value of the bits.

Returns Value of bits.

Return type int

getWriteAddress()

Gets the Ram buffer write address.

This method allows the user to get the RAM buffer write address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

isCountDownEnabled()

Verifies the schedule count down mode.

Returns **enable** – True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCount`, `stopSchedule`, `setScheduleCountDown`

isDebounceEnabled()

Verifies if the input debounce mode is enabled.

Returns **debouce** – True if transitions are being stabilized, False otherwise.

Return type Bool

isLogEventsEnabled()

Verifies if the transition log events mode is enabled.

Returns **enable** – True if transition log events mode is enabled, False otherwise.

Return type Bool

See also:

`setLogEvents`

isLogTimetagsEnabled()

Verifies if the timetag mode is enabled.

Returns **enable** – True if timetag mode is enabled, False otherwise.

Return type Bool

See also:

`setLogTimetags`

isLoopbackEnabled()

Verifies the digital inputs and outputs loopback mode.

Returns **enable** – True if transitions are being stabilized, False otherwise.

Return type Bool

See also:

`setLoopback`

isScheduleRunning()

Verifies if a schedule is currently running on the subsystem.

Returns **schedule_running** – True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startSchedule`, `stopSchedule`

`isStabilizeEnabled`()

Verifies if the input transitions are being stabilized.

Returns stabilize – True if transitions are being stabilized, False otherwise.

Return type Bool

`setBaseAddress`(*address*)

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters address (*int*) – Any value in a range of 0 up to the RAM size.

`setBitDirection`(*bit_mask*)

Sets the port direction mask.

Sets the port direction for each bit. The mask is one value representing all bits from the port. The given `bit_mask` will set the direction of all digital input bits. For each bit which should drive its port, the corresponding `bit_mask` value should be set to 1. An hexadecimal `bit_mask` can be provided.

For example, `bit_mask = 0x0000F` will enable the port for the first 4 bits on the right. All other ports will be disabled. User can then use the first 4 bits to drive the port.

Parameters int – Set bit to 1 to enable the port for that bit. Set bit to 0 to disable it.

`setBufferSize`(*buffer_size*)

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedule to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use `setBaseAddress()`. The given size is in bytes and must be an even value.

Parameters buffer_size (*int*) – Any value in a range of 0 up to the RAM size.

`setDebounce`(*enable*)

Sets the input debounce mode.

This method allows the user to enable or disable the input debounce mode. When a DIN transitions, ignore further DIN transitions for the next 30 milliseconds. This is useful for response buttons. When disabled, the inputs transitions are immediately recognized. In which case, it can be useful to enable debouncing using `setStabilize()`.

Parameters enable (*Bool*) – True if transitions are being debounced, False otherwise.

`setLogEvents`(*enable*)

Sets the transition log events mode on data samples.

This method allows the user to enable or disable the transition log events on acquired data. When enabled, each transition is automatically logged. No schedule is required. This is the best way to log response buttons. When disabled, logging of transitions is done automatically.

Parameters enable (*Bool*) – True to activate the log event mode, False otherwise.

See also:

`isLogEventsEnabled`

`setLogTimetags`(*enable*)
Sets the timetag mode on the data samples.

This method allows the user to enable or disable the timetag on acquired data. When enabled, each buffered sample is preceded with a 64-bit nanosecond timetag. When disabled, buffered data has no timetags.

Parameters **enable** (*Bool*) – True to activate the timetag mode, False otherwise.

See also:

`isLogTimetagsEnabled`

`setLoopback`(*enable*)
Sets the digital inputs and outputs loopback mode.

This method allows the user to enable or disable the loopback between digital output ports and digital inputs. When enabled, the digital outputs send their data to the digital inputs. When disabled, the digital inputs will not get the digital outputs data. debouncing using "setStabilize()".

Parameters **enable** (*Bool*) – True if loopback is enabled, False otherwise.

See also:

`isLoopbackEnabled`

`setOutputStrength`(*strength*)
Sets the strength of the driven outputs.

This method allows the user to set the current (Ampere) strength of the driven outputs. The implementation actual values uses `1/16` up to `16/16`. So minimum strength will be `0.0625` and maximum will be `1`. The strength can be increased by `0.0625` up to `1`. Giving a strength of `0` will thus set it to `0.0625`. Giving a strength between `0.0625` and `0.125` will round the given strength to one of the two increments.

The strength is the same for all bits.

Parameters **strength** (*float*) – Any value in a range of 0 to 1.

`setOutputValue`(*value*)
Sets the data which should be driven on each port.

In order to be able to drive the ports with the given value, the port direction has to be properly enabled. This can be done using the "setBitDirection()" with the appropriate bit mask.

Parameters **value** (*int*) – Any value in a range of 0 to 16777215.

`setScheduleCount`(*count*)
Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of sample. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCount`, `setScheduleCountDown`

`setScheduleCountDown`(*enable*)
Sets the schedule count down mode.

This method allows the user to enable or disable the count down on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if count down is enabled, False otherwise.

See also:

`setScheduleCount`, `stopSchedule`, `isCountDownEnabled`

`setScheduleOnset`(*onset*)

Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method does not need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal to or greater than 0.

See also:

`getScheduleOnset`

`setScheduleRate`(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than 0.
- **unit** (*str*) – hz : samples per second, maximum 1 MHz. video : samples per video frame, maximum 1 MHz. nano : sample period in nanoseconds, minimum 1000 ns.

See also:

`getScheduleRate`, `getScheduleUnit`

`setStabilize`(*enable*)

Sets the input stabilization mode.

This method allows the user to enable or disable the input transitions stabilization. When enabled, input transitions are only recognized after the entire input bus has been stable for 80 ns. This is useful for deskewing parallel busses, and ignoring transmission line reflections. When disabled, the input transitions are immediately recognized. In which case, it can be useful to enable debouncing using `setDebounce()`.

Parameters **enable** (*Bool*) – True if transitions are being stabilized, False otherwise.

`setWriteAddress`(*address*)

Sets the Ram buffer write address.

This method allows the user to set the RAM buffer write address used in schedules. This address is used by the schedule to know where the data should be first written to. The schedule will then write the following data to the address following the RAM buffer write address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`startSchedule`()

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

stopSchedule()

stop the active schedule for a given subsystem.

Depending on how the Schedules are configured, it may not be necessary to call this method. When a schedule is using a count down, it is not required to stop the schedule.

See also:

`startSchedule`, `setWriteAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

CHAPTER

FIFTEEN

DIGITAL OUT

class `digitalOut.`**`DigitalOut`**

Bases: `schedule.Schedule`

The Dout class is used to set and get all values related to the Dout.

`disablePixelMode``()`

Disables pixel mode.

When this function is disabled, the digital ouputs do not show the RGB value of first upper left pixel of the screen. The digital outputs can then be used normally. This is the default mode. This feature is only available on VIEWPixx with firmware revision 31 and higher.

See also:

`enablePixelMode`, `isPixelModeEnabled`

`enablePixelMode``()`

Enables pixel mode.

When this function is enabled, the digital outputs show the RGB value of first upper left pixel of the screen. In this case, digital outputs cannot be used for other purposes. This feature is only available on VIEWPixx with firmware revision 31 and higher.

See also:

`disablePixelMode`, `isPixelModeEnabled`

`getBaseAddress``()`

Gets the Ram buffer start address.

This method allows the user to get the RAM buffer start address used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use setBufferSize()

Returns Any value in a range of 0 up to the RAM size.

Return type int

`getBitValue``()`

Gets the current value of the bits.

Returns value of bit.

Return type int

See also:

`setBitValue`

getBufferSize()

Gets the Ram buffer size.

This method allows the user to get the RAM buffer size used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

getNbrOfBit()

Gets the number of bits available.

Returns Number of bits.

Return type int

getReadAddress()

Gets the Ram buffer read address.

This method allows the user to get the RAM buffer read address used in schedules.

Returns Any value in a range of 0 up to the RAM size.

Return type int

getScheduleCount()

Gets the schedule count value.

This method allows the user to get the current count for a schedule.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`setScheduleCount`, `setScheduleCountDown`

getScheduleOnset()

Gets the schedule onset value.

This method allows the user to get the schedule onset value used in schedules. The onset represents a nanosecond delay between schedule start and first sample.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleRate()

Gets the schedule rate value.

This method allows the user to get the schedule rate value used in schedules. The rate represents the speed at which the schedule updates.

Returns Any positive value equal to or greater than 0.

Return type int

getScheduleRunningState()

Gets the schedule state for the subsystem.

Returns **schedule_state** – "running" if a schedule is currently running, "stopped" otherwise.

Return type str

See also:

`startSchedule`, `stopSchedule`, `isScheduleRunning`

getScheduleUnit()
Gets the schedule unit value.

This method allows the user to get the schedule unit value used in schedules.

Returns Any positive value equal to or greater than 0.

Return type int

See also:

`getScheduleRate`, `setScheduleRate`

isBacklightPulseEnabled()
Verifies if the back light pulse mode is enabled.

Returns **enable** – True if back light pulse mode is enabled, False otherwise.

Return type Bool

See also:

`setBacklightPulse`

isButtonSchedulesEnabled()
Verifies if the schedule start on button press mode is enabled.

Returns **enable** – True if the mode is enabled, False otherwise.

Return type Bool

See also:

`setButtonSchedules`

isCountDownEnabled()
Verifies the schedule count down mode.

Returns **enable** – True if the schedule is decrementing at every sample, False otherwise.

Return type Bool

See also:

`setScheduleCount`, `stopSchedule`, `setScheduleCountDown`

isPixelModeEnabled()
Verifies if the pixel mode is enabled on digital outputs.

Returns **enable** – True if the mode is enabled, False otherwise.

Return type Bool

See also:

`enablePixelMode`, `disablePixelMode`

isScheduleRunning()
Verifies if a schedule is currently running on the subsystem.

Returns **schedule_running** – True if a schedule is currently running, False otherwise.

Return type Bool

See also:

`startSchedule`, `stopSchedule`, `getScheduleRunningState`

setBacklightPulse(*enable*)

Sets the back light pulse mode.

This method allows the user to enable or disable the back light pulse mode. When enabled, the LCD back light LEDs are gated by bit 15 of the digital output. It can be used to make a tachistoscope by pulsing bit 15 of the digital output with a schedule. When disabled, the outputs work normally and are unaffected.

Parameters **enable** (*Bool*) – True to activate the back light pulse mode, False otherwise.

See also:

`isBacklightPulseEnabled`

setBaseAddress(*value*)

Sets the Ram buffer start address.

This method allows the user to set the RAM buffer start address used in schedules. The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

setBitValue(*value*, *bit_mask*)

Sets the value of the bits.

This method allows the user to set the bit value for the subsystem. The mask is one value representing all bits from the port. The given bit_mask will set the direction of all digital input bits. For each bit which should drive its port, the corresponding bit_mask value should be set to 1.

For example, `bit_mask = 0x0000F` will enable the port for the first 4 bits on the right. All other ports will be disabled. User can then use the first 4 bits to drive the port.

Parameters

- **value** (*int*) – value of bits.
- **bit_mask** (*int*) – Set bit to 1 will enable the port for that bit. Set bit to 0 will disable it.

See also:

`getBitValue`

setBufferSize(*buffer_size*)

Sets the Ram buffer size.

This method allows the user to set the RAM buffer size used in schedules. It should only be used if the user wants the schedules to wrap when it has reached its maximum size. When schedules are expected to wrap, the user should also use `setBaseAddress()`. The given size is in bytes and must be an even value.

Parameters **buffer_size** (*int*) – Any value in a range of 0 up to the RAM size.

setButtonSchedules(*enable*)

Sets the schedule start on button press mode.

This method allows the user to enable or disable the transition log events on acquired data. When enabled, digital output schedules are done following a digital input button press. When disabled, digital output schedules have to be started manually.

Parameters **enable** (*Bool*) – True to activate the log event mode, False otherwise.

See also:

`isButtonSchedulesEnabled`

`setReadAddress`(*address*)

Sets the Ram buffer read address.

This method allows the user to set the RAM buffer read address used in schedules. This address is used by schedule to know where the data should be first read from. The schedules will then read the following data to the address following the RAM buffer read address.

The given address must be an even value.

Parameters **address** (*int*) – Any value in a range of 0 up to the RAM size.

`setScheduleCount`(*count*)

Sets the schedule count.

This method allows the user to set the schedule count for a schedule with a fixed number of sample. In which case, the schedule will decrement at a given rate and stop when the count reaches 0.

Parameters **count** (*int*) – Any positive value greater than 0.

See also:

`getScheduleCount`, `setScheduleCountDown`

`setScheduleCountDown`(*enable*)

Sets the schedule count down mode.

This method allows the user to enable or disable the count down on a schedule. When enabled, the schedule decrements at the given rate and stops automatically when the count hits 0. When disabled, the schedule increments at the given rate and is stopped by calling stopSchedule().

Parameters **enable** (*Bool*) – True if count down is enabled, False otherwise.

See also:

`setScheduleCount`, `stopSchedule`, `isCountDownEnabled`

`setScheduleOnset`(*onset*)

Sets the schedule onset value.

This method allows the user to set the nanosecond delay between schedule start and first sample. If no delay is required, this method doesn't need to be used. Default value is 0.

Parameters **onset** (*int*) – Any positive value equal to or greater than 0.

`setScheduleRate`(*rate*, *unit='hz'*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method doesn't need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than 0.
- **unit** (*str*) – hz : rate updates per second, maximum 10 MHz. video : rate updates per video frame, maximum 10 MHz. nano : rate updates period in nanoseconds, minimum 100 ns.

`startSchedule`()

Starts a schedule.

Schedules may be configured in different ways, affecting their behavior. Before a schedule is started, the user should make sure that it is properly set in the right mode.

See also:

`stopSchedule`, `setReadAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

stopSchedule()

Stops the active schedule for a given subsystem.

Depending on how the schedules are configured, it may not be necessary to call this method. When a schedule is using a count down, it is not needed to stop the schedule.

See also:

`startSchedule`, `setReadAddress`, `setBaseAddress`, `setScheduleOnset`, `setScheduleRate`, `setScheduleCountDown`, `setScheduleCount`

CHAPTER

SIXTEEN

DPX DEVICE

class `dpxDevice.`**`DpxDevice`**(*device_type*)

Bases: `object`

Implements the features common for all Devices.

`identification`
Number which identifies the type of device.

`part_number`
Part number of the device.

`firmware_revision`
Firmware revision of the device.

`ram_size`
The size of the Ram found on the device.

`close`()
Closes a VPixx device.

This method is used to release a handle on a VPixx device.

`get12vCurrent`()
Gets the current for the 12 volt Power supply.

Returns Temperature in Celsius.

Return type float

`get12vPower`()
Gets the power for the 12 volt Power supply.

Returns Power in watts.

Return type float

`get12vVoltage`()
Gets the current for the 12 volt Power supply.

Returns Temperature in Celsius.

Return type float

`get5vCurrent`()
Gets the current for the 5 volt Power supply.

Returns Temperature in Celsius.

Return type float

get5vPower()
Gets the current for the 5 volt Power supply.

Returns power in watts.

Return type float

get5vVoltage()
Gets the voltage for the 5 volt Power supply.

Returns Temperature in Celsius.

Return type float

getAssemblyRevision()
Gets the device revision.

Returns The assembly revision of the device.

Return type string

getAvailableSubSystem()
Gets the available subsystems for the device.

Returns Name of subsystems found on the device.

Return type List

getCoreTemperature()
Gets the core temperature from the device.

Returns Temperature in Celsius.

Return type float

getDisplayResolution()
Gets the resolution of the device.

Returns Horizontal resolution followed by vertical resolution.

Return type string

getFirmwareRevision()
Gets the firmware revision.

Returns The firmware revision of the device.

Return type string

getFrameTemperature()
Gets the temperature from the device chassis.

Returns Temperature in Celsius.

Return type float

getFrameTemperature2()
Gets the temperature from the device chassis.

Returns Temperature in Celsius.

Return type float

getInfo()
Gets the device production information.

This method can be used to get information about the current device. Some of these information are different from one device to another. It can be useful when contacting VPixx Technologies.

Returns

info – Any of the predefined constants.

- **Part Number**: The part number of the device.
- **Shipping Date**: The date at which the device was shipped.
- **Assembly REV**: The device assembly revision.
- **Expiration Date**: Date when the warranty expires.
- **S/N**: Serial number of the device.

Return type dict

`getMonitorCableLink()`

Gets the monitor cable link.

This method allows the user to know what kind of cable link is detected by the VPixx device.

Returns

Any of the following predefined constants.

- **DVI-Single Link**.
- **DVI-Dual Link**.

Return type string

`getName()`

Gets the device type.

Returns Device type.

Return type string

`getRamSize()`

Gets the RAM size.

Returns The amount of RAM found on the device.

Return type string

`getSerialNumber()`

Gets the serial number.

Returns The serial number of the device.

Return type string

`getTime()`

Gets the device time since power up.

Returns Time in seconds.

Return type float

`getVideoLine()`

Reads pixels from the VPixx device line buffer, and returns a list containing the data. For each pixel, the buffer contains 16 bit R/G/B/U (where U is thrown away). The returned data is a list containing three lists for the respective R/G/B colors.

Returns lists of list: A list which has [[RED], [GREEN], [BLUE]]

`getVideoSource()`

Get source of video pattern being displayed.

Returns Source of video pattern.

Return type int

isReady()
Verifies if s device has been properly opened.

open()
Opens a VPixx device.

This method is used to get a handle on a VPixx device.

readRam(*address*, *int_list*)
Reads a block of VPixx RAM into a local buffer.

Parameters

- **address** (*int*) – Any even value equal or greater to 0.
- **length** (*int*) – Any of value from 0 to RAM size.

setCustomStartupConfig()
Save the current registers to be used on start up.

This can be useful if you set your projector to Ceiling mode or Rear projection and you want to keep it as such on reboot.

setVideoSource(*vidSource*)
Set source of video to be displayed

Parameters **vidSource** (*str*) – The source we want to display.

- **DVI**: Monitor displays DVI signal.
- **SWTP**: Software test pattern showing image from RAM.
- **SWTP 3D**: 3D Software test pattern flipping between left/right eye images from RAM.
- **RGB SQUARES**: RGB ramps.
- **GRAY**: Uniform gray display having 12-bit intensity.
- **BAR**: Drifting bar.
- **BAR2**: Drifting bar.
- **DOTS**: Drifting dots.
- **RAMP**: Drifting ramp, with dots advancing x*2 pixels per video frame, where x is a 4-bit signed.
- **RGB**: Uniform display with 8-bit intensity nn, send to RGBA channels enabled by mask m.
- **PROJ**: Projector Hardware test pattern.

updateRegisterCache()
Updates the registers and local register cache.

writeRam(*address*, *int_list*)
Writes a local buffer into VPixx RAM.

Parameters

- **address** (*int*) – Any even value equal or greater to 0.
- **int_list** (*int*) – int_list is a list which will fill with RAM data. The length of RAM used is based on the length of int_list. It can’t be bigger than RAM size.

writeRegisterCache()

Writes the registers with local register cache.

CHAPTER SEVENTEEN

DUAL LINK OUT

class `dualLinkOut.`**`DualLinkOut`**

Bases: `object`

Class Definition for the Dual Link Out for each Device.

CHAPTER EIGHTEEN

3D FEATURES

class `threeDFeatures.`**`ThreeDFeatures`**

Bases: `object`

Implements the the different 3D features for your device.

`getVesaOutputSignalEye``()`

Returns the eye on which the VESA outputs an image signal.

Returns **eye** – Eye on which VESA is outputting the signal.

Return type str

`getVesaPhase``()`

Gets the 8-bit unsigned phase of the VESA 3D waveform.

Returns **phase** – Phase of the VESA 3D waveform

Return type int

`getVesaWaveform``()`

Gets the waveform which is being sent to the VESA 3D connector.

Returns **waveform** – Phase of the VESA 3D waveform

Return type str

`setVesaOutputSignalEye``(`*eye*`)`

Sets the VESA output signal eye.

The VESA connector can output image signal on either left or right eye. This method allows the user to choose one.

Parameters **eye** (*string*) – left if VESA connector it to output left-eye signal. right otherwise.

`setVesaPhase``(`*phase*`)`

Sets the 8-bit unsigned phase of the VESA 3D waveform.

Varying this phase from 0-255 will fine tune phase relationship between stereo video and 3D goggle switching. The following combinations have been found to work well:

If you are using a VIEWPIxx/3D, you should set the `phase` to `0x64`. If you are using a CTR with our DATAPixx, it should be set to `0xF5`.

Parameters **phase** (*int*) – Phase of the VESA 3D waveform

`setVesaWaveform``(`*waveform*`)`

Sets the 8-bit unsigned phase of the VESA 3D waveform.

Sets the waveform which will be sent to the DATAPixx VESA 3D connector

Parameters **waveform** (*str*) – If you are using NVIDIA 3D Vision Glasses, you should set the `Waveform` to `NVIDIA`. If you are using Crystaleyes Glasses, it should be set to `CRYSTALEYES`. For a basic L/R square wave, it should be `LR`.

CHAPTER

NINETEEN

LIBDPX WRAPPER

Description: This module is a wrapper of VPixx low-level library for Python. It provides all functions found in "Libdpx.h", as well as the error codes "defined". The first section is the list of all functions available to the user. The second section is the list of all error codes. The last section provides short descriptions for each error codes.

`_libdpx.`**`DPxAutoVidHorizSplit`**`()`

Sets the Horizontal Split mode automatically

DATAPixx will automatically split video across the two VGA outputs if the horizontal resolution is at least twice the vertical resolution (default mode).

Low-level C definition `void DPxAutoVidHorizSplit()`

`_libdpx.`**`DPxAutoVidVertStereo`**`()`

Turns on the automatic mode for Video Vertical Stereo.

Vertical stereo is enabled automatically when vertical resolution > horizontal resolution (default mode)

Low-level C definition `void DPxAutoVidVertStereo()`

`_libdpx.`**`DPxClearError`**`()`

Clear any error on the device.

Low-level C definition `void DPxClearError()`

`_libdpx.`**`DPxClose`**`()`

Close currently selected VPixx device.

Low-level C definition `void DPxClose()`

`_libdpx.`**`DPxDetectDevice`**`(`*devsel*`)`

Verifies if a specific device exists in the system.

Parameters **devsel** (*string*) – Any of the predefined constants.

- **DATAPixx**: DATAPixx.
- **DATAPixx2**: DATAPixx2.
- **VIEWPixx**: VIEWPixx.
- **PROPixx Ctrl**: PROPixx Controller.
- **PROPixx**: PROPixx.

Returns Non-zero if the device exists, 0 if the device does not exist.

Return type int

Low-level C definition `int DPxDetectDevice(int devsel)`

`_libdpx.`**`DPxDisableAdcBuffAllChans`**`()`

Disables RAM buffering of all ADC channels.

Low-level C definition `void DPxDisableAdcBuffAllChans()`

See also:

`DPxEnableAdcBuffChan`, `DPxDisableAdcBuffChan`, `DPxIsAdcBuffChan`

`_libdpx.`**`DPxDisableAdcBuffChan`**`(`*channel*`)`

Disables RAM buffering of an ADC channel.

This function is only available for channels 0-15.

Parameters **channel** (*int*) – Channel number.

Low-level C definition `void DPxDisableAdcBuffChan(int channel)`

See also:

`DPxEnableAdcBuffChan`, `DPxDisableAdcBuffAllChans`, `DPxIsAdcBuffChan`

`_libdpx.`**`DPxDisableAdcCalibRaw`**`()`

Sets the hardware calibration mode.

Disables ADC "raw" mode, causing normal ADC hardware calibration.

Low-level C definition `void DPxDisableAdcCalibRaw()`

See also:

`DPxDisableAdcCalibRaw`, `DPxIsAdcCalibRaw`

`_libdpx.`**`DPxDisableAdcFreeRun`**`()`

Sets the ADC free run mode.

ADCs only convert on schedule ticks, for microsecond-precise sampling.

Low-level C definition `void DPxDisableAdcFreeRun()`

See also:

`DPxEnableAdcFreeRun`, `DPxIsAdcFreeRun`

`_libdpx.`**`DPxDisableAdcLogTimetags`**`()`

Disables ADC timetag mode.

Buffered data has no timetags.

Low-level C definition `void DPxDisableAdcLogTimetags()`

See also:

`DPxDisableAdcLogTimetags`, `DPxIsAdcLogTimetags`

`_libdpx.`**`DPxDisableAdcSchedCountdown`**`()`

Disables ADC schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableAdcSchedCountdown()`

See also:

`DPxEnableAdcSchedCountdown`, `DPxIsAdcSchedCountdown` `_libdpx.`**`DPxDisableAudMicLoopback`**`()`

Disables loopback between audio outputs and microphone inputs

Low-level C definition `void DPxDisableAudMicLoopback()`

`_libdpx.`**`DPxDisableAudSchedCountdown`**`()`

Disables AUD schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableAudSchedCountdown()`

`_libdpx.`**`DPxDisableAuxSchedCountdown`**`()`

Disables the AUX Schedule Countdown

SchedCount increments at SchedRate, and schedule is stopped by calling Sched

Low-level C definition `void DPxDisableAuxSchedCountdown()`

`_libdpx.`**`DPxDisableDacAdcLoopback`**`()`

Disables the loopback between ADC and DAC.

Disables ADC loopback, causes ADC readings to reflect real analog inputs.

Low-level C definition `void DPxDisableDacAdcLoopback()`

See also:

`DPxEnableDacAdcLoopback`, `DPxIsDacAdcLoopback`

`_libdpx.`**`DPxDisableDacBuffAllChans`**`()`

Disables RAM buffering of all DAC channels.

Low-level C definition `void DPxDisableDacBuffAllChans()`

See also:

`DPxEnableDacBuffChan`, `DisableDacBuffChan`

`_libdpx.`**`DPxDisableDacBuffChan`**`(`*channel*`)`

Disables RAM buffering of a DAC channel.

Parameters **channel** (*int*) – Channel number.

Low-level C definition `void DPxDisableDacBuffChan(int channel)`

See also:

`DPxEnableDacBuffChan`, `DPxDisableDacBuffAllChans`

`_libdpx.`**`DPxDisableDacCalibRaw`**`()`

Sets the hardware calibration mode.

Disables DAC “raw” mode, causing normal DAC hardware calibration.

Low-level C definition `void DPxDisableDacCalibRaw()`

See also:

`DPxEnableDacCalibRaw`

`_libdpx.`**`DPxDisableDacSchedCountdown`**`()`

Disables DAC schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableDacSchedCountdown()`

See also:

DPxEnableDacSchedCountdown, DPxIsDacSchedCountdown

`_libdpx.`**`DPxDisableDinDebounce`**`()`

Enables the input debounce mode.

Immediately recognize all DIN transitions (after possible stabilization).

Low-level C definition `void DPxDisableDinDebounce()`

See also:

DPxEnableDinDebounce, DPxIsDinDebounce

`_libdpx.`**`DPxDisableDinLogEvents`**`()`

Disables log events mode.

Disables automatic logging of DIN transitions. A schedule is required to look at DIN transitions.

Low-level C definition `void DPxDisableDinLogEvents()`

See also:

DPxDisableDinLogEvents, DPxIsDinLogEvents

`_libdpx.`**`DPxDisableDinLogTimetags`**`()`

Disables Din timetag mode.

Buffered data has no timetags.

Low-level C definition `void DPxDisableDinLogTimetags()`

See also:

DPxDisableDinLogTimetags, DPxIsDinLogTimetags

`_libdpx.`**`DPxDisableDinSchedCountdown`**`()`

Disables Din schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableDinSchedCountdown()`

See also:

DPxEnableDinSchedCountdown, DPxIsDinSchedCountdown

`_libdpx.`**`DPxDisableDinStabilize`**`()`

Disables the input stabilization mode.

Immediately recognize all DIN transitions, possibly with debouncing.

Low-level C definition `void DPxDisableDinStabilize()`

See also:

DPxEnableDinStabilize, DPxIsDinStabilize

`_libdpx.`**`DPxDisableDoutBacklightPulse`**`()`

Disables the Dout backlight pulse mode.

LCD backlight LEDs are unaffected by DOUT system.

Low-level C definition `void DPxDisableDoutBacklightPulse()`

See also:

DPxEnableDoutBacklightPulse, DPxIsDoutBacklightPulse `_libdpx.`**`DPxDisableDoutButtonSchedules`**`()`

Disables automatic DOUT schedules upon DIN button presses.

Low-level C definition `void DPxDisableDoutButtonSchedules()`

See also:

`DPxEnableDoutButtonSchedules`, `DPxIsDoutButtonSchedules`

`_libdpx.`**`DPxDisableDoutDinLoopback`**`()`

Disables loopback between digital outputs and digital inputs.

Immediately recognize all DIN transitions (after possible stabilization).

Low-level C definition `void DPxDisableDoutDinLoopback()`

See also:

`DPxEnableDoutDinLoopback`, `DPxIsDoutDinLoopback`

`_libdpx.`**`DPxDisableDoutPixelMode`**`()`

Disables pixel mode.

When this function is disabled, the digital ouputs do not show the RGB value of first upper left pixel of the screen. The digital outputs can then be used normally. This is the default mode. This feature is only available on VIEWPixx with firmware revision 31 and higher.

Low-level C definition `void DPxDisableDoutPixelMode()`

See also:

`DPxEnableDoutPixelMode`, `DPxIsDoutPixelMode`

`_libdpx.`**`DPxDisableDoutSchedCountdown`**`()`

Disables Dout schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableDoutSchedCountdown()`

See also:

`DPxEnableDoutSchedCountdown`, `DPxIsDoutSchedCountdown`

`_libdpx.`**`DPxDisableMicSchedCountdown`**`()`

Disables MIC schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxDisableMicSchedCountdown()`

`_libdpx.`**`DPxDisablePPxCeilingMount`**`()`

Disables the PROPixx Ceiling Mount mode.

Low-level C definition `void DPxDisablePPxCeilingMount(void)`

`_libdpx.`**`DPxDisablePPxHotSpotCorrection`**`()`

`_libdpx.`**`DPxDisablePPxLampLed`**`()`

Disables the lamp LED of the PROPixx.

Only available for PROPixx Revision 12 and higher.

Low-level C definition `void DPxDisablePPxLampLed(void)`

`_libdpx.`**`DPxDisablePPxQuietFanMode`**`()`

Disables the quiet fan mode on the PROPixx.

Disabling this mode sets the fans to maximum speed, thus increasing the noise produced by them. Only available for PROPixx Revision 19 and higher.

Low-level C definition `void DPxDisablePPxQuietFanMode(void)`

`_libdpx.`**`DPxDisablePPxRearProjection`**`()`

Disables the Rear Projection of the PROPixx.

Low-level C definition `void DPxDisablePPxRearProjection(void)`

`_libdpx.`**`DPxDisableTouchpixx`**`()`

Disables the TOUCHPixx touch panel hardware subsystem.

Low-level C definition `void DPxDisableTouchpixx()`

See also:

`DPxEnableTouchpixx`, `DPxIsTouchpixx`

`_libdpx.`**`DPxDisableTouchpixxLogContinuousMode`**`()`

Disables Touchpixx continuous logging mode.

TOUCHPixx logging only returns initial press and release events.

Low-level C definition `void DPxDisableTouchpixxLogContinuousMode()`

See also:

`DPxDisableTouchpixxLogContinuousMode`, `DPxIsTouchpixxLogContinuousMode`

`_libdpx.`**`DPxDisableTouchpixxLogEvents`**`()`

Disables log events mode.

Disables automatic logging of Touchpixx transitions. A schedule is needed to log transitions.

Low-level C definition `void DPxDisableTouchpixxLogEvents()`

See also:

`DPxDisableTouchpixxLogEvents`, `DPxIsTouchpixxLogEvents`

`_libdpx.`**`DPxDisableTouchpixxLogTimetags`**`()`

Disables Touchpixx timetag mode.

Buffered data has no timetags.

Low-level C definition `void DPxDisableTouchpixxLogTimetags()`

See also:

`DPxDisableTouchpixxLogTimetags`, `DPxIsTouchpixxLogTimetags`

`_libdpx.`**`DPxDisableVidClutTransparencyColorMode`**`()`

Disables video CLUT transparency color mode

Low-level C definition `void DPxDisableVidClutTransparencyColorMode()`

See also:

`DPxEnableVidClutTransparencyColorMode`, `DPxIsVidClutTransparencyColorMode`, `DPxGetVidClutTransparencyColor`, `DPxSetVidClutTransparencyColor`

`_libdpx.`**`DPxDisableVidHorizOverlay`**`()`

Disables the Horizontal overlay.

Low-level C definition `void DPxDisableVidHorizOverlay()`

`_libdpx.`**`DPxDisableVidHorizSplit`**`()`

Disables Video Horizontal Split.

VGA 1 and VGA 2 both show entire video image (hardware video mirroring)/

Low-level C definition `void DPxDisableVidHorizSplit()`

`_libdpx.`**`DPxDisableVidLcd3D60Hz`**`()`

Returns to normal pixel polarity inversion.

Low-level C definition `int DPxIsVidLcd3D60Hz(void)`

`_libdpx.`**`DPxDisableVidPsyncBlankLine`**`()`

Disables the PSync Blank Line.

Pixel sync raster line is displayed normally when this is disabled.

Low-level C definition `void DPxDisableVidPsyncBlankLine()`

`_libdpx.`**`DPxDisableVidPsyncSingleLine`**`()`

Disables Psync for Single (Raster) Line.

Low-level C definition `void DPxDisableVidPsyncSingleLine()`

`_libdpx.`**`DPxDisableVidRescanWarning`**`()`

Disables VIEWPixx Rescan Warning.

Low-level C definition `void DPxDisableVidRescanWarning()`

`_libdpx.`**`DPxDisableVidScanningBacklight`**`()`

Disables VIEWPixx scanning backlight.

Low-level C definition `void DPxDisableVidScanningBacklight()`

`_libdpx.`**`DPxDisableVidVertStereo`**`()`

Disables the Video Vertical Stereo.

A normal display will be in this mode.

Low-level C definition `void DPxDisableVidVertStereo()`

`_libdpx.`**`DPxDisableVidVesaBlueline`**`()`

Disables the Video blue line mode.

When enabled, the VESA 3D output interprets the middle pixel on last raster line as a blue line code. When disabled, the VESA 3D output is not dependent on video content.

Low-level C definition `void DPxDisableVidVesaBlueline()`

`_libdpx.`**`DPxDisableVidVesaFreeRun`**`()`

Disables PROPixx 3D VESA output freeRun enable bit

Note: PROPixx Rev >= 7 only.

Low-level C definition `void DPxDisableVidVesaFreeRun(void)`

`_libdpx.`**`DPxEnableAdcBuffChan`**`(`*channel*`)`

Enables RAM buffering of an ADC channel.

This function is only available for channels 0-15.

Parameters **channel** (*int*) – Channel number.

Low-level C definition `void DPxEnableAdcBuffChan(int channel)`

See also:

`DPxDisableAdcBuffChan`, `DPxDisableAdcBuffAllChans`, `DPxIsAdcBuffChan`

`_libdpx.`**`DPxEnableAdcCalibRaw`**`()`

Sets the hardware calibration mode.

Enables ADC "raw" mode, causing ADC data to bypass hardware calibration.

Low-level C definition `void DPxEnableAdcCalibRaw()`

See also:

`DPxDisableAdcCalibRaw`, `DPxIsAdcCalibRaw`

`_libdpx.`**`DPxEnableAdcFreeRun`**`()`

Sets the ADC free run mode.

ADCs convert continuously. Can add up to 4 microseconds random latency to scheduled samples.

Low-level C definition `void DPxEnableAdcFreeRun()`

See also:

`DPxDisableAdcFreeRun`, `DPxIsAdcFreeRun`

`_libdpx.`**`DPxEnableAdcLogTimetags`**`()`

Enables ADC timetag mode.

Each buffered ADC sample is preceeded by a 64-bit nanosecond timetag.

Low-level C definition `void DPxEnableAdcLogTimetags()`

See also:

`DPxDisableAdcLogTimetags`, `DPxIsAdcLogTimetags`

`_libdpx.`**`DPxEnableAdcSchedCountdown`**`()`

Enables ADC schedule count down.

SchedCount decrements at SchedRate, and schedule stops automatically when count hits 0.

Low-level C definition `void DPxEnableAdcSchedCountdown()`

See also:

`DPxDisableAdcSchedCountdown`, `DPxIsAdcSchedCountdown`

`_libdpx.`**`DPxEnableAudMicLoopback`**`()`

Enables loopback between audio outputs and microphone inputs

Low-level C definition `void DPxEnableAudMicLoopback()`

`_libdpx.`**`DPxEnableAudSchedCountdown`**`()`

Enables MIC schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxEnableAudSchedCountdown()`

`_libdpx.`**`DPxEnableAuxSchedCountdown`**`()`

Enables the AUX Schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling Sched

Low-level C definition `void DPxEnableAuxSchedCountdown()`

`_libdpx.`**`DPxEnableDacAdcLoopback`**`()`

Sets the loopback between ADC and DAC mode.

ADC data readings are looped back internally from programmed DAC voltages:

- DAC_DATA0 => ADC_DATA0/2/4/6/8/10/12/14
- DAC_DATA1 => ADC_DATA1/3/5/7/9/11/13/15
- DAC_DATA2 => ADC_REF0
- DAC_DATA3 => ADC_REF1

Low-level C definition `void DPxEnableDacAdcLoopback()`

See also:

`DPxDisableDacAdcLoopback`, `DPxIsDacAdcLoopback`

`_libdpx.`**`DPxEnableDacBuffChan`**`(`*channel*`)`

Enables RAM buffering of a DAC channel.

Parameters **channel** (*int*) – Channel number.

Low-level C definition `void DPxEnableDacBuffChan(int channel)`

See also:

`DPxDisableDacBuffChan`, `DPxDisableDacBuffAllChans`

`_libdpx.`**`DPxEnableDacCalibRaw`**`()`

Sets the hardware calibration mode.

Enables DAC "raw" mode, causing DAC data to bypass hardware calibration.

Low-level C definition `void DPxEnableDacCalibRaw()`

See also:

`DPxDisableDacCalibRaw`

`_libdpx.`**`DPxEnableDacSchedCountdown`**`()`

Enables DAC schedule count down.

SchedCount decrements at SchedRate, and schedule stops automatically when count hits 0.

Low-level C definition `void DPxEnableDacSchedCountdown()`

See also:

`DPxDisableDacSchedCountdown`, `DPxIsDacSchedCountdown`

`_libdpx.`**`DPxEnableDinDebounce`**`()`

Enables the input debounce mode.

When a DIN transitions, ignore further DIN transitions for next 30 milliseconds (good for response buttons)

Low-level C definition `void DPxEnableDinDebounce()`

See also:

`DPxDisableDinDebounce`, `DPxIsDinDebounce`

`_libdpx.`**`DPxEnableDinLogEvents`**`()`

Enables log events mode.

Each DIN transition is automatically logged. No schedule is required. Best way to log response buttons.

Low-level C definition `void DPxEnableDinLogEvents()`

See also:

`DPxDisableDinLogEvents`, `DPxIsDinLogEvents`

`_libdpx.`**`DPxEnableDinLogTimetags`**`()`

Enables Din timetag mode.

Each buffered Din sample is preceeded with a 64-bit nanosecond timetag.

Low-level C definition `void DPxEnableDinLogTimetags()`

See also:

`DPxDisableDinLogTimetags`, `DPxIsDinLogTimetags`

`_libdpx.`**`DPxEnableDinSchedCountdown`**`()`

Enables Din schedule count down.

SchedCount decrements at SchedRate, and schedule stops automatically when count hits 0.

Low-level C definition `void DPxEnableDinSchedCountdown()`

See also:

`DPxDisableDinSchedCountdown`, `DPxIsDinSchedCountdown`

`_libdpx.`**`DPxEnableDinStabilize`**`()`

Enables the input stabilization mode.

Low-level C definition `void DPxEnableDinStabilize()`

See also:

`DPxDisableDinStabilize`, `DPxIsDinStabilize`

`_libdpx.`**`DPxEnableDoutBacklightPulse`**`()`

Enables the Dout backlight pulse mode.

LCD backlight LED are controlled by DOUT15. Can be used to make a tachistoscope by pulsing DOUT15 with a schedule.

Low-level C definition `void DPxEnableDoutBacklightPulse()`

See also:

`DPxDisableDoutBacklightPulse`, `DPxIsDoutBacklightPulse`

`_libdpx.`**`DPxEnableDoutButtonSchedules`**`()`

Enables automatic DOUT schedules upon DIN button presses.

Low-level C definition `void DPxEnableDoutButtonSchedules()`

See also:

`DPxDisableDoutButtonSchedules`, `DPxIsDoutButtonSchedules`

`_libdpx.`**`DPxEnableDoutDinLoopback`**`()`

Enables loopback between digital output ports and digital inputs.

Low-level C definition `void DPxEnableDoutDinLoopback()`

See also:

`DPxDisableDoutDinLoopback`, `DPxIsDoutDinLoopback` `_libdpx.`**`DPxEnableDoutPixelMode`**`()`

Enables pixel mode.

When this function is enabled, the digital outputs show the RGB value of first upper left pixel of the screen. In which case, digital outputs cannot be used for other purposes. This feature is only available on VIEWPixx with firmware revision 31 and higher.

Low-level C definition `void DPxEnableDoutPixelMode()`

See also:

`DPxDisableDoutPixelMode`, `DPxIsDoutPixelMode`

`_libdpx.`**`DPxEnableDoutSchedCountdown`**`()`

Enables Dout schedule count down.

SchedCount decrements at SchedRate, and schedule stops automatically when count hits 0.

Low-level C definition `void DPxEnableDoutSchedCountdown()`

See also:

`DPxDisableDoutSchedCountdown`, `DPxIsDoutSchedCountdown`

`_libdpx.`**`DPxEnableMicSchedCountdown`**`()`

Enables MIC schedule count down.

SchedCount increments at SchedRate, and schedule is stopped by calling SchedStop.

Low-level C definition `void DPxEnableMicSchedCountdown()`

`_libdpx.`**`DPxEnablePPxCeilingMount`**`()`

Enables the PROPixx Ceiling Mount mode.

Low-level C definition `void DPxEnablePPxCeilingMount(void)`

`_libdpx.`**`DPxEnablePPxHotSpotCorrection`**`()`

`_libdpx.`**`DPxEnablePPxLampLed`**`()`

Enables the lamp LED of the PROPixx.

Only available for PROPixx Revision 12 and higher.

Low-level C definition `void DPxEnablePPxLampLed(void)`

`_libdpx.`**`DPxEnablePPxQuietFanMode`**`()`

Enables the quiet fan mode on the PROPixx.

Enabling this mode reduces the speed of the fans to reduce noise. Only available for PROPixx Revision 19 and higher.

Low-level C definition `void DPxEnablePPxQuietFanMode(void)`

`_libdpx.`**`DPxEnablePPxRearProjection`**`()`

Enables the Rear Projection of the PROPixx.

Low-level C definition `void DPxEnablePPxRearProjection(void)`

`_libdpx.`**`DPxEnableTouchpixx`**`()`

Enables the TOUCHPixx touch panel hardware subsystem.

Low-level C definition `void DPxEnableTouchpixx()`

See also:

`DPxDisableTouchpixx`, `DPxIsTouchpixx`

`_libdpx.`**`DPxEnableTouchpixxLogContinuousMode`**`()`

Enables Touchpixx continuous logging mode.

TOUCHPixx logging returns continuous position updates during a panel press.

Low-level C definition `void DPxEnableTouchpixxLogContinuousMode()`

See also:

`DPxDisableTouchpixxLogContinuousMode`, `DPxIsTouchpixxLogContinuousMode`

`_libdpx.`**`DPxEnableTouchpixxLogEvents`**`()`

Enables log events mode.

Each Touchpixx transition is automatically logged. No schedule is required. Best way to log response buttons.

Low-level C definition `void DPxEnableTouchpixxLogEvents()`

See also:

`DPxDisableTouchpixxLogEvents`, `DPxIsTouchpixxLogEvents`

`_libdpx.`**`DPxEnableTouchpixxLogTimetags`**`()`

Enables Touchpixx timetag mode.

Each buffered Touchpixx sample is preceeded with a 64-bit nanosecond timetag.

Low-level C definition `void DPxEnableTouchpixxLogTimetags()`

See also:

`DPxDisableTouchpixxLogTimetags`, `DPxIsTouchpixxLogTimetags`

`_libdpx.`**`DPxEnableVidClutTransparencyColorMode`**`()`

Enables video CLUT transparency color mode.

Low-level C definition `void DPxEnableVidClutTransparencyColorMode()`

See also:

`DPxDisableVidClutTransparencyColorMode`, `DPxIsVidClutTransparencyColorMode`, `DPxGetVidClutTransparencyColor`, `DPxSetVidClutTransparencyColor`,

`_libdpx.`**`DPxEnableVidHorizOverlay`**`()`

Enables the Horizontal overlay

VGA 1 and VGA 2 both show an overlay composite of the left/right halves of the video image

Low-level C definition `void DPxEnableVidHorizOverlay()`

`_libdpx.`**`DPxEnableVidHorizSplit`**`()`

Enables Video Horizontal Split.

VGA 1 shows left half of video image, VGA 2 shows right half of video image. The two VGA outputs are perfectly synchronized.

Low-level C definition `void DPxEnableVidHorizSplit()`

`_libdpx.`**`DPxEnableVidLcd3D60Hz`**`()`

Enables 3D pixel polarity inversion

Low-level C definition `void DPxEnableVidLcd3D60Hz(void)`

`_libdpx.`**`DPxEnableVidPsyncBlankLine`**`()`

Enables the PSync Black Line

The sync raster line is always displayed black.

Low-level C definition `void DPxEnableVidPsyncBlankLine()`

`_libdpx.`**`DPxEnableVidPsyncSingleLine`**`()`

Enables Psync for Single (Raster) Line.

Low-level C definition `void DPxEnableVidPsyncSingleLine()`

`_libdpx.`**`DPxEnableVidRescanWarning`**`()`

Enables VIEWPixx Rescan Warning

Low-level C definition `void DPxEnableVidRescanWarning()`

`_libdpx.`**`DPxEnableVidScanningBacklight`**`()`

Enables VIEWPixx scanning backlight

Low-level C definition `void DPxEnableVidScanningBacklight()`

`_libdpx.`**`DPxEnableVidVertStereo`**`()`

Enables Vertical Stereo Video.

Top/bottom halves of input image are output in two sequencial video frames. VESA L/R output is set to 1 when first frame (left eye) is displayed, and set to 0 when second frame (right eye) is displayed.

Low-level C definition `void DPxEnableVidVertStereo()`

`_libdpx.`**`DPxEnableVidVesaBlueline`**`()`

Enables the Video blue line mode.

When enabled, the VESA 3D output interprets the middle pixel on last raster line as a blue line code. When disabled, the VESA 3D output is not dependent on video content.

Low-level C definition `void DPxEnableVidVesaBlueline()`

`_libdpx.`**`DPxEnableVidVesaFreeRun`**`()`

Enables PROPixx 3D VESA output freeRun enable bit.

Note: PROPixx Rev >= 7 only.

Low-level C definition `void DPxEnableVidVesaFreeRun(void)`

`_libdpx.`**`DPxGetAdcBuffBaseAddr`**`()`

Gets the ADC RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetDacBuffBaseAddr()`

See also:

`DPxSetAdcBuffBaseAddr`

`_libdpx.`**`DPxGetAdcBuffChanRef`**`(`*channel*`)`

Gets the reference associated with a channel.

Returns

reference – one of the following predefined constants:

- **gnd**: Referenced to ground.
- **diff**: Referenced to adjacent analog input.
- **ref0**: Referenced to REF0 analog input.

- **ref1**: Referenced to REF1 analog input.

Return type string

Low-level C definition `int DPxGetAdcBuffChanRef(int channel)`

`_libdpx.`**`DPxGetAdcBuffSize`**`()`

Gets the ADC RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetAdcBuffSize()`

See also:

`DPxSetAdcBuffSize`

`_libdpx.`**`DPxGetAdcBuffWriteAddr`**`()`

Gets RAM address from which next ADC datum will be written.

Returns Write address.

Return type int

Low-level C definition `unsigned DPxGetAdcBuffWriteAddr()`

See also:

`DPxSetAdcBuffWriteAddr`

`_libdpx.`**`DPxGetAdcNumChans`**`()`

Gets the number of channel available.

This method returns the number of ADC channels in the system (18 in current implementation)

Returns Number of channels.

Return type int

Low-level C definition `int DPxGetAdcNumChans()`

`_libdpx.`**`DPxGetAdcRange`**`(`*channel*`)`

Gets the voltage range.

This method returns the voltage range; +-10V for all channels

Parameters **channel** (*int*) – Channel number.

Returns Range of channel.

Return type int

Low-level C definition `void DPxGetAdcRange(int channel, double *minV, double *maxV)`

`_libdpx.`**`DPxGetAdcSchedCount`**`()`

Gets ADC schedule update count.

Returns Schedule sample count.

Return type int

Low-level C definition `unsigned DPxGetAdcSchedCount()`

See also:

`DPxSetAdcSchedCount`

`_libdpx.`**`DPxGetAdcSchedOnset`**`()`

Gets the nanosecond delay between schedule start and first ADC sample.

Returns onset.

Return type int

Low-level C definition `unsigned DPxGetAdcSchedOnset()`

See also:

`DPxSetAdcSchedOnset`

`_libdpx.`**`DPxGetAdcSchedRate`**`()`

This function gets the ADC schedule rate and the rate unit. Return value: (schedule rate, rate unit)

Low-level C definition `unsigned DPxGetAdcSchedRate(int *rateUnits)`

`_libdpx.`**`DPxGetAdcValue`**`(`*channel*`)`

Gets the value for one ADC channel.

This method returns the 16-bit 2's complement signed value for one ADC channel. Can be used on channels 0-17.

Parameters **channel** (*int*) – Channel number.

Returns Channel value.

Return type int

Low-level C definition `int DPxGetAdcValue(int channel)`

`_libdpx.`**`DPxGetAdcVoltage`**`(`*channel*`)`

Gets the voltage for an ADC channel.

Parameters **channel** (*int*) – Channel number.

Returns Voltage of channel.

Return type float

Low-level C definition `double DPxGetAdcVoltage(int channel)`

`_libdpx.`**`DPxGetAudBuffBaseAddr`**`()`

Gets the AUD RAM buffer start address.

Returns Base address

Return type int

Low-level C definition `unsigned DPxGetAudBuffBaseAddr()`

`_libdpx.`**`DPxGetAudBuffReadAddr`**`()`

Gets AUD address from which next AUD datum will be read.

Returns Read address.

Return type int

Low-level C definition `unsigned DPxGetAudBuffReadAddr()`

`_libdpx.`**`DPxGetAudBuffSize`**`()`

Gets the AUD RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetAudBuffSize()`

`_libdpx.`**`DPxGetAudCodecOutLeftVolume`**`(`*dBUnits=0*`)`
Gets the volume for the DATAPixx Audio OUT Left channel

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `double DPxGetAudCodecOutLeftVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudCodecOutRightVolume`**`(`*dBUnits=0*`)`
Gets the volume for the DATAPixx Audio OUT Right channel

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `double DPxGetAudCodecOutRightVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudCodecOutVolume`**`(`*dBUnits=0*`)`
Gets the volume for the DATAPixx Audio OUT Left and Right channels

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Returns A tuple containing floats: [left Speaker Volume, Right speaker Volume]

Low-level C definition `double DPxGetAudCodecOutVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudCodecSpeakerLeftVolume`**`(`*dBUnits=0*`)`
Gets the volume for the DATAPixx Speaker Left channel

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `double DPxGetAudCodecSpeakerLeftVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudCodecSpeakerRightVolume`**`(`*dBUnits=0*`)`
Gets the volume for the DATAPixx Speaker Right channel

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Low-level C definition `double DPxGetAudCodecSpeakerRightVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudCodecSpeakerVolume`**`(`*dBUnits=0*`)`
Gets volume for the DATAPixx Speaker Left and Right channels

Parameters **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Returns A tuple containing floats: [left Speaker Volume, Right speaker Volume]

Low-level C definition `double DPxGetAudCodecSpeakerVolume(int DBUnits)`

`_libdpx.`**`DPxGetAudGroupDelay`**`(`*sampleRate*`)`
Gets the CODEC Audio OUT group delay in seconds.

Parameters **sampleRate** (*float*) – The rate at which your schedule is running.

Returns delay in seconds.

Return type float

Low-level C definition `double DPxGetAudGroupDelay(double sampleRate)`

`_libdpx.`**`DPxGetAudLRMode`**`()`
Gets the audio schedule update mode.

Returns

Any of the following predefined constants.

- **mono**: Left schedule data goes to left and right channels.

- **left**: Each schedule data goes to left channel only.
- **right**: Each schedule data goes to right channel only.
- **stereo1**: Pairs of Left data are copied to left/right channels.
- **stereo2**: Left data goes to left channel, Right data goes to right.

Return type String

Low-level C definition `int DPxGetAudLRMode()`

`_libdpx.`**`DPxGetAudLeftValue`**`()`

Get the 16-bit 2's complement signed value for the Left audio output channel

Low-level C definition `int DPxGetAudLeftValue()`

`_libdpx.`**`DPxGetAudLeftVolume`**`()`

Get volume for the Left audio output channel, range 0-1

Low-level C definition `double DPxGetAudLeftVolume()`

`_libdpx.`**`DPxGetAudRightValue`**`()`

Get the 16-bit 2's complement signed value for the Right audio output channel

Low-level C definition `int DPxGetAudRightValue()`

`_libdpx.`**`DPxGetAudRightVolume`**`()`

Get volume for the Right audio output channel, range 0-1

Low-level C definition `double DPxGetAudRightVolume()`

`_libdpx.`**`DPxGetAudSchedCount`**`()`

Gets AUD schedule update count.

Returns The current MIC schedule count.

Return type int

Low-level C definition `unsigned DPxGetAudSchedCount()`

`_libdpx.`**`DPxGetAudSchedOnset`**`()`

Gets the nanosecond delay between schedule start and first AUD update.

Returns The nanosecond onset between the first update and the start of schedule.

Return type int

Low-level C definition `unsigned DPxGetAudSchedOnset()`

`_libdpx.`**`DPxGetAudSchedRate`**`()`

Gets the audio schedule update rate and optionally get rate units.

This method allows the user to get the audio's left schedule update rate and optionally get rate units. The return value is a tuple containing the rate and the rate unit.

The unit can be any of the following predefined constants.

- **hz**: Updates per second, maximum 96 kHz.
- **video**: Updates per video frame, maximum 96 kHz.
- **nano**: Update period in nanoseconds, minimum 10417 ns.

Returns Rate, Unit

Return type Tuple

Low-level C definition `unsigned DPxGetAudSchedRate(int *rateUnits)`

`_libdpx.`**`DPxGetAudVolume`**`()`
Gets the volume for both Left/Right audio channels

Returns A tuple containing floats: [left Speaker Volume, Right speaker Volume]

Low-level C definition `double DPxGetAudVolume()`

`_libdpx.`**`DPxGetAuxBuffBaseAddr`**`()`
Gets the AUX RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetAuxBuffBaseAddr()`

`_libdpx.`**`DPxGetAuxBuffReadAddr`**`()`
Gets RAM address from which next AUX datum will be read.

Returns Read address.

Return type int

Low-level C definition `unsigned DPxGetAuxBuffReadAddr()`

`_libdpx.`**`DPxGetAuxBuffSize`**`()`
Gets the AUX RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetAuxBuffSize()`

`_libdpx.`**`DPxGetAuxSchedCount`**`()`
Gets AUX schedule update count.

Returns The schdule update total count.

Return type int

Low-level C definition `unsigned DPxGetAuxSchedCount()`

`_libdpx.`**`DPxGetAuxSchedOnset`**`()`
Gets the nanosecond delay between schedule start and the first AUX update.

Returns The nanosecond onset between the first update and the start of the schedule.

Return type int

Low-level C definition `unsigned DPxGetAuxSchedOnset()`

`_libdpx.`**`DPxGetAuxSchedRate`**`()`
Gets AUX schedule update rate and the rate units.

Returns rate and unit.

Return type tuple

Low-level C definition `unsigned DPxGetAuxSchedRate(int *rateUnits))`

`_libdpx.`**`DPxGetDacBuffBaseAddr`**`()`
Gets the DAC RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetDacBuffBaseAddr()`

See also:

`DPxSetDacBuffBaseAddr`

`_libdpx.`**`DPxGetDacBuffReadAddr`**`()`

Gets RAM address from which next DAC datum will be read.

Returns Read address.

Return type int

Low-level C definition `unsigned DPxGetDacBuffReadAddr()`

See also:

`DPxSetDacBuffReadAddr`

`_libdpx.`**`DPxGetDacBuffSize`**`()`

Gets the DAC RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetDacBuffSize()`

See also:

`DPxSetDacBuffSize`

`_libdpx.`**`DPxGetDacNumChans`**`()`

Gets the number of channels available.

This method returns the number of DAC channels in the system (4 in current implementation)

Returns Number of channels.

Return type int

Low-level C definition `int DPxGetDacNumChans()`

`_libdpx.`**`DPxGetDacRange`**`(`*channel*`)`

Gets the voltage range.

This method returns the voltage range; For a VIEWPixx: +-10V, for aDATAPixx: +-10V for ch0/1, +-5V for ch2/3

Parameters **channel** (*int*) – Channel number

Returns Number of channel.

Return type int

Low-level C definition `DPxGetDacRange(int channel, double *minV, double *maxV)`

`_libdpx.`**`DPxGetDacSchedCount`**`()`

Gets DAC schedule update count.

Returns Schedule sample count.

Return type int

Low-level C definition `unsigned DPxGetDacSchedCount()`

See also:

`DPxSetDacSchedCount`

`_libdpx.`**`DPxGetDacSchedOnset`**`()`

Gets the nanosecond delay between the schedule start and first DAC update.

Returns onset.

Return type int

Low-level C definition `unsigned DPxGetDacSchedOnset()`

See also:

`DPxSetDacSchedOnset`

`_libdpx.`**`DPxGetDacSchedRate`**`()`

Gets DAC schedule update rate and the rate units.

Returns rate and unit.

Return type tuple

Low-level C definition `unsigned DPxGetDacSchedRate(int *rateUnits)`

See also:

`DPxSetDacSchedRate`

`_libdpx.`**`DPxGetDacValue`**`(`*channel*`)`

Gets the value for one DAC channel.

Parameters **channel** (*int*) – Channel number.

Returns A 16-bit 2's complement signed value.

Return type int

Low-level C definition `int DPxGetDacValue(int channel)`

See also:

`DPxSetDacValue`

`_libdpx.`**`DPxGetDacVoltage`**`(`*channel*`)`

Gets the value for one DAC channel.

Parameters **channel** (*int*) – Channel number.

For channel 0 and 1: $\pm$ 10V. For channel 2 and 3: $\pm$ 5V.

Returns Voltage.

Return type float

Low-level C definition `double DPxGetDacVoltage(int channel)`

See also:

`DPxSetDacValue`

`_libdpx.`**`DPxGetDebug`**`()`

Returns the current debug level

Returns Debug level

Low-level C definition `int DPxGetDebug()`

`_libdpx.`**`DPxGetDinBuffBaseAddr`**`()`

Gets the DIN RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetDinBuffBaseAddr()`

See also:

`DPxSetDinBuffBaseAddr`

`_libdpx.`**`DPxGetDinBuffSize`**`()`

Gets the Din RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetDinBuffSize()`

See also:

`DPxSetDinBuffSize`

`_libdpx.`**`DPxGetDinBuffWriteAddr`**`()`

Gets RAM address from which next DIN datum will be written.

Returns Write address.

Return type int

Low-level C definition `unsigned DPxGetDinBuffWriteAddr()`

See also:

`DPxSetDinBuffWriteAddr`

`_libdpx.`**`DPxGetDinDataDir`**`()`

Gets the port direction mask.

Returns Bit set to 1 is an enabled port. Bit set to 0 is a disabled port.

Return type int

Low-level C definition `int DPxGetDinDataDir()`

See also:

`DPxSetDinDataDir`, `DPxSetDinDataDir`

`_libdpx.`**`DPxGetDinDataOut`**`()`

Gets the data which is being driven on each output port.

This function allows the user to get the data which is currently driven on the output port.

Returns Data which is being driven on each output port.

Return type int

Low-level C definition `int DPxGetDinDataOut()`

See also:

`DPxSetDinDataDir`, `DPxSetDinDataDir`

`_libdpx.`**`DPxGetDinDataOutStrength`**`()`

Gets the strength of the driven outputs.

This function allows the user to get the strength currently driven by the outputs. The implementation actual values uses 1/16 up to 16/16. So minimum strength will be 0.0625 and maximum will be 1. The strength can be increased by 0.0625 up to 1.

Returns Any value in a range of 0 to 1.

Return type float

Low-level C definition `double DPxSetDinDataOutStrength()`

See also:

`DPxGetDinDataOutStrength`

`_libdpx.`**`DPxGetDinNumBits`**`()`

Gets the number of bits available.

This method returns the number of digital input bits in the system (24 in current implementation).

Returns Number of bits.

Return type int

Low-level C definition `int DPxGetDinNumBits()`

`_libdpx.`**`DPxGetDinSchedCount`**`()`

Gets Din schedule update count.

Returns Schedule sample count.

Return type int

Low-level C definition `unsigned DPxGetDinSchedCount()`

See also:

`DPxSetDinSchedCount`

`_libdpx.`**`DPxGetDinSchedOnset`**`()`

Gets the nanosecond delay between schedule start and first Din sample.

Returns onset.

Return type int

Low-level C definition `unsigned DPxGetDinSchedOnset()`

See also:

`DPxSetDinSchedOnset`

`_libdpx.`**`DPxGetDinSchedRate`**`()`

This function gets the Din schedule rate and the rate unit. Return value: (schedule rate, rate unit)

Low-level C definition `unsigned DPxGetDinSchedRate(int *rateUnits)`

`_libdpx.`**`DPxGetDinValue`**`()`

Gets the values of the 24 DIN bits.

Returns Bit values.

Return type int

Low-level C definition `int DPxGetDinValue()`

`_libdpx.`**`DPxGetDoutBuffBaseAddr`**`()`

Gets the Dout RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetDoutBuffBaseAddr()`

See also:

`DPxSetDoutBuffBaseAddr`

`_libdpx.`**`DPxGetDoutBuffReadAddr`**`()`

Gets RAM address from which next Dout datum will be read.

Returns Read address.

Return type int

Low-level C definition `unsigned DPxGetDoutBuffReadAddr()`

See also:

`DPxSetDoutBuffReadAddr`

`_libdpx.`**`DPxGetDoutBuffSize`**`()`

Gets the Dout RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetDoutBuffSize()`

See also:

`DPxSetDoutBuffSize`

`_libdpx.`**`DPxGetDoutNumBits`**`()`

Gets the number of bits available.

This method returns the number of digital output bits in the system (24 in current implementation).

Returns Number of bits.

Return type int

Low-level C definition `int DPxGetDoutNumBits()`

`_libdpx.`**`DPxGetDoutSchedCount`**`()`

Gets Dout schedule update count.

Returns Schedule sample count.

Return type int

Low-level C definition `unsigned DPxGetDoutSchedCount()`

See also:

`DPxSetDoutSchedCount`

`_libdpx.`**`DPxGetDoutSchedOnset`**`()`

Gets the nanosecond delay between schedule start and first Dout update.

Returns onset.

Return type int

Low-level C definition `unsigned DPxGetDoutSchedOnset()`

See also:

`DPxSetDoutSchedOnset`

`_libdpx.`**`DPxGetDoutSchedRate`**`()`

Gets Dout schedule update rate and the rate units.

Returns rate and unit.

Return type tuple

Low-level C definition `unsigned DPxGetDoutSchedRate(int *rateUnits)`

See also:

`DPxSetDoutSchedRate`

`_libdpx.`**`DPxGetDoutValue`**`()`

Gets the values of the 24 Dout bits.

Returns Bit values.

Return type int

Low-level C definition `int DPxGetDoutValue()`

`_libdpx.`**`DPxGetError`**`()`

Returns the error code

Returns Error code if an error occured, otherwise 0.

Low-level C definition `int DPxGetError()`

`_libdpx.`**`DPxGetErrorString`**`()`

Returns the current error string

Returns Latest error string

Low-level C definition `const char* DPxGetErrorString()`

`_libdpx.`**`DPxGetFirmwareRev`**`()`

Gets the VPixx Device firmware revision.

Returns Value higher than 0.

Return type int

Low-level C definition `int DPxGetFirmwareRev()`

`_libdpx.`**`DPxGetID`**`()`

Gets the VPixx device identifier code.

Returns Value higher than 0.

Return type int

Low-level C definition `int DPxGetID()`

`_libdpx.`**`DPxGetMarker`**`()`

Gets the current marker from the register.

This function allows the user to get the marker value previously latched.

Returns Time in seconds.

Return type float

Low-level C definition `double DPxGetMarker()`

See also:

`DPxSetMarker`, `DPxGetNanoMarker`

`_libdpx.`**`DPxGetMicBuffBaseAddr`**`()`

Gets the MIC RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetMicBuffBaseAddr()`

`_libdpx.`**`DPxGetMicBuffSize`**`()`

Gets the DAC RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetMicBuffSize()`

`_libdpx.`**`DPxGetMicBuffWriteAddr`**`()`

Gets RAM address from which next MIC datum will be written.

Returns Write address.

Return type int

Low-level C definition `unsigned DPxGetMicBuffWriteAddr()`

`_libdpx.`**`DPxGetMicGroupDelay`**`(`*sampleRate*`)`

Gets the CODEC Audio OUT group delay in seconds.

Parameters **sampleRate** (*float*) – The sample rate of your schedule

Returns delay in seconds.

Return type float

Low-level C definition `double DPxGetMicGroupDelay(double sampleRate)`

`_libdpx.`**`DPxGetMicLRMode`**`()`

Gets the microphone Left/Right configuration mode.

This method allows the user to get the microphone left and right channels schedule buffer mode.

Returns

Any of the following predefined constants.

- **mono**: Mono data is written to the schedule buffer. The average of the Left/Right CODEC data.
- **left**: Left data is written to the schedule buffer.
- **right**: Right data is written to the schedule buffer.
- **stereo**: Left and Right data are both written to the schedule buffer.

Return type String

Low-level C definition `int DPxGetMicLRMode()`

`_libdpx.`**`DPxGetMicLeftValue`**`()`

Get the 16-bit 2's complement signed value for left MIC channel

Low-level C definition `int DPxGetMicLeftValue()`

`_libdpx.`**`DPxGetMicRightValue`**`()`

Get the 16-bit 2's complement signed value for right MIC channel

Low-level C definition `int DPxGetMicRightValue()`

`_libdpx.`**`DPxGetMicSchedCount`**`()`

Gets MIC schedule update count.

Returns The current MIC schedule count.

Return type int

Low-level C definition `unsigned DPxGetMicSchedCount()`

`_libdpx.`**`DPxGetMicSchedOnset`**`()`

Gets the nanosecond delay between schedule start and first MIC update.

Returns The nanosecond onset between the first update and the start of schedule.

Return type int

Low-level C definition `unsigned DPxGetMicSchedOnset()`

`_libdpx.`**`DPxGetMicSchedRate`**`()`

Gets MIC schedule update rate and the rate units.

Returns rate and unit.

Return type tuple

Low-level C definition `unsigned DPxGetDacSchedRate(int *rateUnits)`

`_libdpx.`**`DPxGetMicSource`**`(`*dBUnits=0*`)`

Gets the source and the gain of the microphone input.

Parameters **dBUnits** (*int, optional*) – Set to non-zero to return the gain in dB. Defaults to 0.

Returns A list containing the [gain value, microphone source]

Low-level C definition `int DPxGetMicSource(int DBUnits)`

`_libdpx.`**`DPxGetNanoMarker`**`()`

Gets the current marker from the register with high precision.

This function allows the user to get the marker value previously latched.

Returns Time in seconds.

Return type tuple

Low-level C definition `void DPxGetNanoMarker(unsigned *nanoHigh32, unsigned *nanoLow32)`

See also:

`DPxSetMarker`, `DPxGetNanoMarker`

`_libdpx.`**`DPxGetNanoTime`**`()`

Gets the current time since power up with high precision.

This function allows the user to get the marker value previously latched.

Returns Time in seconds.

Return type tuple

Low-level C definition `void DPxGetNanoTime(unsigned *nanoHigh32, unsigned *nanoLow32)`

See also:

`DPxSetMarker`, `DPxGetNanoMarker`

`_libdpx.`**`DPxGetPPx3dCrosstalk`**`()`

Gets 3D crosstalk (0-1) which is being subtracted from stereoscopic stimuli.

Warning: This only works with RB3D mode and requires revision 6 of the PROPixx.

Returns A double value for the 3D Crosstalk.

Low-level C definition `double DPxGetPPx3dCrosstalk(void)`

`_libdpx.`**`DPxGetPPxDlpSeqPgrm`**`()`

Get PROPixx DLP Sequencer program.

This method returns the current program loaded in the PROPixx sequencer.

Returns

Any of the following predefined constants.

- **RGB**: Default RGB
- **RB3D**: R/B channels drive grayscale 3D
- **RGB240**: Only show the frame for 1/2 a 120 Hz frame duration.
- **RGB180**: Only show the frame for 2/3 of a 120 Hz frame duration.
- **QUAD4X**: Display quadrants are projected at 4x refresh rate.
- **QUAD12X**: Display quadrants are projected at 12x refresh rate with grayscales.
- **GREY3X**: Converts 640x1080@360Hz RGB to 1920x1080@720Hz Grayscale with blank frames.

Return type String

Low-level C definition `int DPxGetPPxDlpSeqPgrm(void)`

`_libdpx.`**`DPxGetPPxFanPwm`**`()`

Returns the Fans PWN.

Returns Current fan PWN as a double.

Low-level C definition `double DPxGetPPxFanPwm(void)`

`_libdpx.`**`DPxGetPPxFanTachometer`**`(`*fanNum*`)`

Returns the speed at which a fan is rotating.

Parameters **fanNum** (*int*) – The number of the fan.

Low-level C definition `double DPxGetPPxFanTachometer(int fanNum)`

`_libdpx.`**`DPxGetPPxHotSpotCenter`**`()`

`_libdpx.`**`DPxGetPPxLedCurrent`**`(`*ledNum*`)`

Get PROPixx LED Current.

Parameters **ledNum** (*int*) – The number of the LED.

Returns The value of the current of the selected LED as a double in Amps.

Low-level C definition `double DPxGetPPxLedCurrent(int ledNum)` `_libdpx.`**`DPxGetPPxLedMask`**`()`

Get the PROPixx LED mask.

Only available for PROPixx Revision 33 and higher.

Returns

Any of the following number.

- **0**: All LEDs are on.
- **1**: RED is turned off.
- **2**: GREEN is turned off.
- **3**: RED and GREEN are turned off.
- **4**: BLUE is turned off.
- **5**: RED and BLUE are turned off.
- **6**: BLUE and GREEN are turned off.
- **7**: ALL LEDs are off.

Return type Int

Low-level C definition `int DPxGetPPxDlpSeqPgrm()`

`_libdpx.`**`DPxGetPPxTemperature`**`(`*tempNum*`)`

Get a PROPixx temperature for a given sensor number.

Parameters **tempNum** (*int*) – Number of the sensor.

Returns The temperature in Celcius of the sensor.

Low-level C definition `double DPxGetPPxTemperature(int tempNum)`

`_libdpx.`**`DPxGetPPxVoltageMonitor`**`(`*voltageNum*`)`

Gets the Voltage for the given sensor.

Parameters **voltageNum** (*int*) – The number of the sensor.

Returns The volatage in Volts for the chosen monitor.

Low-level C definition `double DPxGetPPxVoltageMonitor(int voltageNum)`

`_libdpx.`**`DPxGetPartNumber`**`()`

Gets the integer part number of the VPixx Device.

Returns Value higher than 0.

Return type int

Low-level C definition `int DPxGetPartNumber()`

`_libdpx.`**`DPxGetRamSize`**`()`

Gets the number of bytes of RAM in the VPixx device.

Returns Value higher than 0.

Return type int

Low-level C definition `int DPxGetRamSize()`

`_libdpx.`**`DPxGetSupply2Current`**`()`

Gets the current being supplied from the +12V power supply.

Returns Value higher than 0 in Amperes.

Return type float

Low-level C definition `double DPxGetSupply2Current()`

`_libdpx.`**`DPxGetSupply2Voltage`**`()`

Gets the voltage being supplied from the 12V power supply.

Returns Value higher than 0 in Volts.

Return type float

Low-level C definition `double DPxGetSupply2Voltage()`

`_libdpx.`**`DPxGetSupplyCurrent`**`()`

Gets the current being supplied from the 5V power supply.

Returns Value higher than 0 in Amperes.

Return type float

Low-level C definition `double DPxGetSupplyCurrent()`

`_libdpx.`**`DPxGetSupplyVoltage`**`()`

Gets the voltage being supplied from +5V supply.

Returns Value higher than 0 in Volt.

Return type float

Low-level C definition `double DPxGetSupplyVoltage()`

`_libdpx.`**`DPxGetTemp2Celcius`**`()`

Gets the temperature inside a VPixx device.

This function gets the board temperature of any VPixx device except the DATAPixx.

Returns Temperature in degrees Celsius.

Return type float

Low-level C definition `double DPxGetTemp2Celcius()`

`_libdpx.`**`DPxGetTemp3Celcius`**`()`

Gets the FPGA temperature inside a VPixx device.

This function cannot be used with a DATAPixx.

Returns Temperature in degrees Celsius.

Return type float

Low-level C definition `double DPxGetTemp3Celcius()`

`_libdpx.`**`DPxGetTempCelcius`**`()`

Gets the temperature inside a VPixx device chassis.

When used with a PROPixx, this function gets the temperature of the Receiver DVI. When used with other devices, it gets the chassis temperature.

Returns Temperature in degrees Celsius.

Return type float

Low-level C definition `double DPxGetTempCelcius()`

`_libdpx.`**`DPxGetTempFarenheit`**`()`

Gets the temperature inside a VPixx device in Fahrenheit.

Returns Temperature in degrees Fahrenheit.

Return type float

Low-level C definition `double DPxGetTempFarenheit()`

`_libdpx.`**`DPxGetTime`**`()`

Gets the device time since last power up.

Returns Time in seconds.

Return type float

Low-level C definition `double DPxGetTime()`

`_libdpx.`**`DPxGetTouchpixxBuffBaseAddr`**`()`

Gets the Touchpixx RAM buffer start address.

Returns Base address.

Return type int

Low-level C definition `unsigned DPxGetDacBuffBaseAddr()`

See also:

`DPxSetTouchpixxBuffBaseAddr`

`_libdpx.`**`DPxGetTouchpixxBuffSize`**`()`

Gets the Touchpixx RAM buffer size in bytes.

Returns buffer size.

Return type int

Low-level C definition `unsigned DPxGetTouchpixxBuffSize()`

See also:

`DPxSetTouchpixxBuffSize`

`_libdpx.`**`DPxGetTouchpixxBuffWriteAddr`**`()`

Gets RAM address from which the next TOUCHPixx datum will be written.

Returns Write address.

Return type int

Low-level C definition `unsigned DPxGetTouchpixxBuffWriteAddr()`

See also:

`DPxSetTouchpixxBuffWriteAddr`

`_libdpx.`**`DPxGetTouchpixxCoords`**`()`

Gets the current touch panel `(X,Y)` coordinates. Returns `(0,0)` if panel not pressed.

Returns A tuple that contains the current pressed `(X,Y)` coordinate. `(0,0)` is returned when nothing is pressed. If there are multiple presses, it returns an average.

Low-level C definition `void DPxGetTouchpixxCoords(int* x, int* y)`

`_libdpx.`**`DPxGetTouchpixxStabilizeDuration`**`()`

Gets the Touchpixx stabilization duration.

Gets the duration in seconds that TOUCHPixx panel coordinates must be stable before being recognized as a touch in `DPxGetTouchpixxCoords`

Returns duration in seconds.

Return type float

Low-level C definition `unsigned DPxGetTouchpixxStabilizeDuration()`

See also:

`DPxSetTouchpixxStabilizeDuration`

`_libdpx.`**`DPxGetVidBacklightIntensity`**`()`

Returns the display current back light intensity.

Returns An integer between 0 and 255.

Example

```
>>> print my_device.getBacklightIntensity()
255
```

See also:

`DPxSetBacklightIntensity`

Low-level C definition `int DPxGetVidBacklightIntensity()`

`_libdpx.`**`DPxGetVidClutTransparencyColor`**`()`

Get 48-bit RGB video CLUT transparency color

This function allows the user to know the current register value for the RGB video CLUT transparency color. The returned value is a tupple which contains the red, green and blue values.

Returns **red** – Pixel color value for the red channel. green (int): Pixel color value for the green channel. blue (int): Pixel color value for the blue channel.

Return type int

Low-level C definition `void DPxGetVidClutTransparencyColor(UInt16* red, UInt16* green, UInt16* blue)`

See also:

`DPxSetVidClutTransparencyColor`

`_libdpx.`**`DPxGetVidDotFreq`**`()`

Gets the dot frequency for the device in Hz.

The dot frequency represents the time taken for a pixel to be switched on. It is calculated by considering all pixels (including those in blanking time) times the refresh rate.

Low-level C definition `double DPxGetVidDotFreq()`

`_libdpx.`**`DPxGetVidHActive`**`()`

Gets the number of visible pixels in one horizontal scan line.

Low-level C definition `int DPxGetVidHActive()`

`_libdpx.`**`DPxGetVidHFreq`**`()`

Gets the video horizontal line rate in Hz

The Horizontal line rate in Hz, which represents the time taken for a whole horizontal line to be switched on, including the blanking time.

Low-level C definition `double DPxGetVidHFreq()`

`_libdpx.`**`DPxGetVidHTotal`**`()`

Get the number of video dot times in one horizontal scan line (includes horizontal blanking interval)

Low-level C definition `int DPxGetVidHTotal()`

`_libdpx.`**`DPxGetVidHorizOverlayBounds`**`()`

Gets the bounding rectangle of horizontal overlay window

Returns A tuple of the form (X1, Y1, X2, Y2)

Low-level C definition `void DPxGetVidHorizOverlayBounds(int* X1, int* Y1, int* X2, int* Y2)`

`_libdpx.`**`DPxGetVidLine`**`(`*Hex=False*`)`

Reads pixels from the VPixx device line buffer, and returns a list containing the data. For each pixel, the buffer contains 16 bits R/G/B/U (where U is thrown away). The returned data is a list containing three lists for the respective R/G/B colors.

Parameters Hex (*bool, optional*) – True returns the value in hexadecimal. Everything else will return the value in decimal.

Returns lists of list: A list which has [[RED], [GREEN], [BLUE]]

Low-level C definition `short* DPxGetVidLine()`

`_libdpx.`**`DPxGetVidMode`**`()`

Gets the video processing mode.

Allows the user to know if it is in the correct mode or which mode is currently used on the device.

Returns

Any of the following predefined constants.

- **L48**: DVI RED[7:0] is used as an index into a 256-entry 16-bit RGB colour lookup table.
- **M16**: DVI RED[7:0] & GREEN[7:0] concatenate into a VGA 16-bit value sent to all three RGB components.
- **C48**: Even/Odd pixel RED/GREEN/BLUE[7:0] concatenate to generate 16-bit RGB components at half the horizontal resolution.
- **L48D**: DVI RED[7:4] & GREEN[7:4] concatenate to form an 8-bit index into a 256-entry 16-bit RGB colour lookup table.
- **M16D**: DVI RED[7:3] & GREEN[7:3] & BLUE[7:2] concatenate into a VGA 16-bit value sent to all three RGB components.
- **C36D**: Even/Odd pixel RED/GREEN/BLUE[7:2] concatenate to generate 12-bit RGB components at half the horizontal resolution.
- **C24**: Straight passthrough from DVI 8-bit (or HDMI "deep" 10/12-bit) RGB to VGA 8/10/12-bit RGB.

Return type String

Low-level C definition `int DPxGetVidMode()`

`_libdpx.`**`DPxGetVidPsyncRasterLine`**`()`

Gets the raster line on which pixel sync sequence is expected.

Returns An integer which represents the line which has the PSync.

Low-level C definition `int DPxGetVidPsyncRasterLine()`

`_libdpx.`**`DPxGetVidSource`**`(`*return_str=False*`)`

Gets source of video being displayed.

Parameters return_str (*Bool*) – When True, the return value is a string describing the video source used. Else, an integer is returned.

Returns

vidSource – The source currently displayed.

- **DVI**: Monitor displays DVI signal.
- **SWTP**: Software test pattern showing image from RAM.
- **SWTP 3D**: 3D Software test pattern flipping between left/right eye images from RAM.
- **RGB SQUARES**: RGB ramps.
- **GRAY**: Uniform gray display having 12-bit intensity.
- **BAR**: Drifting bar.
- **BAR2**: Drifting bar.
- **DOTS**: Drifting dots.
- **RAMP**: Drifting ramp, with dots advancing `x*2` pixels per video frame, where `x` is a 4-bit signed.
- **RGB**: Uniform display with 8-bit intensity nn, send to RGBA channels enabled by mask m.
- **PROJ**: Projector Hardware test pattern.

Return type str

Low-level C definition `int DPxGetVidSource()`

`_libdpx.`**`DPxGetVidVActive`**`()`

Gets number of visible lines in one vertical frame.

Low-level C definition `int DPxGetVidVActive()`

`_libdpx.`**`DPxGetVidVFreq`**`()`

Gets video vertical line rate in Hz.

The vertical line rate in Hz, which represents the time taken for a whole vertical line to be switched on, including the blanking time.

Low-level C definition `double DPxGetVidVFreq()`

`_libdpx.`**`DPxGetVidVPeriod`**`()`

Gets video vertical frame period in nanoseconds

The period is the inverse of the frequency.

Low-level C definition `unsigned DPxGetVidVPeriod()`

`_libdpx.`**`DPxGetVidVTotal`**`()`

Gets number of video lines in one vertical frame (includes vertical blanking interval)

Low-level C definition `int DPxGetVidVTotal()`

`_libdpx.`**`DPxGetVidVesaPhase`**`()`

Gets the 8 bits unsigned phase of the VESA 3D waveform.

Returns **phase** – Phase of the VESA 3D waveform.

Return type int

Low-level C definition `int DPxGetVidVesaPhase()`

`_libdpx.`**`DPxGetVidVesaWaveform`**`()`

Gets the waveform which is being sent to the DATAPixx VESA 3D connector.

Returns

Any of the following predefined constants.

- **LR**: VESA port drives straight L/R squarewave for 3rd party emitter.
- **CRYSTALEYES**: VESA port drives 3DPixx IR emitter for CrystalEyes 3D goggles.
- **NVIDIA**: VESA port drives 3DPixx IR emitter for NVIDIA 3D goggles.

Return type String

Low-level C definition `int DPxGetVidVesaWaveform()`

`_libdpx.`**`DPxInitAudCodec`**`()`

Initialize the required subsystems to play audio.

You are required to call this once before other audio or microphone routines to configure initial audio CODEC state.

Low-level C definition `void DPxInitAudCodec()`

`_libdpx.`**`DPxIs5VFault`**`()`

Verifies the state of the 5V power supply.

This function allows the user to know if the VESA and Analog +5V input/output pins are trying to draw more than 500 mA.

Returns 0 if the current is normal, non-zero otherwise (too much current drawn).

Return type int

Low-level C definition `int DPxIs5VFault()`

`_libdpx.`**`DPxIsAdcBuffChan`**`(`*channel*`)`

Verifies if RAM buffering is enabled for an ADC channel.

This function is only available for channels 0-15.

Parameters **channel** (*int*) – Channel number.

Returns Non-zero if RAM buffering is enabled for an ADC channel.

Return type int

Low-level C definition `int DPxIsAdcBuffChan(int channel)`

See also:

`DPxEnableAdcBuffChan`, `DPxDisableAdcBuffChan`

`_libdpx.`**`DPxIsAdcCalibRaw`**`()`

Verifies if the hardware calibration mode is enabled.

Returns Non-zero if ADC data is bypassing hardware calibration.

Return type int

Low-level C definition `int DPxIsAdcCalibRaw()`

See also:

`DPxEnableAdcCalibRaw`, `DPxDisableAdcCalibRaw`

`_libdpx.`**`DPxIsAdcFreeRun`**`()`

Verifies if the loopback between ADC and DAC is enabled.

Returns Non-zero if ADCs are performing continuous conversions.

Return type int

Low-level C definition `void DPxIsAdcFreeRun()`

See also:

`DPxEnableDacAdcLoopback`, `DPxDisableAdcFreeRun`

`_libdpx.`**`DPxIsAdcLogTimetags`**`()`

Verifies if the ADC timetag mode is enabled.

Returns Non-zero if buffered data is preceeded with nanosecond timetag.

Return type int

Low-level C definition `int DPxIsAdcLogTimetags()`

See also:

`DPxDisableAdcLogTimetags`, `DPxIsAdcLogTimetags`

`_libdpx.`**`DPxIsAdcSchedCountdown`**`()`

Verifies if RAM buffering is enabled for an ADC channel.

Returns Non-zero if SchedCount decrements to 0 and automatically stops schedule.

Return type int

Low-level C definition `int DPxIsAdcSchedCountdown()`

See also:

`DPxEnableAdcSchedCountdown`, `DPxDisableAdcSchedCountdown`

`_libdpx.`**`DPxIsAdcSchedRunning`**`()`

Verifies if an ADC schedule is currently running.

Returns Non-zero if an ADC schedule is currently running, zero if there is one.

Return type int

Low-level C definition `int DPxIsAdcSchedRunning()`

See also:

`DPxStartAdcSched`, `DPxStopAdcSched`

`_libdpx.`**`DPxIsAudMicLoopback`**`()`

Returns non-zero if microphone inputs are driven by audio outputs.

Low-level C definition `int DPxIsAudMicLoopback()`

`_libdpx.`**`DPxIsAudSchedCountdown`**`()`

Returns non-zero if SchedCount decrements to 0 and automatically stops schedule.

Low-level C definition `int DPxIsAudSchedCountdown()`

`_libdpx.`**`DPxIsAudSchedRunning`**`()`

Returns non-zero if AUD schedule is currently running.

Low-level C definition `int DPxIsAudSchedRunning()`

`_libdpx.`**`DPxIsAuxSchedCountdown`**`()`

Returns non-zero if SchedCount decrements to 0 and automatically stops the schedule.

Low-level C definition `int DPxIsAuxSchedCountdown()`

`_libdpx.`**`DPxIsAuxSchedRunning`**`()`

Returns non-zero if an AUX schedule is currently running.

Low-level C definition `int DPxIsAuxSchedRunning()`

`_libdpx.`**`DPxIsCustomStartupConfig`**`()`

Returns True if VPixx device has loaded custom startup register values

Note: PROPixx Rev >= 8 only and VIEWPixx/PROPixxCTRL Rev >= 26 only

Low-level C definition `int DPxIsCustomStartupConfig(void)`

`_libdpx.`**`DPxIsDacAdcLoopback`**`()`

Verifies if the loopback between ADC and DAC is enabled.

Returns Non-zero if ADC inputs are looped back from DAC outputs.

Return type int

Low-level C definition `void DPxIsDacAdcLoopback()`

See also:

`DPxEnableDacAdcLoopback`, `DPxIsDacAdcLoopback`

`_libdpx.`**`DPxIsDacBuffChan`**`(`*channel*`)`

Verifies if RAM buffering is enabled for a DAC channel.

Parameters **channel** (*int*) – Channel number.

Returns Non-zero if RAM buffering is enabled for a DAC channel.

Return type int

Low-level C definition `int DPxIsDacBuffChan(int channel)`

See also:

`DPxEnableDacCalibRaw`, `DPxDisableDacCalibRaw`

`_libdpx.`**`DPxIsDacCalibRaw`**`()`

Verifies if the hardware calibration mode is enabled.

Returns Non-zero if DAC data is bypassing hardware calibration.

Return type int

Low-level C definition `int DPxIsDacCalibRaw()`

See also:

`DPxEnableDacCalibRaw`, `DPxDisableDacCalibRaw`

`_libdpx.`**`DPxIsDacSchedCountdown`**`()`

Verifies if RAM buffering is enabled for a DAC channel.

Returns Non-zero if SchedCount decrements to 0 and automatically stops schedule.

Return type int

Low-level C definition `int DPxIsDacSchedCountdown()`

See also:

`DPxEnableDacSchedCountdown`, `DPxDisableDacSchedCountdown`

`_libdpx.`**`DPxIsDacSchedRunning`**`()`

Verifies if a DAC schedule is currently running.

Returns Non-zero if a DAC schedule is currently running, zero otherwise.

Return type int

Low-level C definition `int DPxIsDacSchedRunning()`

See also:

`DPxStartDacSched`, `DPxStopDacSched`

`_libdpx.`**`DPxIsDatapixx`**`()`

Verifies if the device is a DATAPixx.

Returns Non-zero if a DATAPixx was found, zero otherwise.

Return type int

Low-level C definition `int DPxIsDatapixx()`

`_libdpx.`**`DPxIsDatapixx2`**`()`

Verifies if the device is a DATAPixx2.

Returns Non-zero if a DATAPixx2 was found, zero otherwise.

Return type int

Low-level C definition `int DPxIsDatapixx2()`

`_libdpx.`**`DPxIsDinDebounce`**`()`

Verifies if the DIN debounce mode is enabled.

Returns Non-zero if DIN transitions are being debounced.

Return type int

Low-level C definition `int DPxIsDinDebounce()`

See also:

`DPxEnableDinDebounce`, `DPxDisableDinDebounce`

`_libdpx.`**`DPxIsDinLogEvents`**`()`

Verifies if the Din timetag mode is enable.

Returns Non-zero if DIN transitions are being logged to RAM buffer.

Return type int

Low-level C definition `int DPxIsDinLogEvents()`

See also:

`DPxDisableDinLogEvents`, `DPxEnableDinLogEvents`

`_libdpx.`**`DPxIsDinLogTimetags`**`()`

Verifies if the Din timetag mode is enabled.

Returns Non-zero if buffered data is preceeded with nanosecond timetag.

Return type int

Low-level C definition `int DPxIsDinLogTimetags()`

See also:

`DPxDisableDinLogTimetags`, `DPxIsDinLogTimetags`

`_libdpx.`**`DPxIsDinSchedCountdown`**`()`

Verifies if RAM buffering is enabled for a Din channel.

Returns Non-zero if SchedCount decrements to 0 and automatically stops schedule.

Return type int

Low-level C definition `int DPxIsDinSchedCountdown()`

See also:

`DPxEnableDinSchedCountdown`, `DPxDisableDinSchedCountdown`

`_libdpx.`**`DPxIsDinSchedRunning`**`()`

Verifies if a Din schedule is currently running.

Returns Non-zero if a Din schedule is currently running, zero otherwise.

Return type int

Low-level C definition `int DPxIsDinSchedRunning()`

See also:

`DPxStartDinSched`, `DPxStopDinSched`

`_libdpx.`**`DPxIsDinStabilize`**`()`

Verifies if the hardware calibration mode is enabled.

Returns Non-zero if DIN transitions are being stabilized.

Return type int

Low-level C definition `int DPxIsDinStabilize()`

See also:

`DPxDisableDinStabilize`, `DPxEnableDinStabilize`

`_libdpx.`**`DPxIsDoutBacklightPulse`**`()`

Verifies if the Dout backlight pulse mode is enabled.

Returns Non-zero if LCD backlight LED enables are gated by DOUT15.

Return type int

Low-level C definition `int DPxIsDoutBacklightPulse()`

See also:

`DPxEnableDoutBacklightPulse`, `DPxDisableDoutBacklightPulse`

`_libdpx.`**`DPxIsDoutButtonSchedules`**`()`

Verifies if the DOUT automatic DOUT schedules mode is enabled.

Returns Non-zero if automatic DOUT schedules occur upon DIN button presses.

Return type int

Low-level C definition `int DPxIsDoutButtonSchedules()`

See also:

`DPxEnableDoutButtonSchedules`, `DPxDisableDoutButtonSchedules`

`_libdpx.`**`DPxIsDoutDinLoopback`**`()`

Verifies if the DIN DOUT loopback is enabled.

Returns Non-zero if DIN are driven by digital output ports.

Return type int

Low-level C definition `int DPxIsDoutDinLoopback()`

See also:

DPxEnableDoutDinLoopback, DPxDisableDoutDinLoopback

_libdpx.**DPxIsDoutPixelMode**()

Verifies if the pixel mode is enabled on digital outputs.

Returns Non-zero if pixel mode is enabled, 0 if disabled.

Return type int

Low-level C definition `int DPxIsDoutPixelMode()`

See also:

DPxEnableDoutPixelMode, DPxDisableDoutPixelMode

_libdpx.**DPxIsDoutSchedCountdown**()

Verifies if RAM buffering is enabled for a Dout channel.

Returns Non-zero if SchedCount decrements to 0 and automatically stops schedule.

Return type int

Low-level C definition `int DPxIsDoutSchedCountdown()`

See also:

DPxEnableDoutSchedCountdown, DPxDisableDoutSchedCountdown

_libdpx.**DPxIsDoutSchedRunning**()

Verifies if a Dout schedule is currently running.

Returns Non-zero if a Dout schedule is currently running, zero otherwise.

Return type int

Low-level C definition `int DPxIsDoutSchedRunning()`

See also:

DPxStartDoutSched, DPxStopDoutSched

_libdpx.**DPxIsMicSchedCountdown**()

Returns non-zero if SchedCount decrements to 0 and automatically stops schedule

Low-level C definition `int DPxIsMicSchedCountdown()`

_libdpx.**DPxIsMicSchedRunning**()

Returns non-zero if MIC schedule is currently running

Low-level C definition `int DPxIsMicSchedRunning()`

_libdpx.**DPxIsPPxAwake**()

Check to see if the PROPixx is awake.

Returns 0 (False) if the PROPixx is in sleep mode, 1 (True) otherwise.

Low-level C definition `int DPxIsPPxAwake(void)`

_libdpx.**DPxIsPPxCeilingMount**()

Check to see if the PROPixx is in Ceiling Mount mode.

Returns 0 (False) if the PROPixx is not in Ceiling Mount mode, 1 (True) otherwise.

Low-level C definition `int DPxIsPPxCeilingMount(void)`

_libdpx.**DPxIsPPxHotSpotCorrection**()

_libdpx.**DPxIsPPxLampLedEnabled**()
Check to see if the PROPixx lamp LED is enabled.

Only available for PROPixx Revision 12 and higher.

Returns 0 (False) if the PROPixx lamp LED is disabled, 1 (True) otherwise.

Low-level C definition `int DPxIsPPxLampLedEnabled(void)`

_libdpx.**DPxIsPPxQuietFanMode**()
Check to see if the PROPixx quiet mode is enabled.

Only available for PROPixx Revision 19 and higher.

Returns 0 (False) if the PROPixx quiet mode is disabled, 1 (True) otherwise.

Low-level C definition `int DPxIsPPxQuietFanMode(void)`

_libdpx.**DPxIsPPxRearProjection**()
Check to see if the PROPixx is in Rear projection

Returns 0 (False) if the PROPixx is not in rear projection, 1 (True) otherwise.

Low-level C definition `int DPxIsPPxRearProjection(void)`

_libdpx.**DPxIsPPxVidSeqEnabled**()
Checks to see if the PROPixx Video Sequencer is enabled.

Returns 0 (False) if the PROPixx Video Sequencer is Disabled, 1 (True) if Enabled.

Low-level C definition `int DPxIsPPxVidSeqEnabled(void)`

_libdpx.**DPxIsPropixx**()
Verifies if the device is a PROPixx.

This function will not detect the PROPixx Controller.

Returns Non-zero if the device exists, 0 if the device does not exist.

Return type int

Low-level C definition `int DPxIsPropixx()`

_libdpx.**DPxIsPropixxCtrl**()
Verifies if the device is a PROPixx Controller.

Returns Non-zero if a PROPixx Controller was found, zero otherwise.

Return type int

Low-level C definition `int DPxIsPropixxCtrl()`

_libdpx.**DPxIsPsyncTimeout**()
Verifies if a timeout occurred on the pixel synchronization.

This function allows the user to know if the VESA and Analog +5V input/output pins are trying to draw more than 500 mA.

Returns Non-zero if a timeout occurred, zero otherwise.

Return type int

Low-level C definition `int DPxIsPsyncTimeout()`

_libdpx.**DPxIsRamOffline**()
Verifies if the RAM controller is offline.

Returns Zero if the RAM is online, non-zero otherwise.

Return type int

Low-level C definition `int DPxIsRamOffline()`

`_libdpx.`**`DPxIsReady`**`()`

Verifies if the current device is ready to use.

Returns Non-zero if the device is ready to use, zero otherwise.

Return type int

Low-level C definition `int DPxIsReady()`

`_libdpx.`**`DPxIsTouchpixx`**`()`

Verifies if a Dout schedule is currently running.

Returns Non-zero if TOUCHPixx touch panel hardware is present and enabled.

Return type int

Low-level C definition `int DPxIsTouchpixx()`

See also:

`DPxEnableTouchpixx`, `DPxDisableTouchpixx`

`_libdpx.`**`DPxIsTouchpixxLogContinuousMode`**`()`

Verifies if the TOUCHPixx continuous logging mode is enabled.

Returns Non-zero if the TOUCHPixx is in continuous logging mode, zero otherwise.

Return type int

Low-level C definition `void DPxIsTouchpixxLogContinuousMode()`

See also:

`DPxDisableTouchpixxLogContinuousMode`, `DPxEnableTouchpixxLogContinuousMode`

`_libdpx.`**`DPxIsTouchpixxLogEvents`**`()`

Verifies if the Touchpixx timetag mode is enabled.

Returns Non-zero if Touchpixx transitions are being logged to RAM buffer.

Return type int

Low-level C definition `int DPxIsTouchpixxLogEvents()`

See also:

`DPxDisableTouchpixxLogEvents`, `DPxEnableTouchpixxLogEvents`

`_libdpx.`**`DPxIsTouchpixxLogTimetags`**`()`

Verifies if the Touchpixx timetag mode is enabled.

Returns Non-zero if buffered data is preceeded with nanosecond timetag.

Return type int

Low-level C definition `int DPxIsTouchpixxLogTimetags()`

See also:

`DPxDisableTouchpixxLogTimetags`, `DPxIsTouchpixxLogTimetags`

`_libdpx.`**`DPxIsTouchpixxPressed`**`()`

Returns Non-zero if touch panel is currently pressed

Low-level C definition `int DPxIsTouchpixxPressed()`

`_libdpx.`**`DPxIsTrackpixx`**`()`
Verifies if the device is a TRACKPixx.

Returns Non-zero if a TRACKPixx was found. Else if not.

Return type int

Low-level C definition `int DPxIsTrackpixx()`

`_libdpx.`**`DPxIsVidClutTransparencyColorMode`**`()`
Verifies is the video CLUT transparency color mode is enabled.

This function allows the user to know if the video CLUT transparency color mode is enabled or not.

Returns When the mode is disabled, 0 is returned. When the mode is enabled, it returns Non-zero.

Return type int

Low-level C definition `int DPxIsVidClutTransparencyColorMode()`

See also:

`DPxEnableVidClutTransparencyColorMode`, `DPxDisableVidClutTransparencyColorMode`, `DPxGetVidClutTransparencyColor`, `DPxSetVidClutTransparencyColor`

`_libdpx.`**`DPxIsVidDviActive`**`()`
Returns non-zero if DATAPixx is currently receiving video data over DVI link

Low-level C definition `int DPxIsVidDviActive()`

`_libdpx.`**`DPxIsVidDviActiveDual`**`()`
Returns non-zero if DATAPixx is currently receiving video data over dual-link DVI

Low-level C definition `int DPxIsVidDviActiveDual()`

`_libdpx.`**`DPxIsVidDviLockable`**`()`
Returns non-zero if VIEWPixx is currently receiving video whose timing can directly drive display.

Low-level C definition `int DPxIsVidDviLockable()`

`_libdpx.`**`DPxIsVidHorizOverlay`**`()`
Returns non-zero if the left/right halves of the video image are being overlayed

Low-level C definition `int DPxIsVidHorizOverlay()`

`_libdpx.`**`DPxIsVidHorizSplit`**`()`
Returns non-zero if video is being split across the two VGA outputs.

Low-level C definition `int DPxIsVidHorizSplit()`

`_libdpx.`**`DPxIsVidLcd3D60Hz`**`()`
Returns non-zero if 3D pixel polarity inversion is enabled

Low-level C definition `int DPxIsVidLcd3D60Hz(void)`

`_libdpx.`**`DPxIsVidOverClocked`**`()`
Returns non-zero if DATAPixx is receiving video at clock frequency that is higher than normal.

Low-level C definition `int DPxIsVidOverClocked()`

`_libdpx.`**`DPxIsVidPsyncBlankLine`**`()`
Returns non-zero if pixel sync raster line is always displayed black

Low-level C definition `int DPxIsVidPsyncBlankLine()`

`_libdpx.`**`DPxIsVidPsyncSingleLine`**`()`
Returns non-zero if pixel sync is only recognized on a single raster line.

Low-level C definition `int DPxIsVidPsyncSingleLine()`

`_libdpx.`**`DPxIsVidScanningBacklight`**`()`

Returns non-zero if VIEWPixx scanning backlight is enabled

Low-level C definition `int DPxIsVidScanningBacklight()`

`_libdpx.`**`DPxIsVidVertStereo`**`()`

Returns non-zero if DATAPixx is separating input into sequencial left/right stereo images.

Low-level C definition `int DPxIsVidVertStereo()`

`_libdpx.`**`DPxIsVidVesaBlueline`**`()`

Returns non-zero if VESA 3D output is dependent on video blueline codes

Low-level C definition `int DPxIsVidVesaBlueline()`

`_libdpx.`**`DPxIsVidVesaFreeRun`**`()`

Returns non-zero if PROPixx VESA 3D output is enabled.

Note: PROPixx Rev >= 7 only.

Low-level C definition `int DPxIsVidVesaFreeRun(void)`

`_libdpx.`**`DPxIsVidVesaLeft`**`()`

Returns non-zero if VESA connector has left-eye signal

Low-level C definition `int DPxIsVidVesaLeft()`

`_libdpx.`**`DPxIsViewpixx`**`()`

Verifies if the device is a VIEWPixx.

Returns Non-zero if a VIEWPixx (VIEWPixx, VIEWPixx /EEG or VIEWPixx /3D) is detected, zero otherwise.

Return type int

Low-level C definition `int DPxIsViewpixx()`

`_libdpx.`**`DPxIsViewpixx3D`**`()`

Verifies if the device is a VIEWPixx3D.

Returns Non-zero if a VIEWPixx3D was found, zero otherwise.

Return type int

Low-level C definition `int DPxIsViewpixx3D()`

`_libdpx.`**`DPxIsViewpixxEeg`**`()`

Verifies if the device is a VIEWPixxEEG.

Returns Non-zero if a VIEWPixxEEG was found, zero otherwise.

Return type int

Low-level C definition `int DPxIsViewpixxEeg()`

`_libdpx.`**`DPxOpen`**`()`

Opens VPixx devices.

Must be called before any other DPx functions.

Low-level C definition `void DPxOpen()`

`_libdpx.`**`DPxReadRam`**`(`*address*, *length*`)`
Reads a block of VPixx RAM into a local buffer.

Parameters

- **address** (*int*) – Any even value equal to or greater than zero.
- **length** (*int*) – Any value from zero to the RAM size.

Returns Values taken from RAM.

Return type list

Low-level C definition `void DPxReadRam(unsigned address, unsigned length, void* buffer)`

Example

```
>>> from _libdpx import *
>>>
>>> DPxOpen()
>>>
>>> # Here is a list with the data to write in RAM.
>>> data = [1,2,3,4,5,6]
>>> DPxWriteRam(address= 42, int_list= data)
>>>
>>> DPxUpdateRegCache()
>>>
>>> print'DPxReadRam() = ', DPxReadRam(address= 42, length= 6)
>>>
>>> DPxClose()
```

`_libdpx.`**`DPxRestoreRegs`**`()`
Write the local copy back to the DATAPixx

Low-level C definition `void DPxRestoreRegs()`

`_libdpx.`**`DPxSaveRegs`**`()`
Get all DATAPixx registers, and save them in a local copy

Low-level C definition `void DPxSaveRegs()`

`_libdpx.`**`DPxSelectDevice`**`(`*devsel*`)`
Selects which of the device found in the system should be used.

Parameters **devsel** (*string*) – Any of the predefined constants.

- **Auto**: Lets the low-level choose the device.
- **DATAPixx**: DATAPixx.
- **DATAPixx2**: DATAPixx2.
- **VIEWPixx**: VIEWPixx.
- **PROPixx Ctrl**: PROPixx Controller.
- **PROPixx**: PROPixx.

Returns Non-zero if the device exists, 0 if the device does not exist.

Return type int

Low-level C definition `int DPxSelectDevice(int)`

`_libdpx.`**`DPxSelectDeviceSubName`**`(`*devsel*, *name*`)`
Select which VPixx device to access.

Parameters

- **devsel** (*string*) – Any of the predefined constants.
  - **Auto**: Lets the low-level choose the device.
  - **DATAPixx**: DATAPixx.
  - **DATAPixx2**: DATAPixx2.
  - **VIEWPixx**: VIEWPixx.
  - **PROPixx Ctrl**: PROPixx Controller.
  - **PROPixx**: PROPixx.
- **name** (*string*) – Must me a valid custom device name.

Returns Non-zero if the device exists, 0 if the device does not exist.

Return type int

Low-level C definition `int DPxSelectDeviceSubName(int, char*)`

`_libdpx.`**`DPxSetAdcBuff`**(*buffAddr*, *buffSize*)

Sets base address, write address and buffer size for ADC schedules.

This function is a shortcut which assigns Size/BaseAddr/WriteAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAdcBuff(unsigned buffAddr, unsigned buffSize)`

See also:

`DPxGetAdcSchedOnset`, `DPxSetAdcBuffSize`, `DPxSetAdcBuffWriteAddr`

`_libdpx.`**`DPxSetAdcBuffBaseAddr`**(*buffBaseAddr*)

Sets the ADC RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetAdcBuffBaseAddr(unsigned buffBaseAddr)`

See also:

`DPxGetAdcBuffBaseAddr`

`_libdpx.`**`DPxSetAdcBuffChanRef`**(*channel*, *reference*)

Sets a reference to a channel.

This method allows the user to enable or disable ram buffering on a given channel. When enabled, a given channel is buffered in ram. Enabling RAM buffering can only be done for channels 0 to 15.

Parameters

- **channel** (*int*) – Channel number to associate with a reference.
- **reference** (*str*) – Valid argument is one of the following predefined constants:
  - **gnd**: Referenced to ground.
  - **diff**: Referenced to adjacent analog input.

- **ref0**: Referenced to REF0 analog input.
- **ref1**: Referenced to REF1 analog input.

Low-level C definition `void DPxSetAdcBuffChanRef(int channel, int chanRef)`

`_libdpx.`**`DPxSetAdcBuffSize`**(*buffSize*)

Sets ADC RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAdcBuffSize(unsigned buffSize)`

See also:

`DPxGetAdcBuffSize`

`_libdpx.`**`DPxSetAdcBuffWriteAddr`**(*buffWriteAddr*)

Sets RAM address from which next ADC datum will be written.

Must be an even value.

Parameters **buffWriteAddr** (*int*) – Write address.

Low-level C definition `void DPxSetAdcBuffWriteAddr(unsigned buffWriteAddr)`

See also:

`DPxGetAdcBuffWriteAddr`

`_libdpx.`**`DPxSetAdcSched`**(*onset*, *rateValue*, *rateUnits*, *count*)

Sets ADC schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAdcSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetAdcSchedCount`**(*count*)

Sets ADC schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAdcSchedCount(unsigned count)`

See also:

`DPxGetAdcSchedCount`

`_libdpx.`**`DPxSetAdcSchedOnset`**(*onset*)

Sets nanosecond delay between schedule start and first ADC sample.

Parameters **onset** (*int*) – delay in nanoseconds.

Low-level C definition `void DPxSetAdcSchedOnset(unsigned onset)`

See also:

`DPxGetAdcSchedOnset`

`_libdpx.`**`DPxSetAdcSchedRate`**`(`*rate*, *unit*`)`

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the predefined constants.
  - **hz**: rate updates per second, maximum 200 kHz.
  - **video**: rate updates per video frame, maximum 200 kHz.
  - **nano**: rate updates period in nanoseconds, minimum 5000 ns.

Low-level C definition `void DPxSetAdcSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetAudBuff`**`(`*buffAddr*, *buffSize*`)`

Sets base address, reads address and buffer size for AUD schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAudBuff(unsigned buffAddr, unsigned buffSize)`

`_libdpx.`**`DPxSetAudBuffBaseAddr`**`(`*buffBaseAddr*`)`

Sets the AUD RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetAudBuffBaseAddr(unsigned buffBaseAddr)`

`_libdpx.`**`DPxSetAudBuffReadAddr`**`(`*buffReadAddr*`)`

Sets RAM address from which next AUD datum will be read.

Must be an even value.

Parameters **buffReadAddr** (*int*) – Read address.

Low-level C definition `void DPxSetAudBuffReadAddr(unsigned buffReadAddr)`

`_libdpx.`**`DPxSetAudBuffSize`**`(`*buffSize*`)`

Sets AUD RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAudBuffSize(unsigned buffSize)`

`_libdpx.`**`DPxSetAudCodecOutLeftVolume`**(*volume*, *dBUnits=0*)
Set volume for the DATAPixx Audio OUT Right channel

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `void DPxSetAudCodecOutLeftVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudCodecOutRightVolume`**(*volume*, *dBUnits=0*)
Sets the volume for the DATAPixx Audio OUT Right channel

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `void DPxSetAudCodecOutRightVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudCodecOutVolume`**(*volume*, *dBUnits=0*)
Sets the volume for the DATAPixx Audio OUT Left and Right channels

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `void DPxSetAudCodecOutVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudCodecSpeakerLeftVolume`**(*volume*, *dBUnits=0*)
Sets volume for the DATAPixx Speaker LEft channels

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Low-level C definition `void DPxSetAudCodecSpeakerLeftVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudCodecSpeakerRightVolume`**(*volume*, *dBUnits=0*)
Sets volume for the DATAPixx Speaker Right channels

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0

Low-level C definition `void DPxSetAudCodecSpeakerRightVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudCodecSpeakerVolume`**(*volume*, *dBUnits=0*)
Sets volume for the DATAPixx Speaker Left and Right channels

Parameters

- **volume** (*float*) – The value for the desired volume, between 0 and 1 or in dB.

- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Low-level C definition `void DPxSetAudCodecSpeakerVolume(double volume, int DBUnits)`

`_libdpx.`**`DPxSetAudLRMode`**`(`*mode*`)`

Sets how audio data are updated by schedules.

Parameters **mode** (*str*) – Any of the following predefined constants.

- **mono**: Left schedule data goes to left and right channels.
- **left**: Each schedule data goes to left channel only.
- **right**: Each schedule data goes to right channel only.
- **stereo1**: Pairs of Left data are copied to left/right channels.
- **stereo2**: Left data goes to left channel, Right data goes to right.

Low-level C definition `void DPxSetAudLRMode(int lrMode)`

`_libdpx.`**`DPxSetAudLeftValue`**`(`*volume*`)`

Set the 16-bit 2's complement signed value for the Left audio output channel

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

Low-level C definition `void DPxSetAudLeftValue(int value)`

`_libdpx.`**`DPxSetAudLeftVolume`**`(`*volume*`)`

Sets volume for the Right audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

Low-level C definition `void DPxSetAudRightVolume(double volume)`

`_libdpx.`**`DPxSetAudRightValue`**`(`*volume*`)`

Set the 16-bit 2's complement signed value for the Right audio output channel

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

Low-level C definition `void DPxSetAudRightValue(int value)`

`_libdpx.`**`DPxSetAudRightVolume`**`(`*volume*`)`

Sets volume for the Right audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

Low-level C definition `void DPxSetAudRightVolume(double volume)`

`_libdpx.`**`DPxSetAudSched`**`(`*onset*, *rateValue*, *rateUnits*, *count*`)`

Sets AUD schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAudSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetAudSchedCount`**(*count*)

Sets MIC schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAudSchedCount(unsigned count)`

`_libdpx.`**`DPxSetAudSchedOnset`**(*onset*)

Sets nanosecond delay between schedule start and first MIC update.

Parameters **onset** (*int*) – The onset value.

Low-level C definition `void DPxSetAudSchedOnset(unsigned onset)`

`_libdpx.`**`DPxSetAudSchedRate`**(*rate*, *unit*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the following predefined constants.

Low-level C definition `void DPxSetAudSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetAudVolume`**(*volume*)

Sets the volume for both Left/Right audio channels, range 0-1

Parameters **volume** (*float*) – Value for the desired volume, between 0 and 1.

Low-level C definition `void DPxSetAudVolume(double volume)`

`_libdpx.`**`DPxSetAuxBuff`**(*buffAddr*, *buffSize*)

Sets base address, reads address and buffer size for AUX schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAuxBuff(unsigned buffAddr, unsigned buffSize)`

`_libdpx.`**`DPxSetAuxBuffBaseAddr`**(*buffBaseAddr*)

Sets the AUX RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetAuxBuffBaseAddr(unsigned buffBaseAddr)`

`_libdpx.`**`DPxSetAuxBuffReadAddr`**(*buffReadAddr*)

Sets RAM address from which next AUX datum will be read.

Must be an even value.

Parameters **buffReadAddr** (*int*) – Read address.

Low-level C definition `void DPxSetAuxBuffReadAddr(unsigned buffReadAddr)`

`_libdpx.`**`DPxSetAuxBuffSize`**(*buffSize*)

Sets AUX RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetAuxBuffSize(unsigned buffSize)`

`_libdpx.`**`DPxSetAuxSched`**(*onset*, *rateValue*, *rateUnits*, *count*)

Sets AUX schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAuxSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetAuxSchedCount`**(*count*)

Sets the AUX schedule update count

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetAuxSchedCount(unsigned count)`

`_libdpx.`**`DPxSetAuxSchedOnset`**(*onset*)

Sets nanosecond delay between schedule start and first AUX update.

Parameters **onset** (*int*) – The nanosecond onset between the first update and the start of schedule.

Low-level C definition `void DPxSetAuxSchedOnset(unsigned onset)`

`_libdpx.`**`DPxSetAuxSchedRate`**(*rate*, *unit*)

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the following predefined constants.
  - **hz**: rate updates per second, maximum 96 kHz.
  - **video**: rate updates per video frame, maximum 96 kHz.
  - **nano**: rate updates period in nanoseconds, minimum 10417 ns.

Low-level C definition `void DPxSetAuxSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetCustomStartupConfig`**`()`

Saves the current registers to be used on start up.

This can be useful if you set your projector to ceiling mode or rear projection and you want those settings to persist.

Note: PROPixx Rev >= 6 only and VIEWPixx/PROPixxCTRL Rev >= 25 only

Low-level C definition `void DPxSetCustomStartupConfig(void)`

`_libdpx.`**`DPxSetDacBuff`**`(`*buffAddr*, *buffSize*`)`

Sets base address, read address and buffer size for DAC schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDacBuff(unsigned buffAddr, unsigned buffSize)`

See also:

`DPxGetDacSchedOnset`, `DPxSetDacBuffSize`, `DPxSetDacBuffReadAddr`

`_libdpx.`**`DPxSetDacBuffBaseAddr`**`(`*buffBaseAddr*`)`

Sets the DAC RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetDacBuffBaseAddr(unsigned buffBaseAddr)`

See also:

`DPxGetDacBuffBaseAddr`

`_libdpx.`**`DPxSetDacBuffReadAddr`**`(`*buffReadAddr*`)`

Sets RAM address from which next DAC datum will be read.

Must be an even value.

Parameters **buffReadAddr** (*int*) – Read address.

Low-level C definition `void DPxSetDacBuffReadAddr(unsigned buffReadAddr)`

See also:

`DPxGetDacBuffReadAddr`

`_libdpx.`**`DPxSetDacBuffSize`**`(`*buffSize*`)`

Sets DAC RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDacBuffSize(unsigned buffSize)`

See also:

`DPxGetDacBuffReadAddr`

`_libdpx.`**`DPxSetDacSched`**`(`*onset*, *rateValue*, *rateUnits*, *count*`)`

Sets DAC schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDacSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetDacSchedCount`**`(`*count*`)`

Sets DAC schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDacSchedCount(unsigned count)`

See also:

`DPxGetDacSchedCount`

`_libdpx.`**`DPxSetDacSchedOnset`**`(`*onset*`)`

Sets the nanosecond delay between schedule start and first DAC update.

Parameters **onset** (*int*) – delay in nanoseconds.

Low-level C definition `void DPxSetDacSchedOnset(unsigned onset)`

See also:

`DPxGetDacSchedOnset`

`_libdpx.`**`DPxSetDacSchedRate`**`(`*rate*, *unit*`)`

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the predefined constants.
  - **hz**: rate updates per second, maximum 1 MHz.
  - **video**: rate updates per video frame, maximum 1 MHz.
  - **nano**: rate updates period in nanoseconds, minimum 1000 ns.

Low-level C definition `void DPxSetDacSchedRate(unsigned rateValue, int rateUnits)` `_libdpx.`**`DPxSetDacValue`**(*value*, *channel*)

Sets the current value of a channel.

This method allows the user to set the 16-bit 2's complement signed value for one DAC channel. In other words, this means that you have access to values between -32768 and +32768. For a positive number, you will simply need to leave it as is. For a negative number, since we are using signed, you will need to convert it to the 2's complement representation.

Parameters

- **value** (*int*) – Value of the channel. It is a 16-bit 2's complement signed number.
- **channel** (*int*) – Channel number.

Low-level C definition `void DPxSetDacValue(int value, int channel)`

See also:

`DPxGetDacValue`

`_libdpx.`**`DPxSetDacVoltage`**(*voltage*, *channel*)

Sets the voltage for one DAC channel.

For channels 0 and 1: ± 10V for ch0/1. For channels 2 and 3: ± 5V.

Args:4 voltage (float): Voltage of the channel. channel (int): Channel number.

Low-level C definition `void DPxSetDacVoltage(double voltage, int channel)`

See also:

`DPxGetDacVoltage`

`_libdpx.`**`DPxSetDebug`**(*level*)

Debugging level controls verbosity of debug and trace messages.

Parameters **level** (*int*) – Set to `0` to disable messages, `1` to print libdpx, `2`, to print libusb debug messages.

Low-level C definition `void DPxSetDebug(int level)`

`_libdpx.`**`DPxSetDinBuff`**(*buffAddr*, *buffSize*)

Sets base address, write address and buffer size for Din schedules.

This function is a shortcut which assigns Size/BaseAddr/WriteAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDinBuff(unsigned buffAddr, unsigned buffSize)`

See also:

`DPxGetDinSchedOnset`, `DPxSetDinBuffSize`, `DPxSetDinBuffWriteAddr`

`_libdpx.`**`DPxSetDinBuffBaseAddr`**(*buffBaseAddr*)

Sets the DIN RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetDinBuffBaseAddr(unsigned buffBaseAddr)`

See also:

`DPxGetDinBuffBaseAddr`

`_libdpx.`**`DPxSetDinBuffSize`**`(`*buffSize*`)`

Sets Din RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDinBuffSize(unsigned buffSize)`

See also:

`DPxGetDinBuffSize`

`_libdpx.`**`DPxSetDinBuffWriteAddr`**`(`*buffWriteAddr*`)`

Sets RAM address from which next DIN datum will be written.

Must be an even value.

Parameters **buffWriteAddr** (*int*) – Write address.

Low-level C definition `void DPxSetDinBuffWriteAddr(unsigned buffWriteAddr)`

See also:

`DPxGetDinBuffWriteAddr`

`_libdpx.`**`DPxSetDinDataDir`**`(`*direction_mask*`)`

Sets the port direction mask.

Sets the port direction for each bit in `direction_mask`. The mask is one value representing all bits from the port. The given `direction_mask` will set the direction of all digital input channels. For each bit which should drive its port, the corresponding `direction_mask` value should be set to 1. An hexadecimal `direction_mask` can be given.

For example, `direction_mask = 0x0000F` will enable the port for the first 4 bits on the right. All other ports will be disabled.

Parameters **int** – Value which corresponds in binary to the desired open ports.

Low-level C definition `void DPxSetDinDataDir(int DirectionMask)`

See also:

`DPxGetDinDataDir`, `DPxSetDinDataDir`

`_libdpx.`**`DPxSetDinDataOut`**`(`*dataOut*`)`

Sets the data which should be driven on each port.

In order to be able to drive the ports with the given value, the port direction has to be properly enabled. This can be done using the `DPxSetDinDataDir` with the appropriate bit mask.

Parameters **dataOut** (*int*) – Set bit to 1 will enable the port for that bit. Set bit to 0 will disable it.

Low-level C definition `void DPxSetDinDataOut(int dataOut)`

See also:

`DPxGetDinDataDir`, `DPxSetDinDataDir`

`_libdpx.`**`DPxSetDinDataOutStrength`**`(`*strength*`)`

Sets the strength of the driven outputs.

This function allows the user to set the current (Ampere) strength of the driven outputs. The implementation actual value uses `1/16` up to `16/16`. So minimum strength will be `0.0625` and maximum will be `1`. The strength can be increased by `0.0625` up to `1`. Giving a strength of `0` will thus set it to `0.0625`. Giving a strength between `0.0625` and `0.125` will round the given strength to one of the two increments.

The strength is the same for all channels.

Parameters **strength** (*float*) – Any value in a range of 0 to 1.

Low-level C definition `void DPxSetDinDataOutStrength(double strength)`

See also:

`DPxGetDinDataOutStrength`

`_libdpx.`**`DPxSetDinSched`**`(`*onset*, *rateValue*, *rateUnits*, *count*`)`

Sets Din schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDinSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetDinSchedCount`**`(`*count*`)`

Sets Din schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDinSchedCount(unsigned count)`

See also:

`DPxGetDinSchedCount`

`_libdpx.`**`DPxSetDinSchedOnset`**`(`*onset*`)`

Sets nanosecond delay between schedule start and first Din sample.

Parameters **onset** (*int*) – delay in nanoseconds.

Low-level C definition `void DPxSetDinSchedOnset(unsigned onset)`

See also:

`DPxGetDinSchedOnset`

`_libdpx.`**`DPxSetDinSchedRate`**`(`*rate*, *unit*`)`

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the predefined constants.
  - **hz**: samples per second, maximum 1 MHz.
  - **video**: samples per video frame, maximum 1 MHz.
  - **nano**: sample period in nanoseconds, minimum 1000 ns.

Low-level C definition `void DPxSetDinSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetDoutBuff`**(*buffAddr*, *buffSize*)

Sets base address, read address and buffer size for Dout schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDoutBuff(unsigned buffAddr, unsigned buffSize)`

See also:

`DPxGetDoutSchedOnset`, `DPxSetDoutBuffSize`, `DPxSetDoutBuffReadAddr`

`_libdpx.`**`DPxSetDoutBuffBaseAddr`**(*buffBaseAddr*)

Sets the Dout RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetDoutBuffBaseAddr(unsigned buffBaseAddr)`

See also:

`DPxGetDoutBuffBaseAddr`

`_libdpx.`**`DPxSetDoutBuffReadAddr`**(*buffReadAddr*)

Sets RAM address from which next Dout datum will be read.

Must be an even value.

Parameters **buffReadAddr** (*int*) – Read address.

Low-level C definition `void DPxSetDoutBuffReadAddr(unsigned buffReadAddr)`

See also:

`DPxGetDoutBuffReadAddr`

`_libdpx.`**`DPxSetDoutBuffSize`**(*buffSize*)

Sets Dout RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetDoutBuffSize(unsigned buffSize)`

See also:

`DPxGetDoutBuffReadAddr`

`_libdpx.`**`DPxSetDoutSched`**`(`*onset*, *rateValue*, *rateUnits*, *count*`)`

Sets Dout schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDoutSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetDoutSchedCount`**`(`*count*`)`

Sets Dout schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetDoutSchedCount(unsigned count)`

See also:

`DPxGetDoutSchedCount`

`_libdpx.`**`DPxSetDoutSchedOnset`**`(`*onset*`)`

Sets nanosecond delay between schedule start and first Dout update.

Parameters **onset** (*int*) – delay in nanoseconds.

Low-level C definition `void DPxSetDoutSchedOnset(unsigned onset)`

See also:

`DPxGetDoutSchedOnset`

`_libdpx.`**`DPxSetDoutSchedRate`**`(`*rate*, *unit*`)`

Sets the schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the predefined constants.

  -**hz**: rate updates per second, maximum 10 MHz. -**video**: rate updates per video frame, maximum 10 MHz. -**nano**: rate updates period in nanoseconds, minimum 100 ns.

Low-level C definition `void DPxSetDoutSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetDoutValue`**`(`*bit_value*, *bit_mask*`)`

Sets the port direction mask.

Sets the port direction for each bit in `bit_mask`. The mask is one value representing all bits from the port. The given `bit_mask` will set the direction of all digital input channels. For each bit which should drive its port, the corresponding `direction_mask` value should be set to 1. An hexadecimal `direction_mask` can be given.

For example, `bit_mask = 0x0000F` will enable the port for the first 4 bits on the right. All other ports will be disabled.

Parameters

- **bit_value** (*int*) – value of bits.
- **bit_mask** (*int*) – Turns on or off the specific Digital Out channel. 1 for on, 0 for off.

Low-level C definition `void DPxSetDoutValue(int bitValue, int bitMask)`

`_libdpx.`**`DPxSetError`**`(`*errCode*`)`

Sets the device to the given error code.

Parameters **errCode** (*int*) – Given error code we wish to set the device to.

Low-level C definition `void DPxSetError(int error)`

`_libdpx.`**`DPxSetFactoryStartupConfig`**`()`

Returns your device's configurable registers to their default state. This will reset all your customs setting such as rear projection.

Low-level C definition `void DPxSetFactoryStartupConfig(void)`

`_libdpx.`**`DPxSetMarker`**`()`

Latches the current time value into the marker register.

Low-level C definition `void DPxGetMarker()`

See also:

`DPxSetMarker`, `DPxGetNanoMarker`

`_libdpx.`**`DPxSetMicBuff`**`(`*buffAddr*, *buffSize*`)`

Sets base address, reads address and buffer size for MIC schedules.

This function is a shortcut which assigns Size/BaseAddr/ReadAddr.

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetMicBuff(unsigned buffAddr, unsigned buffSize)`

`_libdpx.`**`DPxSetMicBuffBaseAddr`**`(`*buffBaseAddr*`)`

Sets the MIC RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetMicBuffBaseAddr(unsigned buffBaseAddr)`

`_libdpx.`**`DPxSetMicBuffSize`**`(`*buffSize*`)`

Sets MIC RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetMicBuffSize(unsigned buffSize)`

`_libdpx.`**`DPxSetMicBuffWriteAddr`**`(`*buffWriteAddr*`)`

Sets RAM address from which next MIC datum will be written.

Must be an even value.

Low-level C definition `void DPxSetMicBuffWriteAddr(unsigned buffWriteAddr)`

`_libdpx.`**`DPxSetMicLRMode`**`(`*mode*`)`

Sets the schedule buffer storing mode.

This method allows the user to configure how the microphone left and right channels are stored to the schedule buffer.

Parameters **mode** (*str*) – Any of the following predefined constants.

- **mono**: Mono data is written to the schedule buffer. The average of Left/Right CODEC data.
- **left**: Left data is written to the schedule buffer.
- **right**: Right data is written to the schedule buffer.
- **stereo**: Left and Right data are both written to the schedule buffer.

Low-level C definition `void DPxSetMicLRMode(int lrMode)`

`_libdpx.`**`DPxSetMicSched`**`(`*onset*, *rateValue*, *rateUnits*, *count*`)`

Sets MIC schedule onset, count and rate.

This function is a shortcut which assigns Onset/Rate/Count. If `count` is greater than zero, the count down mode is enabled.

Parameters

- **onset** (*int*) – Schedule onset.
- **rateValue** (*int*) – Rate value.
- **rateUnits** (*str*) – Usually `hz`. Can also be `video` to update every `rateValue` video frames or `nano` to update every `rateValue` nanoseconds.
- **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetMicSched(unsigned onset, unsigned rateValue, int rateUnits, unsigned count)`

`_libdpx.`**`DPxSetMicSchedCount`**`(`*count*`)`

Sets MIC schedule update count.

Parameters **count** (*int*) – Schedule count.

Low-level C definition `void DPxSetMicSchedCount(unsigned count)`

`_libdpx.`**`DPxSetMicSchedOnset`**`(`*onset*`)`

Sets nanosecond delay between schedule start and first MIC update.

Parameters **onset** (*int*) – The onset value.

Low-level C definition `void DPxSetMicSchedOnset(unsigned onset)`

`_libdpx.`**`DPxSetMicSchedRate`**`(`*rate*, *unit*`)`

Sets the MIC schedule rate.

This method allows the user to set the schedule rate. Since the rate can be given with different units, the method also needs to have a unit associated with the rate.

If no delay is required, this method does not need to be used. Default value is 0.

Parameters

- **rate** (*int*) – Any positive value equal to or greater than zero.
- **unit** (*str*) – Any of the following predefined constants.
  - **hz**: rate updates per second, maximum 102.4 kHz.
  - **video**: rate updates per video frame, maximum 102.4 kHz.
  - **nano**: rate updates period in nanoseconds, minimum 9750 ns.

Low-level C definition `void DPxSetMicSchedRate(unsigned rateValue, int rateUnits)`

`_libdpx.`**`DPxSetMicSource`**`(`*source*, *gain*, *dBUnits=0*`)`

Sets the source for the microphone.

Selects the source of the microphone input. Typical gain values would be around 100 for a microphone input, and around 1 for line-level input.

Parameters

- **source** (*str*) – One of the following:
  - `MIC`: Microphone level input
  - `LINE`: Line level audio input.
- **gain** (*int*) – The gain can take the following values depnding on the scale:
  - linear scale : [1, 1000]
  - dB scale : [0, 60] dB
- **dBUnits** (*int, optional*) – Set non-zero to return the gain in dB. Defaults to 0.

Low-level C definition `void DPxSetMicSource(int source, double gain, int dBUnits)`

`_libdpx.`**`DPxSetPPx3dCrosstalk`**`(`*crosstalk*`)`

Sets the 3D crosstalk (0-1) which should be subtracted from stereoscopic stimuli.

Warning: This only works with RB3D mode and requires revision 6 of the PROPixx.

Parameters **crosstalk** (*double*) – A value between 0 and 1 which represents the 3d crosstalk.

Low-level C definition `void DPxSetPPx3dCrosstalk(double crosstalk)`

`_libdpx.`**`DPxSetPPxAwake`**`()`

Turns on the PROPixx.

Only available for PROPixx Revision 12 and higher.

Low-level C definition `void DPxSetPPxAwake()`

`_libdpx.`**`DPxSetPPxDlpSeqPgrm`**`(`*program*`)`

Sets the PROPixx DLP Sequencer program.

Only available for PROPixx Revision 6 and higher.

Parameters **String** – Any of the following predefined constants.

- **RGB**: Default RGB
- **RB3D**: R/B channels drive grayscale 3D

- **RGB240**: Only show the frame for 1/2 a 120 Hz frame duration.
- **RGB180**: Only show the frame for 2/3 of a 120 Hz frame duration.
- **QUAD4X**: Display quadrants are projected at 4x refresh rate.
- **QUAD12X**: Display quadrants are projected at 12x refresh rate with grayscales.
- **GREY3X**: Converts 640x1080@360Hz RGB to 1920x1080@720Hz Grayscale with blank frames.

Low-level C definition `void DPxSetPPxDlpSeqPgrm(int program)`

`_libdpx.`**`DPxSetPPxHotSpotCenter`**`(`*x*, *y*`)`

Sets the hotspot correction location only.

Parameters

- **x** (*int*) – The position in x from the center, + to the left.
- **y** (*int*) – The position in y from the center, + to the bottom.

`_libdpx.`**`DPxSetPPxHotSpotLut`**`(`*x*, *y*, *psi*`)`

Parameters

- **x** –
- **y** –
- **psi** –

`_libdpx.`**`DPxSetPPxLedMask`**`(`*mask*`)`

Sets the PROPixx mask to turn off specific LEDs.

Only available for PROPixx Revision 33 and higher.

Parameters **Int** – Any of the following predefined constants.

- **0**: All LEDs are on.
- 1: RED is turned off.
- 2: GREEN is turned off.
- 3: RED and GREEN are turned off.
- 4: BLUE is turned off.
- 5: RED and BLUE are turned off.
- 6: BLUE and GREEN are turned off.
- 7: ALL LEDs are off.

Low-level C definition `void DPxSetPPxLedMask(int mask)`

`_libdpx.`**`DPxSetPPxSleep`**`()`

Turns off the PROPixx.

Only available for PROPixx Revision 12 and higher.

Low-level C definition `void DPxSetPPxSleep()`

`_libdpx.`**`DPxSetTouchpixxBuff`**`(`*buffAddr*, *buffSize*`)`

Sets base address, write address and buffer size for Touchpixx schedules.

This function is a shortcut which assigns Size/BaseAddr/WriteAddr

Parameters

- **buffAddr** (*int*) – Base address.
- **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetTouchpixxBuff(unsigned buffAddr, unsigned buffSize)`

See also:

`DPxSetTouchpixxBuffSize`, `DPxSetTouchpixxBuffWriteAddr`

`_libdpx.`**`DPxSetTouchpixxBuffBaseAddr`**`(`*buffBaseAddr*`)`

Sets the Touchpixx RAM buffer start address.

Must be an even value.

Parameters **buffBaseAddr** (*int*) – Base address.

Low-level C definition `void DPxSetTouchpixxBuffBaseAddr(unsigned buffBaseAddr)`

See also:

`DPxGetTouchpixxBuffBaseAddr`

`_libdpx.`**`DPxSetTouchpixxBuffSize`**`(`*buffSize*`)`

Sets Touchpixx RAM buffer size in bytes.

Must be an even value. Buffer wraps to Base after Size.

Parameters **buffSize** (*int*) – Buffer size.

Low-level C definition `void DPxSetTouchpixxBuffSize(unsigned buffSize)`

See also:

`DPxGetTouchpixxBuffSize`

`_libdpx.`**`DPxSetTouchpixxBuffWriteAddr`**`(`*buffWriteAddr*`)`

Sets RAM address from which next Touchpixx datum will be written.

Must be an even value.

Parameters **buffWriteAddr** (*int*) – Write address.

Low-level C definition `void DPxSetTouchpixxBuffWriteAddr(unsigned buffWriteAddr)`

See also:

`DPxGetTouchpixxBuffWriteAddr`

`_libdpx.`**`DPxSetTouchpixxStabilizeDuration`**`(`*duration*`)`

Sets the Touchpixx stabilization duration.

This function sets duration in seconds that TOUCHPixx panel coordinates must be stable before being recognized as a touch in `DPxGetTouchpixxCoords`

Parameters **duration** (*int*) – Duration in seconds.

Low-level C definition `void DPxSetTouchpixxStabilizeDuration(double duration)`

See also:

`DPxGetTouchpixxStabilizeDuration`

_libdpx.**DPxSetVidBacklightIntensity**(*intensity*)
Sets the display current back light intensity.

Parameters **intensity** (*int*) – Set to `0` for the lowest intensity level, `255` for the highest, or any other value in between.

Example

```
>>> my_device.setBacklightIntensity(42)
>>> my_device.updateRegisterCache()
>>> print my_device.getBacklightIntensity()
42
```

See also:

`DPxGetBacklightIntensity`

Low-level C definition `void DPxSetBacklightIntensity(int intensity)`

_libdpx.**DPxSetVidClut**(*CLUT*)
Sets the video color lookup table (CLUT).

This function returns immediately; the CLUT is implemented at next vertical blanking interval.

Parameters **CLUT** (*list*) – A list of 3 lists representing the colors [[RED], [GREEN], [BLUE]]

Low-level C definition `void DPxSetVidClut(UInt16* clutData)`

_libdpx.**DPxSetVidClutTransparencyColor**(*red*, *green*, *blue*)
Set 48-bit RGB video CLUT transparency color.

This function allows the user to set the value for the RGB video CLUT transparency color.

Parameters

- **red** (*int*) – Pixel color value for the red channel.
- **green** (*int*) – Pixel color value for the green channel.
- **blue** (*int*) – Pixel color value for the blue channel.

Low-level C definition `void DPxSetVidClutTransparencyColor(UInt16 red, UInt16 green, UInt16 blue)`

See also:

`DPxGetVidClutTransparencyColor`

_libdpx.**DPxSetVidCluts**(*CLUTs*)
Sets the video color lookup tables.

Parameters **CLUT** (*list*) – A list of 3 lists representing the colors [[RED], [GREEN], [BLUE]]

Low-level C definition `void DPxSetVidCluts(UInt16* clutData)`

_libdpx.**DPxSetVidHorizOverlayAlpha**(*int_list*)
Sets 1024 16-bit video horizontal overlay alpha values, in order X0,X1..X511,Y0,Y1...Y511.

Parameters **int_list** (*list*) – A list of integers to set the overlay alpha values, defaults to all zeros

Low-level C definition `void DPxSetVidHorizOverlayAlpha(UInt16* alphaData)`

_libdpx.**DPxSetVidHorizOverlayBounds**(*x1*, *y1*, *x2*, *y2*)
Sets bounding rectangle within left half image whose contents are composited with right half image.

Parameters

- **x1** (*int*) – Bottom left x position.
- **y1** (*int*) – Bottom left y position.
- **x2** (*int*) – Top right x position.
- **y2** (*int*) – Top right y position.

Low-level C definition `void DPxSetVidHorizOverlayBounds(int X1, int Y1, int X2, int Y2)`

`_libdpx.`**`DPxSetVidMode`**`(`*mode*`)`

Sets the video processing mode.

Only available for PROPixx Revision 6 and higher.

Parameters **mode** (*str*) – Any of the following predefined constants.

- **L48**: DVI RED[7:0] is used as an index into a 256-entry 16-bit RGB colour lookup table.
- **M16**: DVI RED[7:0] & GREEN[7:0] concatenate into a VGA 16-bit value sent to all three RGB components.
- **C48**: Even/Odd pixel RED/GREEN/BLUE[7:0] concatenate to generate 16-bit RGB components at half the horizontal resolution.
- **L48D**: DVI RED[7:4] & GREEN[7:4] concatenate to form an 8-bit index into a 256-entry 16-bit RGB colour lookup table.
- **M16D**: DVI RED[7:3] & GREEN[7:3] & BLUE[7:2] concatenate into a VGA 16-bit value sent to all three RGB components.
- **C36D**: Even/Odd pixel RED/GREEN/BLUE[7:2] concatenate to generate 12-bit RGB components at half the horizontal resolution.
- **C24**: Straight passthrough from DVI 8-bit (or HDMI “deep” 10/12-bit) RGB to VGA 8/10/12-bit RGB.

Low-level C definition `void DPxSetVidMode(int vidMode)`

`_libdpx.`**`DPxSetVidPsyncRasterLine`**`(`*line*`)`

Sets the raster line on which the pixel sync sequence is expected.

Parameters **line** (*int*) – The line which will contain de PSync.

Low-level C definition `void DPxSetVidPsyncRasterLine(int line)`

`_libdpx.`**`DPxSetVidSource`**`(`*vidSource*`)`

Sets the source of video to be displayed.

Parameters **vidSource** (*str*) – The source we want to display.

- **DVI**: Monitor displays DVI signal.
- **SWTP**: Software test pattern showing image from RAM.
- **SWTP 3D**: 3D Software test pattern flipping between left/right eye images from RAM.
- **RGB SQUARES**: RGB ramps.
- **GRAY**: Uniform gray display having 12-bit intensity.
- **BAR**: Drifting bar.
- **BAR2**: Drifting bar.
- **DOTS**: Drifting dots.

- **RAMP**: Drifting ramp, with dots advancing `x*2` pixels per video frame, where `x` is a 4-bit signed.
- **RGB**: Uniform display with 8-bit intensity nn, send to RGBA channels enabled by mask m.
- **PROJ**: Projector Hardware test pattern.

Low-level C definition `void DPxSetVidSource(int vidSource)`

`_libdpx.`**`DPxSetVidVesaLeft`**`()`

VESA connector outputs left-eye signal.

Low-level C definition `void DPxSetVidVesaLeft()`

`_libdpx.`**`DPxSetVidVesaPhase`**`(`*phase*`)`

Sets the 8-bit unsigned phase of the VESA 3D waveform.

Varying this phase from 0-255, allows adjustements to the phase relationship between stereo video and 3D goggles switching.

The following combinations have been found to work well:

If you are using a VIEWPIxx/3D, you should set the `phase` to `0x64`. If you are using a CTR with our DATAPixx, it should be set to `0xF5`.

Parameters **phase** (*int*) – Phase of the VESA 3D waveform

Low-level C definition `void DPxSetVidVesaPhase(int phase)`

`_libdpx.`**`DPxSetVidVesaRight`**`()`

VESA connector outputs right-eye signal.

Low-level C definition `void DPxSetVidVesaRight()`

`_libdpx.`**`DPxSetVidVesaWaveform`**`(`*waveform*`)`

Sets the waveform which will be sent to the DATAPixx VESA 3D connector.

Only available for PROPixx Revision 6 and higher.

Parameters **waveform** (*str*) – Any of the following predefined constants.

- **LR**: VESA port drives straight L/R squarewave for 3rd party emitter.
- **CRYSTALEYES**: VESA port drives 3DPixx IR emitter for CrystalEyes 3D goggles.
- **NVIDIA**: VESA port drives 3DPixx IR emitter for NVIDIA 3D goggles.

Low-level C definition `void DPxSetVidVesaWaveform(int waveform)`

`_libdpx.`**`DPxStartAdcSched`**`()`

Starts running an ADC schedule.

Low-level C definition `void DPxStartAdcSched()`

See also:

`DPxStopAdcSched`, `DPxIsAdcSchedRunning`

`_libdpx.`**`DPxStartAudSched`**`()`

Starts running an AUD schedule

Low-level C definition `void DPxStartAudSched()`

`_libdpx.`**`DPxStartAuxSched`**`()`

Starts running an AUX schedule.

Low-level C definition `void DPxStartAuxSched()`

`_libdpx.`**`DPxStartDacSched`**`()`
Starts running a DAC schedule.

Low-level C definition `void DPxStartDacSched()`

See also:

`DPxStopDacSched`, `DPxIsDacSchedRunning`

`_libdpx.`**`DPxStartDinSched`**`()`
Starts running a Din schedule.

Low-level C definition `void DPxStartDinSched()`

See also:

`DPxStopDinSched`, `DPxIsDinSchedRunning`

`_libdpx.`**`DPxStartDoutSched`**`()`
Starts running a Dout schedule.

Low-level C definition `void DPxStartDoutSched()`

See also:

`DPxStopDoutSched`, `DPxIsDoutSchedRunning`

`_libdpx.`**`DPxStartMicSched`**`()`
Starts running a MIC schedule

Low-level C definition `void DPxStartMicSched()`

`_libdpx.`**`DPxStopAdcSched`**`()`
Stops running an ADC schedule.

Low-level C definition `void DPxStopAdcSched()`

See also:

`DPxStartAdcSched`, `DPxIsAdcSchedRunning`

`_libdpx.`**`DPxStopAudSched`**`()`
Stops running an AUD schedule

Low-level C definition `void DPxStopAudSched()`

`_libdpx.`**`DPxStopAuxSched`**`()`
Stops running an AUX schedule.

Low-level C definition `void DPxStopAuxSched()`

`_libdpx.`**`DPxStopDacSched`**`()`
Stops running a DAC schedule.

Low-level C definition `void DPxStopDacSched()`

See also:

`DPxStartDacSched`, `DPxIsDacSchedRunning`

`_libdpx.`**`DPxStopDinSched`**`()`
Stops running a Din schedule.

Low-level C definition `void DPxStopDinSched()`

See also:

`DPxStartDinSched`, `DPxIsDinSchedRunning` `_libdpx.`**`DPxStopDoutSched`**`()`

Stops running a Dout schedule.

Low-level C definition `void DPxStopDoutSched()`

See also:

`DPxStartDoutSched`, `DPxIsDoutSchedRunning`

`_libdpx.`**`DPxStopMicSched`**`()`

Stops running a MIC schedule.

Low-level C definition `void DPxStopMicSched()`

`_libdpx.`**`DPxUpdateRegCache`**`()`

Updates the local register cache.

DPxWriteRegCache, then readback registers over USB into local cache

Low-level C definition `void DPxUpdateRegCache()`

`_libdpx.`**`DPxUpdateRegCacheAfterPixelSync`**`(`*nPixels*, *pixelData*, *timeout*`)`

Writes local register cache to VPixx device over USB, using Pixel Sync timing.

This function is like DPxUpdateRegCache, but waits for a pixel sync sequence before executing.

Parameters

- **nPixels** (*int*) – The number of pixels
- **pixelData** (*list*) – The requested pattern for PSynx
- **timeout** (*int*) – Maximum time to wait before a PSync.

Low-level C definition `void DPxUpdateRegCacheAfterPixelSync(int nPixels, unsigned char* pixelData, int timeout)`

`_libdpx.`**`DPxUpdateRegCacheAfterVideoSync`**`()`

Updates local register cache with VPixx device.

This function is like DPxUpdateRegCache, but the device only writes the registers on the leading edge of the next vertical sync pulse.

Low-level C definition `void DPxUpdateRegCacheAfterVideoSync()`

`_libdpx.`**`DPxVideoScope`**`(`*file_path*`)`

VIEWPixx video source analysis.

Parameters **file_path** (*str*) – Where the file to analyze is located.

Low-level C definition `void DPxVideoScope(int toFile)`

`_libdpx.`**`DPxWriteRam`**`(`*address*, *int_list*`)`

Writes a local buffer into VPixx RAM.

Parameters

- **address** (*int*) – Any even value equal to or greater than zero.
- **int_list** (*int*) – `int_list` is a list which will fill the RAM data. The length of RAM used is based on the length of `int_list`. It can't be greater than RAM size.

Low-level C definition `void DPxWriteRam(unsigned address, unsigned length, void* buffer)`

See also:

`DPxGetRamSize`

Example

```
>>> from _libdpx import *
>>>
>>> DPxOpen()
>>>
>>> # Here is a list with the data to write in RAM.
>>> data = [1,2,3,4,5,6]
>>> DPxWriteRam(address= 42, int_list= data)
>>>
>>> DPxUpdateRegCache()
>>>
>>> print'DPxReadRam() = ', DPxReadRam(address= 42, length= 6)
>>>
>>> DPxClose()
```

`_libdpx.`**`DPxWriteRegCache`**`()`

Write local register cache to VPixx device over USB.

Low-level C definition `void DPxWriteRegCache()`

`_libdpx.`**`DPxWriteRegCacheAfterPixelSync`**`(`*nPixels*, *pixelData*, *timeout*`)`

Write local register cache to VPixx device over USB.

This function is like DPxWriteRegCache, but it waits for a pixel sync sequence.

Parameters

- **nPixels** (*int*) – The number of pixels.
- **pixelData** (*list*) – The requested pattern for PSync.
- **timeout** (*int*) – Maximum time to wait before a PSync.

Low-level C definition `void DPxWriteRegCacheAfterPixelSync(int nPixels, unsigned char* pixelData, int timeout)`

`_libdpx.`**`DPxWriteRegCacheAfterVideoSync`**`()`

Write local register cache to VPixx device over USB on the next video frame.

This function is like DPxWriteRegCache, but the device only writes the registers on leading edge of next vertical sync pulse.

Low-level C definition `void DPxWriteRegCacheAfterVideoSync()`

`_libdpx.`**`DpxStopAllScheds`**`()`

Shortcut to stop running all DAC/ADC/DOUT/DIN/AUD/AUX/MIC schedules.

Low-level C definition `void DPxStopAllScheds()`

`_libdpx.`**`PPxLoadTestPattern`**`(`*image*, *add*, *page*`)`

Loads an image in the PROPixx since the PROPixx is different for software test patterns. :param image: :param add: :param page:

`_libdpx.`**`TPxDisableFreeRun`**`()`

Disable the schedule to automatically record all data

`_libdpx.`**`TPxEnableFreeRun`**`()`

Enable the schedule to automatically record all data

`_libdpx.`**`TPxSaveToCSV`**`(`*last_read_address*, *fileName*`)`

Save from last_read_address up to TPxGetReadAddr(), into the default file for today `_libdpx.`**`TPxSetBuff`**(*buffer_addres*, *buffer_size*)

Shortcut to set up the the base address and buffer size for schedule recording of TRACKPixx data.

CHAPTER

TWENTY

SCANNING BACKLIGHT FEATURES

class `scanningBackLight.`**`ScanningBackLight`**

Bases: `object`

Implements the Scanning Back Light methods for a VIEWPixx.

`getBacklightIntensity``()`

Returns the level of the VIEWPixx back light intensity.

Returns An integer between 0 and 255.

Example

```
>>> print my_device.getBacklightIntensity()
255
```

See also:

`setBacklightIntensity`

`isScanningBackLightEnabled``()`

Gets the scanning back light state.

Returns `True` if scanning back light is enabled, `False` otherwise.

Return type bool

`setBacklightIntensity``(`*intensity*`)`

Sets the display current back light intensity.

Parameters **intensity** (*int*) – Set to `0` for the lowest intensity level, `255` for the highest, or any value in between.

Example

```
>>> my_device.setBacklightIntensity(42)
>>> my_device.updateRegisterCache()
>>> print my_device.getBacklightIntensity()
42
```

See also:

`getBacklightIntensity`

`setScanningBackLight``(`*enable*`)`

Sets the scanning back light mode.

This method allows the user to turn the scanning back light on or off. When the scanning back light is enabled, the screen will look dimmer. When scanning back light is disabled, the screen will look brighter.

Parameters **enable** (*bool*) – Scanning back light mode.

CHAPTER

TWENTYONE

VIDEO FEATURES

class `videoFeatures.`**`VideoFeatures`**

Bases: `object`

Implements the video methods for most Devices.

This Class implements video features used by all devices except the PROPixx. If a video feature can be used in a DATAPixx as well as a VIEWPixx, it is included in this Class.

`getRasterLinePixelSync``()`

Gets the raster line on which the pixel sync sequence is expected.

Returns **line** – line on which pixel sync sequence is expected.

Return type int

`getVideoDotFrequency``()`

Gets the video dot frequency (pixel sync.).

Returns **frequency** – frequency in Hz.

Return type int

`getVideoHorizontalLineFrequency``()`

Gets the video horizontal line frequency.

Returns **frequency** – frequency in Hz.

Return type int

`getVideoHorizontalTotal``()`

Gets the total number of clocks per horizontal line.

This method allows the user to get the total number of clocks in one horizontal scan line. This value includes the horizontal blanking interval.

Returns **number** – number of clocks per horizontal lines.

Return type int

`getVideoMode``()`

Gets the current video processing mode.

This method allows the user to know what is the current display mode of the VPixx device.

Warning: Mode **RB24** is availible on VIEWPixx ONLY, revision 21 or higher.

Returns

mode – video mode is one of the following predefined constants:

- **C24**: Straight pass through from DVI 8-bit (or HDMI "deep" 10/12-bit) RGB to VGA 8/10/12-bit RGB
- **L48**: DVI RED[7:0] is used as an index into a 256-entry 16-bit RGB color lookup table
- **M16**: DVI RED[7:0] & GREEN[7:0] concatenate into a VGA 16-bit value sent to all three RGB components
- **C48**: Even/Odd pixel RED/GREEN/BLUE[7:0] concatenate to generate 16-bit RGB components at half the horizontal resolution
- **L48D**: DVI RED[7:4] & GREEN[7:4] concatenate to form an 8-bit index into a 256-entry 16-bit RGB color lookup table
- **M16D**: DVI RED[7:3] & GREEN[7:3] & BLUE[7:2] concatenate into a VGA 16-bit value sent to all three RGB components
- **C36D**: Even/Odd pixel RED/GREEN/BLUE[7:2] concatenate to generate 12-bit RGB components at half the horizontal resolution
- **RB24**: DVI RED[7:0] & GREEN[7:4] concatenate to form 12-bit RED value, DVI BLUE[7:0] & GREEN[3:0] concatenate to form 12-bit BLUE value, GREEN is forced to 0

Return type string

`getVideoVerticalFrameFrequency``()`

Gets the video vertical frame frequency.

Returns **frequency** – frequency in Hz.

Return type int

`getVideoVerticalFramePeriod``()`

Gets the video vertical frame period.

Returns **period** – Period in nanoseconds.

Return type int

`getVideoVerticalTotal``()`

Gets the total number of clocks per vertical line.

This method allows the user to get the total number of clocks in one vertical frame. This value includes the vertical blanking interval.

Returns **number** – number of clocks per vertical lines.

Return type int

`getVisibleLinePerVerticalFrame``()`

Gets the number of visible lines in one vertical frame.

Returns **number** – Number of visible lines.

Return type int

`getVisiblePixelsPerHorizontalLine``()`

Gets the number of visible pixels in one horizontal scan line.

Returns **number** – Number of pixels.

Return type int

isPixelSyncLineBlackEnabled()

Verifies the pixel synchronization black line mode state.

Allows the user to know if the raster line displayed black mode is enabled or not. When `enable` is True, the raster line displayed black mode is enabled. When disabled, the raster line is displayed normally.

Returns enable – True if black line mode is enabled, False otherwise.

Return type Bool

isPixelSyncOnSingleLineEnabled()

Verifies the pixel synchronization mode state.

Allows the user to know if the pixel synchronization is done on a single raster line or on any line.

Returns enable – True if pixel synchronization is only recognized on a single raster line, False otherwise.

Return type Bool

isVideoOnDualLinkDvi()

Verifies if the device is currently receiving video data over dual-link DVI.

Returns state – True if the device is receiving over dual-link DVI, otherwise False.

Return type Bool

isVideoOnDvi()

Verifies if the device is currently receiving video data over DVI link.

Returns state – True if the device is currently receiving video data over DVI link, otherwise False.

Return type Bool

isVideoTimingDisplayable()

Verifies if the device can display the incoming data.

This method allows the user to know if the device is currently receiving video whose timing can be directly driven by the display.

Returns state – True if the device can display the video signal, otherwise False.

Return type Bool

isVideoTimingTooHigh()

Verifies if the video clock frequency is too high.

This method allows the user to know if the clock frequency of the incoming data is too high for the device to display.

Returns state – True if the clock frequency is too high, otherwise False.

Return type Bool

isVideoVesaBluelineEnabled()

Verifies the video blue line mode state.

Returns enable – True if blue line mode is enabled, False otherwise.

Return type Bool

setCLUT(*CLUT*)

Sets the video color look up table data to be displayed.

Pass 3x256 16-bit video data, [[R1...R256], [G1...G256], [B1...B256]]. Or if you want a different CLUT for the console and the test display, Pass a 3x512 16-bit video, [[R1...R512], [G1...G512], [B1...B512]].

Warning: Since this is 16-bits, the colors RGB must vary between 0 and 65535.

This functions returns immediately, and CLUT is implemented at next vertical blanking interval.

Parameters **CLUT** (*list of lists of int*) – Color look-up table .

setPixelSyncLineBlack(*enable*)

Sets the Video pixel synchronization black line mode.

Allows the user to display the raster line black for video pixel synchronization. When enabled, the raster line is always displayed black. When disabled, the raster line is displayed normally.

Parameters **enable** (*Bool*) – True to enable the black line mode. False to disable it on any line.

setPixelSyncOnSingleLine(*enable*)

Sets the Video pixel synchronization on a single raster line.

Allows the user to set the video pixel synchronization on a single raster line or on any raster line.

Parameters **enable** (*Bool*) – True to enable pixel synchronization on a single raster line. False to allow synchronization on any line.

setRasterLinePixelSync(*line*)

Sets the raster line on which the pixel synchronization sequence is expected.

Parameters **line** (*int*) – line on which the pixel synchronization sequence is expected.

setVideoMode(*mode*)

Sets the video processing mode.

This method allows the user to change the display mode of the VPixx device. Each mode changes how the input video signal is used to display the video on the display.

Warning: Mode **RB24** is availible on VIEWPixx ONLY, revision 21 or higher.

Parameters **mode** (*string*) – video mode is one of the following predefined constants:

- **C24**: Straight pass through from DVI 8-bit (or HDMI "deep" 10/12-bit) RGB to VGA 8/10/12-bit RGB
- **L48**: DVI RED[7:0] is used as an index into a 256-entry 16-bit RGB color lookup table
- **M16**: DVI RED[7:0] & GREEN[7:0] concatenate into a VGA 16-bit value sent to all three RGB components
- **C48**: Even/Odd pixel RED/GREEN/BLUE[7:0] concatenate to generate 16-bit RGB components at half the horizontal resolution
- **L48D**: DVI RED[7:4] & GREEN[7:4] concatenate to form an 8-bit index into a 256-entry 16-bit RGB color lookup table
- **M16D**: DVI RED[7:3] & GREEN[7:3] & BLUE[7:2] concatenate into a VGA 16-bit value sent to all three RGB components
- **C36D**: Even/Odd pixel RED/GREEN/BLUE[7:2] concatenate to generate 12-bit RGB components at half the horizontal resolution
- **RB24**: DVI RED[7:0] & GREEN[7:4] concatenate to form 12-bit RED value, DVI BLUE[7:0] & GREEN[3:0] concatenate to form 12-bit BLUE value, GREEN is forced to 0.

setVideoVesaBlueline(*enable*)

Sets the Video blue line mode.

When enabled, the VESA 3D output interprets the middle pixel on the last raster line as a blue line code. When disabled, the VESA 3D output is not dependent on video content.

Parameters **enable** (*Bool*) – Activate or deactivate the blue line mode.

CHAPTER

TWENTYTWO

VGA OUT

class `vgaOut.`**`VgaOut`**

Bases: `object`

Implements the methods available for a DATAPixx 1.

`name`

definition

`isVideoHorizontalOverlayEnabled``()`

Returns the video horizontal overlay mode state.

Returns **enable** – True if the left/right halves of the video image are being overlayed. False otherwise

Return type Bool

`isVideoHorizontalSplitEnabled``()`

Returns the video horizontal split mode state.

Returns **enable** – True if video is being split across the two VGA outputs. False otherwise

Return type Bool

`isVideoVerticalStereoEnabled``()`

Returns the video vertical stereo mode state.

Returns **enable** – True if stereo mode is enabled. False otherwise

Return type Bool

`setVideoHorizontalOverlay``(`*enable=None*`)`

Sets the Video horizontal overlay mode.

VGA 1 and VGA 2 both show an overlay composite of the left/right halves of the video image. Horizontal overlay is disabled

Parameters **enable** (*Bool*) – True to enable the horizontal overlay, false to disable it.

`setVideoHorizontalSplit``(`*enable=None*`)`

Sets the Video horizontal split mode.

When this feature is enabled, VGA 1 shows left half of video image while VGA 2 shows right half of video image. The two VGA outputs are perfectly synchronized. When this feature is disabled, VGA 1 and VGA 2 both show the entire video image (hardware video mirroring). The device can also be set to automatically split video across the two VGA outputs if the horizontal resolution is at least twice the vertical resolution. This is considered as the automatic mode or default mode.

Parameters **enable** (*Bool, optional*) – True to enable the horizontal split, false to disable it. When no argument is passed, automatic mode is set.

setVideoVerticalStereo(*enable=None*)

Sets the Video vertical stereo mode.

When this feature is enabled, the top/bottom halves of the input image are output in two sequential video frames. VESA L/R output is set to 1 when first frame (left eye) is displayed, and set to 0 when second frame (right eye) is displayed.

When this feature is disabled, the images are displayed normally. The device can also enable vertical stereo automatically if vertical resolution > horizontal resolution. This is considered as the automatic mode or default mode.

Parameters **enable** (*Bool*) – True to enable the vertical stereo mode, false to disable it. When no argument is passed, automatic mode is set.

CHAPTER

TWENTYTHREE

I1DISPLAY

class `i1.`**`I1Display`**

Bases: `object`

Class Definition for the i1Display device.

`measurement`

Measurements taken by the i1Display.

```
>>> from pypixxlib.i1 import I1Display
>>> my_device = I1Display()
>>> my_device.function()
```

`close``()`

Releases the handle on the current I1Display device.

This method should be the last call made on a I1Display object.

`getAllMeasurement``()`

Gets all the measurements taken by the i1Display.

This method returns a list which contains all measurements taken since the creation of the I1Display object. Each item in the list is a dictionary with both chromaticity coordinates and luminance.

Returns **measurements** – All measurements taken by the I1Display.

Return type list

See Also: `getAllMeasurement`, `printLatestMeasurement`

`getCurrentCalibration``()`

Sets the i1Display calibration mode.

This method allows the user to set the calibration mode of the i1Display. The calibration mode should be chosen based on the display that the i1Display will be measuring.

Returns

mode – Calibration is one of the following predefined constants:

- **CCFL**: Calibration used for displays with CCFL backlights.
- **WHITE LED**: Calibration used for displays with white LED backlights.
- **RGB LED**: Calibration used for displays with RGB LED backlights.
- **CRT**: Calibration used for CRT.
- **PROJECTOR**: Calibration used for projectors.
- **VPIXX**: Calibration used for VIEWPixx and VIEWPixx /3D displays and PROPixx projectors.

- **EEG**: Calibration used for VIEWPixx /EEG displays.

Return type string

See Also: `getCurrentCalibration`

`getFirmwareDate()`

Gets the i1Display firmware date.

Returns **revision** – Firmware date.

Return type string

`getFirmwareRevision()`

Gets the i1Display firmware revision.

Returns **revision** – Firmware revision.

Return type string

`getIntegrationTime()`

Gets the integration time used by the i1Display.

This method allows the user to know the current integration time used by the i1Display to take measurements.

Returns **time** – Any value greater than 0.0.

Return type float

`getLatestMeasurement()`

Gets the latest measurements taken by the i1Display.

All measurements are kept in a list. This method returns the last item from that list, which is a dictionary. Using the `chrominance` key, gives the `x` and `y` coordinates of the measurement on the chromaticity chart. Using the `luminance` key gives the luminance value of the measurement.

Returns

measurements –

- **chrominance**: Tuple of floating points coordinates. Ranges from 0 to 1.
- **luminance**: floating point value in cd/m^2.

Return type dict

See Also: `getAllMeasurement`, `printLatestMeasurement`

`isDiffuserOn()`

Verifies if the i1Display's diffuser is currently placed in front of the detector.

Returns **state** – True if the diffuser is in front of the detector, otherwise False.

Return type Bool

`printLatestMeasurement()`

Displays the latest measurements taken by the i1Display.

This method allows the user to display on the monitor the last measurement taken by the I1Display.

Returns **measurements** – Chrominance coordinates and luminance with 4 decimal points.

Return type string

See Also: `getAllMeasurement`, `getLatestMeasurement`

runMeasurement()

Triggers an acquisition on the i1Display.

This method triggers an acquisition on the I1Dsiplay. Since the measurement is affected by the position of the diffuser, the method verifies if the diffuser is located in front of the I1Dsiplay detector. Once the acquisition is done, it is added to the measurement list.

setCurrentCalibration(*calibration*)

Sets the i1Display calibration mode.

This method allows the user to set the calibration mode of the i1Display. The calibration mode should be chosen based on the display that the i1Display will be measuring. The default value is RGB LED. When using a VPixx device, the default value should be kept.

Parameters **mode** (*string*) – Calibration is one of the following predefined constants:

- **CCFL**: Calibration used for displays with CCFL backlights.
- **WHITE LED**: Calibration used for displays with white LED backlights.
- **RGB LED**: Calibration used for displays with RGB LED backlights.
- **CRT**: Calibration used for CRT.
- **PROJECTOR**: Calibration used for projectors.
- **VPIXX**: Calibration used for VIEWPixx and VIEWPixx /3D displays and PROPixx projectors.
- **EEG**: Calibration used for VIEWPixx /EEG displays.

See Also: `getCurrentCalibration`

setIntegrationTime(*time=1.0*)

Sets the integration time for the i1Display.

This method allows the user to change the time taken by the i1Display to take measurements.

Parameters **time** (*float*) – Any value greater than 0.0.

class i1.**I1Pro**

Bases: `object`

Class Definition for the i1Pro device.

spectrum

Spectrum measurements taken by the i1Pro.

tristimulus

Tristimulus measurements taken by the i1Pro.

revision

Revision of the current i1Pro.

serial_number

Serial number of the current i1Pro.

```
>>> from pypixxlib.i1 import I1Pro()
>>> my_device = I1Pro()
>>> my_device.function()
```

calibrate(*measure_mode*)

Calibrates the i1Pro.

This method allows the user to calibrate the i1Pro. It should be noted that when a calibration is done, it is done for a specific 'measurement mode'. The required measurement mode must be specified.

Parameters **measure_mode** (*string*) – mode is one of the following predefined constants:

- **Emission**: Mode for an emission measurement on an emitting probe like a display.
- **Ambiant**: Mode for an ambient light measurement.

`close``()`

Releases the handle on the current i1Pro device.

This methods closes the i1Pro and should be the last call made on a I1Pro object.

`getAllSpectrumMeasurements``()`

Gets all the spectrum measurements taken by the i1Pro.

This method returns a list which contains all spectrum values taken since the creation of the i1Pro object.

Returns **spectrum** – All spectrum measurements taken by the i1Pro.

Return type list

See Also: `getLatestSpectrum`, `getAllTriStimulus`

`getAllTriStimulus``()`

Gets all the triStimulus measurements taken by the i1Pro.

This method returns a list which contains all triStimulus values taken since the creation of the i1Pro object.

Returns **spectrum** – All triStimulus measurements taken by the i1Pro.

Return type list

See Also: `getLatestTriStimulus`

`getColorSpace``()`

Gets the current color space mode for the i1Pro.

This method allows the user to know what is the current color space mode on the i1Pro.

Returns

mode – measurement mode is one of the following predefined constants:

- **CIELab**: Mode for an emission measurement on an emitting probe like a display.
- **CIELCh**: Mode for an ambient light measurement.
- **CIELuv**:
- **CIELChuv**:
- **CIEuvY1960**:
- **CIEuvY1976**:
- **CIEXYZ**:
- **CIExyY**:

Return type string

See Also: `setColorSpace`

`getConnectionStatus``()`

Gets the current connection status on the i1Pro.

This method allows the user to know what is the current connection status on the i1Pro. When an error occurs, this method will give the user an error code which indicates what the problem is.

Returns **mode** – Connection status.

Return type string

`getIlluminationMode``()`

Gets the current color illumination mode on the i1Pro.

This method allows the user to know what is the current illumination mode on the i1Pro.

Returns

mode – mode is one of the following predefined constants:

- **A**
- **B**
- **C**
- **D50**
- **D55**
- **D65**
- **D75**
- **F2**
- **F7**
- **F11**
- **Emission**

Return type string

`getLatestSpectrumMeasurements``()`

Gets the latest spectrum measurements taken by the i1Pro.

All measurements are kept in a list. This method returns the last item from that list.

Returns **measurements** – Latest spectrum measurement taken by he i1Pro.

Return type list

See Also: `getAllSpectrumMeasurements`

`getLatestTriStimulusMeasurements``()`

Gets the latest triStimulus measurements taken by the i1Pro.

All measurements are kept in a list. This method returns the last item from that list.

Returns **measurements** – Latest spectrum measurement taken by he i1Pro.

Return type list

See Also: `getAllSpectrumMeasurements`

`getMeasurementMode``()`

Gets the current measurement mode on the i1Pro.

This method allows the user to know what is the current measurement mode on the i1Pro.

Returns

mode – measurement mode is one of the following predefined constants:

- **Emission**: Mode for an emission measurement on an emitting probe like a display.
- **Ambiant**: Mode for an ambient light measurement.

Return type string

See Also: `setMeasurementMode`

`getTimeUntilCalibrationExpire`()

Gets the time left until the calibration on the i1Pro expires.

This method allows the user to know how much time is left until a calibration needs to be performed on the i1Pro. When an error occurs, this method will give the user an error code which indicates what the problem is.

Returns **time** – Time is a value greater than -1. When -1 is returned, it indicates that the time has expired.

Return type int

`isCalibrationExpired`()

Verifies if the calibration on the i1Pro has expired.

In order to get precise results, the i1Pro needs to be calibrated frequently. This method allows the user to know if the calibration needs to be done. It should be noted that a calibration is required before each use of the device.

Returns **state** – True if the calibration has expired, otherwise False.

Return type Bool

`printLatestMeasurement`()

Displays the latest measurements taken by the i1Pro.

This method allows the user to display on the monitor the last measurement taken by the I1Pro.

Returns **measurements** – Chrominance coordinates and luminance with 4 decimal points.

Return type string

See Also: `getAllTriStimulus`, `getLatestSpectrumMeasurements`

`runMeasurement`()

Triggers an acquisition on the i1Pro.

This method triggers an acquisition on the i1Pro. Once the acquisition is done, it is added to the measurement list.

`setColorSpace`(*mode*)

Sets the color space mode for the i1Pro.

This method allows the user to change the color space mode used during measurements. The mode should be chosen based on the light the source.

Parameters **mode** (*string*) – measurement mode is one of the following predefined constants:

- **CIELab**: Mode for an emission measurement on an emitting probe like a display.
- **CIELCh**: Mode for an ambient light measurement.
- **CIELuv**:
- **CIELChuv**:
- **CIEuvY1960**:
- **CIEuvY1976**:
- **CIEXYZ**:
- **CIExyY**:

See Also: `getColorSpace`

`setIlluminationMode`(*mode='Emission'*)

Sets the illumination mode the i1Pro.

This method allows the user to change the illumination mode used during measurements.

Parameters **mode** (*string*) – mode is one of the following predefined constants:

- **A**
- **B**
- **C**
- **D50**
- **D55**
- **D65**
- **D75**
- **F2**
- **F7**
- **F11**
- **Emission**

`setMeasurementMode`(*mode*, *adaptative=False*)

Sets the measurement mode on the i1Pro.

This method allows the user to change the measurement mode of the i1Pro. The mode should be chosen based on the light source.

Parameters

- **mode** (*string*) – measurement mode is one of the following predefined constants:
  - **Emission**: Mode for an emission measurement on an emitting probe like a display.
  - **Ambiant**: Mode for an ambient light measurement.
- **adaptative** (*Bool*) – When this mode is `True`, a trial measurement is done first to get the best measurement result. When it is `False`, the measurement duration is lower, but the results precision is also lower.

See Also: `getMeasurementMode`

`setModes`(*measurement='Ambiant'*, *color_space='CIExyY'*, *illumination='Emission'*)

Sets the measurement, color space and illumination mode on the i1Pro.

This method allows the user to set the three i1Pro modes in a single method call.

Parameters

- **measurement** (*string*) – One of the following predefined constants:
  - **Emission**: Mode for an emission measurement on an emitting probe like a display.
  - **Ambiant**: Mode for an ambient light measurement.
- **color_space** (*string*) – One of the following predefined constants:
  - **CIELab**: Mode for an emission measurement on an emitting probe like a display.
  - **CIELCh**: Mode for an ambient light measurement.

  - **CIELuv**:
  - **CIELChuv**:
  - **CIEuvY1960**:
  - **CIEuvY1976**:
  - **CIEXYZ**:
  - **CIExyY**:
- **illumination** (*string*) – One of the following predefined constants:
  - **A**
  - **B**
  - **C**
  - **D50**
  - **D55**
  - **D65**
  - **D75**
  - **F2**
  - **F7**
  - **F11**
  - **Emission**

See Also: `setMeasurementMode`, `setColorSpace`, `setIlluminationMode`, `getMeasurementMode`, `getColorSpace`, `getIlluminationMode`,

CHAPTER

TWENTYFOUR

IMPORTANT

For more information about our company and products, visit our Web site at www.vpixx.com

For information, comments or suggestions, please contact us by e-mail at support@vpixx.com

Our offices are located at: 630, Clairevue West, Suite 301 Saint-Bruno, Quebec Canada J3V 6B4

CHAPTER TWENTYFIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*