labscript-devices

Release 3.2.0

labscript suite contributors

Apr 13, 2023

DOCUMENTATION

This portion of the **labscript-suite** contains the plugin architecture for controlling experimental hardware. In particular, this code provides the interface between **labscript** high-level instructions and hardware-specific instructions, the communication interface to send those instructions to the hardware, and the **BLACS** instrument control interface.

CHAPTER

ONE

INTRODUCTION

The **labscript_devices** module contains the low-level hardware interfacing code that intermediates between the labscript API (converting **labscript** instructions into hardware instructions) as well as the BLACS GUI (which communicates directly with the hardware).

Each "device" is made up of four classes that handle the various tasks.

- `labscript_device` (derives from `Device`)
  - Defines the interface between the **labscript** API and generates hardware instructions that can be saved to the shot h5 file.
- `BLACS_tab` (derives from `DeviceTab`)
  - Defines the graphical tab that is present in the **BLACS** GUI. This tab provides graphical widgets for controlling hardware outputs and visualizing hardware inputs.
- `BLACS_worker` (derives from `Worker`)
  - Defines the software control interface to the hardware. The `BLACS_tab` spawns a process that uses this class to send and receive commands with the hardware.
- `runviewer_parser`
  - Defines a software interface that interprets hardware instructions in a shot h5 file and displays them in the runviewer GUI.

The **labscript_suite** provides an extensive *list of device classes* for commercially available hardware. Furthermore, it is simple to add local *user devices* to control instruments not already within the labscript-suite.

CHAPTER

TWO

DEVICES

Here is a list of all the currently supported devices.

2.1 Pseudoclocks

Pseudoclocks provide the timing backbone of the labscript_suite. These devices produce hardware-timed clocklines that trigger other device outputs and acquisitions. Many pseudoclock devices also include other types of outputs, including digital voltage and DDS frequency synthesizers.

2.1.1 Pulseblaster

This labscript device controls the Spincore PulseblaserDDS-II-300-AWG. The Pulseblaster is a programmable pulse generator that is the typical timing backbone of an experiment (ie it generates the pseudoclock timing pulses that control execution of other devices in the experiment). This labscript device is the master implementation of the various Pulseblaster devices. Other Pulseblaster labscript devices subclass this device and make the relevant changes to hard-coded values. Most importantly, the `core_clock_freq` must be manually set to match that of the Pulseblaster being used in order for the timing of the programmed pulses to be correct (in the `labscript_device` and the `BLACS_worker`).

This particular version of Pulseblaster has a 75 MHz core clock frequency and also has DDS synthesizer outputs.

Installation

Use of the Pulseblaster requires driver installation available from the manufacturer here. The corresponding python wrapper, spinapi is available via pip.

```
pip install -U spinapi
```

Usage

```
from labscript import *

from labscript_devices.PulseBlaster import PulseBlaster

PulseBlaster(name='pb',board_number=0,programming_scheme='pb_start/BRANCH')

Clockline(name='pb_clockline_fast', pseudoclock=pb.pseudoclock,connection='flag 0')
Clockline(name='pb_clockline_slow', pseudoclock=pb.pseudoclock,connection='flag 1')

DigitalOut(name='pb_0',parent_device=pb.direct_outputs,connection='flag 2')

PulseBlasterDDS(name='pb_dds_0',parent_device=pb.direct_outputs, 'channel 0')

start()

stop(1)
```

Detailed Documentation

class labscript_devices.PulseBlaster.**PulseBlaster**(*name*, *trigger_device=None*, *trigger_connection=None*, *board_number=0*, *firmware=''*, *programming_scheme='pb_start/BRANCH'*, *pulse_width='symmetric'*, *max_instructions=4000*, *time_based_stop_workaround=False*, *time_based_stop_workaround_extra_time=0.5*, ***kwargs*)

Bases: labscript.labscript.PseudoclockDevice

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.

- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

_check_wait_monitor_ok()

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript.labscript.Pseudoclock'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 8300000.0

clock_resolution = 2.6666666666666667e-08

convert_to_pb_inst(*dig_outputs*, *dds_outputs*, *freqs*, *amps*, *phases*)

core_clock_freq = 75

description = 'PB-DDSII-300'
Brief description of the device.

property direct_outputs

flag_is_clock(*flag*)

flag_valid(*flag*)

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

generate_registers(*hdf5_file*, *dds_outputs*)

get_direct_outputs()
Finds out which outputs are directly attached to the PulseBlaster

get_flag_number(*connection*)

n_flags = 12

pb_instructions = {'BRANCH': 6, 'CONTINUE': 0, 'END_LOOP': 3, 'LONG_DELAY': 7, 'LOOP': 2, 'STOP': 1, 'WAIT': 8}

property pseudoclock

trigger_delay = 2.5e-07

trigger_edge_type = 'falling'

Type of trigger. Must be `'rising'` or `'falling'`.

Type str

wait_delay = 1e-07

write_pb_inst_to_h5(*pb_inst*, *hdf5_file*)

class labscript_devices.PulseBlaster.**PulseBlasterDDS**(**args*, ***kwargs*)

Bases: `labscript.labscript.DDSQuantity`

Instantiates a DDS quantity.

Parameters

- **name** (*str*) – python variable for the object created.
- **parent_device** (`IntermediateDevice`) – Device this output is connected to.
- **connection** (*str*) – Output of parent device this DDS is connected to.
- **digital_gate** (*dict, optional*) – Configures a digital output to use as an enable/disable gate for the output. Should contain keys `'device'` and `'connection'` with arguments for the `parent_device` and `connection` for instantiating the `DigitalOut`.
- **freq_limits** (*tuple, optional*) – `(lower, upper)` limits for the frequency of the output
- **freq_conv_class** (`labscript_utils:labscript_utils.unitconversions`, optional) – Unit conversion class for the frequency of the output.
- **freq_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the frequency of the output.
- **amp_limits** (*tuple, optional*) – `(lower, upper)` limits for the amplitude of the output
- **amp_conv_class** (`labscript_utils:labscript_utils.unitconversions`, optional) – Unit conversion class for the amplitude of the output.
- **amp_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the amplitude of the output.
- **phase_limits** (*tuple, optional*) – `(lower, upper)` limits for the phase of the output

- **phase_conv_class** (labscript_utils:labscript_utils.unitconversions, optional) – Unit conversion class for the phase of the output.
- **phase_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the phase of the output.
- **call_parents_add_device** (*bool, optional*) – Have the parent device run its add_device method.
- ****kwargs** – Keyword arguments passed to Device.__init__().

description = 'PulseBlasterDDS'
Brief description of the device.

hold_phase(*t*)

release_phase(*t*)

class labscript_devices.PulseBlaster.**PulseBlasterDirectOutputs**(*name*, *parent_device*, ***kwargs*)
Bases: labscript.labscript.IntermediateDevice

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (ClockLine) – Parent ClockLine device.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript.labscript.DDS'>, <class 'labscript_devices.PulseBlaster.PulseBlasterDDS'>, <class 'labscript.labscript.DigitalOut'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 8300000.0

description = 'PB-DDSII-300 Direct Outputs'
Brief description of the device.

class labscript_devices.PulseBlaster.**PulseBlasterParser**(*path*, *device*)
Bases: object

_add_pulse_program_row_from_buffer(*traces*, *index*)

_add_pulse_program_row_to_traces(*traces*, *row*, *dds*, *flags=None*)

get_traces(*add_trace*, *parent=None*)

labscript_device_class_name = 'PulseBlaster'

num_dds = 2

num_flags = 12

class labscript_devices.PulseBlaster.PulseBlasterTab(*notebook*, *settings*, *restart=False*)

Bases: blacs.device_base_class.DeviceTab

get_child_from_connection_table(*parent_device_name*, *port*)

initialise_GUI()

labscript_device_class_name = 'PulseBlaster'

reset(**args*, ***kwargs*)

start(**args*, ***kwargs*)

start_run(**args*, ***kwargs*)

status_monitor(**args*, ***kwargs*)

stop(**args*, ***kwargs*)

class labscript_devices.PulseBlaster.PulseblasterWorker(**args*, ***kwargs*)

Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

check_status()

init()

program_manual(*values*)

shutdown()

start_run()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

labscript_devices.PulseBlaster.**profile**(*funct*)

labscript_devices.PulseBlaster.**start_profile**(*name*)

labscript_devices.PulseBlaster.**stop_profile**(*name*)

2.1.2 Pulseblaster (-DDS)

Overview

This labscript device controls the Spincore Pulseblasers that do not have DDS outputs. The Pulseblaster is a programmable pulse generator that is the typical timing backbone of an experiment (ie it generates the pseudoclock timing pulses that control execution of other devices in the experiment). This labscript device inherits from the *Pulseblaster* device. The primary difference is the removal of code handling DDS outputs.

The labscript-suite currently supports a number of no-dds variants of the Pulseblaster device, each with different numbers of outputs and clock frequencies:

- `PulseBlaster_No_DDS`: Has 24 digital outputs and a 100 MHz core clock frequency.
- `PulseBlasterUSB`: Identical to the `PulseBlaster_No_DDS` device
- `PulseBlaster_SP2_24_100_32k`: Has slightly lower `clock_limit` and `clock_resolution` than the standard device. Also supports 32k instructions instead of the standard 4k.
- `PulseBlasterESRPro200`: Has a 200 MHz core clock frequency.
- `PulseBlasterESRPro500`: Has a 500 MHz core clock frequency.

ESR-Pro PulseBlasters

The timing resolution of a normal PulseBlaster is one clock cycle, the minimum interval is typically limited to 5 clock cycles (or nine in the case of the external memory models like the 32k). The ESR-Pro series of PulseBlasters have the Short Pulse Feature, which allows for pulse lengths of 1-5 clock periods. This is controlled using the top three bits (21-23) according to the following table.

Table 1: Short Pulse Control

SpinAPI Define	Bits 21-23	Clock Periods	Pulse Length (ns) at 500 MHz
-	000	-	All outputs low
ONE_PERIOD	001	1	2
TWO_PERIOD	010	2	4
THREE_PERIOD	011	3	6
FOUR_PERIOD	100	4	8
FIVE_PERIOD	101	5	10
ON	111	-	Short Pulse Disabled

Currently, the PulseBlaster labscript device does not use this functionality. However, in order to get any output at all, bits 21-23 must be set high manually.

Installation

Use of the Pulseblaster requires driver installation available from the manufacturer here. The corresponding python wrapper, spinapi is available via pip.

```
pip install -U spinapi
```

Usage

```
from labscript import *

from labscript_devices.PulseBlaster import PulseBlaster

PulseBlaster(name='pb',board_number=0,programming_scheme='pb_start/BRANCH')

Clockline(name='pb_clockline_fast', pseudoclock=pb.pseudoclock,connection='flag 0')
Clockline(name='pb_clockline_slow', pseudoclock=pb.pseudoclock,connection='flag 1')

DigitalOut(name='pb_0',parent_device=pb.direct_outputs,connection='flag 2')

start()

stop(1)
```

Detailed Documentation

labscript_devices.PulseBlaster_No_DDS

labscript_devices.PulseBlasterUSB

labscript_devices.PulseBlaster_SP2_24_100_32k

labscript_devices.PulseBlasterESRPro200

labscript_devices.PulseBlasterESRPro500

PulseBlaster_No_DDS

class labscript_devices.PulseBlaster_No_DDS.**PulseBlaster_No_DDS**(*name*, *trigger_device=None*, *trigger_connection=None*, *board_number=0*, *firmware=''*, *programming_scheme='pb_start/BRANCH'*, *pulse_width='symmetric'*, *max_instructions=4000*, *time_based_stop_workaround=False*, *time_based_stop_workaround_extra_time=0.5*, ***kwargs*)

Bases: *labscript_devices.PulseBlaster.PulseBlaster*

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

clock_limit = 8300000.0

clock_resolution = 2e-08

core_clock_freq = 100

description = 'generic DO only Pulseblaster'
Brief description of the device.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (`h5py.File`) – Handle to shot file.

n_flags = 24

write_pb_inst_to_h5(*pb_inst*, *hdf5_file*)

class labscript_devices.PulseBlaster_No_DDS.**PulseBlaster_No_DDS_Parser**(*path*, *device*)
Bases: *labscript_devices.PulseBlaster.PulseBlasterParser*

labscript_device_class_name = 'PulseBlaster_No_DDS'

num_dds = 0

num_flags = 24

class labscript_devices.PulseBlaster_No_DDS.**PulseblasterNoDDSWorker**(**args*, ***kwargs*)
Bases: `blacs.tab_base_classes.Worker`

abort_buffered()

abort_transition_to_buffered()

check_status()

core_clock_freq = 100

init()

program_manual(*values*)

shutdown()

start_run()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.PulseBlaster_No_DDS.**Pulseblaster_No_DDS_Tab**(**args*, ***kwargs*)
Bases: `blacs.device_base_class.DeviceTab`

get_child_from_connection_table(*parent_device_name*, *port*)

initialise_GUI()

labscript_device_class_name = 'PulseBlaster_No_DDS'

num_DO = 24

reset(**args*, ***kwargs*)

start(**args*, ***kwargs*)

start_run(**args*, ***kwargs*)

status_monitor(**args*, ***kwargs*)

stop(**args*, ***kwargs*)

PulseBlasterUSB

class labscript_devices.PulseBlasterUSB.**PulseBlasterUSB**(*name*, *trigger_device=None*, *trigger_connection=None*, *board_number=0*, *firmware=''*, *programming_scheme='pb_start/BRANCH'*, *pulse_width='symmetric'*, *max_instructions=4000*, *time_based_stop_workaround=False*, *time_based_stop_workaround_extra_time=0.5*, ***kwargs*)

Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS*

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

clock_limit = 8300000.0

clock_resolution = 2e-08

core_clock_freq = 100.0

description = 'SpinCore PulseBlasterUSB'
Brief description of the device.

n_flags = 24

class labscript_devices.PulseBlasterUSB.**PulseBlasterUSBParser**(*path*, *device*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS_Parser*

labscript_device_class_name = 'PulseBlasterUSB'

class labscript_devices.PulseBlasterUSB.**PulseblasterUSBTab**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.Pulseblaster_No_DDS_Tab*

labscript_device_class_name = 'PulseBlasterUSB'

num_DO = 24

class labscript_devices.PulseBlasterUSB.**PulseblasterUSBWorker**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseblasterNoDDSWorker*

core_clock_freq = 100.0

PulseBlaster_SP2_24_100_32k

class labscript_devices.PulseBlaster_SP2_24_100_32k.**PulseBlaster_SP2_24_100_32k**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS*

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

clock_limit = 5000000.0

clock_resolution = 1e-08

core_clock_freq = 100.0

description = 'SpinCore PulseBlaster-SP2-24-100-32k'
Brief description of the device.

n_flags = 24

class labscript_devices.PulseBlaster_SP2_24_100_32k.**PulseBlaster_SP2_24_100_32k_Parser**(*path*, *device*)
Bases: *labscript_devices.PulseBlaster.PulseBlasterParser*

labscript_device_class_name = 'PulseBlaster_SP2_24_100_32k'

num_dds = 0

num_flags = 24

class labscript_devices.PulseBlaster_SP2_24_100_32k.**PulseBlaster_SP2_24_100_32k_Tab**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.Pulseblaster_No_DDS_Tab*

labscript_device_class_name = 'PulseBlaster_SP2_24_100_32k'

num_DO = 24

class labscript_devices.PulseBlaster_SP2_24_100_32k.**PulseBlaster_SP2_24_100_32k_Worker**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseblasterNoDDSWorker*

core_clock_freq = 100.0

PulseBlasterESRPro200

class labscript_devices.PulseBlasterESRPro200.**PulseBlasterESRPro200**(*name*, *trigger_device=None*, *trigger_connection=None*, *board_number=0*, *firmware=''*, *programming_scheme='pb_start/BRANCH'*, *pulse_width='symmetric'*, *max_instructions=4000*, *time_based_stop_workaround=False*, *time_based_stop_workaround_extra_time=0.5*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS*

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

clock_limit = 200000000.0

clock_resolution = 5e-09

core_clock_freq = 200.0

description = 'SpinCore PulseBlaster ESR-PRO-200'
Brief description of the device.

n_flags = 21

class labscript_devices.PulseBlasterESRPro200.**PulseblasterESRPro200Parser**(*path*, *device*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS_Parser*

labscript_device_class_name = 'PulseBlasterESRPro200'

num_dds = 0

num_flags = 21

class labscript_devices.PulseBlasterESRPro200.**PulseblasterESRPro200Worker**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseblasterNoDDSWorker*

core_clock_freq = 200.0

class labscript_devices.PulseBlasterESRPro200.**pulseblasteresrpro200**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.Pulseblaster_No_DDS_Tab*

labscript_device_class_name = 'PulseBlasterESRPro200'

num_DO = 21

PulseBlasterESRPro500

class labscript_devices.PulseBlasterESRPro500.**PulseBlasterESRPro500**(*name*, *trigger_device=None*, *trigger_connection=None*, *board_number=0*, *firmware=''*, *programming_scheme='pb_start/BRANCH'*, *pulse_width='symmetric'*, *max_instructions=4000*, *time_based_stop_workaround=False*, *time_based_stop_workaround_extra_time=0.5*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseBlaster_No_DDS*

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.

- **trigger_connection** (*str, optional*) – Must be provided if `trigger_device` is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to `TriggerableDevice.__init__()`.

clock_limit = 50000000.0

clock_resolution = 4e-09

core_clock_freq = 500.0

description = 'SpinCore PulseBlaster ESR-PRO-500'
Brief description of the device.

n_flags = 21

class labscript_devices.PulseBlasterESRPro500.**PulseblasterESRPro500Worker**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.PulseblasterNoDDSWorker*

core_clock_freq = 500.0

class labscript_devices.PulseBlasterESRPro500.**pulseblasteresrpro500**(**args*, ***kwargs*)
Bases: *labscript_devices.PulseBlaster_No_DDS.Pulseblaster_No_DDS_Tab*

labscript_device_class_name = 'PulseBlasterESRPro500'

num_DO = 21

2.1.3 Cicero Opal-Kelly XEM3001

A pseudoclocking labscript device based on the OpalKelly XEM3001 integration module, which uses a Xilinx Spartan-3 FPGA.

Installation

Firmware (.bit) files for the FPGA are available here and should be placed in the labscript_devices folder along with the `CiceroOpalKellyXEM3001.py` file. The Opal Kelly SDK, which provides the python bindings, is also required. The python bindings will need to either be added to the PATH or manually copied to the site-packages of the virtual environment that BLACS is running in.

Detailed Documentation

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001**(*name*, *trigger_device=None*, *trigger_connection=None*, *serial=''*, *reference_clock='internal'*, *clock_frequency=100000000.0*, *use_wait_monitor=False*, *trigger_debounce_clock_ticks=10*)

Bases: labscript.labscript.PseudoclockDevice

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

add_device(*device*)

Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript_devices.CiceroOpalKellyXEM3001.CiceroOpalKellyXEM3001Pseudoclock'>, <class 'labscript_devices.CiceroOpalKellyXEM3001.CiceroOpalKellyXEM3001DummyPseudoclock'>]

Defines types of devices that are allowed to be children of this device.

Type list

property clockline

description = 'CiceroOpalKellyXEM3001'

Brief description of the device.

generate_code(*hdf5_file*)

Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

property internal_wait_monitor_outputs

max_instructions = 2048

property pseudoclock

trigger_edge_type = 'rising'
Type of trigger. Must be 'rising' or 'falling'.

Type str

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001DummyClockLine**(*name*, *pseudoclock*, *connection*, *ramping_allowed=True*, ***kwargs*)

Bases: labscript.labscript.ClockLine

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.
- **added_properties** (*dict, optional*) –
- **gui** –
- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(**args*, ***kwargs*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001DummyIntermediateDevice**(*name*, *parent_device*, ***kwargs*)

Bases: labscript.labscript.IntermediateDevice

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (ClockLine) – Parent ClockLine device.

add_device(*device*)

Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(**args*, ***kwargs*)

Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001DummyPseudoclock**(*name*, *pseudoclock_device*, *connection*, ***kwargs*)

Bases: labscript.labscript.Pseudoclock

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (PseudoclockDevice) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to Device().

add_device(*device*)

Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(**args*, ***kwargs*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (`h5py.File`) – Handle to shot file.

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001Pseudoclock**(*name*, *pseudoclock_device*, *connection*, ***kwargs*)
Bases: `labscript.labscript.Pseudoclock`

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (`PseudoclockDevice`) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to `Device()`.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (`Device`) – Device to add.

Raises **LabscriptError** – If `device` is not an allowed child of this device.

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001Tab**(*notebook*, *settings*, *restart=False*)
Bases: `blacs.device_base_class.DeviceTab`

close_tab(**args*, ***kwargs*)
Close the tab, terminate subprocesses and join the mainloop thread. If finalise=False, then do not terminate subprocesses or join the mainloop. In this case, callers must manually call finalise_close_tab() to perform these potentially blocking operations

flash_fpga(**args*, ***kwargs*)

get_child_from_connection_table(*parent_device_name*, *port*)

get_save_data()

initialise_GUI()

labscript_device_class_name = 'CiceroOpalKellyXEM3001'

restore_save_data(*data*)

start_run(**args*, ***kwargs*)

status_monitor(**args*, ***kwargs*)

class labscript_devices.CiceroOpalKellyXEM3001.**CiceroOpalKellyXEM3001Worker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort()

abort_buffered()

abort_transition_to_buffered()

flash_FPGA()

init()

program_manual(*values*)

shutdown()

start_run()

status_monitor()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.CiceroOpalKellyXEM3001.**RunviewerClass**(*path*, *device*)
Bases: object

get_traces(*add_trace*, *clock=None*)

labscript_device_class_name = 'CiceroOpalKellyXEM3001'

labscript_devices.CiceroOpalKellyXEM3001.**add_instruction_to_bytearray**(*data*, *instruction*, *on*, *off*, *reps*)

labscript_devices.CiceroOpalKellyXEM3001.**bits_to_int**(*m*, **args*)

labscript_devices.CiceroOpalKellyXEM3001.**int_to_bytes**(*n*, *m*)

2.1.4 Pineblaster

This labscript device controls the PineBlaster open-source digital pattern generator based on the Digilent chipKIT Max32 Prototyping platform.

Detailed Documentation

class labscript_devices.PineBlaster.**PineBlaster**(*name*, *trigger_device=None*, *trigger_connection=None*, *usbport='COM1'*)
Bases: labscript.labscript.PseudoclockDevice

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript_devices.PineBlaster.PineBlasterPseudoclock'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 10000000.0

clock_resolution = 2.5e-08

clock_type = 'fast clock'

property clockline

description = 'PineBlaster'
Brief description of the device.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

max_instructions = 15000

property pseudoclock

trigger_delay = 3.5e-07

wait_delay = 2.5e-06

class labscript_devices.PineBlaster.PineBlasterPseudoclock(*name*, *pseudoclock_device*, *connection*, ***kwargs*)
Bases: labscript.labscript.Pseudoclock

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (PseudoclockDevice) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to Device().

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

class labscript_devices.PineBlaster.PineblasterTab(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

get_child_from_connection_table(*parent_device_name*, *port*)

initialise_GUI()

labscript_device_class_name = 'PineBlaster'

start_run(**args*, ***kwargs*)

status_monitor(**args*, ***kwargs*)

class labscript_devices.PineBlaster.PineblasterWorker(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort()

abort_buffered()

abort_transition_to_buffered()

init()

program_manual(*values*)

shutdown()

start_run()

status_monitor()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.PineBlaster.**RunviewerClass**(*path*, *device*)

Bases: object

clock_resolution = 2.5e-08

clock_type = 'fast clock'

get_traces(*add_trace*, *clock=None*)

labscript_device_class_name = 'PineBlaster'

trigger_delay = 1e-06

wait_delay = 2.5e-06

2.1.5 PrawnBlaster

This labscript device controls the PrawnBlaster open-source digital pattern generator based on the Raspberry Pi Pico platform.

Specifications

The PrawnBlaster takes advantage of the specs of the Pico to provide the following:

- Configurable as 1, 2, 3, or 4 truly independent pseudoclocks.
  - Each clock has its own independent instruction set and synchronization between clocks is not required.
  - Assuming the default internal clock of 100 MHz, each clock has:
    - Minimum pulse half-period of 50 ns
    - Maximum pulse half-period of 42.9 s

        * Half-period resolution of 10 ns
- 30,000 instructions (each with up to 2^32 repetitions) distributed evenly among the configured pseudoclocks; 30,000, 15,000, 10,000, and 7,500 for 1, 2, 3, 4 pseudoclocks respectively.
- Support for external hardware triggers (external trigger common to all pseudoclocks)
  - Up to 100 retriggers (labscript-suite waits) per pseudoclock
  - Each wait can support a timeout of up to 42.9 s
  - Each wait is internally monitored for its duration (resolution of +/-10 ns)
- Can be referenced to an external LVCMOS clock
- Internal clock can be set up to 133 MHz (timing specs scale accordingly)

Installation

In order to turn the standard Pico into a PrawnBlaster, you need to load the custom firmware available in the Github repo onto the board. The simplest way to do this is by holding the reset button on the board while plugging the USB into a computer. This will bring up a mounted folder that you copy-paste the firmware to. Once copied, the board will reset and be ready to go.

Note that this device communicates using a virtual COM port. The number is assigned by the controlling computer and will need to be determined in order for BLACS to connect to the PrawnBlaster.

Usage

The default pinout for the PrawnBlaster is as follows:

- Pseudoclock 0 output: GPIO 9
- Pseudoclock 1 output: GPIO 11
- Pseudoclock 2 output: GPIO 13
- Pseudoclock 3 output: GPIO 15
- External Triggeer input: GPIO 0
- External Clock input: GPIO 20

Note that signal cable grounds should be connected to the digital grounds of the Pico for proper operation.

The PrawnBlaster provides up to four independent clocklines. They can be accessed either by `name.clocklines[int]` or directly by their auto-generated labscript names `name_clock_line_int`.

An example connection table that uses the PrawnBlaster:

```
from labscript import *

from labscript_devices.PrawnBlaster.labscript_devices import PrawnBlaster
from labscript_devices.NI_DAQmx.models.NI_USB_6363 import NI_USB_6363

PrawnBlaster(name='prawn', com_port='COM6', num_pseudoclocks=1)

NI_USB_6363(name='daq', MAX_name='Dev1',
            parent_device=prawn.clocklines[0], clock_terminal='/Dev1/PFI0',
            acquisition_rate=100e3)

AnalogOut('ao0', daq, 'ao0')
AnalogOut('ao1', daq, 'ao1')

if __name__ == '__main__':

   start(0)

   stop(1)
```

Detailed Documentation

class labscript_devices.PrawnBlaster.labscript_devices.**PrawnBlaster**(*name*, *trigger_device=None*, *trigger_connection=None*, *com_port='COM1'*, *num_pseudoclocks=1*, *out_pins=None*, *in_pins=None*, *clock_frequency=100000000.0*, *external_clock_pin=None*, *use_wait_monitor=True*)

Bases: labscript.labscript.PseudoclockDevice

PrawnBlaster Pseudoclock labscript device.

This labscript device creates Pseudoclocks based on the PrawnBlaster, a Raspberry Pi Pico with custom firmware.

Parameters

- **name** (*str*) – python variable name to assign to the PrawnBlaster

- **com_port** (*str*) – COM port assigned to the PrawnBlaster by the OS. Takes the form of `'COMd'`, where `d` is an integer.
- **num_pseudoclocks** (*int*) – Number of pseudoclocks to create. Ranges from 1-4.
- **trigger_device** (`IntermediateDevice`, optional) – Device that will send the hardware start trigger when using the PrawnBlaster as a secondary Pseudoclock.
- **trigger_connection** (*str, optional*) – Which output of the `trigger_device` is connected to the PrawnBlaster hardware trigger input.
- **out_pins** (*list, optional*) – What outpins to use for the pseudoclock outputs. Must have length of at least `num_pseudoclocks`. Defaults to `[9,11,13,15]`
- **in_pins** (*list, optional*) – What inpins to use for the pseudoclock hardware triggering. Must have length of at least `num_pseudoclocks`. Defaults to `[0,0,0,0]`
- **clock_frequency** (*float, optional*) – Frequency of clock. Standard range accepts up to 133 MHz. An experimental overclocked firmware is available that allows higher frequencies.
- **external_clock_pin** (*int, optional*) – If not `None` (the default), the PrawnBlaster uses an external clock on the provided pin. Valid options are `20` and `22`. The external frequency must be defined using `clock_frequency`.
- **use_wait_monitor** (*bool, optional*) – Configure the PrawnBlaster to perform its own wait monitoring.

add_device(*device*)

Adds child devices.

This is automatically called by the labscript compiler.

Parameters **device** (*_PrawnBlasterPseudoclock* or *_PrawnBlasterDummyPseudoclock*) – Instance to attach to the device. Only the allowed children can be attached.

allowed_children = [<class 'labscript_devices.PrawnBlaster.labscript_devices._PrawnBlasterPseudoclock'>, <class 'labscript_devices.PrawnBlaster.labscript_devices._PrawnBlasterDummyPseudoclock'>]

Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 10000000.0

Maximum allowable clock rate.

clock_resolution = 2e-08

Minimum resolvable time for a clock tick.

property clocklines

Returns a list of the automatically generated `ClockLine` objects.

description = 'PrawnBlaster'
Brief description of the device.

generate_code(*hdf5_file*)
Generates the hardware instructions for the pseudoclocks.

This is automatically called by the labscript compiler.

Parameters **hdf5_file** (h5py.File) – h5py file object for shot

input_response_time = 5e-08
Time necessary for hardware to respond to a hardware trigger. Empirically determined to be a ~50 ns buffer on the input.

property internal_wait_monitor_outputs

max_instructions = 30000
Maximum numaber of instructions per pseudoclock. Max is 30,000 for a single pseudoclock.

property pseudoclocks
Returns a list of the automatically generated *_PrawnBlasterPseudoclock* objects.

trigger_delay = 1.3e-07
Processing time delay after trigger is detected. Due to firmware, there is an 80 ns delay between trigger detection and first output pulse.

trigger_minimum_duration = 1.6e-07
Minimum required width of hardware trigger. An overestimate that covers currently unsupported indefinite waits.

wait_delay = 4e-08
Minimum required length of a wait before retrigger can be detected. Corresponds to 4 instructions.

class labscript_devices.PrawnBlaster.labscript_devices.**_PrawnBlasterDummyClockLine**(*name*, *pseudoclock*, *connection*, *ramping_allowed=True*, ***kwargs*)

Bases: labscript.labscript.ClockLine

Dummy Clockline labscript device used internally to allow `WaitMonitor` to work internally to the PrawnBlaster.

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.

- **added_properties** (*dict, optional*) –
- **gui** –
- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(**args*, ***kwargs*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.PrawnBlaster.labscript_devices.**_PrawnBlasterDummyIntermediateDevice**(*name*, *parent_device*, ***kwargs*)
Bases: labscript.labscript.IntermediateDevice

Dummy intermediate labscript device used internally to attach WaitMonitor objects to the PrawnBlaster.

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (ClockLine) – Parent ClockLine device.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(**args*, ***kwargs*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.PrawnBlaster.labscript_devices.**_PrawnBlasterDummyPseudoclock**(*name*, *pseudoclock_device*, *connection*, ***kwargs*)
Bases: labscript.labscript.Pseudoclock

Dummy Pseudoclock labscript device used internally to allow `WaitMonitor` to work internally to the PrawnBlaster.

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (PseudoclockDevice) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to `Device()`.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If `device` is not an allowed child of this device.

generate_code(**args*, ***kwargs*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.PrawnBlaster.labscript_devices.**_PrawnBlasterPseudoclock**(*i*, **args*, ***kwargs*)
Bases: labscript.labscript.Pseudoclock

Customized Clockline for use with the PrawnBlaster.

This Pseudoclock retains information about which hardware clock it is associated with, and ensures only one clockline per pseudoclock.

Parameters **i** (*int*) – Specifies which hardware pseudoclock this device is associated with.

add_device(*device*)

Parameters **device** (ClockLine) – Clockline to attach to the pseudoclock.

class labscript_devices.PrawnBlaster.blacs_tabs.**PrawnBlasterTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

BLACS Tab for the PrawnBlaster Device.

get_child_from_connection_table(*parent_device_name*, *port*)
Finds the attached ClockLines.

Parameters

- **parent_device_name** (*str*) – name of parent_device
- **port** (*str*) – port of parent_device

Returns PrawnBlaster interal Clocklines

Return type ClockLine

initialise_GUI()
Initialises the Tab GUI.

This method is called automatically by BLACS.

initialise_workers()
Initialises the PrawnBlaster Workers.

This method is called automatically by BLACS.

start_run(**args*, ***kwargs*)

status_monitor(**args*, ***kwargs*)

class labscript_devices.PrawnBlaster.blacs_workers.**PrawnBlasterWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

The primary worker for the PrawnBlaster.

This worker handles configuration and communication with the hardware.

abort_buffered()
Aborts a currently running buffered execution.

Returns True is abort is successful.

Return type bool

abort_transition_to_buffered()
Aborts a transition to buffered.

Calls *abort_buffered()*.

check_status()
Checks the operational status of the PrawnBlaster.

This is automatically called by BLACS to update the status of the PrawnBlaster. It also reads the lengths of any accumulated waits during a shot.

Returns

Tuple containing:

- **run_status** (int): Possible values are:
  - 0 : manual-mode
  - 1 : transitioning to buffered execution
  - 2 : buffered execution
  - 3 : abort requested
  - 4 : currently aborting buffered execution
  - 5 : last buffered execution aborted
  - 6 : transitioning to manual mode
- **clock_status** (int): Possible values are:
  - 0 : internal clock
  - 1 : external clock
- **waits_pending** (bool): Indicates if all expected waits have not been read out yet.

Return type (int, int, bool)

init()
Initialises the hardware communication.

This function is automatically called by BLACS and configures hardware communication with the device.

program_manual(*values*)
Manually sets the state of output pins for the pseudoclocks.

Parameters **values** (*dict*) – Dictionary of pseudoclock: value pairs to set.

Returns `values` from arguments on successful programming reflecting current output state.

Return type dict

`read_status()`
Reads the status of the PrawnBlaster.

Returns

Tuple containing

- **run-status** (int): Run status code
- **clock-status** (int): Clock status code

Return type (int, int)

`shutdown()`
Cleanly shuts down the connection to the PrawnBlaster hardware.

`start_run()`
When used as the primary pseudoclock, starts execution in software time to engage the shot.

`transition_to_buffered(`*device_name*, *h5file*, *initial_values*, *fresh*`)`
Configures the PrawnBlaster for buffered execution.

Parameters

- **device_name** (*str*) – labscript name of PrawnBlaster
- **h5file** (*str*) – path to shot file to be run
- **initial_values** (*dict*) – Dictionary of output states at start of shot
- **fresh** (*bool*) – When `True`, clear the local `smart_cache`, forcing a complete reprogramming of the output table.

Returns Dictionary of the expected final output states.

Return type dict

`transition_to_manual()`
Transition the PrawnBlaster back to manual mode from buffered execution at the end of a shot.

Returns `True` if transition to manual is successful.

Return type bool

`wait_for_trigger()`
When used as a secondary pseudoclock, sets the PrawnBlaster to wait for an initial hardware trigger to begin execution.

class labscript_devices.PrawnBlaster.runviewer_parsers.**PrawnBlasterParser**(*path*, *device*)
Bases: object

Runviewer parser for the PrawnBlaster Pseudoclocks.

Parameters

- **path** (*str*) – path to h5 shot file
- **device** (*str*) – labscript name of PrawnBlaster device

get_traces(*add_trace*, *clock=None*)
Reads the shot file and extracts hardware instructions to produce runviewer traces.

Parameters

- **add_trace** (*func*) – function handle that adds traces to runviewer
- **clock** (*tuple, optional*) – clock times from timing device, if not the primary pseudoclock

Returns Dictionary of clocklines and triggers derived from instructions

Return type dict

2.1.6 RFblaster

Another pseudoclock-cable labscript device.

Detailed Documentation

class labscript_devices.RFBlaster.**MultiPartForm**
Bases: object

Accumulate the data to be used when posting a form.

add_field(*name*, *value*)
Add a simple field to the form data.

add_file_content(*fieldname*, *filename*, *body*, *mimetype=None*)

get_content_type()

class labscript_devices.RFBlaster.**RFBlaster**(*name*, *ip_address*, *trigger_device=None*, *trigger_connection=None*)
Bases: labscript.labscript.PseudoclockDevice

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript_devices.RFBlaster.RFBlasterPseudoclock'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 500000.0

clock_resolution = 1.3333333333333334e-08

description = 'RF Blaster Rev1.1'
Brief description of the device.

property direct_outputs

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

property pseudoclock

trigger_delay = 0.00087375

wait_day = 0.00087375

class labscript_devices.RFBlaster.**RFBlasterDirectOutputs**(*name*, *parent_device*, ***kwargs*)
Bases: labscript.labscript.IntermediateDevice

Provides some error checking to ensure parent_device is a `ClockLine`.

Calls `Device.__init__()`.

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (`ClockLine`) – Parent ClockLine device.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (`Device`) – Device to add.

Raises **LabscriptError** – If `device` is not an allowed child of this device.

allowed_children = [<class 'labscript.labscript.DDS'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 500000.0

description = 'RFBlaster Direct Outputs'
Brief description of the device.

class labscript_devices.RFBlaster.**RFBlasterPseudoclock**(*name*, *pseudoclock_device*, *connection*, ***kwargs*)
Bases: `labscript.labscript.Pseudoclock`

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (`PseudoclockDevice`) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to `Device()`.

add_device(*device*)
Adds a child device to this device.

Parameters **device** (`Device`) – Device to add.

Raises **LabscriptError** – If `device` is not an allowed child of this device.

class labscript_devices.RFBlaster.**RFBlasterTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

get_child_from_connection_table(*parent_device_name*, *port*)

initialise_GUI()

labscript_device_class_name = 'RFBlaster'

transition_to_manual(**args*, ***kwargs*)

class labscript_devices.RFBlaster.**RFBlasterWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

check_remote_values()

get_web_values(*page*)

http_request(*form=None*)
Make a HTTP request to the RFBlaster, optionally submitting a form

init()

program_manual(*values*)

restart_kloned(*respawn_netcat=True*)

shutdown()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

2.2 NI DAQS

The NI_DAQmx device provides a generic interface for National Instruments data acquisition hardware. This includes digital and analog voltage I/O. These input/outputs can be either static or hardware-timed dynamically changing variables.

2.2.1 NI DAQs

Overview

This labscript device is a master device that can control a wide range of NI Multi-function data acquistion devices.

Installation

This labscript device requires an installation of the NI-DAQmx module, available for free from NI.

The python bindings are provided by the PyDAQmx package, available through pip.

Adding a Device

While the `NI_DAQmx` device can be used directly by manually specifying the many necessary parameters, it is preferable to add the device via an appropriate subclass. This process is greatly simplified by using the *get_capabilities.py* script followed by the *generate_subclasses.py* script.

To add support for a DAQmx device that is not yet supported, run `get_capabilities.py` on a computer with the device in question connected (or with a simulated device of the correct model configured in NI-MAX). This will introspect the capabilities of the device and add those details to `capabilities.json`. To generate labscript device classes for all devices whose capabilities are known, run `generate_subclasses.py`. Subclasses of NI_DAQmx will be made in the `models` subfolder, and they can then be imported into labscript code with:

```
from labscript_devices.NI_DAQmx.labscript_devices import NI_PCIe_6363
```

or similar. The class naming is based on the model name by prepending "NI_" and replacing the hyphen with an underscore, i.e. 'PCIe-6363' -> NI_PCIe_6363.

Generating device classes requires the Python code-formatting library 'black', which can be installed via pip (Python 3.6+ only). If you don't want to install this library, the generation code will still work, it just won't be formatted well.

The current list of pre-subclassed devices is:

Sub-Classed NI DAQ Models

labscript_devices.NI_DAQmx.models.NI_PCI_6251(...)	Class for NI-PCI-6251
labscript_devices.NI_DAQmx.models.NI_PCI_6534(...)	Class for NI-PCI-6534
labscript_devices.NI_DAQmx.models.NI_PCI_6713(...)	Class for NI-PCI-6713
labscript_devices.NI_DAQmx.models.NI_PCI_6733(...)	Class for NI-PCI-6733

continues on next page

Table 3 – continued from previous page

labscript_devices.NI_DAQmx.models.NI_PCI_DIO_32HS(...)	Class for NI-PCI-DIO-32HS
labscript_devices.NI_DAQmx.models.NI_PCIe_6343(...)	Class for NI-PCIe-6343
labscript_devices.NI_DAQmx.models.NI_PCIe_6363(...)	Class for NI-PCIe-6363
labscript_devices.NI_DAQmx.models.NI_PCIe_6738(...)	Class for NI-PCIe-6738
labscript_devices.NI_DAQmx.models.NI_PXI_6733(...)	Class for NI-PXI-6733
labscript_devices.NI_DAQmx.models.NI_PXIe_4499(...)	Class for NI-PXIe-4499
labscript_devices.NI_DAQmx.models.NI_PXIe_6361(...)	Class for NI-PXIe-6361
labscript_devices.NI_DAQmx.models.NI_PXIe_6363(...)	Class for NI-PXIe-6363
labscript_devices.NI_DAQmx.models.NI_PXIe_6535(...)	Class for NI-PXIe-6535
labscript_devices.NI_DAQmx.models.NI_PXIe_6738(...)	Class for NI-PXIe-6738
labscript_devices.NI_DAQmx.models.NI_USB_6008(...)	Class for NI-USB-6008
labscript_devices.NI_DAQmx.models.NI_USB_6229(...)	Class for NI-USB-6229
labscript_devices.NI_DAQmx.models.NI_USB_6343(...)	Class for NI-USB-6343
labscript_devices.NI_DAQmx.models.NI_USB_6363(...)	Class for NI-USB-6363
labscript_devices.NI_DAQmx.models.NI_USB_6366(...)	Class for NI-USB-6366
labscript_devices.NI_DAQmx.models.generate_subclasses	Reads the capabilities file and generates labscript devices for each known model of DAQ.
labscript_devices.NI_DAQmx.models.get_capabilities	This is a script to update `model_capabilities.json` with the capabilities of all NI-DAQmx devices currently connected to this computer.

labscript_devices.NI_DAQmx.models.NI_PCI_6251

labscript_devices.NI_DAQmx.models.NI_PCI_6251.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 2.5e-07, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 1000000.0, 'max_AI_single_chan_rate': 1250000.0, 'max_AO_sample_rate': 2857142.8571428573, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 16, 'num_AO': 2, 'num_CI': 2, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_PCI_6251.**NI_PCI_6251**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCI-6251

description = 'NI-PCI-6251'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCI_6534

labscript_devices.NI_DAQmx.models.NI_PCI_6534.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': None, 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': None, 'max_DO_sample_rate': 20000000.0, 'min_semiperiod_measurement': None, 'num_AI': 0, 'num_AO': 0, 'num_CI': 0, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': True}, 'port2': {'num_lines': 8, 'supports_buffered': True}, 'port3': {'num_lines': 8, 'supports_buffered': True}, 'port4': {'num_lines': 0, 'supports_buffered': False}, 'port5': {'num_lines': 4, 'supports_buffered': False}}, 'supports_buffered_AO': False, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': False}**

class labscript_devices.NI_DAQmx.models.NI_PCI_6534.**NI_PCI_6534**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCI-6534

description = 'NI-PCI-6534'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCI_6713

labscript_devices.NI_DAQmx.models.NI_PCI_6713.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': 1000000.0, 'max_DO_sample_rate': None, 'min_semiperiod_measurement': 2e-05, 'num_AI': 0, 'num_AO': 8, 'num_CI': 2, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': False, 'supports_semiperiod_measurement': True}**

class labscript_devices.NI_DAQmx.models.NI_PCI_6713.**NI_PCI_6713**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCI-6713

description = 'NI-PCI-6713'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCI_6733

labscript_devices.NI_DAQmx.models.NI_PCI_6733.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': 1000000.0, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 2e-05, 'num_AI': 0, 'num_AO': 8, 'num_CI': 2, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True}**

class labscript_devices.NI_DAQmx.models.NI_PCI_6733.**NI_PCI_6733**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCI-6733

description = 'NI-PCI-6733'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCI_DIO_32HS

labscript_devices.NI_DAQmx.models.NI_PCI_DIO_32HS.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': None, 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': None, 'max_DO_sample_rate': 20000000.0, 'min_semiperiod_measurement': None, 'num_AI': 0, 'num_AO': 0, 'num_CI': 0, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': True}, 'port2': {'num_lines': 8, 'supports_buffered': True}, 'port3': {'num_lines': 8, 'supports_buffered': True}, 'port4': {'num_lines': 0, 'supports_buffered': False}, 'port5': {'num_lines': 4, 'supports_buffered': False}}, 'supports_buffered_AO': False, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': False}**

class labscript_devices.NI_DAQmx.models.NI_PCI_DIO_32HS.**NI_PCI_DIO_32HS**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCI-DIO-32HS

description = 'NI-PCI-DIO-32HS'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCIe_6343

labscript_devices.NI_DAQmx.models.NI_PCIe_6343.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 500000.0, 'max_AI_single_chan_rate': 500000.0, 'max_AO_sample_rate': 917431.1926605505, 'max_DO_sample_rate': 1000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 4, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_PCIe_6343.**NI_PCIe_6343**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCIe-6343

description = 'NI-PCIe-6343'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCIe_6363

labscript_devices.NI_DAQmx.models.NI_PCIe_6363.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 1000000.0, 'max_AI_single_chan_rate': 2000000.0, 'max_AO_sample_rate': 2857142.8571428573, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 4, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_PCIe_6363.**NI_PCIe_6363**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCIe-6363

description = 'NI-PCIe-6363'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PCIe_6738

labscript_devices.NI_DAQmx.models.NI_PCIe_6738.**CAPABILITIES** = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': 1000000.0, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 0, 'num_AO': 32, 'num_CI': 4, 'ports': {'port0': {'num_lines': 2, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True}

class labscript_devices.NI_DAQmx.models.NI_PCIe_6738.**NI_PCIe_6738**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PCIe-6738

description = 'NI-PCIe-6738'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXI_6733

labscript_devices.NI_DAQmx.models.NI_PXI_6733.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': 1000000.0, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 2e-05, 'num_AI': 0, 'num_AO': 8, 'num_CI': 2, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True}**

class labscript_devices.NI_DAQmx.models.NI_PXI_6733.**NI_PXI_6733**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXI-6733

description = 'NI-PXI-6733'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXIe_4499

labscript_devices.NI_DAQmx.models.NI_PXIe_4499.**CAPABILITIES = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': None, 'AI_start_delay_ticks': 64, 'AI_term': 'PseudoDiff', 'AI_term_cfg': {'ai0': ['PseudoDiff'], 'ai1': ['PseudoDiff'], 'ai10': ['PseudoDiff'], 'ai11': ['PseudoDiff'], 'ai12': ['PseudoDiff'], 'ai13': ['PseudoDiff'], 'ai14': ['PseudoDiff'], 'ai15': ['PseudoDiff'], 'ai2': ['PseudoDiff'], 'ai3': ['PseudoDiff'], 'ai4': ['PseudoDiff'], 'ai5': ['PseudoDiff'], 'ai6': ['PseudoDiff'], 'ai7': ['PseudoDiff'], 'ai8': ['PseudoDiff'], 'ai9': ['PseudoDiff']}, 'AO_range': None, 'max_AI_multi_chan_rate': 204800.0, 'max_AI_single_chan_rate': 204800.0, 'max_AO_sample_rate': None, 'max_DO_sample_rate': None, 'min_semiperiod_measurement': None, 'num_AI': 16, 'num_AO': 0, 'num_CI': 0, 'ports': {}, 'supports_buffered_AO': False, 'supports_buffered_DO': False, 'supports_semiperiod_measurement': False, 'supports_simultaneous_AI_sampling': True}**

class labscript_devices.NI_DAQmx.models.NI_PXIe_4499.**NI_PXIe_4499**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXIe-4499

description = 'NI-PXIe-4499'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXIe_6361

labscript_devices.NI_DAQmx.models.NI_PXIe_6361.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 1000000.0, 'max_AI_single_chan_rate': 2000000.0, 'max_AO_sample_rate': 2857142.8571428573, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 16, 'num_AO': 2, 'num_CI': 4, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_PXIe_6361.**NI_PXIe_6361**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXIe-6361

description = 'NI-PXIe-6361'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXIe_6363

labscript_devices.NI_DAQmx.models.NI_PXIe_6363.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 1000000.0, 'max_AI_single_chan_rate': 2000000.0, 'max_AO_sample_rate': 2857142.8571428573, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 4, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_PXIe_6363.**NI_PXIe_6363**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXIe-6363

description = 'NI-PXIe-6363'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXIe_6535

labscript_devices.NI_DAQmx.models.NI_PXIe_6535.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': None, 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': None, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': None, 'num_AI': 0, 'num_AO': 0, 'num_CI': 0, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': True}, 'port2': {'num_lines': 8, 'supports_buffered': True}, 'port3': {'num_lines': 8, 'supports_buffered': True}, 'port4': {'num_lines': 6, 'supports_buffered': False}}, 'supports_buffered_AO': False, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': False}**

class labscript_devices.NI_DAQmx.models.NI_PXIe_6535.**NI_PXIe_6535**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXIe-6535

description = 'NI-PXIe-6535'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_PXIe_6738

labscript_devices.NI_DAQmx.models.NI_PXIe_6738.**CAPABILITIES = {'AI_range': None, 'AI_range_Diff': None, 'AI_start_delay': None, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': None, 'max_AI_single_chan_rate': None, 'max_AO_sample_rate': 1000000.0, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 0, 'num_AO': 32, 'num_CI': 4, 'ports': {'port0': {'num_lines': 2, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True}**

class labscript_devices.NI_DAQmx.models.NI_PXIe_6738.**NI_PXIe_6738**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-PXIe-6738

description = 'NI-PXIe-6738'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_USB_6008

labscript_devices.NI_DAQmx.models.NI_USB_6008.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-20.0, 20.0], 'AI_start_delay': 8.333333333333334e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'Diff'], 'ai1': ['RSE', 'Diff'], 'ai2': ['RSE', 'Diff'], 'ai3': ['RSE', 'Diff'], 'ai4': ['RSE'], 'ai5': ['RSE'], 'ai6': ['RSE'], 'ai7': ['RSE']}, 'AO_range': [0.0, 5.0], 'max_AI_multi_chan_rate': 10000.0, 'max_AI_single_chan_rate': 10000.0, 'max_AO_sample_rate': None, 'max_DO_sample_rate': None, 'min_semiperiod_measurement': None, 'num_AI': 8, 'num_AO': 2, 'num_CI': 1, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': False}, 'port1': {'num_lines': 4, 'supports_buffered': False}}, 'supports_buffered_AO': False, 'supports_buffered_DO': False, 'supports_semiperiod_measurement': False, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_USB_6008.**NI_USB_6008**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-USB-6008

description = 'NI-USB-6008'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_USB_6229

labscript_devices.NI_DAQmx.models.NI_USB_6229.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 2.5e-07, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 250000.0, 'max_AI_single_chan_rate': 250000.0, 'max_AO_sample_rate': 833333.3333333334, 'max_DO_sample_rate': 1000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 2, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_USB_6229.**NI_USB_6229**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-USB-6229

description = 'NI-USB-6229'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_USB_6343

labscript_devices.NI_DAQmx.models.NI_USB_6343.**CAPABILITIES = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 500000.0, 'max_AI_single_chan_rate': 500000.0, 'max_AO_sample_rate': 917431.1926605505, 'max_DO_sample_rate': 1000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 4, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}**

class labscript_devices.NI_DAQmx.models.NI_USB_6343.**NI_USB_6343**(**args*, ***kwargs*)
Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-USB-6343

description = 'NI-USB-6343'
Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_USB_6363

labscript_devices.NI_DAQmx.models.NI_USB_6363.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 7e-08, 'AI_term': 'RSE', 'AI_term_cfg': {'ai0': ['RSE', 'NRSE', 'Diff'], 'ai1': ['RSE', 'NRSE', 'Diff'], 'ai10': ['RSE', 'NRSE'], 'ai11': ['RSE', 'NRSE'], 'ai12': ['RSE', 'NRSE'], 'ai13': ['RSE', 'NRSE'], 'ai14': ['RSE', 'NRSE'], 'ai15': ['RSE', 'NRSE'], 'ai16': ['RSE', 'NRSE', 'Diff'], 'ai17': ['RSE', 'NRSE', 'Diff'], 'ai18': ['RSE', 'NRSE', 'Diff'], 'ai19': ['RSE', 'NRSE', 'Diff'], 'ai2': ['RSE', 'NRSE', 'Diff'], 'ai20': ['RSE', 'NRSE', 'Diff'], 'ai21': ['RSE', 'NRSE', 'Diff'], 'ai22': ['RSE', 'NRSE', 'Diff'], 'ai23': ['RSE', 'NRSE', 'Diff'], 'ai24': ['RSE', 'NRSE'], 'ai25': ['RSE', 'NRSE'], 'ai26': ['RSE', 'NRSE'], 'ai27': ['RSE', 'NRSE'], 'ai28': ['RSE', 'NRSE'], 'ai29': ['RSE', 'NRSE'], 'ai3': ['RSE', 'NRSE', 'Diff'], 'ai30': ['RSE', 'NRSE'], 'ai31': ['RSE', 'NRSE'], 'ai4': ['RSE', 'NRSE', 'Diff'], 'ai5': ['RSE', 'NRSE', 'Diff'], 'ai6': ['RSE', 'NRSE', 'Diff'], 'ai7': ['RSE', 'NRSE', 'Diff'], 'ai8': ['RSE', 'NRSE'], 'ai9': ['RSE', 'NRSE']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 1000000.0, 'max_AI_single_chan_rate': 2000000.0, 'max_AO_sample_rate': 2857142.8571428573, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 32, 'num_AO': 4, 'num_CI': 4, 'ports': {'port0': {'num_lines': 32, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': False}

class labscript_devices.NI_DAQmx.models.NI_USB_6363.**NI_USB_6363**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-USB-6363

description = 'NI-USB-6363'

Brief description of the device.

labscript_devices.NI_DAQmx.models.NI_USB_6366

labscript_devices.NI_DAQmx.models.NI_USB_6366.**CAPABILITIES** = {'AI_range': [-10.0, 10.0], 'AI_range_Diff': [-10.0, 10.0], 'AI_start_delay': 4e-08, 'AI_term': 'Diff', 'AI_term_cfg': {'ai0': ['Diff'], 'ai1': ['Diff'], 'ai2': ['Diff'], 'ai3': ['Diff'], 'ai4': ['Diff'], 'ai5': ['Diff'], 'ai6': ['Diff'], 'ai7': ['Diff']}, 'AO_range': [-10.0, 10.0], 'max_AI_multi_chan_rate': 2000000.0, 'max_AI_single_chan_rate': 2000000.0, 'max_AO_sample_rate': 3333333.3333333335, 'max_DO_sample_rate': 10000000.0, 'min_semiperiod_measurement': 1e-07, 'num_AI': 8, 'num_AO': 2, 'num_CI': 4, 'ports': {'port0': {'num_lines': 8, 'supports_buffered': True}, 'port1': {'num_lines': 8, 'supports_buffered': False}, 'port2': {'num_lines': 8, 'supports_buffered': False}}, 'supports_buffered_AO': True, 'supports_buffered_DO': True, 'supports_semiperiod_measurement': True, 'supports_simultaneous_AI_sampling': True}

class labscript_devices.NI_DAQmx.models.NI_USB_6366.**NI_USB_6366**(**args*, ***kwargs*)

Bases: *labscript_devices.NI_DAQmx.labscript_devices.NI_DAQmx*

Class for NI-USB-6366

description = 'NI-USB-6366'
Brief description of the device.

labscript_devices.NI_DAQmx.models.generate_subclasses

Reads the capabilities file and generates labscript devices for each known model of DAQ.

Called from the command line via

```
python generate_subclasses.py
```

For proper formatting of the code, you will need to install the `black` python package.

labscript_devices.NI_DAQmx.models.generate_subclasses.**main**()
Called when the script is run.

Will attempt to reformat the generated files using *reformat_files()*.

labscript_devices.NI_DAQmx.models.generate_subclasses.**reformat_files**(*filepaths*)
Apply black formatter to a list of source files.

Parameters **filepaths** (*list*) – List of python source files to format.

labscript_devices.NI_DAQmx.models.get_capabilities

This is a script to update `model_capabilities.json` with the capabilities of all NI-DAQmx devices currently connected to this computer.

Run this script to add support for a new model of NI-DAQmx device. Note that this will work with a simulated device configured through NI-MAX as well, so support can be added without actually having the physical device.

Called from the command line via

```
python get_capabilities.py
```

labscript_devices.NI_DAQmx.models.get_capabilities.**AI_filter_delay**(*device_name*)
Determine the filter delay for dynamic signal acquistion devices.

Returns the delay in clock cycles. Absolute delay will vary with sample rate.

Parameters **device_name** (*str*) – NI-MAX device name

Returns Number of analog input delays ticks between task start and acquisition start.

Return type int

labscript_devices.NI_DAQmx.models.get_capabilities.**AI_start_delay**(*device_name*)
Empirically determines the analog inputs' start delay.

Parameters **device_name** (*str*) – NI-MAX device name

Returns Analog input start delay in seconds. None if analog inputs not supported.

Return type float

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAIMaxMultiChanRate**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAIMaxSingleChanRate**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAIPhysicalChans**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAISimultaneousSamplingSupported**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAIVoltageRngs**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAOMaxRate**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAOPhysicalChans**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAOSampClkSupported**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAOVoltageRngs**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevAnlgTrigSupported**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevCIPhysicalChans**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDILines**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDIPorts**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDOLines**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDOMaxRate**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDOPorts**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevDigTrigSupported**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevProductType**(*name=None*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetDevTerminals**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetPhysicalChanAITermCfgs**(*name*)

labscript_devices.NI_DAQmx.models.get_capabilities.**DAQmxGetSysDevNames**(*name=None*)

labscript_devices.NI_DAQmx.models.get_capabilities.**bool_prop**(*func*)
Bool property wrapper.

Parameters **func** (*function*) – PyDAQmx library function that returns a boolean.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**chans**(*func*)
string_prop but splitting the return value into separate channels and stripping the device name from them

Parameters **func** (*function*) – PyDAQmx library function that returns channel string.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**float64_array_prop**(*func*)
Array of floats property wrapper.

Parameters **func** (*function*) – PyDAQmx library function that returns an array of float64s.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**float64_prop**(*func*)
Float property wrapper.

Parameters **func** (*function*) – PyDAQmx library function that returns a float64.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**get_min_semiperiod_measurement**(*device_name*)
Determines the minimum semi-period measurement time supported by the device.

Depending on the timebase used, counter inputs can measure time intervals of various ranges. As a default, we pick a largish range - the one with the fastest timebase still capable of measuring 100 seconds, or the largest time interval if it is less than 100 seconds, and we save the smallest interval measurable with this timebase. Then labscript can ensure it doesn't make wait monitor pulses shorter than this. This should be a sensible default behaviour, though if the user has experiments considerably shorter or longer than 100 seconds, such that they want to use a different timebase, they may pass the min_semiperiod_measurement keyword argument into the DAQmx class, to tell labscript to make pulses some other duration compatible with the maximum wait time in their experiment.

However, since there are software delays in timeouts of waits during a shot, any timed-out waits necessarily will last software timescales of up to ~100ms on a slow computer, preventing one from using very fast timebases with low-resolution counters if there is any possibility of timing out. For now (in the wait monitor worker class) we pessimistically add one second to the expected longest measurement to account for software delays. These decisions can be revisited if there is a need, do not hesitate to file an issue on bitbucket regarding this if it affects you.

Parameters **device_name** (*str*) – NI-MAX device name

Returns Minimum measurement time.

Return type float

labscript_devices.NI_DAQmx.models.get_capabilities.**int32_prop**(*func*)
Int32 property wrapper.

Parameters **func** (*function*) – PyDAQmx library function that returns a int32.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**port_supports_buffered**(*device_name*, *port*, *clock_terminal=None*)
Empirically determines if the digital port supports buffered output.

Parameters

- **device_name** (*str*) – NI-MAX device name
- **port** (*int*) – Which port to intro-spect
- **clock_terminal** (*str, optional*) – String that specifies the clock terminal.

Returns True if `port` supports buffered output.

Return type bool

labscript_devices.NI_DAQmx.models.get_capabilities.**string_prop**(*func*)
String property wrapper.

Parameters **func** (*function*) – PyDAQmx library function that returns a string.

Returns The wrapped function.

Return type function

labscript_devices.NI_DAQmx.models.get_capabilities.**supported_AI_ranges_for_non_differential_input**(*device_name*, *AI_ranges*)
Empirically determine the analog input voltage ranges for non-differential inputs.

Tries AI ranges to see which are actually allowed for non-differential input, since the largest range may only be available for differential input.

Parameters

- **device_name** (*str*) – NI-MAX device name
- **AI_ranges** (*list*) – list of `[Vmin, Vmax]` pairs to check compatibility.

Returns List of lists with the supported voltage ranges.

Return type list

labscript_devices.NI_DAQmx.models.get_capabilities.**supported_AI_terminal_configurations**(*device_name*)

Determine which analong input configurations are supported for each AI.

Valid options are RSE, NRSE, Diff, and PseudoDiff.

Parameters **device_name** (*str*) – NI-MAX device name

Returns Dictionary of analog input channels where each value is a list of the supported input terminations.

Return type dict

labscript_devices.NI_DAQmx.models.get_capabilities.**supports_semiperiod_measurement**(*device_name*)

Empirically determines if the DAQ supports semiperiod measurement.

Parameters **device_name** (*str*) – NI-MAX device name.

Returns True if semi-period measurements are supported by the device.

Return type bool

Usage

NI Multifunction DAQs generally provide hardware channels for `StaticAnalogOut`, `StaticDigitalOut`, `AnalogOut`, `DigitalOut`, and `AnalogIn` labscript quantities for use in experiments. Exact numbers of channels, performance, and configuration depend on the model of DAQ used.

```
from labscript import *

from labscript_devices.DummyPseudoclock.labscript_devices import DummyPseudoclock
from labscript_devices.NI_DAQmx.models.NI_USB_6343 import NI_USB_6343

DummyPseudoclock('dummy_clock',BLACS_connection='dummy')

NI_USB_6343(name='daq',parent_device=dummy_clock.clockline,
                       MAX_name='ni_usb_6343',
```

(continues on next page)

(continued from previous page)

```
                        clock_terminal='/ni_usb_6343/PFI0',
                        acquisition_rate=100e3)

AnalogIn('daq_ai0',daq,'ai0')
AnalogIn('daq_ai1',daq,'ai1')

AnalogOut('daq_ao0',daq,'ao0')
AnalogIn('daq_ai1',daq,'ai1')
```

WaitMonitors

NI DAQs are also used within labscript to provide a `WaitMonitor`. When configured, the `WaitMonitor` allows for arbitrary-length pauses in experiment execution, waiting for some trigger to restart. The monitor provides a measurement of the duration of the wait for use in interpreting the resulting data from the experiment.

Configuration uses three digital I/O connections on the DAQ:

- The parent_connection which sends pulses at the beginning of the experiment, the start of the wait, and the end of the wait.
- The acquisition_connection which must be wired to a counter and measures the time between the pulses of the parent connection.
- The timeout_connection which can send a restart pulse if the wait times out.

An example configuration of a `WaitMonitor` using a NI DAQ is shown here

```
# A wait monitor for AC-line triggering
# This requires custom hardware
WaitMonitor(name='wait_monitor',parent_device=daq,connection='port0/line0',
                        acquisition_device=daq, acquisition_connection='ctr0',
                        timeout_device=daq, timeout_connection='PFI1')
# Necessary to ensure even number of digital out lines in shot
DigitalOut('daq_do1',daq,'port0/line1')
```

Note that the counter connection is specified using the logical label `'ctr0'`. On many NI DAQs, the physical connection to this counter is PFI9. The physical wiring for this configuration would have port0/line0 wired directly to PFI9, with PFI1 being sent to the master pseudoclock retriggering system in case of timeout. If timeouts are not expected/represent experiment failure, this physical connection can be omitted.

In addition to their external ports, some types of NI DAQ modules (PXI, PXIe, CompactDAQ) feature internal ports, known as “terminals” in NI terminology. Terminals include most clocks and triggers in a module, as well as the external PFIN connections. The buffered and static digital IO connections are not terminals. Connections between terminals can be used for sharing clocks or triggers between modules in the same chassis (note: if sufficient clocklines and external inputs are

available, it is likely preferable to simply use a unique clockline for each card). Within labscript, there are two methods for accessing this functionality. For sharing the clock input signal to other cards, the `clock_mirror_terminal` argument in the constructor can be specified. For example, in a system with two PXI-6733 analog cards in a PXI chassis (which supports 8 internal triggers, named `PXI_TrigN`), the connection table entries are

```
NI_PXI_6733(name='dev_1',
                        ...,
                        clock_terminal='/Dev1/PFI0',
                        clock_mirror_terminal='/Dev1/PXI_Trig0',
                        MAX_name='Dev1')

NI_PXI_6733(name='dev_2',
                        ...,
                        clock_terminal='/Dev2/PXI_Trig0',
                        MAX_name='Dev2')
```

However, some NI DAQ modules can not be clocked from certain terminal. To determine this, consult the `Device Routes` tab in NI MAX. If there is not a `Direct Route` or `Indirect Route` between the clock source and clock destination, the best option is to choose a different `clock_mirror_terminal` if possible. For some combinations of modules, there will be no pair of triggers linked to all the cards. To handle this situation, two triggers can be linked using the `connected_terminals` argument. This argument takes a list of tuples of terminal names, and connects the first terminal to the second terminal. For example, to share the clock in the previous with an additional PXIe-6535 digital card (which can not use `PXI_Trig0` as a clock), the connection table entries are

```
NI_PXI_6733(name='dev_1',
                        ...,
                        clock_terminal='/Dev1/PFI0',
                        clock_mirror_terminal='/Dev1/PXI_Trig0',
                        MAX_name='Dev1')

NI_PXI_6733(name='dev_2',
                        ...,
                        clock_terminal='/Dev2/PXI_Trig0',
                        MAX_name='Dev2')

NI_PXIe_6535(name='dev_3',
                        ...,
                        clock_terminal='/Dev3/PXI_Trig7',
                        MAX_name='Dev3',
                        connected_terminals=[('/Dev3/PXI_Trig0', '/Dev3/PXI_Trig7')])
```

In addition to clocking, the `connected_terminals` argument can be used to link output terminals on an NI DAQ module to shared triggers, then link those shared

triggers to input terminals of another NI DAQ module in the same chassis.

AI timing skew

Given how the NI-DAQmx driver currently works, all of the outputs (and generally other hardware) are hardware-timed via direct outputs from the parent pseudoclocks. Under default usage, this is not true for the analog inputs of the DAQs, which are timed via the internal reference oscillator of the DAQ. Synchronization between the two is handled at the end by correlating start times and slicing the AI traces at the appropriate times. This works fine if the reference clocks for the pseudoclock and the DAQ don't drift relative to each other, but that is generally not the case for a longer shot (on the order of 1 second) since the standard clocks for a pulseblaster and a DAQ both have accuracy on the order of 50 ppm.

With version 1.2.0 of the NI-DAQmx driver, this issue can be mitigated by suppling an external sample timebase that is phase synchronous with the DAQ's pseudoclock device. This is done using the DAQmx `SampleClkTimebase` synchronization method. Simply provide an external clocking signal that is faster than the analog input sampling rate, and the DAQ will use an internall PLL to derive the AI sample clock from the provided timebase. Specifying an externally provided sample timebase is done using the `AI_timebase_terminal` and `AI_timebase_rate` arguments, which specify the input terminal (generally a PFI line) and the clock frequency.

Detailed Documentation

class labscript_devices.NI_DAQmx.labscript_devices.**NI_DAQmx**(*name*, *parent_device=None*, *clock_terminal=None*, *MAX_name=None*, *static_AO=None*, *static_DO=None*, *clock_mirror_terminal=None*, *connected_terminals=None*, *acquisition_rate=None*, *AI_range=None*, *AI_range_Diff=None*, *AI_start_delay=0*, *AI_start_delay_ticks=None*, *AI_term='RSE'*, *AI_term_cfg=None*, *AI_timebase_terminal=None*, *AI_timebase_rate=None*, *AO_range=None*, *max_AI_multi_chan_rate=None*, *max_AI_single_chan_rate=None*, *max_AO_sample_rate=None*, *max_DO_sample_rate=None*, *min_semiperiod_measurement=None*, *num_AI=0*, *num_AO=0*, *num_CI=0*, *ports=None*, *supports_buffered_AO=False*, *supports_buffered_DO=False*, *supports_semiperiod_measurement=False*, *supports_simultaneous_AI_sampling=False*, ***kwargs*)

Bases: labscript.labscript.IntermediateDevice

Generic class for NI_DAQmx devices.

Generally over-ridden by device-specific subclasses that contain the introspected default values.

Parameters

- **name** (*str*) – name to assign to the created labscript device
- **parent_device** (*clockline*) – Parent clockline device that will clock the outputs of this device

- **clock_terminal** (*str*) – What input on the DAQ is used for the clockline
- **MAX_name** (*str*) – NI-MAX device name
- **static_AO** (*int, optional*) – Number of static analog output channels.
- **static_DO** (*int, optional*) – Number of static digital output channels.
- **clock_mirror_terminal** (*str, optional*) – Channel string of digital output that mirrors the input clock. Useful for daisy-chaning DAQs on the same clockline.
- **connected_terminals** (*list, optional*) – List of pairs of strings of digital inputs and digital outputs that will be connected. Useful for daisy-chaining DAQs on the same clockline when they do not have direct routes (see Device Routes in NI MAX).
- **acquisiton_rate** (*float, optional*) – Default sample rate of inputs.
- **AI_range** (*iterable, optional*) – A `[Vmin, Vmax]` pair that sets the analog input voltage range for all analog inputs.
- **AI_range_Diff** (*iterable, optional*) – A `[Vmin, Vmax]` pair that sets the analog input voltage range for all analog inputs when using Differential termination.
- **AI_start_delay** (*float, optional*) – Time in seconds between start of an analog input task starting and the first sample.
- **AI_start_delay_ticks** (*int, optional*) – Time in sample clock periods between start of an analog input task starting and the first sample. To use this method, `AI_start_delay` must be set to `None`. This is necessary for DAQs that employ delta ADCs.
- **AI_term** (*str, optional*) – Configures the analog input termination for all analog inputs. Must be supported by the device. Supported options are `'RSE'`, `'NRSE'` `'Diff'`, and ‘`PseudoDiff`'.
- **AI_term_cfg** (*dict, optional*) – Dictionary of analog input channels and their supported terminations. Best to use `get_capabilities.py` to introspect these.
- **AI_timebase_terminal** (*str, optional*) – Channel string that specifies what channel to use for the Sample Clock Timebase signal. If None, use default internal clocks. Must also specify the rate when not using the internal sources.
- **AI_timebase_rate** (*float, optional*) – Supplied clock frequency for the AI timebase. Only specify if using an external clock source.
- **AO_range** (*iterable, optional*) – A `[Vmin, Vmax]` pair that sets the analog output voltage range for all analog outputs.
- **max_AI_multi_chan_rate** (*float, optional*) – Max supported analog input sampling rate when using multiple channels.
- **max_AI_single_chan_rate** (*float, optional*) – Max supported analog input sampling rate when only using a single channel.
- **max_AO_sample_rate** (*float, optional*) – Max supported analog output sample rate.
- **max_DO_sample_rate** (*float, optional*) – Max supported digital output sample rate.
- **min_sermiperiod_measurement** (*float, optional*) – Minimum measurable time for a semiperiod measurement.

- **num_AI** (*int, optional*) – Number of analog inputs channels.
- **num_AO** (*int, optional*) – Number of analog output channels.
- **num_CI** (*int, optional*) – Number of counter input channels.
- **ports** (*dict, optional*) – Dictionarly of DIO ports, which number of lines and whether port supports buffered output.
- **supports_buffered_AO** (*bool, optional*) – True if analog outputs support buffered output
- **supports_buffered_DO** (*bool, optional*) – True if digital outputs support buffered output
- **supports_semiperiod_measurement** (*bool, optional*) – True if device supports semi-period measurements

_check_AI_not_too_fast(*AI_table*)
Check that analog input acquisition rates do not exceed maximums.

_check_bounds(*analogs*)
Check that all analog outputs are in bounds

_check_even_children(*analogs*, *digitals*)
Check that there are an even number of children of each type.

_check_wait_monitor_timeout_device_config()
Check that if we are the wait monitor acquisition device and another device is the wait monitor timeout device, that a) the other device is a DAQmx device and b) the other device has a start_order lower than us and a stop_order higher than us.

_make_analog_input_table(*inputs*)
Collect analog input instructions and create the acquisition table

_make_analog_out_table(*analogs*, *times*)
Collect analog output data and create the output array

_make_digital_out_table(*digitals*, *times*)
Collect digital output data and create the output array

add_device(*device*)
Error checking for adding a child device.

Parameters **device** (*labscript device*) – Child labscript device to attach to this device. Only types of devices in *allowed_children* can be attached.

allowed_children = []
Sets the allowed children types based on the capabilites.

description = 'NI-DAQmx'
Brief description of the device.

generate_code(*hdf5_file*)
Generates the hardware code from the script and saves it to the shot h5 file.

This is called automatically when a shot is compiled.

Parameters **hdf5_file** (*str*) – Path to shot's hdf5 file to save the instructions to.

wait_monitor_supports_wait_completed_events = True

labscript_devices.NI_DAQmx.labscript_devices.**_smallest_int_type**(*n*)
Return the smallest unsigned integer type sufficient to contain n bits

class labscript_devices.NI_DAQmx.blacs_tabs.**NI_DAQmxTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

class labscript_devices.NI_DAQmx.blacs_workers.**NI_DAQmxAcquisitionWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

MAX_READ_INTERVAL = 0.2

MAX_READ_PTS = 10000

abort_buffered()

abort_transition_to_buffered()

extract_measurements(*raw_data*, *waits_in_use*)

init()

program_manual(*values*)

read(*task_handle*, *event_type*, *num_samples*, *callback_data=None*)
Called as a callback by DAQmx while task is running. Also called by us to get remaining data just prior to stopping the task. Since the callback runs in a separate thread, we need to serialise access to instance variables

shutdown()

start_task(*chans*, *rate*)
Set up a task that acquires data with a callback every MAX_READ_PTS points or MAX_READ_INTERVAL seconds, whichever is faster. NI DAQmx calls callbacks in a separate thread, so this method returns, but data acquisition continues until stop_task() is called. Data is appended to self.acquired_data if self.buffered_mode=True, or (TODO) sent to the [whatever the AI server broker is called] if self.buffered_mode=False.

stop_task()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual(*abort=False*)

class labscript_devices.NI_DAQmx.blacs_workers.**NI_DAQmxOutputWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

check_version()
Check the version of PyDAQmx is high enough to avoid a known bug

get_output_tables(*h5file*, *device_name*)
Return the AO and DO tables rom the file, or None if they do not exist.

init()

program_buffered_AO(*AO_table*)

program_buffered_DO(*DO_table*)
Create the DO task and program in the DO table for a shot. Return a dictionary of the final values of each channel in use

program_manual(*front_panel_values*)

set_connected_terminals_connected(*connected*)
Connect the terminals in the connected terminals list. Allows on daisy chaining of the clock line to/from other devices that do not have a direct route (see Device Routes in NI MAX).

set_mirror_clock_terminal_connected(*connected*)
Mirror the clock terminal on another terminal to allow daisy chaining of the clock line to other devices, if applicable

shutdown()

start_manual_mode_tasks()

stop_tasks()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual(*abort=False*)

class labscript_devices.NI_DAQmx.blacs_workers.**NI_DAQmxWaitMonitorWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

init()

program_manual(*values*)

read_edges(*npts*, *timeout=None*)

Wait up to the given timeout in seconds for an edge on the wait monitor and and return the duration since the previous edge. Return None upon timeout.

send_resume_trigger(*pulse_width*)

shutdown()

start_tasks()

stop_tasks(*abort*)

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual(*abort=False*)

wait_monitor()

class labscript_devices.NI_DAQmx.runviewer_parsers.**NI_DAQmxParser**(*path*, *device*)

Bases: object

get_traces(*add_trace*, *clock=None*)

labscript_devices.NI_DAQmx.daqmx_utils.**get_CI_chans**(*device_name*)

labscript_devices.NI_DAQmx.daqmx_utils.**get_devices**()

labscript_devices.NI_DAQmx.daqmx_utils.**get_product_type**(*device_name*)

labscript_devices.NI_DAQmx.daqmx_utils.**incomplete_sample_detection**(*device_name*)

Introspect whether a device has 'incomplete sample detection', described here:

www.ni.com/documentation/en/ni-daqmx/latest/devconsid/incompletesampledetection/

The result is determined empirically by outputting a pulse on one counter and measuring it on another, and seeing whether the first sample was discarded or not. This requires a non-simulated device with at least two counters. No external signal is actually generated by the device whilst this test is performed. Credit for this method goes to Kevin Price, who provided it here:

forums.ni.com/t5/Multifunction-DAQ/_/td-p/3849429

This workaround will hopefully be deprecated if and when NI provides functionality to either inspect this feature's presence directly, or to disable it regardless of its presence.

labscript_devices.NI_DAQmx.daqmx_utils.**is_simulated**(*device_name*)

labscript_devices.NI_DAQmx.daqmx_utils.**supports_period_measurement**(*device_name*)

labscript_devices.NI_DAQmx.utils.**split_conn_AI**(*connection*)
Return analog input number of a connection string such as 'ai1' as an integer, or raise ValueError if format is invalid

labscript_devices.NI_DAQmx.utils.**split_conn_AO**(*connection*)
Return analog output number of a connection string such as 'ao1' as an integer, or raise ValueError if format is invalid

labscript_devices.NI_DAQmx.utils.**split_conn_DO**(*connection*)
Return the port and line number of a connection string such as 'port0/line1 as two integers, or raise ValueError if format is invalid. Accepts connection strings such as port1/line0 (PFI0) - the PFI bit is just ignored

labscript_devices.NI_DAQmx.utils.**split_conn_PFI**(*connection*)
Return PFI input number of a connection string such as 'PFI0' as an integer, or raise ValueError if format is invalid

labscript_devices.NI_DAQmx.utils.**split_conn_port**(*connection*)
Return port number of a string such as 'port0' as an integer, or raise ValueError if format is invalid

2.3 Cameras

The camera devices provide interfaces for using various scientific cameras to acquire hardware-timed images during an experiment. They are organized by the programming API the underlies the communication to the device. The "master" camera class which provides the core functionality and from which the others derive is the IMAQdx class.

2.3.1 IMAQdx Cameras

Overview

The "master" camera device from which all others derive.

labscript_devices.IMAQdxCamera.labscript_devices

labscript_devices.IMAQdxCamera.blacs_tabs

labscript_devices.IMAQdxCamera.blacs_workers

Installation

Usage

Detailed Documentation

class labscript_devices.IMAQdxCamera.labscript_devices.**IMAQdxCamera**(*name*, *parent_device*, *connection*, *serial_number*, *orientation=None*, *pixel_size=[1.0, 1.0]*, *magnification=1.0*, *trigger_edge_type='rising'*, *trigger_duration=None*, *minimum_recovery_time=0.0*, *camera_attributes=None*, *manual_mode_camera_attributes=None*, *stop_acquisition_timeout=5.0*, *exception_on_failed_shot=True*, *saved_attribute_visibility_level='intermediate'*, *mock=False*, ***kwargs*)

Bases: labscript.labscript.TriggerableDevice

A camera to be controlled using NI IMAQdx and triggered with a digital edge.

Parameters

- **name** (*str*) – device name
- **parent_device** (*IntermediateDevice*) – Device with digital outputs to be used to trigger acquisition
- **connection** (*str*) – Name of digital output port on parent device.
- **serial_number** (*str or int*) – string or integer (integer allows entering a hex literal) of the camera's serial number. This will be used to idenitfy the camera.
- **orientation** (*str, optional*) – <name> Description of the camera's location or orientation. This will be used to determine the location in the shot file where the images will be saved. If not given, the device name will be used instead.
- **pixel_size** (*[float,float], optional*) – `[1.0, 1.0]` The x and y size of the pixels in micrometers. This can be used in setting the scale in the blacs image display as well as extracted in lyse for analysis.
- **magnification** (*float, optional*) – `1.0` Imaging system magnification.
- **trigger_edge_type** (*str*) – `'rising'` The direction of the desired edges to be generated on the parent devices's digital output used for triggering. Must be 'rising' or 'falling'. Note that this only determines the edges created on the parent device, it does not program the camera to expect this type of edge. If required, one must configure the camera separately via `camera_attributes` to ensure it expects the type of edge being generated. Default: `'rising'`
- **trigger_duration** (*float or None*) – `None` Duration of digital pulses to be generated by the parent device. This can also be specified as an argument to `expose()` - the value given here will be used only if nothing is passed to `expose()`.
- **minimum_recovery_time** (*float*) – `0` Minimum time between frames. This will be used for error checking during compilation.

- **camera_attributes** (*dict, optional*) – Dictionary of camera attribute names and values to be programmed into the camera. The meaning of these attributes is model-specific. Attributes will be programmed in the order they appear in this dictionary. This can be important as some attributes may not be settable unless another attrbiute has been set first. After adding this device to your connection table, a dictionary of the camera's default attributes can be obtained from the BLACS tab, appropriate for copying and pasting into your connection table to customise the ones you are interested in.
- **manual_mode_camera_attributes** (*dict, optional*) – Dictionary of attributes that will be programmed into the camera during manual mode, that differ from their values in `camera_attributes`. This can be useful for example, to have software triggering during manual mode (allowing the acquisition of frames from the BLACS manual mode interface) but hardware triggering during buffered runs. Any attributes in this dictionary must also be present in `camera_attributes`.
- **stop_acquisition_timeout** (*float*) – `5.0` How long, in seconds, to wait during `transition_to_buffered` for the acquisition of images to complete before giving up. Whilst all triggers should have been received, this can be used to allow for slow image download time.
- **exception_on_failed_shot** (*bool*) – `True`. If acquisition does not complete within the given timeout after the end of a shot, whether to raise an exception. If False, instead prints a warning to stderr (visible in the terminal output pane in the BLACS tab), saves the images acquired so far, and continues. In the case of such a 'failed shot', the HDF5 attribute f['images'][orientation/name].attrs['failed_shot'] will be set to `True` (otherwise it is set to `False`). This attribute is acessible in the lyse dataframe as `df[orientation/name, 'failed_shot']`.
- **saved_attribute_visibility_level** (*str or None*) – 'intermediate' The detail level of the camera attributes saved to the HDF5 file at the end of each shot. If None, no attributes will be saved. Must be one of `'simple'`, `'intermediate'`, `'advanced'`, or `None`. If `None`, no attributes will be saved.
- **mock** (*bool, optional*) – False For testing purpses, simulate a camera with fake data instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (TriggerableDevice).

description = 'IMAQdx Camera'
Brief description of the device.

expose(*t*, *name*, *frametype='frame'*, *trigger_duration=None*)
Request an exposure at the given time. A trigger will be produced by the parent trigger object, with duration trigger_duration, or if not specified, of self.trigger_duration. The frame should have a `name, and optionally a `frametype`, both strings. These determine where the image will be stored in the hdf5 file. `name` should be a description of the image being taken, such as "insitu_absorption" or "fluorescence" or similar. `frametype` is optional and is the type of frame being acquired, for imaging methods that involve multiple frames. For example an absorption image of atoms might have three frames: 'probe', 'atoms' and 'background'. For this one might call expose three times with the same name, but three different frametypes.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (`h5py.File`) – Handle to shot file.

class labscript_devices.IMAQdxCamera.blacs_tabs.**IMAQdxCameraTab**(*notebook*, *settings*, *restart=False*)

Bases: blacs.device_base_class.DeviceTab

get_save_data()

initialise_GUI()

initialise_workers()

on_attr_visibility_level_changed(*value*)

on_attributes_clicked(*button*)

on_continuous_clicked(*button*)

on_copy_clicked(*button*)

on_max_rate_changed(*max_fps*)

on_reset_rate_clicked()

on_snap_clicked(**args*, ***kwargs*)

on_stop_clicked(*button*)

restart(**args*, ***kwargs*)

restore_save_data(*save_data*)

start_continuous(**args*, ***kwargs*)

stop_continuous(**args*, ***kwargs*)

update_attributes(**args*, ***kwargs*)

use_smart_programming = True

worker_class = 'labscript_devices.IMAQdxCamera.blacs_workers.IMAQdxCameraWorker'

class labscript_devices.IMAQdxCamera.blacs_tabs.**ImageReceiver**(**args*, ***kwargs*)

Bases: labscript_utils.ls_zprocess.ZMQServer

ZMQServer that receives images on a zmq.REP socket, replies 'ok', and updates the image widget and fps indicator

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type *Any*

handler(*data*)

labscript_devices.IMAQdxCamera.blacs_tabs.**exp_av**(*av_old*, *data_new*, *dt*, *tau*)
Compute the new value of an exponential moving average based on the previous average av_old, a new value data_new, a time interval dt and an averaging timescale tau. Returns data_new if dt > tau

class labscript_devices.IMAQdxCamera.blacs_workers.**IMAQdxCameraWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

_send_image_to_parent(*image*)
Send the image to the GUI to display. This will block if the parent process is lagging behind in displaying frames, in order to avoid a backlog.

abort()

abort_buffered()

abort_transition_to_buffered()

continuous_loop(*dt*)
Acquire continuously in a loop, with minimum repetition interval dt

get_attributes_as_dict(*visibility_level*)
Return a dict of the attributes of the camera for the given visibility level

get_attributes_as_text(*visibility_level*)
Return a string representation of the attributes of the camera for the given visibility level

get_camera()
Return an instance of the camera interface class. Subclasses may override this method to pass required arguments to their class if they require more than just the serial number.

init()

interface_class
alias of *labscript_devices.IMAQdxCamera.blacs_workers.IMAQdx_Camera*

program_manual(*values*)

set_attributes_smart(*attributes*)
Call self.camera.set_attributes() to set the given attributes, only setting those that differ from their value in, or are absent from self.smart_cache. Update self.smart_cache with the newly-set values

shutdown()

snap()
Acquire one frame in manual mode. Send it to the parent via self.image_socket. Wait for a response from the parent.

start_continuous(*dt*)
Begin continuous acquisition in a thread with minimum repetition interval dt

stop_continuous(*pause=False*)
Stop the continuous acquisition thread

transition_to_buffered(*device_name*, *h5_filepath*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.IMAQdxCamera.blacs_workers.**IMAQdx_Camera**(*serial_number*)
Bases: object

_decode_image_data(*img*)

abort_acquisition()

close()

configure_acquisition(*continuous=True*, *bufferCount=5*)

get_attribute(*name*)
Return current value of attribute of the given name

get_attribute_names(*visibility_level*, *writeable_only=True*)
Return a list of all attribute names of readable attributes, for the given visibility level. Optionally return only writeable attributes

grab(*waitForNextBuffer=True*)

grab_multiple(*n_images*, *images*, *waitForNextBuffer=True*)

set_attribute(*name*, *value*)
Set the value of the attribute of the given name to the given value

set_attributes(*attr_dict*)

snap()
Acquire a single image and return it

stop_acquisition()

class labscript_devices.IMAQdxCamera.blacs_workers.**MockCamera**
Bases: object

Mock camera class that returns fake image data.

abort_acquisition()

close()

configure_acquisition(*continuous=False*, *bufferCount=5*)

get_attribute(*name*)

get_attribute_names(*visibility_level=None*)

grab()

grab_multiple(*n_images*, *images*, *waitForNextBuffer=True*)

set_attributes(*attributes*)

snap()

stop_acquisition()

labscript_devices.IMAQdxCamera.blacs_workers.**_monkeypatch_imaqdispose**()
Monkeypatch a fix to a memory leak bug in pynivision. The pynivision project is no longer active, so we can't contribute this fix upstream. In the long run, hopefully someone (perhaps us) forks it so that bugs can be addressed in the normal way

2.3.2 Pylon Cameras

Overview

This device allows control of Basler scientific cameras via the Pylon API with the PyPylon python wrapper. In order to use this device, both the Basler Pylon API and the PyPylon wrapper must be installed.

labscript_devices.PylonCamera.labscript_devices

labscript_devices.PylonCamera.blacs_tabs

labscript_devices.PylonCamera.blacs_workers

Installation

First ensure that the Basler Pylon SDK is installed. It is available for free here (after signing up for a free account with Basler). It is advisable to use the Pylon Viewer program that comes with the SDK to test communications with the camera.

The python wrapper is installed via pip:

```
pip install -U pypylon
```

At present, the wrapper is tested and confirmed compatible with Pylon 5 for USB3 and GigE interface cameras.

For GigE cameras, ensure that the network interface card (NIC) on the computer with the BLACS controlling the camera has enabled Jumbo Frames. That maximum allowed value (typically 9000) is preferable to avoid dropped frames.

For USB3 cameras, care should be taken to use a USB3 host that is compatible with the Basler cameras. Basler maintains a list of compatible host controllers. The cameras will work on any USB3 port, but non-compatible hosts will not allow for the faster performance.

Usage

Like the *IMAQdxCamera* device, the bulk of camera configuration is performed using a dictionary of kwargs, where the key names and values mirror those provided by the Pylon SDK interface. Which parameters can/need to be set depend on the communication interface. Discovery of what parameters are available can be done in three ways:

1. Careful reading of the Pylon SDK docs.
2. Mirroring the Pylon Viewer parameter names and values.
3. Connecting to the camera with a minimal configuration, viewing the current parameters dictionary, and copying the relevant values to the connection table (preferred).

Below are generic configurations for GigE and USB3 based cameras.

```
from labscript import *

from labscript_devices.PylonCamera.labscript_devices import PylonCamera

PylonCamera('gigeCamera',parent_device=parent,connection=conn,
            serial_number=1234567, # set to the camera serial number
            minimum_recovery_time=20e-3, # the minimum exposure time depends on the camera model & configuration
            camera_attributs={
               'ExposureTimeAbs':1000, #in us
```

(continues on next page)

(continued from previous page)

```
                'ExposureMode':'Timed',
                'ExposureAuto':'Off',
                'GainAuto':'Off',
                'PixelFormat':'Mono12Packed',
                'Gamma':1.0,
                'BlackLevelRaw':0,
                'TriggerSource':'Line 1',
                'TriggerMode':'On'
            },
            manual_camera_attributes={
                'TriggerSource':'Software',
                'TriggerMode':'Off'
            })

PylonCamera('usb3Camera',parent_device=parent,connection=conn,
            serial_number=12345678,
            minimum_recovery_time=36e-3,
            camera_attributs={
                'ExposureTime':1000, #in us
                'ExposureMode':'Timed',
                'ExposureAuto':'Off',
                'GainAuto':'Off',
                'PixelFormat':'Mono12Packed',
                'Gamma':1.0,
                'BlackLevel':0,
                'TriggerSource':'Line 1',
                'TriggerMode':'On',
                'ShutterMode':'Global'
            },
            manual_camera_attributes={
                'TriggerSource':'Software',
                'TriggerMode':'Off'
            })

start()

gigeCamera.expose(t=0.5,'exposure1')
```

(continues on next page)

(continued from previous page)

```
usb3Camera.expose(t=0.45,'exposure2')

stop(1)
```

Utilities

The Pylon labscript device includes a script in the `testing` subfolder that can automatically determine the full-frame sensor readout time and maximum possible framerate. This tool helps in correctly determining the appropriate `minimum_recovery_time` to set for each device. The minimum recovery time is a function of the model used, the communication bus used, and minor details of the setup (such as host controller firmwares, cable lengths, host computer workload, etc). As a result, live testing of the device is often needed to accurately determine the actual recovery time needed between shots.

The script is run from within the testing folder using

```
python ExposureTiming.py [camera_sn]
```

with `[camera_sn]` being the serial number of the camera to connect to and test.

The script reports the minimum recovery time between two shots of 1 ms exposure each, without the use of overlapped exposure mode. Editing the script to include your typical experiment parameters will help in more accurately determining your minimum recovery time. Typically, the minimum recovery time should be slightly longer than the reported sensor readout time.

Note that in overlapped exposure mode, a second exposure is begun before the first exposure has finished reading out and *must* end after the readout of the first exposure frame is complete. This allows for a series of two exposures with shorter delay between them, at the expense of limitations on the length of the second exposure. The script will also report the minimum time between the end of one exposure and the beginning of the second (nominally `readout_time - exposure_time`). Note that this feature is automatically handled at the Pylon API level; this labscript device is not actively aware of it. As a result, incorrect uses of overlapped mode will not be caught at compile time, but rather during the shot as hardware errors.

Detailed Documentation

class labscript_devices.PylonCamera.labscript_devices.**PylonCamera**(*name*, *parent_device*, *connection*, *serial_number*, *orientation=None*, *pixel_size=[1.0, 1.0]*, *magnification=1.0*, *trigger_edge_type='rising'*, *trigger_duration=None*, *minimum_recovery_time=0.0*, *camera_attributes=None*, *manual_mode_camera_attributes=None*, *stop_acquisition_timeout=5.0*, *exception_on_failed_shot=True*, *saved_attribute_visibility_level='intermediate'*, *mock=False*, ***kwargs*)

Bases: *labscript_devices.IMAQdxCamera.labscript_devices.IMAQdxCamera*

A camera to be controlled using NI IMAQdx and triggered with a digital edge.

Parameters

- **name** (*str*) – device name
- **parent_device** (*IntermediateDevice*) – Device with digital outputs to be used to trigger acquisition
- **connection** (*str*) – Name of digital output port on parent device.
- **serial_number** (*str or int*) – string or integer (integer allows entering a hex literal) of the camera's serial number. This will be used to idenitfy the camera.
- **orientation** (*str, optional*) – <name> Description of the camera's location or orientation. This will be used to determine the location in the shot file where the images will be saved. If not given, the device name will be used instead.
- **pixel_size** (*[float,float], optional*) – `[1.0, 1.0]` The x and y size of the pixels in micrometers. This can be used in setting the scale in the blacs image display as well as extracted in lyse for analysis.
- **magnification** (*float, optional*) – `1.0` Imaging system magnification.
- **trigger_edge_type** (*str*) – `'rising'` The direction of the desired edges to be generated on the parent devices's digital output used for triggering. Must be 'rising' or 'falling'. Note that this only determines the edges created on the parent device, it does not program the camera to expect this type of edge. If required, one must configure the camera separately via `camera_attributes` to ensure it expects the type of edge being generated. Default: `'rising'`
- **trigger_duration** (*float or None*) – `None` Duration of digital pulses to be generated by the parent device. This can also be specified as an argument to `expose()` - the value given here will be used only if nothing is passed to `expose()`.
- **minimum_recovery_time** (*float*) – `0` Minimum time between frames. This will be used for error checking during compilation.
- **camera_attributes** (*dict, optional*) – Dictionary of camera attribute names and values to be programmed into the camera. The meaning of these attributes is model-specific. Attributes will be programmed in the order they appear in this dictionary. This can be important as some attributes may not be settable unless another attrbiute has been set first. After adding this device to your connection table, a dictionary of the camera's default attributes can be obtained from the BLACS tab, appropriate for copying and pasting into your connection table to customise the ones you are interested in.
- **manual_mode_camera_attributes** (*dict, optional*) – Dictionary of attributes that will be programmed into the camera during manual mode, that differ from their values in `camera_attributes`. This can be useful for example, to have software triggering during manual mode (allowing the acquisition of frames from the BLACS manual mode interface) but hardware triggering during buffered runs. Any attributes in this dictionary must also be present in `camera_attributes`.
- **stop_acquisition_timeout** (*float*) – `5.0` How long, in seconds, to wait during `transition_to_buffered` for the acquisition of images to complete before giving up. Whilst all triggers should have been received, this can be used to allow for slow image download time.

- **exception_on_failed_shot** (*bool*) – `True`. If acquisition does not complete within the given timeout after the end of a shot, whether to raise an exception. If False, instead prints a warning to stderr (visible in the terminal output pane in the BLACS tab), saves the images acquired so far, and continues. In the case of such a 'failed shot', the HDF5 attribute f['images'][orientation/name].attrs['failed_shot'] will be set to `True` (otherwise it is set to `False`). This attribute is acessible in the lyse dataframe as `df[orientation/name, 'failed_shot']`.
- **saved_attribute_visibility_level** (*str or None*) – 'intermediate' The detail level of the camera attributes saved to the HDF5 file at the end of each shot. If None, no attributes will be saved. Must be one of `'simple'`, `'intermediate'`, `'advanced'`, or `None`. If `None`, no attributes will be saved.
- **mock** (*bool, optional*) – False For testing purpses, simulate a camera with fake data instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (TriggerableDevice).

description = 'Pylon Camera'
Brief description of the device.

class labscript_devices.PylonCamera.blacs_tabs.**PylonCameraTab**(*notebook*, *settings*, *restart=False*)
Bases: *labscript_devices.IMAQdxCamera.blacs_tabs.IMAQdxCameraTab*

worker_class = 'labscript_devices.PylonCamera.blacs_workers.PylonCameraWorker'

class labscript_devices.PylonCamera.blacs_workers.**PylonCameraWorker**(**args*, ***kwargs*)
Bases: *labscript_devices.IMAQdxCamera.blacs_workers.IMAQdxCameraWorker*

Pylon API Camera Worker.

Inherits from IMAQdxCameraWorker. Overloads get_attributes_as_dict to use PylonCamera.get_attributes() method.

get_attributes_as_dict(*visibility_level*)
Return a dict of the attributes of the camera for the given visibility level

interface_class
alias of *labscript_devices.PylonCamera.blacs_workers.Pylon_Camera*

class labscript_devices.PylonCamera.blacs_workers.**Pylon_Camera**(*serial_number*)
Bases: `object`

abort_acquisition()

close()

configure_acquisition(*continuous=True*, *bufferCount=10*)
Configure acquisition by calling StartGrabbing with appropriate grab strategy: LatestImageOnly for continuous, OneByOne otherwise.

get_attribute(*name*)
Return current value of attribute of the given name

get_attributes(*visibility_level*, *writeable_only=True*)
Return a dict of all attributes of readable attributes, for the given visibility level. Optionally return only writeable attributes.

grab(*continuous=True*)
Grab single image during pre-configured acquisition.

grab_multiple(*n_images*, *images*)
Grab n_images into images array during buffered acquistion.

set_attribute(*name*, *value*)
Set the value of the attribute of the given name to the given value

set_attributes(*attributes_dict*)
Sets all attribues in attr_dict. Pylon cameras require that ROI settings be done in correct order, so we do them separately.

snap()
Acquire a single image and return it

stop_acquisition()

2.3.3 FlyCapture2 Cameras

This device allows control of FLIR (formerly Point Grey) scientific cameras via the FlyCapture2 SDK with the now deprecated PyCapture2 wrapper. In order to use this device, both the SDK and the python wrapper must be installed. Note that PyCapture2 only supports up to Python 3.6.

The new FLIR SDK is supported using the *SpinnakerCamera labscript device*.

labscript_devices.FlyCapture2Camera.labscript_devices

labscript_devices.FlyCapture2Camera.blacs_tabs

labscript_devices.FlyCapture2Camera.blacs_workers

Installation

First ensure that the FlyCapture2 SDK is installed.

The python wrapper is available via FLIR and is only released for Python up to 3.6. It must be installed separately, pointed to the correct conda environment during install.

For GigE cameras, ensure that the network interface card (NIC) on the computer with the BLACS controlling the camera has enabled Jumbo Frames. That maximum allowed value (typically 9000) is preferable to avoid dropped frames.

Usage

Like the *IMAQdxCamera* device, the bulk of camera configuration is performed using a dictionary of kwargs, where the key names and values mirror those provided by the FlyCapture2 SDK interface. Which parameters can/need to be set depend on the communication interface. Discovery of what parameters are available can be done in three ways:

1. Careful reading of the FlyCapture2 SDK docs.
2. Mirroring the FlyCap Viewer parameter names and values.
3. Connecting to the camera with a minimal configuration, viewing the current parameters dictionary, and copying the relevant values to the connection table (preferred).

The python structure for setting these values differs somewhat from other camera devices in labscript, taking the form of nested dictionaries. This structure most closely matches the structure of the FlyCapture2 SDK in that each camera property has multiple sub-elements that control the feature. In this implementation, the standard camera properties are set using keys with ALL CAPS. The control of the Trigger Mode and Image Mode properties is handled separately, using a slightly different nesting structure than the other properties.

Below is a generic configuration for a Point Grey Blackfly PGE-23S6M-C device.

```
from labscript import *

from labscript_devices.FlyCapture2Camera.labscript_devices import FlyCapture2Camera

FlyCapture2Camera('gigeCamera',parent_device=parent,connection=conn,
            serial_number=1234567, # set to the camera serial number
            minimum_recovery_time=36e-6, # the minimum exposure time depends on the camera model & configuration
            camera_attributs={
                            'GAMMA':{
                                    'onOff':False,
                                    'absControl':True,
```

(continues on next page)

(continued from previous page)

```
                                        'absValue':1},
                                    'AUTO_EXPOSURE':{
                                        'onOff':True,
                                        'absControl':True,
                                        'autoManualMode':False,
                                        'absValue':0},
                                    'GAIN':{
                                        'autoManualMode':False,
                                        'absControl':True,
                                        'absValue':0},
                                    'SHARPNESS':{
                                        'onOff':False,
                                        'autoManualMode':False,
                                        'absValue':1024},
                                    'FRAME_RATE':{
                                        'autoManualMode':False,
                                        'absControl':True},
                                    'SHUTTER':{
                                        'autoManualMode':False,
                                        'absValue':0},
                                    'TriggerMode':{
                                        'polarity':1,
                                        'source':0,
                                        'mode':1,
                                        'onOff':True},
                                    'ImageMode':{
                                        'width':1920,
                                        'height':1200,
                                        'offsetX':0,
                                        'offsetY':0,
                                        'pixelFormat':'MONO16'}
            },
            manual_camera_attributes={'TriggerMode':{'onOff':False}})

start()

gigeCamera.expose(t=0.5,'exposure1')
```

(continues on next page)

(continued from previous page)

```
stop(1)
```

Detailed Documentation

class labscript_devices.FlyCapture2Camera.labscript_devices.**FlyCapture2Camera**(*name*, *parent_device*, *connection*, *serial_number*, *orientation=None*, *pixel_size=[1.0, 1.0]*, *magnification=1.0*, *trigger_edge_type='rising'*, *trigger_duration=None*, *minimum_recovery_time=0.0*, *camera_attributes=None*, *manual_mode_camera_attributes=None*, *stop_acquisition_timeout=5.0*, *exception_on_failed_shot=True*, *saved_attribute_visibility_level='intermediate'*, *mock=False*, ***kwargs*)

Bases: *labscript_devices.IMAQdxCamera.labscript_devices.IMAQdxCamera*

Thin sub-class of `IMAQdxCamera`.

A camera to be controlled using NI IMAQdx and triggered with a digital edge.

Parameters

- **name** (*str*) – device name
- **parent_device** (*IntermediateDevice*) – Device with digital outputs to be used to trigger acquisition
- **connection** (*str*) – Name of digital output port on parent device.
- **serial_number** (*str or int*) – string or integer (integer allows entering a hex literal) of the camera's serial number. This will be used to idenitfy the camera.
- **orientation** (*str, optional*) – <name> Description of the camera's location or orientation. This will be used to determine the location in the shot file where the images will be saved. If not given, the device name will be used instead.
- **pixel_size** (*[float,float], optional*) – `[1.0, 1.0]` The x and y size of the pixels in micrometers. This can be used in setting the scale in the blacs image display as well as extracted in lyse for analysis.
- **magnification** (*float, optional*) – `1.0` Imaging system magnification.

- **trigger_edge_type** (*str*) – `'rising'` The direction of the desired edges to be generated on the parent devices's digital output used for triggering. Must be 'rising' or 'falling'. Note that this only determines the edges created on the parent device, it does not program the camera to expect this type of edge. If required, one must configure the camera separately via `camera_attributes` to ensure it expects the type of edge being generated. Default: `'rising'`
- **trigger_duration** (*float or None*) – `None` Duration of digital pulses to be generated by the parent device. This can also be specified as an argument to `expose()` - the value given here will be used only if nothing is passed to `expose()`.
- **minimum_recovery_time** (*float*) – `0` Minimum time between frames. This will be used for error checking during compilation.
- **camera_attributes** (*dict, optional*) – Dictionary of camera attribute names and values to be programmed into the camera. The meaning of these attributes is model-specific. Attributes will be programmed in the order they appear in this dictionary. This can be important as some attributes may not be settable unless another attrbiute has been set first. After adding this device to your connection table, a dictionary of the camera's default attributes can be obtained from the BLACS tab, appropriate for copying and pasting into your connection table to customise the ones you are interested in.
- **manual_mode_camera_attributes** (*dict, optional*) – Dictionary of attributes that will be programmed into the camera during manual mode, that differ from their values in `camera_attributes`. This can be useful for example, to have software triggering during manual mode (allowing the acquisition of frames from the BLACS manual mode interface) but hardware triggering during buffered runs. Any attributes in this dictionary must also be present in `camera_attributes`.
- **stop_acquisition_timeout** (*float*) – `5.0` How long, in seconds, to wait during `transition_to_buffered` for the acquisition of images to complete before giving up. Whilst all triggers should have been received, this can be used to allow for slow image download time.
- **exception_on_failed_shot** (*bool*) – `True`. If acquisition does not complete within the given timeout after the end of a shot, whether to raise an exception. If False, instead prints a warning to stderr (visible in the terminal output pane in the BLACS tab), saves the images acquired so far, and continues. In the case of such a 'failed shot', the HDF5 attribute f['images'][orientation/name].attrs['failed_shot'] will be set to `True` (otherwise it is set to `False`). This attribute is acessible in the lyse dataframe as `df[orientation/name, 'failed_shot']`.
- **saved_attribute_visibility_level** (*str or None*) – 'intermediate' The detail level of the camera attributes saved to the HDF5 file at the end of each shot. If None, no attributes will be saved. Must be one of `'simple'`, `'intermediate'`, `'advanced'`, or `None`. If `None`, no attributes will be saved.
- **mock** (*bool, optional*) – False For testing purpses, simulate a camera with fake data instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (TriggerableDevice).

description = 'FlyCapture2 Camera'
Brief description of the device.

class labscript_devices.FlyCapture2Camera.blacs_tabs.**FlyCapture2CameraTab**(*notebook*, *settings*, *restart=False*)
Bases: *labscript_devices.IMAQdxCamera.blacs_tabs.IMAQdxCameraTab*

Thin sub-class of obj:`IMAQdxCameraTab`.

This sub-class only defines *worker_class* to point to the correct FlyCapture2CameraWorker.

worker_class = 'labscript_devices.FlyCapture2Camera.blacs_workers.FlyCapture2CameraWorker'

class labscript_devices.FlyCapture2Camera.blacs_workers.**FlyCapture2CameraWorker**(**args*, ***kwargs*)
Bases: *labscript_devices.IMAQdxCamera.blacs_workers.IMAQdxCameraWorker*

FlyCapture2 API Camera Worker.

Inherits from obj:IMAQdxCameraWorker. Defines *interface_class* and overloads *get_attributes_as_dict* to use FlyCapture2Camera.get_attributes() method.

get_attributes_as_dict(*visibility_level*)
Return a dict of the attributes of the camera for the given visibility level

Parameters **visibility_level** (*str*) – Normally configures level of attribute detail to return. Is not used by FlyCapture2_Camera.

interface_class
alias of *labscript_devices.FlyCapture2Camera.blacs_workers.FlyCapture2_Camera*

class labscript_devices.FlyCapture2Camera.blacs_workers.**FlyCapture2_Camera**(*serial_number*)
Bases: object

The backend hardware interface class for the FlyCapture2Camera.

This class handles all of the API/hardware implementation details for the corresponding labscript device. It is used by the BLACS worker to send appropriate API commands to the camera for the standard BLACS camera operations (i.e. transition_to_buffered, get_attributes, snap, etc).

camera
Handle to connected camera.

Type PyCapture2.Camera

get_props
This list sets which values of each property object are returned when queried by *get_attribute*.

Type list

pixel_formats
An IntEnum object that is automatically populated with the supported pixel types of the connected camera.

Type IntEnum

width
Width of images for most recent acquisition. Used by *_decode_image_data* to format images correctly.

Type int

height
Height of images for most recent acquisition. Used by *_decode_image_data* to format images correctly.

Type int

pixelFormat
Pixel format name for most recent acquisition. Used by *_decode_image_data* to format images correctly.

Type str

_abort_acquisition
Abort flag that is polled during buffered acquisitions.

Type bool

Initialize FlyCapture2 API camera.

Searches all cameras reachable by the host using the provided serial number. Fails with API error if camera not found.

This function also does a significant amount of default configuration.

- It defaults the grab timeout to 1 s
- Ensures use of the API's HighPerformanceRetrieveBuffer
- Ensures the camera is in Format 7, Mode 0 with full frame readout and MONO8 pixels
- If using a GigE camera, automatically maximizes the packet size and warns if Jumbo packets are not enabled on the NIC

Parameters **serial_number** (*int*) – serial number of camera to connect to

_decode_image_data(*img*)
Formats returned FlyCapture2 API image buffers.

FlyCapture2 image buffers require significant formatting. This returns what one would expect from a camera. *configure_acquisition* must be called first to set image format parameters.

Parameters **img** (*numpy.array*) – A 1-D array image buffer of uint8 values to format

Returns

Formatted array based on *width*, *height*, and *pixelFormat*.

Return type numpy.array

_send_format7_config(*image_config*)
Validates and sends the Format7 configuration packet.

Parameters **image_config** (*PyCapture2.Format7ImageSettings*) – Format7ImageSettings object to validate and send to camera.

abort_acquisition()
Sets *_abort_acquisition* flag to break buffered acquisition loop.

close()
Closes *camera* handle to the camera.

configure_acquisition(*continuous=True*, *bufferCount=10*)
Configure acquisition buffer count and grab mode.

This method also saves image width, heigh, and pixelFormat to class attributes for returned image formatting.

Parameters

- **continuous** (bool, optional) – If True, camera will continuously acquire and only keep most recent frames in the buffer. If False, all acquired frames are kept and error occurs if buffer is exceeded. Default is True.
- **bufferCount** (int, optional) – Number of memory buffers to use in the acquistion. Default is 10.

get_attribute(*name*)
Return current values dictionary of attribute of the given name.

Parameters **name** (*str*) – Property name to read

Returns

Dictionary of property values with structure as defined in *set_attribute*.

Return type dict

get_attributes(*visibility_level*, *writeable_only=True*)
Return a nested dict of all readable attributes.

Parameters

- **visibility_level** (*str*) – Not used.
- **writeable_only** (bool, optional) – Not used

Returns Dictionary of property dictionaries

Return type dict

grab()
Grab and return single image during pre-configured acquisition.

Returns Returns formatted image

Return type numpy.array

grab_multiple(*n_images*, *images*)
Grab n_images into images array during buffered acquistion.

Grab method involves a continuous loop with fast timeout in order to poll *_abort_acquisition* for a signal to abort.

Parameters

- **n_images** (*int*) – Number of images to acquire. Should be same number as the bufferCount in *configure_acquisition*.
- **images** (*list*) – List that images will be saved to as they are acquired

set_attribute(*name*, *values*)
Set the values of the attribute of the given name using the provided dictionary values.

Generally, absControl should be used to configure settings. Note that invalid settings tend to coerce instead of presenting an error.

Parameters

- **name** (*str*) –
- **values** (*dict*) – Dictionary of settings for the property. Allowed keys are:
  - 'onOff': bool
  - 'autoManualMode': bool
  - 'absControl': bool
  - 'absValue': float
  - 'valueA': int
  - 'valueB': int
  - 'onePush': bool

set_attributes(*attr_dict*)
Sets all attribues in attr_dict.

FlyCapture does not control all settings through same interface, so we must do them separately. Interfaces are: <Standard PROPERTY_TYPE>, TriggerMode, ImageMode

Parameters **attr_dict** (*dict*) – dictionary of property dictionaries to set for the camera. These property dictionaries assume a specific structure, outlined in *set_attribute*, *set_trigger_mode* and , *set_image_mode* methods.

set_image_mode(*image_settings*)
Configures ROI and image control via Format 7, Mode 0 interface.

Parameters **image_settings** (*dict*) – dictionary of image settings. Allowed keys:

- 'pixelFormat': valid pixel format string, i.e. 'MONO8'
- 'offsetX': int
- 'offsetY': int
- 'width': int
- 'height': int

set_trigger_mode(*trig_dict*)
Configures triggering options via Trigger Mode interface.

Parameters **trig_dict** (*dict*) – dictionary with trigger mode property settings. Allowed keys:

- 'onOff': bool
- 'polarity': 0,1
- 'source': int
- 'mode': int

snap()
Acquire a single image and return it

Returns Acquired image

Return type numpy.array

stop_acquisition()
Tells camera to stop current acquistion.

2.3.4 Spinnaker Cameras

This device allows control of FLIR scientific cameras via the Spinnaker SDK with the PySpin wrapper. In order to use this device, both the SDK and the python wrapper must be installed.

labscript_devices.SpinnakerCamera.labscript_devices

labscript_devices.SpinnakerCamera.blacs_tabs

continues on next page

Table 7 – continued from previous page

labscript_devices.SpinnakerCamera.blacs_workers

Installation

First ensure that the Spinnaker SDK is installed.

The python wrapper is available via FLIR. It must be installed separately and pointed to the correct conda environment during install.

For GigE cameras, ensure that the network interface card (NIC) on the computer with the BLACS controlling the camera has enabled Jumbo Frames. The maximum allowed value (typically 9000) is preferable to avoid dropped frames.

Usage

Like the *IMAQdxCamera* device, the bulk of camera configuration is performed using a dictionary of kwargs, where the key names and values mirror those provided by the Spinnaker SDK interface. Which parameters can/need to be set depend on the communication interface. Discovery of what parameters are available can be done in three ways:

1. Careful reading of the Spinnaker SDK docs.
2. Mirroring the SpinView parameter names and values.
3. Connecting to the camera with a minimal configuration, viewing the current parameters dictionary, and copying the relevant values to the connection table (preferred).

Below is a generic configuration.

```
from labscript import *

from labscript_devices.SpinnakerCamera.labscript_devices import SpinnakerCamera

CCT_global_camera_attributes = {
    'AnalogControl::GainAuto': 'Off',
    'AnalogControl::Gain': 0.0,
    'AnalogControl::BlackLevelEnabled': True,
    'AnalogControl::BlackLevel': 0.0,
    'AnalogControl::GammaEnabled': False,
    'AnalogControl::SharpnessEnabled': False,
    'ImageFormatControl::Width': 1008,
```

(continues on next page)

(continued from previous page)

```
   'ImageFormatControl::Height': 800,
   'ImageFormatControl::OffsetX': 200,
   'ImageFormatControl::OffsetY': 224,
   'ImageFormatControl::PixelFormat': 'Mono16',
   'ImageFormatControl::VideoMode': 'Mode0',
   'AcquisitionControl::TriggerMode': 'Off',
   'AcquisitionControl::TriggerSource': 'Line0',
   'AcquisitionControl::TriggerSelector': 'ExposureActive',
   'AcquisitionControl::TriggerActivation': 'FallingEdge',
 }
CCT_manual_mode_attributes = {
   'AcquisitionControl::TriggerMode': 'Off',
   'AcquisitionControl::ExposureMode': 'Timed',
}
CCT_buffered_mode_attributes = {
   'AcquisitionControl::TriggerMode': 'On',
   'AcquisitionControl::ExposureMode': 'TriggerWidth',
}

SpinnakerCamera('gigeCamera',parent_device=parent,connection=conn,
            serial_number=1234567, # set to the camera serial number
            minimum_recovery_time=36e-6, # the minimum exposure time depends on the camera model & configuration
            trigger_edge_type='falling',
            camera_attributs={**CCT_global_camera_attributes,
                              **CCT_buffered_mode_attributes},
            manual_camera_attributes={**CCT_global_camera_attributes,
                                      **CCT_manual_mode_attributes})

start()

gigeCamera.expose(t=0.5,'exposure1',trigger_duration=0.25)


stop(1)
```

Detailed Documentation

class labscript_devices.SpinnakerCamera.labscript_devices.**SpinnakerCamera**(*name*, *parent_device*, *connection*, *serial_number*, *orientation=None*, *pixel_size=[1.0, 1.0]*, *magnification=1.0*, *trigger_edge_type='rising'*, *trigger_duration=None*, *minimum_recovery_time=0.0*, *camera_attributes=None*, *manual_mode_camera_attributes=None*, *stop_acquisition_timeout=5.0*, *exception_on_failed_shot=True*, *saved_attribute_visibility_level='intermediate'*, *mock=False*, ***kwargs*)

Bases: *labscript_devices.IMAQdxCamera.labscript_devices.IMAQdxCamera*

A camera to be controlled using NI IMAQdx and triggered with a digital edge.

Parameters

- **name** (*str*) – device name
- **parent_device** (*IntermediateDevice*) – Device with digital outputs to be used to trigger acquisition
- **connection** (*str*) – Name of digital output port on parent device.
- **serial_number** (*str or int*) – string or integer (integer allows entering a hex literal) of the camera's serial number. This will be used to idenitfy the camera.
- **orientation** (*str, optional*) – <name> Description of the camera's location or orientation. This will be used to determine the location in the shot file where the images will be saved. If not given, the device name will be used instead.
- **pixel_size** (*[float,float], optional*) – `[1.0, 1.0]` The x and y size of the pixels in micrometers. This can be used in setting the scale in the blacs image display as well as extracted in lyse for analysis.
- **magnification** (*float, optional*) – `1.0` Imaging system magnification.
- **trigger_edge_type** (*str*) – `'rising'` The direction of the desired edges to be generated on the parent devices's digital output used for triggering. Must be 'rising' or 'falling'. Note that this only determines the edges created on the parent device, it does not program the camera to expect this type of edge. If required, one must configure the camera separately via `camera_attributes` to ensure it expects the type of edge being generated. Default: `'rising'`
- **trigger_duration** (*float or None*) – `None` Duration of digital pulses to be generated by the parent device. This can also be specified as an argument to `expose()` - the value given here will be used only if nothing is passed to `expose()`.
- **minimum_recovery_time** (*float*) – `0` Minimum time between frames. This will be used for error checking during compilation.
- **camera_attributes** (*dict, optional*) – Dictionary of camera attribute names and values to be programmed into the camera. The meaning of these attributes is model-specific. Attributes will be programmed in the order they appear in this dictionary. This can be

important as some attributes may not be settable unless another attrbiute has been set first. After adding this device to your connection table, a dictionary of the camera's default attributes can be obtained from the BLACS tab, appropriate for copying and pasting into your connection table to customise the ones you are interested in.

- **manual_mode_camera_attributes** (*dict, optional*) – Dictionary of attributes that will be programmed into the camera during manual mode, that differ from their values in `camera_attributes`. This can be useful for example, to have software triggering during manual mode (allowing the acquisition of frames from the BLACS manual mode interface) but hardware triggering during buffered runs. Any attributes in this dictionary must also be present in `camera_attributes`.
- **stop_acquisition_timeout** (*float*) – `5.0` How long, in seconds, to wait during `transition_to_buffered` for the acquisition of images to complete before giving up. Whilst all triggers should have been received, this can be used to allow for slow image download time.
- **exception_on_failed_shot** (*bool*) – `True`. If acquisition does not complete within the given timeout after the end of a shot, whether to raise an exception. If False, instead prints a warning to stderr (visible in the terminal output pane in the BLACS tab), saves the images acquired so far, and continues. In the case of such a 'failed shot', the HDF5 attribute f['images'][orientation/name].attrs['failed_shot'] will be set to `True` (otherwise it is set to `False`). This attribute is acessible in the lyse dataframe as `df[orientation/name, 'failed_shot']`.
- **saved_attribute_visibility_level** (*str or None*) – 'intermediate' The detail level of the camera attributes saved to the HDF5 file at the end of each shot. If None, no attributes will be saved. Must be one of `'simple'`, `'intermediate'`, `'advanced'`, or `None`. If `None`, no attributes will be saved.
- **mock** (*bool, optional*) – False For testing purpses, simulate a camera with fake data instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (TriggerableDevice).

description = 'Spinnaker Camera'
Brief description of the device.

class labscript_devices.SpinnakerCamera.blacs_tabs.**SpinnakerCameraTab**(*notebook*, *settings*, *restart=False*)
Bases: *labscript_devices.IMAQdxCamera.blacs_tabs.IMAQdxCameraTab*

worker_class = 'labscript_devices.SpinnakerCamera.blacs_workers.SpinnakerCameraWorker'

class labscript_devices.SpinnakerCamera.blacs_workers.**SpinnakerCameraWorker**(**args*, ***kwargs*)
Bases: *labscript_devices.IMAQdxCamera.blacs_workers.IMAQdxCameraWorker*

Spinnaker API Camera Worker.

Inherits from IMAQdxCameraWorker.

interface_class
alias of *labscript_devices.SpinnakerCamera.blacs_workers.Spinnaker_Camera*

class labscript_devices.SpinnakerCamera.blacs_workers.**Spinnaker_Camera**(*serial_number*)
Bases: `object`

Initialize Spinnaker API camera.

Serial number should be of string(?) type.

_decode_image_data(*img*)
Spinnaker image buffers require significant formatting. This returns what one would expect from a camera. configure_acquisition must be called first to set image format parameters.

abort_acquisition()

close()

configure_acquisition(*continuous=True*, *bufferCount=10*)

get_attribute(*name*, *stream_map=False*)
Return current values dictionary of attribute of the given name

get_attribute_names(*visibility*)

grab()
Grab and return single image during pre-configured acquisition.

grab_multiple(*n_images*, *images*)
Grab n_images into images array during buffered acquistion.

set_attribute(*name*, *value*, *stream_map=False*)

set_attributes(*attr_dict*)

set_stream_attribute(*name*, *value*)

snap()
Acquire a single image and return it

stop_acquisition()

trigger()
Execute software trigger

2.3.5 Andor Solis Cameras

A labscript device for controlling Andor scientific cameras via the Andor SDK3 interface.

Presently, this device is hard-coded for use with the iXon camera. Minor modifications can allow use with other Andor cameras, so long as they are compatible with the Andor SDK3 library.

---

labscript_devices.AndorSolis.labscript_devices

---

labscript_devices.AndorSolis.blacs_tabs

---

labscript_devices.AndorSolis.blacs_workers

Installation

The Andor SDK is available from Andor as a paid product (typically purchased with the camera). It must be installed with the SDK directory added to the system path.

Detailed Documentation

class labscript_devices.AndorSolis.labscript_devices.**AndorSolis**(*name*, *parent_device*, *connection*, *serial_number*, *orientation=None*, *pixel_size=[1.0, 1.0]*, *magnification=1.0*, *trigger_edge_type='rising'*, *trigger_duration=None*, *minimum_recovery_time=0.0*, *camera_attributes=None*, *manual_mode_camera_attributes=None*, *stop_acquisition_timeout=5.0*, *exception_on_failed_shot=True*, *saved_attribute_visibility_level='intermediate'*, *mock=False*, ***kwargs*)

Bases: *labscript_devices.IMAQdxCamera.labscript_devices.IMAQdxCamera*

A camera to be controlled using NI IMAQdx and triggered with a digital edge.

Parameters

- **name** (*str*) – device name
- **parent_device** (*IntermediateDevice*) – Device with digital outputs to be used to trigger acquisition
- **connection** (*str*) – Name of digital output port on parent device.

- **serial_number** (*str or int*) – string or integer (integer allows entering a hex literal) of the camera's serial number. This will be used to idenitfy the camera.
- **orientation** (*str, optional*) – <name> Description of the camera's location or orientation. This will be used to determine the location in the shot file where the images will be saved. If not given, the device name will be used instead.
- **pixel_size** (*[float,float], optional*) – `[1.0, 1.0]` The x and y size of the pixels in micrometers. This can be used in setting the scale in the blacs image display as well as extracted in lyse for analysis.
- **magnification** (*float, optional*) – `1.0` Imaging system magnification.
- **trigger_edge_type** (*str*) – `'rising'` The direction of the desired edges to be generated on the parent devices's digital output used for triggering. Must be 'rising' or 'falling'. Note that this only determines the edges created on the parent device, it does not program the camera to expect this type of edge. If required, one must configure the camera separately via `camera_attributes` to ensure it expects the type of edge being generated. Default: `'rising'`
- **trigger_duration** (*float or None*) – `None` Duration of digital pulses to be generated by the parent device. This can also be specified as an argument to `expose()` - the value given here will be used only if nothing is passed to `expose()`.
- **minimum_recovery_time** (*float*) – `0` Minimum time between frames. This will be used for error checking during compilation.
- **camera_attributes** (*dict, optional*) – Dictionary of camera attribute names and values to be programmed into the camera. The meaning of these attributes is model-specific. Attributes will be programmed in the order they appear in this dictionary. This can be important as some attributes may not be settable unless another attrbiute has been set first. After adding this device to your connection table, a dictionary of the camera's default attributes can be obtained from the BLACS tab, appropriate for copying and pasting into your connection table to customise the ones you are interested in.
- **manual_mode_camera_attributes** (*dict, optional*) – Dictionary of attributes that will be programmed into the camera during manual mode, that differ from their values in `camera_attributes`. This can be useful for example, to have software triggering during manual mode (allowing the acquisition of frames from the BLACS manual mode interface) but hardware triggering during buffered runs. Any attributes in this dictionary must also be present in `camera_attributes`.
- **stop_acquisition_timeout** (*float*) – `5.0` How long, in seconds, to wait during `transition_to_buffered` for the acquisition of images to complete before giving up. Whilst all triggers should have been received, this can be used to allow for slow image download time.
- **exception_on_failed_shot** (*bool*) – `True`. If acquisition does not complete within the given timeout after the end of a shot, whether to raise an exception. If False, instead prints a warning to stderr (visible in the terminal output pane in the BLACS tab), saves the images acquired so far, and continues. In the case of such a 'failed shot', the HDF5 attribute f['images'][orientation/name].attrs['failed_shot'] will be set to `True` (otherwise it is set to `False`). This attribute is acessible in the lyse dataframe as `df[orientation/name, 'failed_shot']`.
- **saved_attribute_visibility_level** (*str or None*) – 'intermediate' The detail level of the camera attributes saved to the HDF5 file at the end of each shot. If None, no attributes will be saved. Must be one of `'simple'`, `'intermediate'`, `'advanced'`, or `None`. If `None`, no attributes will be saved.

- **mock** (*bool, optional*) – False For testing purpses, simulate a camera with fake data instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (TriggerableDevice).

description = 'Andor scientific camera'
Brief description of the device.

class labscript_devices.AndorSolis.blacs_tabs.**AndorSolisTab**(*notebook*, *settings*, *restart=False*)
Bases: *labscript_devices.IMAQdxCamera.blacs_tabs.IMAQdxCameraTab*

worker_class = 'labscript_devices.AndorSolis.blacs_workers.AndorSolisWorker'

class labscript_devices.AndorSolis.blacs_workers.**AndorCamera**
Bases: object

_decode_image_data(*img*)

abort_acquisition()

close()

configure_acquisition(*continuous=False*, *bufferCount=None*)

get_attribute(*name*)

get_attribute_names(*visibility_level*, *writeable_only=True*)

grab()
Grab last/single image

grab_multiple(*n_images*, *images*, *waitForNextBuffer=True*)
Grab n_images into images array during buffered acquistion.

set_attribute(*name*, *value*)

set_attributes(*attr_dict*)

snap()
Acquire a single image and return it

stop_acquisition()

class labscript_devices.AndorSolis.blacs_workers.**AndorSolisWorker**(**args*, ***kwargs*)
Bases: *labscript_devices.IMAQdxCamera.blacs_workers.IMAQdxCameraWorker*

get_attributes_as_dict(*visibility_level*)
Return a dict of the attributes of the camera for the given visibility level

get_camera()
Andor cameras may not be specified by serial numbers

interface_class
alias of *labscript_devices.AndorSolis.blacs_workers.AndorCamera*

2.4 Frequency Sources

These devices cover various frequency sources that provide either hardware-timed frequency, amplitude, or phase updates or static frequency outputs.

2.4.1 Novatech DDS 9m

Labscript device for control of the Novatech DDS9m synthesizer. With minor modifications, it can also control the Novatech 409B DDS.

Detailed Documentation

class labscript_devices.NovaTechDDS9M.**NovaTechDDS9M**(*name*, *parent_device*, *com_port=''*, *baud_rate=115200*, *default_baud_rate=None*, *update_mode='synchronous'*, *synchronous_first_line_repeat=False*, *phase_mode='continuous'*, ***kwargs*)

Bases: labscript.labscript.IntermediateDevice

This class is initilzed with the key word argument 'update_mode' – synchronous or asynchronous 'baud_rate', – operating baud rate 'default_baud_rate' – assumed baud rate at startup

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (ClockLine) – Parent ClockLine device.

description = 'NT-DDS9M'
Brief description of the device.

allowed_children = [<class 'labscript.labscript.DDS'>, <class 'labscript.labscript.StaticDDS'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 9990

minimum_clock_high_time = 1e-06

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

get_default_unit_conversion_classes(*device*)
Child devices call this during their __init__ (with themselves as the argument) to check if there are certain unit calibration classes that they should apply to their outputs, if the user has not otherwise specified a calibration class

quantise_freq(*data*, *device*)

quantise_phase(*data*, *device*)

quantise_amp(*data*, *device*)

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.NovaTechDDS9M.**NovatechDDS9MTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

labscript_device_class_name = 'NovaTechDDS9M'

class labscript_devices.NovaTechDDS9M.**NovatechDDS9mWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

init()

check_connection()
Sends non-command and tests for correct response, returns True if connection appears to be working correctly, else returns False

check_remote_values()

program_manual(*front_panel_values*)

program_static(*channel*, *type*, *value*)

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

abort_transition_to_buffered()

abort_buffered()

transition_to_manual(*abort=False*)

shutdown()

class labscript_devices.NovaTechDDS9M.**RunviewerClass**(*path*, *device*)

Bases: `object`

get_traces(*add_trace*, *clock=None*)

labscript_device_class_name = 'NovaTechDDS9M'

2.4.2 QuickSyn FSW-0010 Synthesizer

A labscript device that controlls the NI Quicksyn FSW-0010 Microwave Synthesizer (formerly PhaseMatrix).

Detailed Documentation

class labscript_devices.PhaseMatrixQuickSyn.**QuickSynDDS**(*name*, *parent_device*, *connection*, *freq_limits=None*, *freq_conv_class=None*, *freq_conv_params={}*)

Bases: `labscript.labscript.StaticDDS`

A StaticDDS that supports only frequency control, with no phase or amplitude control.

Instantiates a Static DDS quantity.

Parameters

- **name** (*str*) – python variable for the object created.
- **parent_device** (`IntermediateDevice`) – Device this output is connected to.
- **connection** (*str*) – Output of parent device this DDS is connected to.
- **digital_gate** (*dict, optional*) – Configures a digital output to use as an enable/disable gate for the output. Should contain keys `'device'` and `'connection'` with arguments for the `parent_device` and `connection` for instantiating the `DigitalOut`.
- **freq_limits** (*tuple, optional*) – `(lower, upper)` limits for the frequency of the output

- **freq_conv_class** (labscript_utils:labscript_utils.unitconversions, optional) – Unit conversion class for the frequency of the output.
- **freq_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the frequency of the output.
- **amp_limits** (*tuple, optional*) – (lower, upper) limits for the amplitude of the output
- **amp_conv_class** (labscript_utils:labscript_utils.unitconversions, optional) – Unit conversion class for the amplitude of the output.
- **amp_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the amplitude of the output.
- **phase_limits** (*tuple, optional*) – (lower, upper) limits for the phase of the output
- **phase_conv_class** (labscript_utils:labscript_utils.unitconversions, optional) – Unit conversion class for the phase of the output.
- **phase_conv_params** (*dict, optional*) – Keyword arguments passed to the unit conversion class for the phase of the output.
- **call_parents_add_device** (*bool, optional*) – Have the parent device run its add_device method.
- ****kwargs** – Keyword arguments passed to Device.__init__().

description = 'PhaseMatrix QuickSyn DDS'
Brief description of the device.

allowed_children = [<class 'labscript.labscript.StaticAnalogQuantity'>, <class 'labscript.labscript.StaticDigitalOut'>]
Defines types of devices that are allowed to be children of this device.

Type list

generation = 2

setamp(*value*, *units=None*)
Set the static amplitude of the output.

Parameters

- **value** (*float*) – Amplitude to set to.
- **units** – Units that the value is defined in.

setphase(*value*, *units=None*)
Set the static phase of the output.

Parameters

- **value** (*float*) – Phase to set to.

- **units** – Units that the value is defined in.

enable()
overridden from StaticDDS so as not to provide time resolution - output can be enabled or disabled only at the start of the shot

disable()
overridden from StaticDDS so as not to provide time resolution - output can be enabled or disabled only at the start of the shot

class labscript_devices.PhaseMatrixQuickSyn.**PhaseMatrixQuickSyn**(*name*, *com_port*)
Bases: labscript.labscript.Device

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.
- **added_properties** (*dict, optional*) –
- **gui** –
- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

description = 'QuickSyn Frequency Synthesiser'
Brief description of the device.

allowed_children = [<class 'labscript_devices.PhaseMatrixQuickSyn.QuickSynDDS'>]
Defines types of devices that are allowed to be children of this device.

Type list

generation = 0

quantise_freq(*data*, *device*)

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (`h5py.File`) – Handle to shot file.

class labscript_devices.PhaseMatrixQuickSyn.**PhaseMatrixQuickSynTab**(*notebook*, *settings*, *restart=False*)
Bases: `blacs.device_base_class.DeviceTab`

initialise_GUI()

status_monitor(**args*, ***kwargs*)

update_reference_out(**args*, ***kwargs*)

update_blanking(**args*, ***kwargs*)

update_lock_recovery(**args*, ***kwargs*)

labscript_device_class_name = 'PhaseMatrixQuickSyn'

class labscript_devices.PhaseMatrixQuickSyn.**QuickSynWorker**(**args*, ***kwargs*)
Bases: `blacs.tab_base_classes.Worker`

init()

check_remote_values()

check_status()

program_manual(*front_panel_values*)

update_reference_out(*value*)

update_blanking(*value*)

update_lock_recovery(*value*)

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

abort_transition_to_buffered()

abort_buffered()

transition_to_manual(*abort=False*)

shutdown()

2.5 Miscellaneous

These devices cover other types of devices.

2.5.1 Alazar Tech Board

A labscript device class for data acquisition boards made by Alazar Technologies Inc (ATS).

Installation

This device requires the atsapi.py wrapper. It should be installed into site-packages or kept in the local directory.

It also uses the tqdm progress bar, which is not a standard dependency for the labscript-suite.

Detailed Documentation

2.5.2 Light Crafter DMD

This device allows for control of Light Crafter development module boards. It is currently hard-coded to work with a DLPC300 with a fixed DLP 0.3 WVGA resolution. Extenstion to other models involves subclassing and altering relevant class attributes.

Detailed Documentation

labscript_devices.LightCrafterDMD.**arr_to_bmp**(*arr*)
Convert array to 1 bit BMP, white wherever the array is nonzero, and return a bytestring of the BMP data

class labscript_devices.LightCrafterDMD.**ImageSet**(*name*, *parent_device*, *connection='Mirror'*)
Bases: labscript.labscript.Output

Instantiate an Output.

Parameters

- **name** (*str*) – python variable name to assign the Output to.
- **parent_device** (IntermediateDevice) – Parent device the output is connected to.
- **connection** (*str*) – Channel of parent device output is connected to.

- **limits** (*tuple, optional*) – (min,max) allowed for the output.
- **unit_conversion_class** (labscript_utils:labscript_utils.unitconversions, optional) – Unit concersion class to use for the output.
- **unit_conversion_parameters** (*dict, optional*) – Dictonary or kwargs to pass to the unit conversion class.
- **default_value** (*float, optional*) – Default value of the output if no output is commanded.
- ****kwargs** – Passed to Device.__init__().

Raises **LabscriptError** – Limits tuple is invalid or unit conversion class units don't line up.

description = 'A set of images to be displayed on an SLM or DMD'
Brief description of the device.

width = 608

height = 684

default_value
A black image.

Raw bitmap data hidden from docs.

Type bytes

set_array(*t*, *arr*)

set_image(*t*, *path=None*, *raw=None*)
set an image at the given time, either by a filepath to a bmp file, or by a bytestring of bmp data

expand_timeseries(*all_times*)
We have to override the usual expand_timeseries, as it sees strings as iterables that need flattening! Luckily for us, we should only ever have individual data points, as we won't be ramping or anything, so this function is a lot simpler than the original, as we have more information about the output.

Not 100% sure that this is enough to cover ramps on other devices sharing the clock, come here if there are issues!

class labscript_devices.LightCrafterDMD.**LightCrafterDMD**(*name*, *parent_device*, *server='192.168.1.100'*, *port=21845*)
Bases: labscript.labscript.IntermediateDevice

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device

- **parent_device** (ClockLine) – Parent ClockLine device.

description = 'LightCrafter DMD controller'
Brief description of the device.

allowed_children = [<class 'labscript_devices.LightCrafterDMD.ImageSet'>]
Defines types of devices that are allowed to be children of this device.

Type list

max_instructions = 96

clock_limit = 4000

width = 608

height = 684

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.LightCrafterDMD.**LightCrafterTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

width = 608

height = 684

initialise_GUI()

initialise_workers()

labscript_device_class_name = 'LightCrafterDMD'

class labscript_devices.LightCrafterDMD.**LightCrafterWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

command = {'advance_pattern_sequence': b'\x04\x03', 'display_mode': b'\x01\x01', 'display_pattern': b'\x04\x05', 'pattern_definition': b'\x04\x01', 'sequence_setting': b'\x04\x00', 'start_pattern_sequence': b'\x04\x02', 'static_image': b'\x01\x05', 'version': b'\x01\x00'}

send_packet_type = {'read': b'\x04', 'write': b'\x02'}

receive_packet_type = {b'\x00': 'System Busy', b'\x01': 'Error', b'\x03': 'Write response', b'\x05': 'Read response'}

flag = {'beginning': b'\x01', 'complete': b'\x00', 'end': b'\x03', 'intermediate': b'\x02'}

error_messages = {b'\x01': 'Command execution failed with unknown error', b'\x02': 'Invalid command', b'\x03': 'Invalid parameter', b'\x04': 'Out of memory resource', b'\x05': 'Hardware device failure', b'\x06': 'Hardware busy', b'\x07': 'Not Initialized (any of the preconditions for the command is not met', b'\x08': 'Some object referred by the command is not found. For example, a solution name was not found', b'\t': 'Checksum error', b'\n': 'Packet format error due to insufficient or larger than expected payload size', b'\x0b': 'Command continuation error due to incorrect continuation flag'}

display_mode = {'pattern': b'\x04', 'static': b'\x00'}

init()

send(*type*, *command*, *data*)

_receive()

receive()

program_manual(*values*)

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

abort()

abort_buffered()

abort_transition_to_buffered()

shutdown()

2.5.3 Tektronix Oscilloscope

A device for controlling Tektronix oscilloscopes using the standard VISA interface.

labscript_devices.TekScope.labscript_devices
labscript_devices.TekScope.blacs_tabs
labscript_devices.TekScope.blacs_workers
labscript_devices.TekScope.TekScope

Installation

This wrapper requires PyVISA and a compatible VISA installation. Free versions are provided by NI and Keysight (NI preferred if already using NI DAQs).

Detailed Documentation

class labscript_devices.TekScope.labscript_devices.**TekScope**(*name*, *addr*, *termination='\n'*, *preamble_string='WFMP'*, *timeout=5*, *int16=False*, ***kwargs*)

Bases: labscript.labscript.Device

A labscript_device for Tektronix oscilloscopes using a visa interface. connection_table_properties (set once) termination: character signalling end of response preamble_string: base command for waveform preamble ('WFMO' or 'WFMP')

device_properties (set per shot) timeout: in seconds for response to queries over visa interface int16: download waveform pts as 16 bit integers (returns 1/2 as many pts)

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.

- **added_properties** (*dict, optional*) –
- **gui** –
- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

description = 'Tekstronix oscilloscope'
Brief description of the device.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.TekScope.blacs_tabs.**TekScopeTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

initialise_workers()

class labscript_devices.TekScope.blacs_workers.**TekScopeWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort()

abort_buffered()

abort_transition_to_buffered()

init()

program_manual(*values*)

transition_to_buffered(*device_name*, *h5file*, *front_panel_values*, *refresh*)

transition_to_manual()

class labscript_devices.TekScope.TekScope.**TekScope**(*addr='USB?*::INSTR'*, *timeout=1*, *termination='\n'*)
Bases: object

channels(*all=True*)
Return a dictionary of the channels supported by this scope and whether they are currently displayed. Includes REF and MATH channels if applicable. If "all" is False, only visible channels are returned

close()

get_acquire_state()

get_screenshot(*verbose=False*)

lock(*verbose=False*)

save_screenshot(*filepath*, *verbose=False*)

set_acquire_state(*running=True*)

set_date_time(*verbose=False*)

unlock(*verbose=False*)

waveform(*channel='CH1'*, *preamble_string='WFMO'*, *int16=False*)
Download the waveform of the specified channel from the oscilloscope. All acquired points are downloaded, as set by the 'Record length' setting in the 'Acquisition' menu. A dictionary of waveform formatting parameters is returned in addition to the times and values.

2.5.4 Zaber Stage Controller

Device for controlling a Zaber translation stage.

labscript_devices.ZaberStageController.labscript_devices

labscript_devices.ZaberStageController.blacs_tabs

labscript_devices.ZaberStageController.blacs_workers

labscript_devices.ZaberStageController.utils

Detailed Documentation

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStage**(**args*, *limits=None*, ***kwargs*)
Bases: labscript.labscript.StaticAnalogQuantity

Static Analog output device for controlling the position of a Zaber stage. Can be added as a child device of ZaberStageController. Subclasses for specific models already have model-specific limits set for their values, but you may further restrict these by setting the keyword argument limits=

Parameters

- ***args** – Arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).
- **limits** (*tuple*) – a two-tuple (min, max) for the minimum and maximum allowed positions, in steps, that the device may be instructed to move to via a labscript experiment or the BLACS front panel. If None, the limits set as a class attribute will be used, which are set to the maximal positions allowed by the device if using one of the model-specific subclasses defined in this module, or is (0, inf) otherwise.
- ****kwargs** – Further keyword arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).

description = 'Zaber Stage'
Brief description of the device.

limits = (0, inf)

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStageController**(*name*, *com_port='COM1'*, *mock=False*, ***kwargs*)
Bases: labscript.labscript.IntermediateDevice

Device for controlling a number of Zaber stages connected to a serial port. Add stages as child devices, either by using one of them model-specific classes in this module, or the generic ZaberStage class.

Parameters

- **name** (*str*) – device name
- **com_port** (*str*) – 'COM1' Serial port for communication, i.e. 'COM1' etc on Windows or `'/dev/USBtty0' or similar on unix.
- **mock** (*bool, optional*) – False For testing purpses, simulate a device instead of communicating with actual hardware.
- ****kwargs** – Further keyword arguments to be passed to the __init__ method of the parent class (IntermediateDevice).

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript_devices.ZaberStageController.labscript_devices.ZaberStage'>]
Defines types of devices that are allowed to be children of this device.

Type list

generate_code(*hdf5_file*)

Generate hardware instructions for device and children, then save to h5 file.

Will recursively call `generate_code` for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStageTLS28M**(**args*, *limits=None*, ***kwargs*)

Bases: *labscript_devices.ZaberStageController.labscript_devices.ZaberStage*

Static Analog output device for controlling the position of a Zaber stage. Can be added as a child device of `ZaberStageController`. Subclasses for specific models already have model-specific limits set for their values, but you may further restrict these by setting the keyword argument limits=

Parameters

- ***args** – Arguments to be passed to the `__init__` method of the parent class (StaticAnalogQuantity).
- **limits** (*tuple*) – a two-tuple (min, max) for the minimum and maximum allowed positions, in steps, that the device may be instructed to move to via a labscript experiment or the BLACS front panel. If None, the limits set as a class attribute will be used, which are set to the maximal positions allowed by the device if using one of the model-specific subclasses defined in this module, or is (0, inf) otherwise.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (StaticAnalogQuantity).

description = 'Zaber Stage T-LS28-M'

Brief description of the device.

limits = (0, 282879)

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStageTLSR150D**(**args*, *limits=None*, ***kwargs*)

Bases: *labscript_devices.ZaberStageController.labscript_devices.ZaberStage*

Static Analog output device for controlling the position of a Zaber stage. Can be added as a child device of `ZaberStageController`. Subclasses for specific models already have model-specific limits set for their values, but you may further restrict these by setting the keyword argument limits=

Parameters

- ***args** – Arguments to be passed to the `__init__` method of the parent class (StaticAnalogQuantity).
- **limits** (*tuple*) – a two-tuple (min, max) for the minimum and maximum allowed positions, in steps, that the device may be instructed to move to via a labscript experiment or the BLACS front panel. If None, the limits set as a class attribute will be used, which are set to the maximal positions allowed by the device if using one of the model-specific subclasses defined in this module, or is (0, inf) otherwise.
- ****kwargs** – Further keyword arguments to be passed to the `__init__` method of the parent class (StaticAnalogQuantity).

description = 'Zaber Stage T-LSR150D'

Brief description of the device.

limits = (0, 76346)

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStageTLSR300B**(**args*, *limits=None*, ***kwargs*)
Bases: *labscript_devices.ZaberStageController.labscript_devices.ZaberStage*

Static Analog output device for controlling the position of a Zaber stage. Can be added as a child device of ZaberStageController. Subclasses for specific models already have model-specific limits set for their values, but you may further restrict these by setting the keyword argument limits=

Parameters

- ***args** – Arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).
- **limits** (*tuple*) – a two-tuple (min, max) for the minimum and maximum allowed positions, in steps, that the device may be instructed to move to via a labscript experiment or the BLACS front panel. If None, the limits set as a class attribute will be used, which are set to the maximal positions allowed by the device if using one of the model-specific subclasses defined in this module, or is (0, inf) otherwise.
- ****kwargs** – Further keyword arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).

description = 'Zaber Stage T-LSR150D'
Brief description of the device.

limits = (0, 607740)

class labscript_devices.ZaberStageController.labscript_devices.**ZaberStageTLSR300D**(**args*, *limits=None*, ***kwargs*)
Bases: *labscript_devices.ZaberStageController.labscript_devices.ZaberStage*

Static Analog output device for controlling the position of a Zaber stage. Can be added as a child device of ZaberStageController. Subclasses for specific models already have model-specific limits set for their values, but you may further restrict these by setting the keyword argument limits=

Parameters

- ***args** – Arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).
- **limits** (*tuple*) – a two-tuple (min, max) for the minimum and maximum allowed positions, in steps, that the device may be instructed to move to via a labscript experiment or the BLACS front panel. If None, the limits set as a class attribute will be used, which are set to the maximal positions allowed by the device if using one of the model-specific subclasses defined in this module, or is (0, inf) otherwise.
- ****kwargs** – Further keyword arguments to be passed to the __init__ method of the parent class (StaticAnalogQuantity).

description = 'Zaber Stage T-LSR300D'
Brief description of the device.

limits = (0, 151937)

class labscript_devices.ZaberStageController.blacs_tabs.**ZaberStageControllerTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

initialise_workers()

class labscript_devices.ZaberStageController.blacs_workers.**MockZaberInterface**(*com_port*)
Bases: object

close()

get_position(*device_number*)

move(*device_number*, *position*)

class labscript_devices.ZaberStageController.blacs_workers.**ZaberInterface**(*com_port*)
Bases: object

close()

get_position(*device_number*)

move(*device_number*, *position*)

class labscript_devices.ZaberStageController.blacs_workers.**ZaberWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

check_remote_values()

init()

program_manual(*values*)

shutdown()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

labscript_devices.ZaberStageController.utils.**get_device_number**(*connection_str*)
Return the integer device number from the connection string or raise ValueError if the connection string is not in the format "device <n>" with positive n.

2.6 Other

These devices provide dummy instruments for prototyping and testing purposes of the rest of the labscript_suite as well as the FunctionRunner device which can run arbitrary code post-shot.

2.6.1 Function Runner

A labscript device to run custom functions before, after, or during (not yet implemented) the experiment in software time.

labscript_devices.FunctionRunner.labscript_devices

labscript_devices.FunctionRunner.blacs_tabs

labscript_devices.FunctionRunner.blacs_workers

labscript_devices.FunctionRunner.utils

Detailed Documentation

class labscript_devices.FunctionRunner.labscript_devices.**FunctionRunner**(*name*, ***kwargs*)

Bases: labscript.labscript.Device

A labscript device to run custom functions before, after, or during (not yet implemented) the experiment in software time

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.
- **added_properties** (*dict, optional*) –
- **gui** –

- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

add_function(*t*, *function*, **args*, ***kwargs*)

Add a function to be run at time t. If t='start', then the function will run prior to the shot beginning, and if t='stop', it will run after the experiment has completed. Tip: use `start_order` and `stop_order` keyword arguments when instantiating this device to control the relative order that its 'start' and 'stop' functions run compared to the transition_to_manual and transition_to_buffered functions of other devices. Multiple functions added to run at the same time will be run in the order added. Running functions mid-shot in software time is yet to be implemented.

The function must have a call signature like the following:

def func(shot_context, t, ...): ...

When it is called, a ShotContext instance will be passed in as the first argument, and the time at which the function was requested to run as the second argument. The ShotContext instance will be the same for all calls for the same shot, so it can be used to store state for that shot (but not from one shot to the next), the same way you would use the 'self' argument of a method to store state in an instance. As an example, you might set shot_context.serial to be an open serial connection to a device during a function set to run at t='start', and refer back to it in subsequent functions to read and write data. Other than state stored in shot_context, the functions must be self-contained, containing any imports that they need.

This object has a number of attributes:

- self.globals: the shot globals
- self.h5_file: the filepath to the shot's HDF5 file
- self.device_name: the name of this FunctionRunner

If you want to save raw data to the HDF5 file at the end of a shot, the recommended place to do it is within the group 'data/<device_name>', for .. rubric:: Example

with h5py.File(self.h5_file, 'r+') as f: data_group = f['data'].create_group(self.device_name) # save datasets/attributes within this group

Or, if you are doing analysis and want to save results that will be accessible to lyse analysis routines in the usual way, you can instantiate a lyse.Run object and call Run.save_result() etc:

import lyse run = lyse.Run(shot_context.h5_file) run.save_result('x', 7)

The group that the results will be saved to, which is usually the filename of the lyse analysis routine, will instead be the device name of the FunctionRunner.

The use case for which this device was implemented was to update runmanager's globals immediately after a shot, based on measurement data, such that just-in-time compiled shots imme. This is done by calling the runmanager remote API from within a function to be run at the end of a shot, like so:

import runmanager.remote runmanager.remote.set_globals({'x': 7})

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.FunctionRunner.blacs_tabs.**FunctionRunnerTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_workers()

restore_builtin_save_data(*data*)
Restore builtin settings to be restored like whether the terminal is visible. Not to be overridden.

class labscript_devices.FunctionRunner.blacs_workers.**FunctionRunnerWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

abort_buffered()

abort_transition_to_buffered()

program_manual(*values*)

shutdown()

transition_to_buffered(*device_name*, *h5_file*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.FunctionRunner.blacs_workers.**ShotContext**(*h5_file*, *device_name*)
Bases: object

labscript_devices.FunctionRunner.blacs_workers.**deserialise_function_table**(*function_table*, *device_name*)

labscript_devices.FunctionRunner.utils.**deserialise_function**(*name*, *source*, *args*, *kwargs*, *__name__=None*, *__file__='<string>'*)
Deserialise a function that was serialised by serialise_function. Optional __name__ and __file__ arguments set those attributes in the namespace that the function will be defined.

labscript_devices.FunctionRunner.utils.**serialise_function**(*function*, **args*, ***kwargs*)
Serialise a function based on its source code, and serialise the additional args and kwargs that it will be called with. Raise an exception if the function signature does not begin with (shot_context, t) or if the additional args and kwargs are incompatible with the rest of the function signature

2.6.2 Dummy Pseudoclock

This device represents a dummy labscript device for purposes of testing BLACS and labscript. The device is a PseudoclockDevice, and can be the sole device in a connection table or experiment.

labscript_devices.DummyPseudoclock.labscript_devices

labscript_devices.DummyPseudoclock.blacs_tabs

labscript_devices.DummyPseudoclock.blacs_workers

labscript_devices.DummyPseudoclock.runviewer_parsers

Usage

```
from labscript import *

from labscript_devices.DummyPseudoclock.labscript_devices import DummyPseudoclock
from labscript_devices.DummyIntermediateDevice import DummyIntermediateDevice

DummyPseudoclock(name='dummy_clock',BLACS_connection='dummy')
DummyIntermediateDevice(name='dummy_device',BLACS_connection='dummy2',
                                            parent_device=dummy_clock.clockline)

start()
stop(1)
```

Detailed Documentation

class labscript_devices.DummyPseudoclock.labscript_devices.**DummyPseudoclock**(*name='dummy_pseudoclock'*, *BLACS_connection='dummy_connection'*, ***kwargs*)

Bases: labscript.labscript.PseudoclockDevice

Instantiates a pseudoclock device.

Parameters

- **name** (*str*) – python variable to assign to this device.
- **trigger_device** (DigitalOut) – Sets the parent triggering output. If None, this is considered the master pseudoclock.
- **trigger_connection** (*str, optional*) – Must be provided if trigger_device is provided. Specifies the channel of the parent device.
- ****kwargs** – Passed to TriggerableDevice.__init__().

add_device(*device*)
Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

allowed_children = [<class 'labscript_devices.DummyPseudoclock.labscript_devices._DummyPseudoclock'>]
Defines types of devices that are allowed to be children of this device.

Type list

clock_limit = 10000000.0

clock_resolution = 2.5e-08

property clockline

description = 'Dummy pseudoclock'
Brief description of the device.

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

max_instructions = 100000.0

property pseudoclock

trigger_delay = 3.5e-07

wait_delay = 2.5e-06

class labscript_devices.DummyPseudoclock.labscript_devices.**_DummyPseudoclock**(*name*, *pseudoclock_device*, *connection*, ***kwargs*)
Bases: labscript.labscript.Pseudoclock

Creates a Pseudoclock.

Parameters

- **name** (*str*) – python variable name to assign the device instance to.
- **pseudoclock_device** (PseudoclockDevice) – Parent pseudoclock device
- **connection** (*str*) – Connection on this device that links to parent
- ****kwargs** – Passed to Device().

add_device(*device*)

Adds a child device to this device.

Parameters **device** (Device) – Device to add.

Raises **LabscriptError** – If device is not an allowed child of this device.

class labscript_devices.DummyPseudoclock.blacs_tabs.**DummyPseudoclockTab**(*notebook*, *settings*, *restart=False*)

Bases: blacs.device_base_class.DeviceTab

initialise_workers()

start_run(**args*, ***kwargs*)

wait_until_done(**args*, ***kwargs*)

class labscript_devices.DummyPseudoclock.blacs_workers.**DummyPseudoclockWorker**(**args*, ***kwargs*)

Bases: blacs.tab_base_classes.Worker

abort_buffered()

check_if_done()

program_manual(*values*)

shutdown()

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual()

class labscript_devices.DummyPseudoclock.runviewer_parsers.**DummyPseudoclockParser**(*path*, *device*)

Bases: object

clock_resolution = 2.5e-08

get_traces(*add_trace*, *clock=None*)

trigger_delay = 3.5e-07

```
wait_delay = 2.5e-06
```

2.6.3 Dummy Intermediate Device

Overview

This file represents a dummy labscript device for purposes of testing BLACS and labscript. The device is a Intermediate Device, and can be attached to a pseudoclock in labscript in order to test the pseudoclock behaviour without needing a real Intermediate Device.

You can attach an arbitrary number of outputs to this device, however we currently only support outputs of type AnalogOut and DigitalOut. I would be easy to extend this is anyone needed further functionality.

Usage

```
from labscript import *

from labscript_devices.DummyPseudoclock.labscript_devices import DummyPseudoclock
from labscript_devices.DummyIntermediateDevice import DummyIntermediateDevice

DummyPseudoclock(name='dummy_clock',BLACS_connection='dummy')
DummyIntermediateDevice(name='dummy_device',BLACS_connection='dummy2',
                        parent_device=dummy_clock.clockline)

DigitalOut(name='do1',parent_device=dummy_device,connection='dummy_do1')
DigitalOut(name='do2',parent_device=dummy_device,connection='dummy_do2')

start()
stop(1)
```

Detailed Documentation

class labscript_devices.DummyIntermediateDevice.**DummyIntermediateDevice**(*name*, *parent_device*, *BLACS_connection='dummy_connection'*, ***kwargs*)

Bases: labscript.labscript.IntermediateDevice

Provides some error checking to ensure parent_device is a ClockLine.

Calls Device.__init__().

Parameters

- **name** (*str*) – python variable name to assign to device
- **parent_device** (ClockLine) – Parent ClockLine device.

description = 'Dummy IntermediateDevice'
Brief description of the device.

clock_limit = 1000000.0

allowed_children = [<class 'labscript.labscript.DigitalOut'>, <class 'labscript.labscript.AnalogOut'>]
Defines types of devices that are allowed to be children of this device.

Type list

generate_code(*hdf5_file*)
Generate hardware instructions for device and children, then save to h5 file.

Will recursively call generate_code for all children devices.

Parameters **hdf5_file** (h5py.File) – Handle to shot file.

class labscript_devices.DummyIntermediateDevice.**DummyIntermediateDeviceTab**(*notebook*, *settings*, *restart=False*)
Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

labscript_device_class_name = 'DummyIntermediateDevice'

class labscript_devices.DummyIntermediateDevice.**DummyIntermediateDeviceWorker**(**args*, ***kwargs*)
Bases: blacs.tab_base_classes.Worker

init()

program_manual(*front_panel_values*)

transition_to_buffered(*device_name*, *h5file*, *initial_values*, *fresh*)

transition_to_manual(*abort=False*)

abort_transition_to_buffered()

abort_buffered()

shutdown()

2.6.4 Test Device

A generic test device to aid in testing labscript infrastructure/functionality.

Detailed Documentation

class labscript_devices.test_device.**test_device**(*name*, *DoSomething=False*, ***kwargs*)
Bases: labscript.labscript.Device

Creates a Device.

Parameters

- **name** (*str*) – python variable name to assign this device to.
- **parent_device** (Device) – Parent of this device.
- **connection** (*str*) – Connection on this device that links to parent.
- **call_parents_add_device** (*bool, optional*) – Flag to command device to call its parent device's add_device when adding a device.
- **added_properties** (*dict, optional*) –
- **gui** –
- **worker** –
- **start_order** (*int, optional*) – Priority when starting, sorted with all devices.
- **stop_order** (*int, optional*) – Priority when stopping, sorted with all devices.
- ****kwargs** – Other options to pass to parent.

description = 'test device'
Brief description of the device.

class labscript_devices.test_device.**Tab**
Bases: object

labscript_device_class_name = 'test_device'

class labscript_devices.test_device.**Worker**
Bases: object

class labscript_devices.test_device.**Parser**
Bases: object

```
    labscript_device_class_name = 'test_device'
```

CHAPTER

THREE

USER DEVICES

Adding custom devices for use in the **labscript-suite** can be done using the `user_devices` mechanism. This mechanism provides a simple way to add support for a new device without directly interacting with the **labscript-devices** repository. This is particularly useful when using standard installations of labscript, using code that is proprietary in nature, or code that, while functional, is not mature enough for widespread dissemination.

This is done by adding the **labscript-device** code into the `userlib/user_devices` folder. Using the custom device in a **labscript** connection table is then done by:

```
from user_devices.MyCustomUserDevice.labscript_devices import MyCustomUserDevice
```

This import statement assumes your custom device follows the new device structure organization.

Note that both the `userlib` path and the `user_devices` folder name can be custom configured in the `labconfig.ini` file. The `user_devices` folder must be in the `userlib` path. If a different `user_devices` folder name is used, the import uses that folder name in place of `user_devices` in the above import statement.

Note that we highly encourage everyone that adds support for new hardware to consider making a pull request to **labscript-devices** so that it may be added to the mainline and more easily used by other groups.

3.1 3rd Party Devices

Below is a list of 3rd party devices developed by users of the **labscript-suite** that can be used via the `user_devices` mechanism described above. These repositories are not tested or maintained by the **labscript-suite** development team. As such, there is no guarantee they will work with current or future versions of the **labscript-suite**. They are also not guaranteed to be free of lab-specific implementation details that may prevent direct use in your apparatus. They are provided by users to benefit the community in supporting new and/or unusual devices, and can often serve as a good reference when developing your own devices. Please direct any questions regarding these repositories to their respective owners.

- NAQS Lab

- Vladan Vuletic Group Rb Lab, MIT

If you would like to add your repository to this list, please contact us or make a pull request.

CHAPTER

FOUR

EXAMPLE CONNECTION TABLES

An example connection table for the experiment described in[1]. This connection table makes extensive use of `user_devices`, by name of `naqslab_devices`.

```
from labscript import *
from naqslab_devices.PulseBlasterESRPro300.labscript_device import PulseBlasterESRPro300
from naqslab_devices.NovaTechDDS.labscript_device import NovaTech409B, NovaTech409B_AC
from labscript_devices.NI_DAQmx.models.NI_USB_6343 import NI_USB_6343
from naqslab_devices.SignalGenerator.Models import RS_SMA100B, SRS_SG386
from naqslab_devices import ScopeChannel, StaticFreqAmp
from naqslab_devices.KeysightXSeries.labscript_device import KeysightXScope
#from labscript_devices.PylonCamera.labscript_devices import PylonCamera
from naqslab_devices.KeysightDCSupply.labscript_device import KeysightDCSupply
from naqslab_devices.SR865.labscript_device import SR865

PulseBlasterESRPro300(name='pulseblaster_0', board_number=0, programming_scheme='pb_start/BRANCH')
ClockLine(name='pulseblaster_0_clockline_fast', pseudoclock=pulseblaster_0.pseudoclock, connection='flag 0')
ClockLine(name='pulseblaster_0_clockline_slow', pseudoclock=pulseblaster_0.pseudoclock, connection='flag 1')

NI_USB_6343(name='ni_6343', parent_device=pulseblaster_0_clockline_fast,
            clock_terminal='/ni_usb_6343/PFI0',
            MAX_name='ni_usb_6343',
            acquisition_rate = 243e3, # 500 kS/s max aggregate)
            stop_order = -1) #as clocking device, ensure it transitions first

NovaTech409B(name='novatech_static', com_port="com4", baud_rate = 115200,
```

(continues on next page)

[1] D. H. Meyer, Z. A. Castillo, K. C. Cox, and P. D. Kunz, J. Phys B, **53** 034001 (2020) https://iopscience.iop.org/article/10.1088/1361-6455/ab6051

(continued from previous page)

```
                phase_mode='aligned',ext_clk=True, clk_freq=100, clk_mult=5)
NovaTech409B_AC(name='novatech', parent_device=pulseblaster_0_clockline_slow,
                com_port="com3", update_mode='asynchronous', phase_mode='aligned',
                baud_rate = 115200, ext_clk=True, clk_freq=100, clk_mult=5)

# using NI-MAX alias instead of full VISA name
RS_SMA100B(name='SMA100B', VISA_name='SMA100B')
RS_SMA100B(name='SMA100B2', VISA_name='SMA100B-2')
SRS_SG386(name='SG386', VISA_name='SG386-6181I', output='RF', mod_type='Sweep')

# call the scope, use NI-MAX alias instead of full name
KeysightXScope(name='Scope',VISA_name='DSOX3024T',
        trigger_device=pulseblaster_0.direct_outputs,trigger_connection='flag 3',
        num_AI=4,DI=False)
ScopeChannel('Heterodyne',Scope,'Channel 1')
#ScopeChannel('Absorption',Scope,'Channel 2')
#ScopeChannel('Modulation',Scope,'Channel 4')

# DC Supplies
KeysightDCSupply(name='DCSupply',VISA_name='E3640A',
                 range='HIGH',volt_limits=(0,20),current_limits=(0,1))
StaticAnalogOut('DCBias_Gnd',DCSupply,'channel 0')
KeysightDCSupply(name='DCSupply2',VISA_name='E3644A',
                range='HIGH',volt_limits=(0,20),current_limits=(0,1))
StaticAnalogOut('DCBias_Sig',DCSupply2,'channel 0')

# Lock-In Amplifier
SR865(name='LockIn',VISA_name='SR865')

# Define Cameras
# note that Basler cameras can overlap frames if
# second exposure does not end before frame transfer of first finishes

'''
PylonCamera('CCD_2',parent_device=pulseblaster_0.direct_outputs,connection='flag 6',
            serial_number=21646179,
            mock=False,
```

(continues on next page)

(continued from previous page)

```
            camera_attributes={'ExposureTime':9000,
                               'ExposureMode':'Timed',
                               'Gain':0.0,
                               'ExposureAuto':'Off',
                               'GainAuto':'Off',
                               'PixelFormat':'Mono12',
                               'Gamma':1.0,
                               'BlackLevel':0,
                               'TriggerSource':'Line1',
                               'ShutterMode':'Global',
                               'TriggerMode':'On'},
            manual_mode_camera_attributes={'TriggerSource':'Software',
                                           'TriggerMode':'Off'})
'''
# Define the Wait Monitor for the AC-Line Triggering
# note that connections used here cannot be used elsewhere
# 'connection' needs to be physically connected to 'acquisition_connection'
# for M-Series DAQs, ctr0 gate is on PFI9
WaitMonitor(name='wait_monitor', parent_device=ni_6343,
            connection='port0/line0', acquisition_device=ni_6343,
            acquisition_connection='ctr0', timeout_device=ni_6343,
            timeout_connection='PFI1')

DigitalOut( 'AC_trigger_arm', pulseblaster_0.direct_outputs, 'flag 2')

# define the PB digital outputs
DigitalOut( 'probe_AOM_enable', pulseblaster_0.direct_outputs, 'flag 4')
DigitalOut( 'LO_AOM_enable', pulseblaster_0.direct_outputs, 'flag 5')

# short pulse control channels
DigitalOut(  'bit21', pulseblaster_0.direct_outputs, 'flag 21')
DigitalOut(  'bit22', pulseblaster_0.direct_outputs, 'flag 22')
DigitalOut(  'bit23', pulseblaster_0.direct_outputs, 'flag 23')

AnalogOut( 'ProbeAmpLock', ni_6343, 'ao0')
AnalogOut( 'LOAmpLock', ni_6343, 'ao1')
AnalogOut( 'blueSweep', ni_6343, 'ao2')
```

(continues on next page)

(continued from previous page)

```
AnalogOut( 'MW_Phase', ni_6343, 'ao3')

AnalogIn( 'Homodyne', ni_6343, 'ai0')
AnalogIn( 'AI1', ni_6343, 'ai1')
AnalogIn( 'LockInX', ni_6343, 'ai2')
AnalogIn( 'LockInY', ni_6343, 'ai3')

# this dummy line necessary to balance the digital out for the wait monitor
DigitalOut( 'P0_1', ni_6343, 'port0/line1')

StaticDDS( 'Probe_EOM', novatech_static, 'channel 0')
StaticDDS( 'Probe_AOM', novatech_static, 'channel 1')
StaticDDS( 'LO_AOM', novatech_static, 'channel 2')
StaticDDS( 'LO', novatech_static, 'channel 3')

DDS( 'Probe_BN', novatech, 'channel 0')
DDS( 'dds1', novatech, 'channel 1')
StaticDDS( 'SAS_Mod', novatech, 'channel 2')
StaticDDS( 'SAS_LO', novatech, 'channel 3')

StaticFreqAmp( 'uWaves', SMA100B, 'channel 0', freq_limits=(8e-6,20), amp_limits=(-145,35))
StaticFreqAmp( 'uWavesLO', SMA100B2, 'channel 0', freq_limits=(8e-6,20), amp_limits=(-145,35))
StaticFreqAmp( 'blueEOM', SG386, 'channel 0', freq_limits=(1,6.075e3), amp_limits=(-110,16.5))

start()

stop(1)
```

4.1 References

CHAPTER

FIVE

HOW TO ADD A DEVICE

Adding a **labscript-device** involves implementing interfaces for your hardware to different protions of the **labscript-suite**. Namely, you must provide

- A `labscript_device` that takes **labscript** high-level commands and turns them into instructions that are saved to the shot h5 file.
- A `BLACS_worker` that handles communication with the hardware, in particular interpreting the instructions from the shot h5 file into the necessary hardware commands to configure the device.
- A `BLACS_tab` which provides a graphical interface to control the instrument via **BLACS**

Though not strictly required, you should also consider providing a `runviewer_parser`, which can read the h5 instructions and produce the appropriate shot timings that would occur. This interface is only used in **runviewer**.

5.1 General Strategy

As a general rule, it is best to model new hardware implementations off of a currently implemented device that has similar functionality. If the functionality is similar enough, it may even be possible to simply sub-class the currently implemented device, which is likely preferrable.

Barring the above simple solution, one must work from scratch. It is best to begin by determining the **labscript** device class to inherit from: `Psuedoclock`, `Device`, `IntermediateDevice`. The first is for implementing Psuedoclock devices, the second is for generic devices that are not hardware timed by a pseudoclock, and the last is for hardware timed devices that are connected to another device controlled via labscript.

The `labscript_device` implements general configuration parameters, many of which are passed to the `BLACS_worker`. It also implements the `generate_code` method which converts **labscript** high-level instructions and saves them to the h5 file.

The `BLACS_tab` defines the GUI widgets that control the device. This typically takes the form of using standard widgets provided by **labscript** for controlling **labscript** output primitives (ie `AnalogOut`, `DigitalOut`, `DDS`, etc). This configuration is done in the `initialiseGUI` method. This also specifies which BLACS workers to use and provides necessary instantiation arguments.

The BLACS_worker handles communication with the hardware itself and often represents the bulk of the work required to implement a new labscript device. In general, it should provide five different methods:

- init: This method initialises communications with the device. Not to be confused with the standard python class __init__ method.
- program_manual: This method allows for user control of the device via the BLACS_tab, setting outputs to the values set in the BLACS_tab widgets.
- check_remote_values: This method reads the current settings of the device, updating the BLACS_tab widgets to reflect these values.
- transition_to_buffered: This method transitions the device to buffered shot mode, reading the shot h5 file and taking the saved instructions from labscript_device.generate_code and sending the appropriate commands to the hardware.
- transition_to_manual: This method transitions the device from buffered to manual mode. It does any necessary configuration to take the device out of buffered mode and is used to read any measurements and save them to the shot h5 file as results.

The runviewer_parser takes shot h5 files, reads the saved instructions, and allows you to view them in **runviewer** in order to visualise experiment timing.

5.2 Code Organization

There are currently two supported file organization styles for a labscript-device.

The old style has the labscript_device, BLACS_tab, BLACS_worker, and runviewer_parser all in the same file, which typically has the same name as the labscript_device class name.

The new style allows for arbitrary code organization, but typically has a folder named after the labscript_device with each device component in a different file (ie labscript_devices.py, BLACS_workers.py, etc). With this style, the folder requires an __init__.py file (which can be empty) as well as a register_classes.py file. This file imports labscript_utils.device_registry via

```
from labscript_devices import register_classes
```

This function informs **labscript** where to find the necessary classes during import. An example for the NI_DAQmx device is

```
register_classes(
        'NI_DAQmx',
        BLACS_tab='labscript_devices.NI_DAQmx.blacs_tabs.NI_DAQmxTab',
        runviewer_parser='labscript_devices.NI_DAQmx.runviewer_parsers.NI_DAQmxParser',
)
```

5.3 Contributions to labscript-devices

If you decide to implement a labscript-device for controlling new hardware, we highly encourage you to consider making a pull-request to the **labscript-devices** repository in order to add your work to the **labscript-suite**. Increasing the list of supported devices is an important way for the **labscript-suite** to continue to grow, allowing new users to more quickly get up and running with hardware they may already have.

CHAPTER

SIX

LABSCRIPT SUITE COMPONENTS

The *labscript suite* is modular by design, and is comprised of:

Table 1: Python libraries

	labscript — Expressive composition of hardware-timed experiments
	labscript-devices — Plugin architecture for controlling experiment hardware
	labscript-utils — Shared modules used by the *labscript suite*

Table 2: Graphical applications

	runmanager — Graphical and remote interface to parameterized experiments
	blacs — Graphical interface to scientific instruments and experiment supervision
	lyse — Online analysis of live experiment data
	runviewer — Visualize hardware-timed experiment instructions

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Z