labscript-devices

Release 3.1.0.dev118+g999107d

labscript suite contributors

Feb 05, 2022

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

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

—

PulseBlaster_No_DDS

PulseBlasterUSB

PulseBlaster_SP2_24_100_32k

PulseBlasterESRPro200

PulseBlasterESRPro500

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

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

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
```

(continues on next page)

(continued from previous page)

```
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.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*)

2.1.6 RFblaster

Another pseudoclock-cable labscript device.

Detailed Documentation

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

labscript_devices.NI_DAQmx.models.NI_PCI_6534

labscript_devices.NI_DAQmx.models.NI_PCI_6713

labscript_devices.NI_DAQmx.models.NI_PCI_6733

labscript_devices.NI_DAQmx.models.NI_PCI_DIO_32HS

labscript_devices.NI_DAQmx.models.NI_PCIe_6343

labscript_devices.NI_DAQmx.models.NI_PCIe_6363

labscript_devices.NI_DAQmx.models.NI_PCIe_6738

labscript_devices.NI_DAQmx.models.NI_PXI_6733

labscript_devices.NI_DAQmx.models.NI_PXIe_4499

labscript_devices.NI_DAQmx.models.NI_PXIe_6361

labscript_devices.NI_DAQmx.models.NI_PXIe_6363

labscript_devices.NI_DAQmx.models.NI_PXIe_6535

labscript_devices.NI_DAQmx.models.NI_PXIe_6738

labscript_devices.NI_DAQmx.models.NI_USB_6008

labscript_devices.NI_DAQmx.models.NI_USB_6229

labscript_devices.NI_DAQmx.models.NI_USB_6343

labscript_devices.NI_DAQmx.models.NI_USB_6363

labscript_devices.NI_DAQmx.models.NI_USB_6366

labscript_devices.NI_DAQmx.models.generate_subclasses

labscript_devices.NI_DAQmx.models.get_capabilities

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',
                        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')
```

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.

Detailed Documentation

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.

—

Installation

Usage

Detailed Documentation

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.

—

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
               'ExposureMode':'Timed',
               'ExposureAuto':'Off',
               'GainAuto':'Off',
               'PixelFormat':'Mono12Packed',
               'Gamma':1.0,
               'BlackLevelRaw':0,
               'TriggerSource':'Line 1',
```

(continues on next page)

(continued from previous page)

```
                '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')

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

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*.

—

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,
                                    'absValue':1},
```

(continues on next page)

(continued from previous page)

```
                                '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')


stop(1)
```

Detailed Documentation

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.

—

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,
```

(continues on next page)

(continued from previous page)

```
   'AnalogControl::BlackLevel': 0.0,
   'AnalogControl::GammaEnabled': False,
   'AnalogControl::SharpnessEnabled': False,
   'ImageFormatControl::Width': 1008,
   '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

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.

—

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

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

2.4.2 QuickSyn FSW-0010 Synthesizer

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

Detailed Documentation

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

2.5.3 Tektronix Oscilloscope

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

labscript_devices.TekScope.blacs_tabs

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

initialise_GUI()

initialise_workers()

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.blacs_tabs

labscript_devices.ZaberStageController.utils

Detailed Documentation

class labscript_devices.ZaberStageController.blacs_tabs.**ZaberStageControllerTab**(*notebook*, *settings*, *restart=False*)

Bases: blacs.device_base_class.DeviceTab

initialise_GUI()

initialise_workers()

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.blacs_tabs

labscript_devices.FunctionRunner.utils

Detailed Documentation

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.

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.blacs_tabs

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()
```

(continues on next page)

(continued from previous page)

```
stop(1)
```

Detailed Documentation

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*)

2.6.3 Dummy Intermediate Device

2.6.4 Test Device

A generic test device to aid in testing labscript infrastructure/functionality.

Detailed Documentation

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,
                phase_mode='aligned',ext_clk=True, clk_freq=100, clk_mult=5)
NovaTech409B_AC(name='novatech', parent_device=pulseblaster_0_clockline_slow,
```

(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)

```
                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,
            camera_attributes={'ExposureTime':9000,
                               'ExposureMode':'Timed',
                               'Gain':0.0,
                               'ExposureAuto':'Off',
                               'GainAuto':'Off',
```

(continues on next page)

(continued from previous page)

```
                                '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')
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
```

(continues on next page)

(continued from previous page)

```
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 device 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 links directly to the appropriate BLACS workers.

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 initialised 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 an 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: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

l

INDEX

M

P

R

S

T

U

W

Z