Using the labscript API

Contents

[Introduction 3](#_Toc357527215)

[The Connection table 3](#_Toc357527216)

[Experiment Logic 6](#_Toc357527217)

[Triggering secondary pseudoclocks 6](#_Toc357527218)

[Calling start 6](#_Toc357527219)

[Commanding input and output 6](#_Toc357527220)

[Waits 7](#_Toc357527221)

[Calling stop 8](#_Toc357527222)

[Global variables from runmanager 8](#_Toc357527223)

[API reference 8](#_Toc357527224)

[Class *Device* 9](#_Toc357527225)

[Methods 9](#_Toc357527226)

[Properties 9](#_Toc357527227)

[Class *Pseudoclock* (subclasses *Device*) 9](#_Toc357527228)

[Methods 9](#_Toc357527229)

[Properties 10](#_Toc357527230)

[Class *PulseBlaster* (subclasses *Pseudoclock*) 10](#_Toc357527231)

[Connection Table Instantiation 10](#_Toc357527232)

[Allowed Children 10](#_Toc357527233)

[Methods 10](#_Toc357527234)

[Properties 10](#_Toc357527235)

[Class *PineBlaster* (subclasses *Pseudoclock*) 11](#_Toc357527236)

[Connection Table Instantiation 11](#_Toc357527237)

[Allowed Children 11](#_Toc357527238)

[Class *IntermediateDevice* (subclasses *Device*) 11](#_Toc357527239)

[Class *NIBoard* (subclasses *IntermediateDevice*) 11](#_Toc357527240)

[Class *NI\_PCI\_6733* (subclasses *NIBoard*) 12](#_Toc357527241)

[Connection Table Instantiation 12](#_Toc357527242)

[Allowed Children 12](#_Toc357527243)

[Class *NI\_PCIe\_6363* (subclasses *NIBoard*) 12](#_Toc357527244)

[Connection Table Instantiation 12](#_Toc357527245)

[Allowed Children 13](#_Toc357527246)

[Class *NovaTechDDS9M* (subclasses *IntermediateDevice*) 13](#_Toc357527247)

[Connection Table Instantiation 13](#_Toc357527248)

[Allowed Children 14](#_Toc357527249)

[Class *PhaseMatrixQuickSyn* (subclasses *Device*) 14](#_Toc357527250)

[Connection Table Instantiation 14](#_Toc357527251)

[Allowed Children 14](#_Toc357527252)

[Class *Output* (subclasses *Device*) 14](#_Toc357527253)

[Class *AnalogQuantity* (subclasses *Output*) 15](#_Toc357527254)

[Methods 15](#_Toc357527255)

[Class *AnalogOut* (subclasses *AnalogQuantity*) 21](#_Toc357527256)

[Connection Table Instantiation 21](#_Toc357527257)

[Class *DDS* (subclasses *Device*) 22](#_Toc357527258)

[Connection Table Instantiation 22](#_Toc357527259)

[Methods 23](#_Toc357527260)

[Properties 24](#_Toc357527261)

[Class *StaticDDS* (subclasses *Device*) 25](#_Toc357527262)

[Connection Table Instantiation 25](#_Toc357527263)

[Methods 26](#_Toc357527264)

[Class *AnalogIn* (subclasses *Device*) 27](#_Toc357527265)

[Connection Table Instantiation 27](#_Toc357527266)

[Methods 27](#_Toc357527267)

[Class *DigitalOut* (subclasses *Output*) 28](#_Toc357527268)

[Connection Table Instantiation 28](#_Toc357527269)

[Methods 28](#_Toc357527270)

[Class *Shutter* (subclasses *DigitalOut*) 28](#_Toc357527271)

[Connection Table Instantiation 29](#_Toc357527272)

[Methods 29](#_Toc357527273)

[Class *Camera* (subclasses *DigitalOut*) 30](#_Toc357527274)

[Connection Table Instantiation 30](#_Toc357527275)

[Methods 31](#_Toc357527276)

[Class *WaitMonitor* (subclasses *Trigger*) 31](#_Toc357527277)

[Connection Table Instantiation 31](#_Toc357527278)

[Method *start* 32](#_Toc357527279)

[Method *stop* 32](#_Toc357527280)

[Method *wait* 32](#_Toc357527281)

Introduction

The labscript API is used to define the logic of an experiment that you wish to run. It is recommended that you read our paper before this documentation, so you are familiar with terms like *pseudoclock*. It would also be a good idea to familiarise yourself with the Python programming language and object oriented (OO) programming if you are not already.

To give you an idea of what a sample experiment looks like, the simplest experiment script (that does something) using the labscript API is below. The script consists of two parts, the *connection table* and the *experiment logic* which will be discussed in the next sections

from labscript import \*  
# Connection Table  
PulseBlaster(name='pulseblaster\_0', board\_number=0)  
DigitalOut(name=’my\_digital\_out’, parent\_device=pulseblaster\_0,   
 connection='flag 2')  
  
#Experiment Logic  
start()  
my\_digital\_out.go\_low(t=0) # start low at the start  
my\_digital\_out.go\_high(t=1) # go high at 1s  
stop(2) # stop at 2s

The Connection table

The connection table maps out the way input/output devices are connected to each other in your lab, and the channels (individual inputs/outputs) they have. The devices in your lab should be connected in a similar way to that shown in figure 1.

**Figure 1.** A tree diagram showing example connections between I/O devices in a lab. There are three tiers to the tree, with the highest tier, consisting of pseudoclocks, labelled *tier 1* and the lowest tier, consisting of input/output channels, labelled tier 3. Here *type* refers to the model of the device, or the input/output channel type. The blue arrows and text indicate the connection on the upper tier device that connects to the lower tier device, input or output.

fast clock

slow clock

pulseblaster\_0  
Type: PulseBlaster  
(Pseudoclock)

ni\_card\_0   
Type: NI\_PCIe\_6363  
(I/O board)

ni\_card\_1   
Type: NI\_PCI\_6733  
(output board)

novatech\_DDS9m\_1   
Type: NovaTechDDS9M  
(4x DDS outputs)

pineblaster\_0  
Type: PineBlaster  
(Pseudoclock)

novatech\_DDs9m\_2   
Type: NovaTechDDS9M  
(4x DDS outputs)

dds 1

fast clock

fast clock

dds 0

dds 2

dds 3

dipole\_trap\_AOM  
Type: DDS

dds 0

dds 1

repump\_AOM  
Type: DDS

source\_MOT\_AOM  
Type: DDS

rf\_knife  
Type: DDS

laser\_offset\_AOM  
Type: StaticDDS

imaging\_AOM   
Type: StaticDDS

switch\_1  
Type: DigitalOut

MOT\_coil  
Type: AnalogOut

MOT\_fluorescence  
Type: AnalogIn

bias\_coil\_x  
Type: AnalogOut

bias\_coil\_y  
Type: AnalogOut

bias\_coil\_z  
Type: AnalogOut

port0/line13

ao0

ai3

ao2

ao1

ao0

In figure 1 we see two pseudoclocks in the top tier of the diagram. They do not have a *parent device* that tells them when to update their output (this is true for **all** pseudoclocks). If a device is not a pseudoclock, it must be connected to one, and such devices are pictured in the second tier. In figure 1, we see various *I/O devices* such as an NI\_Pcie\_6363 card or a Novatech DDS9m board connected to the pseudoclocks. In the third tier, we see the input and output channels of the I/O devices.

The diagram in figure 1 can be directly mapped to the Python code (calling the labscript API) required to build a connection table. We’ll start with importing the labscript API and creating the pseudoclocks:

from labscript import \*  
PulseBlaster(name='pulseblaster\_0', board\_number=0)  
PineBlaster(name='pineblaster\_0', usbport='com7',  
 trigger\_device=pulseblaster\_0, trigger\_connection='flag 2')

Those familiar with Python will recognise that PulseBlaster and PineBlaster are classes we are instantiating into *objects*. What will be unfamiliar is that the resulting object is not being assigned to a variable (you would usually write something like x = PulseBlaster(…)). This is because the labscript classes (such as PulseBlaster and PineBlaster) automatically create a variable with the same name as is passed in the name keyword argument at instantiation time. The following illustrates this point in code:

x = PulseBlaster(name='pulseblaster\_0', board\_number=0)  
print x == pulseblaster\_0 # prints True

Continuing with the code example above, we now have two variables, pulseblaster\_0 and pineblaster\_0 available to use in the remainder of our code (you will notice we have already used pulseblaster\_0 as the trigger\_device reference for the PineBlaster). You will also have noticed the names of the devices are highlighted in purple and the device classes are orange. The colouring used here matches the colouring used for the information contained in figure 1, showing how the diagram directly maps to labscript API calls. **Matching information will continue to be highlighted as we create more of the connection table.**

From the example instantiation of two pseudoclocks, you should be able to see some similarities between them already.

1. Both take a name as an argument (these names can be anything, as long as they match the python syntax for variable names. Spaces, for instance, are prohibited in device names for this reason).
2. Both take a description of how the device is connected to the computer (either usbport='COM7' or board\_number=0). This information is used by BLACS to communicate with the device before execution of the experiment under hardware timing.

Lastly, while not obvious from this example, any pseudoclock can be given optional arguments specifying how it is to be triggered by the *master pseudoclock* (trigger\_device=pulseblaster\_0, trigger\_connection='flag 2'). The master pseudoclock is the device which is **not** given these trigger arguments (in this case pulseblaster\_0). The trigger device can also be a device that is clocked by the master pseudoclock (one in tier two of figure 1). **There can only be one master pseudoclock.**

Next we will instantiate the second tier devices:

NI\_PCIe\_6363(name='ni\_card\_0', parent\_device=pulseblaster\_0,   
 clock\_type='fast clock', clock\_terminal='ni\_pcie\_6363\_0/PFI0',  
 MAX\_name='ni\_pcie\_6363\_0', acquisition\_rate=100e3)  
  
NI\_PCI\_6733(name='ni\_card\_1', parent\_device=pulseblaster\_0,   
 clock\_type='fast clock', clock\_terminal='ni\_pcie\_6733\_0/PFI0',   
 MAX\_name='ni\_pci\_6733\_0')  
  
NovaTechDDS9M(name='novatech\_DDS9m\_1', parent\_device=pulseblaster\_0,   
 clock\_type='slow clock', com\_port="com10")  
  
NovaTechDDS9M(name='novatech\_DDS9m\_2', parent\_device=pineblaster\_0,   
 clock\_type='fast clock', com\_port="com11")

You will again notice some similarities between these lines of code:

1. They all have a user defined name, just like the pseudoclocks, which will be used to create a Python variable containing a Python object for this device. We will use these variables which reference tier two devices when creating tier three inputs and outputs, just like we used variables pulseblaster\_0 and pineblaster\_0 when creating our tier two devices (see below).
2. They all have a parent device, which is a reference (the Python variable, not a string) to the pseudoclock which provides the timing signal to this device.
3. All devices have a clock\_type parameter which indicates which output of the pseudoclock will be connected to the clock input of the device. Pseudoclocks like the PulseBlaster have two clocking outputs. The slow clock ticks less often than the fast clock, for use with devices that have a limited instruction buffer. However, slow clocks cannot take advantage of instructions that *ramp* an output value. On the PulseBlaster, the fast clock is output on flag 0 and the slow clock on flag 1 (this may become configurable in a later version). The PineBlaster only has one output, the fast clock.
4. All devices have a parameter which defines how the device is connected to the computer. This might be the name specified in the NI Measurement and Automation Explorer (MAX), max\_name, for National Instruments devices or a com port in the case of the Novatech DDS9m devices.

Finally we come to instantiating the input and output channels (the third tier in figure 1):

DigitalOut( name='switch\_1', parent\_device=ni\_card\_0,   
 connection='port0/line1')  
AnalogOut( name='MOT\_coil', parent\_device=ni\_card\_0, connection='ao0')  
AnalogIn( name='MOT\_fluorescence', parent\_device=ni\_card\_0,  
 connection='ai3')  
  
AnalogOut( name='bias\_coil\_x', parent\_device=ni\_card\_1, connection='ao0')  
AnalogOut( name='bias\_coil\_y', parent\_device=ni\_card\_1, connection='ao1')  
AnalogOut( name='bias\_coil\_z', parent\_device=ni\_card\_1, connection='ao2')  
  
DDS( name='source\_MOT\_AOM', parent\_device=novatech\_DDS9m\_1,   
 connection='dds 0')  
DDS( name='rf\_knife', parent\_device=novatech\_DDS9m\_1,   
 connection='dds 1')  
StaticDDS( name='laser\_offset\_AOM', parent\_device=novatech\_DDS9m\_1,   
 connection='dds 2')  
StaticDDS( name='imaging\_AOM', parent\_device=novatech\_DDS9m\_1,   
 connection='dds 3')  
  
DDS( name='dipole\_trap\_AOM', parent\_device=novatech\_DDS9m\_2,   
 connection='dds 0')  
DDS( name='repump\_AOM', parent\_device=novatech\_DDS9m\_2,   
 connection='dds 1')

Again we see similarities:

1. Each has a user specified name, that is used to create a Python variable for later reference
2. Each has a parent device (at Tier 2), which is specified by the Python variable referencing the Python object created when the device was instantiated.
3. Each has a connection, which is a string specifying the port on the parent device this input/output is using.

You may notice that points 2 and 3 look very similar to points 2 and 3 we discussed for Tier 2. This is not by accident! Specifying a *parent device* and *connection* completely describes the connection to the device at the next tier up.

You now have a complete connection table. Please note that many of these labscript Classes take other arguments at instantiation. These will be covered in more depth at the end of this guide.

Experiment Logic

The logic of the experiment (in other words, the definition of what happens when) is defined by calling functions, provided by the labscript API, that are part of the Python objects created when the connection table was defined. This section will provide some examples, and comment on the similarities between the various function calls. As such, this section should only be taken as an introduction to programming the experiment logic. Once you have understood the basics, the API reference at the end of this document should be your port of call.

## Triggering secondary pseudoclocks

Secondary pseudoclocks must have their initial trigger time set. Usually all pseudoclocks would be triggered at t=0, however there may be instances where you do not wish to start a secondary pseudoclock until far into the experiment (for instance if it has a limited instruction buffer).

Before calling the start function (see the next sub section), you must call the set\_initial\_trigger\_time function on each pseudoclock. For example:

pineblaster\_0.set\_initial\_trigger\_time(t=0)

## Calling start

The start function automatically calls the trigger function for each pseudoclock, creating the required hardware instructions on the master pseudoclock (or child device) at the times specified in the call to set\_initial\_trigger\_time for each pseudoclock. This ultimately means you just call the start function and don’t worry about what it does:

start()

## Commanding input and output

It is generally a god idea to create a variable that keeps track of the current time in your labscript. While it is not necessary to command input/output in chronological order, we recommend you do for readability. As such, we will use the variable t from now on to reference time. If you use t in your code, rather than hard coded times, it allows you to insert new code easily without having to manually offset every subsequent function call accordingly. A good example of this exists in the *global variables from runmanager* section, in this document. It is thus a good idea to define this immediately after calling start:

t=0

To command input and output from channels of hardware, you call functions (instance methods) that are associated with the Python objects created when they were defined in the connection table. For example, to set the state of a digital output you would call either:

switch\_1.go\_high(t)  
or  
switch\_1.go\_low(t)

As another example, to set an analog output to a constant value (0.3 V), you would call:

bias\_coil\_x.constant(t,value=0.3)

Outputs that are connected to a device that are clocked by the *fast clock* can also take advantage of functions that change the output over a range of values, following some functional form. For instance, a linear ramp over 5 seconds, from an output voltage of 1.0V to 3.5V and with a sample rate of 1kHz (5000 updates over the entire ramp duration), can be defined by:

t += MOT\_coil.ramp(t,duration=5,initial=1.0,final=3.5,samplerate=1e3)

You will notice we add the result of the function call to our time variable. In the case of ramp, this just returns the number we specified as the duration (which we knew anyway!). However, in the case of some of the more complex exponential ramps, you can command the truncation of the ramp once the output reaches a certain output value while using the duration of the ramp to define the shape. The length of time the ramp takes then differs to the duration specified.

To command analog acquisition, you should call:

MOT\_fluorescence.acquire(label='', start\_time=, dend\_time=)

## Waits

All current pseudoclocks support *wait* instructions, which pause the execution of an experiment until an external trigger is received. To use waits, you must first create a *wait monitor*, a labscript object which commands a device to report the status of the waits in the experiment. This object should be considered a device, and should be part of your connection table (both the experiment connection table and lab/BLACS connection table). To create a wait monitor, the following call in labscript must be made before the call to start():

WaitMonitor(name='wait\_monitor',   
 parent\_device=ni\_card\_0, connection='port0/line0',  
 acquisition\_device=ni\_card\_0, acquisition\_connection='ctr0',   
 timeout\_device=ni\_card\_0, timeout\_connection='pfi1')

The parent\_device and connection parameters specify the buffered, digital channel that should be used as a flag for the wait monitor to indicate when the hardware is in a wait state. This output will go high while the devices are waiting, and low otherwise. It should not be in use anywhere else in the connection table. The acquisition\_device and acquisition\_connection parameters specify the counter that should be used to determine how long the waits are (necessary to accurately reference acquired signals with the requested output from the experiment). Currently this can only be an NI counter. The digital flag indicating the wait state, specified in the previous two parameters, should be connected to this counter input. Finally, the timeout\_device and timeout\_connection parameters specify the **software timed** digital output used to trigger the master pseudoclock to continue, should the timeout, specified in a particular wait call (see below), be exceeded. This digital output must not be under hardware timed control during the experiment. We recommend using a spare PFI output on an NI card. Note that due to limitations in BLACS, currently the acquisition\_device and timeout\_device must be the same device.

During the experiment logic, you can then call the following function to initiate a *wait* at time t:

t += wait(label='my\_wait', t=t, timeout=2)

As you can see, the wait function must be given a unique label, the time at which to perform the wait, and the maximum time the wait should last for, the timeout, in seconds. The function returns the minimum time the wait will last for, which includes the time necessary to retrigger all pseudoclocks. It is recommended to add this returned time to your time keeping variable t.

## Calling stop

At the end of your experiment logic, you must call the stop function. This tells labscript how long to hold the final instruction, before ending the experiment. As such the only parameter you pass to the stop function is the end time, such should be later than the last instruction by at least the minimum instruction length. In most situations, you can just call stop a few milliseconds after your last instruction, but if you want to be more precise, you can.

stop(t+1e-3)

Global variables from runmanager

Experiments written using the labscript API are compiled to hardware instructions using the interface of runmanager. runmanager allows you to define easily modify parameters from a graphical interface. These parameters are inserted into the Python namespace, just before your experiment script is run. This allows you to you to modify the behaviour of your code, without actually editing the code!

In runmanager, you specify a name for each parameter along with its value. The names must be unique, as they are used to name the Python variable you may access in your code. If for instance you created a parameter in runmanager called MOT\_hold\_time, you could use it in your experiment logic to increment your time variable:

MOT\_coil.constant(t,value=1.0)  
t += MOT\_hold\_time  
MOT\_coil.constant(t,value=0.0)

These parameters can of course be used anywhere in your Python code, including in the connection table and function calls to command input or output.

API reference

This is the API reference for labscript. Here we provide documentation for all the labscript functions we provide for the end user to use in their experiment logic.

For those not familiar with classes,objec ts and inheritance, this API reference is presented in the form of classes and their functions (called methods in Python). Objects are created from classes when the objects are *instantiated* in the connection table. The functions/methods/variables available in the class definitions, are then available to be called on the object as we’ve seen in the previous sections. If a class *subclasses* another labscript class (called the parent class), all the methods available in the parent class are available to the child class, unless a function with the same name is defined in the child class. This is all standard practice in object oriented programming languages, and if you are not familiar with object oriented programming, we again recommend running through some online tutorials!

Methods are usually called in the form:

device.method(parameters)

whereas properties are used like:

device.property

## Class *Device*

The Device class is the parent class of all devices and input/output channels. You usually won’t interact with this class directly (you will never instantiate a pure Device object in the connection table) but it provides some useful functionality that is available in all subclasses.

### Methods

**get\_all\_children()**

This method returns a list of all children, grandchildren, etc of the device, regardless of whether they are an output or not.

**get\_all\_outputs()**

This method returns a list all children, grandchildren, etc of the device that are outputs. An output is usually an object that subclasses the Output class, such as a DigitalOut, Shutter or AnalogOut.

### Properties

**pseudoclock**

This property returns the object for the pseudoclock for this device.

**t0**

This property returns the earliest time output/input can be commanded from the device at the start of the experiment. This is non-zero on devices/outputs/inputs on secondary pseudoclocks due to triggering delays.

## Class *Pseudoclock* (subclasses *Device*)

The Pseudoclock class is the parent class of all pseudoclocks (such as the PineBlaster and PulseBlaster classes). You won’t usually interact with this class directly (you will never instantiate a pure Pseudoclock object in the connection table) but it provides some useful functionality that is available in all subclasses.

### Methods

**set\_initial\_trigger\_time(t)**

This method sets the time of the initial trigger of a secondary pseudoclock. All secondary pseudoclocks must call this method before a call to start() is made. The parameter t specifies the time to wait between starting the master pseudoclock and triggering this secondary pseudoclock to start.

### Properties

**is\_master\_pseudoclock**

This property returns True if the pseudoclock is the master pseudoclock, otherwise it returns False.

## Class *PulseBlaster* (subclasses *Pseudoclock*)

The PulseBlaster class is used to instantiate an object for the SpinCore PulseBlaster DDSII-300 device.

### Connection Table Instantiation

**PulseBlaster(name,board\_number)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This argument should be a string.
* The board\_number argument specifies the board number (an integer, 0 or 1 for example) for this device as given by the SpinCore API. Please see the SpinCore API documentation for how to identify the board number of your device, however if you only have one SpinCore PulseBlaster attached, it will have a board number of 0.

### Allowed Children

The PulseBlaster is a pseudoclock which can output both a *fast clock* and a *slow clock*. Devices which subclass IntermediateDevice can be connected to either the fast or the slow clock, with the clock chosen by the clock\_type argument at instantiation of subclasses of IntermediateDevice. The fast clock is always output on *flag 0* and the slow clock on *flag 1*, but should become configurable in a later release.

The PulseBlaster also has direct outputs, which are considered to be connected to the slow clock:

* Digital outputs can be created on flags 2-11 by instantiating a DigitalOut object with connection=’flag 2’, where two can be a number from 2-11.
* DDS outputs can be created by instantiating a DDS object with connection=’dds 0’ or connection=’dds 1’.

### Methods

**set\_initial\_trigger\_time(t)**

This method sets the time of the initial trigger of a secondary pseudoclock. All secondary pseudoclocks must call this method before a call to start() is made. The parameter t specifies the time to wait between starting the master pseudoclock and triggering this secondary pseudoclock to start.

### Properties

**is\_master\_pseudoclock**

This property returns True if the pseudoclock is the master pseudoclock, otherwise it returns False.

## Class *PineBlaster* (subclasses *Pseudoclock*)

The PineBlaster class is used to instantiate an object for the PineBlaster pseudoclock developed by the BEC group at Monash University.

### Connection Table Instantiation

**PineBlaster(name,** **trigger\_device=None, trigger\_connection=None,   
 usbport='COM1')**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The trigger\_device argument is optional and is used to configure the pseudoclock as a secondary pseudoclock, and specifies the device used to trigger this pseudoclock to start. The argument, if used, should contain a reference to a device object already instantiated in the connection table. This argument is equivalent to the second argument of the DigitalOut instantiation, parent\_device. If this argument is not None (the default), you must also specify a trigger\_connection.
* The trigger\_connection argument is optional and is used to specify the connection (e.g. output port) of the trigger\_device specified. This argument should be a string, and is equivalent to the connection argument of the DigitalOut instantiation. The default value for this argument is None.
* The usbport argument specifies the physical connection between the PineBlaster and PC, used to program the device. It defaults to ‘COM1’ (a string), however this is completely dependent on computer configuration and should not be assumed to be correct. You can identify the COM port used by the PineBlaster in MS Windows by viewing available COM ports within the windows device manager, and unplugging/replugging the device and identifying which COM port disappears and reappears.

### Allowed Children

The PineBlaster is a pseudoclock which can output only a *fast clock*. Devices which subclass IntermediateDevice can be connected to the fast clock, by specifying clock\_type=’fast clock’ as an argument at instantiation of subclasses of IntermediateDevice.

The PineBlaster has no other outputs.

## Class *IntermediateDevice* (subclasses *Device*)

The IntermediateDevice class is used as the base class for all devices that are to be clocked by a pseudoclock. It does not have any useful methods or properties of its own, but may in the future.

## Class *NIBoard* (subclasses *IntermediateDevice*)

The NIBoard class is used as the base class for all National Instruments (NI) boards. It does not have any useful methods or properties of its own, but may in the future.

## Class *NI\_PCI\_6733* (subclasses *NIBoard*)

The NI PCI-6733 is a National Instruments PCI device with 8 buffered analog outputs.

### Connection Table Instantiation

**NI\_PCI\_6733(name,** **parent\_device,clock\_type,clock\_terminal,MAX\_name)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the pseudoclock this device is to be clocked by. The argument should contain a reference to a pseudoclock object already instantiated in the connection table.
* The clock\_type argument specifies whether this device is connected to the *fast clock* or *slow clock* of the pseudoclock. It should be set to a string of the form ‘fast clock’ or ‘slow clock’.
* The clock\_terminal argument specifies the input of the NI device that will receive the clocking signal from the pseudoclock. An example of this would be the string ‘/ni\_pci\_6733\_0/PFI0’ where ni\_pci\_6733\_0 is the name of the device as specified in the NI *Measurement and Automation eXplorer* (MAX). Note that the clock input does not have to be on this device if it is connected to another NI device via an RTSI cable. In the case of an RTSI connection, it could be set to ‘/ni\_pcie\_6363\_0/PFI0’ where ni\_pcie\_6363\_0 is the name of another NI device specified in MAX and, this other device, along with the one you are instantiating here, are connected via an RTSI cable.
* The MAX\_name argument specifies the name of the device as set in the NI *Measurement and Automation eXplorer* (MAX). It should be specified as a string, and not contain any forward slashes.

### Allowed Children

The NI PCI-6733 device can have up to 8 analog outputs attached. Please be aware that this NI device requires there to always be an even number of analog outputs defined. If you require an odd number of outputs, please instantiate an unused AnalogOut in your connection table anyway to keep the number even. The connection argument of AnalogOut should be set to a string of the form ‘ao0’ where the integer 0 can be any integer from 0 to 7.

## Class *NI\_PCIe\_6363* (subclasses *NIBoard*)

The NI PCI-6733 is a National Instruments PCI-Express device with 4 buffered analog outputs, 32 buffered analog inputs and 32 buffered Digital IO channels (currently only output is supported, but a future version of the labscript suite will support Digital Input).

### Connection Table Instantiation

**NI\_PCIe\_6363(name,** **parent\_device, clock\_type, clock\_terminal,   
 MAX\_name, acquisition\_rate=0)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the pseudoclock this device is to be clocked by. The argument should contain a reference to a pseudoclock object already instantiated in the connection table.
* The clock\_type argument specifies whether this device is connected to the *fast clock* or *slow clock* of the pseudoclock. It should be set to a string of the form ‘fast clock’ or ‘slow clock’.
* The clock\_terminal argument specifies the input of the NI device that will receive the clocking signal from the pseudoclock. An example of this would be the string ‘/ni\_pcie\_6363\_0/PFI0’ where ni\_pcie\_6363\_0 is the name of the device as specified in the NI *Measurement and Automation eXplorer* (MAX). Note that the clock input does not have to be on this device if it is connected to another NI device via an RTSI cable. In the case of an RTSI connection, it could be set to ‘/ni\_pci\_6733\_0/PFI0’ where ni\_pci\_6733\_0 is the name of another NI device specified in MAX and, this other device, along the one you are instantiating here, are connected via an RTSI cable.
* The MAX\_name argument specifies the name of the device as set in the NI *Measurement and Automation eXplorer* (MAX). It should be specified as a string, and not contain any forward slashes.
* The acquisition\_rate argument is optional, but required if you wish to use the analog input channels. This parameter specifies the acquisition rate for all analog input channels, and as such should be set to the maximum sample rate you require in a particular experiment. The value should take the form of an integer. While defined as part of the connection table instantiation, this value is not part of any connection table comparison done in BLACS and the value used in an experiment connection table may thus differ from the value specified in the BLACS (lab) connection table.

### Allowed Children

The NI PCIe-6363 device can have up to:

* 4 analog outputs attached. The connection argument of AnalogOut should be set to a string of the form ‘ao0’ where the integer 0 can be any integer from 0 to 3.
* 32 analog inputs attached. The connection argument of AnalogIn should be set to a string of the form ‘ai0’ where the integer 0 can be any integer from 0 to 31.
* 32 digital outputs attached. The connection argument of DigitalOut should be set to a string of the form ‘port0/line0’ where the integer 0 can be any integer from 0 to 31.

## Class *NovaTechDDS9M* (subclasses *IntermediateDevice*)

The NovaTech DDS9m is a serial device with 4 DDS outputs. The first two outputs support buffered output, and can be clocked by a pseudoclock. The last two outputs cannot be clocked by a pseudoclock, but can be set to an initial value during program time at the start of every experiment shot.

### Connection Table Instantiation

**NovaTechDDS9M(name,** **parent\_device, clock\_type, com\_port)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the pseudoclock this device is to be clocked by. The argument should contain a reference to a pseudoclock object already instantiated in the connection table.
* The clock\_type argument specifies whether this device is connected to the *fast clock* or *slow clock* of the pseudoclock. It should be set to a string of the form ‘fast clock’ or ‘slow clock’.
* The com\_port argument specifies the COM port that this NovaTech DDS9m device is connected to. The port is specified as a string of the form ‘COM1’.

### Allowed Children

The NovaTech DDS9m device can have up to 4 DDS outputs attached. The connection argument of DDS or StaticDDS should be set to a string of the form ‘dds 0’ where the integer 0 can be any integer from 0 to 3. ‘dds 0’and ‘dds 1’ should be instantiated as DDS objects while ‘dds 2’ and ‘dds 3’ should be instantiated as StaticDDS objects.

## Class *PhaseMatrixQuickSyn* (subclasses *Device*)

The Phase Matrix QuickSyn is a microwave frequency generator. This device cannot be clocked by a pseudoclock (it has no instruction table) and so the labscript class is not derived from IntermediateDevice. This device can have the frequency (and output gate) set at the beginning of an experiment shot, during program time.

### Connection Table Instantiation

**PhaseMatrixQuickSyn(name, com\_port)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The com\_port argument specifies the COM port that this Phase Matrix QuickSyn device is connected to. The port is specified as a string of the form ‘COM1’.

### Allowed Children

The Phase Matrix QuickSyn device can have 1 StaticDDS output attached. The connection argument of StaticDDS should be set to the string ‘dds 0’.

## 

## Class *Output* (subclasses *Device*)

The Output class is used as the base class for all output classes. It does not have any useful methods or properties of its own, but may in the future.

## Class *AnalogQuantity* (subclasses *Output*)

The AnalogQuantity class is used as the base class for the AnalogOut class. It is also used internally by the DDS class. You should never instantiate an AnalogQuantity class directly in the connection table. Instead use AnalogOut.

### Methods

**constant(t,value,units=None)**

This method sets the output to the value specified, in the units given (if specified) at the time t.

* The t argument specifies the time at which to set the output to the value given. This time is given in seconds, as a float or integer.
* The value argument specifies the value to set the output to.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file

**exp\_ramp(t, duration, initial, final, samplerate, zero=0,   
 trunc=False, trunc\_type='linear', units=None)**

This method changes the output following an exponential curve that is defined by the parameters specified. **This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which will be the duration specified as the second argument to the function unless truncation is used (see below description of parameters for more details).

The final form of the exponential curve is defined by:  
Function(t) = (initial-zero)\*exp(-(1/duration \* log((initial-zero)/(final-  
 zero)))\*t) + zero

And this function is evaluated from t=0 to t=duration where duration may be defined by trunc\_duration if truncation is used (see below description of parameters for more details).

* The t argument specifies the time at which to begin the exponential ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This value should be specified as a float or integer. The actual curve produced may be shorter if trunc is not False. The actual duration of the ramp is returned by the function.
* The initial argument specifies the initial output value (the output value at time t). This value should be specified as a float or integer.
* The final argument specifies the final output value (the output value at time t+duration). This value should be specified as a float or integer. As the ramp may be truncated if trunc is not False, the final value that is output may not actually be the one specified.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The zero argument (by default set to 0) specifies the output value at which the exponential ramp should decay to if it were to continue until . This parameter, along with the initial and final time points completely describes the exponential curve. This value should be specified as a float or integer.
* The trunc argument, False by default, defines the output value at which the output stop. This value should be specified as a float or integer or False. This allows you to define the shape of an exponential curve using initial, final and zero and then truncate the curve at a given output value.
* The trunc\_type argument specifies how the new duration of the ramp is calculated if trunc is not False. If trunc\_type=’linear’ (the default value) then the duration of the ramp is given by:

trunc\_duration = duration\*log((initial-zero)/(trunc-zero))   
 /log((initial-zero)/(final-zero))

If the trunc\_type=’exponential’ then the duration of the ramp is given by:

trunc\_duration = trunc \* duration

* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

**exp\_ramp\_t(t, duration, initial, final, time\_constant, samplerate,   
 trunc=False, units=None)**

This method changes the output following an exponential curve that is defined by the parameters specified. It is similar to the exp\_ramp function, except that time\_constant parameter is used as part of the curve characterisation instead of the zero parameter. **This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which will be the duration specified as the second argument to the function unless truncation is used (see below description of parameters for more details).

The final form of the exponential curve is defined by:  
zero = (final-initial\*exp(-duration/time\_constant)) /   
 (1-exp(-duration/time\_constant))  
Function(t) = (initial-zero)\*exp(-(t)/time\_constant) + zero

And this function is evaluated from t=0 to t=duration where duration may be defined by trunc\_duration if truncation is used (see below description of parameters for more details).

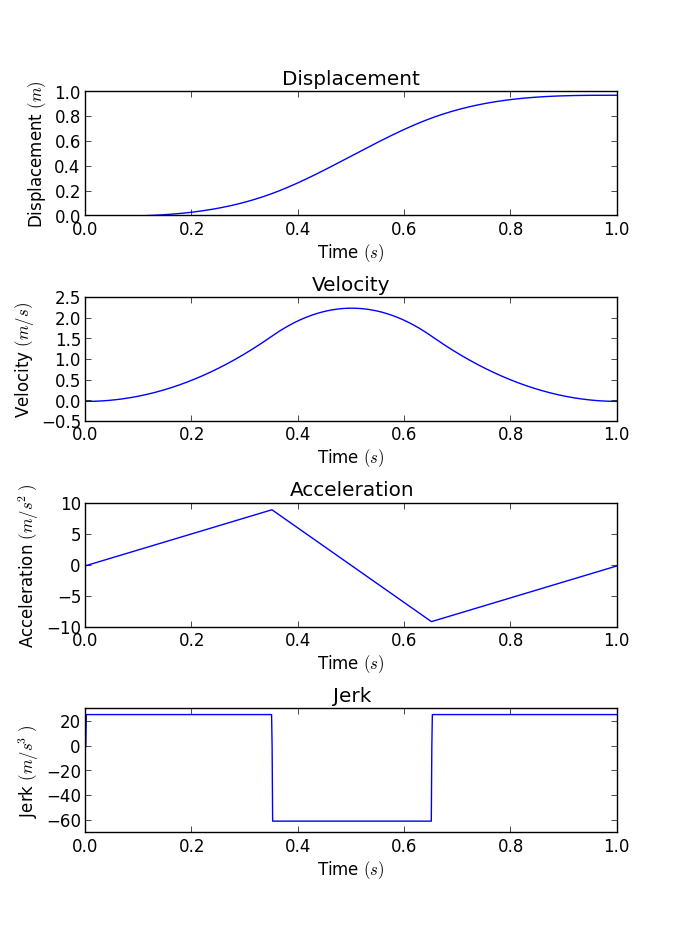
* The t argument specifies the time at which to begin the exponential ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This value should be specified as a float or integer. The actual curve produced may be shorter if trunc is not False. The actual duration of the ramp is returned by the function.
* The initial argument specifies the initial output value (the output value at time t). This value should be specified as a float or integer.
* The final argument specifies the final output value (the output value at time t+duration). This value should be specified as a float or integer. As the ramp may be truncated if trunc is not False, the final value that is output may not actually be the one specified.
* The time\_constant argument specifies the time constant to use in the exponent of the exponential function. This parameter, along with the initial and final time points completely describes the exponential curve. This value should be specified as a float or integer.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The trunc argument, False by default, defines the output value at which the output stop. This value should be specified as a float or integer or False. This allows you to define the shape of an exponential curve using initial, final and time\_constant and then truncate the curve at a given output value. The duration of the ramp, if truncation is used, is given by:

trunc\_duration = time\_constant \* log((initial-zero)/(trunc-zero))

* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

**piecewise\_accel\_ramp(t, duration, initial, final, samplerate,   
 units=None)**

This method changes the output so that the second derivative of the output value with respect to time follows one period of a triangle wave. The figure below shows an example of this, where the output value is used to control something that translates to a displacement (shown on the y-axis of the top graph) and the duration of the ramp is 1 second.



**This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which is equivalent to the duration specified.

The final form of the output curve is defined by:  
Function(t) = initial + (final-initial) \* (  
 (9./2 \* t\*\*3/duration\*\*3) \* (t<duration/3)  
 + (-9\*t\*\*3/duration\*\*3 + 27./2\*t\*\*2/duration\*\*2   
 -9./2\*t/duration + 1./2) \* (t<2\*duration/3)   
 \* (t>=duration/3)  
 + (9./2\*t\*\*3/duration\*\*3 - 27./2 \* t\*\*2/duration\*\*2   
 + 27./2\*t/duration - 7./2) \* (t>= 2\*duration/3))

And this function is evaluated from t=0 to t=duration.

* The t argument specifies the time at which to begin the ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This value should be specified as a float or integer. This duration is returned by the function.
* The initial argument specifies the initial output value (the output value at time t). This value should be specified as a float or integer.
* The final argument specifies the final output value (the output value at time t+duration). This value should be specified as a float or integer.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

**ramp(t, duration, initial, final, samplerate, units=None)**

This method changes the output linearly. **This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which is equivalent to the duration specified.

The final form of the output curve is defined by:  
Function(t) = ((final - initial)/duration)\*t + initial

And this function is evaluated from t=0 to t=duration.

* The t argument specifies the time at which to begin the ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This duration is returned by the function. This value should be specified as a float or integer.
* The initial argument specifies the initial output value (the output value at time t). This value should be specified as a float or integer.
* The final argument specifies the final output value (the output value at time t+duration). This value should be specified as a float or integer.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

**sine(t, duration, amplitude, angfreq, phase, dc\_offset, samplerate,   
 units=None)**

This method changes the output following a sine curve that may go through more than one period. **This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which is equivalent to the duration specified.

The final form of the output curve is defined by:  
Function(t) = amplitude\*sin(angfreq\*(t) + phase) + dc\_offset

And this function is evaluated from t=0 to t=duration.

* The t argument specifies the time at which to begin the ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This duration is returned by the function. This value should be specified as a float or integer.
* The amplitude argument specifies the amplitude of the sine wave. The maximum output value will be equal to amplitude+dc\_offset. This value should be specified as a float or integer.
* The angfreq argument specifies the angular frequency of the sine wave in radians per second. This value should be specified as a float or integer.
* The phase argument specifies the phase of the sine wave in radians. This value should be specified as a float or integer.
* The dc\_offset argument specifies the offset from 0 of the sine wave. This value should be specified as a float or integer.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

**sine\_ramp(t, duration, initial, final, samplerate, units=None)**

This method changes the output through one half period of a sine wave, where the limits (maximum/minimum values) of the sine wave are defined by the initial and final arguments. **This method can only be used if the output is connected to a device that is clocked by the *fast clock* of a pseudoclock.** This function returns the length of time the ramp will take, which is equivalent to the duration specified.

The final form of the output curve is defined by:  
Function(t) = (final-initial)\*(sin(pi\*(t)/(2\*duration)))\*\*2 + initial

And this function is evaluated from t=0 to t=duration.

* The t argument specifies the time at which to begin the ramp. This time is given in seconds, as a float or integer.
* The duration argument specifies the length of the curve in time. This duration is returned by the function. This value should be specified as a float or integer.
* The initial argument specifies the initial output value (the output value at time t). This value should be specified as a float or integer.
* The final argument specifies the final output value (the output value at time t+duration). This value should be specified as a float or integer.
* The samplerate argument specifies the rate at which the output should update (number of updates per second) during the ramp. This value should be given as an integer or float in the units of Hz.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like V or Hz, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS or AnalogOut. For more details, see the documentation for these classes in this document or the separate unit conversion documentation file.

## Class *AnalogOut* (subclasses *AnalogQuantity*)

The AnalogOut class is used to instantiate all Analog outputs on devices (such as those made by NI). It inherits all the methods from AnalogQuantity.

### Connection Table Instantiation

**AnalogOut(name, parent\_device, connection, limits = None,   
 unit\_conversion\_class = None,  
 unit\_conversion\_parameters = None)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘ao0’ for an analog output of the NI PCIe-6363.
* The limits argument is an optional argument that specifies the lower and upper output value for this output. If output is commanded outside of the specified limits during the experiment logic, an error will be thrown. The default is None, which imposes no limits other than those determined by the hardware. If you wish to specify limits, this argument should be set to a tuple, of the form (lower,upper) where lower and upper are specified as floats or integers.
* The unit\_conversion\_class argument is used to specify the Python class to be used for unit conversion (see the unit conversion documentation for how to create such classes). It should be a reference to a Python class (not an instance of a Python class). Specifying a class here allows you to use different units when calling the methods of AnalogQuantity.
* The unit\_conversion\_parameters argument is used to pass in parameters to the class specified by the unit\_conversion\_class argument when it is instantiate by the internals of the labscript API and the internals of BLACS. This argument defaults to None, in which case no parameters are passed in. Otherwise it should take the form of a dictionary, and can contain anything you like as long as it can be represented as a string (you must be able to execute in a Python terminal repr({…my dictionary here…})). The contents of the dictionary will depend on the specific implementation of the unit conversion class.

## Class *DDS* (subclasses *Device*)

The DDS class is used to instantiate DDS outputs on devices that support changing the DDS state during buffered execution. It is a container class, with properties that allow you to access the frequency, amplitude and phase properties which are of class AnalogQuantity. It may also have a gate property, to enable/disable the DDS output, which is of class DigitalOut. Convenience methods are provided to set these the frequency, amplitude and phase to constant values and enable/disable the gate. If the DDS is clocked by the *fast clock* of the pseudoclock, you can access the ramping functionality of the AnalogQuantity’s directly through the object properties.

### Connection Table Instantiation

**DDS(name, parent\_device, connection, digital\_gate={},   
 freq\_limits=None, freq\_conv\_class=None, freq\_conv\_params={},   
 amp\_limits=None, amp\_conv\_class=None, amp\_conv\_params={},   
 phase\_limits=None, phase\_conv\_class=None, phase\_conv\_params={})**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘dds 0’ for a DDS output of the SpinCore PulseBlaster DDSII-300.
* The digital\_gate argument can be used to link a digital output of another device, which is set up to turn on/off the output of the DDS, to the enable/disable methods of this object. For example, a digital output of an NI device might be setup to control a RF switch on the output of a DDS on a NovaTech DDS9m channel. The form of this argument should be a dictionary, with keys ‘device’ and ‘connection’ which correspond to the arguments for parent\_device and connection you would pass were you instantiating the digital output using the DigitalOut class. An example dictionary is {‘device’:ni\_pcie\_6363\_0, ‘connection’:’port0/line12’}, where ni\_pcie\_6363\_0 is a device already instantiated in the connection table. **You should not instantiate the gate as a DigitalOut directly. Labscript will do this internally for you.**
* The \*\_limits arguments (replace the \* with either freq, amp, or phase) are an optional argument that specifies the lower and upper output value for this output. If output is commanded outside of the specified limits during the experiment logic, an error will be thrown. The default is None, which imposes no limits other than those determined by the hardware. If you wish to specify limits, this argument should be set to a tuple, of the form (lower,upper) where lower and upper are specified as floats or integers.
* The \*\_conv\_class arguments (replace the \* with either freq, amp, or phase) are used to specify the Python class to be used for unit conversion (see the unit conversion documentation for how to create such classes). It should be a reference to a Python class (not an instance of a Python class). Specifying a class here allows you to use different units when calling the methods to set the output.
* The \*\_conv\_params arguments (replace the \* with either freq, amp, or phase) are used to pass in parameters to the class specified by the \*\_conv\_class argument when it is instantiate by the internals of the labscript API and the internals of BLACS. This argument defaults to None, in which case no parameters are passed in. Otherwise it should take the form of a dictionary, and can contain anything you like as long as it can be represented as a string (you must be able to execute in a Python terminal repr({…my dictionary here…})). The contents of the dictionary will depend on the specific implementation of the unit conversion class.

### Methods

**setfreq(t,value,units=None)**

This method sets the frequency of the DDS to the value specified, in the units given (if specified) at the time t.

* The t argument specifies the time at which to set the frequency to the value given. This time is given in seconds, as a float or integer.
* The value argument specifies the value to set the frequency to (in Hz).
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units (Hz). If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (freq\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**setamp(t,value,units=None)**

This method sets the amplitude of the DDS to the value specified, in the units given (if specified) at the time t.

* The t argument specifies the time at which to set the amplitude to the value given. This time is given in seconds, as a float or integer.
* The value argument specifies the value to set the amplitude to.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like Vpp or arbitrary units, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (amp\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**setphase(t,value,units=None)**

This method sets the phase of the DDS to the value specified, in the units given (if specified) at the time t.

* The t argument specifies the time at which to set the phase to the value given. This time is given in seconds, as a float or integer.
* The value argument specifies the value to set the phase to (in degrees).
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units (degrees). If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (phase\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**enable(t)**

This method enables the output of the DDS is it supports this capability. If a gate was specified at instantiation time, this calls the go\_high method of the DigitalOut.

* The t argument specifies the time at which to enable the DDS output. This time is given in seconds, as a float or integer.

**disable(t)**

This method disables the output of the DDS is it supports this capability. If a gate was specified at instantiation time, this calls the go\_low method of the DigitalOut.

* The t argument specifies the time at which to disable the DDS output. This time is given in seconds, as a float or integer.

### Properties

**frequency**

This property returns the AnalogQuantity used to control the frequency of the DDS. This allows you to directly access the methods of AnalogQuantity to control the frequency of the DDS.

**amplitude**

This property returns the AnalogQuantity used to control the amplitude of the DDS. This allows you to directly access the methods of AnalogQuantity to control the amplitude of the DDS.

**phase**

This property returns the AnalogQuantity used to control the phase of the DDS. This allows you to directly access the methods of AnalogQuantity to control the phase of the DDS.

## Class *StaticDDS* (subclasses *Device*)

The StaticDDS class is used to instantiate DDS outputs on devices that do not support changing the DDS state during buffered execution. The StaticDDS may also have a gate specified, to enable/disable the DDS output. Convenience methods are provided to enable/disable the gate.

### Connection Table Instantiation

**StaticDDS(name, parent\_device, connection, digital\_gate={},   
 freq\_limits=None, freq\_conv\_class=None, freq\_conv\_params={},   
 amp\_limits=None, amp\_conv\_class=None, amp\_conv\_params={},   
 phase\_limits=None, phase\_conv\_class=None, phase\_conv\_params={})**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘dds 2’ for a DDS output of the NovaTech DDS9m.
* The digital\_gate argument can be used to link a digital output of another device, which is set up to turn on/off the output of the DDS, to the enable/disable methods of this object. For example, a digital output of an NI device might be setup to control a RF switch on the output of a DDS on a NovaTech DDS9m channel. The form of this argument should be a dictionary, with keys ‘device’ and ‘connection’ which correspond to the arguments for parent\_device and connection you would pass were you instantiating the digital output using the DigitalOut class. An example dictionary is {‘device’:ni\_pcie\_6363\_0, ‘connection’:’port0/line12’}, where ni\_pcie\_6363\_0 is a device already instantiated in the connection table. **You should not instantiate the gate as a DigitalOut directly. Labscript will do this internally for you.**
* The \*\_limits arguments (replace the \* with either freq, amp, or phase) are an optional argument that specifies the lower and upper output value for this output. If output is commanded outside of the specified limits during the experiment logic, an error will be thrown. The default is None, which imposes no limits other than those determined by the hardware. If you wish to specify limits, this argument should be set to a tuple, of the form (lower,upper) where lower and upper are specified as floats or integers.
* The \*\_conv\_class arguments (replace the \* with either freq, amp, or phase) are used to specify the Python class to be used for unit conversion (see the unit conversion documentation for how to create such classes). It should be a reference to a Python class (not an instance of a Python class). Specifying a class here allows you to use different units when calling the methods to set the output.
* The \*\_conv\_params arguments (replace the \* with either freq, amp, or phase) are used to pass in parameters to the class specified by the \*\_conv\_class argument when it is instantiate by the internals of the labscript API and the internals of BLACS. This argument defaults to None, in which case no parameters are passed in. Otherwise it should take the form of a dictionary, and can contain anything you like as long as it can be represented as a string (you must be able to execute in a Python terminal repr({…my dictionary here…})). The contents of the dictionary will depend on the specific implementation of the unit conversion class.

### Methods

**setfreq(value,units=None)**

This method sets the frequency of the DDS to the value specified, in the units given (if specified) at the time at during program time of buffered devices (immediately before the experiment shot begins).

* The value argument specifies the value to set the frequency to (in Hz).
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units (Hz). If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (freq\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**setamp(value,units=None)**

This method sets the amplitude of the DDS to the value specified, in the units given (if specified) at during program time of buffered devices (immediately before the experiment shot begins).

* The value argument specifies the value to set the amplitude to.
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units. This is usually an SI unit like Vpp or arbitrary units, and is the first unit in the list of units for the output in BLACS. If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (amp\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**setphase(value,units=None)**

This method sets the phase of the DDS to the value specified, in the units given (if specified) at the time at during program time of buffered devices (immediately before the experiment shot begins).

* The value argument specifies the value to set the phase to (in degrees).
* The units argument specifies the units for the value given. This defaults to None, in which case the units correspond to the device hardware units (degrees). If it is not None, it should correspond to the unit you wish to use, as specified in the unit conversion class used at the instantiation of the DDS (phase\_conv\_class argument). For more details, see the separate unit conversion documentation file.

**enable(t=None)**

This method enables the output of the DDS is it supports this capability. If a gate was specified at instantiation time, this calls the go\_high method of the DigitalOut.

* The t argument specifies the time at which to enable the DDS output. This time is given in seconds, as a float or integer. By default the time is None, which should only be left like this if using the StaticDDS on the Phase Matrix QuickSyn device, as this device has an internal gate that can only get set once during program time, immediately before the experiment shot begins.

**disable(t=None)**

This method disables the output of the DDS is it supports this capability. If a gate was specified at instantiation time, this calls the go\_low method of the DigitalOut.

* The t argument specifies the time at which to disable the DDS output. This time is given in seconds, as a float or integer. By default the time is None, which should only be left like this if using the StaticDDS on the Phase Matrix QuickSyn device, as this device has an internal gate that can only get set once during program time, immediately before the experiment shot begins.

## Class *AnalogIn* (subclasses *Device*)

The AnalogIn class is used to instantiate Analog Inputs on acquisition devices like the NI PCIe-6363

### Connection Table Instantiation

**AnalogIn(name, parent\_device, connection)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘ai2’ for an Analog Input on the NI PCie-6363.

### Methods

**acquire(label, start\_time, end\_time)**

This method sets the frequency of the DDS to the value specified, in the units given (if specified) at the time at during program time of buffered devices (immediately before the experiment shot begins). The function returns the duration of the acquisition, which is the difference between the start and end times specified.

* The label argument specifies a unique label for this acquisition. You can use this name to reference the acquired trace within the lyse framework.
* The start\_time argument specifies the time, in seconds, at which the acquisition should start from. This should be specified as an integer of float.
* The end\_time argument specifies the time, in seconds, at which the acquisition should end. This should be specified as an integer of float.

## Class *DigitalOut* (subclasses *Output*)

The DigitalOut class is used to instantiate a digital output on a device.

### Connection Table Instantiation

**DigitalOut(name, parent\_device, connection)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘port0/line31’ for a digital output on the NI PCie-6363.

### Methods

**go\_high(t)**

This method sets the output to a logical high state (or on state) at the time specified by the argument t.

* The t argument the time at which to turn the output on (set to a logical high state). It should be specified in seconds, as an integer or float.

**go\_low(t)**

This method sets the output to a logical low state (or off state) at the time specified by the argument t.

* The t argument the time at which to turn the output off (set to a logical low state). It should be specified in seconds, as an integer or float.

## Class *Shutter* (subclasses *DigitalOut*)

The Shutter class is almost equivalent to the DigitalOut class. The defining difference is that you can specify a shutter open/close delay at instantiation time, which is taken into account automatically when calling the provided convenience methods open() and close(). This allows you to command that the shutter be closed at a specified time, without having to worry about the delay between the digital signal being sent and the shutter fully closing. Furthermore, the delays times specified at instantiation could be in the form of a runmanager global, which are automatically updated by a user written calibration script. This would allow shutter open/close delays to be automatically updated periodically without needing to edit each experiment script.

### Connection Table Instantiation

**Shutter(name, parent\_device, connection, delay=(0,0))**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘port0/line31’ for a digital output on the NI PCie-6363.
* The delay argument specifies the open and close delay as a tuple. The delays should be specified as the time, in seconds, it takes to open/close the shutter after the digital signal is received by the shutter. An example delay argument would be (3e-3,3.5e-3), which sends the signal to open the shutter 3ms early and the signal to close the shutter 3.5ms early. This then causes the shutter to be fully open/closed at the times specified to the open() and close() methods.

### Methods

**open(t)**

This method is used to open the shutter at the specified time. It will take the open delay time specified at instantiation, into account. Note that is the shutter is commanded very close to zero, the delay time may not be able to be accounted for if that results in the output to the shutter changing before the experiment starts (eg. Using the delay specified in the example instantiation above, the shutter commanded to be open at t=0 would require the digital line to change state at t=-3ms, which is impossible).

* The t argument specifies the time at which to open the shutter. This function is the same as calling go\_high(t-open\_delay) on a DigitalOut.

**close(t)**

This method is used to close the shutter at the specified time. It will take the close delay time specified at instantiation, into account. Note that is the shutter is commanded very close to zero, the delay time may not be able to be accounted for if that results in the output to the shutter changing before the experiment starts (eg. Using the delay specified in the example instantiation above, the shutter commanded to be closed at t=0 would require the digital line to change state at t=-3.5ms, which is impossible).

* The t argument specifies the time at which to open the shutter. This function is the same as calling go\_low(t-close\_delay) on a DigitalOut.

## Class *Camera* (subclasses *DigitalOut*)

The Camera class uses a DigitalOut to trigger a camera under BIAS control to capture an image. Instantiating a Camera object in the lab (BLACS) connection table is necessary before you can use the camera in BIAS.

### Connection Table Instantiation

**Shutter(name, parent\_device, connection, BIAS\_port, serial\_number,  
 SDK, effective\_pixel\_size, exposuretime, orientation)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘port0/line31’ for a digital output on the NI PCie-6363.
* The BIAS\_port argument specifies the network port on which to communicate with the copy of BIAS that is controlling this camera. BIAS will read this information from the connection table and automatically become available on the port specified. It should be an integer between 1024 and 65535. This port must not be in use by any other programs on the pc running BIAS. The network hostname for the PC running BIAS is specified through the BLACS GUI, and is not contained in the connection table.
* The serial\_number argument specifies the serial number for the camera. This information is used by BIAS to uniquely identify the specific camera you wish to access. This should be specified as a string or hexadecimal integer.
* The SDK argument specifies the name of the LabVIEW class BIAS will use to communicate with this camera.
* The effective\_pixel\_size argument specifies the size of each pixel, in units of metres. The labscript suite expects you to have square pixels in your camera. It should be specified as an integer or float.
* The exposuretime argument specifies the length of time the camera will expose the sensor for each image it takes. Note that this is a camera specific parameter, but shorter exposure times can be achieved in certain situations. For instance, in absorption imaging, the exposure time is really defined by how long you illuminate your camera sensor, rather than the length of time the camera exposes the sensor.
* The orientation argument is really just meta data that affects where the images are stored within the HDF file. Images will be stored in the /images/orientation/name/frametype/ group, where name and frametype are specified during the call to the expose method.

### Methods

**expose(name, t, frametype)**

This method is used to command the camera to take an image at the specified time.

* The name argument specifies a name for the image to be taken. It does not have to be unique, but the combination of name and the frametype argument must be unique. This name will be used to access the set of frames in lyse. The name should be specified as a string.
* The t argument specifies the time at which to begin the exposure, in seconds. It should be specified as a float or integer.
* The frametype argument specifies the type of frame to be taken. This is important, for example, when using multiple frames to form an absorption image. The frame type informs BIAS how to combine the images into an absorption image. The frame type should one of the elements of this list: ['atoms','flat','dark','fluoro','clean'].

The dark, flat and atoms frame types are used in absorption imaging. They correspond to:

dark: dark field image where the exposing laser is off.  
flat: exposing lasers on, but no atoms visible.

atoms: exposing lasers on, atoms present to image.

The other frame types are used to tell BIAS not to process the image as part of an absorption image calculation.

fluoro: This can be used when taking a fluorescent image

clean: This can be used to take a dummy frame on cameras where the sensor needs to be read   
 out to clean it.

## Class *WaitMonitor* (subclasses *Trigger*)

The WaitMonitor class subclasses the Trigger class (effectively a DigitalOut class, and not described in this documentation as it is not used directly by the user). The wait monitor is used to instruct BLACS how to monitor the progress of *waits* and resume execution if a *wait* reaches a specified timeout. A WaitMonitor object must be instantiated if you wish to use the wait method described later in this document.

### Connection Table Instantiation

**WaitMonitor(name, parent\_device, connection,   
 acquisition\_device, acquisition\_connection,   
 timeout\_device, timeout\_connection)**

* The name argument is used to create a variable in the Python namespace (with the name provided) for the object created by this call. The variable is placed in the \_\_builtins\_\_ module, and is accessible to all imported Python modules. This parameter should be a string.
* The parent\_device argument specifies the device that this output is connected to. It should be a reference to a device object already instantiated in the connection table.
* The connection argument specifies the output or port of the device that this output is using. This will be a string, and as an example could be ‘port0/line0’ for a digital output on the NI PCie-6363.
* The acquisition\_device argument specifies the device which contains the counter you wish to use to monitor the *wait* periods, and time their length (necessary to successfully perform analog acquisitions after a wait). This should be specified as a device object already instantiated in the connection table. Note that the type of the device object (determined by the labscript API class used to instantiate it) must have support within BLACS to monitor *waits.* Currently the only device supported is the NI PCIe-6363.
* The acquisition\_connection argument specifies the input port on the acquisition\_device that the counter is connected to. For example, this would be ‘ctr0’ on the NI PCIe-6363.
* The timeout\_device argument specifies the device with the **software timed** digital output that is used to resume the master pseudoclock should the timeout be reached. ). This should be specified as a device object already instantiated in the connection table. Note that this device must currently be the same as the device specified by acquisition\_device argument due to limitations within BLACS.
* The timeout\_connection argument specifies the output port of the timeout\_device used to resume the master pseudoclock. This should be specified as a string, and an example of a software timed output would be ‘PFI1’ on the NI PCIe-6363.

## Method *start*

The start method is used to indicate the end of the connection table and the start of the experiment logic. See the start subsection in the experiment logic section of this documentation for an example use case.

**start()**

## Method *stop*

The stop method is used to indicate the end of the experiment logic. See the stop subsection in the experiment logic section of this documentation for an example use case.

**stop(t)**

This method is used to indicate the end of the experiment

* The t argument specifies the time, in seconds, at which the experiment should stop. The final values commanded from outputs are held until the time specified here. This argument should be specified as an integer of float. The time must be at least one minimum length pseudoclock cycle away from the time of the last commanded output.

## Method *wait*

The wait method is used to command the pseudoclocks to pause until resumed by an external trigger, or a timeout is reached.

**wait(label, t, timeout=5)**

This method is used to command a *wait*, and returns the time it takes for all pseudoclocks to resume execution once the wait has completed.

* The label argument specifies a unique name for this *wait*. It should be specified as a string.
* The t argument specifies the time, in seconds, at which the experiment should begin the wait. Note that as far as labscript is concerned, the *wait* takes no time to complete.
* The timeout argument specifies the maximum length of the *wait*, in seconds. After this time, the pseudoclocks will be commanded to resume by BLACS. This argument should be specified as a float or integer as defaults to 5 seconds.