Adding a new device to `labscript`:

Chris Billington

January 31, 2014

Contents

1 Introduction

`Labscript` is a compiler. It translates high-level instructions given by Python function calls into low level instructions suitable for programming into specific devices. Clearly, the form of these low-level instructions is device-specific, and as such code to produce these instructions must be written in order for labscript to work with a new device.

Despite this, many devices are similar, and instructions given to them in a labscript experiment are processed in similar ways. For example, all analog output devices that are externally pseudoclocked will need their outputs interpolated to the times that the clock ticks. Because of this, two analog output devices can benefit from code re-use in labscript.

Depending on how similar your new device is to an existing one, and how well it fits the pseudoclock architecture, you may need to write a lot of code, or very little. The aim of this document is to outline the structure of the labscript module by describing the two main device classes (Sec. 2), and how they fit into the different stages in compilation. With this knowledge a developer writing support for a new device can choose which existing class it is most appropriate to subclass for their device, and what data processing they should write themselves and what they should re-use. An example of each class is provided in Sec. 3

This document assumes familiarity with object oriented programming in Python.

2 Class relationships

Devices are objects in `labscript`. In fact, they are all subclassed from the `labscript.labscript.Device` class. This class contains the logic for adding the device to the compiler 'inventory', so that the compiler knows about it, and putting it into the builtin namespace, making each device object available to the user interpreter-wide. It also contains a few other methods pertaining to connecting devices to each other. See its class definition in `labscript.labscript` to see precisely what all device classes have in common.

Writing device compatibility for `labscript` comprises subclassing `Device` or — more likely — one of its subclasses, and overriding or adding methods as appropriate. The most important method is the `generate_code(hdf5_file)` one. It will be called at compilation time and is responsible for writing low-level instructions for that device to the given HDF5 file. The instructions should be saved as one or more datasets or attributes in the group `'/devices/<device_name>'`, where `<device_name>` is the name of the instance of that device as used in labscript (stored in `self.name`).

Subclasses should generally call their superclass's `__init__` and `generate_code` methods when they are being overridden. Usually, calling the superclass's `generate_code` method does most of the instruction processing for you, at which point the object will have its instructions stored as instance attributes ready for your new code to access and translate into device-specific instructions. See the existing device classes (in `labscript.labscript`) for examples of overwritten `__init__` and `generate_code` methods.

There are two main subclasses of `Device` that you will be likely to subclass: `PseudoClock` and `IntermediateDevice`.

2.1 `labscript.labscript.Pseudoclock`

The first is `labscript.labscript.Pseudoclock`. Most of the control flow during compilation is dictated by methods in this class. Generally, all devices capable of providing their own timing, or providing clocking signals to other devices, should be a `PseudoClock`[1]. A `PseudoClock` object expects to have children which are `Output`s and `IntermediateDevice`s (each with its own `Output`s), and during compilation it calls methods on them to collect data on what the `Output`s have been asked to do in the experiment. From this information it constructs a clocking signal, stored as `self.clock` in an intermediate format as a dictionary. Implementing a pseudoclock involves converting this structure to whatever format the device itself actually requires for programming, and saving the results to the HDF5 file. An example of a clocking signal is:

```
self.clock = [{'start': 0,        'reps': 1,   'step': 1e-3, 'slow_clock_tick':True},
               'WAIT',
              {'start': 1e-3,     'reps': 1,   'step': 1e-6, 'slow_clock_tick':True},
              {'start': 1.001e-3, 'reps': 999, 'step': 1e-6, 'slow_clock_tick':False},
              {'start': 2e-3,     'reps': 1,   'step': 1e-3, 'slow_clock_tick':True}]
```

This means that the pseudoclock should tick once with a period of 1 ms (0.5 ms each high and low), on both the fast and slow clock outputs[2]. It should then halt execution and wait for an external trigger[3]. After that, it should tick once with a period of 1 μs on both outputs, then only the fast clock 999 times at a rate of 1 MHz. It should then tick once more on both clocks with a 1 ms period. The `'start'` key is not needed generally for actually producing signals, and is used only in `labscript` to provide a timestamp in error messages pertaining to producing clocking signals.

So in its simplest form, adding support for a new pseudoclock involves converting this list of dictionaries (or `'WAIT'` strings) into a list of strings to be piped down a serial connection, a list of parameters to be passed to C function calls, or whatever format is is easiest to read and then program into the device once the HDF5 file is being read by `BLACS`.

See `labscript.labscript.PulseBlaster` and `labscript.labscript.PineBlaster` for examples of pseudoclock classes. The former has DDS and digital outputs as well as providing

[1] As is always the case, exceptions are possible but are discouraged.

[2] The `PseudoClock` is assumed to have at most two outputs, one that ticks at a subset of the times that the other ticks. The one that ticks less often is called the *slow clock*. We use this functionality to have more devices clocked off the same pseudoclock than would otherwise be possible. The `PseudoClock` class inserts a slow clock tick for every single-value instruction on an output device, as well as a single slow clock tick at the beginning of ramps. However the slow clock does not tick during ramps. This means that devices attached to the slow clock cannot execute function ramps. If you wish to implement a pseudoclock with only one output, you may simply ignore this distinction and produce hardware instructions for only a fast clock signal. See `labscript.labscript.PineBlaster` for an example of this

[3] Again, your device need not support this, and you can have it simply throw an error upon encountering such an instruction.

a clocking signal, so its code generation is quite involved. The latter produces only a signal clock signal and so is fairly simple. The `PineBlaster` is used as an example in Sec. 3

2.2 `labscript.labscript.IntermediateDevice`

This is the class that you are likely to subclass most often. It represents devices that have a `PseudoClock` as their parent device and `Output`s (analog, digital or DDS) or inputs[4] as their child devices. So for example, National Instruments cards with digital and analog outputs and analog inputs are implemented in `labscript` as `IntermediateDevice`s. These devices are programmed with their output values, but receive their timing from the parent pseudoclock.

By the time an `IntermediateDevice`'s `generate_code` method is called during compilation, the parent `PseudoClock` has already collected the times at which the child `Outputs` change, generated its clocking data, and left behind some useful instance attributes on each `Output` pertaining to what their output values should be at each clock tick. These attributes are what you should use in the `generate_code` method of your `IntermediateDevice`.

You can access the child `Outputs` of an `IntermediateDevice` with `self.child_devices`. Each of these outputs then has an attribute `output.raw_output`, which is simply a `numpy` array of voltages, or in the case of digital outputs, a `numpy` array of ones and zeros[5]. In the case of a `DDS` output, the arrays of amplitudes, frequencies and phases are stored as attributes to three Output-like objects, `output.frequency`, `output.amplitude` and `output.phase`. Each of these is an `AnalogQuantity`[6] and similarly have `raw_output` attributes. DDSs may also have a digital gate for turning them on and off, if so it is stored as `output.gate` and similarly has a `raw_output` attribute for its values.

Child devices may also be `AnalogInputs`. In this case, each input object has an attribute `inputs.acquisitions`, which is a list of dictionaries with the acquisition times that have been requested for that channel.

Your job in implementing an IntermediateDevice is to do any processing necessary on these values, such as converting voltages to integers in some range, converting frequencies to hexadecimal values, packing sets of Booleans into integers, or whatever your device requires. You should also do any error checking that is specific to your device, like checking if the number of instructions is within the capabilities of the device, as are the values themselves. You should raise an informative `labscript.labscript.LabscriptError` in the event that something is not right.

See `labscript.labscript.NIBoard` or `labscript.labscript.NovaTechDDS9M` for examples of `IntermediateDevicess`. The `NIBoard` is used as an example in 3.

3 Examples

3.1 `PseudoClock` example

The following is an actual pseudoclock in use in labscript, commented here to explain what each bit is for. It looks long here but is actually only about 40 lines of actual code. Other pseudoclocks may be more complex, as is `labscript.labscript.PulseBlaster`, owing to its direct digital and DDS outputs as well as a more complex method of programming.

[4] Currently only `AnalogIn`

[5] You can use the `labscript.labscript.bitfield` function to convert a list of these arrays of ones and zeros into a single array of integers (bitfields), if necessary.

[6] Which is a subclass of `Output` and is identical to an `AnalogOut` in every way except for the name. This is so that code can tell the difference between analog outputs that correspond to actual physical outputs, and those that exist only to store the frequency, amplitude or phase data of a DDS.

3.2 `IntermediateDevice` example

The following is a class used in `labscript` for National Instruments cards. This class is not used directly, it is instead subclassed further for specific National Instruments cards with varying numbers of analog and digital outputs, and analog inputs. But it is a more informative example than the more specific classes, as it is the one that contains most of the work of the `generate_code` function.

```python
from labscript import *

class NIBoard(IntermediateDevice):
    # Set what types of child devices this IntermediateDevice can have:
    allowed_children = [AnalogOut, DigitalOut, AnalogIn]

    # Some device specific parameters:
    n_analogs = 4
    n_digitals = 32
    digital_dtype = uint32

    # The maximum rate that the outputs can update:
    clock_limit = 500e3

    # A name for the device:
    description = 'generic_NI_Board'

    def __init__(self, name, parent_device, clock_type, clock_terminal, MAX_name=None, acquisition_rate=0):
        # We pass the relevant parameters to the parent class's __init__ function:
        IntermediateDevice.__init__(self, name, parent_device,clock_type)

        # This implementation only allows analog aquisitions at a constant rate
        self.acquisition_rate = acquisition_rate
        self.clock_terminal = clock_terminal
        self.MAX_name = name if MAX_name is None else MAX_name
        self.BLACS_connection = self.MAX_name

    def convert_bools_to_bytes(self, digitals):
        """converts digital outputs to an array of bitfields stored
        as self.digital_dtype"""
        outputarray = [0]*self.n_digitals
        for output in digitals:
            # output.connection is the string that the user provided at
            # instantiation of the output object.  It is, by convention
            # here, port0/line<n>, where <n> is an integer from 0 to 31
            # indicating which digital output it is:
            port, line = output.connection.replace('port','').replace('line','').split('/')
            port, line  = int(port),int(line)
            if port > 0:
                raise LabscriptError('Ports > 0 on NI Boards not implemented. ' +
                                     'Please use port 0, or file a feature request ' +
                                     'at redmine.physics.monash.edu.au/labscript.')
            # Pack all the 1d arrays of digital output values into their appropriate spot in a list:
            outputarray[line] = output.raw_output
        # Convert this list of arrays of digital values into
        # integer bitfields (the bitfield function is located in
        # labscript.labscript)
        bits = bitfield(outputarray,dtype=self.digital_dtype)
        return bits

    def generate_code(self, hdf5_file):
        # By the time this function is called during compilation, most
        # of the work has already been done.  Calling the parent class's
        # generate_code method actually does nothing at the moment,
        # but this may change in the future, so you should call it anyway.
        Device.generate_code(self, hdf5_file)

        # Now we collect up all the output and input objects from self.child_devices:
        analogs = {}
        digitals = {}
        inputs = {}
        for device in self.child_devices:
            if isinstance(device,AnalogOut):
                analogs[device.connection] = device
            elif isinstance(device,DigitalOut):
                digitals[device.connection] = device
            elif isinstance(device,AnalogIn):
                inputs[device.connection] = device
            else:
                raise Exception('Got unexpected device.')

        # Now we collect up all the output.raw_output arrays from the
        # analog outputs, and load them into a numpy recarray:
        analog_out_table = empty((len(self.parent_device.times),len(analogs)), dtype=float32)
        analog_connections = analogs.keys()
        analog_connections.sort()
        analog_out_attrs = []
        for i, connection in enumerate(analog_connections):
            output = analogs[connection]
            # A bit of error checking:
```

```
            if any(output.raw_output > 10 )  or any(output.raw_output < -10 ):
                # Bounds checking:
                raise LabscriptError('%s %s '%(output.description, output.name) +
                                  'can only have values between -10 and 10 Volts, ' +
                                  'the limit imposed by %s.'%self.name)
            # Put the 1D array of voltages into the table:
            analog_out_table[:,i] = output.raw_output
            # Record the output terminal name to an attribute, so that
            # BLACS knows which ones to program:
            analog_out_attrs.append(self.MAX_name +'/'+connection)

        # Now we make a numpy recarray of all the analog input requests:
        input_connections = inputs.keys()
        input_connections.sort()
        input_attrs = []
        acquisitions = []
        for connection in input_connections:
            input_attrs.append(self.MAX_name+'/'+connection)
            for acq in inputs[connection].acquisitions:
                # Each acquisition request is a dictionary with the
                # following data, we're just putting them all in a list
                # along with the input channel they correspond to:
                acquisitions.append((connection,acq['label'],acq['start_time'],acq['end_time'],
                                     acq['wait_label'],acq['scale_factor'],acq['units']))
        # The 'a256' dtype below limits the string fields to 256
        # characters. Can't imagine this would be an issue, but to not
        # specify the string length (using dtype=str) causes the strings
        # to all come out empty.
        acquisitions_table_dtypes = [('connection','a256'), ('label','a256'), ('start',float),
                                     ('stop',float), ('wait label','a256'),('scale factor',float), ('units','a256')]
        acquisition_table= empty(len(acquisitions), dtype=acquisitions_table_dtypes)
        # OK, now we're putting them all into the numpy array:
        for i, acq in enumerate(acquisitions):
            acquisition_table[i] = acq

        # And finally for digital output:
        digital_out_table = []
        if digitals:
            # We convert the arrays of boolean values to a single
            # array of bitfield integers. This is how many devices need
            # their digital values programmed, though as it happens,
            # the National Instruments cards we use do not. So actually
            # this is just for storage in the HDF5 file and this process
            # is reversed when BLACS reads the data later.
            digital_out_table = self.convert_bools_to_bytes(digitals.values())

        # Create the required group for this device in the HDF5 file:
        grp = hdf5_file.create_group('/devices/'+self.name)

        # Save the analog output table, if it exists (subclasses may have zero outputs and hence an empty table):
        if all(analog_out_table.shape): # Both dimensions must be nonzero
            analog_dataset = grp.create_dataset('ANALOG_OUTS',compression=config.compression,data=analog_out_table)
            # Save the corresponding list of channels:
            grp.attrs['analog_out_channels'] = ', '.join(analog_out_attrs)
        # Save the digital output table, if it exists:
        if len(digital_out_table): # Table must be non empty
            digital_dataset = grp.create_dataset('DIGITAL_OUTS',compression=config.compression,data=digital_out_table)
            # Save the corresponding list of channels:
            grp.attrs['digital_lines'] = '/'.join((self.MAX_name,'port0','line0:%d'%(self.n_digitals-1)))
        # Save the table of acquisitions, if it exists:
        if len(acquisition_table): # Table must be non empty
            input_dataset = grp.create_dataset('ACQUISITIONS',compression=config.compression,data=acquisition_table)
            # Save the channels for analog input:
            grp.attrs['analog_in_channels'] = ', '.join(input_attrs)
            # Save the acquisition rate for analog input:
            grp.attrs['acquisition_rate'] = self.acquisition_rate
        # Save the setting for which terminal this card should expect
        # a clock input on, provided by its parent pseudoclock. BLACS
        # needs this in order to configure the device to respond to the
        # clock ticks:
        grp.attrs['clock_terminal'] = self.clock_terminal

```