How to add a new type of hardware device to BLACS

January 31, 2014

Contents

1 Introduction

This tutorial covers the basics of adding a new hardware device type to BLACS. Existing device types include the PulseBlaster, NovatechDDS9m, NI PCIe 6363 and NI PCI 6733. These classes are used by BLACS to generate the unique tab for each device on the BLACS front panel. Adding support for your device to BLACS does not add support to labscript, which will also need to be done if you wish to use the device in a buffered sequence (see labscript documentation).

This tutorial assumes a working knowledge of Python. Some advanced sections also require a working knowledge of PySide (very similar to PyQt) and QtDesigner. If are not familiar with one or more of these, please attempt a tutorial on the subject before continuing with this guide!

1.1 Creating the Device Classes

All device classes as stored in pythonlib/BLACS/hardware_interfaces. You should create a new file for your device in this folder, with an appropriate name (the name should be the lowercase version of the device class used in labscript). We shall refer to the name in this tutorial as `my_device.py`).

The first thing you need to do in your empty python file is import the required packages and classes. The code below imports the BLACS backend classes for the tab state machine (we'll cover these more as we go). You can also optionally import PySide if you wish to add more PySide widgets to the interface yourself (see section 3.3.13)

```
from BLACS.tab_base_classes import Worker, define_state
from BLACS.tab_base_classes import MODE_MANUAL, MODE_TRANSITION_TO_BUFFERED
from BLACS.tab_base_classes import MODE_TRANSITION_TO_MANUAL, MODE_BUFFERED
from BLACS.device_base_class import DeviceTab
```

Next we need to define two classes. The first class will handle the GUI, and related events. The second will handle the communication with the actual hardware. The first class name (hence forth known as the device class) should be the lower case version of the device class shown in the entry in the connection table for an instance of your device. It should also be the same name as your python file. You may call the second class (hence forth known as the worker class) whatever you wish, though it is advisable to use something understandable!

In this tutorial we will use `my_device` and `MyWorker` as our class names.

```
class my_device(DeviceTab):
    pass

class MyWorker(Worker):
    pass
```

Note: We have used `pass` here as a placeholder. You can assume that the `pass` statements will be removed once the tutorial inserts code at the same indentation level.

2 Implementing a simple Device Class

The device class handles the creation of the GUI and the interaction with the GUI and the Queue Manager. Most of this is handled by the `DeviceTab` class you are subclassing. We'll first discuss the minimum requirements for subclassing `DeviceTab`, followed by an in depth explanation of the internal workings of `DeviceTab` and details on advanced implementation possibilities.

When the device tab is instantiated, the functions described in the next sections are run in the following order:

```
    self.initialise_GUI()
    self.restore_save_data(settings_dictionary)
    self.initialise_workers()
```

2.1 Overriding the `initialise_GUI` function

The `initialise_GUI` function is where you define the output capabilities of your device and generate the graphical interface for manual control of the device through the front panel. The `DeviceTab` class you are subclassing has a lot of function to help you do this. Here we'll walk through a simple example, but for full details see the `DeviceTab` reference in section 3.

The first step is to override the `initialise_GUI` function and define some capabilities of the device:

```python
def initialise_GUI(self):
    # Capabilities
    self.num_DDS = 2
    self.num_DO = 12
    self.ao_base_units     = 'V'
    self.ao_base_min       = -10.0
    self.ao_base_max       =  10.0
    self.ao_base_step      =  0.001
    self.ao_base_decimals  =  3
    self.dds_base_units    = {'freq':'Hz',         'amp':'Vpp',  'phase':'Degrees'}
    self.dds_base_min      = {'freq':0.3,          'amp':0.0,    'phase':0}
    self.dds_base_max      = {'freq':150000000.0, 'amp':1.0,    'phase':360}
    self.dds_base_step     = {'freq':1000000,      'amp':0.01,   'phase':1}
    self.dds_base_decimals = {'freq':1,            'amp':3,      'phase':3}
```

This should look pretty straight forward, we are just storing the number of DDS and digital outputs, followed by the capabilities of the DDSs of the device in **base units**. Base units are considered to be SI units like Volts, Hz, etc. Base units are not necessarily the same as hardware units. If your hardware is not programmed in SI units, we recommend converting from SI to the required unit **within** the worker process methods so that the user never has to deal with the arbitrary units the device manufacturer opted for. All of this information is for your own use, and does not have to follow any particular naming convention, nor does it have to be stored as an instance atrribute using `self`.

Following this you should define 1-3 dictionaries which define the properties of the digital, analog and DDS output channels respectively. If your devices does not have one or more of these types of outputs, you do not need to create the dictionary of properties for it.

For digital outputs, the dictionary should be of the form:

```
    digital_properties = {'hardware_channel_1':{},
                          'hardware_channel_2':{}
                         }
```

where `hardware_channel_x` is the name of the channel connection as specified in your `labscript` implementation. For instance PulseBlasters would use Flag 1, Flag 2, etc. while NI PCIe 6363s would use port0/line0, port0/line1, etc.

For analog outputs, the dictionary should be of the form:

```
    analog_properties = {'hardware_channel_1':{'base_unit':self.ao_base_units,
                                               'min':self.ao_base_min,
                                               'max':self.ao_base_max,
                                               'step':self.ao_base_step,
                                               'decimals':self.ao_base_decimals
                                              },
                         'hardware_channel_2':{'base_unit':self.ao_base_units,
                                               'min':self.ao_base_min,
                                               'max':self.ao_base_max,
                                               'step':self.ao_base_step,
                                               'decimals':self.ao_base_decimals
                                              },
                        }
```

where, again, `hardware_channel_x` is the name of the channel connection as specified in your `labscript` implementation.

DDS properties are combination of three analog properties and a digital property. The DDS property dictionary should thus be of the form:

```
    # The creation of this dictionary could be simplified by one or
    # more for loops. However we chose to write everything out explicitly
    # here so that it is clear what is going on. Consult one of the already
    # implemented device classes to see how it could be done in fewer lines
    # of code (for example the PulseBlaster)
    dds_properties = {'hardware_channel_1':
                          {'freq':{'base_unit':self.dds_base_units['freq'],
                                   'min':self.dds_base_min['freq'],
                                   'max':self.dds_base_max['freq'],
                                   'step':self.dds_base_step['freq'],
                                   'decimals':self.dds_base_decimals['freq']
                                  },
                           'amp':{'base_unit':self.dds_base_units['amp'],
                                   'min':self.dds_base_min['amp'],
                                   'max':self.dds_base_max['amp'],
                                   'step':self.dds_base_step['amp'],
                                   'decimals':self.dds_base_decimals['amp']
                                  },
                                   'min':self.dds_base_min['phase'],
                                   'max':self.dds_base_max['phase'],
                                   'step':self.dds_base_step['phase'],
                                   'decimals':self.dds_base_decimals['phase']
                                  },
                           'gate':{}
                           },

                     'hardware_channel_2':
                          {'freq':{'base_unit':self.dds_base_units['freq'],
                                   'min':self.dds_base_min['freq'],
                                   'max':self.dds_base_max['freq'],
                                   'step':self.dds_base_step['freq'],
                                   'decimals':self.dds_base_decimals['freq']
                                  },
                           'amp':{'base_unit':self.dds_base_units['amp'],
                                   'min':self.dds_base_min['amp'],
                                   'max':self.dds_base_max['amp'],
                                   'step':self.dds_base_step['amp'],
                                   'decimals':self.dds_base_decimals['amp']
                                  },

                                   'min':self.dds_base_min['phase'],
                                   'max':self.dds_base_max['phase'],
                                   'step':self.dds_base_step['phase'],
                                   'decimals':self.dds_base_decimals['phase']
                                  },
                           'gate':{}
                           }
                    }
```

where, again, `hardware_channel_x` is the name of the channel connection as specified in your `labscript` implementation. Importantly, if you omit any of the freq, amp, phase or gate entries, it will be assumed that the DDS output does not have control over that parameter. For instance you may have a frequency source for which you cannot control the amplitude or phase, nor does it have an on/off control (gate). As such you would define your dds dictionary as:

```
    dds_properties = {'hardware_channel_1':
                           {'freq':{'base_unit':self.dds_base_units['freq'],
                                    'min':self.dds_base_min['freq'],
                                    'max':self.dds_base_max['freq'],
                                    'step':self.dds_base_step['freq'],
                                    'decimals':self.dds_base_decimals['freq']
                                   }
                            }
                      }
```

and only the frequency control will be displayed in the DDS widget.

Once the output property dictionaries are defined, you should call one or more of the following functions to create python objects (we'll refer to them as "output objects" or `AO`, `DO` or `DDS` objects for your outputs:

```
    self.create_digital_outputs(digital_properties)
    self.create_analog_outputs(analog_properties)
    self.create_dds_outputs(dds_properties)
```

The objects create exist behind the scenes, and will have automatically looked up relevant entries in the BLACS connection table to get their name and unit conversion class.

You should then create the widgets for each of these outputs. There are several ways you can do this, see sections 3.3.2, 3.3.5, 3.3.9 and 3.3.7. The simplest is to let the `DeviceTab` do it for you, as follows:

```
    dds_widgets,ao_widgets,do_widgets = self.auto_create_widgets()
```

Note that three dictionaries are always returned, regardless of whether you are using any digital, analog and dds outputs on your device. For instance, if you have not create any analog outputs as detailed above, the `ao_widgets` variable will contain an empty dictionary.

You must then place these widgets in the GUI tab for your device. Again, this can be done several ways (see sections 3.3.3 and 3.3.13), but the simplest is to do:

```
    self.auto_place_widgets(dds_widgets,do_widgets)
```

The order in which you pass the dictionaries of widgets determines the order they appear in the device tab. You can split up the dictionaries of widgets if you like, for instance to segregate some digital outputs from the others. You can also specify heading names, and a sorting function for each group of widgets you pass to this function (see section 3.3.3 for details).

If you wish to add additional GUI elements, you may do so before or after the call to `auto_place_widget`
You can access the Qt Layout which contains the main body of the tab using:

```
    self.get_tab_layout()
```

You can then insert/append Qt Widgets or layouts you have created either from code or by loading a Qt UI file created by Qt Designer.

Finally, you should specify whether you implementation will support “smart programming” and/or “remote value checking”. This can be done by calling the methods:

```
    self.supports_remote_value_check(False)
    self.supports_smart_programming(False)
```

where the single argument is `True` or `False` indicating whether support exists. For details on smart programming, see sections 3.3.19 and 5.3. For details on remote value checking, see section 4.3.

2.2 Overriding the `get_save_data` and `restore_save_data` functions

You may find you wish to save and restore some data from your tab when BLACS is closed or the tab is restarted. `DeviceTab` provides two functions to override to implement this behaviour:

- `get_save_data` should be implemented to return a dictionary of key:value pairs you wish saved.
- `restore_save_data` provides a dictionary of key:value pairs to restore as you wish.

The signatures of the functions are:

```
def get_save_data(self):
        # your code here
        pass

def restore_save_data(self, save_data):
        # your code here
        pass
```

The dictionary you return must meet the following requirements:

```
# This expression must be true
eval(repr(my_save_data)) == my_save_data
```

See the Camera device class implementation for an example.

2.3 Overriding the `initialise_workers` function

The `initialise_workers` function is used to tell the device Tab to launch one or more worker processes to communicate with the device. A device tab can have one or more associated worker processes, with one of them being identified as the *primary worker*. When the device tab enters one of the states inbuilt into `DeviceTab` (such as `transition_to_buffered`), the primary worker is communicated with first. Following this, communication with secondary worker processes commences one after the other. The order in which secondary processes are chosen to communicate with depends on the names of all worker processes and the number of worker processes. We simply iterate over a dictionary of worker processes, keyed by the worker process name (watch the PyCon "The mighty dictionary" video online for more information on Python dictionary ordering). If you require secondary worker processes to be communicated with in a specific order, please contact the developers and request this feature to be added.

To launch a worker process, simply call:

```
def initialise_workers(self):
    self.create_worker("my_worker", MyWorker)
```

where `my_worker` is a unique name for this worker process and `MyWorker` is the class name you wish to launch in the new worker process.

`create_worker` also takes a third, optional dictionary of keyword arguments to pass into the worker process (see section 3.3.10). A good piece of information to pass in would be the `BLACS_connection attribute` specified in the connection table. This parameter usually contains information specifying the physical data connection to the PC (Eg. COM12), and is available as an instance attribute of `DeviceClass`:

```
def initialise_workers(self):
    self.create_worker("my_worker", MyWorker,{'com_port':str(self.BLACS_connection))
```

You can access this value from within the Worker Class using `self.com_port`.

Finally, you need to set the primary worker. Do do this, simply write:

```
    self.primary_worker = "my_worker"
```

where `my_worker` is the name you used in the call to `create_worker`

2.4 Overriding the `start_run` function

If your device is a pseudoclock, you will need to override the `start_run` function. This function is called by the Queue Manager to begin a buffered experiment shot. **This function must be a state function (see section 4.1). You will also need to become familiar with the `statemachine_timeout_add` functionality to schedule a regular poll of the device before continuing with this section (see section 4.2).** Your implementation of `start_run` will be passed a queue which it will use to notify the Queue Manager when the experiment shot has finished running. **It is up to you to schedule a poll of your device to determine if the shot has completed.**

Below is the PulseBlaster implementation:

```
@define_state(MODE_BUFFERED,True)
def start_run(self, notify_queue):
    self.statemachine_timeout_remove(self.status_monitor)
    yield(self.queue_work(self.primary_worker,'pb_start'))
    self.statemachine_timeout_add(100,self.status_monitor,notify_queue)
```

In this implementation, we first stop the existing poll of the device status. We then ask the worker process to start the buffered shot. Once the shot has begun, we setup a regular poll of the pulseblaster every 100 ms which will notify the Queue Manager at the end of the shot using the `notify_queue`.

3 `DeviceTab` reference

3.1 Instance Attributes

These attributes are available to instances of `DeviceTab` (and its subclasses).

3.1.1 `BLACS_connection`

This attribute contains the **string**, from the connection table entry of this instantiated device, that indicates how this device is connected to the PC. The contents of the string is determined by the connection table entry in `labscript`. It is likely you will want to pass this value to each worker process you create as one of the arguments at instantiation of the worker process (see section 3.3.10). Note that the `labscript` device class may expose this setting to the user under a different name, for instance the keyword argument `com_port`.

3.1.2 `connection_table`

This attribute contains a reference to the connection table object instantiated when BLACS is started. This object is an instance of class `ConnectionTable` located in 'BLACS/connections.py'.

3.1.3 `device_name`

This attribute contains the name of the device, as specified in the connection table.

3.1.4 `error_message`

This contains the current error message displayed to the user for this device. If no error is present, it will be an empty string. This attribute is a Python property. This means, in theory, you can set modify the error message as you wish, using this attribute, and the changes will automatically be applied to the displayed message. We however recommend you only read from it. If you do wish to modify the error, we recommend appending to the existing error using the += operator. Note that "not_responding" error messages are not stored here, but are instead stored in the `_not_responding_error_message` and `not_responding_for` attributes.

3.1.5 `force_full_buffered_reprogram`

This attribute is passed to the worker process by `DeviceTab.transition_to_buffered`. In the example implementation shown for the worker function `transition_to_buffered` (see section 5.3), the value here is placed in the variable `fresh`. Set this attribute to `True` if you wish a full reprogram of the device buffer to occur at the start of the next buffered shot. If the device supports smart programming (see section 3.3.19), then this attribute will be set to False by `DeviceTab.transition_to_buf`
after each buffered sequence has been programmed.

3.1.6 `logger`

This attribute contains the Python logger object for this device. This can be used to warn the user of errors, or save debug information to the log file or terminal. Python logger objects have methods such as `info`, `debug`, `warning`, etc. which can be passed a string to log. See the Python logging module documentation for more details.

3.1.7 `mode`

This attribute stores the current mode of the device. It will be an integer, either 1, 2, 4 or 8 corresponding to one of the modes described in section 4.1. These modes are one of `MODE_MANUAL`, `MODE_TRANSITION_TO_BUFFERED`, `MODE_BUFFERED` and `MODE_TRANSITION_TO_MANUAL` The mode should only ever be set to one of these variables/values, and should only be set by an implementation of `DeviceTab.transition_to_buffered` or `DeviceTab.transition_to_manual`. You can read this mode at any time you wish.

3.1.8 `primary_worker`

This attribute should be set to the name of the primary worker process, defined when you call `create_worker` (see sections 3.3.10 and 3.2.2). This is generally done in your implementation of `create_worker` (see section 2.3)

3.1.9 `settings`

This attribute contains all the settings information for the Device Tab. It includes current values for all of the `AO`, `DO` and `DDS` objects instantiated (including current units, step size, etc.). It also includes the device name, connection table object and results from the last call to `DeviceTab.get_save_data`. You generally shouldn't need to directly access this attribute, as most of the information is exposed through other instance attributes and methods.

3.1.10 `state`

This attribute contains name of the current state function we are in. You should not modify the contents of this attribute. It is unlikely you will ever need to read from this attribute (we only document it here for completeness).

3.1.11 `_tab_text_colour`

This attribute contains the current colour of the displayed text in the BLACS tab of the notebook. This is black if there is no error, or red if there is. You may change the colour if you like by setting this attribute, however the colour may be later overwritten by the statemachine (for instance if an error occurs) Call `update_tab_text_colour()` after setting this attribute to apply the change.

3.1.12 `_ui`

This attribute contains all the Qt user interface (UI) objects that were loaded from the 'BLACS/tab_frame.ui' file. If you wish to modify the UI significantly, this is what you are going to need to access. However most people should never need to access this attribute.

3.2 Instance Methods to override

These methods should be overridden by your subclass of `DeviceTab`. See section 2 for more details.

3.2.1 `initialise_GUI`

This method should be used to create any `AO`, `DO` and `DDS` objects and associated widgets, and to add any custom elements to the graphical user interface (GUI). More details can be found in section 2.1

The signature of this method is:

```python
def initialise_GUI(self):
```

3.2.2 `initialise_workers`

This method should be used to create one or more worker processes using the `create_worker` method (see section 3.3.10). You should also store the primary worker name (see section 3.1.8) and secondary worker names (if any, see section 3.3.1). More details can be found in section 2.3

The signature of this method is:

```
def initialise_workers(self):
```

3.2.3 `get_save_data`

This method should return a dictionary of any custom save data you wish saved across instances of your device tab (see section 2.2 for more details). This method will be called by BLACS at the appropriate times.

The signature of this method is:

```
def get_save_data(self):
```

3.2.4 `restore_save_data`

This method is passed a dictionary of the custom save data returned by a previous instance of your device tab (using the `get_save_data` method, see section 2.2 for more details). This method will be called by BLACS at the appropriate times. You should restore the data to the appropriate places in your device tab implementation in this method.

The signature of this method is:

```
def restore_save_data(self, data):
```

3.2.5 `get_front_panel_values`

If you are implementing a custom output that is not bound to an `AO`, `DO` or `DDS` object, you may wish to override this method to extend it's capabilities. See section 3.3.12 for more details on this method.

3.3 Instance Methods

3.3.1 `add_secondary_worker`

This method is used to register the names of workers that are not the primary worker. The default implementation of `DeviceTab` communicates with these workers immediately after communicating with the primary worker.

The signature of this method is:

```
def add_secondary_worker(self, worker_name):
```

where `worker_name` is a string containing the name assigned to the worker when you created it using `create_worker` (see section 3.3.10).

3.3.2 auto_create_widgets

This method creates and returns a widget for each of the `DO`, `AO` and `DDS` objects created using the `create_*_outputs` methods (see sections 3.3.4, 3.3.8 and 3.3.6). The return value is a tuple containing 3 elements, each of which is a dictionary of widgets for the `DDS`, `AO` and `DO` objects respectively. These dictionaries are keyed by the hardware channel name used when creating the outputs.

The signature of this method is:

```
def auto_create_widgets(self):
    dds_widgets, ao_widgets, do_widgets = {}, {}, {}
    ...
    return dds_widgets, ao_widgets, do_widgets
```

3.3.3 auto_place_widgets

Calling this method more than once per instance is not recommended. This method places any passed widgets into a layout. This layout is of our own design, and has the following features:

- collapsible/expandable groups,
- automatically adjusts the number of widgets per row so that they fit and display nicely as the tab is resized,
- the widgets in a group have a uniform width, and
- the width of widgets across multiple groups can be linked (this feature not yet exposed, contact the developers if you would like it to be).

The `auto_place_widgets` method can take an arbitrary number of arguments. There are two options for the format of the argument:

1. The argument can be a dictionary of widgets, keyed by the hardware channel name. In this case, the title of the layout group will correspond to the type of widget of the first element of the unsorted dictionary (e.g., "Analog Outputs", "Digital Outputs" or "DDS Outputs"). If a group with this title already exists, widgets will be added to the existing group. The new widgets will be sorted alphabetically by key, and added to this group.

2. The argument can be a tuple. The first element of this tuple should be the title of the group you wish to create or append to. The second element of this tuple should be a dictionary of widgets as described in option 1 above. The tuple can also contain a third, optional element, which specifies a function to be used to sort the order of the new widgets. This function is provided directed to the `key` keyword argument of the `sorted` method in the Python standard library.

This method returns no value.

Example usage:

```
analog_outputs = {'ao1':my_ao_widget}
digital_outputs = {'port0/line1': my_do_widget_1, 'port0/line21':my_do_widget_2}
self.auto_place_widgets(analog_outputs,
                        (``My custom name'', digital_outputs,
                          lambda x: '\%02d'\%x.replace('port0/line','')
                        )
                       )
```

3.3.4 create_analog_outputs

This method creates analog output objects. These objects are the backend for analog widgets, and handle unit conversions, saving/restoring values, etc. The tutorial in section 2 covers how to use this method in depth.

This method takes a dictionary that defines the properties of the analog output objects you wish to instantiate. The dictionary should be keyed by hardware channel name. The values of each key should be a dictionary with the keys `base_unit`, `min`, `max`, `step` and `decimals`.

This method returns no value.

Example usage:

```
    analog_properties = {'hardware_channel_1':{'base_unit':self.ao_base_units,
                                               'min':self.ao_base_min,
                                               'max':self.ao_base_max,
                                               'step':self.ao_base_step,
                                               'decimals':self.ao_base_decimals
                                              },
                         'hardware_channel_2':{'base_unit':self.ao_base_units,
                                               'min':self.ao_base_min,
                                               'max':self.ao_base_max,
                                               'step':self.ao_base_step,
                                               'decimals':self.ao_base_decimals
                                              },
                        }
    self.create_analog_outputs(analog_properties)
```

3.3.5 create_analog_widgets

This method is used to create widgets associated with analog output objects. It is used by `auto_create_widg` (see section 3.3.2) but can be called directly if more control over the widgets are required. This method takes a dictionary which specifies properties of the analog widgets. The dictionary should be keyed by hardware channel name. The value associated with each key should be a dictionary which contains 0 or more of the keys:

- `display_name`: This can be used to display a custom string next to the spinbox in the analog widget. If set to `None`, the string defaults to the channel followed by the name specified in the connection table for this channel (if any). If set to a string, that string is displayed instead. If this key is not specified, it defaults to `None`

- `horizontal_alignment`: This can be used to specify whether the string discussed in the above dot point, is aligned above, or to the left of the spinbox. If not specified, it defaults to `False` (string above the spinbox). If set to `True` it places the string to the left of the spinbox.

- `parent`: If you wish this widget to have a specific parent widget (see Qt documentation on parent widgets) you can set it here. If this key is not specified, it defaults to `None`. Note that any parent set will be overwritten if the widget is added to a layout.

This method returns a dictionary of widgets, keyed by the hardware channel names.
Example usage:

```
widget_properties = {'ao1':{'display_name':'Frequency',
                            'horizontal_alignment': True
                           }
                    }
my_widgets = self.create_analog_widgets(widget_properties)
```

3.3.6 `create_dds_outputs`

This method creates analog output objects. These objects are the backend for DDS widgets, and handle unit conversions, saving/restoring values, etc. The tutorial in section 2 covers how to use this method in depth.

This method takes a dictionary that defines the properties of the DDS output objects you wish to instantiate. The dictionary should be keyed by hardware channel name. The values of each key should be a dictionary with one or more of the keys `freq`, `amp`, `phase` and `gate`. The keys used determine whether controls for those properties are shown. Each of the `freq`, `amp`, `phase` and `gate` keys used should have a value that is a dictionary which follows the format used in `create_analog_outputs` (see section 3.3.4) for `freq`, `amp` and `phase` and follows the format used in `create_digital_outputs` (see section 3.3.8) for the `gate`.

This method returns no value.

Example usage:

```
    # The creation of this dictionary could be simplified by one or
    # more for loops. However we chose to write everything out explicitly
    # here so that it is clear what is going on. Consult one of the already
    # implemented device classes to see how it could be done in fewer lines
    # of code (for example the PulseBlaster)
    dds_properties = {'hardware_channel_1':
                           {'freq':{'base_unit':self.dds_base_units['freq'],
                                    'min':self.dds_base_min['freq'],
                                    'max':self.dds_base_max['freq'],
                                    'step':self.dds_base_step['freq'],
                                    'decimals':self.dds_base_decimals['freq']
                                   },
                            'amp':{'base_unit':self.dds_base_units['amp'],
                                    'min':self.dds_base_min['amp'],
                                    'max':self.dds_base_max['amp'],
                                    'step':self.dds_base_step['amp'],
                                    'decimals':self.dds_base_decimals['amp']
                                   },

                                    'min':self.dds_base_min['phase'],
                                    'max':self.dds_base_max['phase'],
                                    'step':self.dds_base_step['phase'],
                                    'decimals':self.dds_base_decimals['phase']
                                   },
                            'gate':{}
                            },

                      'hardware_channel_2':
                           {'freq':{'base_unit':self.dds_base_units['freq'],
                                    'min':self.dds_base_min['freq'],
                                    'max':self.dds_base_max['freq'],
                                    'step':self.dds_base_step['freq'],
                                    'decimals':self.dds_base_decimals['freq']
                                   },
                             'amp':{'base_unit':self.dds_base_units['amp'],
                                    'min':self.dds_base_min['amp'],
                                    'max':self.dds_base_max['amp'],
                                    'step':self.dds_base_step['amp'],
                                    'decimals':self.dds_base_decimals['amp']
                                   },

                                    'min':self.dds_base_min['phase'],
                                    'max':self.dds_base_max['phase'],
                                    'step':self.dds_base_step['phase'],
                                    'decimals':self.dds_base_decimals['phase']
                                   },
                            'gate':{}
                            }
                     }
    self.create_dds_outputs(dds_properties)
```

3.3.7 **`create_dds_widgets`**

This method is used to create widgets associated with dds output objects. It is used by `auto_create_widget` (see section 3.3.2) but can be called directly. This method takes a dictionary which specifies properties of the dds widgets. The dictionary should be keyed by hardware channel name. The value associated with each key should be an empty dictionary (this is so we can expand configuration options in the future). This method returns a dictionary of widgets, keyed by the hardware channel names.

Example usage:

```
widget_properties = {'dds0':{},
                     'dds1':{}
                    }
my_widgets = self.create_dds_widgets(widget_properties)
```

3.3.8 `create_digital_outputs`

This method creates digital output objects. These objects are the backend for digital widgets, and handle the lock state, saving/restoring values, etc. The tutorial in section 2 covers how to use this method in depth.

This method takes a dictionary that defines the properties of the digital output objects you wish to instantiate. The dictionary should be keyed by hardware channel name. The values of each key should be an empty dictionary (this is so we can expand configuration options in the future).

This method returns no value.

Example usage:

```
    digital_properties = {'hardware_channel_1':{},
                          'hardware_channel_2':{},
                         }
    self.create_digital_outputs(digital_properties)
```

3.3.9 `create_digital_widgets`

This method is used to create widgets associated with digital output objects. It is used by `auto_create_widg` (see section 3.3.2) but can be called directly. This method takes a dictionary which specifies properties of the digital widgets. The dictionary should be keyed by hardware channel name. The value associated with each key should be an empty dictionary (this is so we can expand configuration options in the future). This method returns a dictionary of widgets, keyed by the hardware channel names.

Example usage:

```
widget_properties = {'port0/line1':{},
                     'port0/line21':{}
                    }
my_widgets = self.create_digital_widgets(widget_properties)
```

3.3.10 `create_worker`

This method creates and instantiates an instance of a specified worker process. This results in a new process being spawned on your PC. The signature of this method is:

```
def create_worker(self, worker_name, WorkerClass, worker_arguments = {}):
```

where `worker_name` is a string containing the name you wish to assign to this worker (used to register the worker as a primary or secondary worker, see sections 3.1.8 and 3.3.1). `WorkerClass` is a reference to the **class** (**not** an instance) you wish to launch in the subprocess (this class should subclass the `Worker` class in 'BLACS/tab_base_classes.py'. The `create_worker` method is responsible for creating an instance of the `WorkerClass`. `worker_arguments` is a dictionary of arguments to be passed to the worker process. These arguments become available as instance attributes of the worker instance with the names specified by the keys of the dictionary (subject to change in the future).

3.3.11 get_channel

This method returns an AO, DO or DDS object for a given hardware channel name, or None if it does not exist.

The signature of this method is:

```
def get_channel(self, channel_name):
```

where channel_name is a string containing the hardware channel name of the output object you wish to return.

3.3.12 get_front_panel_values

This method returns a dictionary, keyed by hardware channel name, containing all the current values of the AO, DO and DDS objects.

This method is used to get the values on the BLACS front panel when programming the device in both manual and buffered mode. If you have a custom output not registered as either an AO, DO or DDS object, you may wish to override this method to **extend** the dictionary to contain your custom values.

3.3.13 get_tab_layout

This method takes no arguments and returns a reference to the main layout of the tab. This layout is an instance of QVBoxLayout.

3.3.14 queue_work

This method returns a tuple in the form required to yield from a statefunction (see section 4.1 for more details). It is used to communicate with a worker process.

The signature of this method is:

```
def queue_work(self, worker_name, worker_function_name, *args, **kwargs):
```

where worker_name should be a string containing the name of the worker you wish to communicate wish (as defined when you created the worker process, see section 3.3.10). worker_function_name should be a string containing the name of the method of the worker process you wish to run. You can also specify any number of optional arguments and keyword arguments following this. These arguments will be passed to the method of the worker process you have specified to run.

Example usage:

```
result = yield(self.queue_work(self.primary_worker, 'my_worker_function',
                               1, 7, keyword = 'foo'))

# or alternatively
work = self.queue_work(self.primary_worker, 'my_worker_function',
                       1, 7, keyword = 'foo')
result = yield(work)
```

3.3.15 statemachine_timeout_add

This method sets up a timer which repeatedly calls the specified state-function after a delay. The signature of this method is:

```python
def statemachine_timeout_add(self, delay, statefunction, *args, **kwargs):
```

where `delay` is the minimum time in millseconds between calls to the specified state-function and `statefunction` is a reference to the method to call (this method must be decorated with `@define_state`, see section 4.1). You can also specify any number of optional arguments and keyword arguments following this. These arguments will be passed to the state-function you have specified to run.

Note that the specified delay is the **minimum** time between calls. Immediately after your state-function has been run, another state will execute which sets up a timer object that fires in `delay` milliseconds. There is some overhead in processing this state. Furthermore, when the timer fires, your state-function is queued up in the state machine, but is not guaranteed to immediately run as there may be other state-functions in the queue. However, it **is** guaranteed that the state which sets up the timer, will always run immediately after your state-function. This ensures there will only ever be one entry for your state-function in the state queue at any given time.

Note: You cannot set up multiple timeout callback for the same state-function. Subsequent calls to `statemachine_timeout_add` with the same state-function in the argument, will result in the old timeout being replaced with the delay, etc. of this new call.

3.3.16 statemachine_timeout_remove

This method takes a reference to a state-function as the only argument, and removes the timeout callback created with `statemachine_timeout_add` (see section 3.3.15). This method returns `True` if a timeout was found to remove, otherwise `False`.

The signature of this method is:

```python
def statemachine_timeout_remove(self, statefunction):
```

where `statefunction` is a reference to a function decorated with `@define_state` (see section 4.1).

3.3.17 statemachine_timeout_remove_all

This method takes no arguments, and removes **all** timeout callbacks created with `statemachine_timeout_a` (see section 3.3.15). This method returns `True` if there were timeouts to remove, otherwise `False`.

The signature of this method is:

```python
def statemachine_timeout_remove_all(self):
```

3.3.18 supports_remote_value_check

This method sets an internal flag that determines whether a timeout callback will be setup that runs a state-function which periodically checks the output values programmed into the device and compares them with the BLACS front panel. This is effectively a consistency check to make sure the BLACS front panel is up to date. As not all devices support this feature, you must explicitly enable it if you device does.

3.3.19 `supports_smart_programming`

This method sets an internal flag that determines whether the checkbox to force a full buffered reprogram is shown or hidden. If set to `False`, the checkbox is hidden and left checked permanently. Note that even if set to `False`, you can still programmatically change whether a full buffered reprogram can occur with the `force_full_buffered_reprogram` property (see section 3.1.5).

The signature of this method is:

```
def supports_smart_programming(self, value):
```

where value is `True` or `False`.

3.3.20 `connect_restart_receiver`

This method allows you to register a function to be called when the device tab is restarted. Usually this is only used by components of BLACS outside of the device Tab, though there is no reason why you couldn't call it from within DeviceTab.

The signature of this method is:

```
def connect_restart_receiver(self, function):
```

where function is a reference to the function you wish called. The function you provide should take one argument (the name of the device that has been restarted).

3.3.21 `disconnect_restart_receiver`

This method allows you to deregister a function that was registered using `connect_restart_receiver` (see section 3.3.20). Usually this is only used by components of BLACS outside of the device Tab, though there is no reason why you couldn't call it from within DeviceTab.

The signature of this method is:

```
def disconnect_restart_receiver(self, function):
```

where function is a reference to the function you wish deregister.

3.3.22 `restart`

This method initiates a restart of the device Tab. Usually this is called when the restart button is clicked, however you could call this method programmatically if you wished. The method takes an arbitrary number of keyword arguments (and ignores them all) so it can be connected to GUI signals that pass arguments.

The signature of this method is:

```
def restart(self, *args):
```

3.3.23 update_from_settings

This method restores the settings of the tab from a provided dictionary. This method is called by BLACS on startup and if the user loads a front panel via the file menu. This method will not affect the state of "locked" widgets.

The signature of this method is:

```
def update_from_settings(self, settings):
```

where `settings` is a dictionary with the keys `device_name`, `connection_table`, `saved_data` and `front_panel_settings`. These keys should have the following values:

- `device_name`: A string containing the device name (as stored in the `device_name` attribute).
- `connection_table`: A reference to the connection table object (as stored in the `connection_table` attribute).
- `saved_data`: A dictionary of the form that `get_save_data` is to be passed.
- `front_panel_settings`: A dictionary keyed by hardware channel name. Each of these keys should have a value which is a dictionary containing the settings for the output channel.

3.4 State-functions

State-functions are methods decorated with the `@define_state` decorator. When called, they do not run immediately, but are placed in a queue and executed in the order they were called. See section 4.1 for more details.

The `DeviceTab` class provides some default implementations of state-functions, which should be adequate for most device implementations. We list them below:

- `abort_buffered`
- `abort_transition_to_buffered`
- `check_remote_values` (has an associated method `on_resolve_value_inconsistency` which is not a state-function)
- `destroy`
- `program_device`
- `transition_to_buffered`
- `transition_to_manual`

There are some cases where it may be desirable to override these methods. Please check with us on the mailing list before doing so, in case there is an easy way to achieve your aims. Should you wish to override them, **do not call the state-function method you are overriding in an attempt to extend it's functionality**. If you override, you **must** reimplement yourself, any functionality you want to keep from the `DeviceTab` implementation.

4 Advanced features of the `DeviceTab`

4.1 The state machine

The `Tab` class (`DeviceTab` subclasses `Tab`) contains a state machine which regulates the interprocess communication between the worker process and the GUI process, as well as making sure that the Qt event based architecture is transformed into a deterministic system. The simple Device Class implementation discussed in section 2 is already using the state machine behind the scenes.

Most of the state machine architecture is hidden within the `Tab` class. Any function in the device class prefixed (decorated) with `@define_state` will be queued up appropriately in the state machine when it is called. We will refer to functions prefixed with `@define_state` as *state machine functions*.

The state machine has 4 modes it can be in: `MODE_MANUAL`, `MODE_TRANSITION_TO_BUFFERED`, `MODE_BUFFERED` and `MODE_TRANSITION_TO_MANUAL`. These 4 modes correspond to the following situations for the device:

- `MODE_MANUAL`: The device is programmed to static values that match the front panel interface of BLACS. The device is **not** running a buffered experiment compiled by `labscript`.
- `MODE_TRANSITION_TO_BUFFERED`: The device is being programmed with a buffered sequence from a HDF5 file.
- `MODE_BUFFERED`: The device is either waiting for a trigger to start the execution of a buffered experiment, or is currently executing an experiment.
- `MODE_TRANSITION_TO_MANUAL`: The device has finished executing the buffered experiment and is currently saving any acquired data and preparinfg to entered `MODE_MANUAL`.

In general you should not need to worry about transitioning between these modes unless you are overwriting `transition_to_buffered` or `transition_to_manual`.

A state machine function can be configured to run only in one or more of these modes. A state machine function can also be configured to stay in the queue until the state machine is in a matching mode and/or to only use the most recently queued call of the state machine function. The code `@define_state` is a Python *decorator*, and takes 2 required arguments and 1 optional argument. The arguments are:

- `allowed_modes`: This in the binary OR of the modes in which the following state machine function is allowed to run in. If the statemachine is in a mode not specified here, it will not run until the state machine is in a matching mode
- `queue_state_indefinitely`: This argument should be set to True or False. If True, the state machine function will remain in the queue until the statemachine enters one of the allowed modes. Setting this flag to True guarantees the state machine function will run eventually (unless the device tab is restarted or BLACS is closed). If False, this state machine function will

be removed from the queue if it is the next item in the state machine to be run, but the current state machine mode is not one of the allowed modes.

- `delete_stale_states`: An optional argument that defaults to False. If set to True, the state machine will look for newer versions of the state machine function in the queue, and will run the most recent found before encountering a different state machine function in the queue. If set to False (the default), this state will not be deleted when newer versions exist.

An example use of `@define_state` is shown below:

```
@define_state(allowed_modes=MODE_MANUAL|MODE_BUFFERED,
              queue_state_indefinitely=True,
              delete_stale_states=False)
def start(self,widget=None):
    # some code follows
    time.sleep(5)
```

Note that if we call this function from within our code, E.g.:

```
def foo(self):
    self.start() # This function call returns immediately
```

the call to `start` returns immediately. Some time in the future, the function `start` will be executed, and the main thread (the GUI) will sleep for 5 seconds as expected.

This means that calls to state machine functions (E.g. `start`) cannot return parameters in the conventional way. It is expected that state machine functions will be called upon a Qt event (in which case you will never need to return anything from your state) or from another thread (in which case you can use a Python Queue to block the calling thread until your state machine function puts the return values in the queue)

As mentioned, state machine functions are designed to interact with the worker process. You can call functions within your worker process with the following code:

```
@define_state(MODE_MANUAL,True)
def start(self,widget=None):
    result = yield(self.queue_work(self.primary_worker,'foo'))
```

If you wish to pass arguments to your function, you can instead call something similar to

```
    result = yield(self.queue_work(self.primary_worker,'foo',1,5,x=3))
```

This will call the function `foo` with the arguments `1,5,3` in the worker process name stored in `self.primary_worker`.

It is important to note that you cannot pass objects as arguments to a worker class function. The arguments must be able to be placed into a Python Queue.

You may `yield` to one or more worker process function many times within a single state function. **Note that you can only do this from within a function that is decorated with `@define_state`.**

4.2 Device class state machine, callback on timeout

The state machine architecture of BLACS provides the functionality to register a timeout callback; that is, a function to be called (approximately) every `n` milliseconds. This is generally used for periodic status monitoring of a device.

To add such a timeout callback to your code, you call from within your device class:

```
self.statemachine_timeout_add(delay,self.some_function,userdata1,...)
```

where userdata1 and following arguments are passed to `self.some_function`.

To remove the timeout, call

```
self.statemachine_timeout_remove(self.some_function)
```

Due to the nature of state machines (callbacks are processed one at a time based on the order in the queue), your function is not guaranteed to run as often as you have requested. Initially, because your function is a state machine function, BLACS will add your function to the state machine queue. Once your function has run, it will add a Qt single-shot timer to run an internal function in `delay` milliseconds. This internal function will queue up your function in the state machine again, and, depending on the length of the queue at that time, may not run immediately.

Also note that you can only have one timeout for a given function. Creating a timeout for the same function will replace the existing timeout.

4.3 Remote Value Checking

The default implementation of the `DeviceTab` supports a periodic consistency check on the output values of a device (providing that the device supports this). This can be enabled by setting `supports_remote_value_check` The tab will query the worker process for current output values, and compare these to the values on the BLACS front panel. If any inconsistencies exist, BLACS will ask the user to choose whether to keep the remote values of the device, or the local values shown on the BLACS front panel. Beyond enabling the feature, no further implementation is required in the device class. For the worker class implementation, see section 5.5.

5 The Worker Class

The worker class is used solely to communicated with the hardware for our device. It exists to separate out the code, modules, dll's, etc. from the GUI to provide better stability to the system. The worker class is instantiated inside a separate process (referred to from now on as the worker process), which can be restarted by the user if the device becomes unresponsive. Upon restart, all libraries are completely reloaded as they are only loaded within the worker process. This allows the system to recover from errors in 3rd party API's without the need to restart the entire control system.

The worker class consists of functions which can be executed by the state machine. In your device class you can call:

```
result = yield(self.queue_work('worker_name','some_function',arg1,arg2,kwrd1=arg3))
```

and this will call the function called `some_function` with arguments `arg1,arg2,kwrd1=arg3` in the worker process you have created and named `'my_worker'` (not to be confused with the Worker class name). This is covered in more detail in section 4.1.

5.1 The `init` function

The init function in the worker class is special for two reasons. The first is that it is not called `__init__` as you would expect. This is because we don't want to override the `__init__` function in class `Worker`, which is essential for successful operation.

The second is that this is the function in which you should import the modules, classes, etc. that you wish to access from within the worker class. They are imported here, within the class, so that they only exist within the worker process, and not within the BLACS process. This allows modules to be completely unloaded from the system when the worker process is restarted, and thus recovery of the programming library without the need to completely restart the program (a technique not seen in any other control system to our knowledge). The `init` function will always be the first function in your worker process to be run after it has been created.

If you wish to import a module (for example `h5py`), use the following code:

```
class MyWorker(Worker):
    def init(self):
        global h5py; import h5_lock, h5py
```

This imports both the labscript suite `h5_lock` module (which prevents simultaneous accessing of h5 file which could cause data corruption) and the `h5py` module which is stored as a global variable.

If you wish to import something from within a module (for example one or more functions from the `spinapi` module), you can use:

```
        exec 'from spinapi import *' in globals()
```

or

```
        exec 'from spinapi import pb_start, pb_stop' in globals()
```

You can also set the default values for any class attributes in the `init` function. For instance:

```
        self.smart_cache = {}
```

5.2 The `program_manual` function

This function is called whenever the device needs to be programmed to output values when **not** executing a buffered shot. The most common time this happens is when a value of a digital, analog or DDS output widget on the front panel is changed.

The function is passed an argument which contains all of the current front panel values in a dictionary. The dictionary is keyed by the hardware channel names you defined in the `initialise_GUI` function in your Device GUI class.

The method should return the values for each output value, coerced to the value that the device is actually outputting now. This is to accurately reflect the device quantisation on the front panel of BLACS. The return value should be a dictionary of the same format as the one passed in.

Your function should look something like:

```
    def program_manual(self, front_panel_values):
        # Program the device
        # work out what values the device is actually outputting
        return modified_front_panel_values
```

5.3 The `transition_to_buffered` function

This function is called whenever the Queue Manager requests the device to move into buffered mode in preparation for executing a buffered sequence.

This function is passed the `device_name` to look up in the HDF5 file located at the path contained by `h5file`. The function is also passed the current initial_values, so that your device can be programmed to maintain output continuity until the device is triggered. This may require inserting a dummy instruction at the beginning of the instruction list provided by `labscript`. The final argument passed to this function is a Boolean value that indicates whether the device should have it's entire instruction table overwritten (only applies to device classes that have called `self.supports_smart_pro`
in the `initialise_GUI` function).

The function should return a dictionary containing the final value of each output once the buffered shot has finished execution. The dictionary should be keyed by the hardware channel names you defined in the `initialise_GUI` function in your Device GUI class.

Note: you should avoid holding the HDF5 file open when programming the device. Open the file, copy the data into local memory, close the file and the program the device. This will allow other devices to access the file to begin their programming process, and will minimise your experiment cycle time.

Example:

```
    def transition_to_buffered(self,device_name,h5file,initial_values,fresh):
        # Open HDF5 file
        # copy device data to local memory
        # close HDF5 file
        #
        # check whether we need to do a completely fresh program or not
        # Check whether the data for this shot is similar enough to the last shot
        #       and whether only some instructions need reprogramming
        #
        # build final value dictionary
        return final_experiment_values
```

5.4 The `transition_to_manual` function

This function is called after the master pseudoclock reports that the experiment has finished. This function takes no arguments, should place the device back in the correct mode for operation by the front panel of BLACS, and return a Boolean flag indicating the success of this method. Any acquisitions made during the buffered shot should be saved to the HDF5 file now (you must store a reference to the HDF5 file in `transition_to_buffered` if you need to access it in `transition_to_manual`).

Example:

```
    def transition_to_manual(self):
        # save any acquired data to HDF5 file
        # place device in mode ready for BLACS front panel control
        # return True if this was all successful, or False otherwise
        return True/False
```

5.5 Remote value checking

This `check_remote_values` method should only be present if your tab is configured to check remote values (see section 4.3). The method takes no arguments, and should return a dictionary of remote values, keyed by hardware channel name.

Example:

```
    def check_remote_values(self):
        current_output_values = {}
        # read from the device, the values it is outputting
        # place them in a dictionary, keyed by hardware channel
        return current_output_values
```

5.6 Abort functions

There are two functions that may be called if something went wrong and the experiment shot is to be aborted. The first, `abort_transition_to_buffered`, is called if the experiment shot must be aborted **before** the master pseudoclock has been triggered to begin. The second, `abort_buffered`, is called if the shot must be aborted during the execution of a buffered shot. Both functions take no arguments (other than `self`) and should return `True` or `False` depending on whether they were successful and the device is ready for front panel input from BLACS again. If False is returned, this will cause an error to be displayed requesting the user to **restart** the device tab themselves.

Example

```
    def abort_buffered(self):
        # place the device back in manual mode, while in the middle
        #       of an experiment shot
        # return True if this was all successful, or False otherwise
        return True/False

    def abort_transition_to_buffered(self):
        # place the device back in manual mode, after the device has run
        #       transition_to_buffered, but has not been triggered to
        #       begin the experiment shot.
        # return True if this was all successful, or False otherwise
        return True/False
```

5.7 The shutdown function

The shutdown function is called when BLACS is asked to close. This should put the device in safe state, for example closing any open communication connections with the device. The function should not return any value (the return value is ignored)

Example:

```
    def shutdown(self):
        # close any open connections
        # place the device in a nice state
        # return nothing
```