Starting the psychopy.iohub Process ¶

To use ioHub within your PsychoPy Coder experiment script, ioHub needs to be started at the start of the experiment script. The easiest way to do this is by calling the launchHubServer function.

launchHubServer function

psychopy.iohub.client.**launchHubServer**(***kwargs*)

The launchHubServer function is used to start the ioHub Process by the main psychopy experiment script.

To use ioHub for keyboard and mouse event reporting only, simply use the function in a way similar to the following:

```
from psychopy.iohub import launchHubServer

# Start the ioHub process. The return variable is what is used
# during the experiment to control the iohub process itself,
# as well as any running iohub devices.
io=launchHubServer()

# By default, ioHub will create Keyboard and Mouse devices and
# start monitoring for any events from these devices only.
keyboard=io.devices.keyboard
mouse=io.devices.mouse

# As a simple example, use the keyboard to have the experiment
# wait until a key is pressed.

print "Press any Key to Exit Example....."

keys = keyboard.waitForKeys()

print "Key press detected, exiting experiment."
```

launchHubServer() accepts several kwarg inputs, which can be used when more complex device types are being used in the experiment. Examples are eye tracker and analog input devices.

Please see the psychopy/demos/coder/iohub/launchHub.py demo for examples of different ways to use the launchHubServer function.

ioHubConnection Class

The psychopy.iohub.ioHubConnection object returned from the launchHubServer function provides methods for controlling the iohub process and accessing iohub devices and events.

class psychopy.iohub.client.**ioHubConnection**(*ioHubConfig=None*, *ioHubConfigAbsPath=None*)

ioHubConnection is responsible for creating, sending requests to, and reading replies from the ioHub Process. This class can also shut down and disconnect the ioHub Process.

The ioHubConnection class is also used as the interface to any ioHub Device instances that have been created so that events from the device can be monitored. These device objects can be accessed via the ioHubConnection .devices attribute, providing 'dot name' attribute access, or by using the .deviceByLabel dictionary attribute; which stores the device names as keys,

Using the .devices attribute is handy if you know the name of the device to be accessed and you are sure it is actually enabled on the ioHub Process. The following is an example of accessing a device using the .devices attribute:

```
# get the Mouse device, named mouse
mouse=hub.devices.mouse
current_mouse_position = mouse.getPosition()

print 'current mouse position: ', current_mouse_position

# Returns something like:
# >> current mouse position:  [-211.0, 371.0]
```

getDevice(*deviceName*)

Returns the ioHubDeviceView that has a matching name (based on the device : name property specified in the ioHub_config.yaml for the experiment). If no device with the given name is found, None is returned. Example, accessing a Keyboard device that was named 'kb'

```
keyboard = self.getDevice('kb')
kb_events= keyboard.getEvent()
```

This is the same as using the 'natural naming' approach supported by the .devices attribute, i.e:

```
keyboard = self.devices.kb
kb_events= keyboard.getEvent()
```

However the advantage of using getDevice(device_name) is that an exception is not created if you provide an invalid device name, or if the device is not enabled on the ioHub server (for example if the device hardware was not connected when the ioHub server started). Instead None is returned by this method. This allows for conditional checking for the existance of a requested device within the experiment script, which can be useful in some cases.

Args:

deviceName (str): Name given to the ioHub Device to be returned

Returns:

device (ioHubDeviceView) : the PsychoPy Process represention for the device that matches the name provided.

getEvents(*device_label=None*, *as_type='namedtuple'*)

Retrieve any events that have been collected by the ioHub Process from monitored devices since the last call to getEvents() or clearEvents().

By default all events for all monitored devices are returned, with each event being represented as a namedtuple of all event attributes.

When events are retrieved from an event buffer, they are removed from that buffer as well.

If events are only needed from one device instead of all devices, providing a valid device name as the device_label argument will result in only events from that device being returned.

Events can be received in one of several object types by providing the optional as_type property to the method. Valid values for as_type are the following str values:

- 'list': Each event is sent from the ioHub Process as a list of ordered attributes. This is the most efficient for data transmission, but not for human readability or usability. However, if you do want events to be kept in list form, set as_type = 'list'.
- 'astuple': Each event is converted to a namedtuple object. Event attributes are accessed using natural naming style (dot name style), or by the index of the event attribute for the event type. The namedtuple class definition is created once for each Event type at the start of the experiment, so memory overhead is almost the same as the event value list, and conversion from the event list to the namedtuple is very fast. This is the default, and normally most useful, event representation type.
- 'dict': Each event converted to a dict object, keys equaling the event attribute names, values being, well the attribute values for the event.
- 'object': Each event is converted into an instance of the ioHub DeviceEvent subclass based on the event's type. This conversion process can take a bit of time if the number of events returned is large, and currently there is no real benefit converting events into DeviceEvent Class instances vs. the default namedtuple object type. Therefore this option should be used rarely.

Args:

device_label (str): Indicates what device to retrieve events for. If None (the default) returns device events from all devices.

as_type (str): Indicates how events should be represented when they are returned to the user. Default: 'namedtuple'.

Returns:

tuple: A tuple of event objects, where the event object type is defined by the 'as_type' parameter.

clearEvents(*device_label='all'*)

Clears events from the ioHub Process's Global Event Buffer (by default) so that unneeded events are not sent to the PsychoPy Process the next time getEvents() is called.

If device_label is 'all', (the default), then events from both the ioHub *Global Event Buffer* and all *Device Event Buffer's* are cleared.

If device_label is None then all events in the ioHub *Global Event Buffer* are cleared, but the *Device Event Buffers* are unaffected.

If device_label is a str giving a valid device name, then that *Device Event Buffers* is cleared, but the *Global Event Buffer* is not affected.

Args:

device_label (str): device name, 'all', or None

Returns:

None

sendMessageEvent(*text*, *category=''*, *offset=0.0*, *sec_time=None*)

Create and send an Experiment MessageEvent to the ioHub Server Process for storage with the rest of the event data being recorded in the ioDataStore.

> Note
>
> MessageEvents can be thought of as DeviceEvents from the virtual PsychoPy Process "Device".

Args:

text (str): The text message for the message event. Can be up to 128 characters in length.

category (str): A 0 – 32 character grouping code for the message that can be used to sort or group messages by 'types' during post hoc analysis.

offset (float): The sec.msec offset to apply to the time stamp of the message event. If you send the event before or after the time the event actually occurred, and you know what the offset value is, you can provide it here and it will be applied to the ioHub time stamp for the MessageEvent.

sec_time (float): The time stamp to use for the message in sec.msec format. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer.

Returns:

bool: True

`createTrialHandlerRecordTable`(*trials*)

Create a condition variable table in the ioHub data file based on the a psychopy TrialHandler. By doing so, the iohub data file can contain the DV and IV values used for each trial of an experiment session, along with all the iohub device events recorded by iohub during the session. Example psychopy code usage:

```
# Load a trial handler and
# create an associated table in the iohub data file
#
from psychopy.data import TrialHandler,importConditions

exp_conditions=importConditions('trial_conditions.xlsx')
trials = TrialHandler(exp_conditions,1)

# Inform the ioHub server about the TrialHandler
#
io.createTrialHandlerRecordTable(trials)

# Read a row of the trial handler for
# each trial of your experiment
#
for trial in trials:
    # do whatever...


# During the trial, trial variable values can be updated
#
trial['TRIAL_START']=flip_time

# At the end of each trial, before getting
# the next trial handler row, send the trial
# variable states to iohub so they can be stored for future
# reference.
#
io.addTrialHandlerRecord(trial.values())
```

`addTrialHandlerRecord`(*cv_row*)

Adds the values from a TriaHandler row / record to the iohub data file for future data analysis use.

Parameters:	cv_row –
Returns:	None

`getTime`()

Deprecated Method: Use Computer.getTime instead. Remains here for testing time bases between processes only.

setPriority(*level='normal'*, *disable_gc=False*)

See Computer.setPriority documentation, where current process will be the iohub process.

getPriority()

See Computer.getPriority documentation, where current process will be the iohub process.

enableHighPriority(*disable_gc=False*)

Deprecated Method: Use setPriority('high', disable_gc) instead.

disableHighPriority()

Deprecated Method: Use setPriority('normal') instead.

enableRealTimePriority(*disable_gc=False*)

Deprecated Method: Use setPriority('realtime', disable_gc) instead.

disableRealTimePriority()

Deprecated Method: Use setPriority('normal') instead.

getProcessAffinity()

Returns the current **ioHub Process** Affinity setting, as a list of 'processor' id's (from 0 to getSystemProcessorCount()-1). A Process's Affinity determines which CPU's or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

This method is not supported on OS X at this time.

Args:

None

Returns:

list: A list of integer values between 0 and Computer.getSystemProcessorCount()-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

setProcessAffinity(*processor_list*)

Sets the **ioHub Process** Affinity based on the value of processor_list. A Process's Affinity determines which CPU's or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

The processor_list argument must be a list of 'processor' id's; integers in the range of 0 to Computer.processing_unit_count-1, representing the processing unit indexes that the ioHub Server should be allowed to run on. If processor_list is given as an empty list, the ioHub Process will be able to run on any processing unit on the computer.

This method is not supported on OS X at this time.

Args:

processor_list (list): A list of integer values between 0 and Computer.processing_unit_count-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Returns:

None

`flushDataStoreFile`()

Manually tell the ioDataStore to flush any events it has buffered in memory to disk."

Args:

None

Returns:

None

`shutdown`()

Tells the ioHub Process to close all ioHub Devices, the ioDataStore, and the connection monitor between the PsychoPy and ioHub Processes. Then exit the Server Process itself.

Args:

None

Returns:

None

`quit`()

Same as the shutdown() method, but has same name as PsychoPy core.quit() so maybe easier to remember.