Pyff Documentation

Release 2010.7

Bastian Venthur

July 20, 2010

CONTENTS

This is Pyff's documentation. Feedback developers are probably interested in the documentation of *Feedback base classes*

Contents:

CHAPTER

ONE

LIBRARY DOCUMENTATION

This section contains the documentation for the various Python modules provided by Pyff. Developers of Feedbacks are probably interested only in the documentation of the Feedback base classes.

Contents:

1.1 bcinetwork — Networking between the different Pyff components.

class BciNetwork(*ip, port, myport=None*)

Wrapper for Communication between Feedback Controller and GUI.

getAvailableFeedbacks()

Get available Feedbacks from Feedback Controller.

get_variables()

Get variables (name, type and value) from currently running Feedback.

pause()

Send 'pause' to Feedback Controller.

play()

Send 'play' to Feedback Controller.

quit()

Send 'quit' to Feedback Controller.

receive(*timeout*)

Receive a signal.

send_init(*feedback*)

Send 'send_init(feedback)' to Feedback Controller.

send_signal(*signal*)

Send a signal.

stop()

Send 'stop' to Feedback Controller.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.2 bcixml — Encoding and Decoding of BCI XML messages.

Encoding and decoding of bci-xml packages.

class BciSignal (*data, commands, type*)
Represents a signal from the BCI network.

A BciSignal object can be translated to XML and vice-versa.

exception DecodingError
Message could not be decoded.

exception EncodingError
Something message cound not be encoded.

exception Error
Our own exception type.

class XmlDecoder ()
Parses XML strings and returns BciSignal containing the data of the signal.

Usage: decoder = XmlDecoder() try:

bcisignal = decoder.decode_packet(xml)

except DecodingError: ...

decode_packet (*packet*)
Parse the XML string and return a BciSignal.

A DecodingError is raised when the parsing of the packet failed.

class XmlEncoder ()
Generates an XML string from a BciSignal object.

Usage: enc = XmlEncoder() try:

xml = enc.encode_packet(bcisignal)

except EncodingError: ...

encode_packet (*signal*)
Generates an XML packet from a BciSignal object.

An EncodingError is raised if the encoding failed.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.3 feedbackcontroller — Feedback Controller.

class FeedbackController (*plugin=None, fbpath=None, port=None*)
Feedback Controller.

Controlls the loading, unloading, starting, pausing and stopping of the Feedbacks. Can query the Feedback for it's variables and can as well set them.

handle_signal (*signal*)
Handle incoming signal.

send_to_feedback(*signal*)
Send data to the feedback.

send_to_peer(*signal*)
Send signal to peer.

start()
Start the Feedback Controller's activities.

stop()
Stop the Feedback Controller's activities.

class UDPDispatcher(*fc*)
UDP Message Hanldeer of the Feedback Controller.

handle_read()
Handle incoming signals.

Takes incoming signals, decodes them and forwards them to the Feedback Controller.

send_signal(*signal*)
Send signal to the GUI.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.4 feedbackprocesscontroller — Controls the Feedback Processes.

class FeedbackProcess(*modname, classname, ipcReady, port, fbplugin*)
Process that wrapps the Feedback's activities.

run()
Run the FeedbackProcess' activities in the new process.

class FeedbackProcessController(*plugindirs, baseclass, timeout*)
Takes care of starting and stopping of Feedback Processes.

get_feedbacks()
Return a list of available Feedbacks.

start_feedback(*name, port, fbplugin*)
Starts the given Feedback in a new process.

stop_feedback()
Stops the current Process.

First it tries to join the process with the given timeout, if that fails it terminates the process the hard way.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.5 gstimbox — Driver for g-STIMbox

g-STIMbox is a stimulator digital I/O box with an USB interface manufactured by g.tec medical engineering GmbH. The device features 14 digital inputs and 16 digital outputs. This module encapsulates a proprietary DLL (Dynamic-link library) that operates exclusively under Microsoft Windows.

1.5.1 g-STIMbox Installation

In order to install the device follow these steps:

1. Get `USB_Driver.exe` from the g-STIMbox driver package and execute it with administrator privileges. It will install the driver for the USB serial port.
2. Connect the g-STIMbox device to a free USB socket.
3. Open the *device manager* and find *USB Serial Port* under section *Ports*. It should specify the virtual COM port the device is connected to in brackets (e.g. COM3 meaning port number 3).
4. Now you can already test the device by calling `gSTIMboxDemo1.exe` that is also included in the driver package.

For this Python module to work follow these steps:

1. Make sure the `gSTIMbox.dll` file is available when using the device. The easiest way is to put the file into the system folder (on most Windows installations this is `C:\Windows\System`). Another way is to copy it to the same folder where this module resides.
2. Run the `demo feedback` or have a look at section *Usage*.

For further information refer to the PDF manual shipped with the device.

1.5.2 Usage

Output modes

This example shows both output port operation modes (manual, frequency).

The output ports can be either controlled manually (ON/OFF) or assigned a specific frequency. It's possible to have different ports operate on different modes simultaneously (e.g. port 1 and 3 work in manual mode while 2 and 4 work in frequency mode).

Example code:

```
from sys import stdout
from time import sleep
from gstimbox import GStimbox

comport = 3
b = GStimbox(comport)
print "Connected to g-STIMbox (serial port %d)" % comport
print "Driver version %s, firmware version %s" %          (b.getDriverVersion(), b.getHWVersion())
print "Micro-controller frequency demo running..."
stdout.flush()
port = [0, 1, 2, 3, 4, 5, 6, 7]
freq = [1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5]
mode = [1, 1, 1, 1, 1, 1, 1, 1]
b.reset()
b.setFrequency(port, freq)
b.setMode(port, mode)
sleep(10)
b.reset()
print "Manual ON/OFF demo running..."
stdout.flush()
state = [0 for i in range(16)]
for i in range(5):
```

```
    for j in range(8):
        state[j] = 1
        b.setPortState(state)
        sleep(0.1)
    for j in range(8):
        state[j] = 0
        b.setPortState(state)
        sleep(0.1)
b.reset()
print "Demo finished."
```

Processing input

In order to handle input from the g-STIMbox input ports a callback function must be specified. The callback function receives a single argument, a list of 14 values (e.g. `[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`). Each value maps to the corresponding input port (e.g. list index `0` corresponds to input port 1). In the example input ports 2 and 4 are active while all others are not. Note that the callback function is triggered on every state change (that is on pressing *and* releasing of a button).

The following example prints out the complete input state vector and activates all output ports if button on input port 1 is active (pressed).

Example code:

```
from sys import stdout
from time import sleep
from gstimbox import GStimbox

def input_callback(input_vector):
    global b
    print "Input vector changed:"
    print input_vector
    b.setPortState([input_vector[0] for i in range(16)])
    stdout.flush()

comport = 3
b = GStimbox(comport, input_callback)
print "Connected to g-STIMbox (serial port %d)" % comport
print "Driver version %s, firmware version %s" %          (b.getDriverVersion(), b.getHWVersion())
b.reset()
print "Use a button connected to input port 1 to activate all output ports."
print "This program will exit after 15 seconds."
stdout.flush()
sleep(15)
b.reset()
print "Demo finished."
```

class **GStimbox**(*port=3, callback=None*)

Create a connection to the g-STIMbox.

The serial port number defaults to `3`. A callback function can be specified that handles signals from the input connectors (see *Processing input*).

close()

Close device connection.

getDriverVersion()
Returns API library version.

getHWVersion()
Returns firmware version.

reset()
Reset device.

setFrequency(*port, freq*)
Set the frequencies for output ports.

port - list of port numbers to change. Valid port numbers range from 0 to 15. (eg. `[0, 6, 7]`)

freq - list of frequencies which are to be assigned to the ports. Allowed values range from 1 to 50. The function rounds these frequencies to one digit after the comma. The length of `freq` must equal the length of port list. (eg. `[1, 2.7, 5.8]`)

setMode(*port, mode*)
Set the operation mode of output ports.

port is a list of port numbers to change (eg. `[0, 2, 3]`). Valid port numbers range from 0 to 15.

modes is a list of modes for the ports defined in the port variable. A mode value can be either `0` (controlled manually, see portState()) or `1` (microprocessor controlled frequency, see setFrequency()).

setPortState(*state*)
Set ON/OFF state for ports running in mode `0` (see setMode()).

state is a list with a length of 16. Valid values for a state is an integer of either `0` or `1`.

eg. `state = [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

exception GStimboxError
g-STIMbox device communication error.

moduleauthor:: Mirko Dietrich <mirko.dietrich(AT)bccn-berlin(DOT)de>

1.6 ipc — Inter Process Communication.

Inter Process Communication.

this module provides classes to ease the inter process communication (IPC) between the Feedback Controller and the Feedbacks

class FeedbackControllerIPCChannel(*conn, fc*)
IPC Channel for Feedback Contoller's end.

handle_message(*message*)
Handle message from Feedback.

class FeedbackIPCChannel(*conn, feedback*)
IPC Channel for Feedback's end.

handle_message(*message*)
Handle message from Feedback Controller.

class IPCChannel(*conn*)
IPC Channel.

Base for the channels, the Feedback Controller and the Feedbacks need.

This Class transparently takes care of de-/serialization of the data which goes through the IPC. Derived classes should implement

handle_message(self, message)

to do something useful and use

send_message(self, message)

for sending messages via IPC.

collect_incoming_data(*data*)
Append incoming data to input buffer.

found_terminator()
Process message from peer.

handle_close()
Handle closing of connection.

handle_message(*message*)
Do something with the received message.

This method should be overwritten by derived classes.

send_message(*message*)
Send message to peer.

class IPCConnectionHandler(*fc*)
Waits for incoming connection requests and dispatches a FeedbackControllerIPCChannel.

close_channel()
Close the channel to the Feedback.

handle_accept()
Handle incoming connection from Feedback.

handle_close()
Handle closing of connection.

handle_error()
Handle error.

send_message(*message*)
Send the message via the currently open connection.

get_feedbackcontroller_connection()
Return a connection to the Feedback Controller.

ipcloop()
Start the IPC loop.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.7 PluginController — Finding and Loading Feedbacks.

class PluginController(*plugindirs, baseclass*)
Finds, loads and unloads plugins.

find_plugins()
Returns a list of available plugins.

load_feedback_list (*filename, plugindir*)
Load classnames from file and construct modulename relative to plugindir from plugindir, filename and file entries.

Returns a dictionary: classname -> module.

load_plugin (*name*)
Loads the given plugin and unloads possibly sooner loaded plugins.

test_plugin (*root, filename*)
Test if given module contains a valid plugin instance.

Returns None if not or (name, modulename) otherwise.

unload_plugin ()
Unload currently loaded plugin.

import_module_and_get_class (*modname, classname*)
Import the module and return modname.classname.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.8 RollbackImporter — Importing and Unloading of Modules.

class RollbackImporter ()
RollbackImporter.

RollbackImporter instances install themselves as a proxy for the built-in __import__ function that lies behind the 'import' statement. Once installed, they note all imported modules, and when uninstalled, they delete those modules from the system module list; this ensures that the modules will be freshly loaded from their source code when next imported.

Usage:

if self.rollbackImporter: self.rollbackImporter.uninstall()

self.rollbackImporter = RollbackImporter() # import some modules

uninstall ()
Unload all modules since __init__ and restore the original import.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

CHAPTER

TWO

FEEDBACK BASE CLASSES

This section provides documentation for the Feedback base classes available in Pyff.

Contents:

2.1 `Feedback` — Feedback Base Class

This module contains the Feedback class, which is the baseclass of all feedbacks.

class `Feedback` (*port_num=None*)

Base class for all feedbacks.

This class provides methods which are called by the FeedbackController on certain events. Override the methods as needed.

As a bare minimum you should override the on_play method in your derived class to do anything useful.

To get the data from control signals, you can use the "_data" variable in your feedback which will always hold the latest control signal.

To get the data from the interaction signals, you can use the variable names just as sent by the GUI.

This class provides the send_parallel method which you can use to send arbitrary data to the parallel port. You don't have to override this method in your feedback.

`inject` (*module*)

Inject methods from module to Feedback Controller.

`on_control_event` (*data*)

This method is called after the FeedbackController received a control signal. The FeedbackController parses the signal, extracts the values stores the resulting tuple in the object-variable "data" and calls this method.

Override this method if you want to react on control events.

`on_init` ()

This method is called right after the feedback object was loaded by the FeedbackController.

Override this method to initialize everything you need before the feedback starts.

`on_interaction_event` (*data*)

This method is called after the FeedbackController received a interaction signal. The FeedbackController parses the signal, extracts the variable-value pairs, stores them as object-variables in your feedback and calls this method.

If the FeedbackController detects a “play”, “pause” or “quit” signal, it calls the appropriate on_-method after this method has returned.

If the FeedbackController detects an “init” signal, it calls “on_init” before “on_interaction_event”!

Override this method if you want to react on interaction events.

on_pause()
This method is called by the FeedbackController when it received a “Pause” event via interaction signal.

Override this method to pause your feedback.

on_play()
This method is called by the FeedbackController when it received a “Play” event via interaction signal.

Override this method to actually start your feedback.

on_quit()
This Method is called just before the FeedbackController will destroy the feedback object. The FeedbackController will not destroy the feedback object until this method has returned.

Override this method to cleanup everything as needed or save information before the object gets destroyed.

on_stop()
This method is called by the FeedbackController when it received a “Stop” event.

Override this method to stop your feedback. It should be possible to start again when receiving the on_start event.

send_parallel(*data, reset=True*)
Sends the data to the parallel port.

send_udp(*data*)
Sends the data to UDP

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

2.2 MainloopFeedback — Base Class for Feedbacks with a Mainloop

class MainloopFeedback(*port_num=None*)
Mainloop Feedback Base Class.

This feedback derives from the Feedback Base Class and implements a main loop. More specifically it implements the following methods from it’s base:

on_init on_play on_pause on_stop on_quit

which means that you should not need to re-implement those methods. If you choose to do so anyways, make sure to call MainloopFeedback’s version first:

def on_play(): MainloopFeedback.on_play(self) # your code goes here

MainloopFeedback provides the following new methods:

init pre_mainloop post_mainloop tick pause_tick play_tick

the class takes care of the typical steps needed to run a feedback with a mainloop, starting, pausing, stopping, quiting, etc.

While running it’s internal mainloop it calls tick repeatedly. Additionally it calls either play_tick or pause_tick repeatedly afterwards, depending if the Feedback is paused or not.

init()
Called at the beginning of the Feedback's lifecycle.

More specifically: in Feedback.on_init().

pause_tick()
Called repeatedly in the mainloop if the Feedback is paused.

play_tick()
Called repeatedly in the mainloop if the Feedback is not paused.

post_mainloop()
Called after leaving the mainloop, e.g. after stop or quit.

pre_mainloop()
Called before entering the mainloop, e.g. after on_play.

tick()
Called repeatedly in the mainloop no matter if the Feedback is paused or not.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

F

L

INDEX