Falcon Documentation

Release 1.0

Fabian Kloosterman, Davide Ciliberti

Jun 23, 2020

CONTENTS

Falcon is a software for real-time processing of neural signals to enable short-latency closed-loop feedback in experiments that try to causally link neural activity to behavior. Example use cases are the detection of hippocampal ripple oscillations or online decoding and detection of hippocampal replay patterns.

CHAPTER

ONE

HIGH LEVEL OVERVIEW

At its core, Falcon executes a user-defined data flow graph that consists of multiple connected nodes (processors) that perform computations on one or more streams of input data and produce one or more streams of output data. Some types of processors produce output data without accepting input data (sources), whereas other types of processors consume input data without produce output data (sinks). An example data flow graph is shown in the figure below [TODO].

The data flow graph is specified in YAML text format and defines all the processor nodes and their interconnections. Falcon ships with a number of built-in processors that can be used to construct custom applications (see *Processors*). Falcon can be easily extended with new kinds of processors, although this requires modern C++ programming skills (see *Extending overview*).

By design, Falcon software is only concerned with the execution of data flow graphs and it does not include a graphical user interface. Rather, separate client applications interact with a running Falcon instance through network communication. In this way, dedicated user interfaces may be built in any programming language for particular user applications (as determined by the data flow graph). A generic Python control client is shipped with Falcon (see *Generic control app*) and serves as an example for how to build a user interface.

CHAPTER

TWO

INSTALLATION

2.1 Installation using graphical installer

You just want to run falcon with a set of selected extensions and you don't want to look at the code ? This part is for you. In parallel, we developed a cmake-client python gui to automatically create an installation of Falcon.

```
git clone https://bitbucket.org/kloostermannerflab/fklab-cmake-gui
cd fklab-cmake-gui
conda env create -f falcon.yaml
conda activate falcon
python setup.py build_ext --inplace
pip install -e . --no-deps
fklab-build
```

This environment will install the needed dependencies in a conda environment. You can read the readme.md in the fklab-cmake-gui repository for more information on how the gui is working.

Information specific to the build of falcon asked in the app :

- repository path: https://bitbucket.org/kloostermannerflab/falcon-core.git (the ssh path is also valable)
- (last) version : 1.1.0 => See the changelog to see other available versions.

A grid with available extensions will be display. You can add your own extensions if needed but note that falcon-core does not contains any extensions. If you want to use the core extensions, you need to have the "falcon-fklab-extension" selected. The extensions are stored in this repository.

This step is optional and will allow falcon to more finely control CPU core utilization.

```
sudo setcap 'cap_sys_nice=pe' `which falcon`
```

Once, the app has been installed (without errors) you can continue to the section *Launch Falcon from command line*.

2.2 Manual installation

The building of the falcon app is based on cmake and fetch_dependency to manage libraries and extensions.

2.2.1 Download

Download the latest source from https://bitbucket.org/kloostermannerflab/falcon-core/downloads/?tab=downloads.

2.2.2 Dependencies

- **CMAKE**

The build system is based on CMake (minimum version 3.11). Last version of CMake are available through pip.

```
pip install cmake
```

- **zeromq**

```
sudo apt-get install libzmq3-dev
```

- **g++-5 (or upper)**

G++ v.5 or upper is needed in order to have all libraries of the C++11 standard. In order to install it type in a terminal:

```
sudo apt-get install g++
```

- **External libraries included in source tree** (just for information, you don't need to do anything normally)

g3log cmdline (header only library) disruptor

2.2.3 Build instructions

Compiling falcon has only been tested with GNU g++ compiler. You should use version 5 or upper.

The falcon-core repository does not contains any extensions. You will have to add, at least, the core extension to the CMakeList.txt.

2.2.4 How extensions are found and added to build ?

Extensions are added through the FetchContent feature of CMake. It allows to link in the Falcon CMake the different git repository (or local folder) containing the extension. This extension needs to contain a CMake. This solution allows to use a specific version of an extension by adding a tag version in the option. The core extension are listed in the extension.yaml at the root of the repository.

The CmakeList.txt will read the extension.txt file described below :

```
enable , extension name , extension path , extension version (optional)
1 , extensions , https://bitbucket.org/kloostermannerflab/falcon-fklab-extensions
```

Enable can be 3 different values : 0 (not build)/ 1 (build)/ dev (develop mode)

The build mode will import the repository in the commit state (when not specified, the commit is the last one on the master head). The dev mode will build the repository in its actual local state.

Python install

You can also used the fklab-build tool to build the app in fast mode without using the gui. See installation.

```
fklab-build --gui false
```

Note: Cmake options are available to configure the build.

It can be added with the argument –build_options OPTIONS (without -)

Command line build

So, to compile issue the following commands while in the falcon root directory:

```
mkdir build
cd build
cmake ..
make install
```

For more information on how to integrate third party extension to the build, refer to the build system documentation.

2.2.5 Installation instructions

```
cd falcon
sudo setcap 'cap_sys_nice=pe' ./falcon
```

The last step is optional and will allow falcon to more finely control CPU core utilization.

CHAPTER

THREE

USER GUIDE

3.1 Launch Falcon from command line

3.1.1 Command-line

```
falcon graph.yaml
```

```
```

usage: falcon [options] ... [graph file] ... options:

-c, --config	configuration file (string [=$HOME/.falcon/config.yaml])
-a, --autostart	auto start processing (needs graph)
-d, --debug	show debug messages –noscreenlog disable logging to screen –no-cloudlog disable logging to cloud
-t, --test	turn testing on by default
-v, --version	Show the falcon version number and exit.

-?, –help print this message

A configuration file can be used to specify automatically this options + others used to affine the control in the whole system. Check out the *Configuring falcon*.

Note: The option specified in command line are prioritized against the options specified in the config file.

3.1.2 Control commands with a Falcon running graph

Keyboard commands

key	action
q	quit
i	info
r	start processing graph (run)
t	start processing in test mode
s	stop processing graph
k	stop processing and quit (kill)
y	display the current graph in yaml format

Falcon-client gui

The graph is also controllable from a (remote) client. See this section for more details : *Generic control app*

3.2 Define processing graph

At its core, Falcon is primarily concerned with the execution of a data flow graph. The graph describes how data streams flow from one processor node to the next, and how variables (states) are shared between processor nodes.

The data flow graph, i.e. the processor nodes and their options and the connections between nodes, is specified in YAML format. Here is an example that defines three processor nodes, one node of class *NlxReader* that is called *source* and two nodes of class *DummySink* that are called *sink1 sink2*. The output ports *tt1* and *tt2* of the source node are connected to the input port *data* of *sink1* and *sink2* respectively. Finally, it is specified that the two sinks share their *tickle* state. More information about the syntax for specifying processor nodes, connections and shared states follows below.

The graph is shared in 3 sections : - falcon : could contains in the future some generic options as the version - graph : either the graph path (in remote-side by using uri) or fully defined in this section - options : section to override some specific options in the graph.

Graph - client side : (personalized for each experimentation)

```
falcon:  # could be used for generic falcon options
    version : 1.0  # minimum required falcon version for this graph

graph : graphs://graph_file.yaml (see yaml block below)

options:
    source:
        class: NlxReader
        options:
          channelmap:
            tt1: [1,2,3,4]
            tt2: [5,6,7,8]
```

Graph template - remote side : (template usable by everyone)

```
processors:
  source:
    class: NlxReader
    options:
      batch size: 1
      update interval: 0
      npackets: 1000000
      channelmap:
        tt1: [1,2,3,4]
        tt2: [5,6,7,8]
    advanced:
      threadpriority: 100
      threadcore: 4
  sink(1-2):
    class: DummySink

connections:
  - source.tt(1-2)=p:data.f:sink(1-2)
```

(continues on next page)

(continued from previous page)

```
states:
  - [sink1.tickle, sink2.tickle]
```

We see in this example that the client-side graph will override the channelmap option in the source processor.

3.2.1 Processor nodes

The processor nodes that make up the data flow graph are specified in the *processors* section of the graph definition. Each processor node has a unique user-defined name, a *class* that specified what type of processor node should be created and processor type specific options. In the example above, the first entry in the *processors* section specified a node with the name *source* that is of type *NlxReader*. In addition, a number of options are set that are specific to the NlxReader processor (i.e. *batch_size*, *channelmap*, etc.). See TODO for more information about the specific options for each of the processor classes that are shipped with Falcon. (Note: a number of advanced options are available for each processor to control low-level execution parameters - see TODO for more information)

Sometimes, one needs to define multiple processor nodes of the same class and with the same options. In that case, a short hand notation is available to define a numbered range of nodes with the same base name: `base(start-end)`. Thus, `sink(1-2)` defines two nodes named *sink1* and *sink2*.

3.2.2 Data stream connections between processors

Processor nodes have input ports for receiving data streams and output ports for generating data streams. Input and output ports can have one or more *slots* that handle the 1-to-1 connection between an upstream processor node and a downstream processor node. How connections between processor nodes should constructed is specified by a list of connection rules in the *connections* section.

Each connection rule describes how the output of upstream processors is mapped to the input of downstream processors. In the simplest case, a single output port/slot is connected to a single input port/slot. The general form of such a simple connection rule is `processor.port.slot=processor.port.slot`. Here, the what comes before the = sign is the upstream connection address and the what comes after the = sign is the downstream connection address. A connection address consists of three parts separated by periods that refer to the processor name, port name and slot index. For example, `upstream.out.0=downstream.in.0` defines an explicit connection from the first slot of output port *out* on processor *upstream* to the first slot of input port *in* on processor *downstream*. It is possible to let Falcon select the first available slot on the output or input ports, by leaving out the slot number (e.g. `upstream.out=downstream.in`).

Using the range notation (i.e. `(start-end)` or `(1,2,4-8)`), multiple connections can be specified in one compound rule. All three parts of the connections address (i.e. processor, port and slot) accept a range specifier. For example, the connection rule `upstream(1-2).out=downstream(1-2).in` will be expanded into two connection rules: `upstream1.out=downstream1.in` and `upstream2.out=downstream2.in`. Likewise, `upstream.out(1-2)=downstream.in(1-2)` will be expanded into the simple connection rules: `upstream.out1=downstream.in1` and `upstream.out2=downstream.in2`.

In some case, one may want to map multiple output ports of a single upstream processor to a input port on multiple downstream processors (i.e. fan-out from single processor to multiple processors) or the other way around (i.e. fan-in from multiple processors to a single processor). Such a connection pattern can be specified in a compact way be reordering the address parts in the rule. Since it is assumed by default that the order of the address parts is processor, port, slot, a part identifier has to be explictly added. For example, `upstream(1-2).out=p:in(1-2).f:downstream` says that the *out* port of two upstream processors are mapped to the two *in* ports on the single downstream processor. In this rule, the order of processor and port parts on the right side is changed, such that the ports (prefixed with the `p:` specifier) come first and the processor (prefixed with the `f:` specifier) comes next. This compound rule is equivalent to the following two simple connection rules: `upstream1.out=downstream.in1`

and `upstream2.out=downstream.in2`. In the same way it is possible to map from processors/ports to slots and vice versa using the `s:` part identifier for slots.

3.2.3 Shared states

Shared states are variables that are exposed by processor nodes and which can be shared by multiple nodes. In addition, such states may also be made publicly accessible to clients. For example, the *levelcrossingdetector* processor class exposes a *threshold* state that represents the threshold used internally for detecting a level crossing in an input signal. Clients have write access to the *threshold* state and can both read and update the value while the data flow graph is executed. The *threshold* state of multiple *levelcrossingdetector* processor nodes in the same graph can also be coupled to make sure that they all use the same threshold value.

Which processor states should share their value and under what name this shared state becomes available to clients is specified in the *states* section of the graph definition. The *states* section contains a list of shared state definitions. In its full form, this definition maps an alias to a list of states. In the following example the values of state1 (on processor1) and state2 (on processor2) are shared and the shared state is known under the alias *value*.

```
states:
  - value:
      states: [processor1.state1, processor2.state2]
      permission: read
      description: A shared value between processors
```

The *permission* option in the example sets the external read/write permission for clients. Valid values are *read*, *write* and *none*. The *description* option is a short description of the shared value that clients can present to the user.

If the additional options are not needed, then the shared state can be specified less verbosely with or without alias:

```
states:
  - value: [processor1.state1, processor2.state2]
  - [processor3.state3, processor4.state4]
```

Note: Processor name, shared state, options accept space, -, _ as equivalent. In internal, it is always replace by "-".

3.3 Configuring falcon

Falcon has several options that can be configured through a configuration file that is written in YAML format. By default, Falcon will look for a configuration file in the $HOME/.falcon folder and if this file does not exist, a new one will be created with default values. A configuration file path can also be specified on the command line (see *Launch Falcon from command line*).

The configuration file has a number of sections with configurable options. An example configuration file is shown below:

```
debug:
  enabled: false
testing:
   enabled: false
graph:
  autostart: false
  file: ""
network:
```

(continues on next page)

(continued from previous page)

```
  port: 5555
logging:
  path: "./"
  screen:
    enabled: true
  cloud:
    enabled: true
    port: 5556
server_side_storage:
  environment: "./"
  resources: "$HOME/.falcon"
```

3.3.1 network

Falcon communicates with clients over a network connection. In the network section you can set the *port* that Falcon uses for this communication.

3.3.2 graph

You can specify a graph definition *file* that is loaded when Falcon is started. If a graph file is also specified separately on the command line, then this will override the configured file. If you would like Falcon to automatically start execution of the processing graph (without the user having to manually start execution by pressing a key), then set the *autostart* option to true.

3.3.3 logging

Falcon provides information about its operation through log messages that are saved to a file on disk, shown on screen and broadcast over the network to clients. Use the *path* option to specify the path where the log file should be saved. You can enable/disable logging to screen and network by setting the screen.enabled and cloud.enabled properties to true/false. For logging to the cloud, you can additionally set the network *port*.

3.3.4 server_side_storage

To refer to paths and resources on the computer that runs Falcon, users can configure a mapping between URIs (uniform resource identifiers) and paths. Falcon defines a number of URIs mostly for internal use, including the top level path for storing data generated by individual processors (set by the *environment* option) and the top level path for resources such as digital filter definitions and example graph definitions (set by the *resources* option).

> **Warning:** About the resource server side storage, each extensions will also contained a resource folder with filter definition, graph definitions... etc.
>
> During the build of the app, these resources are gathered and copied in the build folder, then in the installation folder (if make install). Falcon will automatically generate the resource path. Be careful if it is override in the config file, it will lose access to this resource folder.

Users can also specify custom URIs that point to paths on the computer running Falcon. In the *server_side_storage* section of the configuration, the *custom* option can be set to a map that links a URI to a path.

For example based on the example below, a file example.txt in the folder "/path/to/user/location" can be reach in falcon by using "mypath://example.txt".

```
server_side_storage:
  environment: "./"                    # Where to store the output data
  resources: "/"           # Build path where resource from extensions are stored
↪(filter definitions, graph definitions...)
  custom:
    mypath: "/path/to/user/location"
```

All path should be an existing path.

3.4 Frequent errors

ERR exception yaml-cpp: error at line 0, column 0: operator[] call on a scalar

–> check that all processor options are set with *:* and not with =

```
ERROR       Error handling command: buildfile Error: yaml-cpp: error at line 0,
↪column 0: bad conversion
ERR exception yaml-cpp: error at line 0, column 0: bad conversion
```

–> check that your using the correct option name in the config file and make sure all necessary options are listed in the config file

```
ERR exception No storage context "repo" exists.
ERROR       Error handling command: buildfile Error: No storage context "repo" exists.
```

–> Check that you have a custom "repo" path in your config file :

```
server_side_storage:
  environment: path/output_data
  resources: path/input_data
  custom:
    repo: Path/To/Repo/falcon
```

3.5 Debug

To log debug messages while running Falcon, set the *enabled* option to true.

3.6 Test mode

To run Falcon in testing mode, set the *enabled* option to true.

3.7 Test graphs

CHAPTER

FOUR

EXTENDING FALCON

4.1 Extending overview

An extension can contains : - processors - datatype - resources

4.2 Overview of the existing extensions

4.3 Create new processor

The new processor class needs to derive from the IProcessor class.

Two important inputs are :

- const YAML: Node &node : The description from the graph of the node with the different parameters related to the processor
- Context : There is a GlobalContext and a ProcessingContext structure available to be use in the class. (see context documentation)

Virtual methods from the IProcessor class are available to be override:

- *Configure(const YAML::Node &node, const GlobalContext& context)* : The graph (yaml file) describe the node with some parameters specific to the processor. This method is the time to read it, process it (checking errors) and store it for later.
- *CreatePorts()* : This part make use of the internal available methods from Iprocessor (see the API documentation) for creating input port (*create_input_port*), readable sharable state (*create_readable_shared_state*) and output port (*create_output_port*).
- *CompleteStreamInfo()* set extra information for output datastream and parameters specific to the datatype, check additional conditions as for example same numbers of input / output slot if there are related
- *Prepare(GlobalContext& context)* : prepare state of the node aka connecting servor . . . etc.
- *Unprepare(GlobalContext& context)* : undo the prepare method
- *Preprocess(ProcessingContext& context)* : pre-process state of the note aka clear states The Preprocess part is synchronized between processor. So, all processor will wait for that others finished this part. At the difference of prepare step, it is done in their own thread.
- *Process(ProcessingContext& context)* : process state of the node : for loop while the context does not send a terminated signal

```
while(!context.terminated() && others conditions ...){
    // process

    // Don't forget to use LOG
}
```

- *Postprocess(ProcessingContext& context)* : post-process state of the node aka log info, clean up and close communication
- *TestPrepare(ProcessingContext& context)* : use in case of integration test
- *TestFinalize(ProcessingContext& context)* : use in case of integration test

Finally, don't forget to add your processor in the namespace by using

```
REGISTERPROCESSOR(ProcessorName)
```

4.3.1 Documentation :

The documentation of your processor will need to specify what it is doing, its inputs and outputs but also how to describe it in the graph definition yaml file (available options . . . etc.).

To do this "doc.yaml" need to be added next to the .cpp with these entrees:

```
Description: short description

Long description: long description (e.g. explanation of algorithm)

Input ports:
  - name: name
    type: MultiChannelType
    slots: # or [#, #]
    description: description

Output ports: ... same as input ports ...

Options:
  - name: name
    type: double
    default: ...
    description: ...

Methods:
  - name: name
    arguments:
      - name: default value
      - ...
    returns: ...
    description: ...

States:
  static:
    - name: name
      type: double
      initial value: ...
      shared: true/false
      external access: read or write or none
```

(continues on next page)

(continued from previous page)

```
    description: ...

  producer:
    - name: name
      type: double
      initial value: ...
      cooperative: true/false
      external access: read or write or none
      description: ...

  broadcaster:
    - name: name
      type: double
      initial value: ...
      external access: read or write or none
      description: ...

  follower:
    - name: name
      type: double
      initial value: ...
      external access: read or write or none
      description: ...
```

To correctly build the documentation, this file needs to be in yaml format.

4.3.2 Documentation useful for the development

- graph system
- logging system
- build system

4.3.3 Development build

While populating your extension in the falcon CMake, you can override the git LOCAL location with the dev option in the extensions.txt file.

4.4 Create new data type

4.5 Tool extension

4.6 How to package an extension

Example of an extension structure :

```
extension_repository :
    - processors
        - processor_name
```

(continues on next page)

(continued from previous page)

```
            - doc.yaml
            - processor_name.cpp
            - processor_name.hpp
            - CMakeList.txt
    - datatypes
        (similar structure to processors)
    - lib
        - Lib1
            - code
            - CMakeList.txt
        - Lib2
        - CMakeList.txt
    - resources
        - graphs
        - filters
        - others folder (but it will need to configure its own uri to reach it in␣
↪falcon)
    - tools
```

Minimal CMakeList.txt in each extension (lib, processor, datatype) :

```
add_library( name "name.cpp" )
TARGET_LINK_LIBRARIES( name lib_name )
```

Note: lib_name could be lib added in the extension but also already present in the falcon-core.

CHAPTER

FIVE

BUILDING CUSTOM CLIENTS

5.1 Interaction with Falcon

Falcon operation and data flow graph execution can be controlled using keyboard commands (see Usage), but this does not expose the full set of features. Another way to interact with a running Falcon instance is through a network connection.

5.1.1 Send commands to Falcon

there is set of command which can be send to Falcon

- to manage the general app :

command	action
quit	quit the app
kill	stop processing and quit (kill)
info	generate basic information answer as falcon version and state
documentation	generate yaml documentation with a node for each processor registered in Falcon

- to manage the graph : (node the graph prefix for each command)

command	action
graph build	build the graph
graph documentation	generate yaml documentation with a node for each processor in the build graph
graph run	start processing graph (run)
graph test	start processing in test mode
graph stop	stop processing graph
graph yaml	display the current graph in yaml format
graph apply [yaml Node]	apply a exposed method in the processor
graph retrieve [yaml Node]	retrieve a particular processor state/definition
graph update [yaml Node]	update a particular processor state/definition

5.1.2 Receive log messages

5.2 Create custom user interface

5.3 Python library

Nothing has been developped here yet but the objective is to create a graph drawing tool with a processor library. - drawing graph - validate graph - changing options, shared state ... etc.

CHAPTER

SIX

DEVELOPER'S GUIDE

6.1 Code organization and style guide

6.2 Overview

6.3 Logging system

Falcon's logging system is based on the g3log library, which is included in the Falcon source tree.

The logging library is used to provide information about the internal state and operation to the user (or developer). There are a number of different types of log messages (i.e "log levels") defined, each with their own format and usage patter as listed below:

DEBUG debug info

INFO general info

STATE TBD

EVENT TBD

UPDATE TBD

WARNING TBD

ERROR TBD

FATAL TBD

To log messages in the code, one needs to include the *g3log/src/g2log.hpp* header file and then do for example:

```
LOG(INFO) << "my informative message";

LOG_IF(DEBUG, condition) << "If [true], then this text will be logged";
```

In Falcon, three destinations ("sinks") for log messages are defined. First, log messages are always saved to a log file. The path of this file is set using the *logging.path* configuration option (see *Configuring falcon*). Second, log messages are displayed in the terminal in which Falcon was started (but only if the *logging.screen.enabled* configuration option is set to true). Finally, log messages are broadcast to clients using a ZMQ publisher network socket (if the *logging.cloud.enabled* configuration option is true). The network port is configurable (see *Configuring falcon*).

Here is an example in Python how to receive log messages broadcast to port 5556 on the local computer:

```
import zmq

context = zmq.Context()
socket = context.socket( zmq.SUB )

socket.connect( "tcp://localhost:5556" )

socket.setsockopt(zmq.SUBSCRIBE, "")

while True:
    message = socket.recv_multipart()
    print( message )
```

6.4 Command system

namespace commands

Command sources (cloud, command line, keyboard) implemented by three classes that derive from CommandSource base class.

Commands are requested (serially) from sources and handled by CommandHandler class in main thread.

Graph commands are forwarded to GraphManager and handled in graph thread.

Replies are sent to the original requester of the command.

6.5 Configuration system

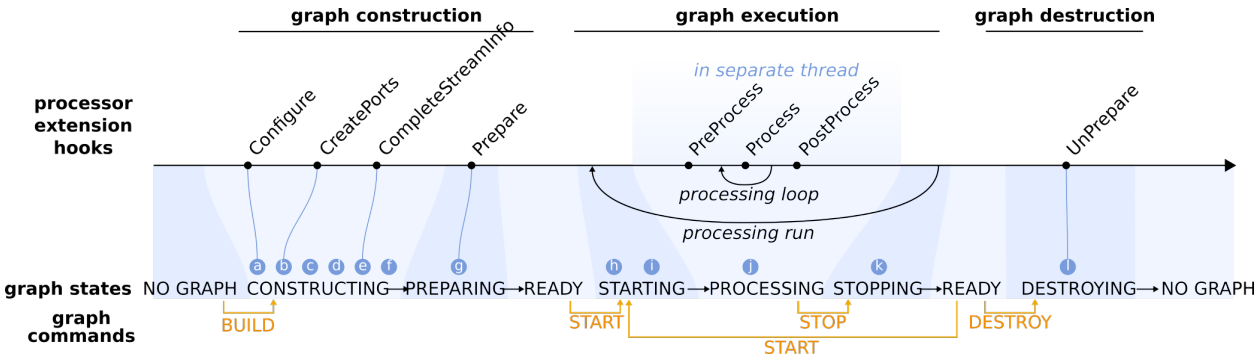
Configuration base class in utilities library.

FalconConfiguration class.

Configuration YAML file.

6.6 Thread management

6.7 Graph life cycle



a. Construction and configuration of processor nodes. Processor nodes are constructed by a factory that has a registry which maps processor names to the associated processor class. The factory calls the default constructor and it is convention that processor extensions do not overload the default constructor. Rather, initialization and configuration of a processor node is done in a separate step. The IProcessor base class will call the Configure hook that provides processor extensions a mechanism for initialization immediately after construction. The YAML user options are passed into the Configure hook, and most processor extensions will parse the options, validate their values and store the option values for future use. The IProcessor base class will take care of the advanced options (i.e. options.test and all advanced options).

b. Creation of input/output ports and shared states. In the next step after construction and configuration, processor nodes can create the input and output ports for handling streaming data, and states for sharing variables. Processor extensions should overload the CreatePorts hook and use the available convenience methods to create the ports and states (create_output_port, create_input_port, create_readable_shared_state and create_writable_shared_state). Often a pointer to the port or state is stored for future use (this is convenient, although not strictly necessary). If the user has explicitly specified ring buffer sizes for the output ports in the advanced options, these will be applied immediately after all ports have been created. The data packets that stream from output to input ports carry specific types of data, and each input/output port will only handle the one data type that was specified during creation.

c. Connecting processor nodes. For all the connections specified in the graph definition, a link is made between the involved output and input ports. In most cases, a slot on the in/output port is automatically created when a connection is made, which makes is easier to make multiple connections with the same ports. Connections between ports are only established after a compatibility check, i.e. the input port should be able to handle the type of data packets that are produced by the output port. This either means that the output port produces data packages of the same type as the input port or a more specialized (derived) data type. In addition, data types can be parameterized, and the specific parameters set on the output port data stream should be compatible with (i.e. within the range of) the capabilities of the input port.

d. Linking shared states. The next step links processor states that should be shared between multiple processor nodes. Optionally, shared states are made accessible to the user through a dedicated name (alias). States can only be linked if they support the same data type and if their access permissions are compatible.

e. Negotiation and validation of streaming data connections. In this step, the specific parameters (i.e. stream rate and data type specific parameters) of the output data streams are finalized. Processor extensions can overload the CompleteStreamInfo hook to customize this step. For each input slot, a validation step is performed to make sure parameters of incoming data are compatible with the input slot data type specific capabilities. It is further checked that all input slots on all input ports are connected to an upstream processor node. Non-connected output slots are acceptable and only emit a warning.

f. Creation of ring buffers. At this point, everything is ready set up the ring buffers for handling the data streams between processors.

g. One-time preparation of processor nodes. The final step in building a graph is to give processor nodes the opportunity to perform preparations, such as resource allocations, file I/O or CPU intensive operations, that only have to be done once in the processor lifetime. Processor extensions can use the Prepare hook for this purpose.

h. Preparing processor nodes and their ports for graph execution. Prior to execution of a graph, all processor nodes and their ports (in particular the ring buffers) are prepared internally and reset to their initial state.

i. Start threads for processor nodes. Next, the dedicated thread for each processor node is started.

j. Run processing loop. The process loop only starts after *all* processor nodes have finished PreProcess-ing.

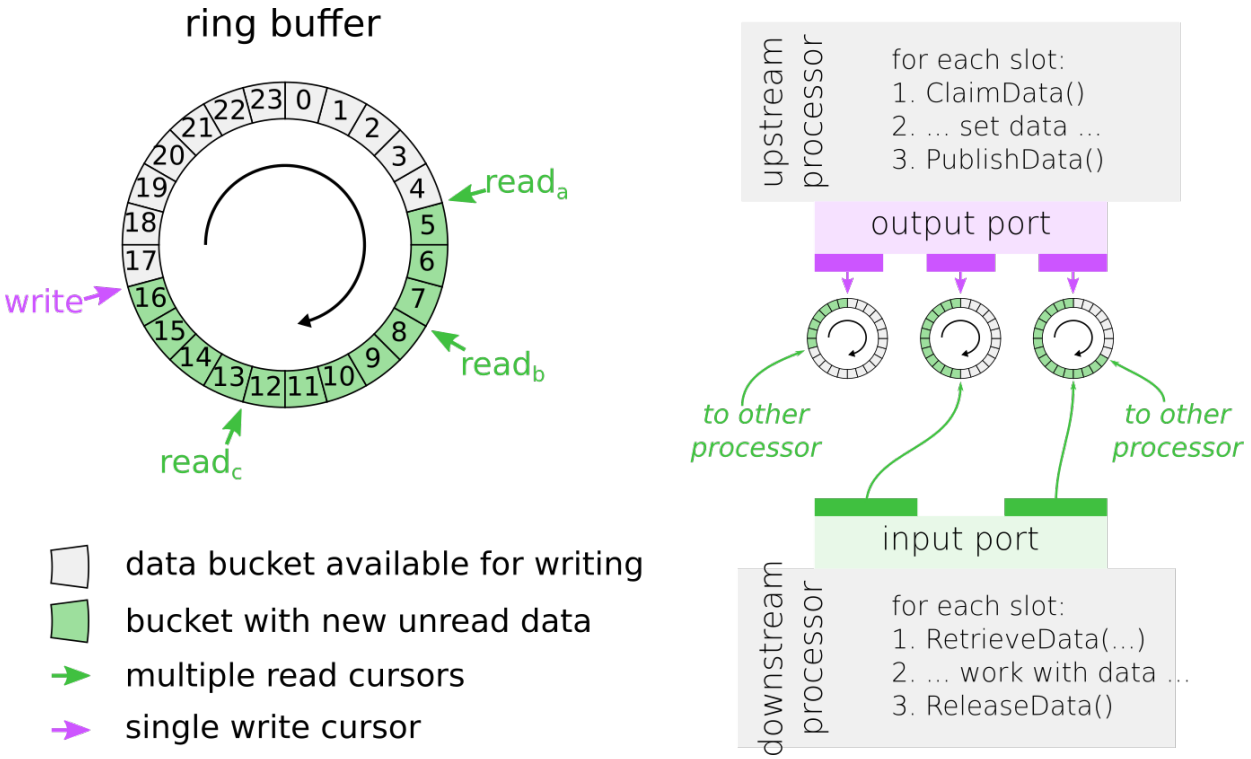
k. Terminate processing loop.

l. Clean up of processor nodes and destruction of graph. Before all graph elements are destructed, processor nodes have the ability to release resources that were previously acquired in the Prepare hook, by overloading the UnPrepare hook.

6.7.1 IProcessor

(Ports, slots, threads, port policies)

6.7.2 Ringbuffer based on disruptor



Read data on input slot

- **Step 1. Retrieve pointers to next data packet(s).** Use RetrieveData, RetrieveDataN or RetrieveDataAll to retrieve respectively one data packet, N data packets or all available data packets. By default, these methods will block until enough data is available. If a time-out has been set and there is still not enough data available after time is up, these methods will either return no data or the cached last data packet (if caching was enabled).
- **Step 2. Use the retrieved data.**

  > **Warning:** Do not overwrite or alter the data, as other read cursors may still need to access the same data.

- **Step 3. Release the data packets and move ahead read cursor.** Always use the ReleaseData method after you are done with the retrieved data packets, so that the data packets can be reused.

Example

For cursor reada, RetrieveData(2, data) will retrieve pointers to the data packets at positions 5 and 6. A subsequent call to ReleaseData() will move the cursor two positions ahead and make positions 5 and 6 available for writing.Note that the read cursor that lags behind the most will determine to what position the write cursor can move.

Write data to output slot

- **Step 1. Claim data packets for writing.** Use ClaimData or ClaimDataN to claim respectively one or N data packets.These methods will always block until enough positions on the ring buffer are available for writing. If needed,the data packets can be cleared automatically so that any previous data is removed.
- **Step 2. Write new data to the data packets.** Don't forget to update the timestamps as well.
- **Step 3. Publish the data to the ring buffer using the the PublishData() method.** Always pair a call to one of the ClaimData methods with a call to PublishData to properly advance the write cursor and make the new data available for readers.

Example

ClaimData() will return a pointer to the next available data packet, in this case at position 17. A subsequent call to PublishData() will advanced the write cursor and make the data at position 17 available to the read cursors.

6.7.3 Data type

(hierarchy, properties, capabilities, compatibility, serialization)

6.7.4 Context

(RunContext, ProcessingContext)

6.7.5 States, atomic variables

6.8 Utilities

6.8.1 ZeroMQ

6.8.2 Time

6.8.3 Math

6.8.4 Options

CHAPTER

SEVEN

FALCON CONTROL CLIENT (SEPARATE REPOSITORY)

7.1 Generic control app

7.1.1 Installation

The falcon-client ui is used to remote control of the falcon system.

Clone the python repository :

```
git clone https://user@bitbucket.org/kloostermannerflab/falcon-client.git
```

Install in your python path :

```
cd falcon-client
python setup.py build_ext --inplace
pip install -e . --no-deps
```

The ui can be launched when the falcon program is already running.

```
simple_client
```

7.2 Graph display app

In the same repository as installed the previous step, you have 4 python clients :

- Plot mua stats in live with the command *live_plot_mua_stats*
- Plot behavior in live with the command *live_plot_behavior*
- Plot decoding error in live with the command *live_plot_decoding_error*
- Plot ripple stats in live with the command *live_plot_ripple_stats*

CHAPTER

EIGHT

RIPPLE DETECTION CLIENT (SEPARATE REPOSITORY)

8.1 Hippocampal ripple detection

8.1.1 Overview

8.1.2 Implementation

8.1.3 processors

- source : NlxReader
- ripple filter 1 : *MultiChannelFilter*
- ripple filter 2 : *MultiChannelFilter*
- HIPPOCAMPUS detector: *RippleDetector*
- CORTEX detector: *RippleDetector*
- **event filter** [*EventFilter*]
  - blocking event : Event from the cortex
  - input event : Event from the hippocampus
- network sink : *ZMQSerializer*
- event sink : *EventLogger*
- datasink ev : *FileSerializer*
- datasink ripple_stats : *FileSerializer*
- ttl output: SerialOutput (not yet ported on the new falcon)

8.1.4 States writable by the user

Processor *RippleDetector*:

- threshold dev (double)
- smooth time (double): integration time for signal statistics. Must be a positive number.
- **detection lockout time (double): Current refractory period following threshold crossing that is not** considered for updating signal statistics and for event detection.
- stream events (bool)

- stream statistics (bool)

Note: As their is two RippleDetectors, these states are available for each processor.

8.1.5 Output

- ripple events : *EventData*
- ripple statistics

8.2 Ripple detection graph

```
processors:
    source:
        class: NlxReader
        advanced:
            threadpriority: 100
            threadcore: 0
        options:
            address: 192.168.3.101
            port: 26090
            batch size: 3
            update interval: 10
            channelmap:
                hp1: [13,20]
                cx1: [9]

    ripple filter(1-2):
        class: MultiChannelFilter
        options:
            filter:
                file: filter://iir_ripple_low_delay.filter

    HIPPOCAMPUS detector:
        class: RippleDetector
        options:
            threshold dev: 14
            smooth time: 7 # in seconds
            detection lockout time: 50 #ms
            stream events: true
            stream statistics: true
            statistics buffer size: 1.0 # sec
            statistics downsample factor: 4
            use power: true

    CORTEX_detector:
        class: RippleDetector
        options:
            threshold dev: 12
            smooth time: 8 # in seconds
            detection lockout time: 40 #ms
            stream events: true
            stream statistics: false
```

(continues on next page)

(continued from previous page)

```
            statistics buffer size: 1.0 # sec
            statistics downsample factor: 4
            use power: true

    event filter:
        class: EventFilter
        options:
            target event: ripple
            blockout duration: 40
            sync time: 1.5
            block wait time: 4 # below 3.5, asynch can occur
            detection criterion: "any" # 'any', 'all' or number
            discard warnings: false

    network sink:
        class: ZMQSerializer
        options:
            encoding: binary
            format: compact

    event sink:
        class: EventSink
        options:
            target event: ripple

    datasink ev:
        class: FileSerializer
        options:
            encoding: yaml
            format: full

    datasink ripplestats:
        class: FileSerializer
        options:
            encoding: binary
            format: compact

    ttl output:
        class: SerialOutput
        options:
            enabled: true
            target event: ripple
            message: "d" # d for delay stim, o for ontime
            lockout period: 250

connections:
    - source.hp1=ripple filter1.data
    - source.cx1=ripple filter2.data
    - ripple filter1.data=HIPPOCAMPUS detector.data
    - ripple filter2.data=CORTEX detector.data
    - HIPPOCAMPUS detector.events=event filter.events
    - CORTEX detector.events=event filter.blocking events
    - HIPPOCAMPUS detector.statistics.0=network sink.data
    - event filter.events.0=ttl output.events
    - event filter.events.0=event sink.events
    - event filter.events.0=datasink ev.data
    - HIPPOCAMPUS detector.statistics.0=datasink ripplestats.data
```

CHAPTER

NINE

FALCON CORE EXTENSION (SEPARATE REPOSITORY)

9.1 Overview

9.2 Data Types

The data streams that flow from processor to processor consist of data packets that carry the data of interest (the payload), a source timestamp (when the data packet was generated) and (optionally) a hardware timestamp (original timestamp of the external hardware that generated the data). Each input and output port on processor nodes only handles dedicated types of data. For example, some processors operate on arrays of analog data.

Data types in Falcon form a hierarchy from generic to specific. At the top of the hierarchy is the most generic data type "IData" that is the base for all other data types. As long as the data type of an input port is the same or more generic than the data type of the upstream output port, a connection can be made. Thus, a processor node with an input port that expects the most generic IData type, can handle incoming data streams of any other type.

Input ports may specify additional requirements for the incoming data. For example, an input port could indicate that it only supports multi-channel analog data with exactly 4 channels. An upstream processor with an output port that serves multi-channel data packets with fewer or more channels will thus not be compatible.

Below is a list of data types that are currently available in Falcon. See *Create new data type* for more information about how to add new data types.

9.2.1 EventData

General description

Data packets of the EventData type hold a string that describes the event. Processors that operate on EventData will usually specify what events they generate on their output ports or what events they expect at their input ports.

Payload details

name	type	description
event	string	event string

API

TODO (put here an overview of the public methods and a short description of what they do. Should we leave this to doxygen??)

Parameters

name	type	description	validation
event	string	default event string	cannot be empty

Capabilities

(none)

Binary Serialization

For serialization formats *FULL* and *COMPACT*, the event string is serialized as a 128-byte long string.

YAML Serialization

For serialization formats *FULL* and *COMPACT*, the following YAML is emitted:

event: [event string]

9.2.2 SpikeData

General description

Data packets of the SpikeData type hold a list of the timestamp and the amplitude of each spike. The list has a maximum size, specified at initialization time, which cannot be overreached.

Payload details

name	type	description
n_detected_spikes	unsigned int	the number of spikes added in the datatype
hw_ts_detected_spikes	vector of unsigned int64	hardware timestamp of each spike
zero_timestamps	vector of 100 (maximum of spikes in the buffer) unsigned int64	
amplitudes	vector of double	
zero_amplitudes	vector of 1600 (maximum of spikes in the buffer x maximum number of channels) double	

API

TODO

Parameters

name	type	description	validation
nchannels	unsigned int	number of channels	needs to be larger than 0
sample_rate	double		needs to be larger than 0
max_nspikes	size_t	buffer size	needs to be larger than 0

Capabilities

(none)

Binary Serialization

For serialization format *FULL*, the number of detected spikes, the hardware, zero timestamps, amplitudes of each spikes are serialized in a string format. For serialization format *COMPACT*, only the number of detected spikes and the amplitudes of each spikes are serialized in a string format.

YAML Serialization

For serialization formats *FULL* and *COMPACT*, the following YAML is emitted:

n_channels [channel number unsigned int] n_detected_spikes [number of spikes unsigned int]

if the previous one is superior to zero : ts_detected_spikes [hardware timestamps] spike_amplitudes [amplitudes]

9.2.3 MultiChannelData

General description

Data packet of the MultiChannelData type contains a generic nsamples-by-nchannels array of data.

Payload details

name	type	description
timestamp	uint64	
signal	vector of any datatype	

API

TODO

Parameters

name	type	description	validation
nsample	unsigned int	0	Number of samples cannot be zero and needs to be in range
nchannels	unsigned int	0	Number of channels cannot be zero and needs to be in range
sample_rate	double	1.0	Sample rate needs to be larger than 0

Capabilities

name	type	default	description
channel range	ChannelRange or Range<unsigned int>	(none)	the number of channels that is supported
sample range	SampleRange or Range<size_t>	[1, (maximum of the datatype can hold)]	the number of samples that is supported

Binary Serialization

For serialization formats *FULL* and *COMPACT*, The timestamps and the data are serialized.

YAML Serialization

For serialization formats *FULL* and *COMPACT*,

the following YAML is emitted:

timestamps: [timestamps (unsigned int)] signal: [data]

9.2.4 MUAData

General description

Data packet of the MUAData type describes the multi-unit activity by the frequency of spikes in a bin.

Payload details

name	type	description
mua	double	number of spikes / size of the bin * 1e3

API

TODO

Parameters

name	type	description	validation
bin_size	double (ms)	Input parameter with default value = 0	needs to be a positive value (>0)
n_spikes	double	Set separately after initialization of the datatype	

Capabilities

(none)

Binary Serialization

For serialization formats *FULL* and *COMPACT*, if compact format, only the mua data as being the number of spike / size of the bin * 1e3 is serialized. if full format, the number of spikes and the bin size are also sent.

YAML Serialization

For serialization formats *FULL* and *COMPACT*,

if compact format: the following YAML is emitted:

MUA: [number of spikes / size of the bin * 1e3]

if full format: the following YAML is emitted:

MUA: [number of spikes / size of the bin * 1e3] n_spikes: [number of spikes (double)] bin_size: [bin size (double)]

9.2.5 VectorData

General description

Data packet of the VectorData type contains a generic vector. The data are set separately, once the vector data has been created by allocating at initialization time in memory the right size of data needed.

Payload details

name	type	description
data	vector of any type	

API

TODO

Parameters

name	type	description	validation
size	unsigned int	size of the vector	cannot be zero

Capabilities

None

Binary Serialization

None

YAML Serialization

None

9.2.6 ScalarData

General description

Data packet of the ScalarData type contains a generic unique value. The value can be modified by a value of the same type.

Payload details

name	type	description
data	any type	scalar value

API

TODO

Parameters

name	type	description	validation
default_value	any type		

Capabilities

None

Binary Serialization

For serialization formats *FULL* and *COMPACT*, the data variable is sent as a string.

YAML Serialization

For serialization formats *FULL* and *COMPACT*,

the following YAML is emitted:

scalar_data [data]

9.3 Processors

Built-in processors can be found in the falcon-fklab-extensions [TODO / Link]

For each processor, a description is given of:

- what data streams it expects on the input ports
- what data streams are generated on the output ports
- the configuration options (used to initialize the processor)
- the shared states that are exposed

9.3.1 Detector

BurstDetector

Detect population bursts using a threshold crossing algorithm.

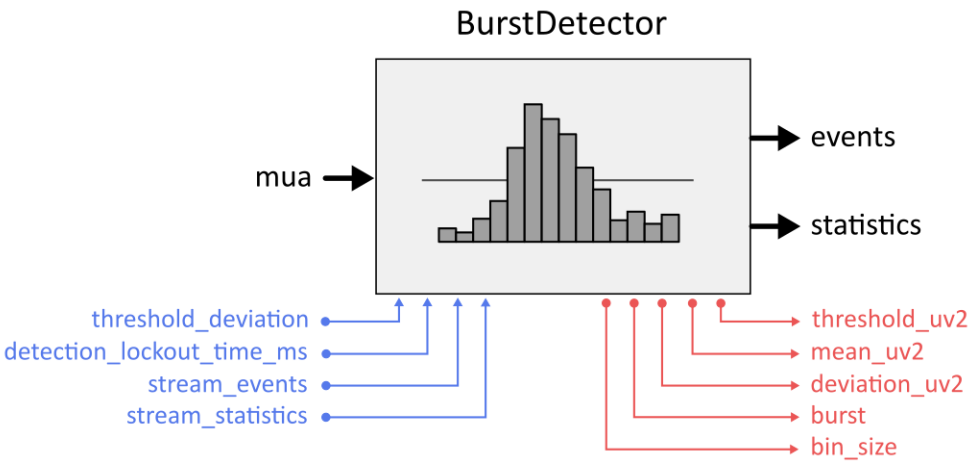


Table 1: **Input port**

port name	data type	slots	description
mua	*MUAData*	1	Binned multi-unit activity in Hz (e.g. from MUAEstimator).

Table 2: **Output port**

port name	data type	slots	description
events	*EventData*	1	A stream of 'burst' events.
statistics	*MultiChannelData* <double>	1	A stream of nsamples-by-2 arrays with the signal test value (first column) and the threshold (second column). The number of samples in each statistics data packet is set by the statistics_buffer_size option.

Table 3: **Options**

port name	data type	default	description
threshold dev	double	6.0	for threshold multiplier. Units: signal standard deviations.
smooth time	double	10.0 seconds	Integration time for estimating signal statistics. Must be a positive number (sec).
detection lockout time	double	30 ms	refractory period after threshold crossing detection that is not considered for updating of statistics and for detecting events. Must greater than 0 ms.
stream events	bool	True	enable/disable burst event output
stream statistics	bool	True	enable/disable streaming of burst detection statistics
statistics buffer size	double	0.5 sec	Buffer size (in seconds) for statistics output stream. This value determines the number of samples that will be collected for each data packet streamed out on the statistics output port. must be either equals or greater than 0.

Table 4: **Broadcaster States**

name	data type	initial value	external access	description
threshold	double	0.0	read-only	Current threshold that needs to be crossed
mean	double	0.0	read-only	Current signal mean. Units: same as input signal.
deviation	double	0.0	read-only	Current signal deviation. Units: same as input signal.
burst	bool	False	read-only	

Table 5: **Static States**

name	data type	initial value	external access	peers access	description
threshold deviation	double	option: threshold deviation	read-only	read/write	
detection lockout time	double	option: detection lockout time	read-only	read/write	Current refractory period following threshold crossing that is not considered for updating signal statistics and for event detection.
stream events	bool	option: stream events	read-only	read/write	
stream statistics	bool	option: stream statistics	read-only	read/write	

Table 6: **Follower State**

name	data type	initial value	external access	description
bin size	double	1.0	read/write	

LevelCrossingDetector

Detect a threshold crossing on any of the channels in the incoming MultiChannelData stream and emits an event in response

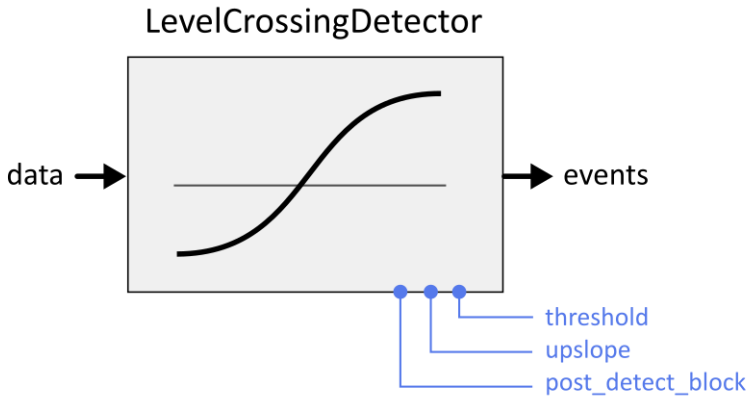


Table 7: **Input port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1	

Table 8: **Output port**

port name	data type	slots	description
events	*EventData*	1	A stream of events.

Table 9: **Options**

port name	data type	default	description
threshold	double	0.0	threshold that needs to be crossed
event	string	"threshold_crossing"	event to emit upon detection of threshold crossing
post detect block	unsigned int	2	refractory period after threshold crossing detection (in number of samples)
upslope	bool	True	whether to look for upward (true) or downward (false) threshold crossings

Table 10: **Static states**

name	data type	initial value	external access	peers access	description
threshold	double	option: threshold	read-only	write/read	Current threshold that needs to be crossed
post detect block	unsigned int	option: post detect block	read-only	write/read	
upslope	bool	option: upslope	read-only	write/read	

RippleDetector

Detect ripples in a MultiChannelData stream and emits a ripple event in response

Table 11: **Input port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1	

Table 12: **Output port**

port name	data type	slots	description
events	*EventData*	1	A stream of 'ripple' events.
statistics	*MultiChannelData* <double>	1	

Table 13: **Options**

port name	data type	default	description
threshold dev	double	6.0	threshold that needs to be crossed
smooth time	double	10.0 seconds	integration time for signal statistics. Must be a positive number.
detection lockout time	double	30 ms	refractory period after threshold crossing detection that is not considered for updating of statistics and for detecting events. Must greater than 0 ms.
statistics buffer size	double	0.5 sec	Buffer size (in seconds) for statistics output buffers. Should be equal larger than zero.
statistics downsample factor	unsigned int	1	downsample factor of streamed statistics signal. Should larger than zero..
stream events	bool	True	enable/disable ripple event output
stream statistics	bool	True	enable/disable streaming of ripple detection statistics
use power	bool	True	

Table 14: **Producer states**

name	data type	initial value	external access	peers access	description
threshold	double	0.0	None	read-only	Current threshold that needs to be crossed
mean	double	0.0	None	read-only	Current signal mean. Units: same as input signal.
deviation	double	0.0	None	read-only	Current signal deviation. Units: same as input signal.

Table 15: **Broadcaster states**

name	data type	initial value	external access	peers access	description
ripple	bool	False	read-only	read-only	

Table 16: **Static states**

name	data type	initial value	external access	peers access	description
threshold dev	double	option:threshold dev	read-only	read/write	
smooth time	double	option:smooth time	read-only	read/write	integration time for signal statistics. Must be a positive number.
detection lockout time	double	option: detection lockout time	read-only	read/write	Current refractory period following threshold crossing that is not considered for updating signal statistics and for event detection.
stream events	bool	option:stream events	read-only	read/write	
stream statistics	bool	option:stream statistics	read-only	read/write	

SpikeDetector

Detect spikes on any of the incoming MultiChannelData stream; sends SpikeData on the output port spikes and an event "spike"/"spikes" on the output port events if one or more spikes have been detected in the received buffer

Table 17: **Input port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1	

Table 18: **Output port**

port name	data type	slots	description
events	*EventData*	1	A stream of events.
spikes	*SpikeData*	1	A stream of detected spikes

Table 19: **Options**

port name	data type	default	description
threshold	double	60.0	threshold that a single channel must cross.
invert signal	bool	True	whether the signal does (true) or does not (false) need to be inverted when detecting spikes
buffer size	double	0.5 ms	amount of data that will be used to look for spikes [ms]
peak lifetime	unsigned int	8 samples	number of samples that will be used to look for a peak
strict time bin check	bool	True	

Table 20: **Static states**

name	data type	initial value	external access	peers access	description
threshold	double	option: threshold	read/write	read-only	
peak lifetime	unsigned int	option:peak lifetime	read/write	read-only	

9.3.2 signal generators, filters, converters

MultiChannelFilter

Table 21: **Input ports**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 22: **Output ports**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 23: **Options**

name	data type	default	description
filter	string OR definition structure	No default value - the definition of this parameter in the graph file is mandatory.	YAML filter definition or name of file that contains filter

Here are some example filter configurations:

```
filter:
    file: filters://butter_lpf_0.1fs.filter

filter:
   type: biquad
   gain: 3.405376527201278e-04
   coefficients:
      - [1.0, 2.0, 1.0, 1.0, -1.032069405319709, 0.275707942472944]
```

(continues on next page)

(continued from previous page)

```
      - [1.0, 2.0, 1.0, 1.0, -1.142980502539903, 0.412801598096190]
      - [1.0, 2.0, 1.0, 1.0, -1.404384890471583, 0.735915191196473]
   description: 6th order butterworth low pass filter with cutoff at 0.1 times the␣
↪sampling frequency

filter:
   type: fir
   description: 101 taps low pass filter with cutoff at 0.1 times the sampling␣
↪frequency
   coefficients: [-6.24626469088e-19, -0.000309386982441, -0.000528204854007, ...]
```

NIxReader

Read raw data from a Neuralynx Digilynx data acquisition system and turns it into multiple MultiChannelData output streams based on a channel mapping

Table 24: **Output ports**

port name	data type	slots	description
(configurable) ports name from channelmap options	*MultiChannelData* <double>	1	create an output port for each channel in the channelmap

The channelmap defines the output port names and for each port lists the AD channels that will be copied to the MultiChannelData buckets on that port. The channelmap option should be specified as follows:

```
channelmap:
  portnameA: [0,1,2,3,4]
  portnameB: [5,6]
  portnameC: [0,5]
```

Table 25: **Options**

name	data type	default	description
address	string	"127.0.0.1"	IP address of Digilynx system
port	unsigned int	5000	port of Digilynx system
channelmap	map of vector<int>	No default value	mapping between AD channels and output ports
npackets	uint64_t	0	number of raw data packets to read before exiting (0 = continuous streaming)
nchannels	unsigned int	128	number of channels in Digilynx system
batch size	unsigned int	1	The number of data packets to concatenate into single multi-channel data bucket.
update interval	unsigned int	20	time interval (in seconds) between log updates
hardware trigger	bool	False	enable use of hardware triggered dispatching
hardware trigger channel	uint8	0	Digital input channel to use as hardware trigger

Rebuffer

Rebuffer and downsample multiple MultiChannelData streams. No anti-aliasing filter is applied before downsampling.

Table 26: **Input port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 27: **Output port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 28: **Options**

port name	data type	default	description
down-sample factor	un-signed int	1	
buffer size	un-signed int	10 samples or equivalent in second based on the downsam-ple factor depending of the buffer unit.	Output buffer size in samples or seconds.

RunningStats

Compute running statistics

Table 29: **Input port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 30: **Output port**

port name	data type	slots	description
data	*MultiChannelData* <double>	1-256	

Table 31: **Options**

port name	data type	default	description
integration time	double	1.0	time window for exponential smoothing
outlier/protection	bool	False	enable outlier protection. Outliers are values larger than a predefined z-score. The contribution of an outlier is reduced by an amount that depends on the magnitude of the outlier
outlier/zscore	double	6.0	z-score that defines an outlier
outlier/half life	double	2.0	the number of standard deviations above the outlier z-score at which the influence of the outlier is halved.

9.3.3 Event

EventFilter

Process neural data

Table 32: **Input port**

port name	data type	slots	description
events	*EventData*	1-256	
blocking events	*EventData*	1-256	

Table 33: **Output port**

port name	data type	slots	description
events	*EventData*	1	

Table 34: **Options**

port name	data type	default	description
target event	string	None	target event to be filtered out
block duration	double	10.0 ms	time during which target events are filtered out
sync time	double	1.5 ms	time used to check if any blocking target event is present after a target event has been received
block wait time	double	3.5 ms	time after blocking event during all arriving event are blocked
discard warnings	bool	false	if true, warnings about discarded events will not be generated
detection criterion	string or unsigned int	"any"	string or number to determine the criterion for a triggering detection; acceptable string values: 'any', 'all' acceptable integer values: any value between 1 (equivalent to 'any') and the number of input slots

EventLogger

Synchronize on the occurrence of a target event on all its input slots, before emitting the same target event

Table 35: **Input ports**

port name	data type	slots	description
events	*EventData*	1-256	

Options

Table 36: **Options**

port name	data type	default	description
target event	string	none (default value set in *EventData*)	

EventSource

Generate an EventData stream by randomly emitting events from a list of candidates at a predefined rate

Table 37: **Output ports**

port name	data type	slots	description
events	*EventData*	1	

Table 38: **Options**

port name	data type	default	description
events	list of string	"default_eventsource_event"	list of events to emit
rate	double	1Hz	(approximate) event rate

EventSource

synchronizes on the occurrence of a target event on all its input slots, before emitting the same target event

Table 39: **Input ports**

port name	data type	slots	description
events	*EventData*	1-256	

Table 40: **Output ports**

port name	data type	slots	description
events	*EventData*	1	

Table 41: **Options**

port name	data type	default	description
target event	list of string	None	The event to synchronize on.

MUAEstimator

Compute the Multi-Unit Activity from the spike counts provided by the spike detectors and outputs MUAData.

Table 42: **Input port**

port name	data type	slots	description
spikes	*SpikeData*	1-64	

Table 43: **Output port**

port name	data type	slots	description
mua	*MUAData*	1	

Table 44: **Options**

port name	data type	default	description
bin size	unsigned int	20 ms	

Table 45: **Static states**

name	data type	initial value	external access	peers access	description
bin size	unsigned int	options:bin size	read-only	read/write	

Table 46: **Broadcaster states**

name	data type	initial value	external access	description
MUA	double	0.0	read-only	

9.3.4 DigitalOutput

Take an EventData stream and sets digital outputs according to an event-specific protocol

Table 47: **Input ports**

port name	data type	slots	description
events	*EventData*	1	

Table 48: **Options**

name	data type	default	description
enabled	bool	enabled state	enable/disable digital output
lockout period	unsigned int	300 ms	set the maximal stimulation frequency. If equal to zero, it disabled the feature.
pulse width	unsigned int	300 ms	duration of digital output pulse in microseconds
enable saving	bool	enabled state	enable/disable saving stimulation events
device	unsigned int	no default type value - Always specify in the graph config file.	map specifying the digital output device. A required "type" key indicates which device should be used. Valid values are "dummy". The dummy device requires an additional "nchannels" key.
print protocol execution updates			maps events to digital output protocols.

The protocols option specifies a map with for each target event a map of actions for selected digital output channels. Note that each channel can only be associated with a single action (even if it is listed more than once). There are 4 possible actions: high, low, toggle and pulse. Events that are not in the protocols map are ignored. Example configuration for protocols option:

```
protocols:
  event_a:
    high: [0,1]
  event_b:
    low: [0]
    toggle: [1]
  event_c:
    pulse: [2]
```

Table 49: **Static states**

name	data type	initial value	external access	peers access	description
enabled	bool	option: enabled	write/read	read-only	
lockout period	bool	option: lockout period	write/read	read-only	

9.3.5 Serializers

FileSerializer

Serializes data streams to file

Table 50: **Input ports**

port name	data type	slots	description
data	IData	1-256	

Table 51: **Options**

port name	data type	default	description
path	string	"run://"	server-side path
encoding	string	"binary"	Only two acceptable keyword: 'binary' or 'yaml'
format	string	"full"	Only tree acceptable keyword: 'full', 'nodata', 'compact' (see serializer.hpp for more informations on this mode)
overwrite	bool	False	overwrite existing file
throttle/enabled	bool	False	throttle saving if we can't keep up
throttle/threshold	double	0.3	upstream ringbuffer fill fraction (between 0-1) at which throttling takes effect
throttle/smooth	double	0.5	smooth level of throttle level (between 0-1)

ZMQSerializer

Serializes data streams to cloud

Table 52: **Input ports**

port name	data type	slots	description
data	IData	1-256	

Table 53: **Options**

port name	data type	default	description
port	unsigned int	7777	
encoding	string	"binary"	Only two acceptable keyword: 'binary' or 'yaml'
format	string	"full"	Only tree acceptable keyword: 'full', 'nodata', 'compact' (see serializer.hpp for more informations on this mode)
interleave	bool	false	whether data streams from different input slots are interleaved

9.3.6 Example

DummySink

Take any data stream and eats it. Mainly used to show and test basic graph processing functionality.

Table 54: **Input port**

port name	data type	slots	description
data	IData	1	

Table 55: **States**

name	data type	initial value	external access	peers access	description
tickle	bool	False	read-only	read/write	trigger a log message when changed : True = "Hi hi, that tickles!" False = "Why stop tickling?

9.4 Connecting to hardware

9.4.1 Neuralynx

9.4.2 Open Ephys

9.4.3 Arduino

9.4.4 Advantech DIO

9.5 Libraries

9.5.1 Digital IO library

9.5.2 Digital Signal Processing (DSP) library

The DSP library included with Falcon contains a number of modular and reusable signal processing algorithms that can be used inside processor nodes.

Detect threshold crossing

Given the value of a single data sample, an instance of the *ThresholdCrosser* class detects if a threshold has been crossed. It does so by keeping track of the previous sample value and testing if the current and previous data samples are on opposite sides of the threshold. For example, for an upward threshold crossing, a crossing is detected if the previous sample is smaller than or equal to the threshold and the current sample is larger than the threshold.

Operation of the ThresholdCrosser is determined by two parameters: the value of the threshold and the direction of the crossing. These parameters are set in the constructor or using member functions (*set_threshold* and *set_slope*). Threshold crossing are detected by repeatedly calling the *has_crossed* member function with the next data sample.

Here is an example of how to use the ThresholdCrosser class:

```
// create instance with threshold = 10. and slope = UP
auto t = dsp::algorithms::ThresholdCrosser( 10., Slope::UP );

for (double k=0.5; k<20; ++k) {
    if (t.has_crossed(k)) {
        std::cout << "Threshold crossed!";
    }
}
```

Compute running statistics of signal

The *RunningStatistics* class supports the computation of running center and dispersion measures of a signal. It is a virtual base class with currently a single concrete subclass *RunningMeanMAD* that overrides the virtual method *update_statistics* to compute the running mean and mean absolute deviation.

The weight of the most recent data sample is set by the *alpha* parameter, which take a value between 0 and 1. The *update_statistics* method takes a data sample and alpha value and updates the internal estimates of the center and dispersion measures. For example, the *RunningMeanMad* class computes the running mean as $(1 - alpha) \times mean + alpha \times sample$.

Optionally, a burn-in period can be set during which the alpha value gradually changes from 1. to the preset alpha value. This has the effect that only weight is given to the data samples and not the initial values of the center and dispersion measures.

Also optionally, data samples that are more than a certain distance (in multiples of the z-score) from the center (i.e. outliers) have reduced influence on the computation of the running statistics.

Here is an example of how to use the *RunningMeanMAD* class:

```
std::vector<double> samples{0.1, 2.0, 1.5, 3.2, 1.3, 2.4};

// set alpha parameter to 0.1, no burn-in or outlier detection
r = dsp::algorithms::RunningMeanMAD(0.1);
r.add_samples(samples);

std::cout << "running mean = " << r.mean() << " and running MAD = " << r.mad();
```

Detect local peaks

The *PeakDetector* class detects local peaks in a signal, gives access to the timestamp and amplitude of the last detected peak and keeps track of the total number of detected peaks.

Exponentially smooth signal

The *ExponentialSmoother* class smooths a signal sample by sample. The integration window for smoothing is determined by the *alpha* parameter that sets the weight of the new data sample, i.e. $value = value * (1 - alpha) + sample * alpha$. Here is an example:

```
double smooth_sample;

// create smoother with alpha = 0.1
auto s = dsp::algorithms::ExponentialSmoother(0.1);

std::vector<double> samples{0.1, 2.0, 1.5, 3.2, 1.3, 2.4};

for (auto k : samples) {
    smooth_sample = s.smooth(k);
}
```

Detect spikes

Filtering

Finite impulse response (FIR) filters

Infinite impulse response (IIR) filters

9.5.3 Neuralynx library

9.6 Tools

9.6.1 Digital filter test

The filter test tool can be used to test the digital filtering library that is included with falcon. Given the definition of a FIR or IIR filter, the tool provides three functions:

1. to filter a signal
2. to time the execution of filtering
3. to compute the filter's impulse reponse

Command-line options

```
usage: ./filtertest [options] ... filter_definition_file
options:
  -s, --signal               signal to filter (string [=])
  -o, --signal_output        output file for filtered signal (string [=filtered_
↪signal.dat])
  -t, --timing               perform timing of filtering operation
  -n, --n_timing_points      number of points for timing (unsigned long [=1000000])
  -c, --n_timing_channels    number of channels for timing (unsigned int [=1])
  -i, --impulse              compute impulse response of filter
  -f, --impulse_output       output file for impulse response (string [=impulse_
↪response.dat])
  -p, --n_impulse_points     number of points for impulse response (0 means number of␣
↪samples is chosen automatically) (unsigned int [=0])
  -?, --help                 print this message
```

Examples

Filter a signal and save the result in *output.dat*:

```
filtertest -s signal.dat -o output.dat /path/to/filter/file
```

Perform timing of the filtering operation using the specified number of time points and channels:

```
filtertest -t -n 1000000 -c 10 /path/to/filter/file
```

Compute impulse response and save result:

```
filtertest -i -f output.dat /path/to/filter/file
```

9.6.2 Neuralynx testbench

The neuralynx testbench is a tool to generate a network stream of raw Neuralynx Digilynx data packets, which can be read by the NlxReader processor. It enables testing of falcon processing graphs without the need to be connected to a live Neuralynx acquisition system.

The testbench comes with a number of built-in signal sources, including a wave generator (sine, square and white noise) and signals read from file. Sources are defined in a configuration file and can be selected using keyboard commands.

Configuration

The testbench tool needs to be configured before use to set the IP address and network port for streaming data packets. By default, the program will look for a configuration file in the $HOME/.nlxtestbench folder and if this file does not exist, a new one will be created with default values. A configuration file can also be specified on the command line (see below).

The configuration file is written in YAML format and generally has three sections (network, stream and sources) with configurable options. An example configuration file is shown below:

```
network:
  ip: 127.0.0.1
  port: 5000
stream:
  rate: 32000
  npackets: 0
  autostart: ""
sources:
  - class: nlx
    file: /path/to/raw/data/file
    cycle: false
  - class: sine
    offset: 0.
    amplitude: 1.
    frequency: 1.
    sampling_rate: 32000
    noise_stdev: 0
  - class: square
    offset: 0.
    amplitude: 1.
    frequency: 1.
    duty_cycle: 0.5
    sampling_rate: 32000
    noise_stdev: 0
  - class: noise
    mean: 0.
    stdev: 1.
    sampling_rate: 32000
```

network options

The *ip* and *port* options specify the destination of the streamed data packets. By default, the IP address is 127.0.0.1 (i.e. local host) and port is 5000.

stream options

The neuralynx testbench tool produces a stream of data packets that are identical to the packets generated by the Digilynx system. Each data packet contains a single data sample for a number of input channels. At the moment, the testbench only supports a fixed number of 128 channels.

The *rate* option defines the desired rate at which data packets should be produced (in data packets per second). Note that data packets will be produced as close as possible to the desired rate, but the actual rate may be lower if the system cannot keep up.

The *npackets* option defines how many data packets should be produced. A value of 0 means that all data packets in the signal source should be streamed out (which in many cases is an infinite data stream).

The *autostart* option sets the signal source that should be streamed immediately when the testbench is started (i.e. without waiting for a keyboard command by the user to select the signal source).

sources

The final section of the configuration file lists predefined signal sources. If no sources are specified, then by default a single sine wave is added.

At present, four different signal sources are available: Neuralynx raw data file (*nlx*), sine wave (*sine*), square wave (*square*) and white noise (*noise*). To configure a signal source, you need to specify the class of the source (i.e. nlx, sine, square or noise) and any additional options.

To define a signal source that reads raw data packets from a previously recorded Neuralynx raw data file, you would use the *nlx* source class and set the *file* option to the path where the raw data file can be found. Importantly, the raw data file should contain recorded signals from exactly 128 channels. This is a limitation of the nlxtestbench tool that will hopefully be removed in the future. All data packets in the file will be streamed, unless a maximum number of data packets has been specified that is less than the number of available data in the file (see stream options). If the *cycle* options is set to true, then streaming of the data in the file will restart automatically once the end of the file was reached.

To define a source that generates a periodic signal, you would use the *sine* and *square* source class. You then specify the *offset* (in microVolt), *amplitude* (in microVolt) and *frequency* (in Hz) of the sine/square wave. In addition, you can specify a sampling rate other than 32 kHz using the *sampling_rate* option. And finally, noise may be added to the signal by setting the *noise_stdev* option to the standard deviation of a Gaussian noise distribution (in microVolt).

To define a source that generates white noise, you would use the *noise* class. You can specify both the *mean* and standard deviation (*stdev*) of the Gaussian noise distribution (in microVolt). You can specify a sampling rate other than 32 kHz using the *sampling_rate* option.

Command-line options

```
usage: ./nlxtestbench [options] ...
options:
  -c, --config       configuration file (string [=$HOME/.nlxtestbench/config.yaml])
  -a, --autostart    source to auto start streaming (int [=-1])
  -r, --rate         data stream rate (Hz) (double [=-1])
  -n, --npackets     maximum number of packets to stream (0 means all packets) (long␣
↪[=-1])
  -?, --help         print this message
```

Keyboard commands

After starting the testbench, the following keys are available:

key	action
<space>	list all defined sources
a-z	select signal to stream
<ESC>	quit

Examples

```
nlxtestbench -c /path/to/config/file -a 1
```

9.7 Example

9.8 Ressources

CHAPTER

TEN

REAL-TIME DECODING EXTENSION (SEPARATE REPOSITORY)

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search