mimoCoRB

Release 1.0.0beta

C. Mayer, K. Heitlinger, G. Quast

Feb 16, 2023

CONTENTS:

1 mimoCoRB - multiple-in multile-out Configurable Ring Buffer: Overview **1**

1.1 Detailed description of components 1

2 Access Classes in module *buffer_control* **5**

2.1 Application example 5

3 Module Documentation **9**

Python Module Index **19**

Index **21**

CHAPTER

ONE

MIMOCORB - MULTIPLE-IN MULTILE-OUT CONFIGURABLE RING BUFFER: OVERVIEW

mimoCoRB: multiple-in multiple-out Configurable Ring Buffer

The package **mimoCoRB** provides a central component of each data acquisition system needed to record and pre-analyze data from randomly occurrig processes. Typical examples are waveform data as provided by single-photon counters or typical detectors common in quantum mechanical measurements or in nuclear, particle physics and astro particle physics, e. g. photo tubes, Geiger counters, avalanche photo-diodes or modern SiPMs.

The random nature of such processes and the need to keep read-out dead times low requires an input buffer and a buffer manager running as a background process. While a data source feeds data into the ringbuffer, consumer processes are fed with an almost constant stream of data to filter, reduce, analyze or simply visualize data and on-line analysis results. Such consumers may be obligatory ones, i. e. data acquisition pauses if all input buffers are full and an obligatory consumer is still busy processing. A second type of random consumers or "observers" receives an event copy from the buffer manager upon request, without pausing the data acquisition process. Typical examples of random consumers are displays of a subset of the wave forms or of intermediate analysis results.

This project originated from an attempt to structure and generalize data acquision for several experiments in advanced physics laboratory courses at Karlruhe Institute of Technology (KIT).

As a simple demonstration, we provide data from simulatd signals as would be recored by a detector for comsmic myons with three detection layers. Occasionally, such muons stop in an absorber between the 2nd and 3rd layer, where they decay at rest and emit a high-energetic electron recorded as a 2nd pulse in one or two of the detection layers. After data acquitision, a search for typical pulses is performed, data with detected double pulses are selected and fed into a second buffer. A third buffer receives data in a reduced format which only contains the parameters of found pulses. These data and the wave forms of all double-pulses are finally stored on disk. This application is a very typical example of the general process of on-line data processing in modern experiments and may serve as a starting point for own applications.

1.1 Detailed description of components

Ring buffer

Writer, Reader and Observer classes

User Access classes wrapping the mimoCoRB classes

Configuraion of DAQ with yaml files

Simple application example (also provided as a unittest)

An application example of *mimo_buffer* is shown below. This code may serve as a starting point for own projects. The set-up is as follows:

2 ring buffers are defined:

- input Buffer RB_1: 10 ch x 1024 slots (int32)
- output Buffer RB_2: 10 ch x 2 slots/ch (float64)

Simple data is filled into RB_1, copied and extended by a process writing data into RB_2, and finally a reader process to check integrity and completenss of thd data. The most complex part of the code is in function *run_control()*, which demonstrates how to set up the buffers, define Reader and Writer instances and start the prallel processes for generating, processing and reading the data.

The example including comment lines for explanation is shown here:

```
import time
import unittest
import numpy as np
from multiprocessing import Process, Value
from mimocorb import mimo_buffer as bm

# global variables
N_requested = 1000  # numer of data injectios ("events")
Time_tick = 0.001   # time between events
Ncpu1 = 2           # number of parallel abalyzer processes

def data_generator(sink_dict):
  """writes continuously rising integers to buffer specified in sink_dict
  """
  sink = bm.Writer(sink_dict)
  n=0
  # inject data
  for x in range(N_requested):
      buffer = sink.get_new_buffer() # get new buffer and pass last item
      #  random wait for next data item
      time.sleep(-Time_tick*np.log(np.random.rand() ))
      # fill "data"
      n += 1
      buffer[:] = n
  # process last data item
  sink.process_buffer()


def analyzer(source_dict, sink_dict):
  """read from source and write first element and a time difference to sink
  """
  source = bm.Reader(source_dict)
  sink = bm.Writer(sink_dict)
  start_time = time.time()

  while True:
      input_data = source.get()
      output_data = sink.get_new_buffer()
      # process data
      output_data[0] = input_data[0]
      # mimick processing time
      time.sleep(2*Time_tick)
      output_data[1] = time.time() - start_time
```

(continues on next page)

(continued from previous page)

```
    #
    sink.process_buffer()


def check_result(source_dict, res):
  """reads RB_2 and sum up the integer content (value should be sum(1 -35) = 630);

     sum is returned as shared memory Value-object
  """
  source = bm.Reader(source_dict)
  sum_rb = 0
  while True:
      input_data = source.get()
      res.value +=int(input_data[0])

def run_control():
  """Setup buffers, start processes and shut_down when 1st writer done
  """

  # Create ring buffers: #2: 10 channel, 2 value per channel
  #    (1: buffer content; 2: time difference as int)
  #    d_type = [('chA', np.float)]  #not necessary: always the same type
  generator_buffer = bm.NewBuffer(10, 1, np.int32)
  eval_buffer = bm.NewBuffer(10, 2, np.float32)

  # create readers first
  source_dic_gen = generator_buffer.new_reader_group()
  source_dic_eval = eval_buffer.new_reader_group()

  # Create worker processes (correct sequence: first action as last)
  process_list = []
  #  evaluation to test ring buffer behavior
  result = Value('i', 0)   # int variable in shared meomry
  process_list.append(Process(target=check_result,
                              args=(source_dic_eval, result)))
  # data transfer between the 2 buffers: generator_buffer -> eval_buffer
  sink_dic_eval = eval_buffer.new_writer()
  # work with all cpu's requested
  number_of_workers = Ncpu1
  for i in range(number_of_workers):
      process_list.append(Process(target=analyzer,
                                  args=(source_dic_gen, sink_dic_eval)))

  # fill buffer (generator_buffer) with data first
  sink_dic_gen = generator_buffer.new_writer()
  process_list.append(Process(target=data_generator,
                              args=(sink_dic_gen,)))

  for p in process_list:
      p.start()
```

(continues on next page)

(continued from previous page)

```
  run_active = True
  while run_active:
     run_active = False if process_list[-1].exitcode==0 else True
     time.sleep(0.1)  # wait

  time.sleep(0.1)  # some grace-time for readers to finish

  generator_buffer.shutdown()
  eval_buffer.shutdown()
  del generator_buffer, eval_buffer

  for p in process_list:
      p.join()

  return result.value


class RPTest(unittest.TestCase):

  def test_process(self):
      # start python test module and check result
      a = run_control()
      expected_result = N_requested*(N_requested+1)//2
      self.assertEqual(a, expected_result)  # expected result: sum(i); i = 1, N_requested


if __name__ == "__main__":
  unittest.main(verbosity=2)
#    print(process_buffer())
```

CHAPTER

TWO

ACCESS CLASSES IN MODULE *BUFFER_CONTROL*

To ease user interaction with the buffer manager, a set of additional classes is provided in the module *buffer_control* to set-up and manage cascades of ringbuffers and the asscociated sub-processes for filling, filtering and extracting data. These classes are of interest for developers wanting to help improving the package.

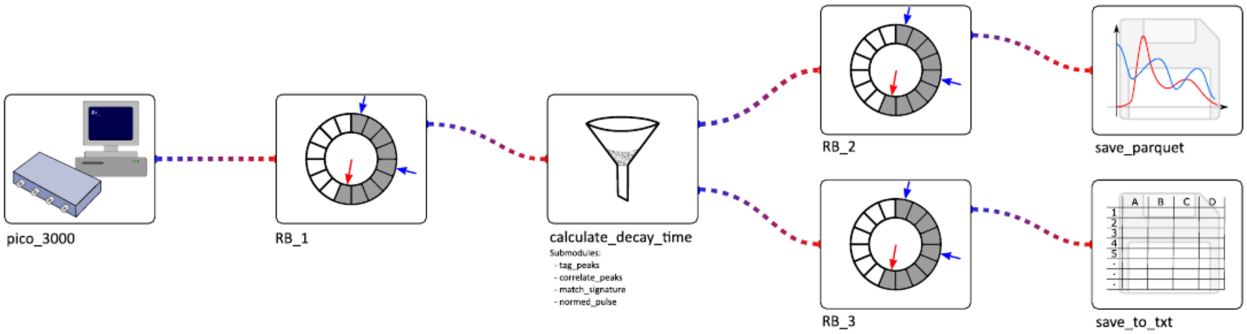
- ***class buffer_control***
  Set-up and management ringbuffers and associated sub-processes
- ***class SourceToBuffer***
  Read data from source (e.g. from file, simulation, PicoScope etc.) and put data in mimo_buffer
- ***class BufferToBuffer***
  Read data from input buffer, filter and write data to output buffer(s)
- ***class BufferToTxtfile*:**
  Save data to file in csv-format
- ***class run_mimoDAQ*:**
  Setup and run data acquisition with mimiCoRB buffer manager

These classes shield much of the complexity from the user, who can thus concentrate on writing the pieces of code need to acquire and prcess the data.

run_mimoDAQ contains most of the code needed to run a real example of a data-acquisition suite defined in a configuraion file with associated, user-defined functions for data provisioning, filtering and storage. It also provides an example on how to user the methods provided by the class *buffer_control*.

2.1 Application example

The subdirectory examples/ contains a rather comlete application use case. Code snippets and configuration data are provided in the subdirectories examples/modules/ and examples/config/, respectively. Waveform data, as provided by, for example, a multi-channel digital oscilloscope, are generated and filled into the first of a cascaded set of three ringbuffers. The raw data are analyzed, and accepted data with a double-pulse signature are selected and directly passed on to a second ringbuffer. A third buffer contains only the information about found signal pulses; a result file in *csv* format contains the data extracted from this buffer. A graphical representation of the set-up is schown in the figure below [source: Master's Thesis Christoph Meyer, ETP 2022]. Note that in the example the oscilloscope ist replaced by a signal simulation.



The buffer layout and the associated functions are defined in the main configuration file *simulsource_setup.py*, which serves as the input to the execution script *run_daq.py* in the top-level directory of the package. The *python* files *simulation_source.py*, *liftime_filter.py* and *save_files.py* contain the user code for data generation, analysis and filtering and extraction of the final data to disk files. The *.yaml* file *simulation_config.yaml* contains configurable parameters provided to these functions.

An observer process receives a sub-set of the data in the second buffer and shows them as an oscilloscope display on screen while data are generated and propagated through the buffers.

This example is executed form the directory examples/ by entering:

> *../run_daq.py simulsource_setup.yaml*

The code needed to run data-acquistion based on the package *mimocorb.buffer_control.run_mimoDAQ* is shown here:

```
# script run_daq.py

from mimocorb.buffer_control import run_mimoDAQ
daq = run_mimoDAQ()
daq.setup()
daq.run()
```

The input *yaml* file for the example provided as part of the package looks as follows:

```
#  Application example for mimoCoRB
#  --------------------------------

RingBuffer:
  # define ring buffers
  - RB_1:
    # raw input data buffer (waveforms from PicoScope, filele_source or simulation_
↪source)
    number_of_slots: 128
    channel_per_slot: 4250
    data_type:
        1: ['chA', "float64"]
        2: ['chB', "float64"]
        3: ['chC', "float64"]
        4: ['chD', "float64"]
  - RB_2:
      # buffer with accepted signatures (here double-pulses)
      number_of_slots: 128
      channel_per_slot: 4250
      data_type:
```

(continues on next page)

(continued from previous page)

```
        1: ['chA', "float64"]
        2: ['chB', "float64"]
        3: ['chC', "float64"]
        4: ['chD', "float64"]
  - RB_3:
      # buffer with pulse parameters (derived from waveforms)
      number_of_slots: 32
      channel_per_slot: 1
      data_type:
        1: ['decay_time', "int32"]
        3: ['1st_chA_h', "float64"]
        4: ['1st_chB_h', "float64"]
        5: ['1st_chC_h', "float64"]
        6: ['1st_chA_p', "int32"]
        7: ['1st_chB_p', "int32"]
        8: ['1st_chC_p', "int32"]
        9: ['1st_chA_int', "float64"]
        10: ['1st_chB_int', "float64"]
        11: ['1st_chC_int', "float64"]
        12: ['2nd_chA_h', "float64"]
        13: ['2nd_chB_h', "float64"]
        14: ['2nd_chC_h', "float64"]
        15: ['2nd_chA_p', "int32"]
        16: ['2nd_chB_p', "int32"]
        17: ['2nd_chC_p', "int32"]
        18: ['2nd_chA_int', "float64"]
        19: ['2nd_chB_int', "float64"]
        20: ['2nd_chC_int', "float64"]
        21: ['1st_chD_h', "float64"]
        22: ['1st_chD_p', "int32"]
        23: ['1st_chD_int', "float64"]
        24: ['2nd_chD_h', "float64"]
        25: ['2nd_chD_p', "int32"]
        26: ['2nd_chD_int', "float64"]

Functions:
  # define functions and assignments
  - Fkt_main:
     config_file: "config/simulation_config.yaml"
  - Fkt_1:
     file_name: "modules/simulation_source"
     kt_name: "simulation_source"
     num_process: 1
     RB_assign:
       RB_1: "write"
  - Fkt_2:
     file_name: "modules/lifetime_filter"
     fkt_name: "calculate_decay_time"
     num_process: 2
     RB_assign:
       RB_1: "read"     # input
       RB_2: "write"    # waveform to save (if double pulse was found)
```

(continues on next page)

(continued from previous page)

```
      RB_3: "write"    # pulse data
  - Fkt_3:
     file_name: "modules/save_files"
     fkt_name: "save_to_txt"
     num_process: 1
     RB_assign:
       RB_3: "read"     # pulse data
  - Fkt_4:
     file_name: "modules/save_files"
     fkt_name: "save_parquet"
     num_process: 1
     RB_assign:
       RB_2: "read"     # waveform to save
  - Fkt_5:
     file_name: "mimocorb/plot"
     fkt_name: "plot_graph"
     num_process: 1
     RB_assign:
       RB_2: "observe"  # double pulse waveform
```

2.1.1 Indices and tables

- genindex
- modindex
- search

CHAPTER

THREE

MODULE DOCUMENTATION

mimo-ringbuffer:

Module implementing a multiple-in multiple-out ringbuffer appropriate for multiprocessing.

The ringbuffer creation and management is handled by the `NewBuffer`-class. Access to the content is possible through the `Reader`, `Writer` and `Observer` classes.

classes:

- NewBuffer: create a new ringbuffer, assign writer(s) and reader(s) or observer(s)

    methods:

    - new_reader_group
- Writer
- Reader
- Observer

class `mimocorb.mimo_buffer.`**`NewBuffer`**(*number_of_slots*, *values_per_slot*, *dtype*, *debug=False*)

Class to create a new ringbuffer object according to the 'FIFO' principle (first-in first-out).

Memory shares, IPC queues, lock and event objects as well as background threads are defined for the multiprocessing ringbuffer management. Methods are provided to build the setup dictionaries (necessary parameter objects) for the `Reader`, `Writer` or `Observer` instances, respectively. Further, methods are provided to allow an index processing (e.g. listeners) and to pause data processing.

important methods:

- __init__() constructor to create a new 'FIFO' ringbuffer
- new_writer() create new writer
- new_reader_group() create reader group
- new_observer() create observer
- buffer_status() display status: event count, processing rate, occupied slots
- pause() disable writer(s) to ringbuffer
- resume() (re-)enable writers
- set_ending() stop data-taking (gives processes time to finish before shutdown)
- close() release shared memory
- shutdown() end connected processes, delete ringbuffer

`buffer_status()`

Processing Rate and approximate number of free slots in this ringbuffer. This method is meant for user information purposes only, as the result may not be completely accurate due to race conditions.

Returns
Number of free slots in this ringbuffer

Return type
int

`event_loop_executor`(*loop: AbstractEventLoop*) → None

Internal method continuously run in a background thread. It runs the asynchronous event loop needed for the websocket based IPC of `Observer`-instances.

`increment_reader_pointer()`

Internal method called by a `reader_queue_listener()`-thread after a new element was marked as 'processing is done'. It is checked whether all reader groups have completed processing the oldest ringbuffer element, and if so, adds it to the 'free ringbuffer elements' queue used by the `Writer`-instances. For this function to work properly and without race conditions self.heap_lock has to be acquired BEFORE entering the function (see `reader_queue_listener()`-method).

`new_observer()`

Method to create a new observer. It's possible to create multiple observers and simply share the setup dictionary between different `Observer`-instances (analogues to the behavior of the `new_reader_group`). It might be possible that data seen by an `Observer` instance are corrupted (especially with a not ideal ringbuffer configuration and/or a heavily loaded PC system).

``Observer``-instances MUST NOT rely on data integrity!!

Returns
The `setup_dict` object passed to an `Observer`-instance to grant access to this ringbuffer.

Return type
dict

`new_reader_group()`

Method to create a new reader group. The processing workload of a group can be distributed to multiple processes by using the same setup dictionary (`setup_dict`) defined for a `Reader`-object. Each ringbuffer element is processed by one reader group process. It's possible to create multiple reader groups per ringbuffer, where each reader group gets every element written to the ringbuffer. If a reader group is created, at least one `Reader`-class instance MUST steadily call its `get()` method to prevent the ringbuffer from blocking and to allow a safe shutdown.

Returns
The `setup_dict` object passed to a `Reader`-instance to grant read access to this ringbuffer.

Return type
dict

`new_writer()`

Method to create a new writer. It is possible to create multiple writers and simply share a setup dictionary definition between different `Writer`-instances (analogues to the behavior of the `new_reader_group`).

Returns
The `setup_dict` object passed to a `Writer`-instance to grant write access to this ringbuffer.

Return type
dict

`async observer_check_active_state()` → None

Internal asynchronous function to check if `NewBuffer.shutdown()` was called

`async observer_main()`

Internal asynchronous method run in the background to handle websocket connections. A websocket server is started on the loopback device, providing IPC between the main process and an `Observer`-instance running in another process.

`async observer_server(`*websocket*, *path*`)`

Internal asynchronous method implementing the `Observer` IPC. As of now: for every message, the current `write_pointer` is sent (index in the shared memory array containing the latest added element to the ringbuffer). **CAUTION!** The ringbuffer element *IS NOT LOCKED*, so it has to be copied as soon as possible in the `Observer`-process. For conventional signal analysis chains and PC setups, this should not be a constraint. But it might be possible that data seen by the `Observer` instance are corrupted (especially with a not ideal ringbuffer configuration and/or a heavily loaded PC system).

``Observer``-instances MUST NOT rely on data integrity!!

`pause()`

Disable writing to ringbuffer (paused)

`reader_queue_listener(`*done_queue*, *done_heap*`)`

Internal method run in a background thread (one for each reader group). It handles dispatching free ringbuffer elements.

Parameters

- **done_queue** – the multiprocessing.queue created in `new_reader_group()`
- **done_heap** – the heap created in `new_reader_group()`

`resume()`

(Re)enable writing to ringbuffer (resume)

`set_ending()`

Stop data flow (before shut-down)

`shutdown()`

Shut down the ringbuffer(s): close background threads, terminate associated processes and release the shared memory definitions.

Affect processes using a `Reader`, `Writer` or `Observer` instance to a ringbuffer.

A 'trickle down' approach is used to have as few ringbuffer elements as possible unprocessed. This may not work correctly with more complex signal analysis chains. So always make sure to shut down the ringbuffers in data flow order (start with first element of the chain, the ringbuffer closest to the signal source).

CAUTION! If there are loops in the signal analysis chain, this method may end in an infinite loop!

`writer_queue_listener()`

Internal method run in a background thread. It takes the index (a 'pointer' in the array) of the 'ready to process' ringbuffer element from the `writer_filled_queue` and distributes it to every reader group (reader_todo_queue).

`class mimocorb.mimo_buffer.Observer(`*setup_dict*`)`

Class for reading selected elements from a ringbuffer.

Ringbuffer elements are structured NumPy arrays. The returned array will not change until the next `Observer.get()`-call, the ringbuffer element is not blocked. The data transfer is implemented via web socket and interfaces with the ringbuffer manager (`NewBuffer`-class).

async **check_active_state**() → None

Internal asynchronous function to check if `NewBuffer.shutdown()` was called in the main process

async **establish_connection**() → None

Internal asynchronous function establishing the websocket connection with the ringbuffer manager in the main process (used for IPC)

event_loop_executor(*loop: AbstractEventLoop*) → None

Internal function executing the event loop for the websocket connection in a different thread.

get()

Get a copy of the latest element added to the ringbuffer by a `Writer` process.

Returns
One element (structured numpy.ndarray) from the ringbuffer as specified in the `NewBuffer()-dtype`-object, or None if run ended

Return type
numpy.ndarray

async **get_new_index**() → None

Internal asynchronous function to query the index of the latest ringbuffer element from the ringbuffer manager

class mimocorb.mimo_buffer.**Reader**(*setup_dict*)

Class to read elements from a ringbuffer (multiple-out part).

Ringbuffer elements are structured NumPy arrays and strictly **read-only**. The returned array won't change until the next `Reader.get()` call is performed, blocking the ringbuffer element for the time being. A program design processing the ringbuffer content has to call the `Reader.get()`-method in a way that minimizes the ringbuffer lock time.

methods:

- get()
- get_metadata():

data_available()

Method to check for new data and avoid blocking of consumers

get()

Get a new element from the ringbuffer. The last element obtained by calling this function is marked as "processing is done". No memory views of old elements may be accessed after calling this function (memory might change, be corrupted or be inconsistent). This function blocks if there are no new elements in the ringbuffer.

Raises
SystemExit – When the `shutdown()`-method of the `NewBuffer` object has been called, a SystemExit is raised which terminates the process.

Returns
One element (structured numpy.ndarray) of the ringbuffer as specified in the `NewBuffer()-dtype`-object.

Return type
numpy.ndarray

`get_metadata()`

Get the metadata defined for a ringbuffer element of the `Reader.get()`-method.

Returns

Currently a 3-tuple is returned with `(counter, timestamp , deadtime)` which is assigned to the latest element of the ringbuffer. The content of these variables is filled by the `Writer`-process. The current convention is:

- counter (int): a unique, 0 based, consecutive integer referencing this element
- timestamp (float): the UTC timestamp
- **deadtime (float): In a live-data environment, the dead time of the first** writer in the analyses chain. This is meant to be the fraction of dead time to active data capturing time (so 0.0 = no dead time whatsoever; 0.99 = only 1% of the time between this and the last element was spent with active data capturing)

Return type

tuple

`class mimocorb.mimo_buffer.Writer(`*setup_dict*`)`

Class to write elements into a ringbuffer (multiple-in part).

Ringbuffer elements are structured NumPy arrays. Writing is triggered by a call of `Writer.process_buffer()` or at the next call of `Writer.get_new_buffer()`. The ringbuffer element is blocked while writes to the NumPy array are permitted. A program design processing the ringbuffer content has to call the `Writer.process_buffer()` or `Writer.get_new_buffer()`-methods in a way that minimizes the ringbuffer lock time.

methods:

- get_new_buffer()
- set_metadata()
- process_buffer()

`get_new_buffer()`

Get a new free element in the ringbuffer.

The last element obtained by calling this function is marked as "ready to be processed". No memory views of old elements may be accessed after calling this function. This function blocks if there are no free elements in the ringbuffer and always returns a valid NumPy array that can be written to.

Raises

`SystemExit` – When the `shutdown()`-method of the `NewBuffer` object has been called, a SystemExit is raised which terminates the process.

Returns

One free ringbuffer element (structured numpy.ndarray) as specified in the `NewBuffer()-dtype`-object. Free elements may contain older data, but they can be safely overwritten.

Return type

numpy.ndarray

`process_buffer()`

Mark the current ringbuffer element as "ready to be processed".

The content of the array MUST NOT be changed after calling this function. If there is no current element, nothing happens. As the ringbuffer element is blocked while writing to the NumPy array it is recommended to call `Writer.process_buffer()` as soon as possible to minimize the ringbuffer lock time.

`set_metadata`(*counter*, *timestamp*, *deadtime*)

Set the metadata defined for the current ringbuffer element. If there is no current ringbuffer element (e.g. because `process_buffer()` has been called or `get_new_buffer()` has not been called yet), nothing happens. Copying metadata from a `Reader` to a `Writer` object (here called `source` and `sink`) can be done with:

```
sink.set_metadata(*source.get_metadata())
```

Parameters

- **counter** (*integer (np.longlong)*) – a unique, 0 based, consecutive integer referencing this element
- **timestamp** (*float (np.float64)*) – the UTC timestamp
- **deadtime** (*float (np.float64)*) – In a live-data environment, the dead time of the first writer in the analyses chain. This is meant to be the fraction of dead time to active data capturing time (so 0.0 = no dead time whatsoever; 0.99 = only 1% of the time between this and the last element was spent with active data capturing)

Collection of classes to set-up, manage and access ringbuffers and associated access funtions

class mimocorb.buffer_control.**BufferData**(*source_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Read data from buffer and send to requesting client (via python yield())

class mimocorb.buffer_control.**BufferToBuffer**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, *ufunc=None*, ***rb_info*)

Read data from input buffer, filter data and write to output buffer(s)

Args:

- buffer configurations (only one source and severals sinks, no observers!)
- function ufunc() must return
  - None if data to be rejected,
  - int if only raw data to be copied to sink[0]
  - list of parameterized data to be copied to sinks[]

Action:

store accepted data in buffers

class mimocorb.buffer_control.**BufferToParquetfile**(*source_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Save data a set of parquet-files packed as a tar archive

class mimocorb.buffer_control.**BufferToTxtfile**(*source_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Save data to file in csv-format

class mimocorb.buffer_control.**ObserverData**(*observe_list=None*, *config_dict=None*, ***rb_info*)

Deliver data from buffer to an observer process

class mimocorb.buffer_control.**SourceToBuffer**(*sink_list=None*, *observe_list=None*, *config_dict=None*, *ufunc=None*, ***rb_info*)

Read data from source (e.g. file, simulation, Picoscope etc.) and put data in mimo_buffer.

class `mimocorb.buffer_control.`**`buffer_control`**(*buffers_dict*, *functions_dict*, *output_directory*)

Set-up and management ringbuffers and associated sub-processes

Class methods:

- setup_buffers()
- setup_workers()
- start_workers()
- pause()
- resume()
- shutdown()

static get_config(*config_file*)

Args:
config_file: defined in main_setup file (yaml) with fixed name key config_file

Returns: yaml configuration file content (dict)

static import_function(*module_path*, *function_name*)

Import a named object defined in a config yaml file from a module.

Parameters:
module_path (str): name of the python module containing the function/class function_name (str): python function/class name

Returns:
(obj): function/method name callable as object

Raises:
ImportError: returns None

pause()

Pause data acquisition

resume()

re-enable data acquisition

set_ending()

stop writing and reading data, allow processes to finish

setup_workers()

Set up all the (parallel) worker functions

shutdown()

Delete buffers, stop processes by calling the shutdown()-Method of the buffer manager

start_workers()

start all of the (parallel) worker functions

class `mimocorb.buffer_control.`**`run_mimoDAQ`**(*verbose=2*)

Setup and run Data Aquisition with mimiCoRB buffer manager

Functions:

- setup
- run

- stop

keyboard_input(*cmd_queue*)

Read keyboard input, run as background-thread to avoid blocking

class tc

define terminal color codes

Collection of classes with graphics functions to plot buffer data

class mimocorb.plot_buffer.**WaveformPlotter**(*conf_dict=None*, *dtypes=None*, *fig=None*)

Oscilloscope-like display of wave from buffer data

The __call__ method of this class updates only the time-depenent input data and redraws the figure.

init()

plot initial line objects to be animated

class mimocorb.plot_buffer.**plotWaveformBuffer**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Plot data using an mimiCoRB Observer

rb_unittest: application example for mimo_buffer

This code may serve as a very basic starting point for own projects

Set-up: 2 ring buffers are defined:

- input Buffer RB_1: 10 ch x 1024 slots (int32)
- output Buffer RB_2: 10 ch x 2 slots/ch (float64)

simulation_source: Generate simulated wave form data

simulation_source.**simulation_source**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Generate simulated data and pass data to buffer The class mimocorb.buffer_control/SourceToBuffer is used to interface to the newBuffer and Writer classes of the package mimoCoRB.mimo_buffer

Parameters

config_dict – configuration dictionary

- events_required: number of events to be simulated or 0 for infinite
- sleeptime: (mean) time between events
- random: random time between events according to a Poission process
- number_of_samples, sample_time_ns, pretrigger_samples and analogue_offset describe the waveform data to be generated (as for oscilloscope setup)

Internal parameters of the simulated physics process (the decay of a muon) are (presently) not exposed to user.

Module **lifetime_filter**

This (rather complex) module filters waveform data to search for valid signal pulses in the channel data. The goal is to clearly identify coincidences of signals in different layers (indiating the passage of a cosmic ray particle, a muon) and find double-pulse signatures that a muon was stopped in or near a detection layer where the resulting decay-electron produced a delayed pulse. The time difference between the initial and the delayed pulses is the individual lifetime of the muon.

Wave forms passing this filter-criterion an passed on to a new buffer; the decay time and the properties of the signal pulses (height, integral and postition in time) are written to another buffer.

The relevant configuration parameters can be found in the section *calculate_decay_time:* of the configuration file.

`lifetime_filter.`**`calculate_decay_time`**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Calculate decay time as time between double pulses

Input:
pulse wave forms

Returns:
None if failed, int or list of pulse parameters if successful

Note: output produced when filter is passed depends on number of defined sinks:

- one sink: input data
- two sinks: input data and double-pulse parameters
- three sinks: input data and double-pulse parameters separately for upwards and for downwards going decay electrons

plot: plotting waveforms from buffer using mimoCoRB.buffer_control.OberserverData

`plot_waveform.`**`plot_graph`**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Plot waveform data from mimiCoRB buffer

Parameters
`input` – configuration dictionary

- plot_title: graphics title to be shown on graph
- min_sleeptime: time between updates
- sample_time_ns, channel_range, pretrigger_samples and analogue_offset describe the waveform data as for oscilloscope setup

read_from_buffer: example of a module reading and analyzing data from a buffer

Since reading blocks when no new data is available, a 2nd thread is started to collect data at the end

`read_from_buffer.`**`read_from_buffer`**(*source_list=None*, *sink_list=None*, *observe_list=None*, *config_dict=None*, ***rb_info*)

Read data from mimiCoRB buffer using the interface class mimo_control.BufferData

Parameters
`input` – configuration dictionary

PYTHON MODULE INDEX

l

lifetime_filter, 16

m

mimocorb, 9
mimocorb.buffer_control, 14
mimocorb.mimo_buffer, 9
mimocorb.plot_buffer, 16

p

plot_waveform, 17

r

rb_unittest, 16
read_from_buffer, 17

s

simulation_source, 16

INDEX

B

buffer_control (*class in mimocorb.buffer_control*), 14
buffer_status() (*mimocorb.mimo_buffer.NewBuffer method*), 9
BufferData (*class in mimocorb.buffer_control*), 14
BufferToBuffer (*class in mimocorb.buffer_control*), 14
BufferToParquetfile (*class in mimocorb.buffer_control*), 14
BufferToTxtfile (*class in mimocorb.buffer_control*), 14

C

calculate_decay_time() (*in module lifetime_filter*), 16
check_active_state() (*mimocorb.mimo_buffer.Observer method*), 11

D

data_available() (*mimocorb.mimo_buffer.Reader method*), 12

E

establish_connection() (*mimocorb.mimo_buffer.Observer method*), 12
event_loop_executor() (*mimocorb.mimo_buffer.NewBuffer method*), 10
event_loop_executor() (*mimocorb.mimo_buffer.Observer method*), 12

G

get() (*mimocorb.mimo_buffer.Observer method*), 12
get() (*mimocorb.mimo_buffer.Reader method*), 12
get_config() (*mimocorb.buffer_control.buffer_control static method*), 15
get_metadata() (*mimocorb.mimo_buffer.Reader method*), 12
get_new_buffer() (*mimocorb.mimo_buffer.Writer method*), 13
get_new_index() (*mimocorb.mimo_buffer.Observer method*), 12

I

import_function() (*mimocorb.buffer_control.buffer_control static method*), 15
increment_reader_pointer() (*mimocorb.mimo_buffer.NewBuffer method*), 10
init() (*mimocorb.plot_buffer.WaveformPlotter method*), 16

K

keyboard_input() (*mimocorb.buffer_control.run_mimoDAQ method*), 16

L

lifetime_filter
    module, 16

M

mimocorb
    module, 9
mimocorb.buffer_control
    module, 14
mimocorb.mimo_buffer
    module, 9
mimocorb.plot_buffer
    module, 16
module
    lifetime_filter, 16
    mimocorb, 9
    mimocorb.buffer_control, 14
    mimocorb.mimo_buffer, 9
    mimocorb.plot_buffer, 16
    plot_waveform, 17
    rb_unittest, 16
    read_from_buffer, 17
    simulation_source, 16

N

new_observer() (*mimocorb.mimo_buffer.NewBuffer method*), 10
new_reader_group() (*mimocorb.mimo_buffer.NewBuffer method*), 10
new_writer() (*mimocorb.mimo_buffer.NewBuffer method*), 10
NewBuffer (*class in mimocorb.mimo_buffer*), 9

O

Observer (*class in mimocorb.mimo_buffer*), 11
observer_check_active_state() (*mimocorb.mimo_buffer.NewBuffer method*), 10
observer_main() (*mimocorb.mimo_buffer.NewBuffer method*), 11
observer_server() (*mimocorb.mimo_buffer.NewBuffer method*), 11
ObserverData (*class in mimocorb.buffer_control*), 14

P

pause() (*mimocorb.buffer_control.buffer_control method*), 15
pause() (*mimocorb.mimo_buffer.NewBuffer method*), 11
plot_graph() (*in module plot_waveform*), 17
plot_waveform
    module, 17
plotWaveformBuffer (*class in mimocorb.plot_buffer*), 16
process_buffer() (*mimocorb.mimo_buffer.Writer method*), 13

R

rb_unittest
    module, 16
read_from_buffer
    module, 17
read_from_buffer() (*in module read_from_buffer*), 17
Reader (*class in mimocorb.mimo_buffer*), 12
reader_queue_listener() (*mimocorb.mimo_buffer.NewBuffer method*), 11
resume() (*mimocorb.buffer_control.buffer_control method*), 15
resume() (*mimocorb.mimo_buffer.NewBuffer method*), 11
run_mimoDAQ (*class in mimocorb.buffer_control*), 15
run_mimoDAQ.tc (*class in mimocorb.buffer_control*), 16

S

set_ending() (*mimocorb.buffer_control.buffer_control method*), 15
set_ending() (*mimocorb.mimo_buffer.NewBuffer method*), 11
set_metadata() (*mimocorb.mimo_buffer.Writer method*), 14
setup_workers() (*mimocorb.buffer_control.buffer_control method*), 15
shutdown() (*mimocorb.buffer_control.buffer_control method*), 15
shutdown() (*mimocorb.mimo_buffer.NewBuffer method*), 11
simulation_source
    module, 16
simulation_source() (*in module simulation_source*), 16
SourceToBuffer (*class in mimocorb.buffer_control*), 14
start_workers() (*mimocorb.buffer_control.buffer_control method*), 15

W

WaveformPlotter (*class in mimocorb.plot_buffer*), 16
Writer (*class in mimocorb.mimo_buffer*), 13
writer_queue_listener() (*mimocorb.mimo_buffer.NewBuffer method*), 11