mimoCoRB2

Release 0.0.3

Julian Baader

Aug 06, 2025

CONTENTS

CHAPTER

ONE

GETTING STARTED

This section will guide you through the process of installing mimoCoRB2 and running the examples.

1.1 Introduction

1.1.1 What is mimoCoRB2?

mimoCoRB2 (multiple in multiple out configurable ringbuffer manager) provides a central component of each data acquisition system needed to record and preanalyse data from randomly occurring processes. Typical examples are waveform data as provided by detectors common in quantum mechanical measurements, or in nuclear, particle 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 for fast collection of data and an efficient buffer manager delivering a constant data stream to the subsequent processing steps. While a data source feeds data into the buffer, consumer processes receive the data to filter, reduce, analyze or simply visualize the recorded data. In order to optimally use the available resources, multi-core and multi-processing techniques must be applied.

This project originated from an effort to structure and generalize data acquisition for several experiments in advanced physics laboratory courses at Karlsruhe Institute of Technology (KIT) and has been extensively tested with Ubuntu Linux.

1.1.2 What can it do?

Amongst the core features of mimoCoRB2 are:

- multiprocessing safe ringbuffer for NumPy structured arrays
- setup of multiple ringbuffers and workers from configuration files
- templates for common interactions between buffers (importing/exporting, filtering, processing, observing)
- pre built functions for common operations (oscilloscope, histogram, pulse height analysis)
- gui for monitoring and controlling the system

Currently the recommended way to install the package is from source (as it is not yet available on PyPI).

1.2 Installation from source

To install mimoCoRB2 from source, you need to clone the repository and install the package using pip. Here are the steps:

1. Clone the repository: `bash git clone https://github.com/JulianBaader/mimoCoRB2.git cd mimoCoRB2 `

2. Install the package using pip: `bash pip install . `

1.3 Examples

mimoCoRB2 comes with a set of examples that can be used to get started quickly. The examples are located in the *examples* directory of the mimoCoRB2 package alongside an explenation of the experiment. Each example is self-contained and can be run independently. The examples stem from experiments currently running at KIT and are meant to be used as a starting point for your own experiments.

To run the examples, you need to have the whole git repository cloned and installed. Then, you can run the examples by using the following command: `` `bash mimocorb2 <setup_file> ` `` where *<setup_file>* is the path to the setup file of the example you want to run. For example, to run the *muon* setup, you would use: `` `bash mimocorb2 examples/muon/spin_setup.yaml ` ``

Available examples include: - `` `examples/mimo_files/write_setup.yaml` ``: A simple example that writes waveform data in the mimo format. - `` `examples/mimo_files/read_setup.yaml` ``: A simple example that reads said waveform data. - `` `examples/redpitaya-spectroscopy/setup.yaml` ``: An example of a redpitaya-based gamma spectroscopy experiment. - `` `examples/muon/spin_setup.yaml` ``: An example of a muon experiment. - `` `examples/simple/setup.yaml` ``: A simple pulseheight analysis example.

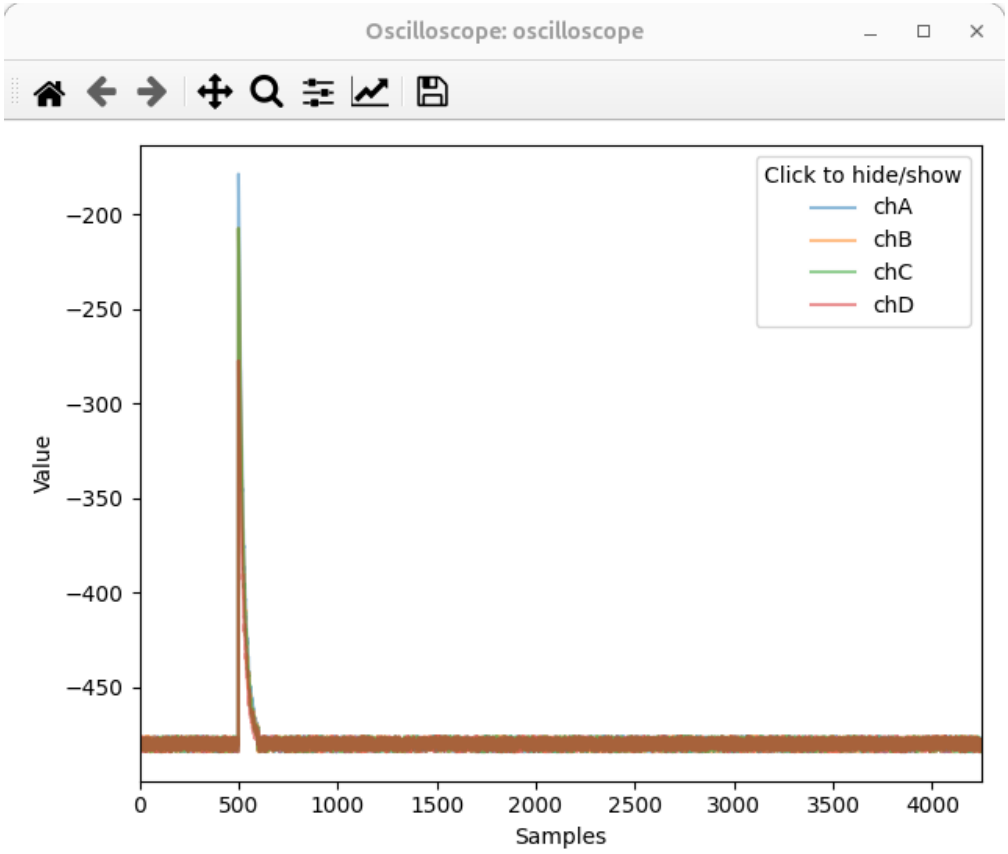
See the respective *examples/<example_name>/README.md* files for more details on each example.

The following section provides a brief overview of the standard interface with the muon example.

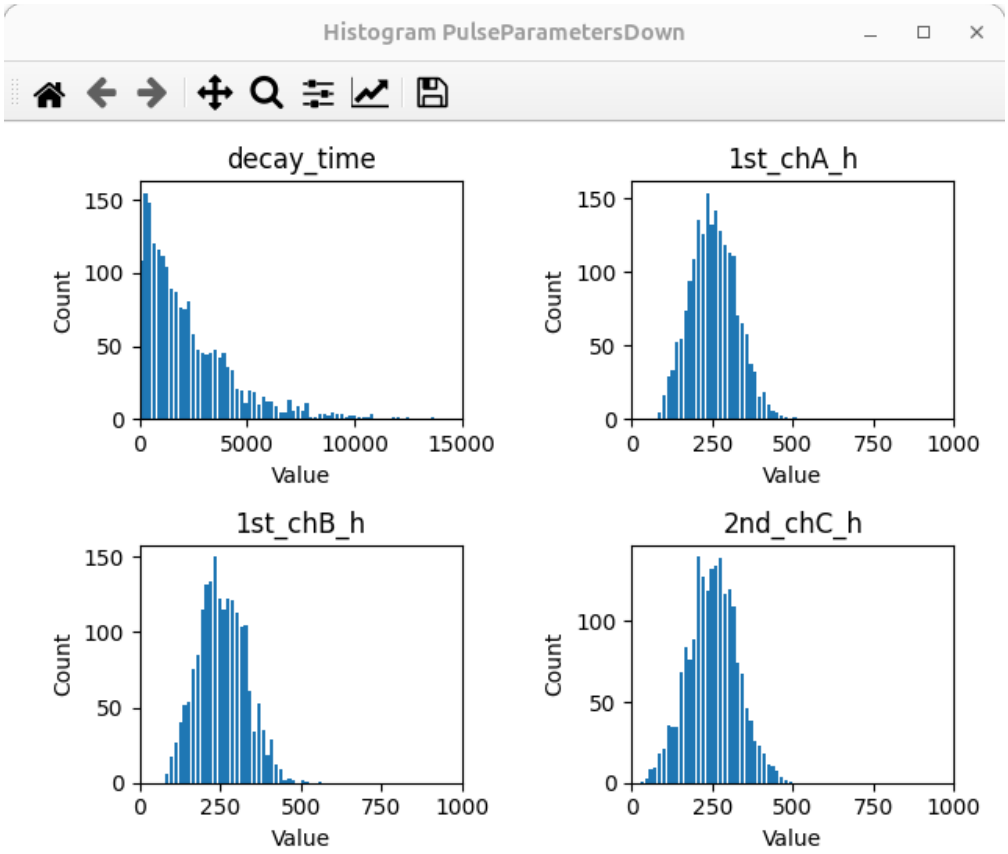
1.4 Muon Example Overview

When running the muon example, multiple windows will open:

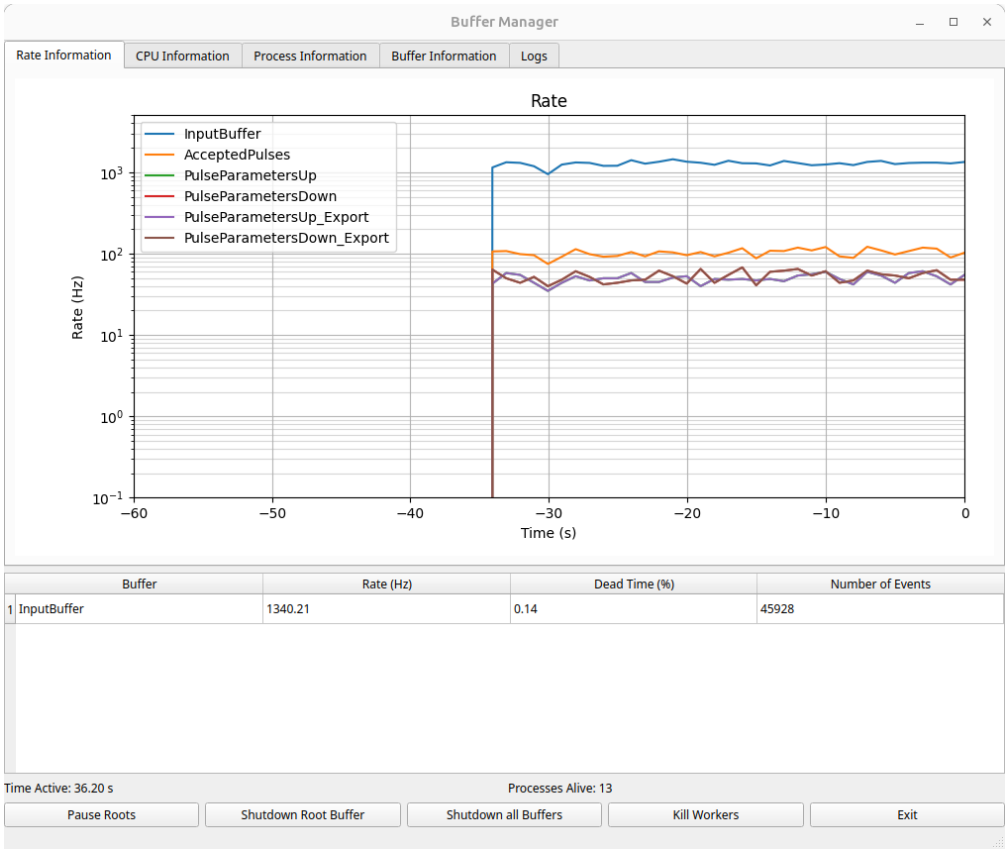
1. **Two Oscilloscopes**: These display the waveforms of the incomming data and the interesting data.



2. **Two Histograms**: These show the distribution of interesting parameters.



3. **Buffer Manager**: This window allows you to manage the whole DAQ suite.



The buffer manager has multiple tabs:

- **Rate Information**: Displays the rate of each buffer.
- **CPU Information**: Shows the CPU usage of each worker.
- **Process Information**: Shows how many processes each worker is running.
- **Buffer Information**: Displays the current state of each buffer.
- **Logs**: Shows the print statements of each worker.

Underneath the tabs, you can find a table, with exact numbers on the 'root buffers' In the row below the table, some additional information is displayed, such as the time the experiment has been running and the number of processes running.

At the bottom of the window, you can find the buttons to control the experiment:

- **Pause/Resume Roots** - Pauses or resumes the root buffers. (-> Any incoming data is discarded while paused.)
- **Shutdown Root Buffer** - Shuts down the root buffers which (ideally) shuts down all other buffers once they are empty.
- **Shutdown all Buffers** - Incase that there are still processes running, this will signal to all buffers to shut down.
- **Shutdown Workers** - Incase after that there are still processes running this will kill them (you might lose data).
- **Exit** - Closes the buffer manager. This is only possible if all processes are finished.

Each run of the experiment creates a new run directory in the *target_directory* specified in the setup file (default is inside the same directory as the setup file).

For every experiment run, the following files are created:

- *data_flow.svg*: A diagram of the Buffers and Workers used in the experiment.
- *setup.yaml*: The complete setup used for the experiment (including configs).
- *stats.csv*: A CSV file containing statistics about the experiment run.

In case of the muon example, the following additional files are created:

- *PulseParametersUp_Export.csv* and *PulseParametersDown_Export.csv*: CSV files containing the parameters of the pulses detected in the up and down direction.
- *AcceptedPulses.mimo*: A MIMO file containing the accepted pulses.
- *Histograms_PulseParametersUp/* and *Histograms_PulseParametersDown/*: Directories containing the histograms of the pulse parameters in the up and down direction, respectively.

CHAPTER

TWO

USER GUIDE

This user guide provides an in-depth description of mimoCoRB2 features required to build your own applications.

2.1 Setup

The setup file which is provided to the main script is a yaml file that defines the buffers and workers that will be used in the application. It describes the buffers and the data flow between them.

```yaml
Buffers:
    buffer_name_1:
        slot_count: int
        data_length: int
        data_dtype:
            field_name_1: field_dtype_1
            ...
    ...

Workers:
    worker_name_1:
        file: path_to_function_file
        function: function_name
        config: dict | str | [str | dict]
        number_of_processes: int
        sources: [str]
        sinks: [str]
        observes: [str]
    ...
base_config: dict | str | [str | dict]
target_directory: str   # Path to the target directory where the run will be stored.
                        # /path/to/target_directory -> absolute path
                        # path/to/target_directory -> relative path to the setup file
                        # ~/path/to/target_directory -> relative path to the user home␣
↪directory
```

2.1.1 Slot Count

The slot count is the number of slots that the buffer will have. Each slot can hold a data packet in the form of a numpy structured array. It should be higher than the number of processes that will be using the buffer, as well as high enough to buffer a reasonable amount of data.

2.1.2 Data Length

The data is stored in the form of a numpy structured array of shape (data_lenght,). It is therefore the number of elements per field in the data.

2.1.3 Data Dtype

The data type of the data stored in the buffer. It is a dictionary where the keys are the field names and the values are the field data types. The field data types are the same as the numpy data types. For example, 'int32', 'float64', 'S10' (string of length 10), etc.

2.1.4 File

This is the path (relative to the setup file) to the file that contains the function that will be used in the worker. If the key is missing or empty, the prebuilt functions will be used.

2.1.5 Function

The name of the function that will be used in the worker. (TODO see prebuilt functions)

2.1.6 Config

The configuration of the worker. It can be a dictionary, a string or a list of strings. If it is a dictionary, it will be passed directly to the worker. If it is a string or a list of strings the yaml file at each path (relative to the setup file) will be loaded in the config dictionary (duplicate keys will be overwritten).

2.1.7 Number of Processes

The number of processes run by the worker.

2.1.8 Sources, Sinks, Observes

The sources, sinks and observes of the worker. They are the names of the buffers that will be used by the worker. Sources are the buffers that will be used to read data from, sinks are the buffers that will be used to write data to and observes are the buffers that will be used to observe data.

2.2 Prebuilt Functions

Mimocorb2 comes with a set of built-in functions for common tasks. To use them, ommit the 'file' keyword or use an empty string in the Worker Setup. The functions are listet below:

2.2.1 Exporters

`exporters.drain()`

mimoCoRB2 Function: Drain buffer

Drains all data from the source buffer and does nothing with it.

Type

Exporter

Buffers

sources
1

sinks
0

observes
0

exporters.**histogram**()

mimoCoRB2 Function: Export data as a histogram.

Saves histograms of the data in the source buffer to npy files in the run_dir for each field in the source buffer. The histograms are saved in a directory named "Histograms_<source_buffer_name>". The directory contains a file named "info.csv" with the histogram configuration and individual npy files for each channel. It is possible to visualize the histograms using the *visualize_histogram* function.

Type

Exporter

Buffers

sources
1 with data_length = 1

sinks
Pass through data without modification to all sinks. Must share same dtype as source buffer.

observes
0

Configs

update_interval
[int, optional (default=1)] Interval in seconds to save the histogram data to files.

bins
[dict] Dictionary where keys are channel names and values are tuples of (min, max, number_of_bins). Channels must be present in the source buffer data.

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> info_df = pd.read_csv('info.csv')
>>> bins = {ch: np.linspace(info_df['Min'][i], info_df['Max'][i], info_df['NBins
↪'][i]) for i, ch in enumerate(info_df['Channel'])}
>>> for ch in info_df['Channel']:
...     data = np.load(f'{ch}.npy')
...     plt.plot(bins[ch][:-1], data, label=ch)
>>> plt.legend()
>>> plt.show()
```

exporters.**csv**()

mimoCoRB2 Function: Save data from the source buffer to a CSV file.

Saves data from the source buffer to a CSV file in the run_dir. Each field in the source buffer is saved as a column in the CSV file.

Type

Exporter

Buffers

sources
1 with data_length = 1

sinks
Pass through data without modification to all sinks. Must share same dtype as source buffer.

observes
0

Configs

save_interval
[int, optional (default=1)] Interval in seconds to save the CSV file.

filename
[str, optional (default='buffer_name')] Name of the CSV file to save the data to. The file will be saved in the run_dir.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> print(pd.read_csv('run_directory/exporter_name.csv'))
```

2.2.2 Visualizers

visualizers.**csv**()

mimoCoRB2 Function: Visualize histograms from the CSV exporter.

Visualizes histograms of the data in the source buffer using matplotlib. The histograms are read from the CSV file saved by the csv exporter.

Type

IsAlive

Buffers

sources
0

sinks
0

observes
1 the same as the source buffer of the exporter

Configs

update_interval
[int, optional (default=1)] Interval in seconds to update the histograms.

histtype
[str, optional (default='bar')] Passed to matplotlib's hist function. Options are 'bar', 'step', or 'stepfilled'.

bins
[dict] Dictionary where keys are channel names and values are tuples of (min, max, number_of_bins). Channels must be present in the source buffer data.

filename
[str, optional (default='buffer_name')] Name of the CSV file to save the data to. The file will be saved in the run_dir.

`visualizers.`**`histogram`**`()`

mimoCoRB2 Function: Visualize histograms from the histogram exporter.

Visualizes histograms of the data in the source buffer using matplotlib. The histograms are read from the npy files saved by the histogram exporter.

Type

IsAlive

Buffers

sources
0

sinks
0

observes
1 the same as the source buffer of the exporter

Configs

update_interval
[int, optional (default=1)] Interval in seconds to update the histograms.

plot_type
[str, optional (default='line')] Type of plot to use for the histograms. Options are 'line', 'bar', or 'step'.

2.2.3 Analyzers

`analyzers.`**`pha`**`()`

mimoCoRB2 Function: Pulse Height Analysis using scipy.signal.find_peaks

Analyzes pulse heights in a given channel of the input data using *scipy.signal.find_peaks*. This function processes the input data to find peaks and their properties based on the provided configuration parameters. Depending on the configuration, it can return various peak properties such as heights, thresholds, prominences, widths, and plateau sizes (see SciPy documentation).

Type

Processor

Buffers

sources
1 source buffer containing the input data with multiple channels

sinks
1 with data_length = 1 Possible field names:

- position
- **peak_heights**
  If *height* is specified in the config, the height of each peak in the specified channel.
- **left_thresholds, right_thresholds**
  If *threshold* is specified, the left and right thresholds of each peak.
- **prominences, left_bases, right_bases**
  If *prominence* is specified, the prominence of each peak, and its bases.
- **widths, width_heights, left_ips, right_ips**
  If *width* is specified, the full width at the specified relative height.
- **plateau_sizes, left_edges, right_edges**
  If *plateau_size* is specified, the size and edges of the peak plateau.

observes
0

Configs

channel
[str, optional (default = 'first channel')] Channel name to analyze. If not specified, the first channel in the input will be used.

height
[float, optional (default = None)] Minimum height of peaks. If None, peak heights are not calculated.

threshold
[float, optional (default = None)] Minimum vertical distance to neighbors. If None, thresholds are not calculated.

distance
[int, optional (default = None)] Minimum number of samples between neighboring peaks.

prominence
[float, optional (default = None)] Minimum prominence of peaks.

width
[float, optional (default = None)] Minimum width of peaks.

wlen
[int, optional (default = None)] Window length for prominence and width calculation.

rel_height
[float, optional (default = 0.5)] Relative height for width calculation (default = 0.5).

plateau_size
[float, optional (default = None)] Minimum size of the plateau at the peak.

2.2.4 Data

data.**export**()

mimoCoRB2 Function: Export data to a mimo file.

Exports data from a source buffer to a mimo file. This function is useful for saving data streams within the mimoCoRB2 framework.

Type

Exporter

Buffers

sources
1

sinks
Pass through data without modification to all sinks. Must share same dtype as source buffer.

observes
0

data.**simulate_importer**()

mimoCoRB2 Function: Simulate an Importer by inputting data according to the timestamps in a mimo file.

Imports data from a mimo file and simulates the Importer behavior by yielding data according to the timestamps in the file. This may lead to bunches of data if the timestamps are not ordered correctly.

Type

Importer

Buffers

sources
0

sinks
1 with the same dtype as the data in the mimo file

observes
0

Configs

filename
[str] Path to the mimo file to be imported.

data.**clocked_importer**()

mimoCoRB2 Function: Simulate an Importer by inputting data at a fixed rate.

Imports data from a mimo file and simulates the Importer behavior by yielding data at a fixed rate. This is useful for testing and simulating data streams in a controlled manner. Can be used to input uniform or poisson distributed data.

Type

Importer

Buffers

sources
0

sinks
1 with the same dtype as the data in the mimo file

observes
0

Configs

rate
[float] Rate at which to yield data in Hz

distribution
[str, optional (default='uniform')] Distribution to use for generating timestamps. Can be 'uniform' or 'poisson'.

filename
[str] Path to the mimo file to be imported.

2.2.5 Misc

`misc.copy()`

mimoCoRB2 Function: Copy data from one source to multiple sinks.

Copys data from a source buffer to multiple sink buffers. This function is useful for duplicating data streams within the mimoCoRB2 framework.

Type

Filter

Buffers

sources
1 source buffer containing the data to be copied

sinks
1 or more sink buffers that will receive the copied data

observes
0

2.2.6 Observers

`observers.oscilloscope()`

mimoCoRB2 Function: Show an Osilloscope plot of the buffer.

Observes data from a buffer and shows it as an oscilloscope plot.

Type

Observer

Buffers

sources
0

sinks
0

observes
1

Configs

ylim
[tuple of float, optional (default=None)] (min, max) of the y-axis. If None, the y-axis will be autoscaled upon each update.

t_scaling
[tuple of float, optional (default=(1, 0, 'Samples'))] (scaling, offset, unit) for the x-axis. The x-axis will be scaled accordingly.

y_scaling
[tuple of float, optional (default=(1, 0, 'Value'))] (scaling, offset, unit) for the y-axis. The y-axis will be scaled accordingly.

channels
[list of str, optional (default=None)] List of channel names to be plotted. If None, all available channels will be plotted.

trigger_level
[float, optional (default=None)] If specified, a horizontal line will be drawn at this level to indicate the trigger level.

update_interval
[float, optional (default=1)] Interval to update the plot in seconds. Default is 1 second.

colors
[list of str, optional (default=None)] List of colors to be used for the channels. If None, default matplotlib colors will be used.

2.2.7 Importers

redpitaya

`redpitaya.waveform()`

mimoCoRB2 Function: Use RedPitaya to acquire waveform data.

<longer description>

Type

Importer

Buffers

sources
0

sinks
1 with data_dtype: {'IN1': int16, 'IN2': int16} data_length decides the total number of samples (must be larger than number_of_samples_before_trigger and less than MAXIMUM_SAMPLES)

observes
0

Configs

ip: str
IP address of the RedPitaya device.

sample_rate: int
Number of samples (125MHz) averaged into one. Must be one of the values in SAMPLE_RATES.

negator_IN1: bool, optional (default=False)
If True, the IN1 input is negated.

negator_IN2: bool, optional (default=False)
If True, the IN2 input is negated.

trigger_slope: str, optional (default='rising')
Slope of the trigger. Must be one of the values in TRIGGER_SLOPES.

trigger_mode: str, optional (default='normal')
Mode of the trigger. Must be one of the values in TRIGGER_MODES.

trigger_level: int
Level of the trigger. Must be between MIN_ADC_VALUE and MAX_ADC_VALUE.

trigger_source: str, optional (default='IN1')
Source of the trigger. Must be one of the values in INPUTS.

number_of_samples_before_trigger: int
Number of samples to acquire before the trigger. Must be an integer between 0 and the total number of samples.

set_size: int, optional (default=100)
Number of sets to acquire in one acquisition.

2.3 Simple Workers

Simple Workers can be built using the classes provided in the **:py:module:`mimocorb2.worker_templates`** module.

In order to document your workers, you can use the following docstring format:

Listing 1: mimoCoRB2 docstring template

```
# This is a template for documenting mimoCoRB2 functions.
from mimocorb2 import BufferIO


def mimoCoRB2_function(buffer_io: BufferIO):
    """mimoCoRB2 Function: <short description of the function>
```

(continues on next page)

(continued from previous page)

```
    <longer description>

    Type
    ----
    <if a worker_template is used (e.g. Importer, Exporter, Filter, Processor, Observer)>

    Buffers
    -------
    sources
        <number and or description of source buffers>
    sinks
        <number and or description of sink buffers>
    observes
        <number and or description of observe buffers>

    Configs
    -------
    <key> : <type>
        <description>
    <key> : <type>, optional (default=<default value>)
        <description>

    Examples
    --------
    <example usage of the function, if applicable. In doctest format>
    """
    pass  # Replace with actual implementation
```

2.3.1 Importer

Importers are used to import data from an external source into the mimocorb2 system. They will automatically add the metadata to each event.

class mimocorb2.worker_templates.**Importer**(*io*)

Worker class for importing data from an external generator.

data_example

Example data from the buffer.

Type
np.ndarray

Examples

```
>>> def worker(buffer_io: BufferIO):
...     importer = Importer(buffer_io)
...     data_shape = importer.data_example.shape
...     def ufunc():
...         for i in range(buffer_io['n_events']):
...             data = np.random.normal(size=shape)
...             yield data
```

(continues on next page)

(continued from previous page)

```
...         yield None
...     importer(ufunc)
```

__call__(*ufunc*)

Start the generator and write data to the buffer.

ufunc must yield data of the same format as the io.data_out_examples[0] and yield None at the end. Metadata (counter, timestamp, deadtime) is automatically added to the buffer.

Parameters
ufunc (*Callable*) – Generator function that yields data and ends with None

Return type
None

__init__(*io*)

Checks the setup.

2.3.2 Exporter

Exporters are used to export data from the mimocorb2 system.

class mimocorb2.worker_templates.**Exporter**(*io*)

Worker class for exporting data and metadata.

If provided with an identical sink events will be copied to allow further analysis.

data_example

Example data from the buffer.

Type
np.ndarray

metadata_example

Example metadata from the buffer.

Type
np.ndarray

Examples

```
>>> def worker(buffer_io: BufferIO):
...     exporter = Exporter(buffer_io)
...     for data, metadata in exporter:
...         print(data, metadata)
```

__init__(*io*)

Checks the setup.

__iter__()

Start the exporter and yield data and metadata.

Yields data and metadata from the buffer until the buffer is shutdown.

Yields

- **data** (*np.ndarray, None*) – Data from the buffer
- **metadata** (*np.ndarray, None*) – Metadata from the buffer

Return type
Generator

2.3.3 Filter

Filters are used to filter data. This means that the data is not modified, but some of the data is removed.

class mimocorb2.worker_templates.**Filter**(*io*)

Worker class for filtering data from one buffer to other buffer(s).

Analyze data using ufunc(data) and copy or discard data based on the result.

data_example

Example data from the buffer.

Type
np.ndarray

Examples

```
>>> def worker(buffer_io: BufferIO):
...     filter = Filter(buffer_io)
...     min_height = buffer_io['min_height']
...     def ufunc(data):
...         if np.max(data) > min_height:
...             return True
...         else:
...             return False
...     filter(ufunc)
```

__call__(*ufunc*)

Start the filter and copy or discard data based on the result of ufunc(data).

Parameters
ufunc (*Callable*) – Function which will be called upon the data (Filter.reader.data_example). The function can return:

bool
True: copy data to every sink False: discard data

list[bool] (mapping to the sinks)
True: copy data to the corresponding sink False: dont copy data to the corresponding sink

Return type
None

__init__(*io*)

Checks the setup.

2.3.4 Processor

Processors are used to process data. This means that the data is modified in some way.

class mimocorb2.worker_templates.**Processor**(*io*)

Worker class for processing data from one buffer to other buffer(s).

Analyze data using ufunc(data) and send results to the corresponding sinks.

Examples

```
>>> def worker(buffer_io: BufferIO):
...     processor = Processor(buffer_io)
...     def ufunc(data):
...         return [data + 1, data - 1]
...     processor(ufunc)
```

__call__(*ufunc*)

Start the processor and process data using ufunc(data).

Parameters

ufunc (*Callable*) – Function which will be called upon the data (io.data_in_examples[0]). When the function returns None the data will be discarded. Otherwise the function must return a list of results, one for each sink. If the result is not None it will be written to the corresponding sink.

Return type

None

__init__(*io*)

Checks the setup.

2.3.5 Observer

Observers are used to observe data. This means that a copy of the data is exported.

class mimocorb2.worker_templates.**Observer**(*io*)

Worker class for observing data from a buffer.

data_example

Example data from the buffer.

Type

np.ndarray

Examples

```
>>> def worker(buffer_io: BufferIO):
...     observer = Observer(buffer_io)
...     generator = observer()
...     while True:
...         data, metadata = next(generator)
...         if data is None:
...             break
...         print(data, metadata)
...         time.sleep(1)
```

__call__()

Start the observer and yield data and metadata.

Yields data and metadata from the buffer until the buffer is shutdown.

Yields

- **data** (*np.ndarray, None*) – Data from the buffer
- **metadata** (*np.ndarray, None*) – Metadata from the buffer

Return type
Generator

__init__(*io*)

Checks the setup.

2.3.6 IsAlive

IsAlive workers are used to check if the system or a specific buffer is still alive.

class mimocorb2.worker_templates.**IsAlive**(*io*)

Worker class for checking if the buffer is alive.

This worker does not read or write any data, it only checks if the buffer provided as an observer is still alive.

Examples

```
>>> def worker(buffer_io: BufferIO):
...     is_alive = IsAlive(mimo_args)
...     while is_alive():
...         print("Buffer is alive")
...         time.sleep(1)
...     print("Buffer is dead")
```

__call__()

Check if the buffer is alive.

Returns
True if the buffer is alive, False otherwise.

Return type
bool

__init__(*io*)

Initialize the IsAlive worker.

Parameters
io (BufferIO) – BufferIO object containing the buffer to check.

2.4 Complex Workers

In case the simple workers are not enough, you can create highly customized workers by interacting with the BufferIO object passed to the worker directly.

2.4.1 BufferIO

class mimocorb2.mimo_worker.**BufferIO**(*name*, *sources*, *sinks*, *observes*, *config*, *setup_dir*, *run_dir*)

Collection of buffers for input/output operations.

Object which is passed to each worker process to provide access to the buffers.

name

The name of the corresponding worker.

Type
str

sources
List of source buffers (read).

Type
list[BufferReader]

sinks
List of sink buffers (write).

Type
list[BufferWriter]

observes
List of observe buffers.

Type
list[BufferObserver]

config
Configuration dictionary for the worker.

Type
Config

setup_dir
Directory where the setup file is located. (Load external data)

Type
Path

run_dir
Directory where the run is located. (Save external data)

Type
Path

logger

Type
logging.Logger

shutdown_sinks()
Shutdown all sink buffers.

__getitem__(*key*)
Get the value of a key from the configuration dictionary.

__str__()
String representation of the BufferIO object.

from_setup(*name*, *setup*, *setup_dir*, *run_dir*, *buffers*)
Create a BufferIO object from a setup dictionary.

Examples

```
>>> with io.write[0] as (metadata, data):
...     # write data and metadata to the buffer
...     pass
>>> with io.read[0] as (metadata, data):
```

(continues on next page)

(continued from previous page)

```
...     # read data and metadata from the buffer
...     pass
>>> with io.observe[0] as (metadata, data):
...     # observe data and metadata from the buffer
...     pass
>>> io.shutdown_sinks()  # Shutdown all sink buffers
```

2.5 Running an Experiment

Once you designed your setup, the required workers and their configs you can run an experiment. The experiment is controlled by the 'Control' class

class mimocorb2.control.**Control**(*setup_dict*, *setup_dir*, *mode='kbd+stats'*)

Central controller for managing buffers, workers, and interfaces in the mimoCorb2 framework.

This class sets up the runtime environment based on a provided setup dictionary or file, initializes and controls all *mimoBuffer* and *mimoWorker* instances, and handles user interaction through terminal, GUI, or logging interfaces. It also tracks and reports system statistics.

__call__()

Start the interfaces, workers, and control loop.

Return type
None

__init__(*setup_dict*, *setup_dir*, *mode='kbd+stats'*)

Initialize the Control instance.

Parameters

- **setup_dict** (dict) – Dictionary containing the setup configuration. See documentation for details.
- **setup_dir** (Path | str) – Directory where the setup file is located. All relative paths (configs and functions) will be resolved relative to this directory.
- **mode** (str (default: 'kbd+stats')) – Modes for the control interface. Can be a combination of: - 'kbd': Terminal interface for command input. - 'gui': GUI interface for control and monitoring. - 'stats': Log statistics to a file. One of kbd or gui is recommended to shutdown the control properly.

classmethod from_setup_file(*setup_file*, *mode='kbd+stats'*)

Create a Control instance from a setup file.

Parameters

- **setup_file** (Path | str) – Path to the YAML setup file containing the configuration for buffers and workers. The file should be structured as described in the documentation.
- **mode** (str (default: 'kbd+stats')) – Modes for the control interface. Can be a combination of: - 'kbd': Terminal interface for command input. - 'gui': GUI interface for control and monitoring. - 'stats': Log statistics to a file. One of kbd or gui is recommended to shutdown the control properly.

Returns
An instance of the Control class initialized with the setup from the file.

Return type
Control

CHAPTER

THREE

DEVELOPER GUIDE

The mimocorb2 library welcomes contributions from the community. This guide provides an overview of the inner workings of the library.

3.1 Buffers

3.1.1 mimo_buffer.py

Multiple In Multiple Out buffer. A module for managing multiprocessing-safe buffers using shared memory. This module is designed for high-performance data processing tasks where data must be shared across multiple processes efficiently.

The buffer itself is implemented as a shared memory which can be accessed by multiple processes through tokens. Each token represents a slot in the buffer. Tokens are either stored in the empty_slots queue (the corresponding slot is empty and can be written to) or in the filled_slots queue (the corresponding slot is filled). The Interfaces Reader, Writer, and Observer provide context management for reading, writing, and observing data in the buffer, respectively. They also handle the management of tokens, ensuring that data can be safely read or written without conflicts.

Classes

mimoBuffer
: Implements a ring buffer using shared memory to manage slots containing structured data and metadata.

Interface
: Base class for interacting with the buffer (Reader, Writer, Observer).

Reader
: Provides context management for reading data from the buffer.

Writer
: Provides context management for writing data to the buffer and sending flush events.

Observer
: Provides context management for observing data from the buffer without modifying it.

Examples

Creating and using a buffer for multiprocessing data handling:

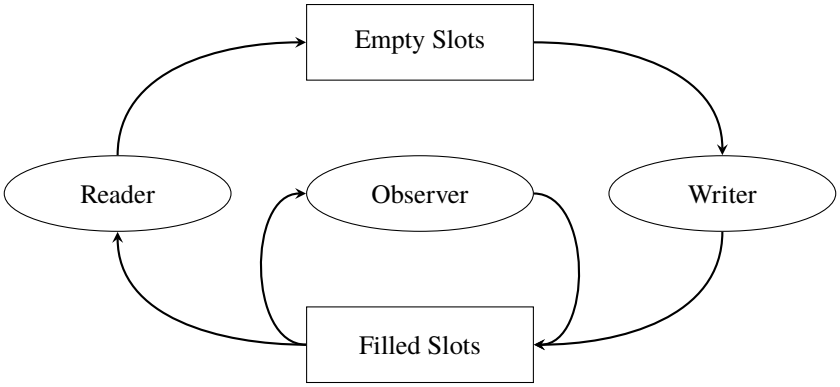
```
>>> import numpy as np
>>> from mimo_buffer import mimoBuffer, Writer, Reader
>>> buffer = mimoBuffer("example", slot_count=4, data_length=10, data_dtype=np.dtype([(
↪'value', '<f4')]))
>>> with Writer(buffer) as (data, metadata):
```

(continues on next page)

(continued from previous page)

```
...     data['value'][:] = np.arange(10)
...     metadata['counter'][0] = 1
>>> with Reader(buffer) as (data, metadata):
...     print(data['value'], metadata['counter'])
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] [1]
```

3.1.2 Token Handeling



PYTHON MODULE INDEX

m

INDEX

Symbols

B

C

D

E

F

H

I

L

M

N

O

P

R

S

W