

---

# **mimoCoRB**

***Release 1.0.***

**C. Mayer, K. Heitlinger, G. Quast**

**Feb 07, 2023**



**CONTENTS:**

<b>1</b>	<b>mimoCoRB Overview:</b>	<b>1</b>
1.1	Detailed desctiption of components . . . . .	1
<b>2</b>	<b>Access Classes in module <i>buffer_control</i></b>	<b>5</b>
2.1	Application example . . . . .	5
<b>3</b>	<b>Module Documentation</b>	<b>9</b>
	<b>Python Module Index</b>	<b>15</b>
	<b>Index</b>	<b>17</b>



## MIMOCORB 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 occurring 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 astroparticle 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 acquisition for several experiments in advanced physics laboratory courses at Karlsruhe Institute of Technology (KIT).

As a simple demonstration, we provide data from simulated signals as would be recorded by a detector for cosmic muons 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 acquisition, 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

Configuration 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 completeness of the 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 parallel 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 # number of data injections ("events")
Time_tick = 0.001 # time between events
Ncpul = 2 # number of parallel analyzer 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]
        # mimic 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()

    run_active = True

```

(continues on next page)

(continued from previous page)

```
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 RPTTest(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())
```



## 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 associated sub-processes for filling, filtering and extracting data.

- **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

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

### 2.1 Application example

The subdirectory *examples/* contains a rather complete application case. Code snippets and configuration data are provided in the subdirectories *examples/modules/* and *examples/config/*, respectively. Waveform data, as provided by a multi-channel oscilloscope, are generated and filled into the first of a cascaded set of three ringbuffers. The raw data are analysed, and accepted data with a double-pulse signature are selected and directly passed on to a second ringbuffer. The third buffer contains only the information about found signal pulses; a result file in *csv* format contains the data extracted from this buffer.

The buffer layout the associated functions are defined in the main configuration file *simultest\_setup.py*, which serves as the input the execution script *run\_daq.py* in the top-level directory of the package. The *python* files *simulation\_source.py*, *lifetime\_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.py* contains configurable data 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 from the directory *examples/* by entering:

```
../run_daq.py simultest_setup.yaml
```

The *yaml* file is shown here:

```

# 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:
            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"]

```

(continues on next page)

(continued from previous page)

```

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)
    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](#)



## MODULE DOCUMENTATION

mimo-buffer: Module implementing a multiprocessing capable buffer as described in ETP-KA/2022-06.

Buffer creation and management is handled by the `NewBuffer`-class, access to the buffer content is handled by the `Reader`, `Writer` and `Observer` classes.

**class** `mimocorb.mimo_buffer.NewBuffer`(*number\_of\_slots, values\_per\_slot, dtype, debug=False*)

Class to create a new ‘FIFO’ buffer object (typically called in the host/parent process).

Creates all the necessary memory shares, IPC queues, etc and launches background threads for buffer management.

The `NewBuffer`-object provides methods to create setup dictionaries for `Reader`, `Writer` or `Observer` instances.

### **buffer\_status()**

Processing Rate and approximate number of free slots in this buffer. 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 buffer

#### **Return type**

int

**event\_loop\_executor**(*loop: AbstractEventLoop*) → None

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

### **increment\_reader\_pointer()**

Internal method called by the `reader_queue_listener()`-threads after a new element was marked as ‘processing is done’. It checks if all reader groups are done with processing the oldest buffer slot, and if so, adds it to the ‘free buffer slots’ 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!

### **new\_observer()**

Method to create a new observer. It’s possible to create multiple observers or simply share the setup dictionary between different `Observer`-instances. It is very much possible for the data seen by the `Observer` instance to be corrupted (especially with non ideal buffer configurations and/or heavily loaded PC systems).

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

#### **Returns**

The `setup_dict` object passed to an `Observer`-instance to grant read access to this buffer.

#### **Return type**

dict

**new\_reader\_group()**

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

**Returns**

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

**Return type**

dict

**new\_writer()**

Method to create a new writer. It's possible to create multiple writers or simply share the setup dictionary between different `Writer`-instances.

**Returns**

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

**Return type**

dict

**async observer\_check\_active\_state()** → None

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

**async observer\_main()**

Internal asyncio 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 in a different process.

**async observer\_server(websocket, path)**

Internal asyncio 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 buffer). **CAUTION!** The buffer 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 is very much possible for the data seen by the `Observer` instance to be corrupted (especially with non ideal buffer configurations and/or heavily loaded PC systems).

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

**pause()**

Disable writing to buffer (paused)

**reader\_queue\_listener(done\_queue, done\_heap)**

Internal method run in a background thread (one for each reader group). It handles dispatching free ring buffer slots :param `done_queue`: the `multiprocessing.queue` created in `new_reader_group()` :param `done_heap`: the heap created in `new_reader_group()`

**resume()**

(Re)enable writing to buffer (resume)

**shutdown()**

Shut down the buffer, closing all background threads, terminating all processes associated with it (all processes using a `Reader`, `Writer` or `Observer` instance to access this buffer) and releasing the shared memory.

A ‘trickel down’ approach is used to have as few buffer elements as possible unprocessed. This may not work correctly with more complex signal analysis chains. So always make sure to shut down the buffers in data flow order (start with first element of the chain, the buffer 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’ buffer element from the ‘filled’-queue and distributes it to every reader groups ‘todo’-queue

**class** mimocorb.mimo\_buffer.**Observer**(*setup\_dict*)

Class to read select elements from a buffer.

The buffer may be created, filled and read by different processes. Buffer elements are structured NumPy arrays, the returned array won’t change until the next `Observer.get()` call, *not* blocking the buffer slot for the time being. Interfaces with the buffer manager via web socket (better error resilience compared to `multiprocessing.SimpleQueue`)

**async check\_active\_state()** → None

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

**async establish\_connection()** → None

Internal asyncio function establishing the websocket connection with the buffer 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 buffer by a `Writer` process.

#### **Returns**

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

#### **Return type**

`numpy.ndarray`

**async get\_new\_index()** → None

Internal asyncio function to query the index of the latest buffer slot from the buffer manager

**class** mimocorb.mimo\_buffer.**Reader**(*setup\_dict*)

Class to read elements from a buffer.

The buffer may be created, filled and read by different processes. Buffer elements are structured NumPy arrays and strictly **read-only**, the returned array won’t change until the next `Reader.get()` call, blocking the buffer slot for the time being. So a program design quickly processing the buffer content and calling `Reader.get()` again as soon as possible is highly advised.

**get()**

**Get a new element from the buffer, marking the last element obtained by calling**

this function 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 buffer.

#### **Raises**

**SystemExit** – If the `shutdown()`-method of the `NewBuffer` object was called, a `SystemExit` is raised, terminating the process this `Reader`-object was created in.

**Returns**

One element (structured `numpy.ndarray`) from the buffer as specified in the `dtype` of the `NewBuffer()`-object. All returned elements have not yet been processed by a process of this *reader group*.

**Return type**

`numpy.ndarray`

**get\_metadata()**

Get the metadata corresponding to the latest element obtained by calling the `Reader.get()`-method.

**Returns**

**Returned is a 3-tuple with (counter, timestamp, deadtime)**

of the latest element obtained from the buffer. The content of these variables is filled by the `Writer`-process (so may be changed), but 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 buffer.

The buffer may be created, filled and read by different processes. Buffer elements are structured NumPy arrays and may only be written to until `Writer.process_buffer()` is called or `Writer.get_new_buffer()` has been called again. The buffer slot is blocked while writes to the NumPy array are permitted, so a program design quickly writing the buffer content and calling `Writer.process_buffer()` or `Writer.get_new_buffer()` again as soon as possible is highly advised.

**get\_new\_buffer()**

**Get a new free spot in the buffer, marking the last element obtained by calling**

this function 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 spots in the buffer, always returning a valid NumPy array to be written to.

**Raises**

**SystemExit** – If the `shutdown()`-method of the `NewBuffer` object was called, a `SystemExit` is raised, terminating the process this `Writer`-object was created in.

**Returns**

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

**Return type**

`numpy.ndarray`



**process\_buffer()**

Mark the current 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 ring buffer slot is blocked while writing to the NumPy array obtained by calling `Writer.get_new_buffer()` is allowed, it is highly advised to call `Writer.process_buffer()` as soon as possible to unblock the the ring buffer.

**Returns**

Nothing

**set\_metadata(counter, timestamp, deadtime)**

Set the metadata corresponding to the current buffer element. If there is no current buffer element (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 (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.BufferToBuffer(source_list=None, sink_list=None, observe_list=None,
                                             config_dict=None, filter=None, **rb_info)
```

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

Args:

- buffer configurations (only one source and severals sinks, no observers!)
- function filter() 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, sink_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, sink_list=None, observe_list=None,
                                              config_dict=None, **rb_info)
```

Save data to file in csv-format

```
class mimocorb.buffer_control.SourceToBuffer(source_list=None, sink_list=None, observe_list=None,
                                             config_dict=None, data_source=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

**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

**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

**rb\_unittest:** application example for mimo\_buffer

This code may serve as a 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)

## PYTHON MODULE INDEX

### m

mimocorb, [9](#)  
mimocorb.buffer\_control, [13](#)  
mimocorb.mimo\_buffer, [9](#)

### r

rb\_unittest, [14](#)



## INDEX

### B

`buffer_control` (class in `mimocorb.buffer_control`), 13  
`buffer_status()` (`mimocorb.mimo_buffer.NewBuffer` method), 9  
`BufferToBuffer` (class in `mimocorb.buffer_control`), 13  
`BufferToParquetfile` (class in `mimocorb.buffer_control`), 13  
`BufferToTxtfile` (class in `mimocorb.buffer_control`), 13

### C

`check_active_state()` (`mimocorb.mimo_buffer.Observer` method), 11

### E

`establish_connection()` (`mimocorb.mimo_buffer.Observer` method), 11  
`event_loop_executor()` (`mimocorb.mimo_buffer.NewBuffer` method), 9  
`event_loop_executor()` (`mimocorb.mimo_buffer.Observer` method), 11

### G

`get()` (`mimocorb.mimo_buffer.Observer` method), 11  
`get()` (`mimocorb.mimo_buffer.Reader` method), 11  
`get_config()` (`mimocorb.buffer_control.buffer_control` static method), 14  
`get_metadata()` (`mimocorb.mimo_buffer.Reader` method), 12  
`get_new_buffer()` (`mimocorb.mimo_buffer.Writer` method), 12  
`get_new_index()` (`mimocorb.mimo_buffer.Observer` method), 11

### I

`import_function()` (`mimocorb.buffer_control.buffer_control` static method), 14

`increment_reader_pointer()` (`mimocorb.mimo_buffer.NewBuffer` method), 9

### M

`mimocorb` module, 9  
`mimocorb.buffer_control` module, 13  
`mimocorb.mimo_buffer` module, 9  
`module`  
`mimocorb`, 9  
`mimocorb.buffer_control`, 13  
`mimocorb.mimo_buffer`, 9  
`rb_unittest`, 14

### N

`new_observer()` (`mimocorb.mimo_buffer.NewBuffer` method), 9  
`new_reader_group()` (`mimocorb.mimo_buffer.NewBuffer` method), 9  
`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), 10  
`observer_server()` (`mimocorb.mimo_buffer.NewBuffer` method), 10

### P

`pause()` (`mimocorb.buffer_control.buffer_control` method), 14  
`pause()` (`mimocorb.mimo_buffer.NewBuffer` method), 10

`process_buffer()` (*mimocorb.mimo\_buffer.Writer*  
*method*), 12

## R

`rb_unittest`  
*module*, 14

`Reader` (*class in mimocorb.mimo\_buffer*), 11

`reader_queue_listener()` (*mimocorb.mimo\_buffer.NewBuffer*  
*method*), 10

`resume()` (*mimocorb.buffer\_control.buffer\_control*  
*method*), 14

`resume()` (*mimocorb.mimo\_buffer.NewBuffer* *method*), 10

## S

`set_metadata()` (*mimocorb.mimo\_buffer.Writer*  
*method*), 13

`setup_workers()` (*mimocorb.buffer\_control.buffer\_control* *method*), 14

`shutdown()` (*mimocorb.buffer\_control.buffer\_control*  
*method*), 14

`shutdown()` (*mimocorb.mimo\_buffer.NewBuffer*  
*method*), 10

`SourceToBuffer` (*class in mimocorb.buffer\_control*), 13

`start_workers()` (*mimocorb.buffer\_control.buffer\_control* *method*), 14

## W

`Writer` (*class in mimocorb.mimo\_buffer*), 12

`writer_queue_listener()` (*mimocorb.mimo\_buffer.NewBuffer*  
*method*), 11