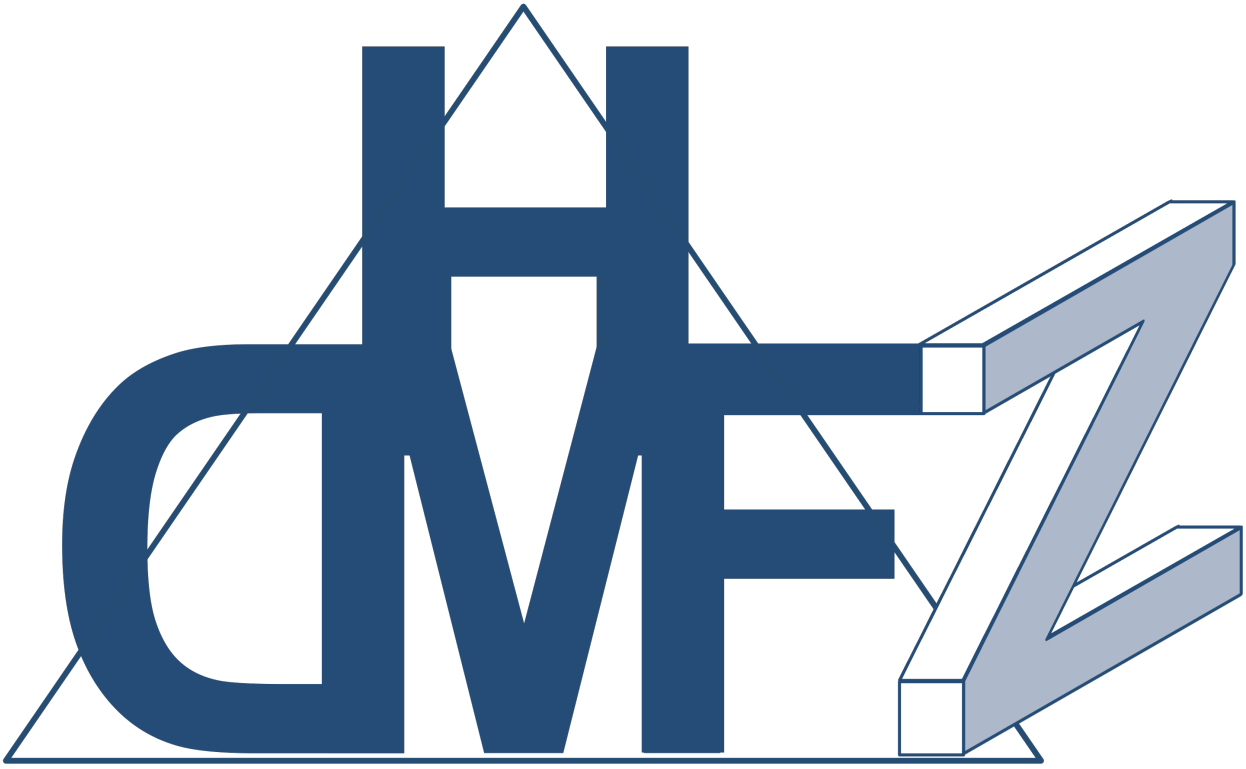

hdmf_zarr

Release 0.6.1.dev9+g72ff80f

Oliver Ruebel, Matthew Avaylon

May 02, 2024

FOR USERS:

hdmf_zarr implements a Zarr backend for HDMF as well as convenience classes for integration of Zarr with PyNWB to support writing of NWB files to Zarr.

Status: The Zarr backend is **under development** and may still change. See the *Overview* section for a description of available features and known limitations of hdmf-zarr.

CHAPTER

ONE

CITING HDMF-ZARR

- A. J. Tritt, O. Ruebel, B. Dichter, R. Ly, D. Kang, E. F. Chang, L. M. Frank, K. Bouchard, *"HDMF: Hierarchical Data Modeling Framework for Modern Science Data Standards,"* 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 165-179, doi: 10.1109/BigData47090.2019.9005648.

1.1 Installation

1.1.1 For Users

Install hdmf-zarr from PyPI

```
pip install hdmf-zarr
```

Install hdmf-zarr from conda-forge

```
conda install -c conda-forge hdmf-zarr
```

1.1.2 For Developers

Install hdmf-zarr from GitHub

The following illustrates how to install both `hdmf` and `hdfm_zarr` from GitHub in a Conda environment. Normally we don't need to install `hdmf` directly, but until `hdmf 3.4.0` is released we need to use the `dev` version of `hdmf`.

```
conda create --name hdmf-zarr-test python=3.9
conda activate hdmf-zarr-test
conda install h5py

git clone --recurse-submodules https://github.com/hdmf-dev/hdmf.git
cd hdmf
pip install -r requirements.txt -r requirements-dev.txt -r requirements-doc.txt -r␣
↪requirements-opt.txt
pip install -e .
cd ..

git clone https://github.com/hdmf-dev/hdmf-zarr.git
```

(continues on next page)

(continued from previous page)

```
cd hdmf-zarr
pip install -r requirements.txt -r requirements-dev.txt -r requirements-doc.txt
pip install -e .
```

Note: Depending on versions, it is possible that when installing `hdmf-zarr` that pip will install HDMF directly from PyPI instead of using the development version of HDMF that is already installed. In that case call `pip uninstall hdmf` and go to the `hdmf` directory and run `pip install -e .` again

1.2 Overview

1.2.1 Zarr Backend and Utilities

hdmf_zarr implements a Zarr backend for HDMF. Some of the key classes relevant for end-users are:

- *ZarrIO* implements an alternative storage backend to store data using HDMF via the Zarr library.
- *NWBZarrIO* uses *ZarrIO* to define a Zarr backend store for integration with PyNWB to simplify the use of *hdmf_zarr* with NWB (similar to `NWBHDF5IO` in PyNWB)
- *utils* implements utility classes for the *ZarrIO* backend. For end-users the *ZarrDataIO* class is relevant for defining advanced I/O options for datasets.

1.2.2 Supported features

- Write/Read of basic data types, strings and compound data types
- Chunking
- Compression and I/O filters
- Links
- Object references
- Writing/loading namespaces/specifications
- Iterative data write using `AbstractDataChunkIterator`
- Parallel write with `GenericDataChunkIterator` (since v0.4)
- Lazy load of datasets
- Lazy load of datasets containing object references (since v0.4)

1.2.3 Known Limitations

- Support for region references is not yet implemented. See also *Region references* for details.
- The Zarr backend is currently experimental and may still change.
- Attributes are stored as JSON documents in Zarr (using the DirectoryStore). As such, all attributes must be JSON serializable. The *ZarrIO* backend attempts to cast types to JSON serializable types as much as possible.
- Currently the *ZarrIO* backend supports Zarr's directory-based stores `DirectoryStore`, `NestedDirectoryStore`, and `TempStore`. Other Zarr stores could be added but will require proper treatment of links and references for those backends as links are not supported in Zarr (see zarr-python issues #389.
- Exporting of HDF5 files with external links is not yet fully implemented/tested. (see hdmf-zarr issue #49.
- Special characters (e.g., `:`, <, >, ", /, \, |, ?, or *) may not be supported by all file systems (e.g., on Windows) and as such should not be used as part of the names of Datasets or Groups as Zarr needs to create folders on the filesystem for these objects.

1.3 Tutorials

1.3.1 Zarr Dataset I/O

To customize data write settings on a per-dataset basis, HDMF supports wrapping of data arrays using `DataIO`. To support defining settings specific to Zarr `hdmf-zarr` provides the corresponding *ZarrDataIO* class.

Create an example DynamicTable Container

As a simple example, we first create a `DynamicTable` container to store some arbitrary data columns.

```
# Import DynamicTable and get the ROOT_NAME
from hdmf.common.table import DynamicTable, VectorData
from hdmf_zarr.backend import ROOT_NAME
from hdmf_zarr import ZarrDataIO
import numpy as np

# Setup a DynamicTable for managing data about users
data = np.arange(50).reshape(10, 5)
column = VectorData(
    name='test_data_default_settings',
    description='Some 2D test data',
    data=data)
test_table = DynamicTable(
    name=ROOT_NAME,
    description='a table containing data/metadata about users, one user per row',
    columns=(column, ),
    colnames=(column.name, )
)
```

Defining Data I/O settings

To define custom settings for write (e.g., for chunking and compression) we simply wrap our data array using *ZarrDataIO*.

```
from numcodecs import Blosc

data_with_data_io = ZarrDataIO(
    data=data * 3,
    chunks=(10, 10),
    fillvalue=0,
    compressor=Blosc(cname='zstd', clevel=1, shuffle=Blosc.SHUFFLE)
)
```

Adding the data to our table

```
test_table.add_column(
    name='test_data_zstd_compression',
    description='Some 2D test data',
    data=data_with_data_io)
```

Next we add a column where we explicitly disable compression

```
data_without_compression = ZarrDataIO(
    data=data*5,
    compressor=False)
test_table.add_column(
    name='test_data_nocompression',
    description='Some 2D test data',
    data=data_without_compression)
```

Note: To control linking to other datasets see the `link_data` parameter of *ZarrDataIO*

Note: In the case of `Data` (or here `VectorData`) we can also set the `DataIO` object to use via the `set_dataio()` function.

Writing and Reading

Reading and writing data with filters works as usual. See the *ZarrIO Overview* tutorial for details.

```
from hdmf.common import get_manager
from hdmf_zarr.backend import ZarrIO

zarr_dir = "example_data.zarr"
with ZarrIO(path=zarr_dir,  manager=get_manager(), mode='w') as zarr_io:
    zarr_io.write(test_table)
```

reading the table from Zarr

```
zarr_io = ZarrIO(path=zarr_dir,  manager=get_manager(), mode='r')
intable = zarr_io.read()
intable.to_dataframe()
```

Check dataset settings used.

```
for c in intable.columns:
    print("Name=%s, Chunks=% s, Compressor=%s" %
          (c.name,
           str(c.data.chunks),
           str(c.data.compressor)))
```

```
Name=test_data_default_settings, Chunks=(10, 5), Compressor=Blosc(cname='lz4', clevel=5,␣
↪shuffle=SHUFFLE, blocksize=0)
Name=test_data_zstd_compression, Chunks=(10, 10), Compressor=Blosc(cname='zstd',␣
↪clevel=1, shuffle=SHUFFLE, blocksize=0)
Name=test_data_nocompression, Chunks=(10, 5), Compressor=None
```

```
zarr_io.close()
```

1.3.2 Converting NWB HDF5 files to/from Zarr

This tutorial illustrates how to convert data between HDF5 and Zarr using a Neurodata Without Borders (NWB) file from the DANDI data archive as an example. In this tutorial we will convert our example file from HDF5 to Zarr and then back again to HDF5. The NWB standard is defined using HDMF and uses the `HDF5IO` HDF5 backend from HDMF for storage.

Setup

Here we use a small NWB file from the DANDI neurophysiology data archive from DANDIset 000009 as an example. To download the file directly from DANDI we can use:

```
from dandi.dandiapi import DandiAPIClient
dandiset_id = "000009"
filepath = "sub-anm00239123/sub-anm00239123_ses-20170627T093549_ecephys+ogen.nwb"   # ~0.
↪5MB file
with DandiAPIClient() as client:
    asset = client.get_dandiset(dandiset_id, 'draft').get_asset_by_path(filepath)
    s3_path = asset.get_content_url(follow_redirects=1, strip_query=True)
    filename = os.path.basename(asset.path)
asset.download(filename)
```

We here use a local copy of a small file from this DANDIset as an example:

```
import os
import shutil
from pynwb import NWBHDF5IO
from hdmf_zarr.nwb import NWBZarrIO
from contextlib import suppress

# Input file to convert
```

(continues on next page)

(continued from previous page)

```
basedir = "resources"
filename = os.path.join(basedir, "sub_anm00239123_ses_20170627T093549_ecephys_and_ogen.
↪nwb")
# Zarr file to generate for converting from HDF5 to Zarr
zarr_filename = "test_zarr_" + os.path.basename(filename) + ".zarr"
# HDF5 file to generate for converting from Zarr to HDF5
hdf_filename = "test_hdf5_" + os.path.basename(filename)

# Delete our converted HDF5 and Zarr file from previous runs of this notebook
for fname in [zarr_filename, hdf_filename]:
    if os.path.exists(fname):
        print("Removing %s" % fname)
        if os.path.isfile(fname):  # Remove a single file (here the HDF5 file)
            os.remove(fname)
        else:  # remove whole directory and subtree (here the Zarr file)
            shutil.rmtree(zarr_filename)
```

Convert the NWB file from HDF5 to Zarr

To convert files between storage backends, we use HMDF's export functionality. As this is an NWB file, we here use the `pynwb.NWBHDF5IO` backend for reading the file from from HDF5 and use the *`NWBZarrIO`* backend to export the file to Zarr.

```
with NWBHDF5IO(filename, 'r', load_namespaces=False) as read_io:  # Create HDF5 IO␣
↪object for read
    with NWBZarrIO(zarr_filename, mode='w') as export_io:         # Create Zarr IO␣
↪object for write
        export_io.export(src_io=read_io, write_args=dict(link_data=False))   # Export␣
↪from HDF5 to Zarr
```

Note: When converting between backends we need to set `link_data=False` as linking from Zarr to HDF5 (and vice-versa) is not supported.

Read the Zarr file back in

```
zr = NWBZarrIO(zarr_filename, 'r')
zf = zr.read()
```

The basic behavior of the `NWBFile` object is the same.

```
# Print the NWBFile to illustrate that
print(zf)
```

```
root pynwb.file.NWBFile at 0x139661578312720
Fields:
  devices: {
    ADunit <class 'pynwb.device.Device'>,
    laser <class 'pynwb.device.Device'>
```

(continues on next page)

(continued from previous page)

```
  }
  electrode_groups: {
    ADunit_32 <class 'pynwb.ecephys.ElectrodeGroup'>
  }
  electrodes: electrodes <class 'hdmf.common.table.DynamicTable'>
  experiment_description: N/A
  experimenter: ['Zengcai Guo']
  file_create_date: [datetime.datetime(2019, 10, 7, 15, 10, 30, 595741,␣
↪tzinfo=tzoffset(None, -18000))]
  identifier: anm00239123_2017-06-27_09-35-49
  institution: Janelia Research Campus
  intervals: {
    trials <class 'pynwb.epoch.TimeIntervals'>
  }
  keywords: <zarr.core.Array '/general/keywords' (6,) object read-only>
  ogen_sites: {
    left-ALM <class 'pynwb.ogen.OptogeneticStimulusSite'>
  }
  related_publications: ['doi:10.1038/nature22324']
  session_description: Extracellular ephys recording of mouse doing discrimination␣
↪task(lick left/right), with optogenetic stimulation plus pole and auditory stimulus
  session_start_time: 2017-06-27 09:35:49-05:00
  subject: subject pynwb.file.Subject at 0x139661439324064
Fields:
  genotype: Ai32 x PV-Cre
  sex: M
  species: Mus musculus
  subject_id: anm00239123

  timestamps_reference_time: 2017-06-27 09:35:49-05:00
  trials: trials <class 'pynwb.epoch.TimeIntervals'>
  units: units <class 'pynwb.misc.Units'>
```

The main difference is that datasets are now represented by Zarr arrays compared to h5py Datasets when reading from HDF5.

```
print(type(zf.trials['start_time'].data))
```

```
<class 'zarr.core.Array'>
```

For illustration purposes, we here show a few columns of the Trials table.

```
zf.trials.to_dataframe()[['start_time', 'stop_time', 'type', 'photo_stim_type']]
zr.close()
```

Convert the Zarr file back to HDF5

Using the same approach as above, we can now convert our Zarr file back to HDF5.

```
with suppress(Exception):  # TODO: This is a temporary ignore on the convert_dtype␣
↪exception.
    with NWBZarrIO(zarr_filename, mode='r') as read_io:  # Create Zarr IO object for read
        with NWBHDF5IO(hdf_filename, 'w') as export_io:  # Create HDF5 IO object for␣
↪write
            export_io.export(src_io=read_io, write_args=dict(link_data=False))  # Export␣
↪from Zarr to HDF5
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hdmf-zarr/envs/dev/lib/python3.9/site-
↪packages/hdmf/build/objectmapper.py:260: DtypeConversionWarning: Spec 'Units/spike_
↪times_index': Value with data type int32 is being converted to data type uint32 (min␣
↪specification: uint8).
  warnings.warn(full_warning_msg, DtypeConversionWarning)
/home/docs/checkouts/readthedocs.org/user_builds/hdmf-zarr/envs/dev/lib/python3.9/site-
↪packages/hdmf/build/objectmapper.py:260: DtypeConversionWarning: Spec 'Units/
↪electrodes_index': Value with data type int32 is being converted to data type uint32␣
↪(min specification: uint8).
  warnings.warn(full_warning_msg, DtypeConversionWarning)
```

Read the new HDF5 file back

Now our file has been converted from HDF5 to Zarr and back again to HDF5. Here we check that we can still read that file.

```
with suppress(Exception):  # TODO: This is a temporary ignore on the convert_dtype␣
↪exception.
    with NWBHDF5IO(hdf_filename, 'r') as hr:
        hf = hr.read()
```

1.3.3 ZarrIO Overview

The *ZarrIO* backend behaves in general much like the standard HDF5IO available with HDMF and is an adaptation of that backend to use Zarr instead of HDF5

Create an example DynamicTable Container

As a simple example, we here create a basic DynamicTable for describing basic user data.

Note: When writing a DynamicTable (or any Container that is normally not intended to be the root of a file) we need to use *hdmf_zarr.backend.ROOT_NAME* as the name for the Container to ensure that link paths are created correctly by *ZarrIO*. This is due to the fact that the top-level Container used during I/O is written as the root of the file. As such, the name of the root Container of a file does not appear in the path to locate it.

```
# Import DynamicTable and get the ROOT_NAME
from hdmf.common.table import DynamicTable
from hdmf_zarr.backend import ROOT_NAME

# Setup a DynamicTable for managing data about users
users_table = DynamicTable(
    name=ROOT_NAME,
    description='a table containing data/metadata about users, one user per row',
)
users_table.add_column(
    name='first_name',
    description='the first name of the user',
)
users_table.add_column(
    name='last_name',
    description='the last name of the user',
)

users_table.add_column(
    name='phone_number',
    description='the phone number of the user',
    index=True,
)

# Add some simple example data to our table
users_table.add_row(
    first_name='Grace',
    last_name='Hopper',
    phone_number=['123-456-7890']
)
users_table.add_row(
    first_name='Alan',
    last_name='Turing',
    phone_number=['555-666-7777', '888-111-2222']
)

# Show the table for validation
users_table.to_dataframe()
```

Writing the table to Zarr

```
from hdmf.common import get_manager
from hdmf_zarr.backend import ZarrIO

zarr_dir = "example.zarr"
with ZarrIO(path=zarr_dir,  manager=get_manager(), mode='w') as zarr_io:
    zarr_io.write(users_table)
```

Reading the table from Zarr

```
zarr_io = ZarrIO(path=zarr_dir,  manager=get_manager(), mode='r')
intable = zarr_io.read()
intable.to_dataframe()
```

```
zarr_io.close()
```

Converting to/from HDF5 using `export`

Exporting the Zarr file to HDF5

To convert our Zarr file to HDF5 we can now simply read the file with our *ZarrIO* backend and the export the file using HDMF's `HDF5IO` backend

```
from hdmf.backends.hdf5 import HDF5IO

with ZarrIO(path=zarr_dir,  manager=get_manager(), mode='r') as zarr_read_io:
    with HDF5IO(path="example.h5", manager=get_manager(), mode='w') as hdf5_export_io:
        hdf5_export_io.export(src_io=zarr_read_io, write_args=dict(link_data=False))  #␣
↪use export!
```

Note: When converting between backends we need to set `link_data=False` as linking between different storage backends (here from HDF5 to Zarr and vice versa) is not supported.

Check that the HDF5 file is correct

```
with HDF5IO(path="example.h5", manager=get_manager(), mode='r') as hdf5_read_io:
    intable_from_hdf5 = hdf5_read_io.read()
    intable_hdf5_df = intable_from_hdf5.to_dataframe()
intable_hdf5_df  # display the table in the gallery output
```

Exporting the HDF5 file to Zarr

In the same way as above, we can now also convert our HDF5 file back to Zarr simply by reading the HDF5 file using HDMF's `HDF5IO` backend and the exporting the file using the *ZarrIO* backend.

```
with HDF5IO(path="example.h5", manager=get_manager(), mode='r') as hdf5_read_io:
    with ZarrIO(path="example_exp.zarr",  manager=get_manager(), mode='w') as zarr_
↪export_io:
        zarr_export_io.export(src_io=hdf5_read_io, write_args=dict(link_data=False))  #␣
↪use export!
```

Check that the Zarr file is correct

```
with ZarrIO(path="example_exp.zarr", manager=get_manager(), mode='r') as zarr_read_io:
    intable_from_zarr = zarr_read_io.read()
    intable_zarr_df = intable_from_zarr.to_dataframe()
intable_zarr_df  # display the table in the gallery output
```

Using custom Zarr storage backends

ZarrIO supports a subset of data stores available for Zarr, e.g., :py:class`~zarr.storage.DirectoryStore`, :py:class`~zarr.storage.TempStore`, and :py:class`~zarr.storage.NestedDirectoryStore`. The supported stores are defined in *SUPPORTED_ZARR_STORES*. A main limitation to supporting all possible Zarr stores in *ZarrIO* is due to the fact that Zarr does not support links and references.

To use a store other than the default, we simply need to instantiate the store and set pass it to *ZarrIO* via the `path` parameter. Here we use a :py:class`~zarr.storage.NestedDirectoryStore` to write a simple `hdmf.common.CSRMatrix` container to disk.

```
from zarr.storage import NestedDirectoryStore
from hdmf.common import CSRMatrix

zarr_nsd_dir = "example_nested_store.zarr"
store = NestedDirectoryStore(zarr_dir)
csr_container = CSRMatrix(
    name=ROOT_NAME,
    data=[1, 2, 3, 4, 5, 6],
    indices=[0, 2, 2, 0, 1, 2],
    indptr=[0, 2, 3, 6],
    shape=(3, 3))

# Write the csr_container to Zarr using a NestedDirectoryStore
with ZarrIO(path=zarr_nsd_dir,  manager=get_manager(), mode='w') as zarr_io:
    zarr_io.write(csr_container)

# Read the CSR matrix to confirm the data was written correctly
with ZarrIO(path=zarr_nsd_dir, manager=get_manager(), mode='r') as zarr_io:
    csr_read = zarr_io.read()
    print(" data=%s\n indices=%s\n indptr=%s\n shape=%s" %
          (str(csr_read.data), str(csr_read.indices), str(csr_read.indptr), str(csr_read.
↪shape)))
```

```
data=[1 2 3 4 5 6]
indices=[0 2 2 0 1 2]
indptr=[0 2 3 6]
shape=[3 3]
```

1.3.4 Creating NWB files using `NWBZarrIO`

Similar to `pynwb.NWBHDF5IO`, the *NWBZarrIO* extends the basic *ZarrIO* to perform default setup for BuildManager, loading or namespaces etc., in the context of the NWB format, to simplify using hdmf-zarr with the NWB data standard. With this we can use *NWBZarrIO* directly with the PyNWB API to read/write NWB files to/from Zarr. I.e., we can follow the standard PyNWB tutorials for using NWB files, and only need to replace `pynwb.NWBHDF5IO` with *NWBZarrIO* for read/write (and replace the use of py:class:*H5DataIO* with *ZarrDataIO*).

Creating and NWB extracellular electrophysiology file

As an example, we here create an extracellular electrophysiology NWB file. This example is derived from Extracellular Electrophysiology tutorial.

```
# Ignore warnings about the development of the ZarrIO backend

from datetime import datetime
from dateutil.tz import tzlocal
import zarr

import numpy as np
from pynwb import NWBFile
from pynwb.ecephys import ElectricalSeries, LFP

# Create the NWBFile
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier="EXAMPLE_ID",
    session_start_time=datetime.now(tzlocal()),
    experimenter="Dr. Bilbo Baggins",
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure with thirteen dwarves "
    "to reclaim vast treasures.",
    session_id="LONELYMTN",
)

# Create a device
device = nwbfile.create_device(
    name="array", description="the best array", manufacturer="Probe Company 9000"
)

# Add electrodes and electrode groups to the NWB file
nwbfile.add_electrode_column(name="label", description="label of electrode")

nshanks = 4
nchannels_per_shank = 3
electrode_counter = 0

for ishank in range(nshanks):
    # create an electrode group for this shank
    electrode_group = nwbfile.create_electrode_group(
        name="shank{}".format(ishank),
        description="electrode group for shank {}".format(ishank),
        device=device,
        location="brain area",
    )
    # add electrodes to the electrode table
    for ielec in range(nchannels_per_shank):
        nwbfile.add_electrode(
            x=5.3, y=1.5, z=8.5, imp=np.nan,
            filtering='unknown',
```

(continues on next page)

(continued from previous page)

```
            group=electrode_group,
            label="shank{}elec{}".format(ishank, ielec),
            location="brain area",
        )
        electrode_counter += 1

# Create a table region to select the electrodes to record from
all_table_region = nwbfile.create_electrode_table_region(
    region=list(range(electrode_counter)),  # reference row indices 0 to N-1
    description="all electrodes",
)

# Add a mock electrical recording acquisition to the NWBFile
raw_data = np.random.randn(50, len(all_table_region))
raw_electrical_series = ElectricalSeries(
    name="ElectricalSeries",
    data=raw_data,
    electrodes=all_table_region,
    starting_time=0.0,  # timestamp of the first sample in seconds relative to the␣
↪session start time
    rate=20000.0,  # in Hz
)
nwbfile.add_acquisition(raw_electrical_series)

# Add a mock LFP processing result to the NWBFile
lfp_data = np.random.randn(50, len(all_table_region))
lfp_electrical_series = ElectricalSeries(
    name="ElectricalSeries",
    data=lfp_data,
    electrodes=all_table_region,
    starting_time=0.0,
    rate=200.0,
)
lfp = LFP(electrical_series=lfp_electrical_series)
ecephys_module = nwbfile.create_processing_module(
    name="ecephys", description="processed extracellular electrophysiology data"
)
ecephys_module.add(lfp)

# Add mock spike times and units to the NWBFile
nwbfile.add_unit_column(name="quality", description="sorting quality")

poisson_lambda = 20
firing_rate = 20
n_units = 10
for n_units_per_shank in range(n_units):
    n_spikes = np.random.poisson(lam=poisson_lambda)
    spike_times = np.round(
        np.cumsum(np.random.exponential(1 / firing_rate, n_spikes)), 5
    )
    nwbfile.add_unit(
        spike_times=spike_times, quality="good", waveform_mean=[1.0, 2.0, 3.0, 4.0, 5.0]
```

(continues on next page)

(continued from previous page)

```
    )
```

Writing the NWB file to Zarr

```
from hdmf_zarr.nwb import NWBZarrIO
import os

path = "ecephys_tutorial.nwb.zarr"
absolute_path = os.path.abspath(path)
with NWBZarrIO(path=path, mode="w") as io:
    io.write(nwbfile)
```

Test opening with the absolute path instead

The main reason for using the `absolute_path` here is for testing purposes to ensure links and references work as expected. Otherwise, using the relative `path` here instead is fine.

```
with NWBZarrIO(path=absolute_path, mode="r") as io:
    infile = io.read()
```

Consolidating Metadata

When writing to Zarr, the metadata within the file will be consolidated into a single file within the root group, *.zmetadata*. Users who do not wish to consolidate the metadata can set the boolean parameter *consolidate_metadata* to *False* within *write*. Even when the metadata is consolidated, the metadata natively within the file can be altered. Any alterations within would require the user to call *zarr.convenience.consolidate_metadata()* to sync the file with the changes. Please refer to the Zarr documentation for more details: https://zarr.readthedocs.io/en/stable/tutorial.html#storage-alternatives

```
zarr.consolidate_metadata(path)
```

```
<zarr.hierarchy.Group '/'>
```

1.4 Resources

1.4.1 sub_anm00239123_ses_20170627T093549_ecephys_and_ogen.nwb

This NWB file was downloaded from DANDIset 000009 The file was modified to replace `:` characters used in the name of the `ElectrodeGroup` called `ADunit:  32` in `'general/extracellular_ephys/` to `'ADunit_32'`. The dataset `general/extracellular_ephys/electrodes/group_name` as part of the electrodes table was updated accordingly to list the appropriate group name. This is to avoid issues on Windows file systems that do not support `:` as part of folder names. The asses can be downloaded from DANDI via:

```
from dandi.dandiapi import DandiAPIClient
dandiset_id = "000009"
filepath = "sub-anm00239123/sub-anm00239123_ses-20170627T093549_ecephys+ogen.nwb"    # ~0.
```

(continues on next page)

(continued from previous page)

```
↪5MB file
with DandiAPIClient() as client:
    asset = client.get_dandiset(dandiset_id, 'draft').get_asset_by_path(filepath)
    s3_path = asset.get_content_url(follow_redirects=1, strip_query=True)
    filename = os.path.basename(asset.path)
asset.download(filename)
```

Resources

sub_anm00239123_ses_20170627T093549_ecephys_and_ogen.nwb

This NWB file was downloaded from DANDIset 000009 The file was modified to replace `:` characters used in the name of the `ElectrodeGroup` called `ADunit:  32` in `'general/extracellular_ephys/` to `'ADunit_32'`. The dataset `general/extracellular_ephys/electrodes/group_name` as part of the electrodes table was updated accordingly to list the appropriate group name. This is to avoid issues on Windows file systems that do not support `:` as part of folder names. The asses can be downloaded from DANDI via:

```
from dandi.dandiapi import DandiAPIClient
dandiset_id = "000009"
filepath = "sub-anm00239123/sub-anm00239123_ses-20170627T093549_ecephys+ogen.nwb"    # ~0.
↪5MB file
with DandiAPIClient() as client:
    asset = client.get_dandiset(dandiset_id, 'draft').get_asset_by_path(filepath)
    s3_path = asset.get_content_url(follow_redirects=1, strip_query=True)
    filename = os.path.basename(asset.path)
asset.download(filename)
```

1.5 Storage Specification

hdmf-zarr currently uses the Zarr DirectoryStore, which uses directories and files on a standard file system to serialize data.

1.5.1 Format Mapping

Here we describe the mapping of HDMF primitives (e.g., Groups, Datasets, Attributes, Links, etc.) used by the HDMF schema language to Zarr storage primitives. HDMF data modeling primitives were originally designed with HDF5 in mind. However, Zarr uses very similar primitives, and as such the high-level mapping between HDMF schema and Zarr storage is overall fairly simple. The main complication is that Zarr does not support links and references (see Zarr issue #389) and as such have to implemented by hdmf-zarr.

Table 1: Mapping of groups

NWB Primitive	Zarr Primitive
Group	Group
Dataset	Dataset
Attribute	Attribute
Link	Stored as JSON formatted Attributes

1.5.2 Mapping of HDMF specification language keys

Here we describe the mapping of keys from the HDMF specification language to Zarr storage objects:

Groups

Table 2: Mapping of groups

NWB Key	Zarr
name	Name of the Group in Zarr
doc	Zarr attribute `doc` on the Zarr group
groups	Zarr groups within the Zarr group
datasets	Zarr datasets within the Zarr group
attributes	Zarr attributes on the Zarr group
links	Stored as JSON formatted attributes on the Zarr Group
quantity	Not mapped; Number of appearances of the group
neurodata_type	Attribute `neurodata_type` on the Zarr Group
namespace ID	Attribute `namespace` on the Zarr Group
object ID	Attribute `object_id` on the Zarr Group

Reserved groups

The *`ZarrIO`* backend typically caches the schema used to create a file in the group `/specifications` (see also *Caching format specifications*)

Datasets

Table 3: Mapping of datasets

HDMF Specification Key	Zarr
name	Name of the dataset in Zarr
doc	Zarr attribute `doc` on the Zarr dataset
dtype	Data type of the Zarr dataset (see *dtype mappings* table) and stored in the `zarr_dtype` reserved attribute
shape	Shape of the Zarr dataset if the shape is fixed, otherwise shape defines the maxshape
dims	Not mapped
attributes	Zarr attributes on the Zarr dataset
quantity	Not mapped; Number of appearances of the dataset
neurodata_type	Attribute `neurodata_type` on the Zarr dataset
namespace ID	Attribute `namespace` on the Zarr dataset
object ID	Attribute `object_id` on the Zarr dataset

Note:

- TODO Update mapping of dims

Attributes

Table 4: Mapping of attributes

HDMF Specification Key	Zarr
name	Name of the attribute in Zarr
doc	Not mapped; Stored in schema only
dtype	Data type of the Zarr attribute
shape	Shape of the Zarr attribute if the shape is fixed, otherwise shape defines the maxshape
dims	Not mapped; Reflected by the shape of the attribute data
required	Not mapped; Stored in schema only
value	Data value of the attribute

Note: Attributes are stored as JSON documents in Zarr (using the DirectoryStore). As such, all attributes must be JSON serializable. The *ZarrIO* backend attempts to cast types (e.g., numpy arrays) to JSON serializable types as much as possible, but not all possible types may be supported.

Reserved attributes

The *ZarrIO* backend defines a set of reserved attribute names defined in `__reserve_attribute`. These reserved attributes are used to implement functionality (e.g., links and object references, which are not natively supported by Zarr) and may be added on any Group or Dataset in the file.

Reserved Attribute Name	Usage
zarr_link	Attribute used to store links. See *Links* for details.
zarr_dtype	Attribute used to specify the data type of a dataset. This is used to implement the storage of object references as part of datasets. See *Object References*

In addition, the following reserved attributes are added to the root Group of the file only:

Reserved Attribute Name	Usage
.specloc	Attribute storing the path to the Group where the scheme for the file are cached. See *SPEC_LOC_ATTR*

Links

Similar to soft links in a file system, a link is an object in a Group that links to another Group or Dataset, either within the same Zarr file or another external Zarr file. Links and reference are not natively supported by Zarr but are implemented in *ZarrIO* in an OS independent fashion using the `zarr_link` reserved attribute (see `__reserve_attribute`) to store a list of dicts serialized as JSON. Each dict (i.e., element) in the list defines a link, with each dict containing the following keys:

- `name` : Name of the link
- `source` : Relative path to the root of the Zarr file containing the linked object. For links pointing to an object within the same Zarr file, the value of source will be `"."`. For external links that point to object in another Zarr file, the value of source will be the path to the other Zarr file relative to the root path of the Zarr file containing the link.
- `path` : Path to the linked object within the Zarr file identified by the `source` key
- `object_id`: Object id of the reference object. May be None in case the referenced object does not have an assigned object_id (e.g., in the case we reference a dataset with a fixed name but without and assigned `data_type` (or `neurodata_type` in the case of NWB).
- `source_object_id`: Object id of the source Zarr file indicated by the `source` key. The `source` should always have an `object_id` (at least if the `source` file is a valid HDMF formatted file).

For example:

```
"zarr_link": [
    {
        "name": "device",
        "source": ".",
        "path": "/general/devices/array",
        "object_id": "f6685427-3919-4e06-b195-ccb7ab42f0fa",
        "source_object_id": "6224bb89-578a-4839-b31c-83f11009292c"
    }
]
```

Table 5: Mapping of links

HDMF Specification Key	Zarr
name	Name of the link
doc	Not mapped; Stored in schema only
target_type	Not mapped. The target type is determined by the type of the target of the link

Hint: In Zarr, attributes are stored in JSON as part of the hidden `.zattrs` file in the folder defining the Group or Dataset.

Hint: In *ZarrIO*, links are written by the `__write_link__()` function, which also uses the helper functions i) `__get_ref()` to construct py:meth:*~hdmf_zarr.utils.ZarrRefernce* and ii) `__add_link__()` to add a link to the Zarr file. `__read_links()` then parses links and also uses the `__resolve_ref()` helper function to resolve the paths stored in links.

Object References

Object reference behave much the same way as Links, with the key difference that they are stored as part of datasets or attributes. This approach allows for storage of large collections of references as values of multi-dimensional arrays (i.e., the data type of the array is a reference type).

Storing object references in Datasets

To identify that a dataset contains object reference, the reserved attribute `zarr_dtype` is set to `'object'` (see also *Reserved attributes*). In this way, we can unambiguously if a dataset stores references that need to be resolved.

Similar to Links, object references are defined via dicts, which are stored as elements of the Dataset. In contrast to links, individual object reference do not have a `name` but are identified by their location (i.e., index) in the dataset. As such, object references only have the `source` with the relative path to the target Zarr file, and the `path` identifying the object within the source Zarr file. The individual object references are defined in the *ZarrIO* as py:class:*~hdmf_zarr.utils.ZarrReference* object created via the `__get_ref()` helper function.

By default, *ZarrIO* uses the `numcodecs.pickles.Pickle` codec to encode object references defined as py:class:*~hdmf_zarr.utils.ZarrReference* dicts in datasets. Users may set the codec used to encode objects in Zarr datasets via the `object_codec_class` parameter of the `__init__()` constructor of *ZarrIO*. E.g., we could use `ZarrIO( ... , object_codec_class=numcodecs.JSON)` to serialize objects using JSON.

Storing object references in Attributes

Object references are stored in a attributes as dicts with the following keys:

- `zarr_dtype` : Indicating the data type for the attribute. For object references `zarr_dtype` is set to `"object"` (or `"region"` for *Region references*)
- `value`: The value of the object references, i.e., here the py:class:*~hdmf_zarr.utils.ZarrReference* dictionary with the `source`, `path`, `object_id`, and `source_object_id` keys defining the object reference, with the definition of the keys being the same as for *Links*.

For example in NWB, the attribute `ElectricalSeries.electrodes.table` would be defined as follows:

```
"table": {
    "value": {
        "path": "/general/extracellular_ephys/electrodes",
        "source": ".",
        "object_id": "f6685427-3919-4e06-b195-ccb7ab42f0fa",
        "source_object_id": "6224bb89-578a-4839-b31c-83f11009292c"
    },
    "zarr_dtype": "object"
}
```

Region references

Region references are similar to object references, but instead of references other Datasets or Groups, region references link to subsets of another Dataset. To identify region references, the reserved attribute `zarr_dtype` is set to `'region'` (see also *Reserved attributes*). In addition to the `source` and `path`, the py:class:*~hdmf_zarr.utils.ZarrReference* object will also need to store the definition of the `region` that is being referenced, e.g., a slice or list on point indices.

Warning: Region references are not yet fully implemented in *ZarrIO*. To implement region references will require updating: 1) py:class:*~hdmf_zarr.utils.ZarrReference* to add a `region` key to support storing the selection for the region, 2) `__get_ref()` to support passing in the region definition to be added to the py:class:*~hdmf_zarr.utils.ZarrReference*, 3) *write_dataset()* already partially implements the required logic for creating region references by checking for `hdmf.build.RegionBuilder` inputs but will likely need updates as well 4) `__read_dataset()` to support reading region references, which may also require updates to `__parse_ref()` and `__resolve_ref()`, and 5) and possibly other parts of *ZarrIO*. 6) The py:class:*~hdmf_zarr.zarr_utils.ContainerZarrRegionDataset* and py:class:*~hdmf_zarr.zarr_utils.ContainerZarrRegionDataset* classes will also need to be finalized to support region references.

dtype mappings

The mappings of data types is as follows

`dtype` spec value	storage type	size
• "float" • "float32"	single precision floating point	32 bit
• "double" • "float64"	double precision floating point	64 bit
• "long" • "int64"	signed 64 bit integer	64 bit
• "int" • "int32"	signed 32 bit integer	32 bit
• "int16"	signed 16 bit integer	16 bit
• "int8"	signed 8 bit integer	8 bit
• "uint32"	unsigned 32 bit integer	32 bit
• "uint16"	unsigned 16 bit integer	16 bit
• "uint8"	unsigned 8 bit integer	8 bit
• "bool"	boolean	8 bit
• "text" • "utf" • "utf8" • "utf-8"	unicode	variable
• "ascii" • "str"	ascii	variable
• "ref" • "reference" • "object"	Reference to another group or dataset. See *Object References*	
• region	Reference to a region of another dataset. See :ref:sec-zarr-storage-references`	
• compound dtype	Compound data type	
• "isodatetime"	ASCII ISO8061 datetime string. For example `2018-09-28T14:43:54.123+02:00`	variable

1.5.3 Caching format specifications

In practice it is useful to cache the specification a file was created with (including extensions) directly in the Zarr file. Caching the specification in the file ensures that users can access the specification directly if necessary without requiring external resources. For the Zarr backend, caching of the schema is implemented as follows.

The `ZarrIO` backend adds the reserved top-level group `/specifications` in which all format specifications (including extensions) are cached. The default name for this group is defined in *DEFAULT_SPEC_LOC_DIR* and caching of specifications is implemented in `ZarrIO.__cache_spec`. The `/specifications` group contains for each specification namespace a subgroup `/specifications/<namespace-name>/<version>` in which the specification for a particular version of a namespace are stored (e.g., `/specifications/core/2.0.1` in the case of the NWB core namespace at version 2.0.1). The actual specification data is then stored as a JSON string in scalar datasets with a binary, variable-length string data type. The specification of the namespace is stored in `/specifications/<namespace-name>/<version>/namespace` while additional source files are stored in `/specifications/<namespace-name>/<version>/<source-filename>`. Here `<source-filename>` refers to the main name of the source-file without file extension (e.g., the core namespace defines `nwb.ephys.yaml` as source which would be stored in `/specifications/core/2.0.1/nwb.ecephys`).

1.5.4 Consolidating Metadata

Zarr allows users to consolidate all metadata for groups and arrays within the given store. By default, every file will consolidate all metadata within into a single *.zmetadata* file, stored in the root group. This reduces the number of read operations when retrieving certain metadata in read mode.

Note: When updating a file, the consolidated metadata will also need to be updated via *zarr.consolidate_metadata(path)* to ensure the consolidated metadata is consistent with the file.

1.6 Integrating New Zarr Data Stores

ZarrIO by default uses the Zarr DirectoryStore via the `zarr.convenience.open()` method. *ZarrIO* further supports all stores listed in *SUPPORTED_ZARR_STORES*. Users can specify a particular store using the `path` parameter when creating a new *ZarrIO* instance. This document discusses key steps towards integrating other data stores available for Zarr with *ZarrIO*.

1.6.1 Updating ZarrIO

1. Import and add the new storage class to the *SUPPORTED_ZARR_STORES*. This will in turn allow instances of your new storage class to be passed as a `path` parameter to `__init__()` and *load_namespaces()* and pass `docval()` validation for these functions.
    - If your store has a `.path` property then the `source` property will be set accordingly in `__init__` in *ZarrIO*, otherwise `__init__` may need to be updated to set a correct `source` (used, e.g., to define links).
2. Update *open()* and *close()* as necessary.
3. Depending on the type of data store, it may also be necessary to update the handling of links and references in *ZarrIO*. In principle, reading and writing of links should not need to change, however, in particular the `__resolve_ref()` and *get_builder_exists_on_disk()* method may need to be updated to ensure references are opened correctly on read for files stored with your new store. The `__get_ref()` function may also need to be updated, in particular in case the links to your store also modify the storage schema for links (e.g., if you need to store additional metadata in order to resolve links to your store).

1.6.2 Updating NWBZarrIO

In most cases we should not need to update *NWBZarrIO* as it inherits directly from *ZarrIO*. However, in particular if the interface for `__init__` has changed for *ZarrIO*, then we may also need to modify *NWBZarrIO* accordingly.

1.6.3 Updating Unit Tests

Much of the core test harness of `hdmf_zarr` is modularized to simplify running existing tests with new storage backends. In this way, we can quickly create a collection of common tests for new backends, and new test cases added to the test suite can be run with all backends. The relevant test class are located in the /tests/unit directory of the hdmf_zarr repository.

test_zarrio.py

base_tests_zarrio.py provides a collection of base classes that define common test cases to test basic functionality of *ZarrIO*. Using these base classes, the test_zarrio.py module then implements concrete tests for various backends. To create tests for a new data store, we need to add the following main classes (while `<MyStore>` in the code below would need to be replaced with the class name of the new data store):

1. **Create tests for new data store:** Add the following main classes (while `<MyStore>` in the code below would need to be replaces with the class name of the new data store):

```
########################################
#  <MyStore> tests
########################################
class TestZarrWriter<MyStore>(BaseTestZarrWriter):
    """Test writing of builder with Zarr using a custom <MyStore>"""
    def setUp(self):
        super().setUp()
        self.store = <MyStore>()
        self.store_path = self.store.path


class TestZarrWriteUnit<MyStore>(BaseTestZarrWriteUnit):
    """Unit test for individual write functions using a custom <MyStore>"""
    def setUp(self):
        super().setUp()
        self.store = <MyStore>()
        self.store_path = self.store.path


class TestExportZarrToZarr<MyStore>(BaseTestExportZarrToZarr):
    """Test exporting Zarr to Zarr using <MyStore>."""
    def setUp(self):
        super().setUp()
        self.stores = [<MyStore>() for i in range(len(self.store_path))]
        self.store_paths = [s.path for s in self.stores]
```

2. **Update ``base_tests_zarrio.reopen_store``** If our new data store cannot be reused after it has been closed via *close()*, then update the method to either reopen or create a new equivalent data store that can be used for read. The function is used in tests that write data, then close the ZarrIO, and create a new ZarrIO to read and validate the data.

3. **Run and update tests** Depending on your data store, some test cases in `BaseTestZarrWriter`, `BaseTestZarrWriteUnit` or `BaseTestExportZarrToZarr` may need to be updated to correctly work with our data store. Simply run the test suite to see if any cases are failing to see whether the `setUp` in your test classes or any specific test cases may need to be updated.

test_io_convert.py

test_io_convert.py uses a collection of mixin classes to define custom test classes to test export from one IO backend to another. As such, the test cases here typically first write to one target and then export to another target and then compare that the data between the two files is consistent.

1. **Update ``MixinTestHDF5ToZarr``, ``MixinTestZarrToZarr``, and ``MixinTestZarrToZarr``** mixin classes to add the new backend to the `WRITE_PATHS` (if Zarr is the initial write target) and/or `EXPORT_PATHS` (if Zarr is the export target) variables to define our store as a write or export store for *ZarrIO*, respectively. Once we have added our new store as write/export targets to these mixins, all test cases defined in the module will be run with our new backend. Specifically, we here commonly need to add an instance of our new data store to:
   - `MixinTestHDF5ToZarr.EXPORT_PATHS`
   - `MixinTestZarrToHDF5.WRITE_PATHS`
   - `MixinTestZarrToZarr.WRITE_PATHS` and `MixinTestZarrToZarr.EXPORT_PATHS`
2. **Update tests and ZarrIO as necessary** Run the test suite and fix any identified issues.

1.7 hdmf_zarr package

1.7.1 Submodules

hdmf_zarr.backend module

Module with the Zarr-based I/O-backend for HDMF

hdmf_zarr.backend.**ROOT_NAME = 'root'**

Name of the root builder for read/write

hdmf_zarr.backend.**SPEC_LOC_ATTR = '.specloc'**

Reserved attribute storing the path to the Group where the schema for the file are cached

hdmf_zarr.backend.**DEFAULT_SPEC_LOC_DIR = 'specifications'**

Default name of the group where specifications should be cached

hdmf_zarr.backend.**SUPPORTED_ZARR_STORES = (<class 'zarr.storage.DirectoryStore'>, <class 'zarr.storage.TempStore'>, <class 'zarr.storage.NestedDirectoryStore'>)**

Tuple listing all Zarr storage backends supported by ZarrIO

class hdmf_zarr.backend.**ZarrIO**(*path*, *mode*, *manager=None*, *synchronizer=None*, *object_codec_class=None*, *storage_options=None*)

Bases: HDMFIO

Parameters

- **path** (str or DirectoryStore or TempStore or NestedDirectoryStore) – the path to the Zarr file or a supported Zarr store

- **mode** (str) – the mode to open the Zarr file with, one of ("w", "r", "r+", "a", "w-") the mode r- is used to force open without consolidated metadata in read only mode.
- **manager** (BuildManager) – the BuildManager to use for I/O
- **synchronizer** (ProcessSynchronizer or ThreadSynchronizer or bool) – Zarr synchronizer to use for parallel I/O. If set to True a ProcessSynchronizer is used.
- **object_codec_class** (*None*) – Set the numcodec object codec class to be used to encode objects.Use numcodecs.pickles.Pickle by default.
- **storage_options** (dict) – Zarr storage options to read remote folders

static can_read(*path*)

Determines whether a given path is readable by this HDMFIO class

property file

The Zarr zarr.hierarchy.Group (or zarr.core.Array) opened by the backend. May be None in case open has not been called yet, e.g., if no data has been read or written yet via this instance.

property path

The path to the Zarr file as set by the user

property abspath

The absolute path to the Zarr file

property synchronizer

property object_codec_class

open()

Open the Zarr file

close()

Close the Zarr file

is_remote()

Return True if the file is remote, False otherwise

classmethod load_namespaces(*namespace_catalog*, *path*, *namespaces=None*)

Load cached namespaces from a file.

Parameters

- **namespace_catalog** (NamespaceCatalog or TypeMap) – the NamespaceCatalog or TypeMap to load namespaces into
- **path** (str or DirectoryStore or TempStore or NestedDirectoryStore) – the path to the Zarr file or a supported Zarr store
- **namespaces** (list) – the namespaces to load

write(*container*, *cache_spec=True*, *link_data=True*, *exhaust_dci=True*, *number_of_jobs=1*, *max_threads_per_process=None*, *multiprocessing_context=None*, *consolidate_metadata=True*)

Overwrite the write method to add support for caching the specification and parallelization.

Parameters

- **container** (Container) – the Container object to write
- **cache_spec** (bool) – cache specification to file
- **link_data** (bool) – If not specified otherwise link (True) or copy (False) Datasets

- **exhaust_dci** (bool) – exhaust DataChunkIterators one at a time. If False, add them to the internal queue self.__dci_queue and exhaust them concurrently at the end
- **number_of_jobs** (int) – Number of jobs to use in parallel during write (only works with GenericDataChunkIterator-wrapped datasets).
- **max_threads_per_process** (int) – Limits the number of threads used by each process. The default is None (no limits).
- **multiprocessing_context** (str) – Context for multiprocessing. It can be None (default), 'fork' or 'spawn'. Note that 'fork' is only available on UNIX systems (not Windows).
- **consolidate_metadata** (bool) – Consolidate metadata into a single .zmetadata file in the root group to accelerate read.

export(*src_io*, *container=None*, *write_args={}*, *clear_cache=False*, *cache_spec=True*, *number_of_jobs=1*, *max_threads_per_process=None*, *multiprocessing_context=None*)

Export data read from a file from any backend to Zarr.

See hdmf.backends.io.HDMFIO.export() for more details.

Parameters

- **src_io** (*HDMFIO*) – the HDMFIO object for reading the data to export
- **container** (Container) – the Container object to export. If None, then the entire contents of the HDMFIO object will be exported
- **write_args** (dict) – arguments to pass to *write_builder()*
- **clear_cache** (bool) – whether to clear the build manager cache
- **cache_spec** (bool) – whether to cache the specification to file
- **number_of_jobs** (int) – Number of jobs to use in parallel during write (only works with GenericDataChunkIterator-wrapped datasets).
- **max_threads_per_process** (int) – Limits the number of threads used by each process. The default is None (no limits).
- **multiprocessing_context** (str) – Context for multiprocessing. It can be None (default), 'fork' or 'spawn'. Note that 'fork' is only available on UNIX systems (not Windows).

get_written(*builder*, *check_on_disk=False*)

Return True if this builder has been written to (or read from) disk by this IO object, False otherwise.

Parameters

- **builder** (*Builder*) – Builder object to get the written flag for
- **check_on_disk** (*bool*) – Check that the builder has been physically written to disk not just flagged as written by this I/O backend

Returns

True if the builder is found in self._written_builders using the builder ID, False otherwise. If check_on_disk is enabled then the function cals get_builder_exists_on_disk in addition to verify that the builder has indeed been written to disk.

get_builder_exists_on_disk(*builder*)

Convenience function to check whether a given builder exists on disk in this Zarr file.

Parameters

builder (Builder) – The builder of interest

get_builder_disk_path(*builder*, *filepath=None*)

Parameters

- **builder** (Builder) – The builder of interest
- **filepath** (str) – The path to the Zarr file or None for this file

write_builder(*builder*, *link_data=True*, *exhaust_dci=True*, *export_source=None*, *consolidate_metadata=True*)

Write a builder to disk.

Parameters

- **builder** (GroupBuilder) – the GroupBuilder object representing the NWBFile
- **link_data** (bool) – If not specified otherwise link (True) or copy (False) Zarr Datasets
- **exhaust_dci** (bool) – Exhaust DataChunkIterators one at a time. If False, add them to the internal queue self.__dci_queue and exhaust them concurrently at the end
- **export_source** (str) – The source of the builders when exporting
- **consolidate_metadata** (bool) – Consolidate metadata into a single .zmetadata file in the root group to accelerate read.

write_group(*parent*, *builder*, *link_data=True*, *exhaust_dci=True*, *export_source=None*)

Write a GroupBuider to file

Parameters

- **parent** (Group) – the parent Zarr object
- **builder** (GroupBuilder) – the GroupBuilder to write
- **link_data** (bool) – If not specified otherwise link (True) or copy (False) Zarr Datasets
- **exhaust_dci** (bool) – exhaust DataChunkIterators one at a time. If False, add them to the internal queue self.__dci_queue and exhaust them concurrently at the end
- **export_source** (str) – The source of the builders when exporting

Returns
the Group that was created

Return type
Group

write_attributes(*obj*, *attributes*, *export_source=None*)

Set (i.e., write) the attributes on a given Zarr Group or Array.

Parameters

- **obj** (Group or Array) – the Zarr object to add attributes to
- **attributes** (dict) – a dict containing the attributes on the Group or Dataset, indexed by attribute name
- **export_source** (str) – The source of the builders when exporting

static get_zarr_paths(*zarr_object*)

For a Zarr object find 1) the path to the main zarr file it is in and 2) the path to the object within the file :param zarr_object: Object for which we are looking up the path :type zarr_object: Zarr Group or Array :return: Tuple of two string with: 1) path of the Zarr file and 2) full path within the zarr file to the object

static get_zarr_parent_path(*zarr_object*)

Get the location of the parent of a zarr_object within the file :param zarr_object: Object for which we are looking up the path :type zarr_object: Zarr Group or Array :return: String with the path

static is_zarr_file(*path*)

Check if the given path defines a Zarr file :param path: Full path to main directory :return: Bool

resolve_ref(*zarr_ref*)

Get the full path to the object linked to by the zarr reference

The function only constructs the links to the targe object, but it does not check if the object exists

Parameters
zarr_ref – Dict with *source* and *path* keys or a *ZarrReference* object

Returns

1) name of the target object
2) the target zarr object within the target file

write_link(*parent*, *builder*)

Parameters

- **parent** (Group) – the parent Zarr object
- **builder** (LinkBuilder) – the LinkBuilder to write

write_dataset(*parent*, *builder*, *link_data=True*, *exhaust_dci=True*, *force_data=None*, *export_source=None*)

Parameters

- **parent** (Group) – the parent Zarr object
- **builder** (DatasetBuilder) – the DatasetBuilder to write
- **link_data** (bool) – If not specified otherwise link (True) or copy (False) Zarr Datasets
- **exhaust_dci** (bool) – exhaust DataChunkIterators one at a time. If False, add them to the internal queue self.__dci_queue and exhaust them concurrently at the end
- **force_data** (*None*) – Used internally to force the data being used when we have to load the data
- **export_source** (str) – The source of the builders when exporting

Returns
the Zarr array that was created

Return type
Array

classmethod get_type(*data*)

read_builder()

Returns
a GroupBuilder representing the NWB Dataset

Return type
GroupBuilder

get_container(*zarr_obj*)

Get the container for the corresponding Zarr Group or Dataset

raises ValueError
When no builder has been constructed yet for the given h5py object

Parameters
zarr_obj (Array or Group) – the Zarr object to the corresponding Container/Data object for

get_builder(*zarr_obj*)

Get the builder for the corresponding Group or Dataset

raises ValueError
When no builder has been constructed

Parameters
zarr_obj (Array or Group) – the Zarr object to the corresponding Builder object for

hdmf_zarr.nwb module

Module with Zarr backend for NWB for integration with PyNWB

class hdmf_zarr.nwb.**NWBZarrIO**(*path*, *mode*, *manager=None*, *synchronizer=None*, *object_codec_class=None*, *storage_options=None*, *load_namespaces=False*, *extensions=None*)

Bases: *ZarrIO*

IO backend for PyNWB for writing NWB files

This class is similar to the NWBHDF5IO class in PyNWB. The main purpose of this class is to perform default setup for BuildManager, loading or namespaces etc., in the context of the NWB format.

Parameters

- **path** (str or DirectoryStore or TempStore or NestedDirectoryStore) – the path to the Zarr file or a supported Zarr store
- **mode** (str) – the mode to open the Zarr file with, one of ("w", "r", "r+", "a", "w-") the mode r- is used to force open without consolidated metadata in read only mode.
- **manager** (BuildManager) – the BuildManager to use for I/O
- **synchronizer** (ProcessSynchronizer or ThreadSynchronizer or bool) – Zarr synchronizer to use for parallel I/O. If set to True a ProcessSynchronizer is used.
- **object_codec_class** (*None*) – Set the numcodec object codec class to be used to encode objects.Use numcodecs.pickles.Pickle by default.
- **storage_options** (dict) – Zarr storage options to read remote folders
- **load_namespaces** (bool) – whether or not to load cached namespaces from given path - not applicable in write mode
- **extensions** (str or TypeMap or list) – a path to a namespace, a TypeMap, or a list consisting paths to namespaces and TypeMaps

export(*src_io*, *nwbfile=None*, *write_args={}*)

Parameters

- **src_io** (HDMFIO) – the HDMFIO object for reading the data to export

- **nwbfile** (*NWBFile*) – the NWBFile object to export. If None, then the entire contents of src_io will be exported
- **write_args** (dict) – arguments to pass to write_builder()

hdmf_zarr.utils module

Collection of utility I/O classes for the ZarrIO backend store.

class hdmf_zarr.utils.**ZarrIODataChunkIteratorQueue**(*number_of_jobs: int = 1*, *max_threads_per_process: None | int = None*, *multiprocessing_context: None | Literal['fork', 'spawn'] = None*)

Bases: deque

Helper class used by ZarrIO to manage the write for DataChunkIterators Each queue element must be a tuple of two elements: 1) the dataset to write to and 2) the AbstractDataChunkIterator with the data :param number_of_jobs: The number of jobs used to write the datasets. The default is 1. :type number_of_jobs: integer :param max_threads_per_process: Limits the number of threads used by each process. The default is None (no limits). :type max_threads_per_process: integer or None :param multiprocessing_context: Context for multiprocessing. It can be None (default), "fork" or "spawn". Note that "fork" is only available on UNIX systems (not Windows). :type multiprocessing_context: string or None

exhaust_queue()

Read and write from any queued DataChunkIterators.

Operates in a round-robin fashion for a single job. Operates on a single dataset at a time with multiple jobs.

append(*dataset*, *data*)

Append a value to the queue :param dataset: The dataset where the DataChunkIterator is written to :type dataset: Zarr array :param data: DataChunkIterator with the data to be written :type data: AbstractDataChunkIterator

static initializer_wrapper(*operation_to_run: callable*, *process_initialization: callable*, *initialization_arguments: Iterable*, *max_threads_per_process: None | int = None*)

Needed as a part of a bug fix with cloud memory leaks discovered by SpikeInterface team.

Recommended fix is to have global wrappers for the working initializer that limits the threads used per process.

static function_wrapper(*args: Tuple[str, str, AbstractDataChunkIterator, Tuple[slice, ...]]*)

Needed as a part of a bug fix with cloud memory leaks discovered by SpikeInterface team.

Recommended fix is to have a global wrapper for the executor.map level.

class hdmf_zarr.utils.**ZarrSpecWriter**(*group*)

Bases: SpecWriter

Class used to write format specs to Zarr

Parameters
group (Group) – the Zarr file to write specs to

static stringify(*spec*)

Converts a spec into a JSON string to write to a dataset

write_spec(*spec*, *path*)

Write a spec to the given path

write_namespace(*namespace*, *path*)

Write a namespace to the given path

class hdmf_zarr.utils.**ZarrSpecReader**(*group*, *source='.'*)

Bases: SpecReader

Class to read format specs from Zarr

Parameters

- **group** (Group) – the Zarr file to read specs from
- **source** (str) – the path spec files are relative to

read_spec(*spec_path*)

Read a spec from the given path

read_namespace(*ns_path*)

Read a namespace from the given path

class hdmf_zarr.utils.**ZarrDataIO**(*data*, *chunks=None*, *fillvalue=None*, *compressor=None*, *filters=None*, *link_data=False*)

Bases: DataIO

Wrap data arrays for write via ZarrIO to customize I/O behavior, such as compression and chunking for data arrays.

Parameters

- **data** (ndarray or list or tuple or Array or Iterable) – the data to be written. NOTE: If an zarr.Array is used, all other settings but link_data will be ignored as the dataset will either be linked to or copied as is in ZarrIO.
- **chunks** (list or tuple) – Chunk shape
- **fillvalue** (*None*) – Value to be returned when reading uninitialized parts of the dataset
- **compressor** (Codec or bool) – Zarr compressor filter to be used. Set to True to use Zarr default.Set to False to disable compression)
- **filters** (list or tuple) – One or more Zarr-supported codecs used to transform data prior to compression.
- **link_data** (bool) – If data is an zarr.Array should it be linked to or copied. NOTE: This parameter is only allowed if data is an zarr.Array

property link_data: bool

Only applies to zarr.Array type data

Type

Bool indicating should it be linked to or copied. NOTE

property io_settings: dict

Dict with the io settings to use

static from_h5py_dataset(*h5dataset*, ***kwargs*)

Factory method to create a ZarrDataIO instance from a h5py.Dataset. The ZarrDataIO object wraps the h5py.Dataset and the io filter settings are inferred from filters used in h5py such that the options in Zarr match (if possible) the options used in HDF5.

Parameters

- **dataset** (*h5py.Dataset*) – h5py.Dataset object that should be wrapped
- **kwargs** – Other keyword arguments to pass to ZarrDataIO.__init__

Returns

ZarrDataIO object wrapping the dataset

static hdf5_to_zarr_filters(*h5dataset*) → list

From the given h5py.Dataset infer the corresponding filters to use in Zarr

static is_h5py_dataset(*obj*)

Check if the object is an instance of h5py.Dataset without requiring import of h5py

class hdmf_zarr.utils.**ZarrReference**(*source=None*, *path=None*, *object_id=None*, *source_object_id=None*)

Bases: dict

Data structure to describe a reference to another container used with the ZarrIO backend

Parameters

- **source** (str) – Source of referenced object. Usually the relative path to the Zarr file containing the referenced object
- **path** (str) – Path of referenced object within the source
- **object_id** (str) – Object_id of the referenced object (if available)
- **source_object_id** (str) – Object_id of the source (should always be available)

property source: str

property path: str

property object_id: str

property source_object_id: str

hdmf_zarr.zarr_utils module

Utilities for the Zarr I/O backend, e.g., for wrapping Zarr arrays on read, wrapping arrays for configuring write, or writing the spec among others

class hdmf_zarr.zarr_utils.**ZarrDataset**(*dataset*, *io*)

Bases: HDMFDataset

Extension of HDMFDataset to add Zarr compatibility

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

property io

property shape

class hdmf_zarr.zarr_utils.**DatasetOfReferences**(*dataset*, *io*)

Bases: *ZarrDataset*, ReferenceResolver

An extension of the base ReferenceResolver class to add more abstract methods for subclasses that will read Zarr references

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

abstract get_object(*zarr_obj*)

A class that maps an Zarr object to a Builder or Container

invert()

Return an object that defers reference resolution but in the opposite direction.

class hdmf_zarr.zarr_utils.**BuilderResolverMixin**

Bases: BuilderResolver

A mixin for adding to Zarr reference-resolving types the get_object method that returns Builders

get_object(*zarr_obj*)

A class that maps an Zarr object to a Builder

class hdmf_zarr.zarr_utils.**ContainerResolverMixin**

Bases: ContainerResolver

A mixin for adding to Zarr reference-resolvinAbstractZarrReferenceDatasetg types the get_object method that returns Containers

get_object(*zarr_obj*)

A class that maps an Zarr object to a Container

class hdmf_zarr.zarr_utils.**AbstractZarrTableDataset**(*dataset*, *io*, *types*)

Bases: *DatasetOfReferences*

Extension of DatasetOfReferences to serve as the base class for resolving Zarr references in compound datasets to either Builders and Containers.

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset
- **types** (list or tuple) – the list/tuple of reference types

property types

property dtype

__getitem__(*arg*)

resolve(*manager*)

class hdmf_zarr.zarr_utils.**AbstractZarrReferenceDataset**(*dataset*, *io*)

Bases: *DatasetOfReferences*

Extension of DatasetOfReferences to serve as the base class for resolving Zarr references in datasets to either Builders and Containers.

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

__getitem__(*arg*)

property dtype

class hdmf_zarr.zarr_utils.**AbstractZarrRegionDataset**(*dataset*, *io*)

Bases: *AbstractZarrReferenceDataset*

Extension of DatasetOfReferences to serve as the base class for resolving Zarr references in datasets to either Builders and Containers.

Note: Region References are not yet supported.

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

__getitem__(*arg*)

property dtype

class hdmf_zarr.zarr_utils.**ContainerZarrTableDataset**(*dataset*, *io*, *types*)

Bases: *ContainerResolverMixin*, *AbstractZarrTableDataset*

A reference-resolving dataset for resolving references inside tables (i.e. compound dtypes) that returns resolved references as Containers

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset
- **types** (list or tuple) – the list/tuple of reference types

classmethod get_inverse_class()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

class hdmf_zarr.zarr_utils.**BuilderZarrTableDataset**(*dataset*, *io*, *types*)

Bases: *BuilderResolverMixin*, *AbstractZarrTableDataset*

A reference-resolving dataset for resolving references inside tables (i.e. compound dtypes) that returns resolved references as Builders

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset
- **types** (list or tuple) – the list/tuple of reference types

classmethod get_inverse_class()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

class hdmf_zarr.zarr_utils.**ContainerZarrReferenceDataset**(*dataset*, *io*)

Bases: *ContainerResolverMixin*, *AbstractZarrReferenceDataset*

A reference-resolving dataset for resolving object references that returns resolved references as Containers

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

classmethod get_inverse_class()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

class hdmf_zarr.zarr_utils.**BuilderZarrReferenceDataset**(*dataset*, *io*)

Bases: *BuilderResolverMixin*, *AbstractZarrReferenceDataset*

A reference-resolving dataset for resolving object references that returns resolved references as Builders

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

classmethod get_inverse_class()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

class hdmf_zarr.zarr_utils.**ContainerZarrRegionDataset**(*dataset*, *io*)

Bases: *ContainerResolverMixin*, *AbstractZarrRegionDataset*

A reference-resolving dataset for resolving region references that returns resolved references as Containers

Note: Region References are not yet supported.

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

classmethod get_inverse_class()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

class hdmf_zarr.zarr_utils.**BuilderZarrRegionDataset**(*dataset*, *io*)

Bases: *BuilderResolverMixin*, *AbstractZarrRegionDataset*

A reference-resolving dataset for resolving region references that returns resolved references as Builders.

Note: Region References are not yet supported.

Parameters

- **dataset** (ndarray or Array or Array) – the Zarr file lazily evaluate
- **io** (ZarrIO) – the IO object that was used to read the underlying dataset

classmethod **get_inverse_class**()

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get_inverse_class should return a class that subclasses ContainerResolver.

ContainerResolver.get_inverse_class should return a class that subclasses BuilderResolver.

1.7.2 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W

Z