Digital Metadata 2.0

William Rideout and Juha Vierinen

Overview

Digital Metadata is a *disk storage* and archival format for time-indexed metadata. The design goals are the following:

- The format and the programming language interfaces are as simple as possible.
- Data files should be self-contained, i.e., interpretation of core properties of a file should not depend on any other file.
- Data files should have a logical namespace structure which, which allows usage of heterogeneous data in a unified manner.
- Directory and file naming conventions should allow efficient file system storage and access over years of stored data.
- Data should be in a format that is cross-platform, i.e., easy to read with different programming languages on different computing hardware.

At the top level, this module stores all metadata indexed to a sample, where the sample is the unix time times the samples per second. At every given sample, metadata is written to a user defined set of field names (strings). These field names do not change for a given metadata channel/directory. When new metadata is to be written, the user passes in two arguments: 1) a sample, and 2) a data dictionary, where the keys match the initial field names, and the values can be any arbitrary numpy object that h5py can convert to a dataset, or another dictionary that contains keys as string and values that are either more dictionaries, or arbitrary numpy objects that h5py can convert to a dataset. Any number of levels of dictionaries are allowed, but all leaf values must be numpy objects. This provides great flexibility in what can be stored - from simple numbers, to complex numpy arrays. The write API also allows array of samples to be passed in with one call.

Digital Metadata 2.0 differs from Digital Metadata 1.0 in that multiple levels of dictionaries can be passed in. Each dictionary passed in is converted to an Hdf5 group, and all numpy objects passed in are converted to Hdf5 datasets. Hdf5 files created with Digital Metadata 1.0 can always be read with Digital Metadata 2.0, but multi-level data created with Digital Metadata 2.0 will cause an exception if read with Digital Metadata 2.0.

To make data access as fast as possible, we store metadata in hdf5 files with a naming convention that can be derived from the requested sample range, so that no IO is required to determine exactly what files will need to be read. When a user initializes a new digital metadata to be written to, they set a subdirectory cadence in seconds, and a file cadence in seconds. The write API enforces the rule that each subdirectory name must be associated with a timestamp such that timestamp % subdirectory_cadence == 0. For example, if subdirectory cadence is 3600, then all subdirectories will be in the form YYYY-MM-DDT00:00:00. Likewise, the file cadence must evenly divide into the subdirectory cadence. For example, if the subdirectory cadence was 3600, a file cadence of 5 would be legal but a file cadence of 7 seconds would not. The file naming convention is <name>@<timestamp>.h5, where name is also passed into the init method. The resulting hdf5 metadata files may not be of equal size unless the use always writes metadata with the same cadence. Also, not all possible file names may exist if the writer did not write any samples in its range. The read API handles this by returning requested data as an OrderedDict with keys = data from all samples found in that range (may be zero).

In this way reading the metadata is unaffected by the size of the total existing metadata, but may be related to the total span of metadata requested. The parameters associated with a given metadata channel (subdirectory cadence, etc) are stored at the top level metadata_dir in a file called metadata.h5.

The read and write API are implemented via a python module called digital_metadata.py located in subversion under RapidSVN/prototypes/digital_rf/trunk/source/. A Matlab reader is available. For the Matlab reader, containers.Map objects replace python dictionaries.

The design of Digital Metadata was the inspiration for Digital RF 2.0.

Python write API

```
class write_digital_metadata:
```

```
    """write_digital_metadata is the class used to write digital_metadata
    """
    def __init__(self, metadata_dir, subdirectory_cadence_seconds, file_cadence_seconds,
                 samples_per_second_numerator, samples_per_second_denominator, file_name):
        """__init__ creates an object to write a single channel of digital_metadata

        Inputs:
            metadata_dir - the top level directory where the metadata will be written. Must already exist.
            subdirectory_cadence_seconds - the integer number of seconds of metadata to store in each
subdirectory. This API will enforce the rule that the timestamp of any subdirectory is
n*subdirectory_cadence_seconds
            file_cadence_seconds - the integer number of seconds to store in each file. Note that N files must
span exactly subdirectory_cadence_seconds, which implies subdirectory_cadence_seconds %
file_cadence_seconds == 0 This API will enforce the rule that file name timestamps are in the list:
range(subdirectory_timestamp, subdirectory_timestamp+subdirectory_cadence_seconds,
file_cadence_seconds)
            samples_per_second_numerator - samples per second numerator (long). Used since
digital_metadata uses samples since 1970 in all indexing.
            samples_per_second_denominator - samples per second denominator (long). Used since
digital_metadata uses samples since 1970 in all indexing.
            file_name - prefix for metadata file names.

        All inputs are saved as class attributes.  Also self._fields to None if no data yet, or data does exist, then
reads  "fields" dataset from at the top level of metadata.h5.  Then self._fields is set to a list of keys (dataset
names)
        """



    def write(self, samples, data_dict):
        """write is the main method used to write new metadata to a metadata channel.

        Inputs:
            samples - A single sample (long) or a list or numpy vector of samples, length = length data value
lists. A sample is the unix time times the sample rate as a long.

            data_dict - a dictionary representing the metadata to write.  keys are the field names, values can be
1) a list of numpy objects or 2) a vector numpy array of length samples, or 3) a single numpy object if
samples is length 1, or 4) another dictionary whose keys are string that are valid Group names and leaf
values are one of the three types above.  Length of list or vector must equal length of samples, and if a
single numpy object, then length of samples must be one. Data must always have the same names for each
call to write.

        """
```

Example Digital Metadata write script

```
"""test of digital_rf_metadata.write_digital_metadata

$Id: test_write_digital_metadata.py 904 2015-12-30 18:39:35Z brideout $
"""
# standard python imports
import os

# third party imports
import numpy

# Millstone imports
import digital_metadata

metadata_dir = '/tmp/test_metadata'
subdirectory_cadence_seconds = 3600
file_cadence_seconds = 60
samples_per_second_numerator = 10
samples_per_second_denominator = 9
file_name = 'rideout'
stime = 1447082580

os.system('mkdir %s' % (metadata_dir))
os.system('rm -r %s/*' % (metadata_dir))

obj = digital_metadata.write_digital_metadata(metadata_dir, subdirectory_cadence_seconds,
file_cadence_seconds, samples_per_second_numerator, samples_per_second_denominator,
 file_name)
print('first create okay')

data_dict = {}
start_idx = long(stime*samples_per_second)
idx_arr = numpy.arange(70, dtype=numpy.int64) + start_idx

int_data = numpy.arange(70, dtype=numpy.int32)
data_dict['int_data'] = int_data
float_data = numpy.arange(70, dtype=numpy.float32)
data_dict['float_data'] = float_data
complex_data = numpy.arange(70, dtype=numpy.complex64)
data_dict['complex_data'] = complex_data
single_int = 5
data_dict['single_int'] = numpy.int32(single_int)
single_float = 6.0
data_dict['single_float'] = numpy.float64(single_float)
```

```
single_complex = 7.0 + 8.0j
data_dict['single_complex'] = numpy.complex(single_complex)

# complex python object
n = numpy.ones((10,4), dtype=numpy.float64)
n[5,:] = 17.0
data_dict['numpy_obj'] = [n for i in range(70)]


obj.write(idx_arr, data_dict)
print('first write_metadata okay')

# write same data again after incrementating inx
idx_arr += 70
```

Python Read API

```
class read_digital_metadata:
    """read_digital_metadata is the class used to access digital_metadata
    """

    def __init__(self, metadata_dir, accept_empty=False):
        """__init__ creates needed class attributes by reading <metadata_dir>/metadata.h5

        If accept_empty is False (the default), raises IOError if metadata.h5 not found or cannot be parsed
        """


    def get_bounds(self):
        """get_bounds returns a tuple of first sample, last sample for this metadata. A sample
            is the unix time times the sample rate as a long.

        Raises IOError if no data
        """


    def get_fields(self):
        """get_fields returns a list of all the field names available in this metadata
        """


    def get_samples_per_second(self):
        """returns the samples per second as a float for this metadata
        """
```

```
    def read(self, sample0, sample1, columns=None):
        """read returns a OrderedDict representing the requested metadata.

        Inputs:
            sample0 - first sample for which to return metadata
            sample1 - last sample for which to return metadata. A sample
                is the unix time times the sample rate as a long.
            columns - either a single string representing one column of metadata to return, or a
                list of column names to return.  If None (the default), return all columns available in files.

        Returns:
            a collections.OrderedDict with ordered keys = all samples found for which there is metadata.
                Value is a simple value if only a single column requested of whatever type that column had in the
                the metadata file.  It multiple columns requested, returns a standard dictionary with keys = column
names,
                values = value for that sample and column name as found in metadata file.
        """


    def read_latest(self):
        """read_latest simply calls read for all columns with samples near the last sample time available
        as returned by get_bounds.  Returns dict with only the largest sample as key

        Returns: dict with key = last sample, value is a dict with keys=column names, values = numpy values
        """
```

Python read example

```
"""test of digital_metadata.read_metadata

$Id: test_read_digital_metadata.py 905 2015-12-30 19:05:34Z brideout $
"""
# third party imports
import numpy

# Millstone imports
import digital_metadata

metadata_dir = '/tmp/test_metadata'
stime = 1447082580

try:
    obj = digital_metadata.read_digital_metadata(metadata_dir)
```

```
except:
    print('Be sure you run test_write_digital_metadata.py before running this test code.')
    raise
print('init okay')

first_sample, last_sample = obj.get_bounds()
print('bounds are %i to %i' % (first_sample, last_sample))

fields = obj.get_fields()
print('Available fields are <%s>' % (str(fields)))

print('first read - just get one column simple_complex')
data_dict = obj.read(stime, stime+2, 'single_complex')
for key in data_dict.keys():
    print((key, data_dict[key]))

print('second read - just 2 columns: simple_complex and numpy_obj')
data_dict = obj.read(stime, stime+2, ('single_complex', 'numpy_obj'))
for key in data_dict.keys():
    print((key, data_dict[key]))

print('third read - get all columns')
data_dict = obj.read(stime, stime+2)
for key in data_dict.keys():
    print((key, data_dict[key]))

print('just get latest metadata')
latest_meta = obj.read_latest()
print(latest_meta)

print('test of get_samples_per_second')
sps = obj.get_samples_per_second()
print(sps)
```

Matlab Read API

```
classdef DigitalMetadataReader
    % class DigitalMetadataReader allows easy read access to  Digital
    % metadata
    %   See testDigitalMetadataReader.m for usage, or run <doc DigitalMetadataReader>
    %
    % $Id: DigitalMetadataReader.m 1272 2017-02-21 19:23:34Z brideout $

    properties
        metadataDir % a string of metadata directory
```

```
        subdirectory_cadence_seconds % a number of seconds per directory
        file_cadence_seconds % number of seconds per filereader
        samples_per_second_numerator % samples per second numerator of metadata
        samples_per_second_denominator % samples per second denominator of metadata
        samples_per_second % float samples per second of metadata as determined by numerator and
denominator
        file_name % file naming prefix
        fields % a char array of field names in metadata
        dir_glob % string to glob for directories
    end % end properties

    methods
        function reader = DigitalMetadataReader(metadataDir)
            % DigitalMetadataReader is the contructor for this class.
            % Inputs - metadataDir - a string of the path to the metadata


        function [lower_sample, upper_sample] = get_bounds(obj)
            % get_bounds returns a tuple of first sample, last sample for this metadata. A sample
            % is the unix time times the sample rate as a integer.


        function fields = get_fields(obj)

        function fields = get_samples_per_second_numerator(obj)r

        function fields = get_samples_per_second_denominator(obj)

        function fields = get_samples_per_second(obj)

        function fields = get_subdirectory_cadence_seconds(obj)

        function fields = get_file_cadence_seconds(obj)

        function fields = get_file_name(obj)

        function data_map = read(obj, sample0, sample1, field)
            % read returns a containers.Map() object containing key=sample as uint64,
            % value = data at that sample for field, or another containers.Map()
            % with its keys = names,values = data or more containers.Maps -
            % no limit to levels
            %
            %   Inputs:
            %       sample0 - first sample for which to return metadata
            %       sample1 - last sample for which to return metadata. A sample
            %           is the unix time times the sample rate as a long.
            %       field - the valid field you which to get
```

Matlab read example

```
% example usage of DigitalMetadataReader.m
% Requires python test_write_digital_metadata.py be run first to create test data
% $Id: testDigitalMetadataReader.m 1273 2017-02-21 19:24:34Z brideout $
metadataDir = '/tmp/test_metadata';

% init the object
reader = DigitalMetadataReader(metadataDir);

% get the sample bounds
[b0, b1] = reader.get_bounds();

% access all the object attributes
fields = reader.get_fields();
disp('The fields are:');
disp(fields);
disp('The samples per sec numerator, denominator, and float values are:');
disp(reader.get_samples_per_second_numerator());
disp(reader.get_samples_per_second_denominator());
disp(reader.get_samples_per_second());
disp('The subdirectory cadence in seconds is:');
disp(reader.get_subdirectory_cadence_seconds());
disp('The file cadence in seconds is:');
disp(reader.get_file_cadence_seconds());
disp('The file name prefix is:');
disp(reader.get_file_name());

% call the main method read for each field
for i=1:length(fields)
    data_map = reader.read(b0, b0+1, char(fields(i)));

    disp(sprintf('Displaying all data relating to field %s', fields{i}));
    recursive_disp_map(data_map);

end

function recursive_disp_map(map)
    % recursive_disp_map is an example method that walks though multiple
    % layers of containers.Map objects, and displaying them
    % Input: map - containers.Map whose values are may also be further
    % containers.Map objects that desend any number of levels
    disp('Displaying a container.Map object');
    keys = map.keys();
```

```
    disp(sprintf('Displaying map with %i keys', length(keys)));
    for i=1:length(keys)
        key = keys{i};
        disp('This key is:');
        disp(key);
        value = map(key);
        if (isprop(value,'ValueType'))
            disp('Value is another map');
            recursive_disp_map(value);
        else
            disp('Value is:');
            disp(value);
        end
    end
end
```