Digital RF 2.0

Juha Vierinen, William Rideout, Frank Lind, Robert Schaefer

Last edit: Aug 30, 2016

Overview

Digital RF 2.0 is a *disk storage* and archival format for radio signals. The design goals are the following:

- The format and the programming language interfaces are as simple as possible.
- Allow easy and efficient random access to multiple heterogenous channels based on global sample index and channel name.
- Allow user to optionally include metadata in a flexible and effortless manner.
- 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.
- Storing of data should use storage space of the natural binary representation of the data, be it 1-bit integer or 128-bit floating point.
- Sparsely sampled data should also achieve data storage proportional to the amount of data actually stored.
- Both real and complex valued signals are supported. Complex data can be built from any data type - integer or floating point.
- Recording multiple subchannels at once is supported. These subchannels must be written simultaneously as a block with each write command. All subchannels must share all the same metadata in the file - starting time, sample rate, complex versus single-valued, etc.
- Allow optional data compression checksumming when needed.
- Allow adding features, while maintaining backwards compatibility.

We intend the data format for various different use cases, including ring buffer on disk, and data archival. The format is flexible enough to support multiple different usage scenarios, such as single channel digital receiver recording, recording of multi channel polyphase filterbank output, or recording of different independent instruments with different sample rates and data formats.

Digital RF 2.0 is *not* a packetized format, such as VDIF or VITA 49. For a discussion on this topic, see section in the end of this document.

This document outlines the underlying Hdf5 file layout along with the file system layout of Digital RF 2.0 Hdf5 files, and describes the C and python interfaces that have been designed to read and write files in this format. Example code is given for all interfaces.

Digital RF 2.0 Hdf5 files differ from Digital RF 1.0 in that Digital RF 2.0 subdirectories and files have predictable names. This allows great simplification in the read API, because the API can predict the filenames it needs to access, and never needs to run glob to access files. Not only

is this simpler to implement, but it is also faster when there are large number of subdirectories or files. Indeed, with Digital RF 2.0, read access is independent of the amount of Digital RF data.

To accomplish this subdirectory and filename predictability, we changed two of the parameters used to write Digital RF data from (files_per_directory and samples_per_file) to (subdirectory_cadence_seconds and file_cadence_milliseconds). In essence, this means that subdirectories and file cover a set sample range, and will contain as many samples in that range as happen to be written. Previously, Digital RF 1.0 subdirectories and files always contained to same number of samples, and the possibility of data gaps made their names unpredictable, since their names were tied to the first sample each contained.

With Digital RF 2.0, subdirectories are always named with a timestamp where subdir_timestamp = subdirectory_cadence_seconds x N, where N is an integer. Files are always named with a millisecond timestamp where (subdir_timestamp*1000 - file_millisecond_timestamp) = file_cadence_milliseconds x M, where M is an integer. To ensure every file in a subdirectory contains the same maximum number of samples, the rule is enforced that subdir_cadence_secs*1000 % file_cadence_millisecs == 0.

If Hdf5 file sizes are allowed to vary, this implies that chunked storage must be used. Chunked storage will slow down read speeds somewhat. The use of compression and/or checksums also requires chunked storage. To allow Digital RF 2.0 to have a mode without chunked storage, and so with the fastest possible read speeds, we added one additional write argument: is_continuous. If the user sets the is_continuous argument to 1, then they can only write continuous blocks of data. When is_continuous == 1 and there is no checksum or compression, the API writes all files of size = maximum number of samples, and chunked storage is not used. If a user starts writing at a sample value in the middle of a file's range, then the preceding samples in that file will all be filled in with filler values (NaN for floats, smallest possible value for ints). This filler value may also appear at the end of Digital RF 2.0 files when chunking is not used.

Underlying HDF5 format

Design background

We choose HDF5 as the underlying file format for Digital RF for a number of reasons. The underlying C I/O code for HDF5 is highly optimized for fast writing. Also, the HDF5 format allows a file to be self-describing, so that users who encounter Digital RF files without access to this documentation or API could still understand this data. Finally, HDF5 is a well-accepted file format in the science community.

We made a design decision in Digital RF to support both continuous and noncontinuous data. For this reason, there are two HDF5 datasets at the top level of every Digital RF HDF5 file - rf_data and rf_data_index. The rf_data dataset contains the recorded values of RF data, independent of the times the samples were recorded. The rf_data_index dataset records the beginning of each continuous block of data in that particular file. The first sample in each Digital RF HDF5 file will always have an entry in rf_data_index, and there will be an additional entry in rf_data_index when a data gap occurs. For continuous data, the rf_data_index will contain only one row.

Another design decision underlying Digital RF is the requirement to support storing RF data in rf_data in a variety of data types. Data can be written as either complex or real (i.e., single-valued) numbers. Data formats can be any data format supported by the HDF5 standard, from bytes to 128 byte floating point numbers. One or more subchannels can be written simultaneously as a block. A complex vector would be a N x num_subchannels table, where each complex subchannel is made up of 'r' and 'i' column. This layout is based on the same layout used by the python module h5py to store complex numbers, but in our format complex values can be floats or any type of integer. A single valued signal would be a N x num_subchannels vector, without column names.

When writing Digital RF 2.0 data, we have used the concept of a channel. A channel is opened by the init method, and is defined by:

1. The top level directory where the channel is to be written
2. The channel name, which is also the subdirectory that is created under the top level directory
3. Whether the data is complex or single valued
4. The data format of the data (independent of whether its complex or single valued)
5. The number of subchannels to be simultaneously recorded (1 or greater)
6. The sample rate in Hz
7. The subdirectory cadence in integer seconds
8. The file cadence in integer millseconds.
9. The starting sample index (which is the unix timestamp times the sample rate - see more later in this document)
10. A user-defined UUID string to help match this channel with other information/metadata.
11. A compression level (0 for no compression, 9 for maximum)
12. A checksum flag
13. An is continuous flag. If is continuous, then gapped data cannot be passed in. This allows Digital 2.0 files to not use slower chunked data storage in the case where the data is continuous and compression and checksum are off.

These properties never change for a given channel, and are recorded as properties at the top level of each HDF5 file. If more than one subchannel is written, all subchannels must share the

above metadata defined in that Hdf5 file - for example, the global sample times and the sample rates.

To make the write API as simple and minimalistic as possible, the API only deals with one channel per instance. While we anticipate most radar data to be continuous, the recorded channel can be written in arbitrary sized vectors with arbitrary sized gaps between the vectors. If there are multiple subchannels in that channel, all subchannels must have the same gaps. This is to support time decimated blocks to accommodate high sample rate acquisition at manageable data rates. To implement functionality such as multi-channel recording where the channels have different times or sample rates, multiple channels are used. See the next section for more details on this.

In the rf_data_index dataset, the data is globally indexed as unsigned 64 bit integer as samples since 1970-01-01T00:00:00.00. This should be enough at 25 MHz for until the year 25367, and 2554 with a 1 GHz sample rate. With 10 GHz, we will run into a wall shortly, in year 2028, but by then we could just switch to 128 bit integers. Global indices will make things like ring buffer type functionality easier to implement, or allow dealing with situations where the data collection program is stopped or killed and restarted, but we are essentially collecting the same stream with a gap. Global indices will also make aligning data across multiple channels easier.

HDF5 layout details

Each file contains the following elements:

/rf_data # MxNx2 or MxNx1 vector of data, where M is the number of subchannels, N is the samples per file, and 2 for complex data, 1 for single valued. Data type can be any valid HDF5 data type.
/rf_data_index # Nx2 uint64, mapping between local (starts at 0) and global sample indices. The first row is always the global sample index and 0, indicating the first sample in the file.

The /rf_data dataset has the following 13 attributes:
/uuid_str # a string containing a UUID generated for that channel. uuid_str saved in each resultant Hdf5 file's metadata.
/seq_number # running number from start of acquisition. used to identify missing files
/is_complex - 1 if complex values, 0 if single valued.
/num_subchannels - 1 or more subchannels in the file. The meaning of the different subchannels is not defined in the rf file, but instead in the metadata file.
/subdir_cadence_secs # the number of seconds of data per subdirectory
/file_cadence_millisecs # the maximum number of millseconds of data per file
/sample_rate # sample rate (double precision) Any integer less than 2^53 can be represented exactly.

/computer_time # Computer time as unix seconds at creation of file.
                # A sanity check and backup in case data acquisition timing fails.
/digital_rf_version # A verrsion number of the Hdf5 RF format. Now 1.0
/digital_rf_time_description # a text description of how time is stored in this format
/epoch # a description of the epoch start. Now 1970-01-01T00:00:00Z
/init_utc_timestamp # the unix timestamp at which the first sample was recorded. Can
    be used to determine the leapsecond offset.
/is_continuous - 1 if data forced to be continuous, 0 for potentially gappy data.

The HDF5 files are written in the following way:

2014-03-30T12-00-00/rf@1396379502.000.h5

A new subdirectory is created at first write. Its name is in the format
YYYY-MM-SSTHH-MM-SS, where the timestamp ts is the largest for which

ts = subdirectory_cadence_seconds x N <= first_sample//sample_rate

is true, where N is an integer. A new subdirectory will be created when

next_sample/sample_rate >= subdirectory_cadence_seconds x (N+1)

The use of subdirectories is to avoid too many files in a directory (which can be very detrimental to performance), and to also naturally divide the amount of data into smaller more manageable bits. This naming convention also means that the sample range in any given subdirectory is predictable.

The file name rf@1396379502.000.h5 includes unix seconds and milliseconds. The millsecond timestamp of every file is determined by the largest millisecond timestamp where

file_ms = file_cadence_millseconds * N <= first_sample // (sample_rate / 1000.0)

is true, where N is an integer. All following samples will be written to the same file until

first_sample // (sample_rate / 1000.0) >= file_cadence_millseconds * (N+1). The maximum number of samples a file can hold is:

max_samples = file_cadence_millisecs * (sample_rate / 1000.0)

To ensure file boundaries and subdirectory boundaries line up, the write API requires subdir_cadence_secs and file_cadence_millisecs to be related by:

(subdir_cadence_secs * 1000) % file_cadence_millisecs == 0

Note that files that are being actively written to have the four characters <tmp.> prepended to their name, such as tmp.rf@1396379502.000.h5. When a new Hdf5 file is opened, the API will automatically rename the file to remove the <tmp.> characters. This renaming will also occur for the last file written when the close method is called. The reason for this special name for files being actively updated is to allow mirroring or backup scripts to determine which files are complete and which might change. See the C examples for how to add a SIGINT handler to remove the tmp file in case of a control-C interrupt for a C writing program.

High level namespace layout

In order to structure the data, we will use the following type of namespace layout:

<top_level_directory>/<channel_name>/2014-03-30T12-00-00/rf@1396379502.000.h5

where the basename of the top_level_directory has the recommended form experiment_name-{timestamp}. Here a timestamp with curly brackets is optional. In some cases it is nice to have (eg., campaign type of recording), where are in other cases not wanted behaviour (eg., a ring buffer that you expect to always reside in some place).

Digital metadata is defined in a separate document, but the Digital RF read api contains a call to read digital metadata if it written in the directory:

<top_level_directory>/<channel_name>/metadata/

While not yet implemented, a future release will make it to make it easy to identify data in digital rf format by placing a file README.digital_rf is placed under the top_level_directory. This file would act as an identifier, as well as a description of the data format.

It is permissible to start acquisitions into the same top_level_directory/channel_name directory.

Writing Digital RF data with C

C write API description

The following four methods represent the low level C RF write API. There are two write methods, one for writing a continuous block, and another with additional arguments to allow for writing a collection of data blocks with a single call.

Init method

```
Digital_rf_write_object * digital_rf_create_write_hdf5(char * directory, hid_t dtype_id,
                                                       uint64_t subdir_cadence_secs,
                                                       uint64_t file_cadence_millisecs,
                                                       uint64_t global_start_sample,
                                                       double sample_rate, char * uuid_str,
                                                       int compression_level, int checksum, int is_complex,
                                                       int num_subchannels, int is_continuous,
                                                       int marching_dots)
```

/* digital_rf_create_write_hdf5 returns an Digital_rf_write_data_object used to write a single channel of RF data to a directory, or NULL with error to standard error if failure.

Inputs:

char * directory - a directory under which to write the resultant Hdf5 files. Must already exist. Hdf5 files will be stored as YYYY-MM-DDTHH-MM-SS/rf@<unix_second>.<3 digit millsecond>.h5

hid_t dtype_id - data type id as defined by hdf5.h

uint64_t subdir_cadence_secs - Number of seconds of data found in one subdir. For example, 3600 subdir_cadence_secs will be saved in each subdirectory

uint64_t files_per_directory - the number of Hdf5 files in each directory of the form YYYY-MM-DDTHH-MM-SS. A new directory will be created when that number is reached. If 0, then subdirectories are instead created on hour boundaries, and will always have the form YYYY-MM-DDTHH:00:00.

uint64_t global_start_sample - The start time of the first sample in units of samples since UT midnight 1970-01-01.

double sample_rate - sample rate in Hz

char * uuid_str - a string containing a UUID generated for that channel. uuid_str saved in each resultant Hdf5 file's metadata.

int compression_level - if 0, no compression used. If 1-9, level of gzip compression. Higher compression means smaller file size and more time used.

int checksum - if non-zero, HDF5 checksum used. If 0, no checksum used.

int is_complex - 1 if complex (IQ) data, 0 if single-valued

int num_subchannels - the number of subchannels of complex or single valued data recorded at once. Must be 1 or greater. Note: A single stream of complex values is one subchannel, not two.

int is_continuous - 1 if all data to be written will be gap free, 0 if there might be gaps. If 1, then any attempt to write gapped data will raise an error

int marching_dots - non-zero if marching dots desired when writing; 0 if not

Continuous data write method

```
int digital_rf_write_hdf5(Hdf5_write_data_object *hdf5_data_object, uint64_t
        global_leading_edge_index, void * vector, uint64_t vector_length)
/*
digital_rf_write_hdf5 writes a continuous block of data from vector into one or more Hdf5 files

 Inputs:

        Digital_rf_write_data_object *hdf5_data_object - C struct created by
digital_rf_create_write_hdf5

        uint64_t global_leading_edge_index - index to write data to.  This is a global index with
zero representing the sample taken at the time global_start_sample specified in the init method.
Note that all values stored in Hdf5 file will have global_start_sample added, and this offset
should NOT be added by the user. Error raised and -1 returned if before end of last write.

        void * vector - pointer into data vector to write

        uint64_t vector_length - number of samples to write to Hdf5

        Affects - Writes data to existing open Hdf5 file.  May close that file and write some or all
of remaining data to new Hdf5 file.

Returns 0 if success, non-zero and error written if failure.
 */
```

Block data write method

```
int digital_rf_write_blocks_hdf5(Hdf5_write_data_object *hdf5_data_object, uint64_t *
        global_index_arr, uint64_t * data_index_arr, uint64_t index_len, void * vector, uint64_t
        vector_length)
/*
digital_rf_write_blocks_hdf5 writes blocks of data from vector into one or more Hdf5 files

Inputs:

        Hdf5_write_data_object *hdf5_data_object - C struct created by
digital_rf_create_write_hdf5

        uint64_t * global_index_arr - an array of global indices into the samples being written.
The global index is the total number of sample periods since data taking began, including gaps.
Note that all values stored in Hdf5 file will have global_start_sample added, and this offset
should NOT be added by the user.  Error is raised if any value is before its expected value
(meaning repeated data).

        uint64_t * data_index_arr - an array of len = len(global_index_arr), where the indices are
related to which sample in the vector being passed in is being referred to in global_index_arr.
First value must be 0 or error raised.  Values must be increasing, and cannot be equal or
greater than index_len or error raised.

        uint_64 index_len - the len of both global_index_arr and data_index_arr.  Must be
greater than 0.

        void * vector - pointer into data vector to write

        uint64_t vector_length - number of samples to write to Hdf5

        Affects - Writes data to existing open Hdf5 file.  May close that file and write some or all
of remaining data to new Hdf5 file.

        Returns 0 if success, non-zero and error written if failure.
 */
```

Close method

```
int digital_rf_close_write_hdf5(Hdf5_write_data_object *hdf5_data_object)
```

```
/* digital_rf_close_write_hdf5 closes open Hdf5 file if needed and releases all memory
associated with hdf5_data_object

Inputs:

        Hdf5_write_data_object *hdf5_data_object - C struct created by
digital_rf_create_write_hdf5
 */
```

Debug methods

```
char * digital_rf_get_last_file_written(Digital_rf_write_object *hdf5_data_object)
/* digital_rf_get_last_file_written returns a malloced string containing the full path to the last hdf5 file written
to

 Inputs:
        Digital_rf_write_object *hdf5_data_object - C struct created by digital_rf_create_write_hdf5

Returns:
        char * containing the full path to the last hdf5 file written to. User is responsible for freeing string
when done.

Returns empty string if no data written.
 */

char * digital_rf_get_last_dir_written(Digital_rf_write_object *hdf5_data_object)
/* digital_rf_get_last_dir_written returns a malloced string containing the full path to the last dir
written * to
 * Inputs:
 *              Digital_rf_write_object *hdf5_data_object - C struct created by
 *                  digital_rf_create_write_hdf5
 *
 *      Returns:
 *              char * containing the full path to the last directory written to. User is responsible for
 *                      freeing string when done.
 *              Returns empty string if no data written.
 */



uint64_t digital_rf_get_last_write_time(Digital_rf_write_object *hdf5_data_object)
/* digital_rf_get_last_write_time returns the unix timestamp of the last write

Inputs:
        Digital_rf_write_object *hdf5_data_object - C struct created by digital_rf_create_write_hdf5
```

```
Returns: uint64_t representing the unix timestamp of the last write.  If no writes occurred yet, returns 0.
 */
```

C write example

```
/*
 * Simple example of writing Digital RF 2.0 data with C API
 *
 * This simple example writes continuous complex data of short ints
 *
 * $Id: example_rf_write_hdf5.c 875 2015-12-02 17:51:40Z brideout $
 */

#include "digital_rf.h"


int main (int argc, char *argv[])
{

        /* local variables */
        Digital_rf_write_object * data_object = NULL; /* main object created by init */
        uint64_t vector_leading_edge_index = 0; /* index of the sample being written starting at zero with
the first sample recorded */
        uint64_t global_start_index; /* start sample (unix time * sample_rate) of first measurement - set
below */
        int i, result;

        /* dummy dataset to write */
        short data_short[100][2];

        /* writing parameters */
        double sample_rate = 100.0; /* 100 Hz sample rate - typically MUCH faster */
        uint64_t subdir_cadence = 4; /* Number of seconds per subdirectory - typically longer */
        uint64_t millseconds_per_file = 400; /* Each subdirectory will have up to 10 400 ms files */
        int compression_level = 1; /* low level of compression */
        int checksum = 0; /* no checksum */
        int is_complex = 1; /* complex values */
        int is_continuous = 1; /* continuous data written */
        int num_subchannels = 1; /* only one subchannel */
        int marching_periods = 0; /* no marching periods when writing */
        char uuid[100] = "Fake UUID - use a better one!";
        uint64_t vector_length = 100; /* number of samples written for each call - typically MUCH longer */

        /* init dataset */
```

```
        for (i=0; i<100; i++)
        {
                data_short[i][0] = 2*i;
                data_short[i][1] = 3*i;
        }

        /* start recording at global_start_sample */
        global_start_index = (uint64_t)(1394368230 * sample_rate) + 1; /* should represent 2014-03-09
12:30:30  and 10 milliseconds*/


        printf("Writing complex short to multiple files and subdirectores in /tmp/hdf5 channel junk0\n");
        system("rm -rf /tmp/hdf5 ; mkdir /tmp/hdf5 ; mkdir /tmp/hdf5/junk0");

        /* init */
        data_object = digital_rf_create_write_hdf5("/tmp/hdf5/junk0", H5T_NATIVE_INT, subdir_cadence,
millseconds_per_file, global_start_index, sample_rate, uuid, compression_level, checksum, is_complex,
num_subchannels, is_continuous, marching_periods);
        if (!data_object)
                exit(-1);


        /* write continuous data */
        for (i=0; i<7; i++) /* writing 700 samples, so should create two subdirectories (each holds 400
samples) */
        {
                result = digital_rf_write_hdf5(data_object, vector_leading_edge_index + i*100, data_short,
vector_length);
                if (result)
                        exit(-1);
        }

        /* close */
        digital_rf_close_write_hdf5(data_object);

        printf("example done - examine /tmp/hdf5 for data\n");
        return(0);
}
```

C write example with SIGINT handling to remove last file

The following example is similar to the one above, except that it includes a SIGINT handler to remove the tmp rf file in case of an interrupt. Because signal handlers take no arguments except the interrupt type, a global string is used to keep track of the last directory being written to. This example assumes only a single Digital_rf_write_object is being created. If this program

was using multiple instances of Digital_rf_write_object, the global would need to contain information about the last directory of each instance.

This example differs in four ways:

1. It includes signal.h and unistd.h.
2. It includes an interrupt handler outside of the main method.
3. It declares a global string to track the last directory written,
4. In the main method, there are three lines of code after the write method to update the global string containing the last directory written to.

Here's the needed includes:

```
#include <signal.h>
#include <unistd.h>
```

Here's the global declaration:

```
char global_last_rf_dir_written[BIG_HDF5_STR];
```

Here's the interrupt handler:

```
void intHandler(int dummy)
{
        char cmd[BIG_HDF5_STR] = "";
        sprintf(cmd, "rm %s/tmp.*", global_last_rf_dir_written);
        system(cmd);
   exit(-1);
}
```

Here's the write call, followed by the code to update the global:

```
result = digital_rf_write_hdf5(data_object, vector_leading_edge_index + i*100, data_short, vector_length);
/* update global last_rf_dir_written after every call */
local_last_dir_written = digital_rf_get_last_dir_written(data_object);
strcpy(global_last_rf_dir_written, local_last_dir_written);
free(local_last_dir_written); /* digital_rf_get_last_dir_written dynamically allocates memory */
```

Here's the full interrupt handling example:

```
/*
 * Simple example of writing Digital RF 2.0 data with C API
 *
 * This simple example writes continuous complex data of short ints,
 * and also removes partially written files when a SIGINT occurs
```

```
 *
 * $Id: example_rf_write_hdf5.c 985 2016-02-02 18:35:23Z brideout $
 */

#include <signal.h>
#include <unistd.h>

#include "digital_rf.h"

/* the following code gives an example of writing a SIGINT handler that deletes the tmp file being written
 * Note that this assumes a non-threaded application because it uses a global to track the last file
 */


char global_last_rf_dir_written[BIG_HDF5_STR];

void intHandler(int dummy)
{
        char cmd[BIG_HDF5_STR] = "";
        sprintf(cmd, "rm %s/tmp.*", global_last_rf_dir_written);
        system(cmd);
   exit(-1);
}



int main (int argc, char *argv[])
{

        /* local variables */
        Digital_rf_write_object * data_object = NULL; /* main object created by init */
        uint64_t vector_leading_edge_index = 0; /* index of the sample being written starting at zero with
the first sample recorded */
        uint64_t global_start_index; /* start sample (unix time * sample_rate) of first measurement - set
below */
        int i, result;
        char * local_last_dir_written; /* used for interrupt handler */

        /* dummy dataset to write */
        short data_short[100][2];

        /* writing parameters */
        double sample_rate = 100.0; /* 100 Hz sample rate - typically MUCH faster */
        uint64_t subdir_cadence = 4; /* Number of seconds per subdirectory - typically longer */
        uint64_t millseconds_per_file = 400; /* Each subdirectory will have up to 10 400 ms files */
        int compression_level = 1; /* low level of compression */
        int checksum = 0; /* no checksum */
        int is_complex = 1; /* complex values */
```

```
        int is_continuous = 1; /* continuous data written */
        int num_subchannels = 1; /* only one subchannel */
        int marching_periods = 0; /* no marching periods when writing */
        char uuid[100] = "Fake UUID - use a better one!";
        uint64_t vector_length = 100; /* number of samples written for each call - typically MUCH longer */

        /* set up signal handling */
        signal(SIGINT, intHandler);

        /* init dataset */
        for (i=0; i<100; i++)
        {
                data_short[i][0] = 2*i;
                data_short[i][1] = 3*i;
        }

        /* start recording at global_start_sample */
        global_start_index = (uint64_t)(1394368230 * sample_rate) + 1; /* should represent 2014-03-09
12:30:30  and 10 milliseconds*/


        printf("Writing complex short to multiple files and subdirectores in /tmp/hdf5 channel junk0\n");
        system("rm -rf /tmp/hdf5 ; mkdir /tmp/hdf5 ; mkdir /tmp/hdf5/junk0");

        /* init */
        data_object = digital_rf_create_write_hdf5("/tmp/hdf5/junk0", H5T_NATIVE_INT, subdir_cadence,
millseconds_per_file,
                        global_start_index, sample_rate, uuid, compression_level, checksum, is_complex,
num_subchannels,
                        is_continuous, marching_periods);
        if (!data_object)
                exit(-1);

        /* write continuous data */
        for (i=0; i<10000; i++)
        {
                result = digital_rf_write_hdf5(data_object, vector_leading_edge_index + i*100, data_short,
vector_length);
                if (result)
                        exit(-1);
                /* update global last_rf_dir_written after every call */
                local_last_dir_written = digital_rf_get_last_dir_written(data_object);
                strcpy(global_last_rf_dir_written, local_last_dir_written);
                free(local_last_dir_written); /* digital_rf_get_last_dir_written dynamically allocates memory
*/
                if (i % 1000 == 0)
                        printf("%i of 10000 written\n", i);
        }
```

```
        /* close */
        digital_rf_close_write_hdf5(data_object);

        printf("example done - examine /tmp/hdf5 for data\n");
        return(0);
}
```

Writing Digital RF data with Python

Python write API description

Digital RF data can also be written using the python digital_rf_hdf5 module. The python API uses the C write API described above. As with C, there are four method - init, rf_write for continuous data, rf_write_blocks for blocked data, and close. They are defined below:

Init

```
class write_hdf5_channel:
    """The class write_hdf5_channel is an object used to write Digital RF 2.0 data to Hdf5 files.
    """

    def __init__(self, directory, dtype_str, subdir_cadence_secs, file_cadence_millisecs,
            start_global_index, sample_rate, uuid_str, compression_level=0, checksum=False,
            is_complex=True, num_subchannels=1, is_continuous=True, marching_periods=True):
        """__init__ creates an write_hdf5_channel

        Inputs:
            directory - the directory where this channel is to be written.  Must already exist and be
writable

            dtype_str - format of numpy data in string format.  String is format as passed into
numpy.dtype().  For example, numpy.dtype('>i4').  For now accepts any legal byte-order
character (No character means native), and one of 'i1', 'u1', 'i2', 'u2', 'i4', 'u4', 'i8', 'u8', 'f', or 'd'.

            samples_per_file - number of samples in each Hdf5 file
```

subdir_cadence_secs - Number of seconds of data found in one subdir. For example, 3600 subdir_cadence_secs will be saved in each subdirectory

file_cadence_millisecs - number of milliseconds of data per file. Rule: subdir_cadence_secs*1000 % file_cadence_millisecs must equal 0

start_global_index - the start time of the first sample in units of (unix_timestamp * sample_rate)

sample_rate - sample rate in Hz

uuid_str - uuid string that will tie the data files to the Hdf5 metadata

compression_level - 0 for no compression (default), 1-9 for varying levels of gzip compression (1 least compression, least CPU, 9 most compression, most CPU)

checksum - if True, use Hdf5 checksum capability, if False (default) no checksum.

is_complex - if True (the default) data is IQ. If false, each sample has a single value.

num_subchannels - number of subchannels to write simultaneously. Default is 1.

is_continuous - True if data always written in continuous blocks. False if data will be written with gapped blocks. If is_continuous True, checksum False and compression_level 0 (all defaults), fastest read speed.

marching_periods - if True, have matching periods written to stdout when writing. False - do not.

"""

Write continuous data

```
def rf_write(self, arr, next_sample=None):
    """rf_write writes a numpy array to Hdf5.  Must have the same number of subchannels as declared in init. For single valued data, number of columns == number of subchannels.  For complex data, there are two types of input arrays that are allowed:
        1. An array without column names with number of columns = 2*num_subchannels.  I/Q are assumed to be interleaved.
        2. A structured array with column names r and i, as stored in the Hdf5 file.  Then the shape will be N * num_subchannels, because numpy considered the r/i data as one piece of data.
```

Here's an example of one way to create a structured numpy array with complex data with dtype int16:

```
arr_data = numpy.ones((num_rows, num_subchannels), dtype=[('r', numpy.int16), ('i', numpy.int16)])
for i in range(num_subchannels):
    for j in range(num_rows):
        arr_data[j,i]['r'] = 2
        arr_data[j,i]['i'] = 3
```

Inputs - arr - numpy array of data of size described above if complex, and size num_rows if not. Error will be raised if its not the same data type set in init.

next_sample - global index of next sample to write to. Default is self._next_avail_sample. Error raised if next_sample < self._next_avail_sample

Returns: self._next_avail_sample

"""

Write blocked data

```
def rf_write_blocks(self, arr, global_sample_arr, block_sample_arr):
    """rf_write_blocks writes a data with interleaved gaps to Hdf5 files

    Inputs - arr - numpy array of data. See rf_write for a complete description.

        global_sample_arr an array len < N, > 0 of type numpy.uint64 that sets the global sample index for each continuous block of data in arr. Must be increasing, and first value must be >= self._next_avail_sample of ValueError raised.

        block_sample_arr an array len = len(global_sample_arr) of type numpy.uint64. Values are the index into arr of each block. Values must be < len(arr). First value must be zero. Increments between value must be > 0 and less than the corresponding increment in global_sample_arr

    Returns: self._next_avail_sample
    """
```

Debug methods

```
def get_last_file_written(self):
```

```
        """get_last_file_written returns the full path to the last file written
        """


    def get_last_dir_written(self):
        """get_last_dir_written returns the full path to the last directory written
        """


    def get_last_utc_timestamp(self):
        """get_last_utc_timestamp returns utc timestamp of the time of the last write
        """
```

Close

```
    def close(self):
        """close frees the C object and closes the last Hdf5 file
        """
```

Python write example

```
"""example_digital_rf_hdf5.py is a simple example of writing Digital RF with python

Writes continuous complex short data.

$Id: example_digital_rf_hdf5.py 811 2015-09-09 19:14:36Z brideout $
"""
# standard python imports
import os

# third party imports
import numpy

# Millstone imports
import digital_rf_hdf5

# writing parameters
sample_rate = 100.0 # 100 Hz sample rate - typically MUCH faster
dtype_str = 'i2' # short int
sub_cadence_secs = 4# Number of seconds of data in a subdirectory - typically MUCH larger
file_cadence_millisecs = 400 # Each fill will have up to 400 ms of data
compression_level = 1 # low level of compression
checksum = False # no checksum
```

```
is_complex = True # complex values
is_continuous = True
num_subchannels = 1 # only one subchannel
marching_periods = False # no marching periods when writing
uuid = "Fake UUID - use a better one!"
vector_length = 100 # number of samples written for each call - typically MUCH longer

# create short data in r/i to test using that to write
arr_data = numpy.ones((vector_length,num_subchannels),
                      dtype=[('r', numpy.int16), ('i', numpy.int16)])
for i in range(len(arr_data)):
    arr_data[i]['r'] = 2*i
    arr_data[i]['i'] = 3*i

# start 2014-03-09 12:30:30 plus one sample
start_global_index = (1394368230 * sample_rate) + 1

# set up top level directory
os.system("rm -rf /tmp/hdf5 ; mkdir /tmp/hdf5");

print("Writing complex short to multiple files and subdirectores in /tmp/hdf5 channel junk0");
os.system("rm -rf /tmp/hdf5/junk0 ; mkdir /tmp/hdf5/junk0");

# init
data_object = digital_rf_hdf5.write_hdf5_channel("/tmp/hdf5/junk0", dtype_str, sub_cadence_secs,
                                        file_cadence_millisecs, start_global_index,
                                        sample_rate, uuid, compression_level, checksum,
                                        is_complex, num_subchannels, is_continuous, marching_periods);
# write
for i in range(7): # will write 700 samples - so creates two subdirectories
    result = data_object.rf_write(arr_data);

# close
data_object.close();
print("done test");
```

Reading Digital RF data with Python

Python read API description

The most basic functionality of the read API would include random access to the recorded RF, while providing guidance to where data exists. All indices in this API refer to unix sample

indices, which is defined as the unix timestamp times the sample rate. Changes of sample rate from the same RF source must be handled by using different channels or separate experiment intervals.

```
class read_hdf5:
    """The class read_hdf5 is an object used to read Digital RF 2.0 data from Hdf5 files.
    This class allows random access to the rf data.

    """
    def __init__(self, top_level_directory_arg):
        """__init__ will verify the data in top_level_directory_arg is as expected.

        Inputs:
            top_level_directory_arg - either a single top level directory, or a list.  A directory can be a file system path or a url, where the url points to a top level directory. Each must be a local path, or start with  http://,  file://, or ftp://

        A top level directory must contain <channel_name>/<YYYY-MM-DDTHH-MM-SS/rf@<unix_seconds>.<%03i milliseconds>.h5

        If more than one top level directory contains the same channel_name subdirectory, this is considered the same channel.  An error is raised if their sample rates differ, or if their time periods overlap.

        This method will create the following attributes:

        self._top_level_dir_dict - a dictionary with keys = top_level_directory string, value = access mode (eg, 'local', 'file', or 'http')

        self._channel_dict - a dictionary with keys = channel_name, and value is a _channel_metadata object.
        """

def get_channels(self):
        """get_channels returns an alphabetically sorted list of channels in this read_hdf5 object
        """

def get_bounds(self, channel_name):
            """get_bounds returns a tuple of (first_unix_sample, last_unix_sample) for a given
            channel name
             “""

def get_rf_file_metadata(self, channel_name):
```

```
        """get_rf_file_metadata returns a dictionary of metadata found as attributes in the Hdf5
file /rf_data dataset for the given channel name. Keys are:

            subdir_cadence_secs (int)
            file_cadence_millisecs (int)
            uuid_str (string)
            sample_rate (double)
            is_complex (0 or 1 - int)
            num_subchannels (int)
            computer_time (unix timestamp from computer when file created) (uint64_t)
            init_utc_timestamp (unix timestamp from computer when first sample written - used to
                determine leapsecond offset), epoch ('1970-01-01 00:00:00 UT')
            digital_rf_time_description - string describing sample time used
            digital_rf_version (now '2.0', was '1.0')
        """


def get_digital_metadata(self, channel_name, top_level_dir=None):
        """get_digital_metadata returns a digital_metadata.read_digital_metadata object
associated with channel_name.

        Assumes digital_metadata is written under one of the top_leve_directories under the
directory <channel_name>/metadata.  If no such directory exists, raises an IOError.  Will return
the first digital_metadata object found in the top_level_directory_arg list if more than one, unless
optional argument top_level_dir given to choose one from among the list.
        """


def get_continuous_blocks(self, start_sample, end_sample, channel_name):
        """get_continuous_blocks returns an OrderedDict, where the keys are the start sample of
each continuous block found between start_sample and end_sample for the channel given, and
values the length of continuous data.

        Very similar to read, except lengths of arrays used instead of arrays themselves
        """


def read(self, start_sample, end_sample, channel_name, sub_channel=None):
        """read is the basic read method of Digital RF.  It returns an OrderedDict, where the keys
are the start sample of each continuous block found between start_sample and end_sample,
(inclusive) and values of numpy arrays of continuous data that start at the key, and the type
matches that of the hdf5 file rf_data dataset.

        If sub_channel is not None, then return only the selected sub_channel (int) (starts at 0)
```

```
        """



    def read_vector(self, unix_sample, vector_length, channel_name, sub_channel=None):
        """read_vector returns a numpy vector of complex8 type, no matter the dtype of the Hdf5
file or the number of channels. Shape is (vector_length, num_subchannels). Single value (real)
files will have the imaginary part set to zero.

        Calls read, then converts result.

        Inputs:
            unix_sample - the number of samples since 1970-01-01 at start of data

            vector_length - the number of continuous samples to include

            channel_name - the channel name to use

            sub_channel - if not None, then return only the selected sub_channel (int) (starts at 0).
                If None (the default) return all channels

        This method will raise an IOError error if the returned vector would include any missing
data.
        """



def read_vector_raw(self, unix_sample, vector_length, channel_name):
        """read_vector_raw returns a numpy array of dim(up to num_samples, num_subchannels)
of the dtype in the Hdf5 files.

        If complex data, real and imag data will have names 'r' and 'i' if underlying data are integers
or be numpy complex data type if underlying data floats.

        Inputs:
            unix_sample - the number of samples since 1970-01-01 at start of data

            vector_length - the number of continuous samples to include

            channel_name - the channel name to use

        This method will raise an IOError error if the returned vector would include any missing
data.
```

```
        """



    def read_vector_c81d(self, unix_sample, vector_length, channel_name, subchannel=0):
        """read_vector_c81d returns a numpy vector of complex8 type, no matter the dtype of the
Hdf5 file or the number of channels. Error thrown if subchannel doesn't exist.

        Inputs:
            unix_sample - the number of samples since 1970-01-01 at start of data

            vector_length - the number of continuous samples to include

            channel_name - the channel name to use

            subchannel - which subchannel to use.  Default is 0 (first)

        This method will raise an IOError error if the returned vector would include any missing
data.
        """

    def get_last_write(self, channel_name):
        """get_last_write returns a tuple of 1. timestamp of last file written, and 2. full path
        to last file written for a given input channel.  Both will be None if no data.
        """
```

Python read example

```
"""example_digital_rf_hdf5.py is an example script using the digital_rf_hdf5 module

Assumes one of the example Digital RF scripts has already been run (C:
example_rf_write_hdf5, or
Python: example_digital_rf_hdf5.py)

$Id: example_read_digital_rf.py 814 2015-09-10 15:52:10Z brideout $
"""
# Millstone imports
import digital_rf_hdf5
```

```
testReadObj = digital_rf_hdf5.read_hdf5(['/tmp/hdf5'])
channels = testReadObj.get_channels()
if len(channels) == 0:
    raise IOError, """Please run one of the example write scripts
        C: example_rf_write_hdf5, or Python: example_digital_rf_hdf5.py
        before running this example"""
print('found channels: %s' % (str(channels)))

print('working on channel junk0')
start_index, end_index = testReadObj.get_bounds('junk0')
print('get_bounds returned %i - %i' % (start_index, end_index))
cont_data_arr = testReadObj.get_continuous_blocks(start_index, end_index, 'junk0')
print('The following is a list of all continuous block of data in (start_sample, length) format: %s' %
(str(cont_data_arr)))

# read data - the first 3 reads of four should succeed, the fourth read will be beyond the
available data
start_sample = cont_data_arr[0][0]
for i in range(4):
    try:
        result = testReadObj.read_vector(start_sample, 200, 'junk0')
        print('read number %i got %i samples starting at sample %i' % (i, len(result), start_sample))
        start_sample += 200
    except IOError:
        print('Read number %i went beyond existing data and raised an IOError' % (i))

# finally, get all the built in rf metadata
rf_dict = testReadObj.get_rf_file_metadata('junk0')
print('Here is the rf metadata built into the Digital RF Hdf5 files: %s' % (str(rf_dict)))
```

Reading Digital RF data with Matlab

Matlab read API description

```
classdef DigitalRFReader
    % class DigitalRFReader allows easy read access to static Digital RF data
    %   See testDigitalRFReader.m for usage, or run <doc DigitalRFReader>
    %
    % $Id: DigitalRFReader.m 970 2016-01-20 20:52:27Z brideout $
```

```
methods
  function reader = DigitalRFReader(topLevelDirectories)
    % DigitalRFReader is the contructor for this class.
    % Inputs - topLevelDirectories - a char array of one or more
    % top level directories, where a top level directory holds
    % channel directories


    % topLevelDirectories - a char array of one or more top level
    %   directories.

  function channels = get_channels(obj)
    % get_channels returns a cell array of channel names found
    % Inputs: None



  function [lower_sample, upper_sample] = get_bounds(obj, channel)
    % get_bounds returns the first and last sample in channel.
    % sample bounds are in samples since 0 seconds unix time
    % (that is, unix time * sample_rate)


  function [data_map] = read(obj, channel, start_sample, end_sample, subchannel)
    % read returns a containers.Map() object containing key= all
    % first samples of continuous block of data found between
    % start_sample and end_sample (inclusive).  Value is an array
    % of the type stored in /rf_data. If subchannel is 0, all
    % channels returned.  If subchannel == -1, length of continuous
    % data is returned instead of data, Else, only subchannel set
    % by subchannel argument returned.


  function subdir_cadence_secs = get_subdir_cadence_secs(obj, channel)
    % get_subdir_cadence_secs returns subdir_cadence_secs for given channel


  function file_cadence_millisecs = get_file_cadence_millisecs(obj, channel)
    % get_file_cadence_millisecs returns file_cadence_millisecs for given channel


  function samples_per_second = get_samples_per_second(obj, channel)
    % get_samples_per_second returns samples_per_second for given channel


  function is_complex = get_is_complex(obj, channel)
    % get_is_complex returns is_complex (1 or 0) for given channel
```

```
    function num_subchannels = get_num_subchannels(obj, channel)
        % get_num_subchannels returns num_subchannels (1 or greater) for given channel


    function vector = read_vector(obj, channel, start_sample, sample_length)
        % read_vector returns a data vector sample_length x num_subchannels.
        % Data type will be complex if data was complex, otherwise data type
        % as stored in same format as in Hdf5 file. Raises error if
        % data gap found.  Simply calls read for all channels, and
        % throws error if more than one block returned.
```

Reading Digital RF data with Matlab example

```
% example usage of DigitalRFReader.m
% Requires python test_digital_rf_hdf5.py be run first to create test data
top_level_directories = char('/tmp/hdf5', '/tmp/hdf52');
reader = DigitalRFReader(top_level_directories);
disp(reader.get_channels());

disp('First test is reading gappy data');
[lower_sample, upper_sample] = reader.get_bounds('junk4.1');
disp([lower_sample, upper_sample]);
disp(reader.get_subdir_cadence_secs('junk4.1'));
disp(reader.get_file_cadence_millisecs('junk4.1'));
disp(reader.get_samples_per_second('junk4.1'));
disp(reader.get_is_complex('junk4.1'));
disp(reader.get_num_subchannels('junk4.1'));

disp('Read data itself - channel 2');
gap_arr = reader.read('junk4.1', 139436843450, 139436843550, 2);
keys = gap_arr.keys();
for i = 1:length(keys)
    key = keys{i}
    gap_arr(key)
end

disp('Now just get block lengths');
gap_arr = reader.read('junk4.1', 139436843450, 139436843550, -1);
keys = gap_arr.keys();
for i = 1:length(keys)
    key = keys{i}
    gap_arr(key)
end
```

Appendix - Technical discussions

Sample rate

What is the most general and exact way to indicate sample rate? The current export format uses sample duration in picoseconds. Other ideas would be sample rate as integer, or perhaps a floating point number.

Sample duration as picoseconds is a good solution for systems that result in nontranscendental sample duration, eg., 100 MHz. However, a sample rate of 30 MHz wouldn't translate well to integer sample duration, but it would translate well to an integer sample rate.

The best way to indicate sample rate would be a symbolic mathematical expression, eg., "⅓ MHz or 1024124534534543242 34/2**32 Hz". This could then be evaluated with necessary precision to reduce the numerical errors of possibly transcendental sample rates or durations to an acceptable level. Information such as this would naturally belong in the metadata.

However, we need a sample rate in our api to calculate file names, based on global sample counts. Double precision numbers have a numerical accuracy of 10^{-16}, which is good enough to print filenames with sufficient accuracy. Also, integer numbers less than 2**53 are represented precisely with float64. Therefore, I think that we should should use double precision to indicate sample rate in the rf files. We should also provide a symbolic expression in the metadata to precisely indicate sample rate.

```
sr = 1e10 ; print("%1.20f"%((numpy.pi*1396445513.0*sr)/(numpy.pi*sr)))
sr = (1.0/3.0)*10e6 ; print("%1.20f"%((numpy.pi*1396445513.0*sr)/(numpy.pi*sr)))
```

Leap seconds

We deal with samples since 1970. This can be mapped uniquely to international atomic time without leap seconds. If leap seconds are needed, in order to align the recording with unix seconds (UTC), the user needs to deal with this in some way. This can be done by adjusting the global sample count since 1970 to align global sample count to UTC. The file write and read APIs is somewhat agnostic of this, although using UTC might encounter some special cases when reading data (eg., starting sampling on a leap second results in ambiguous timing, which probably can be resolved by comparing sample indices to computer timestamps).

If a leap second occurs during a recording, which uses UTC timebase for global sample indices, there will be a one second offset in the file names after the leap second, until sampling is restarted. Anyone that cares about this issue will be aware of it and can deal with it in any way necessary. Again, in order to avoid leap seconds, a timebase without leap seconds should be used.

In either way, leap seconds do not cause any catastrophic problems for us. There is no data loss associated with leap seconds.

VDIF & VITA 49 (Digital IF ; ANSI)

Why not use an existing format, such as VDIF [1] or the ANSI VITA 49 Digital IF standard? Why invent yet another format? Many core concepts of Digital RF are not very dissimilar to those of VDIF or VITA 49. Channels can be easily translated to threads, and the omnipresent timestamps in Digital RF are also an essential part of these formats. VDIF actually has a slightly more flexible way of organizing channels, as multiple channels can be in a thread, as long as they have the same format and sample rate. Some of the different number formats are also allowed by VDIF and VITA 49.

The main difference is that these formats are packet oriented, where each atom of data tightly conforms to a certain set of rules that allow packetizing the data in finite sized packets of data, which are suited for FPGA processing and potentially lossy transport over a network. This nature of the format by necessity restricts the amount of metadata that can be associated with the data in an atomic unit. Digital RF is a file oriented format, with more flexibility on what metadata can be associated with the data atoms. Files are also self-contained, ie., their format can be deduced from information contained within the file itself. For this functionality, we rely on an established data interchange format called HDF5.

Digital RF is random access by nature, allowing fast random access to any channel and point in time data stored on disk. It is possible to build this functionality to access packets stored on disk too, so this is not a major difference. It is also possible to configure Digital RF in a way that allows easy translation of the data stored in the files into a packetized format, such as VDIF or the LOFAR format, but this requires adhering to additional rules on how to use the format. Packetized formats prevents the user from coming up with a configuration that cannot be

efficiently packetized and impose specific realtime metadata requirements which may or may not be sufficient for specific processing.

Digital RF is at its best in an interpreted language environment with high level libraries for reading data (= Python or Matlab), whereas VDIF is more at home on FPGAs or C, where simply casting bytes in memory into a fixed struct will parse the data. Thus, a lower level of expertise is needed to work with Digital RF. Again, an interpreted language reader can be made to access VDIF files easily, and similarly, a easy to use low level read API can also be made to access Digital RF.

Digital RF supports more robust and extensible metadata contained with the voltage data. This is necessary to provide context for data to enable reuse over longer time scales and by multiple users who did not participate in the original data acquisition. Additionally it enables signal processing that transforms voltages to voltages to provide some documentation of the processing associated the transform in the output stream. This is necessary for processing traceability.

In the end, the differences are subtle. The main difference is that Digital RF relies on a widely used and standardized data interchange format HDF5 and with greater metadata support. VITA49 is an ANSI standard with general suitability for realtime stream processing but moderate implementation complexity. VDIF is custom binary format, designed for a specific purpose, with relatively simple FPGA or C code implementation. The fact that there are not many differences is good for interoperability, as data can be translated easily between formats.

Justifying the differences in the formats comes down specifically to the use cases involved. For Digital RF we anticipate use in voltage level processing and archival of data from large scale instrument arrays which can be subsequently processed and analyzed over long time scales (i.e. decades) by multiple users who are often disconnected from the original data acquisition. General purpose computing will be used and additional metadata will enable greater automation of processing workflows, improved traceability of software based signal processing, and the potential for multiple processing workflows to transparently use the same source data.

[1] http://vlbi.org/vdif/docs/2009.06.25_Whitney_e-VLBI_wkshop-Madrid.pdf