Digital RF 1.0

Juha Vierinen, William Rideout, Frank Lind, Robert Schaefer

Overview

Digital RF 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 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 file system layout of Digital RF, and describes the C and python interfaces that have been designed to read and write files in this format.

High level namespace layout

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

experiment_name-{timestamp}/channel_name/2014-03-30T12-35-29/rf@1396379502.000.h5
experiment_name-{timestamp}/channel_name/2014-03-30T12-35-29/metadata@1396379502.000.h5

experiment_name-{timestamp}/channel_name/metadata@1396375000.000.h5

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).
In order to make it easy to identify data in digital rf format, a file README.digital_rf is placed under the root directory: experiment_name-{timestamp}/README.digital_rf. 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 /experiment_name-{timestamp}/channel_name directory. Each time a data acquisition is started, a near ISO 8601 date format (eg., 2014-03-30T12-35-29) directory name is created (dash used instead of hyphen since Windows does not accept a hyphen). Also, a new directory is created every N Hdf5 files (stride) to avoid having too many files in one directory, and to avoid directories that are too large. A directory can be created only every second, or an error is raised.

The metadata associated with channels is stored in files of format metadata@1396378502.000.h5 each time a data acquisition is started. This is to ensure that previous files are not overwritten. The metadata contains channel specific information, which can be e.g., center frequency of channel, or over application specific information.

High level RF Signal API (read)

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 rf data from Hdf5 files as specified
   in the http://www.haystack.mit.edu/pipermail/rapid-dev/2014-February/000273.html email
thread.
```

```
    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.  It will
          analyze metadata about all Hdf5 files so that other methods can return quickly

                Inputs:
                    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.

                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') This attribute is static, that is, it is not updated
          when self.reload() called

                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.
        """
```

```
def get_metadata(self, channel_name, timestamp=None):
        """get_metadata returns a h5py.File object pointing to the metadata*.h5 file at the top level of
the channel directory.  The user is responsible for closing that file when done.  If timestamp ==
None, the latest metadata*.h5 will be returned.  Otherwise it will open the earliest metadata file
with timestamp greater than or equal to timestamp
        """



def get_continuous_blocks(self, start_unix_sample, stop_unix_sample, channel_name):
            """get_continuous_blocks returns a numpy array of dtype u64 and shape (N,2) where the
            first column represents the unix_sample of a continuous block of data, and the second
            column represents the number of samples in that continuous block.  Only samples
            between (start_unix_sample, stop_unix_sample) inclusive will be returned.

            Calls the private method _get_continuous_blocks.  If that raises a _MissingMetadata
            exception, calls  reload to get missing metadata, and then retries
            _get_continuous_blocks.

            Returns IOError if no blocks found

            Inputs:
                        start_unix_sample, stop_unix_sample - only samples between
            (start_unix_sample, stop_unix_sample) inclusive will be returned.  Value of both are
            samples since 1970-01-01

                        channel_name - channel to examine
            """


def read_vector(self, unix_sample, vector_length, channel_name):
            """read_vector 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.

            Calls private method _read_vector.  If that method raises a _MissingMetadata exception,
            calls reload, then tries again.

            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.  It will also raise an IOError is any of the files needed to read the data have been
            deleted.   This is possible because metadata on which this call is based might be out of
            date.
                  """

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 fill or the number of channels. Error thrown if subchannel doesn't exist

            Calls private method _read_vector.  If that method raises a _MissingMetadata exception,
            calls reload, then tries again.

                  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.  It will also raise an IOError is any of the files needed to read the data have been
            deleted.  This is possible because metadata on which this call is based might be out of
            date.
            """

def reload(self, start_sample=None, sample_length=None, channel=None):
         """reload updates the attributes self._channel_location_dict and self._channel_dict.  If
            start_sample and/or sample_length == None, then only get high level metadata.  If
            start_sample and sample_length given, then only updated detailed metadata as needed.

            Inputs:
                        start_sample - sample number if units of samples since 1970-01-01.  If None,
            only refresh high level metadata.
                        sample_length - number of samples.  If given, only refresh low-level metadata
            near sample extent. If None, only refresh high level metadata.
```

```
                channel - channel to update.  If None, only refresh high level metadata.
        """
```

It should be noted that the the GDF 1.0 and 2.0 format are compatible with the proposed API, allowing us to also support old data formats. Some of the features just wouldn't exist (eg., get_continous_blocks() would simple give index zero and length of acquisition, and metadata would be unchanged throughout one data folder).

High level RF Signal API (write)

# creating an experiment from python

Digital RF data can also be written to this Hdf5 standard using the python digital_rf_hdf5 module. 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 rf data to Hdf5 files as specified
    in the http://www.haystack.mit.edu/pipermail/rapid-dev/2014-February/000273.html email
thread.
    """

    def __init__(self, directory, dtype_str, samples_per_file, files_per_directory,
start_global_index, sample_rate, uuid_str, compression_level=0, checksum=False,
is_complex=True, num_subchannels=1, 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
```

files_per_directory - number files per subdirectory in form YYYY-MM-DDTHH-MM-SS before new subdirectory created

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.

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
        """
```

Close

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

Low level HDF5 format

The C API writes a stream of RF to disk in HDF5 format as either complex or real (i.e., single-valued) numbers. One or more subchannels can be written simultaneously as a block. If more than one subchannel is written, all subchannels must share the metadata defined in that Hdf5 file - for example, the global sample times, the sample rates.

We will attempt to make the C API as simple and minimalistic as possible. The low level API only deals with one stream or channel that is running at a certain sample rate. While we anticipate most radar data to be continuous, the recorded stream can be written in arbitrary sized vectors with arbitrary sized gaps between the vectors. In there are multiple subchannels, 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 rtes, multiple instances of the low level writers are used. See the next section for more details on this.

On the low level, 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 eg., 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.

Each file contains an arbitrary number of continuous blocks of one of more subchannels of RF. All of these blocks are contiguously written to /rf_data. 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 layoput 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. The starting index, and associated global index of each continuous block is indicated in the table /rf_data_index.

Each stream of data is associated with a uuid. How this is created is not yet determined. It could be random, or it could be user defined.

Each file contains the following elements:

/rf_data   # Nx2 or Nx1 vector of data, N samples complex or real
/rf_data_index # Nx2 uint64, mapping between local and global sample indices

The /rf_data dataset has the following 11 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.
/samples_per_file # the number of samples stored in that 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.

The files are written by the low level library in the following way:

2014-03-30T12-35-29/rf@1396379502.000.h5

Every N_stride seconds a new directory is created and files are accumulated in it. The name of the directory is a ISO 8601 format date string, which indicates the leading edge of the first sample in the first data file in this directory. The date is determined from the sample index, not computer time. This 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. How often a new directory is created is a user defined parameter.

The file name rf@1396379502.000.h5 includes in unix seconds the time of the leading edge of the first sample in the file. This time is determined from the sample index, as provided by the sampling hardware, not computer time. See later section on behaviour on leap seconds.

Low level C RF write API

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

Hdf5_write_data_object * digital_rf_create_write_hdf5(char * directory, hid_t dtype_id, uint64_t samples_per_file, uint64_t files_per_directory, uint64_t global_start_sample, double sample_rate, char * uuid_str, int compression_level, int checksum, int is_complex, int num_subchannels, int marching_dots)

/* digital_rf_create_write_hdf5 returns an Hdf5_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 samples_per_file - the number of samples in a single Hdf5 file. Exactly that number of samples will be in that file. samples_per_file will be saved in each Hdf5 file's metadata

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.

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 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:

          Hdf5_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
 */

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 102412453453454324234/2**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