GPU spectrometer operation and file formats

J. Barrett

May 31, 2023

1 Operation at Westford

Currently the operation of the GPU spectrometer at Westford is somewhat ad-hoc and requires access to several computers. These are given in the following table.

Hostname	IP	User	Description
gpu1080	192.52.63.49	oper	GPU spectrometer
antenna-encoder	192.52.63.141	westford	Antenna position readout machine
wfmark5-19	192.52.63.119	oper	Utility and database
thumper3	192.52.63.33	oper	Field system

At the moment, the system is configured to record only a single polarization at a 400MHz sampling rate. The number of spectral points is set to 131072 and the number of spectral averages is fixed at 256. These parameters will be user-adjustable in the future, but for current test purposes will remain fixed.

The basic series of operations to get the GPU spectrometer ready for recording is as follows:

1. Log into the oper account on gpu1080 and run the spectrometer daemon launch script with the command `LaunchSpectrometerDaemon`.

2. Log into the westford account on the antenna-encoder machine. Start a screen session with the command `screen` and then launch the encoder read-out program with the command `/home/westford/encoder_software/encrec`. You can detach the screen session with `ctrl-a` followed by `d`.

3. Log into the oper account on wfmark5-19. Start a screen session and run the command `capture-encoder-log.py`. Detach the screen session.

4. On thumper3 start a new field system session. In the field system window create a new log file with the command: `log=<log-name>`.

5. Then on wfmark5-19 start a new screen session, and run the command to capture the field system log: `capture-fs-log.py 192.52.63.33 </path/to/log-name>`. Typically field system logs are stored to the directory `/usr2/log` on thumper3. Detach the screen session.

6. Log into the oper account on gpu1080 and either start the recording schedule with the command `hosecc.py -f <schedule.xml>`. Or start the client interface with with the command `hoseclient.py`. If you are using the client interface to issue recording commands for a session it is import to start the log to database script in the background with the client command `startlog2db`.

In the GPU spectrometer client interface provides the following commands for interactive sessions.

1. Start recording for an unknown duration without specifying an experiment, source or scan name (defaults will be used): `record=on`.

2. Start recording for an unknown duration and specify an experiment, source and scan name: `record=on:experiment_name:source_name:scan_name`.
3. Start recording at a specific start time and duration and specify an experiment, source and scan name: `record=on:experiment_name:source_name:scan_name:YYYYDDDHHMMSS:duration`. Duration must be given in seconds, and the time in UTC.
4. Stop a recording immediately: `record=off`.
5. Get the current recording state (on/of): `record?`.
6. Start the export of spectrometer log data to the InfluxDB database: `startlog2db`.
7. List available commands: `help`.
8. Quit the client interface: `quit`.
9. Quit the client interface and shutdown the spectrometer daemon entirely: `shutdown`.

Spectrometer data files are stored under the data directory: `/home/oper/software/Hose/install/data/`.

2 File/Software Overview

The binary format of the GPU spectrometer output files is fairly straightforward and consists of a binary header followed by formatted binary data. Meta-data files are provided in JSON format for which many open source libraries are available in a variety of languages. For simplicity the binary spectrum files have been separated from the noise-diode power measurement files. The input/output (I/O) library which allows writing and reading this formatted data to and from files is written in C so that it may be accessible in both C++ code and python. Python bindings to this library are provided as part of the Hose package. This software can be downloaded from the MIT github page with:

```
git clone git@github.mit.edu:barrettj/Hose.git
```

provided you have an account with access.

To compile the code you will need a modern C/C++ compiler[1] with support for the C++11 language extensions, and the cmake build system generator. To compile the base libraries (including the I/O library) with default options, change into the Hose source directory (`<Hose>`) and run the commands:

```
mkdir build
cd build/
cmake ../
make install
```

This will compile and install the I/O library and python bindings to `<Hose>/install/lib`. To ensure that the library is your execution path and accessible to external code, you will need to run the script `hosenv.sh` to set up the proper environmental variables. This is done with:

```
source <Hose>/install/bin/hoseenv.sh
```

This can be added to your `.bashrc` file if needed on a regular basis. Once the environmental variables are set up, you should have access to the I/O python bindings. To test this, open a python or ipython window and type `import hose`, if the module is imported you have compiled and installed the library successfully.

Additional optional dependencies of the Hose software can be enabled when desired through the cmake command line interface if you instead run the command `ccmake` instead of `cmake`. These options and their requirements are beyond the scope of this document but will be addressed later.

At the moment no significant effort has been made to ensure portability (correct endianness, data type size, etc) of the binary formats across disparate system, which is something to take into consideration when operating on GPU spectrometer files on other machines.

[1] This has been tested to work with gcc 4.9.2 and gcc 5.4.0, clang/LLVM should work but is untested.

3 Meta-data File Format

During operation, various sets of meta-data from the GPU spectrometer and field system log files are exported to an InfluxDB database. At Westford these meta-data sets are stored in the database named `"gpu_spec"` in an instance of InfluxDB running on `wfmark5-19`. If the GPU spectrometer is operated through the scheduling program `hosecc.py` or the client interface `hoseclient.py`, meta-data associated with each scan will be collected from the database and stored in a JSON file alongside the binary spectrometer files. This is done to allow later file conversion and interpretation without access to the InfluxDB database.

The meta-data file consists of a list of JSON objects, while the number of objects in a file may vary, each object type has a fixed format. Each meta-data object contains a measurement name, a time stamp, and one or more data fields. The JSON format was chosen to allow ease in parsing and conversion to python dictionary objects. The current list of available data objects (measurements) is:

1. `digitizer_config`
2. `spectrometer_config`
3. `noise_diode_config`
4. `udc_status`
5. `source_status`
6. `recording_status`
7. `data_validity`
8. `antenna_target_status`
9. `antenna_position`

The time stamp is given in UTC with date and time in the following format: YYYY-MM-DDTHH:MM:SS.⟨F⟩Z. The number of digits in the fractional part of the second ⟨F⟩ may vary, some examples are: `2018-06-19T18:23:08.289999872Z` or `2018-06-19T18:20:11.24Z`. The data fields of each object are for the most part self-explanatory, examples of each are given as follows: The digitizer configuration specifies the set-up of the digitizer and mirrors some information that is in the binary spectrum file headers. An example is given below:

```
{
   "fields": {
      "n_digitizer_threads": "2",
      "polarization": "X",
      "sampling_frequency_Hz": "4e+08",
      "sideband": "U"
   },
   "measurement": "digitizer_config",
   "time": "2018-06-19T18:15:38.381999872Z"
}
```

The spectrometer configuration specifies the configuration of the spectrometer and also mirrors some information in the binary spectrum file headers for convenience. An example is:

```
{
   "fields": {
      "fft_size": "131072",
      "n_averages": "256",
      "n_spectrometer_threads": "3",
      "n_writer_threads": "1"
   },
   "measurement": "spectrometer_config",
   "time": "2018-06-19T18:15:38.381999872Z"
}
```

The noise diode configuration specifies the configuration of the noise diode, mirroring information in the binary noise power file headers. An example is:

```
{
   "fields": {
      "noise_blanking_period": "5e-08",
      "noise_diode_switching_frequency_Hz": "80"
   },
   "measurement": "noise_diode_config",
   "time": "2018-06-19T18:15:38.381999872Z"
}
```


The UDC status specifies the LO frequency and input attenuation. The attenuation values are given in dB, the extensions `h` and `v` indicate horizontal and vertical polarizations). An example is given below:


```
{
   "fields": {
      "attenuation_h": 20,
      "attenuation_v": 20,
      "frequency_MHz": 7083.1,
      "udc": "c"
   },
   "measurement": "udc_status",
   "time": "2018-06-19T18:20:11.24Z"
}
```


The source status records the source name and position (right-ascension,declination) as reported from the telescope field system log. An example is:


```
{
   "fields": {
      "dec": "+122328.0",
      "epoch": "2000.",
      "ra": "123049.42",
      "source": "virgoa"
   },
   "measurement": "source_status",
   "time": "2018-06-19T18:23:08.289999872Z"
}
```


The spectrometer recording status indicates the 'on/off' state of the GPU spectrometer. It also records the schedule or user specified experiment, source, and scan names. An example is:


```
{
   "fields": {
      "experiment_name": "ExpX",
      "recording": "on",
      "scan_name": "170-182351",
      "source_name": "SrcX"
   },
   "measurement": "recording_status",
   "time": "2018-06-19T18:23:51.933000192Z"
}
```


The data validity flag is an optional user/schedule defined flag to indicate times where the acquired data is deemed valid or invalid (on/off) as follows.


```
{
   "fields": {
      "status": "off"
   },
   "measurement": "data_validity",
   "time": "2018-03-01T18:26:28Z"
}
```


The antenna target acquisition status indicates if the antenna has acquired the specified source, or is slewing. It records a boolean (yes/no) value as to whether the source is acquired in the beam, also available is a status indicator which describes the actual antenna state (acquired, off-source, re-acquired, etc.).


```
{
   "fields": {
      "acquired": "yes",
      "status": "acquired"
   },
   "measurement": "antenna_target_status",
   "time": "2018-06-19T18:23:49.32Z"
}
```

The antenna position position object provides a time stamped location of the antenna in local coordinates (azimuth and elevation). This measurement is typically generated once a second, but the system can be made to report this data at shorter intervals if necessary.

```
{
   "fields": {
      "az": 90.602188,
      "el": 18.869705
   },
   "measurement": "antenna_position",
   "time": "2018-06-19T18:23:52.84024704Z"
}
```

4 Spectrum File Format

The GPU spectrometer generates binary files to store the averaged spectrum results. At recording time, it is assumed each acquisition will be associated with an experiment, source and scan name, if they are not assigned default values will be used. With these names defined the spectrum files are written to disk in real time and are stored in the directory:

```
<Hose>/install/data/<experiment>/<scan>/
```

with the following naming convention:

```
<aquisition_start_time>_<file_sample_start_index>_<sideband><polarization>.spec
```

For example the first spectrum file recording starting at UNIX UTC epoch second 1528140171 for an upper sideband, X polarization recording would be named:

```
1528140171_0_UX.spec
```

This naming convention was chosen to ensure uniqueness and avoid data loss.

The spectrometer file binary format is defined by the following struct:

```
struct HSpectrumFileStruct
{
   struct HSpectrumHeaderStruct fHeader;
   char* fRawSpectrumData;
};
```

with the binary header format defined by:

```
struct HSpectrumHeaderStruct
{
   uint64_t fHeaderSize;
   char fVersionFlag[8];
   char fSidebandFlag[8];
   char fPolarizationFlag[8];
   uint64_t fStartTime;
   uint64_t fSampleRate;
   uint64_t fLeadingSampleIndex;
   uint64_t fSampleLength;
   uint64_t fNAverages;
   uint64_t fSpectrumLength;
   uint64_t fSpectrumDataTypeSize;
   char fExperimentName[256];
   char fSourceName[256];
   char fScanName[256];
};
```

The binary header elements are described in the following table:

Header element name	data type	nominal size (bytes)	Description
fHeaderSize	uint64_t	8	Total size of the header in bytes (equivalently, offset from the start to the data payload)
fVersionFlag	char array	8	Space for header/file format version tracking
fSidebandFlag	char array	8	Polarization indicator
fPolarizationFlag	char array	8	Sideband indicator
fStartTime	uint64_t	8	Start time in seconds from (UNIX) epoch
fSampleRate	uint64_t	8	Data rate (Hz)
fLeadingSampleIndex	uint64_t	8	Index of the first ADC sample in the file
fSampleLength	uint64_t	8	Number of ADC samples collected for this file
fNAverages	uint64_t	8	Number of spectrums averaged together
fSpectrumLength	uint64_t	8	Number of spectral points
fSpectrumDataTypeSize	uint64_t	8	Size in bytes of spectral point data type
fExperimentName	char array	256	Fixed length array to store experiment name
fSourceName	char array	256	Fixed length array to store source name
fScanName	char array	256	Fixed length array to store scan name

The raw spectrum data is stored as an un-typed binary blob and must be cast to the proper data type to be read properly. To enable reconstruction of the stored format two pieces of information are stored. The first is simply the total size (in bytes) of the data type storing each spectral point. This value is saved in `fSpectrumDataTypeSize`. Second two characters are stored the header version flag to indicate real or complex data, and whether the floating point format has (single) float or double precision. The 8 bytes of the version flag are broken down to: For example the string `"001RF"` denotes what is currently the only supported spectrum

Byte index	0	1	2	3	4	5	6	7
Usage	version number			real/complex	float/double	un-assigned		

format. This encodes the version number `"001"` and indicates the spectrum data consists of real valued (`R`) single precision floats (`F`).

Reading a spectrum file in python using the provided bindings is relatively simple. This is demonstrated in the following ipython snippet:

```
In [1]: import hose

In [2]: spectrum_file = hose.open_spectrum_file("../data/ExpX/156-140822/1528207702_0_UX.spec")

In [3]: spectrum_file.printsummary()
spectrum_file_data :
header :
spectrum_file_header :
header_size : 856
version_flag : 001RF
sideband_flag : U
polarization_flag : X
start_time : 1528207702
sample_rate : 400000000
```

```
leading_sample_index : 0
sample_length : 33554432
n_averages : 256
spectrum_length : 65537
spectrum_data_type_size : 4
experiment_name : ExpX
source_name : SrcX
scan_name : 156-140822
raw_spectrum_data : <ctypes.LP_c_char object at 0x7f1e70948b00>
```

Access to the formatted spectrum data in python should be done using:

```
spectrum = spectrum_file.get_spectrum_data()
```

which handles the unpacking of spectrum data from the file into the correct floating point type for manipulation in python.

5 Noise Power File

Along with the spectrum files, the GPU spectrometer also generates binary files to store information for the relative power measurement using the switched noise diode. For each spectrum file, a noise power file will be written to disk into the same directory using the same naming convention as described in the above section but with a different file extension (`".npow"`).

The noise power file binary format is defined by the following struct:

```
struct HNoisePowerFileStruct
{
    struct HNoisePowerHeaderStruct fHeader;
    struct HDataAccumulationStruct* fAccumulations;
};
```

The noise power header contains much of the same information as the spectrum file header (to maintain independence during later processing) and is described as follows.

```
 struct HNoisePowerHeaderStruct
{
    uint64_t fHeaderSize;
    char fVersionFlag[8];
    char fSidebandFlag[8];
    char fPolarizationFlag[8];
    uint64_t fStartTime;
    uint64_t fSampleRate;
    uint64_t fLeadingSampleIndex;
    uint64_t fSampleLength;
    uint64_t fAccumulationLength;
    double fSwitchingFrequency;
    double fBlankingPeriod;
    char fExperimentName[256];
    char fSourceName[256];
    char fScanName[256];
}
```

To avoid redundancy, only the elements which differ from the spectrum file header are described in the table below:

Header element name	data type	nominal size (bytes)	Description
fAccumulationLength	uint64_t	8	Number accumulation periods stored in the file
fSwitchingFrequency	double	8	The noise diode switching frequency (Hz)
fBlankingPeriod	double	8	The switch transition blanking period (sec)

The noise statistics data is stored in the file element `fAccumulations`, and is formatted according to the following struct:

```
struct HDataAccumulationStruct
{
    double sum_x;
    double sum_x2;
    double count;
```

```
    uint64_t state_flag;
    uint64_t start_index;
    uint64_t stop_index;
};
```

and described in the following table:

Header element name	data type	nominal size (bytes)	Description
sum_x	uint64_t	8	accumulated sum of each ADC sample during the accumulation period
sum_x2	double	8	accumulated sum of x^2 for each ADC sample, x, during the accumulation period
count	double	8	total number of samples accumulated
state_flag	uint64_t	8	indicates the data state (diode on=1, diode off=0)
start_index	uint64_t	8	start index of the first sample in the accumulation period
stop_index	uint64_t	8	stop index of the last sample in the accumulation period

Access to the noise power files is also provided by the python bindings. The following ipython snippet demonstrates reading a noise power file and retrieving the mean ADC sample value of the first accumulation period.

```
In [1]: import hose

In [2]: power_file = hose.open_noise_power_file("../data/ExpX/156-140822/1528207702_0_UX.npow")

In [3]: power_file.printsummary()
noise_power_file_data :
header :
noise_power_file_header :
header_size : 856
version_flag : 001
sideband_flag : U
polarization_flag : X
start_time : 1528207702
sample_rate : 400000000
leading_sample_index : 0
sample_length : 33554432
accumulation_length : 14
switching_frequency : 0.0
blanking_period : 0.0
experiment_name : ExpX
source_name : SrcX
scan_name : 156-140822
accumulations : <hose.hinterface_module.LP_accumulation_struct object at 0x7f40acee2b00>

In [4]: first_accumulation_period = power_file.get_accumulation(0)

In [5]: print first_accumulation_period.get_mean()
19008.6903127
```