FLAG Software guide

Richard Black, Mark Ruzindana, Mitchell Burnett

July 2019

Abstract

This document describes the software used to acquire and process data in the FLAG high performance computers.

History

Version	Date	Notes
1.1	July 2019	Added more details on FLAG software (Mark Ruzindana)
1.0	May 2017	Added FLAG Beamformer Backend Section (Richard Black)

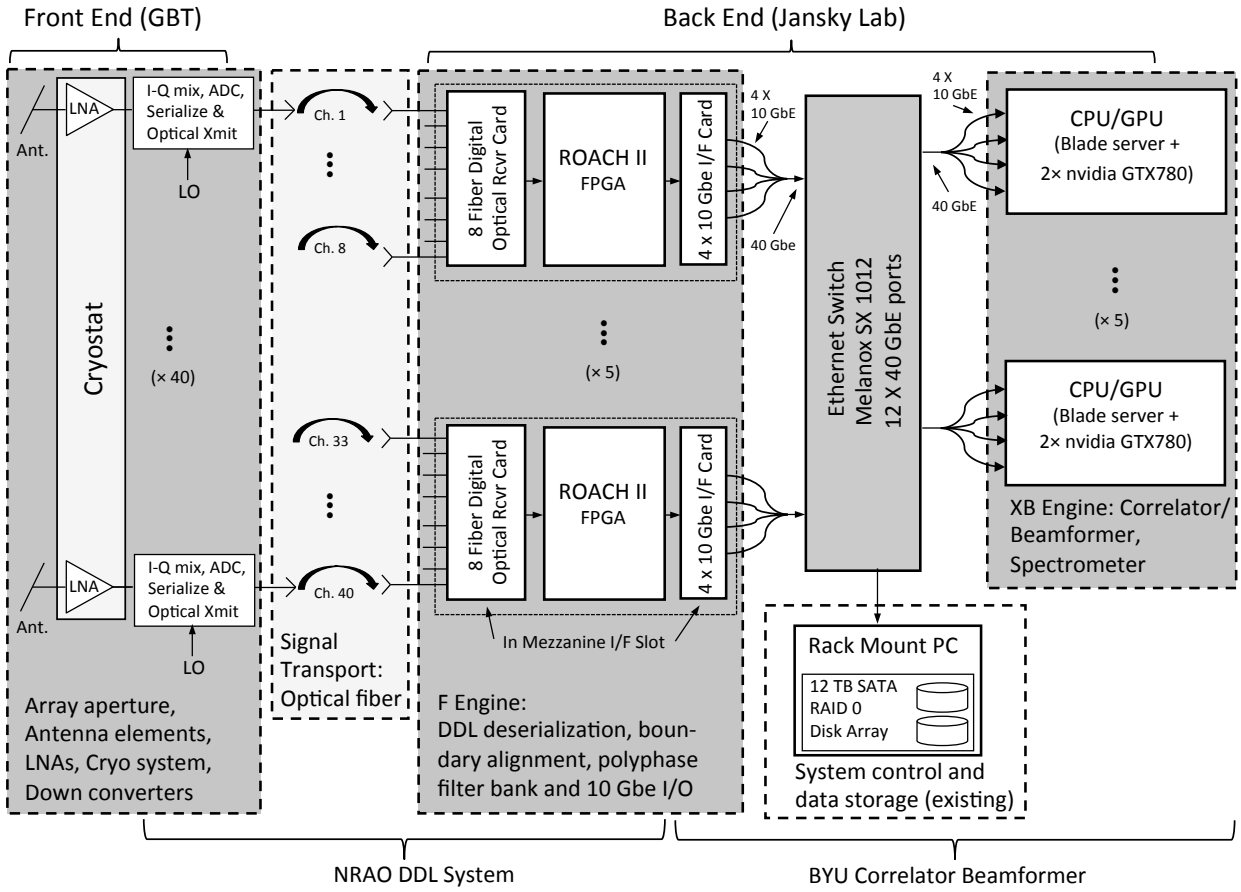

Figure 1: Back End Block Diagram

1 Beamformer Backend

1.1 Introduction

The Focal L-Band Array for the Green Bank Telescope (FLAG) is an in-development 38-element phased array feed (PAF) 150-MHz analog receiver and digital processor for the purposes of HI surveys and radio transient searches. It is expected to be instrumental in discovering more than 50 new pulsars with in the inner galactic plane [fbpn2_pulsarDetection.pdf on beamformer web] and studying diffuse HI around galaxies [fbpn1.pdf on beamformer web].

The entire PAF receiver comes in two parts: the analog receiver (amplification, downconversion, sampling, and signal transport) and the digital processor. A block diagram showing all of the back end is shown in Figure 1. The PAF consists of 19 dual-polarization flared dipoles (see Figure 2) connected to 40 cryogenically cooled low-noise amplifiers (LNAs), two of which are unconnected. The 40 amplified analog bandpass signals are mixed down to baseband, with the inphase (I) and quadrature (Q) components each sampled at 155.52 MHz. The digitized I/Q signals are then serialized and sent over 40 optical fibers to five eight-port optical receiver cards connected to five field-programmable gate array (FPGA) boards each called Reconfigureable Open-Architecture Computing Hardware (ROACH) boards.

The ROACH boards channelize the approximately 150-MHz sampled bandwidth into 512 frequency channels, each with a bandwidth of approximately 303 kHz. The data are then pared down to 500 frequency channels and packetized into 10 user-datagram protocol (UDP) packets each containing 50 frequency samples for eight antennas across 20 time samples. These packets are streamed over 10-GbE/40-GbE breakout cables

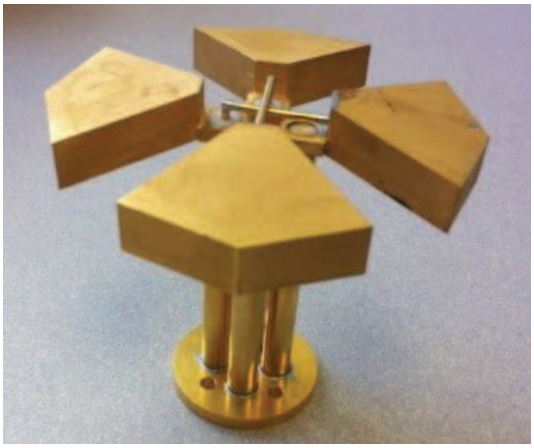

Figure 2: GBT-2 Dipoles

into a 12-port 40-GbE network switch, which redirects packets into five high-performance personal computers (HPCs) such that each HPC receives 100 frequency samples for all 40 antennas. Each HPC then takes these 100 frequency samples and divides them evenly between two graphics processing units (GPUs), which house a real-time beamformer and correlator.

This document intends to explain the inner workings of the HPCs. Section 1.2 describes the different operational modes the HPCs must support for FLAG. Section 1.3 goes over the server's specifications and that of its peripherals (such as GPU card models and specs). Section 1.5 describes the real-time operating system (RTOS) used for thread management and pipelining known as HASHPIPE (or #PIPE). Section 1.6.1 documents the GPU-based real-time correlator library xGPU and polyphase fine-filterbank (PFB) channelization library Grating. Section 1.7 revisits the various operational modes from Section 1.2 with great implementation details.

1.2 Operational Modes

There are two primary modes of operation: coarse PFB mode, fine PFB mode and real-time beamformer. The coarse PFB mode produces spatial covariance matrices across 303-kHz channels for radio transient searches and PAF beamformer weight calibration, and the fine PFB mode correlates across 160 9.5-kHz channels for HI surveys. The real-time beamformer that forms seven beams and accumulates across coarse frequency channels every 0.1 ms. A full block diagram is depicted in Figure 3.

1.2.1 Coarse PFB Mode

In this mode, each GPU receives 50 coarse channels for all 40 antenna elements, totaling 15 MHz of bandwidth. For PAF calibration, correlation matrices are computed for all 50 channels and accumulated for a calibration-specified integration length, where 303 samples would correspond to 1 ms. For transient searches, matrices are computed for only five channels for 0.1 ms, or 30 samples. Only five channels totaling a bandwidth of 1.51 MHz are computed since the memory bandwidth demand for covariance dumps every 0.1 ms approaches the specified limits of the GPU cards.

A real-time beamformer is also run with the coarse PFB, forming seven beams on all 50 coarse channels and dumping every 0.1 ms.

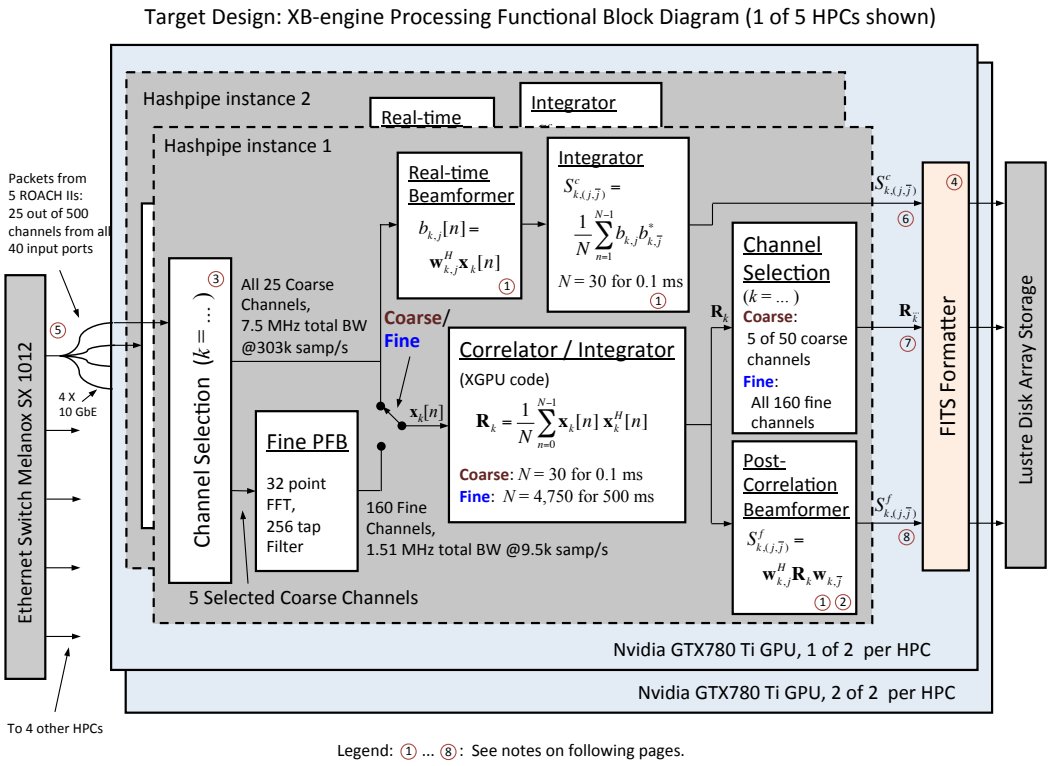

Figure 3: Block diagram describing the various operational modes.

1.2.2 Fine PFB Mode

This mode has the same structure as the coarse PFB mode except that it includes a fine PFB module for further channelization prior to the correlator.

The PFB channelizes five coarse frequency channels (303 kHz each, totaling 1.51 MHz) into 160 finer channels (9.5 kHz each, totaling 1.51 MHz). These are then correlated, and matrices are dumped every 0.5 seconds or approximately 4,734 samples. All 160 fine-channel correlation matrices are loaded off the GPU for FITS file formatting since the relatively large integration time reduces the required memory bandwidth.

1.3 HPC Specifications

Each HPC is a Mercury GPU408 4U GPU Server purchased from Advanced HPC (part number AH-GPU408-SB14). A photo of one of these HPCs is shown in Figure 4. The machine comes with two Intel Xeon six-core processors organized onto two independent PCI-E busses. A table of specifications for each HPC can be found in Table 1. The PCI-E slots are populated with two GPU cards, two dual 10-GbE SFP+ network interface controller (NIC) cards, and an Infiniband NIC card, with part designations summarized in Table 2.

1.3.1 Graphics Cards

There are two flavors of graphics cards used in the FLAG back end: the NVIDIA GeForce GTX 780 Ti and NVIDIA GeForce GTX 980 Ti. Two of the GPUs are the 780 Ti model since they were purchased just prior to their abrupt removal from the market by NVIDIA. Pertinent specifications for both cards are summarized in Tables 3 and 4. The differences between the two cards in all critical performance specifications are minimal, thus we do not anticipate any new throughput bottlenecks resulting from the swap.

Figure 4: A photo of one of the five HPCs

Table 1: HPC Specifications

Feature	**Description**
Processors	2x Intel Xeon E5-2630 v2 2.60 GHz six-core 80 W processors
Memory	32 GB DDR3 ECC
Hard Drives	2x 500 GB SATA 7200 RPM disks in RAID 1 configuration
Drive Bays	8x hot-swappable 3.5 inch drive bays (unpopulated)
PCI-E Slots	4x PCI-E 3.0 x16 slots (double width, two per processor)
	2x PCI-E 3.0 x8 slots (one per processor)
	1x PCI-E 2.0 x4 slot
Network	Integrated Intel i350 dual port GbE LAN
Power Supply	1620 W platinum level efficiency redundant power supply
Height	4U
Rack Mountable	Yes

Table 2: PCI-E Slot Population (out-of-date)

Slot Designation	**Populated With**
CPU 1 Slot 2*	Mellanox QSFP 40-GbE NIC
CPU 1 Slot 4*	EVGA GeForce GTX 780/980 Ti Graphics Card
CPU 2 Slot 6*	Mellanox Infiniband NIC Card (unknown part as of yet)
CPU 2 Slot 8*	EVGA GeForce GTX 780/980 Ti Graphics Card
CPU 2 Slot 9^{+}	Unpopulated
CPU 1 Slot 10^{+}	Unpopulated
CPU 2 Slot 11†	Unpopulated

* PCI-E 3.0 x16 slot. $^{+}$ PCI-E 3.0 x8 slot. † PCI-E 2.0 x4 slot.

* Each of these slots are double wide (to allow for the size of the GPU cards), resulting in missing slot numbers 1,3,5, and 7.

Table 3: GeForce GTX 780 Ti Specifications

Specification	**Value**
# CUDA Cores	2880
Base Clock Speed	875 MHz
Texture Fill Rate	210 GigaTexels/sec
Memory Clock Speed	7.0 Gbps
Memory	3 GB
Memory Interface Width	384 bits
Memory Bandwidth	336 GB/sec
Minimum Power Requirement	600 W

Table 4: GeForce GTX 980 Ti Specifications

Specification	**Value**
# CUDA Cores	2816
Base Clock Speed	1000 MHz
Texture Fill Rate	176 GigaTexels/sec
Memory Clock Speed	7.0 Gbps
Memory	6 GB
Memory Interface Width	384 bits
Memory Bandwidth	336.5 GB/sec
Minimum Power Requirement	600 W

1.3.2 Network Cards

Each HPC is equipped with three network cards: two dual-port SFP+ 10-GbE cards for receiving packets from the ROACH boards and an Infiniband Card for fully processed data transfers to a Lustre disk array (see Figures 3).

1.4 Code Repositories

The following links go to the code repositories:

- HASHPIPE and CUDA code - `https://gitlab.ras.byu.edu/ras-devel/flag`
- Dealer/player code - `https://gitlab.ras.byu.edu/ras-devel/beamformer-back-end`
- MATLAB post-processing - `https://gitlab.ras.byu.edu/ras-devel/matlab-post-processing`
- FITS writer code - `https://github.com/nrao/FLAG-Beamformer-Devel/tree/master/src`

1.5 HASHPIPE

HASHPIPE is an RTOS that specializes in pipeline processing by splitting consecutive tasks into separate threads with semaphore-controlled shared memory buffers in between each thread. A generic HASHPIPE program block diagram is shown in Figure 5. Each task in a process is given its own parallel thread with input and output shared memory buffers.

Each buffer is circular and consists of a number of blocks of data, where each block is usually the minimum size of data needed by the next threads. Each block has an associated semaphore that is initialized to zero (unavailable/free) and is set to one (available/filled) when the block contains valid data and is not being used by any other thread. If the block's semaphore is set to zero, any thread that requests that block's data will hang until the semaphore becomes available.

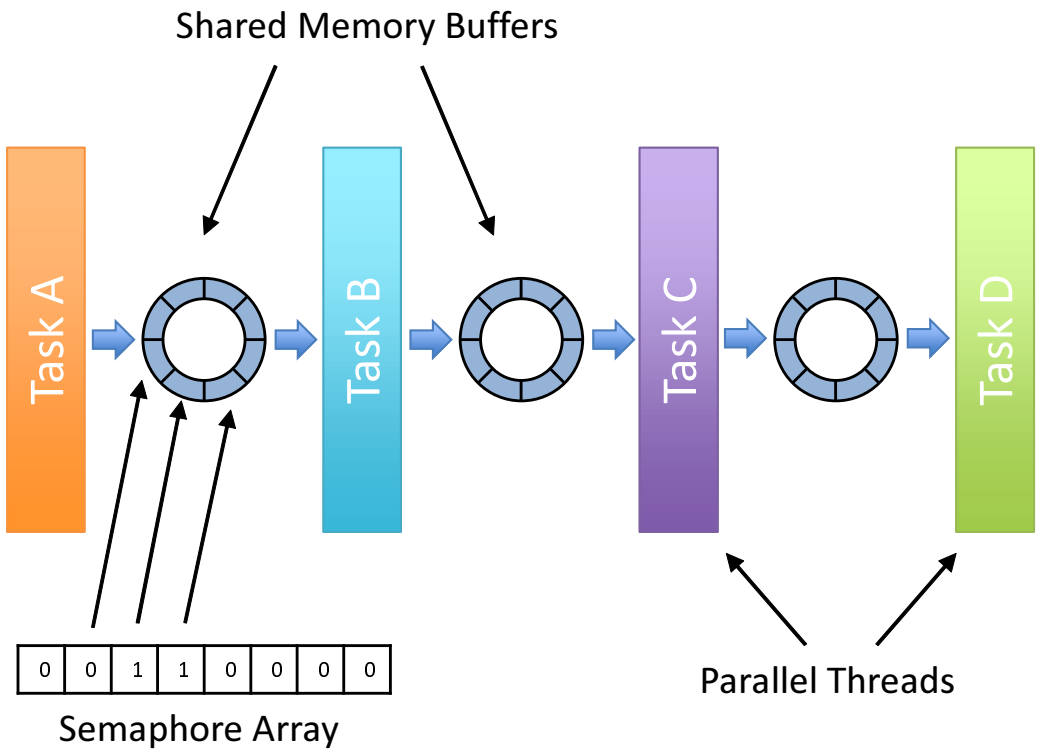

Figure 5: A generic HASHPIPE program.

1.5.1 HASHPIPE Plugins

When HASHPIPE is run on the command line as-is, it will result in nothing since it requires a plugin that specifies the application to be run. In a sense, a HASHPIPE plugin is analogous to a program run on any other conventional RTOS, but is distinct in that a HASHPIPE program is in the form of a shared library. Thus, linking a shared library to HASHPIPE at run-time is like "plugging in" the software; hence the "plugin" terminology.

A HASHPIPE plugin is basically a shared object that defines various threads and shared memory buffer parameters. Each thread is encapsulated in a single .c file that contains a constructor function that is called by hashpipe when its thread is created. An example constructor is shown below.

```
static __attribute__((constructor)) void ctor() {
    register_hashpipe_thread(&thread_desc);
}
```

The argument `thread_desc` is a `hashpipe_thread_desc_t` struct defined in the thread's .c file as follows:

```
static hashpipe_thread_desc_t thread_desc = {
    name: "thread_name", // Name of the thread
    skey: "KEYNAME", // Thread status memory keyword
    init: init_func, // Name of function to run when initializing the thread
    run:  run_func, // Name of main_loop function
    ibuf_desc: {input_buffer_create_func}, // Name of input buffer creation function
    obuf_desc: {output_buffer_create_func} // Name of output buffer creation function
};
```

After compilation, a shared library object (.la) is generated that represents this pipeline with its various threads and semaphores. When the library is placed in a directory found in the LD_LIBRARY_PATH environment variable, it can be executed using the following at a terminal:

```
$ hashpipe -p <library_name>
```

where `<library_name>` is the shared library file name without its .la extension. To run multiple simultaneous instances, one can use the `-I <instance_num>` flag. Shared memory status values can be set using the `-o <KEY>=<value>` flag. Lastly, core affinities for the various threads are set using the `-c <core_num> <thread_name>` flag. For example,

```
$ hashpipe -p flag_beamformer -I 0 -o XID=0 -o BINDHOST=10.10.10.1 -o GPUDEV=0
      -c 0 flag_net_thread -c 1 flag_transpose_thread -c 2 flag_beamformer_thread
```

A simple "Hello World" HASHPIPE plug-in can be found in Appendix A.

1.6 GPU Libraries

The hashpipe plugins use shared libraries that wrap GPU kernel methods.

1.6.1 xGPU

The correlator is based on the open-source GPU library xGPU hosted on GitHub [ref to github repo]. It is a highly optimized code that parallelizes the correlator process by computing several two-by-two correlations and accumulating. The number of antenna inputs must be a multiple of 32. The library is compiled to be C compatible so that it can be linked to hashpipe. The output data structure is a block lower-triangular matrix with two-by-two block matrices in row-major order. The block entries are ordered similarly.

There are three types of correlators that this library must support: (1) a coarse-channel correlator with 8-bit input samples, (2) a fine-channel correlator with floating-point input samples, and (3) a coarse-channel rapid-dump, reduced-bandwidth correlator with 8-bit input samples. To accommodate these various modes, multiple versions of the library are compiled. The `xgpu.so` library supports the coarse-channel correlator; the `xgpu_pfb.so` library supports the find-channel correlator; and the `xgpu_frb.so` library supports the rapid-dump correlator.

1.6.2 Total Power

The total power code computes the time-average power in the entire band on a per-element basis, or

$$P_m = \frac{1}{N} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} |x_{m,k}[n]|^2, \tag{1}$$

where m is the element index, $0 \leq m \leq M-1$, N is the total number of time samples in the integration, k is the frequency channel index $0 \leq k \leq K-1$, and $x_{m,k}[n]$ is the mth element's complex voltage in the kth frequency channel at the nth time sample. For the FLAG back end, $M = 64$, $K = 25$ for each pipeline, and $N = 4000$.

1.6.3 Beamformer

The beamformer computes seven dual-polarization beams across the entire bandwidth with a 40-sample accumulation length resulting in 100 short time integration (STI) windows over 4000 samples. Formulaically, this is

$$P_{k,l,b,p,q} = \frac{1}{N} \sum_{n=0}^{N} \mathbf{w}_{k,b,p}^H \mathbf{x}_k[n + lN] \mathbf{x}_k^H[n + lN] \mathbf{w}_{k,b,q}, \tag{2}$$

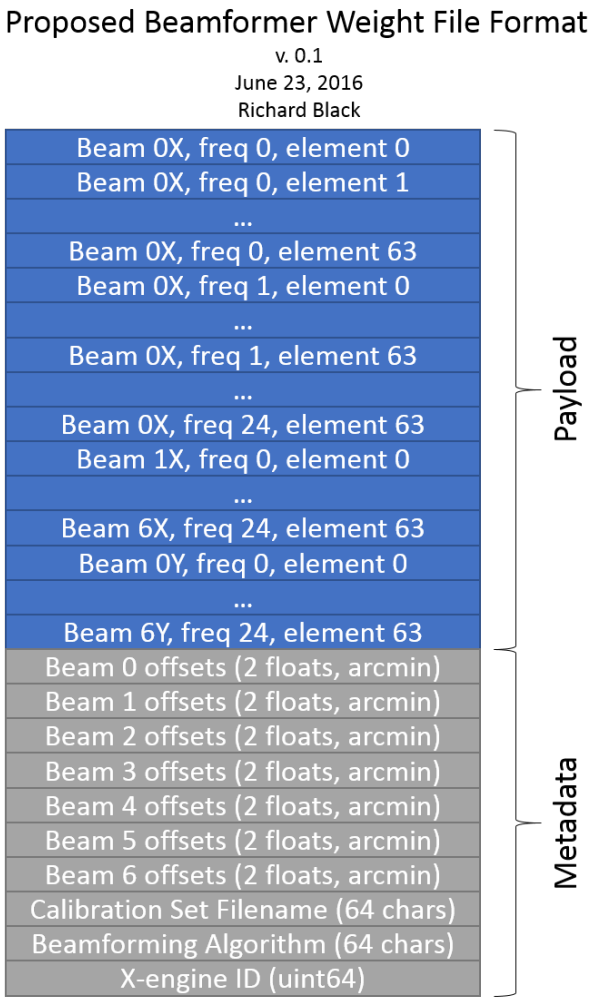

Figure 6: Caption

where $\mathbf{w}_{k,b,p}$ is the beamformer weight vector for the bth beam with polarization p and the kth coarse frequency channel, and l is the STI index.

The weights are stored in a file that is read by the beamformer module, and is formatted as depicted in Figure 6.

The beamformer output dimensions from fastest to slowest moving are beam, polarization, frequency channel and STI index. The number of beams, frequency channels, STI indices/windows are left as stated earlier, but there are 4 polarizations; the self polarizations (xx^* and yy^*) and the cross polarizations (xy^* and yx^*). This can be seen in equation 2.

1.6.4 Polyphase Filter Bank Channelization

In the fine channel correlator mode, only one-fifth of the 500 coarse frequency channels are processed for a zoomed spectrum. Each Hashpipe instance implementing a PFB across the five HPCs is therefore specified to process 5 coarse frequency channels for all 40 elements, using 4000 decimated time samples per data window as input. This results in 5 (coarse channels) x 40 (antenna elements), or 200 decimated time series that need to be independently filtered by a PFB.

Data parallelism allows us to collapse the need for 200 iterations of a PFB into one call to a PFB kernel that runs all 200 iterations simultaneously. This is a powerful capability, because the number of decimated

time series can scale and increase in dimension by either increasing the number of processed frequency bins, or antenna elements and all time series are processed simultaneously, substantially decreasing computation time. As long as there are sufficient GPU resources available to process all of the time series, the compute response time is as if only one PFB is being called.

1.7 Implementation Details

In this section, we expand on the implementation details of the various libraries used in hashpipe. These include xGPU, the total power library, the real-time beamformer library, and the polyphase fiter bank library.

1.7.1 xGPU

The library is compiled to be C compatible so that it can be linked to HASHPIPE. The `xgpu.so` library supports the coarse-channel correlator; the `xgpu_pfb.so` library supports the fine channel correlator; and the `xgpu_frb.so` library supports the rapid-dump correlator.

1.7.2 Total Power

The total power code uses several subsequent GPU kernels to fully reduce the data into element powers. There are three reduction procedures:

1. Get power and reduce blocks of 1024 time samples
2. Reduce the results from each block in 1
3. Reduce the results from 2 across frequency channels

1.7.3 Beamformer

The real-time beamformer is implemented on a GPU similar to the total power code. The specifications for the beamformer are as follows; 19 dipole elements, 25 coarse channels with 15MHz total bandwidth at a sample rate of 303k samp/s, and 7 dual polarization beams. In order to achieve real-time, the total duration of beamforming, and integration must be at most the number of samples (N) divided by the sample rate. The two processes that utilized the GPU the most were beamforming and integration. A data restructure was also processed by the GPU in order to accommodate for a function used by the beamformer. This document describes the functions used to run the real-time beamformer. The code was structured in such a way that it could be easily integrated into HASHPIPE.

The functions used as well as their descriptions are as follows;

init_beamformer() - This function allocates memory to all the arrays used in the code. It also sets up all the arrays used by the *cublasCgemmBatched()* function.

*update_weights(char * filename)* - As the functions name implies, this function updates the weights, but it also transposes the weights to accommodate for the *cublasCgemmBatched()* function. The file name of the weights file is required as an input.

*data_restructure(signed char * data, cuComlplex * data_restruc)* - A kernel that restructures the data to accommodate for the *cublasCgemmBatched()* function and replace the transpose thread in HASHPIPE. The array of pointers, *data*, is required as an input of the kernel and *data_restruc* is its output.

*signed char * data_in(char * input_filename)* - Reads the data file, and returns it as an array of pointers. The file name of the data file is required as an input.

void beamform() - Contains the *cublasCgemmBatched()* function that performs the beamforming operation. The *cublasCgemmBatched()* function is found in the cublas library and performs a matrix-matrix multiplications of an array of matrices.

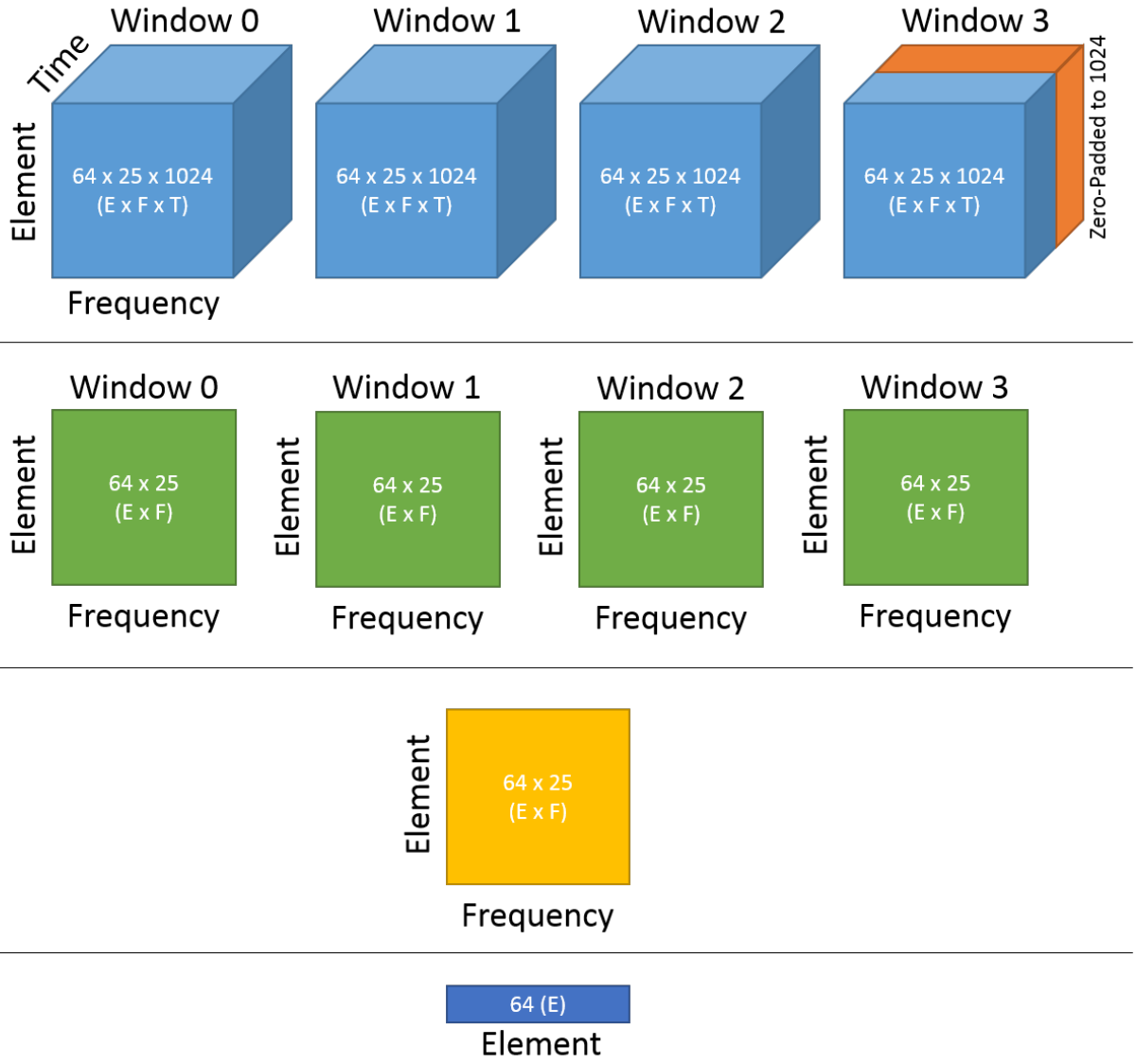

Figure 7: The data as it is manipulated by the total_power code.

*void sti_reduction(cuComplex * data_in, float * data_out)* - A kernel that performs the short time integration using a reduction algorithm commonly used on GPUs to reduce the number of threads per block. The kernel requires the output of the *beamform()* function as its input, *data_in*, and produces an output, *data_out*.

*void run_beamformer(signed char * data_in, float * data_out)* - This function calls the data restructure, beamform and sti reduction functions. The returned value of *data_in()* is required as its input, and its output is the short time integration kernel's output.

1.7.4 Polyphase Filter Bank

Pseudo-code for host PFB implementation:

```
1 void runPFB(char* dataIn_h, float2* dataOut_h, dim3 gridDim, dim3 blockDim)
2 // initialize local variables and flags
3
4 // copy data to device
5
6 // run PFB filtering kernel
7 PFB_kernel<<<gridDim, blockDim>>>(dataIn_d, FFTIn_d);
8 // perform FFT
9 while(!doneFFT)
10 doFFT();
11 countFFT++;
12 // update FFT data pointers
13
14 // check exit condition
15 if (countFFT > N_windows)
16 doneFFT = 1;
17
18
19 // copy data from device to host
20
21 return;
22
23
24 int main()
25 // initialize main variables and flags
26
27 //setup grid and block dimensions
28 dim3 gridDim(N_time_series, N_windows, 1);
29 dim3 blockDim(NFFT, 1, 1);
30 // initialize host and device memory
31
32 // run PFB
33 runPFB(dataIn_h, dataOut_h, gridDim, blockDim);
34 return 0;
35
```

1.8 HASHPIPE Threads

For each specified configuration, there is an associated HASHPIPE plugin, named as follows:

1. `flag_bx.la` → Coarse Channel, Rapid Dump, Reduced Bandwidth Correlator/Coarse Beamformer
2. `flag_bfx.la` → Fine Channel Correlator/Coarse Beamformer

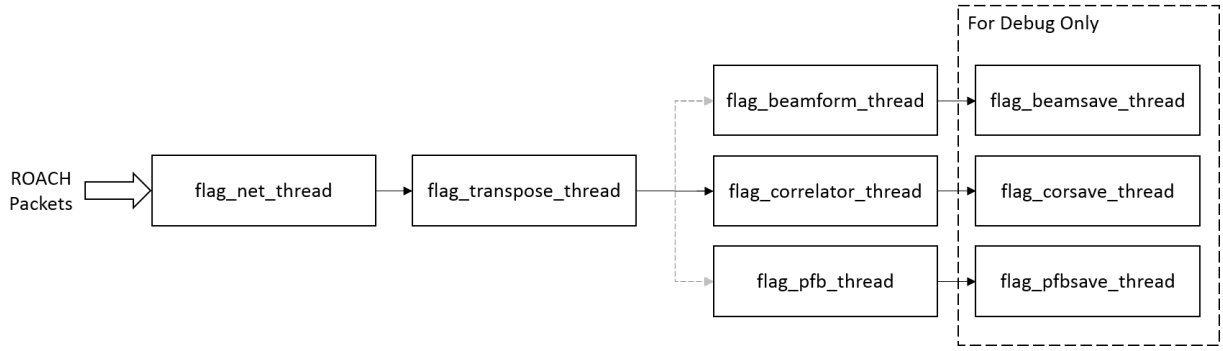

Figure 8: The HASHPIPE thread layout for the FLAG back end.

There are also other reduced capability configurations that are available for simple tests or diagnostics, which are named:

1. `flag_x.la` → Coarse PFB Correlator
2. `flag_x_frb.la` → Coarse PFB Rapid Dump, Reduced Bandwidth Correlator
3. `flag_b.la` → Coarse Beamformer
4. `flag_f.la` → Fine PFB
5. `flag_fx.la` → Fine PFB Correlator

These HASHPIPE plugins use many threads, but they all align into roughly the same structure, shown in Figure 8. In essence, the first thread must always be a network sniffing thread, which listens on the network for packets from the ROACH boards. The second thread must always be a transpose thread, which reformats the data so that all time samples are co-located in memory (details later). The next thread should be selected based on desired operational mode, and the final thread is a save-to-disk thread for debugging purposes. If the FITS writers are being used, the fourth thread should be omitted.

We here enumerate the threads used in each configuration:

The `flag_bx.la` Mode

1. `flag_net_thread`
2. `flag_bx_transpose_thread`
3. `flag_bx_thread`
4. (For Debugging) `flag_bx_save_thread`

The `flag_bfx.la` Mode

1. `flag_net_thread`
2. `flag_transpose_thread`
3. `flag_bfx_thread`
4. (For Debugging) `flag_bfx_corsave_thread`

The `flag_x.la` Mode

1. `flag_net_thread`
2. `flag_transpose_thread`

3. `flag_correlator_thread`
4. (For Debugging) `flag_corsave_thread`

The `flag_x_frb.la` Mode

1. `flag_net_thread`
2. `flag_frb_transpose_thread`
3. `flag_frb_correlator_thread`
4. (For Debugging) `flag_frb_corsave_thread`

The `flag_b.la` Mode

1. `flag_net_thread`
2. `flag_transpose_thread`
3. `flag_beamform_thread`
4. (For Debugging) `flag_beamsave_thread`

The `flag_f.la` Mode

1. `flag_net_thread`
2. `flag_transpose_thread`
3. `flag_pfb_thread`
4. (For Debugging) `flag_pfbsave_thread`

The `flag_fx.la` Mode

1. `flag_net_thread`
2. `flag_transpose_thread`
3. `flag_pfb_thread`
4. `flag_pfb_correlator_thread`
5. (For Debugging) `flag_pfb_corsave_thread`

This section will review each thread and its functionality.

1.8.1 flag_net_thread

The `flag_net_thread` listens for incoming UDP packets from the ROACH boards and stores them in a buffer for further processing. It also handles incoming messages from the Player and subsequently commands the remaining threads in the process.

The internal processing structure is a state machine, which is depicted in Figure 9. Here, the thread waits in an `IDLE` state until the Player issues a "start" command through `stdin`. Once the "start" command is issued, the thread enters the `ACQUIRE` state in which it begins to listen for ROACH packets and stores any that it finds. When it stores a block of data whose starting frame counter (`mcnt`) is greater than or equal to the specified last frame counter (i.e. scan length), or when the Player issues a "stop" command, the thread transitions into a `CLEANUP` state. In `CLEANUP`, the thread issues the same command through shared memory to the other threads and reinitializes the input buffer and all relevant counters to prepare for a new scan. Then the thread returns to `IDLE` to wait for a new "start" command.

Since the full-data-rate system sends packets through a network switch, out-of-order packets are very common. Consequently this thread incorporates a large output buffer with many blocks, where each block contains 200 packets worth of data, which translates to 4000 complex time samples of 25 frequency channels across 64 antenna elements. The data are ordered in the buffer according to the diagram shown in Figure 10.

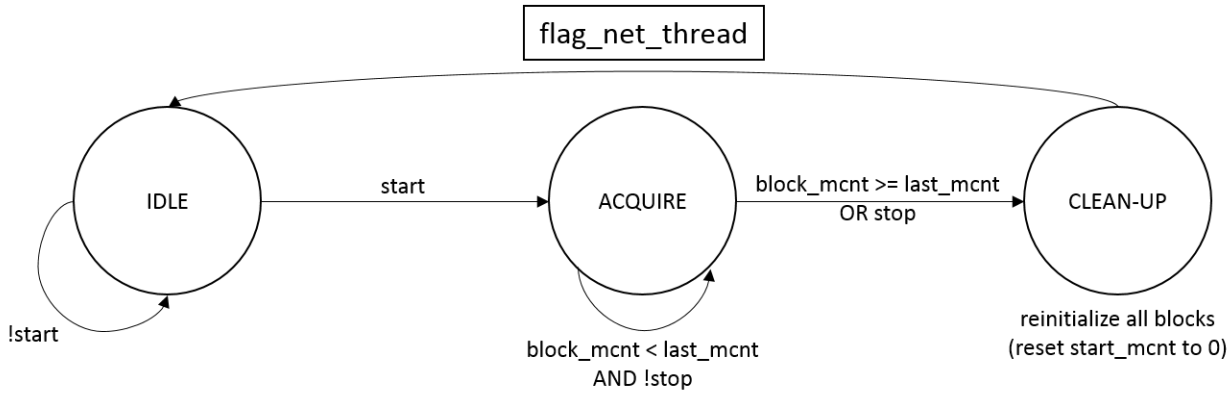

Figure 9: State machine used in net thread.

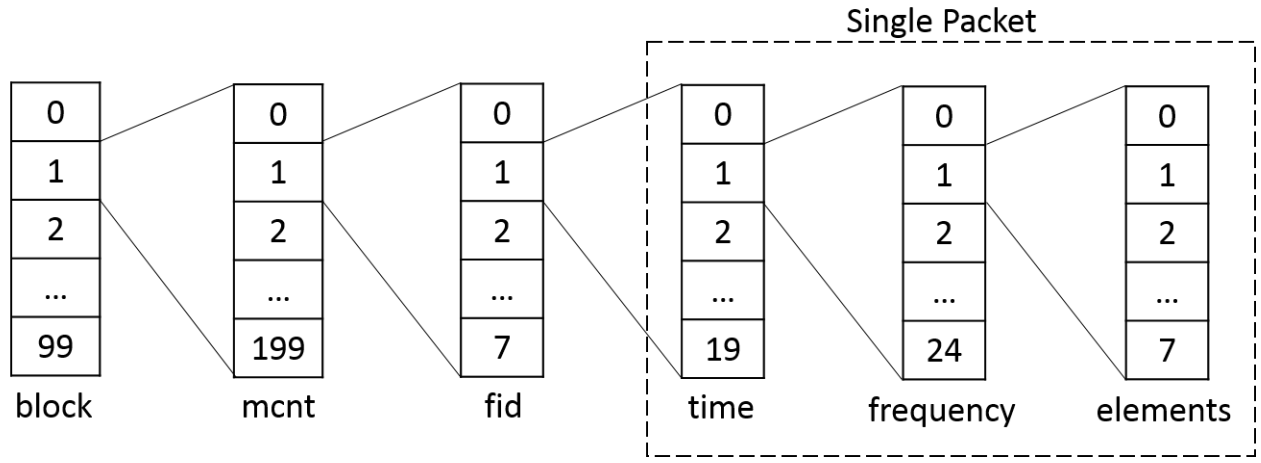

Figure 10: The data coming out of the `flag_net_thread` are ordered such each block contains 200 frames, which each contain eight ROACH packets. Each packet contains 20 time samples of 25 frequency channels across eight antenna elements.

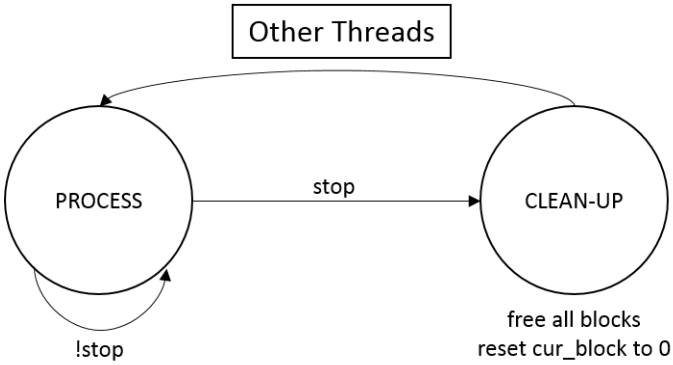

Figure 11: State machine used in other threads.

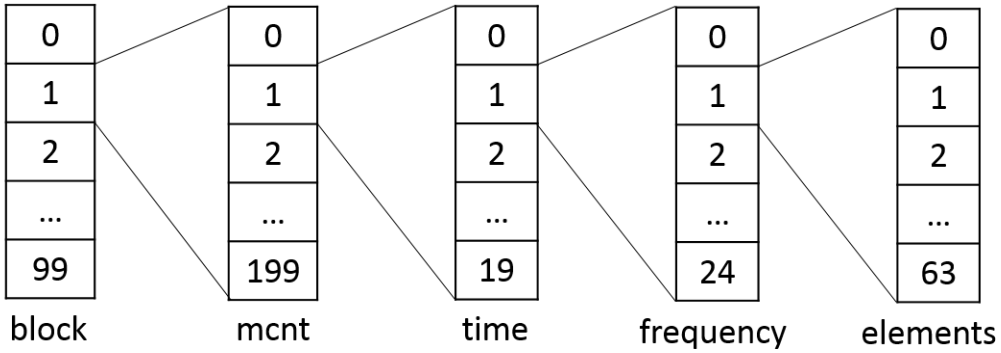

Figure 12: The data format coming out of the `flag_transpose_thread`. Each block contains 4000 time samples of 25 frequency channels across 64 antenna elements.

1.8.2 flag_transpose_thread

Unlike from `flag_net_thread`, this thread uses a much simpler state machine, depicted in Figure 11. Here, the thread resides in an `ACQUIRE` state in which it waits for the input buffer to become populated and reformats the data so as to aggregate the time samples together as depicted in Figure 12. When the `flag_net_thread` goes into a `CLEANUP` state, this thread also enters a `CLEANUP` state in which it resets all relevant counters.

1.8.3 flag_pfb_transpose_thread

As with the `flag_transpose_thread`, this thread uses the same state machine depicted in Figure 11. The operation of the this thread is the same as that of `flag_transpose_thread` except that the output consists of only five of the total 25 frequency channels. One can control which five frequency channels are processed by writing a value to the shared memory status keyword `CHANSEL`, where a value of k, $0 \leq k \leq 4$ yields channels $5k$ through $5k + 4$. These outputs are then marked as ready for the `flag_frb_correlator_thread` to process.

1.8.4 flag_frb_transpose_thread

As with the `flag_transpose_thread`, this thread uses the same state machine depicted in Figure 11. The operation of the this thread is the same as its non-FRB counterpart above except that the output consists of only five of the total 25 frequency channels and is split into 200 blocks per one input block. This creates output blocks that span 40 time samples, or approximately 0.13 milliseconds. One can control which five frequency channels are processed by writing a value to the shared memory status keyword `CHANSEL`, where

a value of k, $0 \leq k \leq 4$ yields channels $5k$ through $5k + 4$. These outputs are then marked as ready for the `flag_frb_correlator_thread` to process.

1.8.5 flag_bx_transpose_thread

As with the other `transpose` threads previously described, this thread uses the same state machine depicted in Figure 11. This thread sources two output buffers, the first for the real-time beamformer operation and the second for the fast-dump, reduced-bandwidth correlator managed by the `flag_bx_thread`. The beamformer's buffer is organized as described in Section 1.8.2, and the correlator's buffer is organized as described in Section 1.8.4

1.8.6 flag_pfb_thread

This thread accesses the 8-bit data outputeed by the flag pfb transpose thread and calls the PFB GPU codes to further channelize the data. The user can specify before run-time the filter length and window type by using other utilities found with the PFB library. Details about the PFB library are beyond the scope of this work and can be found in Mitchell Burnett's thesis.

1.8.7 flag_correlator_thread

This thread accesses the 8-bit data output by the `flag_transpose_thread` and calls the xGPU process that will correlate the data. The user can specify an integration length by writing the desired number of seconds to the `REQSTI` value of the status shared memory. The integration length must be a multiple of 4000 time samples, or approximately 13 ms, so the code rounds up to the nearest multiple of 4000 time samples. As with the `flag_bx_transpose_thread`, this thread uses the state machine depicted in Figure 11. The correlation matrix format has the same structure depicted in Figure 13.

1.8.8 flag_frb_correlator_thread

This thread accesses a single block from the buffer populated by the `flag_frb_transpose_thread` and calls the xGPU process that will correlate the data. In this thread, the xGPU process is linked to the `xgpu_frb.so` library, which supports only five frequency channels and processes blocks of 40 time samples. As with the `flag_correlator_thread`, the output data format is depicted in Figure 13.

1.8.9 flag_pfb_correlator_thread

This thread computes spatial correlation matrices for 160 fine frequency channels provided by the `flag_pfb_thread`. Here, the xGPU process is linked to the `xgpu_pfb.so` library, which expects floating point complex values for 160 frequency channels in 4000-time-sample blocks. As with the other correlator threads, the output data format is given by the format in Figure 13.

1.8.10 flag_beamform_thread

This thread accesses the data output by the `flag_transpose_thread` and processes it using the real-time beamformer GPU kernel. To this end, it manages the beamformer weights and populates shared memory with relevant metadata, which is itemized in Table 5. The thread also uses the state machine from Figure 11. The output data product format is a three-dimensional array with dimensions of $25 \times 4 \times 7$ (frequency channel $\times$ polarization $\times$ beam index), where beam index is the fastest changing dimension followed by polarization ($XX \rightarrow YY \rightarrow XY$).

1.8.11 flag_bx_thread

This thread creates two sub-threads of its own: the first implements the real-time fast-dump correlator as used in the `flag_frb_correlator_thread`, and the second implements the real-time beamformer as used in the `flag_beamform_thread`. To accommodate the different output data products, two data buffers are

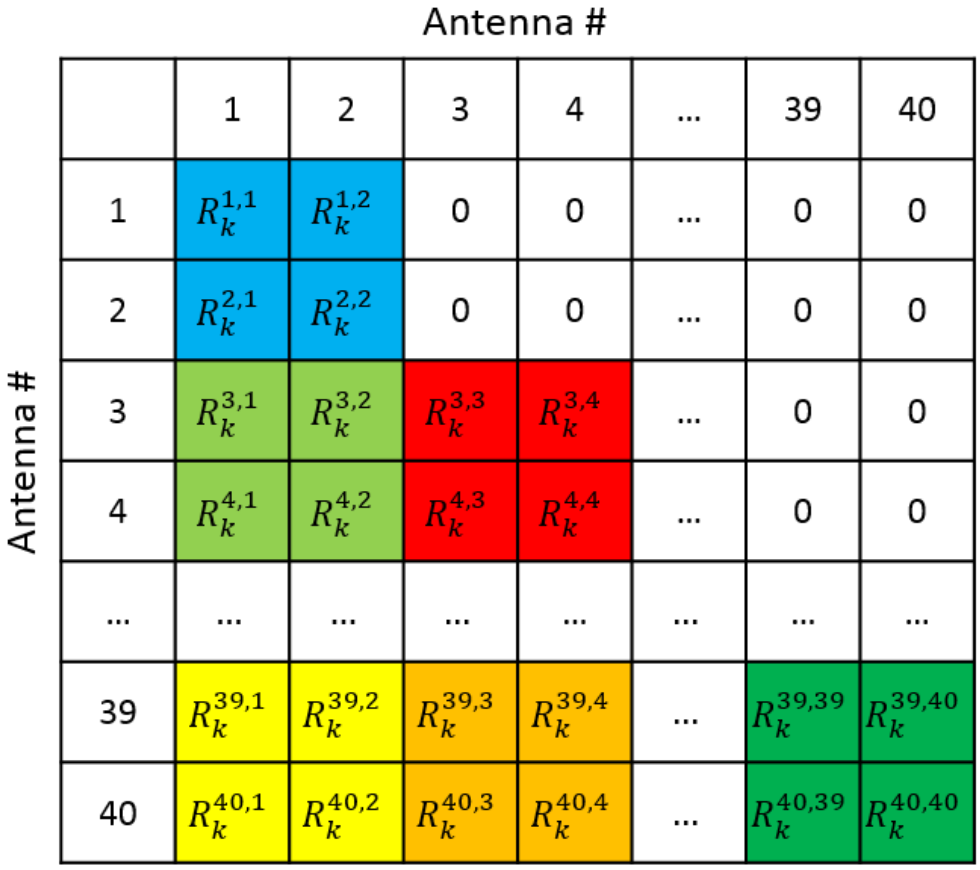

Figure 13: The data format coming out of the `flag_correlator_thread`.

Table 5: Real-Time Beamformer Metadata

Keyword	**Description**
ELOFFX	The elevation offset for beam number X (arcminutes)
AZOFFX	The azimuth offset for beam number X (arcminutes)
BCALFILE	The beamformer calibration data set file name
BALGORIT	The algorithm used in computing the beamformer weights
BWEIFILE	The beamformer weight file name

created, the first being the default associated with the thread and the second created manually before the two sub-threads are created. The two sub-threads independently use the state machine from Figure 11.

1.8.12 flag_bfx_thread

This thread is not yet implemented, but is meant to take data from the flag transpose thread and apply the real-time beamformer, fine-channelization PFB, and correlator in parallel on the data. This single thread approach being developed by Jeff Nybo, an M.S. student at BYU.

1.8.13 flag_corsave_thread

This optional thread must follow the `flag_correlator_thread`. It saves the correlator outputs defined in Figure 13 across 20 frequency channels into a raw text format, with each new line containing a single floating-point sample. Every other sample is real, and the remaining samples are imaginary.

1.8.14 flag_frb_corsave_thread

This optional thread must follow the `flag_frb_correlator_thread`. It saves the short-time dumps of the correlator formatted according to Figure 13 across five fine frequency channels into a raw text format, with each new line containing a single floating-point sample. Every other sample is real, and the remaining samples are imaginary.

1.8.15 flag_pfbsave_thread

This optional thread must follow the flag pfb thread. It saves the PFB channels into a raw text format, with each new line containing a single floating-point sample. Every other sample is real, and the remaining samples are imaginary. The exact format of the data is beyond the scope of this work and can be found in Mitchell Burnett's thesis.

1.8.16 flag_pfb_corsave_thread

This optional thread must follow the `flag_pfb_correlator_thread`. It saves the correlator outputs defined in Figure 13 across 160 frequency channels into a raw text format, with each new line containing a single floating-point sample. Every other sample is real, and the remaining samples are imaginary.

1.8.17 flag_beamsave_thread

This optional thread must follow the `flag_beamform_thread`. It saves the beamformer outputs defined in Figure 13 into a binary file consisting of only float pairs, each representing a complex sample.

1.8.18 flag_bx_save_thread

This optional thread must follow the `flag_bx_thread`. It creates two sub-threads that emulate the behavior of the `flag_frb_corsave_thread` and `flag_beamsave_thread` codes.

1.8.19 flag_bfx_save_thread

This optional thread must follow the `flag_bfx_thread`. It creates two sub-threads that emulate the behavior of the `flag_pfb_corsave_thread` and `flag_beamsave_thread` codes.

1.9 Dealer/Player Implementation Details

The Dealer/Player system is a Python-based code suite that manages ROACH boards and HPC processes and provides mechanisms to issue user commands and control shared memory. Detailed documentation for the Dealer/Player system can be found at [1]. In summary, a "Dealer" acts as a server, issuing commands to its "Players" or clients via zero-MQ socket protocols.

Player Interface.pdf

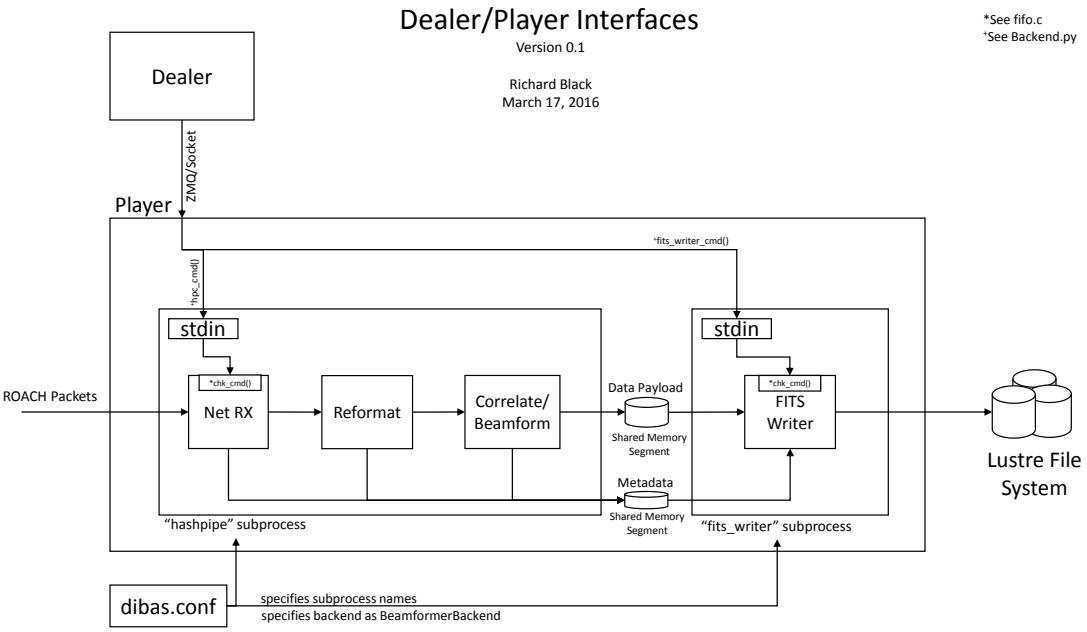

Figure 14: Caption

The FLAG back end system has a single Dealer that communicates with 20 Players that each manage a single instance of our HASHPIPE code and FITS writer. A single Player is shown in Figure 14. Here the Dealer issues commands to the Player, which in turn translates that command into messages that are then fed into the `stdin` of the HASHPIPE and FITS writer processes.

The Dealer/Player interface is controlled by a configuration file named `dibas.conf`, which outlines the parameters for each Player instance, known as BANKs, and the various operational modes that the Player can turn on. An example of a BANK configuration is shown below:

```
[BANKA]
# HPC / Player host & port
hpchost = flag4
player_port = 6677
# ROACH Control:
has_roach = true
katcp_ip = byur2
katcp_port = 7147
# Data flow
data_source_host = byur2
data_source_port = 60000
data_destination_host = 10.10.1.13
data_destination_port = 60000
# Synthesizer:
synth = none
# I'm pretty sure these don't matter if katcp is set as the synthesizer
synth_port = /dev/ttyS1
synth_ref = external
synth_ref_freq = 10000000
synth_vco_range = 2200, 4400
```

Table 6: FLAG Back End Operational Mode Table

Mode	MODENAME	Description
FLAG_HICORR_MODE	hi_correlator	Fine channel real-time correlator
FLAG_CALCORR_MODE	cal_correlator	Coarse channel real-time correlator for PAF calibration
FLAG_FRBCORR_MODE	frb_correlator	Coarse channel rapid-dump real-time correlator
FLAG_RTBF_MODE	pulsar_correlator	Real-time standalone beamformer
FLAG_PFB_MODE	flag_pfb_mode	Fine polyphase filter bank channelizer
FLAG_PFBCORR_MODE	flag_pfbcorr_mode	Fine polyphase filter bank channelizer + fine channel correlator
FLAG_BX_MODE	flag_bx_mode	Coarse-channel, rapid-dump correlator + real-time beamformer
FLAG_BFX_MODE	flag_bfx_mode	Fine-channel correlator + real-time beamformer

```
synth_rf_level = 5
synth_options = 0,0,1,0
# I2C (nonsense for FLAG)
filter_bandwidth_bits = 450, 0x00, 1450, 0x08, 1900, 0x18
# FLAG-specific parameters
xid = 12
instance = 0
gpudev = 0
cpus = 0, 1, 2, 3
```

An example configuration for a mode named FLAG_HICORR_MODE is shown below:

```
[FLAG_HICORR_MODE]
# These values get loaded directly into status shared memory
shmkeys = BACKEND,MODENAME

BACKEND = FLAG
MODENAME = hi_correlator

hpc_program=hashpipe
hpc_program_flags=-p flag_gpu
fits_process = bfFitsWriter

# IP and MAC Addresses
fabric_port = 60000

bof_file = flag_sim3_v1.bof
arm_phase = sync_gen_msync_in,0x0,sync_gen_msync_in,0x1,sync_gen_msync_in,0x0

# These are currently mandatory values
hwexposr = 0.000500395
filter_bw = 1450
frequency = 1500
nchan = 1024
hpc_fifo_name = /tmp/wouldntyouliketoknow.fifo
needed_arm_delay = 2
gigabit_interface_name = 10.2.118.123
dest_ip_register_name = bart
dest_port_register_name = lisa
master_slave_sel = 0,0,0,0,0,0
```

A list of all the modes that the FLAG back end supports are shown in Table 6.

1.10 Scan Overlord

The Scan Overlord is a Python script that monitors the state of the Green Bank Telescope scan coordinator and issues commands to the Dealer. Communication with the scan coordinator is accomplished by creating a map of desired values to observe and the respective callback functions that should be called. For example, the key to get the scan coordinator state is `ScanCoordinator.ScanCoordinator:P:state`. We can then assign a callback function to run whenever that scan coordinator state changes, so that

```
keys = {"ScanCoordinator.ScanCoordinator:P:state": state_callback},
```

where the method `state_callback` is called when the state changes.

Before proceeding, the following `include` statements will be needed:

```
import dealer
from ZMQJSONProxy import ZMQJSONProxyException
import zmq
import sys

from PBDataDescriptor_pb2 import PBDataField
from DataStreamUtils import get_service_endpoints
from DataStreamUtils import get_every_parameter
from datetime import datetime, tzinfo, date, time
```

To connect to the scan coordinator, one must be listening on a socket, which is done as follows:

```
req_url = "tcp://gbtdata.gbt.nrao:5559"
ctx = zmq.Context()
subscriber = ctx.socket(zmq.SUB)
```

This creates a ZMQ subscriber socket listening to the ScanCoordinator. One can then receive all messages from the ScanCoordinator by setting the socket filter option to an empty string.

```
subscriber.setsockopt(zmq.SUBSCRIBE, '')
```

In our case, we are interested in only some messages, which are specified by the key values from the `keys` dictionary, and the socket filters can be incorporated as follows:

```
for key in keys:
    major, minor = key.split(':')[0].split('.')
    sub_url, _, _ = get_service_endpoints(ctx, req_url, major, minor, 0)
    subscriber.connect(sub_url)
    subscriber.setsockopt(zmq.SUBSCRIBE, key)
```

Then one must start a main loop that constantly listens for messages from the scan coordinator, as follows:

```
while (auto_set):
    key, payload = subscriber.recv_multipart()
    df = PBDataField() # Create a message parser/decoder
    df.ParseFromString(payload) # Parse the string message into a struct
    f = keys[key] # Get the callback function name for the message
    f(df) # Call the callback function
```

Note that `PBDataField` is a Google Developer library, where "PB" stands for "Protocol Buffer." This library contains a string-parsing code that converts the received message into a `struct`.

This `struct` is the single argument of the respective callback function. For example, the `struct` received by `state_callback` contains a list of structures called `val_struct`, whose first entry contains a list of strings called `val_string`. Thus, one can print the current state using the following callback function:

```
def state_callback(p):
    print p.val_struct[0].val_string[0]
```

Currently, there are only four callback functions are in place, resulting in the key-callback map of

```
keys = {"ScanCoordinator.ScanCoordinator:P:state": state_callback,
        "ScanCoordinator.ScanCoordinator:P:startTime": start_time_callback,
        "ScanCoordinator.ScanCoordinator:P:scanLength": scan_len_callback,
        "ScanCoordinator.ScanCoordinator:P:projectId": project_id_callback}
```

The other callback functions can access the data as shown below:

```
def start_time_callback(p):
    num_sec = p.val_struct[0].val_struct[0].val_double[0]

def scan_len_callback(p):
    scan_length = p.val_struct[0].val_struct[0].val_double[0]

def project_id_callback(p):
    project_id = p.val_struct[0].val_string[0]
```

1.11 Dealer/player GUI

A GUI has been created and tested in the outdoor test facility (OTF) at GBO. This GUI is located in /home/groups/flag/dibas/versions/stableGUI/lib/python or under the versions/stableGUI/lib/python in the git repository and the file is called `one_punch_gui.py`. It has been created to simplify dealer/player control and has the same functionality described in the previous subsection, but operated from a GUI.

1.12 Commissioning Notes

1.12.1 Bus Address

In order to minimize cross-bus traffic, we must know the bus addresses for the NUMA nodes (CPU cores), GPU cards, and ethernet interfaces.

NUMA Nodes
To locate the NUMA node bus addresses, run the following code:

```
$ dmesg | grep "NUMA node"
```

GPU Cards
To locate the GPU card bus addresses, run the following code, where $CUDADIR is nominally `/usr/local/cuda`:

```
$ $CUDADIR/samples/1_Utilities/deviceQuery/deviceQuery
```

Ethernet Interfaces
To locate the bus addresses for the ethernet interfaces, run the following code:

```
$ sudo lshw -class network -businfo
```

1.13 MATLAB Post-Processing Codes

A suite of post-processing codes for the FLAG receiver were developed and are saved in the remote GIT repository located at

`https://gitlab.ras.byu.edu/ras-devel/matlab-post-processing`

There are six sub-directories:

- kernel
- sensitivity maps
- patterns
- tsys plots
- word lock
- misc

Each subsection that follows will explore the codes in each sub-directory mentioned above.

1.13.1 kernel

The codes located in the kernel sub-directory are functions and scripts that are called by scripts in the other sub-directories. These include scripts that will (1) open FITS files and parse the data into a MATLAB-friendly format, (2) extract antenna position information from ancillary FITS files, and (3) tabulate scan metadata from scan logs.

- **extract_covariances** This function opens a FITS file containing correlations from the coarse-channel correlator and reconstructs the correlation matrix. It then returns the restructured matrix, the corresponding DMJD, XID, and FITS header information. The function declaration is as follows:

  `function [ R, dmjd, xid, info ] = extract_covariances( fits_filename )`

  A note to users and developers: this function is not fast since file I/O in MATLAB is slow. It is possible that some micro optimization could speed this method up.

- **extract_pfb_covariances** This function performs the same operations as in `extract_covariances` except that it is expecting correlations from the fine-channel correlator. The function declaration is as follows:

  `function [ R, dmjd, xid, info ] = extract_pfb_covariances( fits_filename )`

- **extract_bf_output** This function opens a FITS file containing beamformed powers from the real-time beamformer and reconstructs the data into a matrix with dimensions (Beams $\times$ Polarization $\times$ Frequency $\times$ Time). The function is a little quirky since the default MATLAB FITS file reader crashes with large file sizes, and so care must be taken when using this. The function declaration is as follows:

  `function [ B, xid ] = extract_bf_output( fits_filename )`

- **aggregate_banks** This function collects the correlations from every BANK and the corresponding antenna positions, and aggregates them. The correlations are averaged in time based on user input, and the method also linearly interpolates the antenna positions to align with the DMJDs corresponding to the correlations. It then returns the resulting correlation matrix, interpolated offset angles, FITS file meta data, raw offset angles, raw DMJD values, and DMJD values corresponding to the integrated correlations. The function declaration is as follows:

  `function [ R, my_az, my_el, info, az_off, el_off, dmjd, dec_dmjd ] = aggregate_banks( save_dir, ant_dir, tstamp, on_off, Nint )`

  One must specify a directory in which to save the resulting aggregated correlations and positions by setting `save_dir` to the desired directory. The directory that contains the antenna FITS files must also be specified by `ant_dir`. The time stamp (formatted as "yyyy mm dd hh:mm:ss") for the scan to aggregate must also be provided in tstamp. The `on_off` argument is deprecated, but should be set to 1 until downstream code is refactored. Lastly, one specifies the number of correlations per integration by setting `Nint`. If `Nint` is set to -1, all correlations in the scan are averaged.

- **aggregate_banks_pfb** This function does the same thing as `aggregate_banks` except that it expects fine-channel data from the fine-channel correlator. The function declaration is as follows:

  ```
  function [ R, my_az, my_el, info ] = aggregate_banks_pfb( save_dir, ant_dir, tstamp, Nint
  )
  ```

- **aggregate_banks_rb_hack** This function does the same thing as `aggregate_banks` except that it only aggregates data for a single frequency channel. This should only be used when wanting to get a quick look at, say, a sensitivity map for a single frequency bin. The function declaration is as follows:

  ```
  function [ R, my_az, my_el, info ] = aggregate_banks_rb_hack( save_dir, ant_dir, tstamp,
  on_off, Nint )
  ```

- **aggregate_single_ bank** This function does the same thing as `aggregate_banks` except that it only aggregates data for a single BANK. This was done so as to minimize memory usage when processing long scans such as a Daisy scan. The function declaration is as follows:

  ```
  function [ R, my_az, my_el, xid, info ] = aggregate_single_bank( save_dir, ant_dir, tstamp,
  bank, Nint )
  ```

- **get_antenna_positions** This function reads an antenna FITS file and extracts the antenna positions. It also returns the DMJD values corresponding to the antenna positions and the RA/DEC angles. The function declaration is as follows:

  ```
  function [ dmjd, az_off, el_off, ra, dec ] = get_antenna_positions( fits_file, on_off, use_radec
  )
  ```

  The argument `on_off` is deprecated and should always be 1. The argument `use_radec` returns the RA/DEC angles as if they were the azimuth and elevation offset angles.

- **scan_table** This is a script that contains information about each observation session. For example, a table relating the scan numbers with time stamps is generated, and element mappings (i.e., correlation entries to dipole number) are provided. Any dipoles that have failed or frequency channels that were lost are specified here as well. A new entry in the scan table should be created for each observation session.

- **source_table** This is a script that contains information about calibration sources. Each entry is a struct that contains the source name and constants that are used to derive the source's flux density across frequency.

- **stitch_pfb** This is a quick-and-dirty script that reads in some fine-channel correlations and generates a quick spectrum.

- **create_weight_file** This function takes beam angles and weight values and generates a weight file. The function declaration is as follows:

  ```
  function create_weight_file(az, el, wX, wY, cal_filename, X_idx, Y_idx, filename)
  ```

  To use this function, one must also specify the mappings from dipoles to correlation matrix entry in `X_idx` and `Y_idx`.

- **compute_delay** This function estimates the delay between antenna signal paths by examining the phase structure across frequency. This was primarily used to evaluate the effectiveness of our timing alignment algorithm known as "word lock." The function declaration is as follows:

  ```
  function [ delta_n, residual ] = compute_delay(R, faxis, fs, ref_el)
  ```

  One must specify the sample rate (`fs`), center frequencies for each bin (`faxis`), and to which element the delays will be relative (`ref_el`, correlation diagonal index). It then returns the sample offsets in `delta_n` and the residual error from the linear regression in residual. Note: this function would likely be better placed in the `word_lock` directory.

1.13.2 sensitivity_maps

This sub-directory contains scripts and functions that generate sensitivity maps, trajectory plots, and calibrated steering vectors and beamformer weights.

- **plot_trajectories** This script extracts the antenna positions for a series of scans and plots the trajectory traversed by the telescope during those scans. One can also specify "off" or reference scans that are plotted with "x" markers.
- **sensitivity_map** This script computes the formed beam sensitivity for a series of scans in an observation session and plots them as a map. One must specify which session is being processed, the scan numbers for each of the on-source scans and reference ("off") scans. One may also specify a note string that will differentiate the resulting calibration steering vectors and weight file names from other grids in the same observation session. The source must also be specified using one of the defined structs in `source_table`, and the number of correlation matrices to average per grid point. For example, if the correlations were saved with 0.1 second resolution, and the desired resolution of the grid is one second, Nint should be set to 10. The script will then save the resulting calibrated steering vectors, maximum-SNR beamformer weights, and measured T_{sys}/eta values for each grid point. These files are then used when generating real-time beamformer weight files, pattern plots, and calibrated HI spectra.
- **sensitivity_daisy** This script does the same things as `sensitivity_map` except that it processes each BANK separately. All but the normalized system temperatures are then saved.

1.13.3 patterns

This sub-directory contains scripts and functions that generate beam or element patterns or assist with said generation.

- **get_beamformed_patterns** This method computes beam patterns using the calibrated steering vectors for a specified observation session, polarization, and set of weights. It then returns the pattern azimuth and elevation angles and pattern values. The function declaration is as follows:

  ```
  function [AZ, EL, patterns] = get_beamformed_patterns(session, pol, note, w)
  ```

  Note: the note argument is used to distinguish between calibration sets from the same session, and is specified in `sensitivity_map`.
- **get_element_patterns** This is a script that plots element patterns for a specified session and polarization. This should eventually be converted into a function.
- **get_grid_steering_vectors** This function loads in the saved calibrated steering vectors generated when running sensitivity map and returns the vectors closest to the specified azimuth and elevation offset angles. It also returns the angles corresponding to the return steering vectors and the measured normalized system temperature at those angles. The function declaration is as follows:

  ```
  function [a, a_az, a_el, Tsys] = get_grid_steering_vectors(session, pol, note, beam_az,
  beam_el)
  ```
- **get_grid_weights** This function loads in the saved maximum-SNR beamformer weights generated when running `sensitivity_map` and returns the weights for beams pointing in the user-specified directions. It also returns the beam pointing angle corresponding to the returned weight vectors. The function declaration is as follows:

  ```
  function [w, w_az, w_el] = get_grid_weights(session, pol, beam_az, beam_el, note)
  ```
- **plot_beam_patterns** This script will plot beam patterns for seven beams using weights and steering vectors generated from the same observation session's calibration grid. It also generates a binary weight file that can be used in the real-time beamformer.

- **plot_hex** This function plots the beam patterns for a given observation session and set of beamformer weights in a hexagonal pattern. It then returns a handle to the generated figure. The function declaration is as follows:

  `function map_fig = plot_hex(session, AZ, EL, patterns)`

- **plot_lcmv_patterns** This script generates linearly constrained minimum variance beamformer weights with several null constraints and plots the resulting patterns for the standard seven beam layout. It also generates a binary weight file that can be used in the real-time beamformer.

- RTBF_data_analysis Script that quickly looks at the total beamformed power in a time/frequency window. Currently this script is incorrectly placed in this sub-directory and would be better suited for misc.

1.13.4 tsys_plots

This sub-directory contains scripts and functions relating to generating system noise temperature spectra.

- **onoff_table** This script tabulates pairs of on/off source scans. One must specify which session the scans are from, the scan numbers for the two scans, the source struct from `scan_table`, and the LO frequency (Hz). This is then used in other scripts in this directory.

- get_onoff_tsys This function computes the normalized system temperature as a function of frequency for a given pair of on/off scans. One must specify the session struct from `scan_table`, the on/off scan numbers, the source struct from scan table, and the LO frequency (Hz). The function declaration is as follows:

  `function [Tsys_etaX, Tsys_etaY, freqs, wX, wY] = get_onoff_tsys(session, on_scan, off_scan, source, LO_freq)`

  The function returns the system temperatures for both polarization, the center frequencies for each frequency bin, and (for convenience) the maximum-SNR beamformer weights for a beam pointing in the direction of the on source.

- **plot_broad_tsys** This script compiles the system noise temperatures for a set of on/off pairs acquired with different LO frequencies and plots a broad spectrum. One must specify the session struct, the scan pairs, and the desired polarization.

- **run_all_onoff** This script gets the normalized system noise temperature for each pair of on/off scans specified in `onoff_table`. This is a useful script when trying to batch the aggregation process for every pair of on/off scans.

1.13.5 word_lock

This sub-directory contains scripts and functions relating to the "word lock" procedure, which attempts to time-align the element signal paths.

- **snoop_lock** This script reads in a coarse-channel correlation FITS file when observing noise from the noise source (i.e., the noise is coherent across elements) and estimates the sample offsets between the various paths through a phase ramp analysis. The resulting sample offsets are then written to a file to be read later and applied in ROACH boards.

- **word_lock** This script does the same thing as `snoop_lock` except that it instead expects a series of FITS files that were acquired when observing a test tone at different frequencies.

- **compare_lock** This script evaluates the signal path time alignment after the sample offsets resulting from snoop lock have been applied to the ROACH boards. One must specify which reference element was used in `snoop_lock` by changing `ref_el`.

1.13.6 misc

This sub-directory is meant to hold any quick-and-dirty scripts/functions that do not naturally fall into any of the other directories. At the time of this writing, there was only one script in this directory.

- **peak** This script reads in a single scan's worth of data corresponding to a PEAK scan, in which the telescope drifts across a bright point source. It then plots the total power in the first element (correlation diagonal entry 0) in a single BANK against the reported antenna position to evaluate pointing alignment.

A HASHPIPE Hello World Plugin

All of the following codes can be found at `https://github.com/rallenblack/hashpipe_samples.git`.

A.1 hw_databuf.h

This code outlines the specifications for the inter-thread buffer.

```
#ifndef _HW_DATABUF_H
#define _HW_DATABUF_H

#include "hashpipe_databuf.h"

/*  Hello World for hashpipe!
 *  Author: Richard Black
 *  Date: Jan. 11, 2017
 *
 *  Processing structure
 *  ====================================
 *  hw_thread1 -> hw_buffer1 -> hw_thread2
 *  ====================================
 *
 *  thread1 will put the letters "HELLO WORLD!" into buffer1.
 *  thread2 will read buffer1 and print out contents
 *
 *  This header file explains how the buffer is to be structured.
 */


// Macros to maintain cache alignment
#define CACHE_ALIGNMENT (128)
typedef uint8_t hashpipe_databuf_cache_alignment[
    CACHE_ALIGNMENT - (sizeof(hashpipe_databuf_t)%CACHE_ALIGNMENT)
];

// Number of blocks in the first buffer
#define N_BLOCKS1 5

// Create block header struct
typedef struct hw_buffer1_header
    int block_number; // We'll keep track of how many blocks we've processed
 hw_buffer1_header_t;

// Create dummy structure to make header size a multiple of CACHE_ALIGNMENT
```

```
typedef uint8_t hw_buffer1_cache_alignment[
    CACHE_ALIGNMENT - (sizeof(hw_buffer1_header_t)%CACHE_ALIGNMENT)
];

// Create the actual block struct
typedef struct hw_buffer1_block
    hw_buffer1_header_t header; // Put the header in the block
    hw_buffer1_cache_alignment padding; // Force data to be aligned
    char data[4]; // Three characters per block
 hw_buffer1_block_t;

// Create the full buffer
typedef struct hw_buffer1_databuf
    hashpipe_databuf_t header; // All hashpipe buffers must have this header
    hashpipe_databuf_cache_alignment padding; // Force buffer to be aligned
    hw_buffer1_block_t block[N_BLOCKS1];
 hw_buffer1_databuf_t;

/*****************************************
 *  Create buffer control method prototypes
 *  (Definitions in hw_databuf.c)
 *****************************************/

/*
 *  hw_buffer1_databuf_create
 *  Creates an hw_buffer1_databuf_t hashpipe-compatible buffer
 *
 *  @arg    int instance_id
 *          The hashpipe instance id (set by -I on the command line)
 *  @arg    int databuf_id
 *          The buffer's id -- typically set by hashpipe during startup
 *  @return hashpipe_databuf_t *
 *          A pointer to the newly created hashpipe-compatible buffer
 */
hashpipe_databuf_t * hw_buffer1_databuf_create(int instance_id, int databuf_id);


/*
 *  hw_buffer1_databuf_wait_free
 *  Blocking function to wait for a block in the buffer to be marked as processed
 *  @arg hw_buffer1_databuf_t * d
 *       A pointer to the buffer
 *  @arg int block_id
 *       The block index to wait for
 */
int hw_buffer1_databuf_wait_free(hw_buffer1_databuf_t * d, int block_id);

/*
 *  hw_buffer1_databuf_wait_filled
 *  Blocking function to wait for block to be marked as ready for processing
 *  @arg hw_buffer1_databuf_t * d
 *       A pointer to the buffer
 *  @arg int block_id
 *       The block index to wait for
```

```
 */
int hw_buffer1_databuf_wait_filled(hw_buffer1_databuf_t * d, int block_id);

/*
 *  hw_buffer1_databuf_set_free
 *  Function to mark block as having been processed
 *  @arg hw_buffer1_databuf_t * d
 *       A pointer to the buffer
 *  @arg int block_id
 *       The block index to free
 */
int hw_buffer1_databuf_set_free(hw_buffer1_databuf_t * d, int block_id);

/*
 *  hw_buffer1_databuf_set_filled
 *  Function to mark block as ready for processing
 *  @arg hw_buffer1_databuf_t * d
 *       A pointer to the buffer
 *  @arg int block_id
 *       The block index to mark as filled
 */
int hw_buffer1_databuf_set_filled(hw_buffer1_databuf_t * d, int block_id);

#endif
```

A.2 hw_databuf.c

This code defines the methods that were prototyped in `hw_databuf.h`.

```
#include "hw_databuf.h"

hashpipe_databuf_t * hw_buffer1_databuf_create(int instance_id, int databuf_id)
    size_t header_size = sizeof(hashpipe_databuf_t) + sizeof(hashpipe_databuf_cache_alignment);
    size_t block_size = sizeof(hw_buffer1_block_t);
    int n_block = N_BLOCKS1;
    return hashpipe_databuf_create(
        instance_id, databuf_id, header_size, block_size, n_block);


int hw_buffer1_databuf_wait_free(hw_buffer1_databuf_t * d, int block_id)
    return hashpipe_databuf_wait_free((hashpipe_databuf_t *)d, block_id);


int hw_buffer1_databuf_wait_filled(hw_buffer1_databuf_t * d, int block_id)
    return hashpipe_databuf_wait_filled((hashpipe_databuf_t *)d, block_id);


int hw_buffer1_databuf_set_free(hw_buffer1_databuf_t * d, int block_id)
    return hashpipe_databuf_set_free((hashpipe_databuf_t *)d, block_id);


int hw_buffer1_databuf_set_filled(hw_buffer1_databuf_t * d, int block_id)
    return hashpipe_databuf_set_filled((hashpipe_databuf_t *)d, block_id);
```

A.3 hw_thread1.c

This code defines the first thread in the plugin.

```
#include <pthread.h>
#include <string.h>

#include "hashpipe.h"
#include "hw_databuf.h"

/*****************************************************
 * hw_thread1.c
 * Author: Richard Black
 * Date: January 12, 2017
 *
 * This thread creates the string "HELLO WORLD!" and
 * passes that string 3 characters at a time to
 * hw_thread2 through the hw_buffer1_databuf_t buffer.
 *****************************************************/


// Run method for the thread
static void * run(hashpipe_thread_args_t * args)

    // Create a local pointer to the output buffer
    hw_buffer1_databuf_t * db_out = (hw_buffer1_databuf_t *)args->obuf;

    // Get access to status shared memory key/values
    hashpipe_status_t st = args->st;

    // Create string to print in the other thread
    const char test_word[13] = "HELLO WORLD!\0";

    // Create string counter
    int str_idx = 0;

    // Create block index tracker
    int block_idx = 0;

    // Create iteration counter
    int iter_count = 0;

    // Execute the main loop of the thread
    int rv;
    while (run_threads())
        // Wait for the buffer block to be available for writing
        while ((rv=hw_buffer1_databuf_wait_free(db_out, block_idx)) != HASHPIPE_OK)
            // If we time out, print "waiting" to status keyword "THREAD1"
            hashpipe_status_lock_safe(&st); // Gives us exclusive access to shared memory
            hputs(st.buf, "THREAD1", "waiting"); // Put string "waiting" with key "THREAD1"
            hashpipe_status_unlock_safe(&st); // Releases our exclusive access
```

```
        // Once we get here, we have a ready-for-writing block in the buffer

        // Copy part of the string to the buffer
        strncpy(db_out->block[block_idx].data, test_word + (str_idx % 12), 3);

        // Update the string pointer
        str_idx += 3;

        // Write the loop counter
        db_out->block[block_idx].header.block_number = iter_count;

        // Only do this 20 times
        if (iter_count < 20)
            // Update the loop counter
            iter_count++;

            // Mark block as filled so next thread can process the data
            hw_buffer1_databuf_set_filled(db_out, block_idx);

            // Update the block index
            // We use the % operator to make the block_idx circle back
            block_idx = (block_idx + 1) % N_BLOCKS1;

        else
            hashpipe_status_lock_safe(&st);
            hputs(st.buf, "THREAD1", "Done!");
            hashpipe_status_unlock_safe(&st);


        // Check to see if hashpipe is closing
        pthread_testcancel();


    return NULL;



// Thread description for hashpipe
static hashpipe_thread_desc_t hw_thread1 =
    name: "hw_thread1", // The name of the thread for the command line
    skey: "STAT1", // A shared memory keyword for thread status msgs
    init: NULL, // The name of the initialization function (NULL if none)
    run: run, // The name of the main loop function
    ibuf_desc: NULL, // The buffer creation method name for the input buffer (NULL if none)
    obuf_desc: hw_buffer1_databuf_create // The buffer creation name for the output buffer (NULL if none
;

// Hashpipe calls this method to enable this thread
static __attribute__((constructor)) void ctor()
    register_hashpipe_thread(&hw_thread1);
```

A.4 hw_thread2.c

This code defines the second thread in the plugin.

```
#include <pthread.h>
#include <string.h>

#include "hashpipe.h"
#include "hw_databuf.h"

/*****************************************************
 * hw_thread2.c
 * Author: Richard Black
 * Date: January 12, 2017
 *
 * This threads waits for data to become available in
 * the hw_buffer1_databuf_t buffer. It then prints
 * out the contents of the buffer, which should be
 * three characters of the string "HELLO WORLD!"
 *
 * Expected output:
 * -----------------
 *  0: HEL
 *  1: LO
 *  2: WOR
 *  3: LD!
 *  4: HEL
 *  5: LO
 *  6: WOR
 *  7: LD!
 *  ...
 *  19: LD!
 * -----------------
 *****************************************************/

// Run method for the thread
static void * run(hashpipe_thread_args_t * args)

    // Create a local pointer to the input buffer
    hw_buffer1_databuf_t * db_in = (hw_buffer1_databuf_t *)args->ibuf;

    // Get access to status shared memory key/values
    hashpipe_status_t st = args->st;

    // Create placeholder for incoming data
    char new_chars[4];

    // Create iteration count placeholder
    int iter_count = -1;

    // Create block index tracker
    int block_idx = 0;

    // Execute the main loop of the thread
    int rv;
```

```
    while (run_threads())
        // Wait for the buffer block to be available for writing
        while ((rv=hw_buffer1_databuf_wait_filled(db_in, block_idx)) != HASHPIPE_OK)
            // If we time out, print "waiting" to status keyword "THREAD1"
            hashpipe_status_lock_safe(&st); // Gives us exclusive access to shared memory
            hputs(st.buf, "THREAD2", "waiting"); // Put string "waiting" with key "THREAD1"
            hashpipe_status_unlock_safe(&st); // Releases our exclusive access

            pthread_testcancel(); // Check if process is ending


        // Once we get here, we have a ready-for-processing block in the buffer

        // Copy buffer contents to local memory
        strncpy(new_chars, db_in->block[block_idx].data, 3);
        new_chars[3] = '\0';

        // Get the iteration count
        iter_count = db_in->block[block_idx].header.block_number;

        // Print the string to the console
        printf("%d: %s\n", iter_count, new_chars);

        // Mark block as free so it can get new data
        hw_buffer1_databuf_set_free(db_in, block_idx);

        // Update the block index
        block_idx = (block_idx + 1) % N_BLOCKS1;

        // Check to see if hashpipe is closing
        pthread_testcancel();


    return NULL;



// Thread description for hashpipe
static hashpipe_thread_desc_t hw_thread2 =
    name: "hw_thread2",
    skey: "STAT2",
    init: NULL,
    run: run,
    ibuf_desc: hw_buffer1_databuf_create,
    obuf_desc: NULL
;

static __attribute__((constructor)) void ctor()
    register_hashpipe_thread(&hw_thread2);
```

A.5 Makefile.am

This is the required Makefile template needed for automake.

```
ACLOCAL_AMFLAGS = -I m4
AM_CPPFLAGS     =

AM_CPPFLAGS += -I"@HASHPIPE_INCDIR@"

# AM_CFLAGS is used for all C compiles
AM_CFLAGS = -fPIC -O3 -Wall -Werror -fno-strict-aliasing -mavx

hw_databuf = hw_databuf.h \
             hw_databuf.c

hw_threads = hw_thread1.c \
             hw_thread2.c

# This is the flag_gpu plugin itself
lib_LTLIBRARIES = hw_hashpipe.la
hw_hashpipe_la_SOURCES  = $(hw_databuf) $(hw_threads)
hw_hashpipe_la_LIBADD   = -lrt -L/usr/local/cuda/lib64
hw_hashpipe_la_LDFLAGS  = -avoid-version -module -shared -export-dynamic --enable-shared
hw_hashpipe_la_LDFLAGS += -L"@HASHPIPE_LIBDIR@" -Wl,-rpath,"@HASHPIPE_LIBDIR@"
```

A.6 configure.ac

This is the required auto-configure script needed for automake.

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.63])
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
AM_INIT_AUTOMAKE([foreign])
LT_INIT
AM_SILENT_RULES([yes])
#AC_CONFIG_SRCDIR([paper_databuf.h])
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_MACRO_DIR([m4])

# Set CFLAGS to nothing if it is not set by the user.  This prevents AC_PROG_CC
# from setting the (supposedly reserved-for-the-user!) variable CFLAGS in
# Makefile, which prevents AM_CFLAGS in Makefile.am from setting an
# optimization level.  For more details, see
# http://lists.gnu.org/archive/html/autoconf/2006-04/msg00007.html
AS_VAR_SET_IF(CFLAGS,[],[CFLAGS=])

# Checks for programs.
AC_PROG_CC

# Check for HASHPIPE and xGPU and total_power
AX_CHECK_HASHPIPE
AX_CHECK_XGPUINFO
```

```
AX_CHECK_FLAGBEAMFORM
AX_CHECK_FLAGPOW

# Checks for libraries.
AC_CHECK_LIB([pthread], [pthread_create])
AC_CHECK_LIB([rt], [clock_gettime])
AC_CHECK_LIB([z], [crc32])

# Checks for header files.
AC_CHECK_HEADERS([netdb.h stdint.h stdlib.h string.h sys/socket.h sys/time.h unistd.h zlib.h])

# Checks for typedefs, structures, and compiler characteristics.
AC_C_INLINE
AC_TYPE_INT32_T
AC_TYPE_INT64_T
AC_TYPE_OFF_T
AC_TYPE_SIZE_T
AC_TYPE_UINT32_T
AC_TYPE_UINT64_T
AC_TYPE_UINT8_T

# Checks for library functions.
AC_FUNC_MALLOC
AC_CHECK_FUNCS([clock_gettime memset socket crc32])

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

A.7 hashpipe.m4

This script looks for the installed HASHPIPE libraries so the plugin can be installed.

```
# serial 1 hashpipe.m4
AC_DEFUN([AX_CHECK_HASHPIPE],
[AC_PREREQ([2.65])dnl
AC_ARG_WITH([hashpipe],
            AC_HELP_STRING([--with-hashpipe=DIR],
                           [Location of HASHPIPE files (/usr/local)]),
            [HASHPIPEDIR="$withval"],
            [HASHPIPEDIR=/usr/local])

orig_LDFLAGS="$LDFLAGS"
LDFLAGS="$orig_LDFLAGS -L$HASHPIPEDIR/lib"
AC_CHECK_LIB([hashpipe], [hashpipe_databuf_create],
             # Found
             AC_SUBST(HASHPIPE_LIBDIR,$HASHPIPEDIR/lib),
             # Not found there, check HASHPIPEDIR
             AS_UNSET(ac_cv_lib_hashpipe_hashpipe_databuf_create)
             LDFLAGS="$orig_LDFLAGS -L$HASHPIPEDIR"
             AC_CHECK_LIB([hashpipe], [hashpipe_databuf_create],
                          # Found
                          AC_SUBST(HASHPIPE_LIBDIR,$HASHPIPEDIR),
                          # Not found there, error
```

```
                          AC_MSG_ERROR([HASHPIPE library not found])))
LDFLAGS="$orig_LDFLAGS"

AC_CHECK_FILE([$HASHPIPEDIR/include/hashpipe.h],
              # Found
              AC_SUBST(HASHPIPE_INCDIR,$HASHPIPEDIR/include),
              # Not found there, check HASHPIPEDIR
              AC_CHECK_FILE([$HASHPIPEDIR/hashpipe.h],
                            # Found
                            AC_SUBST(HASHPIPE_INCDIR,HASHPIPEDIR),
                            # Not found there, error
                            AC_MSG_ERROR([hashpipe.h header file not found])))

])
```

References

[1] "Dibas 1.0 documentation." http://www.gb.nrao.edu/~mwhitehe/dibas/html/index.html. Accessed: 2017-01-23.