VEGAS
HPC Software Developer Documentation

Simon Scott, UC Berkeley
Jayanth Chennamangalam, WVU

April 9, 2013

Contents

1 VEGAS Overview: ROACHs and HPCs

1.1 Introduction

This section provides an overview of the operation of the ROACHs and HPCs. Since the operation of the ROACH boards is beyond the scope of this document, only those details necessary to understand how the ROACH boards interface with the HPC will be provided.

The ROACH boards can operate in 17 different observing modes. These modes can be divided into two categories: high bandwidth (≥ 250 MHz) and low bandwidth (< 250MHz). In high-bandwidth modes, the ROACH FPGA board acts as a spectrometer, calculating the Fourier Transform of the sampled IF waveform, integrating, and sending the integrated spectrum to the HPC (CPU and GPU) for post-processing and storage. In low-bandwidth modes, the FPGA board simply downconverts and packetizes the IF waveform, before sending it to the HPC. The HPC cluster computes the Fourier Transform on GPUs, before writing to disk.

1.2 Control and Timing Inputs to the ROACH

Each ROACH has four control inputs:

- 1 calibration switching signal
- 2 position/frequency switching signals
- 1 blanking signal

And two timing inputs:

- 1 PPS (1 pulse per second)
- ARM (indicates when FPGA must start integrating spectra)

The calibration switching signal and the two position/frequency switching signals are collectively known as the “switching signals”. These three signals allow the spectrometer to operate in eight different “switching states”. Note that although the ARM signal is regarded as an electrical signal in this document, it is actually implemented as a register on each of the ROACH FPGAs. This register is set by sending an appropriate Ethernet packet to the ROACH.

1.3 Operation of the ROACH

1.3.1 Operation in High-Bandwidth Modes

The FPGA spectrometers are "free running"; they are never started or stopped. The starting and stopping of the integrations is implemented by the HPC software. The FPGA performs an initial, fixed-period integration (on the order of 1ms), in order to reduce the datarate. The integrated spectrum is then sent to the CPU/GPU for longer integration.

The FPGA spectrometer integrates spectra and counts the number of spectrum in an integration. Each integration ends when a maximum number of spectra (called FPGA_NMAX_SPECTRA) is reached, or when one of the switching signals change. In either of these cases, the integration stops at the next spectral boundary, and the integrated spectrum is transmitted to the HPC over the 10Gbe link. The next integration begins immediately.

If the blanking signal goes active at any point, the current spectrum (as outputted from the PFB) will be included in the integration performed on the FPGA, but all subsequent spectra will not be added to the vector accumulators (i.e. will not be integrated) until the blanking signal is cleared. This means that the blanking signal is re-registered to the next spectral boundary on the FPGA.

In summary, while the blanking signal is high, no spectra are added to the vector accumulators on the FPGA, and the counter that counts the number of spectra in the current integration is not incremented.

FPGA_NMAX_SPECTRA will typically be set so that the FPGA integrates 1 millisecond worth of spectra before sending the spectra to the CPU/GPU for further integration.

1.3.2 Operation in Low-Bandwidth Modes

The FPGA is again "free-running". The starting and stopping of the integrations is implemented by the HPC. The FPGA simply performs the necessary digital-downconversion and filtering on the sampled waveform before packetising the data and sending it to the HPC. No spectroscopy is performed in these modes.

Unlike the high-bandwidth mode, the FPGA still sends samples to the HPC when the blanking signal is active. However, when the blanking signal is active, the FPGA sets the blanking bit in the header of the packets sent to the HPC. This informs the HPC that the blanking signal is active and that the spectrum that is currently being computed on the HPC must not be added to the vector accumulators (in the HPC).

1.3.3 Instructing the HPC to Start Integrating

When the FPGA is powered up, the SPEAD packet counter is set to a non-zero value (such as 2048). The GBT M&C starts the spectrometer system by activating the ARM signal on the FPGAs. On the following 1 PPS clock signal, the FPGA simply clears its internal accumulators and resets the packet counter to zero. Therefore, the first packet transmitted after the ARM is activated has packet count zero. This zero packet counter instructs the HPC to start integrating. If the ARM signal deactivates at any point, it has no effect on the FPGA or HPC.

1.4 Architecture of the HPC Software

The work performed by the VEGAS HPC software is implemented in four separate POSIX threads:

1. Network Thread
2. GPU Thread (only used in low-bandwidth modes)
3. CPU Accumulator Thread
4. Disk Thread (not used if disk writing is disabled)

VEGAS uses three shared memory data buffers, one between each of these data processing threads. There is also a shared memory status buffer that is used to configure the software, and report back on the system status. Figure 1 shows the shared memory buffers in high-bandwidth modes, where the GPU is not used. Figure 2 shows the shared memory buffers in low-bandwidth modes, where the GPU is used to perform the spectroscopy. The only difference between these two diagrams is that the second one has an additional GPU thread and GPU data buffer.

In both diagrams, the processing threads are yellow, the shared memory data buffers are blue and the status shared memory is purple. In high-bandwidth modes, the ROACH sends 1ms-integrated spectra to the HPC, in network packets. The Network thread reads these packets, re-assembles them to form complete spectra, and writes the spectra to the CPU buffer. The CPU thread then accumulates these spectra for a specified amount of time, and writes the accumulated spectra to the Disk buffer. The Disk thread writes the accumulated spectra to disk.

In low-bandwidth modes, the FPGA instead sends time samples to the HPC. The GPU thread then performs a PFB/FFT, accumulates the spectra for 1ms, and writes the 1ms-integrated spectra to the CPU buffer. The CPU and Disk threads operate as previously

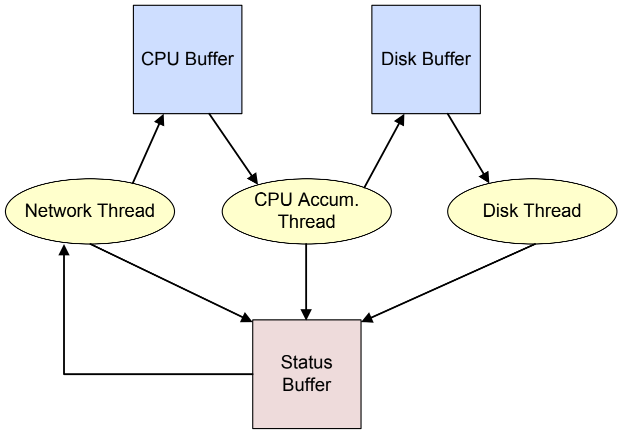


Figure 1: VEGAS shared memory buffers for high-bandwidth modes

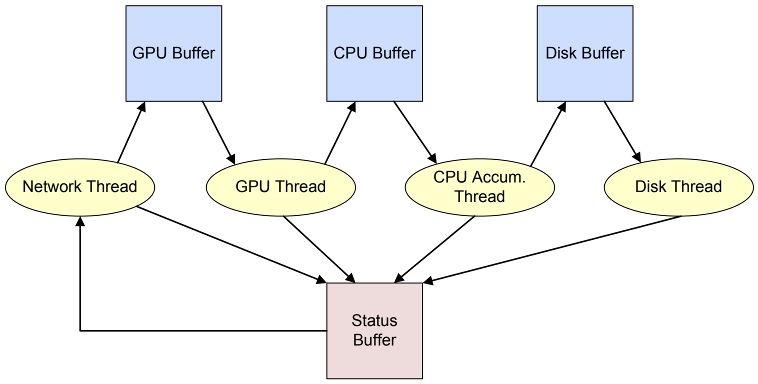


Figure 2: VEGAS shared memory buffers for low-bandwidth modes

described. In all modes, the Disk thread is optional: if the disk writing is performed by the GBT M&C, then the disk thread will be disabled (see this CASPER VEGAS wiki page for details on how to disable disk writing).

The data buffers are ring buffers, used to transfer data from one thread to the next. The status buffer contains the settings for the HPC software, such as number of frequency channels, number of sub-bands, and which network ports to use. The network thread reads these settings and writes them to the FITS header, at the top of each data buffer block (explained later). These settings must be configured by an external application before the HPC software is started. All of the processing threads are also able to report their status (such as number of packets dropped or how fill each of the ring buffers are) by writing to the status shared memory. Both the data buffers and the status buffer are shared memory, and can be accessed from external applications using the POSIX shared-memory API.

2 Data Structures

This section describes the structure of the data structures used within the VEGAS HPC, namely the SPEAD network packet format, the shared memory data buffers and the shared memory status buffer.

2.1 The SPEAD Packet Format

Data is transmitted from the ROACH boards to the HPCs in UDP packets, formatted according to the SPEAD protocol. Although three different packet formats are used, they are all SPEAD compliant. Figure 3 shows the SPEAD packets for high-bandwidth modes, Figure 4 shows the SPEAD packets for low-bandwidth modes with multiple subbands and Figure 5 shows the SPEAD packets for low-bandwidth modes with single subbands.

The header fields for the various packet formats are explained below:

- **Heap counter:** in high-bandwidth modes, each heap represents a single spectrum. Therefore, a heap may be multiple packets long. In low-bandwidth modes, a heap is just a single packet containing a set of time samples.
- **Heap size:** the size of the heap (which may be multiple packets), in bytes.
- **Heap offset:** if the heap is multiple packets long (as in high-bandwidth modes), the offset field indicates where this packet's payload fits into the larger heap.
- **Packet payload length:** length of this packet's payload data, in bytes.
- **Time counter:** this counter is incremented at the FPGA clock rate. The field records the FPGA time counter at the clock cycle when spectrum is outputted from the PFB (high-bandwidth modes) or when the time samples are recorded (low-bandwidth modes). It is reset at the 1 PPS tick after an ARM command is issued.
- **Spectrum counter:** this counter is incremented every spectrum, even blanked spectra. It counts the spectra coming out of the PFB. It is reset at the 1 PPS tick after an ARM command is issued.
- **Integration size:** this counter is incremented every spectrum, unless that spectrum is blanked. It is reset at the start of each FPGA integration. This counter therefore indicates the number of spectra that were integrated on the FPGA, before the integrated spectrum was transmitted to the HPC. For most packets, this counter has the value FPGA_NMAX_SPECTRA. However, if the status bits change during the integration, this counter value will be less than FPGA_NMAX_SPECTRA.

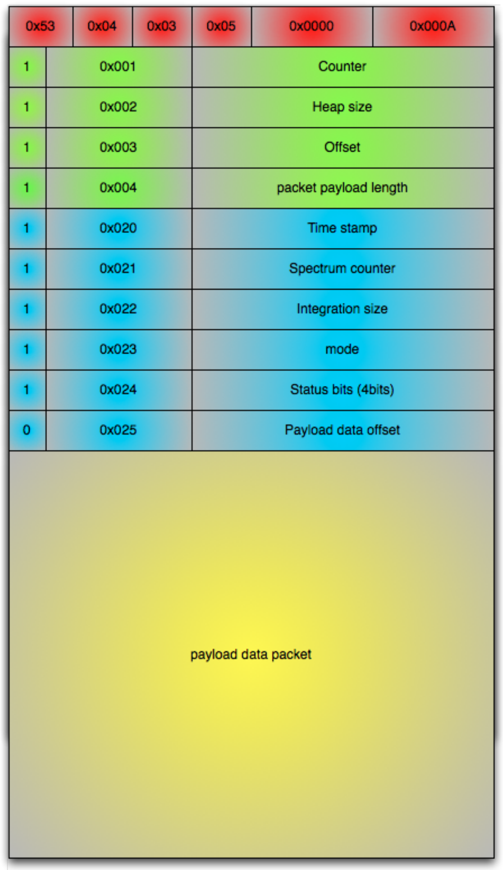
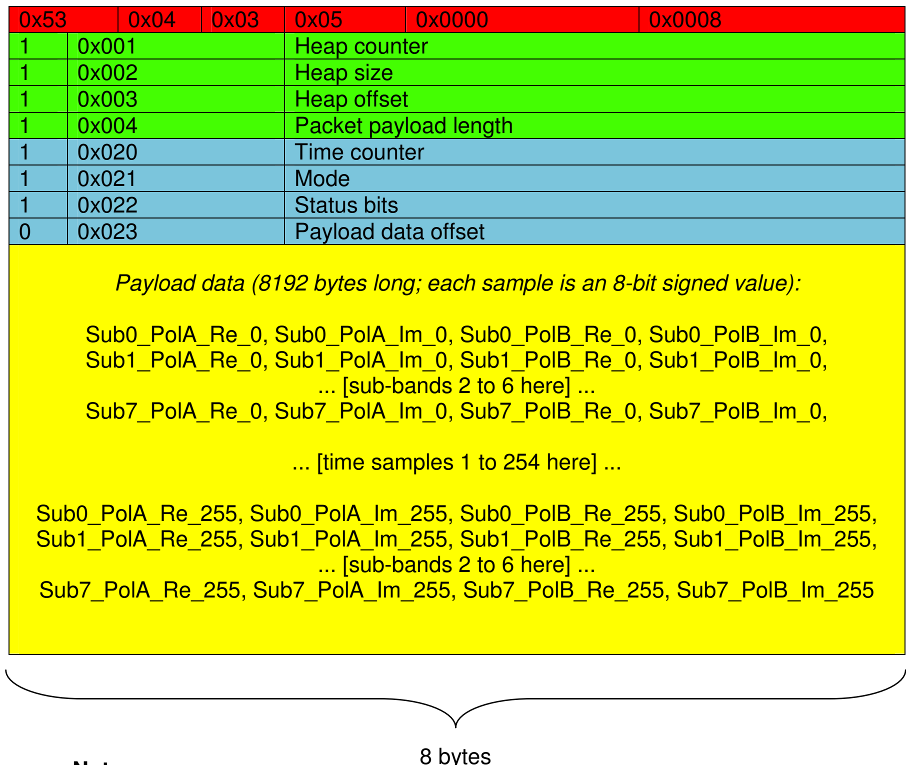


Figure 3: Format of SPEAD packets for high-bandwidth modes

- **Mode:** the mode in which the FPGA is operating (a number from 1 to 17).
- **Status bits:** the state if the three switching signals and the blanking signal. Bit 3 is the blanking signal, while bits 2 to 0 are the calibration, position and frequency switching signals.
- **Payload data offset:** the number of bytes between the end of the *Packet payload length* field and the beginning of the payload data section, in this packet.

2.1.1 The Payload Data Field for High-Bandwidth Modes

As mentioned, a single spectrum (or heap) may be split over multiple SPEAD packets. Within the *Payload Data* field of a single packet, the data is stored as follows:



0x53	0x04	0x03	0x05	0x0000	0x0008

1	0x001	Heap counter
1	0x002	Heap size
1	0x003	Heap offset
1	0x004	Packet payload length
1	0x020	Time counter
1	0x021	Mode
1	0x022	Status bits
0	0x023	Payload data offset

Payload data (8192 bytes long; each sample is an 8-bit signed value):

Sub0_PolA_Re_0, Sub0_PolA_Im_0, Sub0_PolB_Re_0, Sub0_PolB_Im_0,
Sub1_PolA_Re_0, Sub1_PolA_Im_0, Sub1_PolB_Re_0, Sub1_PolB_Im_0,
... [sub-bands 2 to 6 here] ...
Sub7_PolA_Re_0, Sub7_PolA_Im_0, Sub7_PolB_Re_0, Sub7_PolB_Im_0,

... [time samples 1 to 254 here] ...

Sub0_PolA_Re_255, Sub0_PolA_Im_255, Sub0_PolB_Re_255, Sub0_PolB_Im_255,
Sub1_PolA_Re_255, Sub1_PolA_Im_255, Sub1_PolB_Re_255, Sub1_PolB_Im_255,
... [sub-bands 2 to 6 here] ...
Sub7_PolA_Re_255, Sub7_PolA_Im_255, Sub7_PolB_Re_255, Sub7_PolB_Im_255

8 bytes

Figure 4: Format of SPEAD packets for high-bandwidth modes (multiple subbands)

```
Ch0_XX*, Ch0_YY*, Re(Ch0_X*Y), Im(Ch0_X*Y),
Ch1_XX*, Ch1_YY*, Re(Ch1_X*Y), Im(Ch1_X*Y),
...
Ch2047_XX*, Ch2047_YY*, Re(Ch2047_X*Y), Im(Ch2047_X*Y)
```

Note that each value is a 32-bit signed integer. `Ch[0-2047]` represents the spectral channel. For modes where only 1024 spectral channels are used, the packet will only be 4kB long.

`XX*` is a real number and represents the power in X plane. `YY*` is a real number and represents the power in Y plane. `X*Y` is a complex number and represents the cross-correlation. `XY*` is not transmitted.

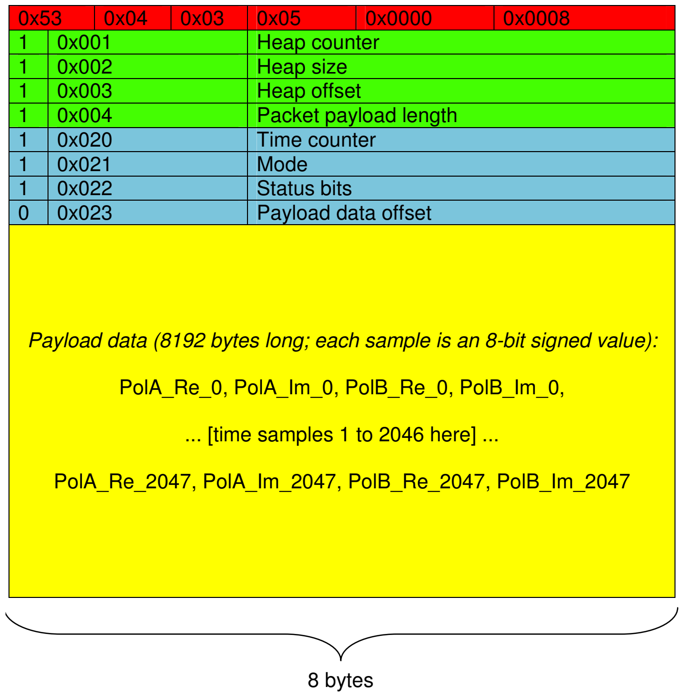

0x53
0x04
0x03
0x05
0x0000
0x0008
1 0x001 Heap counter
1 0x002 Heap size
1 0x003 Heap offset
1 0x004 Packet payload length
1 0x020 Time counter
1 0x021 Mode
1 0x022 Status bits
0 0x023 Payload data offset
Payload data (8192 bytes long; each sample is an 8-bit signed value):
PolA_Re_0, PolA_Im_0, PolB_Re_0, PolB_Im_0,
... [time samples 1 to 2046 here] ...
PolA_Re_2047, PolA_Im_2047, PolB_Re_2047, PolB_Im_2047
8 bytes

Figure 5: Format of SPEAD packets for low-bandwidth modes (single subband)

2.1.2 The Payload Data Field for Low-Bandwidth Modes (multiple subbands)

A heap is always one packet long in these modes. Therefore, the heap counter is the same as the packet counter, and the heap offset is always zero.

The ordering of the data within the payload is indicated in the diagram above. For example, `Sub0_PolA_Re_0` is interpreted as:
`Sub0` = sub-band 0
`PolA` = polarisation A
`Re` = the real component of the signal
`0` = sample at time instant 0

Each packet therefore contains samples from 256 time instances and 8 different sub-bands.

Finally, the centre frequencies and bandwidth of each sub-band are not stored in the packet header. They are instead passed to the HPC software via the status shared memory.

2.1.3 The Payload Data Field for High-Bandwidth Modes (single subband)

A heap is always one packet long in these modes. Therefore, the heap counter is the same as the packet counter, and the heap offset is always zero.

The ordering of the data within the payload is indicated in the diagram above. For example, `PolA_Re_0` is interpreted as:
`PolA` = polarisation A
`Re` = the real component of the signal
`0` = sample at time instant 0

Each packet therefore contains samples from 2048 time instances.

2.2 Structure of the Shared Memory Data Buffers

As described earlier, the shared memory data buffers are used to pass time/frequency samples from one thread to another. Each data buffer is in fact a ring buffer, containing a number of independent blocks protected by semaphores. All three shared memory data buffers have the following common structure, as shown in Figure 6:

The **guppi_databuf structure** stores the size of the FITS header blocks, the indexes and the data blocks. It also keeps count of the number of data blocks within the buffer. Finally, it contains the semaphore ID for the shared memory buffer.

The **FITS header** contains a snapshot of the status shared memory buffer, taken at the time that the network packets were first written to the corresponding data block. This means that the FITS header stores information such as antenna position, local machine timestamps and number of frequency channels. There is one FITS header for each data block, and each header is 184 320 bytes.

The **index** is used for locating individual spectra within a single data block. There is one index per data block.

The **data blocks** store the actual time samples or integrated spectra. The size of a single data block can vary, but it is typically 32MB in size. There are also typically 64 data blocks per buffer.

The contents of the data blocks and indexes do however vary, depending on the buffer type. These are described in the following sections.

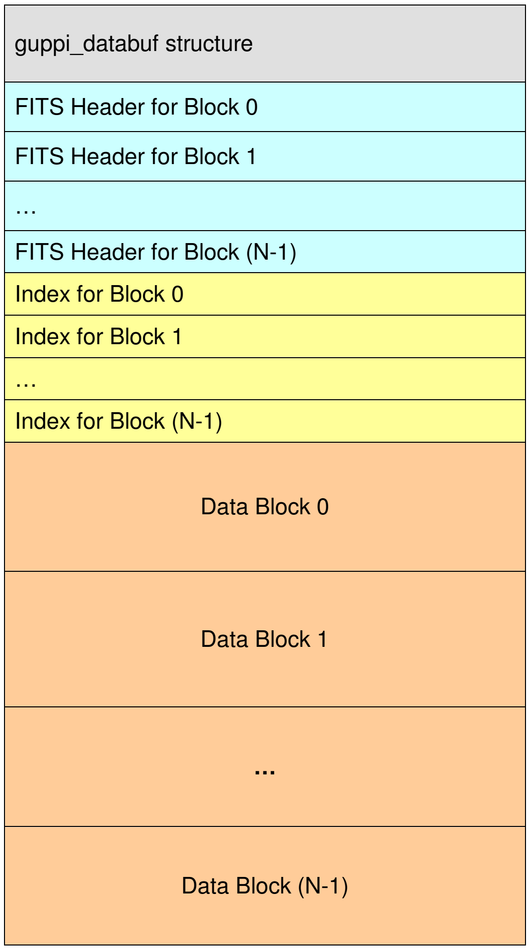
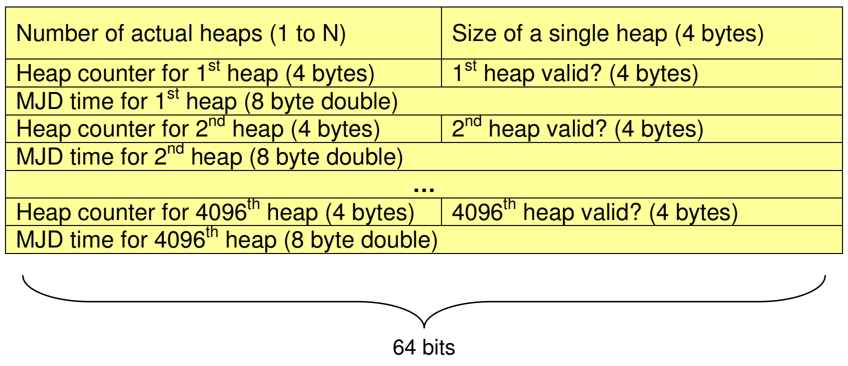


Figure 6: Structure of the Data Buffers

2.3 Structure of the Buffer at Input to the GPU or CPU Thread

The shared memory data buffers at the input to the GPU and CPU accumulation threads store individual SPEAD heaps. Each heap contains a complete spectrum (CPU input buffer) or a block of consecutive time samples (GPU input buffer), along with associated meta data. Since there are typically a few hundred heaps per data block, and index is used to access individual heaps within a single data block. The structure of the index is shown in Figure 7.

Although the index supports up to 4096 heaps per data block, the actual number of heaps within the block may be considerably less. All heaps within a block are of identical



Number of actual heaps (1 to N)	Size of a single heap (4 bytes)
Heap counter for 1st heap (4 bytes)	1st heap valid? (4 bytes)
MJD time for 1st heap (8 byte double)	
Heap counter for 2nd heap (4 bytes)	2nd heap valid? (4 bytes)
MJD time for 2nd heap (8 byte double)	
...	
Heap counter for 4096th heap (4 bytes)	4096th heap valid? (4 bytes)
MJD time for 4096th heap (8 byte double)	

Figure 7: Index for a *single* data block in the CPU and GPU input buffers

size. Each heap has an associated *heap counter*, *heap valid* flag and *MJD time* within the index. Consecutive heaps should have incrementing *heap counters*, except when the ARM signal occurs (then the *heap counter* is reset to zero). If a heap contains a corrupted packet, it is marked as invalid ($heap\ valid = 0$); otherwise the *heap valid flag* is set to 1. The *MJD time* field records the time (in 64-bit Modified Julian Date format with sub-second resolution) that the first network packet in that spectrum was received by the HPC.

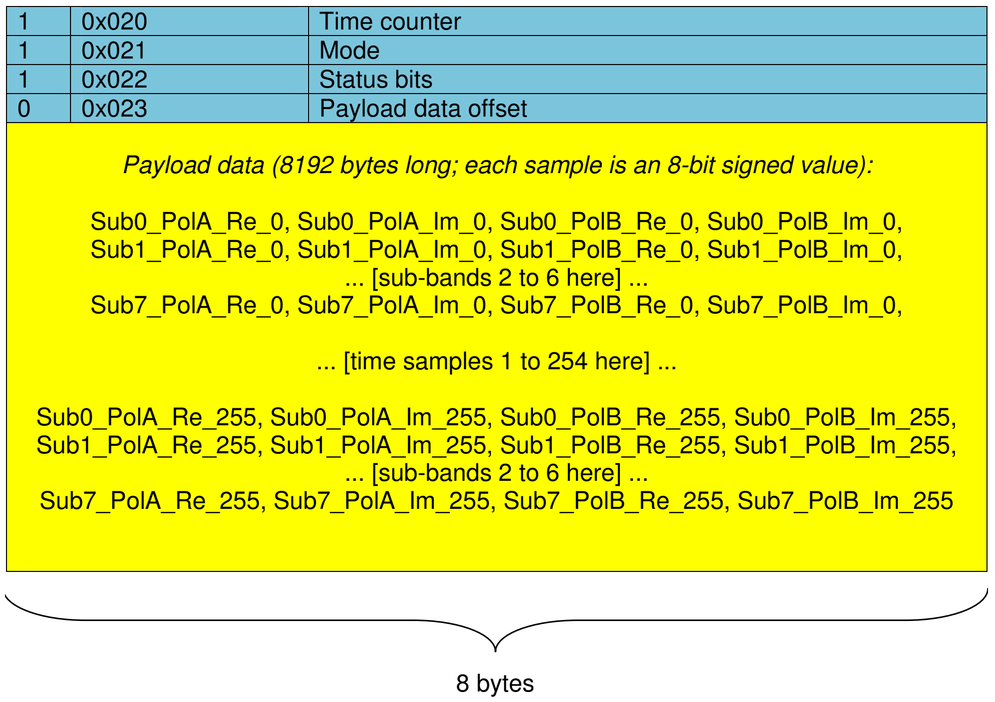
Each data block simply contains many heaps, arranged one after the other, without any inter-heap space. The structure of a single heap is different for the GPU and CPU input buffers. These heap structures are described below.

2.3.1 GPU Heap

A single data block in the GPU buffer contains many GPU heaps. A single heap is simply a SPEAD packet for low-bandwidth VEGAS modes, with the SPEAD packet headers removed. A heap is therefore always one packet long in these modes. One such heap is shown in Figure 8.

Each heap has four header fields that were described in Section 2.1. The ordering of the time samples within the heap is indicated in Figure 8. For example, `Sub0_PolA_Re_0` is interpreted as:
`Sub0` = sub-band 0
`PolA` = polarisation A
`Re` = the real component of the signal
`0` = sample at time instant 0

Each heap therefore contains samples from 256 time instances and 8 different sub-bands;



1	0x020	Time counter
1	0x021	Mode
1	0x022	Status bits
0	0x023	Payload data offset

Payload data (8192 bytes long; each sample is an 8-bit signed value):

Sub0_PolA_Re_0, Sub0_PolA_Im_0, Sub0_PolB_Re_0, Sub0_PolB_Im_0,
Sub1_PolA_Re_0, Sub1_PolA_Im_0, Sub1_PolB_Re_0, Sub1_PolB_Im_0,
... [sub-bands 2 to 6 here] ...
Sub7_PolA_Re_0, Sub7_PolA_Im_0, Sub7_PolB_Re_0, Sub7_PolB_Im_0,

... [time samples 1 to 254 here] ...

Sub0_PolA_Re_255, Sub0_PolA_Im_255, Sub0_PolB_Re_255, Sub0_PolB_Im_255,
Sub1_PolA_Re_255, Sub1_PolA_Im_255, Sub1_PolB_Re_255, Sub1_PolB_Im_255,
... [sub-bands 2 to 6 here] ...
Sub7_PolA_Re_255, Sub7_PolA_Im_255, Sub7_PolB_Re_255, Sub7_PolB_Im_255

Figure 8: A single heap within the GPU shared memory data buffer

or from 2048 time instances in just 1 sub-band, depending on the mode.

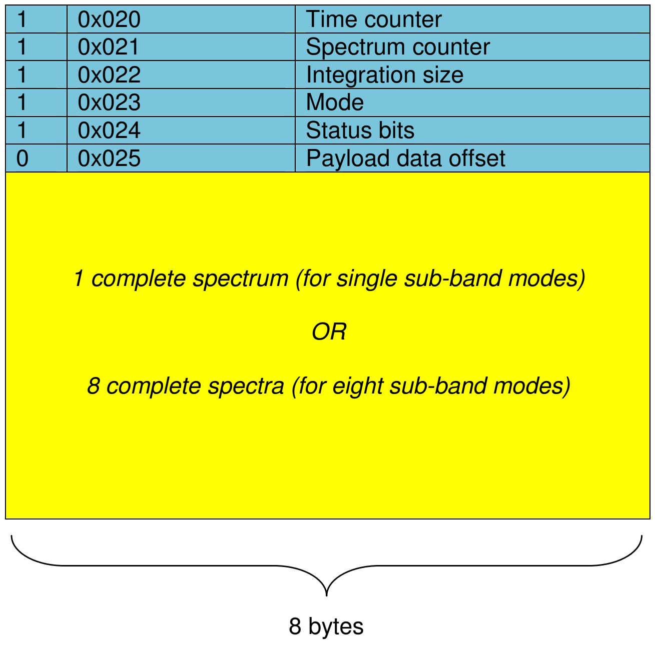
2.3.2 CPU Heap

Each data block in the CPU input buffer contains multiple heaps. Each heap contains a complete 1ms-integrated spectrum in single sub-band modes, or 8 complete 1ms-integrated spectra in 8 sub-band modes. The structure of a single heap is shown in Figure 9.

Each heap has a number of header fields, as described in Section 2.1. The structure of the spectrum data within the payload section of the heap is also described in Section 2.1. The size of a single heap depends on the number of sub-bands and spectral channels.

2.4 Structure of the Buffer at Input to the Disk Thread

The shared memory buffer at the input to the disk thread stores integrated spectra that are ready to be written to disk. Timestamps, frequency information and switching signal state information are also included for each integrated spectra. A single spectrum (or set



1	0x020	Time counter
1	0x021	Spectrum counter
1	0x022	Integration size
1	0x023	Mode
1	0x024	Status bits
0	0x025	Payload data offset

1 complete spectrum (for single sub-band modes)
OR
8 complete spectra (for eight sub-band modes)
8 bytes

Figure 9: A single heap within the CPU shared memory data buffer

of 8 spectra, in the modes where 8 sub-bands are used) is called a *data array*. The meta data associated with a single data array is stored in a `fits_data_columns` structure. Since a single data block may contain hundreds of data arrays and structures, an index is used to access each spectrum. The index for a single data block in disk shared memory buffer is shown in Figure 10.

Number of datasets (1 to N)	Size of single data array
Offset of 1st structure	Offset of 1st data array
Offset of 2nd structure	Offset of 2nd data array
...	
Offset of 8192th structure	Offset of 8192th data array

Figure 10: Index for a *single* data block in the disk input buffer

A dataset is defined as an instance of the `fits_data_columns` structure, followed by the associated spectral data (the data array). The index supports up to 8192 datasets, but again the actual number of datasets within a data block may be less. The size of a single data array is specified within the index, as it depends on the number of frequency channels. The size of the `fits_data_columns` structure is always fixed.

The offset fields, in the index, indicate the relative position of the start of the specified structure or data array. The offset is defined as the number of bytes from the beginning of the data block (and not from the beginning of the data buffer).

The actual data block itself simply contains many datasets, one after the other. As mentioned, a dataset is a structure followed by a data array. C code for the `fits_data_columns` structure is given below. Note that the `accumid` field has a value from 0 to 7, indicating the state of the switching signals for this integration.

```
struct sdfits_data_columns
{
    double time;            // MJD at start of integration (from Linux time)
    int time_counter;       // FPGA time counter at start of integration
    int integ_num;          // The integration number (a specific integ. period)
    float exposure;         // Effective integration time (seconds)
    char object[16];        // Object being viewed
    float azimuth;          // Commanded azimuth
    float elevation;        // Commanded elevation
    float bmaj;             // Beam major axis length (deg)
    float bmin;             // Beam minor axis length (deg)
    float bpa;              // Beam position angle (deg)

    int accumid;            // ID of the accumulator from where spectrum came
    int sttspec;            // SPECTRUM_COUNT of first spectrum in integration
    int stpspec;            // SPECTRUM_COUNT of last spectrum in integration

    float centre_freq_idx;  // Index of the NCHAN/2 frequency bin (0-indexed)
    double centre_freq[8];  // Frequency at centre of each sub-band
    double ra;              // RA mid-integration
    double dec;             // DEC mid-integration

    char data_len[16];      // Length of the data array
    char data_dims[16];     // Data matrix dimensions
    unsigned char *data;    // Ptr to the raw data (used internally only)
};
```

The *data array* can be regarded as a 3-dimensional array of floats, with the following structure:

```
float data[NUM_SUBBANDS][NUM_CHANS][NUM_STOKES]
```

where:
NUM_SUBBANDS = number of sub-bands specified mode [1 - 8]
NUM_SUBCHANS = number of frequency channels/bins [1024 - 32768]
NUM_STOKES = 4.

The number of sub-bands and number of frequency channels can be obtained from the FITS header for the data block (see section 2.2).

2.5 Status Shared Memory Buffer

The status shared memory buffer stores a number of variables in FITS format (i.e. as ASCII strings in 80-character fields). The buffer should therefore be read and written using a FITS-compatible reader. The fields in Table 1 must be set before the VEGAS HPC software is started, as these fields affect how the software operates. Note that these are all input parameters.

It is recommended that a script be used to set these parameters, according to the observation mode. See *Memo on the Critical Settings for the HPC* for the recommended values for these parameters, for each operating mode.

The fields in Table 2 are the telescope observation parameters, that are read from the M&C server. These fields are written to the Disk shared memory buffer, so that they can be written to the output FITS file. Therefore, none of these parameters are actually used by the HPC software. Again, these parameters are all inputs to the software.

Table 3 gives the output fields that indicate the status of the HPC software. These fields are written by the HPC software, and can be monitored by an external application. Most of these fields are self explanatory, with the possible exception of the status fields (NETSTAT, etc). These fields will typically take one of the following values: “init”, “waiting” (for incoming data), “receiving” (network packets), “processing”, “blocked” (no output block available) and “writing” (to disk).

The *BLKIN and *BLKOU fields indicate to/from which block (in a particular ring buffer) each thread is writing/reading. For a 24 block ring buffer, the blocks would have the ID numbers 0 to 23. These fields allow the number of free blocks in each ring buffer to be calculated in realtime, and hence identify any potential problems. For example, to calculate the number of *filled* blocks in the disk buffer, use:

```
(PFBBLKOU - DSKBLKIN) % NUM_BLOCKS_IN_BUF.
```

Table 1: Status Buffer Fields for Critical Settings [Inputs]

Field Name	**Data Type**	**Description**
NSUBBAND	Integer	Number of sub-bands (1 to 8)
NPOL	Integer	Number of antenna polarisations (must be 2)
NCHAN	Integer	Number of frequency channels per sub-band
CHAN_BW	Double	Width of each spectral channel/bin [Hz]
EXPOSURE	Float	Required integration time [s]
HWEXPOSR	Float	Hardware integration time (on FPGA or GPU) [s]
FPGACLK	Float	FPGA clock rate [Hz]
EFSAMPFR	Float	Effective sampling frequency (after decimation) [Hz]
SUB0FREQ	Double	Centre frequency of sub-band 0 [Hz]
SUB1FREQ	Double	Centre frequency of sub-band 1 [Hz]
SUB2FREQ	Double	Centre frequency of sub-band 2 [Hz]
SUB3FREQ	Double	Centre frequency of sub-band 3 [Hz]
SUB4FREQ	Double	Centre frequency of sub-band 4 [Hz]
SUB5FREQ	Double	Centre frequency of sub-band 5 [Hz]
SUB6FREQ	Double	Centre frequency of sub-band 6 [Hz]
SUB7FREQ	Double	Centre frequency of sub-band 7 [Hz]
DATAHOST	String	Hostname of attached ROACH board
DATAPORT	Integer	UDP port to which ROACH board transmits packets
PKTFMT	String	Network packet format (must be SPEAD)
DATADIR	String	FITS output directory (only when disk thread used)
FILENUM	Integer	File number in multi-file scan (reset to zero for each scan)

2.6 Programmatic Access to the Shared Memory Segments

The shared memory segments can be programmatically accessed using their keys and identifiers. The following description applies to the dual-instance version of VEGAS.

On a PC on which dual-instance shared memory is set up, running `ipcs` should show something similar to the following:

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x4019ccf5 131072     owner      666        184320     0
0x8019ccf5 163841     owner      666        1081745664 0
0x8019ccf6 196610     owner      666        1081745664 0
0x8019ccf7 229379     owner      666        408658112  0
0x4119ccf5 262148     owner      666        184320     0
0x8119ccf5 294917     owner      666        1081745664 0
```

Table 2: Status Buffer Fields for Telescope Parameters [Inputs]

Field Name	**Data Type**	**Description**
TELESCOP	String	Name of telescope
PROJID	String	Project ID No
OBSFREQ	Double	Centre frequency of observation [Hz]
OBSBW	Double	Entire backend bandwidth for observation [Hz]
OBSNCHAN	Double	Number of original frequency channels/bins
FRONTEND	String	Name of observation frontend
INSTRUME	String	Name of observation backend (always "VEGAS")
SCANNUM	Integer	Scan number in single observation
OBJECT	String	Object being viewed
STTMJD	Double	Commanded observation start time [MJD double]
TSYS	Double	System temperature
FILTNEP	Float	Filter noise-equivalent parameter (noise of PFB)
AZ	Float	Commanded azimuth [deg]
ELEV	Float	Commanded elevation [deg]
RA	Double	Right-ascension [deg]
DEC	Double	Declination [deg]
BMAJ	Float	Beam major-axis length [deg]
BMIN	Float	Beam minor-axis length [deg]
BPA	Float	Beam position angle [deg]
CAL_MODE	String	Calibration mode (OFF, SYNC, EXT)
CAL_FREQ	Double	Calibration modulation frequency [Hz]
CAL_DCYC	Double	Calibration duty cycle
CAL_PHS	Double	Calibration phase (w.r.t. start time)

```
0x8119ccf6 327686     owner      666        1081745664 0
0x8119ccf7 360455     owner      666        408658112  0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x8019ccf5 65538      owner      666        32
0x8019ccf6 98307      owner      666        32
0x8019ccf7 131076     owner      666        1024
0x8119ccf5 163845     owner      666        32
0x8119ccf6 196614     owner      666        32
0x8119ccf7 229383     owner      666        1024
```

The first four shared memory segments and the first three semaphores correspond to

Table 3: Status Buffer Fields for Reporting on Status of HPC Software [Outputs]

Field Name	**Data Type**	**Description**
NPKT	Integer	Number of packets received from ROACH
NDROP	Integer	Number of packets dropped
DROPAVG	Double	Current packet drop rate
DROPTOT	Double	Overall packet drop rate
NETSTAT	String	Status of network thread
GPUSTAT	String	Status of GPU thread
ACCSTAT	String	Status of CPU accumulator thread
DISKSTAT	String	Status of disk thread
NETBLKOU	Integer	ID number of current output block for network thread
PFBBLKIN	Integer	ID number of current input block for PFB thread
PFBBLKOU	Integer	ID number of current output block for PFB thread
ACCBLKIN	Integer	ID number of current input block for accumulator thread
ACCBLKOU	Integer	ID number of current output block for accumulator thread
DSKBLKIN	Integer	ID number of current input block for disk thread
DSKEXPWR	Integer	The number of exposures written to disk
M_STTMJD	Double	Measured observation start time (MJD; microsec resolution)
M_STTOFF	Double	Measured observation start time offset (fraction of second)
SWVER	String	Version number of the VEGAS HPC software

instance 0, while the rest corresponding to instance 1. The key generation functionality is implemented in the files `vegas_ipckey.c` and `vegas_ipckey.h`. Keys are generated using the `ftok()` function (see manpage), which takes a `proj_id` built in the following way:

For data buffers, `proj_id = (instance_id & 0x3f) | 0x80`, and for status buffers, `proj_id = (instance_id & 0x3f) | 0x40`. For instance IDs 0 and 1, this would correspond to `0x80` and `0x81` respectively for data buffers and `0x40` and `0x41` respectively for status buffers. That is, the keys starting with `0x40` and `0x80` correspond to instance 0, and those starting with `0x41` and `0x81` correspond to instance 1.

The instance ID is passed as a command-line argument to `vegas_hpc_hbw` and `vegas_hpc_lbw`, and this is used by their processes to attach the relevant shared memory segments to their address spaces, and use the relevant semaphores, as demonstrated by the following example:

```
struct vegas_status st;
rv = vegas_status_attach(instance_id, &st);
...
struct vegas_databuf *db;
db = vegas_databuf_attach(instance_id, args->output_buffer);
```

Internally, these functions use `instance_id` and the shared memory key to get the shmid, as shown below.

```
shmid = shmget(key + databuf_id - 1, 0, 0666);
```

Here, `databuf_id` may be 1, 2, or 3 depending on whether the data buffer shared memory identifier in question is at the output of the network thread, the GPU thread, or the accumulation thread, respectively.

3 The CPU-based HPC Processing Threads

In this section, the four data processing threads within the VEGAS HPC software, that run on the CPU, will be discussed. Since the GPU Thread operates quite differently to the other four threads, it is described separately in Section 4.

3.1 The Main Threads

The two "main" threads are `vegas_hpc_hbw` and `vegas_hpc_lbw`. These threads are responsible for starting the data processing threads depicted in Figures 1 and 2. Note that only one of these two threads never run simultaneously.

These two threads first attempt to attach to the shared data buffers. If they do not exist yet, they are created with default block sizes. The data blocks in the disk input buffer are then resized according to the EXPOSURE parameter (see below for more details). The network, PFB, accumulator and disk threads are then launched. Depending on the mode and command line parameters, not all four threads may be launched.

The main thread then waits for the global variable `run` to be set to zero. This happens if an error occurs in one of the data processing threads, or if the user presses Control-C. In either case, the main thread kills all the other threads and closes the program.

3.1.1 Dynamic Block Resizing

If the data blocks in the disk input buffer are large, and a long integration time is used, data blocks will be written to hard disk very infrequently (possibly only once an hour). This means that if the program were to crash, up to an hour of data could be lost. To prevent this, the main thread resizes the data blocks in the disk input buffer so that these data blocks fill up every 20 seconds.

The size of the blocks in the disk buffer are calculated as follows (where EXPOSURE, NCHAN and NSUBBAND are all parameters that are set via the status shared memory):

```
DISK_WRITE_INTERVAL = 20 seconds
num_exp_per_blk = DISK_WRITE_INTERVAL / EXPOSURE
exposure_data_size = NCHAN * NSUBBAND * 4 * 4
disk_block_size = num_exp_per_blk * exposure_data_size
disk_block_size = min(disk_block_size, 32*1024*1024)
```

The last step ensures that the disk block size does not exceed 32MB. Once the calculation is done, the disk buffer is then reconfigured so that it uses the new block size. The number of data blocks in the buffer is adjusted accordingly, so that the overall buffer size remains constant (as determined by the initial user configuration).

When the buffers are first created (typically using the scripts), an array of 1024 semaphores is created for the disk input buffer. This therefore limits the number of blocks per buffer to 1024. Typically, there will be far less than 1024 blocks in the disk buffer, resulting in only a few of the 1024 semaphores being used. However, if the resizing operation results in blocks that are very small, causing there to be more than 1024 buffer blocks, this will be limited to 1024 blocks, resulting in a reduction in the overall disk buffer size.

3.2 Operations Common to all Four Data Processing Threads

The four data processing threads, described in the sections below, have common initialization operations. These common operations are:

1. Each thread sets its CPU affinity, so that each thread runs on a different physical processor. Since the target processor uses hyperthreading, this means that each thread is allocated to every second CPU.
2. The thread then attaches to the status shared memory, and reads the parameters from the shared memory into `vegas_params` and `sdfits` structures. This allows the thread to configure its internal operations according to the parameters in the status shared memory.
3. The thread attaches to its input and output shared data buffers.
4. The thread then enters an infinite loop in which the actual work is done. This loop terminates when one of the threads sets the global `run` variable to zero.

3.3 The Network Thread

The network thread performs the following main operations in its processing loop:

1. Waits for a UDP packet to arrive, using the Linux `poll` command.
2. The packet is checked to ensure that it has a valid header, and that the packet is the correct length.
3. The *heap counter* and *heap offset* fields are combined to form a unique packet number. This packet number is compared to the packet number of the previously received

packet. If the packet number decreased by more than 1024, then the observation begins (i.e. the ARM signal was sent to the FPGA, causing the FPGA to reset its heap counter).

4. If the packet number is the same as the previously received packet, the packet is discarded (duplicate packet). If the packet number is slightly smaller than the previous packet, the packets were sent out of order, and are discarded.

5. If the software has not yet detected the start of the observation (as indicated by the condition in point 3), the packet is discarded. If the observation has begun, the packet is processed according to the following steps.

6. The received packet is then written into the CPU or GPU input buffer (depending on the mode). Within a particular data block in the buffer, all the SPEAD headers (excluding the first 40 bytes) are placed at the beginning, while the payloads are placed at the end. This is to allow fast bulk-copying of data onto the GPU.

   Note that the GPU and CPU buffers store heaps, and not buffers. This means that in high-bandwidth modes, all the packets that form a single heap (i.e. single spectrum) will have just one SPEAD header, and all their payloads will be concatenated together to form a single block of spectral data. Furthermore, in high-bandwidth modes, the byte ordering of the floating-point numbers is reversed when writing the data to the CPU input buffers, to correct the endianess..

7. The index is also updated with the *heap counter*, the MJD time (obtained from Linux time) and whether any packets were dropped within this heap (*valid* flag).

8. If the buffer block is not yet full, go to Step 1. If however the buffer block is full, the semaphore corresponding to that block in the shared semaphore array is set, informing the next thread that it can process the data in the block. The network thread then waits for an empty block to become available in the GPU or CPU input buffer, before going to Step 1.

3.4 The PFB Thread

The main loop in the `vegas_pfb_thread.c` file simply waits for data blocks in the GPU input buffer to become available. Whenever one becomes available, it calls the `do_pfb(...)` function in the GPU code with a pointer to the data block. When the function returns, it marks the input block as available (so that the network thread can refill it at some later point) and then goes back to waiting for another input block.

3.5 The CPU Accumulator Thread

Before entering the main processing loop, `vegas_accum_thread.c` dynamically allocates memory for the 8 vector accumulators (one for each switching state). The 8 vector accumulators are stored as a 2D floating-point array that can be described by:

```
float accumulator[NUM_SW_STATES][num_chan * num_subband * NUM_STOKES]
```

Note that the accumulators always store floating-point numbers, and that `NUM_SW_STATES = 8`. Even though the vector accumulators are stored as a single 2D array, they are logically indexed as a 4D array with the following structure:

```
float accumulator[NUM_SW_STATES][num_chan][num_subband][NUM_STOKES]
```

There are also two other arrays that are associated with the accumulators:

```
char accum_dirty[NUM_SW_STATES]
struct sdfits_data_columns data_cols[NUM_SW_STATES]
```

The `accum_dirty` array stores flags indicating whether a particular accumulator has had anything added to it this integration cycle. The `data_cols` array stores the metadata associated with each integrated spectrum, such as the timestamp at the start of the integration, the total integration time and the antenna position.

After these additional initialization steps, the CPU Accumulator Thread enters its main loop, performing the following steps:

1. Wait for a block in the CPU Input shared memory buffer to become full.
2. For each heap in the input buffer block:
   (a) If the heap is marked as bad (invalid) in the index in the CPU input buffer, it is ignored. Also, if the blanking bit is high (bit 3 of *Status bits* field in SPEAD heap header), the heap is ignored.
   (b) Provided the heap is not ignored, it is added to one of the 8 vector accumulators, based on bits 2:0 of the *Status bits* field in SPEAD heap header. In the case of high-bandwidth modes (where the FFT is done on FPGA), the 32-bit integers are converted to 32-bit floats before being adding to the accumulators.

(c) If the accumulator was all zeroes before adding this heap (spectrum), the dirty bit for that accumulator is set. The timestamp for the integration is also set using the timestamp from the input heap. This means that the vector accumulator (for that particular switching state) has the timestamp of the first spectrum in the accumulation.

(d) The total accumulation time (`accum_time`) for this integration cycle is updated using:

```
pfb_rate = EFSAMPFR / (2 * NCHAN)
accum_time += integ_size / pfb_rate
```

where EFSAMPFR is the effective sampling frequency of the FPGA (set via status shared memory), NCHAN is the number of channels in the FFT (also set via the status shared memory), and `integ_size` is the number of spectra that were added together, in either the FPGA or the GPU, to produce the input heap (spectrum). The `integ_size` parameter is obtained from the SPEAD header of the input heap.

(e) The exposure for that particular vector accumulator is also increased, using the same formula as above.

(f) If the total `accum_time` $\geq$ EXPOSURE (i.e. the EXPOSURE parameter in the status shared memory), all the *dirty* accumulators are written to the output buffer (i.e. the Disk input buffer). Each dirty accumulator is written as a separate dataset, with its own `fits_data_columns` structure and data array, to the disk input buffer. Non-dirty accumulators are not written. All the dirty accumulators are then zeroed out for the start of the next integration cycle.

3. The block in the CPU input buffer is marked as free, so that it can be reused by the Network Thread or PFB Thread at some later time.

4. Go to Step 1.

3.6 The Disk Thread

The disk thread performs the following main operations in its processing loop:

1. Wait for a block in the Disk Input shared memory buffer to become full.

2. Each dataset in the input buffer block is passed to the `sdfits_write_subint` function, which writes that particular dataset to the SDFITS output file. Note that each dataset appears as a separate, single line in the SDFITS output file.

3. The block in the Disk input buffer is marked as free, so that it can be reused by the CPU Accumulation Thread at some later time.

4. Go to Step 1.

4 The GPU Thread

The GPU thread is instantiated only for the low-bandwidth modes of operation of VEGAS. The GPU thread reads data from the GPU buffer, performs PFB/FFT and accumulation for 1ms, and writes the accumulated spectra to the CPU buffer for the consumption of the CPU thread.

4.1 Files

The following are the files relevant to the GPU thread. In `$VEGAS/vegas_hpc/src`:

`vegas_pfb_thread.c`: Entry point of the GPU thread
`pfb_gpu.cu`: Host functions
`pfb_gpu.h`: Host functions header file
`pfb_gpu_kernels.cu`: Device kernels
`pfb_gpu_kernels.h`: Device kernels header file

In `$VEGAS/gpu_dev`:

`vegas_gencoeff.py`: Filter coefficient generation script

4.2 Code Flow

The basic code flow of the GPU thread is listed below. Here, 'host' refers to CPU/RAM and 'device' refers to GPU/associated memory. The term 'kernel' refers to the CUDA/C function that runs on the GPU. For further details on CUDA programming, see the NVIDIA CUDA C Programming Guide.

1. Initialization (`pfb_gpu.cu`: `init_gpu()`)
    (a) Choose a CUDA device (hardcoded to device 0).
    (b) Get device properties.
    (c) Allocate memory for filter coefficients array, read filter coefficients file (see §4.4), and load values.
    (d) Allocate memory for data arrays.
    (e) Calculate CUDA kernel parameters.
    (f) Create FFT plan (one plan for all FFTs, using `cufftPlanMany()`).

2. Set status to 'waiting' and wait for input data buffer to fill up.
3. Set status to 'processing' and process data (`pfb_gpu.cu`: `do_pfb()`).
   (a) Copy entire data in current input block to device.
   (b) If all heaps that go into one PFB operation (`VEGAS_NUM_TAPS * g_iNumSubBands * g_nchan` samples) are valid, perform polyphase filtering (for details of the algorithm, see the CASPER Memo on the PFB technique). If there is any invalid heap, skip all heaps that go into this PFB.
   (c) Perform FFTs. All FFTs (2 for the 1-sub-band modes, and 16 for the 8-sub-band modes) are executed in parallel.
   (d) If the blanking bit is not set, accumulate for 1ms, copy the accumulated spectrum back to the host, and write it to the CPU buffer. If the blanking bit just turned on, copy whatever has been accumulated till then back to the host and dump it to the CPU buffer. Zero the accumulators.
   (e) If the current output block is full, set status to 'blocked' and wait for the next one to be available.

The number of spectra to accumulate, corresponding to a time of 1ms, is computed in `vegas_pfb_thread.c` as

```
acc_len = abs(CHAN_BW) * HWEXPOSR
```

where CHAN_BW is the channel bandwidth and HWEXPOSR is the hardware integration time, both of which are set by the user in the status shared memory.

The dual-polarization complex samples, that are interleaved as explained in the previous sections, are read into a CUDA `char4` array. To elaborate, if the variable is named `c4Data`, the four elements that make up this variable would contain the following data:

`c4Data.x`: Real(X-pol.)
`c4Data.y`: Imag(X-pol.)
`c4Data.z`: Real(Y-pol.)
`c4Data.w`: Imag(Y-pol.)

The spectra are written in the following format. The length of each block is given in parentheses, in units of samples.

```
---------------------------------------------------
| PowX (1) | PowY (1) | Re(XY*) (1) | Im(XY*) (1) | (Interleaved samples)
---------------------------------------------------
```

4.3 Notes on Notation

A quasi-Hungarian notation is used in most of the CUDA code. Example:

```
#define DEF_NFFT    1024    /* ‘DEF_’ denotes default values */

int g_iVar; /* global variable */

<ret-type> Function(<args>)
{
    float fVar;
    int iVar;
    double dVar;

    /* CUDA types */
    char4 c4Var;
    float2 f2Var;
    dim3 dimVar;

    /* pointers */
    char *pcVar;
    int *piVar;

    /* arrays */
    float afVarArray[10];

    ...
}
```

4.4 PFB Filter Coefficients

The PFB filter coefficients are read from a file during initialization. This file is expected to be in the `$VEGAS/vegas_hpc` directory. Depending on the mode, the GPU thread needs to read in different sets of coefficients, saved in different files. The file naming convention is as follows. The file name is composed of different segments, each separated from the neighbouring segments by an underscore, as shown below:
`coeff_<data-type>_<taps>_<nfft>_<sub-bands>.dat`

The first segment is always the string ‘`coeff`’. The second segment is the data type of the coefficients, which, for the purpose of VEGAS, is always single-precision floating point,

and hence, '`float`'. The third segment is the number of taps of the PFB, followed by the number of channels, and the number of sub-bands. As examples, the sets of coefficients that VEGAS needs are pre-computed and saved in the following two files:
`coeff_float_8_4096_8.dat`: 8 taps, 4096 channels, 8 sub-bands (modes 13-17)
`coeff_float_8_32768_1.dat`: 8 taps, 32768 channels, 1 sub-band (modes 6-12)

Generation of filter coefficients afresh is not necessary unless implementing a new mode with a different number of channels and/or sub-bands, in which case, the Python script `vegas_gencoeff.py` can be used for the purpose. The usage of the script is as follows:

```
Usage: vegas_gencoeff.py [options]
    -h  --help                  Display this usage information
    -n  --nfft <value>          Number of points in FFT
    -t  --taps <value>          Number of taps in PFB
    -b  --sub-bands <value>     Number of sub-bands in data
    -d  --data-type <value>     Data type - "float" or "signedchar"
    -p  --no-plot               Do not plot coefficients
```

The data type tells the program whether to output single-precision floating point coefficients or signed chars in the range [-128, 127]. Note that VEGAS can only accept single-precision floating point. The number of sub-bands does not actually affect the coefficients themselves, but is included as an optimization feature – each coefficient repeats that many times, for ease of GPU thread indexing. The output is a binary file.

This Python script requires the NumPy and matplotlib modules.

As for the values of the filter coefficients, they are generated in the following manner:

```
M = NTaps * NFFT
X = numpy.array([(float(i) / NFFT) - (float(NTaps) / 2) for i in range(M)])
PFBCoeff = numpy.sinc(X) * numpy.hanning(M)
```

That is, the coefficients array is a sinc function multiplied by a Hanning window. This code can be modified, if needed, to create different sets of coefficients.