DBE Memo#013
UVLBI MEMO #025
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
HAYSTACK OBSERVATORY
WESTFORD, MASSACHUSETTS 01886
April 26, 2011

To: UVLBI Group
From: G.B. Crew

A Burst Mode Recorder

Notions, Realization and Testing

G. B. Crew

Short Contents

Table of Contents

Overview

This document (available as uVLBI Memo #25 from Haystack) attempts to collect the various considerations leading to an implementation of a burst mode recorder (BMR) system for use with existing VLBI systems. Such systems are evolving to deliver ever increasing **sustained** data recording rates (4 Gbps and greater) in a variety of data formats (Mark 5B, Mark 5C, VDIF, ...). However, there are many reasons to want a system that can capture data at still higher rates for brief periods separated by non-capture intervals for an agregate rate comparable to what can be achieved with a sustained recording system. Indeed, while the VLBI technique requires carefully-timed recordings at remote stations, it generally does not require continuous recording for scans of long duration—atmospheric turbulence usually limits coherence to at most a few minutes.

This document is aimed towards pinning down some of the various options in order to produce a workable system in the near-term that can be extensible to most foreseen future needs. It then describes an initial implementation (designed for testing the capabilities of the hardware) that could also be used for field observations. Some basic tests with this system were performed in the lab and are described here.

For the purposes of this discussion, we can summarize the basic goal as the ability to capture data from some number (D) of digital data sources (DDSs) generating data at a continuous rate of 4 Gbps in the form of UDP packets delivered on a 10 Gbps ethernet link. Thus we are talking about a per-station recording rate of 4D Gbps. For example, the iBob DBE deployed at a number of sites supports two 512 MHz IFs, which, with 2bit sampling at the Nyquist rate would be one such DDS if it provided data via the 10 Gbps ethernet interface.

Within an order of magnitude, we want a capture session lasting about a minute. Ignoring manufacturing hype, current **sustained** disk storage rates are on the order of 125 MBps to a single disk. The captured data needs to be deliverable to some correlator system in a convenient fashion (disks, disk files, network sockets, ...).

For reference, these numbers (crudely) imply:

```
4 Gbps = 1.8 TBph = 11 TB / 6h night
1 m at 4 Gbps = 30 GB
30 GB / 125 MBps = 4 m
```

for 4 Gbps systems, and

```
8 Gbps = 3.7 TBph = 22 TB / 6h night
1 m at 8 Gbps = 60 GB
60 GB / 125 MBps = 8 m
```

for 8 Gbps systems. (Here we use s, m, h for seconds, minutes, hours; b, B for bits and bytes, and M, G, T are the usual mega, giga, tera metric prefixes, but with **approximate** usage: in VLBI parlance, 1 Gbps usually means 1024000000 bits per second which is neither fish nor fowl in the usual (SI units or computer) usage.)

Thus a 1:1 duty cycle (i.e. one minute of capture only followed by one minute of recording only) requires 4 disks, each with a 2.7 TB capacity at 4 Gbps. For 8 Gbps, we'd need 8 3 TB disks. Obviously, various bits of cleverness (or lack of it) as well as variations to the

observing program will improve or degrade these basic numbers. These numbers should then be multiplied by (D S) for an S-station network.

An additional point to make is that, while targetted towards the burst application, the implementation should not preclude an evolution to a(n eventual) sustained recording system.

With these basic numbers in mind, the next sections cover the existing hardware and some future options, then some current benchmark numbers, sections on the software design, and finally some results from testing an initial implementation using a modified Roach DBE (`RDBE-S`).

1 Burst Mode Hardware

Before moving on to the software considerations, we briefly document the initial testbed hardware and the second generation burst-mode hardware.

1.1 Testbed System Components

A prototype test-bed system (`Maxwell`) was purchased and assembled from components then available (ca. 2009) to provide the baseline test numbers mentioned in the next section of this document. It consists of a Debian ("Lenny") Linux system with:

- SuperMicro Computer (SC745TQ-R800B) Chassis
- SuperMicro serverboard (X7DWN+: UIO, 1600FSB dual Quad-Core Xeon [Seaburg], Intel 4500 chipset)
- Quad-Core Intel Xeon CPU E5462 @ 2.80GHz
- 8+16 GB DDR2 800 SDRAM Fully Buffered DIMM (FB-DIMM; Kingston ValueRam KVR800D2D4F5/2G modules in 4 slots, Kingston ValueRam KVR800D2D4F5/4G modules in 4 slots, w/8 empty slots)
- SuperMicro 10 Gb Ethernet Card (AOC-STG-i2) with a dual port Intel 82598EB controller and two CX4 connectors, in a PCI-E x8 slot
- SuperMicro 8-port SATA Card (AOC-SAT2-MV8) via PCI-X bus
- (one) Barracuda 7200.11 SATA 3Gb/s 640-GB
- (eight) Hitachi HDT721032SLA360 320 GB disks
- (one) INTEL SSDSA2MH16 solid state disk
- et cetera.

The E5462 is of the "Harpertown" series of the Xeon processor family. This CPU and its peripherals are connected via the bus architecture similar to that shown in Figure 1.1.

The front side bus (FSB) is 1600 MHz in our system, and is the likely bottleneck for our burst mode recording needs. That is, the data flow from the network interface to RAM memory is via the CPU but it must travel through the memory controller at least twice. (I.e. from a PCI Express [PCIe] card to the CPU and thence to system memory.) While there is a motherboard socket for an additional CPU and a second ethernet card, it's not at all clear that the performance would double—rather the load on the memory controller hub is merely doubled.

For testing, a slower companion machine (`mjg-pc`) was used. Its components were not selected for this program, except for the same 10 Gb Ethernet Card. Its performance was considerably worse.

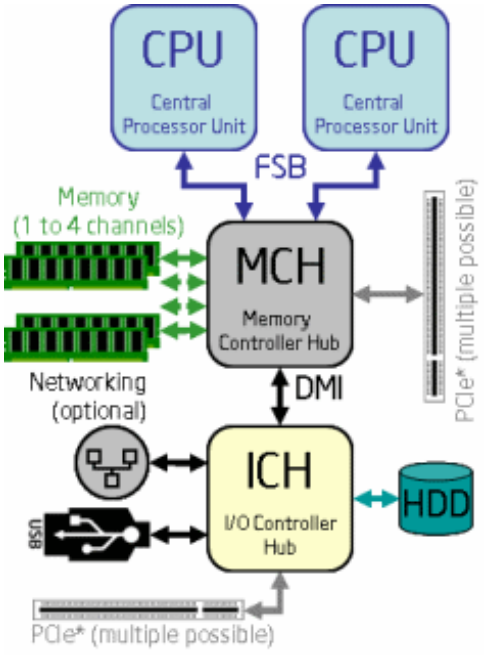

Figure 1.1: Bus Architecture for Initial Test-bed System (Maxwell).

1.2 Next Generation Testbed System

Intel has (of course) been aware of the system i/o bottleneck for some time—the more recent "Nehalem" architecture rearranges the controllers as shown in Figure 1.2 so that there is a faster path between the PCI Express bus and the core processors and also a faster path between the processor cores and memory.

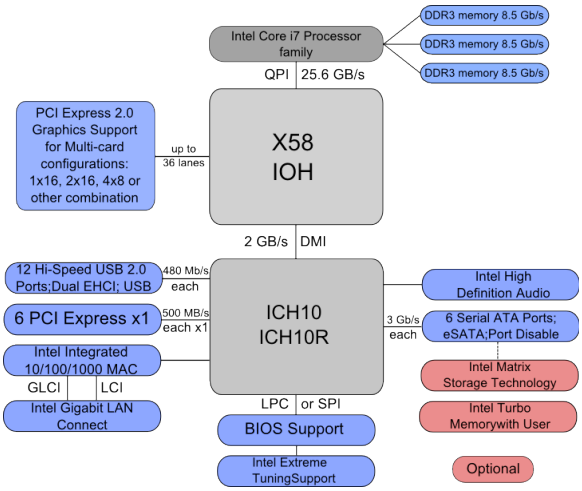

Figure 1.2: Nehalem Bus Architecture in Next Generation System (Monarth).

In this architecture the (DDR3) memory installed is directly accessible directly to the CPU cores. Now, during the capture process the data need only make one trip through the I/O handler (X58 in the Figure 1.2. It is thus likely to be much more efficient. The Quick Path Interconnect (QPI) architecture is rated for 25.6 GB/s (bi-directional) so the hardware speed limit in this architecture is likely to be the number of 10 Gb ethernet devices that can be usefully connected. We assembled a system (`Monarth`) using this architecture, with:.

- SuperMicro Computer (745TQ-R1200B) Chassis
- SuperMicro motherboard (X8DAH+) which supports up to 192GB RAM (18 slots), 2 (x16) PCI-E 2.0, 4 (x8) PCI-E 2.0 (1 in x 16 slot), 1(x4) PCI-E (in x8 slot)
- Quad-Core Intel Xeon E5630 CPU @ 2.54GHz (12M cache, 5.86 GT/s QPI)
- 24 GB DDR3 1066MHz Ram
- SuperMicro 10 Gb Ethernet Card (AOC-STG-i2) with a dual port Intel 82598EB controller and two CX4 connectors, in a PCI-E x8 slot
- SuperMicro 8-port SATA Card (AOC-SASLP-MV8) via PCI-E bus
- (eight) Western Digital WE20EARS 2 TB (Caviar Green) disks
- et cetera.

In particular, we were able to find a motherboard (`X8DAH+`) that supports two processors and uses two of the X58 chipsets. With one CPU and one ethernet card it **should** do no worse than `Maxwell` which appears capable of 8 Gbps in burst mode. Assuming the two "sides" (i.e. one CPU + X58) do not interfere, a 16 Gbps capability should thus obtain with the new machine.

1.3 Additional Hardware Options

Additional ethernet cards could be added to the PCI-Express bus in the Nehalem-based testbed. While the QPI network should not be a bottleneck, it is possible the shared cache of the CPU cores, or the memory access might become the new bottleneck. This is still to be investigated.

Moving beyond data capture, if the data is to be stored within this system there is the basic choice of whether it is to be stored on internal disks, or else onto external disks. Either way, there is the additional question of whether a hardware RAID controller should be part of the mix, or whether the disks are RAIDed together in software, or else are accessed individually.

Some of the potential sites are at high altitude where disk performance becomes an issue (e.g. Mauna Kea in Hawaii). Potentially, disks can be found which will still work (as has been done in the past). Otherwise, some means of pressurizing (and controlling heat) either the entire system or merely the (external) disk pack would need to be found.

In any case, the SATA card (`AOC-SASLP-MV8`) can connect 8 disks to the system via a pair of SAS connectors that subdivide into 4 separate connectors for the disks. Thus it should be straightforward to connect up external disks (suitably) packaged in multiples of 4.

We will leave hardware for now and move on to software. (Obviously with a candidate software system it will be necessary to revisit the hardware choices for deployment in the field, but that's a topic for later.)

2 Initial Benchmarks on Maxwell

With the initial testbed system (`Maxwell`), we established that UDP packets of the type expected by the Mark5C (5008B / packet) can be captured to RAM reliably at 4 Gbps and that 8 Gbps (so packetized) initially did not appear to be possible. A simulator (`Mark5CPacEmu`) running on a slower machine (`Mjg-pc`) was used, so the numbers in the next two sections largely reflect the limitations of that system.

The initial testbed system appears to be able to manage a sustained data write just short of 4 Gbps across a RAID0 built with 8 disks.

2.1 Status on June 25, 2010

Some initial numbers using simple programs `push_test`, `grab_test` and `m5L2`:

Test Name	**MarkPac5CEmu@5.5Gbps (50 GB buffer)**	**push_test@10.0Gbps (50 GB buffer)**
maxwell: recv only	5.7 top, 5.5 avg	5.2 top, 4.6 avg
maxwell: recv to RAM	5.7	4.6
maxwell: SSD	0.6	0.6
maxwell: primary HD	0.7	0.7
maxwell: RAID0	4.3 top, 3.4 avg	3.7 top, 3.4 avg
maxwell: RAID5	2.1	2.1

A couple of things worth mentioning:

- Tests near the top of the range tended to be erratic, varying by up to 1 Gbps.
- The CPU% for the grab process never exceeded ~60-70%.
- For the RAID0 tests, there were a few other processes taking up to 7% CPU each: pdflush, kswapd0, xfsdatad, kjournald. These are related to page swapping and disk management. (Does it make sense that we should be paging out a lot?)
- `push_test` seems to yield lower rates than `m5L2`, but delivers increasing returns as you bump up the rate–is that due to the timing issues? I tried to take it over 10.0, but it wouldn't run (I'm guessing maybe it's running into a data type problem, but I haven't checked yet.)

Bottom line, I'm still kind of unhappy with the inconsistency of the results. These numbers are averages of several ~15-minute tests though, so they should be okay in that respect.

2.2 Status on July 21, 2010

A working GRAB plus PUSH process (`bmr_buffer`, see Section 5.2 [bmr_buffer], page 21) has been implemented and appears to work on `Maxwell` using packet source and sinks on `Mjg-pc`. Commands such as the following:

```
# these commands:
mjg-pc$ grab_test -v -H 192.168.1.9:2652 -n 10000000 grab
maxwell$ while true ; do sleep 1 ; echo counters ; done |\
```

```
         bmr_buffer -v \
            -c grab:addr=192.168.1.10 -c push:addr=192.168.1.9 \
            -c grab:gbps=5.0 -c push:gbps=2.5
mjg-pc$ Mark5CPacEmu/server/m5L2 -u 192.168.1.10 \
    -l 5008 -fb -t 10000 -r 5000
# produces:
  1279737169.369644768  GrabRT: 5.00913 Gbps (19.999950969 s)
  1279737169.369645608  SendRT: 2.50456 Gbps (39.999976627 s)
  1279737169.369644768  Active: 12522794464B 2500558P 0z 0e 0c
  1279737169.369644768 Passive: 1991716656B 397707P 0z 0e 0c
  1279737169.369645608 Sending: 12522794464B 2500558P 0z 0e 4022770c
  1279737169.369645608  Idling: 0B 0P 0z 0e 35989826c
  1279737169.369645608 TooSoon: 0rw 1522212inv 0rl 2423916slp
```

suggest that `bmr_buffer` could receive at 5 Gbps and dump to a Mark5C at 2.5 Gbps. The `Active` and `Sending` refer to bytes and packets in the the capture and delivery parts of the cycle—the other counters refer to various other debugging diagnostics.

```
# these commands:
mjg-pc$ grab_test -v -H 192.168.1.9:2652 -n 10000000 grab
maxwell$ while true ; do sleep 1 ; echo counters ; done |\
         bmr_buffer -v \
            -c grab:addr=192.168.1.10 -c push:addr=192.168.1.9 \
            -c grab:gbps=5.0 -c push:gbps=2.5
mjg-pc$ push%test -v -H 192.168.1.10:2650 \
    -n 100000000 -z 5008 -r 0.0 send
# produces:
  1279740211.884492708  GrabRT: 4.5677 Gbps (20.000076978 s)
  1279740211.884493441  SendRT: 2.28386 Gbps (39.999905258 s)
  1279740211.884492708  Active: 11419296688B 2280211P 0z 0e 0c
  1279740211.884492708 Passive: 2188440912B 436989P 0z 0e 0c
  1279740211.884493441 Sending: 11419276656B 2280207P 4z 4e 7778607c
  1279740211.884493441  Idling: 0B 0P 0z 0e 29763220c
  1279740211.884493441 TooSoon: 0rw 5498396inv 0rl 2213853slp
```

The `Mark5CPacEmu` does a more careful rate emulation; `push_test` has a lower maximum rate, but is still able to deliver 4.5 Gbps to `bmr_buffer`. In this case, there were errors sending (only) 4 of the (2+ million) captured packets.

2.3 Status on Sept 16, 2010

It was subsequently determined that the network card was not plugged into an x8 PCI-Express slot (rather, an x4 slot). That was one limitation. Further, the linux kernel normally expects to swap pages of memory out to disk well before memory is fully in use. There are ways to disable this (in software); however it is rather straightforward to find "safe" operating conditions (i.e. shorter grab periods) where the kernel is willing to leave the pages being used for capture in memory.

A further consideration is that the CPU cores are able to operate with some parallelism, despite the apparent bottleneck in transferring packets from the NIC card to the CPU cache. Thus, operating with 2 receiving processes results in a significant improvement in capture capability.

Thus two CX4 cables given separate ethernet addresses can be carrying packets addressed for capture by 2 separate cores. Finally, the maximum ethernet packet size is 9000B. Using a larger VDIF packet (8KB) rather than 2 smaller Mark5B packets results in less work-per-

packet for the CPUs. This last step of testing was only possible when the initial version of the `Astro8Gbps` personality for the `RDBE` (see Chapter 6 [rdbeusage], page 29) became available.

Testing with `bmr_record` (see Section 5.3 [bmr_record], page 22):

```
# grab:secs=4 grab:flush=1 grab:period=10
A-9: 1284668054.968918370  GrabRT: 4.10459 Gbps (57.395267084 s)
A-9: 1284668054.968918915 WriteRT: 1.92739 Gbps (122.348702018 s)
A-9: 1284668054.968918370  Active: 29448037248B 3577264P 0z 0e 0c
A-9: 1284668054.968918370 Passive: 326898454B 39515P 6z 0e 0c
A-9: 1284668054.968918915  Saving: 29476655360B 3577264P 0z 0e 3577415c
A-9: 1284668054.968918915  Idling: 0B 0P 0z 0e 0c
A-9: 1284668054.968918915 NoWrite: 151rw 0inv 0rl 3577264ok

B-9: 1284668054.968656534  GrabRT: 4.09523 Gbps (57.532007885 s)
B-9: 1284668054.968657204 WriteRT: 1.92949 Gbps (122.226842610 s)
B-9: 1284668054.968656534  Active: 29450844360B 3577605P 0z 0e 0c
B-9: 1284668054.968656534 Passive: 331473378B 40065P 3z 0e 0c
B-9: 1284668054.968657204  Saving: 29479465200B 3577605P 0z 0e 3577754c
B-9: 1284668054.968657204  Idling: 0B 0P 0z 0e 0c
B-9: 1284668054.968657204 NoWrite: 149rw 0inv 0rl 3577605ok
```

In the output above, `GrabRT` refers to the packet capture rate, `WriteRT` refers to the packet write rate, and the other fields provide diagnostics on the numbers of bytes (B) or packets (P) seen in the various modes (described in the next section).

Thus with this final improvement, 8 Gbps burst mode recording is possible. The write-to-disk was not pushed above an agregate 3.8 Gbps for this example.

3 Software Tasks

For design purposes, it is convient to decompose the work of the burst mode system into three essential **performance-critical** tasks

- the task ("GRAB") of capturing the data
- the task ("SAVE") of storing the data
- the task ("DUMP") of delivering the data

for which we need to be careful to achieve optimal behavior. These form the basis of the architectural discussion that follows (see Chapter 4 [architecture], page 13). There certainly are a variety of lesser tasks (commanding, reporting, monitoring, disk characterization, data checking &c).

3.1 GRAB Task

The flow of data in the GRAB task is shown in Figure 3.1.

Figure 3.1: The data in the GRAB task originates in the DDS and is captured to RAM.

Here some (VLBI) digital data source (DDS) generates ethernet or UDP packets which are picked up by the 10 Gb ethernet card (ETHC) and transferred by the CPU to RAM. In general, the same BMR might have multiple (E) ethernet cards and multiple processing cores (P). Assuming there are independent data pathways, the better performance should follow from assigning each ethernet card to its own processing core.

It is not clear that we need to consider data sources other than UDP. (The Mark5C is prepared to accept VLBI data directly encapsulated in the Layer 2 MAC packets.) The UDP packet size is limited to be less than 9000 by the ethernet layer. In the Mark5B emulation mode, the VLBI data payload is 10000 B, so this is divided into two ~5000 B packets linked by a serial number and only one of the packets has the timing information. If VDIF is used, then the packet size can be somewhat more arbitrary in size.

The most optimal choice for the GRAB process is to capture everything that meets trivial filter critera (e.g. the packet is intended for "me", or the size is "right").

3.2 SAVE Task

Once the data is captured in RAM, the BMR system should transfer it to more permanent media as show in Figure 3.2

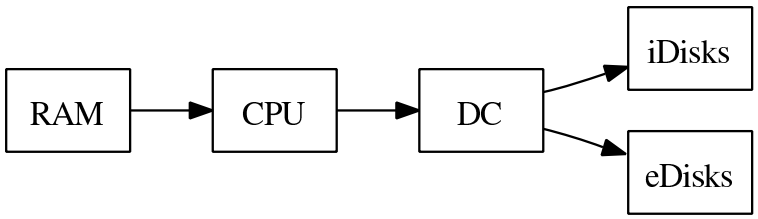

Figure 3.2: Data saved in RAM (by the GRAB process) flows through the CPU on its way to permanent storage in disk filesystems. The disks might be internal or external.

Here we are assuming either internal (iDisks) or external (eDisks) managed by some disk controller (DC). As indicated in the overview, current disks cannot write data at sufficient sustained rates for our needs. So the DC is either some explicit management of the writes across several disks, or else a software or hardware RAID controller.

The advantage to an explicit scheme is that one has complete knowledge of the disk contents. The disadvantage is that there is more code to write and test. (It is probably less efficient than a hardware RAID controller as well.)

Conversely, the hardware RAID controller may offer the best performance for redundant storage (e.g. RAID 5).

From the software perspective, there is no real distinction between internal and external disks. However, some mechanism for tracking what data is on what disks is desirable in any case.

An efficient (real-time) filesystem is desirable. (One could attempt raw data access to the disks devices, but that is probably not advisable.) There is also a trade-off to be made on speed of disk writes at the expense of a greater bookkeepping burden and a more complicated playback mechanism.

3.3 DUMP Tasks

Finally, the saved data will at some point need to be used somehow. The two obvious choices are either a Mark 5C (M5C) unit (possibly more than one) or else some other consumer, e.g. direct replay into a software correlator system. This is shown in Figure 3.3.

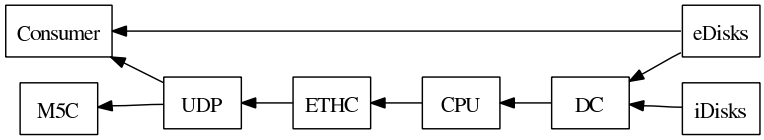

Figure 3.3: The data that was stored on disks will eventually need to pass to some "consumer", (e.g. a software correlator), or to some other (more robust) recorder (e.g. a Mark5C).

If the data is on external disks, then those disks can be directly transferred to the consumer (e.g. to a software correlator as disk mounts); however, unless the data was written in some expected format, some translation of the data may be needed.

If the data is on internal disks, then the burst mode recorder must be prepared to generate data packets (e.g. UDP) or else a higher level TCP socket connection to the consumer. The easiest data product to emit is the same set of UDP packets that it received—since this stream was presumably recordable by an M5C, it could be delivered to an M5C without a minimum of effort.

These three tasks could be operated exclusively—i.e. only one task active at a time. If performance permits the GRAB and SAVE tasks to operate in parallel, the duration of the record-only part of the duty cycle could be shortened. That is not really a goal in this work.

The "DUMP" task can be partitioned into two performance critical portions; call them LOAD and PUSH. The former (LOAD) reads the data back off the disks from their saved format and places it in RAM (see Figure 3.4).

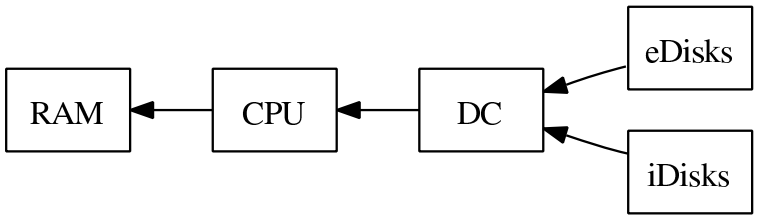

Figure 3.4: Data that was temporarily stored on disks is reloaded into RAM.

The latter (PUSH) takes data in RAM and re-packetizes it and delivers it out the ethernet interface at some measured rate (see Figure 3.5).

Figure 3.5: Data resident in RAM is, in this example, PUSHed out to a Mark5C.

This activity comes in two flavors: one for pushing the packets out as received (e.g. suitable for storage in a Mark5C), and another for delivering data via (TCP) socket for use in a software correlator.

In summary, one can thus view the data as flowing through through the system in all phases as shown in the following diagram (see Figure 3.6). The various task pieces can be assembled to form several, separate processes to carry out complete, high-level activities. For completeness, a variant of LOAD that merely simulates data (FAKE) is also shown. Such a task is useful in testing.

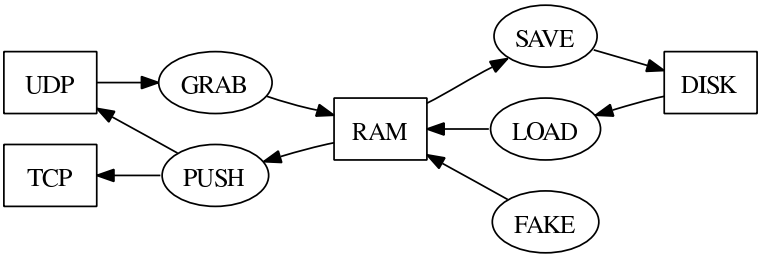

Figure 3.6: Complete flow of data through the burst mode system.

3.4 Those Other Tasks

The lesser tasks must necessarily operate in parallel, however, they should not be computationally intensive. Likewise, the operating sytem task load should be adjusted so that it is not capable of (independently) launching significant processor activity that might interfere with the GRAB task.

Some interface to these processes is needed. The Mark5C `drs` application uses a command set (VSI-S) that has been used throughout the Mark5 series of recorders–it is presumed the users would appreciate a similar command set. However, the "burst" mode of operation is not directly supported in that command set (except via explicit scheduling of the recording scans). We will find it useful to develop the capability first (with the essential commands) and adapt it to the specific needs of VSI-S, later. (I.e. the existing VSI-S command set can simply be augmented by a small number of additional commands to support burst mode recording.)

From the VSI-S command set, we will likely want to support these:

Command	**Meaning**
error?	Get error number/message (query only)
status?	Get system status (query only)
record	Turn recording on|off; assign scan label
dir_info?	Get directory information (query only)
disk_size?	Get disk sizes (query only)
rtime?	Get remaining record time on current disk set (query only)

and possibly the following:

Command	**Meaning**
DTS_id?	Get system information (query only)
OS_rev?	Get details of operating system (query only)
protect	Set/remove erase protection for active module
data_check?	Check data starting at position of start-scan pointer (query only)
scan_check?	Check data between start-scan and stop-scan pointers (query only)
scan_set	Set start-scan and stop-scan pointers
disk_model?	Get disk model numbers (query only)
disk_serial?	Get disk serial numbers (query only)
get_stats?	Get disk-performance statistics (query only)
start_stats	Start gathering disk-performance statistics.

4 Architecture

Building on the preceding discussion we now fill in some details. We will omit some "nice to have" features at this time and focus on the essential requirements. For the present discussion, we'll even omit some implementation details.

4.1 GRAB Task Requirements

The grab task obviously needs to know the where (`udp_addr` and `udp_port`) to claim the UDP packets. The various VLBI specifications suggest that it can insist on a fixed size of packet (`udp_size`) for any recording session and that they be arriving at no more than some fixed rate (e.g. `udp_rate)`, the rate in packets per second, which can be computed directly from the nominal bit rate). This size and rate could be fixed from the first packets received, but it would be better to specify this by command.

The GRAB task will be most efficient if it can be agnostic about the packet contents. However the RAM usage and packet order are possibly of interest to the eventual SAVE and DUMP tasks, so a simple (64-bit) serial number suffices to track packets. This can be either external to the packets, or possibly inserted into packets as they arrive. Eight bytes are certainly more than adequate for this. The GRAB task can then increment this number (`next_seqn`) after each packet and would alternate serial number and packet contents for a total of `mem_chunk` bytes written to RAM per packet.

The GRAB task needs to know where to put the data in RAM (`mem_*`). If there are multiple CPUs sharing RAM, the allocation of RAM buffers will need to be coordinated (trivially with `malloc`, most likely). Several pointers and sizes (`mem_*`) would be needed to manage this. Ideally the buffer size should be chosen (reduced) to hold an integral number of packets:

```
mem_chunk = udp_size + sizeof(int64_t)
mem_size  = grab_secs * udp_rate * mem_chunk
```

where `grab_secs` is the duration of the packet capture phase. In practice, adding a few extra seconds to the buffer size prevents it from wrapping during a single acquisition cycle.

The GRAB task also needs to know when to start, how long to ignore packets (after the capture phase), and how many times to repeat the cycle before stopping. (We assume here that the DDS is continuously emitting packets. Obviously the logic can be made more complex through commanding of the DDS to turn packets on/off.) We can use parameters such as `grab_start`, `grab_stop`, and `grab_period` to make this unambigous across all the BMRs involved in the observation. We assume the start time is guaranteed to be correct through a system clock synchronized via NTP. Additional parameters, `grab_idle` and `grab_cycles` are trivially computed:

```
grab_idle   = grab_period - grab_secs
grab_cycles = floor((0.5 + grab_stop - grab_start) / grab_period)
```

(where 1/2 second of rounding forces the expected number of cycles in most cases). Equivalently, `grab_cycles` or `grab_idle` could be given, with `grab_stop` and `grab_period` as the computed quantities. Once programmed, a decrementing cycle count and working UNIX time variables can be set to switch between the capture and idle modes.

Thus a structure which includes the following suffices to give the GRAB task its marching orders:

```
typedef struct grab_state {
    char        udp_addr[128];  /* binding address of UDP */
    int         udp_port;       /* the Mark5C uses 2650 */
    int         udp_size;       /* must be less than 9000 */
    long        udp_rate;       /* in packets per second */

    int64_t     next_seqn;      /* a sequence counter */
    void        *mem_start;     /* start of a buffer */
    ssize_t     mem_size;       /* of some size */
    void        *mem_write;     /* place for next packet */
    int         mem_chunk;      /* bytes written per packet */

    time_t      grab_start;     /* UNIX time to start */
    time_t      grab_stop;      /* UNIX time to stop */
    int         grab_secs;      /* grabbing duration */
    int         grab_idle;      /* secs between grabs */
    int         grab_period;    /* grab_secs+grab_idle */
    int         grab_cycles;    /* how many rep times */

    /* additional working state variables */
} GrabState;
```

Presumably some socket could be listening for the "stop" command to abort the GRAB task in the middle of things. Likewise (assuming the other parameters were specified somehow) a version of the "start" command could support the "start now" capability (i.e. the VSI-S "record=on" command).

Note that there may be insufficient RAM memory to accomplish the commanded task. In that case, it should either be objecting strenously or else be accomodating. (One could argue for different policy choices here—the initial implementation should strive to cope, I think.)

The serial number could be seeded (e.g. using the high bits) with station/band information so that all packets captured within the experiment are unique. (Exactly how many bits are available here depends on the total number of bits/packet, channels/station and number of stations.)

Additional meta-data might arrive in real time (e.g. PPS events). If these are not provided to the same UDP target then there is no (great) need for the GRAB task to capture these, rather some other task can be delegated to handle this. If such meta-data arrives via the same UDP target, then the GRAB task needs to know how to distinguish these packets (and what to do with them). This could be handled by a flag bit in the sequence number.

The RAM memory buffer could be either a circular buffer or a pair of ping-pong buffers. The former requires more address arithmetic during the capture cycle (a test for wrap after each increment), but transitions nicely into continuous operation and uses half as

much memory. If we transition to continuous operation, the latter will require an artificial division into recording sessions (for ping or pong). Unless there is a reason to do otherwise, we'll begin with a circular buffer implementation. Note that the GRAB task need not trouble itself to maintain a `mem_read` pointer—it can/should assume that the SAVE task is watching the `mem_write` pointer and coping as needed.

The GRAB process could insert fill packets if packets are found not to be arriving at the specified rate. (And then, only if needed by the "consumer".) Alternatively, this functionality can be passed along to either the SAVE or DUMP tasks, so (in the interest of reducing the load the the GRAB task) we shall assume one of them handles it.

4.2 SAVE Task Requirements

The SAVE task needs to be aware of what the GRAB task is doing so that (as a minimum) it can save data up to the current write point. (The sequence number can play a role here.)

The SAVE task needs to know whether it should be quiescent during the active capture phase of a recording session or not. (A simple binary yes/no `overlap` policy is adequate.) There may be reasons (e.g. efficiency) to write the packets to disk in groups (`pgroup`).

The SAVE task will have some initialization work to do—this is mostly identifying what disks are available, how much space is on them, and how the packet data is to be written. The XFS filesystem has support for real-time applications—we need to investigate whether that helps us here or not.

The SAVE task should probably be annotating some sort of master observational/data log information for the disk set in use. (Does the Mark5 series do this?) Potentially most of this information could be present in this file. This information would be collected to facilitate the eventual replay of the data.

```
GrabState   grab_copy;          /* a copy of grab task information */
typedef struct save_state {
    /* coordination */
    int         overlap;        /* permission to run simultaneously */
    int         ppgroup;        /* number of packets per group-write */
    /* disk information */
    int         save_type;      /* RAID or multiple files */
    long        max_cycles;     /* maximum cyles per file group */
    char        *scan_label;    /* a label for the saved files */
    int         save_size;      /* bytes written per packet */
    int         num_files;      /* number of files/disks to use */
    char        *path[BMR_MAX_FILES];
    /* logging information, &c */
} SaveState;
```

Presumably a recording session (i.e. `grab_cycles` bursts of recording or equivalently from "start" to "stop") will be saved as some number of files on disks, labelled by some scan name. If a RAID system is being used, then the obvious choices are either ONE file or `grab_cycles` files.

If a RAID system is not being used, then the SAVE task will presumably have one file per disk open at a time and farming the packets out among the (JBOD) disks in order to maximize throughput. With R disks, this is then either R files or `R * grab_cycles` files.

The simplest farming scheme is some function of the packet sequence number modulo R. However if `select` is being used, some egalitarian choice among "writable" (selected) files could be made—i.e. disks get used as they become available. This places a burden on the playback, of course. One may worry about disk failures. If they are likely, R should be prime (or some quasi randomization of the writes should be implemented).

It may be desirable in any case to limit the size of files written (in case of disk corruption); a simple way to do this would be to put at most `max_cycles` per file (or JBOD collection of files): every `max_cycles` of recording, the files would be closed, the file name index incremented, and the new files opened.

Note that from the perspective of the SAVE task, there is no real difference between having only one disk to farm with and a RAID.

If the data is to be DUMPed from the burst mode system via its own software, then it is free to use whatever disk format is convenient. (I.e. as long as it can locate and produce the data as needed, there is no fixed requirement for exactly how the data is written to disk and efficiency of write is probably the prime consideration.)

On the other hand, it may be able to edit the data prior to the disk write (so that what is on disk is written once, and thereafter only consulted read-only). Either approach involves the same software, discussed below (see Section 4.3 [filereq], page 16).

The GRAB and SAVE tasks have real-time schedules to meet that could be met within one process which `selects` across multiple file descriptors: the DDS socket, one or more disks, and perhaps a control socket. This select may have a time-out to ensure that monitoring information can be broadcast (e.g. to a central control point).

Alternatively, all monitoring could be provided only in response to a query (avoiding the time-out handling).

Alternatively, these tasks (GRAB, SAVE and monitoring) could be implemented as separate threads sharing memory.

4.3 File Format Options

If the data is to be replayed as the original packets (e.g. to a Mark5C) then any internal file format will work.

If the files are to be delivered directly to, e.g. a software correlator, then the packets will need either to be saved in some recognized file format (Mark5B, VDIF, ...) or else converted to that format at playback time. In this case it matters whether the data packets are already in some recognized format (in which case the sequence of packets without any additional header or sequence number needs to be written) or else this reformatting needs to be done.

The VDIF specification includes a 32-byte header of which 16 bytes are available for "extended" data versions (EDV). As long as the DDS isn't writing 32-byte headers in the data packets, then we can register an EDV for use with the burst mode and be assured

of enough space to save the 8-byte sequence number. The SAVE task can then write the packets in a fashion such that the 32-byte headers can be finalized in place.

To do this, the SAVE task then needs to only be able to identify the header (if present) and data array portion of the packets.

As mentioned in See Section 3.1 [grabtask], page 9, the Mark5B compatibility mode emits packets pairwise so the SAVE task needs to be able to pair them up properly. (This is not an issue if Mark5B compatibility is not required and if the DDS packets are guaranteed to be legal VDIF.)

The `RDBE` can be programmed to provide an 8-byte header (PSN) which precedes the VLBI payload (i.e. the VDIF or Mark5B packet). This header includes a 4-byte packet serial number in the lower bits. (It is zero-padded to 8 bytes for Mark5 legacy reasons.) An upper bit may be set to flag data to be considered not worth saving (invalid or otherwise). It is probably not necessary to save both serial numbers in whatever is written to file.

Note that there is some advantage to providing the PSN within the VDIF EDV if at all possible (fewer bytes to transfer).

In this mode, the packets are 5008 bytes in size when the PSN is not present, and 5016 when it is (assuming a 16-byte data header is present)—two packets are required to assemble a complete data frame (see Figure 4.1).

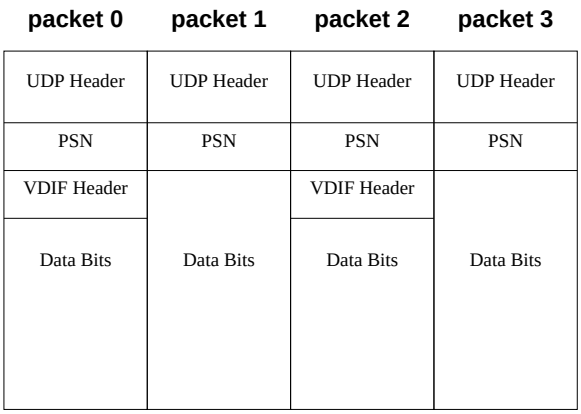

Figure 4.1: The Mark 5C packet stream consists of pairs of packets—the first part has the Mark5B/VDIF header, the second the remainder of the data. The PSN can be omitted or placed in arbitrary locations based on command.

4.4 DUMP Task Requirements

The DUMP task is really a collection of utilities that can make the data available in a variety of contexts. As indicated, if the data is written by the SAVE task in legal VDIF formatted files and the files can be mounted where they are needed, there **is** no DUMP task. Otherwise, the following software capabilities are possibly needed:

- A cleanup utility that can finalize the VDIF packet headers on the files written by the SAVE task. (Needed if the SAVE task does not write legal VDIF.)

- A playback daemon that can read the data (as is) and arrange to play the bytes back to the correlator through some socket connection—possibly in response to time-organized requests.
- A PUSH task that can replay the packets (at a lower rate than originally transmitted) to a Mark5C.

This playback daemon includes a task (LOAD, see Section 4.6 [loadreq], page 18) that will need to know how the data was organized on disk by the SAVE (see Section 4.2 [savereq], page 15) task.

The PUSH task is discussed next.

4.5 PUSH Task Requirements

The PUSH task needs to know what network protocol to be using (e.g. UDP to a Mark5C or TCP to a software correlator). It has similar network addressing needs as the GRAB (see Section 4.1 [grabreq], page 13) task. It uses the same memory buffer that the GRAB (or LOAD) task uses, and also needs access to the GRAB scheduling (at least at the initialization stage).

Thus it is governed by something like this:

```
typedef struct push_state {
    char        push_addr[128]; /* network address of UDP/TCP */
    int         push_port;      /* the Mark5C uses 2650 */
    int         push_size;      /* must be less than 9000 */
    long        push_rate;      /* packets per second */
    int         push_type;      /* UDP or TCP protocol */

    int64_t     last_seqn;      /* last seqn sent */
    void        *mem_start;     /* start of a buffer */
    ssize_t     mem_size;       /* of some size */
    void        *mem_read;      /* next packet to send */
    int         mem_chunk;      /* bytes read per packet */

    int         overlap;        /* permission to run simultaneously */
    int         source;         /* who is loading mem? GRAB or LOAD */
    int         push_period;    /* a copy of grab's value */

    /* additional working state variables */
} PushState;
```

The `mem_read` and `last_seqn` play an analogous role to GRAB's corresponding variables. If GRAB invalidates the data under its write pointer immediately following a `mem_write` advancement, PUSH can detect whether it has valid data to send without reference to GRAB's state.

4.6 LOAD Task Requirements

The LOAD task will need to know how the data was organized on disk by the SAVE (see Section 4.2 [savereq], page 15) task:

```
typedef struct load_state {
    /* disk information */
    int         load_type;      /* RAID or multiple files */
    char        *scan_label;    /* a label for the saved files */
    int         load_size;      /* bytes read per packet */
    int         num_files;      /* number of files/disks to use */
    long        file_index;     /* sequential file counter */
    char        *path[BMR_MAX_FILES];
    /* log information, &c */
} LoadState;
```

It will need further information depending on whether playback is sequential (i.e. how to locate the packets based on serial numbers and files written as recorded in the log file) or time-ordered (in which case it will need to build an index of the times ranges of files and offsets). In any case, it will need to coordinate its activities with the (PUSH) task responsible for delivering the packets.

4.7 Control Requirements

There is a need for local control (of one burst mode unit) and global control (of many such units); both needs can be met with a common "client" application that talks to the burst mode servers (e.g. on a TCP/IP socket). This client could take input from `stdin` to issue commands as required.

A GUI may be desirable.

It may be that the burst mode system also controls its corresponding DDS.

4.8 Monitor Requirements

Each burst mode system could periodically emit a packet of current status to any attached "clients", or alternatively, it could do this in response to a query request.

Autonomous monitoring of the disk system is desirable.

It may be that the burst mode system also monitors its corresponding DDS.

4.9 Initial Implementation

As an initial implementation, the GRAB/SAVE tasks could be combined into one (Linux) process (`bmr_record`) selecting on a command (TCP/IP) port, a UDP socket, and (possibly multiple) disk file descriptors. A separate client process could be used for commanding/monitoring.

For playback, a LOAD/PUSH tasks could be combined into a process (`bmr_replay`) that could read what the SAVE task has written and provide a metered replay of the data in the same order as it was captured, but limited to some lower data rate. This task could then playback data to a single Mark5C. Again, it makes sense to implement this with a client/server model.

The working part of PUSH could also be integrated with GRAB to provide a diskless GRAB/PUSH version of the burst mode (`bmr_buffer`) that would store data to the Mark5C.

For testing purposes, a FAKE module could be used as a replacement for LOAD or GRAB.

All of these tasks could use a simple, common command/response protocol (e.g. numbered commands and numbered responses) to implement the commands of Section 3.4 [misctask], page 12. If polled status were required, the client could be made responsible for emitting periodic requests.

The initial implementation could be single-threaded; however, it may ultimately be most efficient to isolate the separate tasks in individual threads of a multithreaded application.

5 Implementation Details

In this section we discuss actual implementation details.

5.1 Source Code Tree

Conventionally the code source references a tree root `$BMR_ROOT` which can be anywhere. You need not define it to carry out the following bootstrap steps—the variable is used in the examples here for clarity.

The source code is in an SVN repository available from `vault.haystack.mit.edu` (as `svn+ssh://vault/svnrepos/burst/trunk` checked out to `$BMR_ROOT/trunk`).

From withn the source tree, one runs the usual configuration scripts

```
$ aclocal
$ autoconf
$ autoheader
$ automake -a
```

to set up the source directory and create the configure script. Then the build is configured in a *separate* build directory (e.g. `$BMR_ROOT/bld`)

```
$ mkdir $BMR_ROOT/bld
$ cd $BMR_ROOT/bld
$ $BMR_ROOT/trunk/configure
$ make check install
```

This performs (currently minimal) checks and installs build products into a versioned architecture-dependent subdirectory of `$BMR_ROOT`. A script `bmr.bash` is also created to set up your environment (i.e. add the appropriate `bin` directory to your `PATH`). After sourcing the script, the tools can be run from anywhere (and the build directory is removeable).

The GNU configuration scripts allow the manufacture of a tarball distribution (`bmr-VERSION.tar.gz`, which is extracted as `./bmr-VERSION`), typically into `$BMR_ROOT`.

The tarball includes the necessary `Makefile.in` and `configure` script (so you don't have to run `aclocal`, `autoconf`, `autoheader` or `automake`). The build process is similar:

```
$ mkdir $BMR_ROOT/bld-VERSION
$ cd $BMR_ROOT/bld-VERSION
$ $BMR_ROOT/bmr-VERSION/configure
$ make check install
```

There are SVN checkouts of this software on both of the burst machines. (See `/home/gbc/HBMR` on `maxwell` and `monarth`). The GNU `auto`-software also makes tarballs which are useful for release purposes to capture specific versions.

5.2 A Buffering Server

By combining the GRAB and PUSH tasks, we obtain a server that captures packets and then immediately pushes them out again. This is the simplest such server, and it is also useful for some testing.

```
# to get command-line help
$ bmr_buffer --help
Usage: bmr_buffer [options]
where the options are:
  -v            verbose, may be repeated for more
  -c cmd        initialization command
  -f file       file of initialization commands

The -c and -f arguments allow bmr_buffer to be configured
before any clients attach.  Both are repeatable, and
will provoke a fatal exit if not properly interpreted.
```

The idea here is that some commands are required to specify the network connectivity before any clients can attach. Once the server starts, all commands can be issued from simple TCP clients. The server also converses with `stdin`/`stdout`.

For example:

```
# to get command help
$ (echo help; sleep 1; echo quit) | bmr_buffer
bmr_buffer commands:
  quit          exits the buffer server
  help          provides this help
  help:server   provides help on server configuration
  help:grab     provides help on grab configuration
  help:push     provides help on push configuration

  cfg:<cmd>     caches a (re)configuration command
                <cmd> is (server|grab|push):<cmd>
  exec:<file>   load and cache these commands from some file
  dump:<file>   saves working configuration to file
  reset         applies the caches commands

  show          shows internal state information
  info          shows state and additional data
  counters      shows various packet/byte counters
  rates         shows grab / push rates

# to get a loadable inititalization file test.bmr
$ (echo dump:test.bmr ; sleep 1; echo quit) | bmr_buffer
```

Notice that in addition to the `grab` and `push` configuration commands, there is also a `server` portion that binds them together.

Sample invocation examples were shown in See Section 2.2 [bench1], page 6.

5.3 A Burst Recorder Server

By combining the GRAB and SAVE tasks, we obtain a server that captures packets and then, on the off cycle, writes the packets to disk.

```
# to get command-line help
$ bmr_record --help
Usage: bmr_record [options]
where the options are:
  -v            verbose, may be repeated for more
  -c cmd        initialization command
  -f file       file of initialization commands

The -c and -f arguments allow bmr_record to be configured
before any clients attach.  Both are repeatable, and
will provoke a fatal exit if not properly interpreted.
```

As with `bmr_buffer`, some commands are required to specify the network connectivity before any clients can attach. Once the server starts, all commands can be issued from simple TCP clients. The server also converses with `stdin`/`stdout`.

For example:

```
# to get command help
$ (echo help; sleep 1; echo quit) | bmr_record
bmr_record commands:
  quit          exits the record server
  help          provides this help
  help:server   provides help on server configuration
  help:grab     provides help on grab configuration
  help:save     provides help on save configuration

  cfg:<cmd>     caches a (re)configuration command
                <cmd> is (server|grab|save):<cmd>
  exec:<file>   load and cache these commands from some file
  dump:<file>   saves working configuration to file
  reset         applies the caches commands

  show          shows internal state information
  info          shows state and additional data
  counters      shows various packet/byte counters
  rates         shows grab / save rates

# to get a loadable inititalization file test.bmr
$ (echo dump:test.bmr ; sleep 1; echo quit) | bmr_record
```

Notice that in addition to the `grab` and `save` configuration commands, there is also a `server` portion that binds them together.

For testing purposes, it has been convenient to drive `bmr_record` using a simple shell script. (In the course of testing, there are many options that get tried, so the script ends up being rather more complicated than would be used for a simple observing run.) The most recent such scripts (found in the `testcode` area of the source) are `record-X.sh` and `record-vdif.sh`. The latter was used with the `Astro8Gbps` `RDBE-S` image, and was typically invoked as follows:

```
( pushd /home/gbc/HBMR/Test ;
  for S in CC DD EE FF ; do record-vdif.sh $S eval 5 4 20 & done )
```

In this incantation, four `bmr_record` processes were launched—one for each of 4 threads. For testing reasons, the threads were considered to be data from 4 separate stations (CC, DD, EE and FF). The numbers select 5 cycles of 4-second captures with 20-second periods. (Obviously the numbers can be varied.) The default "start" time is 5 seconds from "now". The script automatically numbers the files so that previous results are not overwritten, and it polls the servers periodically for performance statistics. (This is a testing mode.)

In addition, it is useful to make various checks of the recorded data—a testing code `fchk_test` was written to test various aspects of the various data packet types recorded:

```
$ fchk_test --help
Usage: fchk_test [options] action
where the options are:
  -v            verbose, may be repeated for more
  -c <string>   configure the checker

will check files and report on what it finds.
```

The various configuration commands (all are intended for checking the data or the burst mode software, and are rather *ad hoc*):

```
$ fchk_test -c help
Available configuration options:
  f=<int><string>  format of the data frames
                   (int = 0, 1, ... for raw, pkt, ...)
                   (string = m5b or vdx)
  n=<float>        number of frames to read (0 = all)
  p=<int>          level of packet display (0, 1, ...)
  s=<int>          size of data frame
  t=<int>          0 = use thread ID, 1 = usechannel id
  x=<int>          mask of channel/thread to skip (1,2,4,8)
  h=<int>          size of skip to notice (0 == any)
  r=<int>          frames per second (31250)
  g=<int>,<int>    frames to process,skip for bit stats (0,0)
```

so consulting the source code is presently required to make sense of the output. (This is a work in progress....)

5.4 Code Sources

There are quite a number of header and code files for such a simple server. In this section we mention what is to be found in each file. Note that the LOAD task is at present undeveloped, so those files are mostly full of boilerplate, starter code.

The headers are

bmr_common.h	Includes common things needed everywhere.
bmr_util.h	Declarations of some simple utilities.
bmr_server.h	General server declarations.

buffer_server.h	Buffer-specific declarations.
grab_task.h	Declarations for the GRAB task.
push_task.h	Declarations for the PUSH task.
save_task.h	Declarations for the SAVE task.
load_task.h	Declarations for the LOAD task.

The general server code modules are:

bmr_util.c	Implements some simple utilities.
bmr_cache.c	Caches command-line commands for eventual setup.
bmr_parser.c	Parsers general server commands.
bmr_server.c	Implements the general client-server code.

The `bmr_buffer`-specific code modules are:

buffer_main.c	The main() boilerplate.
buffer_server.c	Implements the GRAB+PUSH server.
buffer_parser.c	Interprets commands.
buffer_info.c	Generates printouts of internal state.

Finally, each of the tasks (`GRAB`, `PUSH`, `SAVE` ...) includes three modules:

TASK_task.c	The core implementation of the task.
TASK_parser.c	Routines for parsing its commands.
TASK_info.c	Routines for describing its internal state.

where `TASK` refers to `grab`, `push`, `save` .... Not all modules have been developed or tested in all possible cases.

5.5 Debian System Modifications

The newest burst mode system began life as a "Lenny" distribution, then was reinstalled as a "Sid" (unstable) distribution, and finally as a "Squeeze" (6.0) Debian distribution. Starting from the vanilla distribution, the following changes were required:

- local network changes (the domain and name-server information in `/etc/resolv.conf`, adding local ntp server(s) in `/etc/ntp.conf`)
- 10 GbE network changes (`/etc/network/interfaces`)
- packet parameter tuning (`/etc/sysctl.d/bmr.conf`)
- CPU frequency tuning (it is disabled by default, so it must be enabled; in addition the governor must be changed from "ondemand" to "performance" in `/etc/default/cpufrequtils`; there is a BIOS option that must also be activated to enable this)

At present the network ARP/IP addressing scheme is completely manual. Aside from assignment of the local network interface to something sensible, the `/etc/network/interfaces` file is typically modified to specify the network addressing of the 10 GbE ports. Note that the UDEV device management rules sometimes renumber the network interfaces, so a file such as `70-persistent-net.rules` placed in `/etc/udev/rules.d` is typically desirable for consistent performance across reboots. This file has entries of the form:

```
# PCI device 0x8086:0x10dd (ixgbe)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
```

```
ATTR{address}=="00:25:90:0b:c6:7e", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth2"
```

(all one line, however) to force the association of the port with MAC address `00:25:90:0b:c6:7e` to device `eth2`. Then in the `/etc/network/interfaces` file an entry such as:

```
allow-hotplug eth2
iface eth2 inet static
  address 192.168.5.102
  netmask 255.255.255.248
  network 192.168.5.96
  broadcast 192.168.5.103
  mtu 9000
```

provides the information for one of the 10 GbE CX4 cables so that this interface can be easily managed with the `ifup` and `ifdown` command scripts.

The packet tuning is of the "net.core" `sysctl` parameters:

```
net.core.rmem_max = 16777216
net.core.rmem_default = 8388608
net.core.netdev_max_backlog = 1000
```

Finally, a rather long list of packages were installed as approprate to a development system. This system was also used as part of a `DiFX` software correlation cluster (more convenient for analyzing the data), and also with HOPS, and also for `RDBE` development; so a number of additional modules are needed for those.

The complete list of `apt` commands applied (more or less in this order) was:

```
apt-get install apt-file
apt-file update

# general system tools:
apt-get install sysfsutils mdadm xfsprogs xfsdump sysstat tcpdump

apt-get install gcc make
apt-get install autoconf automake libtool libtool-doc
apt-get install rsync subversion subversion-tools

# things needed for DiFX:
apt-get install g++
apt-get install g++-multilib
apt-get install libstdc++6-4.4-dbg libstdc++6-4.4-doc
apt-get install lib32stdc++6-4.4-dbg lib32mudflap0

apt-get install gfortran
apt-get install gfortran-multilib

apt-get install flex bison gdb
```

```
apt-get install valgrind
apt-get install vim vim-scripts vim-doc

apt-get install openmpi-bin openmpi-common \
                openmpi-dbg openmpi-dev openmpi-libs0 \
openmpi-checkpoint openmpi-mpidoc
apt-get install libexpat1-dev fftw3 fftw3-dev
apt-get install sysv-rc-conf chkconfig
apt-get install kernel-package ia32-libs ia32-libs-dev

# required for HOPS:
apt-get install pgplot5
apt-get install libx11-dev libpng12-dev
apt-get install ghostscript-x ghostscript-doc
apt-get install psutils

# required for RDBE development:
apt-get install ckermit gkermit
apt-get install gnuplot-x11 python-gnuplot
apt-get install wireshark wireshark-common wireshark-dbg wireshark-dev
aptitude install ethtool
apt-get install libgsl0-dev libgsl0-dbg
apt-get install python-tk python-pmw python-matplotlib

# for disk performance testing:
aptitude install fio

apt-get update
```

The older system has a similar set of packages. However, the kernel on that machine was built (as an optimization) without a number of things (e.g. X11) so it is not as easy to use for some things. Note also that the `DiFX` operation is presently in 32-bit mode on these machines (requiring the 32-bit emuation packages that are not otherwise needed).

No other tuning of the operating system is required for testing or short scans. However, the creation of **hugepages** and their use with the application would probably be a logical next step to provide more robust use of the full installed memory of the system. The application can then use these pages if it is given appropriate permissions, and uses `mmap` for access to the memory rather than `malloc`. This has not yet been done.

5.6 Setting up Software RAID

From the viewpoint of disks misbehaving, it is best to write the data out to independent filesystems (so that if one disk goes bad, the others are not affected). For testing in environments where the disks can be expected to behave (i.e. well below 10000 feet) a sofware raid system can be fast and efficient. We have used the XFS filesystem—it is relatively efficient for our needs.

With multiple disks a larger "volume" for the filesystem can be assembled with the `mdadm` command. (You may need to install the `mdadm` package to get the necessary tools.) The `cfdisk` is preferred (these days) for partitioning/labelling the bare disks.

The raid is created/managed with `mdadm`, e.g. with:

```
mdadm --create /dev/md0 --level=0 --raid-devices=8 /dev/sd[b-i]1
```

Thereafter, the kernel will automatically start the array at boot time. You can stop/start/study it with

```
mdadm -S /dev/md0
mdadm -A /dev/md0
mdadm --query --detail /dev/md0
mdadm --detail --scan
```

and so forth—see the man page for further details. The array state is typically captured in `/etc/mdadm/mdadm.conf` using information gathered in a "scan". The system log file `/var/log/messages` has additional detail on what is involved. The `mdadm` mechanisms do appear to be able to identify disks so that they may be removed and re-inserted in arbitrary locations without destroying the data.

Once the raid is created, an XFS filesystem can be built with:

```
mkfs.xfs -f /dev/md0
mount -t xfs /dev/md0 /mnt/raid
```

and mounted in the usual way. It can of course be added to `/etc/fstab` for mounting at boot time.

6 Some RDBE-S Usage Notes

In this section are captured a few usage notes for the RDBE-S environment. There are often many ways to accomplish the same thing (e.g. loading a personality) and the tools are still under development, so this chapter reflects what was done, rather than what **should** have been done.

There may be better/more complete documentation for things-RDBE-S elsewhere.

6.1 General RDBE-S Considerations

The **R**oach Digital Back End (RDBE) connects to the analog RF feeds (IFs) from the receiver/downconverter, samples and packetizes the signal, and presents the digital data as UDP packets on one or more 10 GbE interfaces. While the term RDBE refers to a specific hardware configuration (as defined by NRAO and Haystack), more generally developmental RDBEs differ both in terms of the hardware options (e.g. number of ADC boards, type of synthesizer, &c) as well as the "personality" loaded into the FPGA that does the post-sampling digital work. The work described in this memo is focussed on one particular "personality" (Astro8Gbps), viz a version that works with an RDBE-S (a particular configuration of hardware not to be confused with an RDBE) equipped with 4 512 MHz inputs, sampling driven by the Haystack 1024 MHz synthesizer to produce 4 "threads" of data on two CX4 10 GbE cables (RDBE-S). More about all that later (see Section 6.3 [astro_rdbe], page 30). Hereafter we'll use the term RDBE-S to refer to the hardward with this personality.

The RDBE-S boots with support from a network filesystem or from a local USB drive—the network IP address is often assigned by DHCP. If the system does not come up right away, you may need to connect to it on a serial line via Kermit (see Section 6.2 [kermit_rdbe], page 30).

The "personality" is stored as a binary file, which should be stored in the subdirectory personalities in the roach user directory. In addition, a configuration file (with a similar name) can be stored in the conf subdirectory. This file can be executed at initialization time. After reboot, or if it dies, the server for the RDBE-S needs to be launched, typically with:

```
bin/rdbe_server 5000 6
```

specifying the port (and number of allowed clients) at which the server will listen for commands.

There are some diagnostics available over the normal network (1 GbE) interface. The script bpplotter_rdbe.py provides diagnositics on the raw sampling.

When the server is running, one may connect to it with any of the various clients to carry on a text-based conversation. For example:

```
$ ./rdbe_client.py -h 192.52.61.117
Host: 192.52.61.117 port: 5000
>> dbe_personality?
<<  !dbe_personality?0:pfba:RoachAstro8GbsJan26.bin;

>> dbe_execute=init;
```

```
<<  !dbe_execute=0;
```

should connect to the RDBE-S at 192.52.61.117 and have it load and initialize the named personality. The dbe_execute=init command will only work properly if valid configuration file exists for the personality loaded. Alternatively the commands to configure the RDBE-S personality can be loaded one at a time via one or more scripts.

6.2 Access to RDBE-S via Kermit

If you need to talk to the RDBE-S directly over the serial port, kermit is useful. Typically you'll have something like the following in the initialization file

```
$ cat /home/$user/kermrc
set modem type none
set line /dev/ttyUSB0
set speed 115200
set parity none
set stop-bits 1
set carrier-watch off
set handshake none
set flow-control none
robust
```

so that kermit /home/$user/kermrc will just connect you to a command line prompt. From there you can figure out what's wrong and fix it. If it's a simple DHCP issue, ifconfig can tell you what address was assigned, and you can use that to connect normally.

6.3 Astro RDBE-S Considerations

The Astro8Gbps personality went through a number of revisions—the version current at this time this memo was written is RoachAstro8GbpsJan26. It presents the samples from the four IFs as four VDIF "threads" of UDP packets presented to two of the 10 GbE interfaces. There are commands to support some customization of the header fields for the VDIF packets, and also a means of setting the seconds-of-epoch counter.

A number of "future" commands and capabilities for this personality have yet to be implemented. (I.e. we focussed on the minimum necessary for this testing.)

We found it convenient (and less confusing) to properly place each cable on its own (3-bit) subnet to avoid any confusion in the IP address space. One can then easily verify the network traffic with either tcpdump (command-line) or wireshark (graphical):

```
$ tcpdump -c 1 -x -s 84 -i eth4
...
09:58:24.813516 IP 192.168.5.101.65535 > 192.168.5.102.4203: UDP, length 8224
        0x0000:  4500 203c 0000 4000 ff11 cf94 c0a8 0565
        0x0010:  c0a8 0566 ffff 106b 2028 0000 1e00 0000
        0x0020:  be54 0014 0404 0000 4545 0204 0000 0000
        0x0030:  0000 0000 24df 7110 ecde adab
...
09:58:31.169577 IP 192.168.5.101.65535 > 192.168.5.102.4201: UDP, length 8224
        0x0000:  4500 203c 0000 4000 ff11 cf94 c0a8 0565
        0x0010:  c0a8 0566 ffff 1069 2028 0000 2500 0000
        0x0020:  6005 0014 0404 0000 4343 0004 0000 0000
```

```
        0x0030:  0000 0000 44e6 7410 eade adab
...
$ tcpdump -c 1 -x -s 84 -i eth5
...
09:58:35.609527 IP 192.168.5.109.65535 > 192.168.5.110.4202: UDP, length 8224
        0x0000:  4500 203c 0000 4000 ff11 cf84 c0a8 056d
        0x0010:  c0a8 056e ffff 106a 2028 0000 2900 0000
        0x0020:  8f3a 0014 0404 0000 4444 0104 0000 0000
        0x0030:  0000 0000 bb03 7710 ebde adab
...
09:58:37.641568 IP 192.168.5.109.65535 > 192.168.5.110.4204: UDP, length 8224
        0x0000:  4500 203c 0000 4000 ff11 cf84 c0a8 056d
        0x0010:  c0a8 056e ffff 106c 2028 0000 2b00 0000
        0x0020:  363e 0014 0404 0000 4646 0304 0000 0000
        0x0030:  0000 0000 86fb 7710 edde adab
...
```

which allows the verification that the VDIF headers are also ok. So for example, the hexdump from `tcpdump` for the first packet (noting that `tcpdump` outputs in network order, not host or VDIF order, which is littlendian for both in this case):

```
1e00 0000   word0  ok (secs from ref epoch)
be54 0014   word1  0000 ref epoch is 0014 (early 2010)
0404 0000   word2  ok (data len/8)
4646 0304   word3  ok (station FF, thread 3, bpsm1)
0000 0000   word4  ok (unused)
0000 0000   word5  ok (unused)
24df 7110   word6  ok (psn)
ecde adab   word7  ok (sanity clause: abad dee[abcd])
```

may be compared with the VDIF header as shown in Figure 6.1.

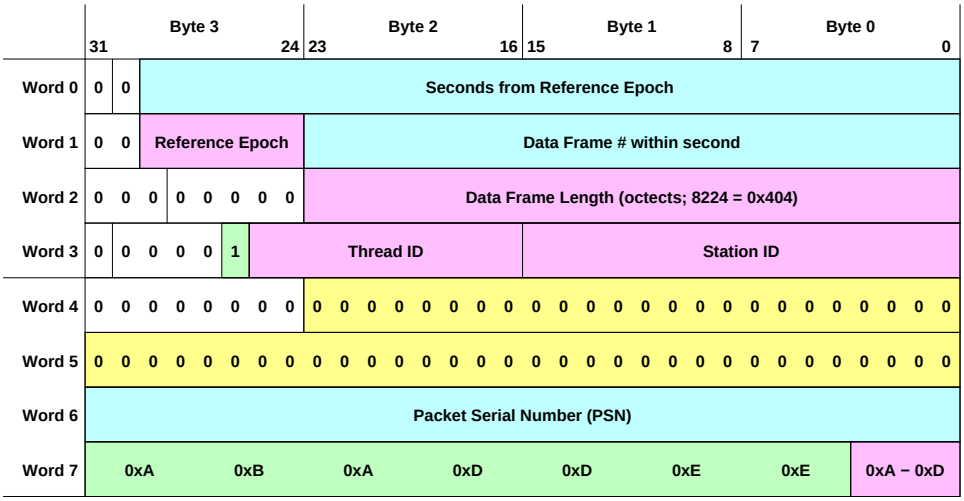

Figure 6.1: VDIF Header as used by the `Astro8Gbps` image. The red-shaded areas are configurable, the blue-shaded areas are dynamic (i.e. changing with each packet), and the green areas are fixed by the personality.

The current burst mode software writes the bits of its internal sequence number into the unused bits of the extended header (3 lowest bytes of word 4 and all of word 5) for debugging purposes, shaded yellow in Figure 6.1. In this example, the 4 cables are on subnets of `192.168.5`, and the 4 threads were arbitrarily also labelled as stations CC, DD, EE and FF. The PSN is embedded in unused words of the Extended header (word6) and the last word is a magic number (`0xabaddeed`) that helps identify the channels and endianism.

There are several paths to setting up the `Astro8Gbps` image. In development a number of scripts were used to program the `Astro8Gbps` image using direct writes to the FPGA registers. (E.g. `BRAINS.sh` to shoot it in the head, `STATUS.sh` to get status—these scripts reside on `Nugget`, the development host.)

For testing purposes, it was (ultimately) useful to modify the client script to take input from a configuration generator (`make-rdbe-conf.sh`), and to pass the result to a parser to calculate the bit states. Then a script could be launched to capture sample data which could be post-processed to check the bit states. Finally, with suitable "noise" data, a zero-baseline test could be processed with `DiFX` (see Chapter 7 [difxzbt], page 37).

6.4 An Example Test Session

A complete (zero-baseline) test session can thus be launched with:

```
$ ssh maxwell
...

# not essential, but the configuration is written out here
cd ~/HBMR/Test

# launch the raw sample statistics analyzer
bpplotter_rdbe.py &

# program the RDBE-S with this personality:
make-rdbe-conf.sh ccddeeff temp RoachAstro8GbsJan26.bin |\
rdbe_setup.py -h 192.52.61.117 -v 1 | bitstatus.py

# launch the recording script 4 times; the environment
# variables provide hints to the script
#  export SAVE_VERS=$NN or unset SAVE_VERS
export SAVE_PATH=/mnt/raid/vdif3
export LABEL=vv
for S in cc dd ee ff ; do record-vdif.sh $S eval 1 3 6 & done
bstats-vdif.sh {NN..MM}
```

As mentioned previously, `bpplotter_rdbe.py` is a tool to examine the "raw capture" data.

The `make-rdbe-conf.sh` is a script to manufacture the "init" configuration commands. The existing version makes an attempt to set the epoch using a helper program `bmr_time` to convert from the local (NTP-sync'd) UNIX clock to the VDIF six-month epoch and seconds of epoch time. For convenience, this time is represented as `Epoch@seconds.subseconds` in printouts.

This script ends by launching commands to report the bit states captured in the gain/threshold calculation—the script `bitstatus.py` processes the raw hex output in a relatively trivial to get more useful quantities to look at. The environment variables (`SAVE_PATH` and `LABEL`) are used to pass file-naming hints to the record-vdif.sh.

The `bstats-vdif.sh` post-processes the data files captured and provides some input to streamline the `DiFX` analysis. In particular, these test were done without setting the `RDBE-S` time to the correct absolute time, so the time of the scan start needs to be found. So, at the time of the reset, the `RDBE-S` thinks it is at the beginning of the epoch.

So, for example, on "Run 11" the scans started 86 seconds into epoch 22 (that would be the first half of 2011—because that's what it was told, and the reset was 86 seconds prior to this example being captured):

```
# this is the output from bitstatus.py:
ifplus    ifminus  counts     gain
+0.0773926 -0.0773926  697413  1.03643
+0.0778885 -0.0778885  705020  1.02140
+0.0701981 -0.0701981  692614  1.04591
+0.0698853 -0.0698853  726007  0.97993

# this is the output from bstats-vdif.sh 11
Run 11 (bit statistics on 1000 frames)
22@86.535168 BS[0] 0.181 0.325 0.340 0.153 (66.5% 32.768 Ms) [697475/1Ms]
22@86.535104 BS[1] 0.209 0.346 0.330 0.115 (67.5% 32.768 Ms) [708204/1Ms]
22@86.535168 BS[2] 0.182 0.319 0.342 0.157 (66.2% 32.768 Ms) [693650/1Ms]
22@86.535168 BS[3] 0.167 0.348 0.353 0.132 (70.2% 32.768 Ms) [735627/1Ms]
vv*11) NN=11
     mjdStart=55562.0010015643
      mjdStop=55562.0010478605
    scanStart=2011y001d00h01m26s
 ;;
```

The "counts" column from `bitstatus.py` and the final column of the `bstats-vdif.sh` script output show that both the ADC gain programming and the resultant data are consistent. (E.g. thread 0 has 66.5% of the data in the low states 697475 of 1024^2 samples).

The ADC logic is attempting to place the same number of samples into the low states—that it is not working as well as expected for the examples shown in this memo is a known bug that is being resolved.

The companion to the output from the `bstats-vdif.sh` script is a `DiFX` launcher script, `./vdifnoise.sh`, which takes input from a file (`./vdifn_par.sh`) providing the start/stop times (via a huge `case` statement) for the individual runs. Then

```
$ ssh smm-pc
cd ~/DiFX/Test
DIFX_GRINDERS='smm-pc maxwell monarth' DIFX_NPROCESS=11 NCHAN=512 \
    ./vdifnoise.sh vv11
```

is all that is required to launch a `DiFX` correlation/analysis run to process "Run 11". The `DiFX` results are discussed later (see Chapter 7 [difxzbt], page 37).

It should be noted that the burst recorder is running "open-loop". The `RDBE-S` is configured to **always** generate packets, so the GRAB/SAVE logic is driven by the host computer clock (NTP synchronized).

The RDBE-S itself (when deployed in the field) should have valid PPS and 10 MHz inputs so that the data is properly time-tagged. (Well, in this case it's off by some large number of seconds-of-epoch since we haven't told the RDBE-S what time it is.) However, in testing, the PPS is provided by a noise generator with an arbitrary offset from proper UTC ticks.

So, the start of data and end of data for each cycle of the burst mode scan are not precisely correlated with the 1-second boundary on the host computer, and in any case are only near the PPS boundary provided: they are "sloppy" by several ms due to packets already queued for reception at the time the GRAB cycle starts and the lack of precision with which the GRAB cycles are launched.

The fchk_test has a number of capabilities for checking the data. The incantation (for a later "Run 26"):

```
for f in /mnt/raid/vdif3/vdif_??_26.vdif
do fchk_test -cf=4vdx -cs=8224 -cg=1000 $f ; done
```

produces a listing of scans in each of the 4 files:

```
Scan-0 0 22@15230.622176 to 22@15233.619168 for 2.996992 s, 93658 fr File 0
Scan-0 1 22@15243.610912 to 22@15246.606368 for 2.995456 s, 93609 fr File 0
Scan-0 2 22@15256.598080 to 22@15259.593568 for 2.995488 s, 93608 fr File 0
Scan-0 3 22@15269.606912 to 22@15272.580800 for 2.973888 s, 92935 fr File 0
Scan-0 4 22@15282.572576 to 22@15285.567936 for 2.995360 s, 93606 fr File 0
Scan-0 5 22@15295.572832 to 22@15298.555104 for 2.982272 s, 93195 fr File 0
Scan-1 0 22@15230.622144 to 22@15233.619072 for 2.996928 s, 93656 fr File 1
Scan-1 1 22@15243.610944 to 22@15246.606336 for 2.995392 s, 93353 fr File 1
Scan-1 2 22@15256.598080 to 22@15259.593536 for 2.995456 s, 93607 fr File 1
Scan-1 3 22@15269.586336 to 22@15272.580704 for 2.994368 s, 93575 fr File 1
Scan-1 4 22@15282.576480 to 22@15285.567904 for 2.991424 s, 93482 fr File 1
Scan-1 5 22@15295.567808 to 22@15298.555136 for 2.987328 s, 93355 fr File 1
Scan-2 0 22@15230.622112 to 22@15233.619136 for 2.997024 s, 93659 fr File 2
Scan-2 1 22@15243.616448 to 22@15246.606336 for 2.989888 s, 93435 fr File 2
Scan-2 2 22@15256.616448 to 22@15259.593568 for 2.977120 s, 93035 fr File 2
Scan-2 3 22@15269.585248 to 22@15272.580768 for 2.995520 s, 93610 fr File 2
Scan-2 4 22@15282.584512 to 22@15285.567904 for 2.983392 s, 93230 fr File 2
Scan-2 5 22@15295.559712 to 22@15298.555168 for 2.995456 s, 93608 fr File 2
Scan-3 0 22@15230.622080 to 22@15233.619168 for 2.997088 s, 93661 fr File 3
Scan-3 1 22@15243.623296 to 22@15246.606304 for 2.983008 s, 93220 fr File 3
Scan-3 2 22@15256.616544 to 22@15259.593568 for 2.977024 s, 93033 fr File 3
Scan-3 3 22@15269.602304 to 22@15272.580704 for 2.978400 s, 93076 fr File 3
Scan-3 4 22@15282.590208 to 22@15285.567968 for 2.977760 s, 93056 fr File 3
Scan-3 5 22@15295.559648 to 22@15298.555168 for 2.995520 s, 93609 fr File 3
```

So in this case, 6 cycles of 3-second scans is falling short by a dozen ms. DiFX does not care about this. (Should anyone else care, this can be solved by adding the record on/off capability back into the Astro8Gbps personality.)

6.5 Sample BMR Commands for Astro Test

For completeness, it's probably worth reviewing the BMR commands used in "Run 11". The record-vdif.sh script launches 4 copies of bmr_record (one to listen for each thread). Each is configured, and a copy of the configuration is dumped to a file. Here is the content of that file, with comments inserted to explain what the commands mean (in the current version of the code—remember, not all the functionality has been implemented).

```
##
```

```
## GRAB commands
##
grab:addr=192.168.5.102       # this is the IP address for packets
grab:port=4201                # this is the UDP Port for packets
grab:size=8224                # i.e. 32 header + 8192 data bytes
grab:rate=30398               # limit: (approx) packets per second
grab:mbps=1999.95             # limit: (approx) MB per second
grab:gbps=1.99995             # limit: (approx) GB per second
grab:type=1                   # mem_type: see bmr_files.h
grab:start=1297373092         # unix clock start time: 2011y041d21h24m52.0000s
grab:stop=1297373267          # unix clock stop time:  2011y041d21h27m47.0000s
grab:secs=8                   # capture duration (secs)
grab:flush=1                  # 1-sec of packet flushing prior to grab
grab:period=35                # cycle duration (secs)
grab:tognet=1                 # activates an optimization on sockets
##
## SAVE commands
##
save:save_type=1              # mem_type: see bmr_files.h
save:expr_name=vdif           # name of experiment
save:stn_name=cc              # name of station
save:scan_name=11             # name of scan
save:scan_suffix=             # optional suffix to scan lable
save:scan_label=vdif_cc_11    # complete scan (file) label
save:num_files=1              # number of files to use
save:path[0]=/mnt/raid/vdif2  # path to the files
save:rate=74880               # limit: (approx) packets per second
save:mbss=4926.5              # limit: (approx) MB per second
save:gbss=4.9265              # limit: (approx) GB per second
save:ppgroup=128              # packets per group write (unused)
save:file_fmt=4               # output file format, see bmr_files.h
##
## SRVR commands
##
server:std_input=1            # allow stdin commanding
server:tcp_port=4401          # listen port for clients
server:max_clients=10         # 11's a crowd
server:sel_to=0.500000        # polling timeout on commands
server:lfile=(null)           # log file
server:dfile=(null)           # debugging file
##
## eof
##
```

As indicated, some of these commands exist for future developments, or to support (earlier) developmental options. In particular, the original implementation supported Mark5B emulation, but that mode hasn't been used in a while, so not all the pathways are (correctly) supported for that data....

As indicated previously (see Section 5.3 [bmr_record], page 22), limited help on the various tasks is available. For the GRAB task:

```
$ bmr_record -c grab:help
The GRAB configuration commands are:
        grab:addr=<addr>  # specify receiving IP addr or ETH device
        grab:port=<int>   # specify the UDP port receiving packets
        grab:size=<int>   # specify size of all UDP packets
        grab:rate=<int>   # specify packet rate (packets per sec)
```

```
        grab:mbps=<float> # specify packet rate (Mbps)
        grab:gbps=<float> # specify packet rate (Gbps)
        grab:type=<int>   # specify packet memory model
        grab:start=<time> # specify starting time
        grab:stop=<time>  # specify stopping time
        grab:secs=<int>   # specify capturing duration (secs)
        grab:flush=<int>  # specify packet flushing (secs)
        grab:period=<int> # specify cycle duration (secs)
        grab:tognet=<int> # specify data socket toggling

        grab:help         # provides this message
        grab:show         # displays GRAB state
        grab:default      # resets to a default state
        grab:apply        # applies all commands
        grab:quit         # provokes a fatal error
```

and for the SAVE task:

```
$ bmr_record -c save:help
The SAVE configuration commands are:
        save:save_type=<int>     # (0|1|2) for (Files|Raid0|Raid5)
        save:scan_label=<string> # name of the scan for files
        save:expr_name=<string>  # name of the experiment
        save:stn_name=<string>   # name of the station (files/vdif)
        save:scan_name=<string>  # name of the scan (for files)
        save:scan_suffix=<string># optional suffix to scan label
        save:num_files=<int>     # number of output files to use
        save:path[N]=<string>    # path to the Nth directory
        save:rate=<long>         # specify packet rate (pps)
        save:mbps=<float>        # specify packet rate (Mbps)
        save:gbps=<float>        # specify packet rate (Gbps)
        save:ppgroup=<int>       # number of packets to write together
        save:file_fmt=<int>      # 0: raw packet with seq nums
                                 # 1: pkt packet only is saved
                                 # 2: vdf version 0 VDIF
                                 # 3: vdx extended VDIF

        save:help                # provides this message
        save:show                # displays SAVE state
        save:default             # resets to a default state
        save:apply               # applies all commands
        save:quit                # provokes a fatal error
```

and finally for the server portion (should there be a need to configure it):

```
$ bmr_record -c server:help
The SRVR configuration commands are:
        server:std_input=<int>   # nonzero to connect stdio/stdout
        server:tcp_port=<int>    # TCP port server listens on (2651)
        server:max_clients=<int> # max number of TCP/UDP fd's
        server:sel_to=<float>    # max commanding latency (<1.0s)
        server:lfile=<file>      # file for general status logging
        server:dfile=<file>      # file for server dispatch debugging

        server:help              # provides this message
        server:show              # displays SRVR state
        server:default           # resets to a default state
        server:apply             # applies all commands
        server:quit              # provokes a fatal error
```

7 Four-station Zero-Baseline Test with DiFX

This section describes some of the testing with the Astro8Gbps image carried out from Nov 2010 through Feb 2011. This "personality" was spawned from the geodesy development thread and eventually matured into something rather different. The two most significant differences are that the data packets are in the VDIF format and that there is no internal filterbank to subdivide the 512 MHz IFs into multiple channels. On the other hand, this personality is able to process 4 such IFs.

A number of minor issues were found and fixed in the development process; we won't dwell on those here. Rather we'll focus on results obtained with the "Jan26" revision of the personality. Aside a few minor commanding issues (i.e. the convenient "arming" of the RDBE-S with the UTC time), there are at present two significant known issues:

- There is an unpredictable sample misalignement (typically 2 ns) between the two ADC cards
- The setting of the gain/thresholds for the proper bit statistics is not as accurate as it should be.

These effects result from timing issues at the ADC/FPGA interface, and fixes are under discussion and are likely to be resolved in the next revision. Here we will focus on the results obtained (Feb 2011) with the "Jan26" revision.

7.1 Test Hardware

A single noise source (see Figure 7.1) was available for these tests. In addition to the noise output, the unit also provides a 5 MHz clock and a 1 PPS tick. Switches on the front panel can attenuate the noise output from a nominal level down by an addition 31.5 dB.

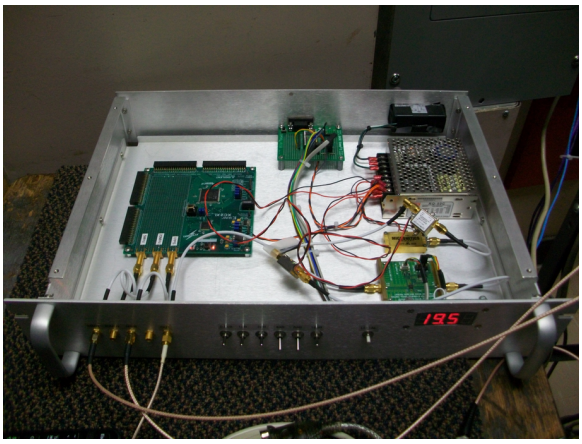

Figure 7.1: The noise source used in these tests provides suitibly amplified noise, 5 MHz and 1 PPS signals on 3 cables.

With only one source, and only one RDBE-S, we decided to treat the four 512 MHz threads as if they were separate "stations". Thus this was in effect a zero baseline test with 6 baselines. Since this involves splitting the noise signal four ways (6 dB loss), an amplifier was added to the noise source.

The four cables from the noise source were connected to the RDBE-S as shown in Figure See Figure 7.2. The PPS and 5 MHz lines were directly connected to the synthesizer (nearest board on the right). The noise signal was connected to the middle, source port (3) of the Minicircuits ZN4PD1-50+ splitter. The four output ports (1, 2, 4 and 5) were then connected to the four IF inputs on the 2 ADC boards. The details of the cable arrangements are discussed later (see Section 7.5 [zbt_cable_delays], page 44).

For some of the testing, a signal generator (not shown) was coupled (with a Minicircuits ZFSC-2-2 coupler) into the noise source to provide a broad-band signal with a single tone.

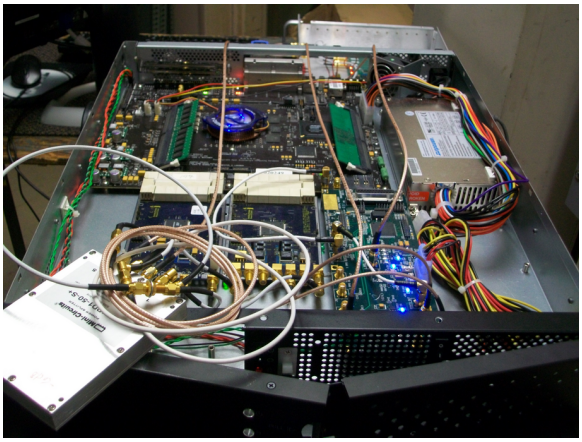

Figure 7.2: The RDBE-S used for testing. The 3 cables draped across the unit come from the noise source and provide signal, 5 MHz and PPS.

The signal paths and data streams are as shown in Figure See Figure 7.3. The labelling of the cables connecting the splitter and the ZDOC inputs (delta, epsilon and zeta) is explained later (see Section 7.5 [zbt_cable_delays], page 44).

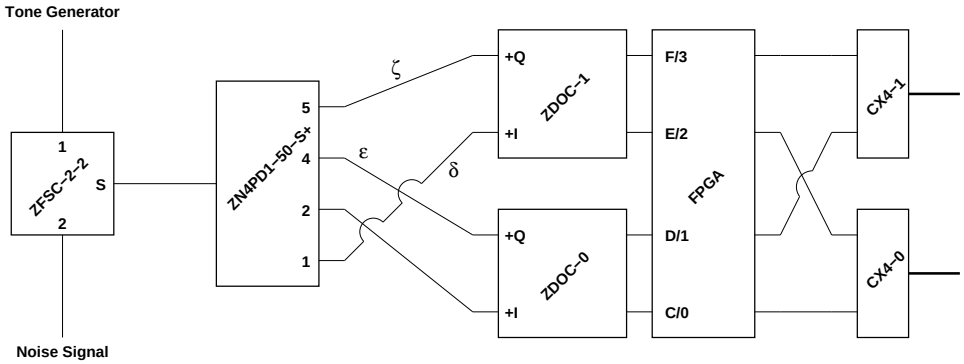

Figure 7.3: The signal paths present in the test are show.

On the output side of the RDBE-S, the two CX4 cables (each carrying two logical threads) are connected to the two ports of the 10 GbE PCI-Express card of the burst mode recorder (see Figure 7.4). The Astro8Gbps personality has a fixed association of ADC data streams with the threads present on the CX4 cables. (I.e. the placement of threads on network IP addresses is not arbitrarily programmable.) The personality was configured to provide

names in the VDIF headers for these streams as threads 0, 1, 2 and 3 and also as stations CC, DD, EE, and FF (and lower case was also used). It was ultimately established that the labelling of threads and stations was consistent.

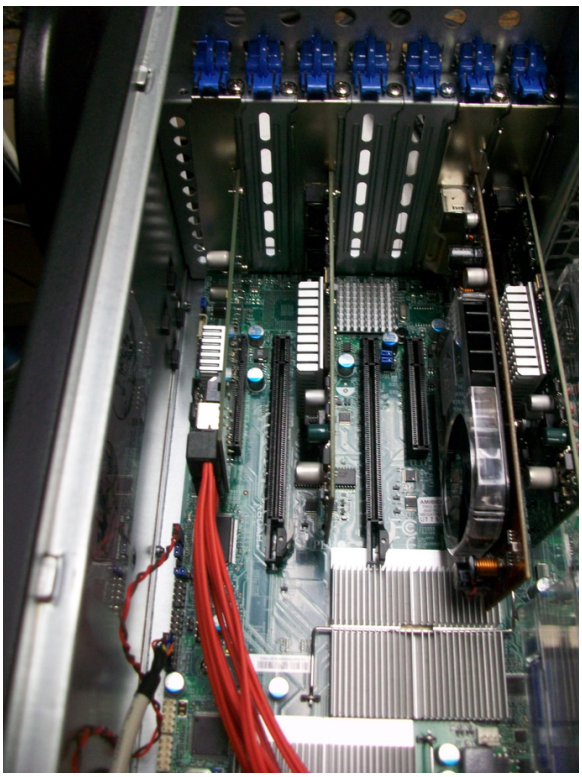

Figure 7.4: The data packets arrive on the 10 GbE PCI-Express card at the center of the picture, and are ultimately stored to disk via the I/O card on the left. The video card and spare 10 GbE card at the right are unused.

7.2 DiFX Setup

As indicated it was sensible to treat the four threads as separate stations (CC, DD, EE and FF; all in Westford) for correlation with DiFX. An evolving script (vdifnoise.sh) was written to set up DiFX runs on the nascent Haystack DiFX cluster. Aside from a number of tedious setup chores, this script manufactures a VEX file suitable for the files present in each run and makes adjustments to the DiFX input file to handle some issues with the version of DiFX (ca 2.0 trunk) used.

In particular, these were high SNR tests, so long DiFX runs were not needed—in practice 0.1s of data was adequate to find the single band delays between the "stations". Since DiFX is expecting integrations to be of multi-second durations (i.e. not arbitrary), this was accomplished by having 1-second long scans that terminated prematurely. (DiFX has adequate logic to catch and handle this properly.) Generally, the correlation was carried out with 512 frequency channels with no averaging. The scan start/stop times were drawn from the RDBE-S timing (i.e. not reflective of real UTC) and the source was equally bogus

(4C66.NN at 3h RA, 81 dec Decl. with NN being the number of the run). The "sky" frequency was arbitrarily set at 8000, and it was checked that the correlations were suitably indifferent to permutations of USB/LSB, polarization &c. as could be specified in the VEX file.

After each DiFX correlation, fourfit was run on the output to report the single-band delay and the correlation strength. The double letter station names were compressed to a single letter (e.g. CC to C) for the fringe plots. However, as these were single-channel "observations", the multiband delay and per-channel data was uniformly uninteresting. It was convenient to carve off the portions of the fringe plot that conveyed useful information. An example is shown in Figure 7.5.

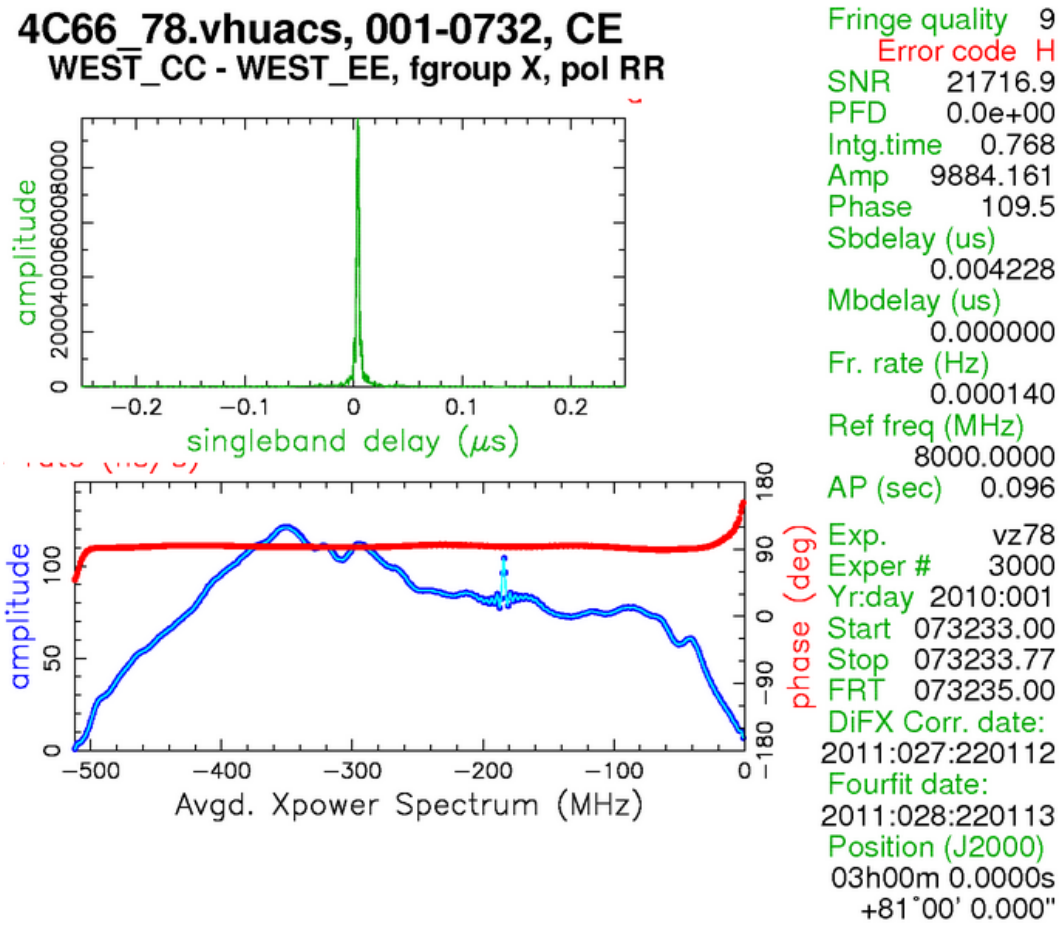

Figure 7.5: The useful bits of a sample fringe plot for two threads ("stations" CC and EE). A single tone at 840 MHz was coupled with the broadband noise source.

As can be seen in the figure, the noise spectrum is broad, but not flat. In this case, a tone at 840 MHz (i.e. the -184 MHz in the LSB Xpower Spectrum plot) was added. The singleband delay was 4.228 ns as will be discussed later. The correlation was very good (98.8%).

7.3 Frequency Checking

A simple VDIF generation tool (`vnoise`) was written to simulate VDIF output for special cases (gaussian noise plus one or more pure tones). The processing script was modified to handle these files equivalently with the "real" data produced by the `RDBE-S` to provide additional checks. (Only two stations are needed for this work—adding extra stations is more processing for no real gain.)

This was most useful to track down minor issues with earlier `Astro8Gbps` personalities as well as minor issues with the VDIF tools themselves. For the aid of future debuggers, the main things that could be done wrong and produce weird behaviors:

- Endian issues (e.g. wrong order of bits/bytes within words)
- Sample representation (i.e. the mapping of the bits of the samples: 00, 01, 10, 11 to input signal levels)

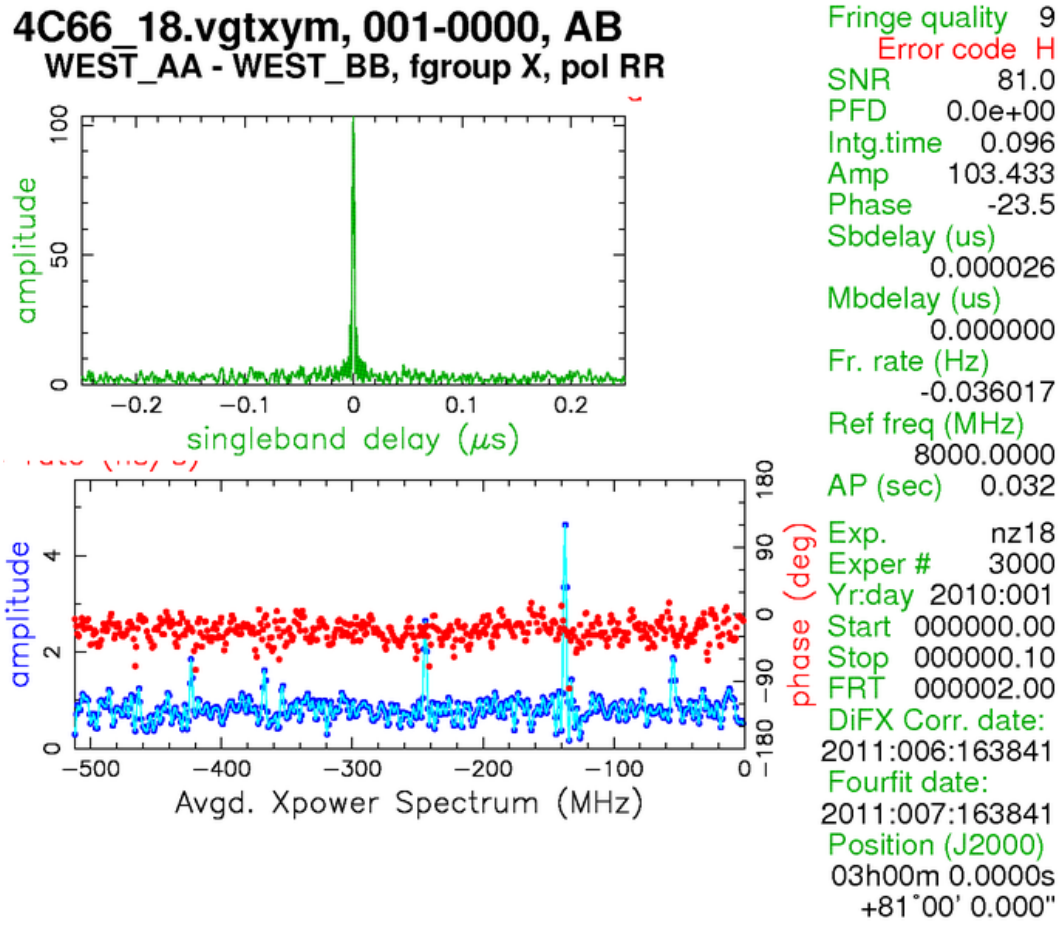

Figure 7.6: Simulated data for two stations (AA and BB) consisting of 1% correlated gaussian noise plus five tones was generated, correlated with `DiFX` and processed with `fourfit`. The tones are clearly visible in the cross-power spectrum.

An example is shown in Figure 7.6 where simulated data for two stations (AA and BB) was constructed from the command:

```
vnoise -vvv -d 0.064 \
    -t 55,0.01 -t 137,0.02 -t 244,0.015 -t 367,0.008 -t 423,0.009 \
    noise_AA_18.vdif noise_BB_18.vdif
```

where the command-line arguments indicate that (in addition to the default 1%-correlated gaussian noise) the sample should last 0.064 seconds, and have additional pure sinusoidal tones place at 55, 137, 244, 367 and 423 MHz with relative amplitudes 0.01, 0.02, 0.015, 0.008 and 0.009. (The 0.064 duration was processed with two 0.032 DiFX sub-integration intervals.)

As can be seen in the figure, the five tones are clearly identifiable in the cross-power spectrum plot (aliased to negative frequencies). The residual single-band delay of 26 picoseconds is a numerical artifact. The amplitude of the cross correlation (1.034%) is dominated by the gaussian component (1.000%).

As indicated, the experimental setup allowed the coupling of a single tone into the noise signal. Through repeated scans with the RDBE-S/BMR setup, one could verify that the tones were appearing at the proper locations in frequency space (see Figure 7.7). The figure is constructed from the raw DiFX output (Swinbourne format) for a single correlation interval (0.032 seconds) on each of six scans.

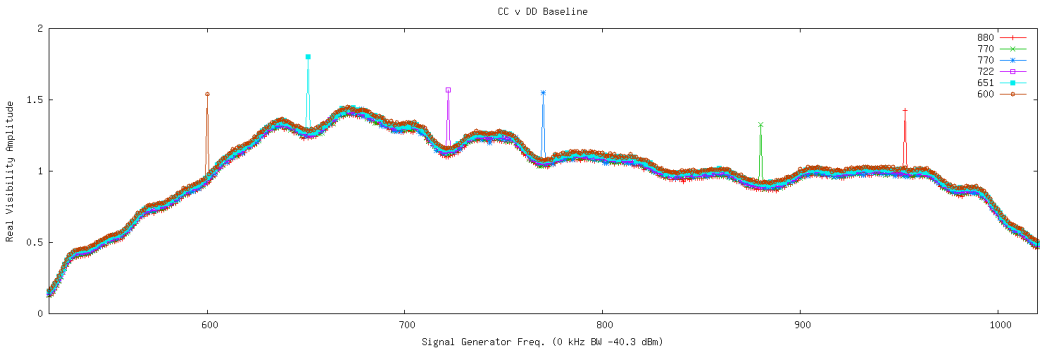

Figure 7.7: A number of pure tones (one per scan) were coupled with the noise source, captured by the RDBE-S/BMR and processed with DiFX. This plot shows an overplot of six such spectral from the DiFX Swinbourne format data files.

And for completeness, in the cross-power spectrum showed up with the same structure in all of the baselines as shown in See Figure 7.8. In this case, the injected tone was at 880 MHz, which aliases to -144 MHz in these LSB plots. (The noise source attenuator was set to 21 dB, and the tone power was -40.3 dBm with zero BW.)

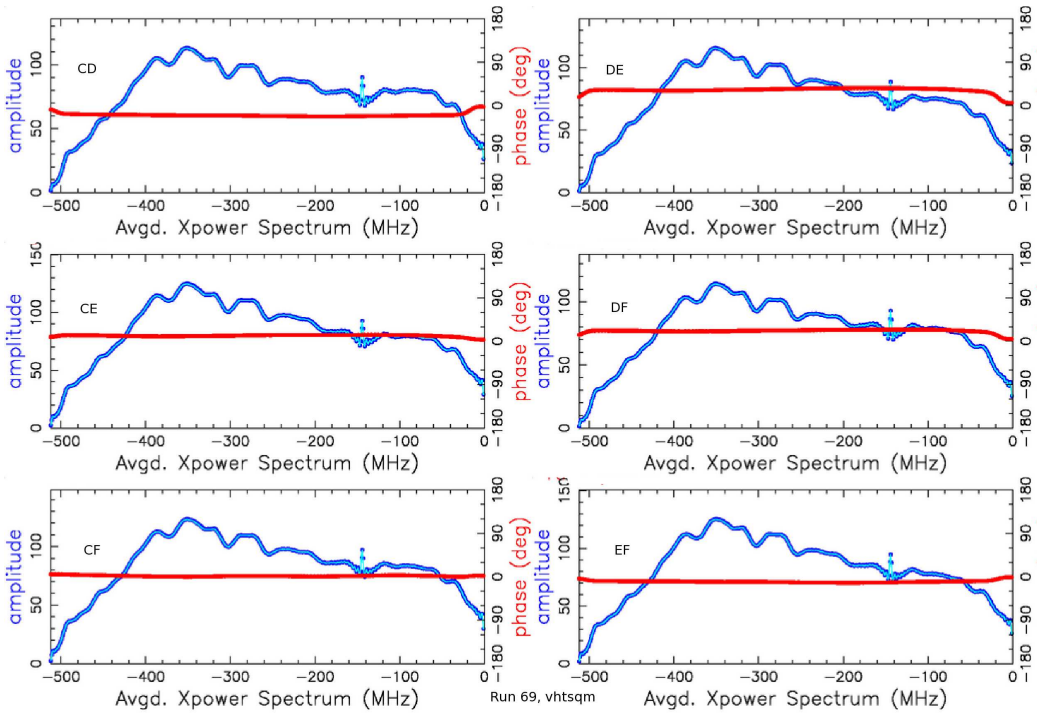

Figure 7.8: The cross-power spectrum for one 880 MHz tone (aliases to -144 MHz) is shown for the six baselines for the 4 threads.

7.4 Bit State Checking

One of the outstanding issues with the "Jan26" revision of the `Astro8Gbps` personality is that the distribution of the bit states does not match what it is being commanded to be. I.e. there is logic in the FPGA to ensure that 68% of the samples are in the middle two states of the gaussian, and the remainder outside.

One possible explanation that was considered was that the input signal was not using the full 8-bit dynamic range of the ADC. To test this, a series of samples were gathered with the attenuation of the noise source varied from 9.5 dB to 21.0 dB (in steps of 0.5 dB) using the front-panel switches. The results are shown in Figure 7.9.

These histograms are as reported through the raw-data capture mechanism and are as plotted with `bpplotter_rdbe.py`. It will be observed from the montage that at 9.5 dB the full dynamic range was being used, and that at 21 dB perhaps 5 of the 8 bits were being used.

Nevertheless, the population of the middle group of states varied from 64.9% through 75.3% with no discernable correlation with the input noise power setting.

At this time, it is believed that the issue is a corruption of some (relatively small) fraction of the data samples.

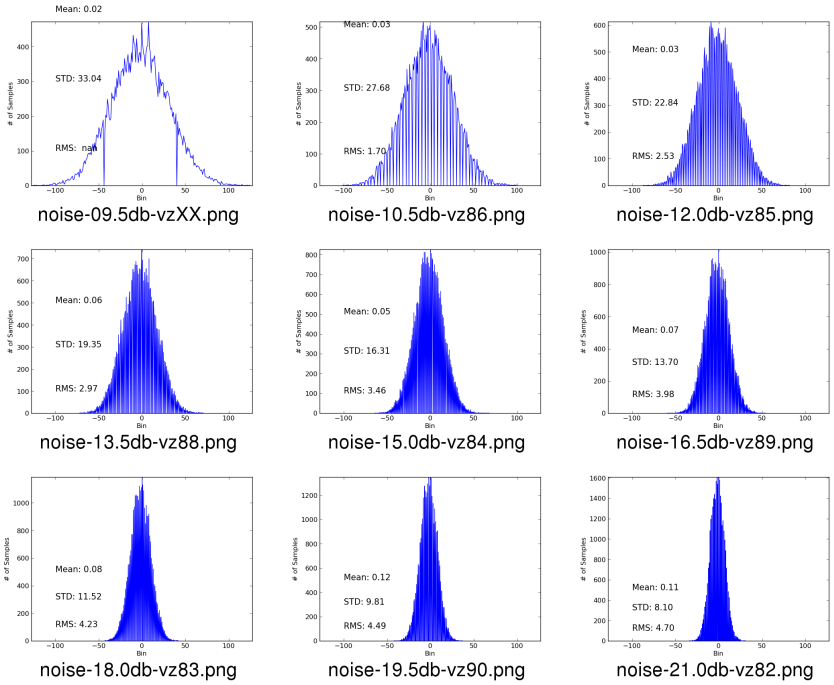

Figure 7.9: Histograms of the input samples were captured with `bpplotter_rdbe.py`. This montage is of the histograms with the input noise attenuation (front panel switches) varied from 9.5 dB to 21.0 dB.

7.5 Single-Band Delays

As mentioned in Section 7.1 [zbt_hw_config], page 37 3 cables were inserted between the 4-way splitter and the IF inputs to allow unambiguous identification of the signal paths. These cables, code-named delta, epsilon and zeta were of approximate lengths 45 cm, 76 cm and 122 cm and would be expected to produce proportionate singleband delays of a few nanoseconds. These cables were added one at a time between runs and the delays measured with fourfit. The sequence of delays on the six baselines is shown in the following table:

Run	**cables**	**CD(ns)**	**CE(ns)**	**CF(ns)**	**DE(ns)**	**DF(ns)**	**EF(ns)**
vz74	none	–0.025	+1.984	+1.971	+2.013	+1.998	–0.013
vz75	d	–0.031	**+4.160**	+1.965	**+4.225**	+1.997	**–2.193**
vz76	e	**+3.632**	+1.983	+1.967	**–1.631**	**–1.669**	–0.017
vz77	z	–0.025	+1.990	**+7.795**	+2.020	**+7.821**	**+5.798**
vz78	dez	+3.629	+4.228	+7.784	+0.543	+4.111	+3.530
vz82	dez	+3.627	–3.581	–0.028	–7.269	–3.699	+3.529
vv11	dez	+3.624	+5.203	+8.759	+1.517	+5.087	+3.527
vv12	dez	+3.624	+4.229	+7.783	+0.539	+4.110	+3.526
vv13	dez	+3.624	+4.229	+7.783	+0.540	+4.110	+3.526
vv14	dez	+3.626	+4.225	+7.783	+0.541	+4.110	+3.527

The autocorrelations were all 0.0 ns and are not shown. Run vz74 did not have any extra cables. For Runs vz75 through vz77 the cables were added one-at-a-time; the delays that changed substantially from Run vz74 are emphasized in the table. Note that even without the cables, the baselines between IFs on different ADCs are delayed by 2 ns—this is almost exactly 2 samples (1024 MHz sampling is 0.9766 ns/sample).

With all three cables installed, the timings are as with Run vz78, and this was the typical case. The arrangement is shown in Figure 7.10.

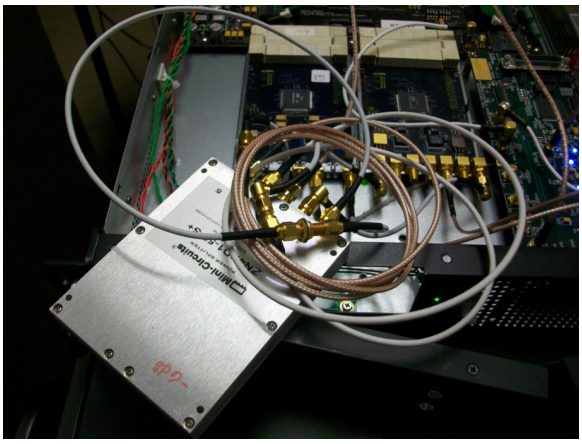

Figure 7.10: Three cables of various lengths were inserted on three of the four splitter outputs to produce different single-band delays on the various baselines.

However, the timings were occasionally different following a reset of the RDBE-S as seen in Runs vz82 and vv11 which are consistent with -6 samples and 3 samples respectively. These were relatively rare cases—in the majority of cases the delay following a reset was 2 ns. Reviewing the last few lines in the table we see that the results are reproduceable at the one or two picosecond level.

The delays between "stations" can be expressed by the following equations:

```
time(C-D) = epsilon
time(C-E) = delta + num * tsamp
time(C-F) = zeta  + num * tsamp
time(D-E) = delta - epsilon + num * tsamp
time(D-F) = zeta  - epsilon + num * tsamp
time(E-F) = zeta  - delta
```

where the shift between the ADC cards is expressed as `num * tsamp` where `tsamp` is 1/1024 MHz.

Reducing these results with equations for the delays, we find that the cables insert delays of 2.275 ns (delta), 3.625 ns (epsilon) and 5.830 ns (zeta) respectively—fully consistent with the speed of light in the cable (reduced from vacuum by 30% or so).

Note that the equations are over-determined. The first three equations were invariably solved, and the remainder were checked with the results. Although the solutions for `delta`, `epsilon` and `zeta` were consistent to within a few picoseconds, the "checks" were typically

off by a few dozen picoseconds. That could be a systematic issue with how fourfit calculates the singleband delays.

The singleband delay plots, with all cables inserted are shown in Figure 7.11. The actual values come from the associated fourfit labelling in the fringe plots (not shown).

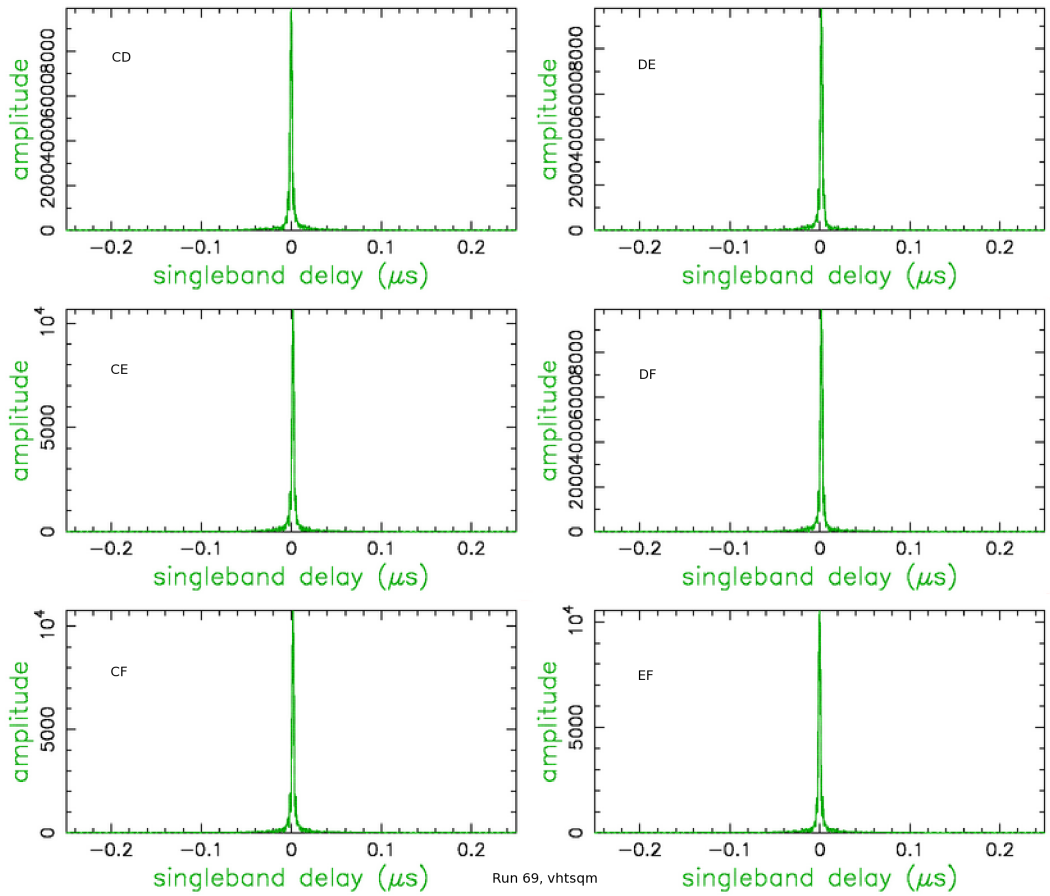

Figure 7.11: The singleband delay observed with fourfit for the six baselines corresponding to Figure 7.8 are shown. The delays, on the order of nanoseconds, are barely visible in these plots.

As indicated at the outset the non-zero, occasionally variable delay between the cards following a reset is a property of the low-level characteristics of the boards and the FPGA programming. A fix for this (detecting and adjusting) is planned.

Acknowledgements

This work was supported by the National Science Foundation.

Beyond that, although this memo was written by its author (who takes full responsiblity for errors/ommissions/&c), none of this would have been possible without considerable support, encouragement, advice, &c. from quite a number of sources.

Most particularly, the `Astro8Gbps` personality is the work of Alan Hinton; and Trevor Capallo helped in many ways throughout many phases of the burst mode project. Roger Cappallo vetted the initial architecture discussions and helped with invaluable advice throughout the debugging of all the various components of the system(s).

The burst mode hardware was designed, purchased, and assembled with the invaluable help of Kevin Dudevoir, Jason Soohoo, Richard Crowley and Dave Fields. The `RDBE-S` used is a development system of the larger `RDBE` development effort variously involving Arthur Niell, Alan Hinton, Chester Ruszczyk, Chris Beaudoin, Alan Rodgers (and many other individuals, both here at Haystack and at NRAO and elsewhere). The correlations were performed with `DiFX`; discussions with various members of that group are gratefully acknowledged.

List of Figures

Index