Borealis Documentation

Release 1.0

SuperDARN Canada

Dec 07, 2022

CONTENTS

CHAPTER

ONE

SUPERDARN CANADA SYSTEM SPECIFICATIONS

1.1 Digital Radio Equipment

- NOTE : ALL cables are phase matched unless specified otherwise
- **17x Ettus USRP N200 (16 and 1 spare)**
  - 17x Ettus LFTX daughterboards
  - 17x Ettus LFRX daughterboards
- 1x Ettus Octoclock-g (includes GPSDO)
- 2x Ettus Octoclock
- 51x ~8 1/4” SMA bulkhead Female to Male RG-316 for daughterboards
- 18x 48” SMA Male to Male RG-316 for PPS signals
- 18x 48” SMA Male to Male RG-316 for 10MHz REF signals
- 1x SMA Male to 0.1” pin header RG-316 for PPS signal input to motherboard
- GPS Antenna (Male SMA connector)
- 17x Custom TXIO Revision 5.0 board (for transmitter interfacing)
- 22x Mini-Circuits ZFL-500LN pre-amps (20 and 2 spare)
- 8x coax cables and adapters for to/from INTF (interferometer) pre-amps
- 32x coax cables for to/from main array filters and pre-amps inside transmitter
- 1x 15V, 0.5A power supply (INTF pre-amps)

1.2 Control Computer

- 1x GeForce GTX 2080 or better
- 2x 16GB DDR4
- 1x Monitor
- 1x Power supply, 1000W 80 Plus Gold or better
- 1x Intel Core i9 10 core or better
- 1x Cpu liquid cooling unit
- 1x CPU socket compatible motherboard with serial port header for PPS discipline

- 1x 256GB SSD
- 1x 1TB HDD
- 1x Intel X550-T2 10Gb PCIe network card **NOTE**: Intel 82579LM controllers WILL NOT WORK

1.3 Networking

- 3x Netgear XS708E-200NES (North American model #) 10Gb switches (parent model name is XS708Ev2)
- 27x SSTP CAT 6a 7ft cables or better*
- 2x SSTP CAT 6a 15ft cables*

Note that the network cables needs to be verified for the whole system
as not all cables seem to work reliably.

Models tested and known to work include:

- Cab-CAT6AS-05[GR|BK|GY|RE|WH]
- Cab-CAT6AS-15GR

Models that were tested and do not work:

- CAT 5e cables
- Non SSTP cables (not dual shielded)
- Cab-Cat7-6BL
- Cab-Cat7-6WH

1.4 Rack and Cabling

- 4x 8 outlet rackmount PDU
- 2x APC AP7900B rackmount PDU
- 1x 4 post 42U rack
- 4x custom-made USRP N200 rackmount shelves (or Ettus ones)
- 1x rackmount shelf for interferometer pre-amps

CHAPTER

TWO

FULL SYSTEM SETUP PROCEDURES

Here are the notes on SuperDARN Canada's Borealis setup procedures.

2.1 Hardware

2.1.1 System Overview and Rack Setup

Below is a recommended configuration in comparison to a common SuperDARN system:

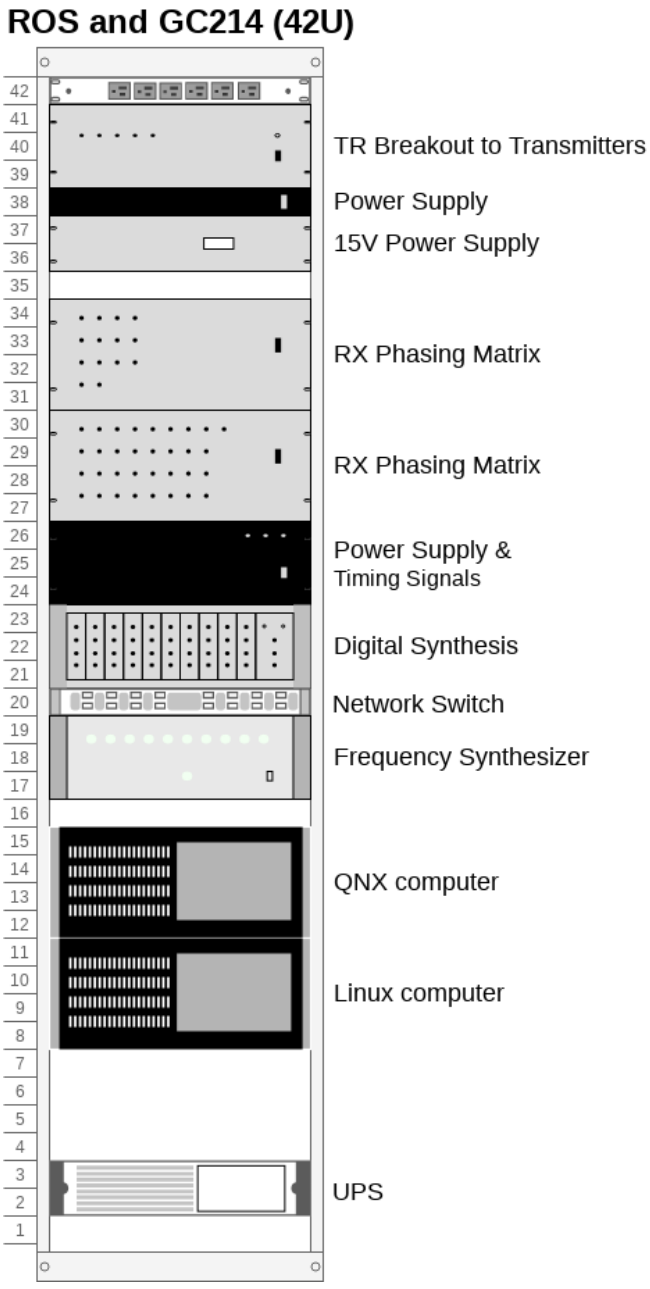
Here is an actual rack configuration as installed by SuperDARN Canada at the Saskatoon (SAS) SuperDARN site. Note that space has been allowed between the rackmount items to allow for cable routing. There is a lot of cabling involved at the front of the devices.

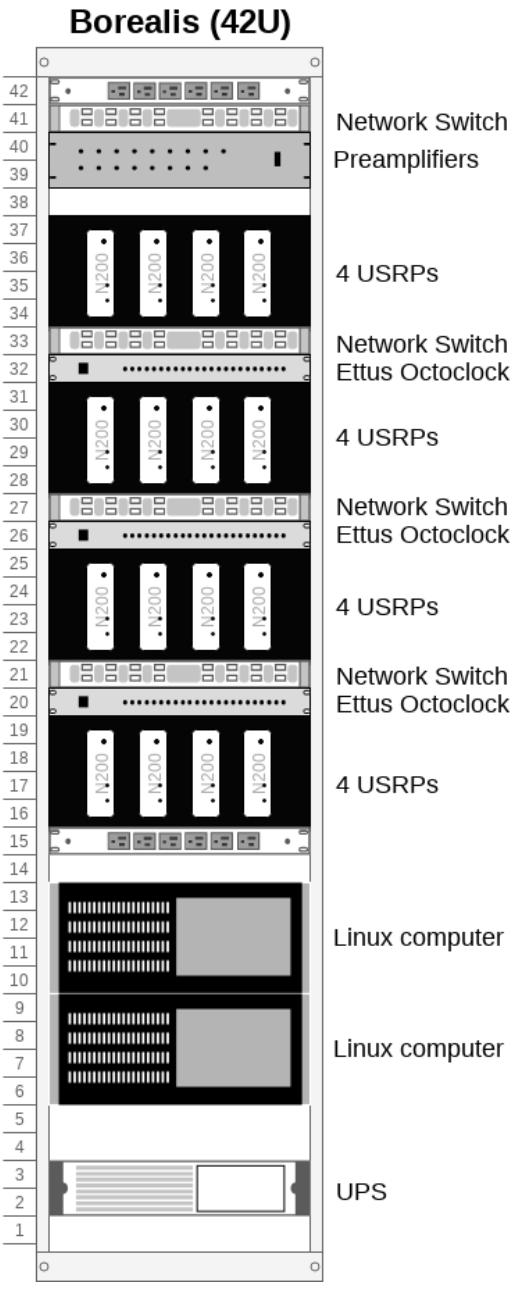
The items installed in the rack at the Saskatoon site are listed below in order from top to bottom in the rack:

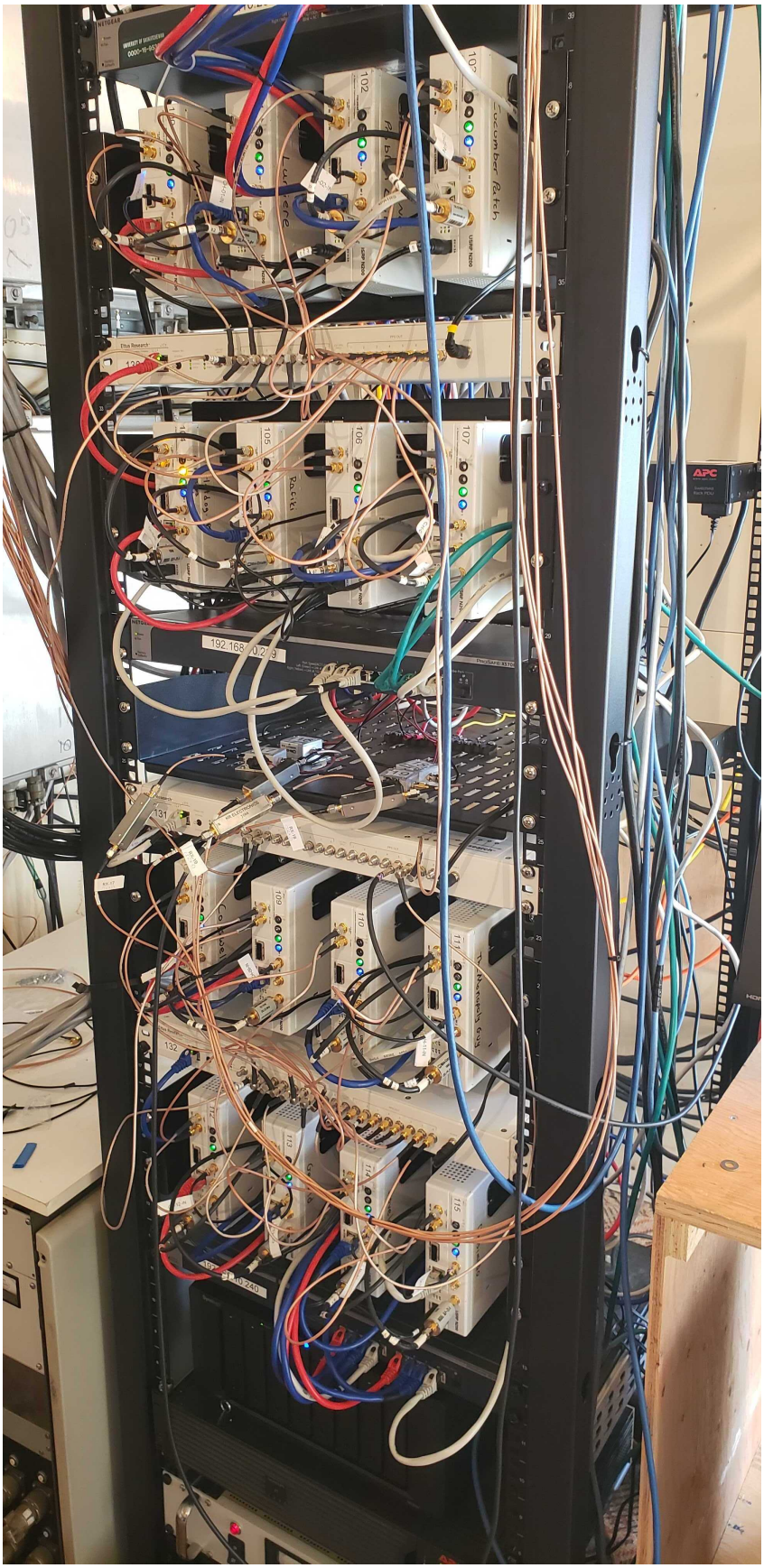
- Netgear XS708E 10Gb switch
- USRP rackmount shelf (in-house design) with 4 x N200s
- Ettus Octoclock
- USRP rackmount shelf (in-house design) with 4 x N200s
- Netgear XS708E 10Gb switch
- Rackmount shelf with 4 x low-noise amplifiers for the interferometer array channels, and a terminal strip for power (supplied by 15V Acopian)
- Ettus Octoclock-G (with GPSDO)
- USRP rackmount shelf (in-house design) with 4 x N200s
- Ettus Octoclock
- USRP rackmount shelf (in-house design) with 4 x N200s
- Netgear XS708E 10Gb switch
- Synology Network Attached Storage device
- APC Smart UPS
- 15V Acopian power supply

(3 x APC PDUs (AP7900B) are mounted at the back of the rack)

The Borealis computer is not in a rackmount case, instead it is placed to the right of the rack.


ROS and GC214 (42U)
TR Breakout to Transmitters
Power Supply
15V Power Supply
RX Phasing Matrix
RX Phasing Matrix
Power Supply &
Timing Signals
Digital Synthesis
Network Switch
Frequency Synthesizer
QNX computer
Linux computer
UPS


Borealis (42U)
Network Switch
Preamplifiers
4 USRPs
Network Switch
Ettus Octoclock
4 USRPs
Network Switch
Ettus Octoclock
4 USRPs
Network Switch
Ettus Octoclock
4 USRPs
Linux computer
Linux computer
UPS


192.168.0.219
APC

2.1.2 USRPs

This guide assumes set up of a brand new, unopened Ettus N200.

Initial Test of the Unit

Install Daughterboards

1. Open the unit and install the LFTX and LFRX daughtercards using hardware provided. The main USRP PCB is clearly marked with where to connect TX and RX daughterboards, and there is only one way they can fit while still allowing all the screw holes to line-up. The RX daughterboard is located directly above the fan power connection at the back of the motherboard.
2. Connect the output of TXA using an SMA cable to the custom-added SMA connection point on the front of the USRP using one of the SMA Male to female bulkhead SMA cables. Connect the output of RXA to RF1 and RXB to RF2 on the front of the USRP using two more SMA Male to female bulkhead cables.
3. Verify that the jumper J510 on the N200 motherboard is jumping the two 0.1" header pins furthest from the board edge. The jumper is located behind the CLK_REF (REF IN) SMA connector on the front of the N200. This ensures that the reference clock input is coming from the front-panel SMA connector, and not the secondary SMA connector located on the motherboard labeled 'J507 CLK_REF 2'.

Connect to the USRP

4. USRPs have a default IP address of *192.168.10.2*. Assign a computer network interface an address that can communicate in this subnet. Connect the USRP to the computer's network interface either directly or through one of the switches from the system specifications. Connect the USRP power supply.
5. Verify the board powers on and is discoverable. The USRP should be discoverable by pinging 192.168.10.2. Ettus' USRP UHD library supplies a tool called *uhd_usrp_probe* which should also be able to detect the device. See software setup for notes on installing UHD. The USRP may require a firmware upgrade.
6. Connect an SMA T connection (F-M-F) to the TX output from the front of the N200, connect another SMA T (F-M-F) to the first T. Connect one end of the second SMA T to RX1, and the other end to RX2 with phase matched SMA M-M cables. Connect the free SMA output of the first SMA T to the scope. Connect the Octoclock PPS and 10MHz reference signals to the USRP. Make sure that the jumper on J510 is in the rightmost position connecting the front panel 10MHz as the system reference.

Test the USRP

7. Use the UHD utilities *rx_samples_to_file*, *tx_bursts* and *txrx_loopback_to_file* to verify the USRP works. Use the scope to see the transmit signal. The RX samples will be a binary file that can be quickly read in a plotted with Numpy/Matplotlib. While testing, watch the front panel LEDs to see that they work as expected.

Disassembly for Enclosure Modifications

8. If the USRP is working correctly, the inner motherboard, fan, daughtercards and RF cables can all be removed from the unit. Carefully peel the product sticker and store with the motherboard, this contains the MAC address, SN and PN of the unit. All removed components and the sticker can be stored in the anti-static bags that were supplied with the unit. The enclosure is ready for machining the additional holes. Ensure that you note which way the fan was installed for reinstallation later.

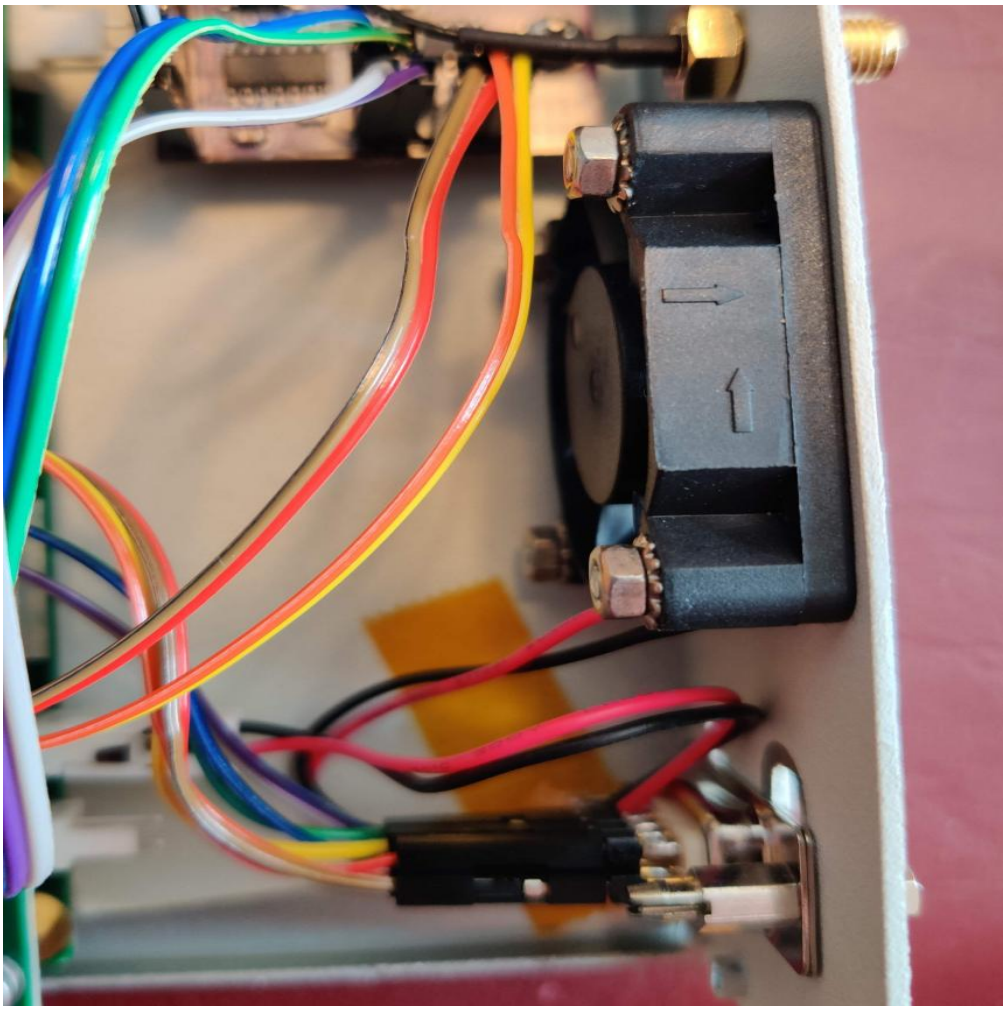
Custom Enclosure Modifications

The custom machining involves the following machining steps

1. Five extra SMA holes that are 'D' shaped to fit most standard SMA bulkhead connectors. Four of these holes are on the back of the N200, and one is on the front, in line with the two existing RF1 and RF2 SMA bulkhead holes.
2. A DSUB shaped hole for a DE9 connector at the rear of the unit for connection to existing SuperDARN transmitters.
3. Four holes for standard 5mm LED clips (6.35 +/-0.05mm diameter) with 9.5mm centers to appropriately space them.

Installing the Custom-Made TXIO Board

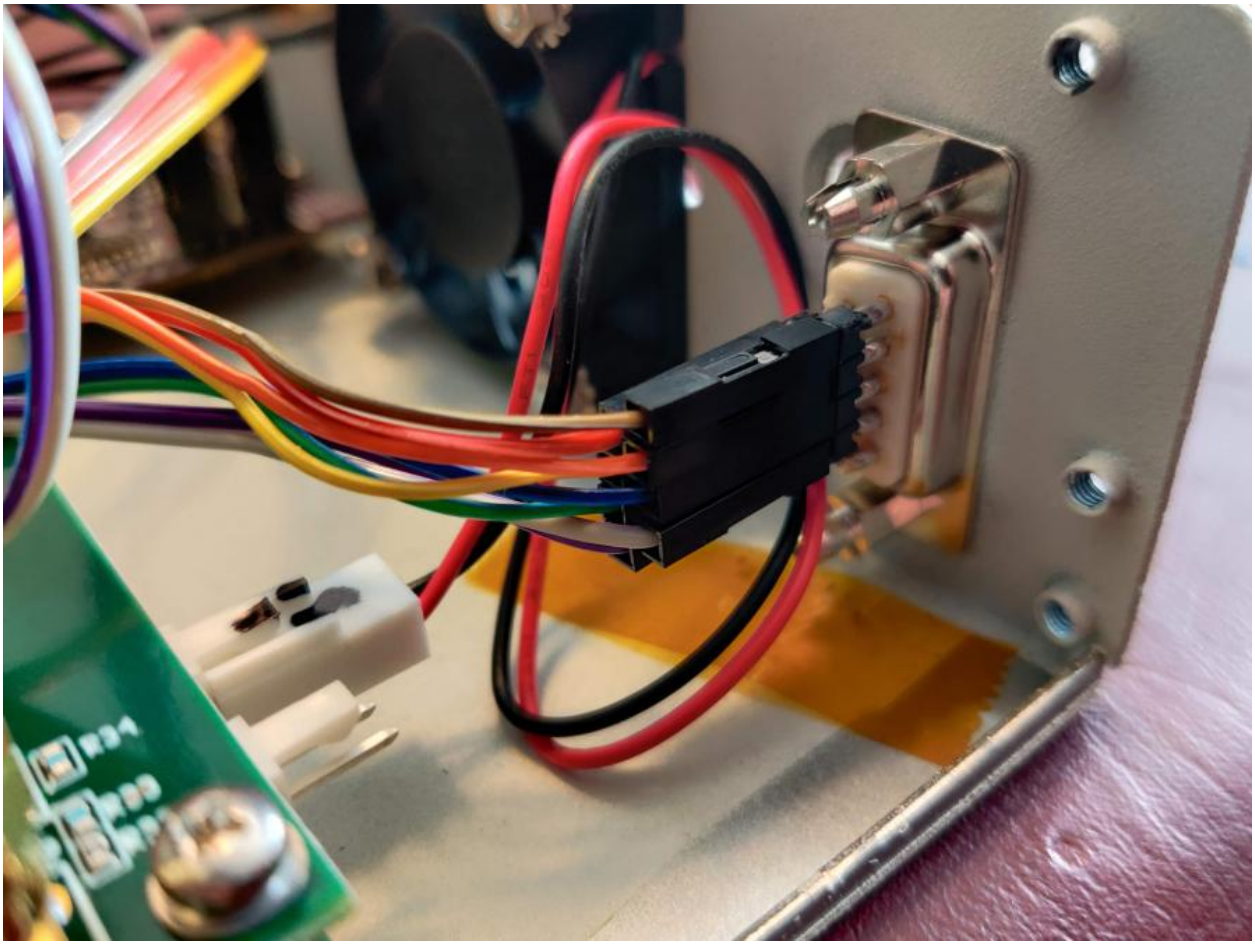
1. Once the enclosures are machined, the electronics and components can all be reinstalled. Place the product sticker back in place on the rear of the unit. There are slight indentations in the case to indicate where the product sticker goes. Connect RXA to port RF1, connect RXB to port RF2, and connect TXA to the additional front panel hole that was added.
2. Install the LEDs (TODO: Add description of how to install LED clip here) into their corresponding holes. The order of the LED install patterns from left to right are the TX only indicator (RED), the IDLE indicator (YELLOW), the RX only indicator (GREEN) and the TR indicator (BLUE). Optionally, add labels to the LEDs on the front panel.
3. Install the fan, making sure to re-install it the same way it was originally installed.

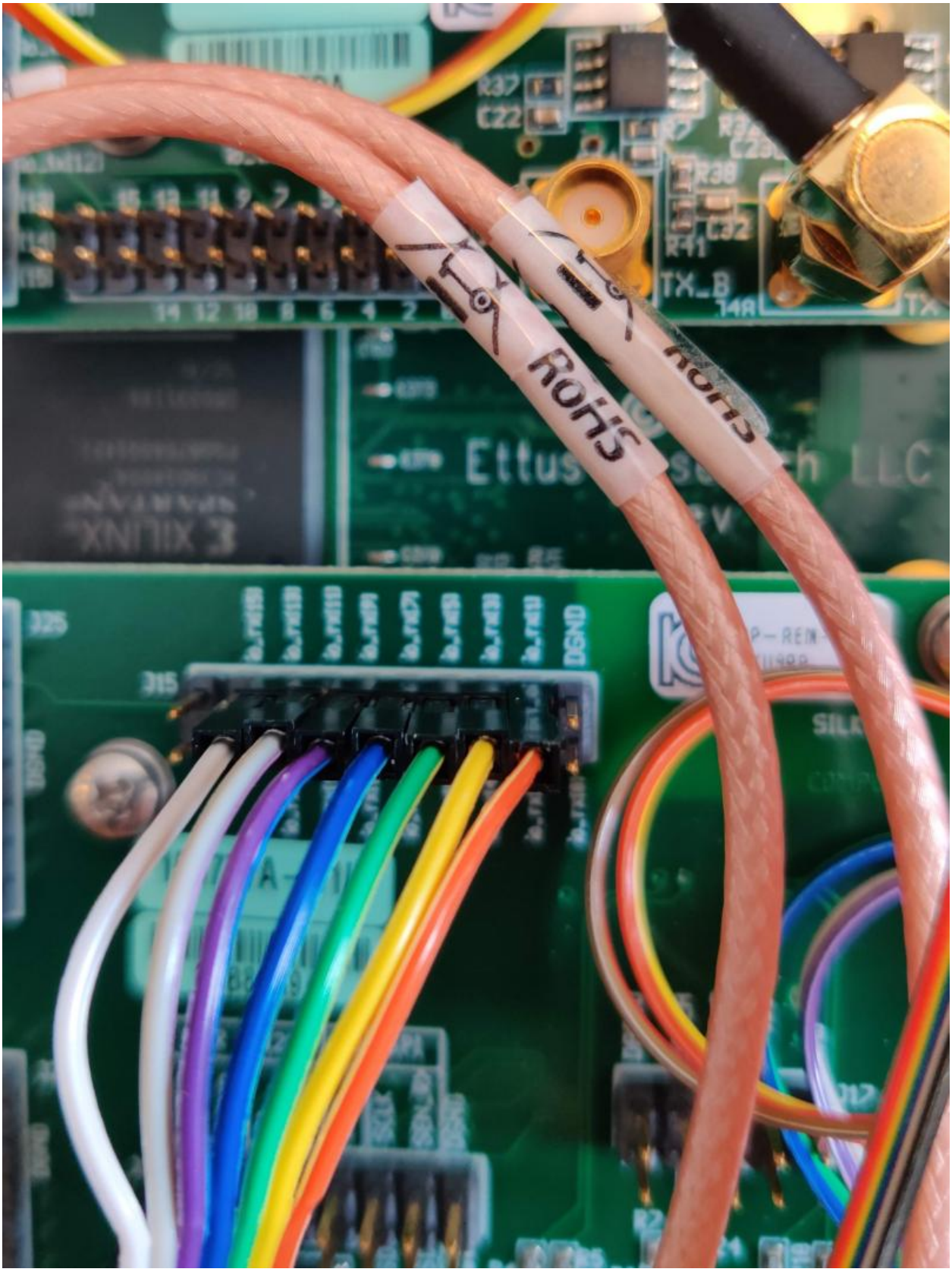


Pre-Assemble the TXIO board before installation into the N200

4. Begin by connecting eight 0.1” female-female jumper cables to pins 1-4 and 6-9 of the D-sub connector. The other ends of these wires connects to header J2 on the TXIO board

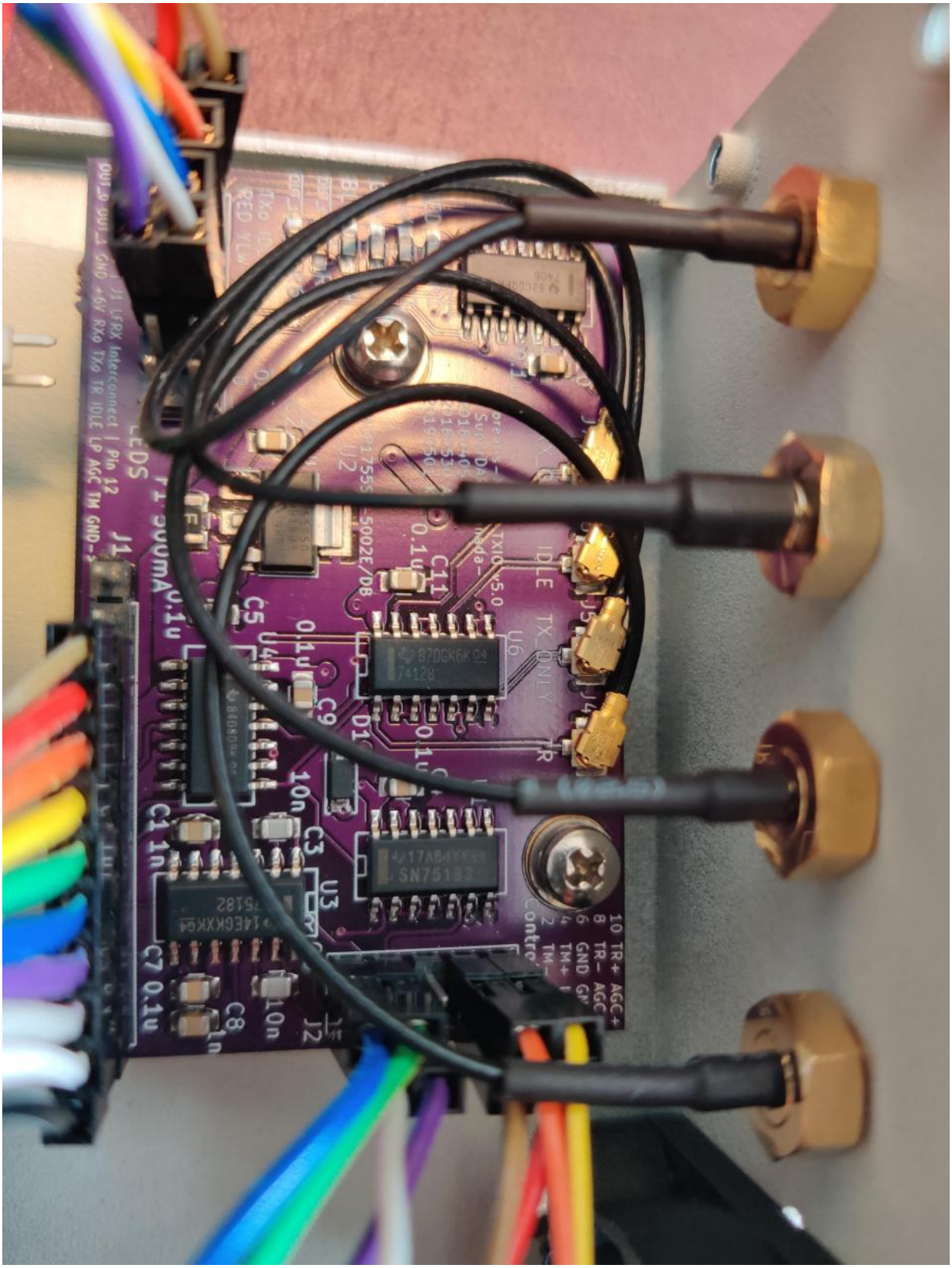
Colour	Sig	DSUB	J2
Brown	AGC-	1	7
Orange	TR-	2	8
Blue	TM-	3	2
Grey	LP-	4	1
[NC]	[NC]	5	[NC]
Red	AGC+	6	9
Yellow	TR+	7	10
Green	TM+	8	4
Purple	LP+	9	3





5. Connect the four U.Fl to SMA female bulkhead cables to J4, J5, J6 and J7 of the TXIO board. Orientation of the

cables doesn't matter, as they will fit in the N200 case if rotated properly.



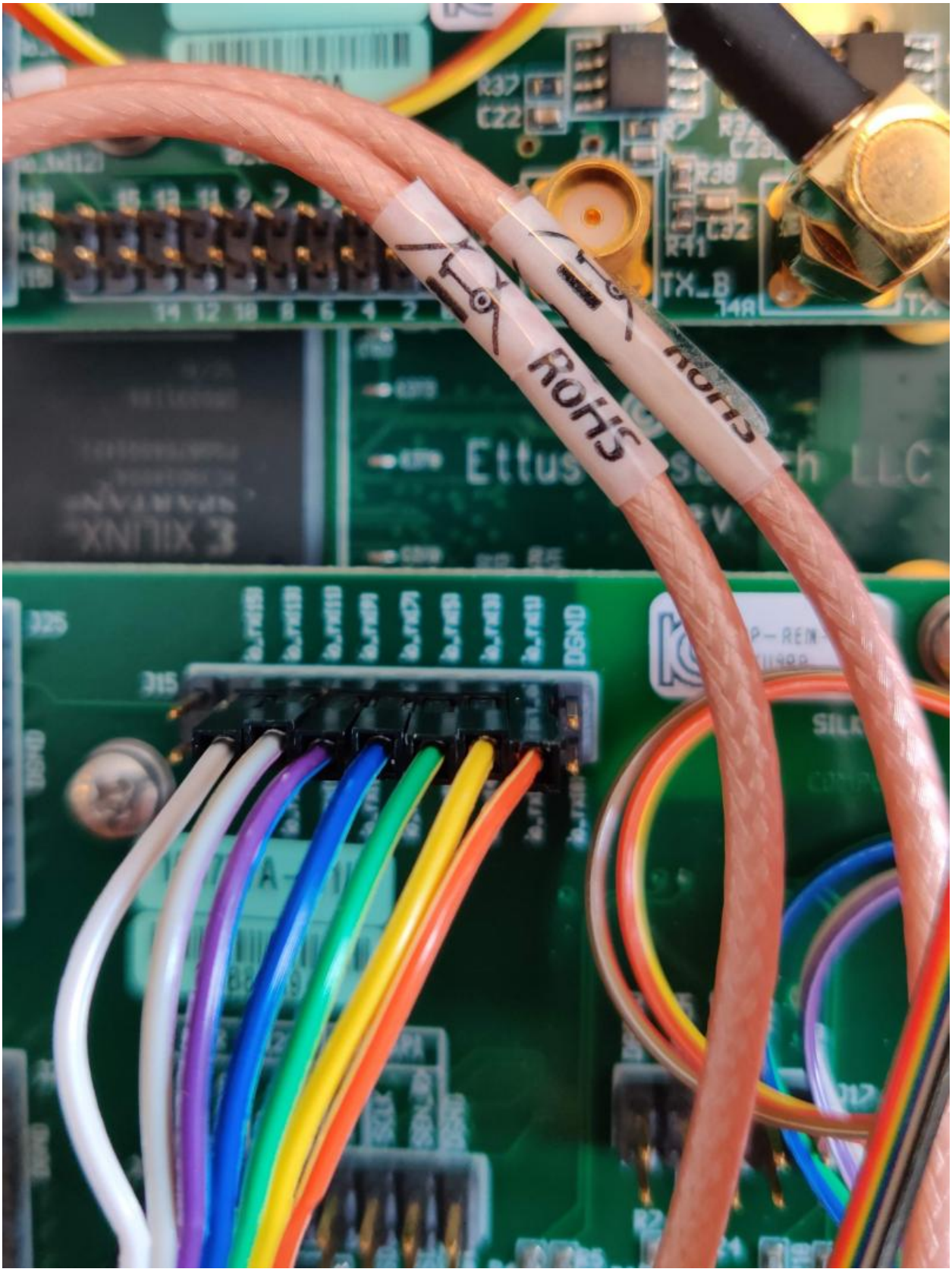
6. Connect 4 pairs of 0.1" female to female jumper wires to header J3 on the TXIO board. THe other ends will connect to the LEDs already installed in the N200 case. There is no need to connect anything to the 4 rightmost pins on J3, these are expansion headers and two are connected (label 'OUT') to the leftover open collector pins on the LED driver chip U5 (SN7406D), the other two (labels '_0' and '_1') are connected to the 5V rail via pullup resistors R5 and R6. **NOTE** If you use your own voltage supply with the open-collector outputs, be aware that the maximum voltage is 30V, and the maximum current sink is 40mA. See the SN7406D datasheet for more details.

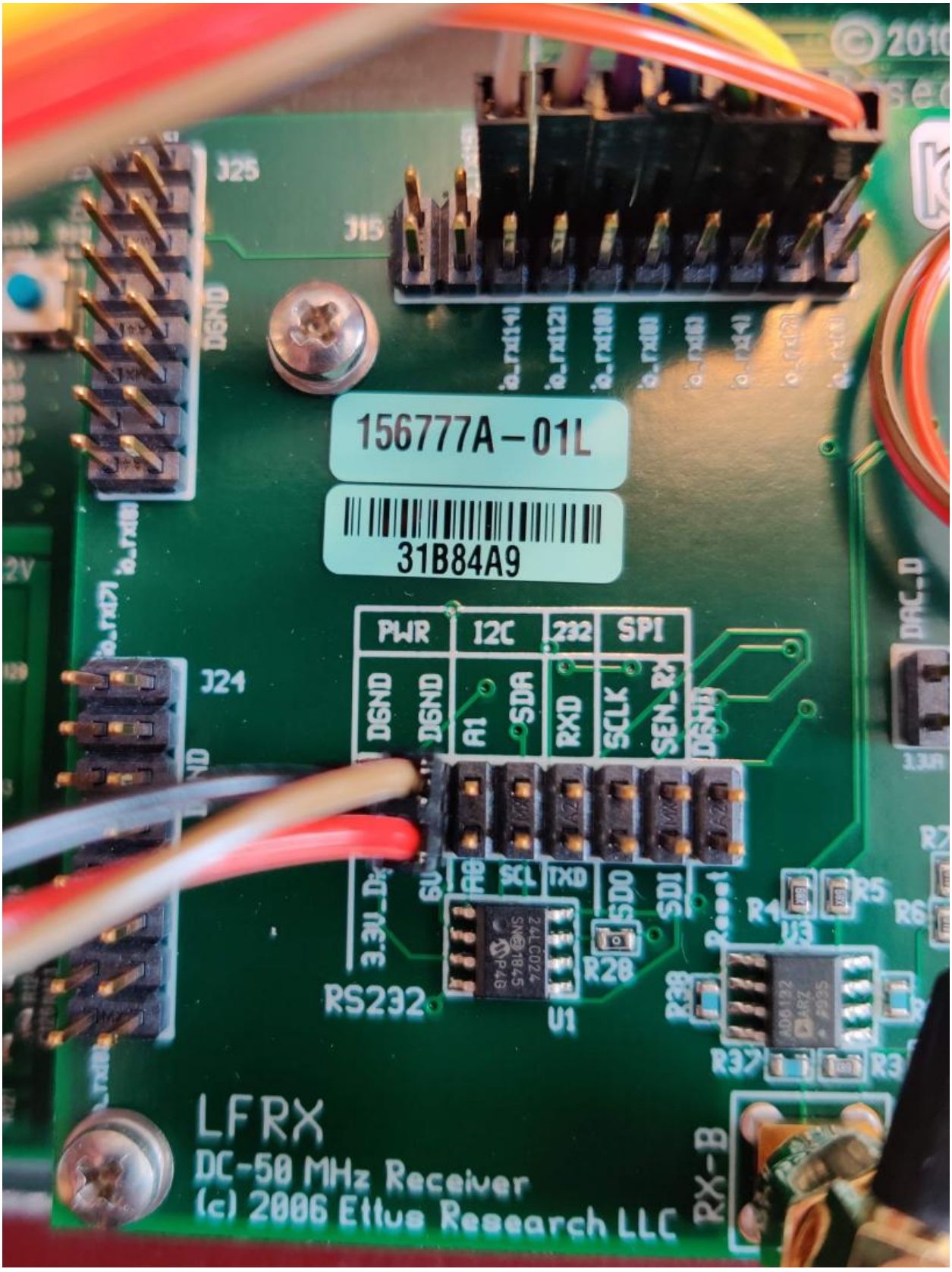
J3 Pin label	Wire Colour	LED Connection
TXo	Brown	RED-
RED	Red	RED+
IDLE	Orange	Yellow-
YLW	Yellow	Yellow+
RX	Blue	Green-
GRN	Green	Green+
TX	Grey	Blue-
BLU	Purple	Blue+

NOTE '-' means cathode, '+' means anode

7. **Connect 10 0.1" female to female jumper wires to J1, the other ends will connect to the LFRX daughterboard pin headers.**

J1 Pin	Pin label	Wire colour	LFRX header	LFRX Pin
1	OUT_0	[NC]	[NC]	[NC]
2	OUT_1	[NC]	[NC]	[NC]
3	GND	Brown	J16	'DGND'
4	+6V	Red	J16	'6V'
5	RXo	Orange	J15	io_rx[1]
6	Txo	Yellow	J15	io_rx[3]
7	TR	Green	J15	io_rx[5]
8	IDLE	Blue	J15	io_rx[7]
9	LP	Purple	J15	io_rx[9]
10	AGC	Grey	J15	io_rx[11]
11	TM	White	J15	io_rx[13]
12	GND	Black	J16	'DGND'


RoHS
RoHS
TX_B
TX
XILINX


156777A-01L
31B84A9
J25
J15
J24
PWR
I2C
232
SPI
DGND
DGND
A1
SDA
RXD
SCLK
SEN_RX
SCL
TXD
SDO
SDI
RS232
U1
R4
R5
R6
R28
R38
R37
LFRX
DC-50 MHz Receiver
(c) 2006 Ettus Research LLC
RX-B

8. **Install the TXIO board by screwing it into place on the USRP housing with the two provided holes. The TXIO board uses the same size and style of screw that the N200 motherboard and daughtercards do.**
    - Install the DSUB connector with the provided standoff screws. **NOTE** some models of DSUB will have split lock washers, but we've found that the thickness of the N200 case is too thick to use them. The DSUB standoff screws are notoriously easy to snap as well, so be careful.
    - Install the 4x SMA female bulkhead cables at the back of the N200, when facing the rear of the N200 case the order from left to right is: J4, J5, J6, J7 (the same order as on the PCB, so no wires should cross each-other).
    - Finally, connect the LFRX jumper wires from J1 and LED wires from J3 to complete the installation.

9. Follow the testing procedure below to run a simple test of the TXIO outputs.

TXIO OUTPUT TESTS

- Connect a needle probe to channel one of your oscilloscope and set it to trigger on the rising edge of channel one.
- Run test_txio_gpio.py located in borealis/testing/n200_gpio_test. Usage is as follows:

  python3 test_txio_gpio.py <N200_ip_address>

- When prompted to enter the pins corresponding to the TXIO signals, press enter to accept the default pin settings. This will begin the tests. Pressing CTRL+C and entering "y" will tell the program to run the next test.
- Insert the needle probe into the SMA output corresponding to RXO. The scope signal should be the inverse of the pattern flashed by the GREEN front LED. Then, proceed to the next test (CTRL+C, then enter "y").

- Insert the needle probe into the SMA output corresponding to TXO. The scope signal should be the inverse of the pattern flashed by the RED and BLUE front LEDs. Then, proceed to the next test (CTRL+C, then enter "y").
- Insert the needle probe into the SMA output corresponding to TR. The scope signal should be the inverse of the pattern flashed by the BLUE and GREEN front LEDs. Then, proceed to the next test (CTRL+C, then enter "y").
  - Insert the needle probe into the hole corresponding to pin 7 of the D-Sub connector (TR+). The scope signal should follow the pattern flashed by the BLUE and GREEN front LEDs.
  - Insert the needle probe into the hole corresponding to pin 2 of the D-Sub connector (TR-). The scope signal should be the inverse of the pattern flashed by the BLUE and GREEN front LEDs.
- Insert the needle probe into SMA output corresponding to IDLE. The scope signal should be the inverse of the pattern flashed by the YELLOW front LED. Then, proceed to the next test (CTRL+C, then enter "y").
- Insert the needle probe into the hole corresponding to pin 8 of the D-Sub. The scope signal should follow the sequence of numbers being printed to your terminal (high when the number is non-zero, low when the number is zero).
  - Insert the needle probe into the hole corresponding to pin 3 of the D-Sub. The scope signal should be the inverse of the sequence of numbers being printed to your terminal. Then, proceed to the next test (CTRL+C, then enter "y").
- To properly perform the loopback tests of the differential signals, connect the D-Sub pins to each other in the following configuration:
  - Pin 6 to pin 7
  - Pin 1 to pin 2
  - Pin 8 to pin 9
  - Pin 3 to pin 4
- Once connected ensure that during the TR, AGC loopback test, the hex digit is non zero when the terminal indicates the output pin is low, and vice versa. Then, proceed to the next test (CTRL+C, then enter "y").
- Ensure that during the TM, LP loopback test, the hex digit is non zero when the terminal indicates the output pin is low, and vice versa. Press CTRL+C, then enter "y" to end the tests.
- This concludes the tests! If any of these signal output tests failed, additional troubleshooting is needed. To check the entire logic path of each signal, follow the testing procedures found in the TXIO notes document.

5. Install enclosure cover lid back in place, ensuring that no wires are pinched.

Configuring the Unit for Borealis

1. Use UHD utility usrp_burn_mb_eeprom to assign a unique IP address for the unit. Label the unit with the device IP address.
2. The device should be configured and ready for use.

2.1.3 Pre-amps

For easy debugging, pre-amps are recommended to be installed inside existing SuperDARN transmitters where possible for SuperDARN main array channels. SuperDARN transmitters typically have a 15V supply and the low-noise amplifiers selected for pre-amplification (Mini-Circuits ZFL-500LN) operate at 15V, with max 60mA draw. The cable from the LPTR (low power transmit/receive) switch to the bulkhead on the transmitter can be replaced with a couple of cables to and from a filter and pre-amp.

Note that existing channel filters (typically custom 8-20MHz filters) should be placed ahead of the pre-amps in line to avoid amplifying noise.

It is also recommended to install all channels the same for all main array channels to avoid varying electrical path lengths in the array which will affect beamformed data.

Interferometer channels will need to be routed to a separate plate and supplied with 15V by a separate supply capable of supplying the required amperage for a minimum of 4 pre-amps.

2.1.4 Computer and Networking

To be able to run Borealis at high data rates, a powerful CPU with many cores and a high number of PCI lanes is needed. The team recommends an Intel i9 10 core CPU or better. Likewise a good NVIDIA GPU is needed for fast data processing. The team recommends a GeForce 1080TI/2080 or better. Just make sure the drivers are up to date on Linux for the model. A 10Gb(or multiple 1Gb interfaces) or better network interface is also required.

Not all networking equipment works well together or with USRP equipment. Some prototyping with different models may be required.

Once these components are selected, the supporting components such as motherboard, cooling and hard drives can all be selected. Assemble the computer following the instructions that come with the motherboard.

2.2 Software

SuperDARN Canada uses OpenSUSE for an operating system, but any Linux system that can support the NVIDIA drivers for the graphics card will work. The current latest version of OpenSuSe (15.1) is known to work.

1. Install the latest version of the NVIDIA drivers (see https://en.opensuse.org/SDB:NVIDIA_drivers). The driver must be able to support running the GPU selected and must also be compatible with the version of CUDA that supports the compute capability version of the GPU. Getting the OS to run stable with NVIDIA is the most important step. You may need to add your linux user account to the 'video' group after installation.
2. Use the BIOS to find a stable over-clock for the CPU. Usually the recommended turbo frequency is a good place to start. This step is optional, but will help system performance when it comes to streaming high rates from the USRP. Do not adjust higher over-clock settings without doing research.
3. Use the BIOS to enable boot-on-power. The computer should come back online when power is restored after an outage. This setting is typically referred to as *Restore on AC/Power Loss*
4. Use cpupower to ungovern the CPU and run at the max frequency. This should be added to a script that occurs on reboot.
    - cpupower frequency-set -g performance.
5. To verify that the CPU is running at maximum frequency:
    - cpupower frequency-info

6. Use ethtool to set the interface ring buffer size for both rx and tx. This should be added to a script that occurs on reboot for the interface used to connect to the USRPs. This is done to help prevent packet loss when the network traffic exceeds the capacity of the network adapter.
    - ethtool -G eth0 tx 4096 rx 4096.
7. To see that this works as intended, and that it persists across reboots, you can execute the following, which will output the maximums and the current settings.
    - ethtool -g eth0
8. Use the network manager or a line in the reboot script to change the MTU of the interface for the interface used to connect to the USRPs. A larger MTU will reduce the amount of network overhead. An MTU larger than 1500 bytes allows what is known as Jumbo frames, which can use up to 9000 bytes of payload.
    - ip link set eth0 mtu 9000
9. To verify that the MTU was set correctly:
    - ip link show eth0
10. Use sysctl to adjust the kernel network buffer sizes. This should be added to a script that occurs on reboot for the interface used to connect to the USRPs.
    - sysctl -w net.core.rmem_max=50000000
    - sysctl -w net.core.wmem_max=2500000
11. Verify that the kernel network buffer sizes are set:
    - cat /proc/sys/net/core/rmem_max
    - cat /proc/sys/net/core/wmem_max
12. Install tuned. Use tuned-adm (as root) to set the system's performance to network-latency.
    - sudo zypper in tuned
    - su
    - systemctl enable tuned
    - systemctl start tuned
    - tuned-adm profile network-latency
13. To verify the system's new profile:
    - tuned-adm profile_info
14. Add an environment variable called BOREALISPATH that points to the cloned git repository in .bashrc or .profile and re-source the file. For example:
    - export BOREALISPATH=/home/radar/borealis/
    - source .profile
15. Clone the Borealis software to a directory.
    - git clone https://github.com/SuperDARNCanada/borealis.git
    - If Usask, git submodule init && git submodule update. Create symlink *config.ini* in borealis directory and link to the site specific config file.
    - cd ${BOREALISPATH} && ln -svi ${BOREALISPATH}/borealis_config_files/[radarcode]_config.ini config.ini

- If not Usask, use a Usask *config.ini* file as a template or the config file documentation to create your own file in the borealis directory.

16. The Borealis software has a script called *install_radar_deps_opensuse.sh* to help install dependencies. This script has to be run by the root user. This script can be modified to use the package manager of a different distribution. Make sure that the version of CUDA is up to date and supports your card. This script makes an attempt to correctly install Boost and create symbolic links to the Boost libraries the UHD (USRP Hardware Driver) understands. If UHD does not configure correctly, an improper Boost installation or library naming convention is the likely reason.

17. Set up NTP. The *install_radar_deps_opensuse.sh* script already downloads and configures a version of *ntpd* that works with incoming PPS signals on the serial port DCD line. An example configuration of ntp is shown below for */etc/ntp.conf*. These settings use *tick.usask.ca* as a time server, and PPS (via the *127.127.22.0* lines). It also sets up logging daily for all stats types.

```
driftfile /var/log/ntp/ntp.drift

statsdir /var/log/ntp/ntpstats/
logfile /var/log/ntp/ntp_log
logconfig =all
statistics loopstats peerstats clockstats cryptostats protostats rawstats sysstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
filegen cryptostats file cryptostats type day enable
filegen protostats file protostats type day enable
filegen rawstats file rawstats type day enable
filegen sysstats file sysstats type day enable

restrict -4 default kod notrap nomodify nopeer noquery limited
restrict -6 default kod notrap nomodify nopeer noquery limited

restrict 127.0.0.1
restrict ::1

restrict source notrap nomodify noquery

server tick.usask.ca prefer
server 127.127.22.0 minpoll 4 maxpoll 4
fudge 127.127.22.0 time1 0.2 flag2 1 flag3 0 flag4 1

keys /etc/ntp.keys
trustedkey 1
requestkey 1
controlkey 1
```

1. As part of the realtime capabilities, the hdw.dat repo will be cloned to the computer(default will be /usr/local/hdw.dat). The hdw.dat files are also used for radar operation. Create a symbolic link for this radar in the $BOREALISPATH directory.

   - ln -s /usr/local/hdw.dat/hdw.dat.[radarcode] $BOREALISPATH/hdw.dat.[radarcode]

2. Edit /etc/security/limits.conf to add the following line that allows UHD to set thread priority. UHD automatically tries to boost its thread scheduling priority, so it will fail if the user executing UHD doesn't have permission.

   - @users - rtprio 99

3. Assuming all dependencies are resolved, use *scons* to build the system. Use the script called *mode* to change the build environment to debug or release depending on what version of the system should be run. *SCONSFLAGS* variable can be added to *.bashrc/.profile* to hold any flags such as *-j* for parallel builds. For example, run the following:
    - *source mode [release|debug]*
    - If first time building, run *scons -c* to reset project state.
    - *scons* to build
4. Add the Python scheduling script, *start_radar.sh*, to the system boot scripts to allow the radar to follow the schedule.
5. Finally, add the GPS disciplined NTP lines to the root start up script.
    - /sbin/modprobe pps_ldisc && /usr/bin/ldattach 18 /dev/ttyS0 && /usr/local/bin/ntpd
6. Verify that the PPS signal incoming on the DCD line of ttyS0 is properly routed and being received. You'll get two lines every second corresponding to an 'assert' and a 'clear' on the PPS line along with the time in seconds since the epoch.

```
sudo ppstest /dev/pps0
[sudo] password for root:
trying PPS source "/dev/pps0"
found PPS source "/dev/pps0"
ok, found 1 source(s), now start fetching data...
source 0 - assert 1585755247.999730143, sequence: 200 - clear  1585755247.199734241,␣
↪sequence: 249187
source 0 - assert 1585755247.999730143, sequence: 200 - clear  1585755248.199734605,␣
↪sequence: 249188
```

1. For further reading on networking and tuning with the USRP devices, see https://files.ettus.com/manual/page_transport.html and https://kb.ettus.com/USRP_Host_Performance_Tuning_Tips_and_Tricks. Also see http://doc.ntp.org/current-stable/drivers/driver22.html for information about the PPS ntp clock discipline, and the man pages for:
    - tuned
    - cpupower
    - ethtool
    - ip
    - sysctl
    - modprobe
    - ldattach

CHAPTER

THREE

STARTING AND STOPPING THE RADAR

3.1 Manual Start-up

To more easily start the radar, there is a script called *steamed_hams.sh*. The name of this script is a goofy reference to a scene in an episode of The Simpsons in which Principal Skinner claims there is an aurora happening in his house. The script takes two arguments and can be invoked as follows:

- $BOREALISPATH/steamed_hams.sh experiment_name code_environment

An example invocation to run twofsound in release mode would be:

- /home/radar/borealis/steamed_hams.sh twofsound release

Another example invocation running normalscan in debug mode:

- /home/radar/borealis/steamed_hams.sh normalscan debug

The experiment name must match to an experiment in the *experiment* folder, and does not include the *.py* extension. The code environment is the type of compilation environment that was compiled using *scons* such as release, debug, etc. **NOTE** This script will kill the Borealis software if it is currently running, before it starts it anew.

The script will boot all the radar processes in a detached *screen* window that runs in the background. This window can be reattached in any terminal window locally or over ssh (*screen -r*) to track any outputs if needed.

If starting the radar in normal operation according to the schedule, there is a helper script called *start_radar.sh*.

3.2 Automated Start-up

In order to start the radar automatically, the script *start_radar.sh* should be added to a startup script of the Borealis computer. It can also be called manually by the non-root user (typically *radar*). The scheduling Python script, *remote_server.py*, is responsible for automating the control of the radar to follow the schedule, and is started via the *start_radar.sh* script with the appropriate arguments

```
#!/bin/bash

/usr/bin/pkill -9 -f remote_server.py
source $HOME/.profile

NOW=`date +'%Y%m%d %H:%M:%S'`

/usr/bin/nohup /usr/bin/python3 $BOREALISPATH/scheduler/remote_server.py --scd-dir=/home/
↪radar/borealis_schedules --emails-filepath=/home/radar/borealis_schedules/emails.txt >/
↪home/radar/logs/scd.out 2>&1 &
```

(continues on next page)

(continued from previous page)

```
retVal=$?
if [[ $retVal -ne 0 ]]; then
        echo "${NOW} START: Could not start radar." | tee -a /data/borealis_logs/start_
↪stop.log
else
        echo "${NOW} START: Radar processes started." | tee -a /data/borealis_logs/start_
↪stop.log
fi
```

This script should be added to the control computer boot-up scripts so that it generates a new set of scheduled commands.

3.3 Stopping the Radar

There are several ways to stop the Borealis radar. They are ranked here from most acceptable to last-resort:

1. Run the script *stop_radar.sh* from the Borealis project directory. This script kills the scheduling server, removes all entries from the schedule and kills the screen session running the Borealis software modules.
2. While viewing the screen session running the Borealis software modules, type *ctrl-A, ctrl-*. This will kill the screen session and all software modules running within it.
3. Restart the Borealis computer. **NOTE** In a normal circumstance, the Borealis software will start back up again once the computer reboots.
4. Shut down the Borealis computer.

CHAPTER

FOUR

SCHEDULING

Borealis scheduling is made of several components to help automate and reduce overhead in scheduling. The idea here is to have a script that runs locally at the institution which generates new schedules, a cloud syncing service to automatically upload the new schedules to the radar sites, and then a remote script on site that converts the schedules to actual radar commands.

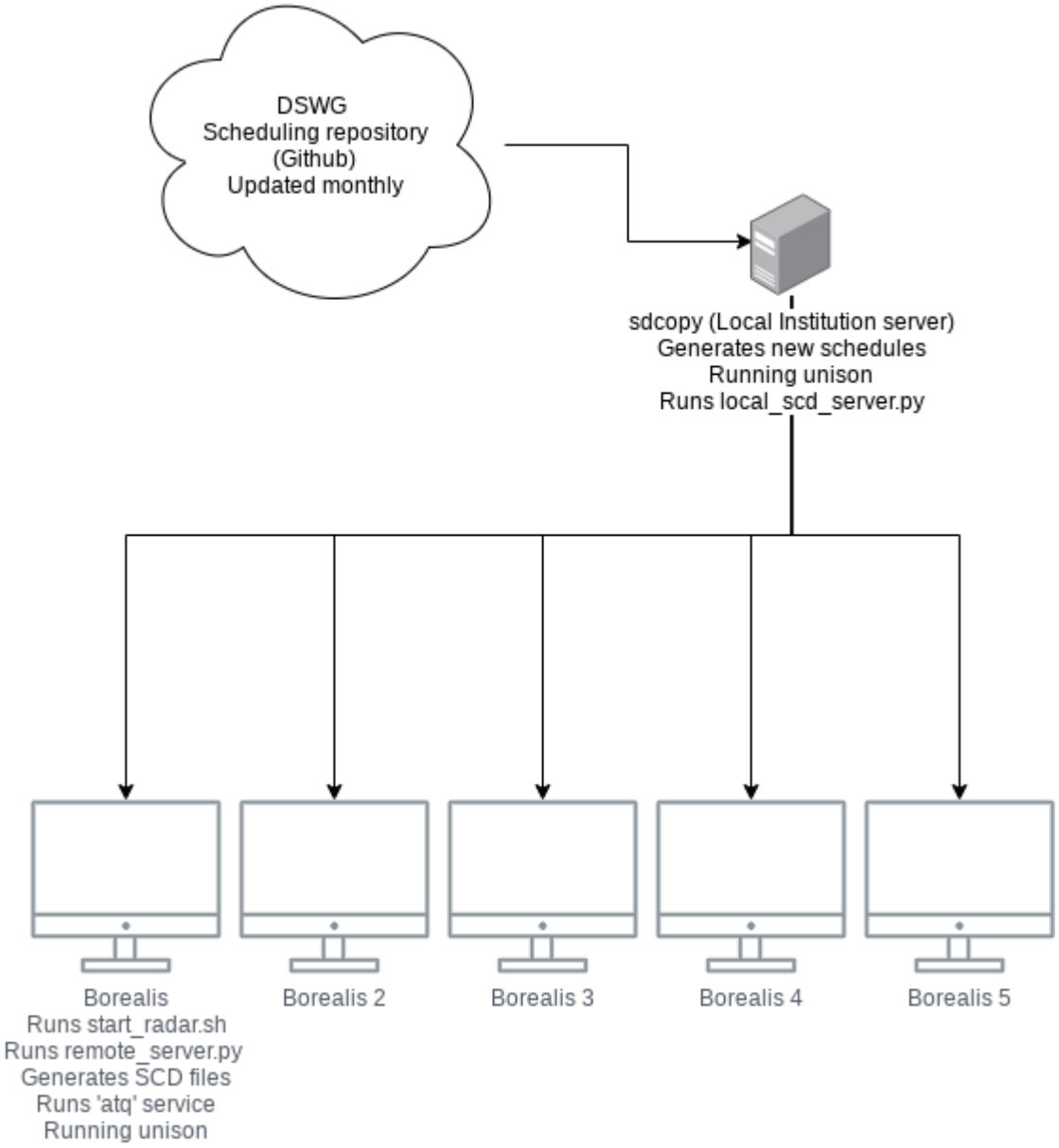
The local script will monitor the Scheduling Working Group (SWG) web link for new uploads and then grab them if there is anything new. At the time of writing, these files are hosted at https://github.com/SuperDARN/schedules. This automated script will then parse the lines from the file and convert them to schedule file (SCD) commands.

The schedule files need to be synced to the radar sites. It is recommended to set up a directory which is cloud shared using a service such as Nextcloud or Owncloud. The SCD files that the local script adds to should all be in this directory so that syncing is all automated.

The remote script will check for changes to any synced files and then generate *at* command arguments for Borealis experiments to run. This allows us to utilize scheduling utilities already available in Linux.

These scripts are configured with logging and email capability so that maintainers can track if scheduling is successful. There is also a utility script called *schedule_modifier.py* that should be used to add or remove lines from the schedule so that no errors are made in the schedule file. It is not recommended to manually modify any schedule files.

Here is a simple diagram for how scheduling works. It starts with the DSWG repository, which is accessed via a local server, which then uses unison to sync with all Borealis radars.



Here are the steps to configure scheduling:

1. Configure a local institution server to build schedules.
    - Configure a cloud/network syncing service such as unison or NFS. Configure this service to share a directory where schedules and logs are to be stored.
    - Git clone a copy of Borealis.
    - Edit the *local_scd_server.py* with the correct experiments and radars belonging to your institution.
    - Configure a system service or reboot *cron* task to run the python3 script *local_scd_server.py* at boot. This script requires the argument *–scd-dir* for the schedules directory as well as *–emails-filepath* which should be a text file of emails on each line where scheduling status will be sent.
    - The *local_scd_server.py* script has an option for running manually the first time to properly configure the scheduling directory with the schedules for the latest files available.

- Example: *python3 ./local_scd_server.py –first-run –scd-dir=/data/borealis_schedules –emails-filepath=/data/borealis_schedules/emails.txt*

2. Configure the Borealis computer.
    - unison will execute on the remote and connect to this machine to sync.
    - Schedule a reboot task via *cron* to run the *start_radar.sh* helper script in order to run the radar according the radar schedule.
    - Enable and start *atq* service.

CHAPTER

FIVE

BUILDING AN EXPERIMENT

Borealis has an extensive set of features and this means that experiments can be designed to be very simple or very complex. To help organize writing of experiments, we've designed the system so that experiments can be broken into smaller components, called slices, that interface together with other components to perform desired functionality. An experiment can have a single slice or several working together, depending on the complexity.

Each slice contains the information needed about a specific pulse sequence to run. The parameters of a slice contain features such as pulse sequence, frequency, fundamental time lag spacing, etc. These are the parameters that researchers will be familiar with. Each slice can be an experiment on its own, or can be just a piece of a larger experiment.

5.1 Introduction to Borealis Slices

Slices are software objects made for the Borealis system that allow easy integration of multiple modes into a single experiment. Each slice could be an experiment on its own, and averaged products are produced from each slice individually. Slices can be used to create separate frequency channels, separate pulse sequences, separate beam scanning order, etc. that can run simultaneously. Slices can be interfaced in four different ways.

The following parameters are unique to a slice:

- tx or rx frequency
- pulse sequence
- tau spacing (mpinc)
- pulse length
- number of range gates
- first range gate
- beam directions
- beam order

A slice is defined using a dictionary and the necessary slice keys. For a complete list of keys that can be used in a slice, see below 'Slice Keys'.

The other necessary part of an experiment is specifying how slices will interface with each other. Interfacing in this case refers to how these two components are meant to be run. To understand the interfacing, lets first understand the basic building blocks of a SuperDARN experiment. These are:

Sequence (integration)

Made up of pulses with a specified spacing, at a specified frequency, and with a specified receive time following the transmission (to gather information from the number of ranges specified). Researchers might be familiar with a common

SuperDARN 7 or 8 pulse sequence design. The sequence definition here is the time to transmit one sequence and the time for receiving echoes from that sequence.

Averaging period (integration time)

A time where the sequences are repeated to gather enough information to average and reduce the effect of spurious emissions on the data. These are defined by either number of sequences, or a length of time during which as many sequences as possible are transmitted. For example, researchers may be familiar with the standard 3 second averaging period in which ~30 pulse sequences are sent out and received in a single beam direction.

Scan

A time where the averaging periods are repeated, traditionally to look in different beam directions with each averaging period. A scan is defined by the number of beams or integration times.

5.2 Interfacing Types Between Slices

Knowing the basic building blocks of a SuperDARN-style experiment, the following types of interfacing are possible, arranged from highest level to lowest level:

1. **SCAN**

   The scan by scan interfacing allows for slices to run a scan of one slice, followed by a scan of the second. The scan mode of interfacing typically means that the slice will cycle through all of its beams before switching to another slice.

   There are no requirements for slices interfaced in this manner.

2. **INTTIME**

   This type of interfacing allows for one slice to run its integration period (also known as integration time or averaging period), before switching to another slice's integration period. This type of interface effectively creates an interleaving scan where the scans for multiple slices are run 'at the same time', by interleaving the integration times.

   Slices which are interfaced in this manner must share:

   - the same SCANBOUND value.

3. **INTEGRATION**

   Integration interfacing allows for pulse sequences defined in the slices to alternate between each other within a single integration period. It's important to note that data from a single slice is averaged only with other data from that slice. So in this case, the integration period is running two slices and can produce two averaged datasets, but the sequences (integrations) within the integration period are interleaved.

   Slices which are interfaced in this manner must share:

   - the same SCANBOUND value.
   - the same INTT or INTN value.
   - the same BEAM_ORDER length (scan length)

4. **PULSE**

   Pulse interfacing allows for pulse sequences to be run together concurrently. Slices will have their pulse sequences layered together so that the data transmits at the same time. For example, slices of different frequencies can be mixed simultaneously, and slices of different pulse sequences can also run together at the cost of having more blanked samples. When slices are interfaced in this way the radar is truly transmitting and receiving the slices simultaneously.

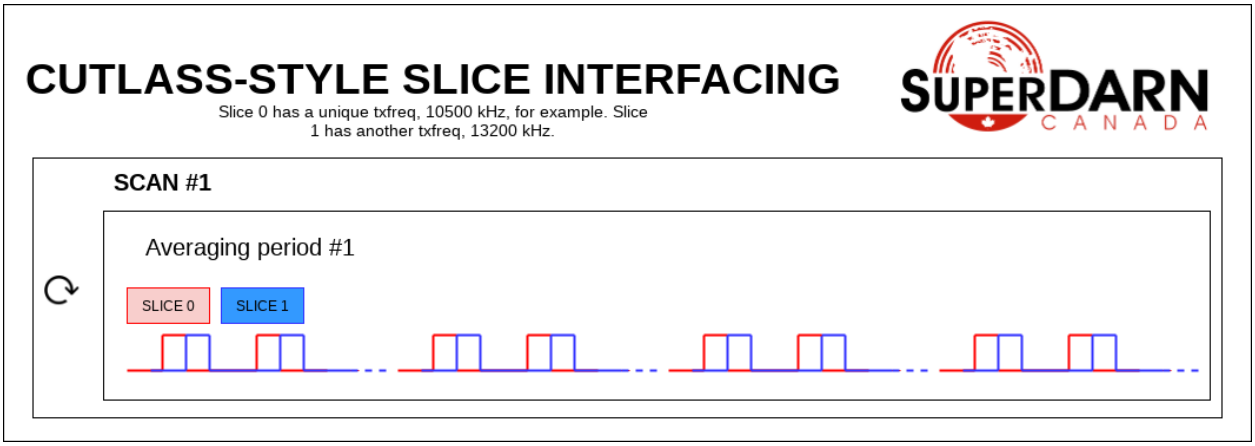
Slices which are interfaced in this manner must share:

- the same SCANBOUND value.
- the same INTT or INTN value.
- the same BEAM_ORDER length (scan length)

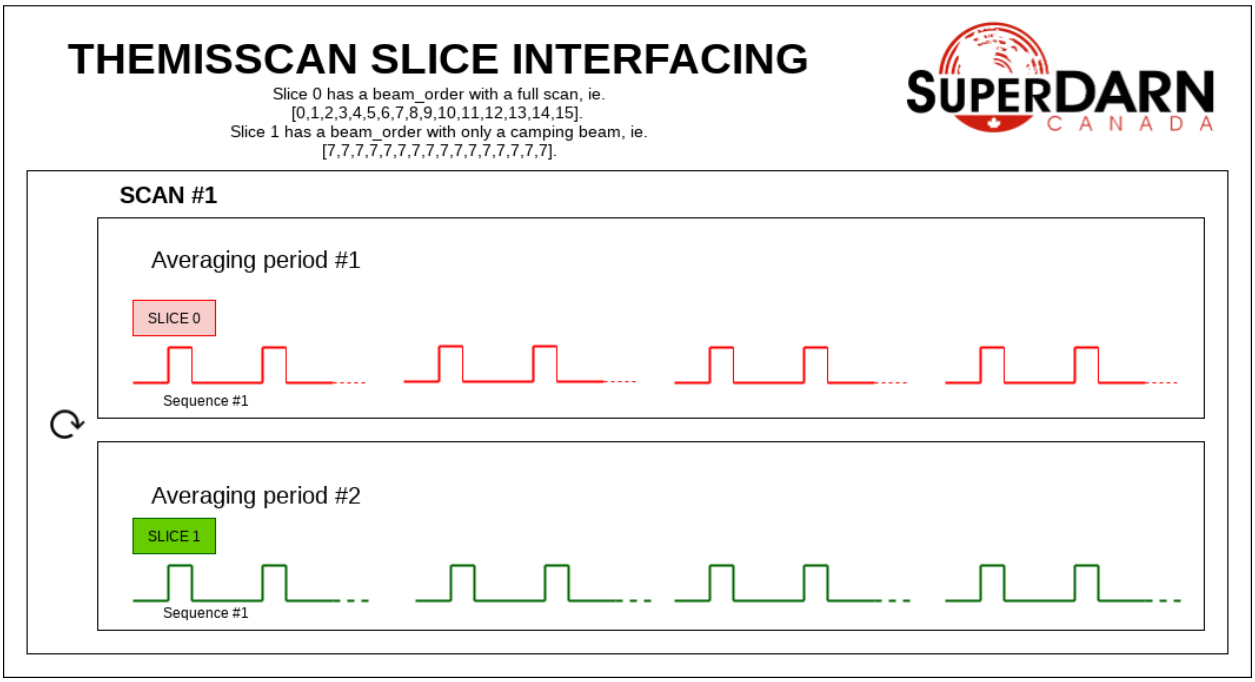
5.3 Slice Interfacing Examples

Let's look at some examples of common experiments that can easily be separated into multiple slices.

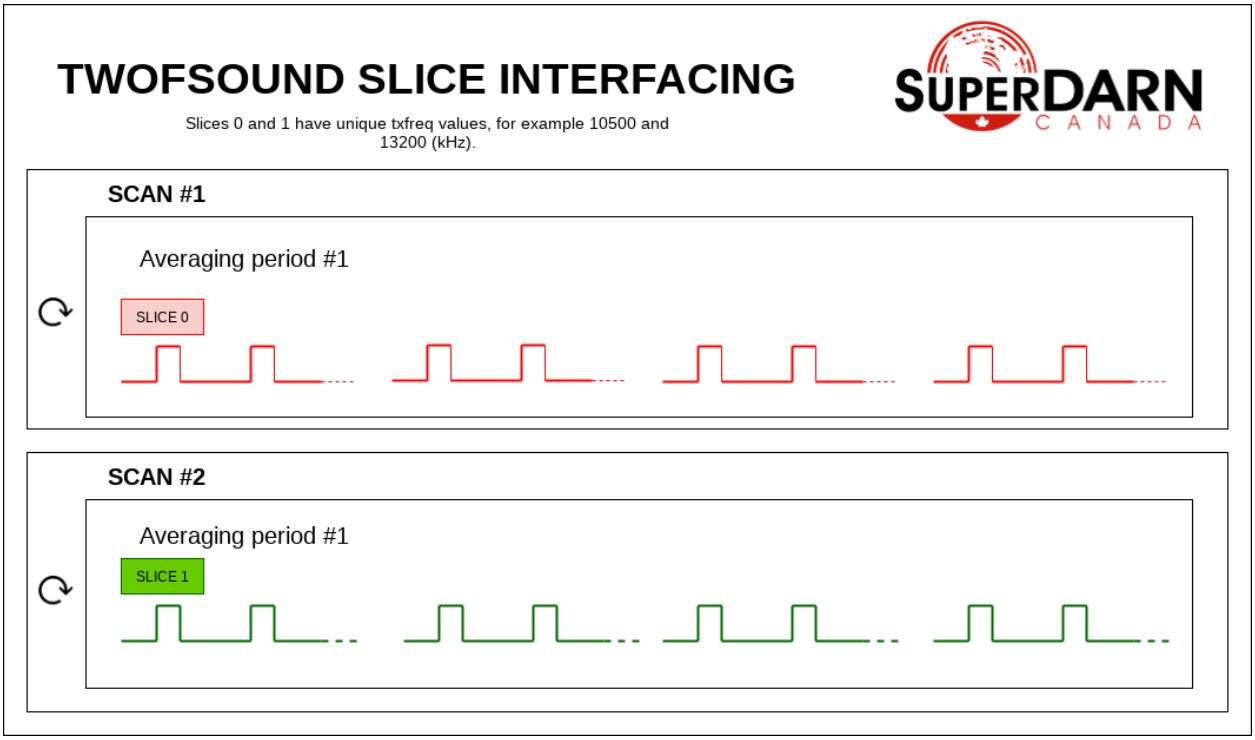
In a CUTLASS-style experiment, the pulse in the sequence is actually two pulses of differing transmit frequency. This is a 'quasi'-simultaneous multi-frequency experiment where the frequency changes in the middle of the pulse. To build this experiment, two slices can be PULSE interfaced. The pulses from both slices are combined into a single set of transmitted samples for that sequence and samples received from those sequences are used for both slices (filtering the raw data separates the frequencies).



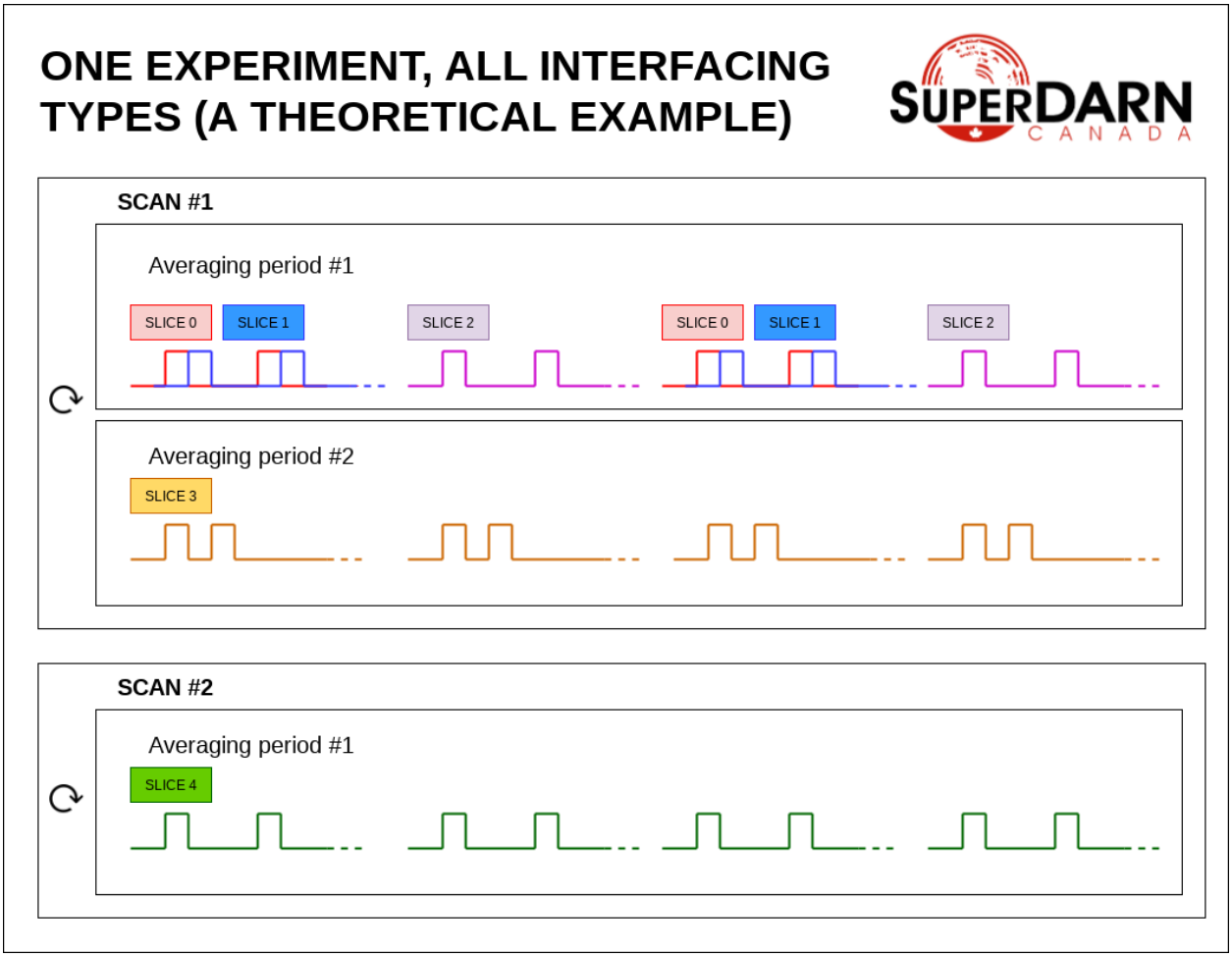
In a themisscan experiment, a single beam is interleaved with a full scan. The beam_order can be unique to different slices, and these slices could be INTTIME interfaced to separate the camping beam data from the full scan, if desired. With INTTIME interfacing, one averaging period of one slice will be followed by an averaging period of another, and so on. The averaging periods are interleaved. The resulting experiment runs beams 0, 7, 1, 7, etc.



In a twofsound experiment, a full scan of one frequency is followed by a full scan of another frequency. The txfreq are unique between the slices. In this experiment, the slices are SCAN interfaced. A full scan of slice 0 runs followed by a full scan of slice 1, and then the process repeats.



Here's a theoretical example showing all types of interfacing. In this example, slices 0 and 1 are PULSE interfaced. Slices 0 and 2 are INTEGRATION interfaced. Slices 0 and 3 are INTTIME interfaced. Slices 0 and 4 are SCAN interfaced.



5.3.1 Writing an Experiment

All experiments must be written as their own class and must be built off of the built-in ExperimentPrototype class.

This means the ExperimentPrototype class must be imported at the start of the experiment file:

```
from experiments.experiment_prototype import ExperimentPrototype
```

5.4 Experiment-Wide Attributes

cpid ***required***
: The only experiment-wide attribute that is required to be set by the user when initializing is the CPID, or control program identifier. This should be unique to the experiment. You will need to request this from your institution's radar operator. You should clearly document the name of the experiment and some operating details that correspond to the CPID.

output_rx_rate ***defaults***
: The sampling rate of the output data. The default is 10.0e3/3 Hz, or 3.333 kHz.

rx_bandwidth ***defaults***
: The sampling rate of the USRPs (before decimation). The default is 5.0e6 Hz, or 5 MHz.

tx_bandwidth *defaults*
The output sampling rate of the transmitted signal. The default is 5.0e6 Hz, or 5 MHz.

txctrfreq *defaults*
The center frequency of the transmit chain. The default is 12000.0 kHz, or 12 MHz. Note that this is tuned so will be set to a quantized value, which in general is not exactly 12 MHz, and the value can be accessed by the user at this attribute after the experiment begins.

rxctrfreq *defaults*
The center frequency of the receive chain. The default is 12000.0 kHz, or 12 MHz. Note that this is tuned so will be set to a quantized value, which in general is not exactly 12 MHz, and the value can be accessed by the user at this attribute after the experiment begins.

decimation_scheme *defaults*
The decimation scheme for the experiment, provided by an instance of the class DecimationScheme. There is a default scheme specifically set for the default rates and center frequencies above.

comment_string *defaults*
A comment string describing the experiment. It is highly encouraged to provide some description of the experiment for the output data files. The default is ‘’, or an empty string.

Below is an example of properly inheriting the prototype class and defining your own experiment:

```
class MyClass(ExperimentPrototype):

    def __init__(self):
        cpid = 123123  # this must be a unique id for your control program.
        super(MyClass, self).__init__(cpid,
            comment_string='My experiment explanation')
```

The experiment handler will create an instance of your experiment when your experiment is scheduled to start running. Your class is a child class of ExperimentPrototype and because of this, the parent class needs to be instantiated when the experiment is instantiated. This is important because the experiment_handler will build the scans required by your class in a way that is easily readable and iterable by the radar control program. This is done by methods that are set up in the ExperimentPrototype parent class.

The next step is to add slices to your experiment. An experiment is defined by the slices in the class, and how the slices interface. As mentioned above, slices are just dictionaries, with a preset list of keys available to define your experiment. The keys that can be used in the slice dictionary are described below.

5.5 Slice Keys

These are the keys that are set by the user when initializing a slice. Some are required, some can be defaulted, and some are set by the experiment and are read-only.

Slice Keys Required by the User

pulse_sequence *required*
The pulse sequence timing, given in quantities of tau_spacing, for example normalscan = [0, 14, 22, 24, 27, 31, 42, 43].

tau_spacing *required*
multi-pulse increment in us, Defines minimum space between pulses.

pulse_len *required*
length of pulse in us. Range gate size is also determined by this.

num_ranges ***required***
Number of range gates.

first_range ***required***
distance to the first range gate, in km

intt ***required or intn required***
duration of an integration, in ms. (maximum)

intn ***required or intt required***
number of averages to make a single integration, only used if intt = None.

beam_angle ***required***
list of beam directions, in degrees off azimuth. Positive is E of N. The beam_angle list length = number of beams. Traditionally beams have been 3.24 degrees separated but we don't refer to them as beam -19.64 degrees, we refer as beam 1, beam 2. Beam 0 will be the 0th element in the list, beam 1 will be the 1st, etc. These beam numbers are needed to write the beam_order list. This is like a mapping of beam number (list index) to beam direction off boresight. Typically you can use the radar's common beam angle list. For example, at Saskatoon site the beam angles are a standard 16-beam list: [-26.25, -22.75, -19.25, -15.75, -12.25, -8.75,

-5.25, -1.75, 1.75, 5.25, 8.75, 12.25, 15.75, 19.25, 22.75, 26.25]

beam_order ***required***
beam numbers written in order of preference, one element in this list corresponds to one integration period. Can have lists within the list, resulting in multiple beams running simultaneously in the averaging period, so imaging. A beam number of 0 in this list gives us the direction of the 0th element in the beam_angle list. It is up to the writer to ensure their beam pattern makes sense. Typically beam_order is just in order (scanning W to E or E to W, ie. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. You can list numbers multiple times in the beam_order list, for example [0, 1, 1, 2, 1] or use multiple beam numbers in a single integration time (example [[0, 1], [3, 4]], which would trigger an imaging integration. When we do imaging we will still have to quantize the directions we are looking in to certain beam directions.

clrfrqrange ***required or txfreq or rxfreq required***
range for clear frequency search, should be a list of length = 2, [min_freq, max_freq] in kHz. **Not currently supported.**

txfreq ***required or clrfrqrange or rxfreq required***
transmit frequency, in kHz. Note if you specify clrfrqrange it won't be used.

rxfreq ***required or clrfrqrange or txfreq required***
receive frequency, in kHz. Note if you specify clrfrqrange or txfreq it won't be used. Only necessary to specify if you want a receive-only slice.

Defaultable Slice Keys

acf ***defaults***
flag for rawacf and generation. The default is False. If True, the following fields are also used: - averaging_method (default 'mean') - xcf (default True if acf is True) - acfint (default True if acf is True) - lagtable (default built based on all possible pulse combos)

acfint ***defaults***
flag for interferometer autocorrelation data. The default is True if acf is True, otherwise False.

averaging_method ***defaults***
a string defining the type of averaging to be done. Current methods are 'mean' or 'median'. The default is 'mean'.

comment ***defaults***
a comment string that will be placed in the borealis files describing the slice. Defaults to empty string.

lag_table ***defaults***
used in acf calculations. It is a list of lags. Example of a lag: [24, 27] from 8-pulse normalscan. This defaults

to a lagtable built by the pulse sequence provided. All combinations of pulses will be calculated, with both the first pulses and last pulses used for lag-0.

pulse_phase_offset ***defaults***
Allows phase shifting between pulses, enabling encoding of pulses. Default all zeros for all pulses in pulse_sequence.

range_sep ***defaults***
a calculated value from pulse_len. If already set, it will be overwritten to be the correct value determined by the pulse_len. This is the range gate separation, in azimuthal direction, in km.

rx_int_antennas ***defaults***
The antennas to receive on in interferometer array, default is all antennas given max number from config.

rx_main_antennas ***defaults***
The antennas to receive on in main array, default is all antennas given max number from config.

scanbound ***defaults***
A list of seconds past the minute for integration times in a scan to align to. Defaults to None, not required. If you set this, you will want to ensure that there is a slightly larger amount of time in the scan boundaries than the integration time set for the slice. For example, if you want to align integration times at the 3n second marks, you may want to have a set integration time of ~2.9s to ensure that the experiment will start on time. Typically 50ms difference will be enough. This is especially important for the last integration time in the scan, as the experiment will always wait for the next scan start boundary (potentially causing a minute of downtime). You could also just leave a small amount of downtime at the end of the scan.

seqoffset ***defaults***
offset in us that this slice's sequence will begin at, after the start of the sequence. This is intended for PULSE interfacing, when you want multiple slice's pulses in one sequence you can offset one slice's sequence from the other by a certain time value so as to not run both frequencies in the same pulse, etc. Default is 0 offset.

tx_antennas ***defaults***
The antennas to transmit on, default is all main antennas given max number from config.

xcf ***defaults***
flag for cross-correlation data. The default is True if acf is True, otherwise False.

Read-only Slice Keys

clrfrqflag ***read-only***
A boolean flag to indicate that a clear frequency search will be done. **Not currently supported.**

cpid ***read-only***
The ID of the experiment, consistent with existing radar control programs. This is actually an experiment-wide attribute but is stored within the slice as well. This is provided by the user but not within the slice, instead when the experiment is initialized.

rx_only ***read-only***
A boolean flag to indicate that the slice doesn't transmit, only receives.

slice_id ***read-only***
The ID of this slice object. An experiment can have multiple slices. This is not set by the user but instead set by the experiment automatically when the slice is added. Each slice id within an experiment is unique. When experiments start, the first slice_id will be 0 and incremented from there.

slice_interfacing ***read-only***
A dictionary of slice_id : interface_type for each sibling slice in the experiment at any given time.

Not currently supported and will be removed

wavetype ***defaults***
string for wavetype. The default is SINE. **Not currently supported.**

iwavetable *defaults*
: a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

qwavetable *defaults*
: a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

5.6 Experiment Example

An example of adding a slice to your experiment is as follows:

```
self.add_slice({  # slice_id will be 0, there is only one slice.
    "pulse_sequence": [0, 9, 12, 20, 22, 26, 27],
    "tau_spacing": tau_spacing,  # us
    "pulse_len": 300,  # us
    "num_ranges": 75,  # range gates
    "first_range": 180,  # first range gate, in km
    "intt": 3500,  # duration of an integration, in ms
    "beam_angle": [-26.25, -22.75, -19.25, -15.75, -12.25, -8.75,
                   -5.25, -1.75, 1.75, 5.25, 8.75, 12.25, 15.75, 19.25, 22.75,
                   26.25],
    "beam_order": [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
    "scanbound": [i * 3.5 for i in range(len(beams_to_use))], #1 min scan
    "txfreq" : 10500, #kHz
    "acf": True,
    "xcf": True,  # cross-correlation processing
    "acfint": True,  # interferometer acfs
})

self.add_slice(slice_1)
```

This slice would be assigned with slice_id = 0 if it's the first slice added to the experiment. The experiment could also add another slice:

```
slice_2 = copy.deepcopy(slice_1)
slice_2['txfreq'] = 13200 #kHz
slice_2['comment'] = 'This is my second slice.'

self.add_slice(slice_2, interfacing_dict={0: 'SCAN'})
```

Notice that you must specify interfacing to an existing slice when you add a second or greater order slice to the experiment. To see the types of interfacing that can be used, see above section 'Interfacing Types Between Slices'.

This experiment is very similar to the twofsound experiment. To see examples of common experiments, look at *experiments package*.

CHAPTER

SIX

CONFIG PARAMETERS

Config field	Example entry
site_id	sas
gps_octoclock_addr	addr=192.168.10.131
devices	recv_frame_size=4000, addr0=192.168.10.100, addr1=192.168.10.101, addr2=192.168.1
main_antenna_count	16
interferometer_antenna_count	4
main_antenna_usrp_rx_channels	0,2,4,6,8,10,12,14,16, 18,20,22,24,26,28,30
interferometer_antenna_usrp_rx_channels	1,3,5,7
main_antenna_usrp_tx_channels	0,1,2,3,4,5,6,7,8,9, 10,11,12,13,14,15
main_antenna_spacing	15.24
interferometer_antenna_spacing	15.24
min_freq	8.00E+06
max_freq	20.00E+06
minimum_pulse_length	100
minimum_mpinc_length	1
minimum_pulse_separation	125
tx_subdev	A:A
max_tx_sample_rate	5.00E+06
main_rx_subdev	A:A A:B
interferometer_rx_subdev	A:A A:B
max_rx_sample_rate	5.00E+06
pps	external
ref	external
overthewire	sc16
cpu	fc32
gpio_bank	RXA
atr_rx	0x0006
atr_tx	0x0018
atr_xx	0x0060
atr_0x	0x0180
tst_md	0x0600
lo_pwr	0x1800
agc_st	0x6000
max_usrp_dac_amplitude	0.99
pulse_ramp_time	1.00E-05
tr_window_time	6.00E-05
agc_signal_read_delay	0
usrp_master_clock_rate	1.00E+08

Config field	Example entry
max_output_sample_rate	1.00E+05
max_number_of_filter_taps_per_stage	2048
router_address	tcp://127.0.0.1:6969
radctrl_to_exphan_identity	RADCTRL_EXPHAN_IDEN
radctrl_to_dsp_identity	RADCTRL_DSP_IDEN
radctrl_to_driver_identity	RADCTRL_DRIVER_IDEN
radctrl_to_brian_identity	RADCTRL_BRIAN_IDEN
radctrl_to_dw_identity	RADCTRL_DW_IDEN
driver_to_radctrl_identity	DRIVER_RADCTRL_IDEN
driver_to_dsp_identity	DRIVER_DSP_IDEN
driver_to_brian_identity	DRIVER_BRIAN_IDEN
exphan_to_radctrl_identity	EXPHAN_RADCTRL_IDEN
exphan_to_dsp_identity	EXPHAN_DSP_IDEN
dsp_to_radctrl_identity	DSP_RADCTRL_IDEN
dsp_to_driver_identity	DSP_DRIVER_IDEN
dsp_to_exphan_identity	DSP_EXPHAN_IDEN
dsp_to_dw_identity	DSP_DW_IDEN
dspbegin_to_brian_identity	DSPBEGIN_BRIAN_IDEN
dspend_to_brian_identity	DSPEND_BRIAN_IDEN
dw_to_dsp_identity	DW_DSP_IDEN
dw_to_radctrl_identity	DW_RADCTRL_IDEN
brian_to_radctrl_identity	BRIAN_RADCTRL_IDEN
brian_to_driver_identity	BRIAN_DRIVER_IDEN
brian_to_dspbegin_identity	BRIAN_DSPBEGIN_IDEN
brian_to_dspend_identity	BRIAN_DSPEND_IDEN
ringbuffer_name	data_ringbuffer
ringbuffer_size_bytes	200.00E+06
data_directory	/data/borealis_data

CHAPTER

SEVEN

BOREALIS PROCESSES

7.1 Runtime Processes

7.1.1 experiment_handler package

The experiment_handler package contains a single module, experiment_handler, that is a standalone program.

experiment_handler process

This program runs a given experiment. It will use the experiment's build_scans method to create the iterable ScanClassBase objects that will be used by the radar_control block, then it will pass the experiment to the radar_control block to run.

It will be passed some data to use in its update method at the end of every integration time. This has yet to be implemented but will allow experiment_prototype to modify itself based on received data as feedback.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

`experiment_handler.experiment_handler.`**`experiment_handler`**`(`*semaphore*`)`

Run the experiment. This is the main process when this program is called.

This process runs the experiment from the module that was passed in as an argument. It currently does not exit unless killed. It may be updated in the future to exit if provided with an error flag.

This process begins with setup of sockets and retrieving the experiment class from the module. It then waits for a message of type RadarStatus to come in from the radar_control block. If the status is 'EXPNEEDED', meaning an experiment is needed, experiment_handler will build the scan iterable objects (of class ScanClassBase) and will pass them to radar_control. Other statuses will be implemented in the future.

In the future, the update method will be implemented where the experiment can be modified by the incoming data.

`experiment_handler.experiment_handler.`**`experiment_parser`**`()`

Creates the parser to retrieve the experiment module.

Returns
parser, the argument parser for the experiment_handler.

`experiment_handler.experiment_handler.`**`printing`**`(`*msg*`)`

experiment_handler.experiment_handler.**retrieve_experiment**(*experiment_module_name*)

Retrieve the experiment class from the provided module given as an argument.

Parameters
experiment_module_name – The name of the experiment module to run from the Borealis project's experiments directory.

Raises
ExperimentException – if the experiment module provided as an argument does not contain a single class that inherits from ExperimentPrototype class.

Returns
Experiment, the experiment class, inherited from ExperimentPrototype.

experiment_handler.experiment_handler.**send_experiment**(*exp_handler_to_radar_control*, *iden*, *serialized_exp*)

Send the experiment to radar_control module.

Parameters

- **exp_handler_to_radar_control** – socket to send the experiment on
- **iden** – ZMQ identity
- **serialized_exp** – Either a pickled experiment or a None.

experiment_handler.experiment_handler.**usage_msg**()

Return the usage message for this process.

This is used if a -h flag or invalid arguments are provided.

Returns
the usage message

Usage

7.1.2 radar_control package

The radar_control package contains a single module, radar_control, that is a standalone program.

7.1.3 Brian

Brian is an administrator process for Borealis. It acts as a router for all messages in the system and it is responsible for controlling the flow of logic. This process was originally called Brain, but after repeated misspellings, the name Brian stuck.

Brian implements a ZMQ router in order for all the other processes to connect. Using a ZMQ router lets us to use a feature of ZMQ for named sockets. The premise of named sockets is that we can connect a single router address, and when we connect we can supply a name for the socket we are connecting from. ZMQ's router process will then automatically know how to send data to that socket if another socket sends to the identity instead of an address. This makes following the flow of messages much easier to track. By having all messages flow through a router, its possible to log the flow of data between sockets to make sure that the pipeline of messages is occuring in the correct order, and if not it is a helpful tool in debugging.

Brian is also responsible for rate controlling the system. Since all the traffic routes through this module, it is an ideal place to make sure that the pipeline isn't being overwhelmed by any modules. This step is very important to make sure that the GPU processing isn't being overloaded with work or that too many new requests enter the USRP driver.

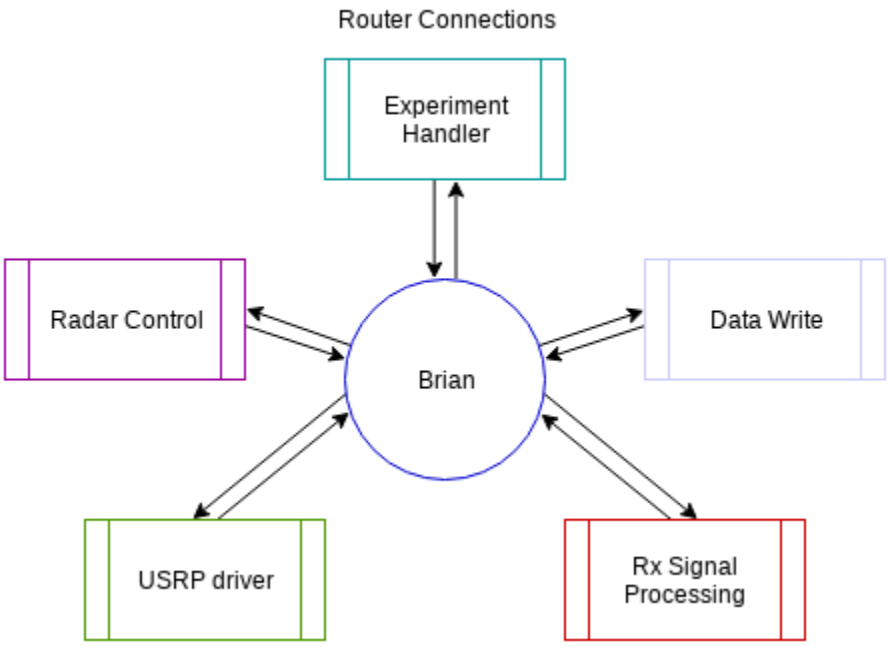


Fig. 1: Block diagram of ZMQ connections

7.1.4 Rx Signal Processing

The Borealis radar receive side signal processing is mostly moved into software using the digital radar design. The RX DSP block is designed to utilize a GPU and threading in order to maximize parallelism to be able to process as much data as possible in real-time.

Borealis experiments give lots of flexibility for filtering. Filter coefficients are generated as part of decimation schemes at the experiment level. The DSP block receives the coefficients for each stage of decimation from Radar Control. The DSP block has been designed to be able to run as many decimation stages as are configured in the decimation scheme. This allows SuperDARN users to have as much control as they want in designing filter characteristics.

Sampled data stored in shared memory is then opened, and operation of the GPU is configured. The GPU programming is set up in an asynchronous mode, meaning that more than one stream can run at once. The GPU does not have enough computation resources to be able to process data from more than one sequence, but in asynchronous mode data from one sequence can be copied to the GPU memory while another sequence is being processed. Asynchronous mode also allows for a callback function that executes when the stream is finished executing without interrupting operation of the main thread. GPU operations works as follows:

1. Memory is allocated on device to hold data for each stage of decimation.
2. Parallelized filtering convolution calculations are performed for each stage.
3. A callback function is run once the GPU is finished processing.

The GPU stream callback runs in a new thread and copies the processed samples back to the host machine. The processed samples are then sent to another process to be written to file. A final deconstructor is run that frees all associated memory resources for the completed sequence.

Filtering convolutions involve many multiply and add operations over a set of data, and many of these operations can be run concurrently. Key to understanding how GPU parallel processing occurs requires reading and studying the CUDA programming guide. In this application, two different kernels are used.

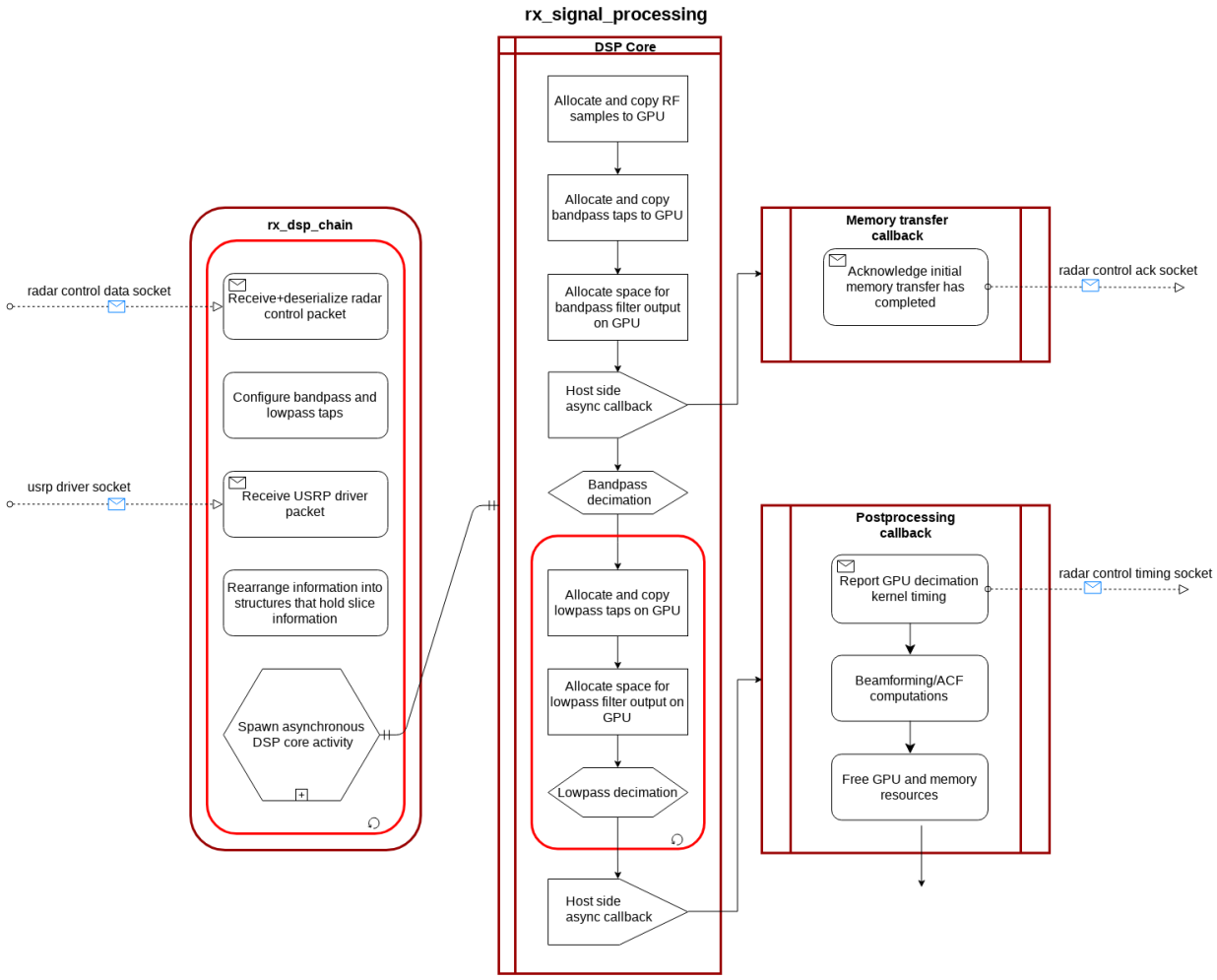


Fig. 2: Block diagram of RX DSP software

A bandpass filtering CUDA kernel is written to perform the convolutions and the GPU is configured with a two dimensional grid of two dimension blocks. In this case there is a single input data set, and one output data set for each frequency. The grid has a block for each decimated output sample by number of antennas. Each block is made up of set of threads and there is a thread for each filter coefficient by number of total filters. What this all means is that the GPU will attempt to process as many output samples that the device can run at once. Each output sample calculation will have all the multiplies and adds done concurrently for the filters for each frequency.

A lowpass filtering CUDA kernel is similar in operation, however since there is now potentially one or more data sets that get individually reduced, the kernel dimensions get slightly changed. The grid now adds a third dimension for frequency data set and the block now only has one set of threads for a single lowpass filter.

Another representation of Frerking's method

Frerking's method is found in Frerking, M. E., *Digital Signal Processing in Communications Systems*, Chapman & Hall, 1994, pp. 171-174. It is a method for creating a frequency-translating FIR filter by translating the filter coefficients to a bandpass filter and then convolving with the input samples (to simultaneously mix to baseband and decimate). The method involves creating multiple bandpass filters so as to maintain the linear phase property of the FIR filter. The number of bandpass filters (sets of coefficients) required is defined as P, and this value is also, therefore, the number of unique ϕ as shown below. The method can really be defined as doing the following:

$$b_k[n] = h[n]e^{j(\phi_k + 2\pi n \frac{f}{F_s})}$$

where b_k are the bandpass filters from $k = 0$ to $k = P$. $h[n]$ is the original low pass filter coefficient set of length N, f is the translation frequency, and F_s is the input sampling frequency. ϕ_k is the starting phase of the NCO (numerically controlled oscillator) being multiplied element by element with the low pass filter where

$$\phi_k = 2\pi Rk\frac{f}{F_s}$$

and where the minimum integer value P is determined by the equation given by Frerking:

$$PR\frac{f}{F_s} = int, \;\; 1 \leq P \leq F_s$$

where R is the integer decimation rate. The maximum value of P would then be F_s, assuming f and F_s are integers.

Then, to filter and decimate,

$$y[m] = y[Rl] = \sum_{n=0}^{N} x[Rl - n]b_{(n \bmod P)}[n]$$

where $y[m]$ is each baseband decimated sample, and $x[l]$ is the input samples. By decimation, the output number of samples, $M = \frac{L}{R}$ where L is the input number of samples (although to avoid zero-padding for convolution, $M < \frac{L}{R}$).

Our new sampling rate will be

$$F_{new} = \frac{F_s}{R}$$

However, by using a single bandpass filter, a new method could be used. The starting phase of the NCO on the filter coefficient set is pulled out from the sum, and then phase correction is done on the decimated samples after the convolution step.

$$b[n] = h[n]e^{j(2\pi n \frac{f}{F_s})}$$

$$y[m] = y[Rl] = e^{j\phi_k} \sum_{n=0}^{N} x[Rl - n]b[n], \;\; k = m \bmod P$$

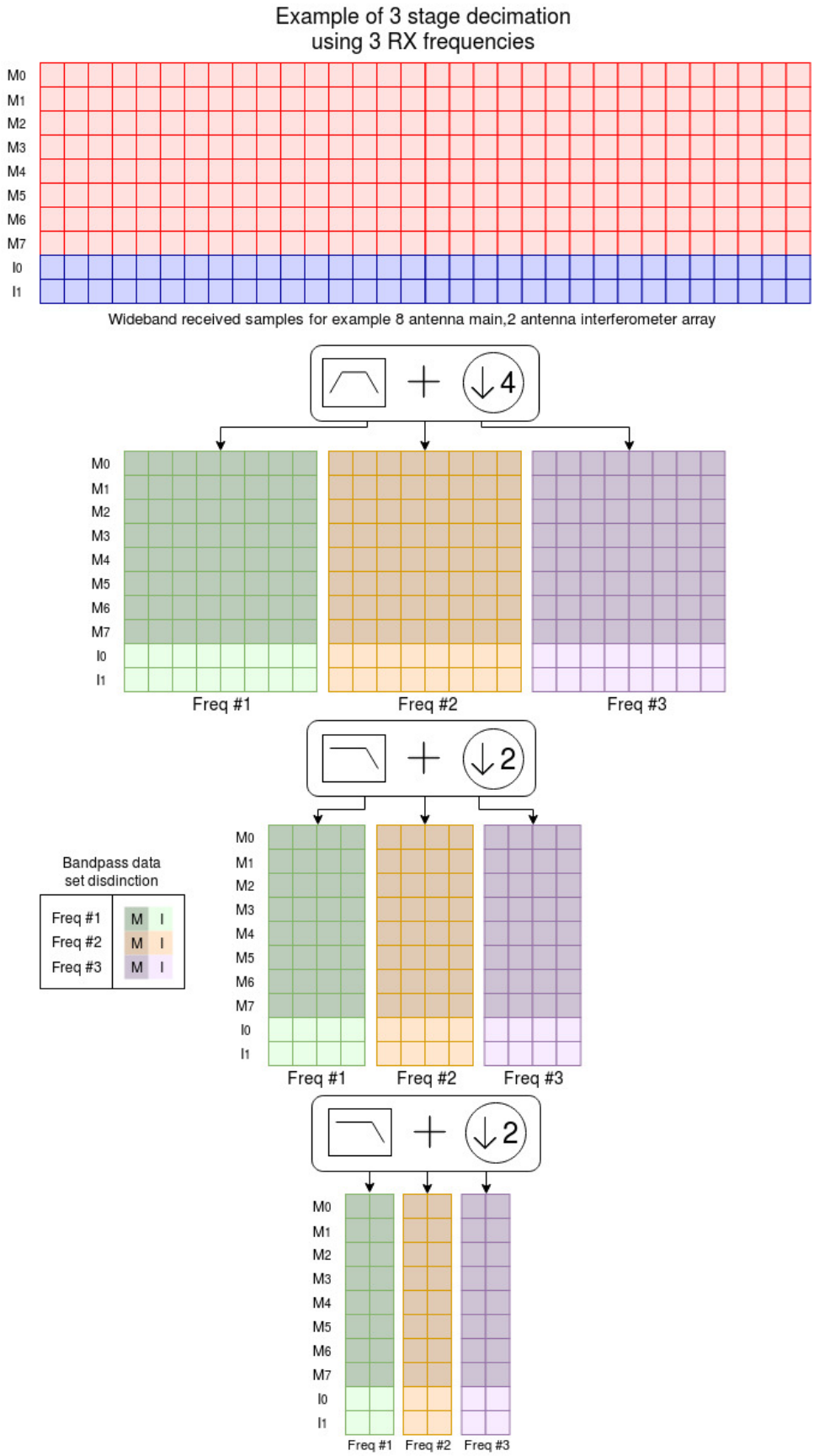


Fig. 3: Diagram of Rx DSP data flow during decimation

Both methods are equivalent:

$$e^{j\phi_k}\sum_{n=0}^{N}x[Rl-n]h[n]e^{j(2\pi n\frac{f}{F_s})}=\sum_{n=0}^{N}x[Rl-n]h[n]e^{j(\phi_k+2\pi n\frac{f}{F_s})}$$

Frerking's method requires NP multiplications before convolution, and for it to be most computationally efficient, it requires storing P sets of N coefficients. For a small value of P and a large value of M output samples, the number of multiplications would be minimized by this method. However, the worst case for using Frerking's method is a large value of F_s, $M \geq F_s$, and an unknown f, meaning that the storage requirements would be for $P = F_s$ number of sets of filter coefficients.

For the case when there exists a small value of M or a large value of P or N, the new modified method might be more computationally efficient, as $N + M - \lfloor \frac{M}{P} \rfloor$ multiplications are required in this method. However, the new method is more memory efficient in all cases where $P > 1$ because only one set of filter coefficients is required to be stored in all cases.

For an unknown integer value f and an unknown decimation rate (or where R is not a submultiple of F_s), processing would have to accommodate $P = F_s$, and so Frerking would be optimal where

$$NF_s < N + M - \lfloor \frac{M}{F_s} \rfloor$$

and the new method would be optimal for

$$NF_s > N + M - \lfloor \frac{M}{F_s} \rfloor$$

File dsp.cu

Functions

std::vector<cudaDeviceProp> **get_gpu_properties**()

Gets the properties of each GPU in the system.

Returns
The gpu properties.

void **print_gpu_properties**(std::vector<cudaDeviceProp> gpu_properties)

Prints the properties of each cudaDeviceProp in the vector.

More info on properties and calculations here: https://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/

Parameters
gpu_properties – **[in]** A vector of cudaDeviceProp structs.

File dsp.hpp

Defines

gpuErrchk(ans)

Typedefs

typedef struct *rx_slice* **rx_slice**

Functions

inline void **throw_on_cuda_error**(cudaError_t code, const char *file, int line)

std::vector<cudaDeviceProp> **get_gpu_properties**()

Gets the properties of each GPU in the system.

Returns
The gpu properties.

void **print_gpu_properties**(std::vector<cudaDeviceProp> gpu_properties)

Prints the properties of each cudaDeviceProp in the vector.

More info on properties and calculations here: https://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/

Parameters
gpu_properties – **[in]** A vector of cudaDeviceProp structs.

void **postprocess**(DSPCore *dp)

struct **rx_slice**

#include <dsp.hpp>

Public Functions

inline **rx_slice**(double rx_freq, uint32_t slice_id, uint32_t num_ranges, uint32_t beam_count, float first_range, float range_sep, uint32_t tau_spacing)

Public Members

double **rx_freq**

uint32_t **slice_id**

uint32_t **num_ranges**

uint32_t **beam_count**

float **first_range**

float **range_sep**

uint32_t **tau_spacing**

std::vector<*lag*> **lags**

struct **lag**

#include <dsp.hpp>

Public Functions

inline **lag**(uint32_t pulse_1, uint32_t pulse_2, uint32_t lag_num)

Public Members

uint32_t **pulse_1**

uint32_t **pulse_2**

uint32_t **lag_num**

class **DSPCore**

#include <dsp.hpp> Contains the core DSP work done on the GPU.

Public Functions

void **cuda_postprocessing_callback**(uint32_t total_antennas, uint32_t num_samples_rf, std::vector<uint32_t> samples_per_antenna, std::vector<uint32_t> total_output_samples)

Add the postprocessing callback to the stream.

This function allocates the host space needed for filter stage data and then copies the data from GPU into the allocated space. Certain *DSPCore* members needed for post processing are assigned such as the rx freqs, the number of rf samples, the total antennas and the vector of samples per antenna(each stage).

void **initial_memcpy_callback**()

Adds the callback to the CUDA stream to acknowledge the RF samples have been copied.

explicit **DSPCore**(zmq::context_t &context, SignalProcessingOptions &options, uint32_t sq_num, double rx_rate, double output_sample_rate, std::vector<std::vector<float>> filter_taps, std::vector<cuComplex> beam_phases, double driver_initialization_time, double sequence_start_time, std::vector<uint32_t> dm_rates, std::vector<*rx_slice*> slice_info)

Initializes the parameters needed in order to do asynchronous DSP processing.

The constructor creates a new CUDA stream and initializes the timing events. It then opens the shared memory with the received RF samples for a pulse sequence.

Parameters

- **context** – ZMQ's application context from which to create sockets.

- **sig_options** – The signal processing options.
- **sequence_num** – **[in]** The pulse sequence number for which will be acknowledged.
- **rx_rate** – **[in]** The *USRP* sampling rate.
- **output_sample_rate** – **[in]** The final decimated output sample rate.
- **filter_taps** – **[in]** The filter taps for each stage.
- **beam_phases** – **[in]** The beam phases.
- **driver_initialization_time** – **[in]** The driver initialization time.
- **sequence_start_time** – **[in]** The sequence start time.
- **dm_rates** – **[in]** The decimation rates.
- **slice_info** – **[in]** The slice info given as a vector of rx_slice structs.

~DSPCore()

Frees all associated pointers, events, and streams. Removes and deletes shared memory.

void **allocate_and_copy_frequencies**(void *freqs, uint32_t num_freqs)

Allocates device memory for the filtering frequencies and then copies them to device.

Parameters

- **freqs** – A pointer to the filtering freqs.
- **num_freqs** – **[in]** The number of freqs.

void **allocate_and_copy_rf_samples**(uint32_t total_antennas, uint32_t num_samples_needed, int64_t extra_samples, uint32_t offset_to_first_pulse, double time_zero, double start_time, uint64_t ringbuffer_size, std::vector<cuComplex*> &ringbuffer_ptrs_start)

Allocates device memory for the RF samples and then copies them to device.

Samples are being stored in a shared memory ringbuffer. This function calculates where to index into the ringbuffer for samples and copies them to the gpu. This function will also copy the samples to a shared memory section that data write, or another process can access in order to work with the raw RF samples.

Parameters

- **total_antennas** – **[in]** The total number of antennas.
- **num_samples_needed** – **[in]** The number of samples needed from each antenna ringbuffer.
- **extra_samples** – **[in]** The number of extra samples needed for filter propagation.
- **offset_to_first_pulse** – **[in]** Offset from sequence start to center of first pulse.
- **time_zero** – **[in]** The time the driver began collecting samples. seconds since epoch.
- **start_time** – **[in]** The start time of the pulse sequence. seconds since epoch.
- **ringbuffer_size** – **[in]** The ringbuffer size in number of samples.
- **ringbuffer_ptrs_start** – A vector of pointers to the start of each antenna ringbuffer.

void **allocate_and_copy_bandpass_filters**(void *taps, uint32_t total_taps)

Allocate and copy bandpass filters for all rx freqs to gpu.

Parameters

- **taps** – A pointer to the filter taps.
- **total_taps** – **[in]** The total amount of filter taps.

std::vector<cuComplex*> **get_filter_outputs_h**()

Gets the vector of host side filter outputs.

Returns
The filter outputs host vector.

cuComplex ***get_last_filter_output_d**()

Gets the last filter output d.

Returns
The last filter output d.

std::vector<cuComplex*> **get_lowpass_filters_d**()

cuComplex ***get_last_lowpass_filter_d**()

Gets the last pointer stored in the lowpass filters vector.

Returns
The last lowpass filter pointer inserted into the vector.

std::vector<uint32_t> **get_samples_per_antenna**()

Gets the samples per antenna vector. Vector contains an element for each stage.

Returns
The samples per antenna vector.

std::vector<uint32_t> **get_dm_rates**()

Gets the vector of decimation rates.

Returns
The dm rates.

cuComplex ***get_bp_filters_p**()

Gets the bandpass filters device pointer.

Returns
The bandpass filter pointer.

void **allocate_and_copy_lowpass_filter**(void *taps, uint32_t total_taps)

Allocate and copy a lowpass filter to the gpu.

Parameters

- **taps** – A pointer to the filter taps.
- **total_taps** – **[in]** The total amount of filter taps.

void **allocate_output**(uint32_t num_output_samples)

Allocate a filter output on the GPU.

Parameters
num_output_samples – **[in]** The number output samples

std::vector<std::vector<float>> **get_filter_taps**()

The vector containing vectors of filter taps for each stage.

Returns
The filter taps vectors for each stage.

uint32_t **get_num_antennas**()

Gets the number of antennas.

Returns
The number of antennas.

float **get_total_timing**()

Gets the total GPU process timing in milliseconds.

Returns
The total process timing.

float **get_decimate_timing**()

Gets the total decimation timing in milliseconds.

Returns
The decimation timing.

void **allocate_and_copy_host**(uint32_t num_output_samples, cuComplex *output_d)

Allocate a host pointer for decimation stage output and then copy data.

Parameters

- **num_output_samples** – **[in]** The number output samples needed.
- **output_d** – The device pointer from which to copy from.

void **clear_device_and_destroy**()

cuComplex ***get_rf_samples_p**()

Gets the device pointer to the RF samples.

Returns
The RF samples device pointer.

std::vector<cuComplex> **get_rf_samples_h**()

Gets the host pointer to the RF samples.

Returns
The rf samples host pointer.

double ***get_frequencies_p**()

Gets the device pointer to the receive frequencies.

Returns
The frequencies device pointer.

uint32_t **get_num_rf_samples**()

Gets the number of rf samples.

Returns
The number of rf samples.

uint32_t **get_sequence_num**()

Gets the sequence number.

Returns
The sequence number.

double **get_rx_rate**()

Gets the rx sample rate.

Returns
The rx sampling rate (samples per second).

double **get_output_sample_rate**()

Gets the output sample rate.

Returns
The output decimated and filtered rate (samples per second).

double **get_driver_initialization_time**()

Gets the driver initialization timestamp.

Returns
The driver initialization timestamp.

double **get_sequence_start_time**()

Gets the sequence start timestamp.

Returns
The sequence start timestamp.

std::vector<*rx_slice*> **get_slice_info**()

Gets the vector of slice information, rx_slice structs.

Returns
The vector of rx_slice structs with slice information.

cudaStream_t **get_cuda_stream**()

Gets the CUDA stream this *DSPCore*'s work is associated to.

Returns
The CUDA stream.

std::vector<cuComplex> **get_beam_phases**()

Gets the vector of beam phases.

Returns
The beam phases.

std::string **get_shared_memory_name**()

Gets the name of the shared memory section.

Returns
The shared memory name string.

void **start_decimate_timing**()

Starts the timing before the GPU kernels execute.

void **stop_timing**()

Stops the timers that the constructor starts.

void **send_ack**()

Sends the acknowledgment to the radar control that the RF samples have been transfered.

RF samples of one pulse sequence can be transfered asynchronously while samples of another are being processed. This means that it is possible to start running a new pulse sequence in the driver as soon as the samples are copied. The asynchronous nature means only timing constraint is the time needed to run the GPU kernels for decimation.

void **send_timing**()

Sends the GPU kernel timing to the radar control.

The timing here is used as a rate limiter, so that the GPU doesn't become backlogged with data. If the GPU is overburdened, this will result in less averages, but the system wont crash.

void **send_processed_data**(processeddata::ProcessedData &pd)

Sends a processed data packet to data write.

Parameters
pd – A processeddata protobuf object.

Public Members

SignalProcessingOptions **sig_options**

Filtering ***dsp_filters**

Private Functions

void **allocate_and_copy_rf_from_device**(uint32_t num_rf_samples)

Private Members

cudaStream_t **stream**

CUDA stream the work will be associated with.

uint32_t **sequence_num**

Sequence number used to identify and acknowledge a pulse sequence.

double **rx_rate**

Rx sampling rate for the data being processed.

double **output_sample_rate**

Output sampling rate of the filtered, decimated, processed data.

std::vector<zmq::socket_t> **zmq_sockets**

The unique sockets for communicating between processes.

float **total_process_timing_ms**

Stores the total GPU process timing once all the work is done.

float **decimate_kernel_timing_ms**

Stores the decimation timing.

double *freqs_d

Pointer to the device rx frequencies.

cuComplex *rf_samples_d

Pointer to the RF samples on device.

cuComplex *bp_filters_d

Pointer to the first stage bandpass filters on device.

std::vector<cuComplex*> lp_filters_d

Vector of device side lowpass filter pointers.

std::vector<cuComplex*> filter_outputs_d

Vector of device side filter output pointers.

std::vector<cuComplex*> filter_outputs_h

Vector of host side filter output pointers.

std::vector<uint32_t> samples_per_antenna

Vector of the samples per antenna at each stage of decimation.

std::vector<uint32_t> dm_rates

Vector of decimation rates at each stage.

std::vector<std::vector<float>> filter_taps

Vector that holds the vectors of filter taps at each stage.

cudaEvent_t initial_start

CUDA event to timestamp when the GPU processing begins.

cudaEvent_t kernel_start

CUDA event to timestamp when the kernels begin executing.

cudaEvent_t stop

CUDA event to timestamp when the GPU processing stops.

cudaEvent_t mem_transfer_end

Cuda event to timestamp the transfer of RF samples to the GPU.

float mem_time_ms

Stores the memory transfer timing.

std::vector<cuComplex*> ringbuffers

A vector of pointers to the start of ringbuffers.

std::vector<cuComplex> **rf_samples_h**

A host side vector for the rf samples.

uint32_t **num_antennas**

The number of total antennas.

uint32_t **num_rf_samples**

The number of rf samples per antenna.

std::vector<cuComplex> **beam_phases**

A set of beam angle phases for each beam direction.

SharedMemoryHandler **shm**

A handler for a shared memory section.

double **driver_initialization_time**

Timestamp of when the driver began sampling. Seconds since epoch.

double **sequence_start_time**

Timestamp of when the sequence began. Seconds since epoch.

std::vector<*rx_slice*> **slice_info**

Slice information given from rx_slice structs.

File decimate.cu

Functions

inline __device__ cuComplex __shfl_down_sync (cuComplex var, unsigned int srcLane, int width=32)

Overloads __shfl_down to handle cuComplex.

__shfl can only shuffle 4 bytes at time. This overload utilizes a trick similar to the below link in order to shuffle 8 byte values. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/ http://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions

Parameters

- **var** – **[in]** cuComplex value to shuffle.
- **srcLane** – **[in]** Relative lane from within the warp that should shuffle its variable down.
- **width** – **[in]** Section of the warp to shuffle. Defaults to full warp size.

Returns

Shuffled cuComplex variable.

`__device__ cuComplex parallel_reduce (cuComplex *data, uint32_t tap_offset)`

Performs a parallel reduction to sum a series of values together.

NVIDIA supplies many versions of optimized parallel reduction. This is a slightly modified version of reduction #5 from NVIDIA examples. /usr/local/cuda/samples/6_Advanced/reduction

Parameters

- `data` – A pointer to a set of cuComplex data to reduce.
- `tap_offset` – **[in]** The offset into the data from which to pull values.

Returns

Final sum after reduction.

`__device__ __forceinline__ cuComplex _exp (cuComplex z)`

cuComplex version of exponential function.

Parameters

`z` – **[in]** Complex number.

Returns

Complex exponential of input.

`__global__ void bandpass_decimate1024 (cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, double F_s, double *freqs)`

Performs decimation using bandpass filters on a set of input RF samples if the total number of filter taps for all filters is less than 1024.

This function performs a parallel version of filtering+downsampling on the GPU to be able process data in realtime. This algorithm will use 1 GPU thread per filter tap if there are less than 1024 taps for all filters combined. Only works with power of two length filters, or a filter that is zero padded to a power of two in length. This algorithm takes a single set of wide band samples from the *USRP* driver, and produces an output data set for each RX frequency. The phase of each output sample is corrected to after decimating via modified Frerking method.

gridDim.x - Total number of output samples there will be after decimation. gridDim.y - Total number of antennas.

blockIdx.x - Decimated output sample index. blockIdx.y - Antenna index.

blockDim.x - Number of filter taps in the lowpass filter. blockDim.y - Total number of filters. Corresponds to total receive frequencies.

threadIdx.x - Filter tap index. threadIdx.y - Filter index.

Parameters

- `original_samples` – **[in]** A pointer to original input samples from each antenna to decimate.
- `decimated_samples` – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- `filter_taps` – **[in]** A pointer to one or more filters needed for each frequency.
- `dm_rate` – **[in]** Decimation rate.

- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **F_s** – **[in]** The sampling frequency in hertz.
- **freqs** – **[in]** A pointer to the frequencies used in mixing.

__global__ void bandpass_decimate2048 (cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, double F_s, double *freqs)

Performs decimation using bandpass filters on a set of input RF samples if the total number of filter taps for all filters is less than 2048.

This function performs a parallel version of filtering+downsampling on the GPU to be able process data in realtime. This algorithm will use 1 GPU thread to process two filter taps if there are less than 2048 taps for all filters combined. Intended to be used if there are more than 1024 total threads, as that is the max block size possible for CUDA. Only works with power of two length filters, or a filter that is zero padded to a power of two in length. This algorithm takes a single set of wide band samples from the *USRP* driver, and produces a output data set for each RX frequency.

gridDim.x - Total number of output samples there will be after decimation. gridDim.y - Total number of antennas.

blockIdx.x - Decimated output sample index. blockIdx.y - Antenna index.

blockDim.x - Number of filter taps in each filter / 2. blockDim.y - Total number of filters. Corresponds to total receive frequencies.

threadIdx.x - Every second filter tap index. threadIdx.y - Filter index.

Parameters

- **original_samples** – **[in]** A pointer to original input samples from each antenna to decimate.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **F_s** – **[in]** The sampling frequency in hertz.
- **freqs** – **[in]** A pointer to the frequencies used in mixing.

void **bandpass_decimate1024_wrapper**(cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, double F_s, double *freqs, cudaStream_t stream)

This function wraps the bandpass_decimate1024 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to original input samples from each antenna to decimate.

- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequencies.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **F_s** – **[in]** The original sampling frequency.
- **freqs** – A pointer to the frequencies being filtered.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

void **bandpass_decimate2048_wrapper**(cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, double F_s, double *freqs, cudaStream_t stream)

This function wraps the bandpass_decimate2048 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to original input samples from each antenna to decimate.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequencies.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **F_s** – **[in]** The original sampling frequency.
- **freqs** – A pointer to the frequencies being filtered.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

__global__ void lowpass_decimate1024 (cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna)

Performs decimation using a lowpass filter on one or more sets of baseband samples corresponding to each RX frequency. This algorithm works on filters with less that 1024 taps.

This function performs a parallel version of filtering+downsampling on the GPU to be able process data in realtime. This algorithm will use 1 GPU thread per filter tap if there are less than 1024 taps for all filters combined. Only works with power of two length filters, or a filter that is zero padded to a power of two in length. This algorithm takes one or more baseband datasets corresponding to each RX frequency and filters each one using a single lowpass filter before downsampling.

gridDim.x - The number of decimated output samples for one antenna in one frequency data set. gridDim.y - Total number of antennas. gridDim.z - Total number of frequency data sets.

blockIdx.x - Decimated output sample index. blockIdx.y - Antenna index. blockIdx.z - Frequency dataset index.

blockDim.x - Number of filter taps in the lowpass filter.

threadIdx.x - Filter tap indices.

Parameters

- **original_samples** – **[in]** A pointer to input samples for one or more baseband datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency dataset after decimation.
- **filter_taps** – **[in]** A pointer to a lowpass filter used for further decimation.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.

__global__ void lowpass_decimate2048 (cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna)

Performs decimation using a lowpass filter on one or more sets of baseband samples corresponding to each RX frequency. This algorithm works on filters with less that 2048 taps.

This function performs a parallel version of filtering+downsampling on the GPU to be able process data in realtime. This algorithm will use 1 GPU thread to process two filter taps if there are less than 2048 taps for all filters combined. Intended to be used if there are more than 1024 total threads, as that is the max block size possible for CUDA. Only works with power of two length filters, or a filter that is zero padded to a power of two in length. This algorithm takes one or more baseband datasets corresponding to each RX frequency and filters each one using a single lowpass filter before downsampling.

gridDim.x - The number of decimated output samples for one antenna in one frequency data set. gridDim.y - Total number of antennas. gridDim.z - Total number of frequency data sets.

blockIdx.x - Decimated output sample index. blockIdx.y - Antenna index. blockIdx.z - Frequency dataset index.

blockDim.x - Number of filter taps in the lowpass filter / 2.

threadIdx.x - Every second filter tap index.

Parameters

- **original_samples** – **[in]** A pointer to input samples for one or more baseband datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency dataset after decimation.
- **filter_taps** – **[in]** A pointer to a lowpass filter used for further decimation.
- **dm_rate** – **[in]** Decimation rate.

- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.

void **lowpass_decimate1024_wrapper**(cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, cudaStream_t stream)

This function wraps the lowpass_decimate1024 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to one or more baseband frequency datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one lowpass filter.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in each data set.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequency datasets.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

void **lowpass_decimate2048_wrapper**(cuComplex *original_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, cudaStream_t stream)

This function wraps the lowpass_decimate2048 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to one or more baseband frequency datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one lowpass filter.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in each data set.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequency datasets.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

File decimate.hpp

Enums

enum **DecimationType**

Values:

enumerator **lowpass**

enumerator **bandpass**

Functions

void **bandpass_decimate1024_wrapper**(cuComplex *input_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, double F_s, double *freqs, cudaStream_t stream)

This function wraps the bandpass_decimate1024 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to original input samples from each antenna to decimate.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequencies.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **F_s** – **[in]** The original sampling frequency.
- **freqs** – A pointer to the frequencies being filtered.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

void **bandpass_decimate2048_wrapper**(cuComplex *input_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, double F_s, double *freqs, cudaStream_t stream)

This function wraps the bandpass_decimate2048 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to original input samples from each antenna to decimate.

- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the original set of samples.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequencies.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **F_s** – **[in]** The original sampling frequency.
- **freqs** – A pointer to the frequencies being filtered.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

void **lowpass_decimate1024_wrapper**(cuComplex *input_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, cudaStream_t stream)

This function wraps the lowpass_decimate1024 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to one or more baseband frequency datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one lowpass filter.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in each data set.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequency datasets.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

void **lowpass_decimate2048_wrapper**(cuComplex *input_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, cudaStream_t stream)

This function wraps the lowpass_decimate2048 kernel so that it can be called from another file.

Parameters

- **original_samples** – **[in]** A pointer to one or more baseband frequency datasets.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one lowpass filter.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in each data set.

- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequency datasets.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **stream** – **[in]** CUDA stream with which to associate the invocation of the kernel.

template<*DecimationType* **type**>
void **call_decimate**(cuComplex *input_samples, cuComplex *decimated_samples, cuComplex *filter_taps, uint32_t dm_rate, uint32_t samples_per_antenna, uint32_t num_taps_per_filter, uint32_t num_freqs, uint32_t num_antennas, double F_s, double *freqs, const char *output_msg, cudaStream_t stream)

Selects which decimate kernel to run.

Parameters

- **input_samples** – **[in]** A pointer to original input samples from each antenna to decimate.
- **decimated_samples** – **[in]** A pointer to a buffer to place output samples for each frequency after decimation.
- **filter_taps** – **[in]** A pointer to one or more filters needed for each frequency. If using lowpass, one filter is used. If using bandpass, there is one filter for each RX frequency.
- **dm_rate** – **[in]** Decimation rate.
- **samples_per_antenna** – **[in]** The number of samples per antenna in the input set of samples for one frequency.
- **num_taps_per_filter** – **[in]** Number of taps per filter.
- **num_freqs** – **[in]** Number of receive frequencies.
- **num_antennas** – **[in]** Number of antennas for which there are samples.
- **F_s** – **[in]** The original sampling frequency.
- **freqs** – A pointer to the filtering freqs.
- **output_msg** – **[in]** A simple character string that can be used to debug or distinguish different stages.
- **stream** – **[in]** The CUDA stream for which to run a run a kernel.

```
        Based off the total number of filter taps, this function␣
↪will choose what decimate
        kernel to use.
```

Template Parameters

type – { description }

File filtering.hpp

class **Filtering**

#include <filtering.hpp> Class for filtering.

Public Functions

Filtering() = default

explicit **Filtering**(std::vector<std::vector<float>> input_filter_taps)

The constructor finds the number of filter taps for each stage and then a lowpass filter for each stage.

Parameters

input_filter_taps – **[in]** The filter taps sent from radar control.

void **save_filter_to_file**(const std::vector<std::complex<float>> &filter_taps, std::string name)

Writes out a set of filter taps to file in case they need to be tested.

Parameters

- **filter_taps** – **[in]** A reference to a vector of filter taps.
- **name** – **[in]** A output file name.

void **mix_first_stage_to_bandpass**(const std::vector<double> &rx_freqs, double initial_rx_rate)

Mixes the first stage lowpass filter to bandpass filters for each RX frequency.

Creates a flatbuffer with a bandpass filter for each RX frequency to be used in decimation.

Parameters

- **rx_freqs** – **[in]** rx_freqs A reference to a vector of RX frequencies in Hz.
- **initial_rx_sample_rate** – **[in]** initial_rx_sample_rate The *USRP* RX sampling rate in Hz.

std::vector<std::vector<std::complex<float>>> **get_mixed_filter_taps**()

Gets the mixed filter taps at each stage.

A temp vector is created. The first stage taps are replaced with the bandpass taps.

Returns

The mixed filter taps.

std::vector<std::vector<std::complex<float>>> **get_unmixed_filter_taps**()

Gets the unmixed filter taps at each stage.

The unmixed filter taps are returned.

Returns

The unmixed filter taps.

Private Functions

std::vector<std::complex<float>> **fill_filter**(std::vector<float> &filter_taps)

Fills the lowpass filter taps with zero to a size that is a power of 2.

Parameters

filter_taps – **[in]** The filter taps provided, will be real.

Returns

A vector of filter taps. Filter is real, but represented using complex<float> form R + i0 for each tap. The vector is filled with zeros at the end to reach a length that is a power of 2 for processing.

Private Members

std::vector<std::vector<std::complex<float>>> **filter_taps**

Vector that holds the vectors of filter taps at each stage.

std::vector<std::complex<float>> **bandpass_taps**

A vector to hold the bandpass taps after first stage filter has been mixed.

7.1.5 USRP N200 Driver

The N200 driver is a C++ application that controls the operation of the USRP N200 transceivers. The driver is responsible for using Ettus' UHD software to configure a multi-USRP device and configure the device for SuperDARN operation.

As part of the driver, a C++ class was written to abstract the configuration of the N200s. The driver configures the N200s using certain options from the config file as well as options related to the experiment. All runtime options and control are defined by the Radar Control module.

The driver consists of the main function and three worker threads. The main function is responsible for instantiating a USRP object, and configuring some initial runtime options such as which physical devices to use, the GPIO bank, the timing signal masks, the clock source, the subdevs for TX and RX, and the time source. These options are configured once at runtime and then not changed during operation. The main function then starts the transmit, and receive worker threads.

Transmit Thread

On a driver packet indicating the start of a new sequence(SOB is true), the transmit thread will configure some multi-USRP parameters such as what TX channels(antennas) to use, the TX center frequency, and the buffer of samples to send as a pulse. The driver requires these all be set once but can be omitted in future sequences if they are repeated. No need to continually serialize and deserialize duplicated information. Each driver packet in the sequence contains a relative time from the start of the sequence to when the pulse should be transmitted. If SOB is true, then a sequence start time is created by using the UHD current time as a reference to when pulses should start. A slight delay is added to allow for some CPU time to finish configuring the pulse. Once the pulse time relative to time zero is calculated, the multi-USRP object is configured to send the pulse samples at that time.

TR switching signals are generated using the USRP ATR functionality. The ATR pins are only triggered exactly when the USRP is sending or receiving, so in order to properly window the RF signal, zeros are padded to the start and end of the signal. From testing, the zeros do not create any issues such as higher noise, etc. They purely allow us to create a window for TR signals. The actual TR signal is ATR_XX. We are receiving during the whole sequence, so

the full-duplex pin is the pin that goes high when we are transmitting while receiving. The current version of borealis does not allow for transmitting only.

After all pulses are sent all the parameters needed for processing the received samples are sent to the DSP unit. The ringbuffer initialization time and the sequence start time are included so that the DSP unit can properly select where the sequence samples start in the ringbuffer.

Receive Thread

Under heavy load, the USRP does not seem to respond well to timed receive events. We use a continuous receive ringbuffer system to minimize dropped samples. Instead of using a time triggered receive event, we start sampling continuously at a time. We then use the timestamp of the transmit pulse sequence to calculate where in the ringbuffer the pulse sequence samples are located.

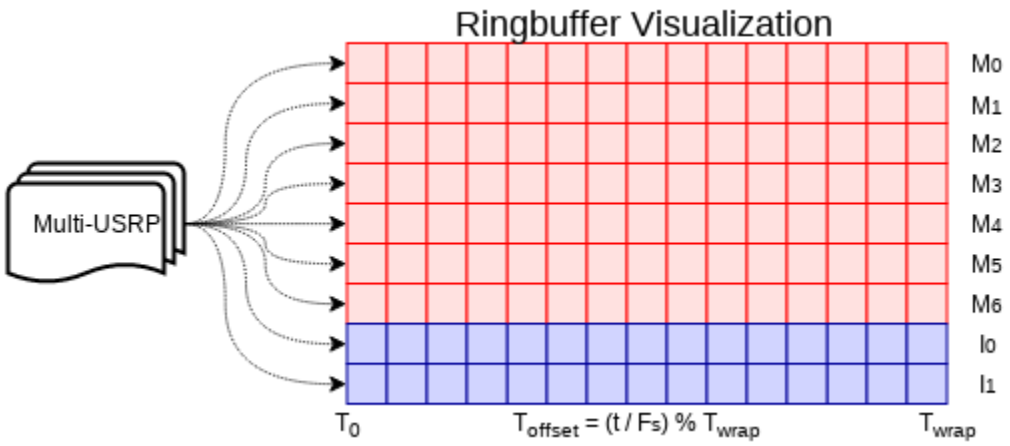


Fig. 4: Diagram of Ringbuffer

The diagram above shows that each USRP receive channel maps to an individual ringbuffer implemented in memory. Each square represents a sample, and for brevity only a few channels are shown. T_0 is the beginning sample in time at which the ringbuffer is initialized. T_{wrap} is the sample in time at which the ringbuffer wraps around, or the length of ringbuffer. By knowing exactly when the ringbuffer was initialized and the exact time the pulse sequence began, we can calculate where in the ringbuffer the sequence samples are by calculating how many times the buffer wrapped. If t is the amount of time passed from when the buffer was initialized to the time the sequence was sent, then T_{offset} is calculated by dividing t by the sampling frequency, F_s, to convert time to number of samples, then dividing this result by T_{wrap} and taking the remainder. T_{offset} is then used as the offset into the buffer from which samples are later copied out for further processing.

Once the multi-USRP is configured, a shared memory handler is created. Received samples are put directly into shared memory that can be accessed by both the driver and Rx Signal Processing. This minimizes the amount of interprocess copying needed. Once the shared memory is created, pointer offsets into the shared memory are calculated for where each channel buffer begins.

The receive thread configures the ringbuffer and shared memory sections. It then initializes the USRP streaming mode. Right before it begins streaming it sends the initialization time and ringbuffer size to transmit thread. This acts as as "go" signal for the transmit thread to begin and the transmit thread needs to send this info to the DSP unit.

File usrp_driver.cpp

Defines

SET_TIME_COMMAND_DELAY

TUNING_DELAY

Functions

std::vector<std::vector<std::complex<float>>> **make_tx_samples**(const driverpacket::DriverPacket &driver_packet, const DriverOptions &driver_options)

Makes a set of vectors of the samples for each TX channel from the driver packet.

Values in a protobuffer have no contiguous underlying storage so values need to be parsed into a vector.

Parameters

- **driver_packet** – **[in]** A received driver packet from radar_control.
- **driver_options** – **[in]** The parsed config options needed by the driver.

Returns

A set of vectors of TX samples for each *USRP* channel.

void **transmit**(zmq::context_t &driver_c, USRP &usrp_d, const DriverOptions &driver_options)

void **receive**(zmq::context_t &driver_c, USRP &usrp_d, const DriverOptions &driver_options)

Runs in a seperate thread to control receiving from the USRPs.

Parameters

- **driver_c** – **[in]** The driver ZMQ context.
- **usrp_d** – **[in]** The multi-USRP SuperDARN wrapper object.
- **driver_options** – **[in]** The driver options parsed from config.

int32_t **UHD_SAFE_MAIN**(int32_t argc, char *argv[])

UHD wrapped main function to start threads.

Creates a new multi-USRP object using parameters from config file. Starts control, receive, and transmit threads to operate on the multi-USRP object.

Returns

EXIT_SUCCESS

Variables

uhd::time_spec_t **box_time**

File usrp.hpp

class **USRP**

#include <usrp.hpp> Contains an abstract wrapper for the *USRP* object.

Public Functions

explicit **USRP**(const DriverOptions &driver_options, float tx_rate, float rx_rate)

Creates the multiUSRP abstraction with the options from the config file.

Parameters

- **driver_options** – **[in]** The driver options parsed from config
- **tx_rate** – **[in]** The transmit rate in Sps (samples per second, Hz).
- **rx_rate** – **[in]** The receive rate in Sps (samples per second, Hz).

void **set_usrp_clock_source**(std::string source)

Sets the *USRP* clock source.

Parameters

source – **[in]** A string for a valid *USRP* clock source.

void **set_tx_subdev**(std::string tx_subdev)

Sets the *USRP* transmit subdev specification.

Parameters

tx_subdev – **[in]** A string for a valid transmit subdev.

double **set_tx_rate**(std::vector<size_t> chs)

Sets the transmit sample rate.

Parameters

chs – **[in]** A vector of *USRP* channels to tx on.

Returns

Actual set tx rate.

double **get_tx_rate**(uint32_t channel = 0)

Gets the *USRP* transmit sample rate.

Returns

The transmit sample rate in Sps.

double **set_tx_center_freq**(double freq, std::vector<size_t> chs, uhd::time_spec_t tune_delay)

Sets the transmit center frequency.

The *USRP* uses a numbered channel mapping system to identify which data streams come from which *USRP* and its daughterboard frontends. With the daughtboard frontends connected to the transmitters,

controlling what *USRP* channels are selected will control what antennas are used and what order they are in. To synchronize tuning of all boxes, timed commands are used so that everything is done at once.

Parameters

- **freq** – **[in]** The frequency in Hz.
- **chs** – **[in]** A vector of which *USRP* channels to set a center frequency.
- **tune_delay** – **[in]** The amount of time in future to tune the devices.

Returns
The actual set tx center frequency for the USRPs

double **get_tx_center_freq**(uint32_t channel = 0)

Gets the transmit center frequency.

Returns
The actual center frequency that the USRPs are tuned to.

void **set_main_rx_subdev**(std::string main_subdev)

Sets the receive subdev for the main array antennas.

Will set all boxes to receive from first *USRP* channel of all mboards for main array.

Parameters
main_subdev – **[in]** A string for a valid receive subdev.

void **set_interferometer_rx_subdev**(std::string interferometer_subdev, uint32_t interferometer_antenna_count)

Sets the interferometer receive subdev.

Override the subdev spec of the first mboards to receive on a second channel for the interferometer.

Parameters

- **interferometer_subdev** – **[in]** A string for a valid receive subdev.
- **interferometer_antenna_count** – **[in]** The interferometer antenna count.

double **set_rx_rate**(std::vector<size_t> rx_chs)

Sets the receive sample rate.

Parameters
rx_chs – **[in]** The *USRP* channels to rx on.

Returns
The actual rate set.

double **get_rx_rate**(uint32_t channel = 0)

Gets the *USRP* transmit sample rate.

Returns
The transmit sample rate in Sps.

double **set_rx_center_freq**(double freq, std::vector<size_t> chs, uhd::time_spec_t tune_delay)

Sets the receive center frequency.

The *USRP* uses a numbered channel mapping system to identify which data streams come from which *USRP* and its daughterboard frontends. With the daughtboard frontends connected to the transmitters, controlling what *USRP* channels are selected will control what antennas are used and what order they are in. To simplify data processing, all antenna mapped channels are used. To synchronize tuning of all boxes, timed commands are used so that everything is done at once.

Parameters

- **freq** – **[in]** The frequency in Hz.
- **chs** – **[in]** A vector of which *USRP* channels to set a center frequency.
- **tune_delay** – **[in]** The amount of time in future to tune the devices.

Returns
The actual center frequency that the USRPs are tuned to.

double **get_rx_center_freq**(uint32_t channel = 0)

Gets the receive center frequency.

Returns
The actual center frequency that the USRPs are tuned to.

void **set_time_source**(std::string source, std::string clk_addr)

Sets the *USRP* time source.

Uses the method Ettus suggests for setting time on the x300. https://files.ettus.com/manual/page_gpsdo_x3x0.html Falls back to Juha Vierinen's method of latching to the current time by making sure the clock time is in a stable place past the second if no gps is available. The *USRP* is then set to this time.

Parameters

- **source** – **[in]** A string with the time source the *USRP* will use.
- **clk_addr** – **[in]** IP address of the octoclock for gps timing.

void **check_ref_locked**()

Makes a quick check that each *USRP* is locked to a reference frequency.

void **create_usrp_rx_stream**(std::string cpu_fmt, std::string otw_fmt, std::vector<size_t> chs)

Creates an *USRP* receive stream.

Parameters

- **cpu_fmt** – **[in]** The cpu format for the tx stream. Described in UHD docs.
- **otw_fmt** – **[in]** The otw format for the tx stream. Described in UHD docs.
- **chs** – **[in]** A vector of which *USRP* channels to receive on.

void **create_usrp_tx_stream**(std::string cpu_fmt, std::string otw_fmt, std::vector<size_t> chs)

Creates an *USRP* transmit stream.

Parameters

- **cpu_fmt** – **[in]** The cpu format for the tx stream. Described in UHD docs.
- **otw_fmt** – **[in]** The otw format for the tx stream. Described in UHD docs.
- **chs** – **[in]** A vector of which *USRP* channels to transmit on.

void **set_command_time**(uhd::time_spec_t cmd_time)

Sets the command time.

Parameters

cmd_time – **[in]** The command time to run a timed command.

void **clear_command_time**()

Clears any timed *USRP* commands.

std::vector<uint32_t> **get_gpio_bank_high_state**()

Gets the state of the GPIO bank represented as a decimal number.

std::vector<uint32_t> **get_gpio_bank_low_state**()

Gets the state of the GPIO bank represented as a decimal number.

uhd::time_spec_t **get_current_usrp_time**()

Gets the current *USRP* time.

Returns

The current *USRP* time.

uhd::rx_streamer::sptr **get_usrp_rx_stream**()

Gets a pointer to the *USRP* rx stream.

Returns

The *USRP* rx stream.

uhd::tx_streamer::sptr **get_usrp_tx_stream**()

Gets a pointer to the *USRP* tx stream.

Returns

The *USRP* tx stream.

uhd::usrp::multi_usrp::sptr **get_usrp**()

Gets the usrp.

Returns

The multi-USRP shared pointer.

std::string **to_string**(std::vector<size_t> tx_chs, std::vector<size_t> rx_chs)

Returns a string representation of the *USRP* parameters.

Parameters

- **tx_chs** – **[in]** *USRP* TX channels for which to generate info for.
- **rx_chs** – **[in]** *USRP* RX channels for which to generate info for.

Returns

String representation of the *USRP* parameters.

void **invert_test_mode**(uint32_t mboard = 0)

Inverts the current test mode signal. Useful for testing.

Parameters

mboard – **[in]** The *USRP* to invert test mode on. Default 0.

void **set_test_mode**(uint32_t mboard = 0)

Sets the current test mode signal HIGH.

Parameters

mboard – **[in]** The *USRP* to set test mode HIGH on. Default 0.

void **clear_test_mode**(uint32_t mboard = 0)

Clears the current test mode signal LOW.

Parameters

mboard – **[in]** The *USRP* to clear test mode LOW on. Default 0.

Private Functions

void **set_atr_gpios**()

Sets the *USRP* automatic transmit/receive states on GPIO for the given daughtercard bank.

void **set_output_gpios**()

Sets the pins mapping the test mode signals as GPIO outputs.

void **set_input_gpios**()

Sets the pins mapping the AGC and low power signals as GPIO inputs.

Private Members

uhd::usrp::multi_usrp::sptr **usrp_**

A shared pointer to a new multi-USRP device.

std::string **gpio_bank_high_**

A string representing what GPIO bank to use on the USRPs for active high sigs.

std::string **gpio_bank_low_**

A string representing what GPIO bank to use on the USRPs for active low sigs.

uint32_t **scope_sync_mask_**

The bitmask to use for the scope sync GPIO.

uint32_t **atten_mask_**

The bitmask to use for the attenuator GPIO.

uint32_t **tr_mask_**

The bitmask to use for the TR GPIO.

uint32_t **atr_xx_**

Bitmask used for full duplex ATR.

uint32_t **atr_rx_**

Bitmask used for rx only ATR.

uint32_t **atr_tx_**

Bitmask used for tx only ATR.

uint32_t **atr_0x_**

Bitmask used for idle ATR.

uint32_t **agc_st_**

Bitmask used for AGC signal.

uint32_t **lo_pwr_**

Bitmask used for lo pwr signal.

uint32_t **test_mode_**

Bitmask used for test mode signal.

float **tx_rate_**

The tx rate in Hz.

float **rx_rate_**

The rx rate in Hz.

uhd::tx_streamer::sptr **tx_stream_**

uhd::rx_streamer::sptr **rx_stream_**

class **TXMetadata**

#include <usrp.hpp> Wrapper for the *USRP* TX metadata object.

Used to hold and initialize a new tx_metadata_t object. Creates getters and setters to access properties.

Public Functions

TXMetadata()

Constructs a blank *USRP* TX metadata object.

uhd::tx_metadata_t **get_md**()

Gets the TX metadata oject that can be sent the USRPs.

Returns
The *USRP* TX metadata.

void **set_start_of_burst**(bool start_of_burst)

Sets whether this data is the start of a burst.

Parameters
start_of_burst – **[in]** The start of burst boolean.

void **set_end_of_burst**(bool end_of_burst)

Sets whether this data is the end of the burst.

Parameters
end_of_burst – **[in]** The end of burst boolean.

void **set_has_time_spec**(bool has_time_spec)

Sets whether this data will have a particular timing.

Parameters
has_time_spec – **[in]** Indicates if this metadata will have a time specifier.

void **set_time_spec**(uhd::time_spec_t time_spec)

Sets the timing in the future for this metadata.

Parameters
time_spec – **[in]** The time specifier for this metadata.

Private Members

uhd::tx_metadata_t **md_**

A raw *USRP* TX metadata object.

class **RXMetadata**

#include <usrp.hpp> Wrapper for the *USRP* RX metadata object.

Used to hold and initialize a new tx_metadata_t object. Creates getters and setters to access properties.

Public Functions

RXMetadata() = default

uhd::rx_metadata_t &**get_md**()

Gets the RX metadata object that will be retrieved on receiving.

Returns
The *USRP* RX metadata object.

bool **get_end_of_burst**()

Gets the end of burst.

Returns
The end of burst.

uhd::rx_metadata_t::error_code_t **get_error_code**()

Gets the error code from the metadata on receive.

Returns
The error code.

size_t **get_fragment_offset**()

Gets the fragment offset. The fragment offset is the sample number at start of buffer.

Returns
The fragment offset.

bool **get_has_time_spec**()

Gets the has time specifier status.

Returns
The has time specifier boolean.

bool **get_out_of_sequence**()

Gets out of sequence status. Queries whether a packet is dropped or out of order.

Returns
The out of sequence boolean.

bool **get_start_of_burst**()

Gets the start of burst status.

Returns
The start of burst.

uhd::time_spec_t **get_time_spec**()

Gets the time specifier of the packet.

Returns
The time specifier.

Private Members

uhd::rx_metadata_t **md_**

A raw *USRP* RX metadata object.

7.1.6 data_write package

The data_write package contains the utilities to parse protobuf packets containing antennas_iq data, bfiq data, rawacf data, etc and write that data to HDF5 or JSON files.

Submodules

experiment_prototype

This is the base module for all experiments. An experiment will only run if it inherits from this class.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

class experiment_prototype.experiment_prototype.**ExperimentPrototype**

Bases: object

The base class for all experiments.

A prototype experiment class composed of metadata, including experiment slices (exp_slice) which are dictionaries of radar parameters. Basic, traditional experiments will be composed of a single slice. More complicated experiments will be composed of multiple slices that interface in one of four pre-determined ways, as described under interface_types.

This class is used via inheritance to create experiments.

Some variables shouldn't be changed by the experiment, and their properties do not have setters. Some variables can be changed in the init of your experiment, and can also be modified in-experiment by the class method 'update' in your experiment class. These variables have been given property setters.

The following are the user-modifiable attributes of the ExperimentPrototype that are used to make an experiment:

- xcf: boolean for cross-correlation data. A default can be set here for slices, but any slice can override this setting with the xcf slice key.
- acf: boolean for auto-correlation data on main array. A default can be set here for slices, but any slice can override this setting with the acf slice key.
- acfint: boolean for auto-correlation data on interferometer array. A default can be set here for slices, but any slice can override this setting with the acfint slice key.
- slice_dict: modifiable only using the add_slice, edit_slice, and del_slice methods.
- interface: modifiable using the add_slice, edit_slice, and del_slice methods, or by updating the interface dict directly.

Other parameters are set in the init and cannot be modified after instantiation.

property acf

The default auto-correlation flag boolean.

This provides the default for slices where this key isn't specified.

property acfint

The default interferometer autocorrelation boolean.

This provides the default for slices where this key isn't specified.

add_slice(*exp_slice*, *interfacing_dict={}*)

Add a slice to the experiment.

Parameters

- **exp_slice** – a slice (dictionary of slice_keys) to add to the experiment.
- **interfacing_dict** – dictionary of type {slice_id : INTERFACING , ... } that defines how this slice interacts with all the other slices currently in the experiment.

Raises

ExperimentException if slice is not a dictionary or if there are errors in setup_slice.

Returns

the slice_id of the new slice that was just added.

build_scans()

Build the scan information, which means creating the Scan, AveragingPeriod, and Sequence instances needed to run this experiment.

Will be run by experiment handler, to build iterable objects for radar_control to use. Creates scan_objects in the experiment for identifying which slices are in the scans.

check_new_slice_interfacing(*interfacing_dict*)

Checks that the new slice plays well with its siblings (has interfacing that is resolvable). If so, returns a new dictionary with all interfacing values set.

The interfacing assumes that the interfacing_dict given by the user defines the closest interfacing of the new slice with a slice. For example, if the slice is to be PULSE combined with slice 0, the interfacing dict should provide this information. If only 'SCAN' interfacing with slice 1 is provided, then that will be assumed to be the closest and therefore the interfacing with slice 0 will also be 'SCAN'.

If no interfacing_dict is provided for a slice, the default is to do 'SCAN' type interfacing for the new slice with all other slices.

Parameters
interfacing_dict – the user-provided interfacing dict, which may be empty or incomplete. If empty, all interfacing is assumed to be = 'SCAN' type. If it contains something, we ensure that the interfacing provided makes sense with the values already known for its closest sibling.

Returns
full interfacing dictionary.

Raises
ExperimentException if invalid interface types provided or if interfacing can not be resolved.

check_slice(*exp_slice*)

Check the slice for errors.

This is the first test of the dictionary in the experiment done to ensure values in this slice make sense. This is a self-check to ensure the parameters (for example, txfreq, antennas) are appropriate. All fields should be full at this time (whether filled by the user or given default values in set_slice_defaults). This was built to be useable at any time after setup. :param: exp_slice: a slice to check :raise: ExperimentException: When necessary parameters do not exist or = None (would have to have been overridden by the user for this, as defaults all set when this runs).

check_slice_minimum_requirements(*exp_slice*)

Check the required slice keys.

Check for the minimum requirements of the slice. The following keys are always required: "pulse_sequence", "tau_spacing", "pulse_len", "num_ranges", "first_range", (one of "intt" or "intn"), "beam_angle", and "beam_order". This function may modify the values in this slice dictionary to ensure that it is able to be run and that the values make sense.

Parameters
exp_slice – slice to check.

check_slice_specific_requirements(*exp_slice*)

Set the specific slice requirements depending.

Check the requirements for the specific slice type as identified by the identifiers rxonly and clrfrqflag. The keys that need to be checked depending on these identifiers are "txfreq", "rxfreq", and "clrfrqrange". This function may modify these keys.

Parameters
exp_slice – the slice to check, before adding to the experiment.

property comment_string

A string related to the experiment, to be placed in the experiment's files.

This is read-only once established in instantiation.

property cpid

This experiment's CPID (control program ID, a term that comes from ROS).

The CPID is read-only once established in instantiation. It may be modified at runtime by the set_scheduling_mode function, to set it to a negative value during discretionary time.

property decimation_scheme

The decimation scheme, of type DecimationScheme from the filtering module. Includes all filtering and decimating information for the signal processing module.

This is read-only once established in instantiation.

del_slice(*remove_slice_id*)

Remove a slice from the experiment.

Parameters
remove_slice_id – the id of the slice you'd like to remove.

Returns
a copy of the removed slice.

Raises
exception if remove_slice_id does not exist in the slice dictionary.

edit_slice(*edit_slice_id*, ***kwargs*)

Edit a slice.

A quick way to edit a slice. In reality this is actually adding a new slice and deleting the old one. Useful for quick changes. Note that using this function will remove the slice_id that you are changing and will give it a new id. It will account for this in the interfacing dictionary.

Parameters

- **edit_slice_id** – the slice id of the slice to be edited.
- **kwargs** – dictionary of slice parameter to slice value that you want to change.

Returns new_slice_id
the new slice id of the edited slice, or the edit_slice_id if no change has occurred due to failure of new slice parameters to pass experiment checks.

Raises
exceptions if the edit_slice_id does not exist in slice dictionary or the params or values do not make sense.

property experiment_name

The experiment class name.

get_scan_slice_ids()

Organize the slice_ids by scan.

Take my own interfacing and get info on how many scans and which slices make which scans. Return a list of lists where each inner list contains the slices that are in an averagingperiod that is inside this scan. ie. len(nested_slice_list) = # of averagingperiods in this scan, len(nested_slice_list[0]) = # of slices in the first averagingperiod, etc.

:return list of lists. The list has one element per scan. Each element is a list of slice_ids signifying which slices are combined inside that scan. The list returned could be of length 1, meaning only one scan is present in the experiment.

get_slice_interfacing(*slice_id*)

Check the experiment's interfacing dictionary for all interfacing that pertains to a given slice, and return the interfacing information in a dictionary. :param slice_id: Slice ID to search the interface dictionary for. :return: interfacing dictionary for the slice.

property interface

The dictionary of interfacing for the experiment slices.

Interfacing should be set up for any slice when it gets added, ie. in add_slice, except for the first slice added. The dictionary of interfacing is setup as:

[(slice_id1, slice_id2) : INTERFACING_TYPE, (slice_id1, slice_id3) : INTERFACING_TYPE, . . .]

for all current slice_ids.

property new_slice_id

The next unique slice id that is available to this instance of the experiment.

This gets incremented each time it is called to ensure it returns a unique ID each time.

property num_slices

The number of slices currently in the experiment.

Will change after methods add_slice or del_slice are called.

property options

The config options for running this experiment.

These cannot be set or removed, but are specified in the config.ini, hdw.dat, and restrict.dat files.

property output_rx_rate

The output receive rate of the data, Hz.

This is read-only once established in instantiation.

printing(*msg*)

property rx_bandwidth

The receive bandwidth for this experiment, in Hz.

This is read-only once established in instantiation.

property rx_maxfreq

The maximum receive frequency.

This is the maximum tx frequency possible in this experiment (maximum given by the center frequency and sampling rate), as license doesn't matter for receiving. The maximum is slightly less than that allowed by the center frequency and rxrate, to stay away from the edges of the possible receive band where the signal may be distorted.

property rx_minfreq

The minimum receive frequency.

This is the minimum rx frequency possible in this experiment (minimum given by the center frequency and sampling rate) - license doesn't restrict receiving. The minimum is slightly more than that allowed by the center frequency and rxrate, to stay away from the edges of the possible receive band where the signal may be distorted.

property rxctrfreq

The receive center frequency that USRP is tuned to (kHz).

property rxrate

The receive bandwidth for this experiment, or the receive sampling rate (of I and Q samples) In Hz.

This is read-only once established in instantiation.

property scan_objects

The list of instances of class Scan for use in radar_control.

These cannot be modified by the user, but are created using the slice dictionary.

property scheduling_mode

Return the scheduling mode time type that this experiment is running in. Types are listed in possible_scheduling_modes. Initialized to 'unknown' until set by the experiment handler.

self_check()

Check that the values in this experiment are valid.

Checks all slices.

set_slice_defaults(*exp_slice*)

Set up defaults in case of some parameters being left blank.

Parameters
exp_slice – slice to set defaults of

Returns slice_with_defaults
updated slice

static set_slice_identifiers(*exp_slice*)

Set the hidden slice keys to determine how to run the slice.

This function sets up internal identifier flags 'clrfrqflag' and 'rxonly' in the slice so that we know how to properly set up the slice and know which keys in the slice must be specified and which are unnecessary. If these keys are ever written by the user, they will be rewritten here.

Parameters
exp_slice – slice in which to set identifiers

setup_slice(*exp_slice*)

Check slice for errors and set defaults of optional keys.

Before adding the slice, ensure that the internal parameters are set, remove unnecessary keys and check values of keys that are needed, and set defaults of keys that are optional.

The following are always able to be defaulted, so are optional: "tx_antennas", "rx_main_antennas", "rx_int_antennas", "pulse_phase_offset", "scanboundflag", "scanbound", "acf", "xcf", "acfint", "wavetype", "seqoffset", "averaging_method"

The following are always required for processing acf, xcf, and acfint which we will assume we are always doing: "pulse_sequence", "tau_spacing", "pulse_len", "num_ranges", "first_range", "intt", "intn", "beam_angle", "beam_order"

The following are required depending on slice type: "txfreq", "rxfreq", "clrfrqrange"

Param
exp_slice: a slice to setup

Returns
complete_slice : a checked slice with all defaults

slice_beam_directions_mapping(*slice_id*)

A mapping of the beam directions in the given slice id.

Parameters
slice_id – id of the slice to get beam directions for.

Returns mapping
enumeration mapping dictionary of beam number to beam direction(s) in degrees off boresight.

property slice_dict

The dictionary of slices.

The slice dictionary can be updated in add_slice, edit_slice, and del_slice. The slice dictionary is a dictionary of dictionaries that looks like:

{ slice_id1 : {slice_key1 : x, slice_key2 : y, ... }, slice_id2 : {slice_key1 : x, slice_key2 : y, ... }, ... }

property slice_ids

The list of slice ids that are currently available in this experiment.

This can change when add_slice, edit_slice, and del_slice are called.

property slice_keys

The list of slice keys available.

This cannot be updated. These are the keys in the current ExperimentPrototype slice_keys dictionary (the parameters available for slices).

property transmit_metadata

A dictionary of config options and experiment-set values that cannot change in the experiment, that will be used to build pulse sequences.

property tx_bandwidth

The transmission sample rate to the DAC (Hz), and the transmit bandwidth.

This is read-only once established in instantiation.

property tx_maxfreq

The maximum transmit frequency.

This is the maximum tx frequency possible in this experiment (either maximum in our license or maximum given by the center frequency, and sampling rate). The maximum is slightly less than that allowed by the center frequency and txrate, to stay away from the edges of the possible transmission band where the signal is distorted.

property tx_minfreq

The minimum transmit frequency.

This is the minimum tx frequency possible in this experiment (either minimum in our license or minimum given by the center frequency and sampling rate). The minimum is slightly more than that allowed by the center frequency and txrate, to stay away from the edges of the possible transmission band where the signal is distorted.

property txctrfreq

The transmission center frequency that USRP is tuned to (kHz).

property txrate

The transmission sample rate to the DAC (Hz).

This is read-only once established in instantiation.

property xcf

The default cross-correlation flag boolean.

This provides the default for slices where this key isn't specified.

experiment_prototype.experiment_prototype.**hidden_key_set = frozenset({'clrfrqflag', 'rxonly', 'slice_interfacing'})**

These are used by the build_scans method (called from the experiment_handler every time the experiment is run). If set by the user, the values will be overwritten and therefore ignored.

experiment_prototype.experiment_prototype.**interface_types = ('SCAN', 'INTTIME', 'INTEGRATION', 'PULSE')**

The types of interfacing available for slices in the experiment.

Interfacing in this case refers to how two or more components are meant to be run together. The following types of interfacing are possible:

1. SCAN. The scan by scan interfacing allows for slices to run a scan of one slice, followed by a scan of the second. The scan mode of interfacing typically means that the slice will cycle through all of its beams before switching to another slice.

There are no requirements for slices interfaced in this manner.

2. INTTIME. This type of interfacing allows for one slice to run its integration period (also known as integration time or averaging period), before switching to another slice's integration period. This type of interface effectively creates an interleaving scan where the scans for multiple slices are run 'at the same time', by interleaving the integration times.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.

3. INTEGRATION. Integration interfacing allows for pulse sequences defined in the slices to alternate between each other within a single integration period. It's important to note that data from a single slice is averaged only with other data from that slice. So in this case, the integration period is running two slices and can produce two averaged datasets, but the sequences (integrations) within the integration period are interleaved.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.
- the same INTT or INTN value.
- the same BEAM_ORDER length (scan length)

4. PULSE. Pulse interfacing allows for pulse sequences to be run together concurrently. Slices will have their pulse sequences summed together so that the data transmits at the same time. For example, slices of different frequencies can be mixed simultaneously, and slices of different pulse sequences can also run together at the cost of having more blanked samples. When slices are interfaced in this way the radar is truly transmitting and receiving the slices simultaneously.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.
- the same INTT or INTN value.
- the same BEAM_ORDER length (scan length)

`experiment_prototype.experiment_prototype.slice_key_set = frozenset({'acf', 'acfint', 'averaging_method', 'beam_angle', 'beam_order', 'clrfrqrange', 'comment', 'cpid', 'first_range', 'intn', 'intt', 'iwavetable', 'lag_table', 'num_ranges', 'pulse_len', 'pulse_phase_offset', 'pulse_sequence', 'qwavetable', 'range_sep', 'rx_int_antennas', 'rx_main_antennas', 'rxfreq', 'scanbound', 'seqoffset', 'slice_id', 'tau_spacing', 'tx_antennas', 'txfreq', 'wavetype', 'xcf'})`

These are the keys that are set by the user when initializing a slice. Some are required, some can be defaulted, and some are set by the experiment and are read-only.

Slice Keys Required by the User

pulse_sequence ***required***
: The pulse sequence timing, given in quantities of tau_spacing, for example normalscan = [0, 14, 22, 24, 27, 31, 42, 43].

tau_spacing ***required***
: multi-pulse increment (mpinc) in us, Defines minimum space between pulses.

pulse_len ***required***
: length of pulse in us. Range gate size is also determined by this.

num_ranges ***required***
Number of range gates.

first_range ***required***
first range gate, in km

intt ***required or intn required***
duration of an integration, in ms. (maximum)

intn ***required or intt required***
number of averages to make a single integration, only used if intt = None.

beam_angle ***required***
list of beam directions, in degrees off azimuth. Positive is E of N. The beam_angle list length = number of beams. Traditionally beams have been 3.24 degrees separated but we don't refer to them as beam -19.64 degrees, we refer as beam 1, beam 2. Beam 0 will be the 0th element in the list, beam 1 will be the 1st, etc. These beam numbers are needed to write the beam_order list. This is like a mapping of beam number (list index) to beam direction off boresight.

beam_order ***required***
beam numbers written in order of preference, one element in this list corresponds to one integration period. Can have lists within the list, resulting in multiple beams running simultaneously in the averaging period, so imaging. A beam number of 0 in this list gives us the direction of the 0th element in the beam_angle list. It is up to the writer to ensure their beam pattern makes sense. Typically beam_order is just in order (scanning W to E or E to W, ie. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. You can list numbers multiple times in the beam_order list, for example [0, 1, 1, 2, 1] or use multiple beam numbers in a single integration time (example [[0, 1], [3, 4]], which would trigger an imaging integration. When we do imaging we will still have to quantize the directions we are looking in to certain beam directions.

clrfrqrange ***required or txfreq or rxfreq required***
range for clear frequency search, should be a list of length = 2, [min_freq, max_freq] in kHz. **Not currently supported.**

txfreq ***required or clrfrqrange or rxfreq required***
transmit frequency, in kHz. Note if you specify clrfrqrange it won't be used.

rxfreq ***required or clrfrqrange or txfreq required***
receive frequency, in kHz. Note if you specify clrfrqrange or txfreq it won't be used. Only necessary to specify if you want a receive-only slice.

Defaultable Slice Keys

acf ***defaults***
flag for rawacf and generation. The default is False. If True, the following fields are also used: - averaging_method (default 'mean') - xcf (default True if acf is True) - acfint (default True if acf is True) - lagtable (default built based on all possible pulse combos) - range_sep (will be built by pulse_len to verify any provided value)

acfint ***defaults***
flag for interferometer autocorrelation data. The default is True if acf is True, otherwise False.

averaging_method ***defaults***
a string defining the type of averaging to be done. Current methods are 'mean' or 'median'. The default is 'mean'.

comment ***defaults***
a comment string that will be placed in the borealis files describing the slice. Defaults to empty string.

lag_table ***defaults***
used in acf calculations. It is a list of lags. Example of a lag: [24, 27] from 8-pulse normalscan. This

defaults to a lagtable built by the pulse sequence provided. All combinations of pulses will be calculated, with both the first pulses and last pulses used for lag-0.

pulse_phase_offset ***defaults***
Allows phase shifting of pulses, enabling encoding of pulses. Default all zeros for all pulses in pulse_sequence. Pulses can be shifted with a single phase shift for each pulse or with a phase shift specified for each sample in the pulses of the slice.

range_sep ***defaults***
a calculated value from pulse_len. If already set, it will be overwritten to be the correct value determined by the pulse_len. Used for acfs. This is the range gate separation, in azimuthal direction, in km.

rx_int_antennas ***defaults***
The antennas to receive on in interferometer array, default is all antennas given max number from config.

rx_main_antennas ***defaults***
The antennas to receive on in main array, default is all antennas given max number from config.

scanbound ***defaults***
A list of seconds past the minute for integration times in a scan to align to. Defaults to None, not required.

seqoffset ***defaults***
offset in us that this slice's sequence will begin at, after the start of the sequence. This is intended for PULSE interfacing, when you want multiple slice's pulses in one sequence you can offset one slice's sequence from the other by a certain time value so as to not run both frequencies in the same pulse, etc. Default is 0 offset.

tx_antennas ***defaults***
The antennas to transmit on, default is all main antennas given max number from config.

xcf ***defaults***
flag for cross-correlation data. The default is True if acf is True, otherwise False.

Read-only Slice Keys

clrfrqflag ***read-only***
A boolean flag to indicate that a clear frequency search will be done. **Not currently supported.**

cpid ***read-only***
The ID of the experiment, consistent with existing radar control programs. This is actually an experiment-wide attribute but is stored within the slice as well. This is provided by the user but not within the slice, instead when the experiment is initialized.

rx_only ***read-only***
A boolean flag to indicate that the slice doesn't transmit, only receives.

slice_id ***read-only***
The ID of this slice object. An experiment can have multiple slices. This is not set by the user but instead set by the experiment when the slice is added. Each slice id within an experiment is unique. When experiments start, the first slice_id will be 0 and incremented from there.

slice_interfacing ***read-only***
A dictionary of slice_id : interface_type for each sibling slice in the experiment at any given time.

Not currently supported and will be removed

wavetype ***defaults***
string for wavetype. The default is SINE. **Not currently supported.**

iwavetable ***defaults***
a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

qwavetable *defaults*
a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

experiment_exception

This is the exception that is raised when there are problems with the experiment that cannot be remedied using experiment_prototype methods.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

exception experiment_prototype.experiment_exception.**ExperimentException**(*message*, **args*)

Bases: Exception

Is raised for the exception where an experiment cannot be run due to setup errors.

list_tests

Basic tests for use in checking slices.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

experiment_prototype.list_tests.**has_duplicates**(*list_to_check*)

Check if the list has duplicate values.

Parameters
list_to_check – A list to check.

Returns
boolean True if duplicates exist, False if not.

experiment_prototype.list_tests.**is_increasing**(*list_to_check*)

Check if list is increasing.

Parameters
list_to_check – a list of numbers

Returns
boolean True if is increasing, False if not.

Subpackages

experiment_prototype.scan_classes package

scan_class_base

This is the base module for all ScanClassBase types (iterable for an experiment given certain parameters). These types include the Scan class, the AveragingPeriod class, and the Sequence class.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

class experiment_prototype.scan_classes.scan_class_base.**ScanClassBase**(*object_keys*, *object_slice_dict*, *object_interface*, *transmit_metadata*)

Bases: object

The base class for the classes Scan, AveragingPeriod, and Sequence.

Scans are made up of AveragingPeriods, these are typically a 3sec time of the same pulse sequence pointing in one direction. AveragingPeriods are made up of Sequences, typically the same sequence run ave. 20-30 times after a clear frequency search. Sequences are made up of pulses, which is a list of dictionaries where each dictionary describes a pulse.

Parameters

- **object_keys** – list of slice_ids that need to be included in this scan_class_base type.
- **object_slice_dict** – the slice dictionary that explains the parameters of each slice that is included in this scan_class_base type. Keys are the slice_ids included and values are dictionaries including all necessary slice parameters as keys.
- **object_interface** – the interfacing dictionary that describes how to interface the slices that are included in this scan_class_base type. Keys are tuples of format (slice_id_1, slice_id_2) and values are of interface_types set up in experiment_prototype.
- **transmit_metadata** – a dictionary of the experiment-wide transmit metadata for building

pulse sequences. The keys of the transmit_metadata are:
'output_rx_rate' [Hz], 'main_antenna_count', 'tr_window_time' [s], 'main_antenna_spacing' [m], 'pulse_ramp_time' [s], 'max_usrp_dac_amplitude' [V peak], 'rx_sample_rate' [Hz], 'minimum_pulse_separation' [us], 'txctrfreq' [kHz], 'txrate' [Hz]

prep_for_nested_scan_class()

Retrieve the params needed for the nested class (also with base ScanClassBase).

This class reduces duplicate code by breaking down the ScanClassBase class into the separate portions for the nested instances. For Scan class, the nested class is AveragingPeriod, and we will need to break down the parameters given to the Scan instance because there may be multiple AveragingPeriods within. For AveragingPeriod, the nested class is Sequence.

Returns
params for the nested class's instantiation.

static slice_combos_sorter(*list_of_combos*, *all_keys*)

Sort keys of a list of combinations so that keys only appear once in the list.

This function modifes the input list_of_combos so that all slices that are associated are associated in the same list. For example, if input is list_of_combos = [[0,1], [0,2], [0,4], [1,4], [2,4]] and all_keys = [0,1,2,4,5] then the output should be [[0,1,2,4], [5]]. This is used to get the slice dictionary for nested class instances. In the above example, we would then have two instances of the nested class to create: one with slices 0,1,2,4 and another with slice 5.

Parameters

- **list_of_combos** – list of lists of length two associating two slices together.
- **all_keys** – list of all keys included in this object (scan, ave_period, or sequence).

Returns

list of combos that is sorted so that each key only appears once and the lists within the list are of however long necessary

scans

This is the module containing the Scan class. The Scan class contains the ScanClassBase members, as well as a scanbound (to be implemented), a beamdir dictionary and scan_beams dictionary which specify beam direction angle and beam order in a scan, respectively, for individual slices that are to be combined in this scan. Beam direction information gets passed on to the AveragingPeriod.

copyright

2018 SuperDARN Canada

author

Marci Detwiller

class experiment_prototype.scan_classes.scans.**Scan**(*scan_keys*, *scan_slice_dict*, *scan_interface*, *transmit_metadata*)

Bases: *ScanClassBase*

Set up the scans.

A scan is made up of AveragingPeriods at defined beam directions, and some other metadata for the scan itself.

The unique members of the scan are (not a member of the scanclassbase):

scanbound

A list of seconds past the minute for scans to align to.

get_inttime_slice_ids()

Return the slice_ids that are within the AveragingPeriods in this Scan instance.

Take the interface keys inside this scan and return a list of lists where each inner list contains the slices that are in an averagingperiod that is inside this scan. ie. len(nested_slice_list) = # of averagingperiods in this scan, len(nested_slice_list[0]) = # of slices in the first averagingperiod, etc.

Returns

the nested_slice_list which is used when creating the AveragingPeriods for this scan.

prep_for_nested_scan_class()

Override of base method to give more information about beamorder and beamdir.

Beam order and beamdir are required for instantiation of the nested class AveragingPeriod so we need to extract this information as well to fill self.aveperiods.

Returns
a list of lists of parameters that can be directly passed into the nested ScanClassBase type, AveragingPeriod. the params_list is of length = # of AveragingPeriods in this scan.

averaging_periods

This is the module containing the AveragingPeriod class. The AveragingPeriod class contains the ScanClassBase members, as well as clrfrqflag (to be implemented), intn (number of integrations to run), or intt(max time for integrations), and it contains sequences of class Sequence.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

class experiment_prototype.scan_classes.averaging_periods.**AveragingPeriod**(*ave_keys*, *ave_slice_dict*, *ave_interface*, *transmit_metadata*, *slice_to_beamorder_dict*, *slice_to_beamdir_dict*)

Bases: *ScanClassBase*

Set up the AveragingPeriods.

An averagingperiod contains sequences and integrates one or multiple pulse sequences together in a given time frame or in a given number of averages, if that is the preferred limiter.

The unique members of the averagingperiod are (not a member of the scanclassbase):

slice_to_beamorder
passed in by the scan that this AveragingPeriod instance is contained in. A dictionary of slice: beam_order for all slices contained in this aveperiod.

slice_to_beamdir
passed in by the scan that this AveragingPeriod instance is contained in. A dictionary of slice: beamdir(s) for all slices contained in this aveperiod.

clrfrqflag
Boolean, True if clrfrqsearch should be performed.

clrfrqrange
The range of frequency to search if clrfrqflag is True. Otherwise empty.

intt
The priority limitation. The time limit (ms) at which time the aveperiod will end. If None, we will use intn to end the aveperiod (a number of sequences).

intn
Number of averages (# of times the sequence transmits) to end after for the averagingperiod.

sequences
The list of sequences included in this aveperiod. This does not indicate how many averages will be transmitted in the aveperiod. If there are multiple sequences in the list, they will be alternated between until the end of the aveperiod.

one_pulse_only
boolean, True if this averaging period only has one unique set of pulse samples in it. This is true if there is only one sequence in the averaging period, and all pulses after the first pulse in the sequence have the

isarepeat key = True. This boolean can be used to speed up the process of sending data to the driver which means we can get more averages in less time.

build_sequences(*slice_to_beamdir_dict*)

Build a list of sequences to iterate through when transmitting.

This includes building all pulses within the sequences, so it then contains all pulse samples data to iterate through when transmitting. If there is only one sequence type in the averaging period, this list will be of length 1. That would mean that that one sequence gets repeated throughout the averagingperiod (intn and intt still apply).

Returns
sequence_dict_list, list of lists of pulse dictionaries.

get_sequence_slice_ids()

Return the slice_ids that are within the Sequences in this AveragingPeriod instance.

Take the interface keys inside this averagingperiod and return a list of lists where each inner list contains the slices that are in a sequence that is inside this averagingperiod. ie. len(nested_slice_list) = # of sequences in this averagingperiod, len(nested_slice_list[0]) = # of slices in the first sequence, etc.

Returns
the nested_slice_list which is used when creating the sequences in this averagingperiod.

set_beamdirdict(*beamiter*)

Get a dictionary of 'slice_id' : 'beamdir(s)' for this averaging period.

At a given beam iteration, this averagingperiod instance will select the beam directions that it will shift to.

Parameters
beamiter – the index into the beam_order list, or the index of an averaging period into the scan

Returns
dictionary of slice to beamdir where beamdir is always a list (may be of length one though). Beamdir is azimuth angle.

sequences

This is the module containing the Sequence class. The Sequence class contains the ScanClassBase members, as well as a list of pulse dictionaries, the total_combined_pulses in the sequence, power_divider, last_pulse_len, ssdelay, seqtime, which together give sstime (scope synce time, or time for receiving, and numberofreceivesamples to sample during the receiving window (calculated using the receive sampling rate).

copyright
2018 SuperDARN Canada

author
Marci Detwiller

class experiment_prototype.scan_classes.sequences.**Sequence**(*seqn_keys*, *sequence_slice_dict*, *sequence_interface*, *transmit_metadata*)

Bases: *ScanClassBase*

Set up the sequence class.

The members of the sequence are:

pulses
a list of pre-combined, pre-sampled pulse dictionaries (one dictionary = one basic pulse of single frequency). The dictionary keys are: isarepeat, pulse_timing_us, slice_id, slice_pulse_index, pulse_len, intra_pulse_start_time, combined_pulse_index, pulse_shift, iscombined, combine_total, and combine_index.

total_combined_pulses
the total number of pulses to be sent by the driver. This may not be the sum of pulses in all slices in the sequence, as some pulses may need to be combined because they are overlapping in timing. This is the number of pulses in the combined sequence, or the number of times T/R signal goes high in the sequence.

power_divider
the power ratio per slice. If there are multiple slices in the same pulse then we must reduce the output amplitude to potentially accommodate multiple frequencies.

last_pulse_len
the length of the last pulse (us)

ssdelay
delay past the end of the sequence to receive for (us) - function of num_ranges and pulse_len. ss stands for scope sync.

seqtime
the amount of time for the whole sequence to transmit, until the logic signal switches low on the last pulse in the sequence (us).

sstime
ssdelay + seqtime (total time for receiving) (us).

numberofreceivesamples
the number of receive samples to take, given the rx rate, during the sstime.

first_rx_sample_time
The location of the first sample for the RX data, in time, from the start of the TX data. This will be calculated as the time at center sample of the first pulse. In seconds.

blanks
A list of sample indices that should not be used for acfs because they were samples taken when transmitting.

Pulses is a list of pulse dictionaries. The pulse dictionary keys are:

isarepeat
Boolean, True if the pulse is exactly the same as the last pulse in the sequence.

pulse_timing_us
The time past the start of sequence for this pulse to start at (us).

slice_id
The slice_id that corresponds to this pulse and gives the information about the experiment and pulse information (frequency, num_ranges, first_range, etc.).

slice_pulse_index
The index of the pulse in its own slice's sequence.

pulse_len
The length of the pulse (us)

intra_pulse_start_time
If the pulse is combined with another pulse and they transmit in a single USRP burst, then we need to know if there is an offset from one pulse's samples being sent and the other pulse's samples being sent.

combined_pulse_index
The combined_pulse_index is the index corresponding with actual number of pulses that will be sent to driver, after combinations are completed. Multiple pulse dictionaries in self.pulses can have the same combined_pulse_index if they are combined together, ie are close enough in timing that T/R will not go low between them, and we will combine the samples of both pulses into one set to send to the driver.

pulse_shift
Phase shift for this pulse, for doing pulse coding.

iscombined
Boolean, true if there is another pulse with the same combined_pulse_index.

combine_total
Total number of pulse dictionaries that have the same combined_pulse_index as this one. (minimum number = 1, itself).

combine_index
Index of this pulse dictionary in regards to all the other pulse dictionaries that have the same combined_pulse_index.

`build_pulse_transmit_data`(*slice_to_beamdir_dict*)

Build a list of ready-to-transmit pulse dictionaries (with samples) to send to driver.

Param
slice_to_beamdir_dict: dictionary of slice id to beam direction(s) for a single averaging period (i.e. if the list len > 1, we're imaging).

Returns sequence_list
list of combined pulse dictionaries in correct order. The keys in the ready-to-transmit pulse dictionary are:

startofburst
Boolean, True if this is the first pulse in the sequence.

endofburst
Boolean, True if this is the last pulse in the sequence.

pulse_antennas
The antennas to transmit on

samples_array
a list of arrays - each array corresponds to an antenna (the samples are phased). All arrays are the same length for a single pulse on that antenna. The length of the list is equal to main_antenna_count (all samples are calculated). If we are not using an antenna, that index is a numpy array of zeroes.

timing
The time to send the pulse at (past the start of sequence, us)

isarepeat
Boolean, True if this pulse is the same as the last pulse except for its timing.

`find_blanks`()

Sets the blanks. Must be run after first_rx_sample_time is set inside the build_pulse_transmit_data function. Called from inside the build_pulse_transmit_data function.

7.2 Experiment Components

7.2.1 experiment_prototype package

The experiment_prototype package contains the building blocks of experiments, which includes the ExperimentPrototype base class, the scan_classes subpackage including the ScanClassBase classes, and the ExperimentException. There is also a list_tests module which is used by the ExperimentPrototype class.

Submodules

experiment_prototype

This is the base module for all experiments. An experiment will only run if it inherits from this class.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

`class experiment_prototype.experiment_prototype.ExperimentPrototype`

Bases: `object`

The base class for all experiments.

A prototype experiment class composed of metadata, including experiment slices (exp_slice) which are dictionaries of radar parameters. Basic, traditional experiments will be composed of a single slice. More complicated experiments will be composed of multiple slices that interface in one of four pre-determined ways, as described under interface_types.

This class is used via inheritance to create experiments.

Some variables shouldn't be changed by the experiment, and their properties do not have setters. Some variables can be changed in the init of your experiment, and can also be modified in-experiment by the class method 'update' in your experiment class. These variables have been given property setters.

The following are the user-modifiable attributes of the ExperimentPrototype that are used to make an experiment:

- xcf: boolean for cross-correlation data. A default can be set here for slices, but any slice can override this setting with the xcf slice key.
- acf: boolean for auto-correlation data on main array. A default can be set here for slices, but any slice can override this setting with the acf slice key.
- acfint: boolean for auto-correlation data on interferometer array. A default can be set here for slices, but any slice can override this setting with the acfint slice key.
- slice_dict: modifiable only using the add_slice, edit_slice, and del_slice methods.
- interface: modifiable using the add_slice, edit_slice, and del_slice methods, or by updating the interface dict directly.

Other parameters are set in the init and cannot be modified after instantiation.

`property acf`

The default auto-correlation flag boolean.

This provides the default for slices where this key isn't specified.

property acfint

The default interferometer autocorrelation boolean.

This provides the default for slices where this key isn't specified.

add_slice(*exp_slice*, *interfacing_dict={}*)

Add a slice to the experiment.

Parameters

- **exp_slice** – a slice (dictionary of slice_keys) to add to the experiment.
- **interfacing_dict** – dictionary of type {slice_id : INTERFACING , ... } that defines how this slice interacts with all the other slices currently in the experiment.

Raises
ExperimentException if slice is not a dictionary or if there are errors in setup_slice.

Returns
the slice_id of the new slice that was just added.

build_scans()

Build the scan information, which means creating the Scan, AveragingPeriod, and Sequence instances needed to run this experiment.

Will be run by experiment handler, to build iterable objects for radar_control to use. Creates scan_objects in the experiment for identifying which slices are in the scans.

check_new_slice_interfacing(*interfacing_dict*)

Checks that the new slice plays well with its siblings (has interfacing that is resolvable). If so, returns a new dictionary with all interfacing values set.

The interfacing assumes that the interfacing_dict given by the user defines the closest interfacing of the new slice with a slice. For example, if the slice is to be PULSE combined with slice 0, the interfacing dict should provide this information. If only 'SCAN' interfacing with slice 1 is provided, then that will be assumed to be the closest and therefore the interfacing with slice 0 will also be 'SCAN'.

If no interfacing_dict is provided for a slice, the default is to do 'SCAN' type interfacing for the new slice with all other slices.

Parameters
interfacing_dict – the user-provided interfacing dict, which may be empty or incomplete. If empty, all interfacing is assumed to be = 'SCAN' type. If it contains something, we ensure that the interfacing provided makes sense with the values already known for its closest sibling.

Returns
full interfacing dictionary.

Raises
ExperimentException if invalid interface types provided or if interfacing can not be resolved.

check_slice(*exp_slice*)

Check the slice for errors.

This is the first test of the dictionary in the experiment done to ensure values in this slice make sense. This is a self-check to ensure the parameters (for example, txfreq, antennas) are appropriate. All fields should be full at this time (whether filled by the user or given default values in set_slice_defaults). This was built to be useable at any time after setup. :param: exp_slice: a slice to check :raise: ExperimentException: When necessary parameters do not exist or = None (would have to have been overridden by the user for this, as defaults all set when this runs).

check_slice_minimum_requirements(*exp_slice*)

Check the required slice keys.

Check for the minimum requirements of the slice. The following keys are always required: "pulse_sequence", "tau_spacing", "pulse_len", "num_ranges", "first_range", (one of "intt" or "intn"), "beam_angle", and "beam_order". This function may modify the values in this slice dictionary to ensure that it is able to be run and that the values make sense.

Parameters
exp_slice – slice to check.

check_slice_specific_requirements(*exp_slice*)

Set the specific slice requirements depending.

Check the requirements for the specific slice type as identified by the identifiers rxonly and clrfrqflag. The keys that need to be checked depending on these identifiers are "txfreq", "rxfreq", and "clrfrqrange". This function may modify these keys.

Parameters
exp_slice – the slice to check, before adding to the experiment.

property comment_string

A string related to the experiment, to be placed in the experiment's files.

This is read-only once established in instantiation.

property cpid

This experiment's CPID (control program ID, a term that comes from ROS).

The CPID is read-only once established in instantiation. It may be modified at runtime by the set_scheduling_mode function, to set it to a negative value during discretionary time.

property decimation_scheme

The decimation scheme, of type DecimationScheme from the filtering module. Includes all filtering and decimating information for the signal processing module.

This is read-only once established in instantiation.

del_slice(*remove_slice_id*)

Remove a slice from the experiment.

Parameters
remove_slice_id – the id of the slice you'd like to remove.

Returns
a copy of the removed slice.

Raises
exception if remove_slice_id does not exist in the slice dictionary.

edit_slice(*edit_slice_id*, ***kwargs*)

Edit a slice.

A quick way to edit a slice. In reality this is actually adding a new slice and deleting the old one. Useful for quick changes. Note that using this function will remove the slice_id that you are changing and will give it a new id. It will account for this in the interfacing dictionary.

Parameters

- **edit_slice_id** – the slice id of the slice to be edited.
- **kwargs** – dictionary of slice parameter to slice value that you want to change.

Returns new_slice_id
the new slice id of the edited slice, or the edit_slice_id if no change has occurred due to failure of new slice parameters to pass experiment checks.

Raises
exceptions if the edit_slice_id does not exist in slice dictionary or the params or values do not make sense.

`property experiment_name`

The experiment class name.

`get_scan_slice_ids()`

Organize the slice_ids by scan.

Take my own interfacing and get info on how many scans and which slices make which scans. Return a list of lists where each inner list contains the slices that are in an averagingperiod that is inside this scan. ie. len(nested_slice_list) = # of averagingperiods in this scan, len(nested_slice_list[0]) = # of slices in the first averagingperiod, etc.

:return list of lists. The list has one element per scan. Each element is a list of slice_ids signifying which slices are combined inside that scan. The list returned could be of length 1, meaning only one scan is present in the experiment.

`get_slice_interfacing(slice_id)`

Check the experiment's interfacing dictionary for all interfacing that pertains to a given slice, and return the interfacing information in a dictionary. :param slice_id: Slice ID to search the interface dictionary for. :return: interfacing dictionary for the slice.

`property interface`

The dictionary of interfacing for the experiment slices.

Interfacing should be set up for any slice when it gets added, ie. in add_slice, except for the first slice added. The dictionary of interfacing is setup as:

[(slice_id1, slice_id2) : INTERFACING_TYPE, (slice_id1, slice_id3) : INTERFACING_TYPE, ...]

for all current slice_ids.

`property new_slice_id`

The next unique slice id that is available to this instance of the experiment.

This gets incremented each time it is called to ensure it returns a unique ID each time.

`property num_slices`

The number of slices currently in the experiment.

Will change after methods add_slice or del_slice are called.

`property options`

The config options for running this experiment.

These cannot be set or removed, but are specified in the config.ini, hdw.dat, and restrict.dat files.

`property output_rx_rate`

The output receive rate of the data, Hz.

This is read-only once established in instantiation.

`printing(msg)`

property rx_bandwidth

The receive bandwidth for this experiment, in Hz.

This is read-only once established in instantiation.

property rx_maxfreq

The maximum receive frequency.

This is the maximum tx frequency possible in this experiment (maximum given by the center frequency and sampling rate), as license doesn't matter for receiving. The maximum is slightly less than that allowed by the center frequency and rxrate, to stay away from the edges of the possible receive band where the signal may be distorted.

property rx_minfreq

The minimum receive frequency.

This is the minimum rx frequency possible in this experiment (minimum given by the center frequency and sampling rate) - license doesn't restrict receiving. The minimum is slightly more than that allowed by the center frequency and rxrate, to stay away from the edges of the possible receive band where the signal may be distorted.

property rxctrfreq

The receive center frequency that USRP is tuned to (kHz).

property rxrate

The receive bandwidth for this experiment, or the receive sampling rate (of I and Q samples) In Hz.

This is read-only once established in instantiation.

property scan_objects

The list of instances of class Scan for use in radar_control.

These cannot be modified by the user, but are created using the slice dictionary.

property scheduling_mode

Return the scheduling mode time type that this experiment is running in. Types are listed in possible_scheduling_modes. Initialized to 'unknown' until set by the experiment handler.

self_check()

Check that the values in this experiment are valid.

Checks all slices.

set_slice_defaults(*exp_slice*)

Set up defaults in case of some parameters being left blank.

Parameters
exp_slice – slice to set defaults of

Returns slice_with_defaults
updated slice

static set_slice_identifiers(*exp_slice*)

Set the hidden slice keys to determine how to run the slice.

This function sets up internal identifier flags 'clrfrqflag' and 'rxonly' in the slice so that we know how to properly set up the slice and know which keys in the slice must be specified and which are unnecessary. If these keys are ever written by the user, they will be rewritten here.

Parameters
exp_slice – slice in which to set identifiers

`setup_slice`(*exp_slice*)

Check slice for errors and set defaults of optional keys.

Before adding the slice, ensure that the internal parameters are set, remove unnecessary keys and check values of keys that are needed, and set defaults of keys that are optional.

The following are always able to be defaulted, so are optional: "tx_antennas", "rx_main_antennas", "rx_int_antennas", "pulse_phase_offset", "scanboundflag", "scanbound", "acf", "xcf", "acfint", "wavetype", "seqoffset", "averaging_method"

The following are always required for processing acf, xcf, and acfint which we will assume we are always doing: "pulse_sequence", "tau_spacing", "pulse_len", "num_ranges", "first_range", "intt", "intn", "beam_angle", "beam_order"

The following are required depending on slice type: "txfreq", "rxfreq", "clrfrqrange"

Param
exp_slice: a slice to setup

Returns
complete_slice : a checked slice with all defaults

`slice_beam_directions_mapping`(*slice_id*)

A mapping of the beam directions in the given slice id.

Parameters
slice_id – id of the slice to get beam directions for.

Returns mapping
enumeration mapping dictionary of beam number to beam direction(s) in degrees off boresight.

`property slice_dict`

The dictionary of slices.

The slice dictionary can be updated in add_slice, edit_slice, and del_slice. The slice dictionary is a dictionary of dictionaries that looks like:

{ slice_id1 : {slice_key1 : x, slice_key2 : y, ... }, slice_id2 : {slice_key1 : x, slice_key2 : y, ... }, ... }

`property slice_ids`

The list of slice ids that are currently available in this experiment.

This can change when add_slice, edit_slice, and del_slice are called.

`property slice_keys`

The list of slice keys available.

This cannot be updated. These are the keys in the current ExperimentPrototype slice_keys dictionary (the parameters available for slices).

`property transmit_metadata`

A dictionary of config options and experiment-set values that cannot change in the experiment, that will be used to build pulse sequences.

`property tx_bandwidth`

The transmission sample rate to the DAC (Hz), and the transmit bandwidth.

This is read-only once established in instantiation.

property tx_maxfreq

The maximum transmit frequency.

This is the maximum tx frequency possible in this experiment (either maximum in our license or maximum given by the center frequency, and sampling rate). The maximum is slightly less than that allowed by the center frequency and txrate, to stay away from the edges of the possible transmission band where the signal is distorted.

property tx_minfreq

The minimum transmit frequency.

This is the minimum tx frequency possible in this experiment (either minimum in our license or minimum given by the center frequency and sampling rate). The minimum is slightly more than that allowed by the center frequency and txrate, to stay away from the edges of the possible transmission band where the signal is distorted.

property txctrfreq

The transmission center frequency that USRP is tuned to (kHz).

property txrate

The transmission sample rate to the DAC (Hz).

This is read-only once established in instantiation.

property xcf

The default cross-correlation flag boolean.

This provides the default for slices where this key isn't specified.

experiment_prototype.experiment_prototype.**hidden_key_set = frozenset({'clrfrqflag', 'rxonly', 'slice_interfacing'})**

These are used by the build_scans method (called from the experiment_handler every time the experiment is run). If set by the user, the values will be overwritten and therefore ignored.

experiment_prototype.experiment_prototype.**interface_types = ('SCAN', 'INTTIME', 'INTEGRATION', 'PULSE')**

The types of interfacing available for slices in the experiment.

Interfacing in this case refers to how two or more components are meant to be run together. The following types of interfacing are possible:

1. SCAN. The scan by scan interfacing allows for slices to run a scan of one slice, followed by a scan of the second. The scan mode of interfacing typically means that the slice will cycle through all of its beams before switching to another slice.

There are no requirements for slices interfaced in this manner.

2. INTTIME. This type of interfacing allows for one slice to run its integration period (also known as integration time or averaging period), before switching to another slice's integration period. This type of interface effectively creates an interleaving scan where the scans for multiple slices are run 'at the same time', by interleaving the integration times.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.

3. INTEGRATION. Integration interfacing allows for pulse sequences defined in the slices to alternate between each other within a single integration period. It's important to note that data from a single slice is averaged only with other data from that slice. So in this case, the integration period is running two slices and can produce two averaged datasets, but the sequences (integrations) within the integration period are interleaved.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.
- the same INTT or INTN value.
- the same BEAM_ORDER length (scan length)

4. PULSE. Pulse interfacing allows for pulse sequences to be run together concurrently. Slices will have their pulse sequences summed together so that the data transmits at the same time. For example, slices of different frequencies can be mixed simultaneously, and slices of different pulse sequences can also run together at the cost of having more blanked samples. When slices are interfaced in this way the radar is truly transmitting and receiving the slices simultaneously.

Slices which are interfaced in this manner must share:

- the same SCANBOUND value.
- the same INTT or INTN value.
- the same BEAM_ORDER length (scan length)

experiment_prototype.experiment_prototype.**slice_key_set = frozenset({'acf', 'acfint', 'averaging_method', 'beam_angle', 'beam_order', 'clrfrqrange', 'comment', 'cpid', 'first_range', 'intn', 'intt', 'iwavetable', 'lag_table', 'num_ranges', 'pulse_len', 'pulse_phase_offset', 'pulse_sequence', 'qwavetable', 'range_sep', 'rx_int_antennas', 'rx_main_antennas', 'rxfreq', 'scanbound', 'seqoffset', 'slice_id', 'tau_spacing', 'tx_antennas', 'txfreq', 'wavetype', 'xcf'})**

These are the keys that are set by the user when initializing a slice. Some are required, some can be defaulted, and some are set by the experiment and are read-only.

Slice Keys Required by the User

pulse_sequence ***required***
The pulse sequence timing, given in quantities of tau_spacing, for example normalscan = [0, 14, 22, 24, 27, 31, 42, 43].

tau_spacing ***required***
multi-pulse increment (mpinc) in us, Defines minimum space between pulses.

pulse_len ***required***
length of pulse in us. Range gate size is also determined by this.

num_ranges ***required***
Number of range gates.

first_range ***required***
first range gate, in km

intt ***required or intn required***
duration of an integration, in ms. (maximum)

intn ***required or intt required***
number of averages to make a single integration, only used if intt = None.

beam_angle ***required***
list of beam directions, in degrees off azimuth. Positive is E of N. The beam_angle list length = number of beams. Traditionally beams have been 3.24 degrees separated but we don't refer to them as beam -19.64 degrees, we refer as beam 1, beam 2. Beam 0 will be the 0th element in the list, beam 1 will be the 1st, etc. These beam numbers are needed to write the beam_order list. This is like a mapping of beam number (list index) to beam direction off boresight.

beam_order ***required***
beam numbers written in order of preference, one element in this list corresponds to one integration period. Can have lists within the list, resulting in multiple beams running simultaneously in the averaging period, so imaging. A beam number of 0 in this list gives us the direction of the 0th element in the beam_angle list. It is up to the writer to ensure their beam pattern makes sense. Typically beam_order is just in order (scanning W to E or E to W, ie. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. You can list numbers multiple times in the beam_order list, for example [0, 1, 1, 2, 1] or use multiple beam numbers in a single integration time (example [[0, 1], [3, 4]], which would trigger an imaging integration. When we do imaging we will still have to quantize the directions we are looking in to certain beam directions.

clrfrqrange ***required or txfreq or rxfreq required***
range for clear frequency search, should be a list of length = 2, [min_freq, max_freq] in kHz. **Not currently supported.**

txfreq ***required or clrfrqrange or rxfreq required***
transmit frequency, in kHz. Note if you specify clrfrqrange it won't be used.

rxfreq ***required or clrfrqrange or txfreq required***
receive frequency, in kHz. Note if you specify clrfrqrange or txfreq it won't be used. Only necessary to specify if you want a receive-only slice.

Defaultable Slice Keys

acf ***defaults***
flag for rawacf and generation. The default is False. If True, the following fields are also used: - averaging_method (default 'mean') - xcf (default True if acf is True) - acfint (default True if acf is True) - lagtable (default built based on all possible pulse combos) - range_sep (will be built by pulse_len to verify any provided value)

acfint ***defaults***
flag for interferometer autocorrelation data. The default is True if acf is True, otherwise False.

averaging_method ***defaults***
a string defining the type of averaging to be done. Current methods are 'mean' or 'median'. The default is 'mean'.

comment ***defaults***
a comment string that will be placed in the borealis files describing the slice. Defaults to empty string.

lag_table ***defaults***
used in acf calculations. It is a list of lags. Example of a lag: [24, 27] from 8-pulse normalscan. This defaults to a lagtable built by the pulse sequence provided. All combinations of pulses will be calculated, with both the first pulses and last pulses used for lag-0.

pulse_phase_offset ***defaults***
Allows phase shifting of pulses, enabling encoding of pulses. Default all zeros for all pulses in pulse_sequence. Pulses can be shifted with a single phase shift for each pulse or with a phase shift specified for each sample in the pulses of the slice.

range_sep ***defaults***
a calculated value from pulse_len. If already set, it will be overwritten to be the correct value determined by the pulse_len. Used for acfs. This is the range gate separation, in azimuthal direction, in km.

rx_int_antennas ***defaults***
The antennas to receive on in interferometer array, default is all antennas given max number from config.

rx_main_antennas ***defaults***
The antennas to receive on in main array, default is all antennas given max number from config.

scanbound ***defaults***
A list of seconds past the minute for integration times in a scan to align to. Defaults to None, not required.

seqoffset *defaults*
offset in us that this slice's sequence will begin at, after the start of the sequence. This is intended for PULSE interfacing, when you want multiple slice's pulses in one sequence you can offset one slice's sequence from the other by a certain time value so as to not run both frequencies in the same pulse, etc. Default is 0 offset.

tx_antennas *defaults*
The antennas to transmit on, default is all main antennas given max number from config.

xcf *defaults*
flag for cross-correlation data. The default is True if acf is True, otherwise False.

Read-only Slice Keys

clrfrqflag *read-only*
A boolean flag to indicate that a clear frequency search will be done. **Not currently supported.**

cpid *read-only*
The ID of the experiment, consistent with existing radar control programs. This is actually an experiment-wide attribute but is stored within the slice as well. This is provided by the user but not within the slice, instead when the experiment is initialized.

rx_only *read-only*
A boolean flag to indicate that the slice doesn't transmit, only receives.

slice_id *read-only*
The ID of this slice object. An experiment can have multiple slices. This is not set by the user but instead set by the experiment when the slice is added. Each slice id within an experiment is unique. When experiments start, the first slice_id will be 0 and incremented from there.

slice_interfacing *read-only*
A dictionary of slice_id : interface_type for each sibling slice in the experiment at any given time.

Not currently supported and will be removed

wavetype *defaults*
string for wavetype. The default is SINE. **Not currently supported.**

iwavetable *defaults*
a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

qwavetable *defaults*
a list of numeric values to sample from. The default is None. Not currently supported but could be set up (with caution) for non-SINE. **Not currently supported.**

experiment_exception

This is the exception that is raised when there are problems with the experiment that cannot be remedied using experiment_prototype methods.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

exception experiment_prototype.experiment_exception.**ExperimentException**(*message*, **args*)

Bases: Exception

Is raised for the exception where an experiment cannot be run due to setup errors.

list_tests

Basic tests for use in checking slices.

copyright
2018 SuperDARN Canada

author
Marci Detwiller

`experiment_prototype.list_tests.`**`has_duplicates`**(*list_to_check*)

Check if the list has duplicate values.

Parameters
`list_to_check` – A list to check.

Returns
boolean True if duplicates exist, False if not.

`experiment_prototype.list_tests.`**`is_increasing`**(*list_to_check*)

Check if list is increasing.

Parameters
`list_to_check` – a list of numbers

Returns
boolean True if is increasing, False if not.

Subpackages

7.2.2 experiments package

This is where you would create your experiment that you would like to run on the radar. The following are a couple of examples of current SuperDARN experiments, and a brief discussion of the update() method which will be implemented at a later date.

experiments.normalscan module

Normalscan is a very common experiment for SuperDARN. It does not update itself, so no update() method is necessary. It only has a single slice, as there is only one frequency, pulse_len, beam_order, etc. Since there is only one slice there is no need for an interface dictionary.

```
#!/usr/bin/python

# write an experiment that creates a new control program.

import sys
import os

BOREALISPATH = os.environ['BOREALISPATH']
sys.path.append(BOREALISPATH)

import experiments.superdarn_common_fields as scf
from experiment_prototype.experiment_prototype import ExperimentPrototype

```

(continues on next page)

(continued from previous page)

```
class Normalscan(ExperimentPrototype):

    def __init__(self):
        cpid = 151
        super(Normalscan, self).__init__(cpid)

        if scf.IS_FORWARD_RADAR:
            beams_to_use = scf.STD_16_FORWARD_BEAM_ORDER
        else:
            beams_to_use = scf.STD_16_REVERSE_BEAM_ORDER

        if scf.opts.site_id in ["cly", "rkn", "inv"]:
            num_ranges = scf.POLARDARN_NUM_RANGES
        if scf.opts.site_id in ["sas", "pgr"]:
            num_ranges = scf.STD_NUM_RANGES

        self.add_slice({  # slice_id = 0, there is only one slice.
            "pulse_sequence": scf.SEQUENCE_7P,
            "tau_spacing": scf.TAU_SPACING_7P,
            "pulse_len": scf.PULSE_LEN_45KM,
            "num_ranges": num_ranges,
            "first_range": scf.STD_FIRST_RANGE,
            "intt": 3500,  # duration of an integration, in ms
            "beam_angle": scf.STD_16_BEAM_ANGLE,
            "beam_order": beams_to_use,
            "scanbound": [i * 3.5 for i in range(len(beams_to_use))], #1 min scan
            "txfreq" : scf.COMMON_MODE_FREQ_1, #kHz
            "acf": True,
            "xcf": True,  # cross-correlation processing
            "acfint": True,  # interferometer acfs
        })

```

experiments.twofsound module

Twofsound is a common variant of the normalscan experiment for SuperDARN. It does not update itself, so no update() method is necessary. It has two frequencies so will require two slices. The frequencies switch after a full scan (full cycle through the beams), therefore the interfacing between slices 0 and 1 should be 'SCAN'.

```
#!/usr/bin/python

# write an experiment that creates a new control program.
import os
import sys
import copy

BOREALISPATH = os.environ['BOREALISPATH']
sys.path.append(BOREALISPATH)

from experiment_prototype.experiment_prototype import ExperimentPrototype
import experiments.superdarn_common_fields as scf
```

(continues on next page)

(continued from previous page)

```

class Twofsound(ExperimentPrototype):

    def __init__(self):
        cpid = 3503

        if scf.IS_FORWARD_RADAR:
            beams_to_use = scf.STD_16_FORWARD_BEAM_ORDER
        else:
            beams_to_use = scf.STD_16_REVERSE_BEAM_ORDER

        if scf.opts.site_id in ["cly", "rkn", "inv"]:
            num_ranges = scf.POLARDARN_NUM_RANGES
        if scf.opts.site_id in ["sas", "pgr"]:
            num_ranges = scf.STD_NUM_RANGES

        slice_1 = {  # slice_id = 0, the first slice
            "pulse_sequence": scf.SEQUENCE_7P,
            "tau_spacing": scf.TAU_SPACING_7P,
            "pulse_len": scf.PULSE_LEN_45KM,
            "num_ranges": num_ranges,
            "first_range": scf.STD_FIRST_RANGE,
            "intt": 3500,  # duration of an integration, in ms
            "beam_angle": scf.STD_16_BEAM_ANGLE,
            "beam_order": beams_to_use,
            "scanbound" : [i * 3.5 for i in range(len(beams_to_use))],
            "txfreq" : scf.COMMON_MODE_FREQ_1, #kHz
            "acf": True,
            "xcf": True,  # cross-correlation processing
            "acfint": True,  # interferometer acfs
        }

        slice_2 = copy.deepcopy(slice_1)
        slice_2['txfreq'] = scf.COMMON_MODE_FREQ_2

        list_of_slices = [slice_1, slice_2]
        sum_of_freq = 0
        for slice in list_of_slices:
            sum_of_freq += slice['txfreq']# kHz, oscillator mixer frequency on the USRP␣
↪for TX
        rxctrfreq = txctrfreq = int(sum_of_freq/len(list_of_slices))


        super(Twofsound, self).__init__(cpid, txctrfreq=txctrfreq, rxctrfreq=rxctrfreq,
                comment_string='Twofsound classic scan-by-scan')

        self.add_slice(slice_1)

        self.add_slice(slice_2, interfacing_dict={0: 'SCAN'})

```

7.3 Utils

7.3.1 radar_status package

radar_status.radar_status module

class radar_status.radar_status.**RadarStatus**

Bases: object

Class to define transmit specifications of a certain frequency, beam, and pulse sequence.

errors = ('EXPNEEDED', 'NOERROR', 'WARNING', 'EXITERROR')

Probably will be phased out once administrator is working

radar_status.radar_status.**errortype**()

radar_status.radar_status.**statustype**()

7.3.2 sample_building package

sample_building.sample_building module

sample_building.sample_building.**calculate_first_rx_sample_time**(*first_pulse_num_samples_with_tr*, *txrate*)

The first rx sample time is in the centre of the first pulse, so find the sample number of that time in the TX data so we can align the samples and offset appropriately in the RX decimated data. Assumes window time for TR is the same at front and end of actual non-zero samples. :param first_pulse_num_samples_with_tr: number of samples in the first pulse. :param txrate: The transmitting sample rate, in Hz. :return: first_rx_sample_time, time to centre of first pulse.

sample_building.sample_building.**calculated_combined_pulse_samples_length**(*pulse_list*, *txrate*)

Get the total length of the array for the combined pulse.

Determine the length of the combined pulse in number of samples before combining the samples, using the length of the samples arrays and the starting sample number for each pulse to combine. (Not all pulse samples may start at sample zero due to differing intra_pulse_start_times.)

Parameters

- **pulse_list** – list of pulse dictionaries that must be combined to one pulse.
- **txrate** – sampling rate of transmission going to DAC.

Returns combined_pulse_length
the length of the pulse after combining slices if necessary.

sample_building.sample_building.**create_debug_sequence_samples**(*txrate*, *txctrfreq*, *list_of_pulse_dicts*, *main_antenna_count*, *final_rx_sample_rate*, *ssdelay*)

Build the samples for the whole sequence, to be recorded in datawrite.

Parameters

- **txrate** – The rate at which these samples will be transmitted at, Hz.

- **txctrfreq** – The centre frequency that the N200 is tuned to (and will mix with these samples, kHz).
- **list_of_pulse_dicts** – The list of all pulse dictionaries for pulses included

in this sequence. Pulse dictionaries have all metadata and the samples for the pulse. :param file_path: location to place the json file. :param main_antenna_count: The number of antennas available for transmitting on. :param final_rx_sample_rate: The final sample rate after decimating on the receive side (Hz). :param ssdelay: Receiver time of flight for last echo. This is the time to continue

receiving after the last pulse is transmitted.

Returns

sample_building.sample_building.**create_uncombined_pulses**(*pulse_list*, *power_divider*, *exp_slices*, *beamdir*, *txrate*, *txctrfreq*, *main_antenna_count*, *main_antenna_spacing*, *pulse_ramp_time*, *max_usrp_dac_amplitude*)

Create the samples for a given pulse_list and append those samples to the pulse_list.

Creates a list of numpy arrays where each numpy array is the pulse samples for a given pulse and a given transmit antenna (index of array in list provides antenna number). Adds the list of samples to the pulse dictionary (in the pulse_list list) under the key 'samples'.

If the antenna is listed in the config but is not used in the sequence, it is provided an array of zeroes to transmit.

Parameters

- **pulse_list** – a list of dictionaries, each dict is a pulse. The list includes all pulses that will be combined together. All dictionaries in this list (all 'pulses') will be modified to include the 'samples' key which will be a list of arrays where every array is a set of samples for a specific antenna.
- **power_divider** – an integer for number of pulses combined (max) in the whole sequence, so we can adjust the amplitude of each uncombined pulse accordingly.
- **exp_slices** – slice dictionary containing all necessary slice_ids for this pulse.
- **beamdir** – the slice to beamdir dictionary to retrieve the phasing information for each antenna in a certain slice's pulses.
- **txrate** – transmit sampling rate, in Hz.
- **txctrfreq** – transmit mixing frequency, in kHz.
- **main_antenna_count** – number of main antennas in the array to transmit.
- **main_antenna_spacing** – spacing between main array antennas, assumed uniform.
- **pulse_ramp_time** – time to ramp up the pulse at the start and end of the pulse. This

time counts as part of the total pulse length time (in seconds). :param max_usrp_dac_amplitude: max voltage out of the digital-analog converter on the USRP

sample_building.sample_building.**get_phshift**(*beamdir*, *freq*, *antenna*, *pulse_shift*, *num_antennas*, *antenna_spacing*, *centre_offset=0.0*)

Find the phase shift for a given antenna and beam direction.

Form the beam given the beam direction (degrees off boresite), the tx frequency, the antenna number, a specified extra phase shift if there is any, the number of antennas in the array, and the spacing between antennas.

Parameters

- **beamdir** – the azimuthal direction of the beam off boresight, in degrees, positive beamdir being to the right of the boresight (looking along boresight from ground). This is for this antenna.
- **freq** – transmit frequency in kHz
- **antenna** – antenna number, INDEXED FROM ZERO, zero being the leftmost antenna if looking down the boresight and positive beamdir right of boresight
- **pulse_shift** – in degrees, for phase encoding
- **num_antennas** – number of antennas in this array
- **antenna_spacing** – distance between antennas in this array, in meters
- **centre_offset** – the phase reference for the midpoint of the array. Default = 0.0, in metres. Important if there is a shift in centre point between arrays in the direction along the array. Positive is shifted to the right when looking along boresight (from the ground).

Returns phshift
a phase shift for the samples for this antenna number, in radians.

sample_building.sample_building.**get_samples**(*rate*, *wave_freq*, *pulse_len*, *ramp_time*, *max_amplitude*, *iwave_table=None*, *qwave_table=None*)

Get basic (not phase-shifted) samples for a given pulse.

Find the normalized sample array given the rate (Hz), frequency (Hz), pulse length (s), and wavetables (list containing single cycle of waveform). Will shift for beam later. No need to use wavetable if just a sine wave.

Parameters

- **rate** – tx sampling rate, in Hz.
- **wave_freq** – frequency offset from the centre frequency on the USRP, given in Hz. To be mixed with the centre frequency before transmitting. (ex. centre = 12 MHz, wave_freq = + 1.2 MHz, output = 13.2 MHz.
- **pulse_len** – length of the pulse (in seconds)
- **ramp_time** – ramp up and ramp down time for the pulse, in seconds. Typical 0.00001 s from config.
- **max_amplitude** – USRP's max DAC amplitude. N200 = 0.707 max
- **iwave_table** – i samples (in-phase) wavetable if a wavetable is required (ie. not a sine wave to be sampled)
- **qwave_table** – q samples (quadrature) wavetable if a wavetable is required (ie. not a sine wave to be sampled)

Returns samples
a numpy array of complex samples, representing all samples needed for a pulse of length pulse_len sampled at a rate of rate.

Returns actual_wave_freq
the frequency possible given the wavetable. If wavetype != 'SINE' (i.e. calculated wavetables were used), then actual_wave_freq may not be equal to the requested wave_freq param.

sample_building.sample_building.**get_wavetables**(*wavetype*)

Find the wavetable to sample from for a given wavetype.

If there are ever any other types of wavetypes besides 'SINE', set them up here.

NOTE: The wavetables should sample a single cycle of the waveform. Note that we will have to block frequencies that could interfere with our license, which will affect the waveform. This blocking of frequencies is not currently set up, so beware. Would have to get the spectrum of the wavetable waveform and then block frequencies that when mixed with the centre frequency, result in the restricted frequencies.

Also NOTE: wavetables create a fixed frequency resolution based on their length. This code is from get_samples:

f_norm = wave_freq / rate

sample_skip = int(f_norm * wave_table_len) # THIS MUST BE AN INT, WHICH DEFINES THE FREQUENCY RESOLUTION.

actual_wave_freq = (float(sample_skip) / float(wave_table_len)) * rate

Parameters
wavetype – A string descriptor of the wavetype.

Returns iwavetable
an in-phase wavetable, or None if given 'SINE' wavetype.

Returns qwavetable
a quadrature wavetable, or None if given 'SINE' wavetype.

sample_building.sample_building.**make_pulse_samples**(*pulse_list*, *power_divider*, *exp_slices*, *slice_to_beamdir_dict*, *txrate*, *txctrfreq*, *main_antenna_count*, *main_antenna_spacing*, *pulse_ramp_time*, *max_usrp_dac_amplitude*, *tr_window_time*)

Make all necessary samples for all antennas for this pulse.

Given a pulse_list (list of dictionaries of pulses that must be combined), make and phase shift samples for all antennas, and combine pulse dictionaries into one pulse if there are multiple waveforms to combine (e.g., multiple frequencies).

Parameters

- **pulse_list** – a list of dictionaries, each dict is a pulse. The list only contains pulses that will be sent as a single pulse (ie. have the same combined_pulse_index).
- **power_divider** – an integer for number of pulses combined (max) in the whole sequence, so we can adjust the amplitude of each uncombined pulse accordingly.
- **exp_slices** – this is the slice dictionary containing the slices necessary for the sequence.
- **slice_to_beamdir_dict** – a dictionary describing the beam directions for the slice_ids.
- **txrate** – transmit sampling rate, in Hz.
- **txctrfreq** – transmit mixing frequency, in kHz.
- **main_antenna_count** – number of main antennas in the array to transmit.
- **main_antenna_spacing** – spacing between main array antennas, assumed uniform.
- **pulse_ramp_time** – time to ramp up the pulse at the start and end of the pulse. This

time counts as part of the total pulse length time (in seconds). :param max_usrp_dac_amplitude: max voltage out of the digital-analog converter on the USRP :param tr_window_time: time in seconds to add zero-samples to the transmit waveform in order to count for the transmit/receive switching time. Windows the pulse on both sides. :returns combined_samples: a list of arrays - each array corresponds to an antenna

(the samples are phased). All arrays are the same length for a single pulse on that antenna. The length of the list is equal to main_antenna_count (all samples are calculated). If we are not using an antenna, that index is a numpy array of zeroes.

Returns pulse_channels
The antennas to actually send the corresponding array. If not all transmit antennas, then we will know that we are transmitting zeroes on any antennas not listed in this list but available as identified in the config file.

sample_building.sample_building.**resolve_imaging_directions**(*beamdirs_list*, *num_antennas*, *antenna_spacing*)

Resolve imaging directions to direction per antenna.

This function will take in a list of directions and resolve that to a direction for each antenna. It will return a list of length num_antenna where each element is a direction off orthogonal for that antenna.

Parameters

- **beamdirs_list** – The list of beam directions for this pulse sequence.
- **num_antennas** – The number of antennas to calculate direcitonrs for.
- **antenna_spacing** – The spacing between the antennas.

Returns beamdirs
A list of beam directions for each antenna.

Returns amplitudes
A list of amplitudes for each antenna

sample_building.sample_building.**rx_azimuth_to_antenna_offset**(*beamdir*, *main_antenna_count*, *interferometer_antenna_count*, *main_antenna_spacing*, *interferometer_antenna_spacing*, *intf_offset*, *freq*)

Get all the necessary phase shifts for all antennas for all the beams for a pulse sequence.

Take all beam directions and resolve into a list of phase offsets for all antennas given the spacing, frequency, and number of antennas to resolve for (provided in config).

If the experiment does not use all channels in config, that will be accounted for in the send_dsp_metadata function, where the phase rotation will instead = 0.0 so all samples from that receive channel will be multiplied by zero and therefore not included (in beamforming).

Parameters

- **beamdir** – list of length 1 or more.
- **main_antenna_count** – the number of main antennas to calculate the phase offset for.
- **interferometer_antenna_count** – the number of interferometer antennas to calculate the phase offset for.
- **main_antenna_spacing** – the spacing between the main array antennas (m).
- **interferometer_antenna_spacing** – the spacing between the interferometer antennas (m).
- **intf_offset** – The interferometer offset from the main array (from centre to centre), in Cartesian coordinates. [x, y, z] where x is along line of antennas, y is along array normal and z is altitude difference, in m.
- **freq** – the frequency we are transmitting/receiving at.

Returns beams_antenna_phases
a list of length = beam directions, where each element is a list of length = number of antennas

(main array followed by interferometer array). The inner list contains the phase shift for the corresponding antenna for the corresponding beam.

sample_building.sample_building.**shift_samples**(*basic_samples*, *phshift*, *amplitude*)

Shift samples for a pulse by a given phase shift.

Take the samples and shift by given phase shift in rads and adjust amplitude as required for imaging.

Parameters

- **basic_samples** – samples for this pulse, numpy array
- **phshift** – phase for this antenna to offset by in rads, float
- **amplitude** – amplitude for this antenna (= 1 if not imaging), float

Returns samples

basic_samples that have been shaped for the antenna for the desired beam.

7.3.3 utils package

utils.experiment_options.experimentoptions module

To load the config options to be used by the experiment and radar_control blocks. Config data comes from the config.ini file, the hdw.dat file, and the restrict.dat file.

class utils.experiment_options.experimentoptions.**ExperimentOptions**

Bases: object

property altitude

property analog_atten_stages

property analog_rx_attenuator

property analog_rx_rise

property beam_sep

property boresight

property brian_to_driver_identity

property brian_to_dspbegin_identity

property brian_to_dspend_identity

property brian_to_radctrl_identity

property default_freq

property driver_to_brian_identity

property driver_to_dsp_identity

property driver_to_radctrl_identity

property dsp_to_driver_identity

property dsp_to_dw_identity

property dsp_to_exphan_identity

property dsp_to_radctrl_identity

property dspbegin_to_brian_identity

property dspend_to_brian_identity

property dw_to_dsp_identity

property dw_to_radctrl_identity

property exphan_to_dsp_identity

property exphan_to_radctrl_identity

property geo_lat

property geo_long

property interferometer_antenna_count

property interferometer_antenna_spacing

property intf_offset

property main_antenna_count

property main_antenna_spacing

property max_beams

property max_freq

property max_number_of_filter_taps_per_stage

property max_number_of_filtering_stages

property max_output_sample_rate

property max_range_gates

property max_rx_sample_rate

property max_tx_sample_rate

property max_usrp_dac_amplitude

property min_freq

property minimum_pulse_length

property minimum_pulse_separation

Minimum pulse separation is the minimum before the experiment treats it as a single pulse (transmitting zeroes or no receiving between the pulses)

property minimum_tau_spacing_length

property phase_sign

property pulse_ramp_time

property radctrl_to_brian_identity

property radctrl_to_driver_identity

property radctrl_to_dsp_identity

property radctrl_to_dw_identity

property radctrl_to_exphan_identity

property restricted_ranges

given in tuples of kHz

property router_address

property site_id

property tdiff

property tr_window_time

property usrp_master_clock_rate

property velocity_sign

Config Parameters

site_id	sas
gps_octoclock_addr	addr=192.168.10.131
devices	recv_frame_size=4000,addr0=192.168.10.100, addr1=192.168.10.101,addr2=192.168.10
main_antenna_count	16
interferometer_antenna_count	4
main_antenna_usrp_rx_channels	0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
interferometer_antenna_usrp_rx_channels	1,3,5,7
main_antenna_usrp_tx_channels	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
main_antenna_spacing	15.24
interferometer_antenna_spacing	15.24
min_freq	8.00E+06
max_freq	20.00E+06
minimum_pulse_length	100
minimum_mpinc_length	1
minimum_pulse_separation	125
tx_subdev	A:A
max_tx_sample_rate	5.00E+06
main_rx_subdev	A:A A:B
interferometer_rx_subdev	A:A A:B
max_rx_sample_rate	5.00E+06
pps	external
ref	external
overthewire	sc16
cpu	fc32
gpio_bank	RXA
atr_rx	0x0006

atr_tx	0x0018
atr_xx	0x0060
atr_0x	0x0180
max_usrp_dac_amplitude	0.99
pulse_ramp_time	1.00E-05
tr_window_time	6.00E-05
usrp_master_clock_rate	1.00E+08
max_output_sample_rate	1.00E+05
max_number_of_filter_taps_per_stage	2048
router_address	tcp://127.0.0.1:6969
radctrl_to_exphan_identity	RADCTRL_EXPHAN_IDEN
radctrl_to_dsp_identity	RADCTRL_DSP_IDEN
radctrl_to_driver_identity	RADCTRL_DRIVER_IDEN
radctrl_to_brian_identity	RADCTRL_BRIAN_IDEN
radctrl_to_dw_identity	RADCTRL_DW_IDEN
driver_to_radctrl_identity	DRIVER_RADCTRL_IDEN
driver_to_dsp_identity	DRIVER_DSP_IDEN
driver_to_brian_identity	DRIVER_BRIAN_IDEN
driver_to_mainaffinity_identity	DRIVER_MAINAFFINITY_IDEN
driver_to_txaffinity_identity	DRIVER_TXAFFINITY_IDEN
driver_to_rxaffinity_identity	DRIVER_RXAFFINITY_IDEN
mainaffinity_to_driver_identity	MAINAFFINITY_DRIVER_IDEN
txaffinity_to_driver_identity	TXAFFINITY_DRIVER_IDEN
rxaffinity_to_driver_identity	RXAFFINITY_DRIVER_IDEN
exphan_to_radctrl_identity	EXPHAN_RADCTRL_IDEN
exphan_to_dsp_identity	EXPHAN_DSP_IDEN
dsp_to_radctrl_identity	DSP_RADCTRL_IDEN
dsp_to_driver_identity	DSP_DRIVER_IDEN
dsp_to_exphan_identity	DSP_EXPHAN_IDEN
dsp_to_dw_identity	DSP_DW_IDEN
dspbegin_to_brian_identity	DSPBEGIN_BRIAN_IDEN
dspend_to_brian_identity	DSPEND_BRIAN_IDEN
dw_to_dsp_identity	DW_DSP_IDEN
dw_to_radctrl_identity	DW_RADCTRL_IDEN
brian_to_radctrl_identity	BRIAN_RADCTRL_IDEN
brian_to_driver_identity	BRIAN_DRIVER_IDEN
brian_to_dspbegin_identity	BRIAN_DSPBEGIN_IDEN
brian_to_dspend_identity	BRIAN_DSPEND_IDEN
ringbuffer_name	data_ringbuffer
ringbuffer_size_bytes	200.00E+06
data_directory	/data/borealis_data

CHAPTER

EIGHT

BOREALIS DATA FILES

8.1 Data Generation

The Borealis software module *data_write.py* is responsible for writing all data files. Different flags can be given to the module to write various types of files. See the documentation for *Borealis Processes*

Borealis writes files into HDF5 format. Examples of how to use HDF5 files can be found here for multiple languages: HDF5 Examples

The following data file types can be generated by Borealis in HDF5 format. The standard Borealis *release* mode run by the scheduler generates HDF5 files for rawacf, antennas_iq and bfiq.

8.1.1 Borealis filetypes

These are the Borealis filetypes produced by the radar software, from most processed to least processed.

- **rawacf**
  The correlated data from the main and interferometer arrays. Produced by Borealis in release mode.
- **bfiq**
  The beamformed iq data from both arrays. Produced by Borealis in release mode.
- **antennas_iq**
  The iq data from every antenna. Produced by Borealis in release mode.
- **rawrf**
  The unfiltered, full receive bandwidth data from every antenna. Only produced by Borealis in debug modes.

Post-processed dmap files can be created from the hdf5 rawacf or bfiq files using the pyDARN package.

For more information on the data files and the fields stored within them, check the data file information for the correct Borealis software version.

8.1.2 Borealis current version

The Borealis software version can affect the data fields in the file format so be sure to check if your data is of the most up to date version. The current Borealis software version is v0.5.

rawacf v0.5

This is the most up to date version of this file format produced by Borealis version 0.5, the current version.

For data files from previous Borealis software versions, see here.

The pydarn format class for this format is BorealisRawacf found in the borealis_formats.

The rawacf format is intended to hold beamformed, averaged, correlated data.

Both site files and array-restructured files exist for this file type. Both are described below.

rawacf array files

Array restructured files are produced after the radar has finished writing a file and contain record data in multi-dimensional arrays so as to avoid repeated values, shorten the read time, and improve human readability. Fields that are unique to the record are written as arrays where the first dimension is equal to the number of records recorded. Other fields that are unique to the slice or experiment (and are therefore repeated for all records) are written only once.

The group names in these files are the field names themselves, greatly reducing the number of group names in the file when compared to site files and making the file much more human readable.

The naming convention of the rawacf array-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].rawacf.hdf5

For example: 20191105.1400.02.sas.0.rawacf.hdf5

This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time. It has been array restructured because it does not have a .site designation at the end of the filename.

These files are zlib compressed which is native to hdf5 and no decompression is necessary before reading using your hdf5 library.

The file fields in the rawacf array files are:

FIELD NAME *type* [dimensions]	**description**
averaging_method *unicode*	A string describing the averaging method. Default is 'mean' but an experiment can set this to 'median' to get the median of all sequences in an integration period, and other methods to combine all sequences in an integration period could be added in the future.

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
beam_azms *float64* [num_records x max_num_beams]	A list of the beam azimuths for each beam in degrees off boresite. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
beam_nums *uint32* [num_records x max_num_beams]	A list of beam numbers used in this slice in this record. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
blanked_samples *uint32* [num_records x max_num_blanked_samples]	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence and can differ from record to record if a new slice is added.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.
correlation_descriptors *unicode* [4]	Denotes what each correlation dimension (in main_acfs, intf_acfs, xcfs) represents. = 'num_records', 'max_num_beams', 'num_ranges', 'num_lags'

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
data_normalization_factor *float32*	Scale of all the filters used, multiplied, for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
first_range *float32*	Distance to use for first range in km.
first_range_rtt *float32*	Round trip time of flight to first range in microseconds.
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32* [num_records]	Integration time in seconds.
intf_acfs *complex64* [num_records x max_num_beams x num_ranges x num_lags]	Interferometer array correlations. Note that records that do not have num_beams = max_num_beams will have padded zeros. The num_beams array should be used to determine the correct number of beams to read for the record.

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
intf_antenna_count *uint32*	Number of interferometer array antennas
lags *uint32* [number of lags, 2]	The lags created from two pulses in the pulses array. Values have to be from pulses array. The lag number is lag[1] - lag[0] for each lag pair.
main_acfs *complex64* [num_records x max_num_beams x num_ranges x num_lags]	Main array correlations. Note that records that do not have num_beams = max_num_beams will have padded zeros. The num_beams array should be used to determine the correct number of beams to read for the record.
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *float64* [num_records x max_num_sequences]	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
num_beams *uint32* [num_records]	The number of beams calculated for each record. Allows the user to correctly read the data up to the correct number and remove the padded zeros in the data array.

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
num_blanked_samples *uint32* [num_records]	The number of blanked samples for each record.
num_sequences *int64* [num_records]	Number of sampling periods (equivalent to number sequences transmitted) in the integration time for each record. Allows the user to correctly read the data up to the correct number and remove the padded zeros in the data array.
num_slices *int64* [num_records]	Number of slices used simultaneously in the record by the experiment. If more than 1, data should exist in another file for the same time period as that record for the other slice.
pulses *uint32* [number of pulses]	The pulse sequence in units of the tau_spacing.
range_sep *float32*	Range gate separation (conversion from time (1/rx_sample_rate) to equivalent distance between samples), in km.
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool* [num_records]	Designates if the record is the first in a scan (scan is defined by the experiment).

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').
slice_comment *unicode*	Additional text comment that describes the slice written in this file.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode* [num_records]	The interfacing of this slice to other slices for each record. String representation of the python dictionary of {slice : interface_type, ... }. Can differ between records if slices updated.
sqn_timestamps *float64* [num_records x max_num_sequences]	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come back from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
station *unicode*	Three-letter radar identifier.

continues on next page

Table 1 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.
xcfs *complex64* [num_records x max_num_beams x num_ranges x num_lags]	Cross correlations of interferometer to main array. Note that records that do not have num_beams = max_num_beams will have padded zeros. The num_beams array should be used to determine the correct number of beams to read for the record.

rawacf site files

Site files are produced by the Borealis code package and have the data in a record by record style format. In site files, the hdf5 group names (ie record names) are given as the timestamp in ms past epoch of the first sequence or sampling period recorded in the record.

The naming convention of the rawacf site-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].rawacf.hdf5.site

For example: 20191105.1400.02.sas.0.rawacf.hdf5.site This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time.

These files are often bzipped after they are produced.

The file fields under the record name in rawacf site files are:

Field name *type*	**description**
averaging_method *unicode*	A string describing the averaging method. Default is 'mean' but an experiment can set this to 'median' to get the median of all sequences in an integration period, and other methods to combine all sequences in an integration period could be added in the future.
beam_azms *[float64,]*	A list of the beam azimuths for each beam in degrees off boresite.
beam_nums *[uint32,]*	A list of beam numbers used in this slice in this record.
blanked_samples *[uint32,]*	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.
correlation_descriptors *[unicode,]*	Denotes what each correlation dimension (in main_acfs, intf_acfs, xcfs) represents. ('num_beams, 'num_ranges', 'num_lags')
correlation_dimensions *[uint32,]*	The dimensions in which to reshape the acf or xcf datasets. Dimensions correspond to correlation_descriptors.

continues on next page

Table 2 – continued from previous page

Field name *type*	**description**
data_normalization_factor *float32*	Scale of all the filters used, multiplied for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
first_range *float32*	Distance to use for first range in km.
first_range_rtt *float32*	Round trip time of flight to first range in microseconds.
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32*	Integration time in seconds.
intf_acfs *[complex64,]*	Interferometer array correlations.
intf_antenna_count *uint32*	Number of interferometer array antennas

continues on next page

Table 2 – continued from previous page

Field name *type*	**description**
lags *[[uint32,],]*	The lags created from two pulses in the pulses array. Dimensions are number of lags x 2. Values have to be from pulses array. The lag number is lag[1] - lag[0] for each lag pair.
main_acfs *[complex64,]*	Main array correlations.
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *[float64,]*	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO.
num_sequences *int64*	Number of sampling periods (equivalent to number sequences transmitted) in the integration time.
num_slices *int64*	Number of slices used simultaneously in this record by the experiment. If more than 1, data should exist in another file for this time period for the other slice.
pulses *[uint32,]*	The pulse sequence in units of the tau_spacing.
range_sep *float32*	Range gate separation (conversion from time (1/rx_sample_rate) to equivalent distance between samples), in km.
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.

continues on next page

Table 2 – continued from previous page

Field name *type*	**description**
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool*	Designates if the record is the first in a scan (scan is defined by the experiment).
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').
slice_comment *unicode*	Additional text comment that describes the slice written in this file.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode*	The interfacing of this slice to other slices. String representation of the python dictionary of {slice : interface_type, ... }
sqn_timestamps *[float64,]*	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch.
station *unicode*	Three-letter radar identifier.

continues on next page

Table 2 – continued from previous page

Field name *type*	**description**
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.
xcfs *[complex64,]*	Cross correlations of interferometer to main array.

Site/Array Restructuring

File restructuring to array files is done using an additional code package. Currently, this code is housed within pyDARN. It is expected that this code will be separated to its own IO code package in the near future.

The site to array file restructuring occurs in the borealis BaseFormat _site_to_array class method, and array to site restructuring is done in the same class _array_to_site method. Both can be found here.

rawacf to rawacf SDARN (DMap) Conversion

Conversion to SDARN IO (DMap rawacf) is available but can fail based on experiment complexity. The conversion also reduces the precision of the data due to conversion from complex floats to int of all samples. Similar precision is lost in timestamps.

HDF5 is a much more user-friendly format and we encourage the use of this data if possible. Please reach out if you have questions on how to use the Borealis rawacf files.

The mapping to rawacf dmap files is completed as follows:

rawacf_mapping

RAWACF SDARN FIELDS

This conversion is done in pydarn IO here in the __convert_rawacf_record method: Link to Source

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
radar.revision.major *char* Major version number	*borealis_git_hash* major version number or 255 if not a commit with a version tag
radar.revision.minor *char* Minor version number	*borealis_git_hash* minor version number or 255 if not a commit with a version tag
origin.code *char* Code indicating origin of data	= 100, this can be used as a flag that the origin code was Borealis
origin.time *string* ASCII representation of when the data was generated	*timestamp_of_write* conversion
origin.command *string* The command line or control program used to generate the data	Borealis vXXX + *borealis_git_hash* + *experiment_name*
cp *short* Control program identifier	*experiment_id*, truncated to short
stid *short* Station identifier	*station* conversion
time.yr *short* Year	*sqn_timestamps* [0] conversion

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
time.mo *short* Month	*sqn_timestamps* [0] conversion
time.dy *short* Day	*sqn_timestamps* [0] conversion
time.hr *short* Hour	*sqn_timestamps* [0] conversion
time.mt *short* Minute	*sqn_timestamps* [0] conversion
time.sc *short* Second	*sqn_timestamps* [0] conversion
time.us *short* Microsecond	*sqn_timestamps* [0] conversion
txpow *short* Transmitted power (kW)	= -1 (filler)
nave *short* Number of pulse sequences transmitted	*num_sequences*

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
atten *short* Attenuation level	= 0 (filler)
lagfr *short* Lag to first range (microseconds)	*first_range_rtt*
smsep *short* Sample separation (microseconds)	(*rx_sample_rate*)^ -1
ercod *short* Error code	= 0 (filler)
stat.agc *short* AGC status word	= 0 (filler)
stat.lopwr *short* LOPWR status word	= 0 (filler)
noise.search *float* Calculated noise from clear frequency search	*noise_at_freq* [0] conversion

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
noise.mean *float* Average noise across frequency band	= 0 (filler)
channel *short* Channel number for a stereo radar (zero for all others)	*slice_id*
bmnum *short* Beam number	*beam_nums* [i]
bmazm *float* Beam azimuth	*beam_azms* [i]
scan *short* Scan flag	*scan_start_marker* (0 or 1)
offset *short* Offset between channels for a stereo radar (zero for all others)	= 0 (filler)
rxrise *short* Receiver rise time (microseconds)	= 0.0

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
intt.sc *short* Whole number of seconds of integration time.	*int_time* conversion
intt.us *short* Fractional number of microseconds of integration time	*int_time* conversion
txpl *short* Transmit pulse length (microseconds)	*tx_pulse_len*
mpinc *short* Multi-pulse increment (microseconds)	*tau_spacing*
mppul *short* Number of pulses in sequence	len(*pulses*)
mplgs *short* Number of lags in sequence	*lags*.shape[0]
nrang *short* Number of ranges	*correlation_dimensions*[1]

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
frang *short* Distance to first range (kilometers)	*first_range*
rsep *short* Range separation (kilometers)	*range_sep*
xcf *short* XCF flag	If *xcfs* exist, then =1
tfreq *short* Transmitted frequency	*freq*
mxpwr *int* Maximum power (kHz)	= -1 (filler)
lvmax *int* Maximum noise level allowed	= 20000 (filler)
rawacf.revision.major *int* Major version number of the rawacf format	= 255
rawacf.revision.minor *int* Minor version number of the rawacf format	= 255

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
combf *string* Comment buffer Comment buffer	Original Borealis filename, 'converted from Borealis file beam number ' X, number of beams in this original record (len(beam_nums)), experiment_comment and slice_comment from the file
thr *float* Thresholding factor	= 0.0 (filler)
ptab[mppul] *short* Pulse table	pulses
ltab[2][mplgs] *short* Lag table	np.transpose(*lags*)
pwr0[nrang] *[float]* Lag zero power for main	Calculated from *main_acfs*
slist[0-nrang] *[short]* List of stored ranges, length dependent on SNR. Lists the range gate of each stored ACF	range(0,*correlation_dimensions*.size[1])
acfd[2][mplgs][0-nrang] *[short]* Calculated ACFs	*main_acfs* conversion, real and imag

continues on next page

Table 3 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
xcfd[2][mplgs][0-nrang] *[short]* Calculated XCFs	*xcfs* conversion, real and imag

If blanked_samples != ptab, or pulse_phase_offset contains non-zeroes, no conversion to dmap rawacf is possible.

bfiq v0.5

This is the most up to date version of this file format produced by Borealis version 0.5, the current version.

For data files from previous Borealis software versions, see here.

The pydarn format class for this format is BorealisBfiq found in the borealis_formats.

The bfiq format is intended to hold beamformed I and Q data for the main and interferometer arrays. The data is not averaged.

Both site files and array-restructured files exist for this file type. Both are described below.

bfiq array files

Array restructured files are produced after the radar has finished writing a file and contain record data in multi-dimensional arrays so as to avoid repeated values, shorten the read time, and improve human readability. Fields that are unique to the record are written as arrays where the first dimension is equal to the number of records recorded. Other fields that are unique to the slice or experiment (and are therefore repeated for all records) are written only once.

The group names in these files are the field names themselves, greatly reducing the number of group names in the file when compared to site files and making the file much more human readable.

The naming convention of the bfiq array-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].bfiq.hdf5

For example: 20191105.1400.02.sas.0.bfiq.hdf5

This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time. It has been array restructured because it does not have a .site designation at the end of the filename.

These files are zlib compressed which is native to hdf5 and no decompression is necessary before reading using your hdf5 library.

The file fields in the bfiq array files are:

FIELD NAME *type* [dimensions]	**description**
antenna_arrays_order *unicode* [num_antenna_arrays]	States what order the data is in and describes the data layout for the num_antenna_arrays data dimension
beam_azms *float64* [num_records x max_num_beams]	A list of the beam azimuths for each beam in degrees off boresite. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
beam_nums *uint32* [num_records x max_num_beams]	A list of beam numbers used in this slice in this record. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
blanked_samples *uint32* [num_records x max_num_blanked_samples]	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence and can differ from record to record if a new slice is added.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.

continues on next page

Table 4 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
data *complex64* [num_records x num_antenna_arrays x max_num_sequences x max_num_beams x num_samps]	A set of samples (complex float) at given sample rate. Note that records that do not have num_sequences = max_num_sequences or num_beams = max_num_beams will have padded zeros. The num_sequences and num_beams arrays should be used to determine the correct number of sequences and beams to read for the record.
data_descriptors *unicode* [5]	Denotes what each data dimension represents. = 'num_records', 'num_antenna_arrays', 'max_num_sequences', 'max_num_beams', 'num_samps'
data_normalization_factor *float32*	Scale of all the filters used, multiplied, for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
first_range *float32*	Distance to use for first range in km.
first_range_rtt *float32*	Round trip time of flight to first range in microseconds.

continues on next page

Table 4 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32* [num_records]	Integration time in seconds.
intf_antenna_count *uint32*	Number of interferometer array antennas
lags *uint32* [number of lags, 2]	The lags created from two pulses in the pulses array. Values have to be from pulses array. The lag number is lag[1] - lag[0] for each lag pair.
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *float64* [num_records x max_num_sequences]	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
num_beams *uint32* [num_records]	The number of beams calculated for each record. Allows the user to correctly read the data up to the correct number and remove the padded zeros in the data array.

continues on next page

Table 4 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
num_blanked_samples *uint32* [num_records]	The number of blanked samples for each record.
num_ranges *uint32*	Number of ranges to calculate correlations for.
num_samps *uint32*	Number of samples in the sampling period. Each sequence has its own sampling period. Will also be provided as the last data_dimension value.
num_sequences *int64* [num_records]	Number of sampling periods (equivalent to number sequences transmitted) in the integration time for each record. Allows the user to correctly read the data up to the correct number and remove the padded zeros in the data array.
num_slices *int64* [num_records]	Number of slices used simultaneously in the record by the experiment. If more than 1, data should exist in another file for the same time period as that record for the other slice.
pulse_phase_offset *float32* [number of pulses]	For pulse encoding phase, in degrees offset. Contains one phase offset per pulse in pulses.
pulses *uint32* [number of pulses]	The pulse sequence in units of the tau_spacing.

continues on next page

Table 4 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
range_sep *float32*	Range gate separation (conversion from time (1/rx_sample_rate) to equivalent distance between samples), in km.
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool* [num_records]	Designates if the record is the first in a scan (scan is defined by the experiment).
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').
slice_comment *unicode*	Additional text comment that describes the slice written in this file. The slice number of this file is provided in the filename.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode* [num_records]	The interfacing of this slice to other slices for each record. String representation of the python dictionary of {slice : interface_type, ... }. Can differ between records if slices updated.

continues on next page

Table 4 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
sqn_timestamps *float64* [num_records x max_num_sequences]	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come back from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
station *unicode*	Three-letter radar identifier.
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.

bfiq site files

Site files are produced by the Borealis code package and have the data in a record by record style format. In site files, the hdf5 group names (ie record names) are given as the timestamp in ms past epoch of the first sequence or sampling period recorded in the record.

The naming convention of the bfiq site-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].bfiq.hdf5.site

For example: 20191105.1400.02.sas.0.bfiq.hdf5.site This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time.

These files are often bzipped after they are produced.

The file fields under the record name in bfiq site files are:

Field name *type*	**description**
antenna_arrays_order *[unicode,]*	States what order the data is in and describes the data layout for the num_antenna_arrays data dimension
beam_azms *[float64,]*	A list of the beam azimuths for each beam in degrees off boresite.
beam_nums *[uint32,]*	A list of beam numbers used in this slice in this record.
blanked_samples *[uint32,]*	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.
data *[complex64,]*	A contiguous set of samples (complex float) at given sample rate. Needs to be reshaped by data_dimensions to be correctly read.
data_descriptors *[unicode,]*	Denotes what each data dimension represents. = 'num_antenna_arrays', 'num_sequences', 'num_beams', 'num_samps' for bfiq
data_dimensions *[uint32,]*	The dimensions in which to reshape the data. Dimensions correspond to data_descriptors.

continues on next page

Table 5 – continued from previous page

Field name *type*	**description**
data_normalization_factor *float32*	Scale of all the filters used, multiplied for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
first_range *float32*	Distance to use for first range in km.
first_range_rtt *float32*	Round trip time of flight to first range in microseconds.
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32*	Integration time in seconds.
intf_antenna_count *uint32*	Number of interferometer array antennas
lags *[[uint32,],]*	The lags created from two pulses in the pulses array. Dimensions are number of lags x 2. Values have to be from pulses array. The lag number is lag[1] - lag[0] for each lag pair.

continues on next page

Table 5 – continued from previous page

Field name *type*	**description**
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *[float64,]*	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO.
num_ranges *uint32*	Number of ranges to calculate correlations for.
num_samps *uint32*	Number of samples in the sampling period. Each sequence has its own sampling period. Will also be provided as the last data_dimension value.
num_sequences *int64*	Number of sampling periods (equivalent to number sequences transmitted) in the integration time.
num_slices *int64*	Number of slices used simultaneously in this record by the experiment. If more than 1, data should exist in another file for this time period for the other slice.
pulse_phase_offset *[float32,]*	For pulse encoding phase, in degrees offset. Contains one phase offset per pulse in pulses.
pulses *[uint32,]*	The pulse sequence in units of the tau_spacing.
range_sep *float32*	Range gate separation (conversion from time (1/rx_sample_rate) to equivalent distance between samples), in km.

continues on next page

Table 5 – continued from previous page

Field name *type*	**description**
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool*	Designates if the record is the first in a scan (scan is defined by the experiment).
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').
slice_comment *unicode*	Additional text comment that describes the slice written in this file.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode*	The interfacing of this slice to other slices. String representation of the python dictionary of {slice : interface_type, ... }
sqn_timestamps *[float64,]*	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch.
station *unicode*	Three-letter radar identifier.

continues on next page

Table 5 – continued from previous page

Field name *type*	**description**
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.

Site/Array Restructuring

File restructuring to array files is done using an additional code package. Currently, this code is housed within pyDARN. It is expected that this code will be separated to its own IO code package in the near future.

The site to array file restructuring occurs in the borealis BaseFormat _site_to_array class method, and array to site restructuring is done in the same class _array_to_site method. Both can be found here.

bfiq to iqdat SDARN (DMap) Conversion

Conversion to SDARN IO (DMap iqdat) is available but can fail based on experiment complexity. The conversion also reduces the precision of the data due to conversion from complex floats to int of all samples. Similar precision is lost in timestamps.

HDF5 is a much more user-friendly format and we encourage the use of this data if possible. Please reach out if you have questions on how to use the Borealis bfiq files.

The mapping from bfiq to iqdat dmap files is completed as follows:

iqdat_mapping

IQDAT SDARN FIELDS

This conversion is done in pydarn IO here in the __convert_bfiq_record method: Link to Source

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
radar.revision.major *char* Major version number	*borealis_git_hash* major version number or 255 if not a commit with a version tag

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
radar.revision.minor *char* Minor version number	*borealis_git_hash* minor version number or 255 if not a commit with a version tag
origin.code *char* Code indicating origin of data	= 100, this can be used as a flag that the origin code was Borealis
origin.time *string* ASCII representation of when the data was generated	*timestamp_of_write* conversion
origin.command *string* The command line or control program used to generate the data	Borealis vXXX + *borealis_git_hash* + *experiment_name*
cp *short* Control program identifier	*experiment_id*, truncated to short
stid *short* Station identifier	*station* conversion
time.yr *short* Year	*sqn_timestamps* [0] conversion
time.mo *short* Month	*sqn_timestamps* [0] conversion

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
time.dy *short* Day	*sqn_timestamps* [0] conversion
time.hr *short* Hour	*sqn_timestamps* [0] conversion
time.mt *short* Minute	*sqn_timestamps* [0] conversion
time.sc *short* Second	*sqn_timestamps* [0] conversion
time.us *short* Microsecond	*sqn_timestamps* [0] conversion
txpow *short* Transmitted power (kW)	= -1 (filler)
nave *short* Number of pulse sequences transmitted	*num_sequences*
atten *short* Attenuation level	= 0 (filler)

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
lagfr *short* Lag to first range (microseconds)	*first_range_rtt*
smsep *short* Sample separation (microseconds)	(*rx_sample_rate*)^ -1
ercod *short* Error code	= 0 (filler)
stat.agc *short* AGC status word	= 0 (filler)
stat.lopwr *short* LOPWR status word	= 0 (filler)
noise.search *float* Calculated noise from clear frequency search	*noise_at_freq* [0] conversion
noise.mean *float* Average noise across frequency band	= 0 (filler)

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
channel *short* Channel number for a stereo radar (zero for all others)	*slice_id*
bmnum *short* Beam number	*beam_nums* [i]
bmazm *float* Beam azimuth	*beam_azms* [i]
scan *short* Scan flag	*scan_start_marker* (0 or 1)
offset *short* Offset between channels for a stereo radar (zero for all others)	= 0 (filler)
rxrise *short* Receiver rise time (microseconds)	= 0.0
intt.sc *short* Whole number of seconds of integration time.	*int_time* conversion

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
intt.us *short* Fractional number of microseconds of integration time	*int_time* conversion
txpl *short* Transmit pulse length (microseconds)	*tx_pulse_len*
mpinc *short* Multi-pulse increment (microseconds)	*tau_spacing*
mppul *short* Number of pulses in sequence	len(*pulses*)
mplgs *short* Number of lags in sequence	*lags*.shape[0]
nrang *short* Number of ranges	*num_ranges*
frang *short* Distance to first range (kilometers)	*first_range*

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
rsep *short* Range separation (kilometers)	*range_sep*
xcf *short* XCF flag	If *xcfs* exist, then =1
tfreq *short* Transmitted frequency	*freq*
mxpwr *int* Maximum power (kHz)	= -1 (filler)
lvmax *int* Maximum noise level allowed	= 20000 (filler)
iqdata.revision.major *int* Major version number of the iqdata library	= 1 (meaning Borealis conversion)
iqdata.revision.minor *int* Minor version number of the iqdata library	= 0 (Borealis conversion)

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
combf *string* Comment buffer	Original Borealis filename, 'converted from Borealis file ' , number of beams in this original record (len(beam_nums)), experiment_comment and slice_comment from the file
seqnum *int* Number of pulse sequences transmitted	*num_sequences*
chnnum *int* Number of channels sampled (both I and Q quadrature samples)	len(*antenna_arrays_order*)
smpnum *int* Number of samples taken per sequence	*num_samps*
skpnum *int* Number of samples to skip before the first valid sample	math.ceil(first_range/range_sep). In theory this should =0 due to Borealis functionality(no rise time). However make_raw in RST requires this to be indicative of the first range so we provide this.
ptab[mppul] *short* Pulse table	pulses

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
ltab[2][mplgs] *short* Lag table	np.transpose(*lags*)
tsc[seqnum] *int* Seconds component of time past epoch of pulse sequence	*sqn_timestamps* conversion
tus[seqnum] *int* Microsecond component of time past epoch of pulse sequence	*sqn_timestamps* conversion
tatten[seqnum] *short* Attenuator setting for each pulse sequence	= [0,0. . .] (fillers)
tnoise[seqnum] *float* Noise value for each pulse sequence	*noise_at_freq* conversion
toff[seqnum] *int* Offset into the sample buffer for each pulse sequence	Offset = 2 * num_samps * len(antenna_arrays_order), toff = [i * offset for i in range(v[‘num_sequences’])]
tsze[seqnum] *int* Number of words stored for this pulse sequence	= [offset, offset, offset. . . .]

continues on next page

Table 6 – continued from previous page

SDARN DMAP FIELD NAME *type* SDARN description	**Borealis Conversion**
data[totnum] *int* Array of raw I and Q samples, arranged: [[[smpnum(i), smpnum(q)] * chnnum] * seqnum], so totnum = 2*seqnum*chnnum*smpnum	Data conversion for correct dimensions and scaled to max int (-32768 to 32767)

If *blanked_samples* != *ptab*, or *pulse_phase_offset* contains non-zeroes, no conversion to iqdat is possible.

antennas_iq v0.5

This is the most up to date version of this file format produced by Borealis version 0.5, the current version.

For data files from previous Borealis software versions, see here.

The pydarn format class for this format is BorealisAntennasIq found in the borealis_formats.

The antennas_iq format is intended to hold individual antennas I and Q data. The data is filtered, but is not averaged.

Both site files and array-restructured files exist for this file type. Both are described below.

antennas_iq array files

Array restructured files are produced after the radar has finished writing a file and contain record data in multi-dimensional arrays so as to avoid repeated values, shorten the read time, and improve human readability. Fields that are unique to the record are written as arrays where the first dimension is equal to the number of records recorded. Other fields that are unique to the slice or experiment (and are therefore repeated for all records) are written only once.

The group names in these files are the field names themselves, greatly reducing the number of group names in the file when compared to site files and making the file much more human readable.

The naming convention of the antennas_iq array-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].antennas_iq.hdf5

For example: 20191105.1400.02.sas.0.antennas_iq.hdf5

This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time. It has been array restructured because it does not have a .site designation at the end of the filename.

These files are zlib compressed which is native to hdf5 and no decompression is necessary before reading using your hdf5 library.

The file fields in the antennas_iq array files are:

FIELD NAME *type* [dimensions]	**description**
antenna_arrays_order *unicode* [num_antennas]	States what order the data is in and describes the data layout for the num_antennas data dimension Antennas are recorded main array ascending and then interferometer array ascending
beam_azms *float64* [num_records x max_num_beams]	A list of the beam azimuths for each beam in degrees off boresite. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
beam_nums *uint32* [num_records x max_num_beams]	A list of beam numbers used in this slice in this record. Note that this is padded with zeroes for any record which has num_beams less than the max_num_beams. The num_beams field should be used to read the correct number of beams for each record.
blanked_samples *uint32* [num_records x max_num_blanked_samples]	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence and can differ from record to record if a new slice is added.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.

continues on next page

Table 7 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
data *complex64* [num_records x num_antennas x max_num_sequences x num_samps]	A set of samples (complex float) at given sample rate. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
data_descriptors *unicode* [4]	Denotes what each data dimension represents. = 'num_records', 'num_antennas', 'max_num_sequences', 'num_samps'
data_normalization_factor *float32*	Scale of all the filters used, multiplied, for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32* [num_records]	Integration time in seconds.

continues on next page

Table 7 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
intf_antenna_count *uint32*	Number of interferometer array antennas
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *float64* [num_records x max_num_sequences]	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
num_beams *uint32* [num_records]	The number of beams to calculate for each record.
num_blanked_samples *uint32* [num_records]	The number of blanked samples for each record.
num_samps *uint32*	Number of samples in the sampling period. Each sequence has its own sampling period. Will also be provided as the last data_dimension value.

continues on next page

Table 7 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
num_sequences *int64* [num_records]	Number of sampling periods (equivalent to number sequences transmitted) in the integration time for each record. Allows the user to correctly read the data up to the correct number and remove the padded zeros in the data array.
num_slices *int64* [num_records]	Number of slices used simultaneously in the record by the experiment. If more than 1, data should exist in another file for the same time period as that record for the other slice.
pulse_phase_offset *float32* [number of pulses]	For pulse encoding phase, in degrees offset. Contains one phase offset per pulse in pulses.
pulses *uint32* [number of pulses]	The pulse sequence in units of the tau_spacing.
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool* [num_records]	Designates if the record is the first in a scan (scan is defined by the experiment).
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').

continues on next page

Table 7 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
slice_comment *unicode*	Additional text comment that describes the slice written in this file. The slice number of this file is provided in the filename.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode* [num_records]	The interfacing of this slice to other slices for each record. String representation of the python dictionary of {slice : interface_type, … }. Can differ between records if slices updated.
sqn_timestamps *float64* [num_records x max_num_sequences]	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come back from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch. Note that records that do not have num_sequences = max_num_sequences will have padded zeros. The num_sequences array should be used to determine the correct number of sequences to read for the record.
station *unicode*	Three-letter radar identifier.
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.

continues on next page

Table 7 – continued from previous page

FIELD NAME *type* [dimensions]	**description**
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.

antennas_iq site files

Site files are produced by the Borealis code package and have the data in a record by record style format. In site files, the hdf5 group names (ie record names) are given as the timestamp in ms past epoch of the first sequence or sampling period recorded in the record.

The naming convention of the antennas_iq site-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].[slice_id].antennas_iq.hdf5.site

For example: 20191105.1400.02.sas.0.antennas_iq.hdf5.site This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data for slice 0 of the experiment that ran at that time.

These files are often bzipped after they are produced.

The file fields under the record name in antennas_iq site files are:

Field name *type*	**description**
antenna_arrays_order *[unicode,]*	States what order the data is in and describes the data layout for the num_antennas data dimension. Antennas are recorded main array ascending and then interferometer array ascending.
beam_azms *[float64,]*	A list of the beam azimuths for each beam in degrees off boresite.
beam_nums *[uint32,]*	A list of beam numbers used in this slice in this record.

continues on next page

Table 8 – continued from previous page

Field name *type*	**description**
blanked_samples *[uint32,]*	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.
data *[complex64,]*	A contiguous set of samples (complex float) at given sample rate. Needs to be reshaped by data_dimensions to be correctly read.
data_descriptors *[unicode,]*	Denotes what each data dimension represents. = 'num_antennas', 'num_sequences', 'num_samps' for antennas_iq
data_dimensions *[uint32,]*	The dimensions in which to reshape the data. Dimensions correspond to data_descriptors.
data_normalization_factor *float32*	Scale of all the filters used, multiplied for a total scale to normalize the data by.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.

continues on next page

Table 8 – continued from previous page

Field name *type*	**description**
experiment_name *unicode*	Name of the experiment file.
freq *uint32*	The frequency used for this experiment, in kHz. This is the frequency the data has been filtered to.
int_time *float32*	Integration time in seconds.
intf_antenna_count *uint32*	Number of interferometer array antennas
main_antenna_count *uint32*	Number of main array antennas
noise_at_freq *[float64,]*	Noise at the receive frequency, with dimension = number of sequences. 20191114: not currently implemented and filled with zeros. Still a TODO.
num_samps *uint32*	Number of samples in the sampling period. Each sequence has its own sampling period. Will also be provided as the last data_dimension value.
num_sequences *int64*	Number of sampling periods (equivalent to number sequences transmitted) in the integration time.
num_slices *int64*	Number of slices used simultaneously in this record by the experiment. If more than 1, data should exist in another file for this time period for the other slice.

continues on next page

Table 8 – continued from previous page

Field name *type*	**description**
pulse_phase_offset *[float32,]*	For pulse encoding phase, in degrees offset. Contains one phase offset per pulse in pulses.
pulses *[uint32,]*	The pulse sequence in units of the tau_spacing.
rx_sample_rate *float64*	Sampling rate of the samples in this file's data in Hz.
samples_data_type *unicode*	C data type of the samples, provided for user friendliness. = 'complex float'
scan_start_marker *bool*	Designates if the record is the first in a scan (scan is defined by the experiment).
scheduling_mode *unicode*	The mode being run during this time period (ex. 'common', 'special', 'discretionary').
slice_comment *unicode*	Additional text comment that describes the slice written in this file.
slice_id *uint32*	The slice id of this file.
slice_interfacing *unicode*	The interfacing of this slice to other slices. String representation of the python dictionary of {slice : interface_type, ... }

continues on next page

Table 8 – continued from previous page

Field name *type*	**description**
sqn_timestamps *[float64,]*	A list of GPS timestamps corresponding to the beginning of transmission for each sampling period in the integration time. These timestamps come from the USRP driver and the USRPs are GPS disciplined and synchronized using the Octoclock. Provided in milliseconds since epoch.
station *unicode*	Three-letter radar identifier.
tau_spacing *uint32*	The minimum spacing between pulses in microseconds. Spacing between pulses is always a multiple of this.
tx_pulse_len *uint32*	Length of the transmit pulse in microseconds.

Site/Array Restructuring

File restructuring to array files is done using an additional code package. Currently, this code is housed within pyDARN. It is expected that this code will be separated to its own IO code package in the near future.

The site to array file restructuring occurs in the borealis BaseFormat _site_to_array class method, and array to site restructuring is done in the same class _array_to_site method. Both can be found here.

rawrf v0.5

This is the most up to date version of this file format produced by Borealis version 0.5, the current version.

For data files from previous Borealis software versions, see here.

The pydarn format class for this format is BorealisRawrf found in the borealis_formats.

The rawrf format is intended to hold high bandwidth, non-filtered raw data from every antenna.

This format is only produced in a site-style, record by record format and is only available to be produced on request. Please note that this format can cause radar operating delays and may reduce number of averages in an integration, for example.

rawrf site files

Site files are produced by the Borealis code package and have the data in a record by record style format. In site files, the hdf5 group names (ie record names) are given as the timestamp in ms past epoch of the first sequence or sampling period recorded in the record.

The naming convention of the rawrf site-structured files are:

[YYYYmmDD].[HHMM].[SS].[station_id].rawrf.hdf5.site

For example: 20191105.1400.02.sas.rawrf.hdf5.site

This is the file that began writing at 14:00:02 UT on November 5 2019 at the Saskatoon site, and it provides data the experiment that ran at that time. Since rawrf is not filtered, this data does not need a slice identifier because it contains all the samples being taken at that time. Some familiarity with the experiment may be necessary to understand the data, or some access to the other file types produced concurrently. This is primarily a debug format for engineering purposes and should only be produced for special cases.

These files are often bzipped after they are produced.

The file fields under the record name in rawrf site files are:

Field name *type*	**description**
blanked_samples *uint32* [number of blanked samples]	Samples that should be blanked because they occurred during transmission times, given by sample number (index into decimated data). Can differ from the pulses array due to multiple slices in a single sequence.
borealis_git_hash *unicode*	Identifies the version of Borealis that made this data. Contains git commit hash characters. Typically begins with the latest git tag of the software.
data *[complex64,]*	A contiguous set of samples (complex float) at given sample rate. Needs to be reshaped by data_dimensions to be correctly read.
data_descriptors *[unicode,]*	Denotes what each data dimension represents. = 'num_sequences', 'num_antennas', 'num_samps' for rawrf
data_dimensions *[uint32,]*	The dimensions in which to reshape the data. Dimensions correspond to data_descriptors.
experiment_comment *unicode*	Comment provided in experiment about the experiment as a whole.
experiment_id *int64*	Number used to identify the experiment.
experiment_name *unicode*	Name of the experiment file.
int_time *float32*	Integration time in seconds.

Site/Array Restructuring

File restructuring to array files is not done for this format.

8.1.3 Previous versions

- v0.2, v0.3, and v0.4 follow the v0.4 format.

8.2 Reading Data

To read the files in python, we recommend using PyTables or deepdish packages. If you are looking to generate SuperDARN standard plots, we recommend using the the pyDARN package, which can read Borealis files specifically. After converting to dmap, standard SuperDARN plots including RTI plots and fan plot can be produced.

CHAPTER

NINE

DATA STORAGE AND DELETION

Borealis file sizes can add up quickly to fill all available hard drive space, especially if antennas_iq and/or bfiq data types are being generated. However, it is convenient and recommended to keep a backlog of lower level data products such as antennas_iq for a period of time. These files are useful for debugging hardware issues and reproducing RAWACF files.

A utility script is scheduled via cron to check the filesystem that Borealis files are written to. If the filesystem usage is too high, it searches for and deletes the oldest files in a loop until the filesystem usage goes below the threshold. See the SuperDARN Canada data flow repository for more information.

In order to prevent system failure due to hard drives filling up, a method for deleting the oldest data files is employed for SuperDARN Canada radars. This is referred to as *rotating* the files.

CHAPTER

TEN

BOREALIS MONITORING

The monitoring system implemented for Borealis is a custom configured installation of Nagios Core, working with NRPE. Nagios monitoring behaves according to objects defined in configuration files, all of which have copies in SuperDARN Canada's Nagios repository.

10.1 Nagios

Nagios core runs as a service under apache2. It is easy to install, but a little tricky to configure for specific purposes. The program executes external plugins that obtain information from the system, and then displays the output on locally hosted webpage. Locally, where and which plugins are executed is determined by host and service objects specified in configuration files. This is also done with monitoring on remote machines, with one exception.

The remote server runs plugins using an a service called NRPE (Nagios Remote Plugin Executor). This process runs on port 566 by default, and sends plugin output over the network to the Nagios service running on the central host. The central host accepts this output through a plugin called check_nrpe, usage specified in the commands.cfg config file. This remote host output is then displayed normally alongside the local services.

In our configuration, remote hosts send information on services continuously, allowing connections from hosts specified in their nrpe.cfg file. To operate properly, both the hostname of the remote host, and that of the central Nagios host, must be included on this line.

The last key difference between NRPE and Nagios Core is that commands to be executed on the remote host are defined in that host's nrpe.cfg file. Whereas commands executed by Nagios Core are defined in the commands.cfg by default.

10.2 Installation

Detailed instructions for installing Nagios Core on several operating systems can be found on Nagios' website.

After installing, simply replace the configuration files with those found in this repository.

Installation of NRPE is similarly simple. Detailed instructions can be found in the NRPE.pdf file located in the monitoring folder along with our config files.

CHAPTER

ELEVEN

LAB TESTING

Good lab tests to include before deployement include:

1. Loopback tests at boresight - allow you to see differences in channel power after digitizing - can also verify that boxes are synchronized because boresite with

   equal cable lengths to receive should give you the same output on all antennas (phases should be the same, check with rawrf and antennas_iq)

   - scripts are available under testing/borealis_tests/testing_utils/plot_borealis_hdf5_data/

2. Logic analyzer tests with a transmitter
3. Long-term reliability tests of software
4. Scope output tests - verify pulse shape of TX out - verify GPIO signals (T/R) - verify pulse distances
5. Test for GPS lock on GPS Octoclock
6.

CHAPTER

TWELVE

TOOLS

12.1 NEC

A python script called *nec_sd_generator.py* in the *tools/NEC* Borealis directory contains functionality to produce the correct geometry and other inputs for a NEC engine program like 4nec2 to simulate/model SuperDARN antenna arrays.

This script can be used to generate some common orientations of the SuperDARN antenna arrays for use with a NEC engine. Some programs that can read NEC inputs are 4nec2 or eznec. This script has been tested with the free, latest version of 4nec2, 5.8.17, updated January 2020 and available here: https://www.qsl.net/4nec2/

In order to use this with 4nec2, simply open 4nec2 and go to *File->Open 4nec2 in/out file* and select the file (it must end in '.nec'). Then hit *F7* or go to the *Calculate->NEC Output-data* option. Interpretations of the results are beyond the scope of this help message. Note that 4nec2 requires DOS line-endings, which is why this script outputs DOS line endings.

By default, if you run this script with no options, it will create TTFD main and interferometer arrays with 21-wire reflectors, oriented like the Rankin Inlet radar with the main array in front of the interferometer array. This can be changed with the *–int-x-spacing*, *–int-y-spacing* and *–int-z-spacing* options which take in a floating point number of meters to offset the interferometer from the main array in 3d space. By default, it is 100m behind (y == -100) and centered in both x and z dimensions. Turn off the reflector via the *–without-fence* flag. In summary, the boresite is in the +y direction, the arrays are typically along the x axis, and the +z axis is distance from the ground.

The number of antennas in both the main and interferometer arrays are controlled via the *–antennas* and *–int_antennas* options. Defaults are 16 and 4, like the Rankin Inlet array. Set the *–int_antennas* value to 0 to remove the interferometer array.

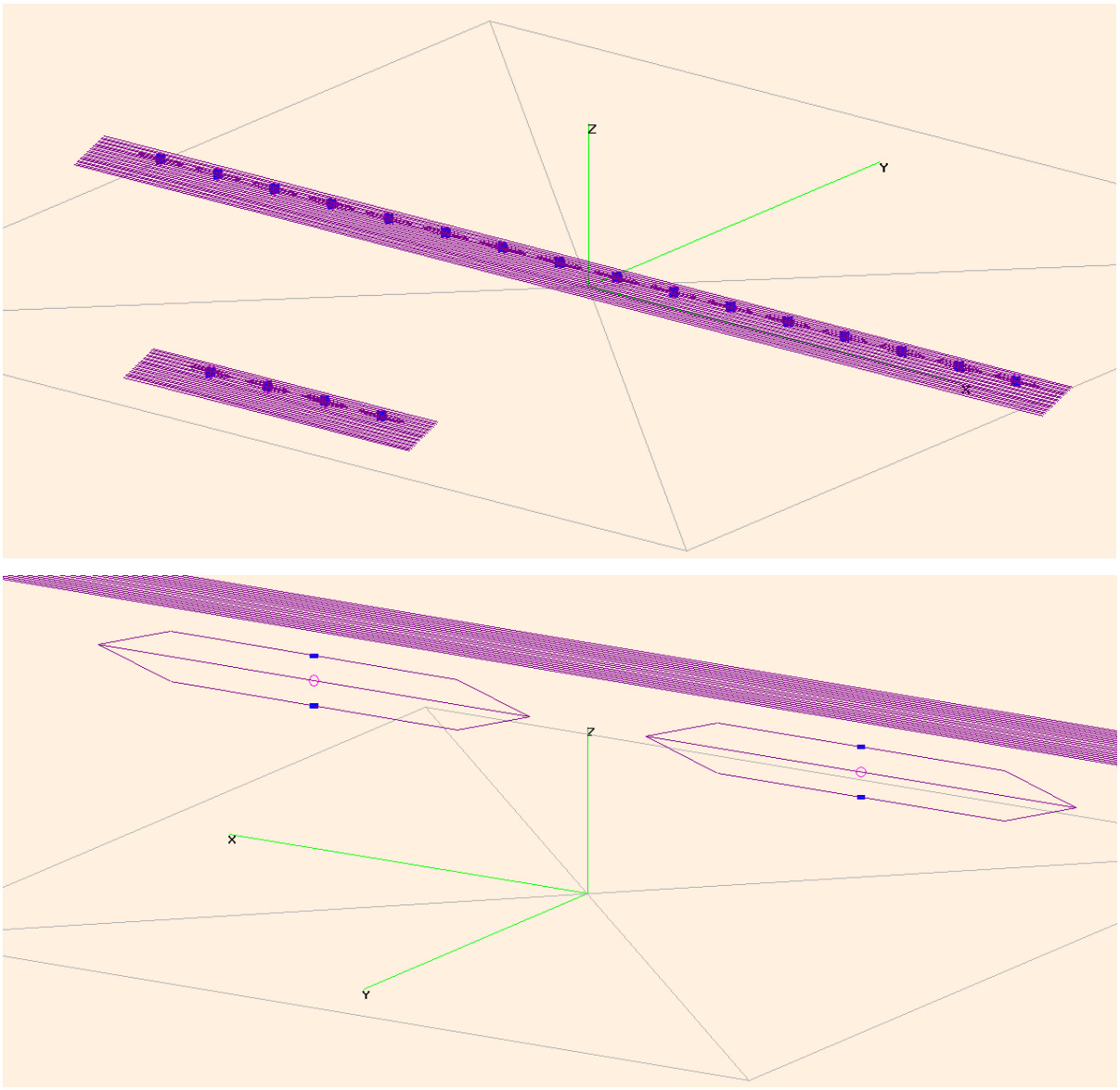
The antenna spacing can be modified from the 15.24m default via the *–antenna_spacing* option.

The beam and frequency used can be changed from the defaults of boresite and 10.5MHz via the *–beam* and *–frequency* options, which take floating point values.

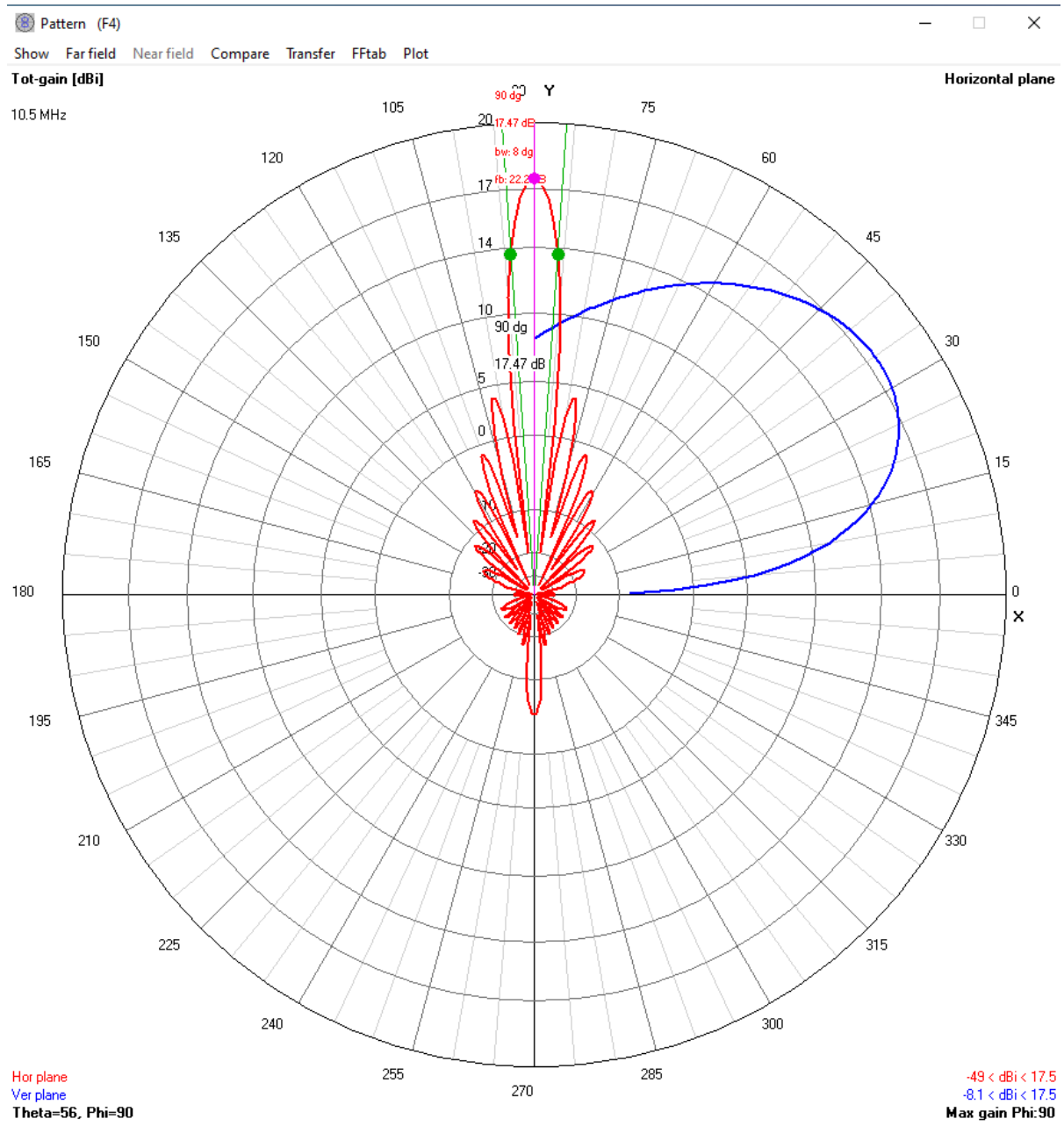
Finally, you can provide a custom name for the file generated by this script using the *–output_file* option. Just be aware that 4nec2 input file naming requirements are strict, no periods, and it must end in '.nec'.

NOTE There are some options that are not supported yet, like log periodic arrays, yagi arrays, as well as different power and phase inputs for the arrays. As well, baluns and feedlines are not implemented, so the signal sources are currently modeled directly where the balun would be on the antennas.

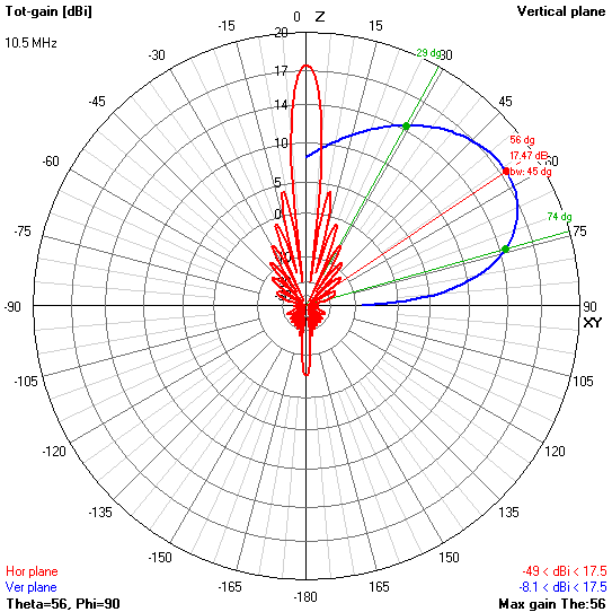
Here are two images that show what the default geometry looks like when importing the default output file into 4nec2. In the first image you can see a wide bird's eye view of the main and interferometer arrays. In the second image you can see a close-up of the main array center two antennas, with the 21 reflector wires in the background. The blue rectangles are the loads and the pink circle on the antenna's is the current source modeled in NEC.

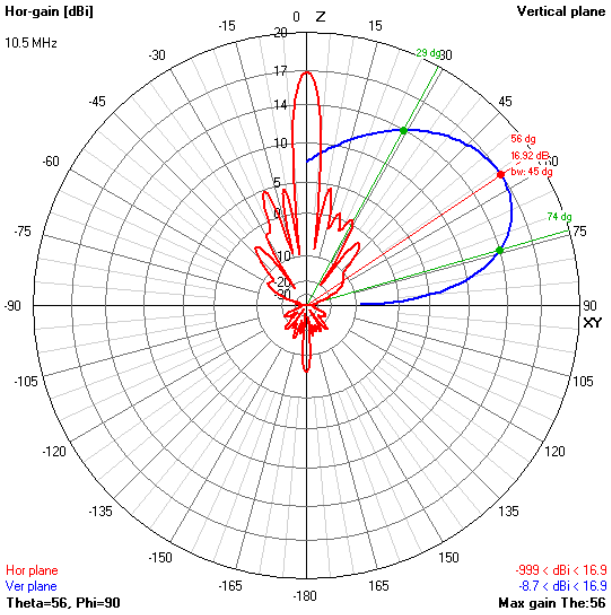


If you calculate this array geometry with the defaults, you'll see this window from 4nec2: This shows the horizontal gain in red, and the vertical gain in blue. You'll notice that the most power is in the main lobe at 0 degrees off azimuth (boresite). This is effectively a beam 7.5 in the standard SuperDARN configuration and shows that the radar array has a F/B ratio of 22dB, a beamwidth of 8 degrees and a gain of 17.47dB. Note this is at a frequency of 10.5MHz.

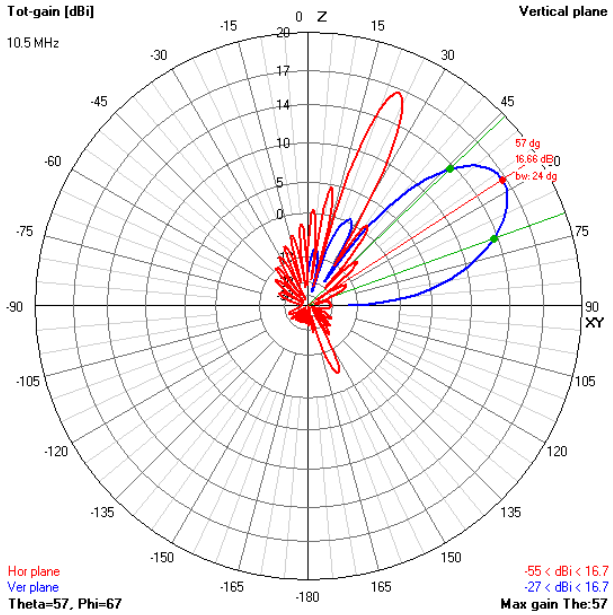


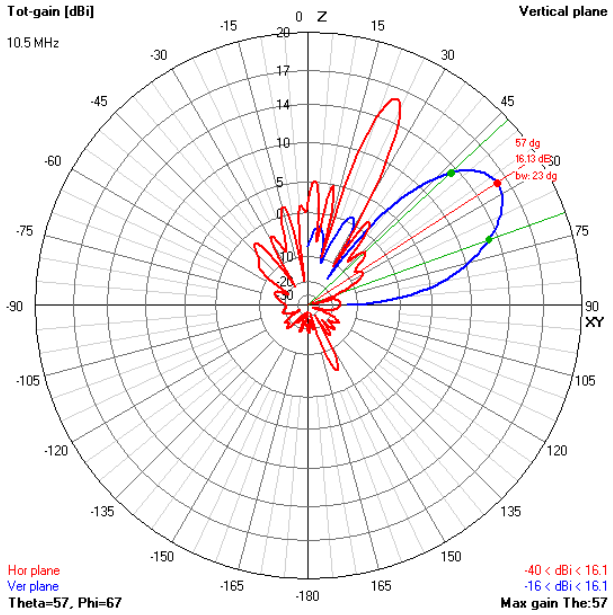
So, why is this considered a tool? What happens when a transmitter goes down? What happens when two transmitters go down? What about if the power output from one transmitter is half of what it should be? How about phase errors? All of these questions are possible to answer with tools like this one. Here's a real example from Rankin Inlet, where transmitters #6 and #12 (indexed from 0) are both down:





The above two images are generated for the radar at Rankin Inlet, the first image shows the standard pattern if everything is working properly at boresite. The second image shows the pattern resulting from transmitters #6 and #12 not contributing to the system. The effects are immediately visible in the higher power sidelobes. The main lobe gain is reduced from 17.47dB to 16.92dB. The main lobe remains the same shape and width in both azimuth and elevation angles.





The above two images are generated for the radar at Rankin Inlet, the first image shows the standard pattern if everything is working properly at beam 1. The second image shows the pattern resulting from transmitters #6 and #12 not contributing to the system. The effects are immediately visible in the higher power sidelobes. The main lobe gain is reduced from 16.66dB to 16.13dB. The main lobe remains the same shape but is slightly smaller (~1 degree) in elevation angle.

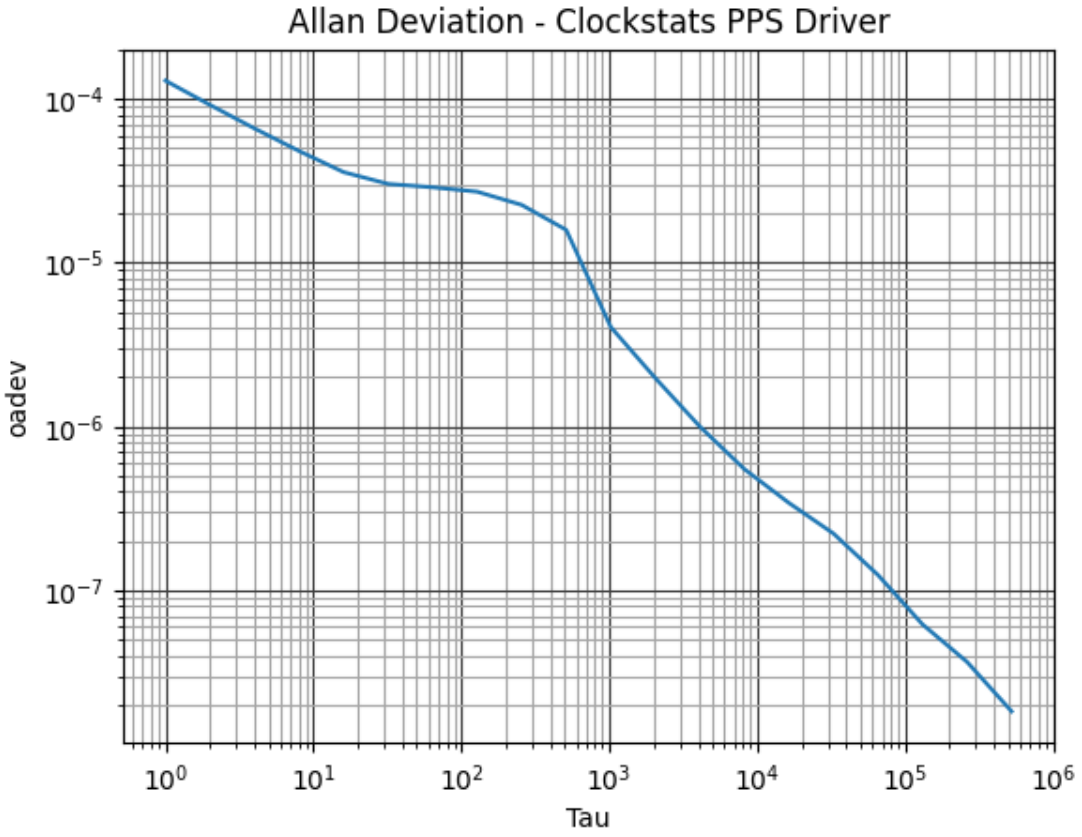
12.2 NTP

A python script called *plot_ntp_stats.py* located in the *tools/NTP* borealis directory contains functionality that can be used to plot some common statistics that the *ntpd* program can produce.

It requires that you've set up *ntpd* to log statistics. Currently supported plots are basic, but still useful. This script also requires the ntp configuration file to be able to accurately calculate the Allan deviation for PPS drivers.

The Allan deviation can be plotted if you have a *clockstats* file. The subject of Allan deviation is beyond the scope of this documentation, but it can give you an indication of your short, mid and long-term stability of your oscillator. In short, if you see a negative relationship between the y axis and the x axis that means that over the long term your oscillator is more stable than it is over the short term. Phase noise and Allan deviation are closely related.

Here is an example of an Allan deviation plot:

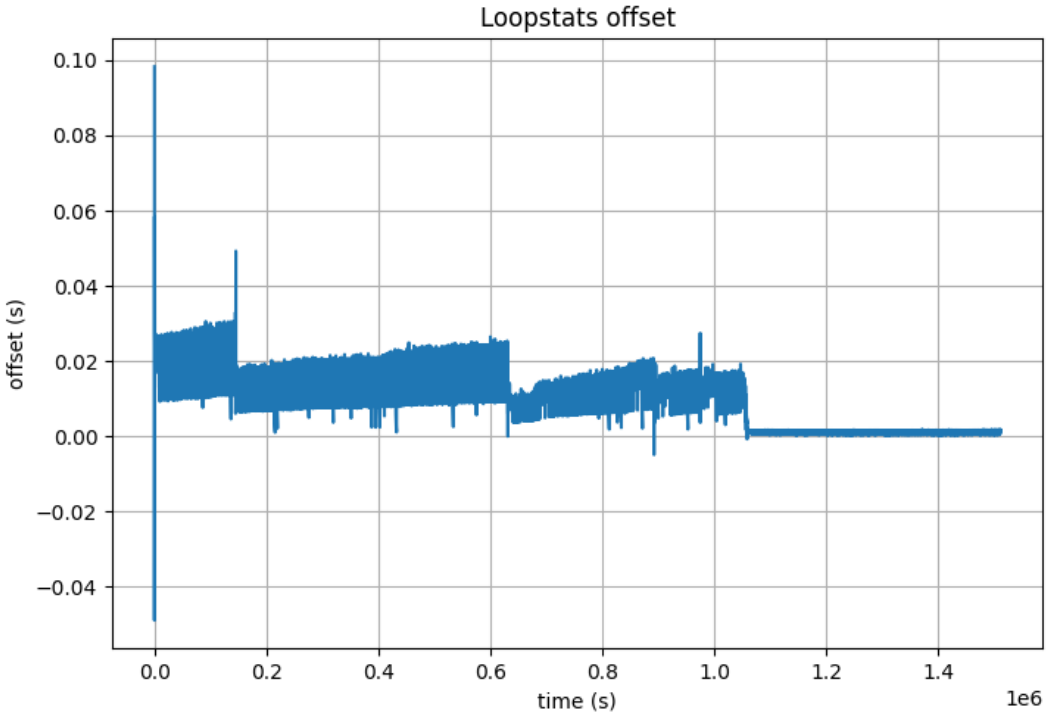


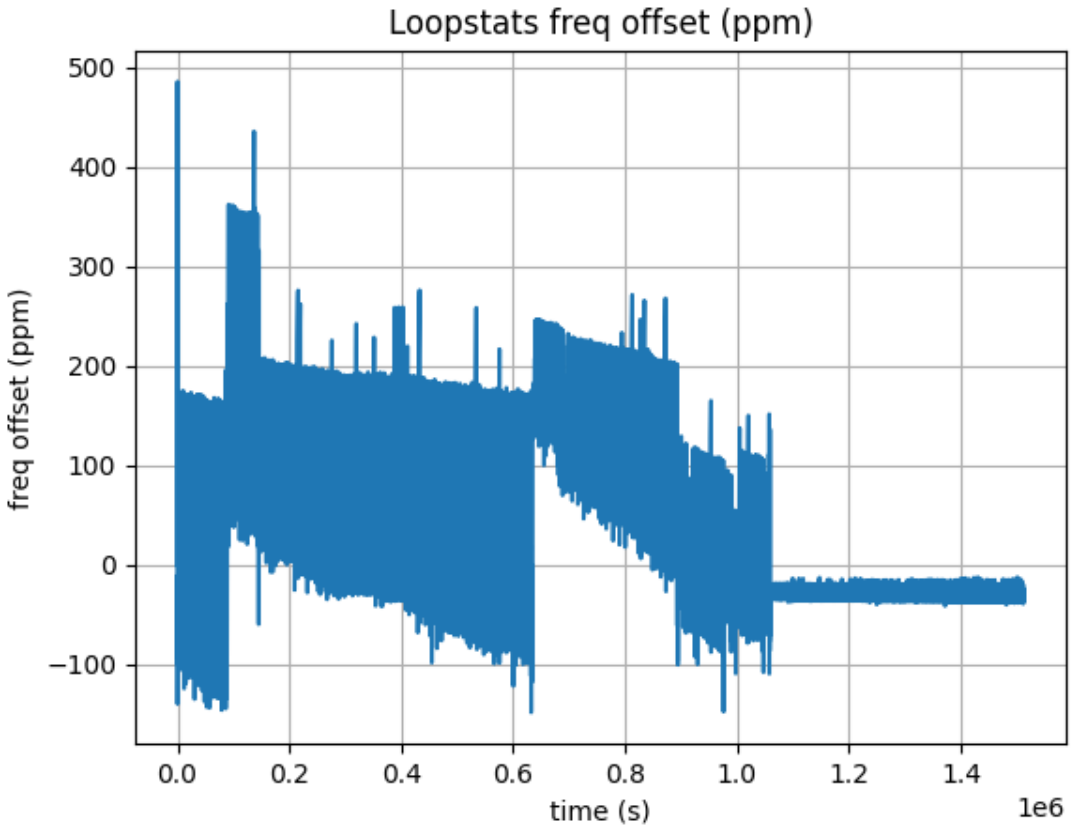
Looking at the above image, it's clear that the clock stats indicate the clock is more stable the longer you view it. This is generally true for GPS disciplined clocks. If you have a piezo crystal oscillator and generated an Allan deviation plot for it, you might see the opposite relationship. Combining the two types of clocks into a GPS disciplined oscillator will get you the best of both short and long term stability.

If you have a *loopstats* input file then you can plot two quantities:

- The *ntpd* estimated time offset from true time in seconds vs time smaller values are better.
- The *ntpd* estimated frequency offset in PPM from a 'true oscillator' (ideal UTC clock) vs time, smaller values are better.

Here are example plots of the *loopstats* offset and frequency offset:

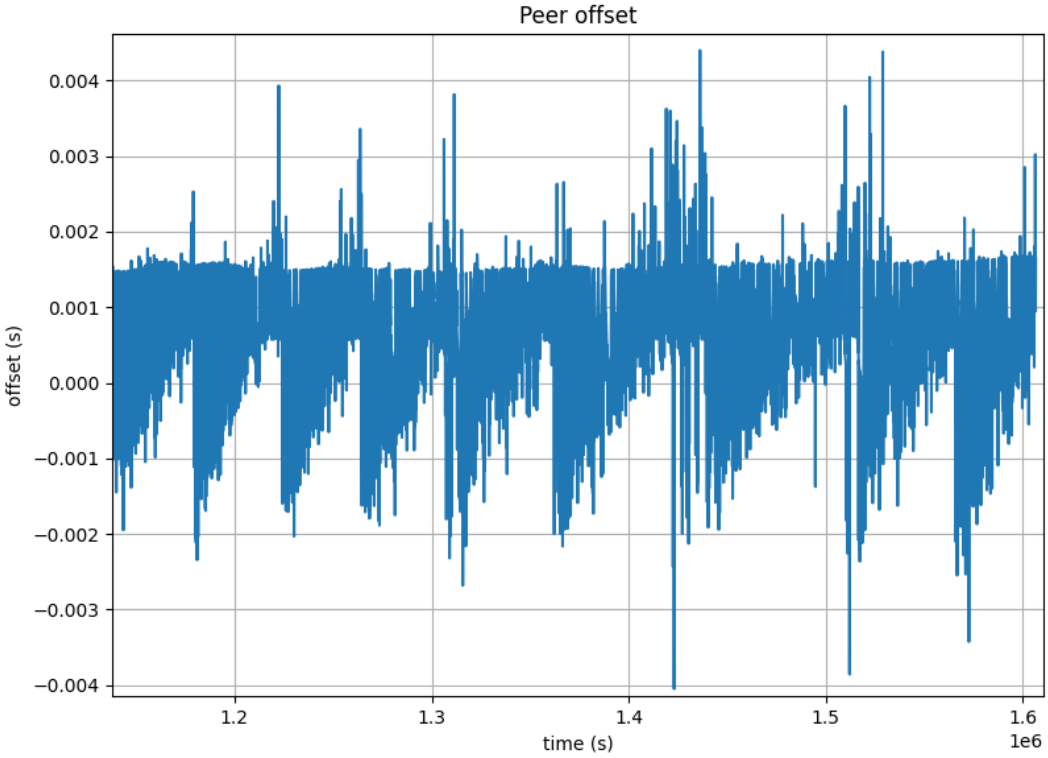

Loopstats offset
offset (s)
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
time (s)
1e6

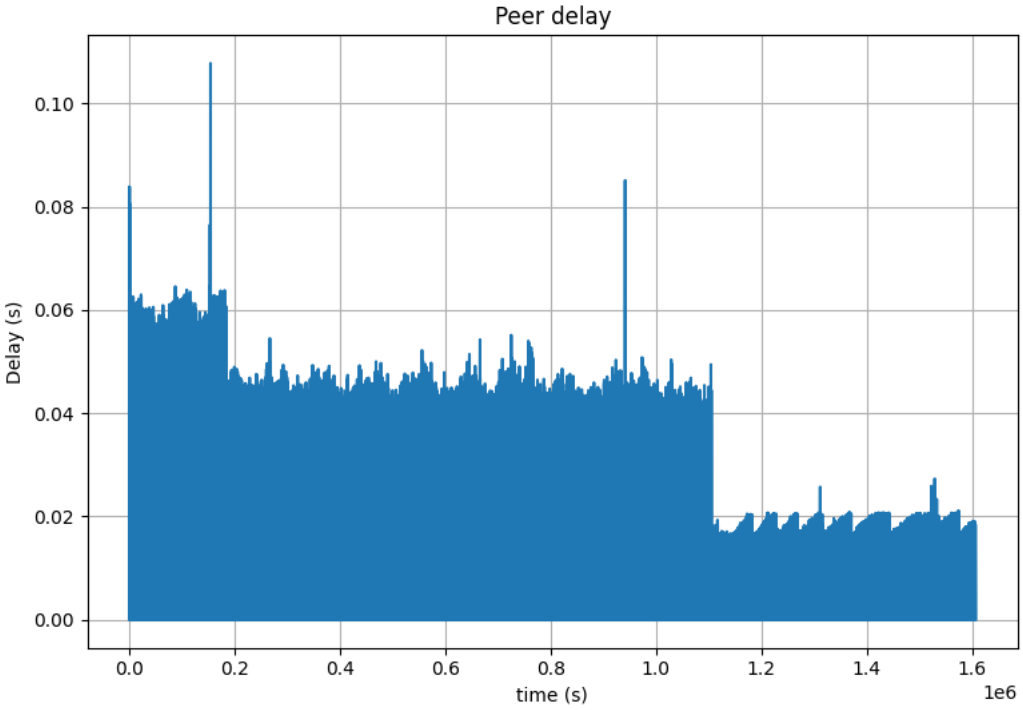

Loopstats freq offset (ppm)
freq offset (ppm)
500
400
300
200
100
0
−100
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
time (s)
1e6

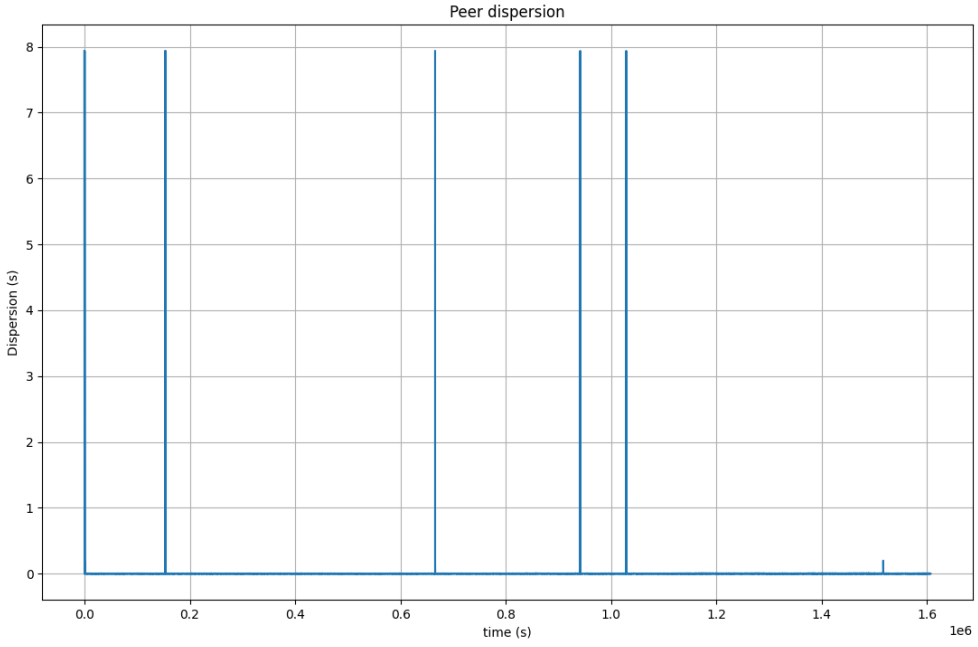
If you have a peerstats input file then you can plot three quantities for each peer:

- The *ntpd* estimated time offset from true time in seconds vs time, smaller values mean *ntpd* thinks it's closer to true time.
- The estimated round-trip time for *ntpd* packets vs time. Very small values would indicate the peer is on the local network.
- The dispersion value (seconds) indicates how spread out the offsets are for this particular peer.

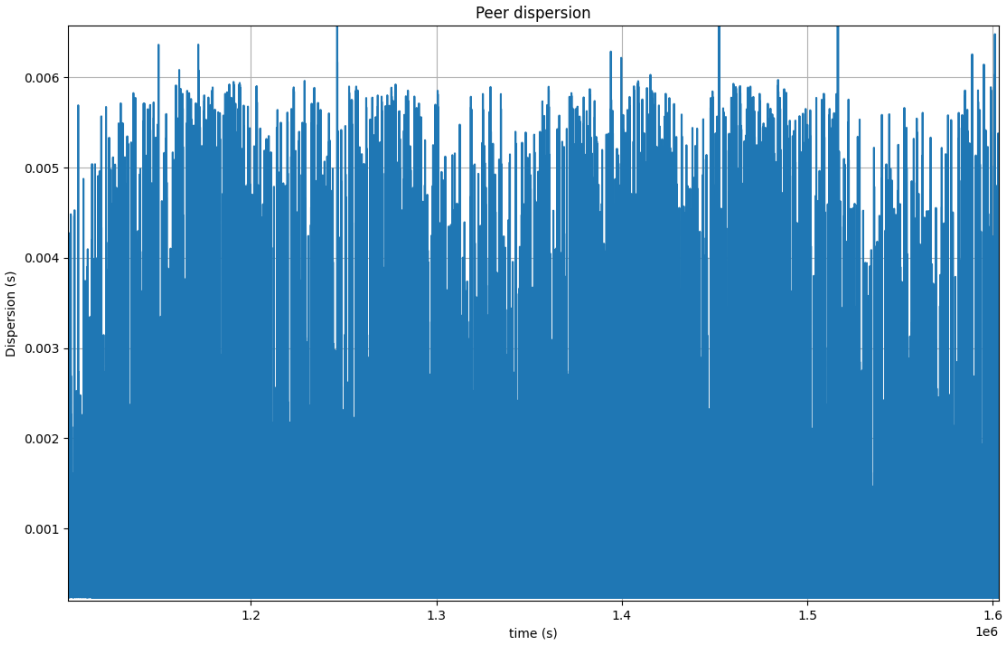
Here are examples of the above three plots:







That dispersion plot looks like there are a few outliers, so lets zoom in on a smaller section:


Peer dispersion
Dispersion (s)
0.006
0.005
0.004
0.003
0.002
0.001
1.2
1.3
1.4
1.5
1.6
1e6
time (s)

CHAPTER

THIRTEEN

COMMON FAILURE MODES

Certain failures of the Borealis system can be generally attributed to a short list of common issues.

13.1 N200 Power loss

If the power to any N200 device that Borealis is currently using is disrupted for a brief time, then the symptoms are typically:

- Driver message: "Timed out! RuntimeError fifo ctrl timed out looking for acks"
- The N200 that lost power will have all front panel aftermarket LEDs ON
- All other N200s will have the green RX LED ON.
- Radar stops

Restart the radar by:

- Ensuring the power is securely connected to all N200s
- */borealis/stop_radar.sh*
- */borealis/start_radar.sh*

13.2 N200 10MHz reference loss

If the 10MHz reference signal to any N200 device that Borealis is currently using is disrupted for an extended time (beyond a few seconds) then the symptoms are:

- Continual 'lates' from the driver ('L' printed out continuously)
- *REF locked* front panel LED will be off for the N200 that lost 10MHz reference
- Upon reconnection of the 10MHz signal, the lates continue
- Radar continues

Restart the radar by:

- Ensuring the 10MHz reference is connected to all N200s
- */borealis/stop_radar.sh*
- */borealis/start_radar.sh*

13.3 N200 PPS reference loss

If the Pulse Per Second (PPS) reference signal to any N200 device that Borealis is currently using is disrupted for an extended time (beyond a few seconds) then the symptoms are:

- None

13.4 N200 Ethernet loss

If the ethernet connection to any N200 device that Borealis is currently using is disrupted for a brief time, the symptoms are typically:

- Borealis software hangs
- After some time, the aftermarket front panel LEDS turn yellow, indicating an IDLE situation
- Radar stops

Restart the radar by:

- Reconnecting the Ethernet
- */borealis/stop_radar.sh*
- */borealis/start_radar.sh*

13.5 Borealis Startup with N200 PPS reference missing

If the Pulse Per Second (PPS) reference signal to any N200 device that Borealis will use upon startup is not connected, the symptoms are:

- Driver initialization doesn't proceed past initialization of the N200s.

NOTE This is as expected as the driver is waiting for a PPS signal to set the time registers

Start the radar by:

- Ensure PPS signal is connected to each N200

13.6 Octoclock GPS Power loss

If the master Octoclock (octoclock-g) unit loses power, then it no longer supplies 10MHz and PPS reference signals to the slave Octoclocks. The symptoms are:

- Octoclock slaves lose PPS and external 10MHz references (only the *power* LED is ON)
- All *ref lock* front panel LEDs on all N200s are OFF
- Continual lates from the driver (may take a few minutes for this symptom to manifest)

Start the radar by:

- Ensure Octoclock-g has power connected, and GPS antenna is connected
- */borealis/stop_radar.sh*
- */borealis/start_radar.sh*

- The driver will wait for GPS lock before initializing the N200s and starting the radar.

NOTE This may take a long time, and depends upon many factors including the antenna view of satellites, how long the octoclock-g has been powered off, temperature, etc. In testing it locked within 20 minutes.

13.7 TXIO Cable disconnect from N200 or Transmitter

If the cable carrying differential signals to/from the transmitters and the N200s is removed, or has failed in some way, then some possible results are:

- Transmitter will not transmit if the T/R signal is missing, this would be most obvious error
- Transmitter Low Power and AGC Status signals may not be valid when read from the N200 GPIO
- Transmitter may not be able to be placed into test mode

To fix this issue, ensure that all connectors are secured.

13.8 Shared memory full/Borealis unable to delete shared memory

NOTE If you've just installed Borealis, this may be caused by a missing *h5copy* binary. Make sure you have it installed for your operating system. For new versions of Ubuntu this means installing *hdf5-tools*. For OpenSuSe it means installing *hdf5*.

This may also be caused by the realtime/datawrite modules not deleting the individual record files. This is tied to issue [#203](https://github.com/SuperDARNCanada/borealis/issues/203), so check that the individual record files in the data output directory are being deleted after being copied, and check the realtime logs to verify that realtime is running properly.

If the shared memory location written to by Borealis is full, or the shared memory files are unable to be deleted by Borealis, then some possible results are:

- N200's may be in RX only mode (green LED on front panel will be on only)
- Borealis may appear to halt when viewing the screen, or Borealis may be getting very few sequences transmitted per integration time (1-2 within seconds)
- Signal processing may quietly die
- Data files, shared memory files and log files will cease being written

To fix this issue and restart the radar:

- Make sure the *h5copy* binary is installed for your system
- remove all Borealis created files in the */dev/shm* directory
- */borealis/stop_radar.sh*
- */borealis/start_radar.sh*

13.9 remote_server.py Segfaults, other programs segfault (core-dump)

This behaviour has been seen several times at the Saskatoon Borealis radar. The root cause is unknown, but symptoms are:

- Radar stops with nothing obvious in the logs or on the screen session
- Attempting to start the radar with *start_radar.sh* results in a segfault
- Attempting to reboot the computer results in segfaults, bus errors, core dumps, etc

To fix this issue and restart the radar:

- Power cycle the machine

13.10 'CPU stuck' messages from kernel, not possible to reboot

This behaviour has been seen once at the Clyde River Borealis radar. The message shown is:

Message from syslogd@clyborealis at Jun 15 00:47:18 ... kernel:[9941421.042914] NMI watchdog: BUG: soft lockup - CPU#19 stuck for 22s! [kworker/u56:0:16764]

The root cause is unknown, but symptoms are:

- Radar stops with the same message across all screens and terminals from the kernel
- Attempting to reboot the computer results in nothing happening etc

To fix this issue and restart the radar:

- Power cycle the machine

CHAPTER

FOURTEEN

GLOSSARY

array
: In SuperDARN data, the array data refers to the data after it has been beamformed and all antennas are combined into one array dataset. Typically the SuperDARN antennas are divided into the main antenna array and one interferometer antenna array.

averaging period
: A time during which sequences are transmitted repeatedly with the intent to average the received samples together. Averaging period is often used interchangeably with integration time.

channel
: This term is often used to denote frequency channels, but in USRPs it is also often used to denote the different transmit and receive physical ports, in which case for SuperDARN the different USRP channels would denote different antennas. We have tried to avoid the use of this term due to the ambiguity.

device
: When using Ettus UHD API this refers to the radio devices, or the N200s in the case of Borealis.

integration time
: The time allocated for an averaging period. An averaging period can be defined by the integration time (during which as many sequences as possible are transmitted); or simply by the number of sequences to transmit for the averaging period. Integration time is often used interchangeably with averaging period.

host
: A local machine; for Borealis this is the Borealis computer.

nave
: number of averages; equivalent to number of sequences transmitted or number of sampling periods received.

record
: A recorded subset of data. In SuperDARN data, a record contains all data for an integration time, and in the rawacf data the data is already averaged from the integration time.

sampling period
: The receive sampling time allocated to a transmitted sequence.

sequence
: A pulse sequence to be transmitted. Each sequence has a sampling period, which extends past the length of the pulse sequence for some time dependent on the number of ranges to be sampled.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

r

s

u

INDEX

E

F

G

O

P

R

S

T

U

V

X