Color Key, Programs
Site Library, Cyan
USRP Server, Orange
Control Program, Blue
CUDA Driver, Green 4
USRP Driver, Magenta
Tasks/Libraries, Yellow 4

Color Key, Status
Under Development, Incomplete, Red
Complete, Tested, Black
Complete. Untested, Purple 5

for information on running the software, see launch_all.sh

launch sequence:
- **run usrp_driver**
- **run cuda_driver**
- **run usrp_server**
- **launch control programs**

control program (e.g uafscan), not modified from stock ros/rst

determine site,
initialize local variables,
load site-specific site library

run SiteStart,
initialize global variables,
parse command line arguments

run SiteSetupRadar,
request channel with SETUP_RADAR_CHAN
run GET_PARAMETERS
create shared memory buffer for base-band samples
initialize logging
start FitACF File

run SiteTimeSeq
Construct pulse sequence timing from global variables using TSGMake
Register pulse sequence with USRP Server with SET_PARAMETERS
Update global variables with pulse sequence information (txpl, smsep, lagfr..)

loop while exitpoll == 0,
- run SiteExit, quit and cleanup if exit_flag is set
- create a new fitacf file if at a two hour mark (TASK_CLOSE, TASK_OPEN)
- set clrfreq flag if clrfreq argument is set
- set xcf flag if xcf argument is set
- set next beam
- read clock, determine daytime/nighttime frequency
- call SiteStartIntt
  - send PING
  - send GET_PARAMETERS
  - overwrite parameters with current beam number, channel, baseband rate
  - send back SET_PARAMETERS with updated struct
- call SiteFCLR if a clear frequency search is warranted
  - send SET_PARAMETERS with current beam, sample rate..
  - send REQUEST_CLEAR_FREQ_SEARCH command search freqeuncy, bandwidth, and averaging
  - send REQUEST_ASSIGNED_FREQ, return assigned tfreq
- call SiteIntegrate
  - exit if exit flag is set or if lag table is invalid
  - initialize sample buffer
  - until the end of the integration period keep requesting pulse sequences
    - send SET_PARAMETERS with tfreq
    - send SET_READY_FLAG
    - send GET_DATA, and if the status is good,
      - receive beam-formed base-band samples from USRP Server
      - receive bad T/R data
      - receive transmitter status
    - send GET_PARAMETERS for actual transmit frequency, etc
    - log time
    - decode phase coding
    - copy samples to buffer
    - calcuate ACF
    - return number of averages in integration period
- populate structs with data, send to tasks
- call SiteEndScan
  - send SET_INACTIVE
  - send PING

log exit
free memory (argtable, strings..)
run SiteExit, send QUIT message

USRP server (usrp_server.py)

radar state machine, start in INIT state
INIT
- enter WAIT state

WAIT
- scan radar channel manager object states
- enter CLRFREQ state if a channel requests it
- enter PRETRIGGER state if a channel requests it
- enter TRIGGER state if a channel requests it
- enter GET_DATA state if a channel requests it

CLRFREQ
- run clrfreq handler on requesting channels
  - unpack clrfreq request information
  - request current usrp time with UHD_GETTIME
  - broadcast CLRFREQ command for clear frequency search
  - average and beamform received samples
  - compute FFT on samples
  - **select clear frequency**
  - pass frequency back to channel manager
- return to WAIT

PRETRIGGER
- if all channels are requesting PRETRIGGER, run pretrigger handler
  - load pulse time
  - calculate sample counts
  - send USRP_SETUP command to usrp driver
  - send CUDA_ADD_CHANNEL for each channel
  - send CUDA_GENERATE_PULSE
- return to WAIT

TRIGGER
- if all channels are requesting TRIGGER, run trigger handler
  - send TRIGGER_PULSE
- return to WAIT

GET_DATA
- if all channels are requesting GET_DATA, run get_data handler
  - create arrays for baseband samples
  - send READY_DATA to check usrp driver status
  - return all channels managers to WAIT state
  - issue CUDA_GET_DATA command
  - **copy samples from CUDA driver to control program**
- return to WAIT

initialize logging
create radar hardware manager object
- open sockets for control programs
- connect to usrp drivers
- connect to CUDA drivers
- initialize RF front end

run radar hardware manager, spawn radar state machine thread
wait for control program connections, spawn radar channel handler threads for each program

radar channel manager

wait for commands from control program, process them and track radar channel state

SET_RADAR_CHAN
receive the channel and radar number from the control program

GET_DATA
wait until channel is in WAIT state, then enter GET_DATA state
transmit baseband samples, **bad T/R times**, and transmitter status

QUERY_INI_SETTINGS
return site.ini parameter, left for compatibility with existing control programs
appears to only be used for ifmode

GET_PARAMETERS
receive parameter struct from control program

REQUEST_CLEAR_FREQUENCY_SEARCH
unpack clear frequency parameter struct from control program
enter clear frequency search request state

SET_ACTIVE
appears unused

SET_PARAMETERS
return current parameters to control program

PING
reply to ping

QUIT
close sockets and cleanly exit

SET_READY
enter trigger state

REGISTER_SEQ
accept sequence struct from control program, unpack run length encoding
extract pulse timing, T/R gates, and phase coding from sequence
create masks for phase coding and T/R gates
create pulse offsets and lengths vectors from pulse sequence

REQUEST_ASSIGNED_FREQ
wait until clear frequency search ends,
then return clear frequency and noise

USRP driver (usrp_driver.cpp, recv_and_hold.cpp, burst_worker.cpp, dio.cpp, usrp_utils.cpp))

parse configuration files (driver_config.ini), extract shared memory buffer sizes
parse command line arguments
enter INIT state
open USRP with UHD driver, setup clock, sync to external 10 MHz and PPS
initialize RF front end
open shared memory sample buffer

wait for socket connection from usrp server
connect, and wait for commands

USRP_SETUP
accept frequency and pulse sequence information from from usrp server
copy in transmit samples from shared memory
if the frequency has changed, retune the USRP
enter READY state

RXFE_SET
accept amplifier and attenuator settings from usrp server
configure RF front end settings using USRP GPIO

TRIGGER_PULSE
if in READY proceed to PULSE state
calculate transmit pulse times from pulse information and current time
launch transmit pulse worker thread (burst_worker.cpp)
launch receive worker thread (recv_and_hold.cpp)

READY_DATA
send status, antenna number, and number of samples to usrp server
copy RF rate receive samples to shared memory with CUDA driver

UHD_GETTIME
return the current synchronized USRP time (relative to last PPS synchronization)

CLRFREQ
get clear frequency search center frequency, time, and bandwidth from usrp server
store existing settings, then retune USRP using UHD to new center frequency
setup USRP to receive sample starting at given time
collect samples, then send them back to usrp server over the socket connection
retune USRP and restore prior sampling rate

EXIT
gracefully exit, close down sockets and semaphores, unmap shared memory

CUDA driver (cuda_driver.py, tx_cuda.cu, rx_cuda.cu)

parse configuration files (usrp_config.ini, driver_config.ini, array_config.ini)
create a GPU object for each USRP
create shared memory buffers for RF samples from each USRP
listen for socket connections from USRP server
wait for commands from USRP server, process with below command handlers

CUDA_SETUP
run cuda setup command on all GPU objects

CUDA_GENERATE_PULSE
synthesize transmit pulse in GPU objects
(see tx_cuda.cu)
copy samples to shared memory

CUDA_ADD_CHANNEL
populate channel information from USRP server, store in GPU objects

CUDA_PULSE_INIT
currently unused/unimplemented

CUDA_REMOVE_CHANNEL
currently unused/unimplemented

CUDA_GET_DATA
retrieve processed baseband samples from GPU
store in memory shared with USRP server

CUDA_PROCESS
process received RF samples in shared memory on GPU
(see rx_cuda.cu)

CUDA_EXIT
clean exit of cuda driver

python libraries

drivermsg_library.py contains code to manage passing commands over sockets

dsp_filters.py is for filters to shape baseband transmit pulses.

phasing_utils.py has a few small functions to help with beamforming.

rosmsg.py contains classes to interpret and replicate rosmsg, dataprm, and other structs sent by control programs.

socket_utils.py has functions for socket communication

shm_library.py has functions for shared memory