CRTS

Generated by Doxygen 1.8.6

Sun Jul 24 2016 04:43:58

Contents

Chapter 1

CRTS

About:

The Cognitive Radio Test System (CRTS) provides a flexible framework for over the air test and evaluation of cognitive radio (CR) networks. Users can rapidly define new testing scenarios involving a large number of CR's and interferers while customizing the behavior of each node individually. Execution of these scenarios is simple and the results can be quickly visualized using octave/matlab logs that are kept throughout the experiment.

CRTS evaluates the performance of CR networks by generating network layer traffic at each CR node and logging metrics based on the received packets. Each CR node will create a virtual network interface so that CRTS can treat it as a standard network device. Part of the motivation for this is to enable development of upper layer protocols. The CR object/process can be anything with such an interface. We are currently working on examples of this in standard SDR frameworks e.g. GNU Radio. A block diagram depicting the test process run on a CR node by CRTS is depicted below.

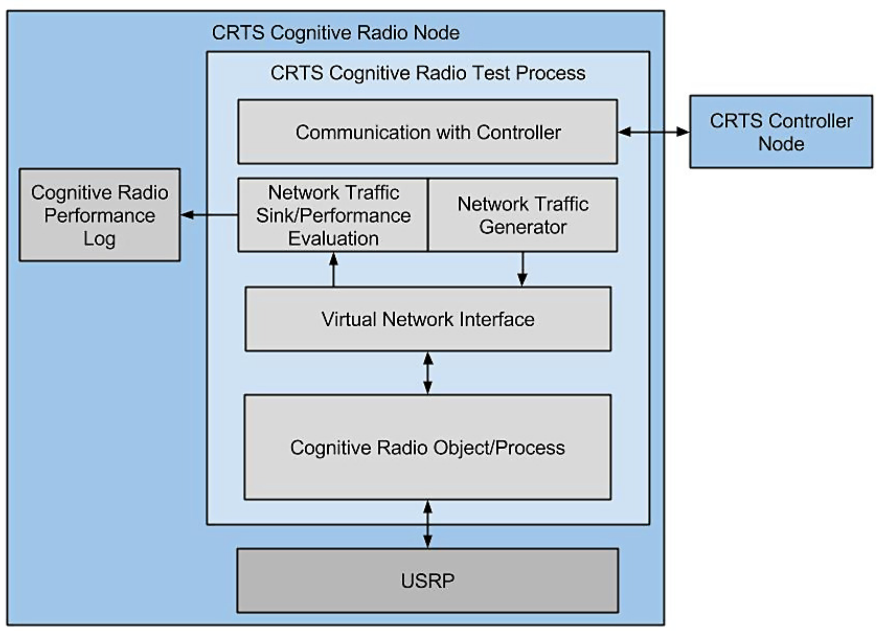

Figure 1.1: Cogntive Radio Test Process

In addition to this CR test framework, a particular CR has been developed with the goal of providing a flexible, generic structure to enable rapid development and evaluation of cognitive engine (CE) algorithms. This CR has been named the Extensible Cognitive Radio (ECR). The basic idea of the ECR is rather simple, a CE is fed data and metrics relating to the current operating point of the radio. It can then make decisions and exert control over the radio to improve its performance. A block diagram of the ECR is shown below.

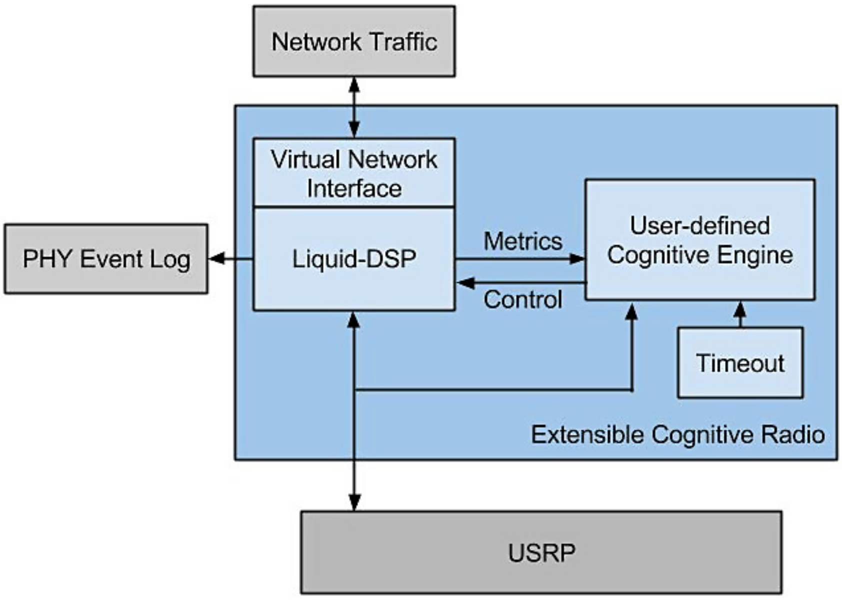

Figure 1.2: The Extensible Cognitive Radio

The ECR uses the `OFDM Frame Generator` of `liquid-dsp` and uses an `Ettus` Univeral Software Radio Peripheral (USRP).

CRTS is being developed using the `CORNET` testbed under Virginia Tech's `Wireless` Research Group.

Installation:

Dependencies

CRTS is being developed on `Ubuntu 14.04` but should be compatible with most Linux distributions. To compile and run CRTS and the ECR, your system will need the following packages. If a version is indicated, then it is recommended because it is being used in CRTS development.

- `UHD Version 3.8.4`
- `liquid-dsp commit a4d7c80d3`
- libconfig-dev

CRTS also relies on each node having network synchronized clocks. On CORNET this is accomplished with Network Time Protocol (NTP). Precision Time Protocol (PTP) would work as well.

Note to CORNET users: These dependencies are already installed for you on all CORNET nodes. You will still need to download the CRTS source repository and compile it however.

Downloading and Configuring CRTS

Official releases of CRTS can be downloaded from the `Releases Page` while the latest development version is available on the main `Git Page`.

Note that because using CRTS involves actively writing and compiling cognitive engine code, it is not installed like traditional software.

Official Releases

In the following commands 'v1.0' should be replaced with the release version you wish to download.

1. Download the latest released version of CRTS from the `Official Releases Page`:

   ```
   $ wget -O crts-v1.0.tar.gz https://github.com/ericps1/crts/archive/v1.0.tar.gz
   ```

2. Unzip the archive and move into the main source tree:

   ```
   $ tar xzf crts-v1.0.tar.gz
   $ cd crts-v1.0/
   ```

3. Compile the code with:

   ```
   $ make
   ```

4. Then configure the system to allow certain networking commands without a password (CORNET users can skip this step):

   ```
   $ sudo make install
   ```

The last step should only ever need to be run once. It configures the system to allow all users to run certain specific networking commands which are necessary for CRTS. They are required because CRTS creates, manipulates, and tears down a virtual network interface upon each run. The commands may be found in the .crts_sudoers file.

To undo these changes, simply run:

```
$ sudo make uninstall
```

Latest Development Version

1. Download the git repository:

   ```
   $ git clone https://github.com/ericps1/crts.git
   ```

2. Move into the main source tree:

   ```
   $ cd crts/
   ```

3. Compile the code with:

   ```
   $ make
   ```

4. Then configure the system to allow certain networking commands without a password (CORNET users should skip this step):

   ```
   $ sudo make install
   ```

The last step should only ever need to be run once. It configures the system to allow all users to run certain very specific networking commands which are necessary for CRTS. They are required because CRTS creates and tears down a virtual network interface upon each run. The commands may be found in the .crts_sudoers file.

To undo these changes, simply run:

```
$ sudo make uninstall
```

An Overview

CRTS is designed to run on a local network of machines, each with their own dedicated USRP. A single node, the `crts_controller`, will automatically launch each radio node for a given scenario and communicate with it as the scenario progresses.

Each radio node in a test scenario could be

1. A member of a CR network (controlled by `crts_cr`) or
2. An interfering node (controlled by `crts_interferer`), generating particular noise or interference patterns against which the CR nodes must operate.

In the next sections we provide an overview of the high level components used by CRTS to enable flexible, scalable testing of CR's.

Scenarios

The crts_controller will run the tests specified by a scenario master configuration file. The default configuration file is scenario_master_template.cfg. A different configuration file may be used by providing a -f option to crts_-controller. The configuration file specifies the number of scenarios to be run, their names, and the number of times each scenario will be run which can be specified once for all scenarios, or for each individual scenario. If both methods are used, the number provided for the specific scenario will take precedence. Syntax for specifying these parameters must align with the provided configuration files.

If you just want to run a single scenario, you may simply provide a -s argument to crts_controller to avoid having to edit a configuration file.

Scenarios are defined by configuration files in the scenarios/ directory. Each of these files will specify the number of nodes in the experiment and the duration of the experiment. Each node will have additional parameters that must be specified. These parameters include but are not limited to:

- The node's type: cognitive radio or interferer.
- The node's local IP address.
- If it is a CR node, it further defines:
  - The type of the CR (e.g. if it uses the ECR or some external CR).
  - The node's virtual IP address in the CR network.
  - The virtual IP address of the node it initially communicates with.
  - The network traffic pattern (stream, burst, or Poisson)
  - If the CR node uses the ECR, it will also specify:
    - Which cognitive engine to use.
    - The initial configuration of the CR.
    - What logs should be kept during the experiment.
- If it is an interferer node, it further defines:
  - The type of interference (e.g. OFDM, GMSK, RRC, etc.).
  - The paremeters of the interferer's operation.
  - What type of data should be logged.

In some cases a user may not care about a particular setting e.g. the forward error correcting scheme. In this case, the setting may be neglected in the configuration file and the default setting will be used.

We've provided a number of scenario files useful for orientation to CRTS and for software testing and performance testing of the radios

Scenario Controllers

Scenario controllers provide a centralized and customizable way to receive feedback and exert control over a scenario's operation in real time. A simple API can be used to enable or disable specific types of feedback from each node involved in the scenario, receive said feedback, and even directly control the scenario test parameters e.g. the network throughput as well as the operating parameters of the radio e.g. its transmit power.

We have found several significant use cases for this functionality. It provides a nice way to automate performance testing of the radios, it provides an easy way to create dynamic test conditions such as network loads, and it creates a central hub through which other applications can interface e.g. CORNET 3D can now be used to visualize tests as they happen and allows for human control of the radios which can be useful for tutorials.

The behavior of the scenario controller is defined by two functions. The initialize node feedback function is called at the beginning of the scenario so that feedback can be setup once since this will often be a static setting. The execute function implements the behavior of the scenario controller. It is triggered whenever feedback is received or after a certain period of time has passed, specified by the sc_timeout_ms parameter in scenario config files.

To make a new scenario controller a user needs to define a new scenario controller subclass. The SC_Template.cpp and SC_Template.hpp can be used as a guide in terms of the structure and API. Once the SC has been defined it can be integrated into CRTS by running $ ./config_scenario_controllers and $ make in the top directory.

The Extensible Cognitive Radio

As mentioned above, the ECR uses an OFDM based waveform defined by liquid-dsp. The cognitive engine will be able to control the parameters of this waveform such as the number of subcarriers, subcarrier allocation, cyclic prefix length, modulation scheme, and more. The cognitive engine will also be able to control the settings of the RF front-end USRP including its gains, sampling rate, center frequency, and digital mixing frequency. See the code documentation for more details.

Currently the ECR does not support much in the way of MAC layer functionality, e.g. there is no ARQ or packet segmentation/concatenation. This is planned for future development.

Cognitive Engines in the ECR

The Extensible Cognitive Radio provides an easy way to implement cognitive engines. This is accomplished through inheritance i.e. a particular cognitive engine can be implemented as a subclass of the cognitive engine base class and seamlessly integrated with the ECR. The general structure is such that the cognitive engine has access to any information related to the operation of the ECR via get() function calls as well as metrics passed from the receiver DSP. It can then control any of the operating parameters of the radio using set() function calls defined for the ECR.

The cognitive engine is defined by an execute function which can be triggered by several events. The engine will need to respond accordingly depending on the type of event that occurred. The event types include the reception of a physical layer frame, a timeout, USRP overflows and underruns, and transmission complete events.

To make a new cognitive engine a user needs to define a new cognitive engine subclass. The CE_Template.cpp and CE_Template.hpp can be used as a guide in terms of the structure, and some of the other examples show how the CE can interact with the ECR. Once the CE has been defined it can be integrated into CRTS by running $ ./config_cognitive_engines and $ make in the top directory.

Other source files in the cognitive_engine directory will be automatically linked into the build process. This way you can define other classes that your CE could instantiate. To make this work, a cpp file that defines a CE must be named beginning with "CE_" as in the examples.

- Any cpp files defining a cognitive engine must begin with "CE_" as in the examples! ∗

Installed libraries can also be used by a CE. For this to work you'll need to manually edit the makefile by adding the library to the variable LIBS which is located at the top of the makefile and defines a list of all libraries being linked in the final compilation.

One particular function that users should be aware of is ECR.set_control_information(). This provides a generic way for cognitive radios to exchange control information without impacting the flow of data. The control information is 6 bytes which are placed in the header of the transmitted frame. It can then be extracted in the cognitive engine at the receiving radio. A similar function can be performed by transmitting a dedicated control packet from the CE.

Examples of cognitive engines are provided in the `cognitive_engines/` directory.

Cognitive Radios in Python

Along with cognitive engines defined by the ECR, CRTS also supports cognitive radios written in python. An example of a very simple python cognitive radio can be found in cognitive_radios/python_txrx.py. When using a python radio, the scenario file must specify the cr_type ("python") and a python_file (python_txrx.py in the example scenario). The python file must be in the top-level cognitive_radios folder. In addition, you can supply arguments to pass to your radio by using the arguments field. An example scenario file for a python radio can be found in scenarios/example_scenarios/python_flowgraph_example.cfg.

Interferers

The testing scenarios for CRTS may involve generic interferers. There are a number of parameters that can be set to define the behavior of these interferers. They may generate CW, GMSK, RRC, OFDM, or AWGN waveforms. Their behavior can be defined in terms of when they turn off and on by the period and duty cycle settings, and there frequency behavior can be defined based on its type, range, dwell time, and increment.

Logs

CRTS will log packet transmission and reception details at the network layer if the appropriate flags are set in the scenario configuration file. Each entry will include the number of bytes sent or received, the packet number, and a timestamp. These may be used to look at network layer metrics such as dropped packets or latency. Note that latency calculations can only be as accurate as the synchronization between the server nodes.

The ECR will also log frame transmission and reception parameters and metrics at the physical layer if the appropriate flags are set in the scenario configuration file.

All of the aforementioned logs are written as binary files in the /logs/bin directory during the scenario's execution. These logs will be automatically converted to either Python or Octave/Matlab scripts and placed in the /logs/octave or /logs/python directories after the scenario has finished. These scripts provide the user with an easy way to import data from experiments. Other scripts can then be written to analyze these results. We've provided some basic Octave/Matlab scripts which plot the contents of the logs as a function of time and display some basic statistics.

In the case where a scenario is run more than once (using the scenario_reps field in the master_scenario_file.cfg), the data from all repetitions will be held in a single Octave script. Rather than a single array for each parameter there will be a cell array for each parameter, each element of the cell array is an array which comprises the results from a particular repetition. This is done to facilitate analysis across the repetitions.

Chapter 2

Provided Scenarios

1. two_channel_dsa (located in example_scenarios/)

This simple DSA scenario assumes that there are two CRs operating in FDD and with two adjacent and equal bandwidth channels (per link) that they are permitted to use. A nearby interferer will be switching between these two channels on one of the links, making it necessary for the CR's to dynamically switch their operating frequency to realize good performance.

2. two_channel_dsa_pu (located in example_scenarios/)

This simple DSA scenario assumes that there are two radios considered primary users (PU) and two cognitive seconday user (SU) radios. There are two adjacent and equal bandwidth channels (per link) that the cognitive radios are permitted to use. The PU's will switch their operating frequency as defined in their "cognitive engines," making it necessary for the SU CR's to dynamically switch their operating frequency to realize good performance and to avoid significantly disrupting the PU links.

3. basic_two_node_network (located in example_scenarios/)

This scenario creates the most basic two node CR network. No actual cognitive/adaptive behavior is defined by the cognitive engines in this scenario, it is intended as the most basic example for a user to become familiar with CRTS. Note how initial subcarrier allocation can be defined in three ways. In this scenario, we use the standard allocation method which allows you to define guard band subcarriers, central null subcarriers, and pilot subcarrier frequency, as well as a completely custom allocation method where we specify each subcarrier or groups of subcarriers. In this example we use both methods to create equivalent subcarrier allocations.

4. fec_adaptation (located in example_scenarios/)

This example scenario defines two CR's that will adapt their transmit FEC scheme based on feedback from the receiver. A dynamic interferer is introduced to make adaptation more important.

5. interferer_test (located in example_scenarios/)

This scenario defines a single interferer (used for development/testing)

6. mod_adaptation (located in example_scenarios/)

This example scenario defines two CR's that will adapt their transmit modulation scheme based on feedback from the receiver. A dynamic interferer is introduced to make adaptation more important.

7. network_loading (located in example_scenarios/)

This example scenario sets up two CR nodes which have asymmetric network loads. The network loads are then periodically swapped by the scenario controller. The cognitive engines used in this scenario will adapt their bandwidths based on the network loads that they detect.

8. python_flowgraph_example (located in example_scenarios/)

This scenario demonstrates how to setup and use a cognitive radio written in python rather than the ECR.

9. subcarrier_alloc_test (located in test_scenarios/)

This example scenario just uses a single node to illustrate how subcarrier allocation can be changed on the fly by the CE. If you run uhd_fft on a nearby node before running this scenario you can observe the initial subcarrier allocation defined in the scenario configuration file followed by switching between a custom allocation and the default liquid-dsp allocation.

Chapter 3

Provided Cognitive Engines

We have put together several example CE's to illustrate some of the features and capabilities of the ECR. Users are encouraged to reference these CE's to get a better understanding of the ECR and how they might want to design their own CE's, but should be aware that there is nothing optimal about these examples.

1. CE_Two_Channel_DSA_Link_Reliability (located in example_engines/)

This CE is intended for the 2 Channel DSA scenario. It operates by switching channels whenever it detects that the link is bad, assuming the source of error to be from the interferer. Once the decision is made at the receiver, the node will update control information transmitted to the other node, indicating the new frequency it should transmit on.

2. CE_Two_Channel_DSA_PU (located in primary_user_engines/)

This CE is used to create a primary user for the 2 Channel DSA PU scenario. The PU will simply switch it's operating frequencies at some regular interval.

3. CE_Two_Channel_DSA_Spectrum_Sensing (located in example_engines/)

This CE is similar to the fist CE listed, but makes its adaptations based on measured channel power rather than based on reliability of the link. The transmitter changes its center frequency based on sensed channel power whereas the receiver will change its center frequency when it has not received any frames for some period of time.

4. CE_FEC_Adaptation (located in example_engines/)

This CE determines which FEC scheme is appropriate based on the received signal quality and updates its control information so that the transmitter will use the appropriate scheme. This is just a demonstration, no particular thought was put into the switching points.

5. CE_Mod_Adaptation (located in example_engines/)

This CE determines which modulation scheme is appropriate based on the received signal quality and updates its control information so that the transmitter will use the appropriate scheme. This is just a demonstration, no particular thought was put into the switching points.

6. CE_Template

This CE makes no adaptations but serves as a template for creating new CE's.

7. CE_Subcarrier_Alloc (located in test_engines/)

This CE illustrates how a CE can change the subcarrier allocation of its transmitter. The method for setting the receiver subcarrier allocation is identical.

8. CE_Network_Loading (located in example_engines/)

When a pair of these CEs are communicating they will negotiate to adapt their occupied bandwidths based on the network loads they detect. Note that this example simply shows bandwidth adaptation based on network load, the bands actually being used by the radios do not overlap so they aren't really sharing spectrum.

9. CE_Simultaneous_RX_And_Sensing (located in test_engines/)

This CE simply demonstrates how the ECR can receive OFDM frames and pass the received samples to the CE for spectrum sensing or other signal processing.

10. CE_Throughput_Test (located in test_engines/)

This CE merely tests the ability of the ECR to predict it's own achieved throughput assuming zero packet errors. From observation the numbers appear to align reasonably well.

11. CE_Control_and_Feedback_Test (located in test_engines/)

This CE periodically turns its transmissions on and off and adjusts its transmit power. This is simply to verify the scenario controller's ability to receive feedback on the CE's parameters.

Chapter 4

Provided Cognitive Radio

1. python_txrx.py

This is a simple python flowgraph that was developed using GNURadio-companion. It reads data from the tunCRTS virtual interface using the TUNTAP_PDU block, and passes it along to a second node via a Socket_PDU block operating in server mode. The second node receives the data from a Socket PDU block operating in client mode, then passes it to the TUNTAP PDU block to write the data to the tunCRTS virtual interface. This radio can be used as a template for other python radios, by replacing the Socket_PDU blocks with USRP blocks to send the data over the air, rather than over the network.

Chapter 5

Tutorial 1: Running CRTS

In this tutorial we go over the basic mechanics of how to configure CRTS to run a test scenario or a batch of test scenarios and view the results.

Begin by selecting a set of three nodes that you will use to run a basic scenario using CRTS. Be sure to choose nodes that are close enough for reliable communication. If you are using CORNET, choose three adjacent nodes. You can view the floorplan {http://www.cornet.wireless.vt.edu/CORNET3D/CORNET3-D/cornet_3d_full/CORNET3D/index.html}{here}. This floorplan also shows the status for each node. Be sure to choose nodes that are bright green, indicating that they have working USRP's.

Once you've selected three nodes, open ssh terminals to them by running the command below. If you are using CORNET, the username will be your CORNET username. Note that the node ports are also displayed on the floorplan.

```
$ ssh -XC -p <node port> <username>@128.173.221.40
```

If you didn't check the CORNET floorplan to see that your nodes had working USRP's or if you're using a testbed other than CORNET, run the following command on each node to double check that the USRP's are available.

```
$ uhd_find_devices
```

If a node does not have access to it's USRP either power cycle the USRP if you have access to it or just try another node.

Navigate to the crts directory on each node (this is assuming you've already followed the installation instructions) and open up the scenario_master_template.cfg file. This file defines the number of scenarios that will be run when CRTS is executed along with their names and optionally how many times these scenarios should be repeated. In this tutorial we're going to run the basic_two_node_network scenario , which consists of two cognitive radio nodes that will communicate with one another. The scenario_master_template file should already be setup to run this scenario. A minimalist version of this would look like the following.

```
num_scenarios = 1;
reps_all_scenarios = 1;
scenario_1 = "example_scenarios/basic_two_node_network";
```

Now open the scenario configuration file scenarios/example_scenarios/basic_two_node_network. As mentioned earlier, this file defines a basic scenario involving two CR nodes. Familiarize yourself with the overall structure; there are some general scenario parameters at the top followed by node declarations which have their own parameters. You may also want to look at the scenarios/scenario_template.cfg file for a more detailed description of all the parameters. In this scenario we use the CE_Template cognitive engine which is basically a placeholder i.e. it does not make any decisions. Check to make sure that all of the print and log flags are set to 1 so that we can view results during and after the scenario runs.

Now that we've looked at how scenarios are configured in CRTS, lets actually run one. First launch the controller. CRTS can be run in a 'manual' or 'automatic' mode. The default behavior is to run in automatic mode; manual mode is specified by a -m flag after the controller command. Manual mode can be very useful for debug purposes when you develop complex cognitive engines later on. If you want to run in automatic mode, make sure the server_ip parameters for both nodes point to the nodes you want to use. On the node you want to act as the controller, run:

```
$ ./crts_controller -m
```

Now you can start the CRTS cognitive radio processes on the other two nodes.

```
$ ./crts_cognitive_radio -a <controller ip>
```

The controller IP needs to be specified so the program knows where to connect. On CORNET the internal ip will be 192.168.1.$<$external port number -6990$>$. You can double check the ip by running ifconfig on the controller node.

Once you've started the two CR nodes, observe that they have received their operating parameters and will shortly begin to exchange frames over the air. Metrics for the received frames should be printed out to both terminals. When you run in automatic mode this output will be stored in the logs/stdout directory.

Once the scenario has finished running go to the /logs/octave directory. You should see several auto-generated .m files starting with basic_two_node_network_$*$. To view a plot of the network throughput vs. time for each node run:

```
$ octave
> basic_two_node_network_node_<node number>_net_rx
> plot_cognitive_radio_net_rx
```

You can also view plots of the physical layer transmitted and received frames.

```
> basic_two_node_network_node_<node number>_phy_tx
> plot_cognitive_radio_phy_tx

> basic_two_node_network_node_<node number>_phy_rx
> plot_cognitive_radio_phy_rx
```

Troubleshooting:

- If you are seeing issues with your radio links e.g. no frames are being received or there is a significant number of frames being received in error, a first measure check would be to look at the transmit and receive gains for each node. Depending on the physical placement of the nodes and the environment you may need to use higher gains to overcome path loss or in some cases you may need to reduce your gain to avoid clipping the ADC of the USRP.
- If you don't see the generated octave log files, return to the scenario file and make sure all of the options including the word log are set equal to 1.

Chapter 6

Tutorial 2: Interferers

In this tutorial we go over how to use an interferer in a CRTS test scenario and the options available to define the interferers behavior.

As in the previous tutorial, select a set of three nearby nodes in your testbed and open ssh terminals to each.

You can either create a new scenario master file to run this scenario, or simply run crts_controller with a -s option. If you create a new scenario master file, it should look like the following:

```
num_scenarios = 1;
reps_all_scenarios = 1;
scenario_1 = "test_scenarios/interferer_test";
```

and you would run the command:

```
$ ./crts_controller -m -f <new scenario master file name without .cfg extension>
```

Running the following command would be equivalent in this case:

```
$ ./crts_controller -m -s test_scenarios/interferer_test
```

Now open the scenarios/test_scenarios/interferer_test.cfg file to see the scenario definition. You may also refer to the scenarios/scenario_template.cfg file for a more detailed description of each parameter for an interferer node. For the first execution let's set the following parameters.

```
tx_rate = 1e6;
interference_type = "rrc";
period = 4.0;
duty_cycle = 1.0;
tx_freq_behavior = "fixed";
```

On another node, open uhd_fft so that we can see the interferer's transmissions; run the following command

```
$ uhd_fft -f <freq> -s <rate> -g <gain>
```

where freq should match the tx_freq defined in the interferer_test.cfg file, rate should be greater than or equal to the tx_rate defined in the Interferer_Test.cfg file, and gain should be set based on the physical separation of the nodes. On CORNET, a gain of 10-20 dB is usually good.

You should now see a plot of the spectrum where the interferer will transmit. If there is already a signal present you may want to change to a different band (one which you have a license for of course). Remember to also change the tx_freq parameter in the scenario file.

Return to the first node and run the CRTS controller as shown above.

Finally, on a third node run

```
$ ./crts_interferer -a <controller ip>
```

You should now see a constant signal in the middle of the spectrum. It should have the root-raised-cosine shape.

Now go back to the scenario file and edit the duty cycle to be 0.5. Rerun CRTS and you should see the same signal which will alternate between being on for duty_cycle∗period seconds and then off for (1-duty_cycle)∗period.

Now let's look at dynamic frequency behavior. Go back to the scenario configuration file and set the following properties:

```
duty_cycle = 1.0;
tx_freq_behavior = "sweep";
tx_freq_min = <tx_freq-5e6>;
tx_freq_max = <tx_freq+5e6>;
tx_freq_dwell_time = 1.0;
tx_freq_resolution = 1.0e6;
```

Also make sure the log flags are set as shown below.

```
log_phy_tx = 1;
generate_octave_logs = 1;
```

Close uhd_fft and rerun it so we can see the full band the interferer will be transmitting in:

```
$ uhd_fft -f <tx_freq> -s 10e6 -g <gain>
```

Rerun CRTS and you should now see a signal which will sweep back and forth across the viewable spectrum, changing frequencies once every second.

Now move into the logs/octave directory. You should see a file called interferer_test_node_1_interferer_phy_tx.m. If you do, run the following to see a plot of the interferer's transmission parameters as a function of time throughout the scenario's execution.

```
$ octave
>> interferer_test_node_1_int_phy_tx
>> plot_interferer_phy_tx
```

If you'd like, play around with some of the settings. You might try changing tx_freq_behavior to "random", changing the interference_type, or trying some combination of dynamic frequency behavior and duty cycle.

Chapter 7

Tutorial 3: Writing a Cognitive Engine

In this tutorial we go through the procedure to define a new cognitive engine, make it available to the ECR, and run a scenario with it. If you haven't already, you may find it useful to review the documentation on the ECR and CE's found in the crts-manual.pdf.

Specifically, we'll be making a simple CE which calculates some statistics and prints them out periodically. We'll also demonstrate how the CE can exert control over the ECR's operation and observe how this impacts the statistics. The statistics we will track include the number of received frames, the average error vector magnitude, the average packet error rate, and the average received signal strength indicator. We'll also modify the transmit gain periodically so that we'll observe some changes in the statistics over time.

Move to the cognitive_engine directory in your local CRTS repository. Make copies of the cognitive engine template files. Note that in order to properly integrate the CE into CRTS, the header and source files should begin with 'CE_' and end in 'hpp' and 'cpp' respectively. This is done to identify the CE sources so that they can be integrated into the ECR code.

```
$ mkdir CE_Tutorial_3
$ cp example_cognitive_engines/CE_Template/CE_Template.cpp CE_Tutorial_3/CE_Tutorial_3.cpp
$ cp example_cognitive_engines/CE_Template/CE_Template.hpp CE_Tutorial_3/CE_Tutorial_3.hpp
```

With these two files we will be defining a new class for our cognitive engine. Edit both files so that each instance of 'CE_Template' is replaced with 'CE_Tutorial_3'. Also edit the define statements at the top of the header file from *CE_TEMPLATE* to *CE_TUTORIAL_3*.

Now open up CE_Tutorial_3.hpp so we can add some necessary class members. We'll need timers in order to know when to print the statistics out and update the transmitter gain along with constants to represent how frequently this should be done and by how much the transmit gain should be increased. We'll also need a counter for the number of frames received and how many were invalid. Finally, we'll need to sum the error vector magnitude and received signal strength indicator. So in total we need to add the following members.

```
const float print_stats_period_s = 1.0;
timer print_stats_timer;
const float tx_gain_period_s = 1.0;
const float tx_gain_increment = 1.0;
time tx_gain_timer;
int frame_counter;
int frame_errs;
float sum_evm;
float sum_rssi;
```

Now open up CE_Tutorial_3.cpp so we can implement our CE. First we need to initialize all of our members in the constructor like so:

```
print_stats_timer = timer_create();
timer_tic(print_stats_timer);
tx_gain_timer = timer_create();
timer_tic(tx_gain_timer);
frame_counter = 0;
frame_errs = 0;
sum_evm = 0.0;
sum_rssi = 0.0;
```

Let's also make sure we clean up the timers in the destructor.

```
timer_destroy(print_stats_timer);
timer_destrpy(tx_gain_timer);
```

Now let's move on to the core of the CE, the execute function. Note that the template has set up a generic structure to deal with each of the possible events which can trigger the CE execution. So at this point we should be considering what we want to happen for each event. We need to update the class members to keep track of the statistics of interest. All of these statistics are based on received frames, so we should update them whenever a PHY event happens. Add the following code under the switch case for PHY events.

```
frame_counter++;
if (!ECR->CE_metrics.payload_valid)
  frame_errs++;
sum_evm += pow(10.0, ECR->CE_metrics.stats.evm/10.0);
sum_rssi += pow(10.0, ECR->CE_metrics.stats.rssi/10.0);
```

Note that EVM and RSSI are reported in dB, but to acquire an average we need to convert them to linear units.

We said that we wanted to print statistics every print_stats_period_s seconds. This doesn't depend on a particular event, so let's write this functionality in a block of code before the event switch. We want to check the elapsed time, print the statistics if enough time has elapsed, and then we'll need to reset the variables used to track statistics. We should also cover the case when zero frames have been received. Something like the following should do the trick.

```
if(timer_toc(print_stats_timer) > print_stats_period_s){
  if (frame_counter>0) {
    printf("Updated Received Frame Statistics:\n");
    printf("  Frames Received: %i\n", frame_counter);
    printf("  Average EVM:     %f\n", 10.0*log10(sum_evm/(float)frame_counter));
    printf("  Average PER:     %f\n", (float)frame_errs/(float)frame_counter);
    printf("  Average RSSI:    %f\n\n", 10.0*log10(sum_rssi/(float)frame_counter));

    // reset timer and statistics
    timer_tic(print_stats_timer);
    frame_counter = 0;
    frame_errs = 0;
    sum_evm = 0.0;
    sum_rssi = 0.0;
  } else {
    printf("Updated Received Frame Statistics:\n");
    printf("  Frames Received: 0\n");
    printf("  Average EVM:     -\n");
    printf("  Average PER:     -\n");
    printf("  Average RSSI:    -\n\n");
  }
}
```

Note that we report EVM and RSSI in dB and so must apply another conversion.

Now that we have written code to track and display some statistics on the received frames, let's make a modification to the ECR's transmission so we can observe changes in the statistics over time. We need to make sure that the gain stays within the possible values, something like below would work. This should be placed above the event switch, just like the other timer-based code.

```
if(timer_toc(tx_gain_timer) > tx_gain_period_s){
  timer_tic(tx_gain_timer);

  float current_tx_gain = ECR->get_tx_gain();
  if(current_tx_gain < 25.0)
    ECR->set_tx_gain(current_gain + tx_gain_increment);
  else
    ECR->set_tx_gain(0.0);
}
```

Now that we've established the desired functionality for our CE, we need to configure CRTS so that we can use it, and recompile the code. This is accomplished simply by running the following from the CRTS root directory.

```
$ ./config_cognitive_engines
$ make
```

You should see you newly defined cognitive engine appear in the list of the included cognitive engines.

Next, we'll need to define a scenario that uses this new CE. Since we'll just be using two CR's, simply copy the scenarios/test_scenarios/basic_two_node_network file to wherever you might like within the scenarios directory e.g.:

```
$ cd scenarios
$ cp example_scenarios/basic_two_node_network.cfg tutorial_3.cfg
```

Open up tutorial_3.cfg. At the very top let's change the run time to be a bit longer.

```
run_time = 60.0;
```

Let's also edit both nodes to have an initial transmit gain of 0, and of course we need to use our new CE. Let's also disable metric printing so we can focus on the statistics we've used in our CE. Make the following changes for both nodes.

```
tx_gain = 0;
CE = "CE_Tutorial_3";
print_metrics = 0;
```

Now we can run the scenario using the same procedure as in the first tutorial. Login to three nodes on your testbed.

On node 1 run:

```
(.sh}
 $ ./crts_controller -m -s tutorial_3
```

On nodes 2 and 3:

```
$ ./crts_cognitive_radio -a <controller ip>
```

You should see updated statistics being printed to the screen once every second on nodes 2 and 3. You should further observe decreasing EVM and increasing RSSI. Note that depending on the distance between the two nodes you may not detect frames at the lower gain settings or you might have distortion/clipping issues at the higher gains levels.

If you are having troubles, here are the completed files that you can compare against.

// CE_Tutorial_3.hpp

```
#ifndef _CE_TUTORIAL_3_
#define _CE_TUTORIAL_3_

#include "CE.hpp"
#include "timer.h"

class CE_Tutorial_3 : public Cognitive_Engine {

private:
  // internal members used by this CE
  const float print_stats_period_s = 1.0;
  timer print_stats_timer;
  const float tx_gain_period_s = 1.0;
  const float tx_gain_increment = 1.0;
  timer tx_gain_timer;
  int frame_counter;
  int frame_errs;
  float sum_evm;
  float sum_rssi;

public:
  CE_Tutorial_3();
  ~CE_Tutorial_3();
  virtual void execute(ExtensibleCognitiveRadio *ECR);
};

#endif
```

// CE_Tutorial_3.cpp

```
#include "ECR.hpp"
#include "CE_Tutorial_3.hpp"

// constructor
CE_Tutorial_3::CE_Tutorial_3(int argc, char **argv, ExtensibleCognitiveRadio *_ECR)
       {

  // save the ECR pointer (this should not be removed)
  ECR = _ECR;

  print_stats_timer = timer_create();
  timer_tic(print_stats_timer);
  tx_gain_timer = timer_create();
  timer_tic(tx_gain_timer);
  frame_counter = 0;
  frame_errs = 0;
  sum_evm = 0.0;
  sum_rssi = 0.0;
}

// destructor
CE_Tutorial_3::~CE_Tutorial_3() {
  timer_destroy(print_stats_timer);
  timer_destroy(tx_gain_timer);
}

// execute function
void CE_Tutorial_3::execute(ExtensibleCognitiveRadio *ECR) {

  if (timer_toc(tx_gain_timer) > tx_gain_period_s) {
    timer_tic(tx_gain_timer);

    float current_tx_gain = ECR->get_tx_gain_uhd();
    if(current_tx_gain < 25.0)
      ECR->set_tx_gain_uhd(current_tx_gain + tx_gain_increment);
    else
      ECR->set_tx_gain_uhd(0.0);
  }

  if (timer_toc(print_stats_timer) > print_stats_period_s) {
    timer_tic(print_stats_timer);

    if (frame_counter>0) {
      printf("Updated Received Frame Statistics:\n");
      printf("  Frames Received: %i\n", frame_counter);
      printf("  Average EVM:     %f\n", 10.0*log10(sum_evm/(float)frame_counter));
      printf("  Average PER:     %f\n", (float)frame_errs/(float)frame_counter);
      printf("  Average RSSI:    %f\n\n", 10.0*log10(sum_rssi/(float)frame_counter));

      // reset statistics
      frame_counter = 0;
      frame_errs = 0;
      sum_evm = 0.0;
      sum_rssi = 0.0;
    } else {
      printf("Updated Received Frame Statistics:\n");
      printf("  Frames Received: 0\n");
      printf("  Average EVM:     -\n");
      printf("  Average PER:     -\n");
      printf("  Average RSSI:    -\n");
    }
  }

  switch(ECR->CE_metrics.CE_event) {
    case ExtensibleCognitiveRadio::TIMEOUT:
      // handle timeout events
      break;
    case ExtensibleCognitiveRadio::PHY:
      // handle physical layer frame reception events
      frame_counter++;
      if (!ECR->CE_metrics.payload_valid)
        frame_errs++;
      sum_evm += pow(10.0, ECR->CE_metrics.stats.evm/10.0);
      sum_rssi += pow(10.0, ECR->CE_metrics.stats.rssi/10.0);
      break;
    case ExtensibleCognitiveRadio::UHD_OVERFLOW:
      // handle UHD overflow events
      break;
    case ExtensibleCognitiveRadio::UHD_UNDERRUN:
      // handle UHD underrun events
      break;
    case ExtensibleCognitiveRadio::USRP_RX_SAMPS:
      // handle samples received from the USRP when simultaneously
      // running the receiver and performing additional sensing
      break;
  }
}
```

// Tutorial_3.cfg

```
// general scenario parameters
num_nodes = 2;
run_time = 60.0;

// Node 1
node1 : {
  // general node parameters
  node_type = "cognitive_radio";
  cognitive_radio_type = "ecr";
  server_ip = "192.168.1.38";

  // network parameters
  crts_ip = "10.0.0.2";
  target_ip = "10.0.0.3";
  net_traffic_type = "stream";
  net_mean_throughput = 2e6;

  // cognitive engine parameters
  cognitive_engine = "CE_Tutorial_3";
  ce_timeout_ms = 200.0;

  // log/report settings
  print_metrics = 0;
  log_phy_rx = 1;
  log_phy_tx = 1;
  log_net_rx = 1;
  log_net_tx = 1;
  generate_octave_logs = 1;

  // initial USRP settings
  rx_freq = 862.5e6;
  rx_rate = 2e6;
  rx_gain = 10.0;
  tx_freq = 857.5e6;
  tx_rate = 2e6;
  tx_gain = 0.0;

  // initial liquid OFDM settings
  tx_gain_soft = -12.0;
  tx_modulation = "bpsk";
  tx_crc = "crc32";
  tx_fec0 = "v27";
  tx_fec1 = "none";
  // tx_cp_len = 16;
  // rx_cp_len = 16;

  tx_subcarriers = 32;
  tx_subcarrier_alloc_method = "standard";
  tx_guard_subcarriers = 4;
  tx_central_nulls = 6;
  tx_pilot_freq = 4;

  rx_subcarriers = 32;
  rx_subcarrier_alloc_method = "standard";
  rx_guard_subcarriers = 4;
  rx_central_nulls = 6;
  rx_pilot_freq = 4;
};

// Node 2
node2 : {
  // general node parameters
  type = "cognitive_radio";
  cognitive_radio_type = "ecr";
  server_ip = "192.168.1.39";

  // virtual network parameters
  crts_ip = "10.0.0.3";
  target_ip = "10.0.0.2";
  net_traffic_type = "stream";
  net_mean_throughput = 2e6;

  // cognitive engine parameters
  cognitive_engine = "CE_Tutorial_3";
  ce_timeout_ms = 200.0;

  // log/report settings
  print_metrics = 0;
  log_phy_rx = 1;
  log_phy_tx = 1;
  log_net_rx = 1;
  log_net_tx = 1;
  generate_octave_logs = 1;

  // initial USRP settings
```

```
  rx_freq = 857.5e6;
  rx_rate = 2e6;
  rx_gain = 10.0;
  tx_freq = 862.5e6;
  tx_rate = 2e6;
  tx_gain = 0.0;

  // initial liquid OFDM settings
  tx_gain_soft = -12.0;
  tx_modulation = "bpsk";
  tx_crc = "crc32";
  tx_fec0 = "v27";
  tx_fec1 = "none";
  tx_delay_us = 1e3;
  // tx_cp_len = 16;
  // rx_cp_len = 16;

  tx_subcarriers = 32;
  tx_subcarrier_alloc_method = "custom";
  tx_subcarrier_alloc : {
    // guard band nulls
    sc_type_1 = "null";
    sc_num_1 = 4;

    // pilots and data
    sc_type_2 = "pilot";
    sc_type_3 = "data";
    sc_num_3 = 3;
    sc_type_4 = "pilot";
    sc_type_5 = "data";
    sc_num_5 = 3;
    sc_type_6 = "pilot";

    // central nulls
    sc_type_7 = "null";
    sc_num_7 = 6;

    // pilots and data
    sc_type_8 = "pilot";
    sc_type_9 = "data";
    sc_num_9 = 3;
    sc_type_10 = "pilot";
    sc_type_11 = "data";
    sc_num_11 = 3;
    sc_type_12 = "pilot";

    // guard band nulls
    sc_type_13 = "null";
    sc_num_13 = 4;
  }

  rx_subcarriers = 32;
  rx_subcarrier_alloc_method = "custom";
  rx_subcarrier_alloc : {
    // guard band nulls
    sc_type_1 = "null";
    sc_num_1 = 4;

    // pilots and data
    sc_type_2 = "pilot";
    sc_type_3 = "data";
    sc_num_3 = 3;
    sc_type_4 = "pilot";
    sc_type_5 = "data";
    sc_num_5 = 3;
    sc_type_6 = "pilot";

    // central nulls
    sc_type_7 = "null";
    sc_num_7 = 6;

    // pilots and data
    sc_type_8 = "pilot";
    sc_type_9 = "data";
    sc_num_9 = 3;
    sc_type_10 = "pilot";
    sc_type_11 = "data";
    sc_num_11 = 3;
    sc_type_12 = "pilot";

    // guard band nulls
    sc_type_13 = "null";
    sc_num_13 = 4;
  }
};
```

Chapter 8

Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 9

Class Documentation

9.1 ce_info Struct Reference

Public Attributes

- std::string **ce_name**
- std::string **ce_dir**

The documentation for this struct was generated from the following file:

- crts/src/config_cognitive_engines.cpp

9.2 CognitiveEngine Class Reference

The base class for the custom cognitive engines built using the ECR (Extensible Cognitive Radio).

`#include <cognitive_engine.hpp>`

Public Member Functions

- virtual void execute ()

  Executes the custom cognitive engine as defined by the user.

Public Attributes

- ExtensibleCognitiveRadio * **ECR**

9.2.1 Detailed Description

The base class for the custom cognitive engines built using the ECR (Extensible Cognitive Radio).

This class is used as the base for the custom (user-defined) cognitive engines (CEs) placed in the cognitive_-engines/ directory of the source tree. The CEs following this model are event-driven: While the radio is running, if certain events occur as defined in ExtensibZZleCognitiveRadio::Event, then the custom-defined execute function (Cognitive_Engine::execute()) will be called.

9.2.2 Member Function Documentation

9.2.2.1 virtual void CognitiveEngine::execute ( ) [virtual]

Executes the custom cognitive engine as defined by the user.

When writing a custom cognitive engine (CE) using the Extensible Cognitive Radio (ECR), this function should be defined to contain the main processing of the CE. An ECR CE is event-driven: When the radio is running, this Cognitive_Engine::execute() function is called if certain events, as defined in ExtensibleCognitiveRadio::Event, occur.

For more information on how to write a custom CE using using the ECR, see TODO:Insert refence here. Or, for direct examples, refer to the source code of the reimplementations listed below (in the cognitive_engines/ directory of the source tree).

The documentation for this class was generated from the following file:

- crts/include/cognitive_engine.hpp

9.3 crts_signal_params Struct Reference

Public Attributes

- int **type**
- int **node**
- int **mod**
- int **crc**
- int **fec0**
- int **fec1**
- double **freq**
- double **bandwidth**
- double **gain**

The documentation for this struct was generated from the following file:

- crts/include/CORNET_3D.hpp

9.4 ExtensibleCognitiveRadio Class Reference

Classes

- struct metric_s

  Contains metric information related to the quality of a received frame. This information is made available to the custom Cognitive_Engine::execute() implementation and is accessed in the instance of this struct: ExtensibleCognitiveRadio::CE_metrics.
- struct rx_parameter_s

  Contains parameters defining how to handle frame reception.
- struct rx_statistics
- struct tx_parameter_s

  Contains parameters defining how to handle frame transmission.

Public Types

- enum CE_Event {
  TIMEOUT = 0, PHY, TX_COMPLETE, UHD_OVERFLOW,
  UHD_UNDERRUN, USRP_RX_SAMPS }

  Defines the different types of CE events.
- enum FrameType { DATA = 0, CONTROL, UNKNOWN }

  Defines the types of frames used by the ECR.

Public Member Functions

- void **set_ce** (char ∗ce, int argc, char ∗∗argv)
- void **start_ce** ()
- void **stop_ce** ()
- void set_ce_timeout_ms (double new_timeout_ms)

  Assign a value to ExtensibleCognitiveRadio::ce_timeout_ms.
- double get_ce_timeout_ms ()

  Get the current value of ExtensibleCognitiveRadio::ce_timeout_ms.
- void set_ce_sensing (int ce_sensing)

  Allows you to turn on/off the USRP_RX_SAMPLES events which allow you to perform custom spectrum sensing in the CE while the liquid-ofdm receiver continues to run.
- void set_ip (char ∗ip)

  Used to set the IP of the ECR's virtual network interface.
- void set_tx_queue_len (int queue_len)

  Allows you to set the tx buffer length for the virtual network interface This could be useful in trading off between dropped packets and latency with a UDP connection.
- int get_tx_queued_bytes ()

  Returns the number of bytes currently queued for transmission.
- void dec_tx_queued_bytes (int n)

  Decrements the count of bytes currently queued for transmission This function is only used as a work around since tun interfaces don't allow you to read the number of queued bytes.
- void inc_tx_queued_bytes (int n)

  Increments the count of bytes currently queued for transmission This function is only used as a work around since tun interfaces don't allow you to read the number of queued bytes.
- void set_tx_freq (double _tx_freq)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_freq.
- void **set_tx_freq** (double _tx_freq, double _dsp_freq)
- void set_tx_rate (double _tx_rate)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_rate.
- void set_tx_gain_soft (double _tx_gain_soft)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_soft.
- void set_tx_gain_uhd (double _tx_gain_uhd)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_uhd.
- void **set_tx_antenna** (char ∗_tx_antenna)
- void set_tx_modulation (int mod_scheme)

  Set the value of `mod_scheme` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- void set_tx_crc (int crc_scheme)

  Set the value of `check` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- void set_tx_fec0 (int fec_scheme)

  Set the value of `fec0` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- void set_tx_fec1 (int fec_scheme)

  Set the value of `fec1` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.

- void set_tx_subcarriers (unsigned int subcarriers)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers.
- void set_tx_subcarrier_alloc (char ∗_subcarrierAlloc)

  Set ExtensibleCognitiveRadio::tx_parameter_s::subcarrierAlloc.
- void set_tx_cp_len (unsigned int cp_len)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::cp_len.
- void set_tx_taper_len (unsigned int taper_len)

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::taper_len.
- void set_tx_control_info (unsigned char ∗_control_info)

  Set the control information used for future transmit frames.
- void set_tx_payload_sym_len (unsigned int len)

  Set the number of symbols transmitted in each frame payload. For now since the ECR does not have any segmentation/concatenation capabilities, the actual payload will be an integer number of IP packets, so this value really provides a lower bound for the payload length in symbols.
- double get_tx_freq ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_freq.
- double get_tx_lo_freq ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_freq.
- int get_tx_state ()

  Return the value of ExtensibleCognitiveRadio::tx_state.
- double get_tx_dsp_freq ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_freq.
- double get_tx_rate ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_rate.
- double get_tx_gain_soft ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_soft.
- double get_tx_gain_uhd ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_uhd.
- char ∗ **get_tx_antenna** ()
- int get_tx_modulation ()

  Return the value of `mod_scheme` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- int get_tx_crc ()

  Return the value of `check` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- int get_tx_fec0 ()

  Return the value of `fec0` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- int get_tx_fec1 ()

  Return the value of `fec1` in ExtensibleCognitiveRadio::tx_parameter_s::fgprops.
- unsigned int get_tx_subcarriers ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers.
- void get_tx_subcarrier_alloc (char ∗subcarrierAlloc)

  Get current ExtensibleCognitiveRadio::tx_parameter_s::subcarrierAlloc.
- unsigned int get_tx_cp_len ()

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::cp_len.
- unsigned int get_tx_taper_len ()
- void **get_tx_control_info** (unsigned char ∗_control_info)
- double **get_tx_data_rate** ()
- void **start_tx** ()
- void **start_tx_burst** (unsigned int _num_tx_frames, float _max_tx_time_ms)
- void **stop_tx** ()
- void **reset_tx** ()
- void transmit_control_frame (unsigned char ∗_payload, unsigned int _payload_len)

  Transmit a control frame.

- void set_rx_freq (double _rx_freq)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_freq.
- void **set_rx_freq** (double _rx_freq, double _dsp_freq)
- void set_rx_rate (double _rx_rate)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_rate.
- void set_rx_gain_uhd (double _rx_gain_uhd)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_gain_uhd.
- void **set_rx_antenna** (char ∗_rx_antenna)
- void set_rx_subcarriers (unsigned int subcarriers)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::numSubcarriers.
- void set_rx_subcarrier_alloc (char ∗_subcarrierAlloc)

  Set ExtensibleCognitiveRadio::rx_parameter_s::subcarrierAlloc.
- void set_rx_cp_len (unsigned int cp_len)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::cp_len.
- void set_rx_taper_len (unsigned int taper_len)

  Set the value of ExtensibleCognitiveRadio::rx_parameter_s::taper_len.
- int get_rx_state ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_state.
- int get_rx_worker_state ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_worker_state.
- double get_rx_freq ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_freq.
- double get_rx_lo_freq ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_freq.
- double get_rx_dsp_freq ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_freq.
- double get_rx_rate ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_rate.
- double get_rx_gain_uhd ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_gain_uhd.
- char ∗ **get_rx_antenna** ()
- unsigned int get_rx_subcarriers ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::numSubcarriers.
- void get_rx_subcarrier_alloc (char ∗subcarrierAlloc)

  Get current ExtensibleCognitiveRadio::rx_parameter_s::subcarrierAlloc.
- unsigned int get_rx_cp_len ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::cp_len.
- unsigned int get_rx_taper_len ()

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::taper_len.
- void **get_rx_control_info** (unsigned char ∗_control_info)
- void **reset_rx** ()
- void **start_rx** ()
- void **stop_rx** ()
- void **start_liquid_rx** ()
- void **stop_liquid_rx** ()
- void **set_rx_stat_tracking** (bool state, float sec)
- float **get_rx_stat_tracking_period** ()
- struct rx_statistics **get_rx_stats** ()
- void **reset_rx_stats** ()
- void **print_metrics** (ExtensibleCognitiveRadio ∗CR)
- void **log_rx_metrics** ()
- void **log_tx_parameters** ()
- void **reset_log_files** ()

Public Attributes

- struct metric_s CE_metrics
  The instance of ExtensibleCognitiveRadio::metric_s made accessible to the Cognitive_Engine.
- std::complex< float > ∗ ce_usrp_rx_buffer
  USRP samples will be written to this buffer if the ce_sensing_flag is set.
- int ce_usrp_rx_buffer_length
  Length of the buffer for USRP samples.
- int **print_metrics_flag**
- int **log_phy_rx_flag**
- int **log_phy_tx_flag**
- char **phy_rx_log_file** [255]
- char **phy_tx_log_file** [255]
- std::ofstream **log_rx_fstream**
- std::ofstream **log_tx_fstream**
- uhd::usrp::multi_usrp::sptr **usrp_tx**
- uhd::tx_metadata_t **metadata_tx**
- uhd::usrp::multi_usrp::sptr **usrp_rx**
- uhd::rx_metadata_t **metadata_rx**

Private Member Functions

- void **update_rx_stats** (bool frame_received)
- void **update_rx_params** ()
- void **update_tx_params** ()
- void **transmit_frame** (unsigned int frame_type, unsigned char ∗_payload, unsigned int _payload_len)

Private Attributes

- Cognitive_Engine ∗ **CE**
- double ce_timeout_ms
  The maximum length of time to go without an event before executing the CE under a timeout event. In milliseconds.
- bool **ce_phy_events**
- int **ce_sensing_flag**
- pthread_t **CE_process**
- pthread_mutex_t **CE_mutex**
- pthread_mutex_t **CE_fftw_mutex**
- pthread_cond_t **CE_cond**
- pthread_cond_t **CE_execute_sig**
- bool **ce_thread_running**
- bool **ce_running**
- struct rx_statistics **rx_stats**
- bool **rx_stat_tracking**
- bool **reset_rx_stats_flag**
- float **rx_stat_tracking_period**
- char **known_net_payload** [CRTS_CR_PACKET_LEN]
- int **tunfd**
- char **tun_name** [IFNAMSIZ]
- int **tx_queued_bytes**
- int **tx_queue_len**
- char **systemCMD** [200]
- struct rx_parameter_s **rx_params**
- bool **update_rx_flag**

- bool **update_usrp_rx**
- bool **recreate_fs**
- bool **reset_fs**
- ofdmflexframesync **fs**
- unsigned int **frame_num**
- unsigned int **frame_uhd_overflows**
- std::complex< float > * **rx_buffer**
- size_t **rx_buffer_len**
- pthread_t **rx_process**
- pthread_mutex_t **rx_mutex**
- pthread_mutex_t **rx_params_mutex**
- pthread_cond_t **rx_cond**
- int **rx_state**
- int **rx_worker_state**
- bool **rx_thread_running**
- unsigned int **tx_frame_counter**
- timer **tx_timer**
- float **max_tx_time_ms**
- tx_parameter_s **tx_params**
- tx_parameter_s **tx_params_updated**
- bool **update_tx_flag**
- bool **update_usrp_tx**
- bool **recreate_fg**
- bool **reset_fg**
- ofdmflexframegen **fg**
- unsigned int **fgbuffer_len**
- std::complex< float > * **fgbuffer**
- unsigned char **tx_header** [8]
- unsigned int **frame_counter**
- unsigned int **numDataSubcarriers**
- double **tx_data_rate**
- int **update_tx_data_rate**
- unsigned int **num_tx_frames**
- pthread_t **tx_process**
- pthread_mutex_t **tx_mutex**
- pthread_mutex_t **tx_state_mutex**
- pthread_mutex_t **tx_params_mutex**
- pthread_cond_t **tx_cond**
- bool **tx_complete**
- bool **tx_thread_running**
- int **tx_worker_state**
- int **tx_state**

Static Private Attributes

- static int **uhd_msg**

Friends

- void * **ECR_ce_worker** (void *)
- void **uhd_msg_handler** (uhd::msg::type_t type, const std::string &msg)
- void * **ECR_rx_worker** (void *)
- int **rxCallback** (unsigned char *, int, unsigned char *, unsigned int, int, framesyncstats_s, void *)
- void * **ECR_tx_worker** (void *)

9.4.1 Member Enumeration Documentation

9.4.1.1 enum ExtensibleCognitiveRadio::CE_Event

Defines the different types of CE events.

The different circumstances under which the CE can be executed are defined here.

Enumerator

- ***TIMEOUT*** The CE had not been executed for a period of time as defined by ExtensibleCognitiveRadio::ce_-timeout_ms. It is now executed as a timeout event.
- ***PHY*** A PHY layer event has caused the execution of the CE. Usually this means a frame was received by the radio.
- ***TX_COMPLETE*** Indicates that the transmit worker has completed transmission of its final frame.
- ***UHD_OVERFLOW*** The receiver processing is not able to keep up with the current settings.
- ***UHD_UNDERRUN*** The transmitter is not providing samples fast enough the the USRP.
- ***USRP_RX_SAMPS*** This event enables the design of custom spectrum sensing which can be employed without interrupting the normal reception of frames.

9.4.1.2 enum ExtensibleCognitiveRadio::FrameType

Defines the types of frames used by the ECR.

Enumerator

- ***DATA*** The frame contains application layer data. Data frames contain IP packets that are read from the virtual network interface and subsequently transmitted over the air.
- ***CONTROL*** The frame was sent explicitly at the behest of another cognitive engine (CE) in the network and it contains custom data for use by the receiving CE. The handling of ExtensibleCognitiveRadio::DATA frames is performed automatically by the Extensible Cognitive Radio (ECR). However, the CE may initiate the transmission of a custom control frame containing information to be relayed to another CE in the network. A custom frame can be sent using ExtensibleCognitiveRadio::transmit_frame().
- ***UNKNOWN*** The Extensible Cognitve Radio (ECR) is unable to determine the type of the received frame. The received frame was too corrupted to determine its type.

9.4.2 Member Function Documentation

9.4.2.1 void ExtensibleCognitiveRadio::get_rx_subcarrier_alloc (char * *subcarrierAlloc*)

Get current ExtensibleCognitiveRadio::rx_parameter_s::subcarrierAlloc.

`subcarrierAlloc` should be a pointer to an array of size ExtensibleCognitiveRadio::rx_parameter_s::numSubcarriers. The array will then be filled with the current subcarrier allocation.

9.4.2.2 void ExtensibleCognitiveRadio::get_tx_subcarrier_alloc (char * *subcarrierAlloc*)

Get current ExtensibleCognitiveRadio::tx_parameter_s::subcarrierAlloc.

`subcarrierAlloc` should be a pointer to an array of size ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers. The array will then be filled with the current subcarrier allocation.

9.4.2.3 unsigned int ExtensibleCognitiveRadio::get_tx_taper_len ( )

Return the value of ExtensibleCognitiveRadio::tx_parameter_s::taper_len.

9.4.2.4 void ExtensibleCognitiveRadio::transmit_control_frame (unsigned char ∗ *_payload*, unsigned int *_payload_len*)

Transmit a control frame.

The cognitive engine (CE) can initiate transmission of a frame dedicated to control information by calling this function. `_payload` is an array of `unsigned char` and can be any length. It can contain any data as would be useful to the CE.

`_payload_len` is the number of elements in `_payload`.

9.4.3 Member Data Documentation

9.4.3.1 double ExtensibleCognitiveRadio::ce_timeout_ms `[private]`

The maximum length of time to go without an event before executing the CE under a timeout event. In milliseconds.

The CE is executed every time an event occurs. The CE can also be executed if no event has occured after some period of time. This is referred to as a timeout event and this variable defines the length of the timeout period in milliseconds.

It can be accessed using ExtensibleCognitiveRadio::set_ce_timeout_ms() and ExtensibleCognitiveRadio::get_ce_-timeout_ms().

The documentation for this class was generated from the following files:

- crts/include/extensible_cognitive_radio.hpp
- crts/src/extensible_cognitive_radio.cpp

9.5 feedback_struct Struct Reference

Public Attributes

- int **type**
- int **node**
- float **frequency**
- float **bandwidth**

The documentation for this struct was generated from the following file:

- crts/include/CORNET_3D.hpp

9.6 Interferer Class Reference

Public Member Functions

- void **start_tx** ()
- void **stop_tx** ()
- void **set_log_file** (char ∗)
- void **log_tx_parameters** ()
- void **UpdateFrequency** ()
- void **TransmitInterference** ()
- void **BuildCWTransmission** ()
- void **BuildNOISETransmission** ()
- void **BuildGMSKTransmission** ()
- void **BuildRRCTransmission** ()
- void **BuildOFDMTransmission** ()
- void **BuildAWGNTransmission** ()

Public Attributes

- int **interference_type**
- double **tx_gain_soft**
- double **tx_gain**
- double **tx_freq**
- double **tx_rate**
- double **period**
- double **duty_cycle**
- int **tx_freq_behavior**
- double **tx_freq_min**
- double **tx_freq_max**
- double **tx_freq_bandwidth**
- double **tx_freq_dwell_time**
- double **tx_freq_resolution**
- timer **duty_cycle_timer**
- timer **freq_dwell_timer**
- bool **log_tx_flag**
- std::ofstream **tx_log_file**
- char **tx_log_file_name** [100]
- std::ofstream **sample_file**
- std::default_random_engine **generator**
- std::normal_distribution< double > **dist**
- resamp2_crcf **interp**
- gmskframegen **gmsk_fg**
- firfilt_crcf **rrc_filt**
- ofdmflexframegenprops_s **fgprops**
- ofdmflexframegen **ofdm_fg**
- uhd::usrp::multi_usrp::sptr **usrp_tx**
- uhd::tx_metadata_t **metadata_tx**
- unsigned int **buffered_samps**
- std::vector< std::complex
  < float > > **tx_buffer**
- pthread_t **tx_process**
- pthread_mutex_t **tx_mutex**
- pthread_cond_t **tx_cond**
- bool **tx_running**
- bool **tx_thread_running**
- int **tx_state**

Friends

- void ∗ **Interferer_tx_worker** (void ∗)

The documentation for this class was generated from the following files:

- crts/include/interferer.hpp
- crts/src/interferer.cpp

9.7 ExtensibleCognitiveRadio::metric_s Struct Reference

Contains metric information related to the quality of a received frame. This information is made available to the custom Cognitive_Engine::execute() implementation and is accessed in the instance of this struct: ExtensibleCognitiveRadio::CE_metrics.

`#include <extensible_cognitive_radio.hpp>`

Public Attributes

- ExtensibleCognitiveRadio::CE_Event CE_event

  Specifies the circumstances under which the CE was executed.
- ExtensibleCognitiveRadio::FrameType CE_frame

  Specifies the type of frame received as defined by ExtensibleCognitiveRadio::FrameType.
- int control_valid

  Indicates whether the `control` information of the received frame passed error checking tests.
- unsigned char control_info [6]

  The control info of the received frame.
- unsigned char * payload

  The payload data of the received frame.
- int payload_valid

  Indicates whether the `payload` of the received frame passed error checking tests.
- unsigned int payload_len

  The number of elements of the `payload` array.
- unsigned int frame_num

  The frame number of the received ExtensibleCognitiveRadio::DATA frame.
- framesyncstats_s stats

  The statistics of the received frame as reported by `liquid-dsp`.
- uhd::time_spec_t time_spec

  The `uhd::time_spec_t` object returned by the `UHD` driver upon reception of a complete frame.

9.7.1 Detailed Description

Contains metric information related to the quality of a received frame. This information is made available to the custom Cognitive_Engine::execute() implementation and is accessed in the instance of this struct: ExtensibleCognitiveRadio::CE_metrics.

The members of this struct will be valid when a frame has been received which will be indicated when the ExtensibleCognitiveRadio::metric_s.CE_event == PHY. Otherwise, they will represent results from previous frames.

The valid members under a ExtensibleCognitiveRadio::PHY event are:

ExtensibleCognitiveRadio::metric_s::CE_frame,

ExtensibleCognitiveRadio::metric_s::control_valid,

ExtensibleCognitiveRadio::metric_s::control_info,

ExtensibleCognitiveRadio::metric_s::payload,

ExtensibleCognitiveRadio::metric_s::payload_valid,

ExtensibleCognitiveRadio::metric_s::payload_len,

ExtensibleCognitiveRadio::metric_s::frame_num,

ExtensibleCognitiveRadio::metric_s::stats, and

ExtensibleCognitiveRadio::metric_s::time_spec

9.7.2 Member Data Documentation

9.7.2.1 ExtensibleCognitiveRadio::CE_Event ExtensibleCognitiveRadio::metric_s::CE_event

Specifies the circumstances under which the CE was executed.

When the CE is executed, this value is set according to the type of event that caused the CE execution, as specified in ExtensibleCognitiveRadio::Event.

9.7.2.2 int ExtensibleCognitiveRadio::metric_s::control_valid

Indicates whether the `control` information of the received frame passed error checking tests.

Derived from `liquid-dsp`. See the `Liquid Documentation` for more information.

9.7.2.3 unsigned int ExtensibleCognitiveRadio::metric_s::frame_num

The frame number of the received ExtensibleCognitiveRadio::DATA frame.

Each ExtensibleCognitiveRadio::DATA frame transmitted by the ECR is assigned a number, according to the order in which it was transmitted.

9.7.2.4 unsigned int ExtensibleCognitiveRadio::metric_s::payload_len

The number of elements of the `payload` array.

Equal to the byte length of the `payload`.

9.7.2.5 int ExtensibleCognitiveRadio::metric_s::payload_valid

Indicates whether the `payload` of the received frame passed error checking tests.

Derived from `liquid-dsp`. See the `Liquid Documentation` for more information.

9.7.2.6 framesyncstats_s ExtensibleCognitiveRadio::metric_s::stats

The statistics of the received frame as reported by `liquid-dsp`.

For information about its members, refer to the `Liquid Documentation`.

9.7.2.7 uhd::time_spec_t ExtensibleCognitiveRadio::metric_s::time_spec

The `uhd::time_spec_t` object returned by the `UHD` driver upon reception of a complete frame.

This serves as a marker to denote at what time the end of the frame was received.

The documentation for this struct was generated from the following file:

- crts/include/extensible_cognitive_radio.hpp

9.8 node_parameters Struct Reference

Public Attributes

- int **node_type**
- int **cognitive_radio_type**
- char **python_file** [100]
- char **python_args** [2048]
- char **team_name** [200]
- char **server_ip** [20]
- char **crts_ip** [20]
- char **target_ip** [20]
- int **net_traffic_type**
- int **net_burst_length**

- double **net_mean_throughput**
- char **cognitive_engine** [100]
- double **ce_timeout_ms**
- char **ce_args** [2048]
- bool **print_rx_frame_metrics**
- bool **log_phy_rx**
- bool **log_phy_tx**
- bool **log_net_rx**
- bool **log_net_tx**
- char **phy_rx_log_file** [260]
- char **phy_tx_log_file** [260]
- char **net_rx_log_file** [260]
- char **net_tx_log_file** [260]
- int **generate_octave_logs**
- int **generate_python_logs**
- double **rx_freq**
- double **rx_rate**
- double **rx_gain**
- double **tx_freq**
- double **tx_rate**
- double **tx_gain**
- int **rx_subcarriers**
- int **rx_cp_len**
- int **rx_taper_len**
- int **rx_subcarrier_alloc_method**
- int **rx_guard_subcarriers**
- int **rx_central_nulls**
- int **rx_pilot_freq**
- char **rx_subcarrier_alloc** [2048]
- double **tx_gain_soft**
- int **tx_subcarriers**
- int **tx_cp_len**
- int **tx_taper_len**
- int **tx_modulation**
- int **tx_crc**
- int **tx_fec0**
- int **tx_fec1**
- int **tx_subcarrier_alloc_method**
- int **tx_guard_subcarriers**
- int **tx_central_nulls**
- int **tx_pilot_freq**
- char **tx_subcarrier_alloc** [2048]
- int **interference_type**
- double **period**
- double **duty_cycle**
- int **tx_freq_behavior**
- double **tx_freq_min**
- double **tx_freq_max**
- double **tx_freq_dwell_time**
- double **tx_freq_resolution**

The documentation for this struct was generated from the following file:

- crts/include/crts.hpp

9.9 node_struct Struct Reference

Public Attributes

- float **frequency**
- float **bandwidth**
- char **team_name** [200]
- int **role**
- int **node**

The documentation for this struct was generated from the following file:

- crts/include/CORNET_3D.hpp

9.10 num_nodes_struct Struct Reference

Public Attributes

- int **num_nodes**

The documentation for this struct was generated from the following file:

- crts/include/CORNET_3D.hpp

9.11 ExtensibleCognitiveRadio::rx_parameter_s Struct Reference

Contains parameters defining how to handle frame reception.

```
#include <extensible_cognitive_radio.hpp>
```

Public Attributes

- unsigned int numSubcarriers

  The number of subcarriers in the OFDM waveform generated by `liquid`.
- unsigned int cp_len

  The length of the cyclic prefix in the OFDM waveform generator from `liquid`.
- unsigned int taper_len

  The overlapping taper length in the OFDM waveform generator from `liquid`.
- unsigned char ∗ subcarrierAlloc

  An array of `unsigned char` *whose number of elements is ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers. Each element in the array should define that subcarrier's allocation.*
- double rx_gain_uhd

  The value of the hardware gain for the receiver. In dB.
- double rx_freq

  The receiver local oscillator frequency in Hertz.
- double rx_dsp_freq

  The transmitter NCO frequency in Hertz.
- double rx_rate

  The sample rate of the receiver in samples/second.

9.11.1 Detailed Description

Contains parameters defining how to handle frame reception.

The member parameters are accessed using the instance of the struct: ExtensibleCognitiveRadio::tx_params.

Note that for frames to be received successfully These settings must match the corresponding settings at the transmitter.

9.11.2 Member Data Documentation

9.11.2.1 unsigned int ExtensibleCognitiveRadio::rx_parameter_s::cp_len

The length of the cyclic prefix in the OFDM waveform generator from `liquid`.

See the `OFDM Framing Tutorial` for details.

9.11.2.2 unsigned int ExtensibleCognitiveRadio::rx_parameter_s::numSubcarriers

The number of subcarriers in the OFDM waveform generated by `liquid`.

See the `OFDM Framing Tutorial` for details.

9.11.2.3 double ExtensibleCognitiveRadio::rx_parameter_s::rx_dsp_freq

The transmitter NCO frequency in Hertz.

The USRP has an NCO which can be used to digitally mix the signal anywhere within the baseband bandwidth of the USRP daughterboard. This can be useful for offsetting the tone resulting from LO leakage of the ZIF receiver used by the USRP.

9.11.2.4 double ExtensibleCognitiveRadio::rx_parameter_s::rx_freq

The receiver local oscillator frequency in Hertz.

It can be accessed with ExtensibleCognitiveRadio::set_rx_freq() and ExtensibleCognitiveRadio::get_rx_freq().

This value is passed directly to `UHD`.

9.11.2.5 double ExtensibleCognitiveRadio::rx_parameter_s::rx_gain_uhd

The value of the hardware gain for the receiver. In dB.

Sets the gain of the hardware amplifier in the receive chain of the USRP. This value is passed directly to `UHD`.

It can be accessed with ExtensibleCognitiveRadio::set_rx_gain_uhd() and ExtensibleCognitiveRadio::get_rx_gain-_uhd().

Run

```
$ uhd_usrp_probe
```

for details about the particular gain limits of your USRP device.

9.11.2.6 double ExtensibleCognitiveRadio::rx_parameter_s::rx_rate

The sample rate of the receiver in samples/second.

It can be accessed with ExtensibleCognitiveRadio::set_rx_rate() and ExtensibleCognitiveRadio::get_rx_rate().

This value is passed directly to `UHD`.

9.11.2.7 unsigned char∗ ExtensibleCognitiveRadio::rx_parameter_s::subcarrierAlloc

An array of `unsigned char` whose number of elements is ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers. Each element in the array should define that subcarrier's allocation.

A subcarrier's allocation defines it as a null subcarrier, a pilot subcarrier, or a data subcarrier.

See `Subcarrier Allocation` in the `liquid` documentation for details.

Also refer to the `OFDM Framing Tutorial` for more information.

9.11.2.8 unsigned int ExtensibleCognitiveRadio::rx_parameter_s::taper_len

The overlapping taper length in the OFDM waveform generator from `liquid`.

See the `OFDM Framing Tutorial` and the `Liquid Documentation Reference` for details.

The documentation for this struct was generated from the following file:

- crts/include/extensible_cognitive_radio.hpp

9.12 ExtensibleCognitiveRadio::rx_statistics Struct Reference

Public Attributes

- int **frames_received**
- int **valid_frames**
- float **evm_dB**
- float **rssi_dB**
- float **per**
- float **ber**
- float **throughput**
- int **uhd_overflows**

The documentation for this struct was generated from the following file:

- crts/include/extensible_cognitive_radio.hpp

9.13 sc_feedback Struct Reference

Public Attributes

- int **node**
- char **fb_type**
- void ∗ **arg**

The documentation for this struct was generated from the following file:

- crts/include/scenario_controller.hpp

9.14 sc_info Struct Reference

Public Attributes

- std::string **sc_name**
- std::string **sc_dir**

The documentation for this struct was generated from the following file:

- crts/src/config_scenario_controllers.cpp

9.15 scenario_parameters Struct Reference

Public Attributes

- int **num_nodes**
- int64_t **start_time_s**
- int64_t **run_time**
- unsigned int **total_num_reps**
- unsigned int **rep_num**
- char **SC** [100]
- float **sc_timeout_ms**
- char **sc_args** [2048]

The documentation for this struct was generated from the following file:

- crts/include/crts.hpp

9.16 ScenarioController Class Reference

Public Member Functions

- virtual void **execute** ()
- virtual void **initialize_node_fb** ()
- void **set_sc_timeout_ms** (float t)
- void **set_node_parameter** (int node, char cont_type, void ∗_arg)
- void **receive_feedback** (int node, char fb_type, void ∗_arg)
- void **start_sc** ()
- void **stop_sc** ()

Public Attributes

- float **sc_timeout_ms** = 1.0
- int ∗ **TCP_nodes**
- struct scenario_parameters **sp**
- struct node_parameters **np** [48]
- struct sc_feedback **fb**
- int **sc_event**

Private Attributes

- pthread_t **sc_process**
- pthread_mutex_t **sc_mutex**
- pthread_cond_t **sc_cond**
- pthread_cond_t **sc_execute_sig**
- bool **sc_thread_running**
- bool **sc_running**

Friends

- void ∗ **sc_worker** (void ∗)

The documentation for this class was generated from the following files:

- crts/include/scenario_controller.hpp
- crts/src/scenario_controller.cpp

9.17 timer_s Struct Reference

Public Attributes

- struct timeval **tic**
- struct timeval **toc**
- int **timer_started**

The documentation for this struct was generated from the following file:

- crts/src/timer.cc

9.18 ExtensibleCognitiveRadio::tx_parameter_s Struct Reference

Contains parameters defining how to handle frame transmission.

`#include <extensible_cognitive_radio.hpp>`

Public Attributes

- unsigned int numSubcarriers

    The number of subcarriers in the OFDM waveform generated by `liquid`.
- unsigned int cp_len

    The length of the cyclic prefix in the OFDM waveform generator from `liquid`.
- unsigned int taper_len

    The overlapping taper length in the OFDM waveform generator from `liquid`.
- unsigned char ∗ subcarrierAlloc

    An array of `unsigned char` *whose number of elements is ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers. Each element in the array should define that subcarrier's allocation.*
- ofdmflexframegenprops_s fgprops

    The properties for the OFDM frame generator from `liquid`.
- double tx_gain_uhd

    The value of the hardware gain for the transmitter. In dB.

- double tx_gain_soft

  The software gain of the transmitter. In dB.
- double tx_freq

  The transmitter local oscillator frequency in Hertz.
- double tx_dsp_freq

  The transmitter NCO frequency in Hertz.
- double tx_rate

  The sample rate of the transmitter in samples/second.
- unsigned int **payload_sym_length**

9.18.1 Detailed Description

Contains parameters defining how to handle frame transmission.

The member parameters are accessed using the instance of the struct: ExtensibleCognitiveRadio::tx_params.

Note that for frames to be received successfully These settings must match the corresponding settings at the receiver.

9.18.2 Member Data Documentation

9.18.2.1 unsigned int ExtensibleCognitiveRadio::tx_parameter_s::cp_len

The length of the cyclic prefix in the OFDM waveform generator from `liquid`.

See the `OFDM Framing Tutorial` for details.

9.18.2.2 ofdmflexframegenprops_s ExtensibleCognitiveRadio::tx_parameter_s::fgprops

The properties for the OFDM frame generator from `liquid`.

See the `Liquid Documentation` for details.

Members of this struct can be accessed with the following functions:

- `check:`
  - ExtensibleCognitiveRadio::set_tx_crc()
  - ExtensibleCognitiveRadio::get_tx_crc().
- `fec0:`
  - ExtensibleCognitiveRadio::set_tx_fec0()
  - ExtensibleCognitiveRadio::get_tx_fec0().
- `fec1:`
  - ExtensibleCognitiveRadio::set_tx_fec1()
  - ExtensibleCognitiveRadio::get_tx_fec1().
- `mod_scheme:`
  - ExtensibleCognitiveRadio::set_tx_modulation()
  - ExtensibleCognitiveRadio::get_tx_modulation().

9.18.2.3 unsigned int ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers

The number of subcarriers in the OFDM waveform generated by `liquid`.

See the `OFDM Framing Tutorial` for details.

9.18.2.4 unsigned char∗ ExtensibleCognitiveRadio::tx_parameter_s::subcarrierAlloc

An array of `unsigned char` whose number of elements is ExtensibleCognitiveRadio::tx_parameter_s::numSubcarriers. Each element in the array should define that subcarrier's allocation.

A subcarrier's allocation defines it as a null subcarrier, a pilot subcarrier, or a data subcarrier.

See `Subcarrier Allocation` in the `liquid` documentation for details.

Also refer to the `OFDM Framing Tutorial` for more information.

9.18.2.5 unsigned int ExtensibleCognitiveRadio::tx_parameter_s::taper_len

The overlapping taper length in the OFDM waveform generator from `liquid`.

See the `OFDM Framing Tutorial` and the `Liquid Documentation Reference` for details.

9.18.2.6 double ExtensibleCognitiveRadio::tx_parameter_s::tx_dsp_freq

The transmitter NCO frequency in Hertz.

The USRP has an NCO which can be used to digitally mix the signal anywhere within the baseband bandwidth of the USRP daughterboard. This can be useful for offsetting the tone resulting from LO leakage of the ZIF transmitter used by the USRP.

9.18.2.7 double ExtensibleCognitiveRadio::tx_parameter_s::tx_freq

The transmitter local oscillator frequency in Hertz.

It can be accessed with ExtensibleCognitiveRadio::set_tx_freq() and ExtensibleCognitiveRadio::get_tx_freq().

This value is passed directly to `UHD`.

9.18.2.8 double ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_soft

The software gain of the transmitter. In dB.

In addition to the hardware gain (ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_uhd), the gain of the transmission can be adjusted in software by setting this parameter. It is converted to a linear factor and then applied to the frame samples before they are sent to `UHD`.

It can be accessed with ExtensibleCognitiveRadio::set_tx_gain_soft() and ExtensibleCognitiveRadio::get_tx_gain_soft().

Note that the values of samples sent to `UHD` must be between -1 and 1. Typically this value is set to around -12 dB based on the peak- to-average power ratio of OFDM signals. Allowing some slight clipping can improve overall signal power at the expense of added distortion.

9.18.2.9 double ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_uhd

The value of the hardware gain for the transmitter. In dB.

Sets the gain of the hardware amplifier in the transmit chain of the USRP. This value is passed directly to `UHD`.

It can be accessed with ExtensibleCognitiveRadio::set_tx_gain_uhd() and ExtensibleCognitiveRadio::get_tx_gain-_uhd().

Run

```
$ uhd_usrp_probe
```

for details about the particular gain limits of your USRP device.

9.18.2.10 double ExtensibleCognitiveRadio::tx_parameter_s::tx_rate

The sample rate of the transmitter in samples/second.

It can be accessed with ExtensibleCognitiveRadio::set_tx_rate() and ExtensibleCognitiveRadio::get_tx_rate().

This value is passed directly to `UHD`.

The documentation for this struct was generated from the following file:

- crts/include/extensible_cognitive_radio.hpp

Index