CRTS

Generated by Doxygen 1.8.6

Thu Oct 29 2015 15:07:54

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 evaluation of UDP and TCP network connections. 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.

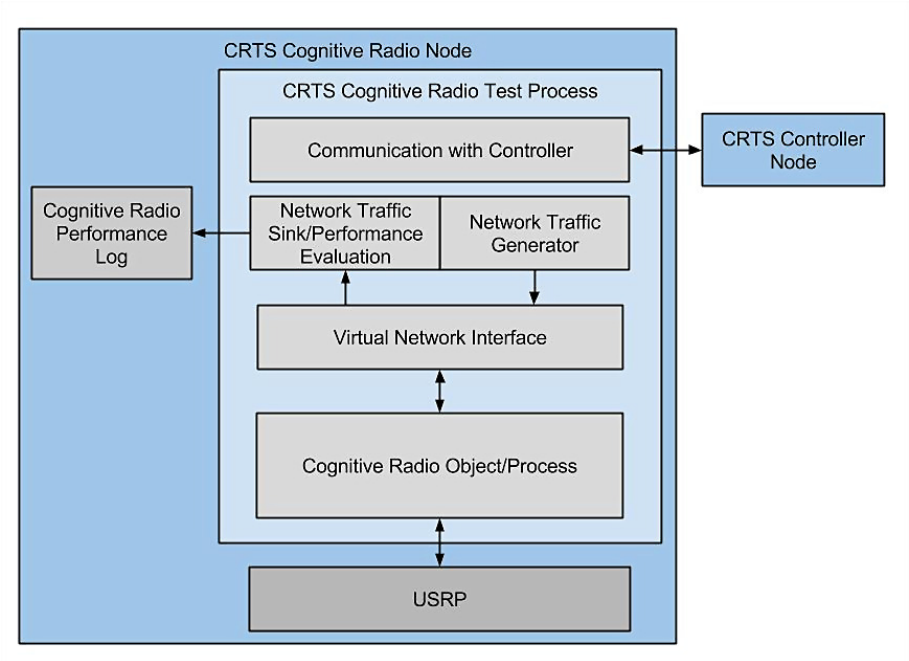

Figure 1.1: Cogntive Radio Test Process

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 is being called the Extensible Cognitive Radio (ECR). In this structure, 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.

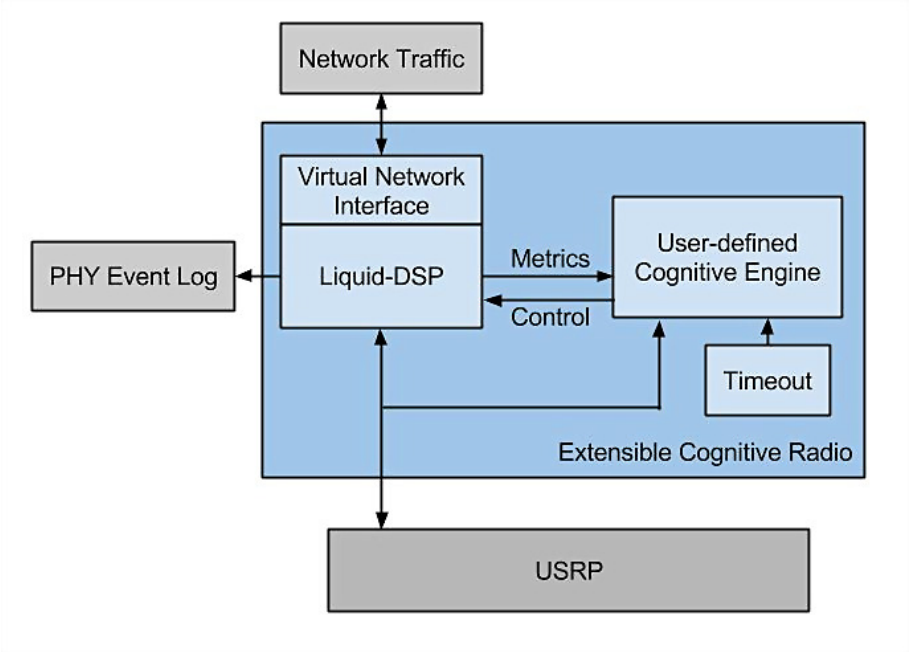

Figure 1.2: The Extensible Cognitive Radio

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.

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

1. Download the Version 1.0 tar.gz 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 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
```

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 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.

Scenarios

The `master_scenario_file.cfg` specifies which scenario(s) should be run for a single execution of the `CRTS_controller`. A single scenario can be run multiple times if desired. The syntax scenario_<#> and reps_scenario_<#> must be used.

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: CR 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.
  - If the CR node uses the ECR, it will also specify:
    - Which cognitive engine to use.
    - The initial configuration of CR.
    - What type of data should be logged.
- 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.

Examples of scenario files are provided in the `scenarios/` directory of the source tree.

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 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 generic 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.

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_CEs 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.

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 Noise 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 automatically log data from several points in the system (provided that the log file options are appropriately specified in the scenario configuration file). These logs include the parameters used by the ECR to create each phyiscal layer frame that is transmitted, the metrics and parameters of each of the physical layer frames received by the ECR, as well as the data received at the network layer (the data read from the virtual network interface by CRTS_CR). Each log entry will include a timestamp to keep events referenced to a common timeline The logs are written as raw binary files in the /logs/bin directory, but will automatically be converted to either Octave/Matlab or Python scripts after the scenario has finished and placed in the logs/Octave or logs/Python directories respectively. This again assumes that the appropriate options were set in the scenario configuration file. These scripts provide the user with an easy way to analyze the results of the experiment. There are some basic Octave/Matlab scripts provided to plot the contents of the logs as a function of time.

Chapter 2

Example Scenarios

1. Two_Channel_DSA

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

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. Two_Node_FDD_Network

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. Three_Node_HD_Network

This scenario defines a 3 node CR network all operating on a single frequency, making it half duplex. This was intended as a demonstration of the networking interfaces of the ECR. Note that there is currently no mechanism to regulate channel access e.g. CSMA.

5. FEC_Adaptation

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.

6. Interferer_Test

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

7. Mod_Adaptation

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.

8. Subcarrier_Alloc_Test

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

Example 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

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

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

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

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

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

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

Chapter 4

Tutorial 1: Running CRTS

Begin by opening three ssh sessions on CORNET using the following command:

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

Choose three nodes that are adjacent to one another. Make sure that the nodes have access to their USRPs by running:

```
$ uhd_find_devices
```

If not, try other nodes. Navigate to the crts directory on each node. Open up the master_scenario_file.cfg file. This file tells the experiment controller how many scenarios to run and their names. Make the values match:

```
NumberofScenarios = 1;
scenario\_1 = "Two_Node_FDD_Network.cfg";
reps\_scenario\_1 = 1;
```

Now open the scenario configuration file 2_Node_FDD_Network.cfg This file defines the most basic scenario involving two CR nodes. The default cognitive engine is an empty template (it does not make any decisions). If you haven't already, take a look at the definition of the cognitive engine being used by this scenario just to see the general structure, more details are provided in the Cognitive Engine section of the documentation.

Now lets actually run CRTS. First launch the controller. CRTS can be run in a 'manual' or 'automatic' mode. The default is to run in automatic mode; manual mode is specified by a -m flag on the controller command. On the node you want to act as the controller, run:

```
$ ./CRTS_controller -m
```

Now you can run the CRTS CR process on the other two nodes.

```
$ ./CRTS_CR -a <controller internal ip>
```

The controller IP needs to be specified so the program knows where to connect.

The internal ip will be 192.168.1.<external port number -6990>. Observe that the two nodes have received their operating parameters and will begin to exchange frames over the air metrics for the received frames should be printed out to both terminals.

Go to /logs/octave and you should see several auto-generated .m files. To view a plot of the network throughput vs. time for each node run:

```
$ octave
> Two_Node_FDD_Network_N1_NET_RX
> Plot_CR_NET_RX
```

Chapter 5

Class Index

5.1 Class List

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

Chapter 6

Class Documentation

6.1 Cognitive_Engine Class Reference

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

`#include <CE.hpp>`

Public Member Functions

- virtual void execute (void *_args)

  Executes the custom cognitive engine as defined by the user.

6.1.1 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 ExtensibleCognitiveRadio::Event, then the custom-defined execute function (Cognitive_Engine::execute()) will be called.

6.1.2 Member Function Documentation

6.1.2.1 void Cognitive_Engine::execute (void * *_args*) [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 files:

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

6.2 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 tx_parameter_s

  Contains parameters defining how to handle frame transmission.

Public Types

- enum Event { TIMEOUT = 0, PHY }

  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)
- void **start_ce** ()
- void **stop_ce** ()
- void set_ce_timeout_ms (float new_timeout_ms)

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

  Get the current value of ExtensibleCognitiveRadio::ce_timeout_ms.
- void set_ip (char *ip)

  Used to set the IP of the ECR's virtual network interface.
- void set_tx_freq (float _tx_freq)

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

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

  Set the value of ExtensibleCognitiveRadio::tx_parameter_s::tx_gain_soft.
- void set_tx_gain_uhd (float _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.
- void increase_tx_mod_order ()

  Increases the modulation order if possible.
- void decrease_tx_mod_order ()

  Decreases the modulation order if possible.
- float get_tx_freq ()

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

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

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

  Return the value of ExtensibleCognitiveRadio::tx_parameter_s::taper_len.
- void **get_tx_control_info** (unsigned char *_control_info)
- void **start_tx** ()
- void **stop_tx** ()
- void **reset_tx** ()
- void transmit_frame (unsigned char *_header, unsigned char *_payload, unsigned int _payload_len)

  Transmit a custom frame.
- void set_rx_freq (float _rx_freq)

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

Set the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_rate.

- void set_rx_gain_uhd (float _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.
- float get_rx_freq ()

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

  Return the value of ExtensibleCognitiveRadio::rx_parameter_s::rx_rate.
- float 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 **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.
- int **print_metrics_flag**
- int **log_phy_rx_flag**
- int **log_phy_tx_flag**
- char **phy_rx_log_file** [100]
- char **phy_tx_log_file** [100]
- 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 Attributes

- Cognitive_Engine * **CE**
- float 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**
- pthread_t **CE_process**
- pthread_mutex_t **CE_mutex**
- pthread_cond_t **CE_cond**
- pthread_cond_t **CE_execute_sig**
- bool **ce_thread_running**
- bool **ce_running**
- int **tunfd**
- char **tun_name** [IFNAMSIZ]
- char **systemCMD** [200]
- struct rx_parameter_s **rx_params**
- ofdmflexframesync **fs**
- unsigned int **frame_num**
- pthread_t **rx_process**
- pthread_mutex_t **rx_mutex**
- pthread_cond_t **rx_cond**
- bool **rx_running**
- bool **rx_thread_running**
- tx_parameter_s **tx_params**
- ofdmflexframegen **fg**
- unsigned int **fgbuffer_len**
- std::complex< float > * **fgbuffer**
- unsigned char **tx_header** [8]
- unsigned int **frame_counter**
- pthread_t **tx_process**
- pthread_mutex_t **tx_mutex**
- pthread_cond_t **tx_cond**
- bool **tx_thread_running**
- bool **tx_running**

Friends

- void * **ECR_ce_worker** (void *)
- void * **ECR_rx_worker** (void *)
- int **rxCallback** (unsigned char *, int, unsigned char *, unsigned int, int, framesyncstats_s, void *)
- void * **ECR_tx_worker** (void *)

6.2.1 Member Enumeration Documentation

6.2.1.1 enum ExtensibleCognitiveRadio::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.

6.2.1.2 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.

6.2.2 Member Function Documentation

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 void ExtensibleCognitiveRadio::transmit_frame (unsigned char * *_header,* unsigned char * *_payload,* unsigned int *_payload_len*)

Transmit a custom frame.

The cognitive engine (CE) can initiate transmission of a custom frame by calling this function. `_header` must be a pointer to an array of exactly 8 elements of type `unsigned int`. The first byte of `_header` **must** be set to ExtensibleCognitiveRadio::CONTROL. For Example:

```
ExtensibleCognitiveRadio ECR;
unsigned char myHeader[8];
unsigned char myPayload[20];
myHeader[0] = ExtensibleCognitiveRadio::CONTROL.
ECR.transmit_frame(myHeader, myPayload, 20);
```

`_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`.

6.2.3 Member Data Documentation

6.2.3.1 float 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/ECR.hpp
- crts/src/ECR.cpp

6.3 Interferer Class Reference

Public Attributes

- int **interference_type**
- float **tx_gain_soft**
- float **tx_gain**
- float **tx_freq**
- float **tx_rate**
- float **period**
- float **duty_cycle**
- int **tx_freq_hop_type**
- float **tx_freq_hop_min**
- float **tx_freq_hop_max**
- float **tx_freq_hop_dwell_time**
- float **tx_freq_hop_increment**
- uhd::usrp::multi_usrp::sptr **usrp_tx**
- uhd::tx_metadata_t **metadata_tx**

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

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

6.4 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 <ECR.hpp>`

Public Attributes

- ExtensibleCognitiveRadio::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.

6.4.1 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

6.4.2 Member Data Documentation

6.4.2.1 ExtensibleCognitiveRadio::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.

6.4.2.2 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.

6.4.2.3 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.

6.4.2.4 unsigned int ExtensibleCognitiveRadio::metric_s::payload_len

The number of elements of the `payload` array.

Equal to the byte length of the `payload`.

6.4.2.5 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.

6.4.2.6 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`.

6.4.2.7 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/ECR.hpp

6.5 node_parameters Struct Reference

Public Attributes

- int **type**
- int **cr_type**
- char **python_file** [100]
- char **arguments** [20][50]
- int **num_arguments**
- char **CORNET_IP** [20]
- char **CRTS_IP** [20]
- char **TARGET_IP** [20]
- char **CE** [100]
- int **print_metrics**
- int **log_phy_rx**
- int **log_phy_tx**
- int **log_net_rx**
- char **phy_rx_log_file** [100]
- char **phy_tx_log_file** [100]
- char **net_rx_log_file** [100]

- float **ce_timeout_ms**
- int **generate_octave_logs**
- int **generate_python_logs**
- float **rx_freq**
- float **rx_rate**
- float **rx_gain**
- float **tx_freq**
- float **tx_rate**
- float **tx_gain**
- int **duplex**
- 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]
- float **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**
- float **tx_delay_us**
- 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**
- float **period**
- float **duty_cycle**
- int **tx_freq_hop_type**
- float **tx_freq_hop_min**
- float **tx_freq_hop_max**
- float **tx_freq_hop_dwell_time**
- float **tx_freq_hop_increment**

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

- crts/include/node_parameters.hpp

6.6 ExtensibleCognitiveRadio::rx_parameter_s Struct Reference

Contains parameters defining how to handle frame reception.

`#include <ECR.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.*
- float rx_gain_uhd

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

  The receiver local oscillator frequency in Hertz.
- float rx_dsp_freq

  The transmitter NCO frequency in Hertz.
- float rx_rate

  The sample rate of the receiver in samples/second.

6.6.1 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.

6.6.2 Member Data Documentation

6.6.2.1 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.

6.6.2.2 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.

6.6.2.3 float 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.

6.6.2.4 float 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.

6.6.2.5 float 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.

6.6.2.6 float 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.

6.6.2.7 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.

6.6.2.8 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/ECR.hpp

6.7 scenario_parameters Struct Reference

Public Attributes

- int **num_nodes**
- time_t **start_time_s**
- time_t **runTime**
- unsigned int **totalNumReps**

- unsigned int **repNumber**

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

- crts/include/read_configs.hpp

6.8 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

6.9 ExtensibleCognitiveRadio::tx_parameter_s Struct Reference

Contains parameters defining how to handle frame transmission.

`#include <ECR.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`.
- float tx_gain_uhd

  The value of the hardware gain for the transmitter. In dB.
- float tx_gain_soft

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

  The transmitter local oscillator frequency in Hertz.
- float tx_dsp_freq

  The transmitter NCO frequency in Hertz.
- float tx_rate

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

6.9.1 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.

6.9.2 Member Data Documentation

6.9.2.1 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.

6.9.2.2 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().

6.9.2.3 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.

6.9.2.4 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.

6.9.2.5 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.

6.9.2.6 float 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.

6.9.2.7 float 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.

6.9.2.8 float 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.

6.9.2.9 float 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.

6.9.2.10 float 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/ECR.hpp

Index