Previous topic

gnuradio.trellis

Next topic

gnuradio.video_sdl

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

gnuradio.uhd

Provides source and sink blocks to interface with the UHD library. Used to send and receive data between the Ettus Research, LLC product line.

gnuradio.uhd.**amsg_source**(**args*, ***kwargs*)

amsg_source_sptr.**msg_to_async_metadata_t**(*amsg_source_sptr self*, *message_sptr msg*) → async_metadata_t

Convert a raw asynchronous message to an asynchronous metatdata object.

gnuradio.uhd.**usrp_sink**(**args*, ***kwargs*)

usrp_sink_sptr.**active_thread_priority**(*usrp_sink_sptr self*) → int

usrp_sink_sptr.**clear_command_time**(*usrp_sink_sptr self*, *size_t mboard=0*)

Clear the command time so future commands are sent ASAP.

usrp_sink_sptr.**declare_sample_delay**(*usrp_sink_sptr self*, *int which*, *int delay*)

declare_sample_delay(usrp_sink_sptr self, unsigned int delay)

usrp_sink_sptr.**get_antenna**(*usrp_sink_sptr self*, *size_t chan=0*) → std::string

Get the antenna in use.

usrp_sink_sptr.**get_antennas**(*usrp_sink_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible antennas on a given channel.

usrp_sink_sptr.**get_bandwidth**(*usrp_sink_sptr self*, *size_t chan=0*) → double

Get the bandpass filter setting on the RF frontend.

usrp_sink_sptr.**get_bandwidth_range**(*usrp_sink_sptr self*, *size_t chan=0*) → meta_range_t

Get the bandpass filter range of the RF frontend.

usrp_sink_sptr.**get_center_freq**(*usrp_sink_sptr self*, *size_t chan=0*) → double

Get the center frequency.

usrp_sink_sptr.**get_clock_rate**(*usrp_sink_sptr self*, *size_t mboard=0*) → double

Get the master clock rate.

usrp_sink_sptr.**get_clock_source**(*usrp_sink_sptr self*, *size_t const mboard*) → std::string

Get the currently set clock source.

usrp_sink_sptr.**get_clock_sources**(*usrp_sink_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible clock sources.

usrp_sink_sptr.**get_dboard_iface**(*usrp_sink_sptr self*, *size_t chan=0*) → dboard_iface_sptr

Get access to the underlying uhd dboard iface object.

usrp_sink_sptr.**get_dboard_sensor**(*usrp_sink_sptr self*, *std::string const & name*, *size_t chan=0*) → sensor_value_t

DEPRECATED use get_sensor.

usrp_sink_sptr.**get_dboard_sensor_names**(*usrp_sink_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

DEPRECATED use get_sensor_names.

usrp_sink_sptr.**get_device**(*usrp_sink_sptr self*) → ::uhd::usrp::multi_usrp::sptr

Get access to the underlying uhd device object.

NOTE: This function is only available in C++.

usrp_sink_sptr.**get_freq_range**(*usrp_sink_sptr self*, *size_t chan=0*) → meta_range_t

Get the tunable frequency range.

usrp_sink_sptr.**get_gain**(*usrp_sink_sptr self*, *size_t chan=0*) → double

get_gain(usrp_sink_sptr self, std::string const & name, size_t chan=0) -> double

Get the actual dboard gain setting.

usrp_sink_sptr.**get_gain_names**(*usrp_sink_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get the actual dboard gain setting of named stage.

usrp_sink_sptr.**get_gain_range**(*usrp_sink_sptr self*, *size_t chan=0*) → meta_range_t

get_gain_range(usrp_sink_sptr self, std::string const & name, size_t chan=0) -> meta_range_t

Get the settable gain range.

usrp_sink_sptr.**get_gpio_attr**(*usrp_sink_sptr self*, *std::string const & bank*, *std::string const & attr*, *size_t const mboard=0*) → boost::uint32_t

Get a GPIO attribute on a particular GPIO bank. Possible attribute names:

usrp_sink_sptr.**get_gpio_banks**(*usrp_sink_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Enumerate GPIO banks on the current device.

usrp_sink_sptr.**get_mboard_sensor**(*usrp_sink_sptr self*, *std::string const & name*, *size_t mboard=0*) → sensor_value_t

Get a motherboard sensor value.

usrp_sink_sptr.**get_mboard_sensor_names**(*usrp_sink_sptr self*, *size_t mboard=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible motherboard sensor names.

usrp_sink_sptr.**get_normalized_gain**(*usrp_sink_sptr self*, *size_t chan=0*) → double

Returns the normalized gain.

The normalized gain is always in [0, 1], regardless of the device. See also set_normalized_gain().

Note that it is not possible to specify a gain name for this function, the result is over the entire gain chain.

usrp_sink_sptr.**get_num_mboards**(*usrp_sink_sptr self*) → size_t

Return the number of motherboards in this configuration.

usrp_sink_sptr.**get_samp_rate**(*usrp_sink_sptr self*) → double

Get the sample rate for this connection to the USRP. This is the actual sample rate and may differ from the rate set.

usrp_sink_sptr.**get_samp_rates**(*usrp_sink_sptr self*) → meta_range_t

Get the possible sample rates for this connection.

usrp_sink_sptr.**get_sensor**(*usrp_sink_sptr self*, *std::string const & name*, *size_t*

chan=0) → sensor_value_t

Get an RF frontend sensor value.

usrp_sink_sptr.**get_sensor_names**(*usrp_sink_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible RF frontend sensor names.

usrp_sink_sptr.**get_subdev_spec**(*usrp_sink_sptr self*, *size_t mboard=0*) → std::string

Get the frontend specification.

usrp_sink_sptr.**get_time_last_pps**(*usrp_sink_sptr self*, *size_t mboard=0*) → time_spec_t

Get the time when the last pps pulse occurred.

usrp_sink_sptr.**get_time_now**(*usrp_sink_sptr self*, *size_t mboard=0*) → time_spec_t

Get the current time registers.

usrp_sink_sptr.**get_time_source**(*usrp_sink_sptr self*, *size_t const mboard*) → std::string

Get the currently set time source.

usrp_sink_sptr.**get_time_sources**(*usrp_sink_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible time sources.

usrp_sink_sptr.**get_usrp_info**(*usrp_sink_sptr self*, *size_t chan=0*) → string_string_dict_t

Returns identifying information about this USRP's configuration. Returns motherboard ID, name, and serial. Returns daughterboard TX ID, subdev name and spec, serial, and antenna.

usrp_sink_sptr.**message_subscribers**(*usrp_sink_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

usrp_sink_sptr.**min_noutput_items**(*usrp_sink_sptr self*) → int

usrp_sink_sptr.**pc_input_buffers_full_avg**(*usrp_sink_sptr self*, *int which*) → float

pc_input_buffers_full_avg(usrp_sink_sptr self) -> pmt_vector_float

usrp_sink_sptr.**pc_noutput_items_avg**(*usrp_sink_sptr self*) → float

usrp_sink_sptr.**pc_nproduced_avg**(*usrp_sink_sptr self*) → float

usrp_sink_sptr.**pc_output_buffers_full_avg**(*usrp_sink_sptr self*, *int which*) → float

pc_output_buffers_full_avg(usrp_sink_sptr self) -> pmt_vector_float

usrp_sink_sptr.**pc_throughput_avg**(*usrp_sink_sptr self*) → float

usrp_sink_sptr.**pc_work_time_avg**(*usrp_sink_sptr self*) → float

usrp_sink_sptr.**pc_work_time_total**(*usrp_sink_sptr self*) → float

usrp_sink_sptr.**sample_delay**(*usrp_sink_sptr self*, *int which*) → unsigned int

usrp_sink_sptr.**set_antenna**(*usrp_sink_sptr self*, *std::string const & ant*, *size_t chan=0*)

Set the antenna to use for a given channel.

usrp_sink_sptr.**set_bandwidth**(*usrp_sink_sptr self*, *double bandwidth*, *size_t*

chan=0)

Set the bandpass filter on the RF frontend.

usrp_sink_sptr.**set_center_freq**(*usrp_sink_sptr self*, *tune_request_t tune_request*, *size_t chan=0*) → tune_result_t

set_center_freq(usrp_sink_sptr self, double freq, size_t chan=0) -> tune_result_t

Tune the selected channel to the desired center frequency.

usrp_sink_sptr.**set_clock_config**(*usrp_sink_sptr self*, *clock_config_t clock_config*, *size_t mboard=0*)

Set the clock configuration.

DEPRECATED for set_time/clock_source.

usrp_sink_sptr.**set_clock_rate**(*usrp_sink_sptr self*, *double rate*, *size_t mboard=0*)

Set the master clock rate.

usrp_sink_sptr.**set_clock_source**(*usrp_sink_sptr self*, *std::string const & source*, *size_t const mboard=0*)

Set the clock source for the usrp device.

This sets the source for a 10 MHz reference clock. Typical options for source: internal, external, MIMO.

usrp_sink_sptr.**set_command_time**(*usrp_sink_sptr self*, *time_spec_t time_spec*, *size_t mboard=0*)

Set the time at which the control commands will take effect.

A timed command will back-pressure all subsequent timed commands, assuming that the subsequent commands occur within the time-window. If the time spec is late, the command will be activated upon arrival.

usrp_sink_sptr.**set_dc_offset**(*usrp_sink_sptr self*, *std::complex< double > const & offset*, *size_t chan=0*)

Set a constant DC offset value. The value is complex to control both I and Q.

usrp_sink_sptr.**set_gain**(*usrp_sink_sptr self*, *double gain*, *size_t chan=0*)

set_gain(usrp_sink_sptr self, double gain, std::string const & name, size_t chan=0)

Set the gain for the selected channel.

usrp_sink_sptr.**set_gpio_attr**(*usrp_sink_sptr self*, *std::string const & bank*, *std::string const & attr*, *boost::uint32_t const value*, *boost::uint32_t const mask=0xffffffff*, *size_t const mboard=0*)

Set a GPIO attribute on a particular GPIO bank. Possible attribute names:

usrp_sink_sptr.**set_iq_balance**(*usrp_sink_sptr self*, *std::complex< double > const & correction*, *size_t chan=0*)

Set the RX frontend IQ imbalance correction. Use this to adjust the magnitude and phase of I and Q.

usrp_sink_sptr.**set_min_noutput_items**(*usrp_sink_sptr self*, *int m*)

usrp_sink_sptr.**set_normalized_gain**(*usrp_sink_sptr self*, *double norm_gain*, *size_t chan=0*)

Set the normalized gain.

The normalized gain is always in [0, 1], regardless of the device. 0 corresponds to minimum gain (usually 0 dB, but make sure to read the device notes in the UHD manual) and 1 corresponds to maximum gain. This will work for any UHD device. Use get_gain() to see which dB value the normalized gain value corresponds to.

Note that it is not possible to specify a gain name for this function.

usrp_sink_sptr.**set_samp_rate**(*usrp_sink_sptr self*, *double rate*)

Set the sample rate for this connection to the USRP.

usrp_sink_sptr.**set_start_time**(*usrp_sink_sptr self*, *time_spec_t time*)

Set the start time for outgoing samples. To control when samples are transmitted, set this value before starting the flow graph. The value is cleared after each run. When not specified, the start time will be:

usrp_sink_sptr.**set_stream_args**(*usrp_sink_sptr self*, *stream_args_t stream_args*)

Update the stream args for this device.

This update will only take effect after a restart of the streaming, or before streaming and after construction. This will also delete the current streamer. Note you cannot change the I/O signature of this block using this function, or it will throw.

It is possible to leave the 'channels' fields of unset. In this case, the previous channels field is used.

usrp_sink_sptr.**set_subdev_spec**(*usrp_sink_sptr self*, *std::string const & spec*, *size_t mboard=0*)

Set the frontend specification.

usrp_sink_sptr.**set_thread_priority**(*usrp_sink_sptr self*, *int priority*) → int

usrp_sink_sptr.**set_time_next_pps**(*usrp_sink_sptr self*, *time_spec_t time_spec*)

Set the time registers at the next pps.

usrp_sink_sptr.**set_time_now**(*usrp_sink_sptr self*, *time_spec_t time_spec*, *size_t mboard=0*)

Sets the time registers immediately.

usrp_sink_sptr.**set_time_source**(*usrp_sink_sptr self*, *std::string const & source*, *size_t const mboard=0*)

Set the time source for the USRP device.

This sets the method of time synchronization, typically a pulse per second or an encoded time. Typical options for source: external, MIMO.

usrp_sink_sptr.**set_time_unknown_pps**(*usrp_sink_sptr self*, *time_spec_t time_spec*)

Sync the time registers with an unknown pps edge.

usrp_sink_sptr.**set_user_register**(*usrp_sink_sptr self*, *uint8_t const addr*, *uint32_t const data*, *size_t mboard=0*)

Perform write on the user configuration register bus. These only exist if the user has implemented custom setting registers in the device FPGA.

usrp_sink_sptr.**thread_priority**(*usrp_sink_sptr self*) → int

gnuradio.uhd.**usrp_source**(**args*, ***kwargs*)

usrp_source_sptr.**active_thread_priority**(*usrp_source_sptr self*) → int

usrp_source_sptr.**clear_command_time**(*usrp_source_sptr self*, *size_t mboard=0*)

Clear the command time so future commands are sent ASAP.

usrp_source_sptr.**declare_sample_delay**(*usrp_source_sptr self*, *int which*, *int delay*)

declare_sample_delay(usrp_source_sptr self, unsigned int delay)

usrp_source_sptr.**finite_acquisition**(*usrp_source_sptr self*, *size_t const nsamps*) → pmt_vector_cfloat

Convenience function for finite data acquisition. This is not to be used with the scheduler; rather, one can request samples from the USRP in python. //TODO assumes fc32

usrp_source_sptr.**finite_acquisition_v**(*usrp_source_sptr self*, *size_t const nsamps*) → gr_vector_vector_complexf

Convenience function for finite data acquisition. This is the multi-channel version of finite_acquisition; This is not to be used with the scheduler; rather, one can request samples from the USRP in python. //TODO assumes fc32

usrp_source_sptr.**get_antenna**(*usrp_source_sptr self*, *size_t chan=0*) → std::string

Get the antenna in use.

usrp_source_sptr.**get_antennas**(*usrp_source_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible antennas on a given channel.

usrp_source_sptr.**get_bandwidth**(*usrp_source_sptr self*, *size_t chan=0*) → double

Get the bandpass filter setting on the RF frontend.

usrp_source_sptr.**get_bandwidth_range**(*usrp_source_sptr self*, *size_t chan=0*) → meta_range_t

Get the bandpass filter range of the RF frontend.

usrp_source_sptr.**get_center_freq**(*usrp_source_sptr self*, *size_t chan=0*) → double

Get the center frequency.

usrp_source_sptr.**get_clock_rate**(*usrp_source_sptr self*, *size_t mboard=0*) → double

Get the master clock rate.

usrp_source_sptr.**get_clock_source**(*usrp_source_sptr self*, *size_t const mboard*) → std::string

Get the currently set clock source.

usrp_source_sptr.**get_clock_sources**(*usrp_source_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible clock sources.

usrp_source_sptr.**get_dboard_iface**(*usrp_source_sptr self*, *size_t chan=0*) → dboard_iface_sptr

Get access to the underlying uhd dboard iface object.

usrp_source_sptr.**get_dboard_sensor**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → sensor_value_t

DEPRECATED use get_sensor.

usrp_source_sptr.**get_dboard_sensor_names**(*usrp_source_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

DEPRECATED use get_sensor_names.

usrp_source_sptr.**get_device**(*usrp_source_sptr self*) → ::uhd::usrp::multi_usrp::sptr

Get access to the underlying uhd device object.

NOTE: This function is only available in C++.

usrp_source_sptr.**get_freq_range**(*usrp_source_sptr self*, *size_t chan=0*) →

meta_range_t

Get the tunable frequency range.

usrp_source_sptr.**get_gain**(*usrp_source_sptr self*, *size_t chan=0*) → double

get_gain(usrp_source_sptr self, std::string const & name, size_t chan=0) -> double

Get the actual dboard gain setting.

usrp_source_sptr.**get_gain_names**(*usrp_source_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get the actual dboard gain setting of named stage.

usrp_source_sptr.**get_gain_range**(*usrp_source_sptr self*, *size_t chan=0*) → meta_range_t

get_gain_range(usrp_source_sptr self, std::string const & name, size_t chan=0) -> meta_range_t

Get the settable gain range.

usrp_source_sptr.**get_gpio_attr**(*usrp_source_sptr self*, *std::string const & bank*, *std::string const & attr*, *size_t const mboard=0*) → boost::uint32_t

Get a GPIO attribute on a particular GPIO bank. Possible attribute names:

usrp_source_sptr.**get_gpio_banks**(*usrp_source_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Enumerate GPIO banks on the current device.

usrp_source_sptr.**get_lo_export_enabled**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → bool

Returns true if the currently selected LO is being exported.

usrp_source_sptr.**get_lo_freq**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → double

Get the current RX LO frequency (Advanced).

usrp_source_sptr.**get_lo_freq_range**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → meta_range_t

Get the LO frequency range of the RX LO.

usrp_source_sptr.**get_lo_names**(*usrp_source_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible LO stage names

usrp_source_sptr.**get_lo_source**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → std::string const

Get the currently set LO source.

usrp_source_sptr.**get_lo_sources**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible LO sources.

usrp_source_sptr.**get_mboard_sensor**(*usrp_source_sptr self*, *std::string const & name*, *size_t mboard=0*) → sensor_value_t

Get a motherboard sensor value.

usrp_source_sptr.**get_mboard_sensor_names**(*usrp_source_sptr self*, *size_t mboard=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible motherboard sensor names.

usrp_source_sptr.**get_normalized_gain**(*usrp_source_sptr self*, *size_t chan=0*) → double

Returns the normalized gain.

The normalized gain is always in [0, 1], regardless of the device. See also set_normalized_gain().

Note that it is not possible to specify a gain name for this function, the result is over the entire gain chain.

usrp_source_sptr.**get_num_mboards**(*usrp_source_sptr self*) → size_t

Return the number of motherboards in this configuration.

usrp_source_sptr.**get_samp_rate**(*usrp_source_sptr self*) → double

Get the sample rate for this connection to the USRP. This is the actual sample rate and may differ from the rate set.

usrp_source_sptr.**get_samp_rates**(*usrp_source_sptr self*) → meta_range_t

Get the possible sample rates for this connection.

usrp_source_sptr.**get_sensor**(*usrp_source_sptr self*, *std::string const & name*, *size_t chan=0*) → sensor_value_t

Get an RF frontend sensor value.

usrp_source_sptr.**get_sensor_names**(*usrp_source_sptr self*, *size_t chan=0*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible RF frontend sensor names.

usrp_source_sptr.**get_subdev_spec**(*usrp_source_sptr self*, *size_t mboard=0*) → std::string

Get the frontend specification.

usrp_source_sptr.**get_time_last_pps**(*usrp_source_sptr self*, *size_t mboard=0*) → time_spec_t

Get the time when the last pps pulse occurred.

usrp_source_sptr.**get_time_now**(*usrp_source_sptr self*, *size_t mboard=0*) → time_spec_t

Get the current time registers.

usrp_source_sptr.**get_time_source**(*usrp_source_sptr self*, *size_t const mboard*) → std::string

Get the currently set time source.

usrp_source_sptr.**get_time_sources**(*usrp_source_sptr self*, *size_t const mboard*) → std::vector< std::string,std::allocator< std::string > >

Get a list of possible time sources.

usrp_source_sptr.**get_usrp_info**(*usrp_source_sptr self*, *size_t chan=0*) → string_string_dict_t

Returns identifying information about this USRP's configuration. Returns motherboard ID, name, and serial. Returns daughterboard RX ID, subdev name and spec, serial, and antenna.

usrp_source_sptr.**issue_stream_cmd**(*usrp_source_sptr self*, *stream_cmd_t cmd*)

Issue a stream command to all channels in this source block.

This method is intended to override the default "always on" behavior. After starting the flow graph, the user should call stop() on this block, then issue any desired arbitrary stream_cmd_t structs to the device. The USRP will be able to enqueue several stream commands in the FPGA.

usrp_source_sptr.**message_subscribers**(*usrp_source_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

usrp_source_sptr.**min_noutput_items**(*usrp_source_sptr self*) → int

usrp_source_sptr.**pc_input_buffers_full_avg**(*usrp_source_sptr self*, *int which*) → float

pc_input_buffers_full_avg(usrp_source_sptr self) -> pmt_vector_float

usrp_source_sptr.**pc_noutput_items_avg**(*usrp_source_sptr self*) → float

usrp_source_sptr.**pc_nproduced_avg**(*usrp_source_sptr self*) → float

usrp_source_sptr.**pc_output_buffers_full_avg**(*usrp_source_sptr self*, *int which*) → float

pc_output_buffers_full_avg(usrp_source_sptr self) -> pmt_vector_float

usrp_source_sptr.**pc_throughput_avg**(*usrp_source_sptr self*) → float

usrp_source_sptr.**pc_work_time_avg**(*usrp_source_sptr self*) → float

usrp_source_sptr.**pc_work_time_total**(*usrp_source_sptr self*) → float

usrp_source_sptr.**sample_delay**(*usrp_source_sptr self*, *int which*) → unsigned int

usrp_source_sptr.**set_antenna**(*usrp_source_sptr self*, *std::string const & ant*, *size_t chan=0*)

Set the antenna to use for a given channel.

usrp_source_sptr.**set_auto_dc_offset**(*usrp_source_sptr self*, *bool const enb*, *size_t chan=0*)

Enable/disable the automatic DC offset correction. The automatic correction subtracts out the long-run average.

When disabled, the averaging option operation is halted. Once halted, the average value will be held constant until the user re-enables the automatic correction or overrides the value by manually setting the offset.

usrp_source_sptr.**set_auto_iq_balance**(*usrp_source_sptr self*, *bool const enb*, *size_t chan=0*)

Enable/Disable the RX frontend IQ imbalance correction.

usrp_source_sptr.**set_bandwidth**(*usrp_source_sptr self*, *double bandwidth*, *size_t chan=0*)

Set the bandpass filter on the RF frontend.

usrp_source_sptr.**set_center_freq**(*usrp_source_sptr self*, *tune_request_t tune_request*, *size_t chan=0*) → tune_result_t

set_center_freq(usrp_source_sptr self, double freq, size_t chan=0) -> tune_result_t

Tune the selected channel to the desired center frequency.

usrp_source_sptr.**set_clock_config**(*usrp_source_sptr self*, *clock_config_t clock_config*, *size_t mboard=0*)

Set the clock configuration.

DEPRECATED for set_time/clock_source.

usrp_source_sptr.**set_clock_rate**(*usrp_source_sptr self*, *double rate*, *size_t mboard=0*)

Set the master clock rate.

usrp_source_sptr.**set_clock_source**(*usrp_source_sptr self*, *std::string const & source*, *size_t const mboard=0*)

Set the clock source for the usrp device.

This sets the source for a 10 MHz reference clock. Typical options for source:

internal, external, MIMO.

usrp_source_sptr.**set_command_time**(*usrp_source_sptr self*, *time_spec_t time_spec*, *size_t mboard=0*)

Set the time at which the control commands will take effect.

A timed command will back-pressure all subsequent timed commands, assuming that the subsequent commands occur within the time-window. If the time spec is late, the command will be activated upon arrival.

usrp_source_sptr.**set_dc_offset**(*usrp_source_sptr self*, *std::complex< double > const & offset*, *size_t chan=0*)

Set a constant DC offset value. The value is complex to control both I and Q. Only set this when automatic correction is disabled.

usrp_source_sptr.**set_gain**(*usrp_source_sptr self*, *double gain*, *size_t chan=0*)

set_gain(usrp_source_sptr self, double gain, std::string const & name, size_t chan=0)

Set the gain for the selected channel.

usrp_source_sptr.**set_gpio_attr**(*usrp_source_sptr self*, *std::string const & bank*, *std::string const & attr*, *boost::uint32_t const value*, *boost::uint32_t const mask=0xffffffff*, *size_t const mboard=0*)

Set a GPIO attribute on a particular GPIO bank. Possible attribute names:

usrp_source_sptr.**set_iq_balance**(*usrp_source_sptr self*, *std::complex< double > const & correction*, *size_t chan=0*)

Set the RX frontend IQ imbalance correction. Use this to adjust the magnitude and phase of I and Q.

usrp_source_sptr.**set_lo_export_enabled**(*usrp_source_sptr self*, *bool enabled*, *std::string const & name*, *size_t chan=0*)

Set whether the LO used by the usrp device is exported For usrps that support exportable LOs, this function configures if the LO used by chan is exported or not.

usrp_source_sptr.**set_lo_freq**(*usrp_source_sptr self*, *double freq*, *std::string const & name*, *size_t chan=0*) → double

Set the RX LO frequency (Advanced).

usrp_source_sptr.**set_lo_source**(*usrp_source_sptr self*, *std::string const & src*, *std::string const & name*, *size_t chan=0*)

Set the LO source for the usrp device. For usrps that support selectable LOs, this function allows switching between them. Typical options for source: internal, external.

usrp_source_sptr.**set_min_noutput_items**(*usrp_source_sptr self*, *int m*)

usrp_source_sptr.**set_normalized_gain**(*usrp_source_sptr self*, *double norm_gain*, *size_t chan=0*)

Set the normalized gain.

The normalized gain is always in [0, 1], regardless of the device. 0 corresponds to minimum gain (usually 0 dB, but make sure to read the device notes in the UHD manual) and 1 corresponds to maximum gain. This will work for any UHD device. Use get_gain() to see which dB value the normalized gain value corresponds to.

Note that it is not possible to specify a gain name for this function.

usrp_source_sptr.**set_recv_timeout**(*usrp_source_sptr self*, *double const timeout*, *bool const one_packet=True*)

Configure the timeout value on the UHD recv() call

This is an advanced use parameter. Changing the timeout value affects the latency

of this block; a high timeout value can be more optimal for high-throughput applications (e.g., 1 second) and setting it to zero will have the best latency. Changing the timeout value may also be useful for custom FPGA modifications, where traffic might not be continuously streaming. For specialized high-performance use cases, twiddling these knobs may improve performance, but changes are not generally applicable.

Note that UHD's recv() call may block until the timeout is over, which means this block might also block for up to the timeout value.

usrp_source_sptr.**set_samp_rate**(*usrp_source_sptr self*, *double rate*)

Set the sample rate for this connection to the USRP.

usrp_source_sptr.**set_start_time**(*usrp_source_sptr self*, *time_spec_t time*)

Set the start time for incoming samples. To control when samples are received, set this value before starting the flow graph. The value is cleared after each run. When not specified, the start time will be:

usrp_source_sptr.**set_stream_args**(*usrp_source_sptr self*, *stream_args_t stream_args*)

Update the stream args for this device.

This update will only take effect after a restart of the streaming, or before streaming and after construction. This will also delete the current streamer. Note you cannot change the I/O signature of this block using this function, or it will throw.

It is possible to leave the 'channels' fields of unset. In this case, the previous channels field is used.

usrp_source_sptr.**set_subdev_spec**(*usrp_source_sptr self*, *std::string const & spec*, *size_t mboard=0*)

Set the frontend specification.

usrp_source_sptr.**set_thread_priority**(*usrp_source_sptr self*, *int priority*) → int

usrp_source_sptr.**set_time_next_pps**(*usrp_source_sptr self*, *time_spec_t time_spec*)

Set the time registers at the next pps.

usrp_source_sptr.**set_time_now**(*usrp_source_sptr self*, *time_spec_t time_spec*, *size_t mboard=0*)

Sets the time registers immediately.

usrp_source_sptr.**set_time_source**(*usrp_source_sptr self*, *std::string const & source*, *size_t const mboard=0*)

Set the time source for the USRP device.

This sets the method of time synchronization, typically a pulse per second or an encoded time. Typical options for source: external, MIMO.

usrp_source_sptr.**set_time_unknown_pps**(*usrp_source_sptr self*, *time_spec_t time_spec*)

Sync the time registers with an unknown pps edge.

usrp_source_sptr.**set_user_register**(*usrp_source_sptr self*, *uint8_t const addr*, *uint32_t const data*, *size_t mboard=0*)

Perform write on the user configuration register bus. These only exist if the user has implemented custom setting registers in the device FPGA.

usrp_source_sptr.**thread_priority**(*usrp_source_sptr self*) → int