Previous topic
gnuradio.comedi
Next topic
gnuradio.dtv
This Page
Show Source
Quick search
Go
Enter search terms or a module, class or function name.

# gnuradio.digital

Blocks and utilities for digital modulation and demodulation.

gnuradio.digital.**additive_scrambler_bb**(*int mask*, *int seed*, *int len*, *int count=0*, *int bits_per_byte=1*, *std::string const & reset_tag_key*) → additive_scrambler_bb_sptr

Scramble an input stream using an LFSR.

This block scrambles up to 8 bits per byte of the input data stream, starting at the LSB.

The scrambler works by XORing the incoming bit stream by the output of the LFSR. Optionally, after bits have been processed, the shift register is reset to the value. This allows processing fixed length vectors of samples.

Alternatively, the LFSR can be reset using a reset tag to scramble variable length vectors. However, it cannot be reset between bytes.

For details on configuring the LFSR, see gr::digital::lfsr.

Constructor Specific Documentation:

Create additive scrambler.

**Parameters:**
- **mask** – Polynomial mask for LFSR
- **seed** – Initial shift register contents
- **len** – Shift register length
- **count** – Number of bytes after which shift register is reset, 0=never
- **bits_per_byte** – Number of bits per byte
- **reset_tag_key** – When a tag with this key is detected, the shift register is reset (when this is set, count is ignored!)

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

additive_scrambler_bb_sptr.**bits_per_byte**(*additive_scrambler_bb_sptr self*) → int

additive_scrambler_bb_sptr.**count**(*additive_scrambler_bb_sptr self*) → int

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

declare_sample_delay(additive_scrambler_bb_sptr self, unsigned int delay)

additive_scrambler_bb_sptr.**len**(*additive_scrambler_bb_sptr self*) → int

additive_scrambler_bb_sptr.**mask**(*additive_scrambler_bb_sptr self*) → int

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

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

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

pc_input_buffers_full_avg(additive_scrambler_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(additive_scrambler_bb_sptr self) -> pmt_vector_float

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

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

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

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

additive_scrambler_bb_sptr.**seed**(*additive_scrambler_bb_sptr self*) → int

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

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

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

gnuradio.digital.**binary_slicer_fb**() → binary_slicer_fb_sptr

Slice float binary symbol producing 1 bit output.

Constructor Specific Documentation:

Make binary symbol slicer block.

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

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

declare_sample_delay(binary_slicer_fb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(binary_slicer_fb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(binary_slicer_fb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**burst_shaper_cc**(*pmt_vector_cfloat taps*, *int pre_padding=0*, *int post_padding=0*, *bool insert_phasing=False*, *std::string const & length_tag_name*) → burst_shaper_cc_sptr

Burst shaper block for applying burst padding and ramping.

This block applies a configurable amount of zero padding before and/or after a burst indicated by tagged stream length tags.

If phasing symbols are used, an alternating pattern of +1/-1 symbols of length ceil(N/2) will be inserted before and after each burst, where N is the length of the taps vector. The ramp- up/ramp-down shape will be applied to these phasing symbols.

If phasing symbols are not used, the taper will be applied directly to the head and tail of each burst.

Length tags will be updated to include the length of any added zero padding or phasing symbols and will be placed at the beginning of the modified tagged stream. Any other tags found at the same offset as a length tag will also be placed at the beginning of the modified tagged stream, since these tags are assumed to be associated with the burst rather than a specific sample. For example, if “tx_time” tags are used to control bursts, their offsets should be consistent with their associated burst’s length tags. Tags at other offsets will be placed with the samples on which they were found.

Constructor Specific Documentation:

Make a burst shaper block.

**Parameters:**

- **taps** – vector of window taper taps; the first ceil(N/2) items are the up flank and the last ceil(N/2) items are the down flank. If taps.size() is odd, the middle tap will be used as the last item of the up flank and first item of the down flank.
- **pre_padding** – number of zero samples to insert before the burst.
- **post_padding** – number of zero samples to append after the burst.
- **insert_phasing** – if true, insert alternating +1/-1 pattern of length ceil(N/2) before and after the burst and apply ramp up and ramp down taps, respectively, to the inserted patterns instead of the head and tail items of the burst.
- **length_tag_name** – the name of the tagged stream length tag key.

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

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

declare_sample_delay(burst_shaper_cc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(burst_shaper_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(burst_shaper_cc_sptr self) -> pmt_vector_float

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

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

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

burst_shaper_cc_sptr.**post_padding**(*burst_shaper_cc_sptr self*) → int

Returns the amount of zero padding inserted after each burst.

burst_shaper_cc_sptr.**pre_padding**(*burst_shaper_cc_sptr self*) → int

Returns the amount of zero padding inserted before each burst.

burst_shaper_cc_sptr.**prefix_length**(*burst_shaper_cc_sptr self*) → int

Returns the total amount of zero padding and phasing symbols inserted before each burst.

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

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

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

burst_shaper_cc_sptr.**suffix_length**(*burst_shaper_cc_sptr self*) → int

Returns the total amount of zero padding and phasing symbols inserted after each burst.

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

gnuradio.digital.**burst_shaper_ff**(*pmt_vector_float taps*, *int pre_padding=0*, *int post_padding=0*, *bool insert_phasing=False*, *std::string const & length_tag_name*) → burst_shaper_ff_sptr

Burst shaper block for applying burst padding and ramping.

This block applies a configurable amount of zero padding before and/or after a burst indicated by tagged stream length tags.

If phasing symbols are used, an alternating pattern of +1/-1 symbols of length ceil(N/2) will be inserted before and after each burst, where N is the length of the taps vector. The ramp- up/ramp-down shape will be applied to these phasing symbols.

If phasing symbols are not used, the taper will be applied directly to the head and tail of each burst.

Length tags will be updated to include the length of any added zero padding or phasing symbols and will be placed at the beginning of the modified tagged stream. Any other tags found at the same offset as a length tag will also be placed at the beginning of the modified tagged stream, since these tags are assumed to be associated with the burst rather than a specific sample. For example, if “tx_time” tags are used to control bursts, their offsets should be consistent with their associated burst’s length tags. Tags at other offsets will be placed with the samples on which they were found.

Constructor Specific Documentation:

Make a burst shaper block.

**Parameters:**
- **taps** – vector of window taper taps; the first ceil(N/2) items are the up flank and the last ceil(N/2) items are the down flank. If taps.size() is odd, the middle tap will be used as the last item of the up flank and first item of the down flank.
- **pre_padding** – number of zero samples to insert before the burst.
- **post_padding** – number of zero samples to append after the burst.
- **insert_phasing** – if true, insert alternating +1/-1 pattern of length ceil(N/2) before and after the burst and apply ramp up and ramp down taps, respectively, to the inserted patterns instead of the head and tail items of the burst.
- **length_tag_name** – the name of the tagged stream length tag key.

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

burst_shaper_ff_sptr.**declare_sample_delay**(*burst_shaper_ff_sptr self*, *int which*, *int delay*)
declare_sample_delay(burst_shaper_ff_sptr self, unsigned int delay)

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

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

burst_shaper_ff_sptr.**pc_input_buffers_full_avg**(*burst_shaper_ff_sptr self*, *int which*) → float
pc_input_buffers_full_avg(burst_shaper_ff_sptr self) -> pmt_vector_float

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

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

burst_shaper_ff_sptr.**pc_output_buffers_full_avg**(*burst_shaper_ff_sptr self*, *int which*) → float
pc_output_buffers_full_avg(burst_shaper_ff_sptr self) -> pmt_vector_float

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

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

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

burst_shaper_ff_sptr.**post_padding**(*burst_shaper_ff_sptr self*) → int
Returns the amount of zero padding inserted after each burst.

burst_shaper_ff_sptr.**pre_padding**(*burst_shaper_ff_sptr self*) → int
Returns the amount of zero padding inserted before each burst.

burst_shaper_ff_sptr.**prefix_length**(*burst_shaper_ff_sptr self*) → int
Returns the total amount of zero padding and phasing symbols inserted before each burst.

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

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

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

burst_shaper_ff_sptr.**suffix_length**(*burst_shaper_ff_sptr self*) → int
Returns the total amount of zero padding and phasing symbols inserted after each burst.

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

gnuradio.digital.**chunks_to_symbols_bc**(*pmt_vector_cfloat symbol_table*, *int const D=1*) → chunks_to_symbols_bc_sptr
Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_bc_sptr.**D**(*chunks_to_symbols_bc_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_bc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(chunks_to_symbols_bc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_bc_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_bc_sptr.**set_symbol_table**(*chunks_to_symbols_bc_sptr self*, *pmt_vector_cfloat symbol_table*)

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

chunks_to_symbols_bc_sptr.**symbol_table**(*chunks_to_symbols_bc_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**chunks_to_symbols_bf**(*pmt_vector_float symbol_table*, *int const D=1*) → chunks_to_symbols_bf_sptr

Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_bf_sptr.**D**(*chunks_to_symbols_bf_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_bf_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(chunks_to_symbols_bf_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_bf_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_bf_sptr.**set_symbol_table**(*chunks_to_symbols_bf_sptr self*, *pmt_vector_float symbol_table*)

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

chunks_to_symbols_bf_sptr.**symbol_table**(*chunks_to_symbols_bf_sptr self*) → pmt_vector_float

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

gnuradio.digital.**chunks_to_symbols_ic**(*pmt_vector_cfloat symbol_table*, *int const D=1*) → chunks_to_symbols_ic_sptr

Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_ic_sptr.**D**(*chunks_to_symbols_ic_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_ic_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(chunks_to_symbols_ic_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_ic_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_ic_sptr.**set_symbol_table**(*chunks_to_symbols_ic_sptr self*, *pmt_vector_cfloat symbol_table*)

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

chunks_to_symbols_ic_sptr.**symbol_table**(*chunks_to_symbols_ic_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**chunks_to_symbols_if**(*pmt_vector_float symbol_table*, *int const D=1*) → chunks_to_symbols_if_sptr

Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_if_sptr.**D**(*chunks_to_symbols_if_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_if_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(chunks_to_symbols_if_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_if_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_if_sptr.**set_symbol_table**(*chunks_to_symbols_if_sptr self*, *pmt_vector_float symbol_table*)

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

chunks_to_symbols_if_sptr.**symbol_table**(*chunks_to_symbols_if_sptr self*) → pmt_vector_float

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

gnuradio.digital.**chunks_to_symbols_sc**(*pmt_vector_cfloat symbol_table*, *int const D=1*) → chunks_to_symbols_sc_sptr

Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_sc_sptr.**D**(*chunks_to_symbols_sc_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_sc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(chunks_to_symbols_sc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_sc_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_sc_sptr.**set_symbol_table**(*chunks_to_symbols_sc_sptr self*, *pmt_vector_cfloat symbol_table*)

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

chunks_to_symbols_sc_sptr.**symbol_table**(*chunks_to_symbols_sc_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**chunks_to_symbols_sf**(*pmt_vector_float symbol_table*, *int const D=1*) → chunks_to_symbols_sf_sptr

Map a stream of unpacked symbol indexes to stream of float or complex constellation points in D dimensions (D = 1 by default)

The combination of gr::blocks::packed_to_unpacked_XX followed by gr::digital::chunks_to_symbols_XY handles the general case of mapping from a stream of bytes or shorts into arbitrary float or complex symbols.

Constructor Specific Documentation:

Make a chunks-to-symbols block.

**Parameters:**
- **symbol_table** – list that maps chunks to symbols.
- **D** – dimension of table.

chunks_to_symbols_sf_sptr.**D**(*chunks_to_symbols_sf_sptr self*) → int

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

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

declare_sample_delay(chunks_to_symbols_sf_sptr self, unsigned int delay)

chunks_to_symbols_sf_sptr.**message_subscribers**(*chunks_to_symbols_sf_sptr self*, *swig_int_ptr*

*which_port*) → swig_int_ptr

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

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

pc_input_buffers_full_avg(chunks_to_symbols_sf_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(chunks_to_symbols_sf_sptr self) -> pmt_vector_float

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

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

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

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

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

chunks_to_symbols_sf_sptr.**set_symbol_table**(*chunks_to_symbols_sf_sptr self*, *pmt_vector_float symbol_table*)

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

chunks_to_symbols_sf_sptr.**symbol_table**(*chunks_to_symbols_sf_sptr self*) → pmt_vector_float

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

gnuradio.digital.**clock_recovery_mm_cc**(*float omega*, *float gain_omega*, *float mu*, *float gain_mu*, *float omega_relative_limit*) → clock_recovery_mm_cc_sptr

Mueller and M?ller (M&M) based clock recovery block with complex input, complex output.

This implements the Mueller and M?ller (M&M) discrete-time error-tracking synchronizer.

The peak to peak input signal amplitude must be symmetrical about zero, as the M&M timing error detector (TED) is a decision directed TED, and this block uses a symbol decision slicer referenced at zero.

The input signal peak amplitude should be controlled to a consistent level (e.g. +/- 1.0) before this block to achieve consistent results for given gain settings; as the TED's output error signal is directly affected by the input amplitude.

The input signal must have peaks in order for the TED to output a correct error signal. If the input signal pulses do not have peaks (e.g. rectangular pulses) the input signal should be conditioned with a matched pulse filter or other appropriate filter to peak the input pulses. For a rectangular base pulse that is N samples wide, the matched filter taps would be [1.0/float(N)]*N, or in other words a moving average over N samples.

This block will output samples at a rate of one sample per recovered symbol, and is thus not outputting at a constant rate.

Output symbols are not a subset of input, but may be interpolated.

The complex version here is based on: Modified Mueller and Muller clock recovery circuit:

G. R. Danesfahani, T.G. Jeans, "Optimisation of modified Mueller and Muller algorithm," Electronics Letters, Vol. 31, no. 13, 22 June 1995, pp. 1032 - 1033.

Constructor Specific Documentation:

Make a M&M clock recovery block.

**Parameters:**
- **omega** – Initial estimate of samples per symbol
- **gain_omega** – Gain setting for omega update loop
- **mu** – Initial estimate of phase of sample
- **gain_mu** – Gain setting for mu update loop
- **omega_relative_limit** – limit on omega

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

clock_recovery_mm_cc_sptr.**declare_sample_delay**(*clock_recovery_mm_cc_sptr self*, *int which*, *int*

*delay*)

declare_sample_delay(clock_recovery_mm_cc_sptr self, unsigned int delay)

clock_recovery_mm_cc_sptr.**gain_mu**(*clock_recovery_mm_cc_sptr self*) → float

clock_recovery_mm_cc_sptr.**gain_omega**(*clock_recovery_mm_cc_sptr self*) → float

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

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

clock_recovery_mm_cc_sptr.**mu**(*clock_recovery_mm_cc_sptr self*) → float

clock_recovery_mm_cc_sptr.**omega**(*clock_recovery_mm_cc_sptr self*) → float

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

pc_input_buffers_full_avg(clock_recovery_mm_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(clock_recovery_mm_cc_sptr self) -> pmt_vector_float

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

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

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

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

clock_recovery_mm_cc_sptr.**set_gain_mu**(*clock_recovery_mm_cc_sptr self*, *float gain_mu*)

clock_recovery_mm_cc_sptr.**set_gain_omega**(*clock_recovery_mm_cc_sptr self*, *float gain_omega*)

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

clock_recovery_mm_cc_sptr.**set_mu**(*clock_recovery_mm_cc_sptr self*, *float mu*)

clock_recovery_mm_cc_sptr.**set_omega**(*clock_recovery_mm_cc_sptr self*, *float omega*)

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

clock_recovery_mm_cc_sptr.**set_verbose**(*clock_recovery_mm_cc_sptr self*, *bool verbose*)

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

gnuradio.digital.**clock_recovery_mm_ff**(*float omega*, *float gain_omega*, *float mu*, *float gain_mu*, *float omega_relative_limit*) → clock_recovery_mm_ff_sptr

Mueller and M?ller (M&M) based clock recovery block with float input, float output.

This implements the Mueller and M?ller (M&M) discrete-time error-tracking synchronizer.

The peak to peak input signal amplitude must be symmetrical about zero, as the M&M timing error detector (TED) is a decision directed TED, and this block uses a symbol decision slicer referenced at zero.

The input signal peak amplitude should be controlled to a consistent level (e.g. +/- 1.0) before this block to achieve consistent results for given gain settings; as the TED's output error signal is directly affected by the input amplitude.

The input signal must have peaks in order for the TED to output a correct error signal. If the input signal pulses do not have peaks (e.g. rectangular pulses) the input signal should be conditioned with a matched pulse filter or other appropriate filter to peak the input pulses. For a rectangular base pulse that is N samples wide, the matched filter taps would be [1.0/float(N)]*N, or in other words a moving average over N samples.

This block will output samples at a rate of one sample per recovered symbol, and is thus not outputting at a constant rate.

Output symbols are not a subset of input, but may be interpolated.

See "Digital Communication Receivers: Synchronization, Channel Estimation and Signal Processing" by

Heinrich Meyr, Marc Moeneclaey, & Stefan Fechtel. ISBN 0-471-50275-8.

Constructor Specific Documentation:

Make a M&M clock recovery block.

**Parameters:**
- **omega** – Initial estimate of samples per symbol
- **gain_omega** – Gain setting for omega update loop
- **mu** – Initial estimate of phase of sample
- **gain_mu** – Gain setting for mu update loop
- **omega_relative_limit** – maximum relative deviation from omega

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

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

declare_sample_delay(clock_recovery_mm_ff_sptr self, unsigned int delay)

clock_recovery_mm_ff_sptr.**gain_mu**(*clock_recovery_mm_ff_sptr self*) → float

clock_recovery_mm_ff_sptr.**gain_omega**(*clock_recovery_mm_ff_sptr self*) → float

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

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

clock_recovery_mm_ff_sptr.**mu**(*clock_recovery_mm_ff_sptr self*) → float

clock_recovery_mm_ff_sptr.**omega**(*clock_recovery_mm_ff_sptr self*) → float

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

pc_input_buffers_full_avg(clock_recovery_mm_ff_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(clock_recovery_mm_ff_sptr self) -> pmt_vector_float

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

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

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

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

clock_recovery_mm_ff_sptr.**set_gain_mu**(*clock_recovery_mm_ff_sptr self*, *float gain_mu*)

clock_recovery_mm_ff_sptr.**set_gain_omega**(*clock_recovery_mm_ff_sptr self*, *float gain_omega*)

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

clock_recovery_mm_ff_sptr.**set_mu**(*clock_recovery_mm_ff_sptr self*, *float mu*)

clock_recovery_mm_ff_sptr.**set_omega**(*clock_recovery_mm_ff_sptr self*, *float omega*)

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

clock_recovery_mm_ff_sptr.**set_verbose**(*clock_recovery_mm_ff_sptr self*, *bool verbose*)

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

gnuradio.digital.**cma_equalizer_cc**(*int num_taps*, *float modulus*, *float mu*, *int sps*) → cma_equalizer_cc_sptr

Implements constant modulus adaptive filter on complex stream.

The error value and tap update equations (for p=2) can be found in:

D. Godard, “Self-Recovering Equalization and Carrier Tracking in Two-Dimensional Data Communication Systems,” IEEE Transactions on Communications, Vol. 28, No. 11, pp. 1867 - 1875, 1980.

Constructor Specific Documentation:

Make a CMA Equalizer block

**Parameters:**
- **num_taps** – Numer of taps in the equalizer (channel size)
- **modulus** – Modulus of the modulated signals
- **mu** – Gain of the update loop
- **sps** – Number of samples per symbol of the input signal

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

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

declare_sample_delay(cma_equalizer_cc_sptr self, unsigned int delay)

cma_equalizer_cc_sptr.**gain**(*cma_equalizer_cc_sptr self*) → float

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

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

cma_equalizer_cc_sptr.**modulus**(*cma_equalizer_cc_sptr self*) → float

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

pc_input_buffers_full_avg(cma_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(cma_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

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

cma_equalizer_cc_sptr.**set_gain**(*cma_equalizer_cc_sptr self*, *float mu*)

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

cma_equalizer_cc_sptr.**set_modulus**(*cma_equalizer_cc_sptr self*, *float mod*)

cma_equalizer_cc_sptr.**set_taps**(*cma_equalizer_cc_sptr self*, *pmt_vector_cfloat taps*)

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

cma_equalizer_cc_sptr.**taps**(*cma_equalizer_cc_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**constellation_decoder_cb**(*constellation_sptr constellation*) → constellation_decoder_cb_sptr

Constellation Decoder.

Decode a constellation's points from a complex space to (unpacked) bits based on the map of the object.

Constructor Specific Documentation:

Make constellation decoder block.

**Parameters:** **constellation** – A constellation derived from class 'constellation'. Use base() method to get a shared pointer to this base class type.

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

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

declare_sample_delay(constellation_decoder_cb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(constellation_decoder_cb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(constellation_decoder_cb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**constellation_receiver_cb**(*constellation_sptr constellation*, *float loop_bw*, *float fmin*, *float fmax*) → constellation_receiver_cb_sptr

This block makes hard decisions about the received symbols (using a constellation object) and also fine tunes phase synchronization.

The phase and frequency synchronization are based on a Costas loop that finds the error of the incoming signal point compared to its nearest constellation point. The frequency and phase of the NCO are updated according to this error.

Message Ports:

set_constellation (input): Receives a PMT any containing a new gr::digital::constellation object. The PMT is cast back to a gr::digital::constellation_sptr and passes this to set_constellation.

rotate_phase (input): Receives a PMT double to update the phase. The phase value passed in the message is added to the current phase of the receiver.

Constructor Specific Documentation:

Constructs a constellation receiver that (phase/fine freq) synchronizes and decodes constellation points specified by a constellation object.

**Parameters:**
- **constellation** – constellation of points for generic modulation
- **loop_bw** – Loop bandwidth of the Costas Loop (~ 2pi/100)
- **fmin** – minimum normalized frequency value the loop can achieve
- **fmax** – maximum normalized frequency value the loop can achieve

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

constellation_receiver_cb_sptr.**advance_loop**(*constellation_receiver_cb_sptr self*, *float error*)

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

declare_sample_delay(constellation_receiver_cb_sptr self, unsigned int delay)

constellation_receiver_cb_sptr.**frequency_limit**(*constellation_receiver_cb_sptr self*)

constellation_receiver_cb_sptr.**get_alpha**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_beta**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_damping_factor**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_frequency**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_loop_bandwidth**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_max_freq**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_min_freq**(*constellation_receiver_cb_sptr self*) → float

constellation_receiver_cb_sptr.**get_phase**(*constellation_receiver_cb_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(constellation_receiver_cb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(constellation_receiver_cb_sptr self) -> pmt_vector_float

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

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

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

constellation_receiver_cb_sptr.**phase_error_tracking**(*constellation_receiver_cb_sptr self*, *float phase_error*)

constellation_receiver_cb_sptr.**phase_wrap**(*constellation_receiver_cb_sptr self*)

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

constellation_receiver_cb_sptr.**set_alpha**(*constellation_receiver_cb_sptr self*, *float alpha*)

constellation_receiver_cb_sptr.**set_beta**(*constellation_receiver_cb_sptr self*, *float beta*)

constellation_receiver_cb_sptr.**set_damping_factor**(*constellation_receiver_cb_sptr self*, *float df*)

constellation_receiver_cb_sptr.**set_frequency**(*constellation_receiver_cb_sptr self*, *float freq*)

constellation_receiver_cb_sptr.**set_loop_bandwidth**(*constellation_receiver_cb_sptr self*, *float bw*)

constellation_receiver_cb_sptr.**set_max_freq**(*constellation_receiver_cb_sptr self*, *float freq*)

constellation_receiver_cb_sptr.**set_min_freq**(*constellation_receiver_cb_sptr self*, *float freq*)

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

constellation_receiver_cb_sptr.**set_phase**(*constellation_receiver_cb_sptr self*, *float phase*)

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

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

constellation_receiver_cb_sptr.**update_gains**(*constellation_receiver_cb_sptr self*)

gnuradio.digital.**constellation_soft_decoder_cf**(*constellation_sptr constellation*) → constellation_soft_decoder_cf_sptr

Constellation Decoder.

Decode a constellation's points from a complex space to soft bits based on the map and soft decision LUT of the object.

Constructor Specific Documentation:

Make constellation decoder block.

**Parameters:** **constellation** – A constellation derived from class 'constellation'. Use base() method to get a shared pointer to this base class type.

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

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

declare_sample_delay(constellation_soft_decoder_cf_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(constellation_soft_decoder_cf_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(constellation_soft_decoder_cf_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**correlate_access_code_bb**(*std::string const & access_code*, *int threshold*) → correlate_access_code_bb_sptr

Examine input for specified access code, one bit at a time.

input: stream of bits, 1 bit per input byte (data in LSB) output: stream of bits, 2 bits per output byte (data in LSB, flag in next higher bit)

Each output byte contains two valid bits, the data bit, and the flag bit. The LSB (bit 0) is the data bit, and is the original input data, delayed 64 bits. Bit 1 is the flag bit and is 1 if the corresponding data bit is the first data bit following the access code. Otherwise the flag bit is 0.

Constructor Specific Documentation:

Make a correlate_access_code block.

**Parameters:**
- **access_code** – is represented with 1 byte per bit, e.g., “010101010111000100”
- **threshold** – maximum number of bits that may be wrong

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

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

declare_sample_delay(correlate_access_code_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(correlate_access_code_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(correlate_access_code_bb_sptr self) -> pmt_vector_float

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

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

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

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

correlate_access_code_bb_sptr.**set_access_code**(*correlate_access_code_bb_sptr self*, *std::string const & access_code*) → bool

Set a new access code.

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

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

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

gnuradio.digital.**correlate_access_code_bb_ts**(*std::string const & access_code*, *int threshold*, *std::string const & tag_name*) → correlate_access_code_bb_ts_sptr

Examine input for specified access code, one bit at a time.

input: stream of bits (unpacked bytes) output: a tagged stream set of bits from the payload following the access code and header.

This block searches for the given access code by reading in the input bits. Once found, it expects the following 32 samples to contain a header that includes the frame length (16 bits for the length, repeated). It decodes the header to get the frame length in order to set up the the tagged stream key information.

The output of this block is appropriate for use with tagged stream blocks.

Constructor Specific Documentation:

**Parameters:**

- **access_code** – is represented with 1 byte per bit, e.g., “0101010101111000100”
- **threshold** – maximum number of bits that may be wrong
- **tag_name** – key of the tag inserted into the tag stream

correlate_access_code_bb_ts_sptr.**access_code**(*correlate_access_code_bb_ts_sptr self*) → unsigned long long

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

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

declare_sample_delay(correlate_access_code_bb_ts_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(correlate_access_code_bb_ts_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(correlate_access_code_bb_ts_sptr self) -> pmt_vector_float

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

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

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

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

correlate_access_code_bb_ts_sptr.**set_access_code**(*correlate_access_code_bb_ts_sptr self*, *std::string const & access_code*) → bool

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

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

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

gnuradio.digital.**correlate_access_code_ff_ts**(*std::string const & access_code*, *int threshold*, *std::string const & tag_name*) → correlate_access_code_ff_ts_sptr

Examine input for specified access code, one bit at a time.

input: stream of floats (generally, soft decisions) output: a tagged stream set of samples from the payload following the access code and header.

This block searches for the given access code by slicing the soft decision symbol inputs. Once found, it expects the following 32 samples to contain a header that includes the frame length (16 bits for the length, repeated). It decodes the header to get the frame length in order to set up the the tagged stream key information.

The output of this block is appropriate for use with tagged stream blocks.

Constructor Specific Documentation:

**Parameters:**
- **access_code** – is represented with 1 byte per bit, e.g., “010101010111000100”
- **threshold** – maximum number of bits that may be wrong
- **tag_name** – key of the tag inserted into the tag stream

correlate_access_code_ff_ts_sptr.**access_code**(*correlate_access_code_ff_ts_sptr self*) → unsigned long long

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

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

declare_sample_delay(correlate_access_code_ff_ts_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(correlate_access_code_ff_ts_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(correlate_access_code_ff_ts_sptr self) -> pmt_vector_float

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

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

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

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

correlate_access_code_ff_ts_sptr.**set_access_code**(*correlate_access_code_ff_ts_sptr self*, *std::string const & access_code*) → bool

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

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

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

gnuradio.digital.**correlate_access_code_tag_bb**(*std::string const & access_code*, *int threshold*, *std::string const & tag_name*) → correlate_access_code_tag_bb_sptr

Examine input for specified access code, one bit at a time.

input: stream of bits, 1 bit per input byte (data in LSB) output: unaltered stream of bits (plus tags)

This block annotates the input stream with tags. The tags have key name [tag_name], specified in the constructor. Used for searching an input data stream for preambles, etc.

Constructor Specific Documentation:

**Parameters:**
- **access_code** – is represented with 1 byte per bit, e.g., “010101010111000100”
- **threshold** – maximum number of bits that may be wrong
- **tag_name** – key of the tag inserted into the tag stream

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

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

declare_sample_delay(correlate_access_code_tag_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(correlate_access_code_tag_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(correlate_access_code_tag_bb_sptr self) -> pmt_vector_float

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

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

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

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

correlate_access_code_tag_bb_sptr.**set_access_code**(*correlate_access_code_tag_bb_sptr self*, *std::string const & access_code*) → bool

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

correlate_access_code_tag_bb_sptr.**set_tagname**(*correlate_access_code_tag_bb_sptr self*, *std::string const & tagname*)

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

correlate_access_code_tag_bb_sptr.**set_threshold**(*correlate_access_code_tag_bb_sptr self*, *int threshold*)

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

gnuradio.digital.**correlate_and_sync_cc**(*pmt_vector_cfloat symbols*, *pmt_vector_float filter*, *unsigned int sps*, *unsigned int nfilts=32*) → correlate_and_sync_cc_sptr

Correlate to a preamble and send time/phase sync info.

Input: Output: This block is designed to search for a preamble by correlation and uses the results of the correlation to get a time and phase offset estimate. These estimates are passed downstream as stream tags for use by follow-on synchronization blocks.

The preamble is provided as a set of symbols along with a baseband matched filter which we use to create the filtered and upsampled symbol that we will receive over-the-air.

The phase_est tag is used to adjust the phase estimation of any downstream synchronization blocks and is currently used by the gr::digital::costas_loop_cc block.

The time_est tag is used to adjust the sampling timing estimation of any downstream synchronization blocks and is currently used by the gr::digital::pfb_clock_sync_ccf block.

Constructor Specific Documentation:

Make a block that correlates against the vector and outputs a phase and symbol timing estimate.

**Parameters:**
- **symbols** – Set of symbols to correlate against (e.g., a preamble).
- **filter** – Baseband matched filter (e.g., RRC)
- **sps** – Samples per symbol
- **nfilts** – Number of filters in the internal PFB

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

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

declare_sample_delay(correlate_and_sync_cc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(correlate_and_sync_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(correlate_and_sync_cc_sptr self) -> pmt_vector_float

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

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

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

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

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

correlate_and_sync_cc_sptr.**set_symbols**(*correlate_and_sync_cc_sptr self*, *pmt_vector_cfloat symbols*)

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

correlate_and_sync_cc_sptr.**symbols**(*correlate_and_sync_cc_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**corr_est_cc**(*pmt_vector_cfloat symbols*, *float sps*, *unsigned int mark_delay*, *float threshold=0.9*) → corr_est_cc_sptr

Correlate stream with a pre-defined sequence and estimate peak.

Input: Output:

This block is designed to search for a sync word by correlation and uses the results of the correlation to get a time and phase offset estimate. These estimates are passed downstream as stream tags for use by follow-on synchronization blocks.

The sync word is provided as a set of symbols after being filtered by a baseband matched filter.

The phase_est tag can be used by downstream blocks to adjust their phase estimator/correction loops, and is currently implemented by the gr::digital::costas_loop_cc block.

The time_est tag can be used to adjust the sampling timing estimate of any downstream synchronization blocks and is currently implemented by the gr::digital::pfb_clock_sync_ccf block.

The caller must provide a “time_est” and “phase_est” tag marking delay from the start of the correlated signal segment, in order to mark the proper point in the sync word for downstream synchronization blocks. Generally this block cannot know where the actual sync word symbols are located relative to “corr_start”, given that some modulations have pulses with intentional ISI. The user should manually examine the primary output and the “corr_start” tag postition to determine the required tag delay settings for the particular modulation, sync word, and downstream blocks used.

For a discussion of the properties of complex correlations, with respect to signal processing, see: Marple, Jr. S. L., “Estimating Group Delay and Phase Delay via Discrete-Time ‘Analytic’ Cross-Correlation”, Volume 47, No. 9, September 1999

Constructor Specific Documentation:

Make a block that correlates against the vector and outputs a phase and symbol timing estimate.

**Parameters:**
- **symbols** – Set of symbols to correlate against (e.g., a sync word).
- **sps** – Samples per symbol
- **mark_delay** – tag marking delay in samples after the corr_start tag
- **threshold** – Threshold of correlator, relative to a 100% correlation (1.0). Default is 0.9.

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

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

declare_sample_delay(corr_est_cc_sptr self, unsigned int delay)

corr_est_cc_sptr.**mark_delay**(*corr_est_cc_sptr self*) → unsigned int

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

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

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

pc_input_buffers_full_avg(corr_est_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(corr_est_cc_sptr self) -> pmt_vector_float

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

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

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

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

corr_est_cc_sptr.**set_mark_delay**(*corr_est_cc_sptr self*, *unsigned int mark_delay*)

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

corr_est_cc_sptr.**set_symbols**(*corr_est_cc_sptr self*, *pmt_vector_cfloat symbols*)

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

corr_est_cc_sptr.**set_threshold**(*corr_est_cc_sptr self*, *float threshold*)

corr_est_cc_sptr.**symbols**(*corr_est_cc_sptr self*) → pmt_vector_cfloat

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

corr_est_cc_sptr.**threshold**(*corr_est_cc_sptr self*) → float

gnuradio.digital.**costas_loop_cc**(*float loop_bw*, *int order*, *bool use_snr=False*) → costas_loop_cc_sptr

A Costas loop carrier recovery module.

The Costas loop locks to the center frequency of a signal and downconverts it to baseband.

More details can be found online:

J. Feigin, “Practical Costas loop design: Designing a simple and inexpensive BPSK Costas loop carrier recovery circuit,” RF signal processing, pp. 20-36, 2002.

The Costas loop can have two output streams: There is a single optional message input:

Constructor Specific Documentation:

Make a Costas loop carrier recovery block.

**Parameters:**
- **loop_bw** – internal 2nd order loop bandwidth (~ 2pi/100)
- **order** – the loop order, either 2, 4, or 8
- **use_snr** – Use or ignore SNR estimates (from noise message port) in measurements; also uses tanh instead of slicing.

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

costas_loop_cc_sptr.**advance_loop**(*costas_loop_cc_sptr self*, *float error*)

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

declare_sample_delay(costas_loop_cc_sptr self, unsigned int delay)

costas_loop_cc_sptr.**error**(*costas_loop_cc_sptr self*) → float

Returns the current value of the loop error.

costas_loop_cc_sptr.**frequency_limit**(*costas_loop_cc_sptr self*)

costas_loop_cc_sptr.**get_alpha**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_beta**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_damping_factor**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_frequency**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_loop_bandwidth**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_max_freq**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_min_freq**(*costas_loop_cc_sptr self*) → float

costas_loop_cc_sptr.**get_phase**(*costas_loop_cc_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(costas_loop_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(costas_loop_cc_sptr self) -> pmt_vector_float

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

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

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

costas_loop_cc_sptr.**phase_wrap**(*costas_loop_cc_sptr self*)

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

costas_loop_cc_sptr.**set_alpha**(*costas_loop_cc_sptr self*, *float alpha*)

costas_loop_cc_sptr.**set_beta**(*costas_loop_cc_sptr self*, *float beta*)

costas_loop_cc_sptr.**set_damping_factor**(*costas_loop_cc_sptr self*, *float df*)

costas_loop_cc_sptr.**set_frequency**(*costas_loop_cc_sptr self*, *float freq*)

costas_loop_cc_sptr.**set_loop_bandwidth**(*costas_loop_cc_sptr self*, *float bw*)

costas_loop_cc_sptr.**set_max_freq**(*costas_loop_cc_sptr self*, *float freq*)

costas_loop_cc_sptr.**set_min_freq**(*costas_loop_cc_sptr self*, *float freq*)

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

costas_loop_cc_sptr.**set_phase**(*costas_loop_cc_sptr self*, *float phase*)

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

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

costas_loop_cc_sptr.**update_gains**(*costas_loop_cc_sptr self*)

gnuradio.digital.**cpmmod_bc**(*gr::analog::cpm::cpm_type type*, *float h*, *int samples_per_sym*, *int L*, *double beta=0.3*) → cpmmod_bc_sptr

Generic CPM modulator.

Examples:

The input of this block are symbols from an M-ary alphabet +/-1, +/-3, ..., +/-(M-1). Usually, M = 2 and therefore, the valid inputs are +/-1. The modulator will silently accept any other inputs, though. The output is the phase-modulated signal.

Constructor Specific Documentation:

Make CPM modulator block.

**Parameters:**

- **type** – The modulation type. Can be one of LREC, LRC, LSRC, TFM or GAUSSIAN. See gr_cpm::phase_response() for a detailed description.
- **h** – The modulation index. is the maximum phase change that can occur between two symbols, i.e., if you only send ones, the phase will increase by every samples. Set this to 0.5 for Minimum Shift Keying variants.
- **samples_per_sym** – Samples per symbol.
- **L** – The length of the phase duration in symbols. For L=1, this yields full- response CPM symbols, for L > 1, partial-response.
- **beta** – For LSRC, this is the rolloff factor. For Gaussian pulses, this is the 3 dB time-bandwidth product.

cpmmod_bc_sptr.**beta**(*cpmmod_bc_sptr self*) → double

Return the value of beta for the modulator.

cpmmod_bc_sptr.**make_gmskmod_bc**(*cpmmod_bc_sptr self*, *int samples_per_sym=2*, *int L=4*, *double beta=0.3*) → cpmmod_bc_sptr

Make GMSK modulator block.

The type is GAUSSIAN and the modulation index for GMSK is 0.5. This are populated automatically by this factory function.

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

swig_int_ptr

cpmmod_bc_sptr.**samples_per_sym**(*cpmmod_bc_sptr self*) → int

Return the number of samples per symbol.

cpmmod_bc_sptr.**taps**(*cpmmod_bc_sptr self*) → pmt_vector_float

Return the phase response FIR taps.

gnuradio.digital.**crc32_async_bb**(*bool check=False*) → crc32_async_bb_sptr

Byte-stream CRC block for async messages.

Processes packets (as async PDU messages) for CRC32. Theparameter determines if the block acts to check and strip the CRC or to calculate and append the CRC32.

The input PDU is expected to be a message of packet bytes.

When using check mode, if the CRC passes, the output is a payload of the message with the CRC stripped, so the output will be 4 bytes smaller than the input.

When using calculate mode (check == false), then the CRC is calculated on the PDU and appended to it. The output is then 4 bytes longer than the input.

This block implements the CRC32 using the Boost crc_optimal class for 32-bit CRCs with the standard generator 0x04C11DB7.

Constructor Specific Documentation:

**Parameters:** **check** – Set to true if you want to check CRC, false to create CRC.

crc32_async_bb_sptr.**active_thread_priority**(*crc32_async_bb_sptr self*) → int

crc32_async_bb_sptr.**declare_sample_delay**(*crc32_async_bb_sptr self*, *int which*, *int delay*)

declare_sample_delay(crc32_async_bb_sptr self, unsigned int delay)

crc32_async_bb_sptr.**message_subscribers**(*crc32_async_bb_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

crc32_async_bb_sptr.**min_noutput_items**(*crc32_async_bb_sptr self*) → int

crc32_async_bb_sptr.**pc_input_buffers_full_avg**(*crc32_async_bb_sptr self*, *int which*) → float

pc_input_buffers_full_avg(crc32_async_bb_sptr self) -> pmt_vector_float

crc32_async_bb_sptr.**pc_noutput_items_avg**(*crc32_async_bb_sptr self*) → float

crc32_async_bb_sptr.**pc_nproduced_avg**(*crc32_async_bb_sptr self*) → float

crc32_async_bb_sptr.**pc_output_buffers_full_avg**(*crc32_async_bb_sptr self*, *int which*) → float

pc_output_buffers_full_avg(crc32_async_bb_sptr self) -> pmt_vector_float

crc32_async_bb_sptr.**pc_throughput_avg**(*crc32_async_bb_sptr self*) → float

crc32_async_bb_sptr.**pc_work_time_avg**(*crc32_async_bb_sptr self*) → float

crc32_async_bb_sptr.**pc_work_time_total**(*crc32_async_bb_sptr self*) → float

crc32_async_bb_sptr.**sample_delay**(*crc32_async_bb_sptr self*, *int which*) → unsigned int

crc32_async_bb_sptr.**set_min_noutput_items**(*crc32_async_bb_sptr self*, *int m*)

crc32_async_bb_sptr.**set_thread_priority**(*crc32_async_bb_sptr self*, *int priority*) → int

crc32_async_bb_sptr.**thread_priority**(*crc32_async_bb_sptr self*) → int

gnuradio.digital.**crc32_bb**(*bool check=False*, *std::string const & lengthtagname*, *bool packed=True*) → crc32_bb_sptr

Byte-stream CRC block.

Input: stream of bytes, which form a packet. The first byte of the packet has a tag with key “length” and the value being the number of bytes in the packet.

Output: The same bytes as incoming, but trailing a CRC32 of the packet. The tag is re-set to the new length.

Constructor Specific Documentation:

**Parameters:**
- **check** – Set to true if you want to check CRC, false to create CRC.
- **lengthtagname** – Length tag key for the tagged stream.
- **packed** – If the data is packed or unpacked bits (default=true).

crc32_bb_sptr.**active_thread_priority**(*crc32_bb_sptr self*) → int

crc32_bb_sptr.**declare_sample_delay**(*crc32_bb_sptr self*, *int which*, *int delay*)
declare_sample_delay(crc32_bb_sptr self, unsigned int delay)

crc32_bb_sptr.**message_subscribers**(*crc32_bb_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

crc32_bb_sptr.**min_noutput_items**(*crc32_bb_sptr self*) → int

crc32_bb_sptr.**pc_input_buffers_full_avg**(*crc32_bb_sptr self*, *int which*) → float
pc_input_buffers_full_avg(crc32_bb_sptr self) -> pmt_vector_float

crc32_bb_sptr.**pc_noutput_items_avg**(*crc32_bb_sptr self*) → float

crc32_bb_sptr.**pc_nproduced_avg**(*crc32_bb_sptr self*) → float

crc32_bb_sptr.**pc_output_buffers_full_avg**(*crc32_bb_sptr self*, *int which*) → float
pc_output_buffers_full_avg(crc32_bb_sptr self) -> pmt_vector_float

crc32_bb_sptr.**pc_throughput_avg**(*crc32_bb_sptr self*) → float

crc32_bb_sptr.**pc_work_time_avg**(*crc32_bb_sptr self*) → float

crc32_bb_sptr.**pc_work_time_total**(*crc32_bb_sptr self*) → float

crc32_bb_sptr.**sample_delay**(*crc32_bb_sptr self*, *int which*) → unsigned int

crc32_bb_sptr.**set_min_noutput_items**(*crc32_bb_sptr self*, *int m*)

crc32_bb_sptr.**set_thread_priority**(*crc32_bb_sptr self*, *int priority*) → int

crc32_bb_sptr.**thread_priority**(*crc32_bb_sptr self*) → int

gnuradio.digital.**descrambler_bb**(*int mask*, *int seed*, *int len*) → descrambler_bb_sptr
Descramber an input stream using an LFSR.

Descramble an input stream using an LFSR. This block works on the LSB only of the input data stream, i.e., on an “unpacked binary” stream, and produces the same format on its output.

Constructor Specific Documentation:

Make a descrambler block.

**Parameters:**
- **mask** – Polynomial mask for LFSR
- **seed** – Initial shift register contents
- **len** – Shift register length

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

descrambler_bb_sptr.**declare_sample_delay**(*descrambler_bb_sptr self*, *int which*, *int delay*)
declare_sample_delay(descrambler_bb_sptr self, unsigned int delay)

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

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

descrambler_bb_sptr.**pc_input_buffers_full_avg**(*descrambler_bb_sptr self*, *int which*) → float
pc_input_buffers_full_avg(descrambler_bb_sptr self) -> pmt_vector_float

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

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

descrambler_bb_sptr.**pc_output_buffers_full_avg**(*descrambler_bb_sptr self*, *int which*) → float
pc_output_buffers_full_avg(descrambler_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**diff_decoder_bb**(*unsigned int modulus*) → diff_decoder_bb_sptr

Differential encoder: y[0] = (x[0] - x[-1]) % M.

Uses current and previous symbols and the alphabet modulus to perform differential decoding.

Constructor Specific Documentation:

Make a differntial decoder block.

**Parameters:** **modulus** – Modulus of code's alphabet

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

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

declare_sample_delay(diff_decoder_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(diff_decoder_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(diff_decoder_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**diff_encoder_bb**(*unsigned int modulus*) → diff_encoder_bb_sptr

Differential decoder: y[0] = (x[0] + y[-1]) % M.

Uses current and previous symbols and the alphabet modulus to perform differential encoding.

Constructor Specific Documentation:

Make a differntial encoder block.

**Parameters:** **modulus** – Modulus of code's alphabet

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

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

declare_sample_delay(diff_encoder_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(diff_encoder_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(diff_encoder_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**diff_phasor_cc**() → diff_phasor_cc_sptr

Differential decoding based on phase change.

Uses the phase difference between two symbols to determine the output symbol:

Constructor Specific Documentation:

Make a differential phasor decoding block.

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

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

declare_sample_delay(diff_phasor_cc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(diff_phasor_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(diff_phasor_cc_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**fll_band_edge_cc**(*float samps_per_sym*, *float rolloff*, *int filter_size*, *float bandwidth*) → fll_band_edge_cc_sptr

Frequency Lock Loop using band-edge filters.

The frequency lock loop derives a band-edge filter that covers the upper and lower bandwidths of a digitally-modulated signal. The bandwidth range is determined by the excess bandwidth (e.g., rolloff factor) of the modulated signal. The placement in frequency of the band-edges is determined by the oversampling ratio (number of samples per symbol) and the excess bandwidth. The size of the filters should be fairly large so as to average over a number of symbols.

The FLL works by filtering the upper and lower band edges into x_u(t) and x_l(t), respectively. These are combined to form cc(t) = x_u(t) + x_l(t) and ss(t) = x_u(t) - x_l(t). Combining these to form the signal e(t) = Re{cc(t) imes ss(t)^*} (where ^* is the complex conjugate) provides an error signal at the DC term that is directly proportional to the carrier frequency. We then make a second-order loop using the error signal that is the running average of e(t).

In practice, the above equation can be simplified by just comparing the absolute value squared of the output

of both filters: abs(x_l(t))^2 - abs(x_u(t))^2 = norm(x_l(t)) - norm(x_u(t)).

In theory, the band-edge filter is the derivative of the matched filter in frequency, (H_be(f) = frac{H(f)}{df}). In practice, this comes down to a quarter sine wave at the point of the matched filter's rolloff (if it's a raised-cosine, the derivative of a cosine is a sine). Extend this sine by another quarter wave to make a half wave around the band-edges is equivalent in time to the sum of two sinc functions. The baseband filter fot the band edges is therefore derived from this sum of sincs. The band edge filters are then just the baseband signal modulated to the correct place in frequency. All of these calculations are done in the 'design_filter' function.

**Note:** We use FIR filters here because the filters have to have a flat phase response over the entire frequency range to allow their comparisons to be valid.

It is very important that the band edge filters be the derivatives of the pulse shaping filter, and that they be linear phase. Otherwise, the variance of the error will be very large.

Constructor Specific Documentation:

Make an FLL block.

**Parameters:**
- **samps_per_sym** – (float) number of samples per symbol
- **rolloff** – (float) Rolloff (excess bandwidth) of signal filter
- **filter_size** – (int) number of filter taps to generate
- **bandwidth** – (float) Loop bandwidth

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

fll_band_edge_cc_sptr.**advance_loop**(*fll_band_edge_cc_sptr self*, *float error*)

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

declare_sample_delay(fll_band_edge_cc_sptr self, unsigned int delay)

fll_band_edge_cc_sptr.**filter_size**(*fll_band_edge_cc_sptr self*) → int

Returns the number of taps of the filter.

fll_band_edge_cc_sptr.**frequency_limit**(*fll_band_edge_cc_sptr self*)

fll_band_edge_cc_sptr.**get_alpha**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_beta**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_damping_factor**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_frequency**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_loop_bandwidth**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_max_freq**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_min_freq**(*fll_band_edge_cc_sptr self*) → float

fll_band_edge_cc_sptr.**get_phase**(*fll_band_edge_cc_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(fll_band_edge_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(fll_band_edge_cc_sptr self) -> pmt_vector_float

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

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

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

fll_band_edge_cc_sptr.**phase_wrap**(*fll_band_edge_cc_sptr self*)

fll_band_edge_cc_sptr.**print_taps**(*fll_band_edge_cc_sptr self*)

Print the taps to screen.

fll_band_edge_cc_sptr.**rolloff**(*fll_band_edge_cc_sptr self*) → float

Returns the rolloff factor used for the filter.

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

fll_band_edge_cc_sptr.**samples_per_symbol**(*fll_band_edge_cc_sptr self*) → float

Returns the number of sampler per symbol used for the filter.

fll_band_edge_cc_sptr.**set_alpha**(*fll_band_edge_cc_sptr self*, *float alpha*)

fll_band_edge_cc_sptr.**set_beta**(*fll_band_edge_cc_sptr self*, *float beta*)

fll_band_edge_cc_sptr.**set_damping_factor**(*fll_band_edge_cc_sptr self*, *float df*)

fll_band_edge_cc_sptr.**set_filter_size**(*fll_band_edge_cc_sptr self*, *int filter_size*)

Set the number of taps in the filter.

This sets the number of taps in the band-edge filters. Setting this will force a recalculation of the filter taps.

This should be about the same number of taps used in the transmitter's shaping filter and also not very large. A large number of taps will result in a large delay between input and frequency estimation, and so will not be as accurate. Between 30 and 70 taps is usual.

fll_band_edge_cc_sptr.**set_frequency**(*fll_band_edge_cc_sptr self*, *float freq*)

fll_band_edge_cc_sptr.**set_loop_bandwidth**(*fll_band_edge_cc_sptr self*, *float bw*)

fll_band_edge_cc_sptr.**set_max_freq**(*fll_band_edge_cc_sptr self*, *float freq*)

fll_band_edge_cc_sptr.**set_min_freq**(*fll_band_edge_cc_sptr self*, *float freq*)

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

fll_band_edge_cc_sptr.**set_phase**(*fll_band_edge_cc_sptr self*, *float phase*)

fll_band_edge_cc_sptr.**set_rolloff**(*fll_band_edge_cc_sptr self*, *float rolloff*)

Set the rolloff factor of the shaping filter.

This sets the rolloff factor that is used in the pulse shaping filter and is used to calculate the filter taps. Changing this will force a recalculation of the filter taps.

This should be the same value that is used in the transmitter's pulse shaping filter. It must be between 0 and 1 and is usually between 0.2 and 0.5 (where 0.22 and 0.35 are commonly used values).

fll_band_edge_cc_sptr.**set_samples_per_symbol**(*fll_band_edge_cc_sptr self*, *float sps*)

Set the number of samples per symbol.

Set's the number of samples per symbol the system should use. This value is used to calculate the filter taps and will force a recalculation.

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

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

fll_band_edge_cc_sptr.**update_gains**(*fll_band_edge_cc_sptr self*)

gnuradio.digital.**framer_sink_1**(*msg_queue_sptr target_queue*) → framer_sink_1_sptr

Given a stream of bits and access_code flags, assemble packets.

input: stream of bytes from digital_correlate_access_code_bb output: none. Pushes assembled packet into target queue

The framer expects a fixed length header of 2 16-bit shorts containing the payload length, followed by the payload. If the 2 16-bit shorts are not identical, this packet is ignored. Better algs are welcome.

The input data consists of bytes that have two bits used. Bit 0, the LSB, contains the data bit. Bit 1 if set, indicates that the corresponding bit is the the first bit of the packet. That is, this bit is the first one after the access code.

Constructor Specific Documentation:

Make a framer_sink_1 block.

**Parameters:** **target_queue** – The message queue where frames go.

framer_sink_1_sptr.**active_thread_priority**(*framer_sink_1_sptr self*) → int

framer_sink_1_sptr.**declare_sample_delay**(*framer_sink_1_sptr self*, *int which*, *int delay*)
declare_sample_delay(framer_sink_1_sptr self, unsigned int delay)

framer_sink_1_sptr.**message_subscribers**(*framer_sink_1_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

framer_sink_1_sptr.**min_noutput_items**(*framer_sink_1_sptr self*) → int

framer_sink_1_sptr.**pc_input_buffers_full_avg**(*framer_sink_1_sptr self*, *int which*) → float
pc_input_buffers_full_avg(framer_sink_1_sptr self) -> pmt_vector_float

framer_sink_1_sptr.**pc_noutput_items_avg**(*framer_sink_1_sptr self*) → float

framer_sink_1_sptr.**pc_nproduced_avg**(*framer_sink_1_sptr self*) → float

framer_sink_1_sptr.**pc_output_buffers_full_avg**(*framer_sink_1_sptr self*, *int which*) → float
pc_output_buffers_full_avg(framer_sink_1_sptr self) -> pmt_vector_float

framer_sink_1_sptr.**pc_throughput_avg**(*framer_sink_1_sptr self*) → float

framer_sink_1_sptr.**pc_work_time_avg**(*framer_sink_1_sptr self*) → float

framer_sink_1_sptr.**pc_work_time_total**(*framer_sink_1_sptr self*) → float

framer_sink_1_sptr.**sample_delay**(*framer_sink_1_sptr self*, *int which*) → unsigned int

framer_sink_1_sptr.**set_min_noutput_items**(*framer_sink_1_sptr self*, *int m*)

framer_sink_1_sptr.**set_thread_priority**(*framer_sink_1_sptr self*, *int priority*) → int

framer_sink_1_sptr.**thread_priority**(*framer_sink_1_sptr self*) → int

gnuradio.digital.**glfsr_source_b**(*int degree*, *bool repeat=True*, *int mask=0*, *int seed=1*) → glfsr_source_b_sptr

Galois LFSR pseudo-random source.

Constructor Specific Documentation:

Make a Galois LFSR pseudo-random source block.

**Parameters:**
- **degree** – Degree of shift register must be in [1, 32]. If mask is 0, the degree determines a default mask (see digital_impl_glfsr.cc for the mapping).
- **repeat** – Set to repeat sequence.
- **mask** – Allows a user-defined bit mask for indexes of the shift register to feed back.
- **seed** – Initial setting for values in shift register.

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

glfsr_source_b_sptr.**declare_sample_delay**(*glfsr_source_b_sptr self*, *int which*, *int delay*)
declare_sample_delay(glfsr_source_b_sptr self, unsigned int delay)

glfsr_source_b_sptr.**mask**(*glfsr_source_b_sptr self*) → int

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

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

glfsr_source_b_sptr.**pc_input_buffers_full_avg**(*glfsr_source_b_sptr self*, *int which*) → float
pc_input_buffers_full_avg(glfsr_source_b_sptr self) -> pmt_vector_float

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

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

glfsr_source_b_sptr.**pc_output_buffers_full_avg**(*glfsr_source_b_sptr self*, *int which*) → float
pc_output_buffers_full_avg(glfsr_source_b_sptr self) -> pmt_vector_float

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

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

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

glfsr_source_b_sptr.**period**(*glfsr_source_b_sptr self*) → unsigned int

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

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

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

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

gnuradio.digital.**glfsr_source_f**(*int degree*, *bool repeat=True*, *int mask=0*, *int seed=1*) → glfsr_source_f_sptr

Galois LFSR pseudo-random source generating float outputs -1.0 - 1.0.

Constructor Specific Documentation:

Make a Galois LFSR pseudo-random source block.

**Parameters:**
- **degree** – Degree of shift register must be in [1, 32]. If mask is 0, the degree determines a default mask (see digital_impl_glfsr.cc for the mapping).
- **repeat** – Set to repeat sequence.
- **mask** – Allows a user-defined bit mask for indexes of the shift register to feed back.
- **seed** – Initial setting for values in shift register.

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

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

declare_sample_delay(glfsr_source_f_sptr self, unsigned int delay)

glfsr_source_f_sptr.**mask**(*glfsr_source_f_sptr self*) → int

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

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

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

pc_input_buffers_full_avg(glfsr_source_f_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(glfsr_source_f_sptr self) -> pmt_vector_float

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

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

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

glfsr_source_f_sptr.**period**(*glfsr_source_f_sptr self*) → unsigned int

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

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

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

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

gnuradio.digital.**hdlc_deframer_bp**(*int length_min*, *int length_max*) → hdlc_deframer_bp_sptr

HDLC deframer which takes in unpacked bits, and outputs PDU binary blobs. Frames which do not pass CRC are rejected.

Constructor Specific Documentation:

Return a shared_ptr to a new instance of digital::hdlc_deframer.

**Parameters:**
- **length_min** – Minimum frame size (default: 32)
- **length_max** – Maximum frame size (default: 500)

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

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

declare_sample_delay(hdlc_deframer_bp_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(hdlc_deframer_bp_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(hdlc_deframer_bp_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**hdlc_framer_pb**(*std::string const frame_tag_name*) → hdlc_framer_pb_sptr

HDLC framer which takes in PMT binary blobs and outputs HDLC frames as unpacked bits, with CRC and bit stuffing added. The first sample of the frame is tagged with the tag frame_tag_name and includes a length field for tagged_stream use.

This block outputs one whole frame at a time; if there is not enough output buffer space to fit a frame, it is pushed onto a queue. As a result flowgraphs which only run for a finite number of samples may not receive all frames in the queue, due to the scheduler's granularity. For flowgraphs that stream continuously (anything using a USRP) this should not be an issue.

Constructor Specific Documentation:

Return a shared_ptr to a new instance of digital::hdlc_framer.

**Parameters:** **frame_tag_name** – The tag to add to the first sample of each frame.

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

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

declare_sample_delay(hdlc_framer_pb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(hdlc_framer_pb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(hdlc_framer_pb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**header_payload_demux**(*int const header_len*, *int const items_per_symbol=1*, *int const guard_interval=0*, *std::string const & length_tag_key*, *std::string const & trigger_tag_key*, *bool const output_symbols=False*, *size_t const itemsize*, *std::string const & timing_tag_key*, *double const samp_rate=1.0*, *std::vector< std::string, std::allocator< std::string > > const & special_tags*, *size_t const header_padding=0*) → header_payload_demux_sptr

Header/Payload demuxer (HPD).

This block is designed to demultiplex packets from a bursty transmission. The typical application for this block is the case when you are receiving packets with yet-to-determine length. This block will pass the header section to other blocks for demodulation. Using the information from the demodulated header, it will then output the payload. The beginning of the header needs to be identified by a trigger signal (see below).

Theory of Operation Input 0 takes a continuous transmission of samples (items). Input 1 is an optional input for the trigger signal (mark beginning of packets). In this case, a non-zero value on input 1 identifies the beginning of a packet. Otherwise, a tag with the key specified in is used as a trigger (its value is irrelevant).

Until a trigger signal is detected, all samples are dropped onto the floor. Once a trigger is detected, a total of items are copied to output 0. The block then stalls until it receives a message on the message port . The message must be a PMT dictionary; all key/value pairs are copied as tags to the first item of the payload (which is assumed to be the first item after the header). The value corresponding to the key specified in is read and taken as the payload length. The payload, together with the header data as tags, is then copied to output 1.

If the header demodulation fails, the header must send a PMT with value pmt::PMT_F. The state gets reset and the header is ignored.

Symbols, Items and Item Sizes To generically and transparently handle different kinds of modulations, including OFDM, this block distinguises between and .

Items are what are consumed at the input. Anything that uses complex samples will therefore use an itemsize of . Symbols are a way of grouping items. In OFDM, we usually don't care about individual samples, but we do care about full OFDM symbols, so we setto the IFFT / FFT length of the OFDM modulator / demodulator. For single-carrier modulations, this value can be set to the number of samples per symbol, to handle data in number of symbols, or to 1 to handle data in number of samples. If specified, items are discarded before every symbol. This is useful for demuxing bursts of OFDM signals.

On the output, we can deal with symbols directly by setting to true. In that case, the output item size is the .

> OFDM with 48 sub-carriers, using a length-64 IFFT on the modulator, and a cyclic-prefix length of 16 samples. In this case, is , because we're receiving complex samples. One OFDM symbol has 64 samples, hence is set to 64, and to 16. The header length is specified in number of OFDM symbols. Because we want to deal with full OFDM symbols, we set to true.
>
> PSK-modulated signals, with 4 samples per symbol. Again,i s because we're still dealing with complex samples. is 4, because one item is one sample. must be set to 0. The header length is given in number of PSK symbols.

Handling timing uncertainty on the trigger By default, the assumption is made that the trigger arrives onthe sample that the header starts. These triggers typically come from timing synchronization algorithms which may be suboptimal, and have a known timing uncertainty (e.g., we know the trigger might be a sample too early or too late).

The demuxer has an option for this case, the . If this value is non-zero, it specifies the number of items that are prepended and appended to the header before copying it to the header output.

Example: Say our synchronization algorithm can be off by up to two samples, and the header length is 20 samples. So we set to 20, and to 2. Now assume a trigger arrives on sample index 100. We copy a total of 24 samples to the header port, starting at sample index 98.

The payload is padded. Let's say the header demod reports a payload length of 100. In the previous examples, we would copy 100 samples to the payload port, starting at sample index 120 (this means the padded samples appended to the header are copied to both ports!). However, the header demodulator has the option to specify a payload offset, which cannot exceed the padding value. To do this, include a key in the message sent back to the HPD. A negative value means the payload starts earlier than otherwise. (If you wanted to always pad the payload, you could set to and increase the reported length of the payload).

Because the padding is specified in number of items, and not symbols, this value can only be multiples of the number of items per symbol either is true, or a guard interval is specified (or both). Note that in practice, it is rare that both a guard interval is specified a padding value is required. The difference between the padding value and a guard interval is that a guard interval is part of the signal, and comes with symbol, whereas the header padding is added to only the header, and is not by design.

Tag Handling Any tags on the input stream are copied to the corresponding output they're on an item that is propagated. Note that a tag on the header items is copied to the header stream; that means the header-parsing block must handle these tags if they should go on the payload. A special case are tags on items that

make up the guard interval. These are copied to the first item of the following symbol. If a tag is situated very close to the end of the payload, it might be unclear if it belongs to this packet or the following. In this case, it is possible that the tag might be propagated twice.

Tags outside of packets are generally discarded. If there are tags that carry important information that must not be list, there are two additional mechanisms to preserve the tags:

Constructor Specific Documentation:

**Parameters:**
- **header_len** – Number of symbols per header
- **items_per_symbol** – Number of items per symbol
- **guard_interval** – Number of items between two consecutive symbols
- **length_tag_key** – Key of the frame length tag
- **trigger_tag_key** – Key of the trigger tag
- **output_symbols** – Output symbols (true) or items (false)?
- **itemsize** – Item size (bytes per item)
- **timing_tag_key** – The name of the tag with timing information, usually 'rx_time' or empty (this means timing info is discarded)
- **samp_rate** – Sampling rate at the input. Necessary to calculate the rx time of packets.
- **special_tags** – A vector of strings denoting tags which shall be preserved (see Tag Handling)
- **header_padding** – A number of items that is appended and prepended to the header.

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

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

declare_sample_delay(header_payload_demux_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(header_payload_demux_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(header_payload_demux_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**kurtotic_equalizer_cc**(*int num_taps*, *float mu*) → kurtotic_equalizer_cc_sptr

Implements a kurtosis-based adaptive equalizer on complex stream.

**Warning:** This block does not yet work.

"Y. Guo, J. Zhao, Y. Sun, "Sign kurtosis maximization based blind equalization algorithm," IEEE Conf. on Control, Automation, Robotics and Vision, Vol. 3, Dec. 2004, pp. 2052 - 2057."

Constructor Specific Documentation:

**Parameters:**
- **num_taps** –
- **mu** –

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

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

declare_sample_delay(kurtotic_equalizer_cc_sptr self, unsigned int delay)

kurtotic_equalizer_cc_sptr.**gain**(*kurtotic_equalizer_cc_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(kurtotic_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(kurtotic_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

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

kurtotic_equalizer_cc_sptr.**set_gain**(*kurtotic_equalizer_cc_sptr self*, *float mu*)

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

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

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

gnuradio.digital.**lms_dd_equalizer_cc**(*int num_taps*, *float mu*, *int sps*, *constellation_sptr cnst*) → lms_dd_equalizer_cc_sptr

Least-Mean-Square Decision Directed Equalizer (complex in/out)

This block implements an LMS-based decision-directed equalizer. It uses a set of weights, w, to correlate against the inputs, u, and a decisions is then made from this output. The error in the decision is used to update the weight vector.

y[n] = conj(w[n]) u[n] d[n] = decision(y[n]) e[n] = d[n] - y[n] w[n+1] = w[n] + mu u[n] conj(e[n])

Where mu is a gain value (between 0 and 1 and usually small, around 0.001 - 0.01.

This block uses the digital_constellation object for making the decision from y[n]. Create the constellation object for whatever constellation is to be used and pass in the object. In Python, you can use something like:

self.constellation = digital.constellation_qpsk()

To create a QPSK constellation (see the digital_constellation block for more details as to what constellations are available or how to create your own). You then pass the object to this block as an sptr, or using “self.constellation.base()”.

The theory for this algorithm can be found in Chapter 9 of: S. Haykin, Adaptive Filter Theory, Upper Saddle River, NJ: Prentice Hall, 1996.

Constructor Specific Documentation:

Make an LMS decision-directed equalizer

**Parameters:**
- **num_taps** – Number of taps in the equalizer (channel size)
- **mu** – Gain of the update loop
- **sps** – Number of samples per symbol of the input signal
- **cnst** – A constellation derived from class ‘constellation’. Use base() method to get a shared pointer to this base class type.

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

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

declare_sample_delay(lms_dd_equalizer_cc_sptr self, unsigned int delay)

lms_dd_equalizer_cc_sptr.**gain**(*lms_dd_equalizer_cc_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(lms_dd_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(lms_dd_equalizer_cc_sptr self) -> pmt_vector_float

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

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

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

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

lms_dd_equalizer_cc_sptr.**set_gain**(*lms_dd_equalizer_cc_sptr self*, *float mu*)

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

lms_dd_equalizer_cc_sptr.**set_taps**(*lms_dd_equalizer_cc_sptr self*, *pmt_vector_cfloat taps*)

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

lms_dd_equalizer_cc_sptr.**taps**(*lms_dd_equalizer_cc_sptr self*) → pmt_vector_cfloat

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

gnuradio.digital.**map_bb**(*std::vector< int, std::allocator< int > > const & map*) → map_bb_sptr

output[i] = map[input[i]]

This block maps an incoming signal to the value in the map. The block expects that the incoming signal has a maximum value of len(map)-1.

-> output[i] = map[input[i]]

Constructor Specific Documentation:

Make a map block.

**Parameters:** **map** – a vector of integers that maps x to map[x].

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

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

declare_sample_delay(map_bb_sptr self, unsigned int delay)

map_bb_sptr.**map**(*map_bb_sptr self*) → std::vector< int,std::allocator< int > >

Apply element-wise to the elements of list and returns a list of the results, in order.

must be a list. The dynamic order in which is applied to the elements of is unspecified.

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

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

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

pc_input_buffers_full_avg(map_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(map_bb_sptr self) -> pmt_vector_float

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

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

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

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

map_bb_sptr.**set_map**(*map_bb_sptr self*, *std::vector< int, std::allocator< int > > const & map*)

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

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

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

gnuradio.digital.**mpsk_receiver_cc**(*unsigned int M*, *float theta*, *float loop_bw*, *float fmin*, *float fmax*, *float mu*, *float gain_mu*, *float omega*, *float gain_omega*, *float omega_rel*) → mpsk_receiver_cc_sptr

This block takes care of receiving M-PSK modulated signals through phase, frequency, and symbol synchronization.

It performs carrier frequency and phase locking as well as symbol timing recovery. It works with (D)BPSK, (D)QPSK, and (D)8PSK as tested currently. It should also work for OQPSK and PI/4 DQPSK.

The phase and frequency synchronization are based on a Costas loop that finds the error of the incoming signal point compared to its nearest constellation point. The frequency and phase of the NCO are updated according to this error. There are optimized phase error detectors for BPSK and QPSK, but 8PSK is done using a brute-force computation of the constellation points to find the minimum.

The symbol synchronization is done using a modified Mueller and Muller circuit from the paper:

“G. R. Danesfahani, T. G. Jeans, “Optimisation of modified Mueller and Muller algorithm,” Electronics Letters, Vol. 31, no. 13, 22 June 1995, pp. 1032 - 1033.”

This circuit interpolates the downconverted sample (using the NCO developed by the Costas loop) every mu samples, then it finds the sampling error based on this and the past symbols and the decision made on the samples. Like the phase error detector, there are optimized decision algorithms for BPSK and QPKS, but 8PSK uses another brute force computation against all possible symbols. The modifications to the M&M used here reduce self-noise.

Constructor Specific Documentation:

Make a M-PSK receiver block.

The constructor also chooses which phase detector and decision maker to use in the work loop based on the value of M.

**Parameters:**

- **M** – modulation order of the M-PSK modulation
- **theta** – any constant phase rotation from the real axis of the constellation
- **loop_bw** – Loop bandwidth to set gains of phase/freq tracking loop
- **fmin** – minimum normalized frequency value the loop can achieve
- **fmax** – maximum normalized frequency value the loop can achieve
- **mu** – initial parameter for the interpolator [0,1]
- **gain_mu** – gain parameter of the M&M error signal to adjust mu (~0.05)
- **omega** – initial value for the number of symbols between samples (~number of samples/symbol)
- **gain_omega** – gain parameter to adjust omega based on the error (~omega^2/4)
- **omega_rel** – sets the maximum (omega*(1+omega_rel)) and minimum (omega*(1+omega_rel)) omega (~0.005)

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

mpsk_receiver_cc_sptr.**advance_loop**(*mpsk_receiver_cc_sptr self*, *float error*)

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

declare_sample_delay(mpsk_receiver_cc_sptr self, unsigned int delay)

mpsk_receiver_cc_sptr.**frequency_limit**(*mpsk_receiver_cc_sptr self*)

mpsk_receiver_cc_sptr.**gain_mu**(*mpsk_receiver_cc_sptr self*) → float

Returns mu gain factor.

mpsk_receiver_cc_sptr.**gain_omega**(*mpsk_receiver_cc_sptr self*) → float

Returns omega gain factor.

mpsk_receiver_cc_sptr.**gain_omega_rel**(*mpsk_receiver_cc_sptr self*) → float

Returns the relative omega limit.

mpsk_receiver_cc_sptr.**get_alpha**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_beta**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_damping_factor**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_frequency**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_loop_bandwidth**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_max_freq**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_min_freq**(*mpsk_receiver_cc_sptr self*) → float

mpsk_receiver_cc_sptr.**get_phase**(*mpsk_receiver_cc_sptr self*) → float

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

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

mpsk_receiver_cc_sptr.**modulation_order**(*mpsk_receiver_cc_sptr self*) → float

Returns the modulation order (M) currently set.

mpsk_receiver_cc_sptr.**mu**(*mpsk_receiver_cc_sptr self*) → float

Returns current value of mu.

mpsk_receiver_cc_sptr.**omega**(*mpsk_receiver_cc_sptr self*) → float

Returns current value of omega.

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

pc_input_buffers_full_avg(mpsk_receiver_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(mpsk_receiver_cc_sptr self) -> pmt_vector_float

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

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

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

mpsk_receiver_cc_sptr.**phase_wrap**(*mpsk_receiver_cc_sptr self*)

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

mpsk_receiver_cc_sptr.**set_alpha**(*mpsk_receiver_cc_sptr self*, *float alpha*)

mpsk_receiver_cc_sptr.**set_beta**(*mpsk_receiver_cc_sptr self*, *float beta*)

mpsk_receiver_cc_sptr.**set_damping_factor**(*mpsk_receiver_cc_sptr self*, *float df*)

mpsk_receiver_cc_sptr.**set_frequency**(*mpsk_receiver_cc_sptr self*, *float freq*)

mpsk_receiver_cc_sptr.**set_gain_mu**(*mpsk_receiver_cc_sptr self*, *float gain_mu*)

Sets value for mu gain factor.

mpsk_receiver_cc_sptr.**set_gain_omega**(*mpsk_receiver_cc_sptr self*, *float gain_omega*)

Sets value for omega gain factor.

mpsk_receiver_cc_sptr.**set_gain_omega_rel**(*mpsk_receiver_cc_sptr self*, *float omega_rel*)

Sets the relative omega limit and resets omega min/max values.

mpsk_receiver_cc_sptr.**set_loop_bandwidth**(*mpsk_receiver_cc_sptr self*, *float bw*)

mpsk_receiver_cc_sptr.**set_max_freq**(*mpsk_receiver_cc_sptr self*, *float freq*)

mpsk_receiver_cc_sptr.**set_min_freq**(*mpsk_receiver_cc_sptr self*, *float freq*)

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

mpsk_receiver_cc_sptr.**set_modulation_order**(*mpsk_receiver_cc_sptr self*, *unsigned int M*)

Sets the modulation order (M) currently.

mpsk_receiver_cc_sptr.**set_mu**(*mpsk_receiver_cc_sptr self*, *float mu*)

Sets value of mu.

mpsk_receiver_cc_sptr.**set_omega**(*mpsk_receiver_cc_sptr self*, *float omega*)

Sets value of omega and its min and max values.

mpsk_receiver_cc_sptr.**set_phase**(*mpsk_receiver_cc_sptr self*, *float phase*)

mpsk_receiver_cc_sptr.**set_theta**(*mpsk_receiver_cc_sptr self*, *float theta*)

Sets value of theta.

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

mpsk_receiver_cc_sptr.**theta**(*mpsk_receiver_cc_sptr self*) → float

Returns current value of theta.

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

mpsk_receiver_cc_sptr.**update_gains**(*mpsk_receiver_cc_sptr self*)

gnuradio.digital.**mpsk_snr_est_cc**(*gr::digital::snr_est_type_t type*, *int tag_nsamples=10000*, *double alpha=0.001*) → mpsk_snr_est_cc_sptr

A block for computing SNR of a signal.

This block can be used to monitor and retrieve estimations of the signal SNR. It is designed to work in a flowgraph and passes all incoming data along to its output.

The block is designed for use with M-PSK signals especially. The type of estimator is specified as the parameter in the constructor. The estimators tend to trade off performance for accuracy, although experimentation should be done to figure out the right approach for a given implementation. Further, the current set of estimators are designed and proven theoretically under AWGN conditions; some amount of error should be assumed and/or estimated for real channel conditions.

Constructor Specific Documentation:

Factory function returning shared pointer of this class

**Parameters:**

- **type** – the type of estimator to use gr::digital::snr_est_type_t “snr_est_type_t” for details about the available types
- **tag_nsamples** – after this many samples, a tag containing the SNR (key=’snr’) will be sent
- **alpha** – the update rate of internal running average calculations

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

mpsk_snr_est_cc_sptr.**alpha**(*mpsk_snr_est_cc_sptr self*) → double

Get the running-average coefficient.

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

declare_sample_delay(mpsk_snr_est_cc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(mpsk_snr_est_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(mpsk_snr_est_cc_sptr self) -> pmt_vector_float

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

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

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

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

mpsk_snr_est_cc_sptr.**set_alpha**(*mpsk_snr_est_cc_sptr self*, *double alpha*)

Set the running-average coefficient.

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

mpsk_snr_est_cc_sptr.**set_tag_nsample**(*mpsk_snr_est_cc_sptr self*, *int n*)

Set the number of samples between SNR tags.

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

mpsk_snr_est_cc_sptr.**set_type**(*mpsk_snr_est_cc_sptr self*, *gr::digital::snr_est_type_t t*)

Set type of estimator to use.

mpsk_snr_est_cc_sptr.**snr**(*mpsk_snr_est_cc_sptr self*) → double

Return the estimated signal-to-noise ratio in decibels.

mpsk_snr_est_cc_sptr.**tag_nsample**(*mpsk_snr_est_cc_sptr self*) → int

Return how many samples between SNR tags.

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

gnuradio.digital.**msk_timing_recovery_cc**(*float sps*, *float gain*, *float limit*, *int osps*) → msk_timing_recovery_cc_sptr

MSK/GMSK timing recovery

This block performs timing synchronization on CPM modulations using a fourth-order nonlinearity feedback method which is non-data-aided. The block does not require prior phase synchronization but is relatively sensitive to frequency offset (keep offset to 0.1x symbol rate).

For details on the algorithm, see: A.N. D'Andrea, U. Mengali, R. Reggiannini: A digital approach to clock recovery in generalized minimum shift keying. IEEE Transactions on Vehicular Technology, Vol. 39, Issue 3.

Constructor Specific Documentation:

Make an MSK timing recovery block.

**Parameters:**
- **sps** – Samples per symbol
- **gain** – Loop gain of timing error filter (try 0.05)
- **limit** – Relative limit of timing error (try 0.1 for 10% error max)
- **osps** – Output samples per symbol

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

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

declare_sample_delay(msk_timing_recovery_cc_sptr self, unsigned int delay)

msk_timing_recovery_cc_sptr.**get_gain**(*msk_timing_recovery_cc_sptr self*) → float

msk_timing_recovery_cc_sptr.**get_limit**(*msk_timing_recovery_cc_sptr self*) → float

msk_timing_recovery_cc_sptr.**get_sps**(*msk_timing_recovery_cc_sptr self*) → float

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

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

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

pc_input_buffers_full_avg(msk_timing_recovery_cc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(msk_timing_recovery_cc_sptr self) -> pmt_vector_float

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

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

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

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

msk_timing_recovery_cc_sptr.**set_gain**(*msk_timing_recovery_cc_sptr self*, *float gain*)

msk_timing_recovery_cc_sptr.**set_limit**(*msk_timing_recovery_cc_sptr self*, *float limit*)

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

msk_timing_recovery_cc_sptr.**set_sps**(*msk_timing_recovery_cc_sptr self*, *float sps*)

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

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

gnuradio.digital.**ofdm_carrier_allocator_cvc**(*int fft_len*, *std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & occupied_carriers*, *std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & pilot_carriers*, *gr_vector_vector_complexf pilot_symbols*, *gr_vector_vector_complexf sync_words*, *std::string const & len_tag_key*, *bool const output_is_shifted=True*) → ofdm_carrier_allocator_cvc_sptr

Create frequency domain OFDM symbols from complex values, add pilots.

This block turns a stream of complex, scalar modulation symbols into vectors which are the input for an IFFT in an OFDM transmitter. It also supports the possibility of placing pilot symbols onto the carriers.

The carriers can be allocated freely, if a carrier is not allocated, it is set to zero. This allows doing OFDMA-style carrier allocations.

Input: A tagged stream of complex scalars. The first item must have a tag containing the number of complex symbols in this frame. Output: A tagged stream of complex vectors of length fft_len. This can directly be connected to an FFT block. Make sure to set this block to 'reverse' for the IFFT. Ifis true, the FFT block must activate FFT shifting, otherwise, set shifting to false. If given, sync words are prepended to the output. Note that sync words are prepended verbatim, make sure they are shifted (or not).

Carrier indexes are always such that index 0 is the DC carrier (note: you should not allocate this carrier). The carriers below the DC carrier are either indexed with negative numbers, or with indexes larger than . Index -1 and index both identify the carrier below the DC carrier.

There are some basic checks in place during initialization which check that the carrier allocation table is valid. However, it is possible to overwrite data symbols with pilot symbols, or provide a carrier allocation that has mismatching pilot symbol positions and -values.

Tags are propagated such that a tag on an incoming complex symbol is mapped to the corresponding OFDM symbol. There is one exception: If a tag is on the first OFDM symbol, it is assumed that this tag should stay there, so it is moved to the front even if a sync word is included (any other tags will never be attached to the sync word). This allows tags to control the transmit timing to pass through in the correct position.

Constructor Specific Documentation:

**Parameters:**
- **fft_len** –
- **occupied_carriers** –
- **pilot_carriers** –
- **pilot_symbols** –
- **sync_words** –
- **len_tag_key** –
- **output_is_shifted** –

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

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

declare_sample_delay(ofdm_carrier_allocator_cvc_sptr self, unsigned int delay)

ofdm_carrier_allocator_cvc_sptr.**fft_len**(*ofdm_carrier_allocator_cvc_sptr self*) → int const

ofdm_carrier_allocator_cvc_sptr.**len_tag_key**(*ofdm_carrier_allocator_cvc_sptr self*) → std::string

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

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

ofdm_carrier_allocator_cvc_sptr.**occupied_carriers**(*ofdm_carrier_allocator_cvc_sptr self*) → std::vector< std::vector< int,std::allocator< int > >,std::allocator< std::vector< int,std::allocator< int > > > >

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

pc_input_buffers_full_avg(ofdm_carrier_allocator_cvc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_carrier_allocator_cvc_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_chanest_vcvc**(*pmt_vector_cfloat sync_symbol1*, *pmt_vector_cfloat sync_symbol2*, *int n_data_symbols*, *int eq_noise_red_len=0*, *int max_carr_offset=-1*, *bool force_one_sync_symbol=False*) → ofdm_chanest_vcvc_sptr

Estimate channel and coarse frequency offset for OFDM from preambles

Input: OFDM symbols (in frequency domain). The first one (or two) symbols are expected to be synchronisation symbols, which are used to estimate the coarse freq offset and the initial equalizer taps (these symbols are removed from the stream). The following are passed through unmodified (the actual equalisation must be done elsewhere). Output: The data symbols, without the synchronisation symbols. The first data symbol passed through has two tags: 'ofdm_sync_carr_offset' (integer), the coarse frequency offset as number of carriers, and 'ofdm_sync_eq_taps' (complex vector). Any tags attached to the synchronisation symbols are attached to the first data symbol. All other tags are propagated as expected.

**Note:** The vector on ofdm_sync_eq_taps is already frequency-corrected, whereas the rest is not.

This block assumes the frequency offset is even (i.e. an integer multiple of 2).

[1] Schmidl, T.M. and Cox, D.C., "Robust frequency and timing synchronization for OFDM", Communications, IEEE Transactions on, 1997. [2] K.D. Kammeyer, "Nachrichtenuebertragung," Chapter. 16.3.2.

Constructor Specific Documentation:

**Parameters:**

- **sync_symbol1** – First synchronisation symbol in the frequency domain. Its length must be the FFT length. For Schmidl & Cox synchronisation, every second sub-carrier has to be zero.
- **sync_symbol2** – Second synchronisation symbol in the frequency domain. Must be equal to the FFT length, or zero length if only one synchronisation symbol is used. Using this symbol is how synchronisation is described in [1]. Leaving this empty forces us to interpolate the equalizer taps. If you are using an unusual sub-carrier configuration (e.g. because of OFDMA), this sync symbol is used to identify the active sub-carriers. If you only have one synchronisation symbol, set the active sub-carriers to a non-zero value in here, and also set parameter to true.
- **n_data_symbols** – The number of data symbols following each set of synchronisation symbols. Must be at least 1.
- **eq_noise_red_len** – If non-zero, noise reduction for the equalizer taps is done according to [2]. In this case, it is the channel influence time in number of samples. A good value is usually the length of the cyclic prefix.
- **max_carr_offset** – Limit the number of sub-carriers the frequency offset can maximally be. Leave this zero to try all possibilities.
- **force_one_sync_symbol** – See .

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

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

declare_sample_delay(ofdm_chanest_vcvc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_chanest_vcvc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_chanest_vcvc_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_cyclic_prefixer**(*size_t input_size*, *size_t output_size*, *int rolloff_len=0*, *std::string const & len_tag_key*) → ofdm_cyclic_prefixer_sptr

Adds a cyclic prefix and performs pulse shaping on OFDM symbols.

Input: OFDM symbols (in the time domain, i.e. after the IFFT). Optionally, entire frames can be processed. In this case, must be specified which holds the key of the tag that denotes how many OFDM symbols are in a frame. Output: A stream of (scalar) complex symbols, which include the cyclic prefix and the pulse shaping. Note: If complete frames are processed, and is greater than zero, the final OFDM symbol is followed by the delay line of the pulse shaping.

The pulse shape is a raised cosine in the time domain.

Constructor Specific Documentation:

**Parameters:**
- **input_size** – FFT length (i.e. length of the OFDM symbols)
- **output_size** – FFT length + cyclic prefix length (in samples)
- **rolloff_len** – Length of the rolloff flank in samples
- **len_tag_key** – For framed processing the key of the length tag

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

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

declare_sample_delay(ofdm_cyclic_prefixer_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_cyclic_prefixer_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_cyclic_prefixer_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_equalizer_base**(**args*, ***kwargs*)

Base class for implementation details of frequency-domain OFDM equalizers.

ofdm_equalizer_base_sptr.**base**(*ofdm_equalizer_base_sptr self*) → ofdm_equalizer_base_sptr

ofdm_equalizer_base_sptr.**equalize**(*ofdm_equalizer_base_sptr self*, *gr_complex * frame*, *int n_sym*, *pmt_vector_cfloat initial_taps*, *tags_vector_t tags*)

Run the actual equalization.

ofdm_equalizer_base_sptr.**fft_len**(*ofdm_equalizer_base_sptr self*) → int

ofdm_equalizer_base_sptr.**get_channel_state**(*ofdm_equalizer_base_sptr self*, *pmt_vector_cfloat taps*)

Return the current channel state.

ofdm_equalizer_base_sptr.**reset**(*ofdm_equalizer_base_sptr self*)

Reset the channel information state knowledge.

gnuradio.digital.**ofdm_equalizer_simpledfe**(*int fft_len*, *constellation_sptr constellation*, *std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & occupied_carriers*, *std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & pilot_carriers*, *gr_vector_vector_complexf pilot_symbols*, *int symbols_skipped=0*, *float alpha=0.1*, *bool input_is_shifted=True*) → ofdm_equalizer_simpledfe_sptr

Simple decision feedback equalizer for OFDM.

Equalizes an OFDM signal symbol by symbol using knowledge of the complex modulations symbols. For every symbol, the following steps are performed:

This equalizer makes a lot of assumptions:

Note that the equalized symbols are on the constellation. This means soft information of the modulation symbols is lost after the equalization, which is suboptimal for channel codes that use soft decision.

Constructor Specific Documentation:

**Parameters:**
- **fft_len** –
- **constellation** –
- **occupied_carriers** –
- **pilot_carriers** –
- **pilot_symbols** –
- **symbols_skipped** –
- **alpha** –
- **input_is_shifted** –

ofdm_equalizer_simpledfe_sptr.**base**(*ofdm_equalizer_simpledfe_sptr self*) → ofdm_equalizer_base_sptr

ofdm_equalizer_simpledfe_sptr.**equalize**(*ofdm_equalizer_simpledfe_sptr self*, *gr_complex * frame*, *int n_sym*, *pmt_vector_cfloat initial_taps*, *tags_vector_t tags*)

Run the actual equalization.

ofdm_equalizer_simpledfe_sptr.**fft_len**(*ofdm_equalizer_simpledfe_sptr self*) → int

ofdm_equalizer_simpledfe_sptr.**get_channel_state**(*ofdm_equalizer_simpledfe_sptr self*, *pmt_vector_cfloat taps*)

Return the current channel state.

ofdm_equalizer_simpledfe_sptr.**reset**(*ofdm_equalizer_simpledfe_sptr self*)

Reset the channel information state knowledge.

gnuradio.digital.**ofdm_equalizer_static**(*int fft_len*, *std::vector< std::vector< int, std::allocator< int > >,*

*std::allocator< std::vector< int, std::allocator< int > > > > const & occupied_carriers, std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & pilot_carriers, gr_vector_vector_complexf pilot_symbols, int symbols_skipped=0, bool input_is_shifted=True*) → ofdm_equalizer_static_sptr

Very simple static equalizer for OFDM.

This is an extremely simple equalizer. It will only work for high-SNR, very, very slowly changing channels.

It simply divides the signal with the currently known channel state. Whenever a pilot symbol comes around, it updates the channel state on that particular carrier by dividing the received symbol with the known pilot symbol.

Constructor Specific Documentation:

**Parameters:**
- **fft_len** –
- **occupied_carriers** –
- **pilot_carriers** –
- **pilot_symbols** –
- **symbols_skipped** –
- **input_is_shifted** –

ofdm_equalizer_static_sptr.**base**(*ofdm_equalizer_static_sptr self*) → ofdm_equalizer_base_sptr

ofdm_equalizer_static_sptr.**equalize**(*ofdm_equalizer_static_sptr self*, *gr_complex * frame*, *int n_sym*, *pmt_vector_cfloat initial_taps*, *tags_vector_t tags*)

Run the actual equalization.

ofdm_equalizer_static_sptr.**fft_len**(*ofdm_equalizer_static_sptr self*) → int

ofdm_equalizer_static_sptr.**get_channel_state**(*ofdm_equalizer_static_sptr self*, *pmt_vector_cfloat taps*)

Return the current channel state.

ofdm_equalizer_static_sptr.**reset**(*ofdm_equalizer_static_sptr self*)

Reset the channel information state knowledge.

gnuradio.digital.**ofdm_frame_acquisition**(*unsigned int occupied_carriers*, *unsigned int fft_length*, *unsigned int cplen*, *pmt_vector_cfloat known_symbol*, *unsigned int max_fft_shift_len=4*) → ofdm_frame_acquisition_sptr

take a vector of complex constellation points in from an FFT and performs a correlation and equalization.

This block takes the output of an FFT of a received OFDM symbol and finds the start of a frame based on two known symbols. It also looks at the surrounding bins in the FFT output for the correlation in case there is a large frequency shift in the data. This block assumes that the fine frequency shift has already been corrected and that the samples fall in the middle of one FFT bin.

It then uses one of those known symbols to estimate the channel response over all subcarriers and does a simple 1-tap equalization on all subcarriers. This corrects for the phase and amplitude distortion caused by the channel.

Constructor Specific Documentation:

Make an OFDM correlator and equalizer.

**Parameters:**
- **occupied_carriers** – The number of subcarriers with data in the received symbol
- **fft_length** – The size of the FFT vector (occupied_carriers + unused carriers)
- **cplen** – The length of the cycle prefix
- **known_symbol** – A vector of complex numbers representing a known symbol at the start of a frame (usually a BPSK PN sequence)
- **max_fft_shift_len** – Set's the maximum distance you can look between bins for correlation

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

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

declare_sample_delay(ofdm_frame_acquisition_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_frame_acquisition_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_frame_acquisition_sptr self) -> pmt_vector_float

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

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

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

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

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

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

ofdm_frame_acquisition_sptr.**snr**(*ofdm_frame_acquisition_sptr self*) → float

Return an estimate of the SNR of the channel.

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

gnuradio.digital.**ofdm_frame_equalizer_vcvc**(*ofdm_equalizer_base_sptr equalizer*, *int cp_len*, *std::string const & tsb_key*, *bool propagate_channel_state=False*, *int fixed_frame_len=0*) → ofdm_frame_equalizer_vcvc_sptr

OFDM frame equalizer.

Performs equalization in one or two dimensions on a tagged OFDM frame.

This does two things: First, it removes the coarse carrier offset. If a tag is found on the first item with the key 'ofdm_sync_carr_offset', this is interpreted as the coarse frequency offset in number of carriers. Next, it performs equalization in one or two dimensions on a tagged OFDM frame. The actual equalization is done by a ofdm_frame_equalizer object, outside of the block.

Note that the tag with the coarse carrier offset is not removed. Blocks downstream from this block must not attempt to also correct this offset.

Input: a tagged series of OFDM symbols. Output: The same as the input, but equalized and frequency-corrected.

Constructor Specific Documentation:

**Parameters:**

- **equalizer** – The equalizer object that will do the actual work
- **cp_len** – Length of the cyclic prefix in samples (required to correct the frequency offset)
- **tsb_key** – TSB key
- **propagate_channel_state** – If true, the channel state after the last symbol will be added to the first symbol as a tag
- **fixed_frame_len** – Set if the frame length is fixed. When this value is given, the TSB tag key can be left empty, but it is useful even when using tagged streams at the input.

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

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

declare_sample_delay(ofdm_frame_equalizer_vcvc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_frame_equalizer_vcvc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_frame_equalizer_vcvc_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_frame_sink**(*pmt_vector_cfloat sym_position*, *std::vector< char, std::allocator< char > > const & sym_value_out*, *msg_queue_sptr target_queue*, *int occupied_tones*, *float phase_gain=0.25*, *float freq_gain=0*) → ofdm_frame_sink_sptr

Takes an OFDM symbol in, demaps it into bits of 0's and 1's, packs them into packets, and sends to to a message queue sink.

NOTE: The mod input parameter simply chooses a pre-defined demapper/slicer. Eventually, we want to be able to pass in a reference to an object to do the demapping and slicing for a given modulation type.

Constructor Specific Documentation:

Make an OFDM frame sink block.

**Parameters:**
- **sym_position** – vector of OFDM carrier symbols in complex space
- **sym_value_out** – vector of bit mapped from the complex symbol space
- **target_queue** – message queue for the packets to go into
- **occupied_tones** – The number of subcarriers with data in the received symbol
- **phase_gain** – gain of the phase tracking loop
- **freq_gain** – gain of the frequency tracking loop

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

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

declare_sample_delay(ofdm_frame_sink_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_frame_sink_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_frame_sink_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_insert_preamble**(*int fft_length*, *gr_vector_vector_complexf preamble*) → ofdm_insert_preamble_sptr

insert “pre-modulated” preamble symbols before each payload.

Constructor Specific Documentation:

Make an OFDM preamble inserter block.

**Parameters:**
- **fft_length** – length of each symbol in samples.
- **preamble** – vector of symbols that represent the pre-modulated preamble.

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

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

declare_sample_delay(ofdm_insert_preamble_sptr self, unsigned int delay)

ofdm_insert_preamble_sptr.**enter_preamble**(*ofdm_insert_preamble_sptr self*)

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

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

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

pc_input_buffers_full_avg(ofdm_insert_preamble_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_insert_preamble_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_mapper_bcv**(*pmt_vector_cfloat constellation*, *unsigned int msgq_limit*, *unsigned int occupied_carriers*, *unsigned int fft_length*) → ofdm_mapper_bcv_sptr

take a stream of bytes in and map to a vector of complex constellation points suitable for IFFT input to be used in an ofdm modulator.

Abstract class must be subclassed with specific mapping.

Constructor Specific Documentation:

Make an OFDM mapper block.

**Parameters:**
- **constellation** – vector of OFDM carrier symbols in complex space
- **msgq_limit** – limit on number of messages the queue can store
- **occupied_carriers** – The number of subcarriers with data in the received symbol
- **fft_length** – The size of the FFT vector (occupied_carriers + unused carriers)

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

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

declare_sample_delay(ofdm_mapper_bcv_sptr self, unsigned int delay)

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

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

ofdm_mapper_bcv_sptr.**msgq**(*ofdm_mapper_bcv_sptr self*) → msg_queue_sptr

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

pc_input_buffers_full_avg(ofdm_mapper_bcv_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_mapper_bcv_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_sampler**(*unsigned int fft_length*, *unsigned int symbol_length*, *unsigned int timeout=1000*) → ofdm_sampler_sptr

does the rest of the OFDM stuff

Constructor Specific Documentation:

Make an OFDM sampler block.

**Parameters:**
- **fft_length** – The size of the FFT vector (occupied_carriers + unused carriers)
- **symbol_length** – Length of the full symbol (fft_length + CP length)
- **timeout** – timeout in samples when we stop looking for a symbol after initial ack.

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

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

declare_sample_delay(ofdm_sampler_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_sampler_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_sampler_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_serializer_vcc**(*int fft_len*, *std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & occupied_carriers*, *std::string const &*

*len_tag_key*, *std::string const & packet_len_tag_key*, *int symbols_skipped=0*, *std::string const & carr_offset_key*, *bool input_is_shifted=True*) → ofdm_serializer_vcc_sptr

make(ofdm_carrier_allocator_cvc_sptr allocator, std::string const & packet_len_tag_key, int symbols_skipped=0, std::string const & carr_offset_key, bool input_is_shifted=True) -> ofdm_serializer_vcc_sptr

Serializes complex modulations symbols from OFDM sub-carriers.

This is the inverse block to the carrier_allocator_cvc. It outputs the complex data symbols as a tagged stream, discarding the pilot symbols.

If given, two different tags are parsed: The first key () specifies the number of OFDM symbols in the frame at the input. The second key () specifies the number of complex symbols that are coded into this frame. If given, this second key is then used at the output, otherwise, is used. If both are given, the packet length specifies the maximum number of output items, and the frame length specifies the exact number of consumed input items.

It is possible to correct a carrier offset in this function by passing another tag with said offset.

Input: Complex vectors of length Output: Complex scalars, in the same order as specified in occupied_carriers.

Constructor Specific Documentation:

**Parameters:**

- **fft_len** – FFT length
- **occupied_carriers** – See ofdm_carrier_allocator_cvc.
- **len_tag_key** – The key of the tag identifying the length of the input frame in OFDM symbols.
- **packet_len_tag_key** – The key of the tag identifying the number of complex symbols in this packet.
- **symbols_skipped** – If the first symbol is not allocated as in [0], set this
- **carr_offset_key** – When this block should correct a carrier offset, specify the tag key of the offset here (not necessary if following an ofdm_frame_equalizer_vcvc)
- **input_is_shifted** – If the input has the DC carrier on index 0 (i.e. it is not FFT shifted), set this to false

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

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

declare_sample_delay(ofdm_serializer_vcc_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(ofdm_serializer_vcc_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(ofdm_serializer_vcc_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**ofdm_sync_sc_cfb**(*int fft_len*, *int cp_len*, *bool use_even_carriers=False*, *float threshold=0.9*) → ofdm_sync_sc_cfb_sptr

Schmidl & Cox synchronisation for OFDM.

Input: complex samples. Output 0: Fine frequency offset, scaled by the OFDM symbol duration. This is in [1]. The normalized frequency offset is then 2.0*output0/fft_len. Output 1: Beginning of the first OFDM symbol after the first (doubled) OFDM symbol. The beginning is marked with a 1 (it's 0 everywhere else).

The evaluation of the coarse frequency offset is done in this block. Also, the initial equalizer taps are not calculated here.

Note that we use a different normalization factor in the timing metric than the authors do in their original work[1]. If the timing metric (8) is we calculate the normalization as i.e., we estimate the energy from half-symbols. This avoids spurious detects at the end of a burst, when the energy level suddenly drops.

[1] Schmidl, T.M. and Cox, D.C., "Robust frequency and timing synchronization for OFDM", Communications, IEEE Transactions on, 1997.

Constructor Specific Documentation:

**Parameters:**
- **fft_len** – FFT length
- **cp_len** – Length of the guard interval (cyclic prefix) in samples
- **use_even_carriers** – If true, the carriers in the sync preamble are occupied such that the even carriers are used (0, 2, 4, ...). If you use all carriers, that would include the DC carrier, so be careful.
- **threshold** – detection threshold. Default is 0.9.

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

ofdm_sync_sc_cfb_sptr.**set_threshold**(*ofdm_sync_sc_cfb_sptr self*, *float threshold*)

ofdm_sync_sc_cfb_sptr.**threshold**(*ofdm_sync_sc_cfb_sptr self*) → float

gnuradio.digital.**packet_header_default**(*long header_len*, *std::string const & len_tag_key*, *std::string const & num_tag_key*, *int bits_per_byte=1*) → packet_header_default_sptr

Default header formatter for digital packet transmission.

For bursty/packetized digital transmission, packets are usually prepended with a packet header, containing the number of bytes etc. This class is not a block, but a tool to create these packet header.

This is a default packet header (see header_formatter()) for a description on the header format). To create other header, derive packet header creator classes from this function.

gr::digital::packet_headergenerator_bb uses header generators derived from this class to create packet headers from data streams.

Constructor Specific Documentation:

**Parameters:**
- **header_len** –
- **len_tag_key** –
- **num_tag_key** –
- **bits_per_byte** –

packet_header_default_sptr.**base**(*packet_header_default_sptr self*) → packet_header_default_sptr

packet_header_default_sptr.**formatter**(*packet_header_default_sptr self*) → packet_header_default_sptr

packet_header_default_sptr.**header_formatter**(*packet_header_default_sptr self*, *long packet_len*, *unsigned char * out*, *tags_vector_t tags*) → bool

Encodes the header information in the given tags into bits and places them into .

Uses the following header format: Bits 0-11: The packet length (what was stored in the tag with key ) Bits 12-23: The header number (counts up everytime this function is called) Bit 24-31: 8-Bit CRC All other bits: Are set to zero

If the header length is smaller than 32, bits are simply left out. For this reason, they always start with the LSB.

However, it is recommended to stay above 32 Bits, in order to have a working CRC.

packet_header_default_sptr.**header_len**(*packet_header_default_sptr self*) → long

packet_header_default_sptr.**header_parser**(*packet_header_default_sptr self*, *unsigned char const * header*, *tags_vector_t tags*) → bool

Inverse function to header_formatter().

Reads the bit stream in and writes a corresponding tag into .

packet_header_default_sptr.**len_tag_key**(*packet_header_default_sptr self*) → swig_int_ptr

packet_header_default_sptr.**set_header_num**(*packet_header_default_sptr self*, *unsigned int header_num*)

gnuradio.digital.**packet_headergenerator_bb**(*packet_header_default_sptr header_formatter*, *std::string const & len_tag_key*) → packet_headergenerator_bb_sptr

make(long header_len, std::string const & len_tag_key) -> packet_headergenerator_bb_sptr

Generates a header for a tagged, streamed packet.

Input: A tagged stream. This is consumed entirely, it is not appended to the output stream. Output: An tagged stream containing the header. The details on the header are set in a header formatter object (of type packet_header_default or a subclass thereof). If only a number of bits is specified, a default header is generated (see packet_header_default).

Constructor Specific Documentation:

**Parameters:**
- **header_formatter** –
- **len_tag_key** –

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

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

declare_sample_delay(packet_headergenerator_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(packet_headergenerator_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(packet_headergenerator_bb_sptr self) -> pmt_vector_float

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

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

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

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

packet_headergenerator_bb_sptr.**set_header_formatter**(*packet_headergenerator_bb_sptr self*, *packet_header_default_sptr header_formatter*)

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

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

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

gnuradio.digital.**packet_header_ofdm**(*std::vector< std::vector< int, std::allocator< int > >, std::allocator< std::vector< int, std::allocator< int > > > > const & occupied_carriers*, *int n_syms*, *std::string const & len_tag_key*, *std::string const & frame_len_tag_key*, *std::string const & num_tag_key*, *int bits_per_header_sym=1*, *int bits_per_payload_sym=1*, *bool scramble_header=False*) → packet_header_ofdm_sptr

Header utility for OFDM signals.

Constructor Specific Documentation:

**Parameters:**
- **occupied_carriers** – See carrier allocator
- **n_syms** – The number of OFDM symbols the header should be (usually 1)
- **len_tag_key** – The tag key used for the packet length (number of bytes)
- **frame_len_tag_key** – The tag key used for the frame length (number of OFDM symbols, this is the tag key required for the frame equalizer etc.)
- **num_tag_key** – The tag key used for packet numbering.
- **bits_per_header_sym** – Bits per complex symbol in the header, e.g. 1 if the header is BPSK modulated, 2 if it's QPSK modulated etc.
- **bits_per_payload_sym** – Bits per complex symbol in the payload. This is required to figure out how many OFDM symbols are necessary to encode the given number of bytes.
- **scramble_header** – Set this to true to scramble the bits. This is highly recommended, as it reduces PAPR spikes.

packet_header_ofdm_sptr.**base**(*packet_header_ofdm_sptr self*) → packet_header_default_sptr

packet_header_ofdm_sptr.**formatter**(*packet_header_ofdm_sptr self*) → packet_header_default_sptr

packet_header_ofdm_sptr.**header_formatter**(*packet_header_ofdm_sptr self*, *long packet_len*, *unsigned char * out*, *tags_vector_t tags*) → bool

Encodes the header information in the given tags into bits and places them into .

Uses the following header format: Bits 0-11: The packet length (what was stored in the tag with key ) Bits 12-23: The header number (counts up everytime this function is called) Bit 24-31: 8-Bit CRC All other bits: Are set to zero

If the header length is smaller than 32, bits are simply left out. For this reason, they always start with the LSB.

However, it is recommended to stay above 32 Bits, in order to have a working CRC.

packet_header_ofdm_sptr.**header_len**(*packet_header_ofdm_sptr self*) → long

packet_header_ofdm_sptr.**header_parser**(*packet_header_ofdm_sptr self*, *unsigned char const * header*, *tags_vector_t tags*) → bool

Inverse function to header_formatter().

Reads the bit stream in and writes a corresponding tag into .

packet_header_ofdm_sptr.**len_tag_key**(*packet_header_ofdm_sptr self*) → swig_int_ptr

packet_header_ofdm_sptr.**set_header_num**(*packet_header_ofdm_sptr self*, *unsigned int header_num*)

gnuradio.digital.**packet_headerparser_b**(*packet_header_default_sptr header_formatter*) → packet_headerparser_b_sptr

make(long header_len, std::string const & len_tag_key) -> packet_headerparser_b_sptr

Post header metadata as a PMT.

In a sense, this is the inverse block to packet_headergenerator_bb. The difference is, the parsed header is not output as a stream, but as a PMT dictionary, which is published to message port with the id "header_data".

The dictionary consists of the tags created by the header formatter object. You should be able to use the exact same formatter object as used on the Tx side in the packet_headergenerator_bb.

If only a header length is given, this block uses the default header format.

Constructor Specific Documentation:

**Parameters:** **header_formatter** – Header object. This should be the same as used for packet_headergenerator_bb.

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

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

declare_sample_delay(packet_headerparser_b_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(packet_headerparser_b_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(packet_headerparser_b_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**packet_sink**(*std::vector< unsigned char, std::allocator< unsigned char > > const & sync_vector*, *msg_queue_sptr target_queue*, *int threshold=-1*) → packet_sink_sptr

process received bits looking for packet sync, header, and process bits into packet

input: stream of symbols to be sliced.

output: none. Pushes assembled packet into target queue

The packet sink takes in a stream of binary symbols that are sliced around 0. The bits are then checked for the to determine find and decode the packet. It then expects a fixed length header of 2 16-bit shorts containing the payload length, followed by the payload. If the 2 16-bit shorts are not identical, this packet is ignored. Better algs are welcome.

This block is not very useful anymore as it only works with 2-level modulations such as BPSK or GMSK. The block can generally be replaced with a correlate access code and frame sink blocks.

Constructor Specific Documentation:

Make a packet_sink block.

**Parameters:**
- **sync_vector** – The synchronization vector as a vector of 1's and 0's.
- **target_queue** – The message queue that packets are sent to.
- **threshold** – Number of bits that can be incorrect in the .

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

packet_sink_sptr.**carrier_sensed**(*packet_sink_sptr self*) → bool

return true if we detect carrier

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

declare_sample_delay(packet_sink_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(packet_sink_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(packet_sink_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.digital.**pfb_clock_sync_ccf**(*double sps*, *float loop_bw*, *pmt_vector_float taps*, *unsigned int filter_size=32*, *float init_phase=0*, *float max_rate_deviation=1.5*, *int osps=1*) → pfb_clock_sync_ccf_sptr

Timing synchronizer using polyphase filterbanks.

This block performs timing synchronization for PAM signals by minimizing the derivative of the filtered signal, which in turn maximizes the SNR and minimizes ISI.

This approach works by setting up two filterbanks; one filterbank contains the signal's pulse shaping matched filter (such as a root raised cosine filter), where each branch of the filterbank contains a different phase of the filter. The second filterbank contains the derivatives of the filters in the first filterbank. Thinking of this in the time domain, the first filterbank contains filters that have a sinc shape to them. We want to align the output signal to be sampled at exactly the peak of the sinc shape. The derivative of the sinc contains a zero at the maximum point of the sinc (sinc(0) = 1, sinc(0)' = 0). Furthermore, the region around the zero point is relatively linear. We make use of this fact to generate the error signal.

If the signal out of the derivative filters is d_i[n] for the ith filter, and the output of the matched filter is x_i[n], we calculate the error as: e[n] = (Re{x_i[n]} * Re{d_i[n]} + Im{x_i[n]} * Im{d_i[n]}) / 2.0 This equation averages the error in the real and imaginary parts. There are two reasons we multiply by the signal itself. First, if the symbol could be positive or negative going, but we want the error term to always tell us to go in the same direction depending on which side of the zero point we are on. The sign of x_i[n] adjusts the error term to do this. Second, the magnitude of x_i[n] scales the error term depending on the symbol's amplitude, so larger signals give us a stronger error term because we have more confidence in that symbol's value. Using the magnitude of x_i[n] instead of just the sign is especially good for signals with low SNR.

The error signal, e[n], gives us a value proportional to how far away from the zero point we are in the derivative signal. We want to drive this value to zero, so we set up a second order loop. We have two variables for this loop; d_k is the filter number in the filterbank we are on and d_rate is the rate which we travel through the filters in the steady state. That is, due to the natural clock differences between the transmitter and receiver, d_rate represents that difference and would traverse the filter phase paths to keep the receiver locked. Thinking of this as a second-order PLL, the d_rate is the frequency and d_k is the phase. So we update d_rate and d_k using the standard loop equations based on two error signals, d_alpha and d_beta. We have these two values set based on each other for a critically damped system, so in the block constructor, we just ask for "gain," which is d_alpha while d_beta is equal to (gain^2)/4.

The block's parameters are:

Reference: f. j. harris and M. Rice, "Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios", IEEE Selected Areas in Communications, Vol. 19, No. 12, Dec., 2001.

Constructor Specific Documentation:

Build the polyphase filterbank timing synchronizer.

**Parameters:**
- **sps** – (double) The number of samples per symbol in the incoming signal
- **loop_bw** – (float) The bandwidth of the control loop; set's alpha and beta.
- **taps** – (vector<int>) The filter taps.
- **filter_size** – (uint) The number of filters in the filterbank (default = 32).
- **init_phase** – (float) The initial phase to look at, or which filter to start with (default = 0).
- **max_rate_deviation** – (float) Distance from 0 d_rate can get (default = 1.5).
- **osps** – (int) The number of output samples per symbol (default=1).

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

pfb_clock_sync_ccf_sptr.**alpha**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the loop gain alpha.

pfb_clock_sync_ccf_sptr.**beta**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the loop gain beta.

pfb_clock_sync_ccf_sptr.**channel_taps**(*pfb_clock_sync_ccf_sptr self*, *int channel*) → pmt_vector_float

Returns the taps of the matched filter for a particular channel

pfb_clock_sync_ccf_sptr.**clock_rate**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the current clock rate.

pfb_clock_sync_ccf_sptr.**damping_factor**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the loop damping factor.

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

declare_sample_delay(pfb_clock_sync_ccf_sptr self, unsigned int delay)

pfb_clock_sync_ccf_sptr.**diff_channel_taps**(*pfb_clock_sync_ccf_sptr self*, *int channel*) → pmt_vector_float

Returns the taps in the derivative filter for a particular channel

pfb_clock_sync_ccf_sptr.**diff_taps**(*pfb_clock_sync_ccf_sptr self*) → std::vector< std::vector< float,std::allocator< float > >,std::allocator< std::vector< float,std::allocator< float > > > >

Returns all of the taps of the derivative filter

pfb_clock_sync_ccf_sptr.**diff_taps_as_string**(*pfb_clock_sync_ccf_sptr self*) → std::string

Return the derivative filter taps as a formatted string for printing

pfb_clock_sync_ccf_sptr.**error**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the current error of the control loop.

pfb_clock_sync_ccf_sptr.**loop_bandwidth**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the loop bandwidth.

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

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

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

pc_input_buffers_full_avg(pfb_clock_sync_ccf_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(pfb_clock_sync_ccf_sptr self) -> pmt_vector_float

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

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

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

pfb_clock_sync_ccf_sptr.**phase**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the current phase arm of the control loop.

pfb_clock_sync_ccf_sptr.**rate**(*pfb_clock_sync_ccf_sptr self*) → float

Returns the current rate of the control loop.

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

pfb_clock_sync_ccf_sptr.**set_alpha**(*pfb_clock_sync_ccf_sptr self*, *float alpha*)

Set the loop gain alpha.

Set's the loop filter's alpha gain parameter.

This value should really only be set by adjusting the loop bandwidth and damping factor.

pfb_clock_sync_ccf_sptr.**set_beta**(*pfb_clock_sync_ccf_sptr self*, *float beta*)

Set the loop gain beta.

Set's the loop filter's beta gain parameter.

This value should really only be set by adjusting the loop bandwidth and damping factor.

pfb_clock_sync_ccf_sptr.**set_damping_factor**(*pfb_clock_sync_ccf_sptr self*, *float df*)

Set the loop damping factor.

Set the loop filter's damping factor to . The damping factor should be sqrt(2)/2.0 for critically damped systems. Set it to anything else only if you know what you are doing. It must be a number between 0 and 1.

When a new damping factor is set, the gains, alpha and beta, of the loop are recalculated by a call to update_gains().

pfb_clock_sync_ccf_sptr.**set_loop_bandwidth**(*pfb_clock_sync_ccf_sptr self*, *float bw*)

Set the loop bandwidth.

Set the loop filter's bandwidth to . This should be between 2*pi/200 and 2*pi/100 (in rads/samp). It must also be a positive number.

When a new damping factor is set, the gains, alpha and beta, of the loop are recalculated by a call to update_gains().

pfb_clock_sync_ccf_sptr.**set_max_rate_deviation**(*pfb_clock_sync_ccf_sptr self*, *float m*)

Set the maximum deviation from 0 d_rate can have

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

pfb_clock_sync_ccf_sptr.**set_taps**(*pfb_clock_sync_ccf_sptr self*, *pmt_vector_float taps*, *std::vector< std::vector< float, std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > > & ourtaps*, *std::vector< gr::filter::kernel::fir_filter_ccf *, std::allocator< gr::filter::kernel::fir_filter_ccf * > > & ourfilter*)

Used to set the taps of the filters in the filterbank and differential filterbank.

WARNING: this should not be used externally and will be moved to a private function in the next API.

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

pfb_clock_sync_ccf_sptr.**taps**(*pfb_clock_sync_ccf_sptr self*) → std::vector< std::vector< float,std::allocator< float > >,std::allocator< std::vector< float,std::allocator< float > > > >

Returns all of the taps of the matched filter

pfb_clock_sync_ccf_sptr.**taps_as_string**(*pfb_clock_sync_ccf_sptr self*) → std::string

Return the taps as a formatted string for printing

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

pfb_clock_sync_ccf_sptr.**update_gains**(*pfb_clock_sync_ccf_sptr self*)

update the system gains from omega and eta

This function updates the system gains based on the loop bandwidth and damping factor of the system. These two factors can be set separately through their own set functions.

pfb_clock_sync_ccf_sptr.**update_taps**(*pfb_clock_sync_ccf_sptr self*, *pmt_vector_float taps*)

Resets the filterbank's filter taps with the new prototype filter.

gnuradio.digital.**pfb_clock_sync_fff**(*double sps*, *float gain*, *pmt_vector_float taps*, *unsigned int filter_size=32*, *float init_phase=0*, *float max_rate_deviation=1.5*, *int osps=1*) → pfb_clock_sync_fff_sptr

Timing synchronizer using polyphase filterbanks.

This block performs timing synchronization for PAM signals by minimizing the derivative of the filtered signal, which in turn maximizes the SNR and minimizes ISI.

This approach works by setting up two filterbanks; one filterbank contains the signal's pulse shaping matched filter (such as a root raised cosine filter), where each branch of the filterbank contains a different phase of the filter. The second filterbank contains the derivatives of the filters in the first filterbank. Thinking of this in the time domain, the first filterbank contains filters that have a sinc shape to them. We want to align the output signal to be sampled at exactly the peak of the sinc shape. The derivative of the sinc contains a zero at the maximum point of the sinc (sinc(0) = 1, sinc(0)' = 0). Furthermore, the region around the zero point is relatively linear. We make use of this fact to generate the error signal.

If the signal out of the derivative filters is d_i[n] for the ith filter, and the output of the matched filter is x_i[n], we calculate the error as: e[n] = (Re{x_i[n]} * Re{d_i[n]} + Im{x_i[n]} * Im{d_i[n]}) / 2.0 This equation averages the error in the real and imaginary parts. There are two reasons we multiply by the signal itself. First, if the symbol could be positive or negative going, but we want the error term to always tell us to go in the same direction depending on which side of the zero point we are on. The sign of x_i[n] adjusts the error term to do this. Second, the magnitude of x_i[n] scales the error term depending on the symbol's amplitude, so larger signals give us a stronger error term because we have more confidence in that symbol's value. Using the magnitude of x_i[n] instead of just the sign is especially good for signals with low SNR.

The error signal, e[n], gives us a value proportional to how far away from the zero point we are in the derivative signal. We want to drive this value to zero, so we set up a second order loop. We have two variables for this loop; d_k is the filter number in the filterbank we are on and d_rate is the rate which we travel through the filters in the steady state. That is, due to the natural clock differences between the transmitter and receiver, d_rate represents that difference and would traverse the filter phase paths to keep the receiver locked. Thinking of this as a second-order PLL, the d_rate is the frequency and d_k is the phase. So we update d_rate and d_k using the standard loop equations based on two error signals, d_alpha and d_beta. We have these two values set based on each other for a critically damped system, so in the block constructor, we just ask for "gain," which is d_alpha while d_beta is equal to (gain^2)/4.

The block's parameters are:

Reference: f. j. harris and M. Rice, “Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios”, IEEE Selected Areas in Communications, Vol. 19, No. 12, Dec., 2001.

Constructor Specific Documentation:

Build the polyphase filterbank timing synchronizer.

**Parameters:**

- **sps** – (double) The number of samples per second in the incoming signal
- **gain** – (float) The alpha gain of the control loop; beta = (gain^2)/4 by default.
- **taps** – (vector<int>) The filter taps.
- **filter_size** – (uint) The number of filters in the filterbank (default = 32).
- **init_phase** – (float) The initial phase to look at, or which filter to start with (default = 0).
- **max_rate_deviation** – (float) Distance from 0 d_rate can get (default = 1.5).
- **osps** – (int) The number of output samples per symbol (default=1).

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

pfb_clock_sync_fff_sptr.**alpha**(*pfb_clock_sync_fff_sptr self*) → float

Returns the loop gain alpha.

pfb_clock_sync_fff_sptr.**beta**(*pfb_clock_sync_fff_sptr self*) → float

Returns the loop gain beta.

pfb_clock_sync_fff_sptr.**channel_taps**(*pfb_clock_sync_fff_sptr self*, *int channel*) → pmt_vector_float

Returns the taps of the matched filter for a particular channel

pfb_clock_sync_fff_sptr.**clock_rate**(*pfb_clock_sync_fff_sptr self*) → float

Returns the current clock rate.

pfb_clock_sync_fff_sptr.**damping_factor**(*pfb_clock_sync_fff_sptr self*) → float

Returns the loop damping factor.

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

declare_sample_delay(pfb_clock_sync_fff_sptr self, unsigned int delay)

pfb_clock_sync_fff_sptr.**diff_channel_taps**(*pfb_clock_sync_fff_sptr self*, *int channel*) → pmt_vector_float

Returns the taps in the derivative filter for a particular channel

pfb_clock_sync_fff_sptr.**diff_taps**(*pfb_clock_sync_fff_sptr self*) → std::vector< std::vector< float,std::allocator< float > >,std::allocator< std::vector< float,std::allocator< float > > > >

Returns all of the taps of the derivative filter

pfb_clock_sync_fff_sptr.**diff_taps_as_string**(*pfb_clock_sync_fff_sptr self*) → std::string

Return the derivative filter taps as a formatted string for printing

pfb_clock_sync_fff_sptr.**loop_bandwidth**(*pfb_clock_sync_fff_sptr self*) → float

Returns the loop bandwidth.

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

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

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

pc_input_buffers_full_avg(pfb_clock_sync_fff_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(pfb_clock_sync_fff_sptr self) -> pmt_vector_float

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

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

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

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

pfb_clock_sync_fff_sptr.**set_alpha**(*pfb_clock_sync_fff_sptr self*, *float alpha*)

Set the loop gain alpha.

Set's the loop filter's alpha gain parameter.

This value should really only be set by adjusting the loop bandwidth and damping factor.

pfb_clock_sync_fff_sptr.**set_beta**(*pfb_clock_sync_fff_sptr self*, *float beta*)

Set the loop gain beta.

Set's the loop filter's beta gain parameter.

This value should really only be set by adjusting the loop bandwidth and damping factor.

pfb_clock_sync_fff_sptr.**set_damping_factor**(*pfb_clock_sync_fff_sptr self*, *float df*)

Set the loop damping factor.

Set the loop filter's damping factor to . The damping factor should be sqrt(2)/2.0 for critically damped systems. Set it to anything else only if you know what you are doing. It must be a number between 0 and 1.

When a new damping factor is set, the gains, alpha and beta, of the loop are recalculated by a call to update_gains().

pfb_clock_sync_fff_sptr.**set_loop_bandwidth**(*pfb_clock_sync_fff_sptr self*, *float bw*)

Set the loop bandwidth.

Set the loop filter's bandwidth to . This should be between 2*pi/200 and 2*pi/100 (in rads/samp). It must also be a positive number.

When a new damping factor is set, the gains, alpha and beta, of the loop are recalculated by a call to update_gains().

pfb_clock_sync_fff_sptr.**set_max_rate_deviation**(*pfb_clock_sync_fff_sptr self*, *float m*)

Set the maximum deviation from 0 d_rate can have

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

pfb_clock_sync_fff_sptr.**set_taps**(*pfb_clock_sync_fff_sptr self*, *pmt_vector_float taps*, *std::vector< std::vector< float, std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > > & ourtaps*, *std::vector< gr::filter::kernel::fir_filter_fff *, std::allocator< gr::filter::kernel::fir_filter_fff * > > & ourfilter*)

Used to set the taps of the filters in the filterbank and differential filterbank.

WARNING: this should not be used externally and will be moved to a private function in the next API.

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

pfb_clock_sync_fff_sptr.**taps**(*pfb_clock_sync_fff_sptr self*) → std::vector< std::vector< float,std::allocator< float > >,std::allocator< std::vector< float,std::allocator< float > > > >

Returns all of the taps of the matched filter

pfb_clock_sync_fff_sptr.**taps_as_string**(*pfb_clock_sync_fff_sptr self*) → std::string

Return the taps as a formatted string for printing

pfb_clock_sync_fff_sptr.**thread_priority**(*pfb_clock_sync_fff_sptr self*) → int

pfb_clock_sync_fff_sptr.**update_gains**(*pfb_clock_sync_fff_sptr self*)

update the system gains from omega and eta

This function updates the system gains based on the loop bandwidth and damping factor of the system. These two factors can be set separately through their own set functions.

pfb_clock_sync_fff_sptr.**update_taps**(*pfb_clock_sync_fff_sptr self*, *pmt_vector_float taps*)

Resets the filterbank's filter taps with the new prototype filter.

gnuradio.digital.**pn_correlator_cc**(*int degree*, *int mask=0*, *int seed=1*) → pn_correlator_cc_sptr

PN code sequential search correlator.

Receives complex baseband signal, outputs complex correlation against reference PN code, one sample per PN code period. The PN sequence is generated using a GLFSR.

Constructor Specific Documentation:

Make PN code sequential search correlator block.

**Parameters:**
- **degree** – Degree of shift register must be in [1, 32]. If mask is 0, the degree determines a default mask (see digital_impl_glfsr.cc for the mapping).
- **mask** – Allows a user-defined bit mask for indexes of the shift register to feed back.
- **seed** – Initial setting for values in shift register.

pn_correlator_cc_sptr.**active_thread_priority**(*pn_correlator_cc_sptr self*) → int

pn_correlator_cc_sptr.**declare_sample_delay**(*pn_correlator_cc_sptr self*, *int which*, *int delay*)

declare_sample_delay(pn_correlator_cc_sptr self, unsigned int delay)

pn_correlator_cc_sptr.**message_subscribers**(*pn_correlator_cc_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

pn_correlator_cc_sptr.**min_noutput_items**(*pn_correlator_cc_sptr self*) → int

pn_correlator_cc_sptr.**pc_input_buffers_full_avg**(*pn_correlator_cc_sptr self*, *int which*) → float

pc_input_buffers_full_avg(pn_correlator_cc_sptr self) -> pmt_vector_float

pn_correlator_cc_sptr.**pc_noutput_items_avg**(*pn_correlator_cc_sptr self*) → float

pn_correlator_cc_sptr.**pc_nproduced_avg**(*pn_correlator_cc_sptr self*) → float

pn_correlator_cc_sptr.**pc_output_buffers_full_avg**(*pn_correlator_cc_sptr self*, *int which*) → float

pc_output_buffers_full_avg(pn_correlator_cc_sptr self) -> pmt_vector_float

pn_correlator_cc_sptr.**pc_throughput_avg**(*pn_correlator_cc_sptr self*) → float

pn_correlator_cc_sptr.**pc_work_time_avg**(*pn_correlator_cc_sptr self*) → float

pn_correlator_cc_sptr.**pc_work_time_total**(*pn_correlator_cc_sptr self*) → float

pn_correlator_cc_sptr.**sample_delay**(*pn_correlator_cc_sptr self*, *int which*) → unsigned int

pn_correlator_cc_sptr.**set_min_noutput_items**(*pn_correlator_cc_sptr self*, *int m*)

pn_correlator_cc_sptr.**set_thread_priority**(*pn_correlator_cc_sptr self*, *int priority*) → int

pn_correlator_cc_sptr.**thread_priority**(*pn_correlator_cc_sptr self*) → int

gnuradio.digital.**probe_density_b**(*double alpha*) → probe_density_b_sptr

This block maintains a running average of the input stream and makes it available as an accessor function. The input stream is type unsigned char.

If you send this block a stream of unpacked bytes, it will tell you what the bit density is.

Constructor Specific Documentation:

Make a density probe block.

**Parameters:** **alpha** – Average filter constant

probe_density_b_sptr.**active_thread_priority**(*probe_density_b_sptr self*) → int

probe_density_b_sptr.**declare_sample_delay**(*probe_density_b_sptr self*, *int which*, *int delay*)

declare_sample_delay(probe_density_b_sptr self, unsigned int delay)

probe_density_b_sptr.**density**(*probe_density_b_sptr self*) → double

Returns the current density value.

probe_density_b_sptr.**message_subscribers**(*probe_density_b_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

probe_density_b_sptr.**min_noutput_items**(*probe_density_b_sptr self*) → int

probe_density_b_sptr.**pc_input_buffers_full_avg**(*probe_density_b_sptr self*, *int which*) → float

pc_input_buffers_full_avg(probe_density_b_sptr self) -> pmt_vector_float

probe_density_b_sptr.**pc_noutput_items_avg**(*probe_density_b_sptr self*) → float

probe_density_b_sptr.**pc_nproduced_avg**(*probe_density_b_sptr self*) → float

probe_density_b_sptr.**pc_output_buffers_full_avg**(*probe_density_b_sptr self*, *int which*) → float

pc_output_buffers_full_avg(probe_density_b_sptr self) -> pmt_vector_float

probe_density_b_sptr.**pc_throughput_avg**(*probe_density_b_sptr self*) → float

probe_density_b_sptr.**pc_work_time_avg**(*probe_density_b_sptr self*) → float

probe_density_b_sptr.**pc_work_time_total**(*probe_density_b_sptr self*) → float

probe_density_b_sptr.**sample_delay**(*probe_density_b_sptr self*, *int which*) → unsigned int

probe_density_b_sptr.**set_alpha**(*probe_density_b_sptr self*, *double alpha*)

Set the average filter constant.

probe_density_b_sptr.**set_min_noutput_items**(*probe_density_b_sptr self*, *int m*)

probe_density_b_sptr.**set_thread_priority**(*probe_density_b_sptr self*, *int priority*) → int

probe_density_b_sptr.**thread_priority**(*probe_density_b_sptr self*) → int

gnuradio.digital.**probe_mpsk_snr_est_c**(*gr::digital::snr_est_type_t type*, *int msg_nsamples=10000*, *double alpha=0.001*) → probe_mpsk_snr_est_c_sptr

A probe for computing SNR of a PSK signal.

This is a probe block (a sink) that can be used to monitor and retrieve estimations of the signal SNR. This probe is designed for use with M-PSK signals especially. The type of estimator is specified as the parameter in the constructor. The estimators tend to trade off performance for accuracy, although experimentation should be done to figure out the right approach for a given implementation. Further, the current set of estimators are designed and proven theoretically under AWGN conditions; some amount of error should be assumed and/or estimated for real channel conditions.

The block has three output message ports that will emit a message every msg_samples number of samples. These message ports are: Some calibration is required to convert dBx of the signal and noise power estimates to real measurements, such as dBm.

Constructor Specific Documentation:

Make an MPSK SNR probe.

Parameters:

**Parameters:**
- **type** – the type of estimator to use see gr::digital::snr_est_type_t for details about the types.
- **msg_nsamples** – [not implemented yet] after this many samples, a message containing the SNR (key='snr') will be sent
- **alpha** – the update rate of internal running average calculations.

probe_mpsk_snr_est_c_sptr.**active_thread_priority**(*probe_mpsk_snr_est_c_sptr self*) → int

probe_mpsk_snr_est_c_sptr.**alpha**(*probe_mpsk_snr_est_c_sptr self*) → double

Get the running-average coefficient.

probe_mpsk_snr_est_c_sptr.**declare_sample_delay**(*probe_mpsk_snr_est_c_sptr self*, *int which*, *int delay*)

declare_sample_delay(probe_mpsk_snr_est_c_sptr self, unsigned int delay)

probe_mpsk_snr_est_c_sptr.**message_subscribers**(*probe_mpsk_snr_est_c_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

probe_mpsk_snr_est_c_sptr.**min_noutput_items**(*probe_mpsk_snr_est_c_sptr self*) → int

probe_mpsk_snr_est_c_sptr.**msg_nsample**(*probe_mpsk_snr_est_c_sptr self*) → int

Return how many samples between SNR messages.

probe_mpsk_snr_est_c_sptr.**noise**(*probe_mpsk_snr_est_c_sptr self*) → double

Return the estimated noise power in decibels.

probe_mpsk_snr_est_c_sptr.**pc_input_buffers_full_avg**(*probe_mpsk_snr_est_c_sptr self*, *int which*) → float

pc_input_buffers_full_avg(probe_mpsk_snr_est_c_sptr self) -> pmt_vector_float

probe_mpsk_snr_est_c_sptr.**pc_noutput_items_avg**(*probe_mpsk_snr_est_c_sptr self*) → float

probe_mpsk_snr_est_c_sptr.**pc_nproduced_avg**(*probe_mpsk_snr_est_c_sptr self*) → float

probe_mpsk_snr_est_c_sptr.**pc_output_buffers_full_avg**(*probe_mpsk_snr_est_c_sptr self*, *int which*) → float

pc_output_buffers_full_avg(probe_mpsk_snr_est_c_sptr self) -> pmt_vector_float

probe_mpsk_snr_est_c_sptr.**pc_throughput_avg**(*probe_mpsk_snr_est_c_sptr self*) → float

probe_mpsk_snr_est_c_sptr.**pc_work_time_avg**(*probe_mpsk_snr_est_c_sptr self*) → float

probe_mpsk_snr_est_c_sptr.**pc_work_time_total**(*probe_mpsk_snr_est_c_sptr self*) → float

probe_mpsk_snr_est_c_sptr.**sample_delay**(*probe_mpsk_snr_est_c_sptr self*, *int which*) → unsigned int

probe_mpsk_snr_est_c_sptr.**set_alpha**(*probe_mpsk_snr_est_c_sptr self*, *double alpha*)

Set the running-average coefficient.

probe_mpsk_snr_est_c_sptr.**set_min_noutput_items**(*probe_mpsk_snr_est_c_sptr self*, *int m*)

probe_mpsk_snr_est_c_sptr.**set_msg_nsample**(*probe_mpsk_snr_est_c_sptr self*, *int n*)

Set the number of samples between SNR messages.

probe_mpsk_snr_est_c_sptr.**set_thread_priority**(*probe_mpsk_snr_est_c_sptr self*, *int priority*) → int

probe_mpsk_snr_est_c_sptr.**set_type**(*probe_mpsk_snr_est_c_sptr self*, *gr::digital::snr_est_type_t t*)

Set type of estimator to use.

probe_mpsk_snr_est_c_sptr.**signal**(*probe_mpsk_snr_est_c_sptr self*) → double

Return the estimated signal power in decibels.

probe_mpsk_snr_est_c_sptr.**snr**(*probe_mpsk_snr_est_c_sptr self*) → double

Return the estimated signal-to-noise ratio in decibels.

probe_mpsk_snr_est_c_sptr.**thread_priority**(*probe_mpsk_snr_est_c_sptr self*) → int

gnuradio.digital.**scrambler_bb**(*int mask*, *int seed*, *int len*) → scrambler_bb_sptr

Scramble an input stream using an LFSR.

This block works on the LSB only of the input data stream, i.e., on an “unpacked binary” stream, and produces the same format on its output.

Constructor Specific Documentation:

Make a scramber block.

**Parameters:**
- **mask** – Polynomial mask for LFSR
- **seed** – Initial shift register contents
- **len** – Shift register length

scrambler_bb_sptr.**active_thread_priority**(*scrambler_bb_sptr self*) → int

scrambler_bb_sptr.**declare_sample_delay**(*scrambler_bb_sptr self*, *int which*, *int delay*)

declare_sample_delay(scrambler_bb_sptr self, unsigned int delay)

scrambler_bb_sptr.**message_subscribers**(*scrambler_bb_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

scrambler_bb_sptr.**min_noutput_items**(*scrambler_bb_sptr self*) → int

scrambler_bb_sptr.**pc_input_buffers_full_avg**(*scrambler_bb_sptr self*, *int which*) → float

pc_input_buffers_full_avg(scrambler_bb_sptr self) -> pmt_vector_float

scrambler_bb_sptr.**pc_noutput_items_avg**(*scrambler_bb_sptr self*) → float

scrambler_bb_sptr.**pc_nproduced_avg**(*scrambler_bb_sptr self*) → float

scrambler_bb_sptr.**pc_output_buffers_full_avg**(*scrambler_bb_sptr self*, *int which*) → float

pc_output_buffers_full_avg(scrambler_bb_sptr self) -> pmt_vector_float

scrambler_bb_sptr.**pc_throughput_avg**(*scrambler_bb_sptr self*) → float

scrambler_bb_sptr.**pc_work_time_avg**(*scrambler_bb_sptr self*) → float

scrambler_bb_sptr.**pc_work_time_total**(*scrambler_bb_sptr self*) → float

scrambler_bb_sptr.**sample_delay**(*scrambler_bb_sptr self*, *int which*) → unsigned int

scrambler_bb_sptr.**set_min_noutput_items**(*scrambler_bb_sptr self*, *int m*)

scrambler_bb_sptr.**set_thread_priority**(*scrambler_bb_sptr self*, *int priority*) → int

scrambler_bb_sptr.**thread_priority**(*scrambler_bb_sptr self*) → int

gnuradio.digital.**simple_correlator**(*int payload_bytesize*) → simple_correlator_sptr

inverse of simple_framer (more or less)

Constructor Specific Documentation:

**Parameters:** **payload_bytesize** –

simple_correlator_sptr.**active_thread_priority**(*simple_correlator_sptr self*) → int

simple_correlator_sptr.**declare_sample_delay**(*simple_correlator_sptr self*, *int which*, *int delay*)

declare_sample_delay(simple_correlator_sptr self, unsigned int delay)

simple_correlator_sptr.**message_subscribers**(*simple_correlator_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

simple_correlator_sptr.**min_noutput_items**(*simple_correlator_sptr self*) → int

simple_correlator_sptr.**pc_input_buffers_full_avg**(*simple_correlator_sptr self*, *int which*) → float

pc_input_buffers_full_avg(simple_correlator_sptr self) -> pmt_vector_float

simple_correlator_sptr.**pc_noutput_items_avg**(*simple_correlator_sptr self*) → float

simple_correlator_sptr.**pc_nproduced_avg**(*simple_correlator_sptr self*) → float

simple_correlator_sptr.**pc_output_buffers_full_avg**(*simple_correlator_sptr self*, *int which*) → float

pc_output_buffers_full_avg(simple_correlator_sptr self) -> pmt_vector_float

simple_correlator_sptr.**pc_throughput_avg**(*simple_correlator_sptr self*) → float

simple_correlator_sptr.**pc_work_time_avg**(*simple_correlator_sptr self*) → float

simple_correlator_sptr.**pc_work_time_total**(*simple_correlator_sptr self*) → float

simple_correlator_sptr.**sample_delay**(*simple_correlator_sptr self*, *int which*) → unsigned int

simple_correlator_sptr.**set_min_noutput_items**(*simple_correlator_sptr self*, *int m*)

simple_correlator_sptr.**set_thread_priority**(*simple_correlator_sptr self*, *int priority*) → int

simple_correlator_sptr.**thread_priority**(*simple_correlator_sptr self*) → int

gnuradio.digital.**simple_framer**(*int payload_bytesize*) → simple_framer_sptr

add sync field, seq number and command field to payload

Takes in enough samples to create a full output frame. The frame is prepended with the GRSF_SYNC (defined in simple_framer_sync.h) and an 8-bit sequence number.

Constructor Specific Documentation:

Make a simple_framer block.

**Parameters:** **payload_bytesize** – The size of the payload in bytes.

simple_framer_sptr.**active_thread_priority**(*simple_framer_sptr self*) → int

simple_framer_sptr.**declare_sample_delay**(*simple_framer_sptr self*, *int which*, *int delay*)

declare_sample_delay(simple_framer_sptr self, unsigned int delay)

simple_framer_sptr.**message_subscribers**(*simple_framer_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

simple_framer_sptr.**min_noutput_items**(*simple_framer_sptr self*) → int

simple_framer_sptr.**pc_input_buffers_full_avg**(*simple_framer_sptr self*, *int which*) → float

pc_input_buffers_full_avg(simple_framer_sptr self) -> pmt_vector_float

simple_framer_sptr.**pc_noutput_items_avg**(*simple_framer_sptr self*) → float

simple_framer_sptr.**pc_nproduced_avg**(*simple_framer_sptr self*) → float

simple_framer_sptr.**pc_output_buffers_full_avg**(*simple_framer_sptr self*, *int which*) → float

pc_output_buffers_full_avg(simple_framer_sptr self) -> pmt_vector_float

simple_framer_sptr.**pc_throughput_avg**(*simple_framer_sptr self*) → float

simple_framer_sptr.**pc_work_time_avg**(*simple_framer_sptr self*) → float

simple_framer_sptr.**pc_work_time_total**(*simple_framer_sptr self*) → float

simple_framer_sptr.**sample_delay**(*simple_framer_sptr self*, *int which*) → unsigned int

simple_framer_sptr.**set_min_noutput_items**(*simple_framer_sptr self*, *int m*)

simple_framer_sptr.**set_thread_priority**(*simple_framer_sptr self*, *int priority*) → int

simple_framer_sptr.**thread_priority**(*simple_framer_sptr self*) → int