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

# gnuradio.fec

Blocks for forward error correction.

gnuradio.fec.**async_decoder**(*generic_decoder_sptr my_decoder*, *bool packed=False*, *bool rev_pack=True*, *int mtu=1500*) → async_decoder_sptr

Creates the decoder block for use in GNU Radio flowgraphs from a given FEC API object derived from the generic_decoder class.

Decodes frames received as async messages over a message port. This decoder deployment expects messages of soft decision symbols in and can produce either packed, PDU messages ( = True) or messages full of unpacked bits ( = False).

This decoder works off a full message as one frame or block to decode. The message length is used to calculate the frame length. To support this, the decoder variable used will have had its frame_size set. This block treats that initial frame_size value as the maximum transmission unit (MTU) and will not process frames larger than that after being decoded.

The packed PDU form of this deployment is designed to work well with other PDU-based blocks to operate within the processing flow of data packets or frames.

Due to differences in how data is packed and processed, this block also offers the ability to change the direction of how bits are packed. All inputs messages are one soft decision per item. By default, the mode is set to True. Using this setup allows the async block to behave with PDUs in the same operation and format as the tagged stream decoders. That is, putting the same data into both the tagged stream decoder deployment and this with the default setting should produce the same data.

Because the block handles data as a full frame per message, this decoder deployment cannot work with any decoders that require history. For example, the gr::fec::code::cc_decoder decoder in streaming mode requires an extra rate*(K-1) bits to complete the decoding, so it would have to wait for the next message to come in and finish processing. Therefore, the streaming mode of the CC decoder is not allowed. The other three modes will work with this deployment since the frame is self-contained for decoding.

Constructor Specific Documentation:

Build the PDU-based FEC decoder block from an FECAPI decoder object.

**Parameters:**
- **my_decoder** – An FECAPI decoder object child of the generic_decoder class.
- **packed** – Sets output to packed bytes if true; otherwise, 1 bit per byte.
- **rev_pack** – If packing bits, should they be reversed?
- **mtu** – The Maximum Transmission Unit (MTU) of the output frame that the block will be able to process. Specified in bytes and defaults to 1500.

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

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

declare_sample_delay(async_decoder_sptr self, unsigned int delay)

async_decoder_sptr.**general_work**(*async_decoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int

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

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

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

pc_input_buffers_full_avg(async_decoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(async_decoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**async_encoder**(*generic_encoder_sptr my_encoder*, *bool packed=False*, *bool rev_unpack=True*, *bool rev_pack=True*, *int mtu=1500*) → async_encoder_sptr

Creates the encoder block for use in GNU Radio flowgraphs with async message from a given FEC API object derived from the generic_encoder class.

Encodes frames received as async messages or as a PDU over a message port. This encoder works off a full message as one frame or block to encode. The message length is used as the frame length. To support this, the encoder variable used will have had its frame_size set. This block treats that initial frame_size value as the maximum transmission unit (MTU) and will not process frames larger than that.

This deployment works off messages and expects them to either be messages full of unpacked bits or PDU messages, which means full bytes of a frame from the higher layers, including things like headers, tails, CRC check bytes, etc. For handling PDUs, set the option of this deployment block to True. The block will then use the FEC API to properly unpack the bits from the PDU, pass it through the encoder, and repack them to output the PDUs for the next stage of processing.

The packed PDU form of this deployment is designed to work well with other PDU-based blocks to operate within the processing flow of data packets or frames.

Due to differences in how data is packed and processed, this block also offers the ability to change the direction of how bits are unpacked and packed, where reading or writing from the LSB or MSB. By default, theand modes are set to True. Using this setup allows the async block to behave with PDUs in the same operation and format as the tagged stream encoders. That is, putting the same data into both the tagged stream encoder deployment and this with these default settings should produce the same data.

Constructor Specific Documentation:

Build the PDU-based FEC encoder block from an FECAPI encoder object.

**Parameters:**
- **my_encoder** – An FECAPI encoder object child of the generic_encoder class.
- **packed** – True if working on packed bytes (like PDUs).
- **rev_unpack** – Reverse the unpacking order from input bytes to bits.
- **rev_pack** – Reverse the packing order from bits to output bytes.
- **mtu** – The Maximum Transmission Unit (MTU) of the input frame that the block will be able to process. Specified in bytes and defaults to 1500.

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

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

declare_sample_delay(async_encoder_sptr self, unsigned int delay)

async_encoder_sptr.**general_work**(*async_encoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int

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

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

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

pc_input_buffers_full_avg(async_encoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(async_encoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**ber_bf**(*bool test_mode=False*, *int berminerrors=100*, *float ber_limit=-7.0*) → ber_bf_sptr

BER block in FECAPI.

This block measures the bit error rate between two streams of packed data. It compares the bits of each streams and counts the number of incorrect bits between them. It outputs the log of the bit error rate, so a value of -X is 10^{-X} bit errors.

When the is set to false (default), it is in streaming mode. This means that the output is constantly producing the current value of the BER. In this mode, there is a single output BER calculation per chunk of bytes passed to it, so there is no exact timing between calculations of BER. In this mode, the other two parameters to the constructor are ignored.

When is true, the block is in test mode. This mode is used in the ber_curve_gen example and for other offline analysis of BER curves. Here, the block waits until at leastare observed and then produces a BER calculation. The parameterhelps make sure that the simulation is controlled. If the BER calculation drops below thesetting, the block will exit and simply return the set limit; the real BER is therefore some amount lower than this.

Note that this block takes in data as packed bytes with 8-bits per byte used. It outputs a stream of floats as the log-scale BER.

Constructor Specific Documentation:

Calculate the BER between two streams of data.

**Parameters:**
- **test_mode** – false for normal streaming mode (default); true for test mode.
- **berminerrors** – the block needs to observe this many errors before outputting a result. Only valid when test_mode=true.
- **ber_limit** – if the BER calculation falls below this limit, produce this value and exit. Only valid when test_mode=true.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ber_bf_sptr.**total_errors**(*ber_bf_sptr self*) → long
Get total number of errors counter value.

gnuradio.fec.**conv_bit_corr_bb**(*std::vector< unsigned long long, std::allocator< unsigned long long > > correlator*, *int corr_sym*, *int corr_len*, *int cut*, *int flush*, *float thresh*) → conv_bit_corr_bb_sptr
Correlate block in FECAPI.

What does this block do?

Constructor Specific Documentation:

**Parameters:**
- **correlator** –
- **corr_sym** –
- **corr_len** –
- **cut** –
- **flush** –
- **thresh** –

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

conv_bit_corr_bb_sptr.**data_garble_rate**(*conv_bit_corr_bb_sptr self*, *int taps*, *float syn_density*) → float

This subroutine will find the encoded data garble rate corresponding to a syndrome density of `target', that is created with an annihilating polynomial with 'taps' number of taps.

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

declare_sample_delay(conv_bit_corr_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(conv_bit_corr_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(conv_bit_corr_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**decode_ccsds_27_fb**() → decode_ccsds_27_fb_sptr

A rate 1/2, k=7 convolutional decoder for the CCSDS standard.

This block performs soft-decision convolutional decoding using the Viterbi algorithm.

The input is a stream of (possibly noise corrupted) floating point values nominally spanning [-1.0, 1.0], representing the encoded channel symbols 0 (-1.0) and 1 (1.0), with erased symbols at 0.0.

The output is MSB first packed bytes of decoded values.

As a rate 1/2 code, there will be one output byte for every 16 input symbols.

This block is designed for continuous data streaming, not packetized data. The first 32 bits out will be zeroes, with the output delayed four bytes from the corresponding inputs.

Constructor Specific Documentation:

decode_ccsds_27_fb_sptr.**active_thread_priority**(*decode_ccsds_27_fb_sptr self*) → int

decode_ccsds_27_fb_sptr.**declare_sample_delay**(*decode_ccsds_27_fb_sptr self*, *int which*, *int delay*)

declare_sample_delay(decode_ccsds_27_fb_sptr self, unsigned int delay)

decode_ccsds_27_fb_sptr.**message_subscribers**(*decode_ccsds_27_fb_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

decode_ccsds_27_fb_sptr.**min_noutput_items**(*decode_ccsds_27_fb_sptr self*) → int

decode_ccsds_27_fb_sptr.**pc_input_buffers_full_avg**(*decode_ccsds_27_fb_sptr self*, *int which*) → float

pc_input_buffers_full_avg(decode_ccsds_27_fb_sptr self) -> pmt_vector_float

decode_ccsds_27_fb_sptr.**pc_noutput_items_avg**(*decode_ccsds_27_fb_sptr self*) → float

decode_ccsds_27_fb_sptr.**pc_nproduced_avg**(*decode_ccsds_27_fb_sptr self*) → float

decode_ccsds_27_fb_sptr.**pc_output_buffers_full_avg**(*decode_ccsds_27_fb_sptr self*, *int which*) → float

pc_output_buffers_full_avg(decode_ccsds_27_fb_sptr self) -> pmt_vector_float

decode_ccsds_27_fb_sptr.**pc_throughput_avg**(*decode_ccsds_27_fb_sptr self*) → float

decode_ccsds_27_fb_sptr.**pc_work_time_avg**(*decode_ccsds_27_fb_sptr self*) → float

decode_ccsds_27_fb_sptr.**pc_work_time_total**(*decode_ccsds_27_fb_sptr self*) → float

decode_ccsds_27_fb_sptr.**sample_delay**(*decode_ccsds_27_fb_sptr self*, *int which*) → unsigned int

decode_ccsds_27_fb_sptr.**set_min_noutput_items**(*decode_ccsds_27_fb_sptr self*, *int m*)

decode_ccsds_27_fb_sptr.**set_thread_priority**(*decode_ccsds_27_fb_sptr self*, *int priority*) → int

decode_ccsds_27_fb_sptr.**thread_priority**(*decode_ccsds_27_fb_sptr self*) → int

gnuradio.fec.**decoder**(*generic_decoder_sptr my_decoder*, *size_t input_item_size*, *size_t output_item_size*) → decoder_sptr

General FEC decoding block that takes in a decoder variable object (derived from gr::fec::general_decoder) for use in a flowgraph.

This block uses a decoder variable object (derived from gr::fec::generic_decoder) to decode data within a flowgraph. This block interacts with the general FECAPI architecture to handle all passing all input and output data in a flowgraph. The decoder variable takes care of understanding the requirements, data types and sizes, and boundary conditions of the specific FEC decoding algorithm.

Generally, this block is used within the fec.extended_decoder Python block to handle

some input/output formatting issues. In the FECAPI, the decoder variable sets properties like the input and output types and sizes and whether the output is packed or unpacked bytes. The fec.extended_decoder uses this information to set up an gr::hier_block2 structure to make sure the I/O to the variable is handled consistently, such as to make sure all inputs are floats with one soft symbol per item and the outputs are unpacked bytes with the bit in the LSB.

See gr::fec::generic_decoder for detail on what information an FECAPI variable object can set if using this block directly and not as part of the fec.extended_decoder.

Constructor Specific Documentation:

Create the FEC decoder block by taking in the FECAPI decoder object as well as input and output sizes.

**Parameters:**

- **my_decoder** – An FECAPI decoder object (See gr::fec::generic_decoder).
- **input_item_size** – The size of the input items (often the my_decoder object can tell us this).
- **output_item_size** – The size of the output items (often the my_decoder object can tell us this).

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

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

declare_sample_delay(decoder_sptr self, unsigned int delay)

decoder_sptr.**fixed_rate_ninput_to_noutput**(*decoder_sptr self*, *int ninput*) → int

decoder_sptr.**fixed_rate_noutput_to_ninput**(*decoder_sptr self*, *int noutput*) → int

decoder_sptr.**forecast**(*decoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items_required*)

decoder_sptr.**general_work**(*decoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int

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

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

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

pc_input_buffers_full_avg(decoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(decoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**depuncture_bb**(*int puncsize*, *int puncpat*, *int delay=0*, *char symbol=127*) → depuncture_bb_sptr

Depuncture a stream of samples.

Depuncture a given block of input samples of . The items produced is based on the pattern . Basically, if:

This block is designed for unpacked bits - that is, every input sample is a bit, either a 1 or 0. It's possible to use packed bits as symbols, but the depuncturing will be done on the symbol level, not the bit level.

> is specified as a 32-bit integer that we can convert into the vector _puncpat used in the algorithm above:

Example:

The gr.fec Python module provides a read_bitlist function that can turn a string of a puncture pattern into the correct integer form. The pattern of 0xEF could be specified as fec.readbitlist("11101111"). Also, this allows us to use puncsize=len("11101111") to make sure that our sizes are set up correctly for the pattern we want.

The fec.extended_decoder takes in the puncture pattern directly as a string and uses the readbitlist inside to do the conversion.

The parameter delays the application of the puncture pattern. This is equivalent to circularly rotating the by . Note that because of the circular shift, the delay should be between 0 and , but this is not enforced; the effective delay will simply be mod . A negative value here is ignored.

Constructor Specific Documentation:

Constructs a depuncture block.

**Parameters:**
- **puncsize** – Size of block of bits to puncture
- **puncpat** – The puncturing pattern
- **delay** – Delayed the puncturing pattern by shifting it
- **symbol** – The symbol to reinsert into the stream (def=127)

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

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

declare_sample_delay(depuncture_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(depuncture_bb_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(depuncture_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**encode_ccsds_27_bb**() → encode_ccsds_27_bb_sptr

A rate 1/2, k=7 convolutional encoder for the CCSDS standard.

This block performs convolutional encoding using the CCSDS standard polynomial ("Voyager").

The input is an MSB first packed stream of bits.

The output is a stream of symbols 0 or 1 representing the encoded data.

As a rate 1/2 code, there will be 16 output symbols for every input byte.

This block is designed for continuous data streaming, not packetized data. There is no provision to "flush" the encoder.

Constructor Specific Documentation:

encode_ccsds_27_bb_sptr.**active_thread_priority**(*encode_ccsds_27_bb_sptr self*) → int

encode_ccsds_27_bb_sptr.**declare_sample_delay**(*encode_ccsds_27_bb_sptr self*, *int which*, *int delay*)

declare_sample_delay(encode_ccsds_27_bb_sptr self, unsigned int delay)

encode_ccsds_27_bb_sptr.**message_subscribers**(*encode_ccsds_27_bb_sptr self*, *swig_int_ptr which_port*) → swig_int_ptr

encode_ccsds_27_bb_sptr.**min_noutput_items**(*encode_ccsds_27_bb_sptr self*) → int

encode_ccsds_27_bb_sptr.**pc_input_buffers_full_avg**(*encode_ccsds_27_bb_sptr self*, *int which*) → float

pc_input_buffers_full_avg(encode_ccsds_27_bb_sptr self) -> pmt_vector_float

encode_ccsds_27_bb_sptr.**pc_noutput_items_avg**(*encode_ccsds_27_bb_sptr self*) → float

encode_ccsds_27_bb_sptr.**pc_nproduced_avg**(*encode_ccsds_27_bb_sptr self*) → float

encode_ccsds_27_bb_sptr.**pc_output_buffers_full_avg**(*encode_ccsds_27_bb_sptr self*, *int which*) → float

pc_output_buffers_full_avg(encode_ccsds_27_bb_sptr self) -> pmt_vector_float

encode_ccsds_27_bb_sptr.**pc_throughput_avg**(*encode_ccsds_27_bb_sptr self*) → float

encode_ccsds_27_bb_sptr.**pc_work_time_avg**(*encode_ccsds_27_bb_sptr self*) → float

encode_ccsds_27_bb_sptr.**pc_work_time_total**(*encode_ccsds_27_bb_sptr self*) → float

encode_ccsds_27_bb_sptr.**sample_delay**(*encode_ccsds_27_bb_sptr self*, *int which*) → unsigned int

encode_ccsds_27_bb_sptr.**set_min_noutput_items**(*encode_ccsds_27_bb_sptr self*, *int m*)

encode_ccsds_27_bb_sptr.**set_thread_priority**(*encode_ccsds_27_bb_sptr self*, *int priority*) → int

encode_ccsds_27_bb_sptr.**thread_priority**(*encode_ccsds_27_bb_sptr self*) → int

gnuradio.fec.**encoder**(*generic_encoder_sptr my_encoder*, *size_t input_item_size*, *size_t output_item_size*) → encoder_sptr

Creates the encoder block for use in GNU Radio flowgraphs from a given FECAPI object derived from the generic_encoder class.

Generally, we would use the fec.extended_encoder Python implementation to instantiate this. The extended_encoder wraps up a few more details, like taking care of puncturing as well as the encoder itself.

Constructor Specific Documentation:

Build the FEC encoder block from an FECAPI encoder object.

**Parameters:**
- **my_encoder** – An FECAPI encoder object child of the generic_encoder class.
- **input_item_size** – size of a block of data for the encoder.
- **output_item_size** – size of a block of data the encoder will produce.

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

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

declare_sample_delay(encoder_sptr self, unsigned int delay)

encoder_sptr.**fixed_rate_ninput_to_noutput**(*encoder_sptr self*, *int ninput*) → int

encoder_sptr.**fixed_rate_noutput_to_ninput**(*encoder_sptr self*, *int noutput*) → int

encoder_sptr.**forecast**(*encoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items_required*)

encoder_sptr.**general_work**(*encoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int

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

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

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

pc_input_buffers_full_avg(encoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(encoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**generic_decoder**(**args*, ***kwargs*)

Parent class for FECAPI objects.

Parent of a decoder variable class for FECAPI that will fit into the gr::fec::decoder block to handle FEC decoding. This class provides the basic information required to fit into the FECAPI structure. It provides information about input and output data types, potential data conversions, and a few other parameters useful to establish the decoder's behavior.

We create objects from FECAPI-derived classes to go into the actual GNU Radio decoder block. Each object contains its own state and so there should be a one-to-one mapping of an FECAPI object and a GR decoder block. Sharing these objects is not guaranteed to be thread-safe.

This is a pure virtual class and must be derived from by a child class.

generic_decoder_sptr.**base_unique_id**(*generic_decoder_sptr self*) → int

generic_decoder_sptr.**d_name**(*generic_decoder_sptr self*) → std::string const &

generic_decoder_sptr.**generic_work**(*generic_decoder_sptr self*, *void * inbuffer*, *void * outbuffer*)

generic_decoder_sptr.**get_history**(*generic_decoder_sptr self*) → int

Sets up history for the decoder when the decoder is required to look ahead in the data stream in order to finish its processing.

The child class MAY implement this function. If not reimplemented, it returns 0.

generic_decoder_sptr.**get_input_conversion**(*generic_decoder_sptr self*) → char const *

Set up a conversion type required to setup the data properly for this decoder. The decoder itself will not implement the conversion and expects an external wrapper (e.g., fec.extended_decoder) to read this value and "do the right thing" to format the data.

The default behavior is 'none', which means no conversion is required. Whatever the get_input_item_size() value returns, the input is expected to conform directly to this.

This may also return 'uchar', which indicates that the wrapper should convert the standard float samples to unsigned characters, either hard sliced or 8-bit soft symbols. See gr::fec::code::cc_decoder as an example decoder that uses this conversion format.

If 'packed_bits', the block expects the inputs to be packed hard bits. Each input item is a unsigned char where each of the 8-bits is a hard bit value.

The child class SHOULD implement this function. If not reimplemented, it returns "none".

generic_decoder_sptr.**get_input_item_size**(*generic_decoder_sptr self*) → int

Sets the size of an input item, as in the size of a char or float item.

The child class SHOULD implement this function. If not reimplemented, it returns sizeof(float) as the decoders typically expect floating point input types.

generic_decoder_sptr.**get_input_size**(*generic_decoder_sptr self*) → int

Returns the input size in items that the decoder object uses to decode a full frame. Often, this number is the number of bits per frame if the input format is unpacked. If the block expects packed bytes, then this value should be the number of bytes (number of bits / 8) per input frame.

The child class MUST implement this function.

generic_decoder_sptr.**get_iterations**(*generic_decoder_sptr self*) → float

Get repetitions to decode.

The child class should implement this function and return the number of iterations required to decode.

generic_decoder_sptr.**get_output_conversion**(*generic_decoder_sptr self*) → char const *

Set up a conversion type required to understand the output style of this decoder. Generally, follow-on processing expects unpacked bits, so we specify the conversion type here to indicate what the wrapper (e.g., fec.extended_decoder) should do to convert the output samples from the decoder into unpacked bits.

The default behavior is 'none', which means no conversion is required. This should mean that the output data is produced from this decoder as unpacked bit.

If 'unpack', the block produces packed bytes that should be unpacked by the wrapper. See gr::fec::code::ccsds_decoder as an example of a decoder that produces packed bytes.

The child class SHOULD implement this function. If not reimplemented, it returns "none".

generic_decoder_sptr.**get_output_item_size**(*generic_decoder_sptr self*) → int

Sets the size of an output item, as in the size of a char or float item.

The child class SHOULD implement this function. If not reimplemented, it returns sizeof(char) as the decoders typically expect to produce bits or bytes.

generic_decoder_sptr.**get_output_size**(*generic_decoder_sptr self*) → int

Returns the output size in items that the decoder object produces after decoding a full frame. Often, this number is the number of bits in the outputted frame if the input format is unpacked. If the block produces packed bytes, then this value should be the number of bytes (number of bits / 8) per frame produced. This value is generally something like get_input_size()/R for a 1/R rate code.

The child class MUST implement this function.

generic_decoder_sptr.**get_shift**(*generic_decoder_sptr self*) → float

Some decoders require the input items to float around a particular soft value. We can set that floating value by setting this value to return some non-zero number.

The fec.extended_decoder block will use this to create an add_const_ff block before the decoder block to adjust all input samples appropriately.

The child class MAY implement this function. If not reimplemented, it returns 0.

generic_decoder_sptr.**my_id**(*generic_decoder_sptr self*) → int

generic_decoder_sptr.**rate**(*generic_decoder_sptr self*) → double

Returns the rate of the code. For every r input bits, there is 1 output bit, so the rate is 1/r. Used for setting things like the encoder block's relative rate.

This function MUST be reimplemented by the child class.

generic_decoder_sptr.**set_frame_size**(*generic_decoder_sptr self*, *unsigned int frame_size*) → bool

Updates the size of a decoded frame.

The child class MUST implement this function and interpret how theinformation affects the block's behavior. It should also provide bounds checks.

gnuradio.fec.**generic_encoder**(**args*, ***kwargs*)

Proxy of C++ gr::fec::generic_encoder class.

generic_encoder_sptr.**base_unique_id**(*generic_encoder_sptr self*) → int

generic_encoder_sptr.**d_name**(*generic_encoder_sptr self*) → std::string const &

generic_encoder_sptr.**generic_work**(*generic_encoder_sptr self*, *void * in_buffer*, *void * out_buffer*)

generic_encoder_sptr.**get_input_conversion**(*generic_encoder_sptr self*) → char const *

Set up a conversion type required to setup the data properly for this encoder. The encoder itself will not implement the conversion and expects an external wrapper (e.g., fec.extended_encoder) to read this value and "do the right thing" to format the data.

The default behavior is 'none', which means no conversion is required. Whatever the get_input_item_size() value returns, the input is expected to conform directly to this. Generally, this means unpacked bytes.

If 'pack', the block expects the inputs to be packed bytes. The wrapper should implement a gr::blocks::pack_k_bits_bb(8) block for this.

The child class MAY implement this function. If not reimplemented, it returns "none".

generic_encoder_sptr.**get_input_size**(*generic_encoder_sptr self*) → int

Returns the input size in items that the encoder object uses to encode a full frame. Often, this number is the number of bits per frame if the input format is unpacked. If the block expects packed bytes, then this value should be the number of bytes (number of bits / 8) per input frame.

The child class MUST implement this function.

generic_encoder_sptr.**get_output_conversion**(*generic_encoder_sptr self*) → char const *

Set up a conversion type required to understand the output style of this encoder. Generally an encoder will produce unpacked bytes with a bit set in the LSB.

The default behavior is 'none', which means no conversion is required and the encoder produces unpacked bytes.

If 'packed_bits', the block produces packed bits and the wrapper should unpack these (using, for instance, gr::block::unpack_k_bits_bb(8)).

The child class MAY implement this function. If not reimplemented, it returns "none".

generic_encoder_sptr.**get_output_size**(*generic_encoder_sptr self*) → int

Returns the output size in items that the encoder object produces after encoding a full frame. Often, this number is the number of bits in the outputted frame if the input format is unpacked. If the block produces packed bytes, then this value should be the number of bytes (number of bits / 8) per frame produced. This value is generally something like R*get_input_size() for a 1/R rate code.

The child class MUST implement this function.

generic_encoder_sptr.**my_id**(*generic_encoder_sptr self*) → int

generic_encoder_sptr.**rate**(*generic_encoder_sptr self*) → double

Returns the rate of the code. For every 1 input bit, there are r output bits, so the rate is 1/r. Used for setting things like the encoder block's relative rate.

This function MUST be reimplemented by the child class.

generic_encoder_sptr.**set_frame_size**(*generic_encoder_sptr self*, *unsigned int frame_size*) → bool

Updates the size of the frame to encode.

The child class MUST implement this function and interpret how theinformation affects the block's behavior. It should also provide bounds checks.

gnuradio.fec.**puncture_bb**(*int puncsize*, *int puncpat*, *int delay=0*) → puncture_bb_sptr

Puncture a stream of unpacked bits.

Puncture a given block of input samples of . The items produced is based on pattern . Basically, if:

This block is designed for unpacked bits - that is, every input sample is a bit, either a 1 or 0. It's possible to use packed bits as symbols, but the puncturing will be done on the symbol level, not the bit level.

is specified as a 32-bit integer that we can convert into the vector _puncpat used in the algorithm above:

Example:

The gr.fec Python module provides a read_bitlist function that can turn a string of a puncture pattern into the correct integer form. The pattern of 0xEF could be specified as fec.readbitlist("11101111"). Also, this allows us to use puncsize=len("11101111") to make sure that our sizes are set up correctly for the pattern we want.

The fec.extended_encoder takes in the puncture pattern directly as a string and uses the readbitlist inside to do the conversion.

Note that due to the above concept, the default setting in the extended encoder of '11' translates into no puncturing.

The parameter delays the application of the puncture pattern. This is equivalent to circularly rotating the by . Note that because of the circular shift, the delay should be between 0 and , but this is not enforced; the effective delay will simply be mod . A negative value here is ignored.

Constructor Specific Documentation:

Constructs a puncture block for unpacked bits.

**Parameters:**
- **puncsize** – Size of block of bits to puncture
- **puncpat** – The puncturing pattern
- **delay** – Delayed the puncturing pattern by shifting it

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

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

declare_sample_delay(puncture_bb_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(puncture_bb_sptr self) -> pmt_vector_float

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

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

puncture_bb_sptr.**pc_output_buffers_full_avg**(*puncture_bb_sptr self*, *int*

*which*) → float

pc_output_buffers_full_avg(puncture_bb_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**puncture_ff**(*int puncsize*, *int puncpat*, *int delay*) → puncture_ff_sptr

Puncture a stream of floats.

For a given block of input samples of , the items produced is based on . Basically, if:

This block is designed for floats, generally 1's and -1's. It's possible to use other float values as symbols, but this is not the expected operation.

> is specified as a 32-bit integer that we can convert into the vector _puncpat used in the algorithm above:

Example:

The gr.fec Python module provides a read_bitlist function that can turn a string of a puncture pattern into the correct integer form. The pattern of 0xEF could be specified as fec.readbitlist("11101111"). Also, this allows us to use puncsize=len("11101111") to make sure that our sizes are set up correctly for the pattern we want.

The fec.extended_encoder takes in the puncture pattern directly as a string and uses the readbitlist inside to do the conversion.

Note that due to the above concept, the default setting in the extended encoder of '11' translates into no puncturing.

The parameter delays the application of the puncture pattern. This is equivalent to circularly rotating the by . Note that because of the circular shift, the delay should be between 0 and , but this is not enforced; the effective delay will simply be mod . A negative value here is ignored.

Constructor Specific Documentation:

Constructs a puncture block for floats.

**Parameters:**
- **puncsize** – Size of block of bits to puncture
- **puncpat** – The puncturing pattern
- **delay** – Delayed the puncturing pattern by shifting it

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

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

declare_sample_delay(puncture_ff_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(puncture_ff_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(puncture_ff_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

gnuradio.fec.**tagged_decoder**(*generic_decoder_sptr my_decoder*, *size_t input_item_size*, *size_t output_item_size*, *std::string const & lengthtagname*, *int mtu=1500*) → tagged_decoder_sptr

General FEC decoding block that takes in a decoder variable object (derived from gr::fec::general_decoder) for use in a flowgraph.

This block uses a decoder variable object (derived from gr::fec::generic_decoder) to decode data within a flowgraph. This block interacts with the general FECAPI architecture to handle all passing all input and output data in a flowgraph. The decoder variable takes care of understanding the requirements, data types and sizes, and boundary conditions of the specific FEC decoding algorithm.

Generally, this block is used within the fec.extended_decoder Python block to handle some input/output formatting issues. In the FECAPI, the decoder variable sets properties like the input and output types and sizes and whether the output is packed or unpacked bytes. The fec.extended_decoder uses this information to set up an gr::hier_block2 structure to make sure the I/O to the variable is handled consistently, such as to make sure all inputs are floats with one soft symbol per item and the outputs are unpacked bytes with the bit in the LSB.

See gr::fec::generic_decoder for detail on what information an FECAPI variable object can set if using this block directly and not as part of the fec.extended_decoder.

Constructor Specific Documentation:

Create the FEC decoder block by taking in the FECAPI decoder object as well as input and output sizes.

**Parameters:**
- **my_decoder** – An FECAPI decoder object (See gr::fec::generic_decoder).
- **input_item_size** – The size of the input items (often the my_decoder object can tell us this).
- **output_item_size** – The size of the output items (often the my_decoder object can tell us this).
- **lengthtagname** – Key name of the tagged stream frame size.
- **mtu** – The Maximum Transmission Unit (MTU) of the output frame that the block will be able to process. Specified in bytes and defaults to 1500.

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

tagged_decoder_sptr.**calculate_output_stream_length**(*tagged_decoder_sptr self*, *gr_vector_int const & ninput_items*) → int

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

declare_sample_delay(tagged_decoder_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(tagged_decoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(tagged_decoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

tagged_decoder_sptr.**work**(*tagged_decoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int

gnuradio.fec.**tagged_encoder**(*generic_encoder_sptr my_encoder*, *size_t input_item_size*, *size_t output_item_size*, *std::string const & lengthtagname*, *int mtu=1500*) → tagged_encoder_sptr

Creates the encoder block for use in GNU Radio flowgraphs from a given FECAPI object derived from the generic_encoder class.

Generally, we would use the fec.extended_encoder Python implementation to instantiate this. The extended_encoder wraps up a few more details, like taking care of puncturing as well as the encoder itself.

Constructor Specific Documentation:

Build the FEC encoder block from an FECAPI encoder object.

**Parameters:**

- **my_encoder** – An FECAPI encoder object child of the generic_encoder class.
- **input_item_size** – size of a block of data for the encoder.
- **output_item_size** – size of a block of data the encoder will produce.
- **lengthtagname** – Key name of the tagged stream frame size.
- **mtu** – The Maximum Transmission Unit (MTU) of the input frame that the block will be able to process. Specified in bytes and defaults to 1500.

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

tagged_encoder_sptr.**calculate_output_stream_length**(*tagged_encoder_sptr self*, *gr_vector_int const & ninput_items*) → int

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

declare_sample_delay(tagged_encoder_sptr self, unsigned int delay)

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

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

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

pc_input_buffers_full_avg(tagged_encoder_sptr self) -> pmt_vector_float

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

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

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

pc_output_buffers_full_avg(tagged_encoder_sptr self) -> pmt_vector_float

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

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

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

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

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

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

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

tagged_encoder_sptr.**work**(*tagged_encoder_sptr self*, *int noutput_items*, *gr_vector_int & ninput_items*, *gr_vector_const_void_star & input_items*, *gr_vector_void_star & output_items*) → int