PhaseNoiseDataProcessorLib

Generated by Doxygen 1.13.2

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 PhaseNoiseDataProcessor Namespace Reference

Classes

- struct Config

Functions

- void convert_samples (int channel, const void *data_bytes)

  Convert raw input sample byte data to voltage values and store to meas buffer for the selected channel.
- void calc_x_values ()

  Calculate and store x-axis frequency output values.
- void calc_y_values (uint32_t num_meas)

  Calculate and store y-axis power spectral density output values.
- void calc_y_values ()
- void dbv_to_dbc (double rf_input_level, double cal_input_level, double cal_output_level)

  Convert y-axis power spectral density output values from dBV/Hz to dBc/Hz based on the specified calibration settings.
- bool read_meas_from_file (char *filename, uint32_t meas_num)

  Read sample data from file for a single measurement and store to data buffers.
- bool write_output_to_file (char *filename)

  Write output data points to file.
- void retrieve_samples (uint32_t set_num, uint32_t smpl_rate_div)

  Retrieve a sample packet containing a single data set from meaurement data buffers and store in sample buffers (only used for fixed window size processing)
- void filter_samples (AntiAliasingFilter &filter_0, AntiAliasingFilter &filter_1)

  Apply a digital filter to sample buffers (only used for fixed window size processing)
- void overwrite_samples (uint32_t packet_num)

  Overwrite selected packet in measurement data buffers with the contents of sample buffers (only used for fixed window size processing)
- void clear_meas_buffer ()

  Set all values in measurement data buffers to 0.
- void clear_output_buffer ()

  Set all values in output buffers to 0.
- void process_fixed_window_size ()

Process sample data from file to produce a set of output data points using a fixed window size.

- void setup (bool cross_corr, double smpl_rate, uint32_t num_meas, uint32_t data_size, uint32_t min_↩ window_size, uint32_t num_bands, uint32_t band_mult, uint32_t band_shift, double conversion_fac)

  Setup processing functions based on the specified parameters.
- void setup (Config new_config)
- const std::vector< double > & get_x_values ()

  Return vector containing x-axis frequency output values.
- const std::vector< double > & get_y_values ()

  Return vector containing y-axis power spectral density output values.
- void process_fixed_smpl_rate ()

  Process sample data from file to produce a set of output data points using a fixed sampling rate.
- void start_fixed_window_size ()

  Start fixed window size processing where sample data is pushed in real time.
- bool push_data_fixed_window_size (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed window size.
- void start_fixed_smpl_rate ()

  Start fixed sample rate processing where sample data is pushed in real time.
- bool push_data_fixed_smpl_rate (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed sample rate.
- void stop ()

  Stop real time processing and release data buffers.

Variables

- std::vector< double > meas_buffer [2]

  Sample buffers for captured data from a single measurement.
- std::vector< double > smpl_buffer [2]

  Sample buffers for a single sample window.
- std::vector< double > fft_re_buffer

  FFT real output buffer.
- std::vector< double > fft_im_buffer

  FFT imaginary output buffer.
- std::vector< double > output_re_buffer

  Buffer to store the sum of real values of cross correlations / auto correlations.
- std::vector< double > output_im_buffer

  Buffer to store the sum of imaginary values of cross correlations / auto correlations.
- std::vector< double > output_x_buffer

  Buffer to store x-axis frequency output values.
- std::vector< double > output_y_buffer

  Buffer to store y-axis power spectral density output values.
- uint32_t meas_count = 0

  Number of measurements that have been processed since processing was started.
- BandData band_data

  Generated data descibing the properties of each measurement band.
- Config config

  Measurement configuration settings.

5.1.1 Function Documentation

5.1.1.1 calc_x_values()

```
void PhaseNoiseDataProcessor::calc_x_values ()
```

Calculate and store x-axis frequency output values.

Values in Hz are calculated based on configuration settings and generated band data. Should be called after selected processing has been started.

5.1.1.2 calc_y_values() [1/2]

```
void PhaseNoiseDataProcessor::calc_y_values ()
```

5.1.1.3 calc_y_values() [2/2]

```
void PhaseNoiseDataProcessor::calc_y_values (
            uint32_t num_meas)
```

Calculate and store y-axis power spectral density output values.

Values in dBV/Hz are calculated based on the combined data of all processed measurements since processing was started. Should be called after measurement data are pushed for real-time monitoring of phase noise.

5.1.1.4 clear_meas_buffer()

```
void PhaseNoiseDataProcessor::clear_meas_buffer ()
```

Set all values in measurement data buffers to 0.

5.1.1.5 clear_output_buffer()

```
void PhaseNoiseDataProcessor::clear_output_buffer ()
```

Set all values in output buffers to 0.

5.1.1.6 convert_samples()

```
void PhaseNoiseDataProcessor::convert_samples (
            int channel,
            const void * data_bytes)
```

Convert raw input sample byte data to voltage values and store to meas buffer for the selected channel.

Byte data should be in the format of consecutive 16-bit half-words each representing one sample. Samples are converted to a voltage in the range of -(*conversion_fac*) to +(*conversion_fac*) using an offset binary representation (not 2's complement).

5.1.1.7 dbv_to_dbc()

```
void PhaseNoiseDataProcessor::dbv_to_dbc (
            double rf_input_level,
            double cal_input_level,
            double cal_output_level)
```

Convert y-axis power spectral density output values from dBV/Hz to dBc/Hz based on the specified calibration settings.

rf_input_level is the level of the carrier signal at the output of the device under test in dBV. *cal_input_level* is the level of the calibration tone applied at the output of the device under test in dBV. *cal_output_level* is the level of the calibration tone measured at the input of the sample capture device in dBV.

5.1.1.8 filter_samples()

```
void PhaseNoiseDataProcessor::filter_samples (
            AntiAliasingFilter & filter_0,
            AntiAliasingFilter & filter_1)
```

Apply a digital filter to sample buffers (only used for fixed window size processing)

This is a 6th-order digital Butterworth filter with a cutoff at (*sampl_rate* / *band_mult*). It allows for under-sampling of the data (such as with the retrieve_samples() function) with reduced aliasing artefacts.

5.1.1.9 get_x_values()

```
const std::vector< double > & PhaseNoiseDataProcessor::get_x_values ()
```

Return vector containing x-axis frequency output values.

5.1.1.10 get_y_values()

```
const std::vector< double > & PhaseNoiseDataProcessor::get_y_values ()
```

Return vector containing y-axis power spectral density output values.

5.1.1.11 overwrite_samples()

```
void PhaseNoiseDataProcessor::overwrite_samples (
            uint32_t packet_num)
```

Overwrite selected packet in measurement data buffers with the contents of sample buffers (only used for fixed window size processing)

5.1.1.12 process_fixed_smpl_rate()

```
void PhaseNoiseDataProcessor::process_fixed_smpl_rate ()
```

Process sample data from file to produce a set of output data points using a fixed sampling rate.

Frequency bands are generated with a fixed sampling rate and variable window size (more accurate).

5.1.1.13 process_fixed_window_size()

```
void PhaseNoiseDataProcessor::process_fixed_window_size ()
```

Process sample data from file to produce a set of output data points using a fixed window size.

Frequency bands are generated with a fixed window size and variable sampling rate (faster).

5.1.1.14 push_data_fixed_smpl_rate()

```
bool PhaseNoiseDataProcessor::push_data_fixed_smpl_rate (
            double * data_0,
            double * data_1)
```

Push sample data for a single measurement to be processed with a fixed sample rate.

5.1.1.15 push_data_fixed_window_size()

```
bool PhaseNoiseDataProcessor::push_data_fixed_window_size (
            double * data_0,
            double * data_1)
```

Push sample data for a single measurement to be processed with a fixed window size.

5.1.1.16 read_meas_from_file()

```
bool PhaseNoiseDataProcessor::read_meas_from_file (
            char * filename,
            uint32_t meas_num)
```

Read sample data from file for a single measurement and store to data buffers.

Byte data should be in the format of consecutive 16-bit half-words each representing one sample. Samples are converted to a voltage in the range of -(*conversion_fac*) to +(*conversion_fac*) using an offset binary representation (not 2's complement).

5.1.1.17 retrieve_samples()

```
void PhaseNoiseDataProcessor::retrieve_samples (
            uint32_t set_num,
            uint32_t smpl_rate_div)
```

Retrieve a sample packet containing a single data set from meaurement data buffers and store in sample buffers (only used for fixed window size processing)

Setting *smpl_rate_div* to a number greater than 1 allows the sample rate to be effectively reduced by skipping samples. (e.g. setting *smpl_rate_div* to 8 will retrieve the first in every 8 samples)

5.1.1.18 setup() [1/2]

```
void PhaseNoiseDataProcessor::setup (
            bool cross_corr,
            double smpl_rate,
            uint32_t num_meas,
            uint32_t data_size,
            uint32_t min_window_size,
            uint32_t num_bands,
            uint32_t band_mult,
            uint32_t band_shift,
            double conversion_fac)
```

Setup processing functions based on the specified parameters.

5.1.1.19 setup() [2/2]

```
void PhaseNoiseDataProcessor::setup (
            Config new_config)
```

5.1.1.20 start_fixed_smpl_rate()

```
void PhaseNoiseDataProcessor::start_fixed_smpl_rate ()
```

Start fixed sample rate processing where sample data is pushed in real time.

5.1.1.21 start_fixed_window_size()

```
void PhaseNoiseDataProcessor::start_fixed_window_size ()
```

Start fixed window size processing where sample data is pushed in real time.

5.1.1.22 stop()

```
void PhaseNoiseDataProcessor::stop ()
```

Stop real time processing and release data buffers.

5.1.1.23 write_output_to_file()

```
bool PhaseNoiseDataProcessor::write_output_to_file (
            char * filename)
```

Write output data points to file.

The file is in in CSV format and contains the properties of each frequency band followed by the list of power spectral density output values against frequency.

5.1.2 Variable Documentation

5.1.2.1 band_data

```
BandData PhaseNoiseDataProcessor::band_data
```

Generated data descibing the properties of each measurement band.

5.1.2.2 config

```
Config PhaseNoiseDataProcessor::config
```

Measurement configuration settings.

5.1.2.3 fft_im_buffer

```
std::vector< double > PhaseNoiseDataProcessor::fft_im_buffer
```

FFT imaginary output buffer.

5.1.2.4 fft_re_buffer

```
std::vector< double > PhaseNoiseDataProcessor::fft_re_buffer
```

FFT real output buffer.

5.1.2.5 meas_buffer

```
std::vector< double > PhaseNoiseDataProcessor::meas_buffer
```

Sample buffers for captured data from a single measurement.

5.1.2.6 meas_count

```
uint32_t PhaseNoiseDataProcessor::meas_count = 0
```

Number of measurements that have been processed since processing was started.

5.1.2.7 output_im_buffer

```
std::vector< double > PhaseNoiseDataProcessor::output_im_buffer
```

Buffer to store the sum of imaginary values of cross correlations / auto correlations.

5.1.2.8 output_re_buffer

```
std::vector< double > PhaseNoiseDataProcessor::output_re_buffer
```

Buffer to store the sum of real values of cross correlations / auto correlations.

5.1.2.9 output_x_buffer

```
std::vector< double > PhaseNoiseDataProcessor::output_x_buffer
```

Buffer to store x-axis frequency output values.

5.1.2.10 output_y_buffer

```
std::vector< double > PhaseNoiseDataProcessor::output_y_buffer
```

Buffer to store y-axis power spectral density output values.

5.1.2.11 smpl_buffer

```
std::vector< double > PhaseNoiseDataProcessor::smpl_buffer
```

Sample buffers for a single sample window.

Chapter 6

Class Documentation

6.1 AntiAliasingFilter Class Reference

6th order Butterworth digital anti-aliasing filter

`#include <filter.h>`

Public Member Functions

- AntiAliasingFilter (int cutoff_div)

  Instatiate the anti-aliasing filter and generate coefficients for the selected cutoff division.
- float pass (double input)

  Passe the next sample into the filter and return the corresponding output sample.
- void reset ()

  Reset all of the internal sample buffers of the filter to 0.
- double output ()

  Return the current output sample.

Protected Attributes

- const double Q1 = 1.931851989228
- const double Q2 = 0.707106562373
- const double Q3 = 0.517637997114
- BiquadFilter filt1
- BiquadFilter filt2
- BiquadFilter filt3
- double output_value = 0

6.1.1 Detailed Description

6th order Butterworth digital anti-aliasing filter

This filter is used to prepare data for under-sampling with reduced aliasing artefacts.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AntiAliasingFilter()

```
AntiAliasingFilter::AntiAliasingFilter (
            int cutoff_div)  [inline]
```

Instatiate the anti-aliasing filter and generate coefficients for the selected cutoff division.

6.1.3 Member Function Documentation

6.1.3.1 output()

```
double AntiAliasingFilter::output ()  [inline]
```

Return the current output sample.

6.1.3.2 pass()

```
float AntiAliasingFilter::pass (
            double input)  [inline]
```

Passe the next sample into the filter and return the corresponding output sample.

6.1.3.3 reset()

```
void AntiAliasingFilter::reset ()  [inline]
```

Reset all of the internal sample buffers of the filter to 0.

6.1.4 Member Data Documentation

6.1.4.1 filt1

```
BiquadFilter AntiAliasingFilter::filt1  [protected]
```

6.1.4.2 filt2

```
BiquadFilter AntiAliasingFilter::filt2  [protected]
```

6.1.4.3 filt3

```
BiquadFilter AntiAliasingFilter::filt3  [protected]
```

6.1.4.4 output_value

```
double AntiAliasingFilter::output_value = 0  [protected]
```

6.1.4.5 Q1

```
const double AntiAliasingFilter::Q1 = 1.931851989228  [protected]
```

6.1.4.6 Q2

```
const double AntiAliasingFilter::Q2 = 0.707106562373  [protected]
```

6.1.4.7 Q3

```
const double AntiAliasingFilter::Q3 = 0.517637997114  [protected]
```

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

- filter.h

6.2 Band Class Reference

Data structure for a single phase noise measurement band.

```
#include <band_data.h>
```

Public Attributes

- uint32_t num_sets = 0

  Number of separate data sets that a single measurement is split into (equal to the number of correlation/autocorrelations performed)
- uint16_t smpl_rate_div = 0

  Ratio of under-sampling relative to the sampling rate of the captured data.
- uint32_t output_start_pos = 0

  Start position in the output data buffers of where output data for this band should be written.
- uint32_t window_size = 0

  The sampling window size for a single data set.
- uint32_t dft_size = 0

  Size of the output of the Fourier transform (only positive frequencies)
- uint32_t dft_start_pos = 0

  Start position of useful output data from Fourier transforms calculated for this band.
- uint32_t dft_end_pos = 0

  End position of useful output data from Fourier transforms calculated for this band.

6.2.1 Detailed Description

Data structure for a single phase noise measurement band.

6.2.2 Member Data Documentation

6.2.2.1 dft_end_pos

```
uint32_t Band::dft_end_pos = 0
```

End position of useful output data from Fourier transforms calculated for this band.

6.2.2.2 dft_size

```
uint32_t Band::dft_size = 0
```

Size of the output of the Fourier transform (only positive frequencies)

6.2.2.3 dft_start_pos

```
uint32_t Band::dft_start_pos = 0
```

Start position of useful output data from Fourier transforms calculated for this band.

6.2.2.4 num_sets

```
uint32_t Band::num_sets = 0
```

Number of separate data sets that a single measurement is split into (equal to the number of correlation/autocorrelations performed)

6.2.2.5 output_start_pos

```
uint32_t Band::output_start_pos = 0
```

Start position in the output data buffers of where output data for this band should be written.

6.2.2.6 smpl_rate_div

```
uint16_t Band::smpl_rate_div = 0
```

Ratio of under-sampling relative to the sampling rate of the captured data.

6.2.2.7 window_size

```
uint32_t Band::window_size = 0
```

The sampling window size for a single data set.

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

- band_data.h

6.3 BandData Class Reference

Generates data for a set of phase noise measurement bands based on the supplied parameters.

```
#include <band_data.h>
```

Public Member Functions

- BandData ()
- BandData (uint16_t num_bands, uint16_t band_mult, uint32_t smpl_data_size, uint32_t min_window_size, uint16_t band_shift, bool fixed_smpl_rate)

  Instatiates the BandData object and generates the Bands.

Public Attributes

- uint32_t smpl_data_size = 0

  Number of samples per channel in a single measurement data capture (must be a power of 2)
- uint16_t band_mult = 1

  Frequency multiplier for subsequent bands (must be a power of 2)
- uint16_t num_bands = 0

  Number of frequency bands to use.
- uint32_t min_window_size = 0

  FFT window size to use for the highest frequency band (must be a power of 2)
- uint16_t band_shift = 0

  Shifts each band down in multiples of band_mult by the specified number (prioritises more correlations over smaller RBW for higher frequencies)
- bool fixed_smpl_rate = true

  True if each band uses a fixed sample rate (more accurate), false if each band uses a fixed window size (faster)
- uint32_t output_size = 0

  Number of output data points that will be generated based on current settings.
- std::vector< Band > bands

  Vector array of band data structures.

6.3.1 Detailed Description

Generates data for a set of phase noise measurement bands based on the supplied parameters.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 BandData() [1/2]

```
BandData::BandData ()  [inline]
```

6.3.2.2 BandData() [2/2]

```
BandData::BandData (
            uint16_t num_bands,
            uint16_t band_mult,
            uint32_t smpl_data_size,
            uint32_t min_window_size,
            uint16_t band_shift,
            bool fixed_smpl_rate)  [inline]
```

Instatiates the BandData object and generates the Bands.

6.3.3 Member Data Documentation

6.3.3.1 band_mult

```
uint16_t BandData::band_mult = 1
```

Frequency multiplier for subsequent bands (must be a power of 2)

6.3.3.2 band_shift

```
uint16_t BandData::band_shift = 0
```

Shifts each band down in multiples of band_mult by the specified number (prioritises more correlations over smaller RBW for higher frequencies)

6.3.3.3 bands

```
std::vector<Band> BandData::bands
```

Vector array of band data structures.

6.3.3.4 fixed_smpl_rate

```
bool BandData::fixed_smpl_rate = true
```

True if each band uses a fixed sample rate (more accurate), false if each band uses a fixed window size (faster)

6.3.3.5 min_window_size

```
uint32_t BandData::min_window_size = 0
```

FFT window size to use for the highest frequency band (must be a power of 2)

6.3.3.6 num_bands

```
uint16_t BandData::num_bands = 0
```

Number of frequency bands to use.

6.3.3.7 output_size

```
uint32_t BandData::output_size = 0
```

Number of output data points that will be generated based on current settings.

6.3.3.8 smpl_data_size

```
uint32_t BandData::smpl_data_size = 0
```

Number of samples per channel in a single measurement data capture (must be a power of 2)

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

- band_data.h

6.4 BiquadFilter Class Reference

Digital biquadratic filter.

```
#include <filter.h>
```

Public Member Functions

- BiquadFilter ()
- BiquadFilter (int cutoff_div, double q)

  Instatiate the biquad filter and generate coefficients for the selected cutoff division and Q factor.
- void generateCoefficients (int cutoff_div, double q)

  Generate coefficients for the selected cutoff division and Q factor.
- double pass (double input)

  Passe the next sample into the filter and return the corresponding output sample.
- void reset ()

  Reset all of the internal sample buffers of the filter to 0.
- double output ()

  Return the current output sample.

Protected Attributes

- double input_coeff [3] = { 0, 0, 0 }
- double feedback_coeff [2] = { 0, 0 }
- double input_buffer [3] = { 0, 0, 0 }
- double feedback_buffer [2] = { 0, 0 }
- double output_value = 0

6.4.1 Detailed Description

Digital biquadratic filter.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 BiquadFilter() [1/2]

```
BiquadFilter::BiquadFilter ()  [inline]
```

6.4.2.2 BiquadFilter() [2/2]

```
BiquadFilter::BiquadFilter (
            int cutoff_div,
            double q)  [inline]
```

Instatiate the biquad filter and generate coefficients for the selected cutoff division and Q factor.

6.4.3 Member Function Documentation

6.4.3.1 generateCoefficients()

```
void BiquadFilter::generateCoefficients (
            int cutoff_div,
            double q)  [inline]
```

Generate coefficients for the selected cutoff division and Q factor.

cutoff_div sets the cutoff frequency of the filter as an integer division relative to the sampling frequency. q sets the quality factor of the resonant peak.

6.4.3.2 output()

```
double BiquadFilter::output ()  [inline]
```

Return the current output sample.

6.4.3.3 pass()

```
double BiquadFilter::pass (
            double input)  [inline]
```

Passe the next sample into the filter and return the corresponding output sample.

6.4.3.4 reset()

```
void BiquadFilter::reset ()  [inline]
```

Reset all of the internal sample buffers of the filter to 0.

6.4.4 Member Data Documentation

6.4.4.1 feedback_buffer

```
double BiquadFilter::feedback_buffer[2] = { 0, 0 }  [protected]
```

6.4.4.2 feedback_coeff

```
double BiquadFilter::feedback_coeff[2] = { 0, 0 }  [protected]
```

6.4.4.3 input_buffer

```
double BiquadFilter::input_buffer[3] = { 0, 0, 0 }  [protected]
```

6.4.4.4 input_coeff

```
double BiquadFilter::input_coeff[3] = { 0, 0, 0 }  [protected]
```

6.4.4.5 output_value

```
double BiquadFilter::output_value = 0  [protected]
```

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

- filter.h

6.5 PhaseNoiseDataProcessor::Config Struct Reference

```
#include <PhaseNoiseDataProcessorLib.h>
```

Public Attributes

- bool cross_corr = true

  True for cross correlation, false for single channel.
- double smpl_rate = 524288.0

  Sample rate of the captured data.
- uint32_t num_meas = 1

  Number of measurements to process.
- uint32_t data_size = 4194304

  Number of samples captured per channel for a single measurement (must be a power of 2)
- uint32_t min_window_size = 524288

  Minimum size of the FFT window (must be a power of 2)
- uint32_t num_bands = 1

  Number of frequency bands to use.
- uint32_t band_mult = 8

  Frequency multiplier for subsequent bands (must be a power of 2)
- uint32_t band_shift = 2

  Shifts each band down in multiples of band_mult by the specified number (prioritises more correlations over smaller RBW for higher frequencies)
- double conversion_fac = 0.0025

  Maximum input voltage reference (bipolar)
- bool window_en = false

  True to apply flat top window to FFT input data (NOT YET IMPLEMENTED), false for no windowing (rectangular)

6.5.1 Member Data Documentation

6.5.1.1 band_mult

```
uint32_t PhaseNoiseDataProcessor::Config::band_mult = 8
```

Frequency multiplier for subsequent bands (must be a power of 2)

6.5.1.2 band_shift

```
uint32_t PhaseNoiseDataProcessor::Config::band_shift = 2
```

Shifts each band down in multiples of band_mult by the specified number (prioritises more correlations over smaller RBW for higher frequencies)

6.5.1.3 conversion_fac

```
double PhaseNoiseDataProcessor::Config::conversion_fac = 0.0025
```

Maximum input voltage reference (bipolar)

6.5.1.4 cross_corr

```
bool PhaseNoiseDataProcessor::Config::cross_corr = true
```

True for cross correlation, false for single channel.

6.5.1.5 data_size

```
uint32_t PhaseNoiseDataProcessor::Config::data_size = 4194304
```

Number of samples captured per channel for a single measurement (must be a power of 2)

6.5.1.6 min_window_size

```
uint32_t PhaseNoiseDataProcessor::Config::min_window_size = 524288
```

Minimum size of the FFT window (must be a power of 2)

6.5.1.7 num_bands

```
uint32_t PhaseNoiseDataProcessor::Config::num_bands = 1
```

Number of frequency bands to use.

6.5.1.8 num_meas

```
uint32_t PhaseNoiseDataProcessor::Config::num_meas = 1
```

Number of measurements to process.

6.5.1.9 smpl_rate

```
double PhaseNoiseDataProcessor::Config::smpl_rate = 524288.0
```

Sample rate of the captured data.

6.5.1.10 window_en

```
bool PhaseNoiseDataProcessor::Config::window_en = false
```

True to apply flat top window to FFT input data (NOT YET IMPLEMENTED), false for no windowing (rectangular)

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

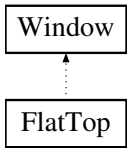
- PhaseNoiseDataProcessorLib.h

6.6 FlatTop Class Reference

Flat top window for minimal scalloping loss.

`#include <window.h>`

Inheritance diagram for FlatTop:



Public Member Functions

- FlatTop (size_t size)

Protected Member Functions

- double rbwFactor () override

  Calculate and return the resolution bandwidth scaling factor.
- void generate ()

  Generate the data points.

6.6.1 Detailed Description

Flat top window for minimal scalloping loss.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 FlatTop()

```
FlatTop::FlatTop (
            size_t size)  [inline]
```

6.6.3 Member Function Documentation

6.6.3.1 generate()

```
void FlatTop::generate ()  [inline], [protected], [virtual]
```

Generate the data points.

Implements Window.

6.6.3.2 rbwFactor()

```
double FlatTop::rbwFactor ()  [inline], [override], [protected], [virtual]
```

Calculate and return the resolution bandwidth scaling factor.

Implements Window.

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

- window.h

6.7 TrigTable Class Reference

Generates a lookup table of precalculated sine/cosine values.

```
#include <trig_table.h>
```

Public Member Functions

- TrigTable (uint32_t size)
- TrigTable ()
- double sin_lookup (uint64_t pos)

  Look up sine value at the given position.
- double cos_lookup (uint64_t pos)

  Look up cosine value at the given position.

6.7.1 Detailed Description

Generates a lookup table of precalculated sine/cosine values.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 TrigTable() [1/2]

```
TrigTable::TrigTable (
            uint32_t size)  [inline]
```

6.7.2.2 TrigTable() [2/2]

```
TrigTable::TrigTable ()  [inline]
```

6.7.3 Member Function Documentation

6.7.3.1 cos_lookup()

```
double TrigTable::cos_lookup (
            uint64_t pos)  [inline]
```

Look up cosine value at the given position.

6.7.3.2 sin_lookup()

```
double TrigTable::sin_lookup (
            uint64_t pos)  [inline]
```

Look up sine value at the given position.

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

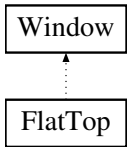
- trig_table.h

6.8 Window Class Reference

Abstract class for a sampling window of given size, which defines the shape and resolution bandwidth scaling factor.

```
#include <window.h>
```

Inheritance diagram for Window:



Public Member Functions

- Window (size_t size)

Protected Member Functions

- virtual double rbwFactor ()=0

  Calculate and return the resolution bandwidth scaling factor.
- virtual void generate ()=0

  Generate the data points.

Protected Attributes

- size_t size = 0

  Size of the window in samples.
- double rbw_factor

  Resolution bandwidth scaling factor (equal to 1.0 for a rectangular window)
- std::vector< double > data

  Data points that define the shape of of the window (all 1.0s for a rectangular window)

6.8.1 Detailed Description

Abstract class for a sampling window of given size, which defines the shape and resolution bandwidth scaling factor.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 Window()

```
Window::Window (
            size_t size)  [inline]
```

6.8.3 Member Function Documentation

6.8.3.1 generate()

```
virtual void Window::generate ()  [protected], [pure virtual]
```

Generate the data points.

Implemented in FlatTop.

6.8.3.2 rbwFactor()

```
virtual double Window::rbwFactor ()  [protected], [pure virtual]
```

Calculate and return the resolution bandwidth scaling factor.

Implemented in FlatTop.

6.8.4 Member Data Documentation

6.8.4.1 data

```
std::vector<double> Window::data  [protected]
```

Data points that define the shape of of the window (all 1.0s for a rectangular window)

6.8.4.2 rbw_factor

```
double Window::rbw_factor  [protected]
```

Resolution bandwidth scaling factor (equal to 1.0 for a rectangular window)

6.8.4.3 size

```
size_t Window::size = 0  [protected]
```

Size of the window in samples.

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

- window.h

Chapter 7

File Documentation

7.1 band_data.h File Reference

```
#include <vector>
```

Classes

- class Band

  Data structure for a single phase noise measurement band.
- class BandData

  Generates data for a set of phase noise measurement bands based on the supplied parameters.

7.2 band_data.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <vector>
00004
00006 class Band
00007 {
00008 public:
00009
00010     uint32_t num_sets = 0;
00011     uint16_t smpl_rate_div = 0;
00012     uint32_t output_start_pos = 0;
00013     uint32_t window_size = 0;
00014     uint32_t dft_size = 0;
00015     uint32_t dft_start_pos = 0;
00016     uint32_t dft_end_pos = 0;
00017 };
00018
00019
00021 class BandData
00022 {
00023 public:
00024
00025     uint32_t smpl_data_size = 0;
00026     uint16_t band_mult = 1;
00027     uint16_t num_bands = 0;
00028     uint32_t min_window_size = 0;
00029     uint16_t band_shift = 0;
00030     bool fixed_smpl_rate = true;
00031
00032     uint32_t output_size = 0;
```

```
00033     std::vector<Band> bands;
00034
00035     BandData() {};
00036
00038     BandData(uint16_t num_bands, uint16_t band_mult, uint32_t smpl_data_size, uint32_t
      min_window_size, uint16_t band_shift, bool fixed_smpl_rate)
00039     {
00040         this->smpl_data_size = smpl_data_size;
00041         this->band_mult = band_mult;
00042         this->num_bands = num_bands;
00043         this->min_window_size = min_window_size;
00044         this->band_shift = band_shift;
00045         this->fixed_smpl_rate = fixed_smpl_rate;
00046
00047
00048         uint16_t smpl_rate_div = 1;
00049         uint32_t num_sets = smpl_data_size / min_window_size;
00050
00051         this->bands.resize(num_bands);
00052
00053         for (int b = (this->num_bands - 1); b >= 0; b--)
00054         {
00055             this->bands[b].smpl_rate_div = smpl_rate_div;
00056             this->bands[b].num_sets = num_sets;
00057
00058             if(!fixed_smpl_rate) smpl_rate_div *= this->band_mult;
00059             num_sets /= this->band_mult;
00060         }
00061
00062         uint32_t pos_count = 0;
00063         uint16_t shift = 1;
00064         for (int i = 0; i < band_shift; i++) shift *= this->band_mult;
00065
00066         for (int b = 0; b < this->num_bands; b++)
00067         {
00068             this->bands[b].window_size = fixed_smpl_rate ? (smpl_data_size / this->bands[b].num_sets)
      : min_window_size;
00069             this->bands[b].dft_size = (this->bands[b].window_size / 2) + 1;
00070
00071             this->bands[b].output_start_pos = pos_count;
00072             this->bands[b].dft_start_pos = (b == 0) ? 0 : (min_window_size / (2 * band_mult * shift));
00073             this->bands[b].dft_end_pos = (b == (this->num_bands - 1)) ? (min_window_size / 2) :
      ((min_window_size / (2 * shift)) - 1);
00074             pos_count += (this->bands[b].dft_end_pos - this->bands[b].dft_start_pos + 1);
00075         }
00076
00077         this->output_size = pos_count;
00078     }
00079 };
```

7.3 fft.cpp File Reference

```
#include "fft.h"
```

Functions

- void set_data_size (uint32_t size)

  Sets the size of the FFT window to be used and precalculates sine/cosine values.
- void dft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Calculate the DFT for an array of real and imaginary data points.
- void rdft (double *x_re, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Calculate the DFT for an array of real data points.
- void fft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t data_size, uint32_t div)
- void fft (double *x_re, double *x_im, double *out_re, double *out_im)

  Calculate the FFT for an array of real and imaginary data points.
- void pfft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void pfft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_num)

Calculate the partial FFT for an array of real and imaginary data points.

- void pfft_dual_real (double *x_re1, double *x_re2, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void pfft_dual_real (double *x_re1, double *x_re2, double *out_re, double *out_im, uint32_t k_num)

  Calculate the partial FFT for two arrays of real data points to be used as the input to a dual-real FFT cross-correlation function.
- void rfft (double *x_re, double *out_re, double *out_im, uint32_t data_size, uint32_t div)
- void rfft (double *x_re, double *out_re, double *out_im)

  Calculate the FFT for an array of real data points.
- void rpfft (double *x_re, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void rpfft (double *x_re, double *out_re, double *out_im, uint32_t k_num)

  Calculate the partial FFT for an array of real data points.
- void cross_corr_dual_real (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Performs the cross correlation of an FFT of two sets of real data.
- void cross_corr_dual_real (double *x_re, double *x_im, double *out_re, double *out_im)
- void cross_corr (double *f_re, double *f_im, double *g_re, double *g_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Performs the cross correlation of complex data sets f and g
- void cross_corr (double *f_re, double *f_im, double *g_re, double *g_im, double *out_re, double *out_im)
- void auto_corr_real (double *x_re, double *x_im, double *out_re, uint32_t k_start, uint32_t k_stop)

  Performs the auto correlation of the result of an FFT of real data.
- void auto_corr_real (double *x_re, double *x_im, double *out_re)

Variables

- uint32_t g_data_size = 1
- TrigTable g_trig = TrigTable()

7.3.1 Function Documentation

7.3.1.1 auto_corr_real() [1/2]

```
void auto_corr_real (
            double * x_re,
            double * x_im,
            double * out_re)
```

7.3.1.2 auto_corr_real() [2/2]

```
void auto_corr_real (
            double * x_re,
            double * x_im,
            double * out_re,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the auto correlation of the result of an FFT of real data.

Only output values in the range *k_start* to *k_stop* are calculated. The output values are the sum of the positive and negative frequency components (i.e. multiplied by 2).

7.3.1.3 cross_corr() [1/2]

```
void cross_corr (
            double * f_re,
            double * f_im,
            double * g_re,
            double * g_im,
            double * out_re,
            double * out_im)
```

7.3.1.4 cross_corr() [2/2]

```
void cross_corr (
            double * f_re,
            double * f_im,
            double * g_re,
            double * g_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the cross correlation of complex data sets *f* and *g*

Only output values in the range *k_start* to *k_stop* are calculated.

7.3.1.5 cross_corr_dual_real() [1/2]

```
void cross_corr_dual_real (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im)
```

7.3.1.6 cross_corr_dual_real() [2/2]

```
void cross_corr_dual_real (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the cross correlation of an FFT of two sets of real data.

The input data is the FFT generated where one of the real data sets is used as the real part of the FFT input, and the other real data set is used as the imaginary part of the FFT input. Only output values in the range *k_start* to *k_stop* are calculated. Unlike the standard cross_corr() function, the output values are the sum of the positive and negative frequency components (i.e. multiplied by 2).

7.3.1.7 dft()

```
void dft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Calculate the DFT for an array of real and imaginary data points.

Only output values in the range *k_start* to *k_stop* are calculated.

7.3.1.8 fft() [1/2]

```
void fft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im)
```

Calculate the FFT for an array of real and imaginary data points.

7.3.1.9 fft() [2/2]

```
void fft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t data_size,
            uint32_t div)
```

7.3.1.10 pfft() [1/2]

```
void pfft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for an array of real and imaginary data points.

Only the first *k_num* output values are calculated.

7.3.1.11 pfft() [2/2]

```
void pfft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.3.1.12 pfft_dual_real() [1/2]

```
void pfft_dual_real (
            double * x_re1,
            double * x_re2,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for two arrays of real data points to be used as the input to a dual-real FFT cross-correlation function.

//! Only the first *k_num* output values are calculated.

7.3.1.13 pfft_dual_real() [2/2]

```
void pfft_dual_real (
            double * x_re1,
            double * x_re2,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.3.1.14 rdft()

```
void rdft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Calculate the DFT for an array of real data points.

Only output values in the range *k_start* to *k_stop* are calculated.

7.3.1.15 rfft() [1/2]

```
void rfft (
            double * x_re,
            double * out_re,
            double * out_im)
```

Calculate the FFT for an array of real data points.

7.3.1.16 rfft() [2/2]

```
void rfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t data_size,
            uint32_t div)
```

7.3.1.17 rpfft() [1/2]

```
void rpfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for an array of real data points.

Only the first *k_num* output values are calculated.

7.3.1.18 rpfft() [2/2]

```
void rpfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.3.1.19 set_data_size()

```
void set_data_size (
            uint32_t size)
```

Sets the size of the FFT window to be used and precalculates sine/cosine values.

Must be set before any transform functions are performed.

7.3.2 Variable Documentation

7.3.2.1 g_data_size

```
uint32_t g_data_size = 1
```

7.3.2.2 g_trig

```
TrigTable g_trig = TrigTable()
```

7.4 fft.h File Reference

```
#include <stdint.h>
#include "trig_table.h"
```

Functions

- void set_data_size (uint32_t size)

  Sets the size of the FFT window to be used and precalculates sine/cosine values.
- void dft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Calculate the DFT for an array of real and imaginary data points.
- void rdft (double *x_re, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Calculate the DFT for an array of real data points.
- void fft (double *x_re, double *x_im, double *out_re, double *out_im)

  Calculate the FFT for an array of real and imaginary data points.
- void fft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t data_size, uint32_t div)
- void pfft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_num)

  Calculate the partial FFT for an array of real and imaginary data points.
- void pfft (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void pfft_dual_real (double *x_re1, double *x_re2, double *out_re, double *out_im, uint32_t k_num)

  Calculate the partial FFT for two arrays of real data points to be used as the input to a dual-real FFT cross-correlation function.
- void pfft_dual_real (double *x_re1, double *x_re2, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void rfft (double *x_re, double *out_re, double *out_im)

  Calculate the FFT for an array of real data points.
- void rfft (double *x_re, double *out_re, double *out_im, uint32_t data_size, uint32_t div)
- void rpfft (double *x_re, double *out_re, double *out_im, uint32_t k_num)

  Calculate the partial FFT for an array of real data points.
- void rpfft (double *x_re, double *out_re, double *out_im, uint32_t k_num, uint32_t data_size, uint32_t div)
- void cross_corr (double *f_re, double *f_im, double *g_re, double *g_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Performs the cross correlation of complex data sets f and g
- void cross_corr (double *f_re, double *f_im, double *g_re, double *g_im, double *out_re, double *out_im)
- void cross_corr_dual_real (double *x_re, double *x_im, double *out_re, double *out_im, uint32_t k_start, uint32_t k_stop)

  Performs the cross correlation of an FFT of two sets of real data.
- void cross_corr_dual_real (double *x_re, double *x_im, double *out_re, double *out_im)
- void auto_corr_real (double *x_re, double *x_im, double *out_re, uint32_t k_start, uint32_t k_stop)

  Performs the auto correlation of the result of an FFT of real data.
- void auto_corr_real (double *x_re, double *x_im, double *out_re)

Variables

- uint32_t g_data_size
- TrigTable g_trig

7.4.1 Function Documentation

7.4.1.1 auto_corr_real() [1/2]

```
void auto_corr_real (
            double * x_re,
            double * x_im,
            double * out_re)
```

7.4.1.2 auto_corr_real() [2/2]

```
void auto_corr_real (
            double * x_re,
            double * x_im,
            double * out_re,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the auto correlation of the result of an FFT of real data.

Only output values in the range *k_start* to *k_stop* are calculated. The output values are the sum of the positive and negative frequency components (i.e. multiplied by 2).

7.4.1.3 cross_corr() [1/2]

```
void cross_corr (
            double * f_re,
            double * f_im,
            double * g_re,
            double * g_im,
            double * out_re,
            double * out_im)
```

7.4.1.4 cross_corr() [2/2]

```
void cross_corr (
            double * f_re,
            double * f_im,
            double * g_re,
            double * g_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the cross correlation of complex data sets *f* and *g*

Only output values in the range *k_start* to *k_stop* are calculated.

7.4.1.5 cross_corr_dual_real() [1/2]

```
void cross_corr_dual_real (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im)
```

7.4.1.6 cross_corr_dual_real() [2/2]

```
void cross_corr_dual_real (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Performs the cross correlation of an FFT of two sets of real data.

The input data is the FFT generated where one of the real data sets is used as the real part of the FFT input, and the other real data set is used as the imaginary part of the FFT input. Only output values in the range *k_start* to *k_stop* are calculated. Unlike the standard cross_corr() function, the output values are the sum of the positive and negative frequency components (i.e. multiplied by 2).

7.4.1.7 dft()

```
void dft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Calculate the DFT for an array of real and imaginary data points.

Only output values in the range *k_start* to *k_stop* are calculated.

7.4.1.8 fft() [1/2]

```
void fft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im)
```

Calculate the FFT for an array of real and imaginary data points.

7.4.1.9 fft() [2/2]

```
void fft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t data_size,
            uint32_t div)
```

7.4.1.10 pfft() [1/2]

```
void pfft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for an array of real and imaginary data points.

Only the first *k_num* output values are calculated.

7.4.1.11 pfft() [2/2]

```
void pfft (
            double * x_re,
            double * x_im,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.4.1.12 pfft_dual_real() [1/2]

```
void pfft_dual_real (
            double * x_re1,
            double * x_re2,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for two arrays of real data points to be used as the input to a dual-real FFT cross-correlation function.

//! Only the first *k_num* output values are calculated.

7.4.1.13 pfft_dual_real() [2/2]

```
void pfft_dual_real (
            double * x_re1,
            double * x_re2,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.4.1.14 rdft()

```
void rdft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_start,
            uint32_t k_stop)
```

Calculate the DFT for an array of real data points.

Only output values in the range *k_start* to *k_stop* are calculated.

7.4.1.15 rfft() [1/2]

```
void rfft (
            double * x_re,
            double * out_re,
            double * out_im)
```

Calculate the FFT for an array of real data points.

7.4.1.16 rfft() [2/2]

```
void rfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t data_size,
            uint32_t div)
```

7.4.1.17 rpfft() [1/2]

```
void rpfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_num)
```

Calculate the partial FFT for an array of real data points.

Only the first *k_num* output values are calculated.

7.4.1.18 rpfft() [2/2]

```
void rpfft (
            double * x_re,
            double * out_re,
            double * out_im,
            uint32_t k_num,
            uint32_t data_size,
            uint32_t div)
```

7.4.1.19 set_data_size()

```
void set_data_size (
            uint32_t size)
```

Sets the size of the FFT window to be used and precalculates sine/cosine values.

Must be set before any transform functions are performed.

7.4.2 Variable Documentation

7.4.2.1 g_data_size

```
uint32_t g_data_size  [extern]
```

7.4.2.2 g_trig

```
TrigTable g_trig  [extern]
```

7.5 fft.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <stdint.h>
00004 #include "trig_table.h"
00005
00006 extern uint32_t g_data_size;
00007 extern TrigTable g_trig;
00008
00010
00012 void set_data_size(uint32_t size);
00013
00015
00017 void dft(double* x_re, double* x_im, double* out_re, double* out_im, uint32_t k_start, uint32_t
      k_stop);
00018
00020
00022 void rdft(double* x_re, double* out_re, double* out_im, uint32_t k_start, uint32_t k_stop);
00023
00025 void fft(double* x_re, double* x_im, double* out_re, double* out_im);
00026 void fft(double* x_re, double* x_im, double* out_re, double* out_im, uint32_t data_size, uint32_t
      div);
00027
00029
00031 void pfft(double* x_re, double* x_im, double* out_re, double* out_im, uint32_t k_num);
```

```
00032 void pfft(double* x_re, double* x_im, double* out_re, double* out_im, uint32_t k_num, uint32_t
      data_size, uint32_t div);
00033
00035
00037 void pfft_dual_real(double* x_re1, double* x_re2, double* out_re, double* out_im, uint32_t k_num);
00038 void pfft_dual_real(double* x_re1, double* x_re2, double* out_re, double* out_im, uint32_t k_num,
      uint32_t data_size, uint32_t div);
00039
00041 void rfft(double* x_re, double* out_re, double* out_im);
00042 void rfft(double* x_re, double* out_re, double* out_im, uint32_t data_size, uint32_t div);
00043
00045
00047 void rpfft(double* x_re, double* out_re, double* out_im, uint32_t k_num);
00048 void rpfft(double* x_re, double* out_re, double* out_im, uint32_t k_num, uint32_t data_size, uint32_t
      div);
00049
00051
00053 void cross_corr(double* f_re, double* f_im, double* g_re, double* g_im, double* out_re, double*
      out_im, uint32_t k_start, uint32_t k_stop);
00054 void cross_corr(double* f_re, double* f_im, double* g_re, double* g_im, double* out_re, double*
      out_im);
00055
00057
00061 void cross_corr_dual_real(double* x_re, double* x_im, double* out_re, double* out_im, uint32_t
      k_start, uint32_t k_stop);
00062 void cross_corr_dual_real(double* x_re, double* x_im, double* out_re, double* out_im);
00063
00065
00068 void auto_corr_real(double* x_re, double* x_im, double* out_re, uint32_t k_start, uint32_t k_stop);
00069 void auto_corr_real(double* x_re, double* x_im, double* out_re);
```

7.6 filter.h File Reference

```
#include <cmath>
#include <stdint.h>
```

Classes

- class BiquadFilter

  Digital biquadratic filter.
- class AntiAliasingFilter

  6th order Butterworth digital anti-aliasing filter

Macros

- #define _USE_MATH_DEFINES

7.6.1 Macro Definition Documentation

7.6.1.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

7.7 filter.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #define _USE_MATH_DEFINES
00004 #include <cmath>
00005 #include <stdint.h>
00006
00008 class BiquadFilter
00009 {
00010     protected:
00011
00012         double input_coeff[3] = { 0, 0, 0 };
00013         double feedback_coeff[2] = { 0, 0 };
00014
00015         double input_buffer[3] = { 0, 0, 0 };
00016         double feedback_buffer[2] = { 0, 0 };
00017
00018         double output_value = 0;
00019
00020     public:
00021
00022         BiquadFilter() {};
00023
00025         BiquadFilter(int cutoff_div, double q)
00026         {
00027             generateCoefficients(cutoff_div, q);
00028         }
00029
00031
00034         void generateCoefficients(int cutoff_div, double q)
00035         {
00036             double k_wa = 1.0 / tanf(M_PI / cutoff_div);
00037             double k_wa_sqr = powf(k_wa, 2);
00038
00039             double fac = 1.0 / (k_wa_sqr + (k_wa / q) + 1.0);
00040
00041             input_coeff[0] = fac;
00042             input_coeff[1] = 2 * fac;
00043             input_coeff[2] = fac;
00044
00045             feedback_coeff[0] = (2.0 * k_wa_sqr - 2.0) * fac;
00046             feedback_coeff[1] = -(k_wa_sqr - (k_wa / q) + 1.0) * fac;
00047         }
00048
00050         double pass(double input)
00051         {
00052             input_buffer[2] = input_buffer[1];
00053             input_buffer[1] = input_buffer[0];
00054             input_buffer[0] = input;
00055
00056             feedback_buffer[1] = feedback_buffer[0];
00057             feedback_buffer[0] = output_value;
00058             output_value = (input_buffer[0] * input_coeff[0]) + (input_buffer[1] * input_coeff[1]) +
    (input_buffer[2] * input_coeff[2]) +
00059                                 (feedback_buffer[0] * feedback_coeff[0]) + (feedback_buffer[1] *
    feedback_coeff[1]);
00060
00061             return output();
00062         }
00063
00065         void reset()
00066         {
00067             input_buffer[2] = 0;
00068             input_buffer[1] = 0;
00069             input_buffer[0] = 0;
00070
00071             feedback_buffer[1] = 0;
00072             feedback_buffer[0] = 0;
00073
00074             output_value = 0;
00075         }
00076
00078         double output() { return output_value; };
00079 };
00080
00082
00084 class AntiAliasingFilter
00085 {
00086     protected:
00087         const double Q1 = 1.931851989228;
00088         const double Q2 = 0.707106562373;
00089         const double Q3 = 0.517637997114;
00090
```

```
00091         BiquadFilter filt1;
00092         BiquadFilter filt2;
00093         BiquadFilter filt3;
00094 
00095         double output_value = 0;
00096 
00097     public:
00098 
00100         AntiAliasingFilter(int cutoff_div)
00101         {
00102             filt1.generateCoefficients(cutoff_div, Q1);
00103             filt2.generateCoefficients(cutoff_div, Q2);
00104             filt3.generateCoefficients(cutoff_div, Q3);
00105         }
00106 
00108         float pass(double input)
00109         {
00110             output_value = filt3.pass(filt2.pass(filt1.pass(input)));
00111             return output();
00112         }
00113 
00115         void reset()
00116         {
00117             filt1.reset();
00118             filt2.reset();
00119             filt3.reset();
00120 
00121             output_value = 0;
00122         }
00123 
00125         double output() { return output_value; };
00126 };
```

7.8 PhaseNoiseDataProcessorLib.cpp File Reference

```
#include "PhaseNoiseDataProcessorLib.h"
```

Namespaces

- namespace PhaseNoiseDataProcessor

Functions

- void PhaseNoiseDataProcessor::convert_samples (int channel, const void *data_bytes)

  Convert raw input sample byte data to voltage values and store to meas buffer for the selected channel.
- void PhaseNoiseDataProcessor::calc_x_values ()

  Calculate and store x-axis frequency output values.
- void PhaseNoiseDataProcessor::calc_y_values (uint32_t num_meas)

  Calculate and store y-axis power spectral density output values.
- void PhaseNoiseDataProcessor::calc_y_values ()
- void PhaseNoiseDataProcessor::dbv_to_dbc (double rf_input_level, double cal_input_level, double cal_↩ output_level)

  Convert y-axis power spectral density output values from dBV/Hz to dBc/Hz based on the specified calibration settings.
- bool PhaseNoiseDataProcessor::read_meas_from_file (char *filename, uint32_t meas_num)

  Read sample data from file for a single measurement and store to data buffers.
- bool PhaseNoiseDataProcessor::write_output_to_file (char *filename)

  Write output data points to file.
- void PhaseNoiseDataProcessor::retrieve_samples (uint32_t set_num, uint32_t smpl_rate_div)

  Retrieve a sample packet containing a single data set from meaurement data buffers and store in sample buffers (only used for fixed window size processing)

- void PhaseNoiseDataProcessor::filter_samples (AntiAliasingFilter &filter_0, AntiAliasingFilter &filter_1)

  Apply a digital filter to sample buffers (only used for fixed window size processing)
- void PhaseNoiseDataProcessor::overwrite_samples (uint32_t packet_num)

  Overwrite selected packet in measurement data buffers with the contents of sample buffers (only used for fixed window size processing)
- void PhaseNoiseDataProcessor::clear_meas_buffer ()

  Set all values in measurement data buffers to 0.
- void PhaseNoiseDataProcessor::clear_output_buffer ()

  Set all values in output buffers to 0.
- void PhaseNoiseDataProcessor::process_fixed_window_size ()

  Process sample data from file to produce a set of output data points using a fixed window size.
- void PhaseNoiseDataProcessor::setup (bool cross_corr, double smpl_rate, uint32_t num_meas, uint32_t data_size, uint32_t min_window_size, uint32_t num_bands, uint32_t band_mult, uint32_t band_shift, double conversion_fac)

  Setup processing functions based on the specified parameters.
- void PhaseNoiseDataProcessor::setup (Config new_config)
- const std::vector< double > & PhaseNoiseDataProcessor::get_x_values ()

  Return vector containing x-axis frequency output values.
- const std::vector< double > & PhaseNoiseDataProcessor::get_y_values ()

  Return vector containing y-axis power spectral density output values.
- void PhaseNoiseDataProcessor::process_fixed_smpl_rate ()

  Process sample data from file to produce a set of output data points using a fixed sampling rate.
- void PhaseNoiseDataProcessor::start_fixed_window_size ()

  Start fixed window size processing where sample data is pushed in real time.
- bool PhaseNoiseDataProcessor::push_data_fixed_window_size (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed window size.
- void PhaseNoiseDataProcessor::start_fixed_smpl_rate ()

  Start fixed sample rate processing where sample data is pushed in real time.
- bool PhaseNoiseDataProcessor::push_data_fixed_smpl_rate (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed sample rate.
- void PhaseNoiseDataProcessor::stop ()

  Stop real time processing and release data buffers.

Variables

- std::vector< double > PhaseNoiseDataProcessor::meas_buffer [2]

  Sample buffers for captured data from a single measurement.
- std::vector< double > PhaseNoiseDataProcessor::smpl_buffer [2]

  Sample buffers for a single sample window.
- std::vector< double > PhaseNoiseDataProcessor::fft_re_buffer

  FFT real output buffer.
- std::vector< double > PhaseNoiseDataProcessor::fft_im_buffer

  FFT imaginary output buffer.
- std::vector< double > PhaseNoiseDataProcessor::output_re_buffer

  Buffer to store the sum of real values of cross correlations / auto correlations.
- std::vector< double > PhaseNoiseDataProcessor::output_im_buffer

  Buffer to store the sum of imaginary values of cross correlations / auto correlations.
- std::vector< double > PhaseNoiseDataProcessor::output_x_buffer

  Buffer to store x-axis frequency output values.
- std::vector< double > PhaseNoiseDataProcessor::output_y_buffer

  Buffer to store y-axis power spectral density output values.

- uint32_t PhaseNoiseDataProcessor::meas_count = 0

  Number of measurements that have been processed since processing was started.
- BandData PhaseNoiseDataProcessor::band_data

  Generated data descibing the properties of each measurement band.
- Config PhaseNoiseDataProcessor::config

  Measurement configuration settings.

7.9 PhaseNoiseDataProcessorLib.h File Reference

```
#include <cmath>
#include <stdio.h>
#include <stdint.h>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "band_data.h"
#include "fft.h"
#include "filter.h"
```

Classes

- struct PhaseNoiseDataProcessor::Config

Namespaces

- namespace PhaseNoiseDataProcessor

Macros

- #define _USE_MATH_DEFINES
- #define MAX_DATA_SIZE 268435456
- #define MAX_WINDOW_SIZE 268435456
- #define MAX_MEAS 65536
- #define SMPL_FILENAME "samples.bin"
- #define OUTPUT_FILENAME "output.csv"

Functions

- void PhaseNoiseDataProcessor::convert_samples (int channel, const void *data_bytes)

  Convert raw input sample byte data to voltage values and store to meas buffer for the selected channel.
- void PhaseNoiseDataProcessor::calc_x_values ()

  Calculate and store x-axis frequency output values.
- void PhaseNoiseDataProcessor::calc_y_values (uint32_t num_meas)

  Calculate and store y-axis power spectral density output values.
- void PhaseNoiseDataProcessor::calc_y_values ()
- void PhaseNoiseDataProcessor::dbv_to_dbc (double rf_input_level, double cal_input_level, double cal_↩ output_level)

Convert y-axis power spectral density output values from dBV/Hz to dBc/Hz based on the specified calibration settings.

- bool PhaseNoiseDataProcessor::read_meas_from_file (char *filename, uint32_t meas_num)

  Read sample data from file for a single measurement and store to data buffers.
- void PhaseNoiseDataProcessor::retrieve_samples (uint32_t set_num, uint32_t smpl_rate_div)

  Retrieve a sample packet containing a single data set from meaurement data buffers and store in sample buffers (only used for fixed window size processing)
- void PhaseNoiseDataProcessor::filter_samples (AntiAliasingFilter &filter_0, AntiAliasingFilter &filter_1)

  Apply a digital filter to sample buffers (only used for fixed window size processing)
- void PhaseNoiseDataProcessor::overwrite_samples (uint32_t packet_num)

  Overwrite selected packet in measurement data buffers with the contents of sample buffers (only used for fixed window size processing)
- void PhaseNoiseDataProcessor::clear_meas_buffer ()

  Set all values in measurement data buffers to 0.
- void PhaseNoiseDataProcessor::clear_output_buffer ()

  Set all values in output buffers to 0.
- void PhaseNoiseDataProcessor::setup (bool cross_corr, double smpl_rate, uint32_t num_meas, uint32_t data_size, uint32_t min_window_size, uint32_t num_bands, uint32_t band_mult, uint32_t band_shift, double conversion_fac)

  Setup processing functions based on the specified parameters.
- void PhaseNoiseDataProcessor::setup (Config new_config)
- const std::vector< double > & PhaseNoiseDataProcessor::get_x_values ()

  Return vector containing x-axis frequency output values.
- const std::vector< double > & PhaseNoiseDataProcessor::get_y_values ()

  Return vector containing y-axis power spectral density output values.
- bool PhaseNoiseDataProcessor::write_output_to_file (char *filename)

  Write output data points to file.
- void PhaseNoiseDataProcessor::process_fixed_window_size ()

  Process sample data from file to produce a set of output data points using a fixed window size.
- void PhaseNoiseDataProcessor::process_fixed_smpl_rate ()

  Process sample data from file to produce a set of output data points using a fixed sampling rate.
- void PhaseNoiseDataProcessor::start_fixed_window_size ()

  Start fixed window size processing where sample data is pushed in real time.
- bool PhaseNoiseDataProcessor::push_data_fixed_window_size (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed window size.
- void PhaseNoiseDataProcessor::start_fixed_smpl_rate ()

  Start fixed sample rate processing where sample data is pushed in real time.
- bool PhaseNoiseDataProcessor::push_data_fixed_smpl_rate (double *data_0, double *data_1)

  Push sample data for a single measurement to be processed with a fixed sample rate.
- void PhaseNoiseDataProcessor::stop ()

  Stop real time processing and release data buffers.

7.9.1 Macro Definition Documentation

7.9.1.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

7.9.1.2 MAX_DATA_SIZE

```
#define MAX_DATA_SIZE 268435456
```

7.9.1.3 MAX_MEAS

```
#define MAX_MEAS 65536
```

7.9.1.4 MAX_WINDOW_SIZE

```
#define MAX_WINDOW_SIZE 268435456
```

7.9.1.5 OUTPUT_FILENAME

```
#define OUTPUT_FILENAME "output.csv"
```

7.9.1.6 SMPL_FILENAME

```
#define SMPL_FILENAME "samples.bin"
```

7.10 PhaseNoiseDataProcessorLib.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #define _USE_MATH_DEFINES
00004 #include <cmath>
00005
00006 #include <stdio.h>
00007 #include <stdint.h>
00008 #include <iostream>
00009 #include <fstream>
00010 #include <string>
00011 #include <vector>
00012
00013 #include "band_data.h"
00014 #include "fft.h"
00015 #include "filter.h"
00016
00017 using namespace std;
00018
00019 #define MAX_DATA_SIZE 268435456
00020 #define MAX_WINDOW_SIZE 268435456
00021 #define MAX_MEAS 65536
00022 #define SMPL_FILENAME "samples.bin"
00023 #define OUTPUT_FILENAME "output.csv"
00024
00025 namespace PhaseNoiseDataProcessor
00026 {
00027     extern std::vector<double> meas_buffer[2];
00028     extern std::vector<double> smpl_buffer[2];
00029     extern std::vector<double> fft_re_buffer;
00030     extern std::vector<double> fft_im_buffer;
00031     extern std::vector<double> output_re_buffer;
00032     extern std::vector<double> output_im_buffer;
00033     extern std::vector<double> output_x_buffer;
00034     extern std::vector<double> output_y_buffer;
00035
00036     extern uint32_t meas_count;
00037
00038     extern BandData band_data;
00039
00040     struct Config
00041     {
00042         bool cross_corr = true;
00043         double smpl_rate = 524288.0;
00044         uint32_t num_meas = 1;
00045         uint32_t data_size = 4194304;
00046         uint32_t min_window_size = 524288;
```

```
00047         uint32_t num_bands = 1;
00048         uint32_t band_mult = 8;
00049         uint32_t band_shift = 2;
00050         double conversion_fac = 0.0025;
00051         bool window_en = false;
00052     };
00053
00054     extern Config config;
00055
00057
00060     void convert_samples(int channel, const void* data_bytes);
00061
00063
00066     void calc_x_values();
00067
00069
00072     void calc_y_values(uint32_t num_meas);
00073     void calc_y_values();
00074
00076
00080     void dbv_to_dbc(double rf_input_level, double cal_input_level, double cal_output_level);
00081
00083
00086     bool read_meas_from_file(char* filename, uint32_t meas_num);
00087
00089
00092     void retrieve_samples(uint32_t set_num, uint32_t smpl_rate_div);
00093
00095
00098     void filter_samples(AntiAliasingFilter& filter_0, AntiAliasingFilter& filter_1);
00099
00101     void overwrite_samples(uint32_t packet_num);
00102
00104     void clear_meas_buffer();
00105
00107     void clear_output_buffer();
00108
00110     void setup(bool cross_corr, double smpl_rate, uint32_t num_meas, uint32_t data_size, uint32_t
      min_window_size, uint32_t num_bands, uint32_t band_mult, uint32_t band_shift, double conversion_fac);
00111     void setup(Config new_config);
00112
00114     const std::vector<double>& get_x_values();
00115
00117     const std::vector<double>& get_y_values();
00118
00120
00122     bool write_output_to_file(char* filename);
00123
00125
00127     void process_fixed_window_size();
00128
00130
00132     void process_fixed_smpl_rate();
00133
00135     void start_fixed_window_size();
00136
00138     bool push_data_fixed_window_size(double* data_0, double* data_1);
00139
00141     void start_fixed_smpl_rate();
00142
00144     bool push_data_fixed_smpl_rate(double* data_0, double* data_1);
00145
00147     void stop();
00148 }
```

7.11 trig_table.h File Reference

```
#include <cmath>
#include <stdint.h>
#include <vector>
```

Classes

- class TrigTable

  Generates a lookup table of precalculated sine/cosine values.

Macros

- #define _USE_MATH_DEFINES

7.11.1 Macro Definition Documentation

7.11.1.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

7.12 trig_table.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #define _USE_MATH_DEFINES
00004 #include <cmath>
00005
00006 #include <stdint.h>
00007 #include <vector>
00008
00010 class TrigTable
00011 {
00012 private:
00013
00014     uint32_t mod_mask;
00015     uint32_t cos_shift;
00016     std::vector<double> sin_table;
00017
00018 public:
00019     TrigTable(uint32_t size)
00020     {
00021         mod_mask = size - 1;
00022         cos_shift = size » 2;
00023         sin_table.resize(size);
00024         for (int i = 0; i < size; i++) sin_table[i] = sin((((double) i / size)) * 2 * M_PI);
00025     }
00026     TrigTable()
00027     {
00028         mod_mask = 0x0;
00029         cos_shift = 0;
00030         sin_table.resize(1);
00031         sin_table[0] = 0.0;
00032     }
00034     double sin_lookup(uint64_t pos)
00035     {
00036         uint32_t pos_m = (uint32_t)(pos & mod_mask);
00037         return sin_table[pos_m];
00038     }
00039
00041     double cos_lookup(uint64_t pos)
00042     {
00043         uint32_t pos_m = (uint32_t)((pos + cos_shift) & mod_mask);
00044         return sin_table[pos_m];
00045     }
00046 };
00047
00048
```

7.13 window.h File Reference

```
#include <cmath>
#include <vector>
```

Classes

- class Window

  Abstract class for a sampling window of given size, which defines the shape and resolution bandwidth scaling factor.
- class FlatTop

  Flat top window for minimal scalloping loss.

Macros

- #define _USE_MATH_DEFINES

7.13.1 Macro Definition Documentation

7.13.1.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

7.14 window.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #define _USE_MATH_DEFINES
00004 #include <cmath>
00005
00006 #include <vector>
00007
00009 class Window
00010 {
00011 protected:
00012
00013     size_t size = 0;
00014     double rbw_factor;
00015     std::vector<double> data;
00016
00017     virtual double rbwFactor() = 0;
00018     virtual void generate() = 0;
00019
00020 public:
00021
00022     Window(size_t size)
00023     {
00024         this->size = size;
00025         data.resize(size);
00026         this->rbw_factor = rbwFactor();
00027         generate();
00028     }
00029 };
00030
00032 class FlatTop : Window
00033 {
00034 protected:
00035
00036     double rbwFactor() override { return 3.8193596; };
00037
00038     void generate()
00039     {
00040         double a0 = 0.21557895;
00041         double a1 = 0.41663158;
00042         double a2 = 0.277263158;
00043         double a3 = 0.083578947;
00044         double a4 = 0.006947368;
00045         double fac = M_PI / size;
00046
00047         for (int i = 0; i < data.size(); i++)
00048         {
00049             data[i] = a0 - (a1 * cos(2 * i * fac)) + (a2 * cos(4 * i * fac)) - (a3 * cos(6 * i * fac))
      + (a4 * cos(8 * i * fac));
00050         }
00051     }
00052
00053 public:
00054
00055     FlatTop(size_t size) : Window(size) {};
00056 };
```

Index