

# **CHAPRO**

## **Compression Hearing-Aid Processing Library API Documentation**

**Stephen Neely, D.Sc.  
Director, Communication Engineering Laboratory**

**Boys Town National Research Hospital  
3 August 2022**

**TABLE OF CONTENTS**

CHAPRO LIBRARY OVERVIEW .....	4
CHAPRO FUNCTION DESCRIPTIONS .....	7
cha_allocate .....	7
cha_cleanup .....	8
cha_data_gen .....	9
cha_data_save .....	10
cha_data_read .....	11
cha_scale .....	12
cha_fft_cr .....	13
cha_fft_rc .....	14
cha_fft .....	15
cha_ifft .....	16
cha_prepare .....	17
cha_version .....	18
cha_chunk_size .....	19
cha_agc_prepare .....	20
cha_agc_input .....	21
cha_agc_channel .....	22
cha_agc_output .....	23
cha_icmp_prepare .....	24
cha_icmp_process .....	25
cha_afc_prepare .....	26
cha_afc_input .....	27
cha_afc_output .....	28
cha_ciirfb_design .....	29
cha_ciirfb_prepare .....	30
cha_ciirfb_analyze .....	31
cha_ciirfb_synthesize .....	32
cha_cfirfb_prepare .....	33
cha_cfirfb_analyze .....	34
cha_cfirfb_synthesize .....	35
cha_firfb_prepare .....	36
cha_firfb_analyze .....	37
cha_firfb_synthesize .....	38
cha_iirfb_design .....	39
cha_iirfb_prepare .....	40
cha_iirfb_analyze .....	41
cha_firfb_synthesize .....	42
cha_nfc_prepare .....	43
cha_nfc_process .....	44
Appendix A. Test programs .....	45
Appendix B. CLS Prescription .....	46
Appendix C. DSL Prescription .....	47
Appendix D. WDRC Parameters .....	48

Appendix E. AFC Parameters .....49

Appendix F. NFC Parameters.....50

## CHAPRO LIBRARY OVERVIEW

CHAPRO is a library of functions that may be used to implement simulations of compression hearing-aid signal processing. Four different types of signal processing strategies are included:

- (1) FIR filter-bank frequency analysis with automatic gain control;
- (2) IIR filter-bank frequency analysis with automatic gain control;
- (3) Complex FIR filter-bank frequency analysis with instantaneous compression;
- (4) Complex IIR filter-bank frequency analysis with instantaneous compression.

A modular design has been adopted to facilitate replacement of library functions with alternative signal-processing implementations. Each of the six major modules contains (1) a *preparation* function that allocates memory and initializes variables and (2) one or more *processing* functions that perform signal processing.

1. FIR filter-bank
  - a. `firfb_prepare`
  - b. `firfb_analyze`
  - c. `firfb_synthesize`
2. IIR filter-bank
  - a. `iirfb_design`
  - b. `iirfb_prepare`
  - c. `iirfb_analyze`
  - d. `iirfb_synthesize`
3. Automatic gain control
  - a. `agc_prepare`
  - b. `agc_input`
  - c. `agc_channel`
  - d. `agc_output`
4. Adaptive feedback cancelation
  - a. `afc_prepare`
  - b. `afc_input`
  - c. `afc_output`
5. Complex IIR filter-bank
  - a. `ciirfb_design`
  - b. `ciirfb_prepare`
  - c. `ciirfb_analyze`
  - d. `ciirfb_synthesize`
6. Instantaneous compression
  - a. `icmp_prepare`
  - b. `icmp_process`
7. Nonlinear frequency compression
  - a. `nfc_prepare`

b. `nfc_process`

For each module, variables are initialized and associated data memory is allocated by the preparation functions. This storage is combined into a single data structure to facilitate creation of firmware for real-time implementation on signal-processing hardware that might not have an operating system. The CHAPRO library includes a function that generates a C-code representation of this initialized data.

- `data_gen`

For desktop simulation, the CHAPRO library includes core functions for memory allocation and disposal.

- `prepare`
- `allocate`
- `cleanup`

Functions for FFT of real signals are included among the core functions.

- `fft_rc`
- `fft_cr`

Finally, the CHAPRO library includes a function that returns a version description string.

- `version`

To simulate complex IIR filter-bank frequency analysis with instantaneous compression, variable initialization and memory allocation is performed by calling the following functions.

- `ciirfb_prepare`
- `icmp_prepare`

Subsequent signal processing is performed by calling the following functions.

- `ciirfb_analyze`
- `icmp_process`
- `ciirfb_synthesize`

Several examples test basic aspects of these functions.

- `tst_cifa` – tests complex IIR filter-bank analysis
- `tst_cifio` – tests simple waveform complex IIR processing
- `tst_cifsc` – tests simple waveform complex IIR processing with soundcard

To simulate FIR filter-bank frequency analysis with AGC compression, variable initialization and memory allocation is performed by calling the following prepare functions.

- `firfb_prepare`
- `agc_prepare`

Subsequent signal processing is performed by calling the following process functions.

- `agc_input`
- `firfb_analyze`

- agc\_channel
- firfb\_synthesize
- agc\_output

Several examples test basic aspects of these functions.

- tst\_ffa – tests filter-bank analysis
- tst\_ffio – tests simple waveform FIR processing
- tst\_ffsc – tests speech-waveform FIR & AGC with soundcard

To simulate IIR filter-bank frequency analysis with AGC compression and adaptive feedback cancelation, variable initialization and memory allocation is performed by calling the following functions.

- afc\_prepare
- agc\_prepare
- iirfb\_prepare

Subsequent signal processing is performed by calling the following functions.

- afc\_input
- agc\_input
- iirfb\_analyze
- agc\_channel
- iirfb\_synthesize
- agc\_output
- afc\_output

Several examples test basic aspects of these functions.

- tst\_ifa – tests filter-bank analysis
- tst\_ifio – tests simple waveform IIR processing
- tst\_ifsc – tests speech-waveform IIR & AGC with soundcard
- tst\_iffb – tests speech-waveform IIR & AFC with soundcard
- tst\_gha – tests speech-waveform IIR & AFC & AGC with soundcard

All examples require the SIGPRO library from BTNRH (<http://audres.org/rc/sigpro>). The soundcard examples also require the ARSC library (<http://audres.org/rc/arsc>).

## CHAPRO FUNCTION DESCRIPTIONS

### *cha\_allocate*

Allocates memory attached to CHAPRO data structure.

(void \*) **cha\_allocate**(CHA\_PTR **cp**, int **cnt**, int **siz**, int **idx**)

#### Function arguments

<b>cp</b>	pointer to CHAPRO data structure
<b>cnt</b>	Number of elements to allocate.
<b>siz</b>	Size of each element.
<b>idx</b>	Index into CHAPPRO data structure.

#### Return Value

Pointer to the allocated memory.

#### Remarks

A pointer to the allocated memory is stored in the CHAPRO data structure at the location specified by **idx**.

#### See Also

**cha\_cleanup**

***cha\_cleanup***

Frees all memory attached to CHAPRO data structure.

(void) **cha\_cleanup**(CHA\_PTR **cp**)

**Function arguments**

**cp**                    pointer to CHAPRO data structure

**Return Value**

none

**Remarks**

Should always be the last function called in the CHAPRO library.

**See Also**

**cha\_allocate**



***cha\_data\_gen***

Generates C code that represents the CHAPRO data structure.

(int) **cha\_data\_gen**(CHA\_PTR **cp**, char \***fn**)

**Function arguments**

<b>cp</b>	pointer to CHAPRO data structure
<b>fn</b>	Pointer to output filename.

**Return Value**

Error code:

- 0 – no error
- 1 – can't open output file
- 2 – data structure not yet initialized
- 3 – data structure contains no data

**Remarks**

The C code generated by this function represents the CHAPRO data structure after variables have been initialized and data memory has been allocated by prior calls to any preparation functions. The code is written to the file specified by **fn**.

**See Also**

**cha\_data\_save, cha\_data\_load**

***cha\_data\_save***

Writes the CHAPRO data structure to a binary file.

(int) **cha\_data\_save**(CHA\_PTR **cp**, char \***fn**)

**Function arguments**

<b>cp</b>	pointer to CHAPRO data structure
<b>fn</b>	Pointer to output filename.

**Return Value**

Error code:

- 0 – no error
- 1 – can't open output file
- 2 – data structure not yet initialized
- 3 – data structure contains no data

**Remarks**

The binary file written by this function represents the CHAPRO data structure after variables have been initialized and data memory has been allocated by prior calls to any preparation functions. The code is written to the file specified by **fn**. A 16-byte file header is prepended to the data.

**See Also**

**cha\_data\_gen, cha\_data\_load**

***cha\_data\_read***

Reads the CHAPRO data structure from a binary file.

(int) **cha\_data\_read**(CHA\_PTR **cp**, char \***fn**)

**Function arguments**

<b>cp</b>	pointer to CHAPRO data structure
<b>fn</b>	Pointer to output filename.

**Return Value**

Error code:

- 0 – no error
- 1 – can't open output file
- 2 – data structure not yet initialized
- 3 – data structure contains no data

**Remarks**

The binary file read by this function represents the CHAPRO data structure after variables have been initialized and data memory has been allocated by prior calls to any preparation functions. The code is written to the file specified by **fn**. A 16-byte file header is prepended to the data.

**See Also**

**cha\_data\_save, cha\_data\_gen**

***cha\_scale***

Applies scales factor to a chunk of the input or output stream.

(void) **cha\_scale**(float \***x**, int **cs**, float **scale**)

**Function arguments**

<b>x</b>	pointer to input signal
<b>cs</b>	chunk size
<b>scale</b>	pointer to CHAPRO data structure

**Return Value**

None.

**Remarks**

The scaled output signal overwrites the input signal.

***cha\_fft\_cr***

Inverse Fourier transform complex frequency components into real signal.

(void) **cha\_fft\_cr**(float \***x**, int **n**)

**Function arguments**

<b>x</b>	Complex frequency components are replaced by real-valued signal.
<b>n</b>	Number of points in the signal.

**Return Value**

None

**Remarks**

The input array must be dimensioned to accommodate **n+2** float values. The number of complex frequency components is **(n+2)/2**.

***cha\_fft\_rc***

Fourier transform real signal into complex frequency components.

(void) **cha\_fft\_rc**(float \***x**, int **n**)

**Function arguments**

<b>x</b>	Real-valued signal is replaced by complex frequency components
<b>n</b>	Number of points in the signal.

**Return Value**

None

**Remarks**

The input array must be dimensioned to accommodate **n+2** float values. The number of complex frequency components is **(n+2)/2**.

***cha\_fft***

Fourier transform a complex time signal to complex frequency components.

(void) **cha\_fft**(float \***x**, int **n**)

**Function arguments**

<b>x</b>	Complex frequency components are replaced by a complex signal.
<b>n</b>	Number of points in the signal.

**Return Value**

None

**Remarks**

The input array must be dimensioned to accommodate **n**×2 float values.

***cha\_iff***

Inverse Fourier transform complex frequency components into a complex-valued signal.

(void) **cha\_iff**(float \***x**, int **n**)

**Function arguments**

<b>x</b>	Complex signal is replaced by complex frequency components
<b>n</b>	Number of points in the signal.

**Return Value**

None

**Remarks**

The input array must be dimensioned to accommodate **n**×2 float values.



***cha\_prepare***

CHAPRO data structure preparation function.

(void) **cha\_prepare**(CHA\_PTR **cp**)

**Function arguments**

**cp**                    pointer to CHAPRO data structure

**Return Value**

None.

**Remarks**

Should be called only once and prior to calling other library functions; however, violations of this rule may be tolerated.

***cha\_version***

Returns a string that describes the current version of the CHAPRO library.

(char \*) **cha\_version**(void)

**Function arguments**

none

**Return Value**

Pointer to version string.

**Remarks**

An example of the return value, "CHAPro version 0.03, 6-Nov-2016".

***cha\_chunk\_size***

Specifies nominal size of input/output stream chunks.

(void) **cha\_chunk\_size**(CHA\_PTR **cp**, int **cs**)

<b>cp</b>	pointer to CHAPRO data structure
<b>cs</b>	number of samples per chunk

**Return Value**

None.

**Remarks**

If no filter-bank prepare function has been called prior to calling AGC prepare, then this function may be used to initialize the internal value of the chunk size. This is needed to ensure sufficient allocation of peak-value history for AGC calculations.

**See Also**

**cha\_agc\_prepare**

### *cha\_agc\_prepare*

Automatic-gain-control preparation function.

(int) **cha\_agc\_prepare**(CHA\_PTR **cp**, CHA\_DSL \***dsl**, CHA\_WDRC \***gha**)

#### **Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>dsl</b>	pointer to DSL prescription structure (see Appendix C)
<b>gha</b>	pointer to WDRC prescription structure (see Appendix D)

#### **Return Value**

Error code:  
0 – no error

#### **Remarks**

Initializes variables and allocates memory for automatic gain control. Chunk size is the number of samples read from the input signal and written to the output signal with each call to **cha\_agc\_process**.

The **cha\_agc\_prepare** function needs to allocate memory for a history of the peak value, so assumes that the chunk size of the input-output stream has already been specified, which may be accomplished by previously calling either one of the filter-bank prepare functions or by calling **cha\_chunk\_size**.

#### **See Also**

**cha\_agc\_process**, **cha\_chunk\_size**

***cha\_agc\_input***

Automatic-gain-control processing function.

(void) **cha\_agc\_input**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs single-channel, automatic-gain-control processing on CHAPRO input signal. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_agc\_prepare**, **cha\_agc\_output**

***cha\_agc\_channel***

Automatic-gain-control processing function.

(void) **cha\_agc\_channel**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs multi-channel, automatic-gain-control processing. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_agc\_prepare**

***cha\_agc\_output***

Automatic-gain-control processing function.

(void) **cha\_agc\_output**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs single-channel, automatic-gain-control processing on CHAPRO output signal. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_agc\_prepare**, **cha\_agc\_input**

### ***cha\_icmp\_prepare***

Instantaneous-compression preparation function.

(int) **cha\_icmp\_prepare**(CHA\_PTR **cp**,  
float \***Lc**, float \***Gc**, double **sr**, double **lr**, int **np**, int **ds**)

#### **Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>Lc</b>	pointer to level array
<b>Lc</b>	pointer to gain array
<b>sr</b>	sampling rate (Hz)
<b>lr</b>	level reference
<b>np</b>	number of points in level and gain arrays
<b>ds</b>	down-sample factor

#### **Return Value**

Error code:  
0 – no error

#### **Remarks**

Initializes variables and allocates memory for instantaneous compression. Chunk size is the number of samples read from the input signal and written to the output signal with each call to `cha_icmp_process`.

#### **See Also**

**cha\_icmp\_process**



***cha\_icmp\_process***

Instantaneous-compression processing function.

(void) **cha\_icmp\_process**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs instantaneous compression. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_icmp\_prepare**

### *cha\_afc\_prepare*

Configure feedback management.

(int) **cha\_afc\_prepare**(CHA\_PTR **cp**, CHA\_AFC \***afc**)

#### Function arguments

<b>cp</b>	pointer to CHA data structure
<b>afc</b>	pointer to AFC parameters & buffers structure (see Appendix E)

#### Return Value

Error code:

0 – no error

#### Remarks

Initializes variables and allocates memory for adaptive feedback cancelation (AFC), which is controlled by contained in the CHA\_AFC structure. In the current version (0.24), the AFC implementation is functional, but incomplete.

Feedback estimation is controlled by three parameters, **mu**, **rho**, & **eps**. For testing purposes, feedback simulation is enabled by setting **fbg**=1 or disabled by setting **fbg**=0. When simulation is enabled, the feedback-filter misalignment error is saved as a quality metric by setting **sqm**=1.

#### See Also

**cha\_afc\_input**, **cha\_afc\_output**

***cha\_afc\_input***

Process input signal to remove feedback.

(void) **cha\_afc\_input**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHAPRO data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Removes estimated feedback from input signal. Chunk size is the number of samples read from the input signal and written to the output signal. Optionally simulates feedback and saves misalignment error as a quality metric.

**See Also**

**cha\_afc\_prepare, cha\_afc\_output**

***cha\_afc\_output***

Save output signal for feedback management.

(void) **cha\_afc\_output**(CHA\_PTR **cp**, float \***x**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHAPRO data structure
<b>x</b>	pointer to input signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Assists feedback management by saving the output of the hearing-aid processing. Chunk size is the number of samples read from the input signal.

**See Also**

**cha\_afc\_prepare, cha\_afc\_input**

***cha\_ciirfb\_design***

Complex IIR filter-bank design function.

```
(int) cha_ciirfb_design(float *z, float *p, float *g, int *d, int nc,
                        double *fc, double *bw, double sr, double td)
```

**Function arguments**

<b>z</b>	pointer to IIR filter complex zeros
<b>p</b>	pointer to IIR filter complex poles
<b>g</b>	pointer IIR filter complex gain
<b>d</b>	pointer IIR filter delay
<b>nc</b>	number of frequency bands
<b>fc</b>	pointer to list of center frequencies (Hz)
<b>bw</b>	pointer to list of bandwidths (Hz)
<b>sr</b>	sampling rate (samples/second)
<b>td</b>	target group delay (ms)

**Return Value**

Error code:

0 – no error

**Remarks**

Computes zeros and poles for the complex IIR filter-bank. The filterbank design is based on fourth-order gammatone bandpass filters. The center frequency (**fc**) and bandwidth (**bw**) arrays are input arguments and their size is equal to the number of frequency bands (**nc**). The zero (**z**) and pole (**p**) arrays are output arguments and are complex numbers, so are stored as sequential real and imaginary parts. The size of the zeros & poles arrays is two times the product of the number of frequency bands (**nc**) and the number of zeros (and poles) per band, which is always equal to 4. The gain array (**g**) is complex, so its size two times the number of frequency bands (**nc**). The size of the filter delay array (**d**) is equal to the number of frequency bands (**nc**). The filter-design process attempts to align the impulse response of each frequency band to the target delay (**td**).

**See Also**

**cha\_ciirfb\_prepare**, **cha\_ciirfb\_analyze**, **cha\_ciirfb\_synthesize**

***cha\_ciirfb\_prepare***

Complex IIR filter-bank preparation function.

```
(int) cha_ciirfb_prepare(CHA_PTR cp, float *z, float *p, float *g, int *d,
    int nc, int nz, double sr, int cs)
```

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>z</b>	pointer to IIR filter complex zeros
<b>p</b>	pointer to IIR filter complex poles
<b>g</b>	pointer IIR filter complex gain
<b>d</b>	pointer IIR filter delay
<b>nc</b>	number of frequency bands
<b>nz</b>	number of zeros (and poles) for each band
<b>sr</b>	sampling rate (samples/second)
<b>cs</b>	chunk size

**Return Value**

Error code:

0 – no error

**Remarks**

Initializes variables and allocates memory for the IIR filter-bank. The zero (**z**) and pole (**p**) arrays are output arguments and are complex numbers, so are stored as sequential real and imaginary parts. The size of the zero & pole arrays is two times the product of the number of frequency bands (**nc**) and the number of zeros per band (**nz**). The number of zeros (and poles) per band (**nz**) should be an even number and should include conjugate pairs, so that second-order sections will have real coefficients. The gain array (**g**) is complex, so its size two times the number of frequency bands (**nc**). The size of the filter delay array (**d**) is equal to the number of frequency bands (**nc**). Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_ciirfb\_design**, **cha\_ciirfb\_analyze**, **cha\_ciirfb\_synthesize**

***cha\_ciirfb\_analyze***

Complex IIR filter-bank frequency-analysis function.

(void) **cha\_ciirfb\_analyze**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to complex output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs complex IIR filter-bank analysis. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_ciirfb\_prepare, cha\_ciirfb\_synthesize**

***cha\_ciirfb\_synthesize***

Complex IIR filter-bank frequency-synthesis function.

(void) **cha\_ciirfb\_synthesize**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to complex input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs complex filter-bank frequency-synthesis. Frequency bands are summed and only the real part is output. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_ciirfb\_prepare**, **cha\_ciirfb\_analyze**



***cha\_cfirfb\_prepare***

Complex FIR filter-bank preparation function.

(int) **cha\_cfirfb\_prepare**(CHA\_PTR **cp**, double \***cf**, int **nc**, double **sr**,  
int **nw**, int **wt**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>cf</b>	list frequency band edges (kHz)
<b>nc</b>	number of frequency bands
<b>sr</b>	sampling rate (samples/second)
<b>nw</b>	window size (samples)
<b>wt</b>	window type (0=Hamming, 1=Blackman)
<b>cs</b>	chunk size

**Return Value**

Error code:  
0 – no error

**Remarks**

Initializes variables and allocates memory for the complex FIR filter-bank. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_cfirfb\_analyze**, **cha\_cfirfb\_synthesize**

***cha\_cfirfb\_analyze***

Complex FIR filter-bank frequency-analysis function.

(void) **cha\_cfirfb\_analyze**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to real input signal
<b>y</b>	pointer to complex output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs complex FIR filter-bank analysis. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_cfirfb\_prepare**, **cha\_cfirfb\_synthesize**

***cha\_cfirfb\_synthesize***

Complex FIR filter-bank frequency-synthesis function.

(void) **cha\_cfirfb\_synthesize**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to complex input signal
<b>y</b>	pointer to real output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs complex FIR filter-bank synthesis. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_cfirfb\_prepare**, **cha\_cfirfb\_analyze**

***cha\_firfb\_prepare***

FIR filter-bank preparation function.

(int) **cha\_firfb\_prepare**(CHA\_PTR **cp**,  
double \***cf**, int **nc**, double **sr**, int **nw**, int **wt**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>cf</b>	list frequency band edges (kHz)
<b>nc</b>	number of frequency bands
<b>sr</b>	sampling rate (samples/second)
<b>nw</b>	window size (samples)
<b>wt</b>	window type (0=Hamming, 1=Blackman)
<b>cs</b>	chunk size

**Return Value**

Error code:  
0 – no error

**Remarks**

Initializes variables and allocates memory for the FIR filter-bank. Chunk size is the number of samples read from the input and written to the output.

**See Also**

**cha\_firfb\_analyze**, **cha\_firfb\_synthesize**

***cha\_firfb\_analyze***

FIR filter-bank frequency-analysis function.

(void) **cha\_firfb\_analyze**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs FIR filter-bank analysis. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_firfb\_prepare, cha\_firfb\_synthesize**

***cha\_firfb\_synthesize***

FIR filter-bank frequency-synthesis function.

(void) **cha\_firfb\_synthesize**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs FIR filter-bank synthesis. Chunk size is the number of samples read from the input signal and written to the output signal.

***cha\_iirfb\_design***

IIR filter-bank design function.

```
(int) cha_iirfb_design(float *z, float *p, float *g, int *d,
    double *cf, int nc, int nz, double sr, double td)
```

**Function arguments**

<b>z</b>	pointer to IIR filter complex zeros
<b>p</b>	pointer to IIR filter complex poles
<b>g</b>	pointer to IIR filter gain
<b>d</b>	pointer to IIR filter delay
<b>cf</b>	pointer to cross-over frequencies
<b>nc</b>	number of frequency bands
<b>nz</b>	number of zeros (and poles) for each band
<b>sr</b>	sampling rate (samples/second)
<b>td</b>	impulse response target delay (millisecond)

**Return Value**

Error code:

0 – no error

**Remarks**

Computes zeros and poles for the IIR filter-bank. The filterbank design is based on Butterworth bandpass filters sandwiched between Butterworth low-pass and high-pass filters. The array of cross-over frequencies (**cf**) is an input argument and its size is one less than the number of frequency bands (**nc**). The zero (**z**) and pole (**p**) arrays are output arguments and are complex numbers, so are stored as sequential real and imaginary parts. The size of the zeros & poles arrays is two times the product of the number of frequency bands (**nc**) and the number of zeros per band (**nz**). The number of zeros (and poles) per band (**nz**) should be an even number and should include conjugate pairs, so that second-order sections have real coefficients. The number of filter gains and filter delays (**g** & **d**) is the number of frequency bands (**nc**). The filter-design process attempts to align the impulse response of each frequency band to the target delay (**td**).

**See Also**

**cha\_iirfb\_prepare**, **cha\_iirfb\_analyze**, **cha\_iirfb\_synthesize**

### ***cha\_iirfb\_prepare***

IIR filter-bank preparation function.

(int) **cha\_iirfb\_prepare**(CHA\_PTR **cp**, float \***z**, float \***p**, float \***g**, int \***d**,  
int **nc**, int **nz**, double **sr**, int **cs**)

#### **Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>z</b>	pointer to IIR filter complex zeros
<b>p</b>	pointer to IIR filter complex poles
<b>g</b>	pointer IIR filter gain
<b>d</b>	pointer IIR filter delay
<b>nc</b>	number of frequency bands
<b>nz</b>	number of zeros (and poles) for each band
<b>sr</b>	sampling rate (samples/second)
<b>cs</b>	chunk size

#### **Return Value**

Error code:

0 – no error

#### **Remarks**

Initializes variables and allocates memory for the IIR filter-bank. The zero (**z**) and pole (**p**) arrays are output arguments and are complex numbers, so are stored as sequential real and imaginary parts. The size of the zero & pole arrays is two times the product of the number of frequency bands (**nc**) and the number of zeros per band (**nz**). The number of zeros (and poles) per band (**nz**) should be an even number and should include conjugate pairs, so that second-order sections will have real coefficients. The filter gains and filter delays (**g** & **d**) are input arguments and have array sizes equal to the number of frequency bands (**nc**). Chunk size is the number of samples read from the input signal and written to the output signal.

#### **See Also**

**cha\_iirfb\_design**, **cha\_iirfb\_analyze**, **cha\_iirfb\_synthesize**



***cha\_iirfb\_analyze***

IIR filter-bank frequency-analysis function.

(void) **cha\_iirfb\_analyze**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs IIR filter-bank analysis. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_iirfb\_prepare**, **cha\_iirfb\_synthesize**

***cha\_firfb\_synthesize***

IIR filter-bank frequency-synthesis function.

(void) **cha\_iirfb\_synthesize**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

**Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

**Return Value**

none

**Remarks**

Performs IIR filter-bank synthesis. Chunk size is the number of samples read from the input signal and written to the output signal.

**See Also**

**cha\_iirfb\_prepare, cha\_iirfb\_analyze**

### ***cha\_nfc\_prepare***

Nonlinear-frequency-compression (NFC) preparation function.

(int) **cha\_nfc\_prepare**(CHA\_PTR **cp**, CHA\_NFC **nfc**)

#### **Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>nfc</b>	pointer to NFC data structure (see Appendix F)

#### **Return Value**

Error code:  
0 – no error  
1 – invalid chunk size

#### **Remarks**

Initializes variables and allocates memory for nonlinear frequency compression. Chunk size (**nfc.cs**) is the number of samples read from the input signal and written to the output signal with each call to **cha\_nfc\_process**.

The frequency-mapping may be defined by specifying the lower (**nfc.fl**) and upper (**nfc.f2**) bounds of the output frequency range into which input frequencies will be compressed. Input frequencies below the lower bound will not be altered by the mapping. The range of input frequencies between the lower bound and the Nyquist frequency (i.e., half the sampling rate) will be compressed into the specified output frequency range.

Alternatively, the frequency mapping may be defined by an array of indices (at **nfc.mm** of size **nfc.nm**) that specify the upper and lower bounds of the input frequency bins that go into each output frequency bin. Indices are assumed to have monotonically increasing values.

Optionally, gain may be applied to each frequency bin before (**nfc.g1**) and/or after (**nfc.g2**) the frequency mapping. The number of frequency bins before mapping is equal to the window size (**nfc.nw**) and this number is reduced after mapping by a factor of  $2 \cdot \text{nfc.f2} / \text{nfc.sr}$ . No gain is applied when gain pointers are set to zero.

#### **See Also**

**cha\_nfc\_process**

### ***cha\_nfc\_process***

Nonlinear-frequency-compression (NFC) processing function.

(void) **cha\_nfc\_process**(CHA\_PTR **cp**, float \***x**, float \***y**, int **cs**)

#### **Function arguments**

<b>cp</b>	pointer to CHA data structure
<b>x</b>	pointer to input signal
<b>y</b>	pointer to output signal
<b>cs</b>	chunk size

#### **Return Value**

none

#### **Remarks**

Performs nonlinear frequency compression. Chunk size is the number of samples read from the input signal and written to the output signal.

The signal-processing method currently implemented by this function may be covered by a patent ([US20040264721](#)) that has an anticipated expiration in 2023.

#### **See Also**

**cha\_nfc\_prepare**

## Appendix A. Test programs

Several examples that test basic aspects of complex IIR filter-bank and instantaneous-compression.

- `tst_cifa` – tests complex IIR filter-bank analysis
- `tst_cifio` – tests simple waveform complex IIR processing
- `tst_cifsc` – tests simple waveform complex IIR processing with soundcard

Several examples that test basic aspects of FIR filter-bank and automatic-gain-control.

- `tst_ffa` – tests FIR filter-bank analysis
- `tst_ffio` – tests simple waveform FIR processing
- `tst_ffsc` – tests speech-waveform FIR & AGC with soundcard

Several examples that test basic aspects of IIR filter-bank with automatic-gain-control and adaptive feedback cancelation.

- `tst_ifa` – tests IIR filter-bank analysis
- `tst_ifio` – tests simple waveform IIR processing
- `tst_ifsc` – tests speech-waveform IIR & AGC
- `tst_gha` – tests speech-waveform IIR & AGC & AFC with soundcard

These test programs are all written in C and produce results that are written to a subfolder called “test.” Each test program has a corresponding MATLAB script for viewing the results. The test programs that contain IIR processing require filter coefficients to be precomputed by a MATLAB script called `iirfb`.

## Appendix B. CLS Prescription

Structure `CHA_CLS` specifies the CLS prescription.

```
#define CLS_MXCH 32          // maximum number of channels

typedef struct {
    int cm;                  // compression mode
    int nc;                  // number of channels
    double fc[CLS_MXCH];    // center frequency
    double bw[CLS_MXCH];    // bandwidth
    double Gcs[CLS_MXCH];   // gain at compression start
    double Gcm[CLS_MXCH];   // gain at compression middle
    double Gce[CLS_MXCH];   // gain at compression end
    double Gmx[CLS_MXCH];   // maximum gain
    double Lcs[CLS_MXCH];   // level at compression start
    double Lcm[CLS_MXCH];   // level at compression middle
    double Lce[CLS_MXCH];   // level at compression end
    double Lmx[CLS_MXCH];   // maximum output level
} CHA_CLS;
```

## Appendix C. DSL Prescription

Structure `CHA_DSL` specifies the DSL prescription.

```
#define DSL_MXCH 32                // maximum number of channels

typedef struct {
    double attack;                 // attack time (ms)
    double release;               // release time (ms)
    double maxdB;                 // maximum output (dB SPL)
    int ear;                     // 0=left, 1=right
    int nchannel;                // number of channels
    double cross_freq[DSL_MXCH]; // cross frequencies (Hz)
    double tkgain[DSL_MXCH];     // compression-start gain
    double cr[DSL_MXCH];         // compression ratio
    double tk[DSL_MXCH];         // compression-start kneepoint
    double bolt[DSL_MXCH];       // broadband output limiting threshold
} CHA_DSL;
```

## Appendix D. WDRC Parameters

Structure `CHA_WDRC` specifies single-channel WDRC parameters

```
typedef struct {  
    double attack;           // attack time (ms)  
    double release;          // release time (ms)  
    double fs;               // sampling rate (Hz)  
    double maxdB;            // maximum signal (dB SPL)  
    double tkgain;           // compression-start gain  
    double tk;               // compression-start kneepoint  
    double cr;               // compression ratio  
    double bolt;             // broadband output limiting threshold  
} CHA_WDRC;
```



## Appendix E. AFC Parameters

Structure `CHA_AFC` specifies single-channel AFC parameters

```
typedef struct {
    // simulation parameters
    double fbg;                // simulated-feedback gain
    // AFC parameters
    double rho;                // forgetting factor
    double eps;                // power threshold
    double mu;                 // step size
    int afl;                   // adaptive-filter length
    int wfl;                   // whiten-filter length
    int pfl;                   // pre-emphasis-filter length
    int fbl;                   // simulated-feedback length
    int hdel;                  // output/input hardware delay
    // feedback filter buffers
    float *efbp;               // estimated-feedback buffer pointer
    float *sfbp;               // simulated-feedback buffer pointer
    float *wfrp;               // whiten-filter buffer pointer
    float *ffrp;               // pre-emphasis buffer pointer
    // quality metric buffers & parameters
    float *merr;               // chunk-error buffer pointer
    float *qm;                 // quality-metric buffer pointer
    int nqm;                   // quality-metric buffer size
    int iqm;                   // quality-metric index
    int sqm;                   // save quality metric ?
    CHA_PTR pcp;               // previous CHA_PTR
} CHA_AFC;
```

## Appendix F. NFC Parameters

Structure `CHA_NFC` specifies single-channel NFC parameters

```
typedef struct {  
    int32_t  cs;           // chunk size  
    int32_t  nw;           // window size (pow2)  
    int32_t  wt;           // window type: 0=Hamming, 1=Blackman  
    int32_t  nm;           // frequency-map size  
    double   sr;           // sampling rate (Hz)  
    double   f1;           // compression-lower-bound frequency (Hz)  
    double   f2;           // compression-upper-bound frequency (Hz)  
    int32_t  *mm;          // frequency-map pointer  
    float    *g1;          // pre-map gain pointer  
    float    *g2;          // pre-map gain pointer  
} CHA_NFC;
```