

# Channel mapping for Studio 114

Henrik Frisk

September 28, 2018

<b>author</b>	Henrik Frisk	
<b>basics.lib/name</b>	Faust Basic Element Library	
<b>basics.lib/version</b>	0.0	
<b>copyright</b>	(c) dinergy 2018	
<b>filename</b>	KMH114_channel_map_C	
<b>license</b>	BSD	
<b>maths.lib/author</b>	GRAME	This document pro-
<b>maths.lib/copyright</b>	GRAME	
<b>maths.lib/license</b>	LGPL with exception	
<b>maths.lib/name</b>	Faust Math Library	
<b>maths.lib/version</b>	2.1	
<b>name</b>	Channel mapping for Studio 114	
<b>version</b>	0.1	

vides a mathematical description of the Faust program text stored in the `src/KMH114_channel_map_C.dsp` file. See the notice in Section 3 (page 3) for details.

## 1 Mathematical definition of process

The *KMH114\_channel\_map\_C* program evaluates the signal transformer denoted by **process**, which is mathematically defined as follows:

1. Output signals  $y_i$  for  $i \in [1, 15]$  such that

$$y_1(t) = x_1(t)$$

$$y_2(t) = x_3(t)$$

$$y_3(t) = x_2(t)$$

$$y_4(t) = x_{15}(t)$$

$$y_5(t) = x_8(t)$$

$$y_6(t) = x_5(t)$$

$$y_7(t) = x_9(t)$$

$$y_8(t) = x_4(t)$$

$$y_9(t) = x_7(t)$$

$$y_{10}(t) = x_6(t)$$

$$y_{11}(t) = x_{10}(t)$$

$$y_{12}(t) = x_{11}(t)$$

$$y_{13}(t) = x_{13}(t)$$

$$y_{14}(t) = x_{12}(t)$$

$$y_{15}(t) = x_{14}(t)$$

2. Input signals  $x_i$  for  $i \in [1, 15]$

3. Intermediate signals  $r_i$  for  $i \in [1, 15]$  such that

$$r_1(t) = \max(r_1(t-1) - k_1, |x_1(t)|)$$

$$r_2(t) = \max(r_2(t-1) - k_1, |x_3(t)|)$$

$$r_3(t) = \max(r_3(t-1) - k_1, |x_2(t)|)$$

$$r_4(t) = \max(r_4(t-1) - k_1, |x_{15}(t)|)$$

$$r_5(t) = \max(r_5(t-1) - k_1, |x_8(t)|)$$

$$r_6(t) = \max(r_6(t-1) - k_1, |x_5(t)|)$$

$$r_7(t) = \max(r_7(t-1) - k_1, |x_9(t)|)$$

$$r_8(t) = \max(r_8(t-1) - k_1, |x_4(t)|)$$

$$r_9(t) = \max(r_9(t-1) - k_1, |x_7(t)|)$$

$$r_{10}(t) = \max(r_{10}(t-1) - k_1, |x_6(t)|)$$

$$r_{11}(t) = \max(r_{11}(t-1) - k_1, |x_{10}(t)|)$$

$$r_{12}(t) = \max(r_{12}(t-1) - k_1, |x_{11}(t)|)$$

$$r_{13}(t) = \max(r_{13}(t-1) - k_1, |x_{13}(t)|)$$

$$r_{14}(t) = \max(r_{14}(t-1) - k_1, |x_{12}(t)|)$$

$$r_{15}(t) = \max(r_{15}(t-1) - k_1, |x_{14}(t)|)$$

4. Constant  $k_1$  such that

$$k_1 = \frac{1}{\min(192000, \max(1, f_S))}$$

## 2 Block diagram of process

The block diagram of `process` is shown on Figure 1 (page 35).

### 3 Notice

- This document was generated using Faust version 2.6.3 on September 28, 2018.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the `process` identifier is bound to input signals, running at the  $f_S$  sampling frequency.
- Faust (*Functional Audio Stream*) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal  $s$  in  $S$  is a function mapping<sup>1</sup> times  $t \in \mathbb{Z}$  to values  $s(t) \in \mathbb{R}$ , while a signal transformer is a function from  $S^n$  to  $S^m$ , where  $n, m \in \mathbb{N}$ . See the Faust manual for additional information (<http://faust.grame.fr>).
- Every mathematical formula derived from a Faust expression is assumed, in this document, to having been normalized (in an implementation-dependent manner) by the Faust compiler.
- A block diagram is a graphical representation of the Faust binding of an identifier  $I$  to an expression  $E$ ; each graph is put in a box labeled by  $I$ . Subexpressions of  $E$  are recursively displayed as long as the whole picture fits in one page.
- The `KMH114_channel_map-C-mdoc/` directory may also include the following subdirectories:
  - `cpp/` for Faust compiled code;
  - `pdf/` which contains this document;
  - `src/` for all Faust sources used (even libraries);
  - `svg/` for block diagrams, encoded using the Scalable Vector Graphics format (<http://www.w3.org/Graphics/SVG/>);
  - `tex/` for the L<sup>A</sup>T<sub>E</sub>X source of this document.

### 4 Faust code listings

This section provides the listings of the Faust code used to generate this document, including dependencies.

---

<sup>1</sup>Faust assumes that  $\forall s \in S, \forall t \in \mathbb{Z}, s(t) = 0$  when  $t < 0$ .

Listing 1: KMH114\_channel\_map\_C.dsp

```

1 declare name "Channel mapping for Studio 114";
2 declare version " 0.1 ";
3 declare author " Henrik Frisk " ;
4 declare license " BSD ";
5 declare copyright "(c) dinergy 2018 ";
6
7 //-----'Channel mapping plugin' -----
8 //
9 // Channel mapping plugin that takes 15 channels of input (center speaker included)
10 // and maps it to the channel/speaker configuration of the studio 114 according to:
11 //
12 // * 1 -> 1 (L)
13 // * 2 -> 2 (R)
14 // * 3 -> 3 (C)
15 // * 4 -> 5 (LSR)
16 // * 5 -> 6 (RSR)
17 // * 6 -> 7 (LSF)
18 // * 7 -> 8 (RSF)
19 // * 8 -> 9 (RL)
20 // * 9 -> 10 (RR)
21 // * 10 -> 11 (ULF)
22 // * 11 -> 12 (URF)
23 // * 12 -> 13 (URL)
24 // * 13 -> 14 (URR)
25 // * 14 -> 15 (VOG)
26 //
27 //-----
28
29 import("stdfaust.lib");
30
31 vmeter(x) = attach(x, envelop(x) : vbargraph("[unit:dB]", -70, +5));
32 hmeter(x) = attach(x, envelop(x) : hbargraph("[2][unit:dB]", -70, +5));
33 envelop = abs : max ~ -(1.0/ma.SR) : max(ba.db2linear(-70)) : ba.linear2db;
34
35 process(L, C, R, RSF, RSR, RR, RL, LSR, LSF, ULF, URF, URL, URR, VOG, x) =
36   vgroup("", (L, R, C, x, LSR, RSR, LSF, RSF, RL, RR) : hgroup("lower ring", par(i, 10,
37     vgroup("%i", vmeter)))),
38   vgroup("", (ULF, URF, URR, URL) : hgroup("upper ring", par(i, 4, vgroup("%i", vmeter)))),
39   vgroup("", (VOG) : hgroup("vog", par(i, 1, vgroup("%i", vmeter))));

```

Listing 2: stdfaust.lib

```

1 //##### stdfaust.lib #####
2 // The purpose of this library is to give access to all the Faust standard libraries
3 // through a series of environment.
4 //#####
5
6 an = library("analyzers.lib");
7 ba = library("basics.lib");
8 co = library("compressors.lib");
9 de = library("delays.lib");
10 dm = library("demos.lib");
11 dx = library("dx7.lib");
12 en = library("envelopes.lib");
13 fi = library("filters.lib");
14 ho = library("hoa.lib");
15 ma = library("maths.lib");
16 ef = library("misceffects.lib");
17 os = library("oscillators.lib");
18 no = library("noises.lib");
19 pf = library("phaflangers.lib");
20 pm = library("physmodels.lib");

```

```

21 re = library("reverbs.lib");
22 ro = library("routes.lib");
23 sp = library("spats.lib");
24 si = library("signals.lib");
25 so = library("soundfiles.lib");
26 sy = library("synths.lib");
27 ve = library("vaeffects.lib");
28 sf = library("all.lib");

```

### Listing 3: maths.lib

```

1  //##### maths.lib #####
2  // Mathematic library for Faust. Its official prefix is 'ma'.
3  //#####
4  // Some functions are implemented as Faust foreign functions of 'math.h' functions
5  // that are not part of Faust's primitives. Defines also various constants and several
6  // utilities.
7  //#####
8
9  // ## History
10 // * 06/13/2016 [RM] normalizing and integrating to new libraries
11 // * 07/08/2015 [YO] documentation comments
12 // * 20/06/2014 [SL] added FTZ function
13 // * 20/06/2014 [SL] added FTZ function
14 // * 22/06/2013 [YO] added float/double/quad variants of some foreign functions
15 // * 28/06/2005 [YO] postfix functions with 'f' to force float version instead of double
16 // * 28/06/2005 [YO] removed 'modf' because it requires a pointer as argument
17
18 /*****
19 *****/
20 FAUST library file
21 Copyright (C) 2003-2016 GRAME, Centre National de Creation Musicale
22 -----
23
24 This program is free software; you can redistribute it and/or modify
25 it under the terms of the GNU Lesser General Public License as
26 published by the Free Software Foundation; either version 2.1 of the
27 License, or (at your option) any later version.
28
29 This program is distributed in the hope that it will be useful,
30 but WITHOUT ANY WARRANTY; without even the implied warranty of
31 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
32 GNU Lesser General Public License for more details.
33
34 You should have received a copy of the GNU Lesser General Public
35 License along with the GNU C Library; if not, write to the Free
36 Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
37 02111-1307 USA.
38
39 EXCEPTION TO THE LGPL LICENSE : As a special exception, you may create a
40 larger FAUST program which directly or indirectly imports this library
41 file and still distribute the compiled code generated by the FAUST
42 compiler, or a modified version of this compiled code, under your own
43 copyright and license. This EXCEPTION TO THE LGPL LICENSE explicitly
44 grants you the right to freely choose the license for the resulting
45 compiled code. In particular the resulting compiled code has no obligation
46 to be LGPL or GPL. For example you are free to choose a commercial or
47 closed source license or any other license if you decide so.
48 *****/
49
50 declare name "Faust Math Library";
51 declare version "2.1";
52 declare author "GRAME";
53 declare copyright "GRAME";

```

```

54 declare license "LGPL with exception";
55
56 //=====Functions Reference=====
57 //=====
58
59
60 //-----'(ma.)SR'-----
61 // Current sampling rate (between 1000Hz and 192000Hz). Constant during
62 // program execution.
63 //
64 // #### Usage
65 //
66 // '''
67 // SR : _
68 // '''
69 //-----
70 SR      = min(192000.0, max(1.0, fconstant(int fSamplingFreq, <math.h>)));
71
72
73 //-----'(ma.)BS'-----
74 // Current block-size. Can change during the execution.
75 //
76 // #### Usage
77 //
78 // '''
79 // BS : _
80 // '''
81 //-----
82 BS      = fvariable(int count, <math.h>);
83
84
85 //-----'(ma.)PI'-----
86 // Constant PI in double precision.
87 //
88 // #### Usage
89 //
90 // '''
91 // PI : _
92 // '''
93 //-----
94 PI      = 3.1415926535897932385;
95
96 //-----'(ma.)INFINITY'-----
97 // Constant INFINITY inherited from 'math.h'.
98 //
99 // #### Usage
100 //
101 // '''
102 // INFINITY : _
103 // '''
104 //-----
105 INFINITY = fconstant(float INFINITY, <math.h>);
106
107 //-----'(ma.)FTZ'-----
108 // Flush to zero: force samples under the "maximum subnormal number"
109 // to be zero. Usually not needed in C++ because the architecture
110 // file take care of this, but can be useful in javascript for instance.
111 //
112 // #### Usage
113 //
114 // '''
115 // _ : ftz : _
116 // '''
117 //
118 // See : <http://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html>
119 //-----
120 FTZ(x)   = x * (abs(x) > 1.17549435e-38);
121

```

```

122 //-----'(ma.)neg'-----
123 // Invert the sign (-x) of a signal.
124 //
125 // #### Usage
126 //
127 // '''
128 // _ : neg : _
129 // '''
130 //-----
131 neg(x) = -x;
132
133
134 //-----'(ma.)sub(x,y)'-----
135 // Subtract 'x' and 'y'.
136 //-----
137 sub(x,y) = y-x;
138
139
140 //-----'(ma.)inv'-----
141 // Compute the inverse (1/x) of the input signal.
142 //
143 // #### Usage
144 //
145 // '''
146 // _ : inv : _
147 // '''
148 //-----
149 inv(x) = 1/x;
150
151
152 //-----'(ma.)cbrt'-----
153 // Computes the cube root of of the input signal.
154 //
155 // #### Usage
156 //
157 // '''
158 // _ : cbrt : _
159 // '''
160 //-----
161 cbrt = ffunction(float cbrtf|cbrt|cbrtl (float), <math.h>,"");
162
163
164 //-----'(ma.)hypot'-----
165 // Computes the euclidian distance of the two input signals
166 // sqrt(x*x+y*y) without undue overflow or underflow.
167 //
168 // #### Usage
169 //
170 // '''
171 // _,_ : hypot : _
172 // '''
173 //-----
174 hypot = ffunction(float hypotf|hypot|hypotl (float, float), <math.h>,"");
175
176
177 //-----'(ma.)ldexp'-----
178 // Takes two input signals: x and n, and multiplies x by 2 to the power n.
179 //
180 // #### Usage
181 //
182 // '''
183 // _,_ : ldexp : _
184 // '''
185 //-----
186 ldexp = ffunction(float ldexpf|ldexp|ldexpl (float, int), <math.h>,"");
187
188
189

```

```

190 //-----'(ma.)scalb'-----
191 // Takes two input signals: x and n, and multiplies x by 2 to the power n.
192 //
193 // #### Usage
194 //
195 // '''
196 // _ : scalb : _
197 // '''
198 //-----
199 scalb      = ffunction(float scalbnf|scalbn|scalbnl (float, int), <math.h>,"");
200
201
202 //-----'(ma.)log1p'-----
203 // Computes log(1 + x) without undue loss of accuracy when x is nearly zero.
204 //
205 // #### Usage
206 //
207 // '''
208 // _ : log1p : _
209 // '''
210 //-----
211 log1p      = ffunction(float log1pf|log1p|log1pl (float), <math.h>,"");
212
213
214 //-----'(ma.)logb'-----
215 // Return exponent of the input signal as a floating-point number.
216 //
217 // #### Usage
218 //
219 // '''
220 // _ : logb : _
221 // '''
222 //-----
223 logb       = ffunction(float logbf|logb|logbl (float), <math.h>,"");
224
225
226 //-----'(ma.)ilogb'-----
227 // Return exponent of the input signal as an integer number.
228 //
229 // #### Usage
230 //
231 // '''
232 // _ : ilogb : _
233 // '''
234 //-----
235 ilogb      = ffunction(int ilogbf|ilogb|ilogbl (float), <math.h>,"");
236
237
238 //-----'(ma.)log2'-----
239 // Returns the base 2 logarithm of x.
240 //
241 // #### Usage
242 //
243 // '''
244 // _ : log2 : _
245 // '''
246 //-----
247 log2(x) = log(x)/log(2.0);
248
249
250 //-----'(ma.)expm1'-----
251 // Return exponent of the input signal minus 1 with better precision.
252 //
253 // #### Usage
254 //
255 // '''
256 // _ : expm1 : _
257 // '''

```



```

258 //-----'(ma.)acosh'-----
259 expm1    = ffunction(float expm1f|expm1|expm1l (float), <math.h>,"");
260
261
262 //-----'(ma.)acosh'-----
263 // Computes the principle value of the inverse hyperbolic cosine
264 // of the input signal.
265 //
266 // #### Usage
267 //
268 // '''
269 // _ : acosh : _
270 // '''
271 //-----
272 acosh     = ffunction(float acoshf|acosh|acoshl (float), <math.h>, "");
273
274
275 //-----'(ma.)asinh'-----
276 // Computes the inverse hyperbolic sine of the input signal.
277 //
278 // #### Usage
279 //
280 // '''
281 // _ : asinh : _
282 // '''
283 //-----
284 asinh     = ffunction(float asinhf|asinh|asinh1 (float), <math.h>, "");
285
286
287 //-----'(ma.)atanh'-----
288 // Computes the inverse hyperbolic tangent of the input signal.
289 //
290 // #### Usage
291 //
292 // '''
293 // _ : atanh : _
294 // '''
295 //-----
296 atanh     = ffunction(float atanhf|atanh|atanhl (float), <math.h>, "");
297
298
299 //-----'(ma.)sinh'-----
300 // Computes the hyperbolic sine of the input signal.
301 //
302 // #### Usage
303 //
304 // '''
305 // _ : sinh : _
306 // '''
307 //-----
308 sinh      = ffunction(float sinhf|sinh|sinhl (float), <math.h>, "");
309
310
311 //-----'(ma.)cosh'-----
312 // Computes the hyperbolic cosine of the input signal.
313 //
314 // #### Usage
315 //
316 // '''
317 // _ : cosh : _
318 // '''
319 //-----
320 cosh      = ffunction(float coshf|cosh|cosh1 (float), <math.h>, "");
321
322
323 //-----'(ma.)tanh'-----
324 // Computes the hyperbolic tangent of the input signal.
325 //

```

```

326 // #### Usage
327 //
328 // '''
329 // _ : tanh : _
330 // '''
331 //-----
332 tanh      = ffunction(float tanhf|tanh|tanh1(float), <math.h>,"");
333
334
335 //-----'(ma.)erf'-----
336 // Computes the error function of the input signal.
337 //
338 // #### Usage
339 //
340 // '''
341 // _ : erf : _
342 // '''
343 //-----
344 erf       = ffunction(float erff|erf|erfl(float), <math.h>,"");
345
346
347 //-----'(ma.)erfc'-----
348 // Computes the complementary error function of the input signal.
349 //
350 // #### Usage
351 //
352 // '''
353 // _ : erfc : _
354 // '''
355 //-----
356 erfc      = ffunction(float erfcf|erfc|erfcl(float), <math.h>,"");
357
358
359 //-----'(ma.)gamma'-----
360 // Computes the gamma function of the input signal.
361 //
362 // #### Usage
363 //
364 // '''
365 // _ : gamma : _
366 // '''
367 //-----
368 gamma     = ffunction(float tgammaf|tgamma|tgammal(float), <math.h>,"");
369
370
371 //-----'(ma.)lgamma'-----
372 // Calculates the natural logarithm of the absolute value of
373 // the gamma function of the input signal.
374 //
375 // #### Usage
376 //
377 // '''
378 // _ : lgamma : _
379 // '''
380 //-----
381 lgamma    = ffunction(float lgammaf|lgamma|lgammal(float), <math.h>,"");
382
383
384 //-----'(ma.)J0'-----
385 // Computes the Bessel function of the first kind of order 0
386 // of the input signal.
387 //
388 // #### Usage
389 //
390 // '''
391 // _ : J0 : _
392 // '''
393 //-----

```

```

394 J0      = ffunction(float j0(float), <math.h>,"");
395
396
397 //-----'(ma.)J1'-----
398 // Computes the Bessel function of the first kind of order 1
399 // of the input signal.
400 //
401 // #### Usage
402 //
403 // '''
404 // _ : J1 : _
405 // '''
406 //-----
407 J1      = ffunction(float j1(float), <math.h>,"");
408
409
410 //-----'(ma.)Jn'-----
411 // Computes the Bessel function of the first kind of order n
412 // (first input signal) of the second input signal.
413 //
414 // #### Usage
415 //
416 // '''
417 // _ : Jn : _
418 // '''
419 //-----
420 Jn      = ffunction(float jn(int, float), <math.h>,"");
421
422
423 //-----'(ma.)Y0'-----
424 // Computes the linearly independent Bessel function of the second kind
425 // of order 0 of the input signal.
426 //
427 // #### Usage
428 //
429 // '''
430 // _ : Y0 : _
431 // '''
432 //-----
433 Y0      = ffunction(float y0(float), <math.h>,"");
434
435
436 //-----'(ma.)Y1'-----
437 // Computes the linearly independent Bessel function of the second kind
438 // of order 1 of the input signal.
439 //
440 // #### Usage
441 //
442 // '''
443 // _ : Y1 : _
444 // '''
445 //-----
446 Y1      = ffunction(float y1(float), <math.h>,"");
447
448
449 //-----'(ma.)Yn'-----
450 // Computes the linearly independent Bessel function of the second kind
451 // of order n (first input signal) of the second input signal.
452 //
453 // #### Usage
454 //
455 // '''
456 // _ : Yn : _
457 // '''
458 //-----
459 Yn      = ffunction(float yn(int, float), <math.h>,"");
460
461

```

```

462 //-----'(ma.)fabs', '(ma.)fmax', '(ma.)fmin
463 // Just for compatibility...
464 //
465 // '''
466 // fabs = abs
467 // fmax = max
468 // fmin = min
469 // '''
470 //-----
471 fabs = abs;
472 fmax = max;
473 fmin = min;
474
475 //-----'(ma.)np2'-----
476 // Gives the next power of 2 of x.
477 //
478 // ### Usage
479 //
480 // '''
481 // np2(n) : _
482 // '''
483 //
484 // Where:
485 //
486 // * 'n': an integer
487 //-----
488 np2 = -(1) <: >>(1)|_ <: >>(2)|_ <: >>(4)|_ <: >>(8)|_ <: >>(16)|_ : +(1);
489
490
491 //-----'(ma.)frac'-----
492 // Gives the fractional part of n.
493 //
494 // ### Usage
495 //
496 // '''
497 // frac(n) : _
498 // '''
499 //
500 // Where:
501 //
502 // * 'n': a decimal number
503 //-----
504 frac(n) = n - floor(n);
505 decimal = frac;
506 // NOTE: decimal does the same thing as frac but using floor instead. JOS uses frac a lot
507 // in filters.lib so we decided to keep that one... decimal is declared though for
508 // backward compatibility.
509 // decimal(n) = n - floor(n);
510
511 //-----'(ma.)modulo'-----
512 // Modulus operation.
513 //
514 // ### Usage
515 //
516 // '''
517 // modulo(x,N) : _
518 // '''
519 //
520 // Where:
521 //
522 // * 'x': the numerator
523 // * 'N': the denominator
524 //-----
525 modulo(x,N) = (x % N + N) % N;
526
527
528 //-----'(ma.)isnan'-----

```

```

529 // Return non-zero if and only if x is a NaN.
530 //
531 // #### Usage
532 //
533 // '''
534 // isnan(x)
535 // _ : isnan : _
536 // '''
537 //
538 // Where:
539 //
540 // * 'x': signal to analyse
541 //-----
542 isnan      = ffunction(int isnan (float), <math.h>, "");
543 nextafter  = ffunction(float nextafter(float, float), <math.h>, "");
544
545
546 //-----'(ma.)chebychev'-----
547 // Chebychev transformation of order n.
548 //
549 // #### Usage
550 //
551 // '''
552 // _ : chebychev(n) : _
553 // '''
554 //
555 // Where:
556 //
557 // * 'n': the order of the polynomial
558 //
559 // #### Semantics
560 //
561 // '''
562 // T[0](x) = 1,
563 // T[1](x) = x,
564 // T[n](x) = 2x*T[n-1](x) - T[n-2](x)
565 // '''
566 //
567 // #### Reference
568 //
569 // <http://en.wikipedia.org/wiki/Chebyshev_polynomial>
570 //-----
571 chebychev(0) = !:1;
572 chebychev(1) = _;
573 chebychev(n) = _ <: *(2)*chebychev(n-1)-chebychev(n-2);
574
575
576 //-----'(ma.)chebypoly'-----
577 // Linear combination of the first Chebyshev polynomials.
578 //
579 // #### Usage
580 //
581 // '''
582 // _ : chebypoly((c0,c1,...,cn)) : _
583 // '''
584 //
585 // Where:
586 //
587 // * 'cn': the different Chebyshev polynomials such that:
588 // chebypoly((c0,c1,...,cn)) = Sum of chebychev(i)*ci
589 //
590 // #### Reference
591 //
592 // <http://www.csounds.com/manual/html/chebypoly.html>
593 //-----
594 chebypoly(lcoef) = _ <: L(0,lcoef) :> _
595   with {
596     L(n,(c,cs)) = chebychev(n)*c, L(n+1,cs);

```

```

597     L(n,c)      = chebychev(n)*c;
598 };
599
600
601 //-----'(ma.)diffn'-----
602 // Negated first-order difference.
603 //
604 // #### Usage
605 //
606 // '
607 // _ : diffn : _
608 // '
609 //-----
610 diffn(x) = x' - x; // negated first-order difference
611
612 //-----'(ma.)signum'-----
613 // The signum function signum(x) is defined as
614 // -1 for x<0, 0 for x==0, and 1 for x>0;
615 //
616 // #### Usage
617 //
618 // '
619 // _ : signum : _
620 // '
621 //-----
622 signum(x) = (x>0)-(x<0);

```

#### Listing 4: basics.lib

```

1  //##### basics.lib #####
2  // A library of basic elements. Its official prefix is 'ba'.
3  //#####
4  // A library of basic elements for Faust organized in 5 sections:
5  //
6  // * Conversion Tools
7  // * Counters and Time/Tempo Tools
8  // * Array Processing/Pattern Matching
9  // * Selectors (Conditions)
10 // * Other Tools (Misc)
11
12 //#####
13
14 /*****
15 *****/
16 FAUST library file, GRAME section
17
18 Except where noted otherwise, Copyright (C) 2003-2017 by GRAME,
19 Centre National de Creation Musicale.
20 -----
21 GRAME LICENSE
22
23 This program is free software; you can redistribute it and/or modify
24 it under the terms of the GNU Lesser General Public License as
25 published by the Free Software Foundation; either version 2.1 of the
26 License, or (at your option) any later version.
27
28 This program is distributed in the hope that it will be useful,
29 but WITHOUT ANY WARRANTY; without even the implied warranty of
30 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
31 GNU Lesser General Public License for more details.
32
33 You should have received a copy of the GNU Lesser General Public
34 License along with the GNU C Library; if not, write to the Free
35 Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA

```

```

36 02111-1307 USA.
37
38 EXCEPTION TO THE LGPL LICENSE : As a special exception, you may create a
39 larger FAUST program which directly or indirectly imports this library
40 file and still distribute the compiled code generated by the FAUST
41 compiler, or a modified version of this compiled code, under your own
42 copyright and license. This EXCEPTION TO THE LGPL LICENSE explicitly
43 grants you the right to freely choose the license for the resulting
44 compiled code. In particular the resulting compiled code has no obligation
45 to be LGPL or GPL. For example you are free to choose a commercial or
46 closed source license or any other license if you decide so.
47 *****
48 *****/
49
50 ma = library("maths.lib");
51 ro = library("routes.lib");
52 ba = library("basics.lib"); // so functions here can be copy/pasted out
53
54 declare name "Faust Basic Element Library";
55 declare version "0.0";
56
57 //=====Conversion Tools=====
58 //=====
59
60
61 //-----'(ba.)samp2sec'-----
62 // Converts a number of samples to a duration in seconds.
63 // 'samp2sec' is a standard Faust function.
64 //
65 // ### Usage
66 //
67 // ''
68 // samp2sec(n) : _
69 // ''
70 //
71 // Where:
72 //
73 // * 'n': number of samples
74 //-----
75 samp2sec = /(ma.SR);
76
77
78 //-----'(ba.)sec2samp'-----
79 // Converts a duration in seconds to a number of samples.
80 // 'sec2samp' is a standard Faust function.
81 //
82 // ### Usage
83 //
84 // ''
85 // sec2samp(d) : _
86 // ''
87 //
88 // Where:
89 //
90 // * 'd': duration in seconds
91 //-----
92 sec2samp = *(ma.SR);
93
94
95 //-----'(ba.)db2linear'-----
96 // Converts a loudness in dB to a linear gain (0-1).
97 // 'db2linear' is a standard Faust function.
98 //
99 // ### Usage
100 //
101 // ''
102 // db2linear(l) : _
103 // ''

```

```

104 //
105 // Where:
106 //
107 // * 'l': loudness in dB
108 //-----
109 db2linear(n) = pow(10, n/20.0);
110
111
112 //-----'(ba.)linear2db'-----
113 // Converts a linear gain (0-1) to a loudness in dB.
114 // 'linear2db' is a standard Faust function.
115 //
116 // ### Usage
117 //
118 // ''
119 // linear2db(g) : _
120 // ''
121 //
122 // Where:
123 //
124 // * 'g': a linear gain
125 //-----
126 linear2db(n) = 20*log10(n);
127
128
129 //-----'(ba.)lin2LogGain'-----
130 // Converts a linear gain (0-1) to a log gain (0-1).
131 //
132 // ### Usage
133 //
134 // ''
135 // _ : lin2LogGain : _
136 // ''
137 //-----
138 lin2LogGain = _ <: _*_;
139
140
141 //-----'(ba.)log2LinGain'-----
142 // Converts a log gain (0-1) to a linear gain (0-1).
143 //
144 // ### Usage
145 //
146 // ''
147 // _ : log2LinGain : _
148 // ''
149 //-----
150 log2LinGain = sqrt;
151
152 // end GRAME section
153 //#####
154 //*****
155 FAUST library file, jos section
156
157 Except where noted otherwise, The Faust functions below in this
158 section are Copyright (C) 2003-2017 by Julius O. Smith III <jos@ccrma.stanford.edu>
159 ([jos](http://ccrma.stanford.edu/~jos/)), and released under the
160 (MIT-style) [STK-4.3](#stk-4.3-license) license.
161
162 The MarkDown comments in this section are Copyright 2016-2017 by Romain
163 Michon and Julius O. Smith III, and are released under the
164 [CCIA4I](https://creativecommons.org/licenses/by/4.0/) license (TODO: if/when Romain agrees)
165
166 *****/
167
168 //-----'(ba.)tau2pole'-----
169 // Returns a real pole giving exponential decay.
170 // Note that t60 (time to decay 60 dB) is ~6.91 time constants.
171 // 'tau2pole' is a standard Faust function.

```



```

172 //
173 // #### Usage
174 //
175 // '''
176 // _ : smooth(tau2pole(tau)) : _
177 // '''
178 //
179 // Where:
180 //
181 // * 'tau': time-constant in seconds
182 //-----
183 tau2pole(tau) = exp(-1.0/(tau*ma.SR));
184
185
186 //-----'(ba.)pole2tau'-----
187 // Returns the time-constant, in seconds, corresponding to the given real,
188 // positive pole in (0,1).
189 // 'pole2tau' is a standard Faust function.
190 //
191 // #### Usage
192 //
193 // '''
194 // pole2tau(pole) : _
195 // '''
196 //
197 // Where:
198 //
199 // * 'pole': the pole
200 //-----
201 pole2tau(pole) = -1.0/(log(pole)*ma.SR);
202
203
204 //-----'(ba.)midikey2hz'-----
205 // Converts a MIDI key number to a frequency in Hz (MIDI key 69 = A440).
206 // 'midikey2hz' is a standard Faust function.
207 //
208 // #### Usage
209 //
210 // '''
211 // midikey2hz(mk) : _
212 // '''
213 //
214 // Where:
215 //
216 // * 'mk': the MIDI key number
217 //-----
218 midikey2hz(mk) = 440.0*pow(2.0, (mk-69.0)/12);
219
220
221 //-----'(ba.)hz2midikey'-----
222 // Converts a frequency in Hz to a MIDI key number (MIDI key 69 = A440).
223 // 'hz2midikey' is a standard Faust function.
224 //
225 // #### Usage
226 //
227 // '''
228 // hz2midikey(f) : _
229 // '''
230 //
231 // Where:
232 //
233 // * 'f': frequency in Hz
234 //-----
235 hz2midikey(f) = 12*ma.log2(f/440.0) + 69.0;
236
237
238 //-----'(ba.)pianokey2hz'-----
239 // Converts a piano key number to a frequency in Hz (piano key 49 = A440).

```

```

240 //
241 // #### Usage
242 //
243 // '''
244 // pianokey2hz(pk) : _
245 // '''
246 //
247 // Where:
248 //
249 // * 'pk': the piano key number
250 //-----
251 pianokey2hz(pk) = 440.0*pow(2.0, (pk-49.0)/12);
252
253
254 //-----'(ba.)hz2pianokey'-----
255 // Converts a frequency in Hz to a piano key number (piano key 49 = A440).
256 //
257 // #### Usage
258 //
259 // '''
260 // hz2pianokey(f) : _
261 // '''
262 //
263 // Where:
264 //
265 // * 'f': frequency in Hz
266 //-----
267 hz2pianokey(f) = 12*ma.log2(f/440.0) + 49.0;
268
269 // end jos section
270 //#####
271 /*****
272 FAUST library file, GRAME section 2
273 *****/
274
275 //=====Counters and Time/Tempo Tools=====
276 //=====
277
278 //-----'(ba.)countdown'-----
279 // Starts counting down from n included to 0. While trig is 1 the output is n.
280 // The countdown starts with the transition of trig from 1 to 0. At the end
281 // of the countdown the output value will remain at 0 until the next trig.
282 // 'countdown' is a standard Faust function.
283 //
284 // #### Usage
285 //
286 // '''
287 // countdown(n,trig) : _
288 // '''
289 //
290 // Where:
291 //
292 // * 'count': the starting point of the countdown
293 // * 'trig': the trigger signal (1: start at 'n'; 0: decrease until 0)
294 //-----
295 countdown(count, trig) = \(c).(if(trig>0, count, max(0, c-1))) ~_;
296
297
298 //-----'(ba.)countup'-----
299 // Starts counting up from 0 to n included. While trig is 1 the output is 0.
300 // The countup starts with the transition of trig from 1 to 0. At the end
301 // of the countup the output value will remain at n until the next trig.
302 // 'countup' is a standard Faust function.
303 //
304 // #### Usage
305 //
306 // '''
307 // countup(n,trig) : _

```

```

308 // '''
309 //
310 // Where:
311 //
312 // * 'count': the maximum count value
313 // * 'trig': the trigger signal (1: start at 0; 0: increase until 'n')
314 //-----
315 countup(count, trig) = \c.(if(trig>0, 0, min(count, c+1))) ~_;
316
317 //-----'(ba.)sweep'-----
318 // Counts from 0 to 'period' samples repeatedly, while 'run' is 1.
319 // Outputs zero while 'run' is 0.
320 //
321 //
322 // ### Usage
323 //
324 // '''
325 // sweep(period,run) : _
326 // '''
327 //-----
328 // Author: Jonatan Liljedahl, markdown by RM
329 sweep = %(int(*:max(1)))~+(1);
330
331 //-----'(ba.)time'-----
332 // A simple timer that counts every samples from the beginning of the process.
333 // 'time' is a standard Faust function.
334 //
335 //
336 // ### Usage
337 //
338 // '''
339 // time : _
340 // '''
341 //-----
342 time = +(1)~_ - 1;
343
344 //-----'(ba.)tempo'-----
345 // Converts a tempo in BPM into a number of samples.
346 //
347 //
348 // ### Usage
349 //
350 // '''
351 // tempo(t) : _
352 // '''
353 //
354 // Where:
355 //
356 // * 't': tempo in BPM
357 //-----
358 tempo(t) = (60*ma.SR)/t;
359
360 //-----'(ba.)period'-----
361 // Basic sawtooth wave of period 'p'.
362 //
363 //
364 // ### Usage
365 //
366 // '''
367 // period(p) : _
368 // '''
369 //
370 // Where:
371 //
372 // * 'p': period as a number of samples
373 //-----
374 // NOTE: may be this should go in oscillators.lib
375 period(p) = %(int(p))~+(1');

```

```

376
377
378 //-----'(ba.)pulse'-----
379 // Pulses (10000) generated at period 'p'.
380 //
381 // #### Usage
382 //
383 // '''
384 // pulse(p) : _
385 // '''
386 //
387 // Where:
388 //
389 // * 'p': period as a number of samples
390 //-----
391 // NOTE: may be this should go in oscillators.lib
392 pulse(p) = period(p)==0;
393
394
395 //-----'(ba.)pulsen'-----
396 // Pulses (11110000) of length 'n' generated at period 'p'.
397 //
398 // #### Usage
399 //
400 // '''
401 // pulsen(n,p) : _
402 // '''
403 //
404 // Where:
405 //
406 // * 'n': the length of the pulse as a number of samples
407 // * 'p': period as a number of samples
408 //-----
409 // NOTE: may be this should go in oscillators.lib
410 pulsen(n,p) = period(p)<n;
411
412
413 //-----'(ba.)cycle'-----
414 // Split nonzero input values into 'n' cycles.
415 //
416 // #### Usage
417 //
418 // '''
419 // _ : cycle(n) <:
420 // '''
421 //
422 // Where:
423 //
424 // * 'n': the number of cycles/output signals
425 //-----
426 // Author: Mike Olsen
427 cycle(n) = _ <: par(i,n,resetCtr(n,(i+1)));
428
429
430 //-----'(ba.)beat'-----
431 // Pulses at tempo 't'.
432 // 'beat' is a standard Faust function.
433 //
434 // #### Usage
435 //
436 // '''
437 // beat(t) : _
438 // '''
439 //
440 // Where:
441 //
442 // * 't': tempo in BPM
443 //-----

```

```

444 beat(t) = pulse(tempo(t));
445
446
447 //-----'(ba.)pulse_countup'-----
448 // Starts counting up pulses. While trig is 1 the output is
449 // counting up, while trig is 0 the counter is reset to 0.
450 //
451 // #### Usage
452 //
453 // '''
454 // _ : pulse_countup(trig) : _
455 // '''
456 //
457 // Where:
458 //
459 // * 'trig': the trigger signal (1: start at next pulse; 0: reset to 0)
460 //-----
461 //TODO: author "Vince"
462 pulse_countup(t) = + ~ _ * t ;
463
464
465 //-----'(ba.)pulse_countdown'-----
466 // Starts counting down pulses. While trig is 1 the output is
467 // counting down, while trig is 0 the counter is reset to 0.
468 //
469 // #### Usage
470 //
471 // '''
472 // _ : pulse_countdown(trig) : _
473 // '''
474 //
475 // Where:
476 //
477 // * 'trig': the trigger signal (1: start at next pulse; 0: reset to 0)
478 //-----
479 //TODO: author "Vince"
480 pulse_countdown(t) = - ~ _ * t ;
481
482
483 //-----'(ba.)pulse_countup_loop'-----
484 // Starts counting up pulses from 0 to n included. While trig is 1 the output is
485 // counting up, while trig is 0 the counter is reset to 0. At the end
486 // of the countup (n) the output value will be reset to 0.
487 //
488 // #### Usage
489 //
490 // '''
491 // _ : pulse_countup_loop(n,trig) : _
492 // '''
493 //
494 // Where:
495 //
496 // * 'n': the highest number of the countup (included) before reset to 0.
497 // * 'trig': the trigger signal (1: start at next pulse; 0: reset to 0)
498 //-----
499 //TODO: author "Vince"
500 pulse_countup_loop(n, t) = + ~ cond(n)*t
501 with {
502   cond(n) = _ <: _ * (_ <= n) ;
503 };
504
505
506 //-----'(ba.)resetCtr'-----
507 // Function that lets through the mth impulse out of
508 // each consecutive group of 'n' impulses.
509 //
510 // #### Usage
511 //

```

```

512 // ''
513 // _ : resetCtr(n,m) : _
514 // ''
515 //
516 // Where:
517 //
518 // * 'n': the total number of impulses being split
519 // * 'm': index of impulse to allow to be output
520 //-----
521 // Author: Mike Olsen
522 resetCtr(n,m) = _ <: (_,pulse_countup_loop(n-1,1)) : (_,(_==m)) : *;
523
524
525 //-----'(ba.)pulse_countdown_loop'-----
526 // Starts counting down pulses from 0 to n included. While trig is 1 the output
527 // is counting down, while trig is 0 the counter is reset to 0. At the end
528 // of the countdown (n) the output value will be reset to 0.
529 //
530 // #### Usage
531 //
532 // ''
533 // _ : pulse_countdown_loop(n,trig) : _
534 // ''
535 //
536 // Where:
537 //
538 // * 'n': the highest number of the countup (included) before reset to 0.
539 // * 'trig': the trigger signal (1: start at next pulse; 0: reset to 0)
540 //-----
541 //TODO: author "Vince:
542 pulse_countdown_loop(n, t) = - ~ cond(n)*t
543 with {
544   cond(n) = _ <: _ * (_ >= n) ;
545 };
546
547
548 //=====Array Processing/Pattern Matching=====
549 //=====
550
551 //-----'(ba.)count'-----
552 // Count the number of elements of list l.
553 // 'count' is a standard Faust function.
554 //
555 // #### Usage
556 //
557 // ''
558 // count(l)
559 // count ((10,20,30,40)) -> 4
560 // ''
561 //
562 // Where:
563 //
564 // * 'l': list of elements
565 //-----
566 count ((xs, xxs)) = 1 + count(xxs);
567 count (xx) = 1;
568
569
570 //-----'(ba.)take'-----
571 // Take an element from a list.
572 // 'take' is a standard Faust function.
573 //
574 // #### Usage
575 //
576 // ''
577 // take(e,l)
578 // take(3,(10,20,30,40)) -> 30
579 // ''

```

```

580 //
581 // Where:
582 //
583 // * 'p': position (starting at 1)
584 // * 'l': list of elements
585 //-----
586 take (1, (xs, xxs)) = xs;
587 take (1, xs)       = xs;
588 take (nn, (xs, xxs)) = take (nn-1, xxs);
589
590
591 //-----'(ba.)subseq'-----
592 // Extract a part of a list.
593 //
594 // #### Usage
595 //
596 // '''
597 // subseq(l, p, n)
598 // subseq((10,20,30,40,50,60), 1, 3) -> (20,30,40)
599 // subseq((10,20,30,40,50,60), 4, 1) -> 50
600 // '''
601 //
602 // Where:
603 //
604 // * 'l': list
605 // * 'p': start point (0: begin of list)
606 // * 'n': number of elements
607 //
608 // #### Note:
609 //
610 // Faust doesn't have proper lists. Lists are simulated with parallel
611 // compositions and there is no empty list
612 //-----
613 subseq((head, tail), 0, 1) = head;
614 subseq((head, tail), 0, n) = head, subseq(tail, 0, n-1);
615 subseq((head, tail), p, n) = subseq(tail, p-1, n);
616 subseq(head, 0, n)        = head;
617
618
619 //=====Selectors (Conditions)=====
620 //=====
621
622 //-----'(ba.)if'-----
623 // if-then-else implemented with a select2.
624 //
625 // #### Usage
626 //
627 // * 'if(c, t, e) : _'
628 //
629 // Where:
630 //
631 // * 'c': condition
632 // * 't': signal selected while c is true
633 // * 'e': signal selected while c is false
634 //-----
635 if(cond,thn,els) = select2(cond,els,thn);
636 // TODO: perhaps it would make more sense to have an if(a,b) and an ifelse(a,b,c)?
637
638 //-----'(ba.)selector'-----
639 // Selects the ith input among n at compile time.
640 //
641 // #### Usage
642 //
643 // '''
644 // selector(i,n)
645 // _,-,-,- : selector(2,4) : _ // selects the 3rd input among 4
646 // '''
647 //

```

```

648 // Where:
649 //
650 // * 'i': input to select ('int', numbered from 0, known at compile time)
651 // * 'n': number of inputs ('int', known at compile time, 'n > i')
652 //
653 // There is also cselector for selecting among complex input signals of the form (real,imag).
654 //
655 //-----
656 selector(i,n) = par(j, n, S(i, j)) with { S(i,i) = _; S(i,j) = !; };
657 cselector(i,n) = par(j, n, S(i, j)) with { S(i,i) = (_,_); S(i,j) = (!,!); }; // for complex
        numbers
658
659
660 //-----'(ba.)selectn'-----
661 // Selects the ith input among N at run time.
662 //
663 // #### Usage
664 //
665 // ''
666 // selectn(N,i)
667 // _-,_,_ : selectn(4,2) : _ // selects the 3rd input among 4
668 // ''
669 //
670 // Where:
671 //
672 // * 'N': number of inputs (int, known at compile time, N > 0)
673 // * 'i': input to select (int, numbered from 0)
674 //
675 // #### Example test program
676 //
677 // ''
678 // N=64;
679 // process = par(n,N, (par(i,N,i) : selectn(N,n)));
680 // ''
681 //-----
682 selectn(N,i) = S(N,0)
683 with {
684     S(1,offset) = _;
685     S(n,offset) = S(left, offset), S(right, offset+left) : select2(i >= offset+left)
686         with {
687             right = int(n/2);
688             left = n-right;
689         };
690 };
691
692
693 //-----'(ba.)select2stereo'-----
694 // Select between 2 stereo signals.
695 //
696 // #### Usage
697 //
698 // ''
699 // _-,_,_ : select2stereo(bpc) : _-,_,_
700 // ''
701 //
702 // Where:
703 //
704 // * 'bpc': the selector switch (0/1)
705 //-----
706 select2stereo(bpc) = ro.cross2 : select2(bpc), select2(bpc) : _,_;
707 //=====Other=====
708 //=====
709
710 //-----'(ba.)latch'-----
711 // Latch input on positive-going transition of "clock" ("sample-and-hold").
712 //
713 // #### Usage
714 //

```



```

715 // ''
716 // _ : latch(clocksig) : _
717 // ''
718 //
719 // Where:
720 //
721 // * 'clocksig': hold trigger (0 for hold, 1 for bypass)
722 //-----
723 latch(c,x) = x * s : + ~ *(1-s) with { s = ((c'<=0)&(c>0)); };
724
725
726 //-----'(ba.)sAndH'-----
727 // Sample And Hold.
728 // 'sAndH' is a standard Faust function.
729 //
730 // #### Usage
731 //
732 // ''
733 // _ : sAndH(t) : _
734 // ''
735 //
736 // Where:
737 //
738 // * 't': hold trigger (0 for hold, 1 for bypass)
739 //-----
740 // Author: RM
741 sAndH(t) = select2(t,_,_)~_;
742
743
744 //-----'(ba.)downSample'-----
745 // Down sample a signal. WARNING: this function doesn't change the
746 // rate of a signal, it just holds samples...
747 // 'downSample' is a standard Faust function.
748 //
749 // #### Usage
750 //
751 // ''
752 // _ : downSample(freq) : _
753 // ''
754 //
755 // Where:
756 //
757 // * 'freq': new rate in Hz
758 //-----
759 // Author: RM
760 downSample(freq) = sAndH(hold)
761 with{
762   hold = time%int(ma.SR/freq) == 0;
763 };
764
765
766 //-----'(ba.)peakhold'-----
767 // Outputs current max value above zero.
768 //
769 // #### Usage
770 //
771 // ''
772 // _ : peakhold(mode) : _;
773 // ''
774 //
775 // Where:
776 //
777 // 'mode' means: 0 - Pass through. A single sample 0 trigger will work as a reset.
778 // 1 - Track and hold max value.
779 //-----
780 // TODO: author Jonatan Liljedahl, revised by RM
781 peakhold = (*,_.max) ~ _;
782

```

```

783 //-----'(ba.)peakholder'-----
784 // Tracks abs peak and holds peak for 'holdtime' samples.
785 //
786 // #### Usage
787 //
788 // '''
789 // _ : peakholder(holdtime) : _;
790 // '''
791 //-----
792 // TODO: author Jonatan Liljedahl
793 peakholder(holdtime) = peakhold2 ~ reset : (!,_) with {
794   reset = ba.sweep(holdtime) > 0;
795   // first out is gate that is 1 while holding last peak
796   peakhold2 = _,abs <: peakhold,!,_ <: >=,_,!;
797 };
798
799
800 //-----'(ba.)impulsify'-----
801 // Turns the signal from a button into an impulse (1,0,0,... when
802 // button turns on).
803 // 'impulsify' is a standard Faust function.
804 //
805 // #### Usage
806 //
807 // '''
808 // button("gate") : impulsify ;
809 // '''
810 //-----
811 impulsify = _ <: _,mem : - : >(0);
812
813
814 //-----'(ba.)automat'-----
815 // Record and replay to the values the input signal in a loop.
816 //
817 // #### Usage
818 //
819 // '''
820 // hslider(...) : automat(bps, size, init) : _
821 // '''
822 //-----
823 automat(bps, size, init, input) = rwttable(size+1, init, windex, input, rindex)
824 with {
825   clock = beat(bps);
826   rindex = int(clock) : (+ : %(size)) ~ _; // each clock read the next entry of the table
827   windex = if (timeToRenew, rindex, size); // we ignore input unless it is time to renew
828   timeToRenew = int(clock) & (inputHasMoved | (input <= init));
829   inputHasMoved = abs(input-input') : countfrom(int(clock)') : >(0);
830   countfrom(reset) = (+ : if(reset, 0, _) ~ _;
831 };
832
833
834 //-----'(ba.)bpf'-----
835 // bpf is an environment (a group of related definitions) that can be used to
836 // create break-point functions. It contains three functions :
837 //
838 // * 'start(x,y)' to start a break-point function
839 // * 'end(x,y)' to end a break-point function
840 // * 'point(x,y)' to add intermediate points to a break-point function
841 //
842 // A minimal break-point function must contain at least a start and an end point :
843 //
844 // '''
845 // f = bpf.start(x0,y0) : bpf.end(x1,y1);
846 // '''
847 //
848 // A more involved break-point function can contains any number of intermediate
849 // points:

```

```

851 //
852 // '''
853 // f = bpf.start(x0,y0) : bpf.point(x1,y1) : bpf.point(x2,y2) : bpf.end(x3,y3);
854 // '''
855 //
856 // In any case the 'x_{i}' must be in increasing order (for all 'i', 'x_{i} < x_{i+1}').
857 // For example the following definition :
858 //
859 // '''
860 // f = bpf.start(x0,y0) : ... : bpf.point(xi,yi) : ... : bpf.end(xn,yn);
861 // '''
862 //
863 // implements a break-point function f such that :
864 //
865 // * 'f(x) = y_{0}' when 'x < x_{0}'
866 // * 'f(x) = y_{n}' when 'x > x_{n}'
867 // * 'f(x) = y_{i} + (y_{i+1}-y_{i})*(x-x_{i})/(x_{i+1}-x_{i})' when 'x_{i} <= x'
868 // and 'x < x_{i+1}'
869 //
870 // 'bpf' is a standard Faust function.
871 //-----
872 bpf = environment
873 {
874   // Start a break-point function
875   start(x0,y0) = \x.(x0,y0,x,y0);
876   // Add a break-point
877   point(x1,y1) = \x0,y0,x,y.(x1, y1, x, if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(
878     x1-x0), y1));
879   // End a break-point function
880   end (x1,y1) = \x0,y0,x,y.(if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(x1-x0), y1));
881 };
882
883 //-----'(ba.)listInterp'-----
884 // Linearly interpolates between the elements of a list.
885 //
886 // #### Usage
887 //
888 // '''
889 // foo = listInterp((800,400,350,450,325),index);
890 // i = 1.69; // range is 0-4
891 // process = foo(i);
892 // '''
893 //
894 // Where:
895 //
896 // * 'index': the index (float) to interpolate between the different values.
897 // The range of 'index' depends on the size of the list.
898 //-----
899 // Author: RM
900 listInterp(v) =
901   bpf.start(0,take(1,v)) :
902   seq(i,count(v)-2,bpf.point(i+1,take(i+2,v))) :
903   bpf.end(count(v)-1,take(count(v),v));
904
905 //-----'(ba.)bypass1'-----
906 // Takes a mono input signal, route it to 'e' and bypass it if 'bpc = 1'.
907 // 'bypass1' is a standard Faust function.
908 //
909 // #### Usage
910 //
911 // '''
912 // _ : bypass1(bpc,e) : _
913 // '''
914 //
915 // Where:
916 //
917 //

```

```

918 // * 'bpc': bypass switch (0/1)
919 // * 'e': a mono effect
920 //-----
921 // Author: JJS
922 // License: STK-4.3
923 bypass1(bpc,e) = _ <: select2(bpc,(inswitch:e),_)
924   with {
925     inswitch = select2(bpc,_,0);
926   };
927
928 //-----'(ba.)bypass2'-----
929 // Takes a stereo input signal, route it to 'e' and bypass it if 'bpc = 1'.
930 // 'bypass2' is a standard Faust function.
931 //
932 // ### Usage
933 //
934 // '''
935 // _ : bypass2(bpc,e) : _,_
936 // '''
937 //
938 // Where:
939 //
940 // * 'bpc': bypass switch (0/1)
941 // * 'e': a stereo effect
942 //-----
943 // Author: JJS
944 // License: STK-4.3
945 bypass2(bpc,e) = _,_ <: ((inswitch:e),_,_) : select2stereo(bpc)
946   with {
947     inswitch = _,_ : (select2(bpc,_,0), select2(bpc,_,0)) : _,_;
948   };
949
950 //-----'(ba.)bypass1to2'-----
951 // Bypass switch for effect 'e' having mono input signal and stereo output.
952 // Effect 'e' is bypassed if 'bpc = 1'.
953 // 'bypass1to2' is a standard Faust function.
954 //
955 // ### Usage
956 //
957 // '''
958 // _ : bypass1(bpc,e) : _,_
959 // '''
960 //
961 // Where:
962 //
963 // * 'bpc': bypass switch (0/1)
964 // * 'e': a mono-to-stereo effect
965 //-----
966 // Author: JJS
967 // License: STK-4.3
968 bypass1to2(bpc,e) = _ <: ((inswitch:e),_,_) : ba.select2stereo(bpc)
969   with {
970     inswitch = select2(bpc,_,0);
971   };
972
973 //-----'(ba.)toggle'-----
974 // Triggered by the change of 0 to 1, it toggles the output value
975 // between 0 and 1.
976 //
977 // ### Usage
978 //
979 // '''
980 // _ : toggle : _
981 // '''
982 // ### Examples
983 //
984 // '''
985 // button("toggle") : toggle : v bargraph("output", 0, 1)

```

```

986 // (an.amp_follower(0.1) > 0.01) : toggle : vbargraph("output", 0, 1) // takes audio input
987 // '''
988 //
989 //-----
990 // TODO: author "Vince"
991 toggle = trig : loop
992   with {
993     trig(x) = (x-x') == 1;
994     loop = ( _ != _ ) ~ _ ;
995   };
996
997
998 //-----'(ba.)on_and_off'-----
999 // The first channel set the output to 1, the second channel to 0.
1000 //
1001 // #### Usage
1002 //
1003 // '''
1004 // _ , _ : on_and_off : _
1005 // '''
1006 //
1007 // #### Example
1008 //
1009 // '''
1010 // button("on"), button("off") : on_and_off : vbargraph("output", 0, 1)
1011 // '''
1012 //
1013 //-----
1014 // TODO: author "Vince"
1015 on_and_off(a, b) = (a : trig) : loop(b)
1016   with {
1017     trig(x) = (x-x') == 1;
1018     loop(b) = + ~ ( _ >= 1 ) * ((b : trig) == 0) ;
1019   };
1020
1021
1022 //-----'(ba.)selectoutn'-----
1023 // Route input to the output among N at run time.
1024 //
1025 // #### Usage
1026 //
1027 // '''
1028 // _ : selectoutn(n, s) : _,...n
1029 // '''
1030 //
1031 // Where:
1032 //
1033 // * 'n': number of outputs (int, known at compile time, N > 0)
1034 // * 's': output number to route to (int, numbered from 0) (i.e. slider)
1035 //
1036 // #### Example
1037 //
1038 // '''
1039 // process = 1 : selectoutn(3, sel) : par(i,3,bar) ;
1040 // sel = hslider("volume",0,0,2,1) : int;
1041 // bar = vbargraph("v.bargraph", 0, 1);
1042 // '''
1043 //-----
1044 // TODO: author "Vince"
1045 selectoutn(n, s) = _ <: par(i,n, _*(s==i) ) ;
1046
1047 //=====Sliding Reduce=====
1048 // Provides various operations on the last N samples using a high order
1049 // 'slidingReduce(op,N,maxN,disabledVal,x)' fold-like function :
1050 //
1051 // * 'slidingSumN(n,maxn)': the sliding sum of the last n input samples
1052 // * 'slidingMaxN(n,maxn)': the sliding max of the last n input samples
1053 // * 'slidingMinN(n,maxn)': the sliding min of the last n input samples

```

```

1054 // * 'slidingMeanN(n,maxn)': the sliding mean of the last n input samples
1055 // * 'slidingRMSn(n,maxn)': the sliding RMS of the last n input samples
1056 //
1057 // #### Working Principle
1058 //
1059 // If we want the maximum of the last 8 values, we can do that as:
1060 //
1061 // '''
1062 // simpleMax(x) =
1063 // (
1064 //   (
1065 //     max(x@0,x@1),
1066 //     max(x@2,x@3)
1067 //   ):max
1068 // ),
1069 // (
1070 //   (
1071 //     max(x@4,x@5),
1072 //     max(x@6,x@7)
1073 //   ):max
1074 // )
1075 // :max;
1076 // '''
1077 //
1078 // 'max(x@2,x@3)' is the same as 'max(x@0,x@1)@2' but the latter re-uses a
1079 // value we already computed,so is more efficient. Using the same trick for
1080 // values 4 trough 7, we can write:
1081 //
1082 // '''
1083 // efficientMax(x)=
1084 // (
1085 //   (
1086 //     max(x@0,x@1),
1087 //     max(x@0,x@1)@2
1088 //   ):max
1089 // ),
1090 // (
1091 //   (
1092 //     max(x@0,x@1),
1093 //     max(x@0,x@1)@2
1094 //   ):max@4
1095 // )
1096 // :max;
1097 // '''
1098 //
1099 // We can rewrite it recursively, so it becomes possible to get the maximum at
1100 // have any number of values, as long as it's a power of 2.
1101 //
1102 // '''
1103 // recursiveMax =
1104 // case {
1105 //   (1,x) => x;
1106 //   (N,x) => max(recursiveMax(N/2,x) , recursiveMax(N/2,x)@(N/2));
1107 // };
1108 // '''
1109 //
1110 // What if we want to look at a number of values that's not a power of 2?
1111 // For each value, we will have to decide whether to use it or not.
1112 // If N is bigger than the index of the value, we use it, otherwise we replace
1113 // it with ('0-(ma.INFINITY)'):
1114 //
1115 // '''
1116 // variableMax(N,x) =
1117 // max(
1118 //   max(
1119 //     (
1120 //       (x@0 : useVal(0)),
1121 //       (x@1 : useVal(1))

```

```

1122 //      ):max,
1123 //      (
1124 //          (x@2 : useVal(2)),
1125 //          (x@3 : useVal(3))
1126 //      ):max
1127 //  ),
1128 //  max(
1129 //      (
1130 //          (x@4 : useVal(4)),
1131 //          (x@5 : useVal(5))
1132 //      ):max,
1133 //      (
1134 //          (x@6 : useVal(6)),
1135 //          (x@7 : useVal(7))
1136 //      ):max
1137 //  )
1138 // )
1139 // with{
1140 //   useVal(i) = select2( (N>=i) , (0-(ma.INFINITY)),_);
1141 // };
1142 // '''
1143 //
1144 // Now it becomes impossible to re-use any values. To fix that let's first look
1145 // at how we'd implement it using recursiveMax, but with a fixed N that is not
1146 // a power of 2. For example, this is how you'd do it with 'N=3':
1147 //
1148 // '''
1149 // binaryMaxThree(x) =
1150 // (
1151 //   recursiveMax(1,x)@0, // the first x
1152 //   recursiveMax(2,x)@1 // the second and third x
1153 // ):max;
1154 // '''
1155 //
1156 // 'N=6'
1157 //
1158 // '''
1159 // binaryMaxSix(x) =
1160 // (
1161 //   recursiveMax(2,x)@0, // first two
1162 //   recursiveMax(4,x)@2 // third through sixth
1163 // ):max;
1164 // '''
1165 //
1166 // Note that 'recursiveMax(2,x)' is used at a different delay than in
1167 // 'binaryMaxThree', since it represents 1 and 2, not 2 and 3. Each block is
1168 // delayed the combined size of the previous blocks.
1169 //
1170 // 'N=7'
1171 //
1172 // '''
1173 // binaryMaxSeven(x) =
1174 // (
1175 //   (
1176 //     recursiveMax(1,x)@0, // first x
1177 //     recursiveMax(2,x)@1 // second and third
1178 //   ):max,
1179 //   (
1180 //     recursiveMax(4,x)@3 // fourth through seventh
1181 //   )
1182 // ):max;
1183 // '''
1184 //
1185 // To make a variable version, we need to know which powers of two are used,
1186 // and at which delay time.
1187 //
1188 // Then it becomes a matter of:
1189 //

```

```

1190 // * lining up all the different block sizes in parallel: the first 'par()'
1191 // statement
1192 // * delaying each the appropriate amount: 'sumOfPrevBlockSizes()'
1193 // * turning it on or off: 'useVal()'
1194 // * getting the maximum of all of them: 'combine()'
1195 //
1196 // In faust, we can only do that for a fixed maximum number of values: 'maxN'
1197 //
1198 // '''
1199 // variableBinaryMaxN(N,maxN,x) =
1200 // par(i,maxNrBits,recursiveMax(pow2(i),x)@sumOfPrevBlockSizes(N,maxN,i) : useVal(i)) :
1201 //   combine(maxNrBits) with {
1202 //     // The sum of all the sizes of the previous blocks
1203 //     sumOfPrevBlockSizes(N,maxN,0) = 0;
1204 //     sumOfPrevBlockSizes(N,maxN,i) = (ba.subseq((allBlockSizes(N,maxN)),0,i):>_);
1205 //     allBlockSizes(N,maxN) = par(i, maxNrBits, pow2(i) * isUsed(i) );
1206 //     maxNrBits = int2nrOfBits(maxN);
1207 //     // get the maximum of all blocks
1208 //     combine(2) = max;
1209 //     combine(N) = max(combine(N-1),_);
1210 //     // Decide whether or not to use a certain value, based on N
1211 //     useVal(i) = select2( isUsed(i), (0-(ma.INFINITY)),_);
1212 //     isUsed(i) = ba.take(i+1,(int2bin(N,maxN)));
1213 //   };
1214 // '''
1215 // =====
1216 // Section contributed by Bart Brouns (bart@magnetophon.nl).
1217 // SPDX-License-Identifier: GPL-3.0
1218 // Copyright (C) 2018 Bart Brouns
1219 // -----
1220 // Fold-like high order function. Apply a commutative binary operation '<op>' to
1221 // the last '<n>' consecutive samples of a signal '<x>'. For example :
1222 // 'slidingReduce(max,128,128,-(ma.INFINITY))' will compute the maximum of the last
1223 // 128 samples. The output is updated each sample, unlike reduce, where the
1224 // output is constant for the duration of a block
1225 //
1226 // ### Usage
1227 //
1228 // '''
1229 // _ : slidingReduce(op,N,maxN,disabledVal) : _
1230 // '''
1231 //
1232 // Where:
1233 //
1234 // * 'N': the number of values to process
1235 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1236 // * 'op': the operator. Needs to be a commutative one.
1237 // * 'disabledVal': the value to use when we want to ignore a value.
1238 //
1239 // In other words, 'op(x,disabledVal)' should equal to 'x'. For example,
1240 // '+ (x,0)' equals 'x' and 'min(x,ma.INFINITY)' equals 'x'. So if we want to
1241 // calculate the sum, we need to give 0 as 'disabledVal', and if we want the
1242 // minimum, we need to give 'ma.INFINITY' as 'disabledVal'.
1243 // -----
1244 slidingReduce(op,N,maxN,disabledVal,x) =
1245   par(i,maxNrBits,fixedDelayOp(pow2(i),x)@sumOfPrevBlockSizes(N,maxN,i)
1246     : useVal(i)) : combine(maxNrBits)
1247   with {
1248     // apply <op> to the last <N> values of <x>, where <N> is fixed
1249     fixedDelayOp = case {
1250       (1,x) => x;
1251       (N,x) => op(fixedDelayOp(N/2,x) , fixedDelayOp(N/2,x)@(N/2));
1252     };
1253     // The sum of all the sizes of the previous blocks
1254     sumOfPrevBlockSizes(N,maxN,0) = 0;
1255     sumOfPrevBlockSizes(N,maxN,i) = (subseq((allBlockSizes(N,maxN)),0,i):>_);
1256     allBlockSizes(N,maxN) = par(i, maxNrBits, (pow2(i)) * isUsed(i) );

```



```

1257 maxNrBits = int2nrOfBits(maxN);
1258 // Apply <op> to <N> parallel inputsignals
1259 combine(2) = op;
1260 combine(N) = op(combine(N-1),_);
1261 // Decide wether or not to use a certain value, based on N
1262 // Basically only the second <select2> is needed,
1263 // but this version also works for N == 0
1264 // 'works' in this case means 'does the same as reduce
1265 useVal(i) =
1266     _ <: select2(
1267         (i==0) & (N==0) ,
1268         select2( isUsed(i) , disabledVal,_),
1269         _
1270     );
1271 // useVal(i) =
1272 //     select2( isUsed(i) , disabledVal,_);
1273 isUsed(i) = take(i+1,(int2bin(N,maxN)));
1274 pow2(i) = 1<<i;
1275 // same as:
1276 // pow2(i) = int(pow(2,i));
1277 // but in the block diagram, it will be displayed as a number, instead of a formula
1278
1279 // convert N into a list of ones and zeros
1280 int2bin(N,maxN) = par(j,int2nrOfBits(maxN),int(floor(N/(pow2(j))))%2);
1281 // calculate how many ones and zeros are needed to represent maxN
1282 int2nrOfBits(0) = 0;
1283 int2nrOfBits(maxN) = int(floor(log(maxN)/log(2))+1);
1284 };
1285
1286
1287 //-----'(ba.)slidingSumN'-----
1288 // The sliding sum of the last n input samples.
1289 //
1290 // #### Usage
1291 //
1292 // '''
1293 // _ : slidingSumN(N,maxN) : _
1294 // '''
1295 //
1296 // Where:
1297 //
1298 // * 'N': the number of values to process
1299 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1300 //-----
1301 slidingSumN(n,maxn) = slidingReduce(+,n,maxn,0);
1302
1303
1304 //-----'(ba.)slidingMaxN'-----
1305 // The sliding maximum of the last n input samples.
1306 //
1307 // #### Usage
1308 //
1309 // '''
1310 // _ : slidingMaxN(N,maxN) : _
1311 // '''
1312 //
1313 // Where:
1314 //
1315 // * 'N': the number of values to process
1316 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1317 //-----
1318 slidingMaxN(n,maxn) = slidingReduce(max,n,maxn,-(ma.INFINITY));
1319
1320
1321 //-----'(ba.)slidingSumN'-----
1322 // The sliding minimum of the last n input samples.
1323 //
1324 // #### Usage

```

```

1325 //
1326 // '''
1327 // _ : slidingMinN(N,maxN) : _
1328 // '''
1329 //
1330 // Where:
1331 //
1332 // * 'N': the number of values to process
1333 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1334 //-----
1335 slidingMinN(n,maxn) = slidingReduce(min,n,maxn,ma.INFINITY);
1336
1337
1338 //-----'(ba.)slidingMeanN'-----
1339 // The sliding mean of the last n input samples.
1340 //
1341 // ### Usage
1342 //
1343 // '''
1344 // _ : slidingMeanN(N,maxN) : _
1345 // '''
1346 //
1347 // Where:
1348 //
1349 // * 'N': the number of values to process
1350 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1351 //-----
1352 slidingMeanN(n,maxn) = slidingSumN(n,maxn)/n;
1353
1354
1355 //-----'(ba.)slidingRMSn'-----
1356 // The root mean square of the last n input samples.
1357 //
1358 // ### Usage
1359 //
1360 // '''
1361 // _ : slidingRMSn(N,maxN) : _
1362 // '''
1363 //
1364 // Where:
1365 //
1366 // * 'N': the number of values to process
1367 // * 'maxN': the maximum number of values to process, needs to be a power of 2
1368 //-----
1369 slidingRMSn(n,maxn) = pow(2):slidingMeanN(n,maxn) : sqrt;
1370
1371
1372 ///////////////////////////////////////////////////////////////////Deprecated Functions/////////////////////////////////////////////////////////////////
1373 // This section implements functions that used to be in music.lib but that are now
1374 // considered as "deprecated".
1375 ///////////////////////////////////////////////////////////////////
1376
1377 millisec = ma.SR/1000.0;
1378
1379 time1s = hslider("time", 0, 0, 1000, 0.1)*millisec;
1380 time2s = hslider("time", 0, 0, 2000, 0.1)*millisec;
1381 time5s = hslider("time", 0, 0, 5000, 0.1)*millisec;
1382 time10s = hslider("time", 0, 0, 10000, 0.1)*millisec;
1383 time21s = hslider("time", 0, 0, 21000, 0.1)*millisec;
1384 time43s = hslider("time", 0, 0, 43000, 0.1)*millisec;

```

