

Contents

1	Technical spec	3
2	MIDI Implementation.....	4
3	WaveSet	4
3.1	Note Sector:	5
3.2	Intensity Layer:.....	7
3.3	Morphing.....	8
4	Wave Table	10
4.1	Wave Table Increment.....	11
4.2	Pitch resolution	12
4.3	Increment lookup for MIDI notes	13
4.4	Populating the Wave Table	13
4.5	Skipping calculation of unheard frequencies.....	14
4.6	Use of a 'rough' WaveTable	14
4.7	Wave Table refresh rate	15
5	Body resonance filter	16
5.1	Band gain levels	16
5.2	Fitting frequencies to bands	18
5.3	Interpolation of filtering	18
5.3.1	Calculation of filter gain for a given frequency.....	19
6	Noise reduction.....	21
6.1	Audio circuit considerations	21
6.2	Pre-emphasis and de-emphasis	21
6.2.1	Circuit de-emphasis.....	21
6.2.2	Tone processor pre-emphasis.....	22
6.3	Automatic Gain control (AGC)	23
6.3.1	AGC Ring buffer and windows	23
6.3.2	AGC output and HAAS delay	25
6.3.3	Calculating the required envelope 'target' gain	26
7	Tone Loudness and Intensity	27
8	Low Frequency Oscillators (LFOs)	28
8.1	Tables	28
8.2	MIDI Controllers and LFO Depth Envelopes.....	29
8.3	Tremolo LFO.....	29
8.3.1	Tremolo frequency.....	29

8.3.2	Tremolo LFO gain	30
8.3.3	Tremolo LFO offset gain	31
8.4	Timbre LFO	33
8.4.1	Timbre envelopes.....	33
8.4.2	Timbre LFO frequency.....	33
8.4.3	Timbre LFO gain	34
8.4.4	Timbre LFO offset gain	34
9	Performances and PatchSets	35

1 Technical spec

Specification	Value
Amplitude limiting method	Look-ahead AGC with two windows of 128 samples each. The AGC guarantees that there is no amplitude overshoot.
Audio Sampling rate	41.7KHz
Connectors	Un-balanced 3.5mm Stereo jack (audio out), USB, MIDI input, 3.5mm power (tip is +ve)
Digital to Analog Controller (DAC)	16-bit resolution (14-bit accuracy), Second-order digital Delta-Sigma modulator, 256x oversampling ratio
Dimensions (PCB)	144.8mm x 94.0mm
Frequency response	27.5Hz (Note A0) to 15kHz (the harmonics are limited by code and prevented from being generated if above 15kHz).
MIDI Control-change refresh rate on module	100Hz. MIDI control-change messages might be sent to the Module at a higher frequency, but internally the Module processes the changes at 100 Hz. This applies to pitch-bend also.
MIDI note range	A0 (MIDI note 21) to C8 (MIDI note 108). This is the range of a standard 88-note keyboard.
Note pitch error	+/- 5% max (which is undetectable by most humans)
Patches	Maximum of 18 saved on the Module, but unlimited in the Patch Editor software.
Performances	Maximum of 7 saved on the Module, but unlimited in the Patch Editor software.
Power consumption	1.3W
Signal-To-Noise (SNR) ratio	Typically around 60dB but can vary depending on the type of waveforms played
Supply current	~ 400mA but can vary depending on tasks being performed
Supply voltage	5V 'wall-wart' regulated DC supply, dropped down to a 3.3V regulated DC board voltage
Synthesis method	Additive synthesis using 32 harmonics in total. This generates a WaveTable per note. The WaveTable refreshes during play enabling 'Timbre Morphing'. Parallel processing of tone generation across 6 'Tone Processor' dsPICs, mixed and outputted by a 'Mixer' dsPIC via an on-board DAC.
System clock	A 16MHz crystal on the PCB is the global clock source for all dsPIC chips. However this frequency is multiplied up internally in each chip to much higher frequencies. The dsPICs run at 70MIPS (Tone Processors) and 40MIPS (Mixer).
System memory	64K bytes, held on a dedicated EEPROM chip. Patch data for up to 18 patches is held on this chip.
Timbre refresh rate	Varies depending on the number of notes (instances) being handled by a Tone Processor chip, and also the number of harmonics being processed, but typically around 50 Hz.
Volume control method	Simple bit-scaling, so there is a reduction in quality at low volume
Weight (PCB)	~100g

2 MIDI Implementation

Function ...	Transmitted	Recognized	Remarks
Basic Channel	-	01-Jun	Depending on Performance
Note number	-	21 - 108	
Velocity			
Note ON	-	0 v=0 - 127	
Note OFF	-	0 v=0 - 127	
Pitch Bend	-	0 0 - 8,192	Bank Select
Control Change			Modulation
0,32	-	0	Breath Controller
1	-	0	Foot Controller
2	-	0	Expression
4	-	0	Effect Controller1
11	-	0	Sustain pedal
12	-	0	18 Patches in memory
64	-	0	
Program Change	-	0 0 - 17	
System Exclusive	-	X	
Common	-	X	
System Real Time	-	X	
Aux Messages			
All Sound Off	-	X	
Reset All Cntrl's	-	X	Turns all note instances off
Local ON/OFF	-	X	
All Notes OFF	-	0	
Active Sense			
Reset	-	X	

0 : Yes

X : No

3 WaveSet

A patch has an associated WaveSet, which is comprised of a set of waveforms. There are actually **75 waveforms** in a WaveSet, split into 15 blocks of 5 :

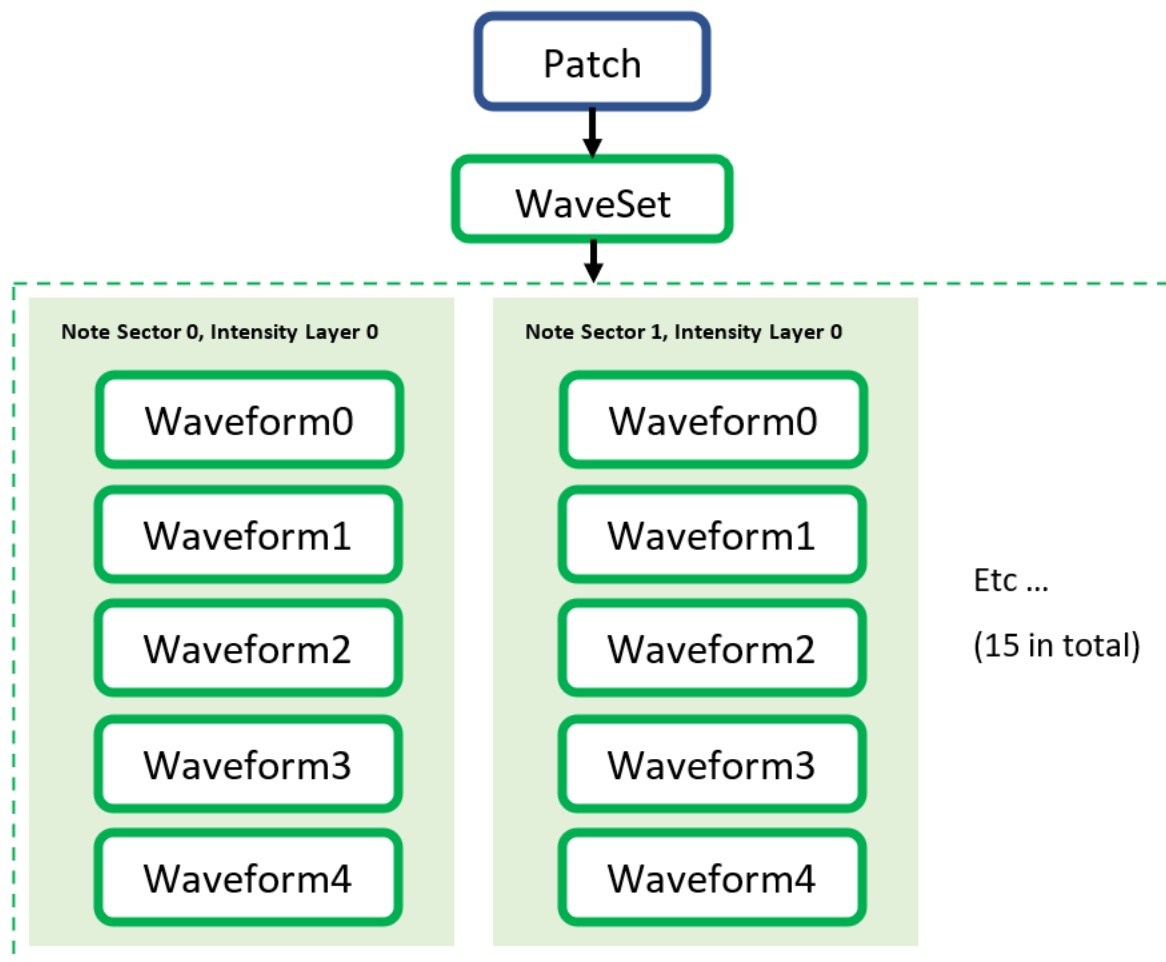


Figure 1 : Waveforms in a WaveSet

Each block represents a specific combination of 'Note Sector' and 'Intensity Layer' .

Within each block are 5 waveforms. These waveforms are made by summing harmonics together (the basis of additive synthesis).

All waveforms can be different in a WaveSet, offering the possibility of a whole range of sounds depending on the variables of Intensity, Note Sector and Waveform control. The Waveform chosen can be based on the value of the Timbre Envelope, or on a MIDI 'continuous controller' value.

3.1 Note Sector:

The notes on a keyboard are divided into 'Sectors'.

Sector		MIDI Note Num	Symbol
Sector 0	0	21	A0
	1	22	A#0

Spectral Additive Synthesis Module – Technical Reference Manual

	2	23	B0
	3	24	C1
	4	25	C#1
	5	26	D1
	6	27	D#1
	7	28	E1
	8	29	F1
	9	30	F#1
	10	31	G1
	11	32	G#1
	12	33	A1
	13	34	A#1
	14	35	B1
	15	36	C2
Sector 1	0	37	C#2
	1	38	D2
	2	39	D#2
	3	40	E2
	4	42	F2
	5	42	F#2
	6	43	G2
	7	44	G#2
	8	45	A2
	9	46	A#2
	10	47	B2
	11	48	C3
	12	49	C#3
	13	50	D3
	14	51	D#3
Sector 2	0	52	E3
	0	53	F3
	1	54	F#3
	2	55	G3
	3	56	G#3
	4	57	A3
	5	58	A#3
	6	59	B3
	7	60	C4
	8	61	C#4
	9	62	D4
	10	63	D#4
	11	64	E4
	12	65	F4
	13	66	F#4
	14	67	G4

Spectral Additive Synthesis Module – Technical Reference Manual

	15	68	G#4
Sector 3	0	69	A4
	1	70	A#4
	2	71	B4
	3	72	C5
	4	73	C#5
	5	74	D5
	6	75	D#5
	7	76	E5
	8	77	F5
	9	78	F#5
	10	79	G5
	11	80	G#5
	12	81	A5
	13	82	A#5
	14	83	B5
	15	84	C6
Sector 4	0	85	C#6
	1	86	D6
	2	87	D#6
	3	88	E6
	4	89	F6
	5	90	F#6
	6	91	G6
	7	92	G#6
	8	93	A6
	9	94	A#6
	10	95	B6
	11	96	C7
	12	97	C#7
	13	98	D7
	14	99	D#7
	15	100	E7
		101	F7
		102	F#7
		103	G7
		104	G#7
		105	A7
		106	A#7
		107	B7
		108	C8

3.2 Intensity Layer:

The intensity that a note is played is slit into 'Layers'. There are 3 layers.

The intensity control doesn't have to be MIDI velocity, which can be very useful for simulating non-keyboard instruments :

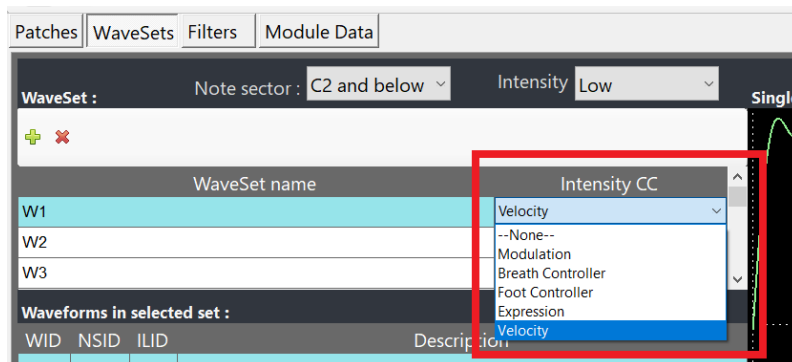


Figure 2 : Intensity control

3.3 Morphing

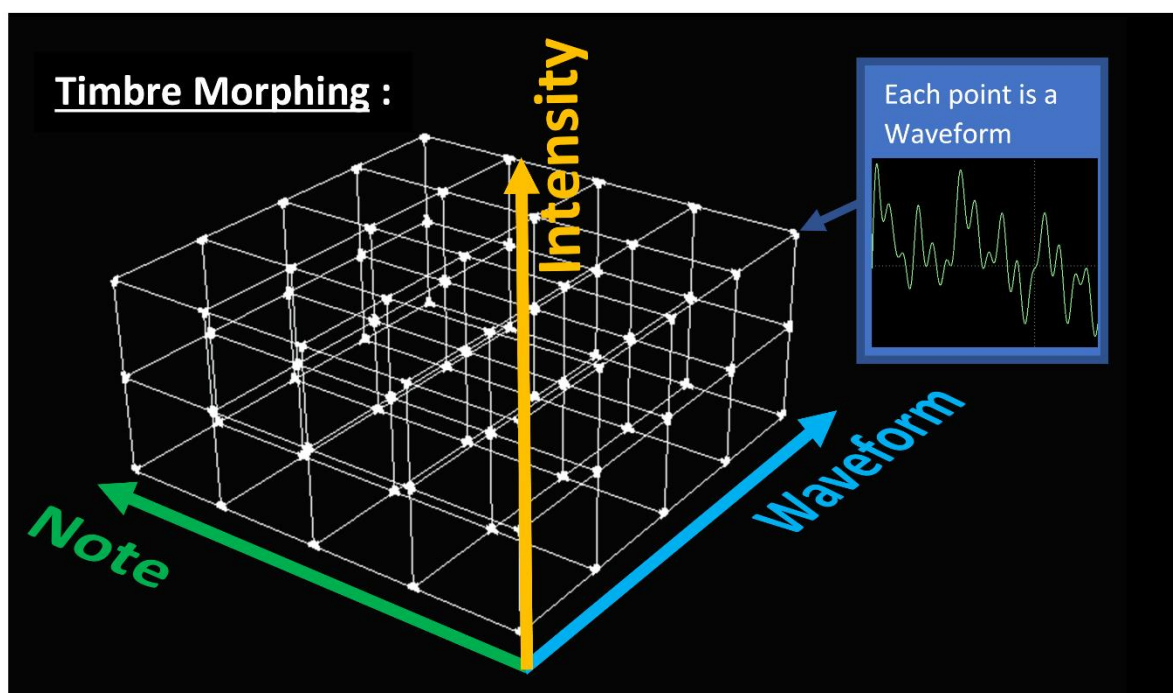


Figure 3 : WaveSet Waveforms are across 3 dimensions

In reality the waveform that is actually played is calculated by interpolating in 3 dimensions as shown in the diagram above. This 'Timbre' therefore morphs as the values of the 3 variables change. The rate of this morphing is typically 50Hz and this is adequate for human hearing to identify the change as 'smooth'.

When Timbres are refreshed in a Tone Processor WaveTable (in method ***CalculateInstanceWavetable***), the calculation involves a number of linear interpolation steps :

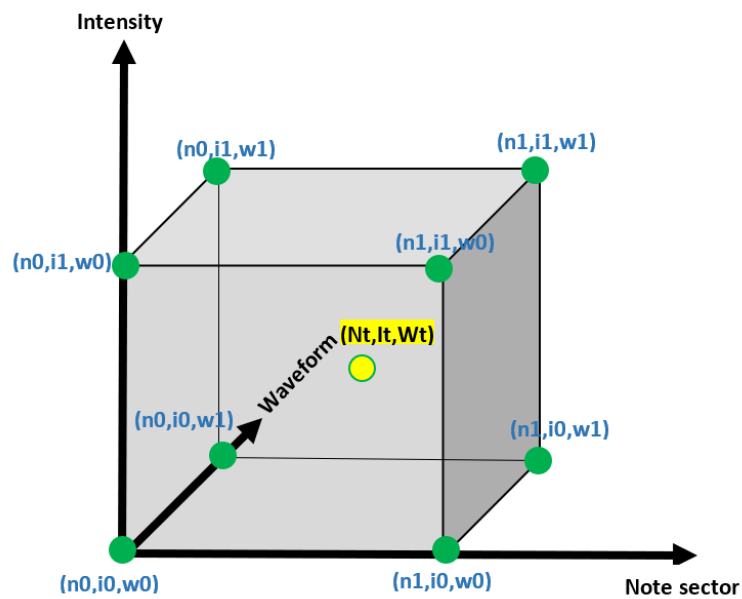


Figure 4 : Calculating the harmonic level at (Nt, It, Wt) using linear interpolation

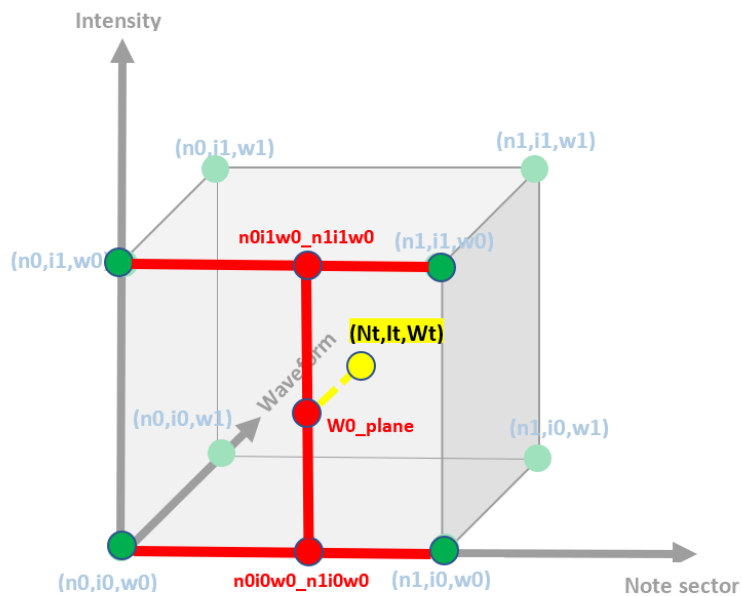


Figure 5 : Step1: Interpolate on $W0$ plane

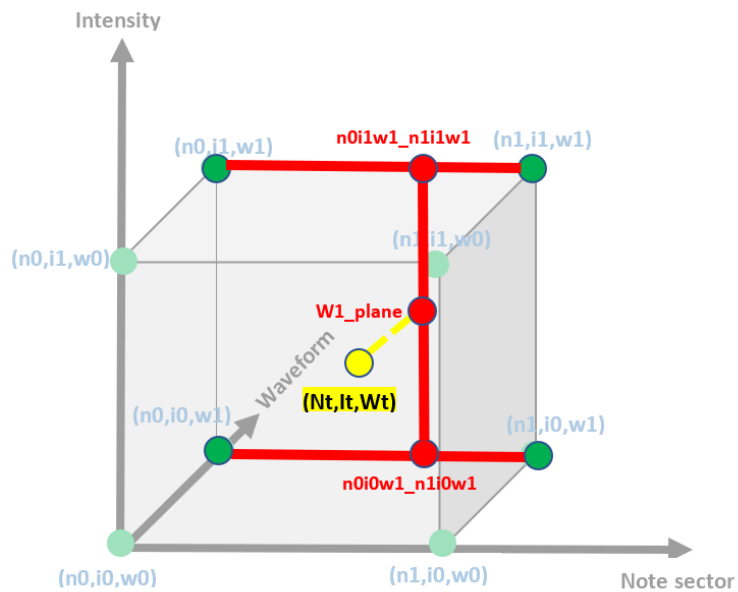


Figure 6 : Step2: Interpolate on W1 plane

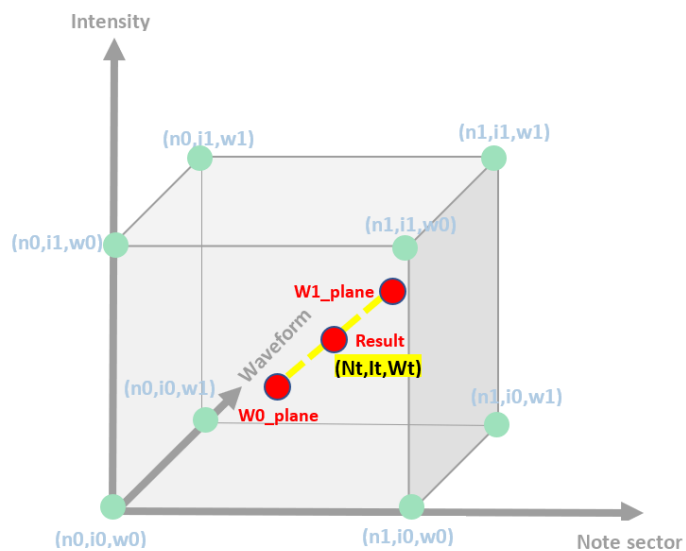


Figure 7 : Step3: Interpolate the final result

4 Wave Table

The **Spectral Sound Module** generates sound by dynamically creating the waveform of a played instrument note in a Wave Table, and then looping through this waveform to generate sound. This waveform might regularly keep updating as the sound progresses, depending on the patch.

ID	Sample (16bit Word)
0	4562
1	4582
2	356
3	22
...	...
2047	-32100

Figure 8: Wave Table

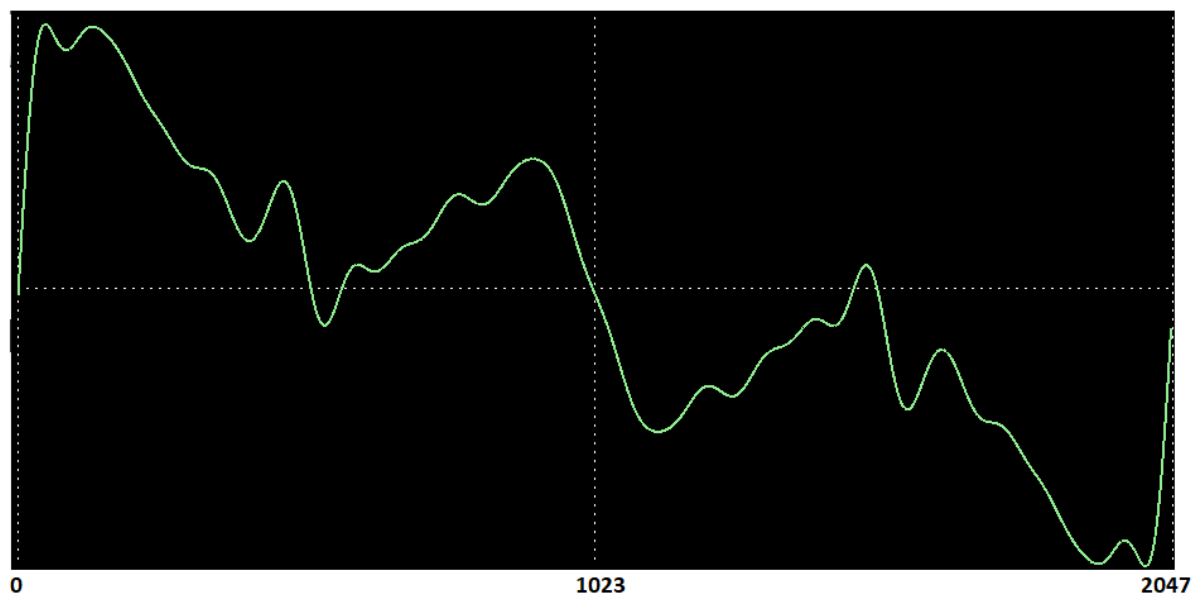


Figure 9: Single Waveform Cycle defined by Wave Table

A Wave Table is simply an array holding 2048 Word values (signed 16 bit), or ‘samples’. The Wave Table samples hold a single cycle of a waveform. A **Wave Table Index** variable points to the current sample.

Each instance of a played note has a separate Wave Table. A Tone Processor chip on the Module holds 4 Wave Tables, because it has 4 note polyphony – i.e. it can play 4 notes simultaneously.

4.1 Wave Table Increment

If the Wave Table values are sampled sequentially at the sampling rate of the module (41,666.66Hz), then the number of waveform cycles generated in a second(Hz) = $41,666.66/2048 = 20.345$ Hz. This is therefore the lowest frequency that the module is capable of generating.

Higher frequencies of the waveform are generated by incrementing the Wave Table index by more than one. The increment value is held in a special **Wave Table Increment** variable.

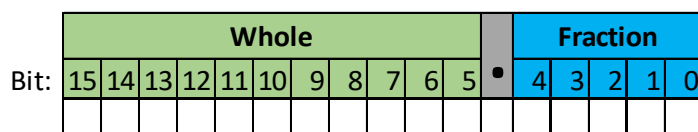


Figure 10: Wave Table Increment

This variable is a single, unsigned, 16-bit Word value where the increment value has a whole and fractional component. The whole component being the upper 11 bits and the fractional the lower 5 bits. This means that the precision of the increment value is $1/2^5$ or $1/32$.

The lowest increment has a whole component of 1, which is the lowest frequency of 20.345Hz mentioned above. The highest increment has a whole component of $2^{11}-1$ or 2047, which is a frequency of $20.345 \times 2047\text{Hz} = 41,646\text{Hz}$. However the real upper frequency used by the module is much less, since the Nyquist theorem limits the useable upper frequency to half the sampling rate, i.e $41,666.66/2 = 20,833\text{Hz}$.

Bit:	Whole											Fraction					Waveform
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Frequency
	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	20.345Hz
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	41,646Hz

Figure 11: Increment frequency range

4.2 Pitch resolution

The fractional component of the increment is used to improve pitch accuracy when generating the waveform.

In humans the threshold at which a change in pitch is just noticeable is about 5% of a semitone, which is 5 ‘cents’ in musical terms, which is a 20^{th} of a semitone. A rise of a semitone corresponds to multiplying the frequency by $2^{1/12}$ or approximately 1.0595, which is a 6% increase. So humans can discriminate an increase of a 20^{th} of this, which is approximately a 0.3% increase in frequency.

The formula for the Wave Table waveform frequency is :

$$\begin{aligned}
 \text{Frequency(Hz)} &= \left(\frac{\text{SamplingFrequency}}{\text{WaveTableSamples}} \right) * \left(\frac{\text{WaveTableIncrement}}{32} \right) \\
 &= \left(\frac{41,666.66}{2048} \right) * \left(\frac{\text{WaveTableIncrement}}{32} \right) \\
 &= \mathbf{0.63578 * WaveTableIncrement}
 \end{aligned}$$

So the finest resolution of frequency is 0.63578Hz. For very low frequencies this level of resolution is problematic, whereas for high frequencies it is more than adequate. We can calculate the frequency where a change in the increment Word by 1 will be perceptible to humans:

$$\begin{aligned}
 \frac{0.63578 + f}{f} &= 1.003 \\
 0.63578 + f &= 1.003f \\
 f &= 212 \text{ Hz (roughly G\# below middle C)}
 \end{aligned}$$

We can plot the resolution of the Wave Table Increment against frequency :

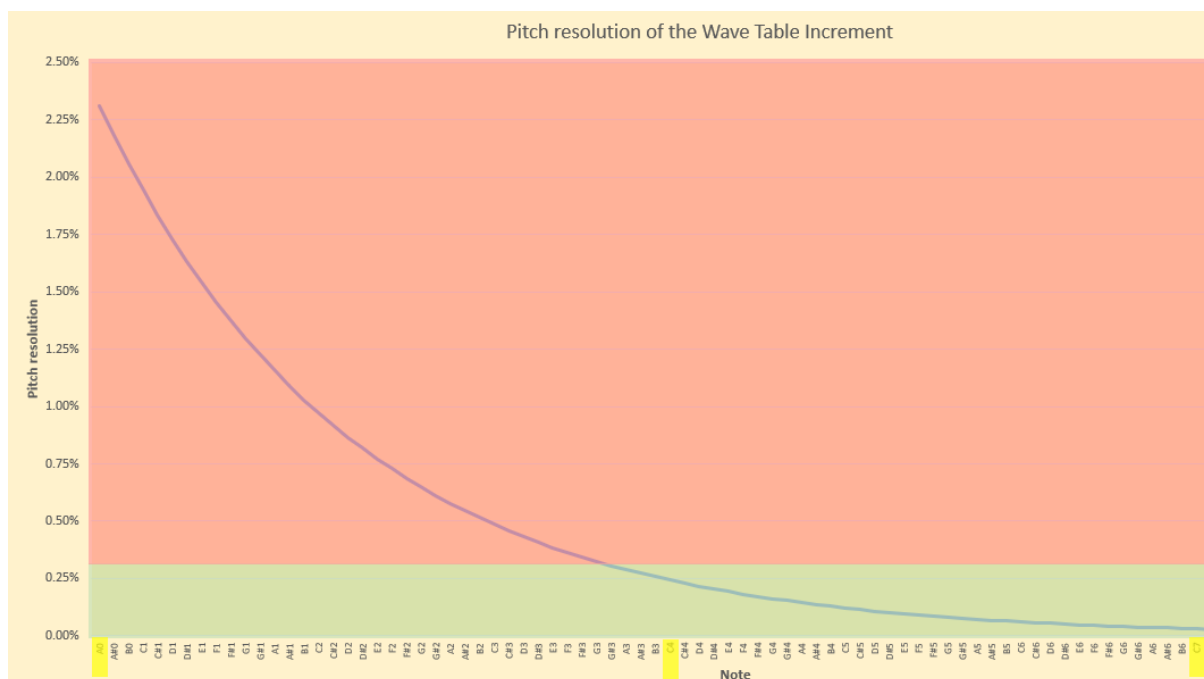


Figure 12: Pitch resolution

So anything below G#3 has a less-than-ideal resolution, whereas everything above is fine.

4.3 Increment lookup for MIDI notes

The Module uses an 88 note lookup table, **NoteSTIncLookup**, for the Wave Table Increment values. Standard full-size pianos have 88 keys.

Each MIDI note number from 21 (A0) to 108 (C8) has a corresponding (pre-calculated) increment value, representing the fundamental frequency of the note.

4.4 Populating the Wave Table

The Wave Table waveform is calculated in the Module by adding the waveforms of all the relevant harmonics. The 'Timbre' is the word used to describe the collection of harmonics used in a sound, and their levels. These harmonic levels are specified for the waveforms relating to a patch, but they are also interpolated levels, arising from multi-dimensional parameters used by the Module (note intensity, note-sector etc).

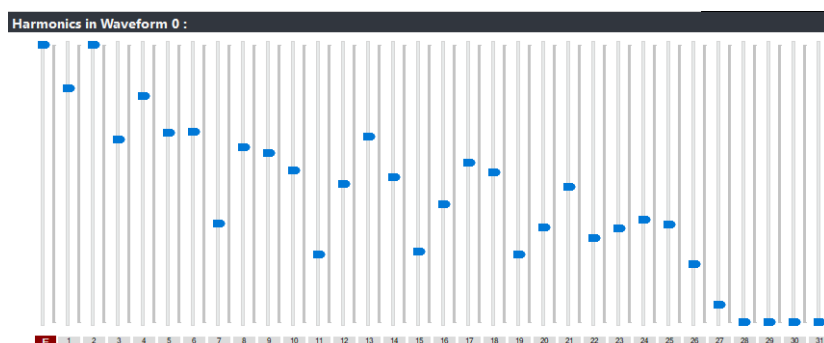


Figure 13 : Specifying harmonic levels in the app

The Spectral app, and also the internal Module mathematics, ensure that the summed waveform is 'normalised' to a range of a signed 16-bit Word (a sample in the Wave Table). Sometimes additional tweaking is necessary via a 'patch gain' parameter in the Spectral app to ensure that clipping doesn't occur.

Each harmonic is a Sine wave at a specified level. The Module does not worry about the phase of each harmonic, only the level. This is because human sound perception is very good at ignoring the phase component of sound – and has to be because of sound reflection. Phase 'can' make a difference to perceived sound, especially sounds interacting together, but largely it is fine to ignore it.

The Module populates the Wave Table samples by first populating harmonic 0 (the fundamental), by looking up the Sine values and adjusting by the harmonic level. It then looping through all remaining, relevant harmonics, adding to what's already in the Wave Table.

Sine values are obtained from lookup table ***SineLookup***. Using a lookup is the fastest method. However for harmonics above the fundamental the Sine value used is obtained via an ***interpolated_sine_lookup*** function. This function interpolates between Sine values in the SinLookup table and this interpolation comes into play for odd harmonics. It's debateable how this nuance might be noticeable in the end sounds, but it's relatively easy to achieve in code !

4.5 Skipping calculation of unheard frequencies

When the Module plays a note, the fundamental frequency of the note being played, in Hz, is also looked up, via the ***NoteHzLookup*** table. The frequencies of all the harmonics used can be simply calculated from this value because all harmonics are exact integer multiples of this fundamental frequency (note that the module does not cater for in-harmonic partials).

If the frequency of any harmonic is above the ***max_audio_freq*** then no calculation is performed.

This frequency has been set to be 15kHz, which is under the Nyquist requirement of half the sampling frequency ($\frac{41,666.66Hz}{2} = 20,833.33Hz$). The lower the max_audio_freq value, the more calculations will be dropped and the faster the waveform calculation, however this needs to be balanced against the 'often-quoted' human hearing range of 20Hz-20kHz. The upper average limit in adults is more like 15kHz and so this frequency is used !

The Module can specify 31 harmonics above the fundamental, i.e. the upper harmonic is 31 x fundamental frequency.

- Note A#4 has a frequency of 466.1Hz. 31 x 466.1Hz = 14,449Hz.
- Note B4 has a frequency of 466.1Hz. 31 x 493.9Hz = 15,311Hz.

Therefore harmonics will be dropped, and waveform calculation faster, for any notes played that are above A#4.

4.6 Use of a 'rough' WaveTable

Each Tone Processor chip has a 4 note polyphony, meaning that 4 Wave Tables can be used for up to 4 notes played simultaneously. However it wouldn't be possible to calculate the contents of a Wave Table at the same time as it was being played ! Therefore the Module uses a separate 'rough' Wave Table in which to perform calculations.

- Once calculations are complete and this Wave Table is ready to be used, then the system points the desired note instance to this table, now as a 'live' table.
- The previous Wave Table pointed to by the instance now becomes the new 'rough' table.

This method of using pointers is very fast and avoids naively physically copying data between a rough and live Wave Table.

4.7 Wave Table refresh rate

Calculating a new waveform is a background task on the Tone Processor chip and due to the complexity of foreground tasks (primarily actually looking up and modifying waveform samples for the Module's output) the time to calculate a waveform is variable. As with all calculations 'the faster the better', but from a human perception point of view it's unlikely that a human can discern a timbre changing with a period less than 10ms. The majority of timbre recalculation on the Module is within this.

5 Body resonance filter

For real musical instruments the body of the instrument greatly affects the sound produced. For example a violin body resonates at different frequencies. This is in effect a ‘master EQ filter’ applied across the generated sound.

Creating anything but the simplest filters electronically is complex. However a very useful aspect of additive synthesis is that filtering can be achieved by simply scaling the harmonics being added. So the Module has the ability to define ‘Body Resonance Filters’. The filter looks like a graphic EQ :

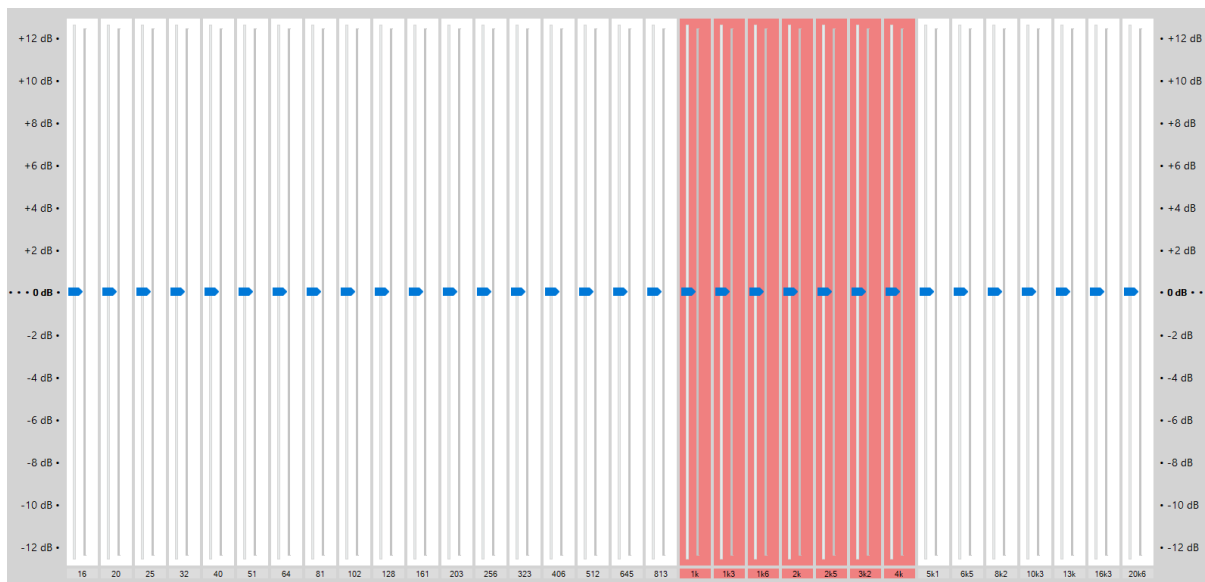


Figure 14: Body resonance filter bands

A traditional graphic EQ has frequency bands defined logarithmically, because humans have a non-linear perception of sound. Typically there is a third of an octave interval between each band. Since raising a note by an octave is doubling the frequency, then to raise a frequency by a third of an octave is achieved by multiplying the frequency by $2^{1/3}$.

The Module has 32 ‘third of an octave’ bands, covering a frequency range of 16Hz to 20.6KHz. The first band is actually irrelevant because it is below the lowest frequency that the module can generate (20.345Hz). The graphical display highlights the range 1KHz to 4kHz in red, because this is the range that human hearing is most sensitive to.

5.1 Band gain levels

The Band gain levels are held as unsigned 16-bit Word values, both in the Spectral app and also in the Module. The levels displayed in the app are logarithmic and the table below shows the mapping between the values held in the app table (‘App Table Value’) and the dB level on the app display, and also the actual band level slider value within the app :

Scaling Factor : 13606.22

A	B	C	D	E
App Table Value	A/256	Log10(B)	dB level (20Log10(B))	App slider value (C * scaling factor)
65536	256	2.408239965	48.16479931	32767
32768	128	2.10720997	42.14419939	28671
16384	64	1.806179974	36.12359948	24575
8192	32	1.505149978	30.10299957	20479
4096	16	1.204119983	24.08239965	16383
2048	8	0.903089987	18.06179974	12287
1024	4	0.602059991	12.04119983	8191
512	2	0.301029996	6.020599913	4095
256	1	0	0	0
128	0.5	-0.301029996	-6.020599913	-4096
64	0.25	-0.602059991	-12.04119983	-8192
32	0.125	-0.903089987	-18.06179974	-12288
16	0.0625	-1.204119983	-24.08239965	-16384
8	0.03125	-1.505149978	-30.10299957	-20480
4	0.015625	-1.806179974	-36.12359948	-24576
2	0.0078125	-2.10720997	-42.14419939	-28672
1	0.00390625	-2.408239965	-48.16479931	-32768

Figure 15 : Mapping of Band values to dB and to app slider values

Note that the range of cut and boost available in the app is only +/- 12dB, shown in yellow. This is a table value range of 64 to 1024 and a slider range of -8192 to 8191. This limited range is because the system only works well for relatively gentle filtering.

Once the levels are transmitted to the Model, the Module only applies the Band gain levels by attenuation. The levels set in the Spectral app table are scaled by a calculated ***filter_scaling_factor*** before being sent to the Module.

This factor is calculated by scanning all the band levels and finding the maximum level, then :

$$filter\ scaling\ factor = \frac{65535}{\max(level)}$$

‘Normalising’ all Band levels by multiplying with this factor ensures that the maximum Band level is a value of 65535.

In the Module, the filter gain for an harmonic falling in a frequency band b is therefore :

$$filter\ gain = \frac{level_b}{65536}$$

Or in code :

```
hg = (uint16_t)(__builtin_muluu(ih[0],get_filter_gain(freq_hz))>>16);
```

Note that although the filter displayed in the Spectral app shows a positive and negative dB scale, this method translates the end result into just attenuation, which has the same overall effect.

5.2 Fitting frequencies to bands

The Module must be able to determine the Band of any given frequency very efficiently. You might think that it would simply be a matter of having the Band in a lookup table of the MIDI notes played. However because the Module can apply vibrato (pitch modulation) and pitch-bend to waveforms then it needs to be able to calculate the Band 'on the fly'.

The method used involves two lookup tables, one for frequencies below 272Hz and one for frequencies above.

For the 'upper' lookup table (EqBandLookupOver271Hz) : The index to the lookup table is first calculated by dividing the frequency by 16 (shift right by 4). This table holds 1024 lookup Band values covering frequencies of 16Hz to 16384Hz (each 16Hz apart). The lower 16 entries (16Hz to 256Hz) are unused but the code is simpler to just have these unused values in place.

For the 'lower' table (EqBandLookup271HzAndLess): If we used the same approach for frequencies below 272Hz we find that this wouldn't cover bands 1,2, and 5. So instead, this lookup is for all the 256 individual frequencies between 16Hz and 271Hz.

The code to establish the Band that a given frequency occupies, is then simply :

```
if (freq_hz <= 271)
    {b= EqBandLookup271HzAndLess[freq_hz - 16];}
else
    {b= EqBandLookupOver271Hz[freq_hz>>4];}
```

5.3 Interpolation of filtering

The graphic EQ filtering needs to be smooth across frequencies, because we can't have sudden jumps in applied filtering between bands. To achieve this, the Module uses straight-line interpolation between the gain levels set by adjacent Bands. The table that holds the band gain levels also has a 'slope' value that indicates the slope of the interpolation line between one band level and the next.

Band :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Level :																																
Slope :																																

Figure 16: Body resonance bands table

The Band gain level is an unsigned 16-bit Word value. The Slope is a signed 16-bit Word value.

The slope values are calculated by the Spectral app and entered into the table, where :

$$\text{Slope value} = \frac{\text{Change in level}}{\text{Change in frequency}}$$

So if we have Band_b and Band_{b+1} then :

- The change in level is the level at Band_{b+1} minus the level at Band_b.
- The change in frequency is the frequency of Band_{b+1} minus the the frequency of Band_b.

Pre-calculating the slope values in the Spectral app just makes the Module faster.

The actual slope value has a whole and fractional component. It is a signed value with a range between $-32768/32$ to $32767/32 = -1024$ to 1023

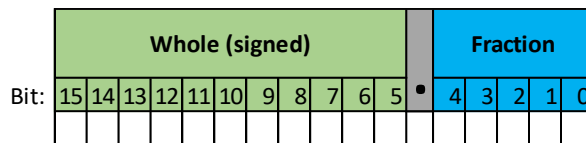


Figure 17: Slope value

The use of a fractional component is necessary in order to cope with shallow slopes.

This fairly small range is a limitation of the Module, however the expectation is that graphic EQ filtering wouldn't be too 'notchy' and the filtering would involve smoother transitions from one band to the next, and hence shallower slopes.

The Spectral app includes a 'View Module calculated filter' button that displays a graph highlighting where there is going to be overload, and hence inaccuracy of the slope values.

5.3.1 Calculation of filter gain for a given frequency

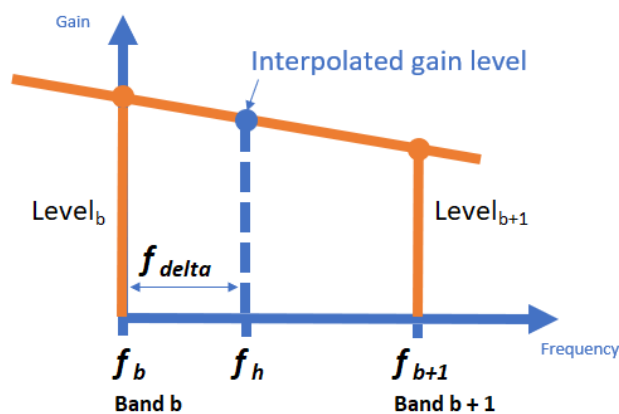


Figure 18: Interpolated filter gain

The calculation of the filter gain level, at a given frequency is relatively straightforward :

$$\text{Filter gain} = \text{Level}_b + (\text{Slope} * f_{\text{delta}})$$

Expressed in code :

```
fdelta= freq_hz - band_freq_lookup[b];
```

Note that fdelta is always a positive value.

Then we calculate the FilterGain value by interpolating :

```
uint16_t fdelta= freq_hz - band_freq_lookup[b];
```

```
if (slope > 0)
{
```

Spectral Additive Synthesis Module – Technical Reference Manual

```
    filter_gain = (uint16_t)(channel.patch.body_resonance_bands[b].level +  
(uint16_t)(__builtin_mulus(fdelta,slope)>>5));  
}  
else  
{  
    filter_gain = channel.patch.body_resonance_bands[b].level - (uint16_t)(0 -  
(int16_t)(__builtin_mulus(fdelta,slope)>>5));  
}
```

6 Noise reduction

6.1 Audio circuit considerations

The Module's audio circuitry include op-amps that must be isolated as much as possible from the digital noise generated by all the digital integrated circuits. Standard design steps have been taken :

- Separate power supply regulator for the audio circuit.
- The audio and digital power regulators are linear and not switched mode (which are inherently noisy).
- Audio circuitry kept separate from digital as much as possible with its own ground plane.

6.2 Pre-emphasis and de-emphasis

The Module uses an old audio trick of pre-emphasis and de-emphasis in order to reduce noise: The generated sounds have pre-emphasis (effectively a high-pass filter) applied, then the in-circuit audio amplifier has matching de-emphasis applied (low-pass filter). The noise that the circuit introduces (especially digital electronics) is suppressed by the low-pass filter.

6.2.1 Circuit de-emphasis

The Module electronics has two sets of differential outputs from the DAC in the 'Mixer' DSPIC chip, one for the audio Right channel and one for the Left. Each differential pair goes into a 'differential amplifier' circuit as shown below.

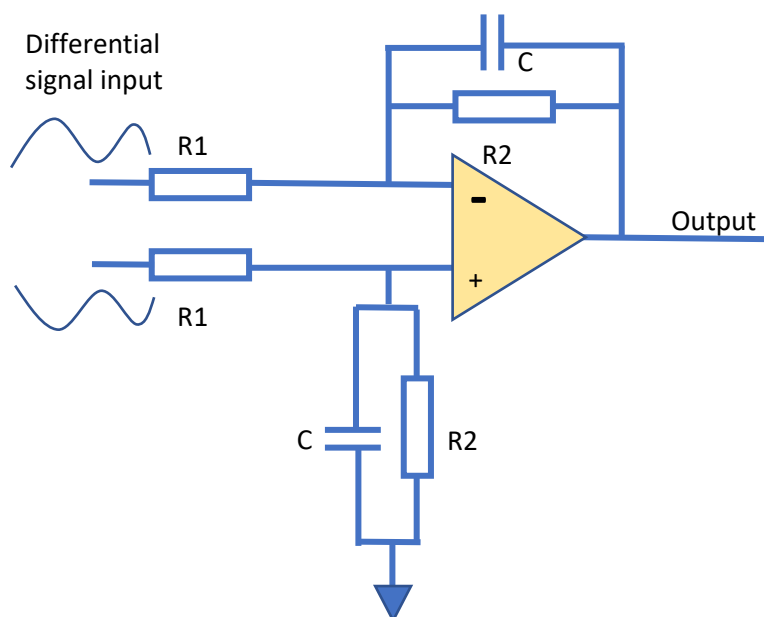


Figure 19 : Module's Differential amplifier and low-pass filter

$$\text{Gain of circuit, **without** capacitors} = \frac{R2}{R1}$$

$$\text{Impedance of a capacitor on its own (in Ohms)} = Z_c = \frac{1}{2\pi FC}$$

$$\text{Impedance of } C \text{ in parallel with a resistor } R = Z_{RC} = \frac{Z_C * R}{Z_C + R}$$

$$\text{So gain of circuit with capacitors} = \frac{Z_{R2C}}{R_1} = \left(\frac{Z_C * R_2}{Z_C + R_2} \right) \frac{1}{R_1}$$

The Module electronics have the following values, that give the frequency response shown below :

$$R_1 = 3300 \text{ Ohms}, R_2 = 4700 \text{ Ohms}, C = 0.1 \mu F$$

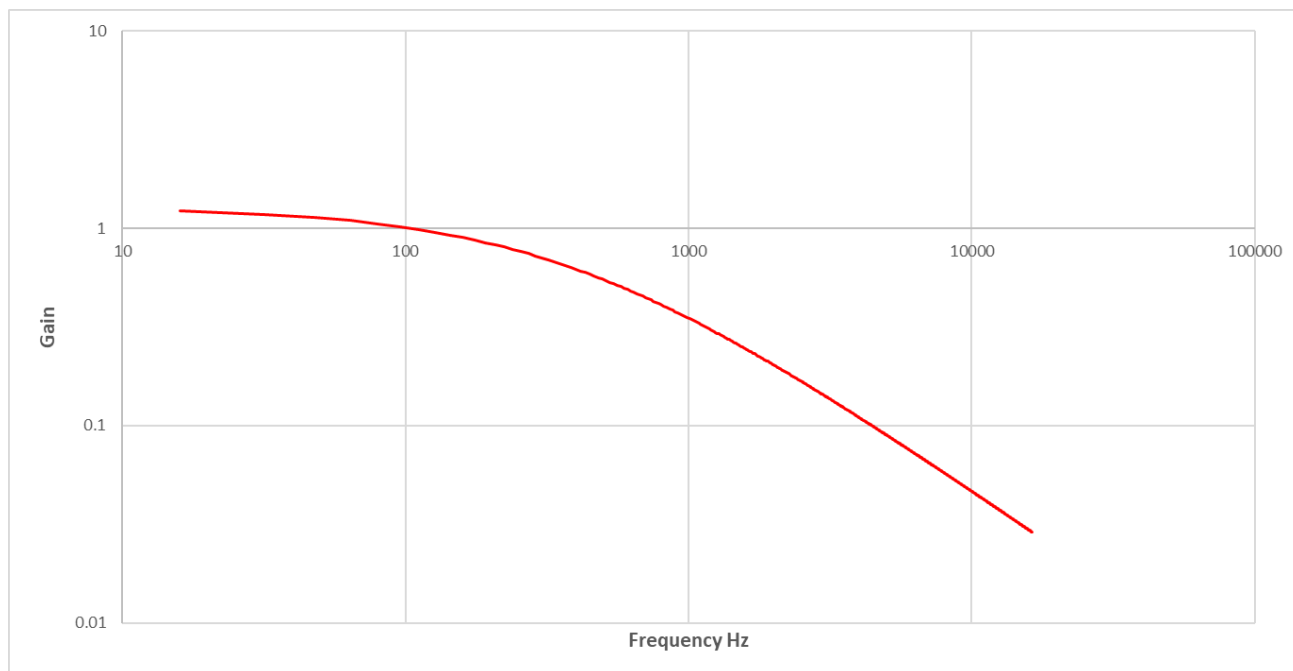


Figure 20 : Amplifier frequency response

This is the ideal response based on exact resistor and capacitor values. Metal film resistors are used with a tolerance of +/- 0.25% which is far better than standard resistor's +/-5%. However the capacitors used are ceramic multi-layer with a tolerance of +/-5%. It is difficult to get better tolerance for through-hole capacitors. Capacitors are measured for their accuracy before being used however.

This filtering suppresses the high frequency element of circuit noise, which is the whole purpose, but brings the wanted musical tones (that have had emphasis applied, boosting high frequencies) back to their wanted levels.

6.2.2 Tone processor pre-emphasis

The Tone processors add pre-emphasis, effectively boosting high frequencies, which mirrors the circuit de-emphasis. However instead of actually boosting high frequencies, low frequencies are attenuated. In a digital audio system where samples values need to be constrained within limited 'headroom', it's easier to attenuate.

The circuit frequency response has been analysed and a simple method of attenuation chosen:

- 400-element *bass_atten_lookup* table.
- The index to the lookup table is based on the frequency/16.
- It only attenuates frequencies below 6384 Hz.
- The attenuation values are 0-65535

Attenuation is a simple lookup and calculation, for frequencies below 6384Hz :

$$Gain = Gain * \left(\frac{attenuation\ value}{65536} \right)$$

This is the Module code (within the **get_filter_gain** function) :

```
if (freq_hz <= 6384)
{
filter_gain = (uint16_t)((__builtin_muluu(filter_gain,bass_atten_lookup[freq_hz>>4])) >> 16);
}
```

6.3 Automatic Gain control (AGC)

The Module uses an AGC in code, on the ‘Mixer’ DSPIC chip, to suppress peaks in the audio output, effectively acting as a ‘limiter’. This effectively raises the overall signal-to-noise ratio of the device.

This isn’t the best thing to do from a purist audio perspective, because summed sounds should always be just that, without alteration. For example, if you play two notes on the piano, the resulting summed sound won’t be massaged in volume – it will always be the exact sum.

However there are huge, very noticeable benefits in noise reduction by using a limiter and an argument can be made that digital audio often gets limited (compressed) further down stream anyway ! The only caveat is that the AGC should try not to be too noticeably ‘aggressive’.

In the Module that AGC’s mission is to suppress peaks and scale the output down when necessary so that the digital output value NEVER overflows. This is achieved via a 256 sample lookahead buffer and applying a linear ramped gain envelope to ensure this criteria is always met.

A buffer of 256 samples, at our sampling rate of 41,666.66Hz contributes a latency of 6.14ms into the system, but it’s still very worthwhile.

The operation of the AGC is quite complex but the code to implement it is quite efficient.

6.3.1 AGC Ring buffer and windows

The AGC uses a ring buffer of 256 samples, split into two equal ‘Windows’ :

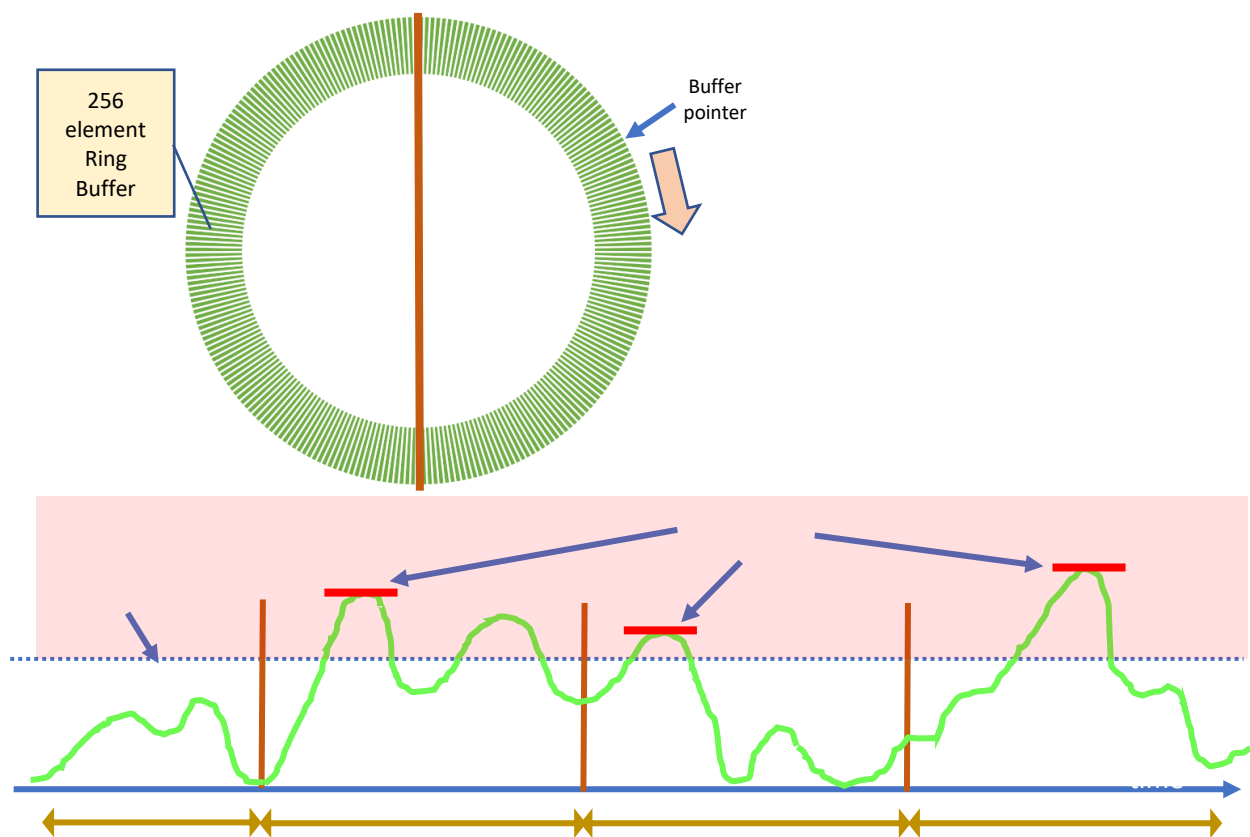


Figure 21 : AGC Ring Buffer and Windows

The buffer index is sequentially incremented at the sampling rate and for each increment of the index a '256 old' sample is read from the buffer index and the new sample is written to the buffer index.

During the writing of a windows-worth of data, the peak of the signal is monitored, so that by the end of the window we know what the peak was for that window.

We then use this peak to calculate a gain value, such that when this window of data is eventually output, this gain value will ensure that 'Output x GainValue' does not exceed a given maximum threshold (it does not exceed an int16 Word value). We know that we have 128 samples of data still to read before we need to apply this gain value (the output is 256 samples behind the input), so we also work out the gain envelope slope, so that the output gain can smoothly transition to this gain value.

So we effectively 'lookahead' and use the peak information to ensure that we adjust the output gain just in time to avoid any overload :

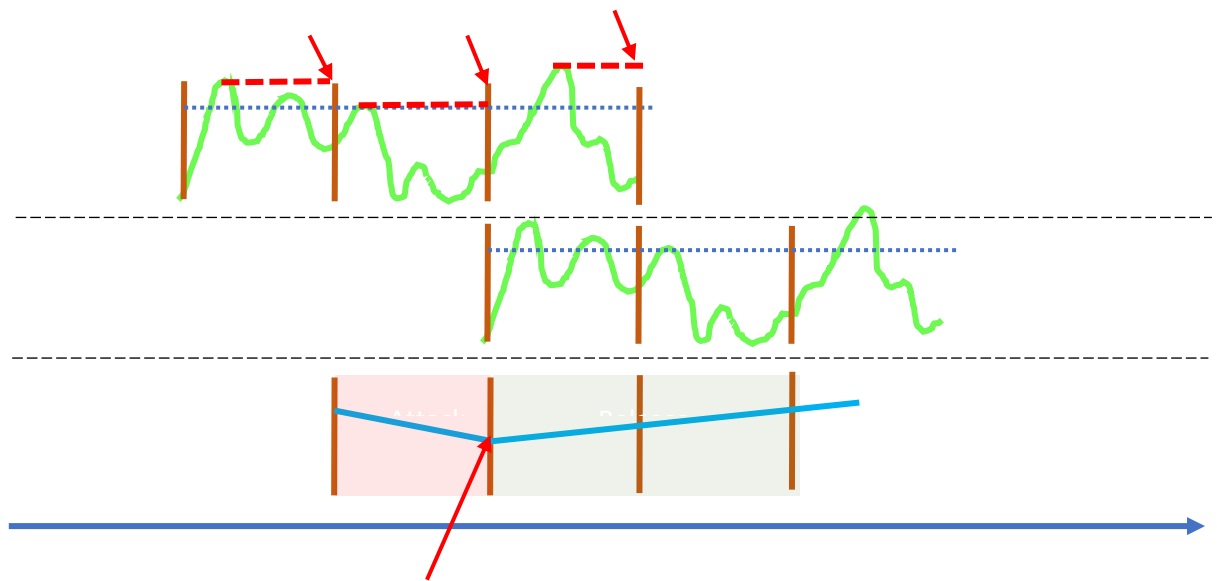


Figure 22 : Lookahead over time

If a peak is on the horizon then the gain slope is negative and this is termed ‘attack’. The slope of this gain envelope is as steep as it has to be to guarantee that there won’t be overload. We ideally don’t want this attack to be too aggressive, which is audibly noticeable, and the more lookahead the better in this respect. However we also don’t want to add much latency to the system, because this is a live musical instrument. So an AGC ring buffer size of 256 samples is a compromise.

If the signal level drops at the input then the AGC can ‘release’ and the gain level can return to unity gain (i.e. no attenuation). However we don’t want to release immediately because :

- A quick release would be audible
- A ‘release’ envelope gain slope ‘could’ result in system overload if the slope is too steep.
-

6.3.2 AGC output and HAAS delay

The AGC buffer output is multiplied by the gain envelope, and as mentioned, this envelope ensures that the output is not overloaded. This output is also scaled by a global Module gain value, which relates to the Module’s volume potentiometer.

The output is then made into ‘pseudo-stereo’ using the ‘HAAS effect’. This is simply adding a sub-40ms delay to the audio fed to one ear, which makes the audio appear to be in stereo, even though it’s come from a mono source.

The Haas Effect, also sometimes called the precedence effect, is a psychoacoustic phenomenon that causes a listener to perceive a space and direction of a sound when there is a slight delay between stereo channels. The listener perceives that the sound takes place in the direction of the first, or preceding, channel—even if the delay between the two channels is only a few milliseconds.

The Module allows the user to specify the HAAS delay up to the buffer size of 1100 samples, which is a delay of $1100/41,666.66$ seconds = 26.4ms (limited to 25ms in the app).

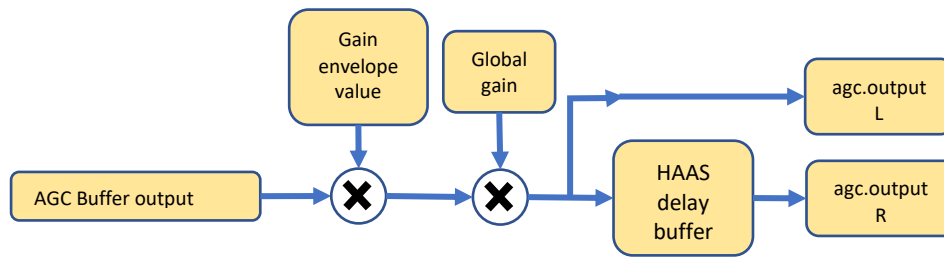


Figure 23 : Output from the AGC

6.3.3 Calculating the required envelope ‘target’ gain

The threshold is set to 32700. Any signal (int16 Word value) over that will be suppressed by an envelope ‘target’ gain value that is looked up from a carefully constructed lookup table called **AGCNumeratorLookup**. This table is indexed as follows:

Target = AGCNumeratorLookup[(uint16_t)((uint32_t)(agc.window_peak - threshold)>>11)]

This is the target gain that the current gain needs to change to, by linearly changing over time.

7 Tone Loudness and Intensity

The MIDI velocity received by the Module has to be translated into appropriate 'loudness', in human terms. The loudness curve, relating MIDI note velocity to signal amplitude, is traditionally more a square-law relationship than logarithmic (there is research online, looking at a number of synth responses). So this Module follows suit.

A Tone Processor in the Module translates the MIDI note velocity to a 16-bit, **velocity16** value by use of a lookup table called **Velocity16Lookup**:

```
uint16_t velocity16 = Velocity16Lookup[inon.velocity_id];
```

$$V = ((V_{midi})^2 * 0.937008) / 127 + 8$$

$$velocity16 = V * 512 + 511$$

V is between 8 and 127. Velocity16 is between 4607 and 65535. The lookup table holds 128 values of Velocity16 that map the MIDI note velocities, 0 to 127.

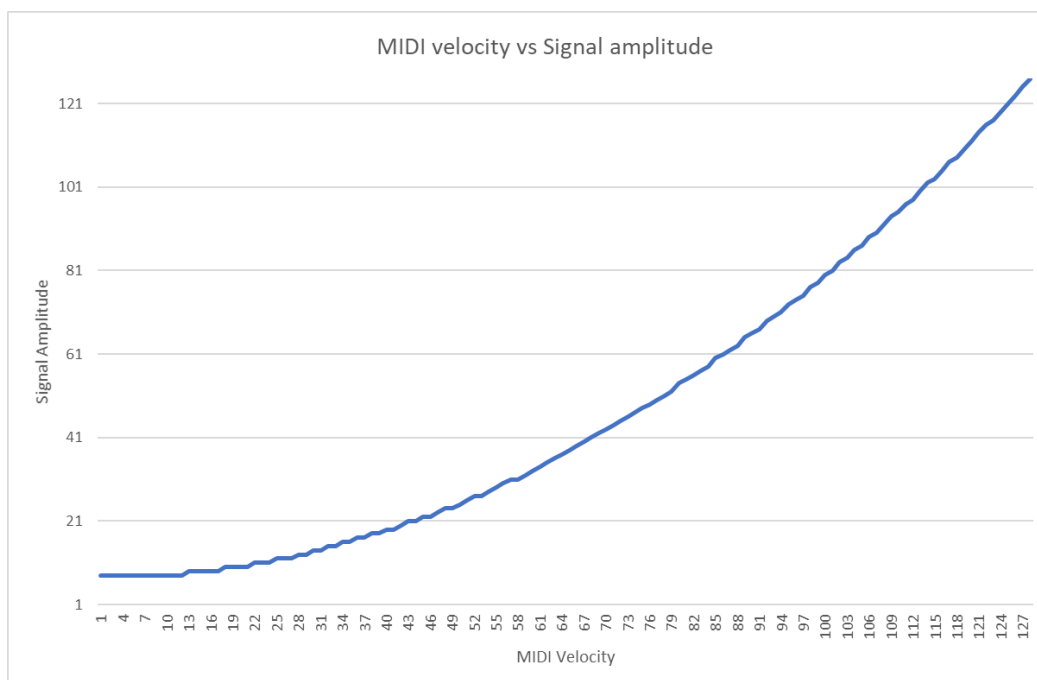


Figure 24 : Mapping loudness (linear scale)

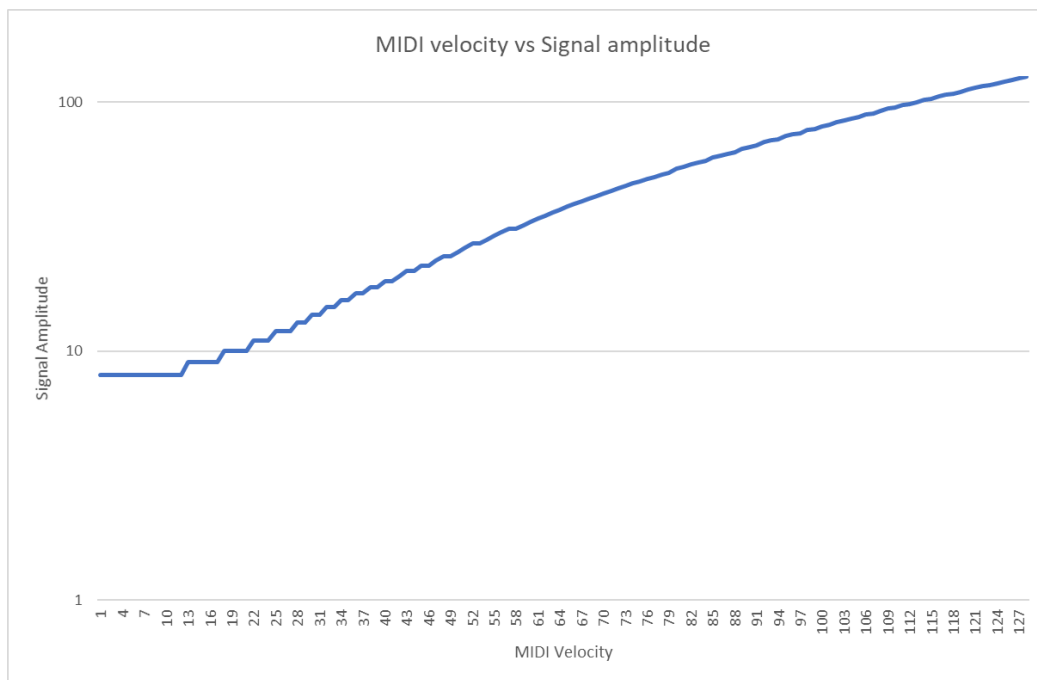


Figure 25 : Mapping loudness (log scale)

The curve on a logarithmic scale looks approximately linear, which is what we want, because humans perceive loudness approximately logarithmically.

Note that the volume of played notes is also affected by the MIDI channel volume, that's specified on the Module as part of the Performance configuration. The raw MIDI notes velocity (e.data2 below) is scaled by the channel volume (0 to 255) as necessary :

```
if (performance_id != performance_id_not_set)
{
    m.velocity_id = (uint8_t)((e.data2 *
performance[performance_id].pc[midi_channel].volume)>>8);
}
else
{
    m.velocity_id = e.data2;
}
```

8 Low Frequency Oscillators (LFOs)

8.1 Tables

Table : lfo_envelope_config

Low Frequency Oscillators (LFOs) :

LFO	Enabled	Wave type	Freq Hz	Freq CC	Depth CC
Tremolo	<input type="checkbox"/>	Sine	2.0	--None--	--None--
Vibrato	<input type="checkbox"/>	Sine	0.0	--None--	--None--
Timbre lfo	<input checked="" type="checkbox"/>	Sine	2.5	--None--	--None--

Envelope gain controllers :

Envelope	Initial Level%	Gain Controller
Amplitude	0	Velocity
Noise gain	0	--None--
Noise cut-off freq	100	--None--
Timbre morph	0	--None--
Sample gain	0	Velocity
Pitch shift	0	--None--
Tremolo depth	0	--None--
Vibrato depth	0	--None--
Timbre lfo depth	100	--None--

Table : env_gain_CC

8.2 MIDI Controllers and LFO Depth Envelopes

The LFOs have MIDI Controllers that affect :

- Frequency of the LFO
- Depth of the LFO signal

Each LFO has an associated depth envelope, and since all envelopes have associated 'gain' controllers, these do too. However these gain controllers make no sense here !

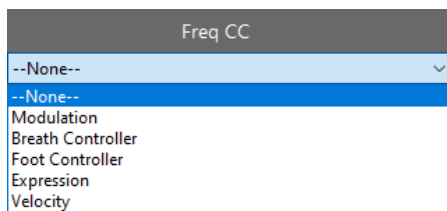
8.3 Tremolo LFO

8.3.1 Tremolo frequency

The frequency of the Tremolo LFO is set by the user in the app. However this frequency can also be changed by an associated MIDI controller. A Tone Processor updates the frequency every 10ms in *process_10ms_event*.

```
if (channel.patch.lfo_envelope_config[lfo_tremolo].enabled == 1)
{
    ucc_value = GetLFOFreqEnvCCValue16(i,lfo_tremolo);
    instance[i].lfo_env_current[lfo_tremolo].current_wt_inc_q11_5 =
    (uint16_t) (__builtin_muluu(channel.patch.lfo_envelope_config[lfo_tremolo].default_wt_inc_q11_5,ucc_value)>>16);
}
```

The user specified the MIDI Continuous Controller relating to the LFO frequency :



GetLFOFreqEnvCCValue16 returns :

- 65535 if the controller is 'None'
- The Velocity16 value (unsigned 16 bit) if the controller is set to Velocity. Note this is fixed at the start of a note being played, since it's derived from the MIDI velocity.
- Otherwise the current controller value (unsigned 16 bit)

ucc_value is the current value of this 'used' continuous controller value.

default_wt_inc_q11_5 is the default tremolo frequency, set by the user in the app. This value is actually the Wave Table Increment value corresponding to frequency.

The new LFO frequency, **current_wt_inc_q11_5**, is calculated by scaling this value by **ucc_value**. i.e :

$$current = default \times \left(\frac{UCCvalue}{65536} \right)$$

8.3.2 Tremolo LFO gain

The Tremolo LFO oscillates at the calculated frequency. The magnitude of the LFO signal at any given time is controlled by a number of factors :

- LFO envelope
- LFO oscillator 'signal' (waveform and frequency)
- Key scaling
- Current MIDI controller value affecting LFO depth

The LFO signal is termed a 'gain' signal because it is applied to the tone being produced by the Tone Processor.

The **lfo_env_current[lfo_tremolo].gain** value is the 'current', instantaneous amplitude of the LFO at a moment in time, based on these factors. It is a signed 16-bit value that oscillates about zero with an amplitude that depends on the envelope. It is updated every 1ms in the method called **process_1ms_event**, which executes the following code :

```
StepLFOEnvelope(&channel.patch.lfo_envelope_config[lfo_tremolo], &channel.patch.adsr_section[a].adsr_section_envelope_config[env_tremolo], &instance[i].lfo_env_current[lfo_tremolo], a, i);
```

```
IEC0bits.T2IE = 0;
```

(See Stage 1 below)

```
instance[i].trem_offset_gain =
(uint16_t)((int32_t)instance[i].lfo_env_current[lfo_tremolo].gain +
((int32_t)(65535) -
((int32_t)instance[i].lfo_env_current[lfo_tremolo].depth_env_value>>1)));
```

(See Stage 2 below)

```
instance[i].trem_offset_gain =
(uint16_t) (__builtin_muluu(instance[i].trem_offset_gain, GetEnvCCValue16(i, e
nv_tremolo)) >> 16);

(See Stage 3 below)
instance[i].trem_offset_gain =
(uint16_t) (__builtin_muluu(instance[i].trem_offset_gain, channel.volume) >> 16
);

IEC0bits.T2IE = 1;
```

StepLFOEnvelope is a generic method that updates the *lfo_env_current[lfo_tremolo].gain* value. It also updates the current LFO envelope value, called **depth_env_value** (an unsigned 16-bit value between 0 and 65535). The gain value is scaled by the MIDI controller affecting LFO depth (if any).

8.3.3 Tremolo LFO offset gain

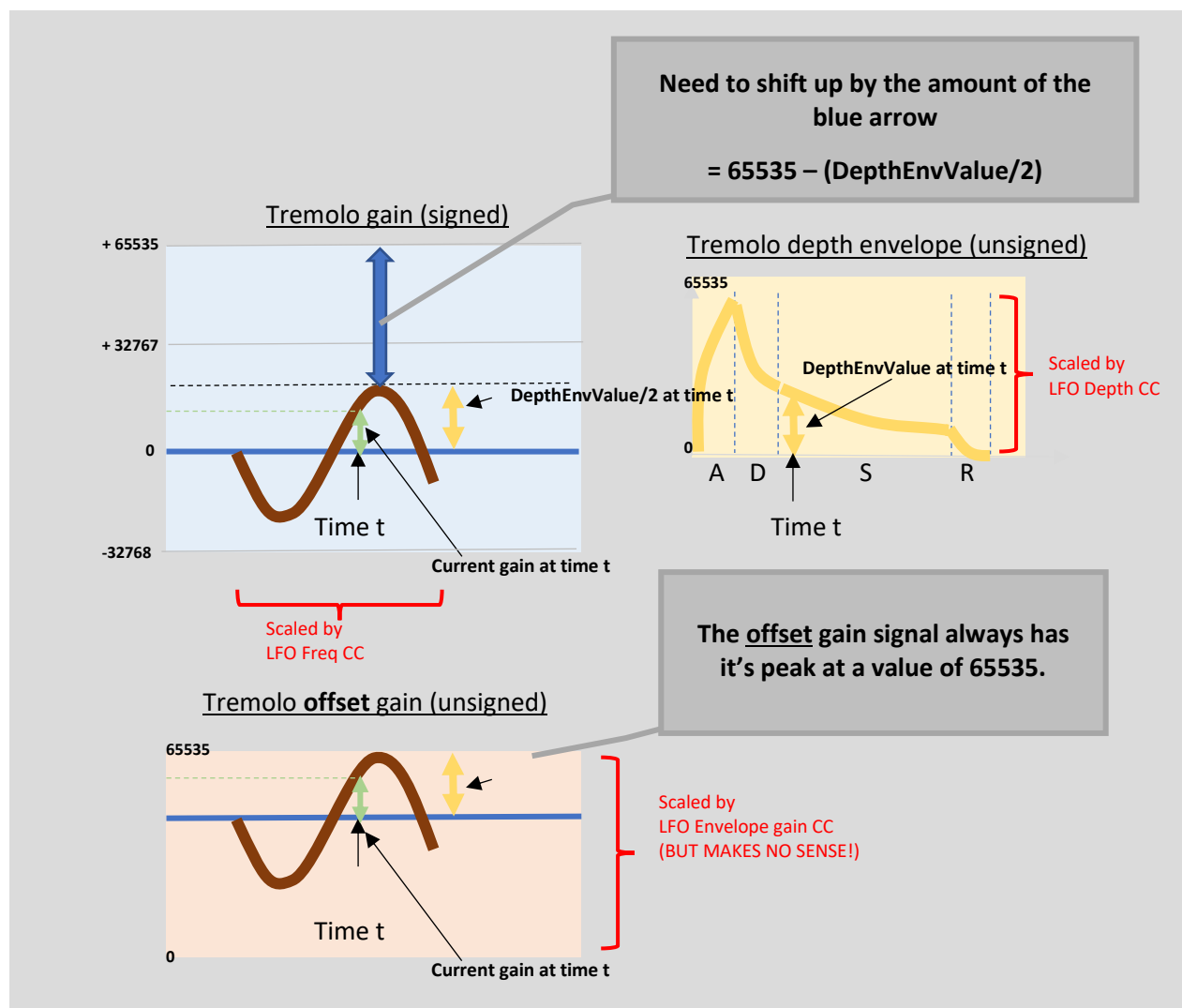
For the gain value to be useable, it needs to be offset to lie in a range of 0 to 65535.

trem_offset_gain is the adjusted, unsigned 16-bit, gain value, and is calculated in 3 stages :

Stage 1 : Simple level shift

$$TremoloOffsetGain = CurrentTremoloGain + \left(65535 - \left(\frac{DepthEnvValue}{2} \right) \right)$$

This translates the gain value from a signed to unsigned value, plus level shifts. 65535 - (DepthEnvValue/2) is the amount to shift the level :



Stage 2 : Scale by Tremolo Depth Gain Controller value

$$TremoloOffsetGain = TremoloOffsetGain * \left(\frac{EnvCCValue16}{65536} \right)$$

GetEnvCCValue16 is a function that returns an unsigned 16-bit integer, for the 'gain controller' value (in the *env_gain_cc* table) relating to the Tremolo depth envelope :

```
uint16_t GetEnvCCValue16(uint16_t i, uint16_t envelope_id)
{
    uint16_t ucc_id = channel.patch.env_gain_CC[envelope_id];
    if(ucc_id == ucc_none)
        {return 65535;}
    else if(ucc_id == ucc_velocity)
        {return instance[i].velocity16;}
    else
        {return channel.ucc_current_values[ucc_id];}
}
```


NOTE: This controller makes no sense for Tremolo !

Stage 3 : Scale by the channel volume

This last stage is not really related to tremolo, but is an intermediate step in the final calculation of the **overall_gain** value for the Tone Processor instance being played . It scales by the MIDI channel volume :

$$TremoloOffsetGain = TremoloOffsetGain * \left(\frac{ChannelVolume}{65536} \right)$$

This final **trem_offset_gain** value is used to scale the output signal in the core Tone Processor sample generation code (a method called **UpdateSampleValue**) :

```
instance[i].overall_gain =
__builtin_muluu(instance[i].trem_offset_gain,amplitude_depth_env_value)>>16
;
```

8.4 Timbre LFO

8.4.1 Timbre envelopes

Unlike Tremolo and Vibrato, Timbre has two envelopes :

```
enum envelope
{
    env_amplitude = 0,
    env_noise = 1,
    env_noise_cutoff_level = 2,
    env_timbre = 3,
    env_sample = 4,
    env_pitch = 5,
    env_portamento = 6,
    env_tremolo = 7,
    env_vibrato = 8,
    env_timbre_lfo = 9
};
```

- **Env_timbre** is displayed as ‘Timbre Morph’ and is used to Morph between waveforms in a Waveform Block.
- **Env_timbre_lfo** is displayed a ‘Timbre lfo depth’ and as it’s name implies is used to vary the Timbre LFO depth.

8.4.2 Timbre LFO frequency

Just as for Tremolo, the frequency of the Timbre LFO is set by the user in the app and this frequency can also be changed by an associated MIDI controller. A Tone Processor updates this frequency every 10ms in **process_10ms_event**.

```
ucc_value = GetLFOFreqEnvCCValue16(i,lfo_timbre);

instance[i].lfo_env_current[lfo_timbre].current_wt_inc_q11_5 =
(uint16_t)(__builtin_muluu(channel.patch.lfo_envelope_config[lfo_timbre].de
fault_wt_inc_q11_5,ucc_value)>>16);
```

8.4.3 Timbre LFO gain

Again, just like Tremolo, the magnitude of the Timbre LFO signal at any given time is controlled by a number of factors :

- LFO envelope
- LFO oscillator 'signal' (waveform and frequency)
- Key scaling
- Current MIDI controller value affecting LFO depth

For the Timbre LFO, the gain and ***lfo_timbre_offset_gain*** values are calculated every 10ms (not 1ms) in the ***process_10ms_event*** method. The method of calculation is the same as for Tremolo, except there is only a single calculation stage for the offset gain :

```
StepLFOEnvelope(&channel.patch.lfo_envelope_config[lfo_timbre], &channel.patch.adsr_section[a].adsr_section_envelope_config[env_timbre_lfo], &instance[i].lfo_env_current[lfo_timbre], a, i);

IEC0bits.T2IE = 0;

instance[i].lfo_timbre_offset_gain =
(uint16_t)((int32_t)instance[i].lfo_env_current[lfo_timbre].gain +
((int32_t)(65535) -
((int32_t)instance[i].lfo_env_current[lfo_timbre].depth_env_value>>1)));

IEC0bits.T2IE = 1;
```

8.4.4 Timbre LFO offset gain

This final ***timbre_offset_gain*** value is used when the Wave Table is next recalculated, in the ***CalculateInstanceWavetable*** method :

```
uint16_t timbre_gain =
__builtin_muluu(instance[instance_id].lfo_timbre_offset_gain, instance[instance_id].env_current[env_timbre].depth_env_value_scaled_by_CC)>>16;

uint16_t w0 = timbre_gain >>14;
uint16_t w1 = w0 + 1;
uint16_t w = (uint16_t)((timbre_gain - (w0 << 14))>>6); //Range 0 to 255

if(w > 128){w++;}
uint16_t vw = 256 - w;
```

The ***timbre_offset_gain*** value is first scaled by the 'Timbre morph' envelope depth value(***depth_env_value_scaled_by_CC***), which has been scaled by the 'Gain controller' associated with the Timbre Morph envelope (***env_timbre***).

The waveforms to interpolate across (w0 and w1), are then calculated from this ***timbre_offset_gain*** value.

9 Performances and PatchSets

The Module can store a number of instrument sounds, or 'patches' internally. It can also store configurations of 'performances' where multiple patches can be played in layers or splits across the keyboard.

The way the performances are configured takes a bit of understanding, because of the physical limitations of the device. However the method used gives maximum flexibility which can be very useful.

Although there can be many patches held in a Spectral App file, the module itself can only store a maximum of 18 patches. So there is a concept of 'PatchSets' that are collections of up to 18 patches. The module is loaded with a PatchSet.

The patches on the module can further be used in 'Performances'. There can be a maximum of 8 performances saved to the module. A performance specifies which patches are used on MIDI channels 1 to 6, plus which channel each 'Tone Processor' chip on the module is assigned to :

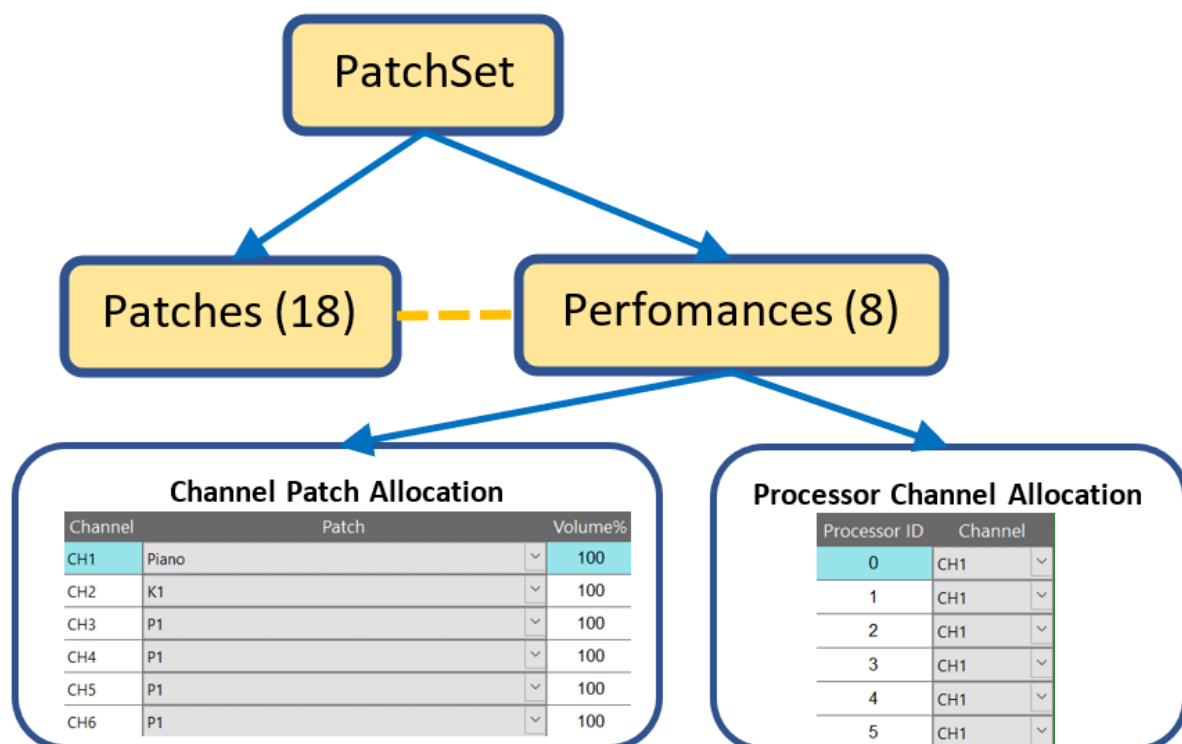



Figure 26: PatchSets and Performances

There are 6 Tone Processor chips. Each chip can only be assigned to one MIDI channel. Therefore there is a maximum of 6 MIDI channels that can be used.

An important point is that multiple Tone Processors can be assigned to the same MIDI channel. This means that you can distribute the processing power as you see fit. As an

example, suppose you wanted to play a simple bass accompaniment in the left hand but piano chords in the right hand. The processing requirements of the bass are much less and so would need fewer processors.

Patches and Performances are changed through MIDI **Bank and Program Change messages**.

Data in Module Memory (when saved) :  Save

Patches Performances

Patches in the selected PatchSet :

PSPID	Patch
0	K1
1	Piano
2	Xylophone1
3	P1
4	P1
5	P1
6	P1
7	P1

Figure 27: Patches in a PatchSet

Patches Performances

Performances in the selected PatchSet :

ID	Performance name	HAAS delay ms
0	Perf1	10
1	Perf2	25
2	<None>	25
3	<None>	25
4	<None>	25
5	<None>	25
6	<None>	25
7	<None>	25

Figure 28: Performances in a PatchSet

Each of 6 MIDI channels can be assigned a Patch, plus have an associated volume :

Channel patch allocation		Processor channel allocation	
Channel	Patch		Volume%
CH1	Piano	▼	100
CH2	Xylophone1	▼	100
CH3	P1	▼	100
CH4	P1	▼	100
CH5	P1	▼	100
CH6	P1	▼	100

Figure 29: Channel patch allocation

Each of the 6 'Tone Processors' on the module can be assigned to one of the 6 MIDI channels. Multiple processors can be assigned to the same MIDI channel :

Set :	Channel patch allocation		Processor channel allocation	
HAAS delay ms	Processor ID	Channel		
	0	CH1	▼	
10	1	CH1	▼	
25	2	CH1	▼	
25	3	CH2	▼	
25	4	CH2	▼	
25	5	CH2	▼	
25				
25				

Figure 30: Processor Channel allocation

When notes are played, the module receives the MIDI channel of the note and looks across all the Tone Processors that are associated with the channel, in order to work out which Processor should handle this 'Note Instance'. If all instances are used up then 'note-stealing' takes place, where the note in that channel that was played the earliest is dropped to make way for the new note.

Since different channels can be used, the module does this calculation on each channel separately. Therefore some channels can be witnessing note-stealing and others not.

Since there are 6 Tone Processors, then there can be a maximum of 6 different patches being played, across six MIDI channels. If this was the case then since each Tone Processor can handle a maximum of 3 Note Instances, then the system would be 3-note-polyphonic on each channel !

So although quite complex, this system is very flexible.