

## Bytebeat per il principiante assoluto!

V1.5, 4 Gennaio 2020

scritto da:

The Tuesday Night Machines

[hello@nightmachines.tv](mailto:hello@nightmachines.tv)

<http://nightmachines.tv/youtube>

tradotto da:

[pangrus](#)  
<https://www.peal.space/>

Buongiorno!

Grazie per aver scaricato la mia guida alla programmazione di bytebeat.

Quando ho scoperto il bytebeat, sono rimasto meravigliato di come pochi caratteri di testo potessero generare una così grande varietà di suoni.

Non sono un buon programmatore o un matematico, quindi all'inizio credetti che non sarei mai stato in grado di programmare un bytebeat senza ricorrere interamente alla semplice pratica dei tentativi e errori.

Naturalmente scrivere codice casuale era comunque molto divertente, ma dopo poco tempo, ho davvero sentito il bisogno di capire perchè le cose funzionassero in un certo modo, per avere la possibilità di aggiungere deterministicamente degli elementi ai miei bytebeat. Sfortunatamente, informazioni specifiche per principianti erano sorprendentemente difficili da trovare online, quindi dopo un ripasso di matematica e programmazione ho finalmente avuto qualche epifania che si è trasformata in una serie di annotazioni e poi in questo documento.

Questa guida è per i musicisti elettronici senza, o quasi, esperienza di programmazione, che si divertano a sperimentare con dei suoni inconsueti.

Basta sapere qualcosa riguardo alle forme d'onda come la dente di sega, l'onda quadra e l'onda triangolare, le frequenze e le ampiezze in campo musicale e qualche concetto base di aritmetica come addizione, sottrazione, moltiplicazione e divisione.

Cercherò di spiegare il resto in modo semplice, intervallando la spiegazione a divertenti esercizi.

Fatemi sapere se questa guida vi è stata utile.

Happycoding :-)

Felix / The Tuesday Night Machines

## ***Indice***

<b>Cosa è un bytebeat?</b>	<b>4</b>
Esercizio 01: Sperimentazione	
<b>Come si programma un bytebeat?</b>	<b>5</b>
<b>Cosa fa sì che un'espressione sia anche un bytebeat?</b>	<b>6</b>
Esercizio 02: Operatori	
<b>Capire il tuo primo bytebeat</b>	<b>7</b>
<b>Programmare con operatori matematici</b>	<b>10</b>
Esercizio 03: Moltiplicazioni e divisioni	
Esercizio 04: Operatore Modulo %	
Esercizio 05: Addizione e sottrazione	
<b>Programmare con gli operatori bit per bit</b>	<b>13</b>
Esercizio 06: AND &	
Esercizio 07: OR	
Esercizio 08: AND &, OR  , X O R ^	
Esercizio 09: Bitshifts << >>	
<b>Programmare con gli operatori relazionali</b>	<b>18</b>
Esercizio 10: Operatori relazionali	
<b>Mettiamo tutto insieme</b>	<b>19</b>
Programmare uno step sequencer	
Programmare un'onda quadra con PWM	
Programmare un'onda triangolare con wave folding	
<b>Rompere le catene</b>	<b>24</b>
<b>Fonti e ulteriori letture</b>	<b>25</b>

## Cosa è un bytebeat?

I bytebeat sono creazioni musicali generate da poche linee di codice, tipicamente originate da un flusso audio di byte (8bit unsigned) a una frequenza di 8kHz. Questo significa che un'espressione matematica viene processata dal computer 8000 volte al secondo, generando una forma d'onda udibile con una risoluzione di 256 valori dal silenzio (0) alla massima ampiezza (255).

per esempio:

```
(t*4|t|t>>3&t+t/4&t*12|t*8>>10|t/20&t+140)&t>>4
```

(premere [qui](#) per ascoltarlo nel browser)

Oh! Sembra super astratto e complicato! Come funziona?

Oonestamente, quando l'ho scritto, non avevo idea di come funzionasse. L'esempio qui sopra è il risultato di tentativi e errori e io credo che molta gente programmi i propri bytebeat solo in questo modo, che è comunque molto divertente, veloce e può condurre a molti incidenti felici. Un ottimo modo per generare nuovi suoni per il proprio fabbisogno giornaliero!

Proviamoci insieme ora!

### Esercizio 01: Sperimentazione

Apri il Greggman's HTML 5 bytebeat Player usando questo [link](#)

La variabile **t** genera un'onda a dente di sega. Lasciala dov'è e copia qualcuna delle linee seguenti subito dopo. Ascolta il risultato.

```
|t*4  
|t%128*10  
|t*5^t/30  
|t/20  
&120  
^t*7
```

Sentiti libero di riscrivere il codice, cambiare valori e operatori.

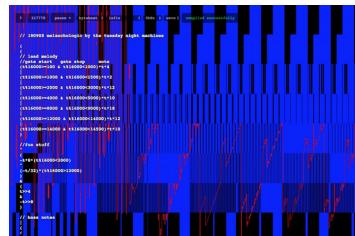
Nota che tutte le linee qui sopra cominciano con degli operatori logici come OR |, AND & e XOR ^. Questi sono necessari per combinare altre istruzioni dopo la **t** iniziale.  
Se trovi melodie o suoni interessanti, salva il codice in un file di testo.

Spero sia stato divertente! Chi ha bisogno di saper programmare?

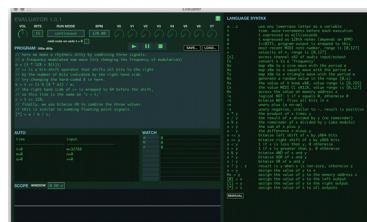
Comunque, in questa guida cercherò di insegnarti le basi della programmazione bytebeat, in modo da poter combinare errori e elementi più prevedibili.

## Come programmare bytebeat?

Esistono varie applicazioni per scrivere, suonare o registrare I bytebeat. Ecco quattro esempi:



[Greggman's HTML 5 bytebeat Player](#) ti permette di creare bytebeat dal tuo browser.



[Damien Quartz' Evaluator](#) è un player bytebeat con supporto MIDI per macOS o Windows. E' disponibile come plugin vst o standalone.



[Kymatica's BitWiz](#) è un'applicazione iOS che offre una interfaccia multilinea e un oscilloscopio.



[Single Cell's Caustic 3](#) per Android, iOS, Windows and macOS è una DAW che include un sintetizzatore bytebeat chiamato "8BitSynth":

## Cosa fa sì che un'espressione sia anche un bytebeat?

Ho scritto che un bytebeat è una espressione matematica, che genera un flusso audio a 8 bit con una frequenza di 8kHz, quindi si può pensare di scrivere qualsiasi cosa ci venga in mente. Ma che cosa fa di un'espressione un bytebeat?

Il bytebeat può essere considerato una forma d'arte a bassa complessità ([low-complexity art](#)) con strette connessioni alla [Demoscene](#), dove si prova a creare arte in strutture incredibilmente limitate.

Per il bytebeat, le limitazioni tipicamente sono:

- una sola espressione
- una sola variabile: t
- operatori matematici: () + - \* / %
- operatori bit per bit: & | ^ << >>
- operatori relazionali: < > <= >= == !=

Naturalmente queste sono limitazioni arbitrarie e ognuno è libero di fare ciò che vuole. Se non sono chiare alcune delle cose accennate qui sopra, non preoccuparti, le impareremo più avanti.

Ma prima, proviamo gli operatori descritti in combinazione a numeri casuali, come abbiamo fatto nel primo esercizio.

Vediamo cosa ne esce, salvate i vostri primi pezzetti di bytebeat!

### Esercizio 02: Operatori

Ecco un nuovo [link](#) al bytebeat Player con **t/20** come punto di partenza. Aggiungete una o più linee di quelle seguenti e sperimentate cambiando numeri e operatori:

```
|t*4*(t%10000>2000)&t>>4  
|t&64  
|t*t%128*(t%15000<2000)  
|t
```

### Suggerimento : Commenti

Dopo aver cliccato sul link, avrai notato che la prima linea comincia con i caratteri **//**. Questo denota un "commento" che è ignorato dal compilatore. Qualsiasi linea che comincia con **//** non modifica il risultato finale. Usa i due caratteri **//** per commentare parti di codice senza cancellarle e tieni traccia di cosa fanno le varie parti di codice quando le cose diventano particolarmente complesse.

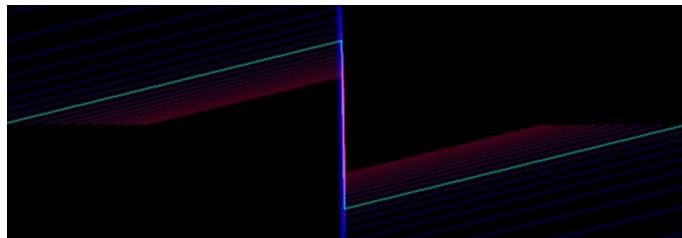
## Capire il tuo primo bytebeat

Bene, vediamo se riusciamo davvero a capire perchè con un certo pezzo di codice si ottiene un certo risultato.

Il più semplice bytebeat, come già sai, è il seguente:

t

(Premi [qui](#) per ascoltarlo nel tuo browser)



E' una dente di sega! Ma perchè?

La variabile predefinita **t** è un timer, o contatore, che comincia da 0 e incrementa di 1 a ogni audio frame.

Quindi a una frequenza di campionamento di 8kHz, t incrementa di 8000 ogni secondo, continuando a incrementare. Questo da solo non crea una dente di sega, ma solo un numero che incrementa costantemente. Per farla diventare una dente di sega, dobbiamo dare uno sguardo all'uscita audio a 8 bit.

Un numero binario a 8 bit significa che ci sono 8 cifre, che possono essere 0 o 1 e che tutti insieme rappresentano un numero da 0 a 255. Ognuno degli 8 bit rappresenta un numero:

128,64,32,16,8,4,2,1

I numeri corrispondenti ai bit che sono allo stato logico 1 sono sommati insieme per ottenere in risultante numero da 0 a 255

Ecco alcuni esempi:

00000000 = 0 (tutti i bit a 0)

00000001 = 1 (primo bit = 1)

00000010 = 2 (secondo bit = 2)

00000011 = 3 (secondo e primo bit = 2 + 1)

...

00010000 = 16 (quinto bit = 16)

00010001 = 17 (quinto e primo bit = 16 + 1)

...

10000000 = 128 (ottavo bit = 128)

10000001 = 129 (ottavo e primo bit = 128 + 1)

...

10001011 = 139 (ottavo ,quarto ,secondo e primo bit= 128 + 8 + 2 + 1)

...

11111111 = 255 (otto bit = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1)

Ricordi che **t** cresce di 8000 unità al secondo? Questo è molto di più del nostro range da 0 a 255. Possiamo usare numeri maggiori di 255 nel nostro codice, ma l'uscita audio fa riferimento solo ai primi 8 bit. Guardando al processo in binario, possiamo capire perchè viene generata la forma d'onda a dente di sega.

Diciamo, per esempio, di avere 10 bit disponibili, che ci permetterebbero di contare da 0 a 1023. Se abbiamo 10 bit ognuno rappresenterà i seguenti numeri:

512,256,128,64,32,16,8,4,2,1

Se contiamo a partire da 0, i 10 bit si riempiranno di 1, da destra a sinistra:

0000000000	= 0
0000000001	= 1
0000000010	= 2
0000000011	= 3
0000000100	= 4
0000000101	= 5
0000000110	= 6
0000000111	= 7
0000001000	= 8
0000001001	= 9
0000001010	= 10
0000001011	= 11
0000001100	= 12
0000001101	= 13
0000001110	= 14
0000001111	= 15
0000010000	= 16
0000010001	= 17
0000010010	= 18
0000010011	= 19
0000010100	= 20
...	
0011111110	= 244
0011111111	= 255

Come abbiamo detto, usiamo 10 bit in questo esempio e possiamo contare fino a 1023. Ma la nostra uscita prende in considerazione solo i primi 8 bit, dunque ecco cosa succede quando il nostro contatore eccede il valore di 255:

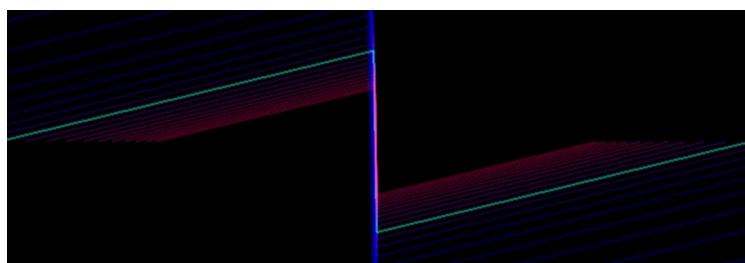
`0011111111 = 255 (255 in 10bit)`  
`0011111111 = 255 (255 troncato dopo 8 bit, nulla cambia)`

`0100000000 = 256 (256 in 10bit)`  
`0100000000 = 0 (256 troncato dopo 8 bit)`

`0100000001 = 257 (257 in 10bit)`  
`0100000001 = 1 (257 troncato dopo 8 bit)`

`0100000010 = 258 (258 in 10bit)`  
`0100000010 = 2 (258 troncato dopo 8 bit)`

Quindi il nostro contatore interno `t` può assumere valori maggiori di 255, ma il risultato dell'espressione viene troncato a 8 bit, quindi va da 0 a 255, ritorna a 0, poi sale fino a 255. E' una dente di sega!



Guardiamo alla catena di eventi nel nostro bytebeat player usando la semplice espressione `t*50`:

Input	Espressione	Risultato	Output	Output binario
<code>t=0</code>	<code>t*50</code>	0	0	(0000000000)
<code>t=1</code>	<code>t*50</code>	50	50	(0000110010)
<code>t=2</code>	<code>t*50</code>	100	100	(0001100100)
<code>t=3</code>	<code>t*50</code>	150	150	(0010010110)
<code>t=4</code>	<code>t*50</code>	200	200	(0011001000)
<code>t=5</code>	<code>t*50</code>	250	250	(0011111010)
<code>t=6</code>	<code>t*50</code>	300	44	(0100101100)
<code>t=7</code>	<code>t*50</code>	350	94	(0101011110)
<code>t=8</code>	<code>t*50</code>	400	144	(0110010000)

Quanto può crescere `t`? Fino a 2147483647, il massimo numero rappresentabile con 32 bit, sembra essere la norma per i bytebeat player. Quindi a raggiungere questo valore si impiegheranno circa 74 ore. E poi che succederà? Non so, riprenderà da 0 o aprirà il portale alla dimensione ChipTune...provate voi!

## Programmare con operatori matematici

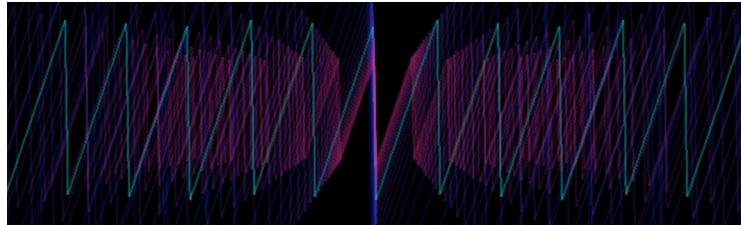
Ora modifichiamo la nostra onda a dente di sega con gli operatori matematici:

\* / + - %

**Moltiplicare t** per un numero incrementa la frequenza della nostra forma d'onda.

Per esempio:

t\*12



**Dividere t** per un numero diminuirà la frequenza della nostra dente di sega. Questo ci porterà al di sotto della soglia di udibilità, nei territori degli LFO.

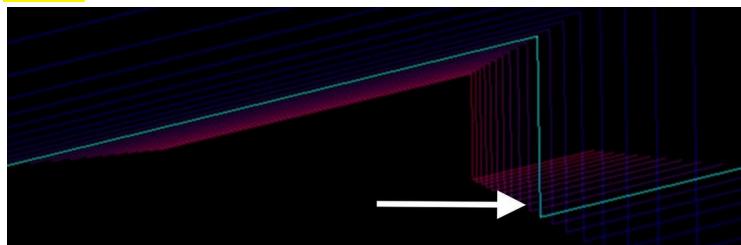
Per esempio:

t/30

**Sommare o sottrarre numeri** applica un offset a t, che non cambia niente dal punto di vista sonoro.

Per esempio:

t+200



Questo non ha un effetto udibile ma sposta la fase della forma d'onda, e diventa apparente quando si combinano forme d'onda di fase diversa. Lo proveremo presto!

Un'altro uso dell'operatore meno è quello di invertire un valore, come se fosse moltiplicato per -1.

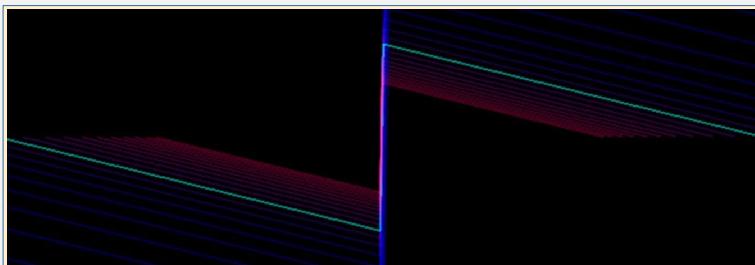
### Esercizio 03: Moltiplicazione e divisione

Apri il tuo bytebeat player preferito e usa qualcuno degli esempi precedenti giocando con moltiplicazione e divisione.

Prova anche:

-t

Per l'inversione della forma d'onda.



**Modulo.** Ora le cose si fanno interessanti! L'operatore modulo restituisce il resto di una divisione. Cos'è il resto?

Un pò di matematica:

$$15 / 4 = 3.75$$

Significa che il 4 sta nel 15 per 3 volte intere, più un pochettino. Questo è il resto.  
 $0.75 * 4 = 3$

Per ottenere il resto puoi scrivere la seguente espressione:

$$15 \% 4 = 3$$

Ma...a cosa può servirci per la programmazione bytebeat?

Grazie per aver chiesto! Guardiamo qualche risultato di operazioni con il modulo e cerchiamo di individuare un pattern:

$$4 \% 4 = 0$$

$$5 \% 4 = 1$$

$$6 \% 4 = 2$$

$$7 \% 4 = 3$$

$$8 \% 4 = 0$$

$$9 \% 4 = 1$$

$$10 \% 4 = 2$$

$$11 \% 4 = 3$$

$$12 \% 4 = 0$$

Notate nulla? E' come una piccola dente di sega. Il modulo di qualsiasi numero diviso per quattro è sempre 0 o 1 o 2 o 3.

Parlando più genericamente:

$x \% y$  è sempre compreso tra 0 e  $y - 1$

Programmiamo una piccola dente di sega e sentiamola!

#### Esercizio 04 : Modulo %

Apri il tuo bytebeat player e scrivi:

`t%128`

Il risultato sarà compreso tra 0 e 127, che significa che stiamo usando la metà del range totale, riducendo l'ampiezza della forma d'onda(il "volume").

Prova anche altri valori invece di 128.

Ben fatto. Hai notato che l'onda non cambia solo in ampiezza ma anche in frequenza?  
Questo significa che si può usare il modulo per cambiare la frequenza di valori piccoli.

Il modulo funziona come la nostra uscita a 8 bit, cioè riduce il valore in ingresso a un range specifico.

## Programmare con gli operatori bit per bit.

Usare gli operatori matematici probabilmente non è una grossa novità, quindi proviamo ora gli operatori bit per bit (bitwise). Questi compiono operazioni basate sulla rappresentazione binaria di un numero.

### AND &

Compara il valore dei singoli bit di due numeri binari e restituisce 1 solo se entrambi i singoli bit sono a 1.

010**1** = 5  
& 001**1** = 3  
= 000**1** = 1 = 5&3

10**11** = 11  
& 00**11** = 3  
= 00**11** = 3 = 11&3

0011 = 3  
& 0100 = 4  
= 0000 = 0 = 3&4

E' un po' come moltiplicare i singoli bit:

0 \* 0 = 0  
1 \* 0 = 0  
1 \* 1 = 1

Prova a individuare un pattern in questi esempi:

111111100 = 1020  
& 0010000000 = 128  
= 0010000000 = 128

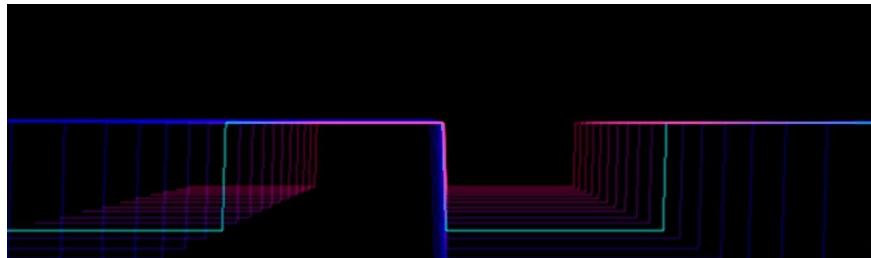
1100001110 = 782  
& 0010000000 = 128  
= 0000000000 = 0

1011011101 = 733  
& 0010000000 = 128  
= 0010000000 = 128

1001011010 = 602  
& 0010000000 = 128  
= 0000000000 = 0

Quindi se scriviamo **t&128**, avremo solo due valori che si alternano, 128 e 0. Sembra un'onda quadra, giusto?

Infatti, è un'onda quadra che usa solo metà del range di 8 bit.



### Esercizio 06: AND &

Inserisci ora:

t&128

E' un onda quadra! Rimpiazza 128 con un altro numero e guarda che cosa succede.

Ora prova a combinare una dente di sega con un'altra più lenta:

t&t/30

Che cosa succede se inverti il segno della seconda forma d'onda?

t&-t/30

Un effetto dell'operazione di AND & è che il risultato non può mai eccedere l'operando più piccolo.

Per esempio:

t&140

non potrà mai superare il valore di 140.

### OR |

Compara il valore dei bit e restituisce 1 se uno o l'altro dei bit è a 1. Viene chiamato anche OR inclusivo.

0101=5

|0011=3

=0111=8=5|3

1011=11

|0011=3

=1011=11=11|3

**0011=3**  
**|0100=4**  
**=0111=8=3|4**

E' un poco come sommare i due bit.

0 + 0 = 0  
1 + 0 = 1  
1 + 1 = uhm...

Beh, non proprio, ma è importante sapere che si può usare OR per sommare due forme d'onda come in un audio mixer super lo-fi.

### Esercizio 07: OR |

Apri il tuo bytebeat player e scrivi:

**t**  
poi aggiungi:  
**|t\*4**  
e poi:  
**|t\*12**

Usa OR | per aggiungere altre forme d'onda di frequenze differenti (**t** moltiplicato o diviso per un numero).

### XOR ^

chiamato anche OR esclusivo, restituisce 1 solo se i bit sono diversi tra loro.

**0101 = 5**  
**^ 0011 = 3**  
**= 0110 = 6= 5^3**

**1011 = 11**  
**^ 0011 = 3**  
**= 1000 = 8= 10^3**

**0011 = 3**  
**^ 0100 = 4**  
**= 0110 = 6= 3^4**

L'operatore XOR non limita il risultato a un minimo o a un massimo valore.

### Esercizio 08: AND &, OR |, X OR ^

Se non hai ancora provato, procedi mixando qualche forma d'onda utilizzando gli operatori logici AND &, OR | and XOR ^.

Ecco un divertente esempio, per cominciare. Prima di tutto, cerca di capire che tipo di forma d'onda genera ciascuna linea di codice (veloce o lenta? dente di sega o quadra?) e poi ascolta il risultato di quando le combini utilizzando gli operatori logici.

```
t&128  
|  
t&64&t  
/20  
^t/15  
|t*2
```

Come già detto, l'operatore OR | produce risultati più simili a quelli di un mixer audio. Sonicamente, anche AND & può funzionare per sommare delle forme d'onda mentre XOR ^ può spezzare il segnale. Dipende tutto dall'espressione che stai utilizzando, quindi non smettere mai di sperimentare.

Bene, ora parliamo dei restanti operatori!

### Bitshift left << and Bitshift right >>

sono leggermente differenti dagli operatori logici, non comparano numeri binari ma spostano i bit del primo operando di una certa quantità specificata dal secondo operando.

```
12<<3  
00001100=12  
      <---  
01100000=96
```

Tutti i bit vengono semplicemente spostati di tre posizioni a sinistra. I bit aggiunti sono a 0.

L'altro operatore è >>, il bitshift a destra:

```
12>>3  
00001100=12  
      ---  
>00000001=1
```

Qui tutti i bit vengono spostati a destra.

$12/2^3=1$   
or  
 $12/(2*2*2)=1$

Perchè utilizzare il bitshift invece di normali moltiplicazioni e divisioni? Il bitshift viene eseguito più velocemente dal computer ma questo non è molto importante. Però l'uso del bitshift può condurre a risultati particolari, come vedremo ora.

### Esercizio 09 : Bitshifts << >>

Prova questi esempi. Come cambia la frequenza effettuando il biteshift di 1,2 oppure 3 ?

t<<1  
t<<2  
t<<3

t<<1  
t<<2  
t<<3

Un bitshift di 1 produce un salto di ottava. Buono a sapersi!

## Programmare con gli operatori relazionali

Una nuova categoria! Gli operatori relazionali stabiliscono se una certa relazione tra due numeri è vera o falsa e restituiscono rispettivamente 1 o 0.

```
2 > 1 = 1 ("due è maggiore di uno")
2 < 1 = 0 ("due è minore di uno")
2 == 1 = 0 ("due è uguale a uno")
2 != 1 = 1 ("due è diverso da uno")
2 >= 2 = 1 ("due è maggiore o uguale a uno")
2 <= 3 = 1 ("due è minore o uguale a tre")
```

Possiamo usare gli operatori relazionali per creare degli interruttori on/off per delle parti del nostro codice, che può portare i nostri bytebeat a un altro livello.

### Esercizio 10: Operatori relazionali

Ferma il bytebeat player e assicurati che **t** sia a 0 (nel Greggman's HTML5 bytebeat player, basta cliccare sul valore tra il punto interrogativo e il pulsante di play). Ora copia il seguente codice:

```
(t>8000)*t
|(t>16000)*t*2
|(t>24000)*t*6
|(t>32000)*t*12
|(t>40000)*t*40
|(t>48000)*t/20
```

Dopo sei secondi, puoi premere stop e ripartire da **t = 0**, se vuoi.

Ricordi che l'output di un'operazione relazionale è 0 o 1? Moltiplicandola per una parte della nostra espressione, noi possiamo attivare o disattivare quella parte.

Nell'esempio abbiamo 8000 campioni di silenzio, che a una frequenza di 8kHZ è un secondo. Poi, quando **t** diventa maggiore di 8000, la nostra prima linea di codice viene interpretata come:

```
(1)*t
```

Un secondo dopo, quando t supera 16000, abbiamo:

```
(1)*t
|(1)*t*2
```

Una seconda forma d'onda viene sommata alla prima grazie all'operatore OR **|**. Ora sperimenta con altri operatori e non dimenticare di annotare le tue scoperte.

## Mettiamo tutto insieme!

In questo capitolo, cercherò di illustrare alcuni esempi in cui applicheremo i vari concetti imparati finora.

### Programmare uno step sequencer

Costruiamo un tradizionale step sequencer a 16 step, che farà suonare 16 suoni diversi. Per far ciò, abbiamo bisogno di un contatore che indichi gli step della nostra sequenza. Abbiamo già lavorato con un contatore di questo tipo, ricordi?

Guarda questo pezzo di codice:

```
t%16
```

L'operatore modulo! In questo caso, otteniamo proprio i valori tra 0 e 15. Però alla frequenza di 8kHz, questi 16 step sono molto brevi, perciò rendiamoli più lunghi:

```
t%16000
```

Ora ciascuno dei nostri step avrà durata di 1000 frame, cioè 1/8 di secondo.

Poi, dobbiamo definire gli step:

*Step1 deve durare da 0 a 999*

*Step2 da 1000 a 1999*

*Step3 da 2000 a 2999*

...

*Step15 da 14000 a 14999*

*Step16 da 15000 a 15999*

Come implementare questi step? Con gli operatori relazionali:

```
(t%16000>=0 & t%16000<1000)
```

Il codice qui sopra produce 1 finchè il contatore di step **t%16000** è **minore o uguale** a 0 e **minore** di 1000.

Questo non produce nessun suono, ma basterà moltiplicarlo per la nostra vecchia variabile **t**:

```
(t%16000>=0 & t%16000<1000)*t
```

Prova questa istruzione e poi aggiungi una seconda linea preceduta dall'operatore logico OR | per il secondo step:

```
|(t%16000>=1000 & t%16000<2000)*t*3
```

E continua per i rimanenti 14 step.

Ecco il codice completo per i 16 step, che suona una semplice melodia.

```
(t%16000 >= 0 & t%16000 < 1000) * t  
|(t%16000 >= 1000 & t%16000 < 2000) * t*3  
|(t%16000 >= 2000 & t%16000 < 3000) * t*5  
|(t%16000 >= 3000 & t%16000 < 4000) * t*8  
|(t%16000 >= 4000 & t%16000 < 5000) * t*2  
|(t%16000 >= 5000 & t%16000 < 6000) * t  
|(t%16000 >= 6000 & t%16000 < 7000) * t*12  
|(t%16000 >= 7000 & t%16000 < 8000) * t*8  
|(t%16000 >= 8000 & t%16000 < 9000) * t*3  
|(t%16000 >= 9000 & t%16000 < 10000) * t  
|(t%16000 >= 10000 & t%16000 < 11000) * t*4  
|(t%16000 >= 11000 & t%16000 < 12000) * t*10  
|(t%16000 >= 12000 & t%16000 < 13000) * t*7  
|(t%16000 >= 13000 & t%16000 < 14000) * t*6  
|(t%16000 >= 14000 & t%16000 < 15000) * t*5  
|(t%16000 >= 15000 & t%16000 < 16000) * t*6
```

La forma d'onda a dente di sega ora sta cominciando a darmi sui nervi, quindi per favore, cambiamola in un'onda quadra. Questo può essere fatto facilmente, mettendo tutto il codice del sequencer tra parentesi e aggiungendo **&128**.

**(codice sequencer)&128**

Questa melodia può essere ulteriormente variata; per esempio, come possiamo trasporla di un'ottava ogni 16 step?

Avremo bisogno di

- codice per la trasposizione
- un contatore e un'operazione relazionale che attui la trasposizione

Ecco la nuova linea di codice:

**<<(t%32000<=16000)**

Per trasporre di un'ottava l'uscita del nostro sequencer, possiamo semplicemente effettuare un bitshift a sinistra di 1 bit. Per attivare il bitshift usiamo il counter  $t\%32000$ , lungo il doppio del counter del nostro sequencer e controlliamo che sia minore o uguale a 16000.

Perchè funzioni correttamente, dobbiamo inserire la nostra linea di codice tra il sequencer e la conversione a onda quadra:

**(codice sequencer)<<(t%32000<=16000)&128**

Questo è un approccio molto deterministico alla programmazione di un sequencer, senza procedere per tentativi e errori. Da qui si può partire e modificare il codice a proprio piacere, per esempio cambiando il suono a ognuno dei 16 step, o la lunghezza degli step...

### Suggerimento : Interruzioni di riga

Le istruzioni scritte in una sola linea possono impressionare ma spesso non forniscono indicazioni su cosa succede davvero nel codice. Usare i commenti e le interruzioni di linea può essere molto d'aiuto.

Poco chiara:

```
t*30&(t%3000<1500)*t/10
```

Chiara:

```
t*30 //dente di sega molto alta  
&  
(t%3000<1500)*t/10 //interrompe ritmicamente
```

## Programmare un'onda quadra con PWM

Le onde quadre sono ancora più interessanti se applichiamo una modulazione di larghezza dell'impulso (pulse width modulation).

Vediamo come implementare questo effetto in bytebeat. Prima di tutto, scriviamo un'espressione che ci dia la possibilità di controllare il tempo per cui l'onda rimane allo stato alto.

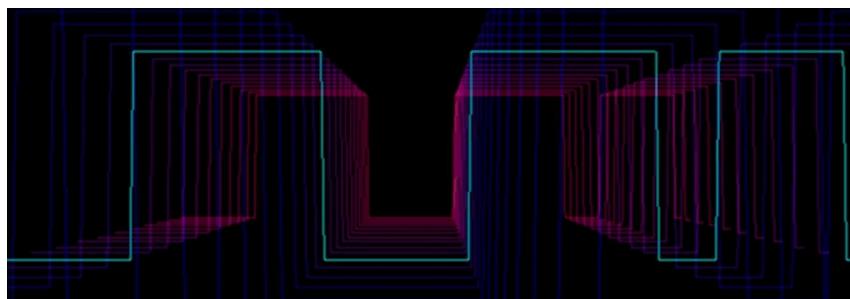
Ecco una possibile soluzione:

`(t%100>=50)*255`

C'è un contatore `t%100` che conta da 0 a 99. Testiamo se il contatore è maggiore o uguale a 50 e moltiplichiamo il risultato per l'ampiezza che vogliamo ottenere, dove 0 è il silenzio e 255 la massima ampiezza ottenibile. Cambiando il numero 50 possiamo agire sulla larghezza dell'impulso dell'onda quadra.

Per ottenere la modulazione, possiamo scrivere:

`(t%100<=t/200%100)*255`



Qui il numero 50 è sostituito da `t/200%100`, un altro contatore da 0 a 100 che conta molto più lentamente, cioè a `t/200`. Prova a cambiare questi valori e ascolta il risultato.

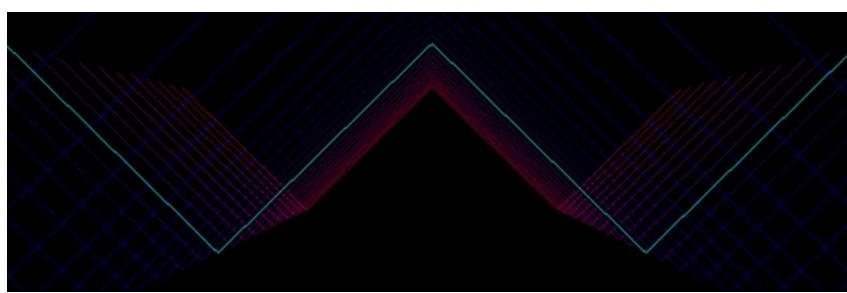
## Programmare un onda triangolare con wave folding

Finora abbiamo lavorato solo con forme d'onda molto grzze; scriviamo ora il codice per un'onda triangolare

Sappiamo che **t** può generare una dente di sega crescente e **-t** un'inversione della stessa. Se a ogni ciclo passiamo da **t** a **-t** possiamo ottenere un'onda triangolare. Il ciclo della nostra dente di sega è 256, quindi dobbiamo cambiare forma d'onda ogni 256 audio frame.

Ecco l'espressione che possiamo usare:

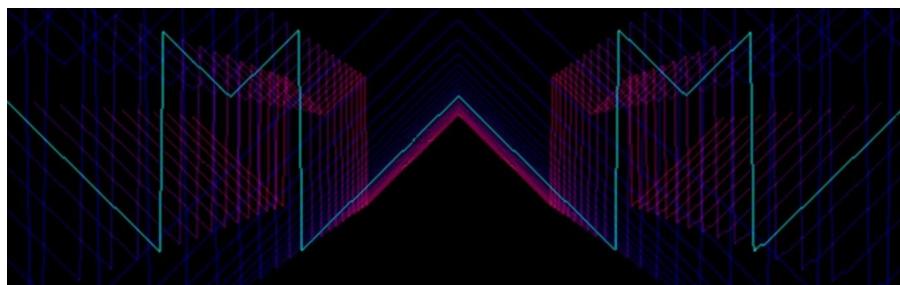
```
t      *(t%512>=256)  
|  
(-t-1)*(t%512<2 5 6 )
```



Ora proviamo il wave folding, mettendo tra parentesi il codice dell'onda triangolare e aggiungendo un offset di **t/60** che viene sommato costantemente alla forma d'onda, spostandola verso l'alto.

Nota che la variabile **t** è stata anche moltiplicata per 4 per alzare la frequenza della forma d'onda.

```
(  
t*4*(t*4%512>=256)  
|  
(-t*4-1)*(t*4%512<256)  
)  
+t/60
```



## Rompere le catene

Finora siamo stati dentro alle comunemente accettate e arbitrariamente definite regole della programmazione bytebeat. Diamo un'occhiata fuori dal recinto!

Qualche volta puoi voler alterare parte del bytebeat mentre viene eseguito senza digitare sulla tastiera, per esempio durante una live performance. Un modo semplice per ottenere questo effetto è usare altre variabili oltre a **t**.

[Bitwiz Audio synth di Kymatika](#), per esempio, offre un pad multi-touch XY e la possibilità di ricevere dati MIDI. Per esempio puoi settare l'applicazione per ricevere il control change relativo alla modulation wheel della tua tastiera MIDI e associarlo a una variabile che cambi la frequenza della forma d'onda generata.

Un'altra limitazione che possiamo ignorare è "solo un'espressione". Nel [Single Cell's Caustic 8bitSynth](#), puoi scrivere due espressioni e miscelarne le due uscite separatamente.

Altri editor ti permettono di scrivere più espressioni o dichiarare delle variabili, separandole con la virgola.

Per esempio:

```
s=6,  
(t*s)*(t*s%512>256)  
|  
(-t*s-1)*(t*s%512<256)
```

Per concludere, alcuni bytebeat player come [Greggman's HTML 5 Player](#), può accettare altri operatori rispetto a quelli introdotti in questa guida, come **sin**, **cos**, **tan**, **sqrt...**

## Fonti e ulteriori letture

Molte grazie per aver letto questa guida! Spero ti sia piaciuto imparare a programmare un bytebeat. Se hai commenti o suggerimenti riguardo questo PDF, per favore fammeli sapere.

Naturalmente c'è ancora molto da dire sull'argomento, quindi ecco una lista di link interessanti.

Testi di Ville-Matias "viznut" Heikkila, che ha iniziato l'intera faccenda del bytebeat nel 2011:

<http://viznut.fi/texts-en/>

[bytebeat\\_algorithmic\\_symphonies.html](http://viznut.fi/texts-en/bytebeat_algorithmic_symphonies.html)

[http://viznut.fi/texts-en/bytebeat\\_exploring\\_space.pdf](http://viznut.fi/texts-en/bytebeat_exploring_space.pdf)

[http://viznut.fi/texts-en/bytebeat\\_deep\\_analysis.html](http://viznut.fi/texts-en/bytebeat_deep_analysis.html)

<https://countercomplex.blogspot.com/2011/10/some-deep-analysis-of-one-line-music.html>

Articolo con un sacco di altri link:

<http://canonical.org/~kragen/bytebeat/>

Post su Reddit con molti esempi di codice:

[https://www.reddit.com/r/bytebeat/comments/20km9l/cool\\_equations/?](https://www.reddit.com/r/bytebeat/comments/20km9l/cool_equations/?)

[st=jm4me4ki&sh=d5e70bef](#)

Bytebeat software:

<http://coleingraham.com/2013/04/28/bytebeat-shell-script/>

<https://github.com/greggman/html5bytebeathttps://damikyu.itch.io/evaluator>

<http://kymatica.com/Software/BitWiz>

<http://www.singlecellsoftware.com/caustic>

Articoli su Wikipedia:

[https://en.wikipedia.org/wiki/Low-complexity\\_art](https://en.wikipedia.org/wiki/Low-complexity_art)

<https://en.wikipedia.org/wiki/>

[https://en.wikipedia.org/wiki/Binary\\_number](https://en.wikipedia.org/wiki/Binary_number)

<https://en.wikipedia.org/wiki/>

[Bitwise\\_operations\\_in\\_C](#)

The Tuesday Night Machines:

<https://nightmachines.tv/youtube>

support TTNM :

<http://nightmachines.tv/support.html>

Get the latest version of this PDF

here:<https://github.com/TuesdayNightMachines/bytebeats>