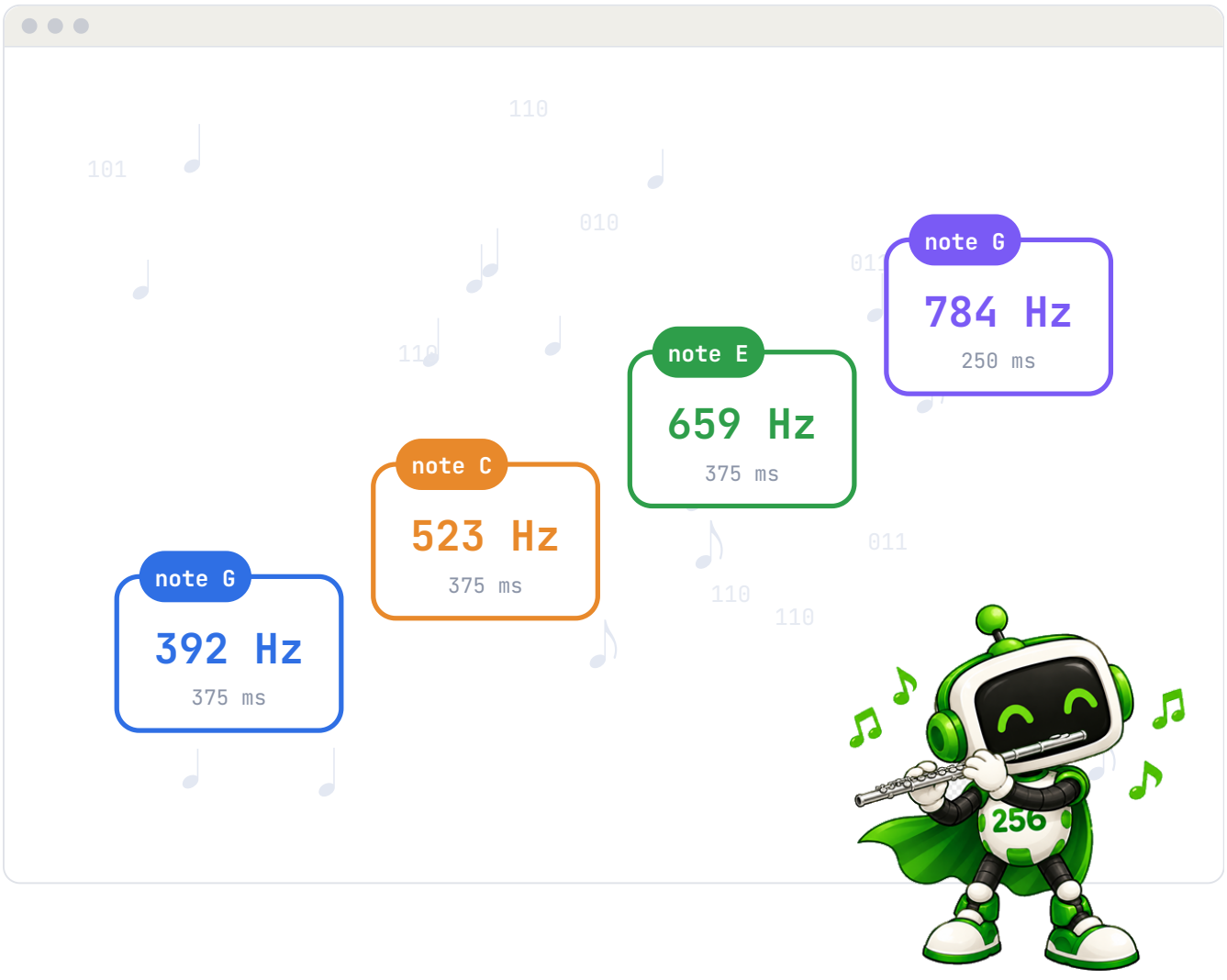
LEARN TO CODE · SOUND

Chapter 4

Sound and Music

Now that you have colour, graphics and variables, it's time to make some noise. In this chapter you will find out what sound is, play notes of any pitch and length, and turn a piece of sheet music into a program that plays a tune.



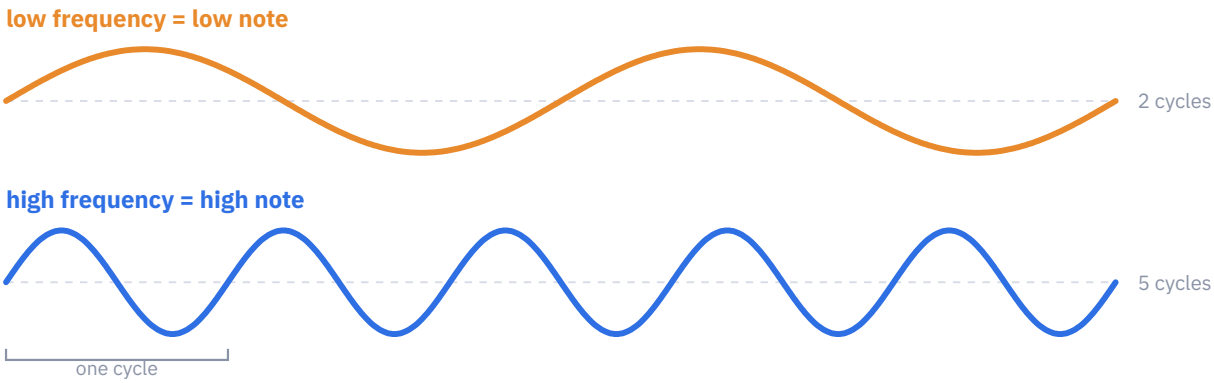
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

Sound basics

Before making any noise, let's look at what a sound actually is.

Sound is made by vibrating air striking your eardrum. These vibrations are called **sound waves**. When the air vibrates quickly you hear a high note; when it vibrates slowly you hear a low note. How fast the air vibrates is called the **frequency**.



In the same amount of time the high note makes more ups and downs (cycles) than the low note.



NEW CONCEPT

Frequency is measured in **hertz** (Hz): the number of cycles (ups and downs) a wave makes in one second. A person with good hearing can hear from about **20 Hz**, a deep rumble, up to about **20,000 Hz**, a very high whistle. The A that an orchestra tunes to is 440 Hz.

Another property of a sound is its **length**, or duration. Computers measure time very precisely, so BASIC-256 measures the length of a sound in **milliseconds** (ms). A millisecond is one thousandth of a second: 1000 ms make one second, and 500 ms is half a second.

Good to know: BASIC-256 will play any frequency you give it, but your speakers and your ears set the real limits. Small laptop speakers often can't play very low notes, and very high tones are hard to hear — or just unpleasant. Most tunes stay between about 100 Hz and 2000 Hz.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

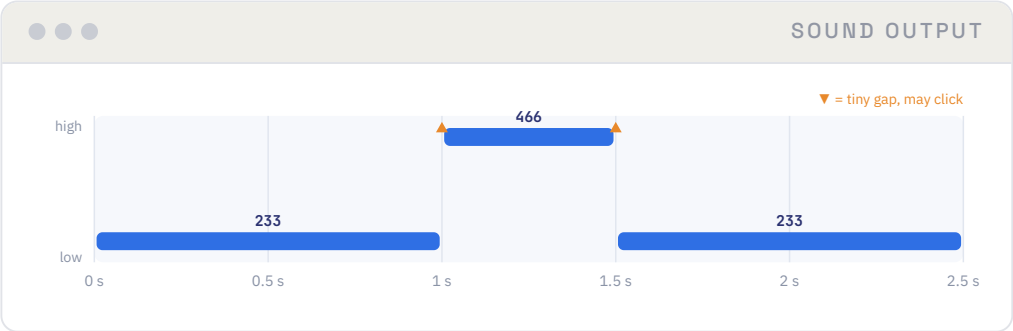
Playing a sound

The `sound` statement plays a note of the frequency and length you choose.

Let’s make some sounds.

sounds.kbs BASIC-256

```
# sounds.kbs - play three notes
sound 233, 1000
sound 466, 500
sound 233, 1000
```



Program 26 · Play Three Individual Notes

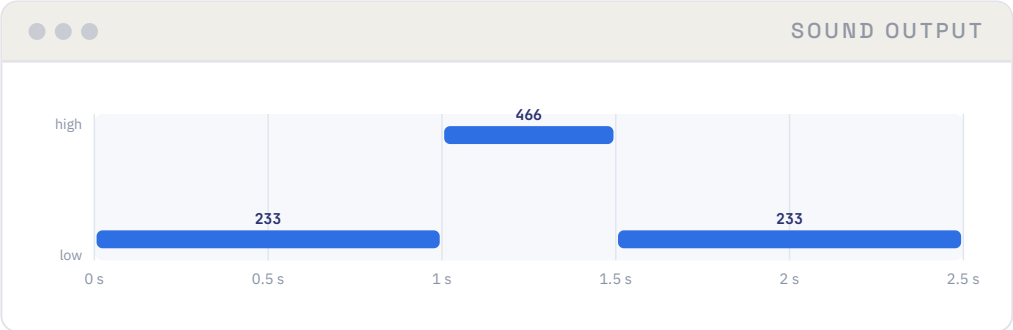
Each `sound` statement plays one note: first 233 Hz for 1000 ms (one second), then 466 Hz for half a second, then 233 Hz again. Notice that 466 is exactly double 233 — doubling a frequency gives the same note, one **octave** higher. You’ll see more of that in Section 3.

You may have heard a small click between the notes. Each `sound` statement makes a separate little sound: the wave stops suddenly at the end of one note, and a fresh wave starts for the next. To smooth out the change from one note to the next, the `sound` statement can also take a **list** of frequencies and durations.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

soundslist.kbs BASIC-256

```
# soundslist.kbs - play a list of notes
sound {233, 1000, 466, 500, 233, 1000}
```



Program 27 · List of Sounds

The first two values are the frequency and duration of the first note. Once that note has played, the next two values are used, and so on. This program plays the same three notes for the same lengths, but BASIC-256 builds them into one continuous sound before it starts playing, so the notes flow smoothly into each other.



NEW CONCEPT

```
sound frequency, duration
sound {frequency1, duration1, frequency2, duration2 …}
sound numeric_array[]
```

The basic **sound** statement takes two values: (1) the frequency of the sound in hertz and (2) the length of the tone in milliseconds.

The second form uses a single **list** in curly braces `{ }`. The values come in pairs — frequency, duration, frequency, duration — so a list must always have an even number of values. It's easy to lose track in a long list, so be careful.

The third form uses an **array** holding frequencies and durations. Arrays are covered in a later chapter.

Music also has **rests**: a rest means play nothing for a certain length of time. In a list of sounds you insert a rest by giving a frequency of **zero** (0) and the length of the silence.
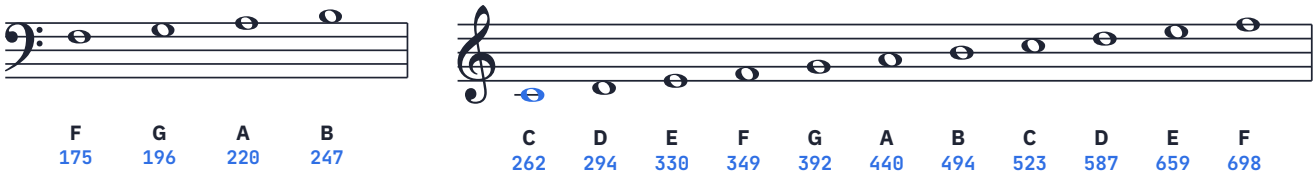
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

From sheet music to frequencies
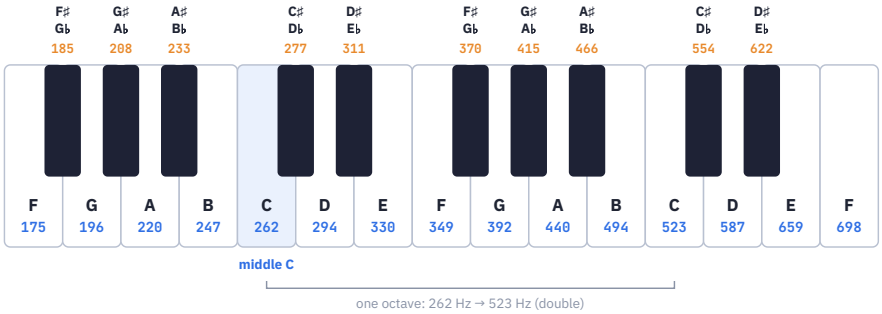
How do we get BASIC-256 to play a tune? First, turn each note on the music staff into a frequency.

Music is written as notes on a **staff** of five lines. The higher a note sits on the staff, the higher it sounds. Each note has a letter name from A to G. The chart below shows the notes from the F below middle C up to the F an octave and a half higher, with the frequency each one makes, rounded to a whole number.



The notes on the staff, their names and their frequencies in Hz. Middle C (blue) sits on a short extra line of its own.

On a piano, the white keys are the plain letter notes. A black key is a **sharp** (♯) of the white key to its left and a **flat** (♭) of the white key to its right — so C♯ and D♭ are the same key, with the same frequency.



Every note in the chart on a keyboard. The black keys are labelled with both of their names.



NEW CONCEPT

From one note to the same note an **octave** higher, the frequency **doubles**: middle C is 262 Hz and the next C up is 523 Hz. Going an octave down halves it.

So if you need a note that isn't on the chart, take the same note from one octave away and double it (to go higher) or halve it (to go lower). The G above the chart is 2 × 392 = **784 Hz**.



EXPLORE

BASIC-256 can also work out the frequency for you. Instead of a number, write the note's letter and octave number as a string: middle C is `"C4"`, and the A above it is `"A4"`, 440 Hz. A sharp or flat goes *after* the octave number: `"F4#"`, `"B4b"`. Try `sound {"G4", 500, "C5", 500, "E5", 500}`.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

How long is a note?

Now that we have the frequencies, we need the duration of each note.

Each kind of note lasts a certain number of **beats**. The speed of the music, its **tempo**, is given in beats per minute (BPM). If you know the tempo, you can work out a note's duration in milliseconds:

$$\text{note duration (ms)} = \frac{1000 \times 60}{\text{beats per minute}} \times \text{length in beats}$$

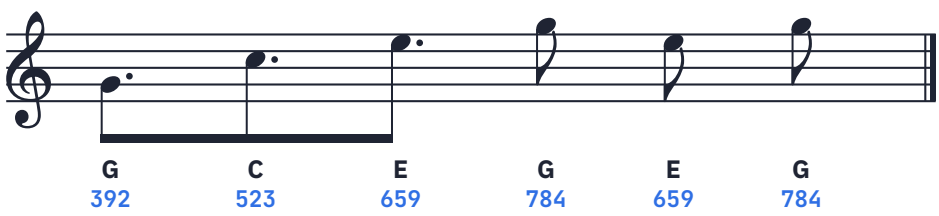
1000 × 60 is the number of milliseconds in a minute. Dividing it by the tempo gives the length of one beat.

The table shows the common notes and rests, how long they are compared to each other, and their durations at three typical speeds.

NOTE NAME	NOTE	REST	BEATS	AT 100 BPM	AT 120 BPM	AT 140 BPM
Dotted whole	𝅝.	𝄻.	6.000	3600 ms	3000 ms	2571 ms
Whole	𝅝	𝄻	4.000	2400 ms	2000 ms	1714 ms
Dotted half	𝅗𝅥.	𝄼.	3.000	1800 ms	1500 ms	1286 ms
Half	𝅗𝅥	𝄼	2.000	1200 ms	1000 ms	857 ms
Dotted quarter	♩.	𝄽.	1.500	900 ms	750 ms	643 ms
Quarter	♩	𝄽	1.000	600 ms	500 ms	429 ms
Dotted eighth	♪.	𝄾.	0.750	450 ms	375 ms	321 ms
Eighth	♪	𝄾	0.500	300 ms	250 ms	214 ms
Dotted sixteenth	𝅘𝅥𝅯.	𝄿.	0.375	225 ms	188 ms	161 ms
Sixteenth	𝅘𝅥𝅯	𝄿	0.250	150 ms	125 ms	107 ms

Durations are rounded to the nearest millisecond.

Now let's have the computer play "Charge!". The music is below. The high G is not on our chart, so we double the G one octave lower: 2 × 392 = 784 Hz. At 120 BPM a dotted eighth note lasts 375 ms and an eighth note 250 ms.



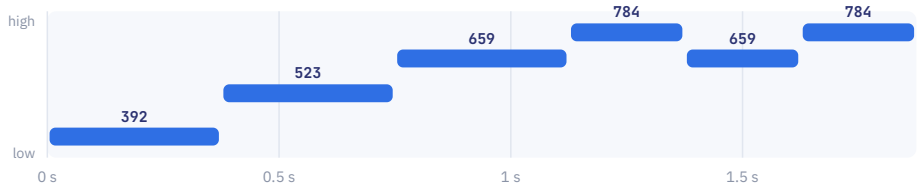
"Charge!" — three dotted eighth notes and three eighth notes.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

charge.kbs BASIC-256

```
# charge.kbs - play Charge!
sound {392, 375, 523, 375, 659, 375, 784, 250, 659, 250, 784, 250}
say "Charge!"
```

SOUND OUTPUT



says **“Charge!”**

Program 28 · Charge!

Notice that BASIC-256 only says “Charge!” after the music has finished. The `sound` statement waits until the whole sound has played before the program moves on to the next line.

Instead of working out the note durations by hand, let’s use variables to calculate and store them. Here is “Charge!” again, using the formula from the start of this section.

charge2.kbs BASIC-256

```
# charge2.kbs - play Charge! using variables
bpm = 120                # beats per minute
bms = 1000 * 60 / bpm    # milliseconds per beat
dottedeighth = bms * 0.75
eighth = bms * 0.5

sound {392, dottedeighth, 523, dottedeighth, 659, dottedeighth,
       784, eighth, 659, eighth, 784, eighth}
say "Charge!"
```

SOUND OUTPUT

♪ plays exactly the same tune as Program 28

says **“Charge!”**

Program 29 · Charge! with Variables

`bms` works out to 500.0 ms, the length of one beat at 120 BPM, so `dottedeighth` is 375.0 and `eighth` is 250.0. The variables can go straight into the list, in place of numbers.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



TRY IT

Change `bpm = 120` to `bpm = 60` and run the program again, then try `bpm = 200`. One number changes the speed of the whole tune.

Sound waits for the music

`sound` is a **blocking** statement: BASIC-256 plays the whole sound, and only then goes on to the next line. That's perfect for playing notes and tunes on your computer, like every program in this chapter.

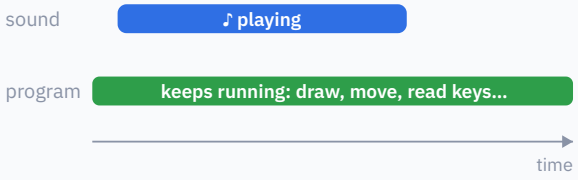
It is **not** suitable for games and animations. While the sound is playing, the program is frozen: nothing moves, the screen isn't updated and no keys are read until the sound has ended. For those, BASIC-256 has a non-blocking statement, `soundplay`. It starts the sound and lets the program carry straight on while it plays.

sound



the program **waits** — tunes and sound effects

soundplay



the program **keeps going** — games and animations

Coming later: `soundplay` and the statements that go with it are covered in the **Advanced Sound and Music** chapter.
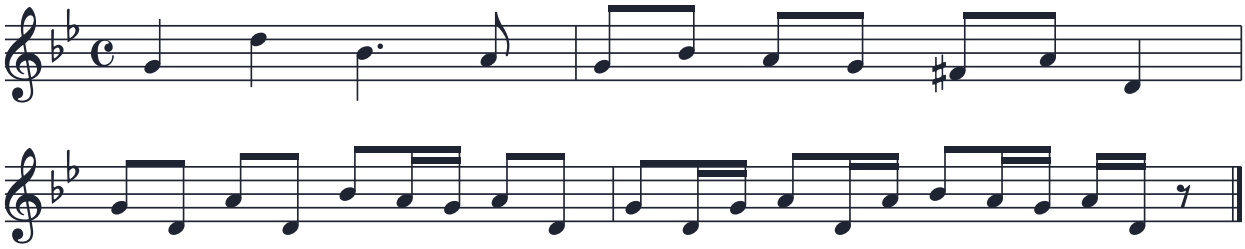
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

Bach's Little Fugue

For this chapter's big program, let's take a piece of music by Johann Sebastian Bach and write a program to play the opening of his *Little Fugue in G minor*.



The first four measures of J.S. Bach's Little Fugue in G minor. The two flats at the start mean every B and E is played flat; the ♯ before one F makes it F♯ (370 Hz).

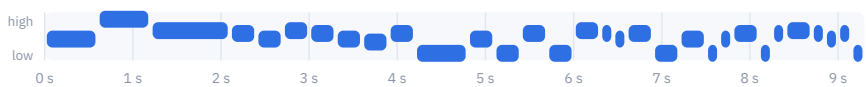
littlefugue.kbs — BASIC-256

```
# littlefugue.kbs - J.S. Bach, Little Fugue in G minor (BWV 578)

tempo = 100               # beats per minute
msmin = 1000 * 60         # milliseconds in a minute
q = msmin / tempo         # a quarter note is one beat
e = q / 2                 # eighth note: half a quarter
s = q / 4                 # sixteenth note: a quarter of a quarter
dq = q + e                # dotted quarter: quarter + eighth

sound {392, q, 587, q, 466, dq, 440, e,
       392, e, 466, e, 440, e, 392, e, 370, e, 440, e, 294, q,
       392, e, 294, e, 440, e, 294, e,
       466, e, 440, s, 392, s, 440, e, 294, e,
       392, e, 294, s, 392, s, 440, e, 294, s, 440, s,
       466, e, 440, s, 392, s, 440, s, 294, s}
```

SOUND OUTPUT



31 notes, about 9.3 seconds

Program 30 · Big Program — Little Fugue

Each note length is worked out once and given a short name: `q` is a quarter note (600 ms at 100 BPM), `e` an eighth, `s` a sixteenth. Change `tempo` to speed the piece up or slow it down. A list may run over several lines, as long as each line ends with a comma.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

c	b	r	a	c	e	s	u	a	p	n	f	j
r	e	t	r	a	u	q	z	t	r	e	h	d
i	r	g	v	s	i	x	t	e	e	n	t	h
e	n	m	i	l	l	i	s	e	c	o	n	d
n	f	n	j	n	v	g	h	t	h	g	i	e
f	r	c	o	d	v	i	b	r	a	t	e	b
e	y	b	h	t	y	p	h	n	d	s	b	k
l	o	b	w	v	e	s	g	n	s	y	c	k
o	o	f	y	c	n	e	u	q	e	r	f	b
h	t	s	e	r	m	o	r	o	p	m	e	t
w	k	n	o	f	s	b	c	i	s	u	m	k
g	k	e	m	i	h	o	c	t	a	v	e	r
d	u	r	a	t	i	o	n	m	d	d	p	w
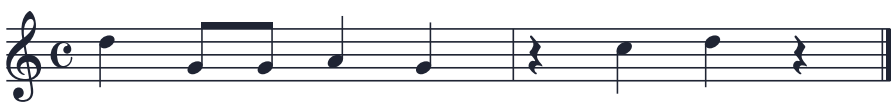
Find: braces, duration, eighth, frequency, hertz, millisecond, music, note, octave, quarter, rest, sixteenth, sound, tempo, vibrate, whole



Problems

Try these — they only use what this chapter covered.

1 Write a program using a single `sound` statement to play “Shave and a Haircut”. Remember to include the quarter rests in the second measure as a frequency of zero with the duration of a quarter note.

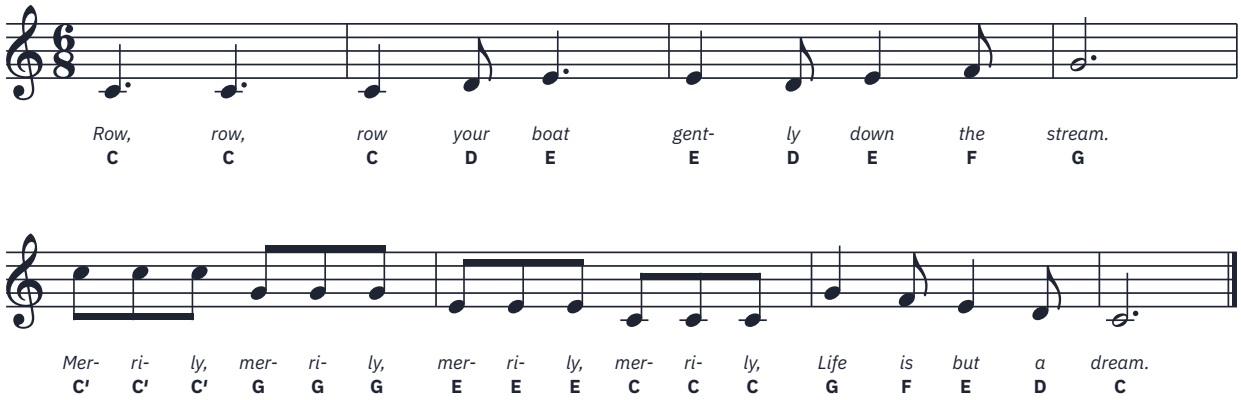


Shave	*and*	*a*	*hair*	*cut*		*two*	*bits*	
D	**G**	**G**	**A**	**G**	**rest**	**C**	**D**	**rest**
587	392	392	440	392	0	523	587	0

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

2 Type the `sound` statement below, and add variable assignments before it, to play “Row, Row, Row Your Boat”. The variables `c`, `d`, `e`, `f`, `g` and `cc` hold the frequencies of the notes (C′ in the music is `cc`, the C an octave higher). `n4` holds the length of a quarter note in milliseconds; `n2` is twice `n4` and `n8` is half of it.

```
sound {c,n4+n8, c,n4+n8, c,n4, d,n8, e,n4+n8,
  e,n4, d,n8, e,n4, f,n8, g,n2+n4, cc,n8, cc,n8,
  cc,n8, g,n8, g,n8, g,n8, e,n8, e,n8, e,n8, c,n8,
  c,n8, c,n8, g,n4, f,n8, e,n4, d,n8, c,n2+n4}
```



3 In Program 29, what value of `bpm` plays “Charge!” twice as fast? And twice as slow? Work out how long the whole tune lasts at 120 BPM, then check your answer by timing it.

4 Make a siren: in one `sound` statement, switch between 650 Hz and 900 Hz, 300 ms each, four times over. Then try it with a short rest after every note.

5 Rewrite Program 27 with note names instead of numbers. Hint: 233 Hz is `"A3#"` and 466 Hz is `"A4#"`. Does it sound the same?

Chapter 4 · Sound and Music. Modernised layout of the BASIC-256 beginner’s guide, updated for BASIC-256 2.3: `sound` accepts any frequency, notes can be given by name (`"C4"`), and `sound` is blocking — its non-blocking twin `soundplay` follows in a later chapter. Program numbers continue from Chapter 3, the note-length table is rounded to the nearest millisecond, and “Row, Row, Row Your Boat” now ends E–D–C. Every program in this chapter was run in BASIC-256 2.3. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Music set in Bravura (SIL Open Font License). Mascot: BitBot.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout