C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

Course topics

Programming topics	Music/audio topics
Working in real time	Oscillators
Buffers and arrays	Samples
Parameter control	Wavetables
Classes and objects	Control voltages
Analog and digital I/O	Gates and triggers
Filtering	Filters
Timing in real time	Metronomes and clocks
Circular buffers	Delays and delay-based effects
State machines	Envelopes
MIDI	ADSR
Block-based processing	MIDI
Threads	Additive synthesis
Fixed point arithmetic	Phase vocoders
ARM assembly language	Impulse reverb

Today

Lecture 9: Timing

What you'll learn today:

Counting time with sampled audio
Creating delays in real time
Events and intervals

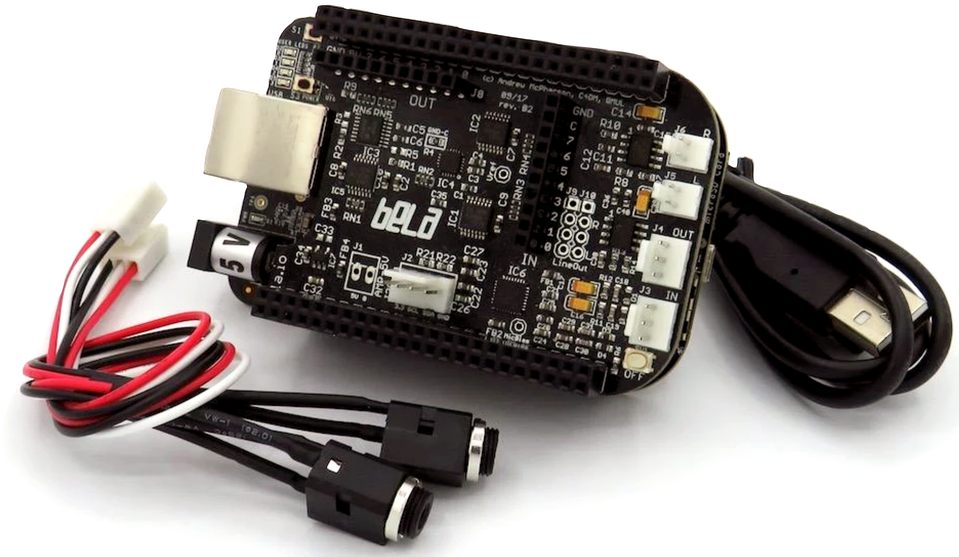
What you'll make today:

A metronome and a step sequencer

Companion materials:

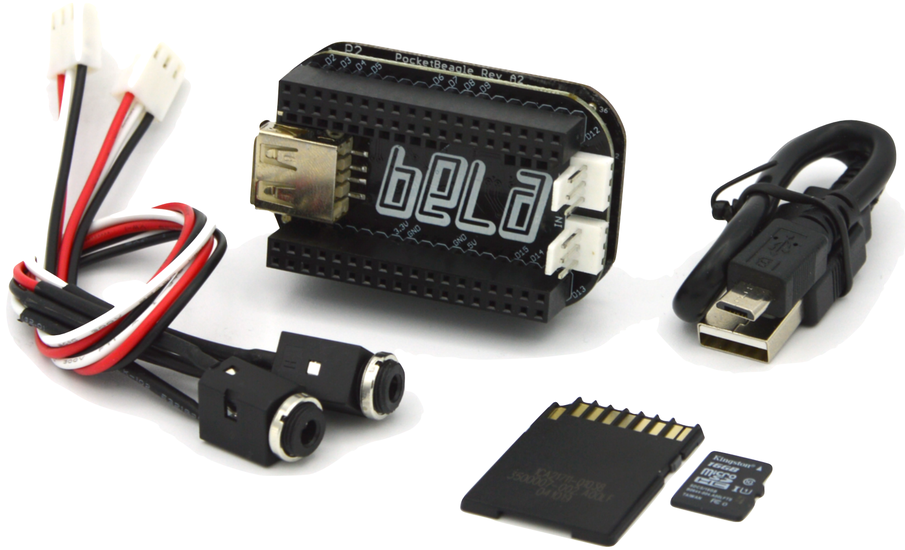
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

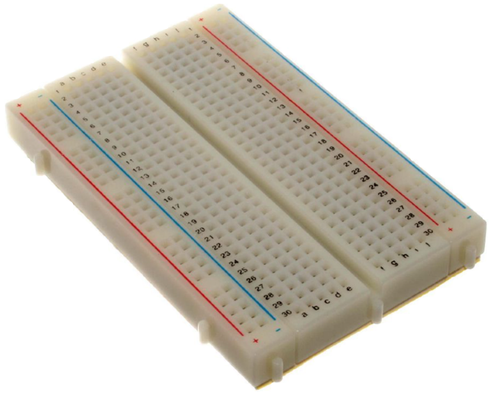
or

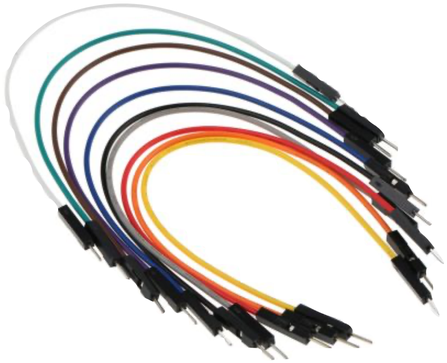


Bela Mini Starter Kit

[shop.bela.io]

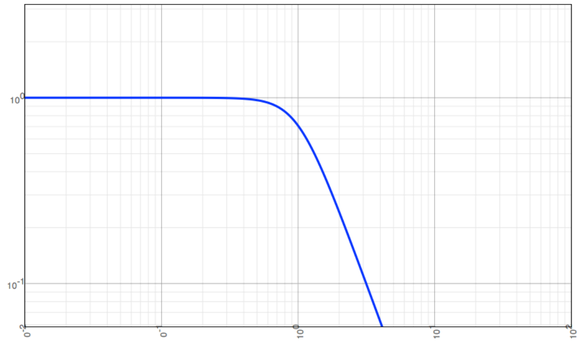
Optional for this lecture:



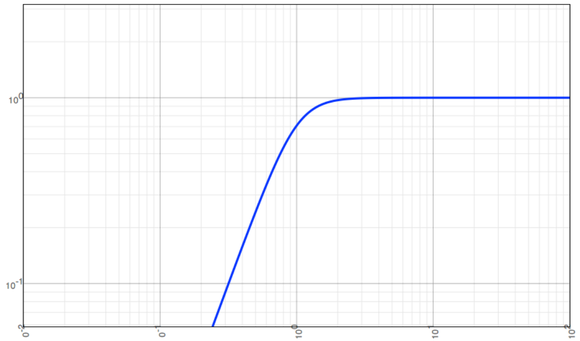


Review: Filters

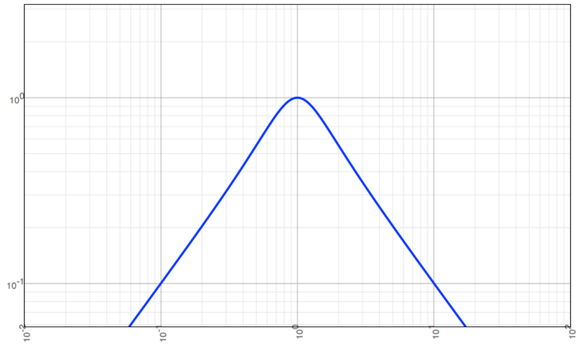
- Last lecture, we looked at creating digital filters
  - Many types of filters used in audio:



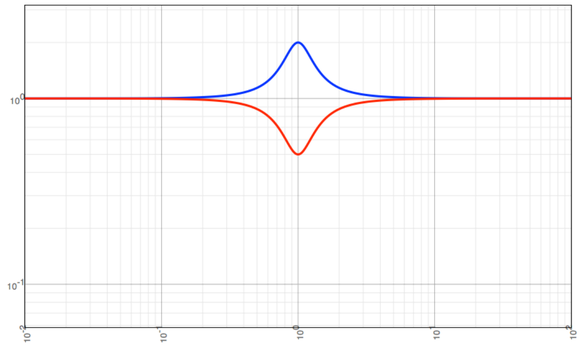
Lowpass



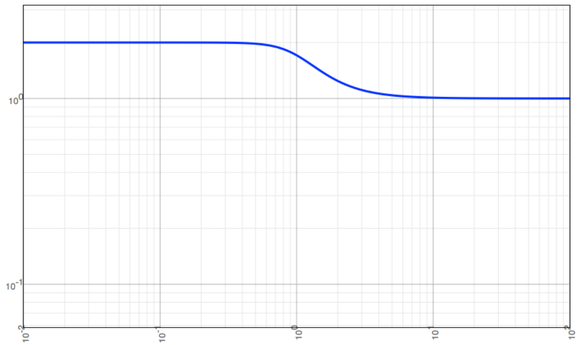
Highpass



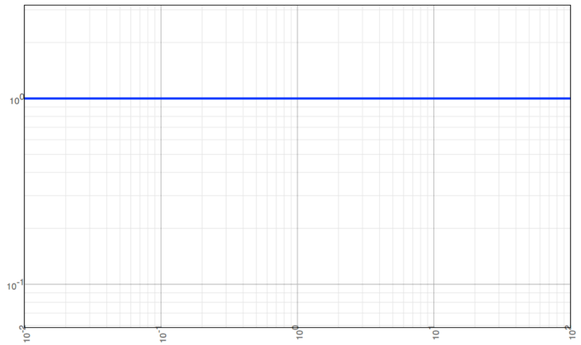
Bandpass



Peak / Notch



Shelving

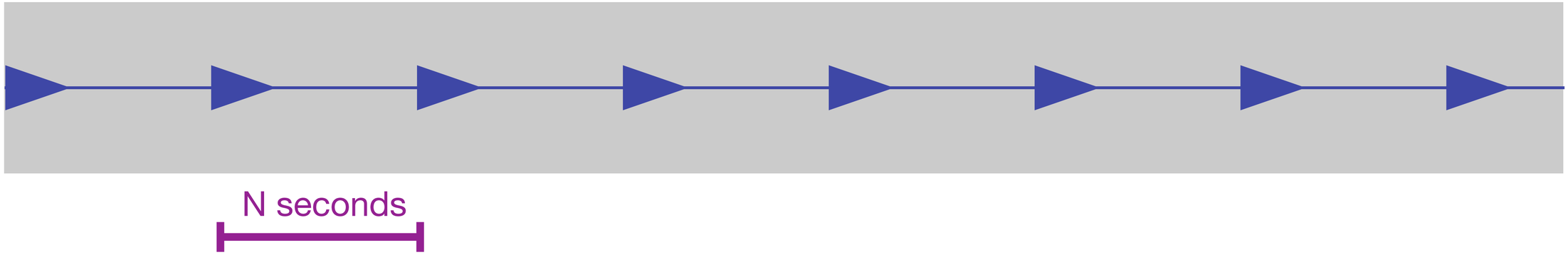


Allpass

- Filters were calculated as a weighted sum of previous inputs and outputs
  - A filter's frequency and phase response is specified by its coefficients

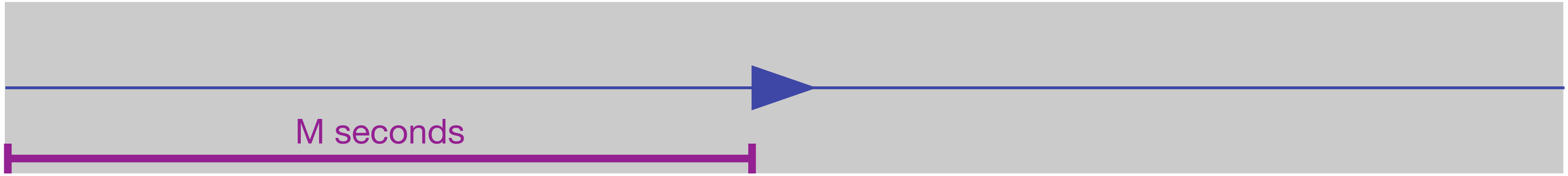
Handling time

- Filters are linear time-invariant (LTI) systems
  - Linearity means that filtering the sum of two signals is the same as filtering each signal independently and adding the outputs: $f(x_1 + x_2) = f(x_1) + f(x_2)$
  - Time invariance means:
    - A time shift in the input signal produces a corresponding time shift in the output signal
    - Effectively, the system always behaves in the same way for all time
  - Typically, we only think about time in an implicit way in implementing these systems
    - For example, keeping track of previous inputs and outputs to calculate a filter
- In practice, we often have to explicitly deal with time in signal processing
  - For example: “generate a signal every N seconds”:

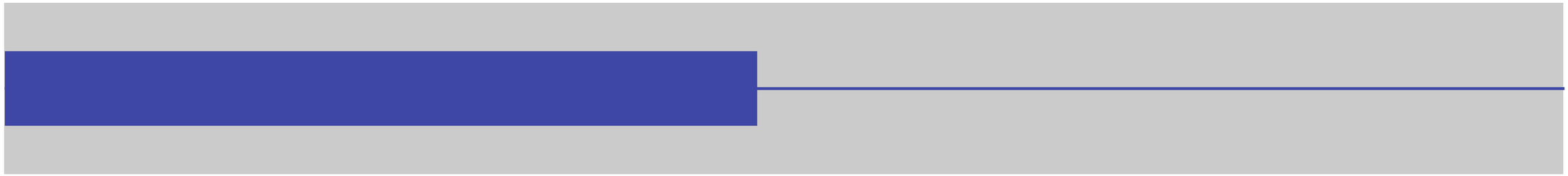


Handling time

- Other examples of working with time in signal processing:
  - “Wait M seconds and then generate a signal”:



  - “Generate a signal for the next M seconds and then stop”:

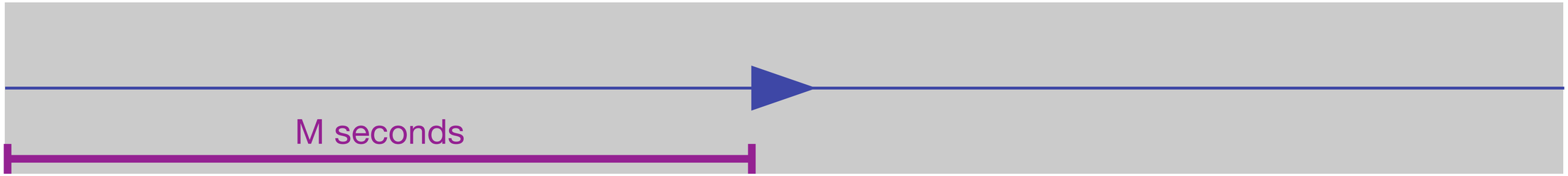


  - “Measure the time difference T between events A and B in an input signal”:

Timing, in real time

- The question is how we keep track of time in a real-time system
- Consider the case of generating a signal after a delay:

  “Wait M seconds and then generate a signal”



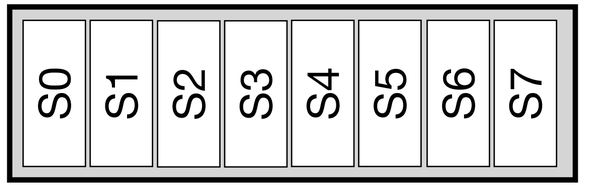
- Does this (pseudocode) solution work? **No.** (why not?)

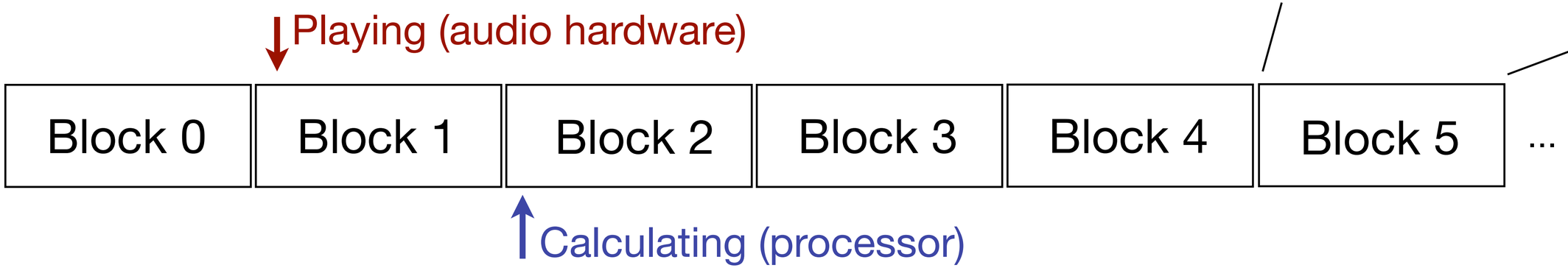
```
void render(BelaContext *context, void *userData)
{
    delay(5.0);        // Wait 5 seconds

    for (unsigned int n = 0; n < context->audioFrames; ++n)
    {
        // Generate my signal 5 seconds later
    }
}
```

Review: block-based processing

- A common real-time approach: process in blocks of several samples
  - Generate enough samples at a time to get through the next few milliseconds
  - Typical block sizes on GPOSes: 32 to 512 samples
    - Usually a power of 2 for reasons of driver efficiency
  - Typical block sizes on Bela: 2 to 32 samples
    - Default is 16; can be changed in the Settings tab of the IDE
    - Bela runs the audio code in hard real time, so we can make stronger timing guarantees
- While the audio hardware is busy playing one block, our code can start calculating the next one so it's ready on time:

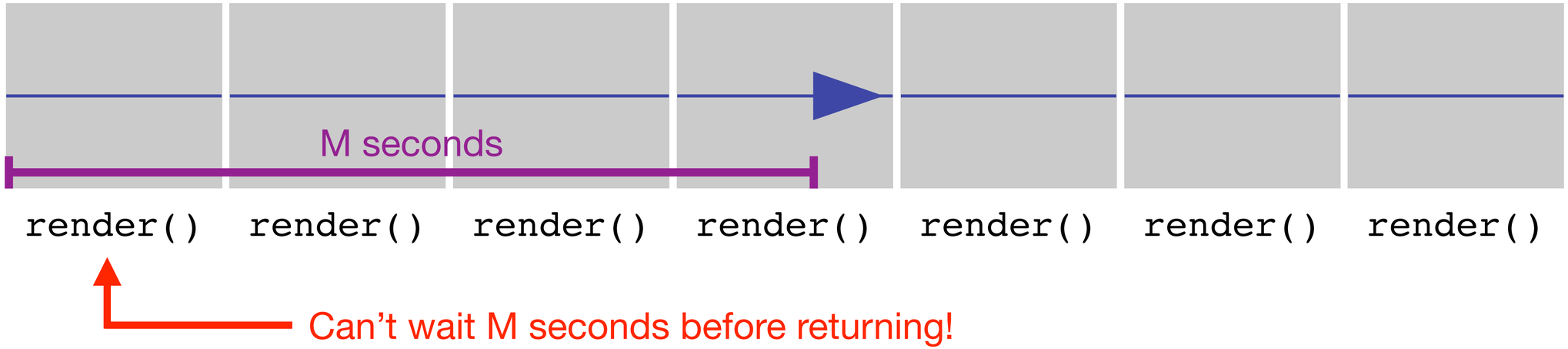
each block contains several samples





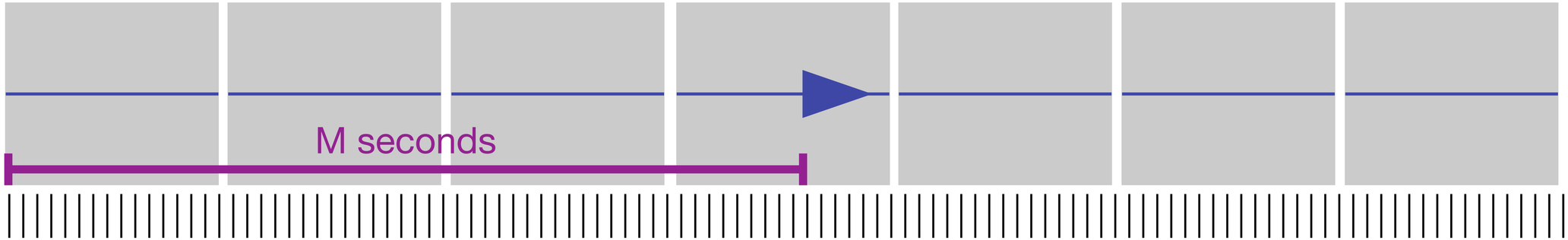
Timing, in real time

- We are always processing signals one block at a time
  - Our `render()` function is only responsible for what happens in the next block
  - We have strict timing limitations on how long `render()` can take to finish
  - If we miss our deadlines, we don't get a delay, we get an underrun



- The time when the computation happens is not the same as the signal time
  - By the time `render()` is called, the input was already captured
  - On the other hand, the output signals we generate don't appear until the next block

Timing without delay



- Alternative to delay: count elapsed time as we go along
  - Perhaps we want something like the `millis()` function on Arduino, which gives the number of milliseconds since the board powered up
  - There are system calls on Linux that give us the clock time, though many of these are not safe to run in Xenomai primary mode (i.e. they would cause mode switches on Bela)
- What is a simpler way to keep track of elapsed time within a digital signal?
  - Count samples!
  - (elapsed time in seconds) = (elapsed samples) * (sample rate)
  - This also has the advantage of telling us the signal time rather than the computation time

Counting samples

- To schedule an event, we need to know how many samples in the future it should take place
  - For example, at 44.1kHz sample rate, how many samples for 0.1 seconds? **4410**
- The event might well be in a future block
  - Therefore, we need to use a global variable
- Two possible (equivalent) approaches:
  1. Start the variable at 0 at the beginning of interval, increment it each sample until it reaches the desired delay time (in samples)
  2. Start the variable at the desired delay time (in samples), decrement it each sample until it reaches 0
- Task: using the metronome example
  - Play the sound file every 0.5 seconds by counting samples (use a new variable)
  - To start the sound playing, call `gPlayer.trigger()`

Metronome code

```
// [other global variables omitted]
unsigned int gMetronomeInterval = 22050; // Interval between events
unsigned int gMetronomeCounter = 0;      // Number of elapsed samples

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        if(++gMetronomeCounter >= gMetronomeInterval) {
            // Counter reached the target
            gMetronomeCounter = 0;          // Reset the counter
            gPlayer.trigger();              // Start a new tick
        }

        // Get the next sample of the file if it's playing
        float out = gAmplitude * gPlayer.process();

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

Increment counter and check if we've reached the interval

Resetting the counter starts the timing over again

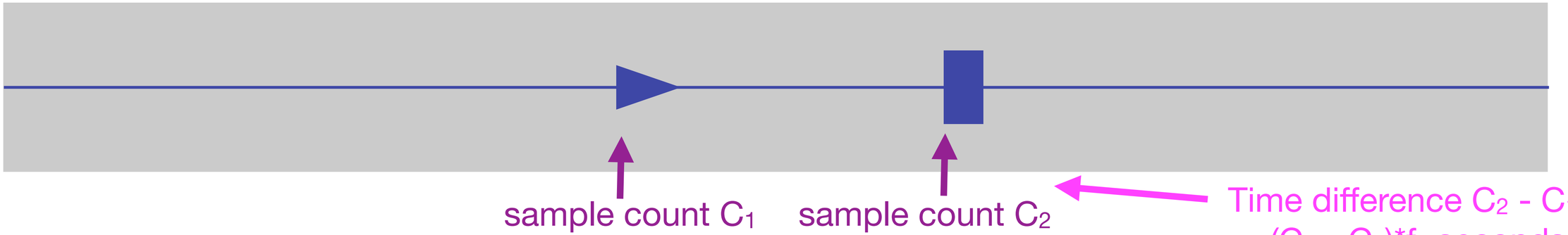
Calling `gPlayer.trigger()` resets the read pointer inside `MonoFilePlayer`

Metronome code (alternative version)

```
// [other global variables omitted]
unsigned int gMetronomeInterval = 22050;               // Interval between events
unsigned int gMetronomeCounter = gMetronomeInterval;  // Number of remaining samples

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        if(--gMetronomeCounter == 0) {
            // Counter reached the target
            gMetronomeCounter = gMetronomeInterval; // Reset the counter
            gPlayer.trigger();                      // Start a new tick
        }

        // Get the next sample of the file if it's playing
        float out = gAmplitude * gPlayer.process();

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

Counting samples on Bela

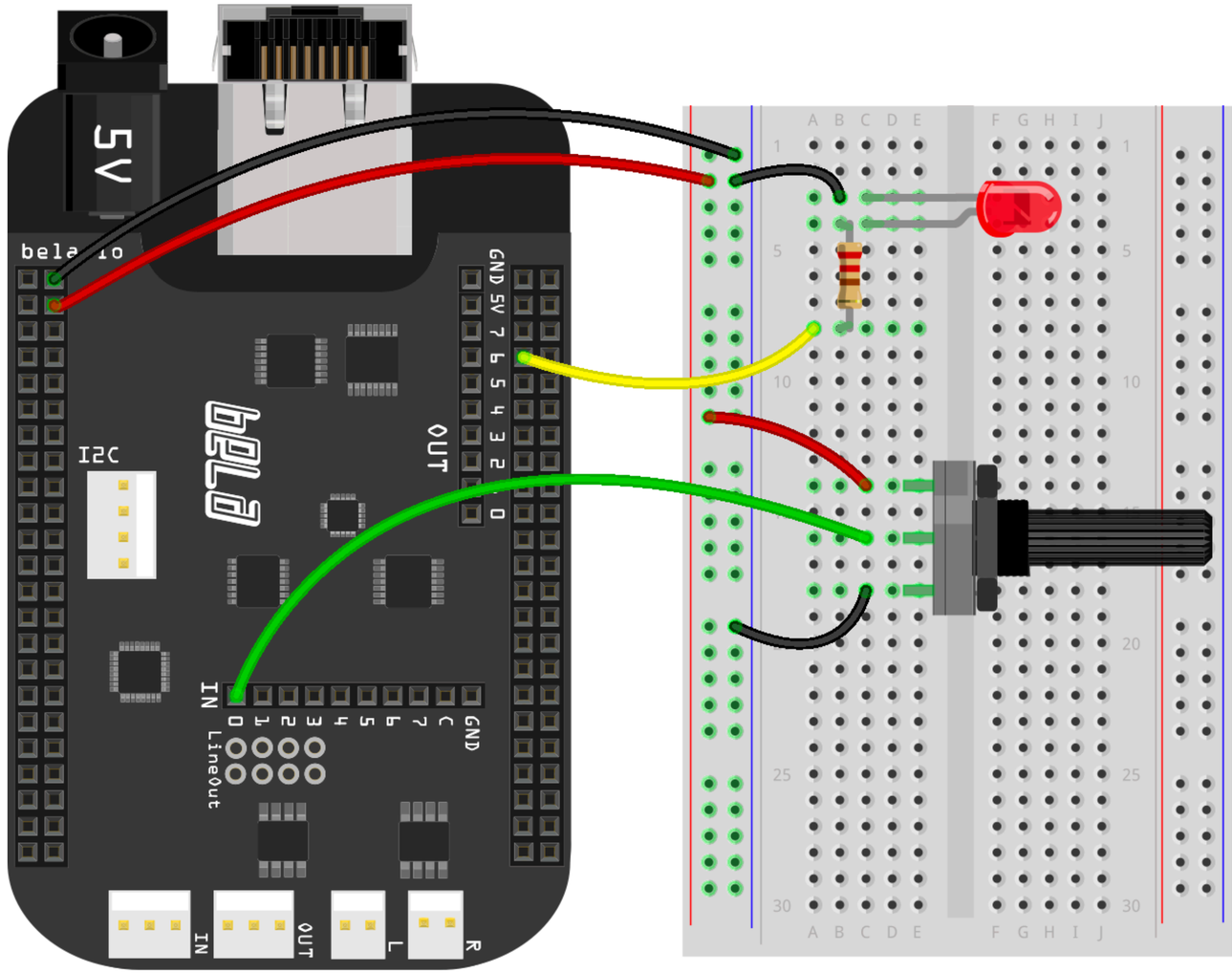
- Bela provides a cumulative number of samples since the program began:
  - `context->audioFramesElapsed`
  - This is a variable of type `uint64_t` (64-bit unsigned int)
    - It will roll over every 2^{64} samples (once every 13.2 million years!)
    - By contrast, a 32-bit value would roll over every 1.1 days
  - `audioFramesElapsed` is updated at the beginning of every block
    - To find the total number of samples for a sample within the block, use `(audioFramesElapsed + n)`
- In practice, we are often more interested in relative times between events



  - Save the time of the first event, then look at the difference in time
  - Alternatively (like the last exercise), reset the counter at the first event

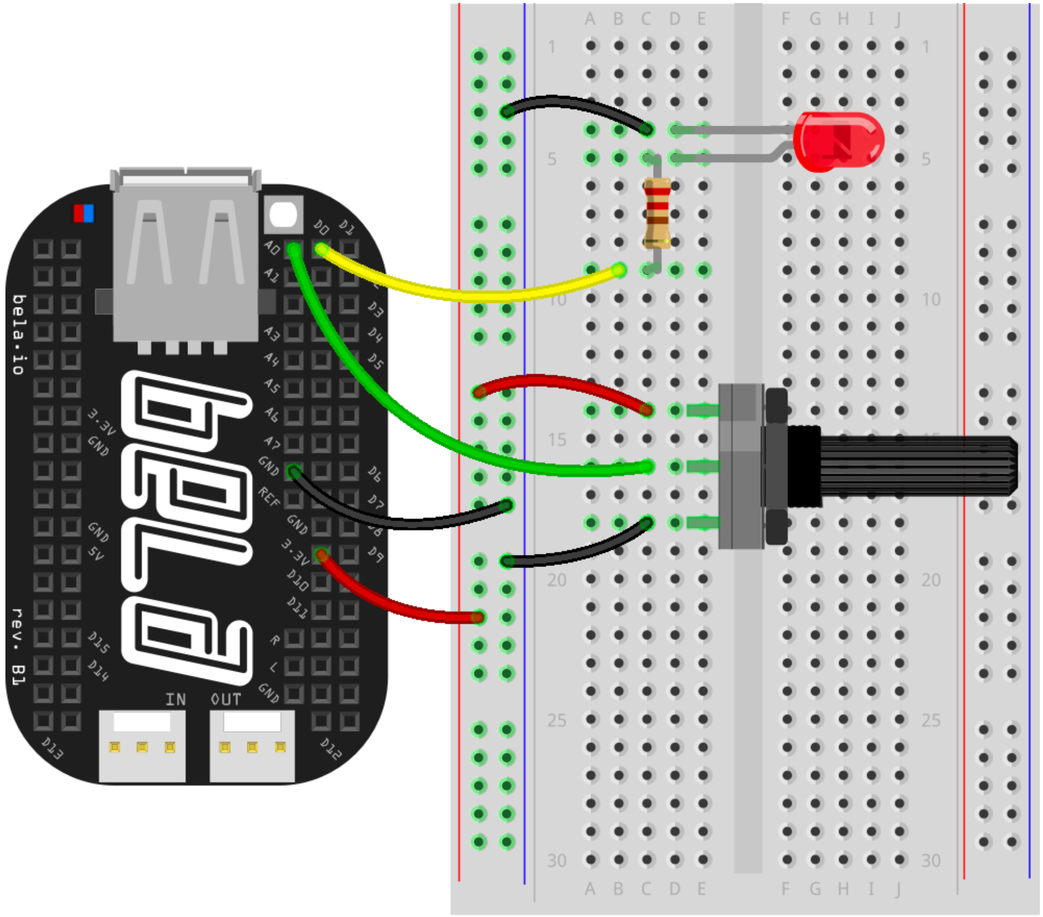
Metronome tasks

- How to convert beats per minute (BPM) to an interval in samples?
  - Suppose the tempo is *X* BPM
  - How many minutes per beat? 1/X
  - How many seconds per beat? 60/X
  - How many samples per beat? $60*f_s/X$
- Task: make the metronome speed adjustable with a potentiometer
  - Reminder: outer leads to +3.3V and ground; middle lead to analog input
  - Use `map()` to scale the interval between 40 and 208 BPM



Metronome tasks

- How to convert beats per minute (BPM) to an interval in samples?
  - Suppose the tempo is *X* BPM
  - How many minutes per beat? 1/X
  - How many seconds per beat? 60/X
  - How many samples per beat? $60*f_s/X$
- Task: make the metronome speed adjustable with a potentiometer
  - Reminder: outer leads to +3.3V and ground; middle lead to analog input
  - Use `map()` to scale the interval between 40 and 208 BPM



Tempo-adjustable metronome code

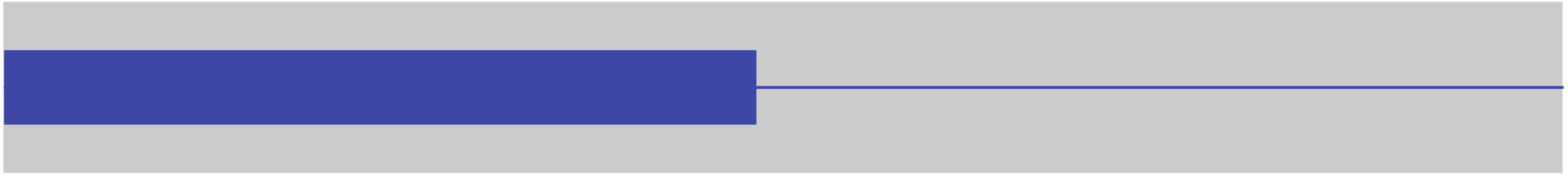
```
unsigned int gMetronomeCounter = 0;          // The current metronome count
unsigned int gMetronomeInterval = 0;         // How long between metronome ticks?

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Read the analog input to get the current tempo
        float input = analogRead(context, n/2, kInputTempo);
        float bpm = map(input, 0, 3.3/4.096, 40, 208);
        gMetronomeInterval = 60.0 * context->audioSampleRate / bpm;

        // Check if enough time has elapsed
        if(++gMetronomeCounter >= gMetronomeInterval) {
            // Reset the counter and trigger the sample
            gMetronomeCounter = 0;
            gPlayer.trigger();
        }

        // Get the next sample of the file if it's playing (out = 0 otherwise)
        float out = gAmplitude * gPlayer.process();

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

Notice: index of analog frame is **n/2** since analog is half the sample rate (verify this in `setup()`)

BPM to samples

Intervals

"Generate a signal for the next M seconds and then stop"

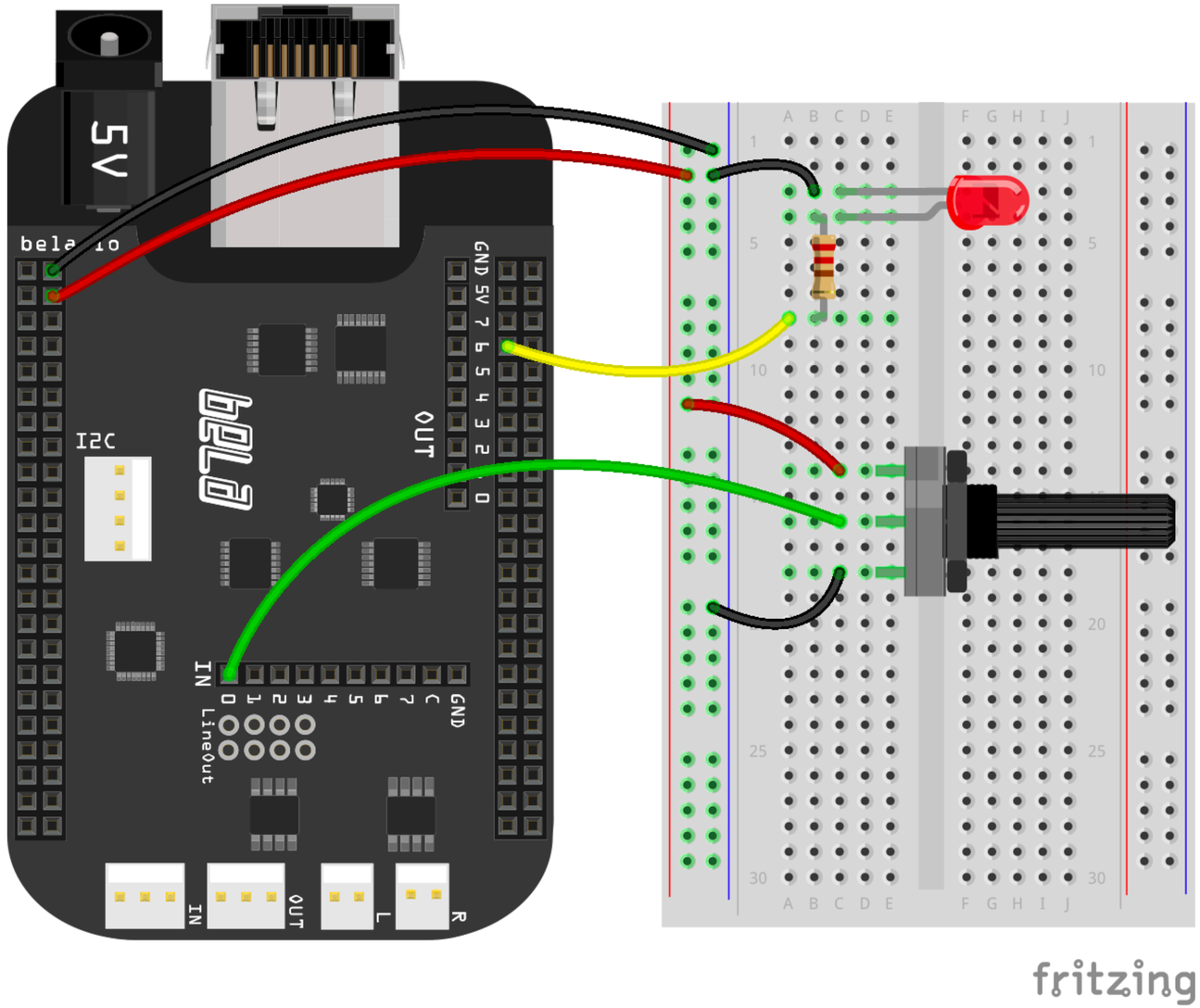


- The previous examples looked at a discrete trigger after a period of time
- What about having a signal persist for a certain interval?
  - For example, how do we make a pulse that lasts 0.1 seconds?
- Same idea as before: count samples until a certain number elapse
  - This time, instead of having a condition occur once when the count elapses, make the condition true every sample *until* the count elapses
  - e.g. inside the `for()` loop:

```
counter++;
if(counter < interval) {
    // Generate a signal
}
```

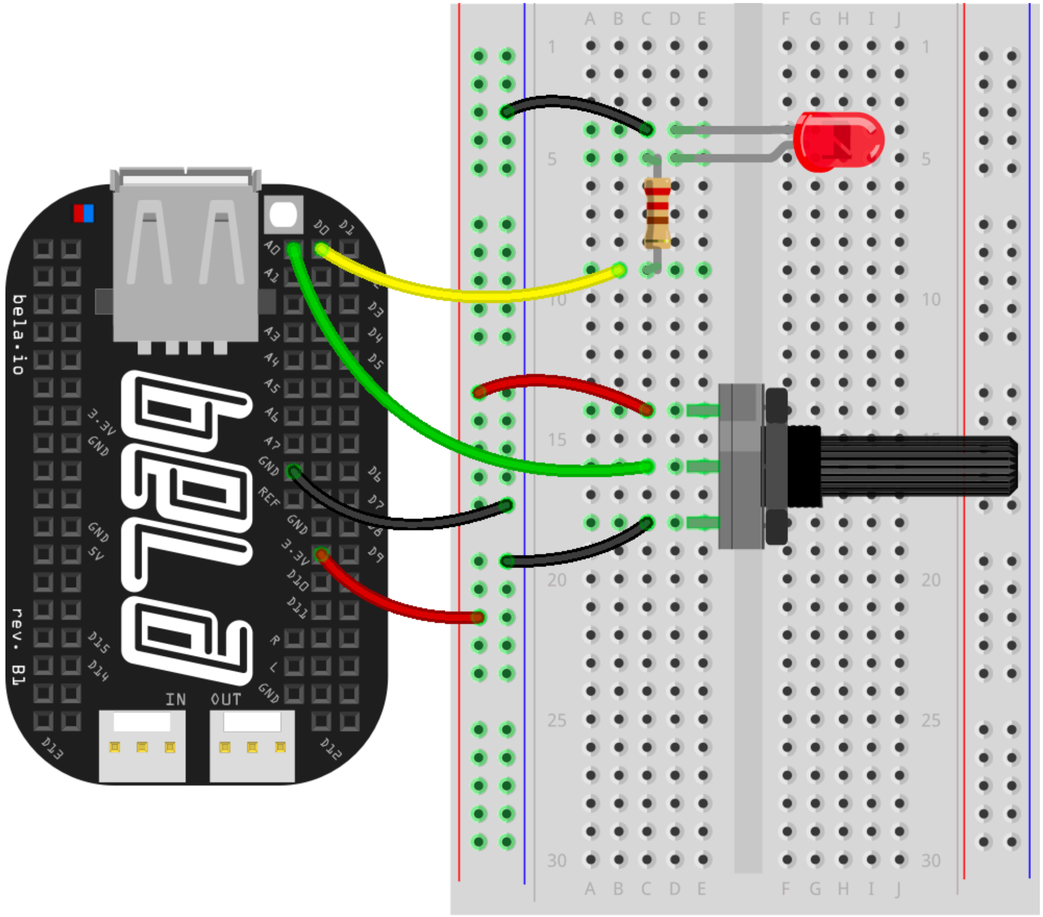
Interval task

- Task: using the metronome example
  - Add a blinking LED on each tick
  - The LED should be on for the first 50 milliseconds after the tick (regardless of metronome speed)
  - Count samples and use `digitalWrite()` to set the LED on or off for each sample
  - Hint: you can use the same variable to time the LED blink and to wait for the next tick



Interval task

- Task: using the metronome example
  - Add a blinking LED on each tick
  - The LED should be on for the first 50 milliseconds after the tick (regardless of metronome speed)
  - Count samples and use `digitalWrite()` to set the LED on or off for each sample
  - Hint: you can use the same variable to time the LED blink and to wait for the next tick



Metronome with LED

```
// [other global variables omitted]
const unsigned int kLEDPin = 0;           // Pin number for LED
unsigned int gMetronomeCounter = 0;       // Number of elapsed samples
unsigned int gMetronomeInterval = 22050; // Interval between events [actually calculated in render()]
unsigned int gLEDInterval = 2205;         // How many samples for the LED to be on [calc. in setup()]

// Don't forget pinMode() inside of setup()!

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // [tempo adjustment code omitted]
        if(++gMetronomeCounter >= gMetronomeInterval) {
            gMetronomeCounter = 0;
            gPlayer.trigger();
        }

        if(gMetronomeCounter < gLEDInterval) {
            digitalWriteOnce(context, n, kLEDPin, HIGH);
        }
        else {
            digitalWriteOnce(context, n, kLEDPin, LOW);
        }
        // [audio output code omitted]
    }
}
```

Check if counter is within the LED blink interval

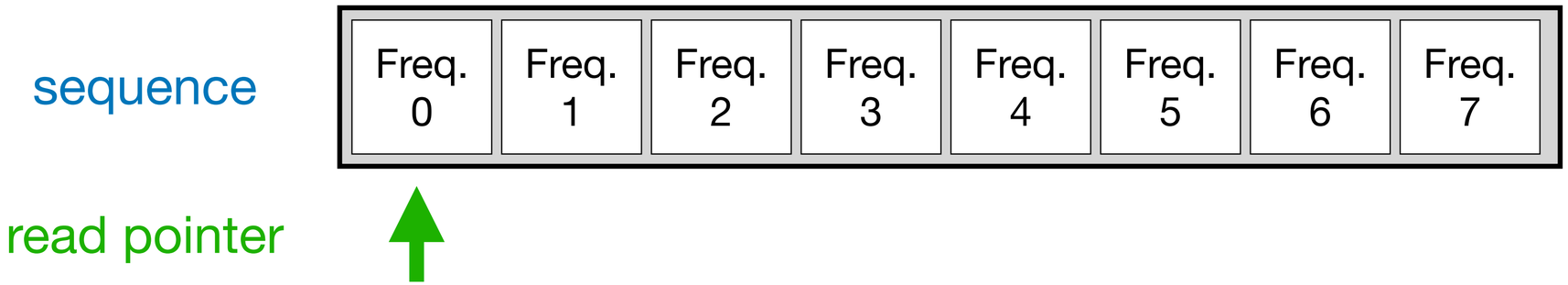
Review: step sequencer

- A step sequencer plays a set pattern of notes in a loop
  - Step sequencers commonly store patterns of 8 or 16 notes
  - Every time it receives a trigger, it advances to the next note
- For now, let's focus just on controlling pitch (frequency)
  - We could implement a step sequencer like this:

  - We store 8 frequencies in a buffer, and the read pointer keeps track of which one we are currently reading. That frequency then controls an oscillator.
  - Very similar to playing a wavetable!
  - The difference is that we move the read pointer only when we receive a trigger

Step sequencer task

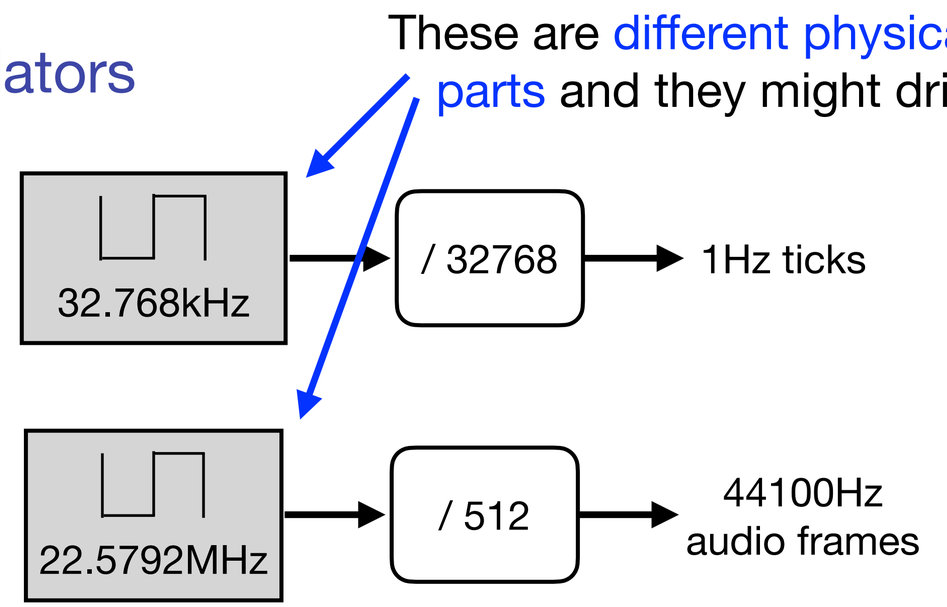
- For a step sequencer, our buffer holds frequencies (or more generally, CV values)
  - We move the read pointer only when we receive a trigger



- In Lecture 7, we used a button press to create a trigger
  - We looked for a falling edge (high to low transition)
- Task: in the step-sequencer-metronome example,
  - Count samples to generate a trigger at a regular interval
  - When the counter reaches the correct interval, advance the read pointer, looping as needed
  - Implement the same code as before to blink the LED each step

Timing limitations

- Counting samples is a (mostly) great way to measure elapsed time
  - (elapsed time in seconds) = (elapsed samples) * (sample rate)
  - But this depends on an absolutely precise sample rate!
- On many systems, the system clock (wall time) is not precisely synchronised with the audio frame clock (sample rate)
  - They can be clocked from different physical oscillators
  - Even where they are ultimately derived from the same oscillator, the system clock might periodically update from an external time server
  - A similar issue applies to connecting two audio devices together: might have different frame clocks that drift from each other
  - The effect is typically small
    - e.g. 20ppm = .002% = about ±1 sample/sec at 44100Hz



Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources