C++ Real-Time Audio Programming with Bela

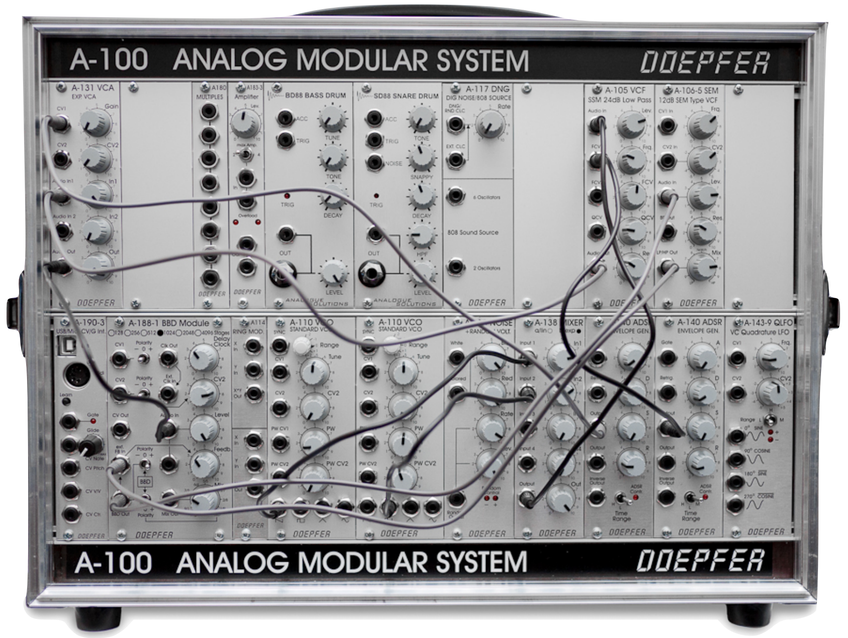
Dr Andrew McPherson

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

Founder and Director, Bela

C++ Real-Time Audio Programming with Bela

Modular synthesis



C++

Embedded hardware

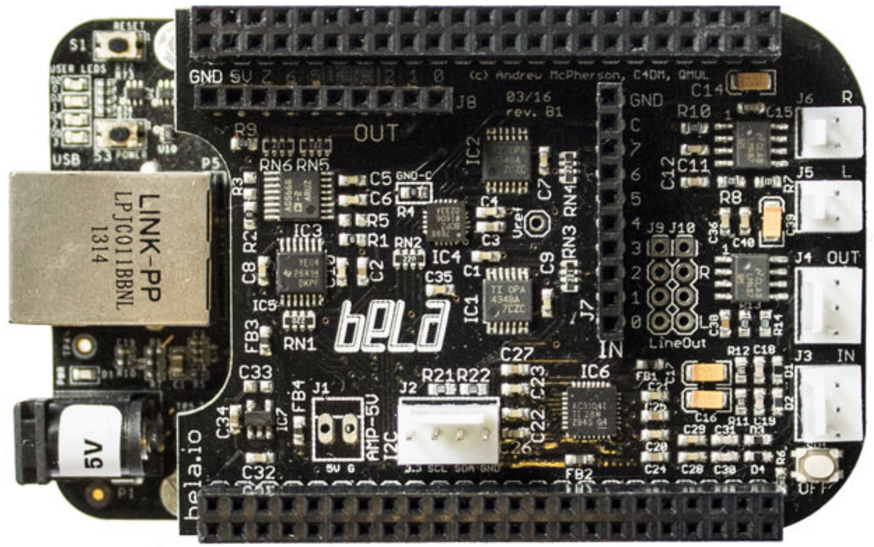


Image credit: Nina Richards, wikimedia (CC-BY 3.0)

The course in a nutshell

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

Lecture 1: Real time

What you'll learn today:

What is real-time audio?
How to write real-time C++ code

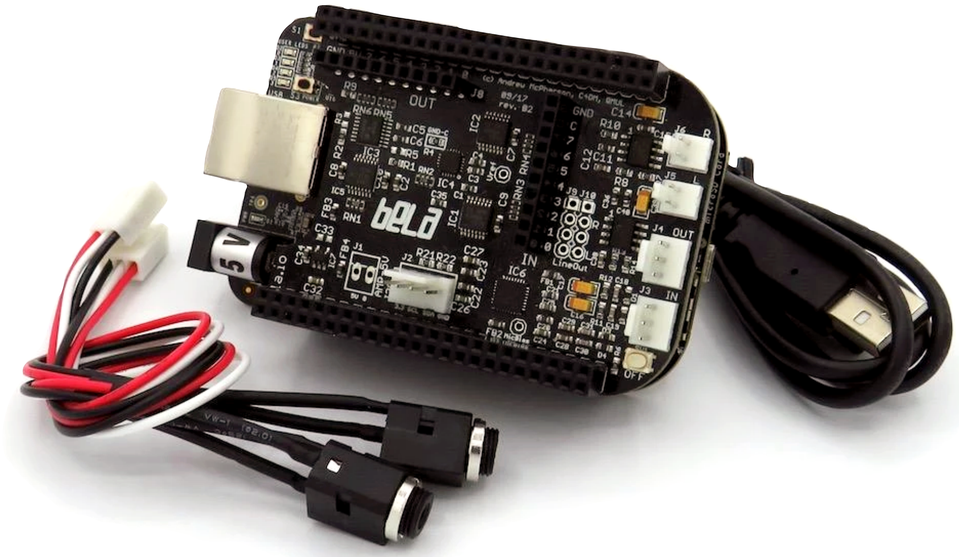
What you'll make today:

Sine-wave oscillator

Companion materials:

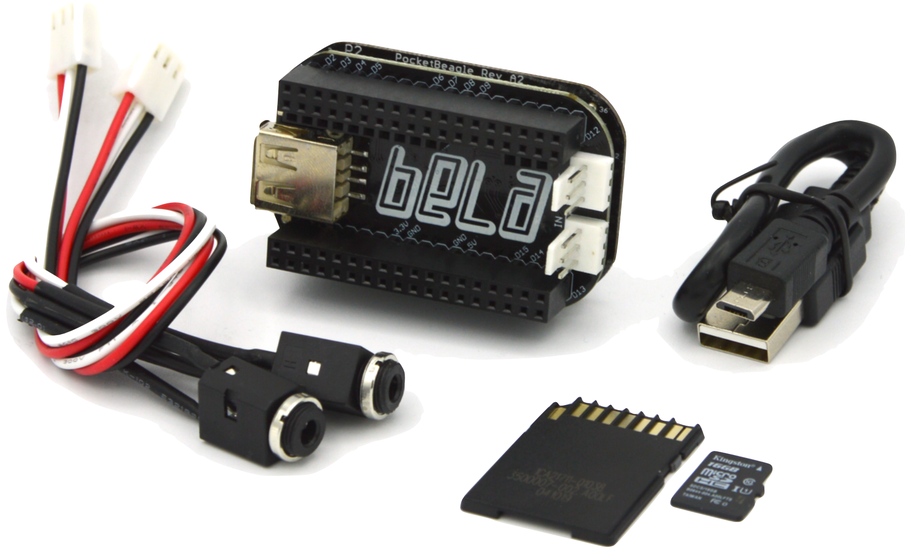
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

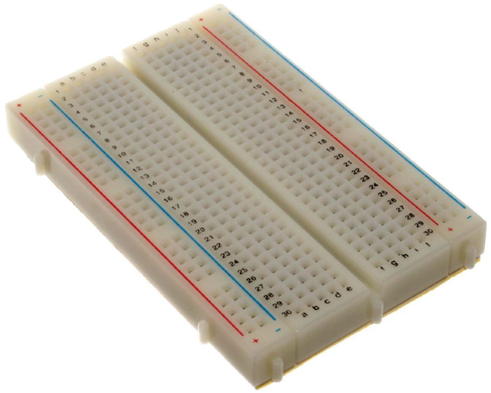
or

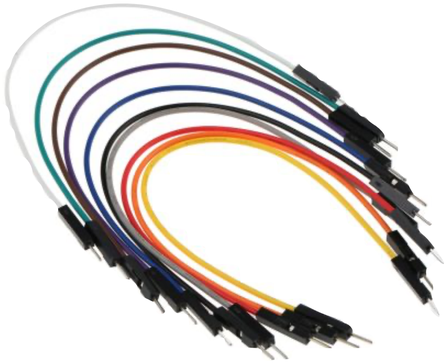


Bela Mini Starter Kit

[shop.bela.io]

Recommended for some lectures:

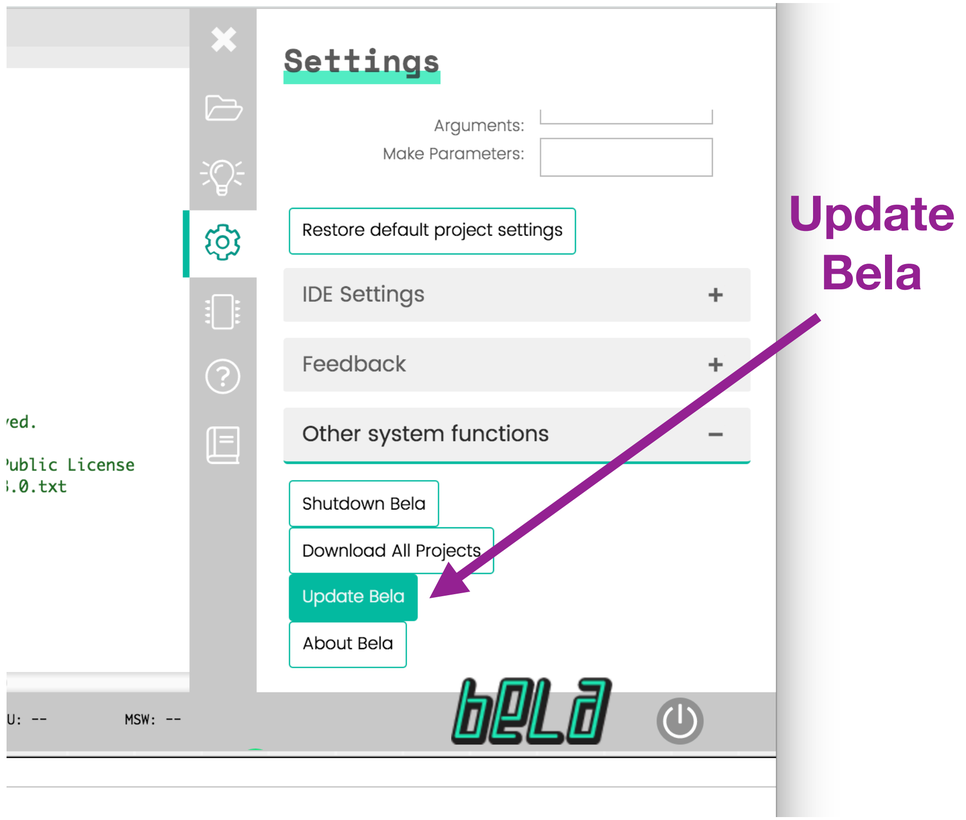




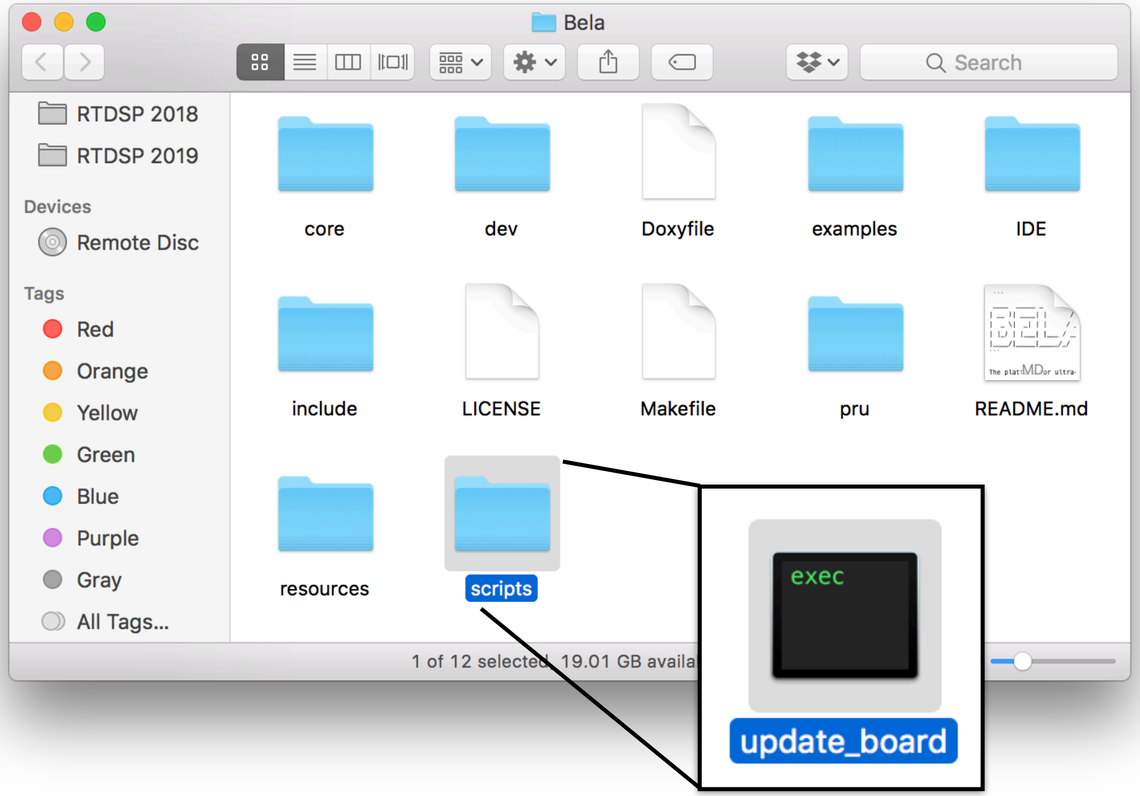
Updating Bela

- Download the latest Bela core software from: **learn.bela.io/update**
- Two ways to update the board:

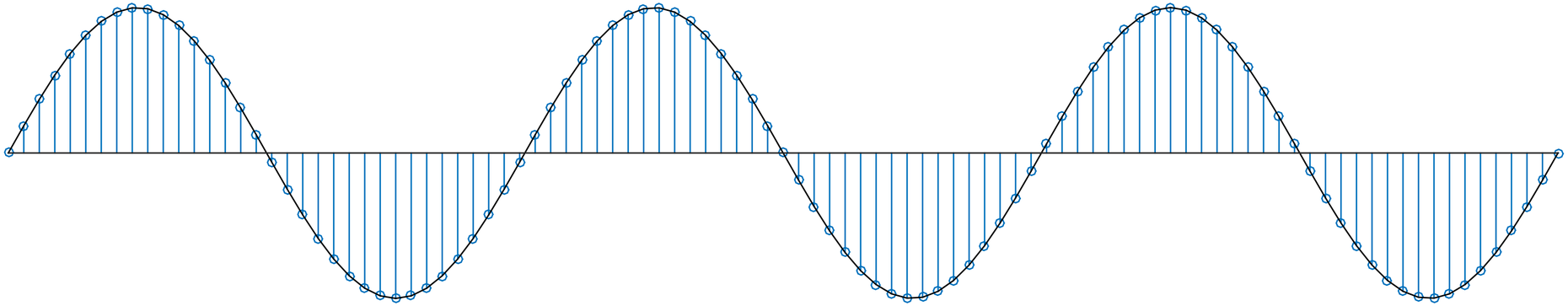
By IDE (recommended)



By Script (Mac and Linux)
unzip the archive



Basics of digital signals



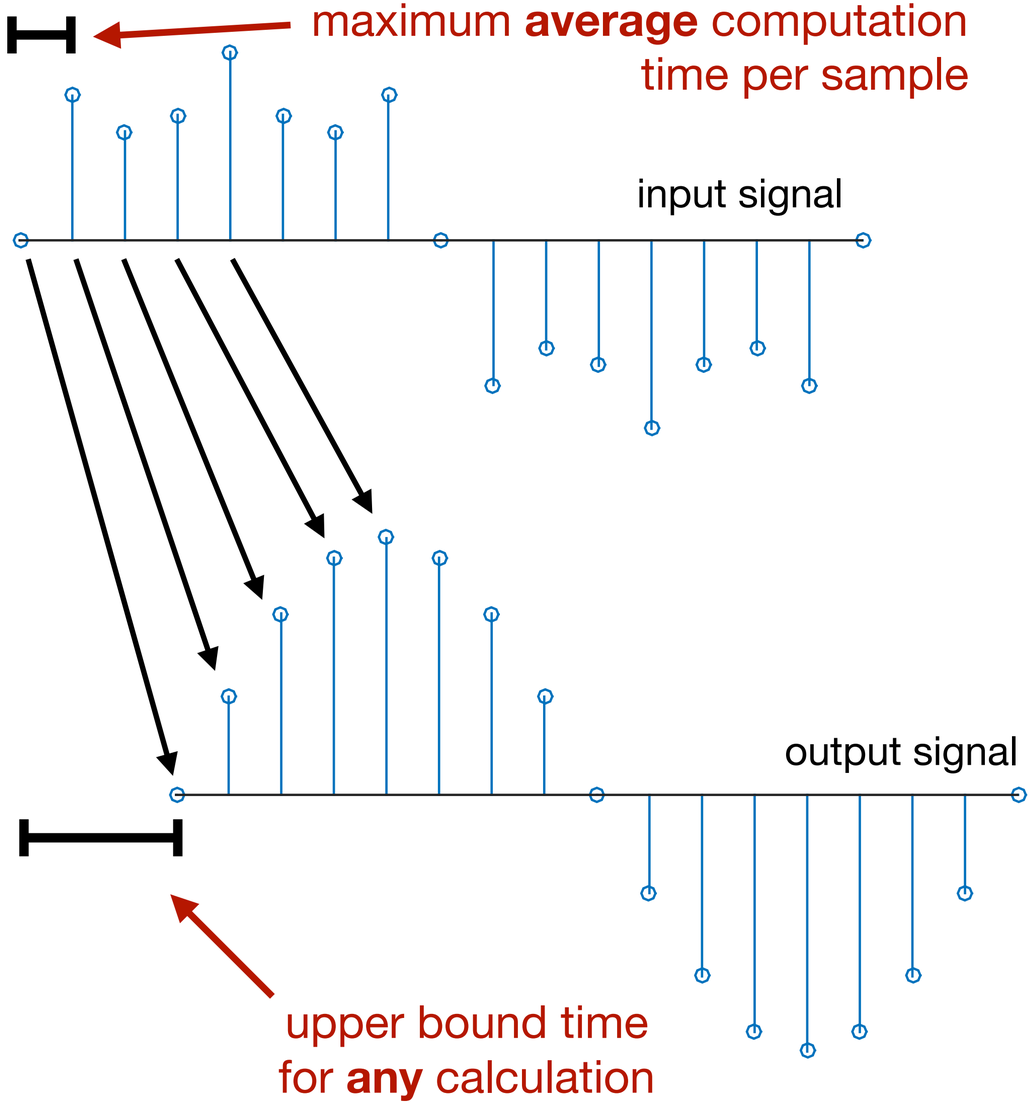
- A digital signal is composed of a series of samples (numbers) in discrete time
  - The signal $x[n]$ is defined for integer *n*: $n = ..., -2, -1, 0, 1, 2, ...$
  - $x[n]$ is undefined when *n* is not an integer
- In informal usage (especially for audio), digital signals are often considered as sampled versions of an analog, continuous-time signal
  - The signal $x(t)$ is a continuous-time signal defined for all real values of *t*
  - In the above example, we could write $x[n] = x(nT)$ where *T* is the sampling period
  - The inverse of the sampling period is the sample rate: $f_s = 1/T$

Real time

- This course covers **real-time** audio programming. What do we mean by that?
- Continuous processing of digital audio signals as they come in
  - Or equivalently, continuous generation of signals as needed
- Suppose $x[n]$ is a signal where the samples are arriving over time
  - $x[n] = x(nT)$ for some sampling period *T*, and these samples are arriving every *T* seconds
  - Then our system must be able to process $x[n]$ piece-by-piece as new samples arrive
- Contrast this to a non-real-time system (e.g. an audio editor)
  - Might have the luxury (or even the requirement!) of accessing the entire signal at once
- These are practical implementation considerations, not mathematical ones
  - DSP theory generally doesn't care what *n* "means" in the real world, or when/how the signal arrives to a computer
  - One exception: any real-time system must be causal (i.e. not depend on future values)

Real time

- Real-time performance depends on computation time
- Specifically: every computation must finish in bounded time
  - In a real-time system, results are only useful if they arrive on time
  - This upper bound could be large or small, but it must be consistent and guaranteed
- On average, processing one sample must take no longer than the sampling period
  - Otherwise, the delay between input and output would grow without bound



Real time: example

- What is this equation? $y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$

an IIR filter

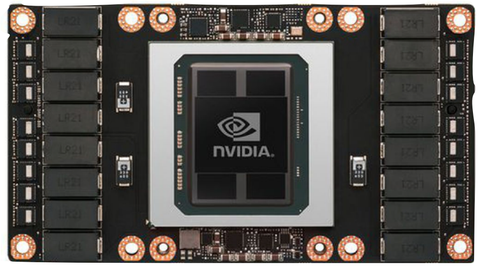
If you haven't seen this notation before, consult a DSP textbook or online tutorial.

- Suppose that the filter order $N = M = 8$. Is this filter real time?
  - That depends! Not enough information. How is it implemented?
- Suppose it takes our implementation 10ms to compute a single value of $y[n]$ for each new value of $x[n]$. Is it real time?
  - That depends! What more do we need to know?
  - What is the sample rate? Is the sampling period longer than 10ms?
    - In other words, is the sample rate lower than 100Hz?
    - We are assuming, as we saw earlier, that $x[n]$ corresponds to some real-world signal unfolding in time. Mathematically, digital signals don't need to correspond to values over time.
- Suppose $f_s = 50\text{Hz}$. Now is it real time?
  - It can be, but it still depends on how it's implemented!

Types of processors

- General-Purpose Processor (GPP)
  - Designed for a broad range of tasks
  - Found in computers and mobile devices
  - Very small, low power version called a microcontroller (MCU)

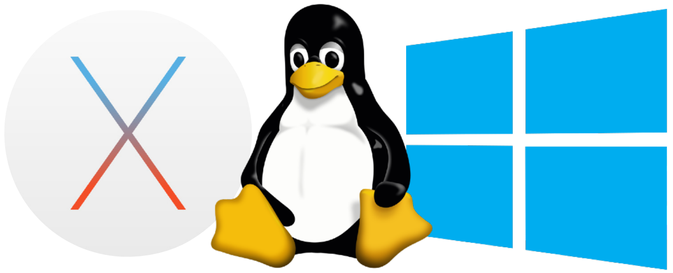
- Digital Signal Processor (DSP)
  - Optimised for signal processing maths
  - Good at fast, repeated computations
  - Found in dedicated hardware and as co-processors

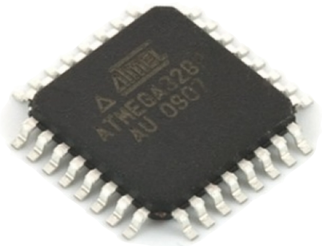


- Graphics Processing Unit (GPU)
  - Massively parallel processor cores
  - Good for high bandwidth data with little interdependence (e.g. video pixels)
  - Can be used for some audio processing tasks

- Field Programmable Gate Array (FPGA)
  - Function is encoded into the hardware
  - Extremely fast, but very little flexibility
  - Found in specialist applications and certain high-performance computing scenarios

Types of operating system





- General-Purpose Operating System (GPOS)
  - Designed for typical computers and mobile devices
  - Shares time between many simultaneous processes in a way that maximises overall system performance
  - Real-time processes can have higher priority, but timing guarantees are generally not possible
- Real-Time Operating System (RTOS)
  - Designed for mission critical systems and application-specific devices
  - Also allows simultaneous processes, but with strict scheduling rules
  - Can offer guarantees on reaction time to incoming data and events
- No Operating System (bare metal)
  - Typically found on small microcontrollers in single-function applications
  - Only one task required, so no scheduler needed and timing is guaranteed
  - Asynchronous events can still be handled with interrupts

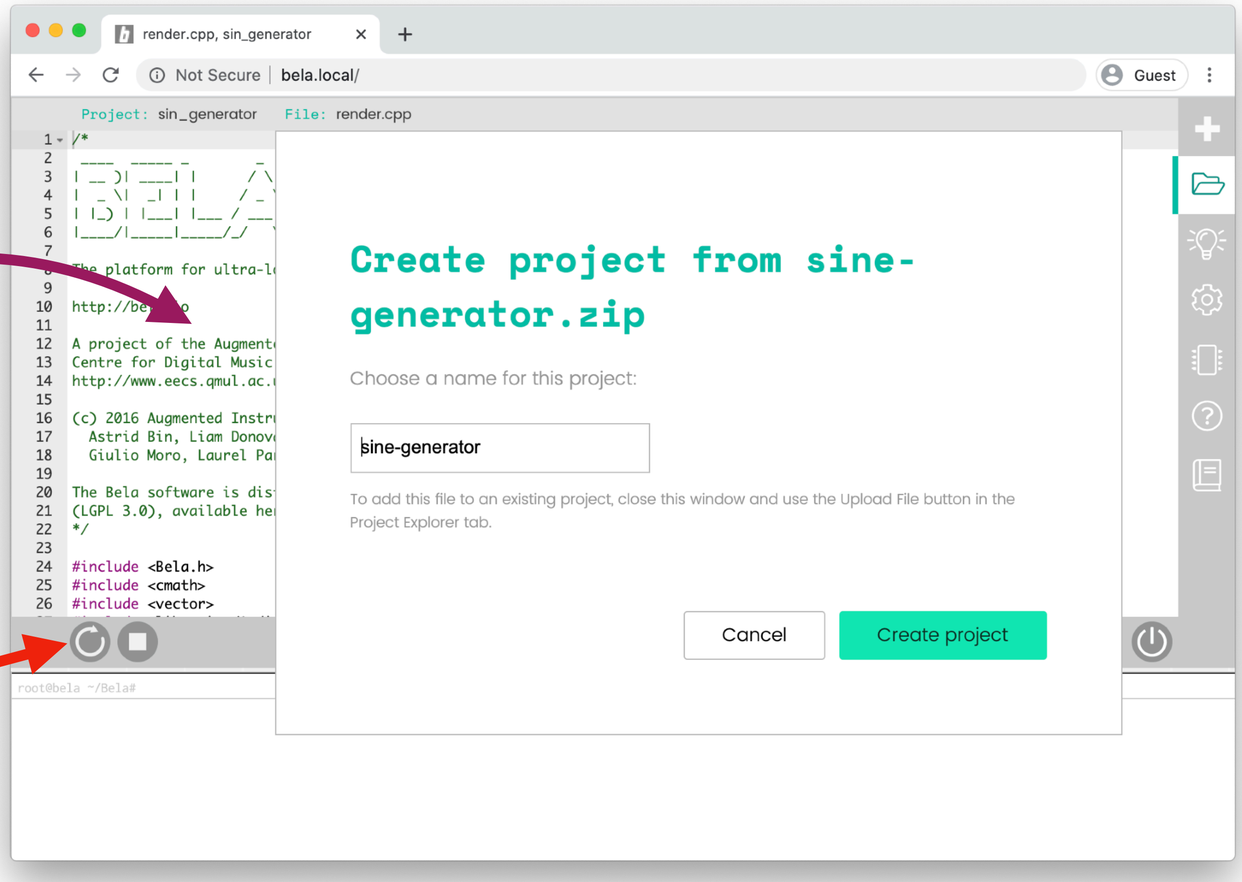
Categories of “real time”

- Hard real time (RTOS, bare metal)
  - Calculations are guaranteed to finish within a certain defined period of time
  - Delay from input to output is less than some fixed time *T* for every sample, **always**
  - Examples: aircraft control systems; medical devices
- Soft real time (GPOS)
  - Calculations most of the time finish within a certain defined period of time
  - For any fixed delay *T*, the system may take longer than *T* to respond to some samples (usually infrequent) because it cannot guarantee timing in all cases
  - Missed deadlines mean the calculation is useless and may degrade performance, but will not cause the whole system to fail catastrophically
  - Examples: streaming audio/video; video games

Uploading the sine generator project

1. Download the example `sine-generator.zip` from **github.com/BelaPlatform/bela-online-course**
2. Drag the `.zip` file onto the Bela IDE window

3. Bela will ask you to name the new project. *(If it doesn't, update to the latest software.)*
4. Click Run to start



First test program

- sine-generator does not generate audio in real time
  - Instead, it calculates the whole signal at once, saves it to a file, then plays it back
    - The audio playback itself is real time, but that's not our focus right now!

This runs once, to generate the whole sound

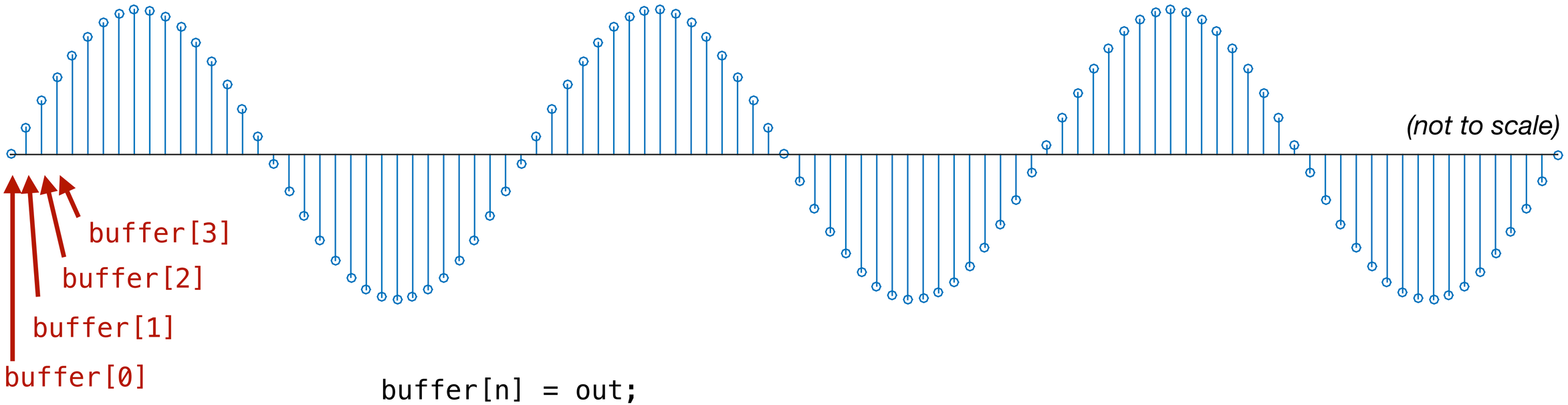
This runs numSamples times, to generate each sample

```
float gFrequency = 440.0; // Frequency of the sine wave in Hz
float gAmplitude = 0.6;   // Amplitude of the sine wave (1.0 is maximum)
float gDuration = 5.0;    // Length of file to generate

void calculate_sine(float *buffer, int numSamples, float sampleRate)
{
    // Generate the sine wave sample-by-sample for the whole duration
    for (int n = 0; n < numSamples; n++) {
        // Calculate one sample of the sine wave
        float out = gAmplitude * sin(2.0 * M_PI * n * gFrequency / sampleRate);

        // Store the sample in the buffer
        buffer[n] = out;
    }
}
```

This is an array holding the signal
The value in [ ] indicates the index (which element)

First test program

- sine-generator does not generate audio in real time
  - Instead, it calculates the whole signal at once, saves it to a file, then plays it back
    - The audio playback itself is real time, but that's not our focus right now!




```
buffer[n] = out;
```

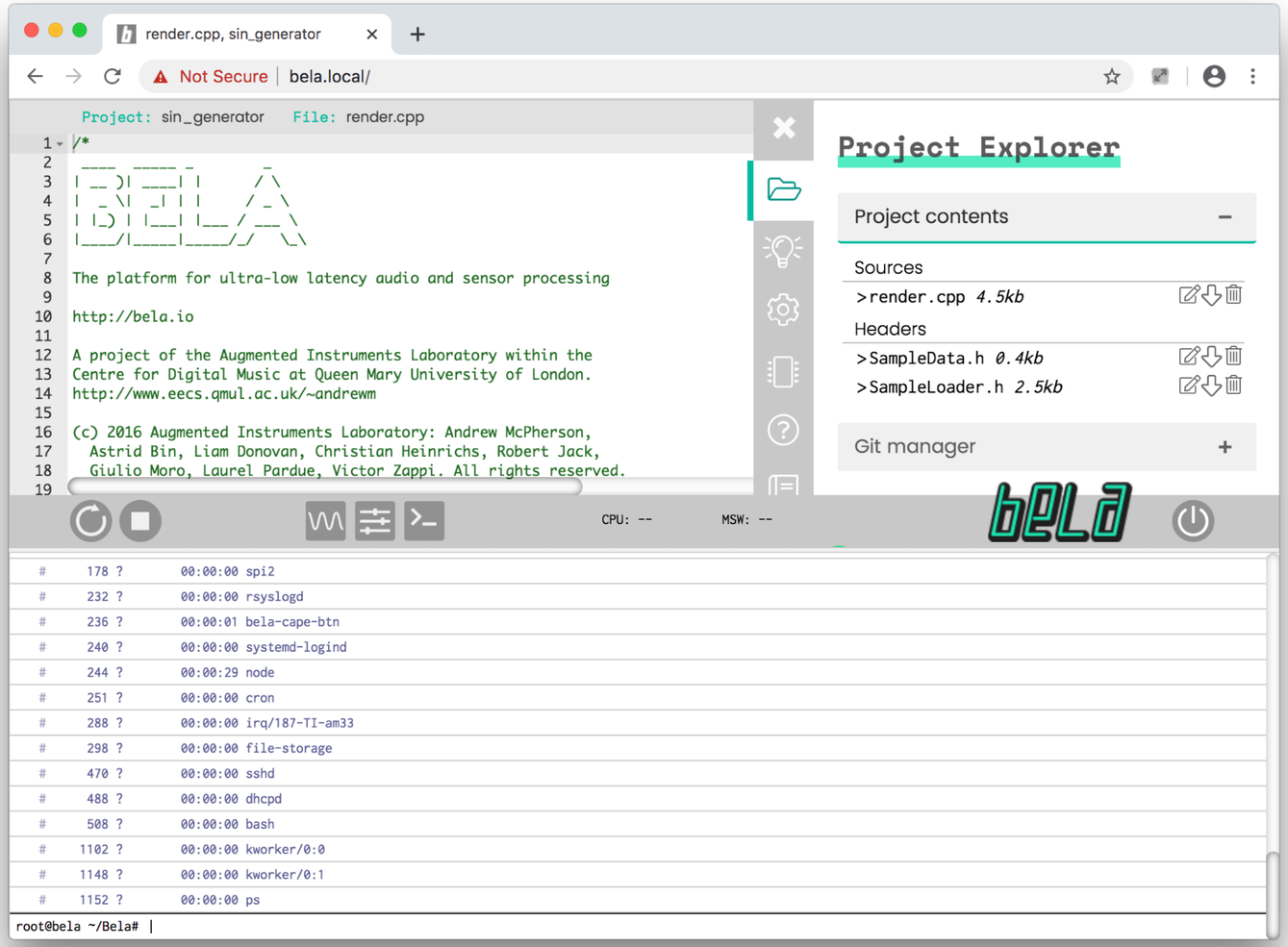
This is an array holding the signal
The value in `[ ]` indicates the index (which element)

Moving to real time

- sine-generator generates the sine wave offline (non-real time)
- Our goal is to re-implement it online (real time)
  - Let's generate the audio as we need it!
  - But what does it take to do this?
- Digital audio is composed of samples
  - In this case, our sample rate is 44100 samples per second (44.1kHz)
  - That means we need a new sample every 1/44100 seconds
    - That's every **22.7µs**, which is our sampling period
- Possible approach: run a short bit of code every time we need a new sample
  - What might be a drawback of this approach?
  - The computer runs lots of different processes at the same time
  - Can we guarantee that we'll get to every sample within 22.7µs?

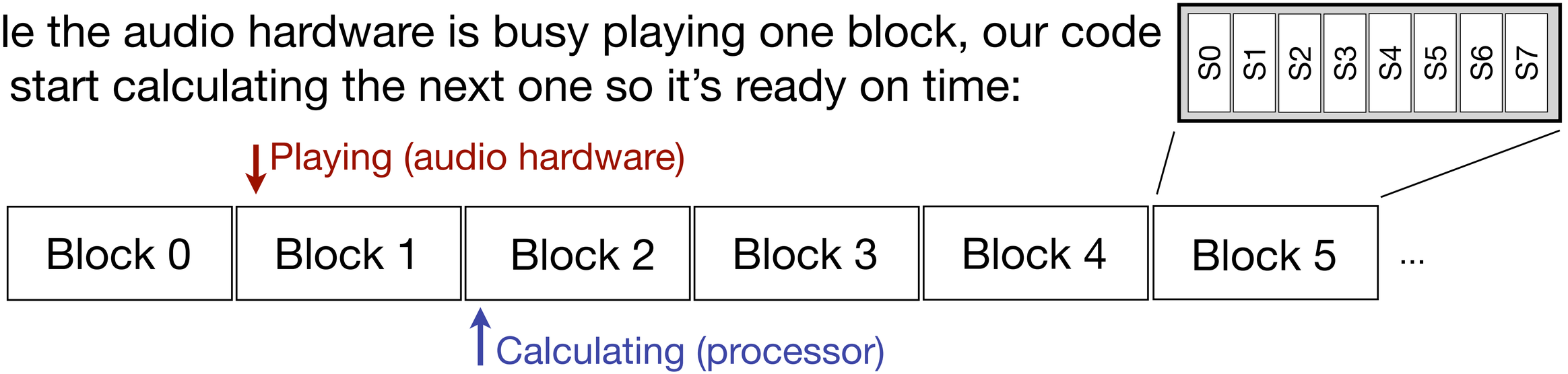
Processes

- In the box at the bottom of the Bela IDE, type **`ps`** (and hit enter)
  - Scroll through the resulting text
- These are all the processes running on the board
  - Each one takes CPU and memory
  - The OS switches rapidly from one to the next
  - You can find out more by running **`top`** from a terminal window
- Every process might itself contain multiple threads
- We can't guarantee our audio calculation will happen on time!



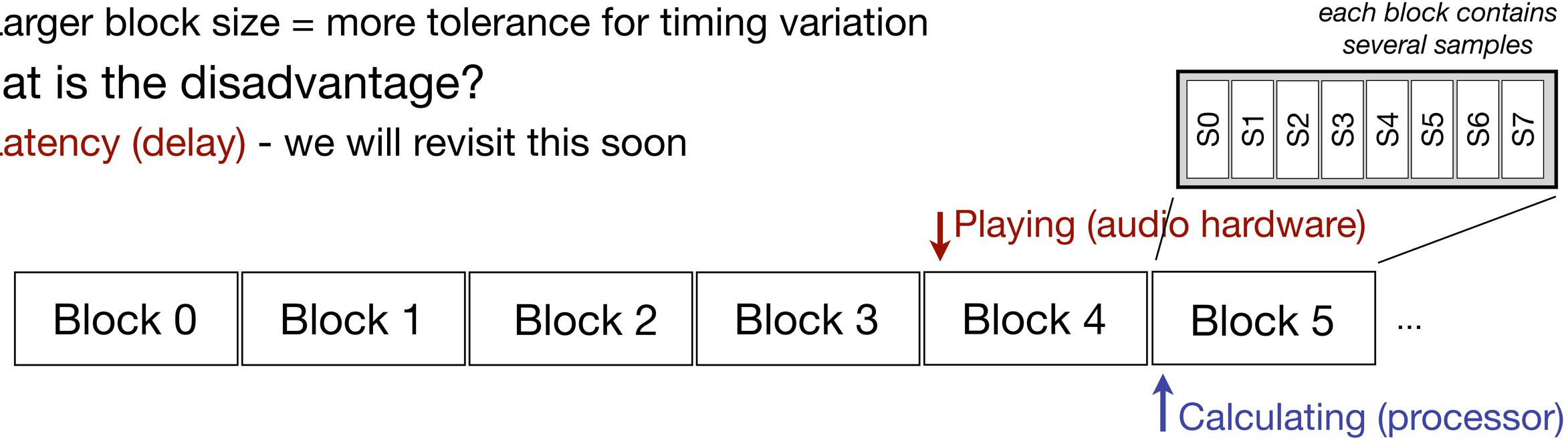
Block-based processing

- A better real-time option: 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:



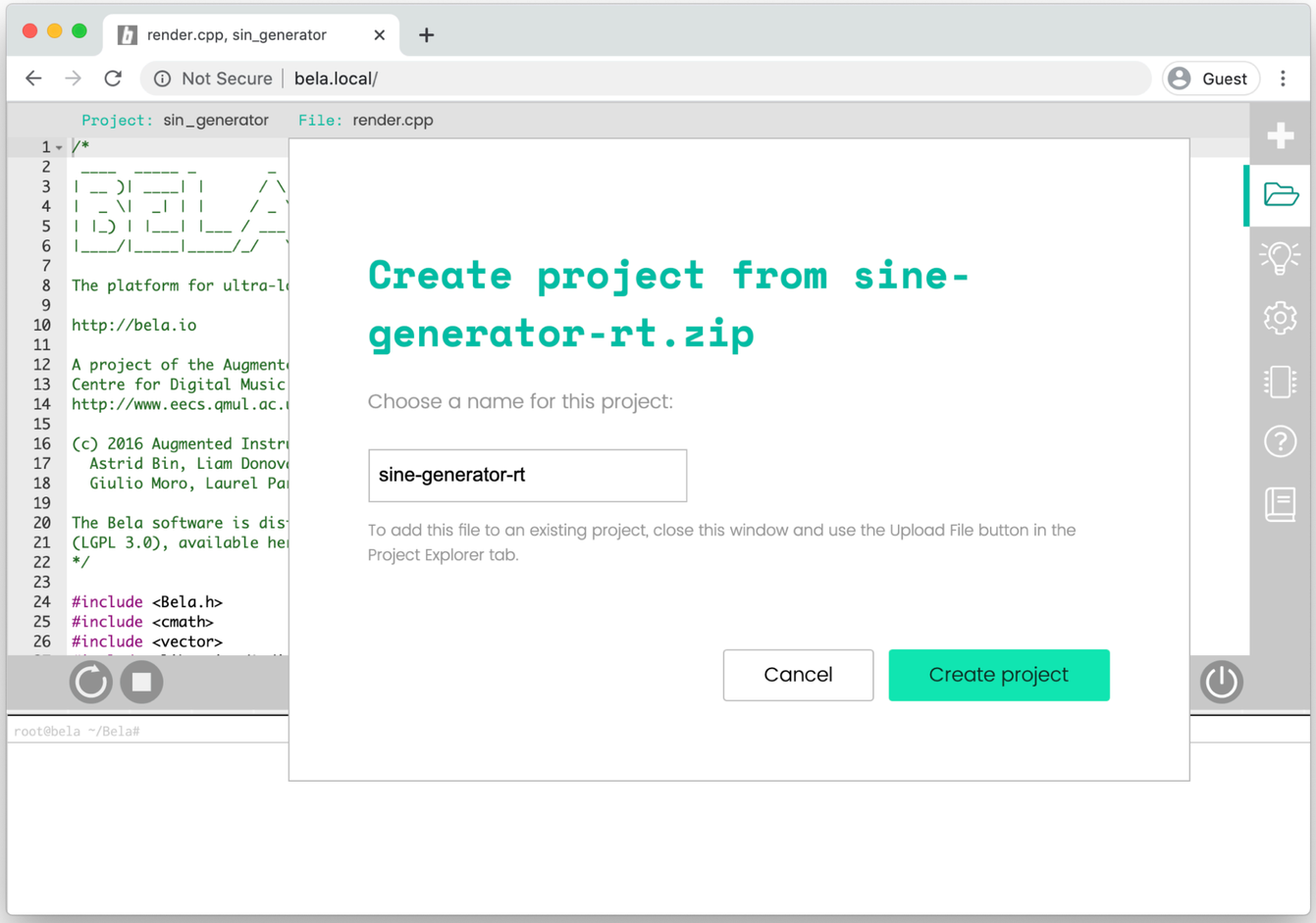
Block-based processing

- Advantages of blocks vs. processing 1 sample at a time:
  - We need to run our function less often
  - We always generate 1 block ahead of what is actually playing
  - Suppose that one block lasts 5ms, and running our code takes 1ms
  - Now, we can tolerate a delay of up to 4ms if the OS is busy with other tasks
  - Larger block size = more tolerance for timing variation
- What is the disadvantage?
  - Latency (delay) - we will revisit this soon



From non-RT to RT

- Download **sine-generator-rt.zip** from the GitHub companion materials and drag it into the IDE
- Copy the calculation for **`out`** from the previous project sine-generator
  - Replace `sampleRate` with `context->audioSampleRate`
- We'll now be generating samples one block at a time
  - Run this and see how it sounds



Handling calculation by blocks

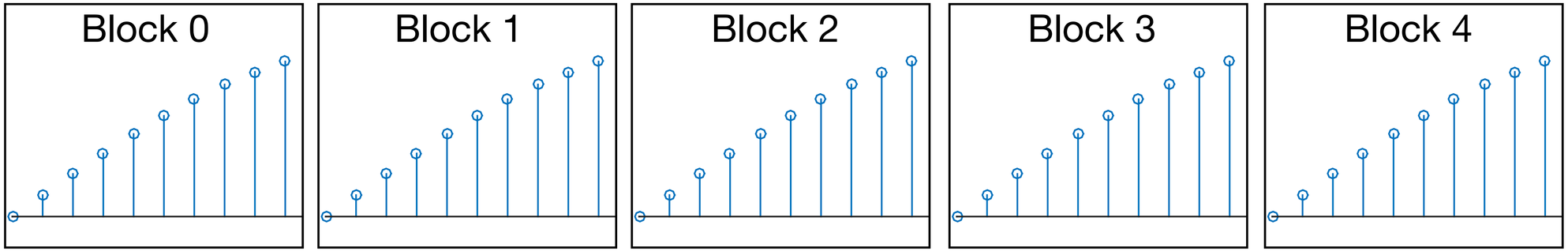
- Why does it sound distorted?
- Consider the phase of the `sin()` function: $x[n] = \sin(2\pi n f / f_s)$
  - The phase depends on `n`, which is declared here:

```
float out = gAmplitude * sin(2.0 * M_PI * n * gFrequency / context->audioSampleRate);
```

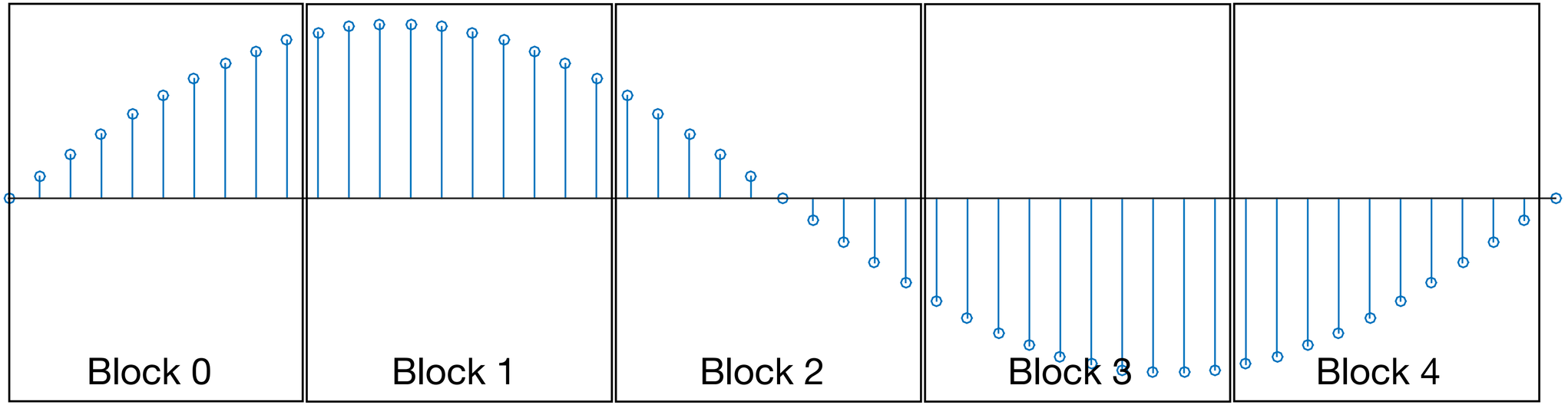
- The default block size on Bela is 16 samples
  - What range does `n` take for each block?
    - 0 to 15
    - Each block starts over at `n` = 0
  - Try changing the block size in the Settings tab
    - Sound should be different. Why?

Blocks and phase

- What we get is something like this:



- But what we want is this:



- How should we implement this?

Blocks and phase: task

- We need to preserve state between calls to `render()`
  - When `render()` runs a second time, it should remember when it left off the first time
  - But local variables in the function all disappear when the function returns!
  - Solution: use global variables to save the state
    - Okay, cleaner solutions exist: static variables; structure that you pass as an argument to `render()`
    - Or you could use instance variables (variables that are declared inside a class rather than inside a function or method). But that's a topic for a different time.
- Try it now!
  - Declare a global variable to keep track of the total number of samples that have elapsed
  - Use that variable, together with `n`, to calculate the phase for `sin()`

Preserving state: code

- Back to our oscillator. Here we use a global variable to keep track of how many samples have gone by:

Global variables remember their value

```
unsigned int gTotalSamples = 0;    // Keep track of total samples

// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
  // This for() loop goes through all the samples in the block
  for (unsigned int n = 0; n < context->audioFrames; n++) {
    // Increment the total number of samples
    gTotalSamples++;

    // Calculate a sample of the sine wave
    float out = gAmplitude * sin(2.0 * M_PI * gTotalSamples * gFrequency /
                                 context->audioSampleRate);

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

This runs once per block

This runs once per sample within the block (default 16 times on Bela)

Local variables forget their value after the end of the {} where they are defined

Preserving state: code

- Back to our oscillator. Here we use a global variable to keep track of how many samples have gone by:

Global variables remember their value

```
unsigned int gTotalSamples = 0;    // Keep track of total samples

// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
  // This for() loop goes through all the samples in the block
  for (unsigned int n = 0; n < context->audioFrames; n++) {
    // Increment the total number of samples
    gTotalSamples++;

    // Calculate a sample of the sine wave
    float out = gAmplitude * sin(2.0 * M_PI * gTotalSamples * gFrequency /
                                 context->audioSampleRate);

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

This runs once per block

This runs once per sample within the block (default 16 times on Bela)

Local variables forget their value after the end of the {} where they are defined

Preserving state: code

- Here's another way to write the same thing:
  - Update the global variable once at the end of the block rather than each sample

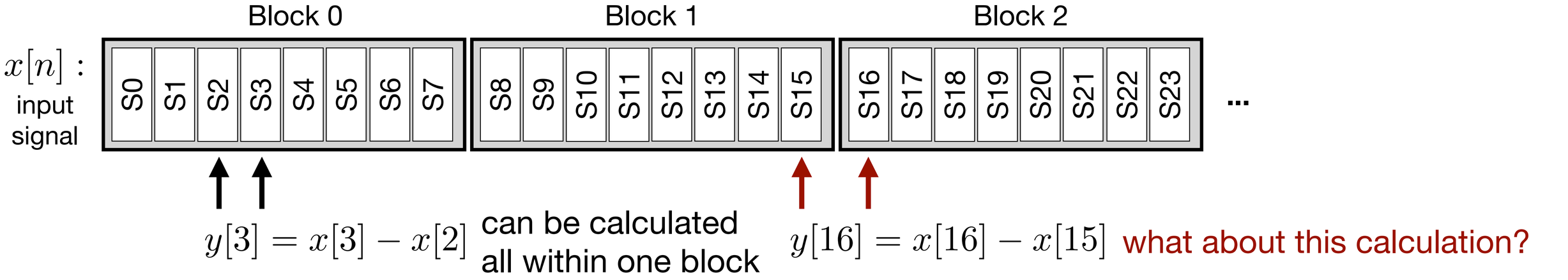
```
unsigned int gTotalSamples = 0;      // Keep track of total samples

// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
  // This for() loop goes through all the samples in the block
  for (unsigned int n = 0; n < context->audioFrames; n++) {
     // Calculate a sample of the sine wave
     float out = gAmplitude * sin(2.0 * M_PI * (gTotalSamples + n) * gFrequency /
                                  context->audioSampleRate);

     // Store the sample in the audio output buffer
     for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
        audioWrite(context, n, channel, out);
     }
  }

  // Increment the total number of samples by one block size
  gTotalSamples += context->audioFrames;
}
```

Preserving state

- To run in real time, we need to remember what we've done before
  - The **only** kind of DSP system that can be implemented in real time without preserving state is a memoryless system
    - i.e. a system where $y[n]$ depends only on the current $x[n]$ and no previous values
  - Why is this true? Does it still apply when we process a whole block at a time?
  - Take the equation for a first-order difference:

$$y[n] = x[n] - x[n-1]$$

  - Why do we need to preserve our state across calls to `render()`?



Which state do we preserve?

- In the previous code examples, what happens if the frequency changes?
  - $x[n] = \sin(2\pi n f / f_s)$
  - If `gFrequency` (f) changes to a new value after 5 seconds, what is a possible problem we might encounter?

$$\sin(2\pi 440(220500/44100)) = 0$$

$$\sin(2\pi 440.01(220500/44100)) \approx 0.309$$

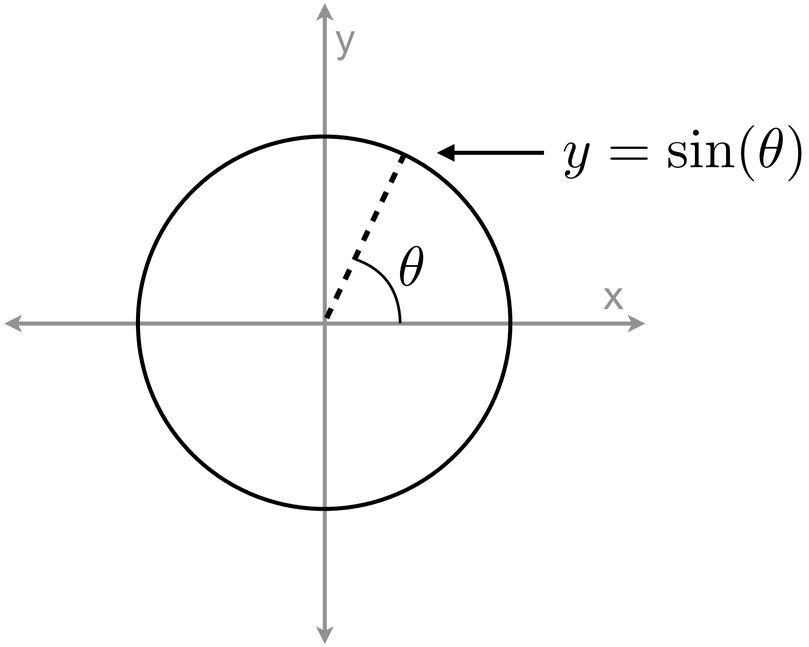
  - The phase might jump when the frequency changes even a tiny bit
  - Let's try it:

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

  gFrequency *= 1.00001;
}
```

Add this line to the end of `render()`

Frequency and phase

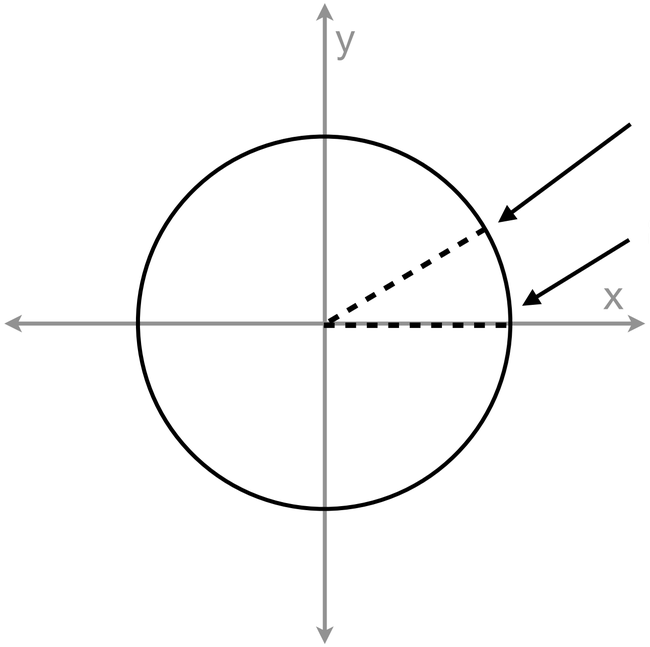
- The argument to the $\sin()$ function is known as the phase
  - Possible range of (unique) values: 0 to 2π
  - Analogous to the y-coordinate on the unit circle for a particular angle



- What does frequency correspond to in this analogy?
  - How fast we spin around the circle: the derivative of the phase

Frequency and phase

- Why is our old code problematic when the frequency changes?
  - We recalculate the phase every time based on how many samples have gone by
  - If we change the frequency, the phase can jump, which produces a click



$$\sin(2\pi 440.01(220500/44100)) \approx 0.309$$

$$\sin(2\pi 440(220500/44100)) = 0$$

- What should we do instead?
  - Remember the phase, not the number of samples

Preserving phase

- Previously we saved the total number of samples
  - We then recalculated $2\pi n f / f_s$ for each sample
- Let's try this instead:
  - At the beginning: $\phi = 0$
  - Then for each sample: $x[n] = sin(\phi)$
    $\phi = \phi + 2\pi f / f_s$
- To preserve state, keep track of ϕ between calls to `render()`
- Try it now!

Preserving phase

Global variable remembers phase

```
float gPhase = 0;     // Keep track of the phase

// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {
      // Increment the phase by one sample's worth at this frequency
      gPhase += 2.0 * M_PI * gFrequency / context->audioSampleRate;

      // Calculate a sample of the sine wave
      float out = gAmplitude * sin(gPhase);

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

Notice: no reference to `n` anymore (why?)

There's a subtle issue as `gPhase` gets larger...

...loss of precision

This gets a lot simpler!

Preserving phase

Wrap the phase to remain between 0 and 2π

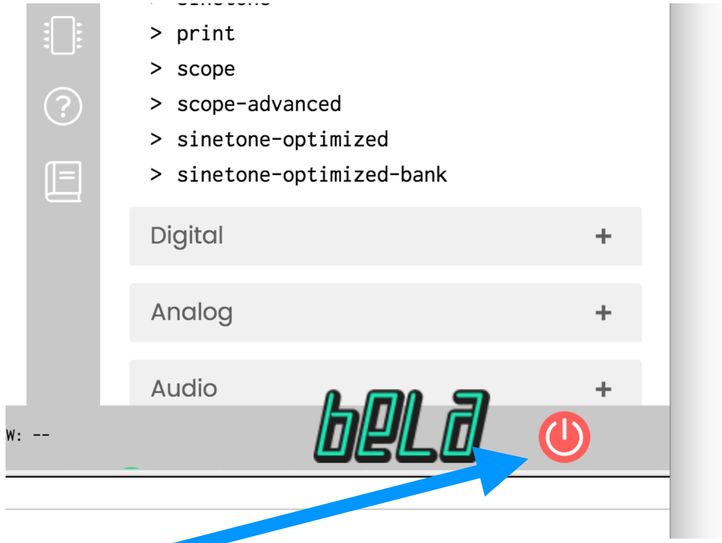
```
float gPhase = 0;     // Keep track of the phase

// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {
      // Increment the phase by one sample's worth at this frequency
      gPhase += 2.0 * M_PI * gFrequency / context->audioSampleRate;
      if(gPhase >= 2.0 * M_PI)
         gPhase -= 2.0 * M_PI;

       // Calculate a sample of the sine wave
      float out = gAmplitude * sin(gPhase);

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

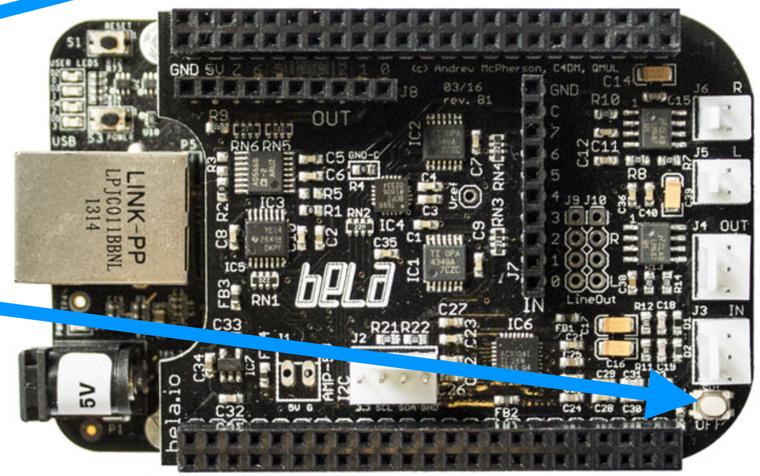
Bonus task

- Update your project to have separate frequencies for each channel
  - 440Hz on the left channel (channel 0)
  - 660Hz on the right channel (channel 1)
- Get rid of the `for()` loop across the number of channels
  - Just use two `audioWrite()` statements, one for each channel
- Some questions:
  - How many extra global variables do you need?
  - What else needs to change in `render()`?

Turning off Bela

- When you're done,
  don't just pull the power!
- Bela is a full Linux computer which needs to be shut down
- Two options:
  - In the IDE, click on the Shutdown button
  - Hold the white OFF button on the Bela cape for several seconds





Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources