C++ Real-Time Audio Programming with Bela

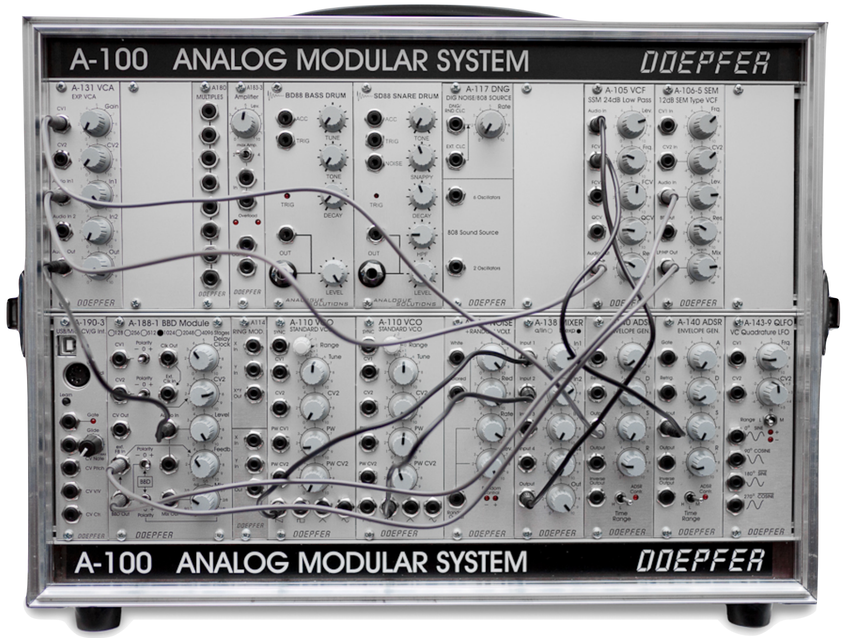
Dr Andrew McPherson

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

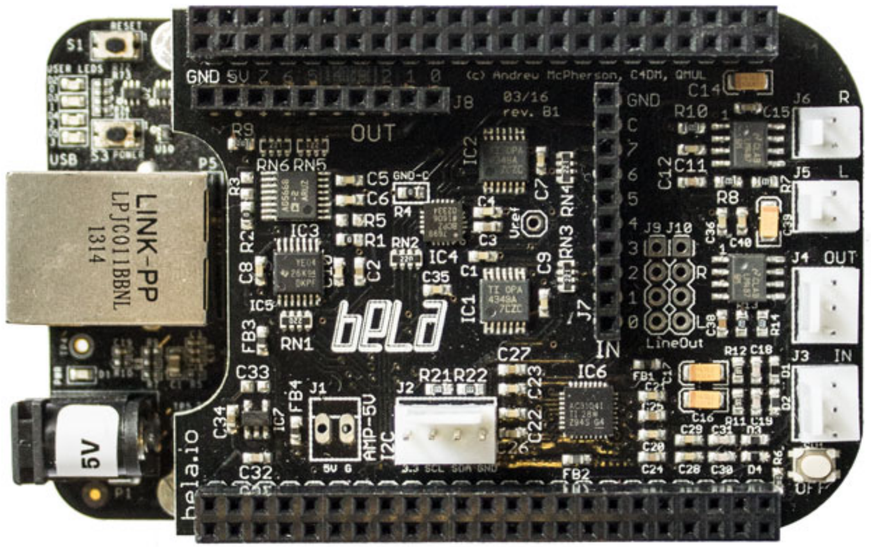


Image credit: Nina Richards, wikimedia (CC-BY 3.0)

Course topics

Programming topics	Music/audio topics
Working in real time	Oscillators
Buffers and arrays	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

Today

Lecture 2: Playing recorded samples

What you'll learn today:

The Bela C++ API
Working with buffers (arrays)

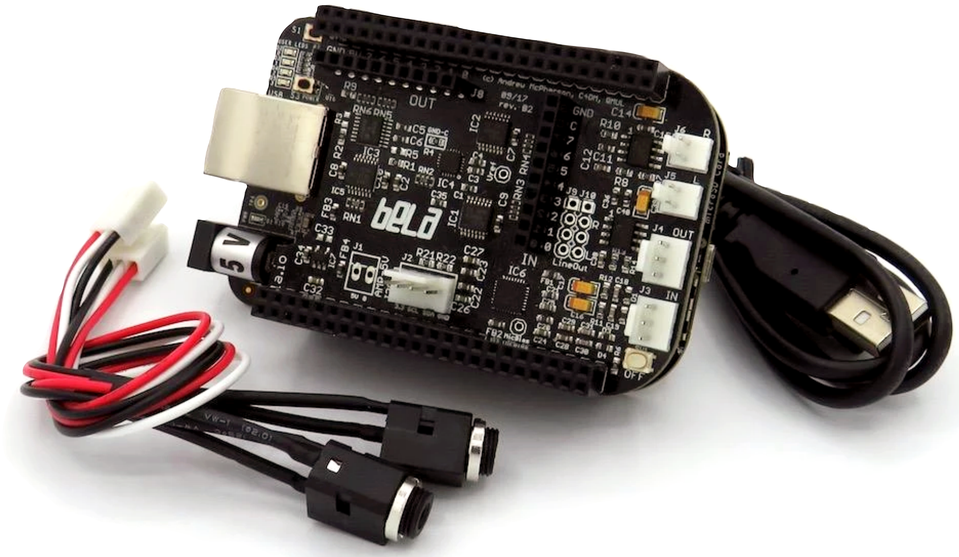
What you'll make today:

Sound-file player

Companion materials:

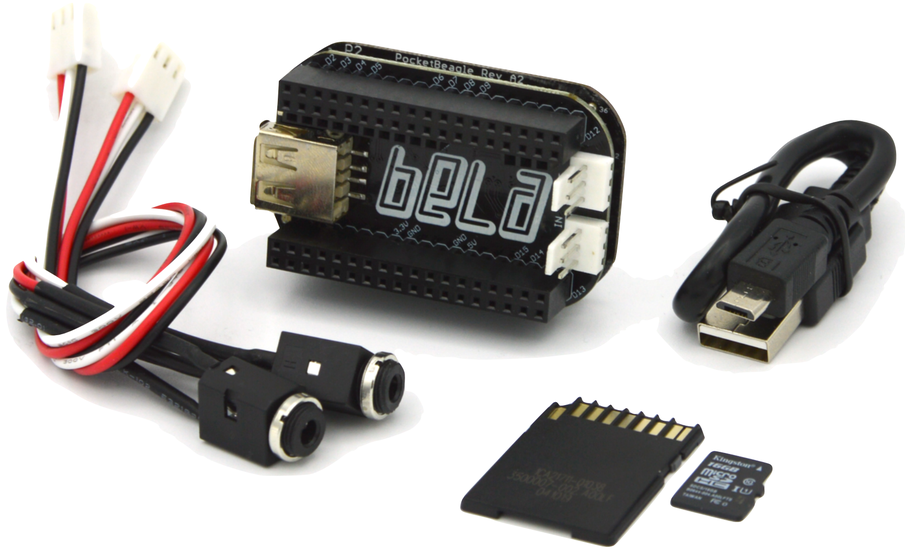
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

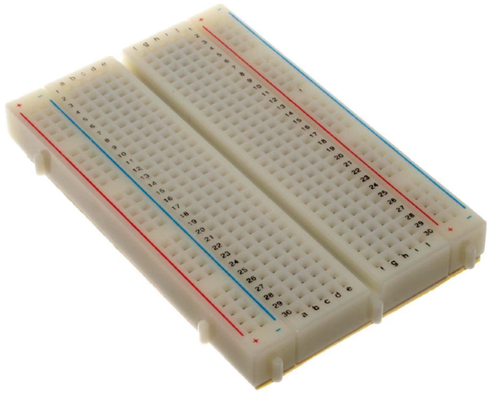
or

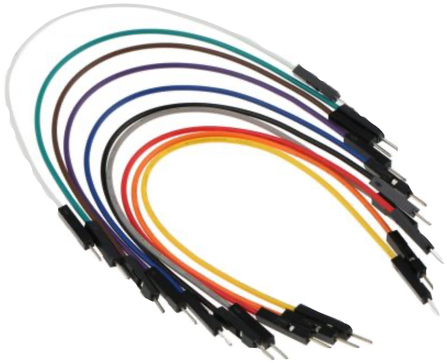


Bela Mini Starter Kit

[shop.bela.io]

Recommended for some lectures:

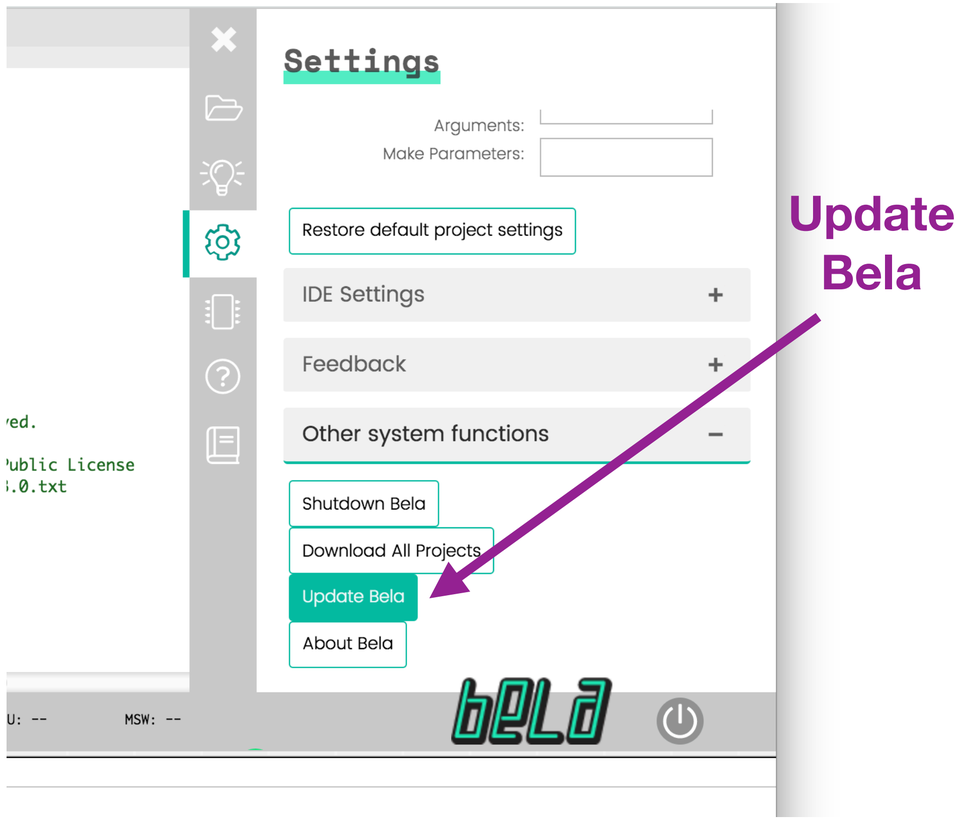




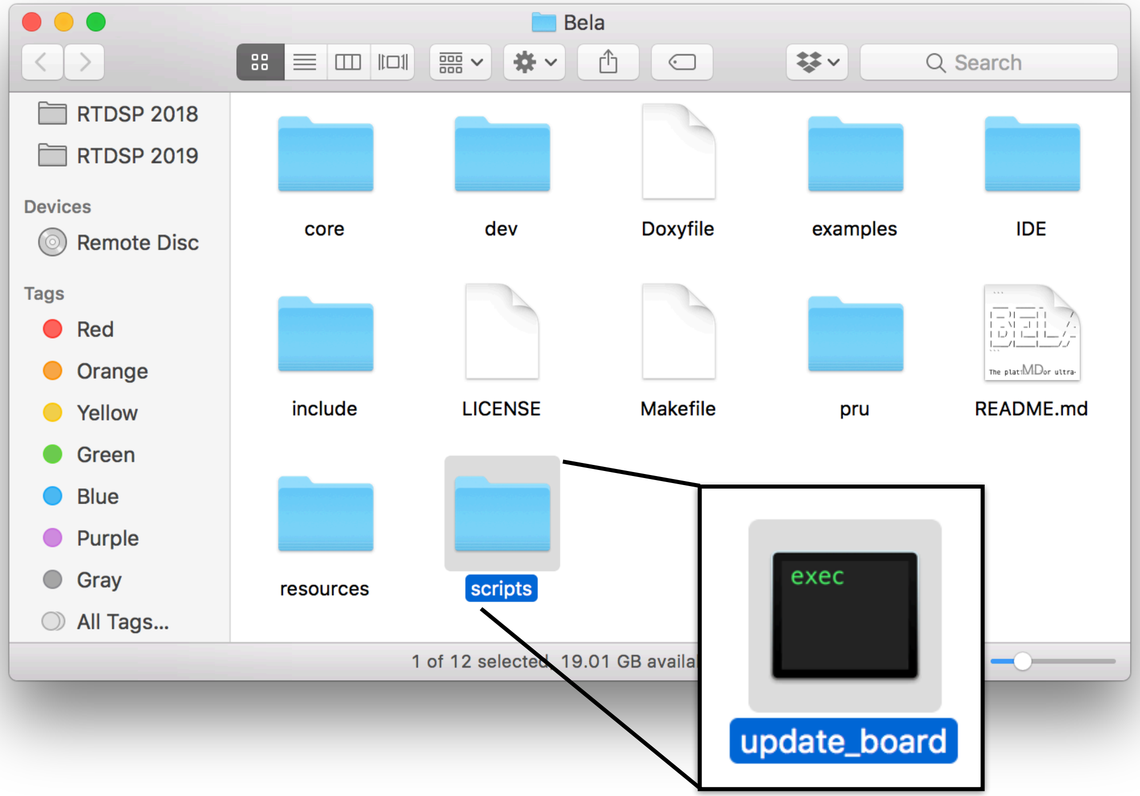
Updating Bela

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

By IDE (recommended)



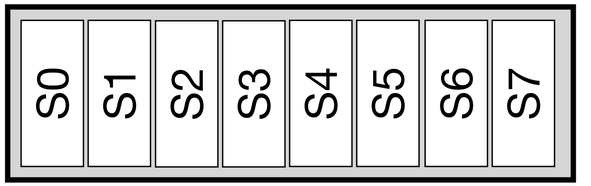
By Script (Mac and Linux)
unzip the archive

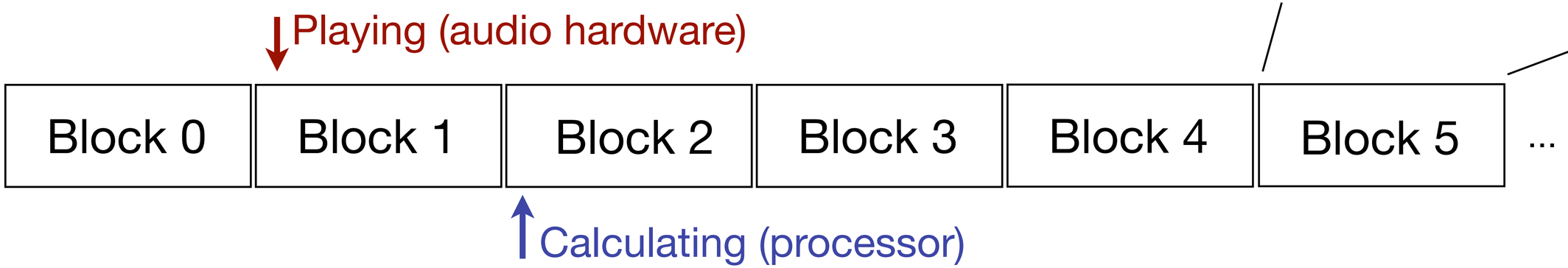


Review: block-based processing

- Real-time systems process audio in blocks, rather than one sample at a time
  - Generate enough samples at a time to get through the next few milliseconds
  - Typical block sizes on general-purpose operating systems: 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:

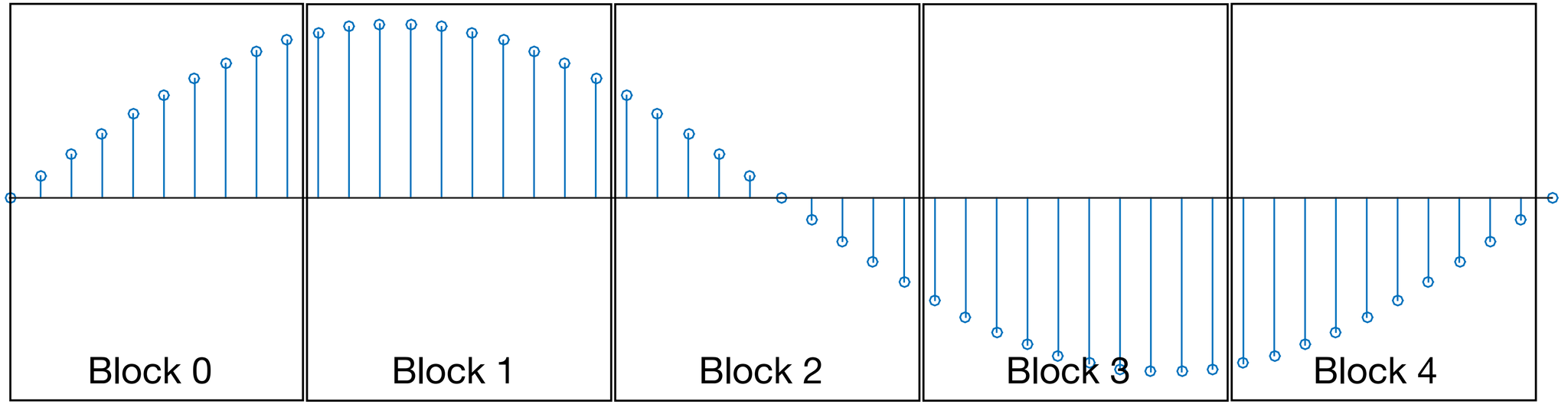
each block contains several samples



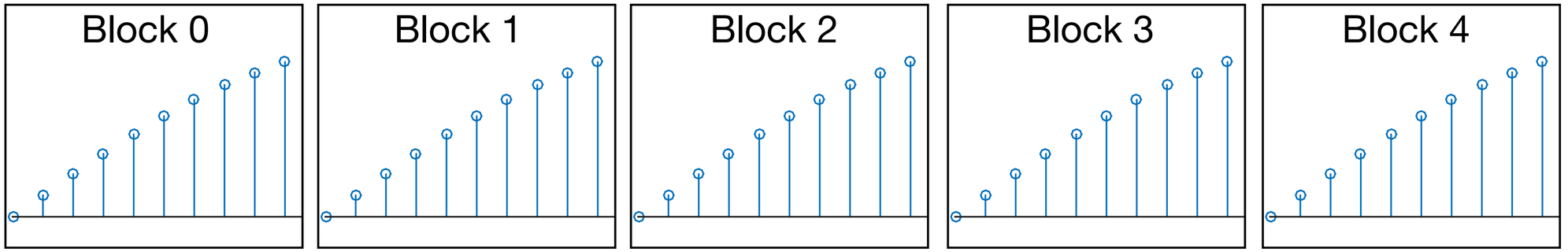


A real-time sine oscillator

- Last lecture, we wanted to generate a sine wave in real time, like this:



- But when we started, what we got was this:



- The solution was to remember the phase between calls to `render()`

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);
      }
    }
}
```

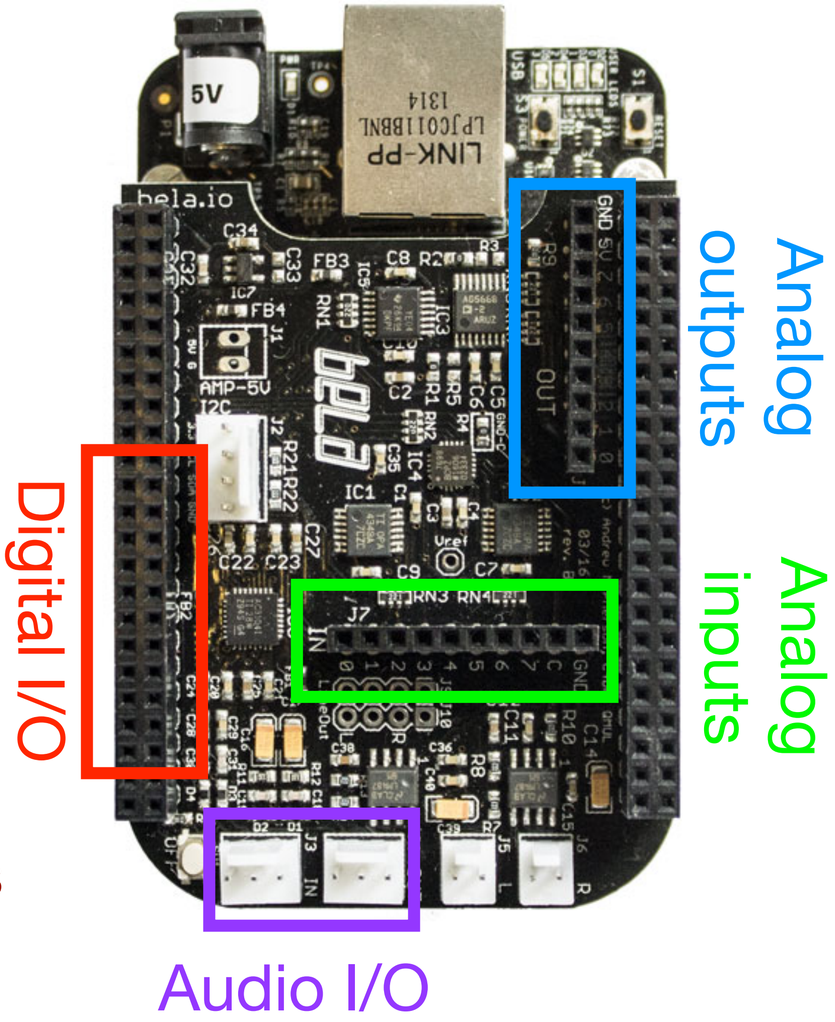
What's all this stuff?

The Bela C / C++ API

- Every Bela program has three main functions:
- `bool setup(BelaContext *context, void *userData)`
  - Runs once at the beginning, before audio starts
  - Data structure `context` holds info on channels, sample rates, block sizes
  - Return `true` if initialisation was successful (`false` stops the program)
- `void render(BelaContext *context, void *userData)`
  - The audio callback function
  - Called automatically by the Bela system for each new block
  - Where most of your code goes: process the samples and return as quickly as possible!
- `void cleanup(BelaContext *context, void *userData)`
  - Runs once at the end, when the program stops
  - Use to release any resources you allocated in `setup()`

The Bela C / C++ API

- `BelaContext` is a type of data structure (`struct`)
- It holds lots of useful information about the outside world:
  - Block sizes
  - Sample rates
  - Numbers of channels
  - The sample data itself!
- It holds information on audio, analog and digital I/O
  - This means we don't need to call any special functions to send and receive samples
  - All we need to do is write or read the appropriate elements inside this data structure

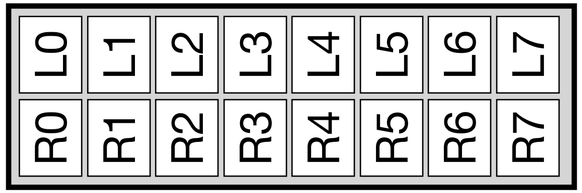


BelaContext

- See the Documentation tab in the IDE or **learn.bela.io** for full description

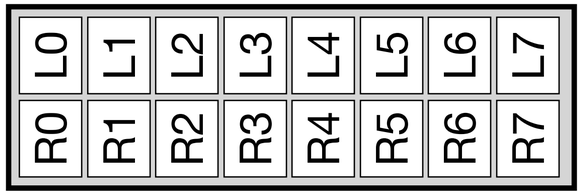
`struct BelaContext`

`float* audioIn`



buffer of audio input samples for all channels (we'll look at how this is organised soon)

`float* audioOut`



buffer of audio output samples for all channels

`int audioFrames` — 8 — *the audio block size*

`int audioInChannels` — 2 — *how many audio input channels*

`int audioOutChannels` — 2 — *how many audio output channels*

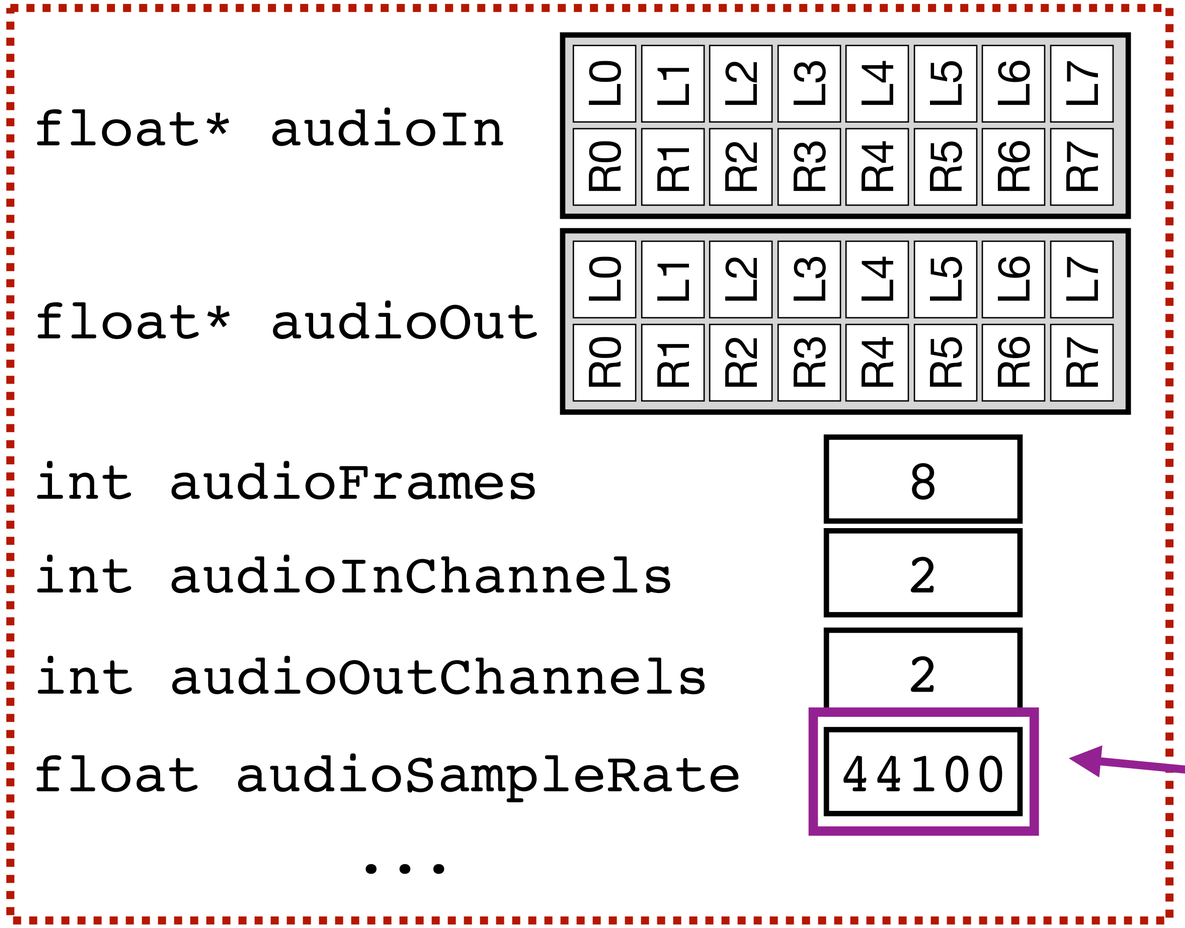
`float audioSampleRate` — 44100 — *sample rate of audio data*

`...`

BelaContext

- See the Documentation tab in the IDE or **learn.bela.io** for full description

`struct BelaContext`



all this information is passed to `render()`

```
void render(BelaContext *context, void *userData)
```

`render()` gets a pointer to the data structure. That means the data lives somewhere in memory, and this tells us where to find it

To access an element inside `context`:

```
context->audioSampleRate
```

Preserving 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;
      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);
      }
    }
}
```

16

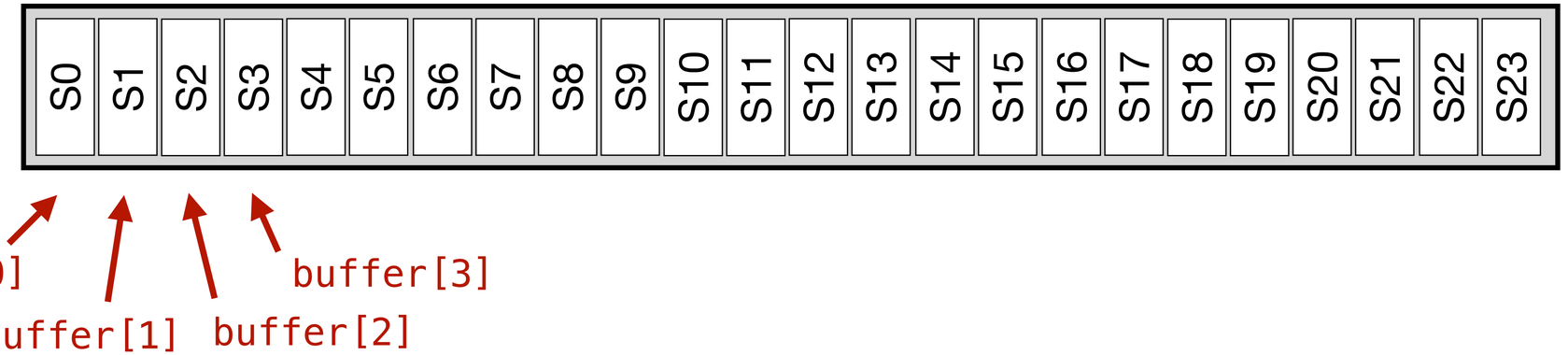
44100

2

Playing audio samples

- Suppose we want to play a clip of recorded audio
  - For example, something that we load from a WAV file
  - Let's not worry yet about how we load the sound from storage
- The sound, once loaded, will be stored in a buffer (i.e. array)

```
float buffer[24];
```



- An array is just a series of values of a particular type, stored one after another
  - Access individual elements using numbers in square brackets
  - A more sophisticated array is the C++ `vector` class, e.g.:

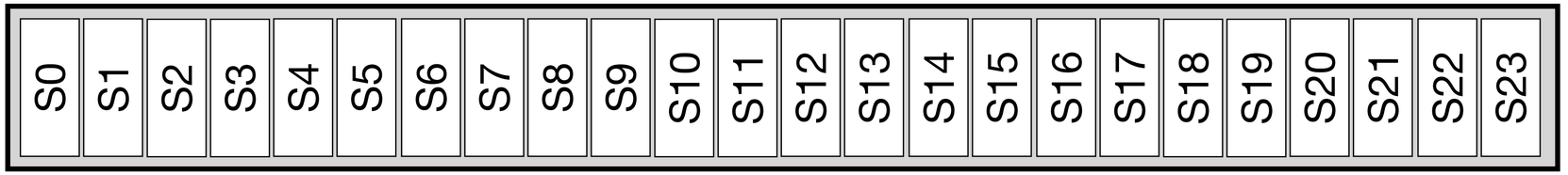
```
std::vector<float> buffer(24);
```

← *vector of type* `float`*, initial size 24*

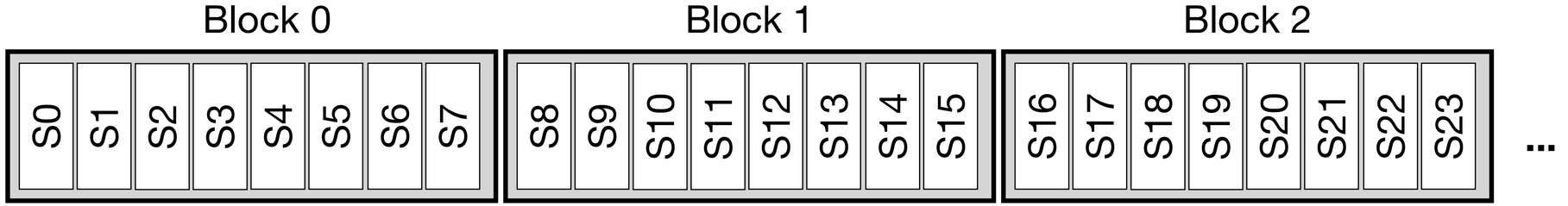
Playing audio samples

- Suppose we want to play a clip of recorded audio
  - For example, something that we load from a WAV file
  - Let's not worry yet about how we load the sound from storage
- The sound, once loaded, will be stored in a buffer (i.e. array)

```
float buffer[24];
```



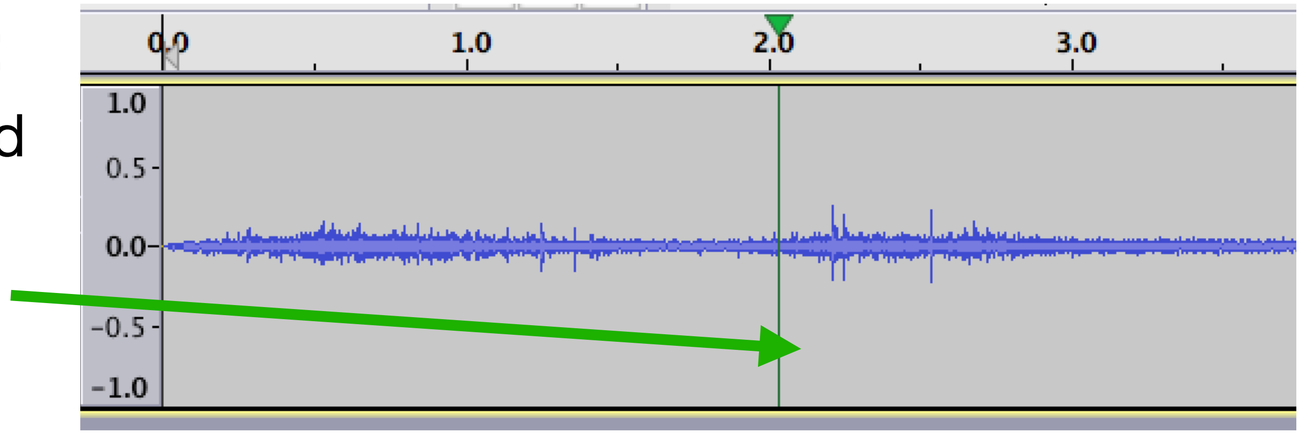
  - The buffer is almost certainly longer than the block size of our real-time system
- We need to copy these samples block-by-block into our audio output



- Besides the buffer itself, what do we need to keep track of to play this sound in real time?

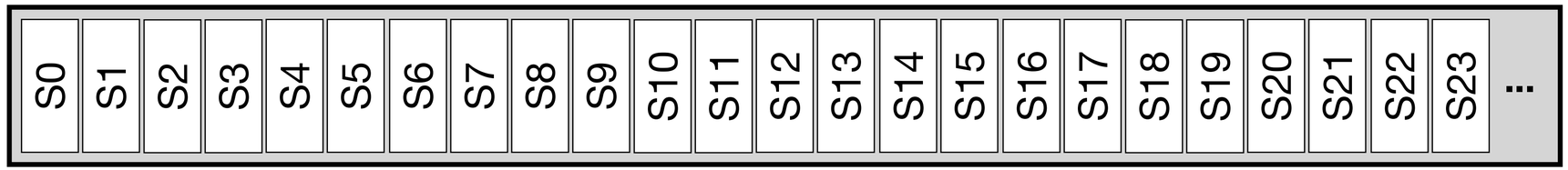
Keeping track of position

- Audio editors show the play head:
  - The location in the buffer being played
  - As the sound plays, the play head moves forward at constant speed



- In DSP programming, the play head is a special case of a read pointer
  - A reference indicating which index in the buffer to play next
  - Just like the phase of an oscillator, we want to remember this from one block to the next
    - So it should be a global variable!

the buffer doesn't change as we play it



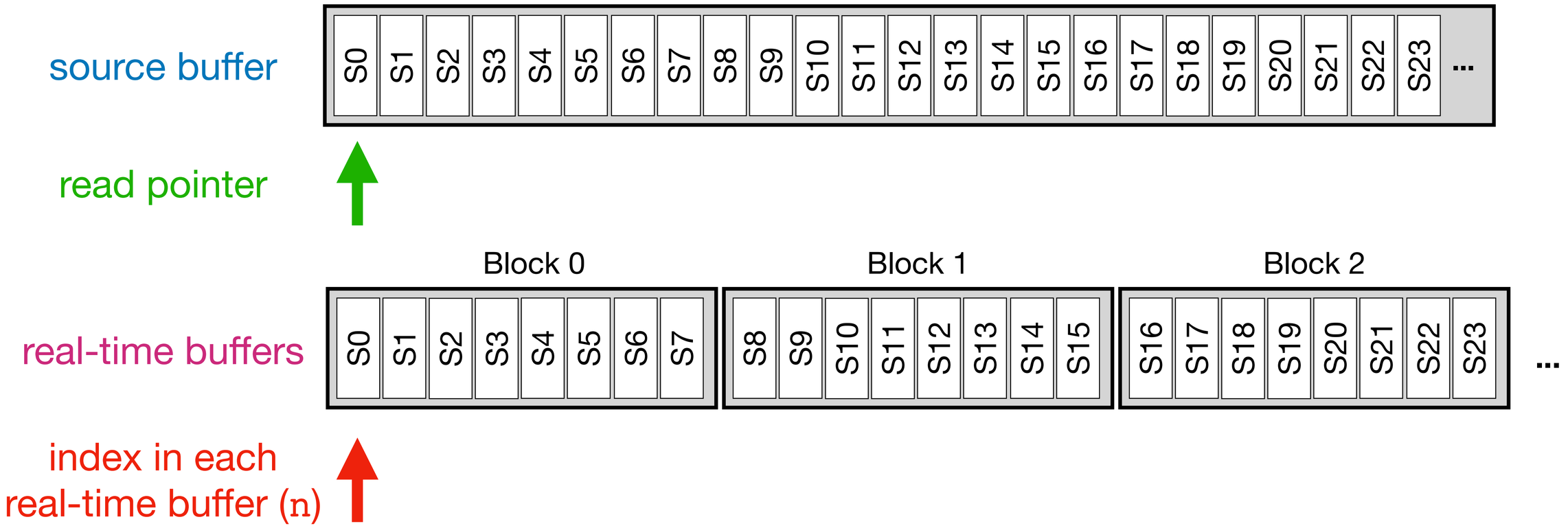
but the read pointer moves

read pointer

Fun with indexing

- One of the hardest parts of working with buffers can be keeping track of what each index means
- In this case, we've got two different kinds of buffers to think about:
  1. The recorded sound (let's call it the source buffer)
     - Only one buffer whose contents don't change
     - Length: number of samples in the source sound (possibly long)
  2. The buffer for each real-time audio block
     - A new buffer each time `render()` is called, accessed via `audioWrite()`
     - Length: block size of the real-time system (e.g. 16)
- Therefore, we need to keep track of two indexes:
  1. Where are we playing in the source buffer? (read pointer or play head)
  2. Where are we writing in the output buffer? (starts over from 0 each block)

Fun with indexing



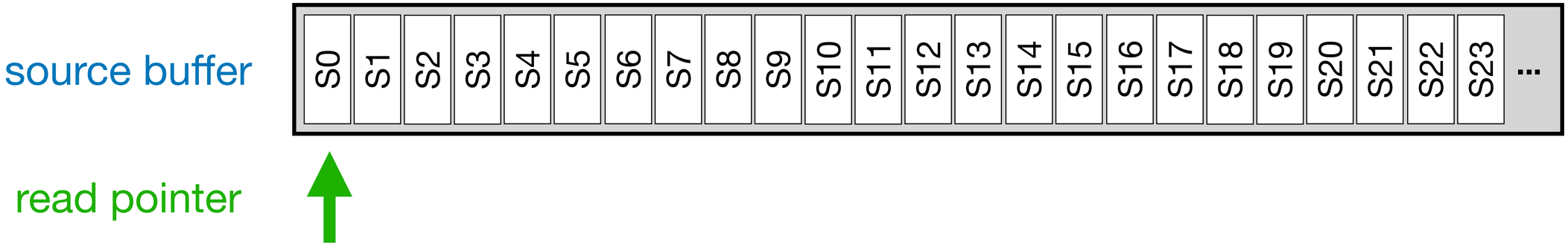
`render()` [1st time]
`n` goes from 0 to 7
`gReadPointer` goes from 0 to 7

`render()` [2nd time]
`n` goes from 0 to 7
`gReadPointer` goes from 8 to 15

`render()` [3rd time]
`n` goes from 0 to 7
`gReadPointer` goes from 16 to 23

Practical considerations

- What should we do when we reach the end of the source buffer?
  - If we fall off the end, we end up accessing a part of memory we shouldn't



  - This is called a segmentation fault and it will make our program crash
- What could we do instead?
  - Stop playing (no longer move the read pointer or copy samples)
  - Return the read pointer back to 0 and start over again
- Task: using the `sample-player` example, implement playback from the source buffer (`gSampleBuffer`), returning to 0 when the read pointer gets to the end

Playing audio samples

Global variables keep track of source buffer, read pointer

```
std::vector<float> gSampleBuffer;
int gReadPointer = 0;
```

```
// Buffer that holds the sound file
// Position of the last frame we played
```

`gReadPointer` keeps track of where we are in the source buffer

```
void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Load a sample from the buffer which was loaded from the file
        float out = gSampleBuffer[gReadPointer];

        // Increment read pointer and reset to 0 when end of file is reached
        gReadPointer++;
        if(gReadPointer >= gSampleBuffer.size())
            gReadPointer = 0;

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

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

update `gReadPointer`, whose value persists

`n` is used to specify which frame of the output buffer to write

What's this?

Audio buffers

- `audioWrite()` is a convenience function that accesses the audio buffer
  - This way we don't have to remember how the samples are organised in memory

```
void audioWrite(BelaContext *context, int frame, int channel, float value)
```

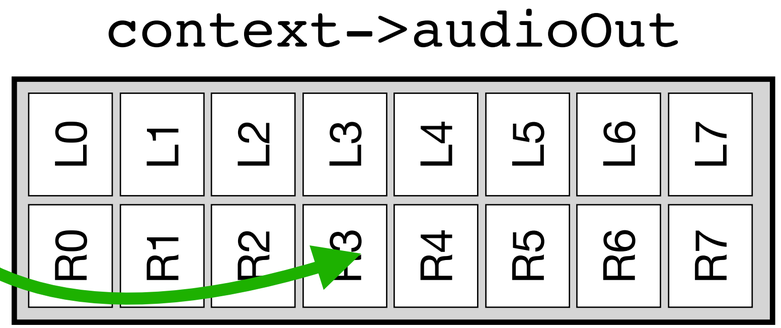
Reference to the `BelaContext` structure, which holds the actual data we want

Which frame (i.e. sample) within the buffer to write

Which channel (e.g. left or right) to write

What value to write

For example: `audioWrite(context, 3, 1, 0.5);`



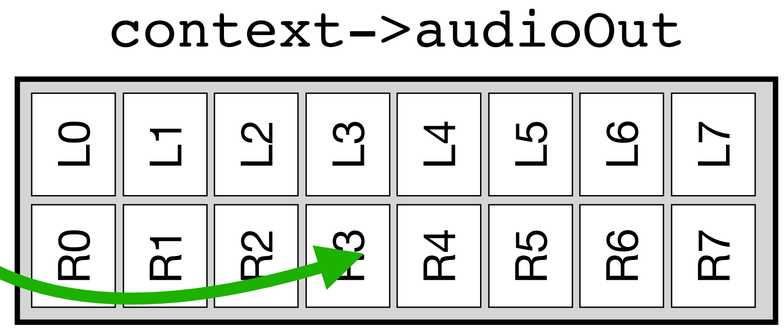
Audio buffers

- `audioWrite()` is a convenience function that accesses the audio buffer
  - This way we don't have to remember how the samples are organised in memory

```
void audioWrite(BelaContext *context, int frame, int channel, float value) {
  context->audioOut[frame * context->audioOutChannels + channel] = value;
}
```

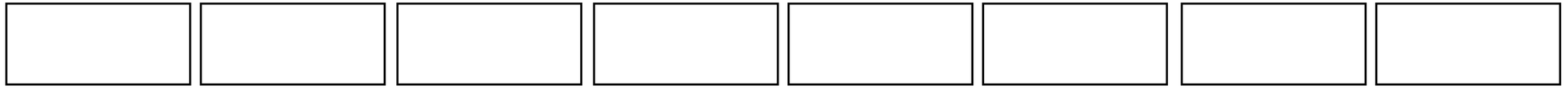
Can we work out how `audioOut` is organised based on this code?

For example: `audioWrite(context, 3, 1, 0.5);`

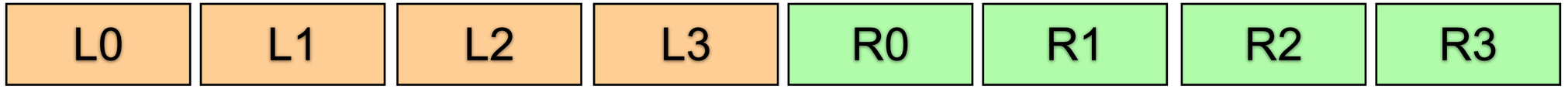


Interleaving

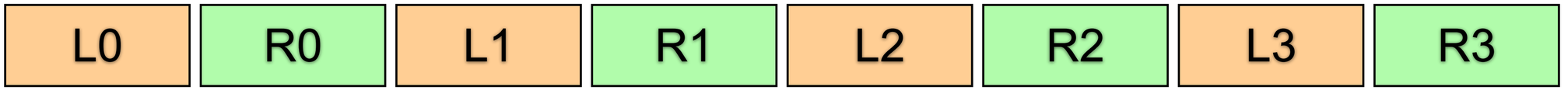
`float* audioOut` ⟵ a one-dimensional array (buffer)



- Two ways to organise a buffer containing multiple channels of data
- Non-interleaved
  - All the frames (samples) of channel 0, followed by all the frames of channel 1, etc.
  - A contiguous block of samples for each channel, one block after another



- Interleaved
  - All the channels of frame 0, followed by all the channels of frame 1, etc.
  - Alternating between channels, with the buffer organised by time (earlier samples first)



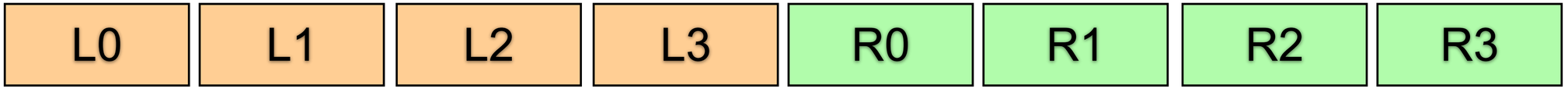
Interleaving

Which one does this code implement?

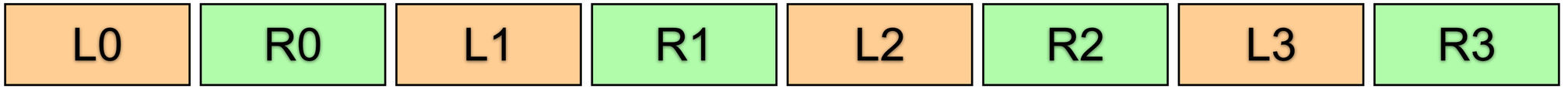
```
void audioWrite(BelaContext *context, int frame, int channel, float value) {
  context->audioOut[frame * context->audioOutChannels + channel] = value;
}
```

Interleaved

- Two ways to organise a buffer containing multiple channels of data
- Non-interleaved
  - All the frames (samples) of channel 0, followed by all the frames of channel 1, etc.
  - A contiguous block of samples for each channel, one block after another



- Interleaved
  - All the channels of frame 0, followed by all the channels of frame 1, etc.
  - Alternating between channels, with the buffer organised by time (earlier samples first)



Loading WAV files

- Bela has a set of straightforward functions for loading files: `AudioFile`

```
#include <libraries/AudioFile/AudioFile.h>
```

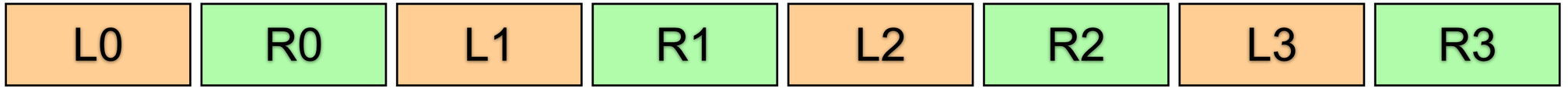
  - This is an example of a Bela library: pre-made code you can include in your project
  - See the IDE Libraries tab for a list of Bela libraries
- The `AudioFileUtilities::loadMono()` function loads a single channel from a WAV file

```
std::vector<float> gSampleBuffer = AudioFileUtilities::loadMono(gFilename);
```

  - `std::` and `AudioFileUtilities::` are namespaces: prefixes that help avoid ambiguities in our code when entities have similar names
  - Returns `std::vector<float>`, an array of `float` values with range -1 to 1
  - Check for loading error by seeing if the vector is empty: `gSampleBuffer.size() == 0`
- The `AudioFile` Bela library uses `libsndfile` *(https://github.com/erikd/libsndfile/)*
  - `libsndfile` also supports other formats (AIFF, FLAC, Ogg/Vorbis), but not MP3

Multichannel audio files

- Stereo or multichannel WAV files are stored in interleaved format



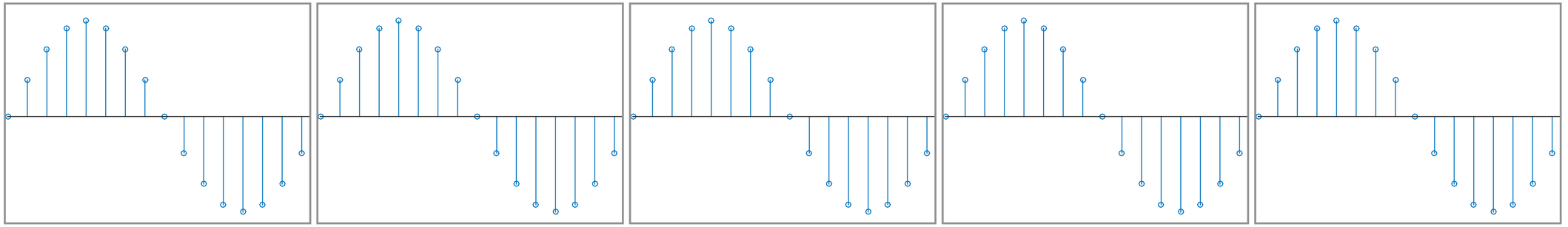
- Use `AudioFileUtilities::load()` to load a stereo or multichannel file
  - Return type is `std::vector<std::vector <float> >` -- what does this mean?
  - An array of arrays
  - First index is the channel, second index is the frame (sample) number
    - e.g. `gSampleBuffer[0][100]` would refer to channel 0 (left channel), frame 100
- In other words, once we load the file, we get separate buffers per channel
  - To play this back, we need to change the structure of our `for()` loops inside `render()`
- Task: using the project sample-player-stereo: implement stereo playback
  - Only need to change `render()`. The `setup()` function is done for you.
  - Notice the new type of `gSampleBuffer`!

Other considerations

- For very large sound files, we may not want to load it all into memory at once
  - The BeagleBone has 512MB of RAM. How much audio would this hold?
    - One `float` is 32 bits (4 bytes)
    - 512MB RAM will hold 128 million samples. How long is this at 44100Hz sample rate?
    - Assuming a mono signal, 2902 seconds = 48.4 minutes
- The alternative is to stream the file off the flash storage
  - Load smaller chunks from storage as needed
  - This is subtle because accessing the filesystem is not real-time safe (more on this later)
  - See the Bela example `Audio/sample-streamer` (on the board) for an implementation
- We have assumed our WAV files are the same sample rate as Bela (44.1kHz)
  - If not, we would need to resample them, either when we load or during playback
  - We will see more about resampling in the next lecture

Uses of repeating buffers

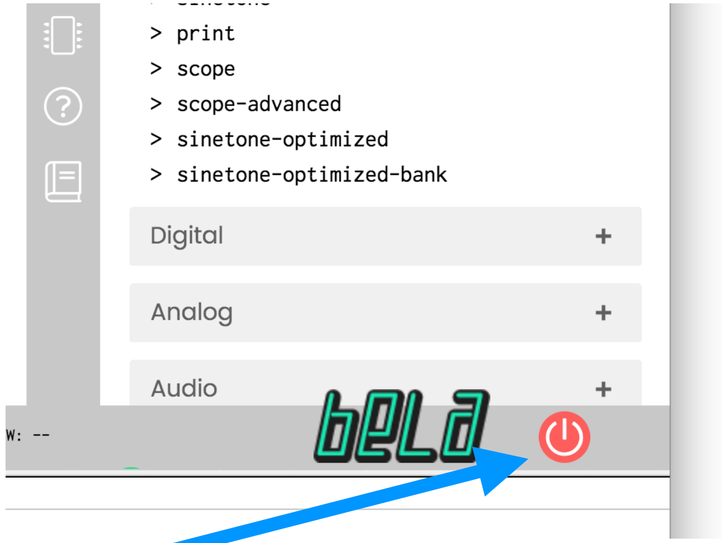
- There's another name for playing a buffer of samples over and over

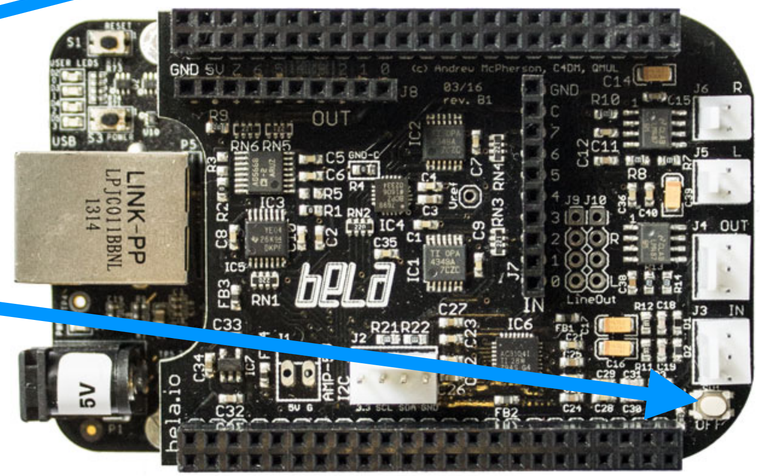


- **Wavetable**
- This is another way of building an oscillator
  - Rather than calculating the output mathematically every sample, play it from a buffer
  - Another way of thinking about it: a look-up table of output values
- More about wavetables in the next lecture!

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