Bela

Ultra-Low Latency Audio + Sensors on BeagleBone Black

A project by the **C4DM**
Augmented Instruments Lab

Andrew McPherson
Victor Zappi
Giulio Moro
Liam Donovan
Christian Heinrichs
Astrid Bin
Robert Jack
Laurel Pardue

Centre for Digital Music
Queen Mary University of London

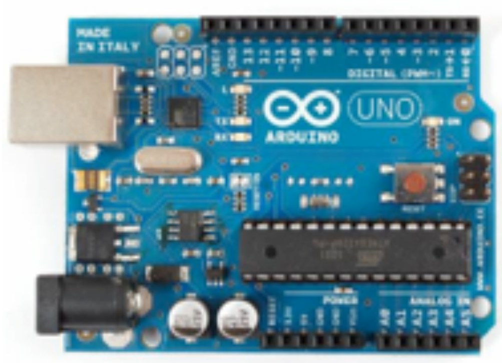
5 December 2015

http://beaglert.cc

EPSRC centre for digital music Queen Mary University of London

The Goal:

High-performance, self-contained audio and sensor processing



ARDUINO

- Easy low-level hardware connectivity
- No OS = precise control of timing
- Very limited CPU (8-bit, 16MHz)
- Not good for audio processing

- Reasonable CPU (up to 1GHz ARM)
- High-level hardware (USB, network etc.)
- Limited low-level hardware
- Linux OS = high-latency / underruns

- Fast CPU
- High-level hardware (USB, network etc.)
- Arduino for low-level
- USB connection = high-latency, jitter
- Bulky, not self-contained

Bela *features*

1ms round-trip audio latency without underruns

High sensor bandwidth: digital I/Os sampled at 44.1kHz; analog I/Os sampled at 22.05kHz

Jitter-free alignment between audio and sensors

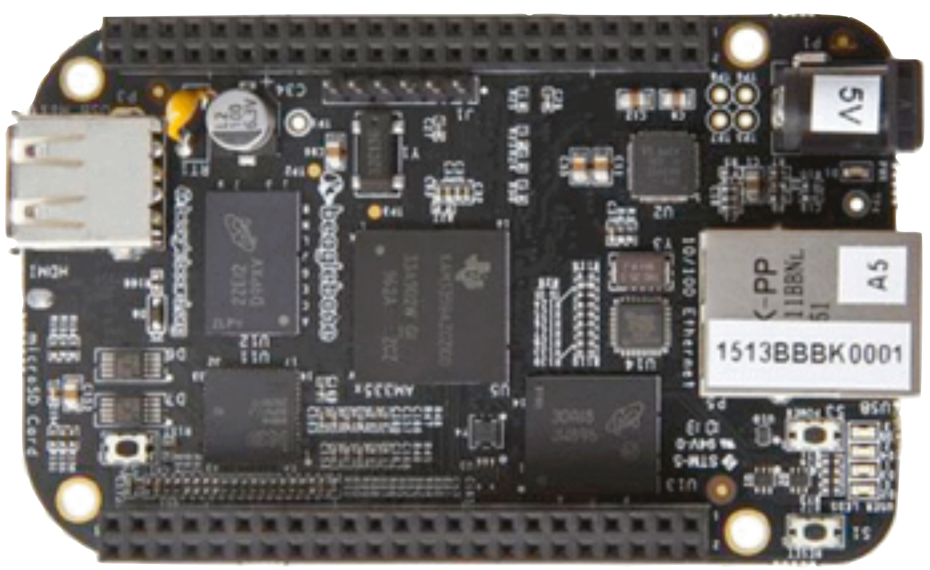
Hard real-time audio+sensor performance, but full Linux APIs still available

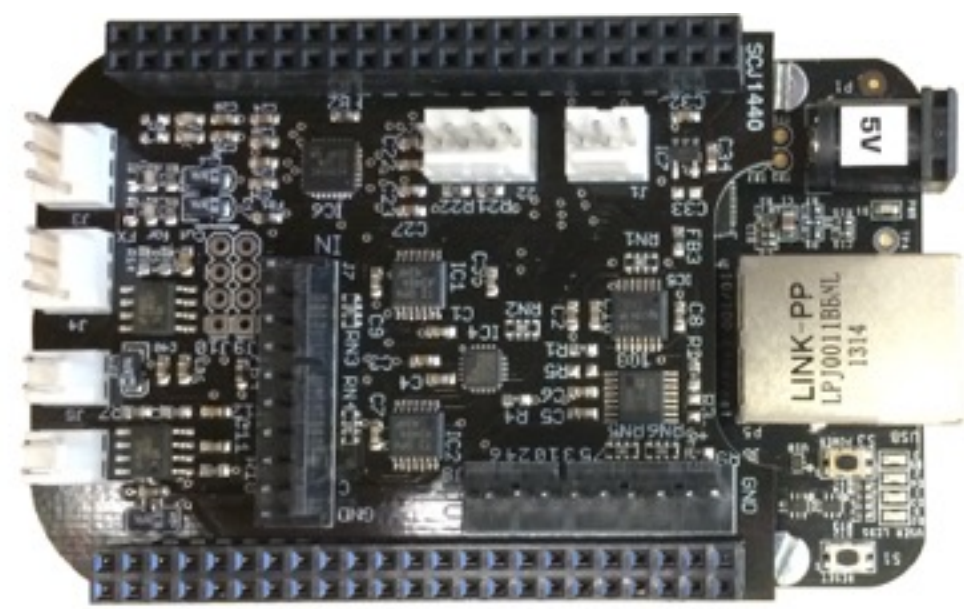
Programmable using **C/C++ or Pd**

Designed for **musical instruments and live audio**

bela

hardware





BeagleBone Black

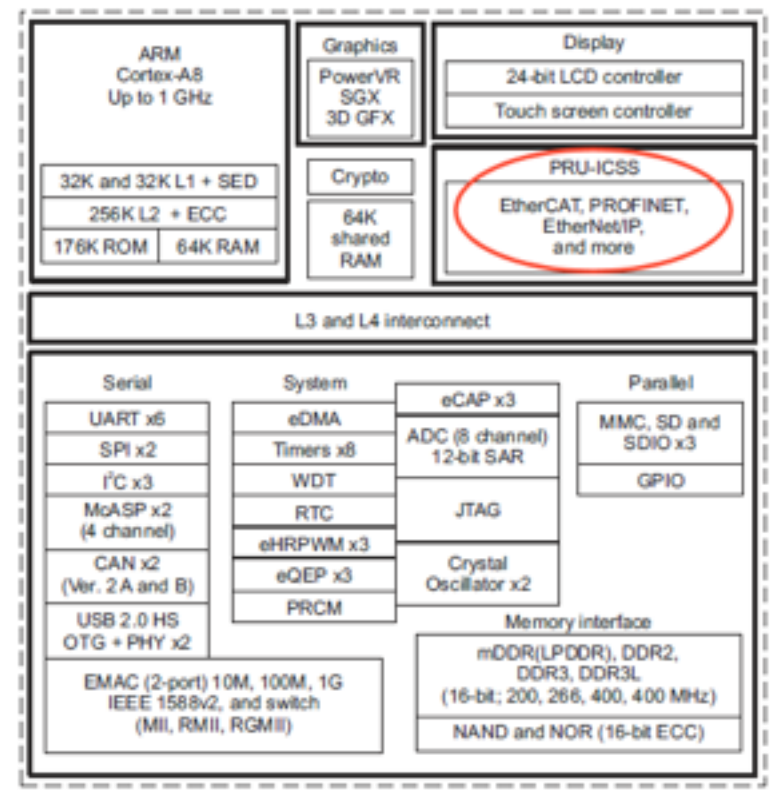
1GHz ARM Cortex-A8
NEON vector floating point
PRU real-time microcontrollers
512MB RAM

Custom Bela Cape

Stereo audio in + out
Stereo 1.1W speaker amps
8x 16-bit analog in + out
16x digital in/out

bela

software



Xenomai Linux kernel

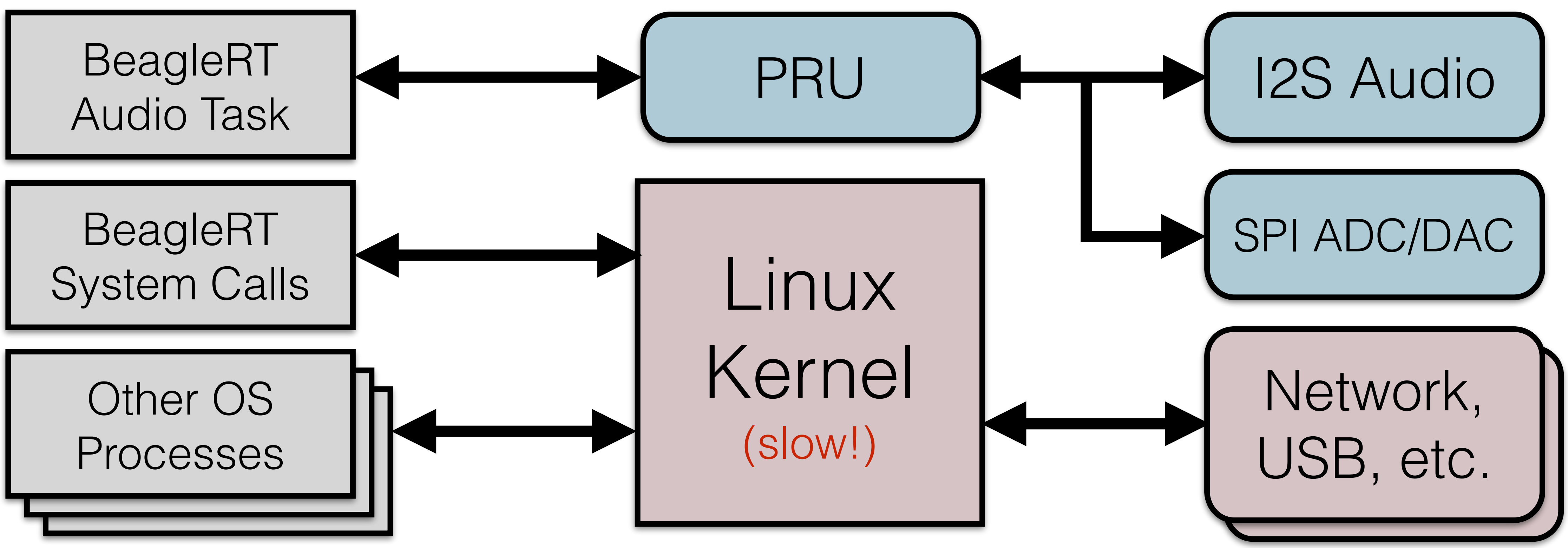
Debian distribution
Xenomai hard real-time
extensions

C++ programming API

Uses PRU for audio/sensors
Runs at higher priority
than kernel = *no dropouts*
Buffer sizes as small as **2**

Bela software

- Hard real-time environment using Xenomai Linux kernel extensions
- Use BeagleBone Programmable Realtime Unit (PRU) to write straight to hardware



- Sample all matrix ADCs and DACs at half audio rate (22.05kHz)
- Buffer sizes as small as 2 samples (90µs latency)

Materials

what you need to get started...

- **BeagleBone Black** (BBB)
- **Bela Cape**
- **SD card** with Bela image
- 3.5mm headphone jack **adapter cable**
- **Mini-USB cable** (to attach BBB to computer)
- Also useful for hardware hacking: **breadboard**, **jumper wires**, etc.

Step 1

install BBB drivers and Bela software

BeagleBone Black drivers:

(already installed on lab machines)

http://beagleboard.org

Bela code (for later today):

http://beaglert.cc --> Downloads --> bela_4-12-2015.zip

Bela code (in general):

http://beaglert.cc --> Repository

instructions:

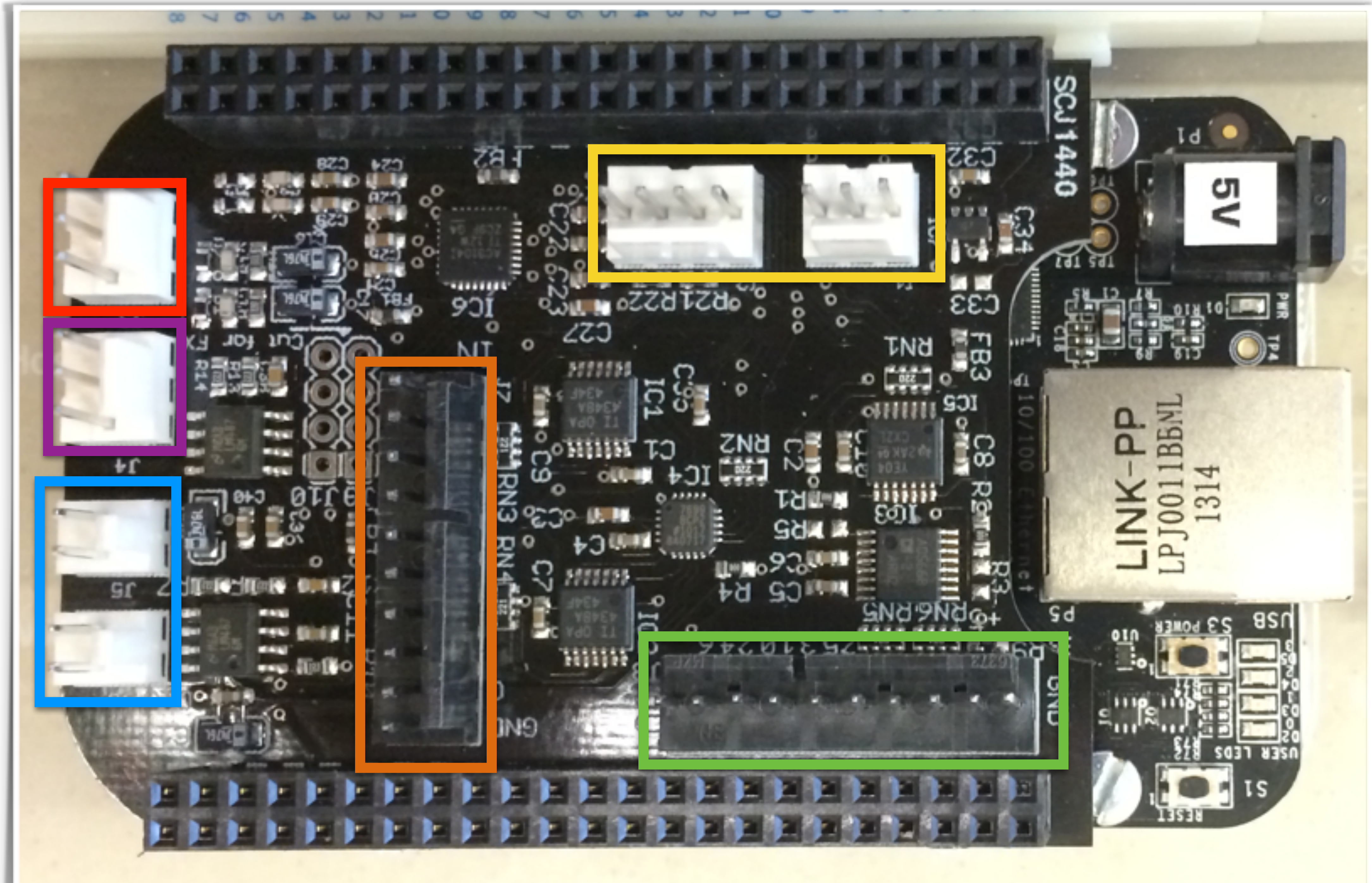
http://beaglert.cc --> Wiki --> Getting Started

Step 2

build a project

1. **Web interface**: http://192.168.7.2:3000
   Edit and compile code on the board
2. **Build scripts** (within repository)
   Edit code on your computer; build on the board
   No special tools needed except a text editor
3. **Eclipse** and cross-compiler (http://eclipse.org)
   Edit and compile on your computer; copy to board
4. **Heavy Pd-to-C compiler** (https://enzienaudio.com)
   Make audio patches in Pd-vanilla, translate to C and compile on board

Bela Cape


I2C and GPIO
Audio In
Audio Out
(headphone)
Speakers
8-ch. 16-bit ADC
8-ch. 16-bit DAC

API introduction

- In **`render.cpp`**....
- Three main functions:
- `setup()`
  runs once at the beginning, before audio starts
  gives channel and sample rate info
- `render()`
  called repeatedly by Bela system ("callback")
  passes input and output buffers for audio and sensors
- `cleanup()`
  runs once at end
  release any resources you have used

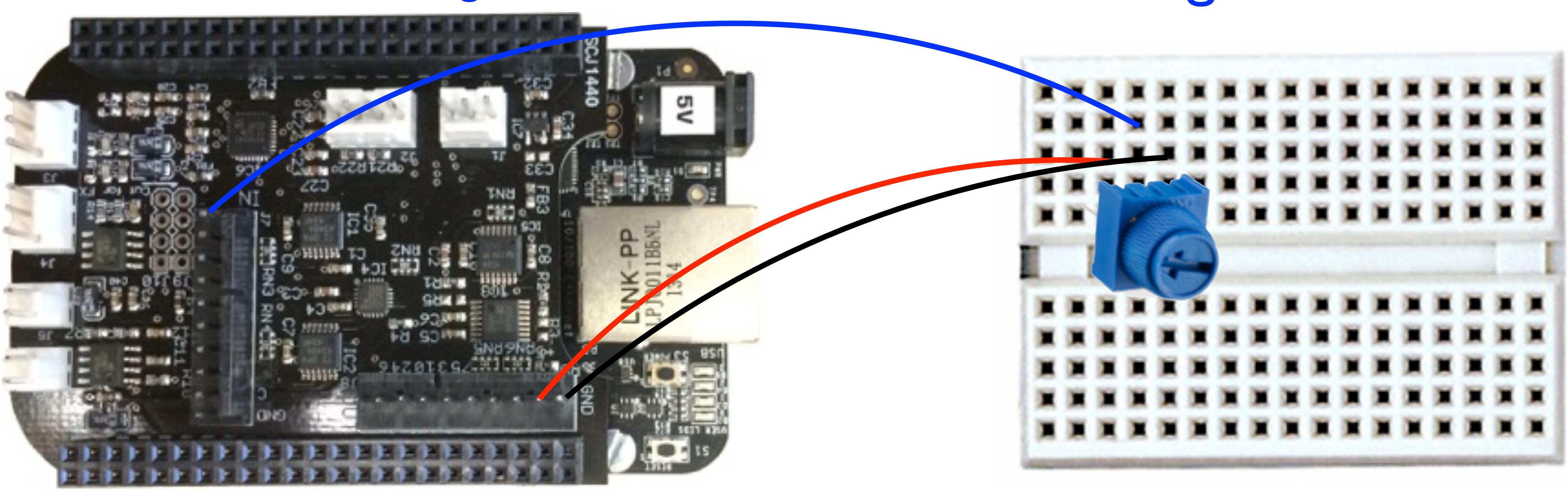
Connect a Potentiometer

a.k.a. a "pot" or knob

Interactive pinout: http://www.astridbin.com/bbb_diagram/

The pot has 3 pins
5V and GND on the outside
Bela analog in in the middle

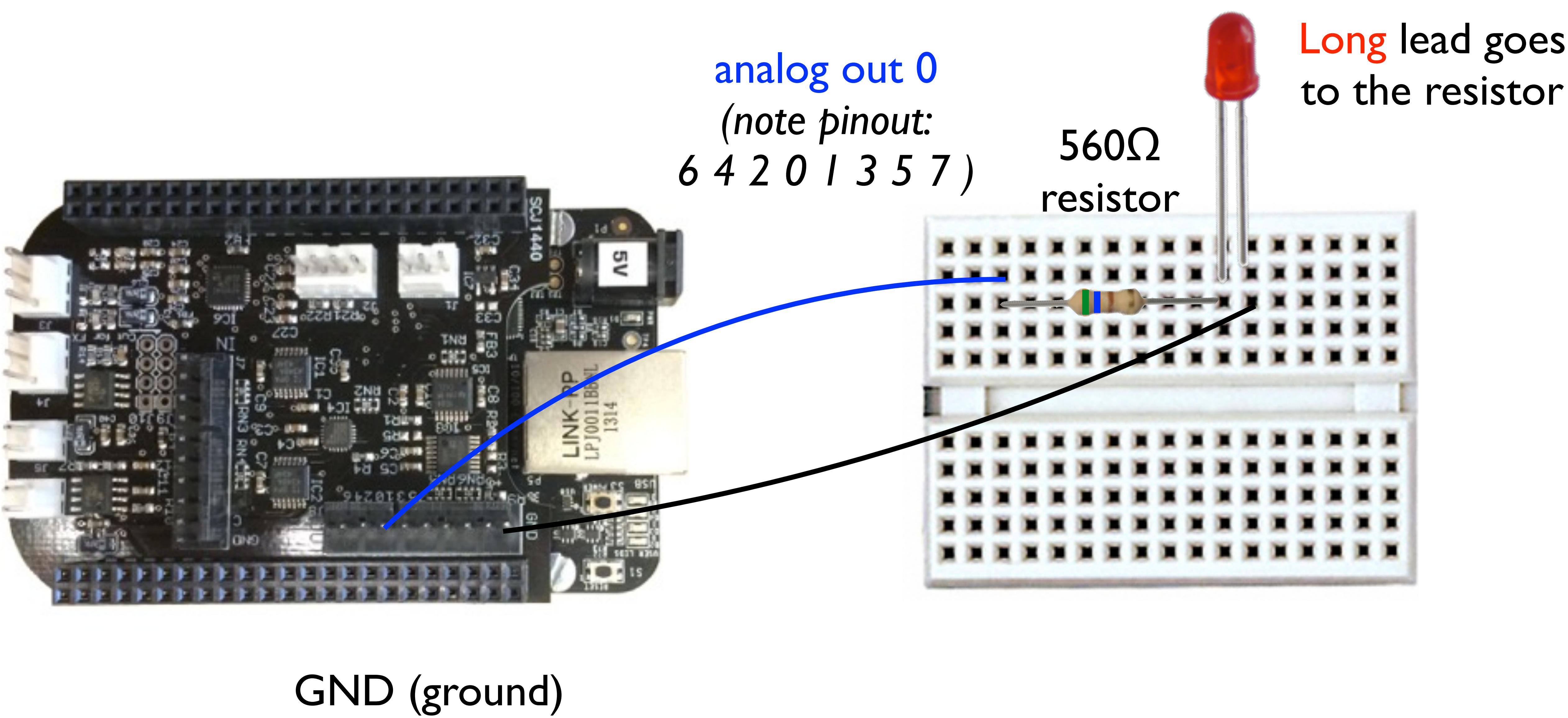
analog in 0



5V GND (ground)

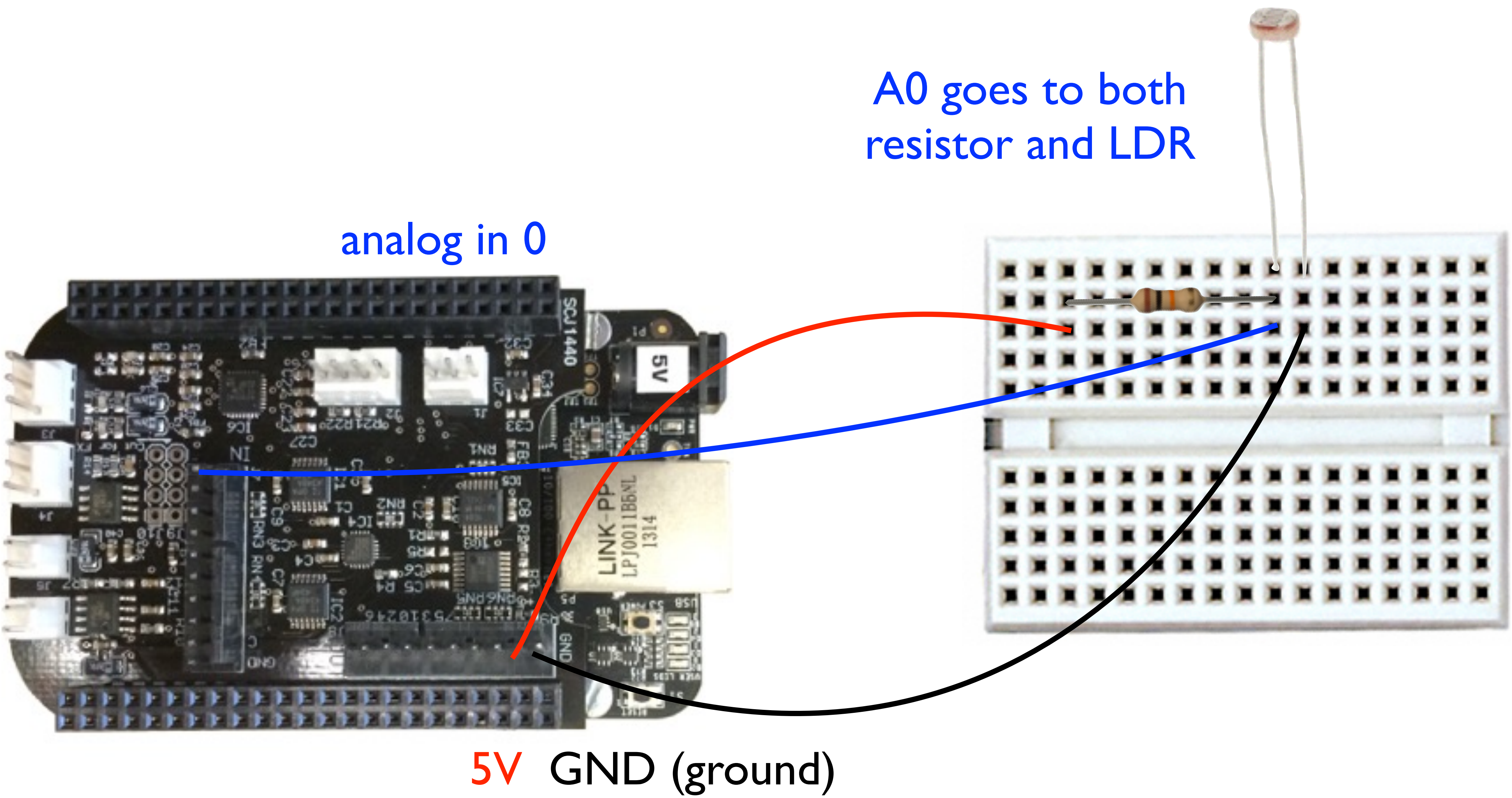
Connect an LED*

* Light-Emitting Diode

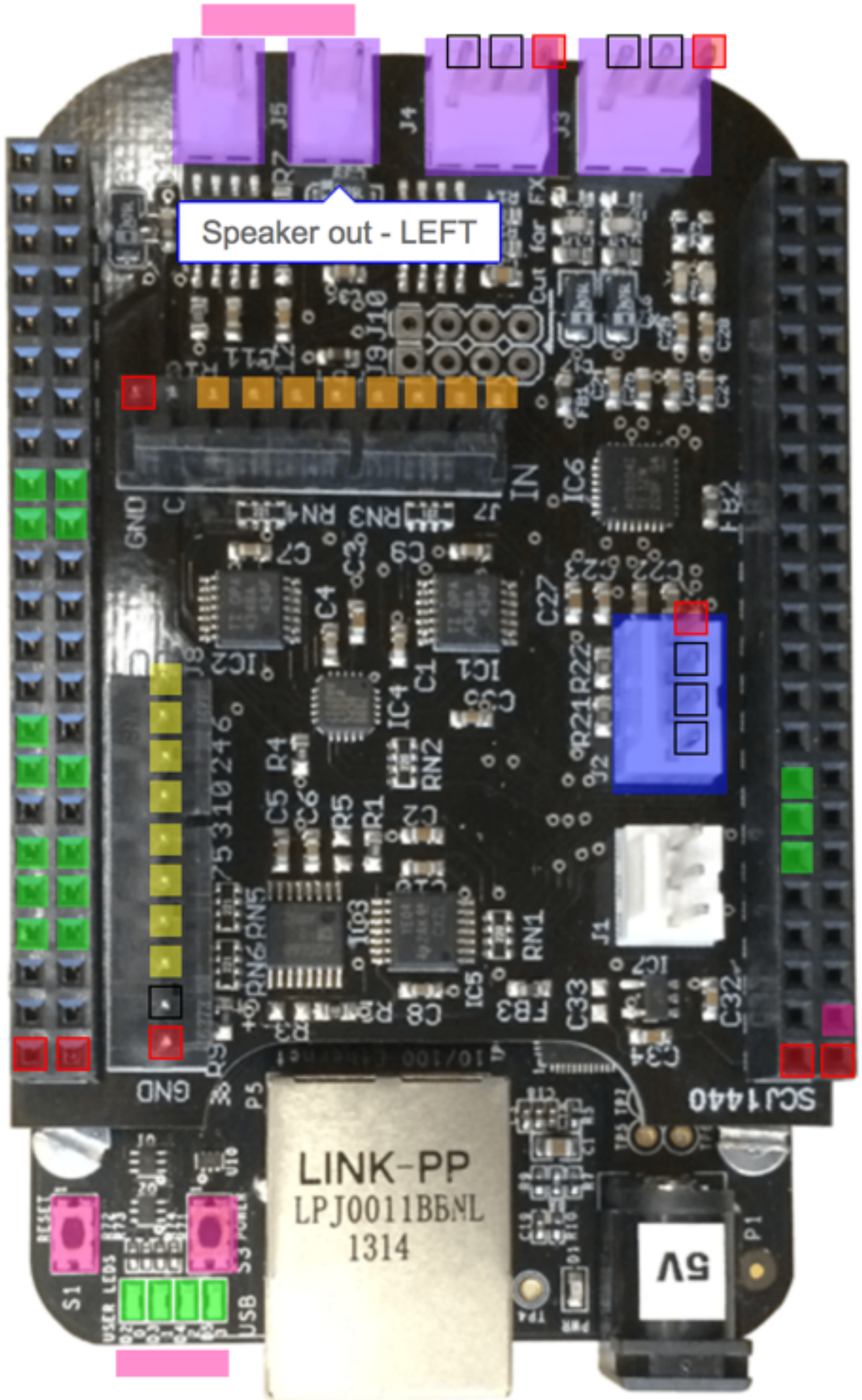


Connect a LDR/FSR*

* Light-Dependent Resistor / Force-Sensing Resistor



Green pins can be used for digital I/O



API introduction

- In **`render.cpp`**....
- Three main functions:
- **`setup()`**
  runs once at the beginning, before audio starts
  gives channel and sample rate info
- **`render()`**
  called repeatedly by Bela system ("callback")
  passes input and output buffers for audio and sensors
- **`cleanup()`**
  runs once at end
  release any resources you have used

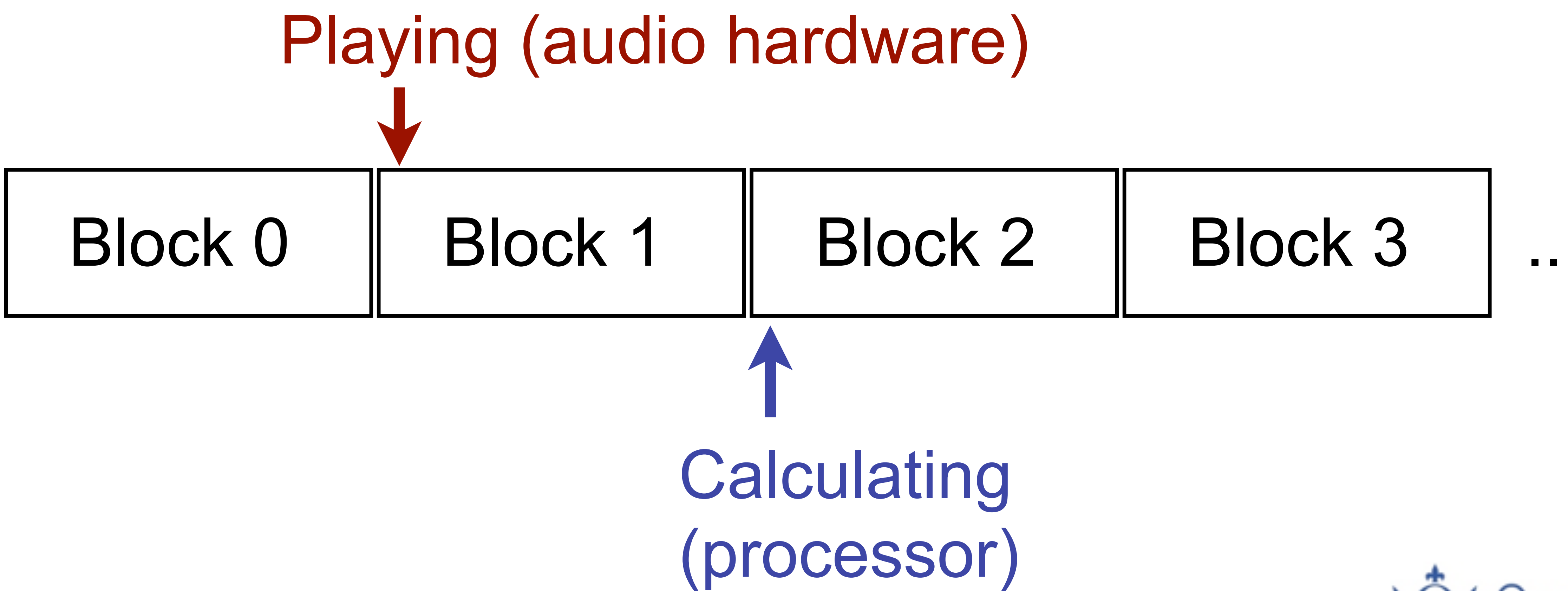
Real-time audio

- Suppose we have code that runs offline
  - (non-real time)
- Our goal is to re-implement it online (real time)
  - Generate audio as we need it!
  - Why couldn't we just generate it all in advance, and then play it when we need it?
- Digital audio is composed of samples
  - 44100 samples per second in our example
  - That means we need a new sample every 1/44100 seconds (about every 23μs)
  - So option #1 is to run a short bit of code every sample whenever we want to know what to play next
  - What might be some drawbacks of this approach?
    - Can we guarantee we'll be ready for each new sample?

Queen Mary
University of London

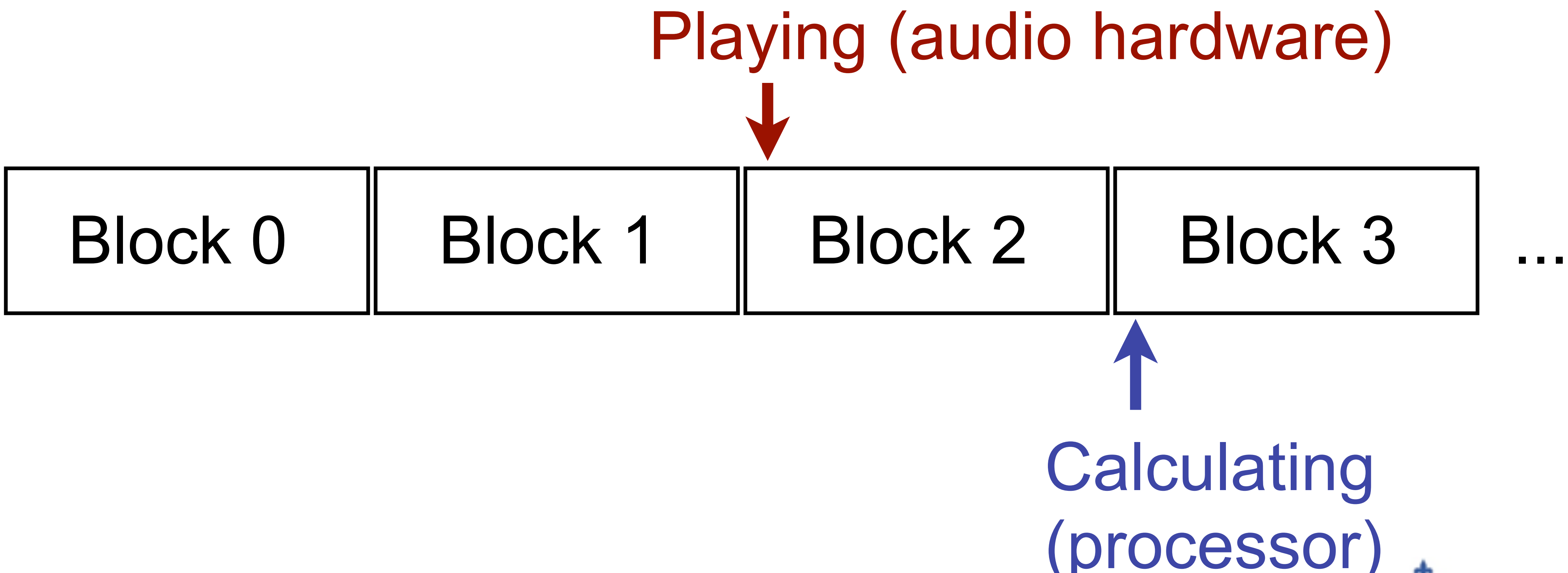
Block-based processing

- Option #2: Process in blocks of several samples
  - Basic idea: generate enough samples to get through the next few milliseconds
  - Typical block sizes: 32 to 1024 samples
    - Usually a power of 2 for reasons having to do with hardware
  - While the audio hardware is busy playing one block, we can start calculating the next one so it's ready on time:



Block-based processing

- Option #2: Process in blocks of several samples
  - Basic idea: generate enough samples to get through the next few milliseconds
  - Typical block sizes: 32 to 1024 samples
    - Usually a power of 2 for reasons having to do with hardware
  - While the audio hardware is busy playing one block, we can start calculating the next one so it's ready on time:



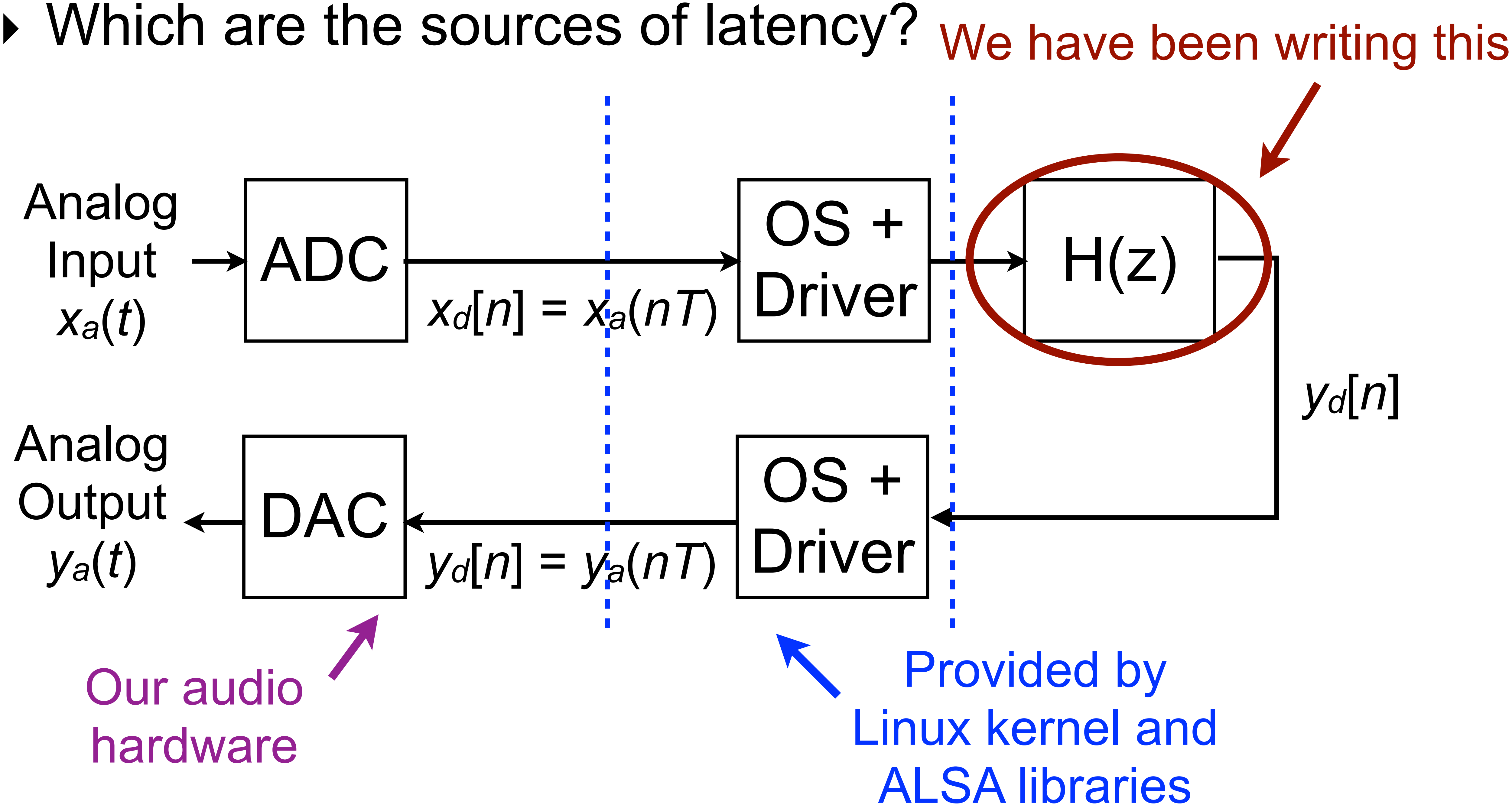
Queen Mary University of London

Block-based processing

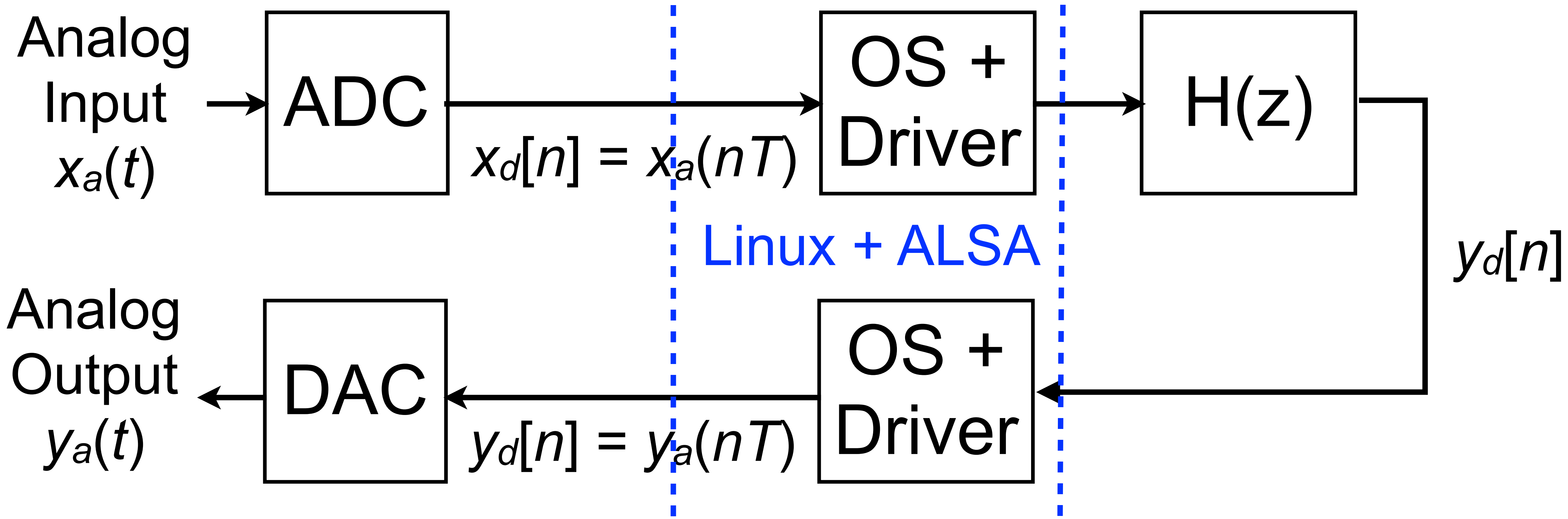
- Advantages of blocks over individual samples
  - We need to run our function less often
  - We always generate one block ahead of what is actually playing
  - Suppose one block of samples lasts 5ms, and running our code takes 1ms
  - Now, we can tolerate a delay of up to 4ms if the OS is busy with other tasks
  - Larger block size = can tolerate more variation in timing
- What is the disadvantage?
  - Latency (delay)

Latency

- Primary tradeoff for buffering: latency
  - There will be a delay from input to output
- Let's consider a full-duplex system (in and out)
  - Which are the sources of latency?



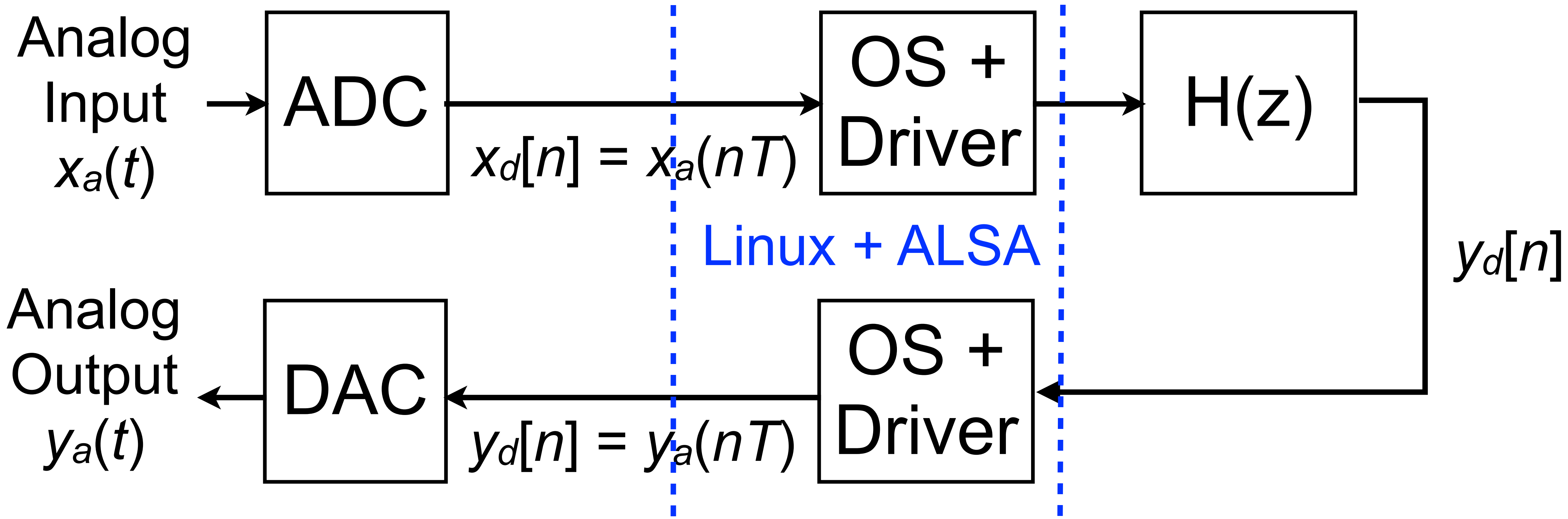
Latency: the role of buffering



- Block-based processing introduces latency
  - This is in addition to whatever was generated by H(z)
- On input side: how is a block of samples created?
  - For block of size N: we wait until N samples have arrived from ADC....

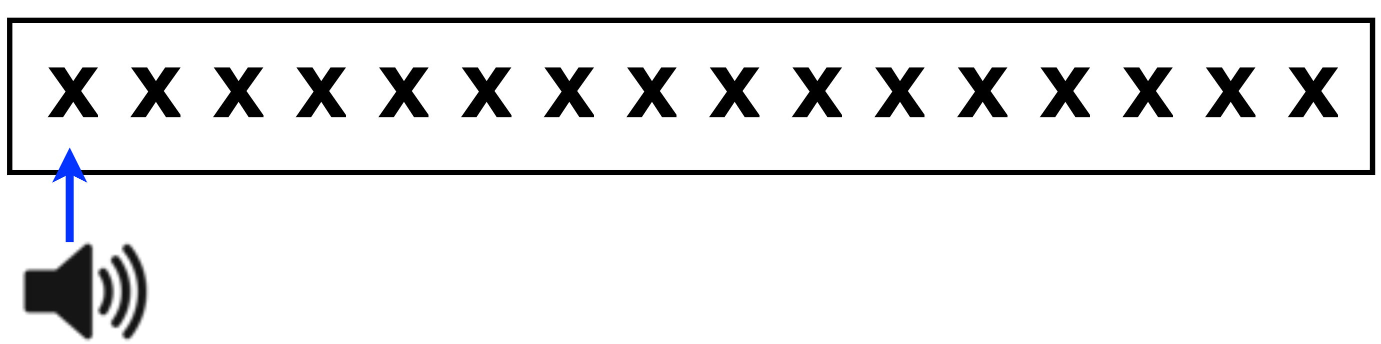
In other words: first sample in the block is already N samples old by the time we get it

Latency: the role of buffering

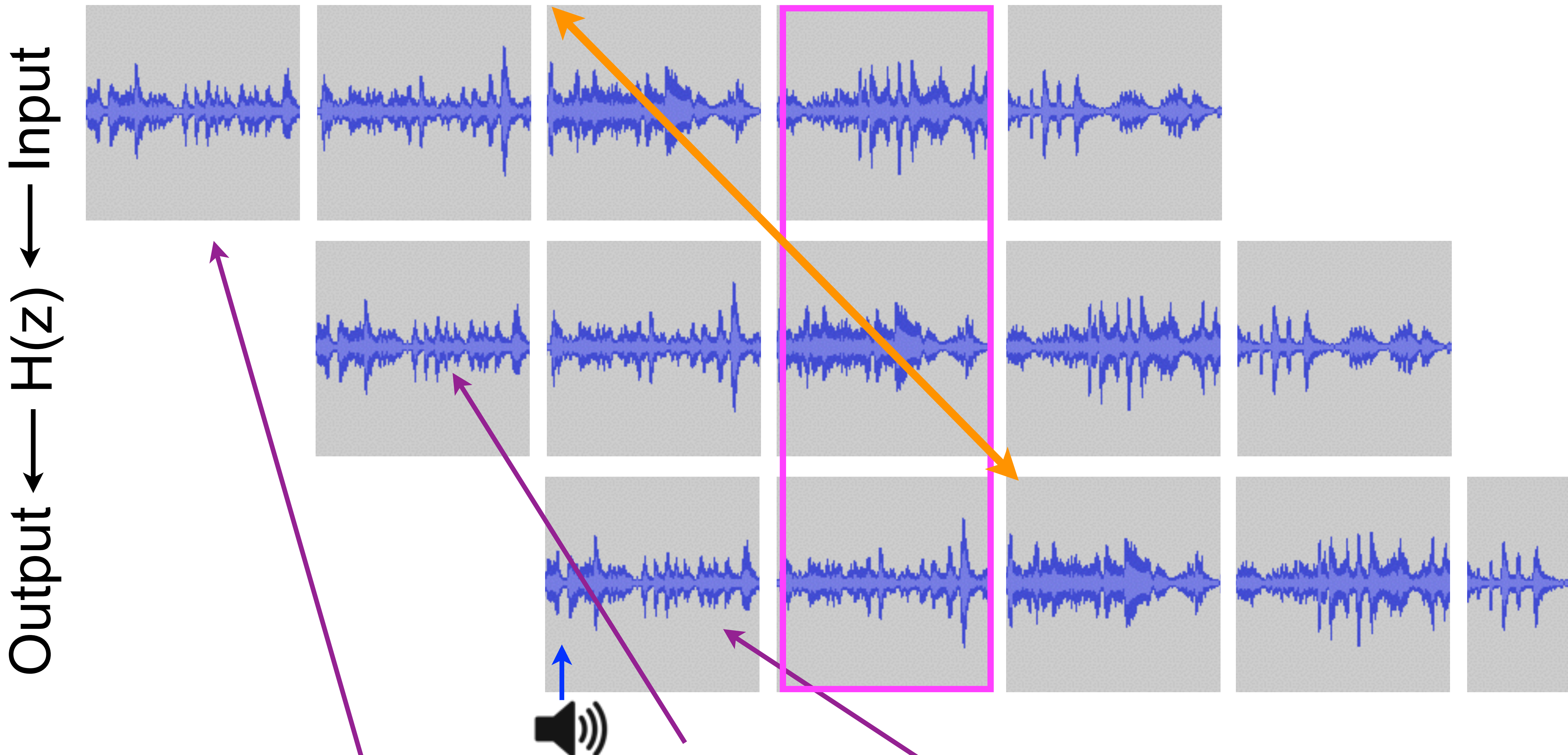


- On output side: how is a block played by DAC?
  - We can only start playing once the block arrives!
  - So how long until the last sample is played?
    - N samples after the the block is sent to the hardware

Block



Buffering illustration



1. First we fill up a buffer of samples

2. We process this buffer while the next one fills up

3. Next time, we send this buffer to the output

At any given time, we are reading from ADC, processing a block, and writing to DAC

Total latency is 2x buffer length

API introduction

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

- Sensor ("matrix" = ADC+DAC) data is gathered automatically alongside audio
- Audio runs at 44.1kHz; sensor data at 22.05kHz
- `context` holds buffers plus information on number of frames and other info
- Your job as programmer: render one buffer of audio and sensors and finish as soon as possible!
- API documentation: http://beaglert.cc

First test program

```
float gPhase; /* Phase of the oscillator (global variable) */

void render(BeagleRTContext *context, void *userData)
{
    /* Iterate over the number of audio frames */
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        /* Calculate the output sample based on the phase */
        float out = 0.8f * sinf(gPhase);

        /* Update the phase according to the frequency */
        gPhase += 2.0 * M_PI * gFrequency * gInverseSampleRate;
        if(gPhase > 2.0 * M_PI)
            gPhase -= 2.0 * M_PI;

        /* Store the output in every audio channel */
        for(unsigned int channel = 0;
                channel < context->audioChannels; channel++)
            context->audioOut[n * context->audioChannels
                                + channel] = out;
    }
}
```

This runs once per block

This runs once per sample in the block (audioFrames gives the number)

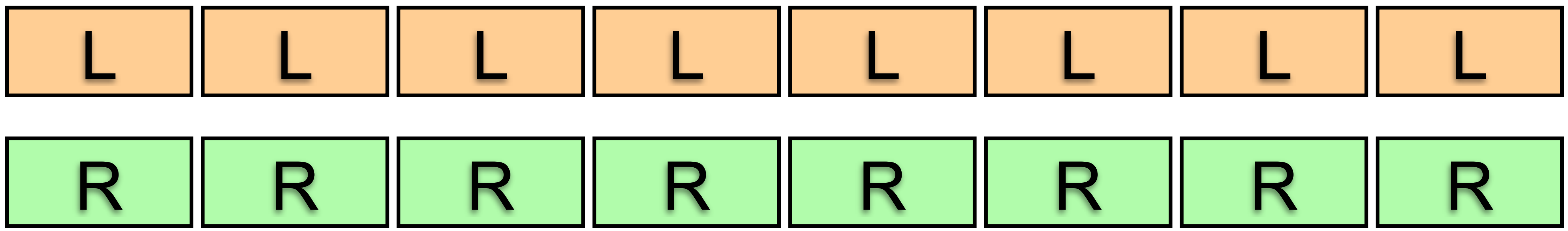
This runs twice per frame, once for each channel

One-dimensional array holding interleaved audio data

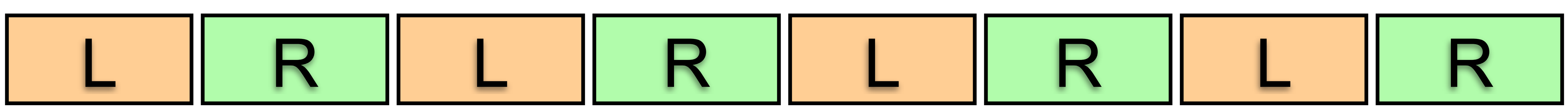
Queen Mary University of London

Interleaving

- Two ways for multichannel audio to be stored
  - Way 1: Separate memory buffers per channel



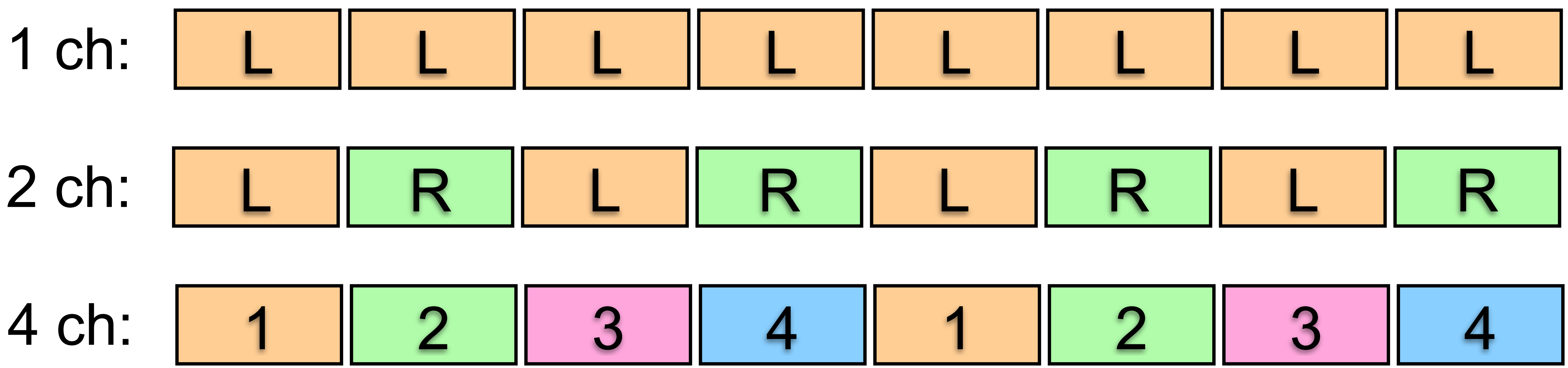
    - This is known as non-interleaved format
    - Typically presented in C as a two-dimensional array: `float **sampleBuffers`
  - Way 2: One memory buffer for all channels
    - Alternating data between channels



    - This is known as interleaved format
    - Typically presented in C as a one-dimensional array: `float *sampleBuffer`

Interleaving

- We accessed non-interleaved data like this:
  - `float in = sampleBuffers[channel][n];`
- How do we do the same thing with interleaving?
  - `float in = sampleBuffers[***what goes here?***];`
  - What else do we need to know?
    - Number of channels



  - `float in = sampleBuffers[numChannels*n + channel];`
  - Each sample advances numChannels in the buffer
    - The offset tells us which channel we're reading

First test program

```
float gPhase; /* Phase of the oscillator (global variable) */

void render(BeagleRTContext *context, void *userData)
{
    /* Iterate over the number of audio frames */
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        /* Calculate the output sample based on the phase */
        float out = 0.8f * sinf(gPhase);

        /* Update the phase according to the frequency */
        gPhase += 2.0 * M_PI * gFrequency * gInverseSampleRate;
        if(gPhase > 2.0 * M_PI)
            gPhase -= 2.0 * M_PI;

        /* Store the output in every audio channel */
        for(unsigned int channel = 0;
                channel < context->audioChannels; channel++)
            context->audioOut[n * context->audioChannels
                                    + channel] = out;
    }
}
```

This runs once per block

This runs once per sample in the block (audioFrames gives the number)

This runs twice per frame, once for each channel

One-dimensional array holding interleaved audio data

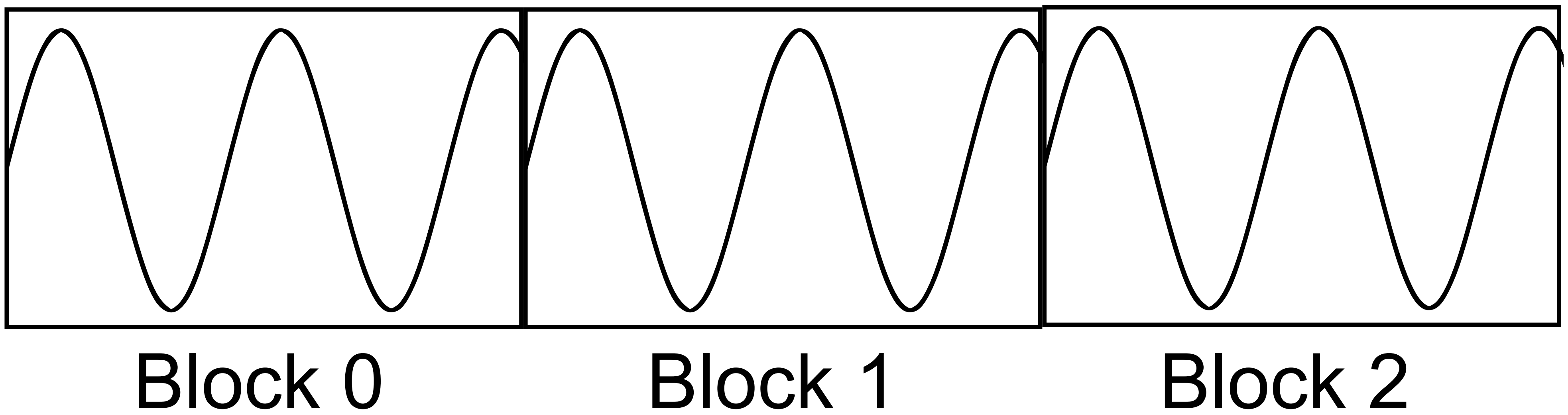
Queen Mary University of London

Blocks and phase: task

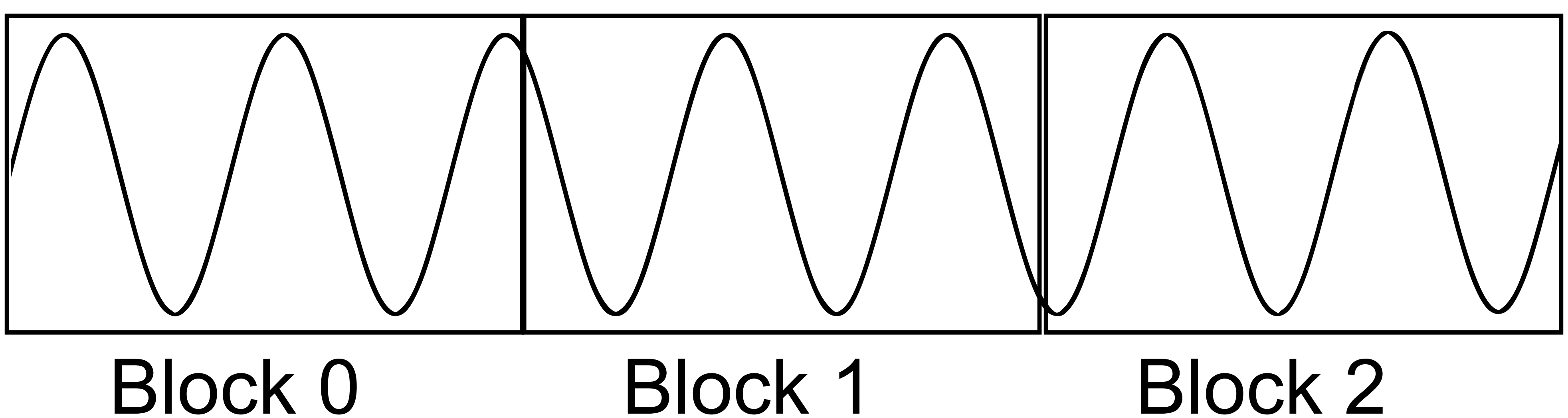
- Need to preserve state between calls to render()
  - When you call `render()` a second time, it should remember where it left off the first time
  - But local variables in the function all disappear when the function returns!
  - Solution: use global variables to save the state
    - Okay, cleaner solutions exist: keep a structure that you pass by pointer as an argument to `render()`. Save your state there.
    - Or in C++, use instance variables (variables that are declared in the class rather than within a method). But we'll save that for later.

Blocks and phase

- If we don't store phase in a global variable, we get:



- But what we want is this:



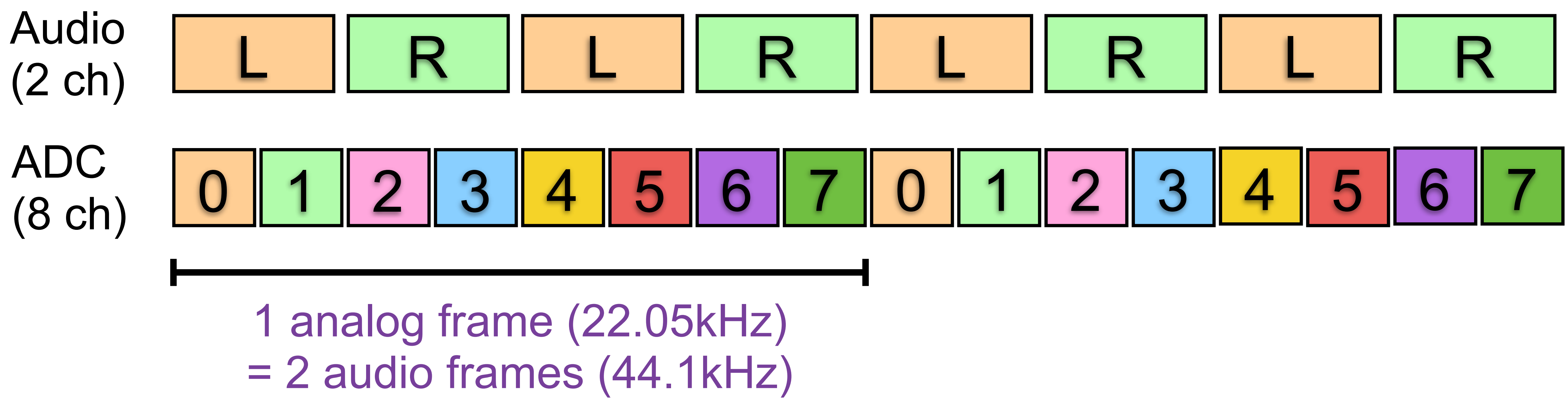
First test program

This remembers where we left off

```
float gPhase; /* Phase of the oscillator (global variable) */

void render(BeagleRTContext *context, void *userData)
{
    /* Iterate over the number of audio frames */
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        /* Calculate the output sample based on the phase */
        float out = 0.8f * sinf(gPhase);

        /* Update the phase according to the frequency */
        gPhase += 2.0 * M_PI * gFrequency * gInverseSampleRate;
        if(gPhase > 2.0 * M_PI)
            gPhase -= 2.0 * M_PI;

        /* Store the output in every audio channel */
        for(unsigned int channel = 0;
                channel < context->audioChannels; channel++)
            context->audioOut[n * context->audioChannels
                                    + channel] = out;
    }
}
```

This updates the phase each sample and keeps it in the 0 to 2π range

Analog input data format



- Data type is `float`: just like audio
  - But range is 0.0 to 1.0
    - This is internally converted from raw values 0 to 65535
  - Compare this to audio, which is -1.0 to 1.0

Analog input

```
float gPhase;
float gInverseSampleRate;           /* Pre-calculated for convenience */
int gAudioFramesPerAnalogFrame;

extern int gSensorInputFrequency;   /* Which analog pin controls frequency */
extern int gSensorInputAmplitude;   /* Which analog pin controls amplitude */

void render(BeagleRTContext *context, void *userData)
{
    float frequency = 440.0;
    float amplitude = 0.8;

    for(unsigned int n = 0; n < context->audioFrames; n++) {
        /* There are twice as many audio frames as matrix frames since
           audio sample rate is twice as high */
        if(!(n % gAudioFramesPerAnalogFrame)) {
            /* Every other audio sample: update frequency and amplitude */
            frequency = map(analogReadFrame(context,
                                n/gAudioFramesPerAnalogFrame,
                                gSensorInputFrequency),
                        0, 1, 100, 1000);
            amplitude = analogReadFrame(context,
                        n/gAudioFramesPerAnalogFrame,
                        gSensorInputAmplitude);
        }

        float out = amplitude * sinf(gPhase);

        for(unsigned int channel = 0; channel < context->audioChannels; channel++)
            context->audioOut[n * context->audioChannels + channel] = out;

        gPhase += 2.0 * M_PI * frequency * gInverseSampleRate;
        if(gPhase > 2.0 * M_PI)
            gPhase -= 2.0 * M_PI;
    }
}
```

This runs every other sample

Read the analog input at the specified frame

Map the 0-1 input range to a frequency range

Digital I/O

```
void render(BeagleRTContext *context, void *userData)
{
    static int count = 0; // counts elapsed samples
    float interval = 0.5; // how often to toggle the LED (in seconds)
    static int status = GPIO_LOW;

    for(unsigned int n = 0; n < context->digitalFrames; n++) {
        /* Check if enough samples have elapsed that it's time to
           blink to the LED */
        if(count == context->digitalSampleRate * interval) {
            count = 0; // reset the counter
            if(status == GPIO_LOW) {
                /* Toggle the LED */
                digitalWriteFrame(context, n, P8_07, status);
                status = GPIO_HIGH;
            }
            else {
                /* Toggle the LED */
                digitalWriteFrame(context, n, P8_07, status);
                status = GPIO_LOW;
            }
        }

        /* Increment the count once per frame */
        count++;
    }
}
```

This runs once per digital frame

Write the digital output at the specified frame

To manage timing, count samples rather than using delays