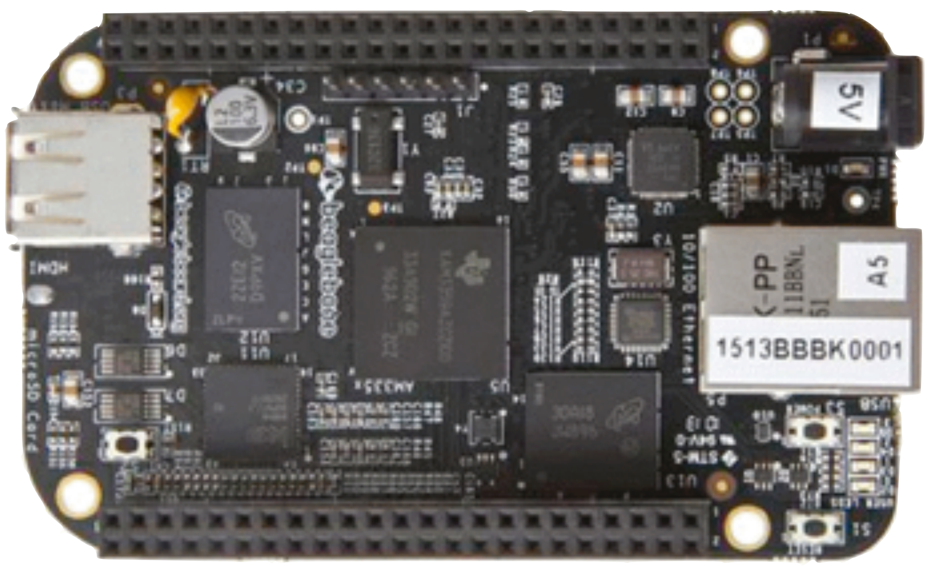
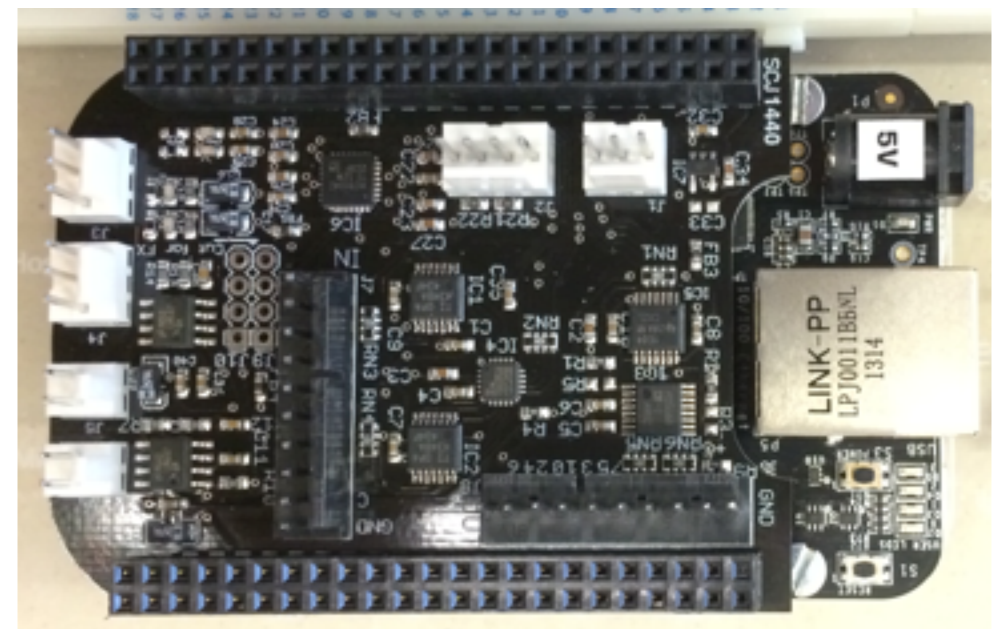
BeagleRT

hardware



BeagleBone Black

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

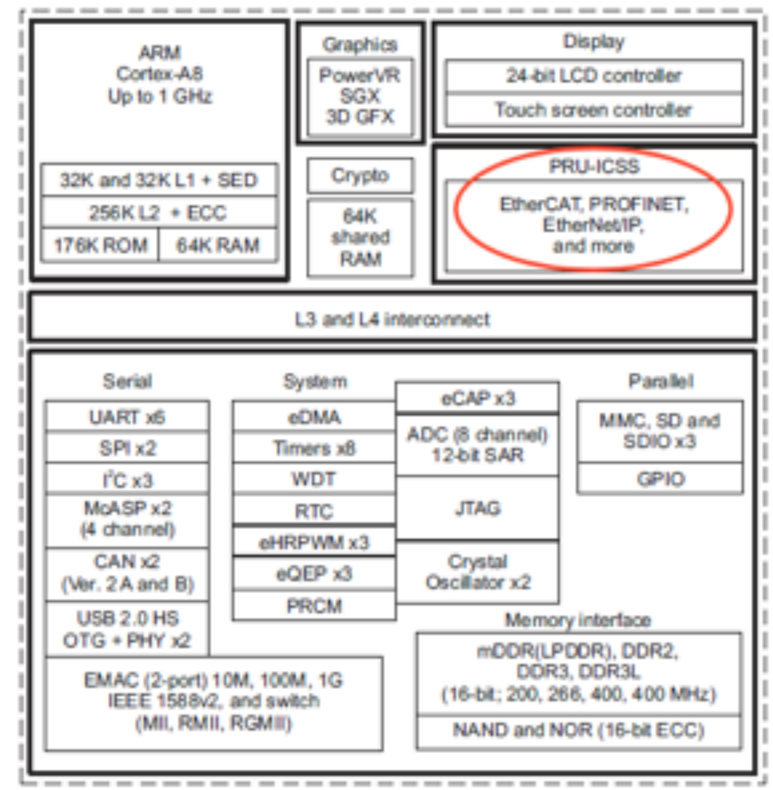


Custom BeagleRT Cape

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

BeagleRT

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**

BeagleRT

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**

Materials

what you need to get started...

- **BeagleBone Black** (BBB)
- **BeagleRT Cape**
- **SD card** with BeagleRT image
  (image can be downloaded from wiki at beaglert.cc)
- 3.5mm headphone jack **adapter cable**
- The D-Box already contains all of the above...
- **Mini-USB cable** (to attach BBB to computer)
- Also useful for hardware hacking: **breadboard**, **jumper wires**, etc.

Step 1

install BBB drivers and BeagleRT software

BeagleBone Black drivers:
http://beagleboard.org

BeagleRT code:
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

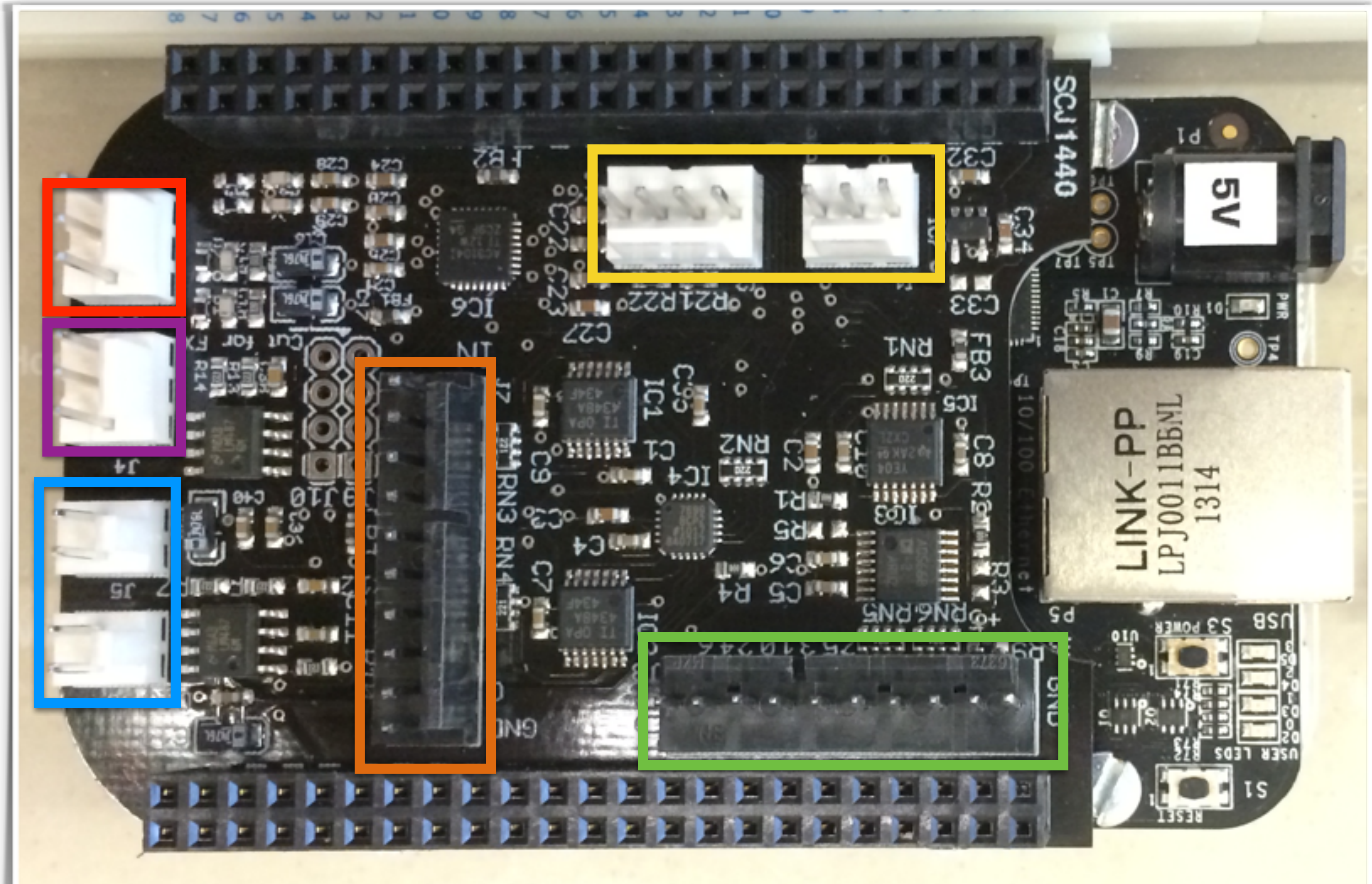
BeagleRT/D-Box Cape

I2C and GPIO

Audio In

Audio Out
(headphone)

Speakers



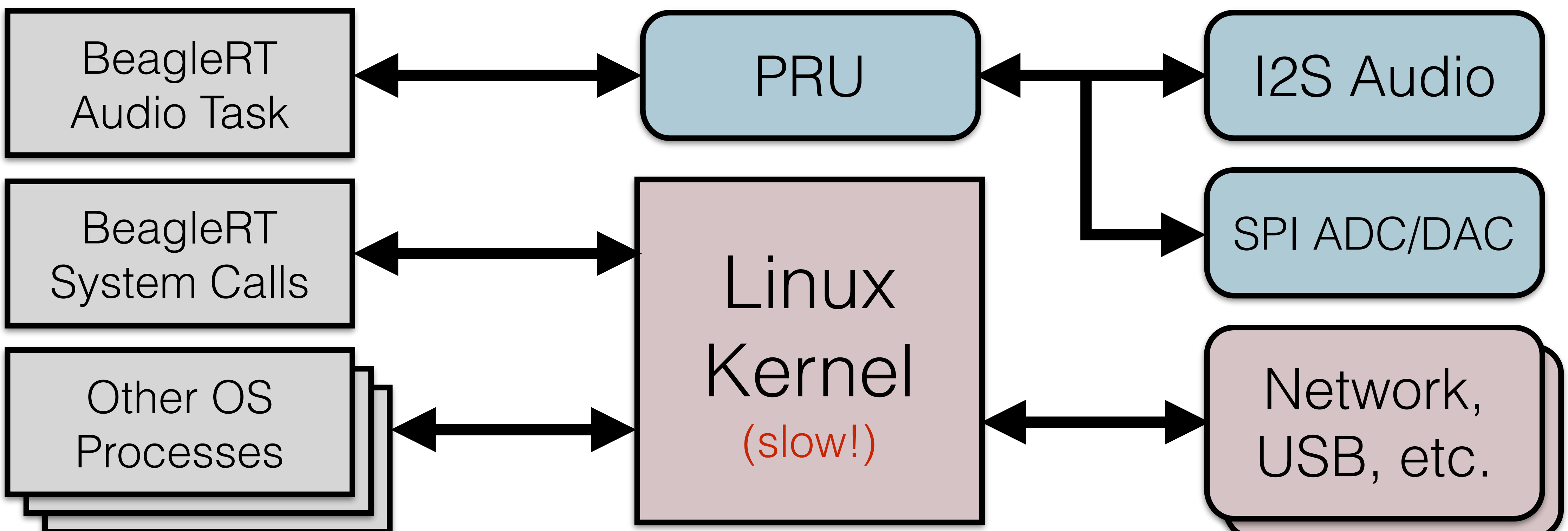
8-ch. 16-bit ADC

8-ch. 16-bit DAC

BeagleRT software

Xenomai

- 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)

API introduction

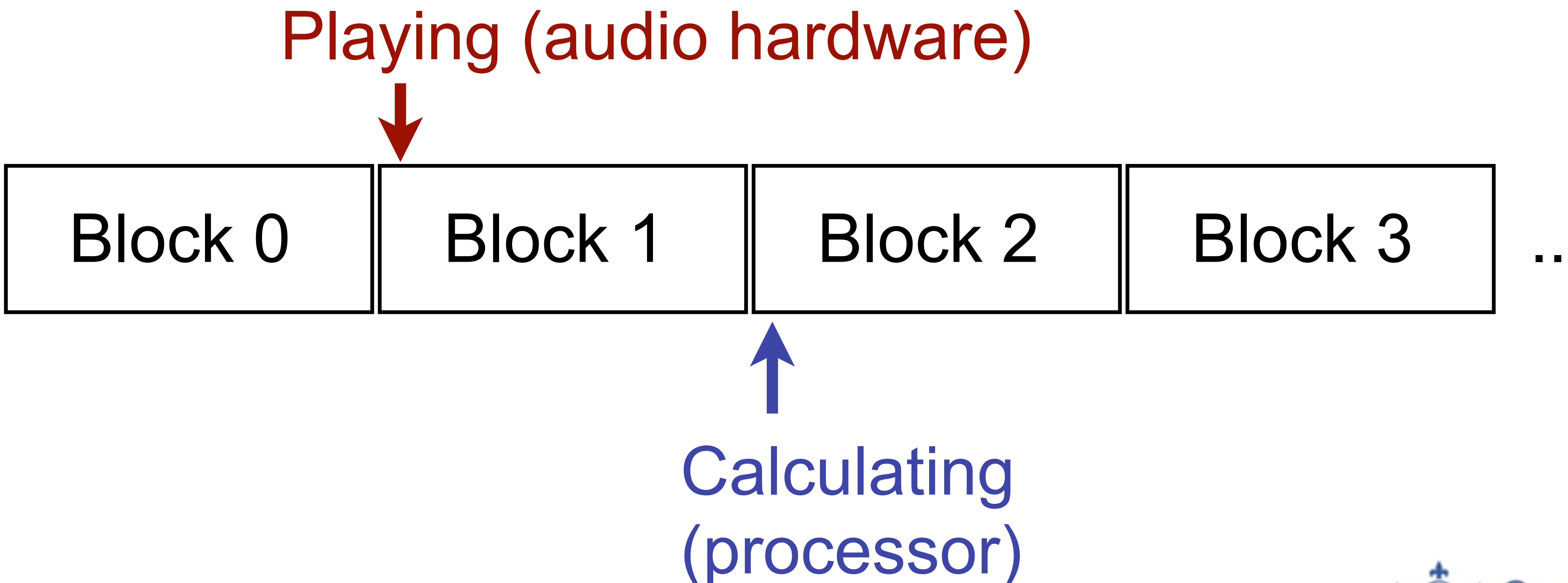
- In `render.cpp`....
- Three main functions:
- `setup()`
  runs once at the beginning, before audio starts
  gives channel and sample rate info
- `render()`
  called repeatedly by BeagleRT system ("callback")
  passes input and output buffers for audio and sensors
- `cleanup()`
  runs once at end
 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

Block-based processing

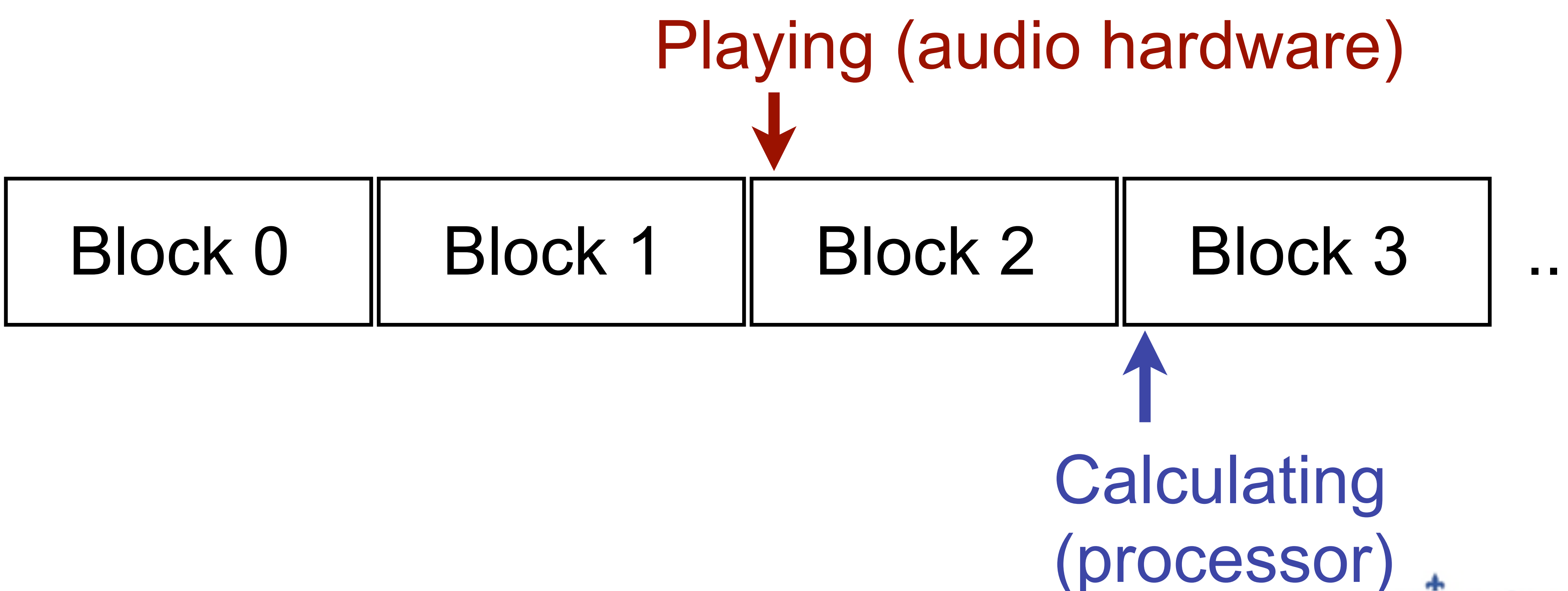
- Option #2: Process in blocks of several samples
  - Basic idea: generate enough samples to get through the next few milliseconds
  - Typical block sizes: 32 to 1024 samples
    - Usually a power of 2 for reasons having to do with hardware
  - While the audio hardware is busy playing one block, we can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



Queen Mary University of London

Block-based processing

- 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:

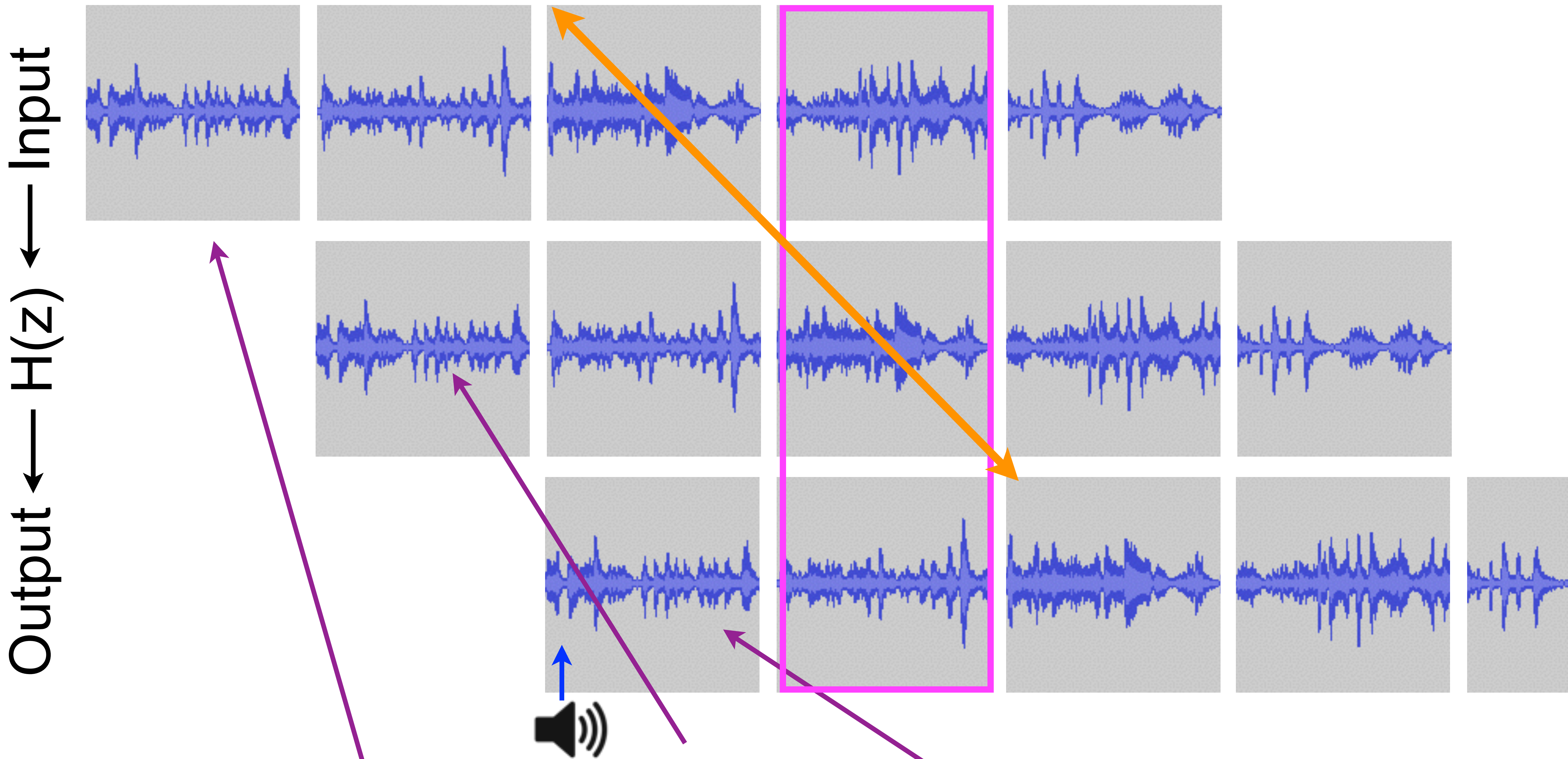


Queen Mary University of London

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)

Buffering illustration



1. First we fill up a buffer of samples

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

3. Next cycle, 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

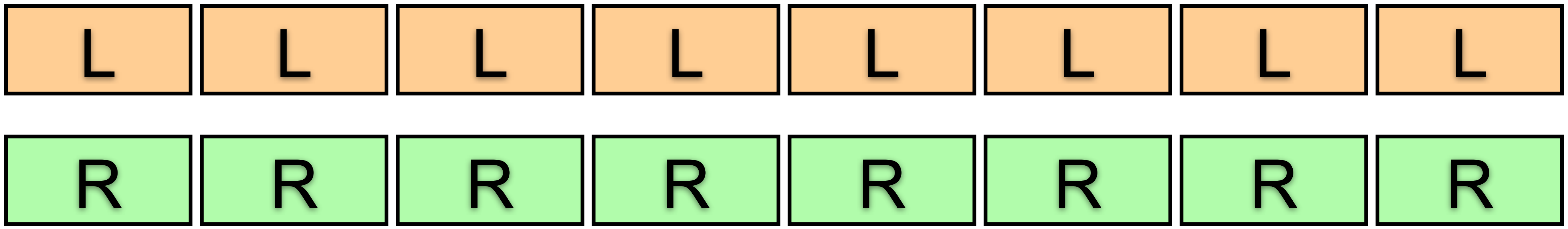
API introduction

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

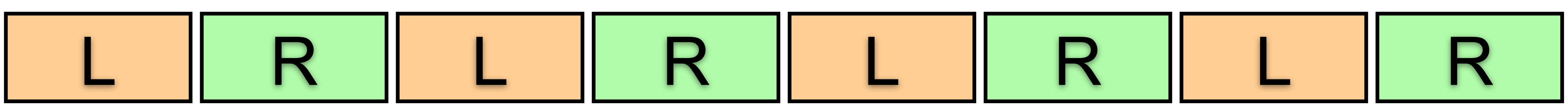
- Sensor ("matrix" = ADC+DAC) data is gathered automatically alongside audio
- Audio runs at 44.1kHz; sensor data at 22.05kHz
- `context` holds buffers plus information on number of frames and other info
- Your job as programmer: render one buffer of audio and sensors and finish as soon as possible!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

Interleaving

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



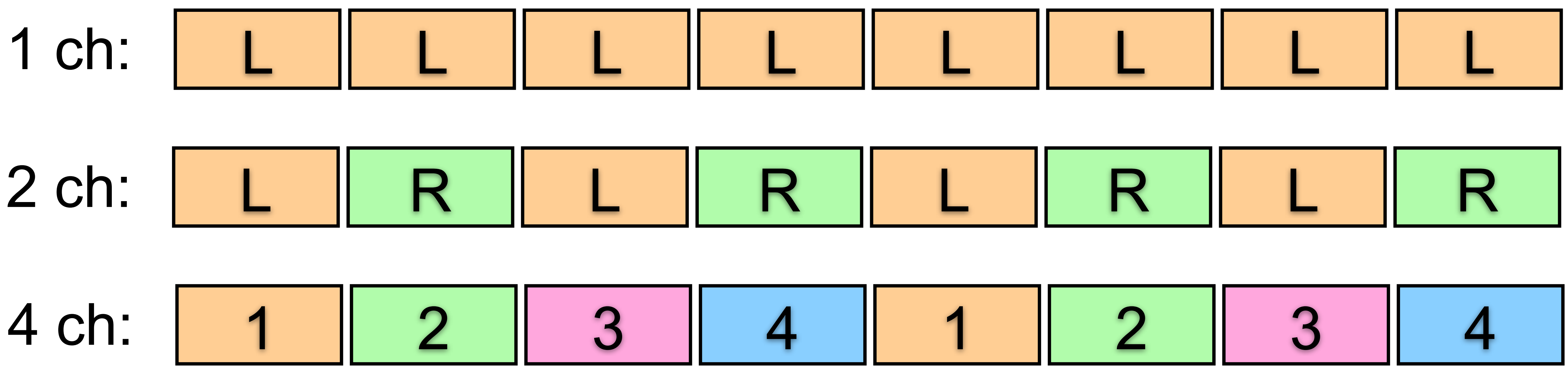
    - This is known as non-interleaved format
    - Typically presented in C as a two-dimensional array:
      `float **sampleBuffers`
  - Way 2: One memory buffer for all channels
    - Alternating data between channels



    - 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