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

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Timing in real time
Circular buffers
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Today

Music/audio topics

Oscillators
Samples
Wavetables
Control voltages
Gates and triggers
Filters
Metronomes and clocks
Delays and delay-based effects
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Lecture 16: MIDI part 2

What you'll learn today:

Handling the MIDI pitch wheel
Handling MIDI control change messages
Low-frequency oscillators and portamento

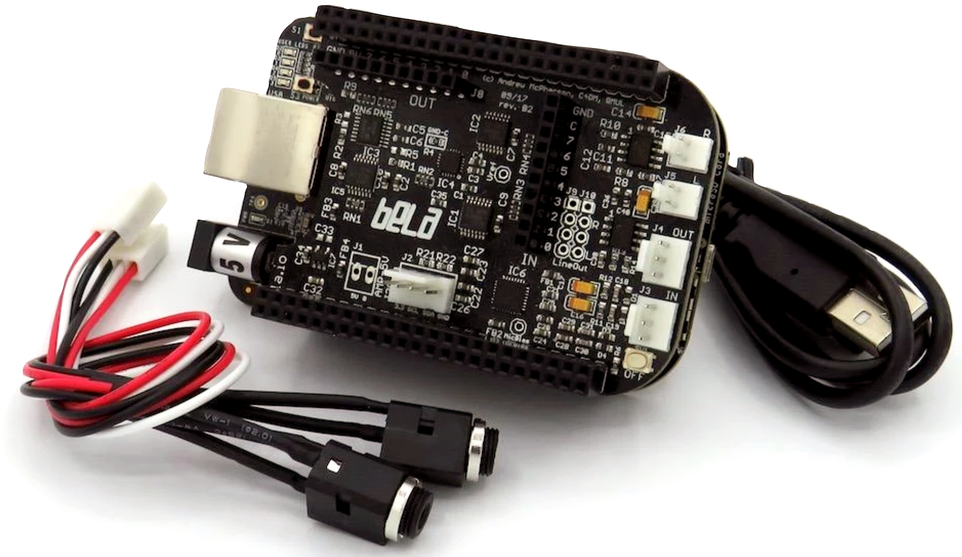
What you'll make today:

Analog-style monophonic MIDI synth

Companion materials:

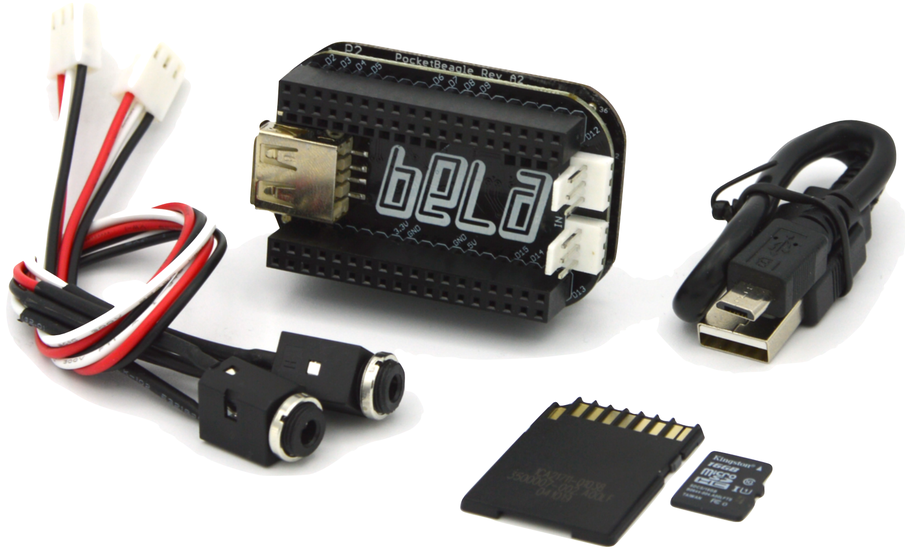
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

or



Bela Mini Starter Kit

[shop.bela.io]

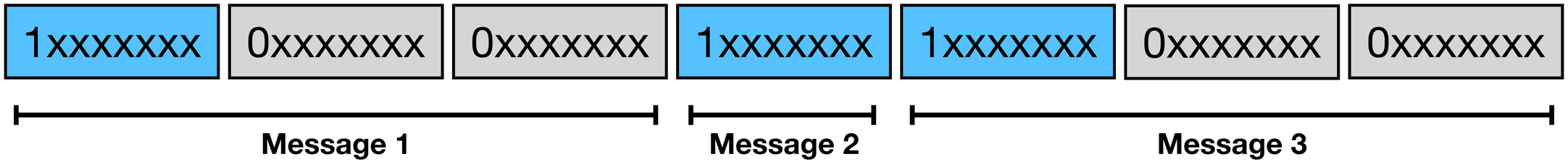
Also needed for this lecture:

or virtual keyboard software:

MIDIKeys (Mac)
vmpk (cross-platform)

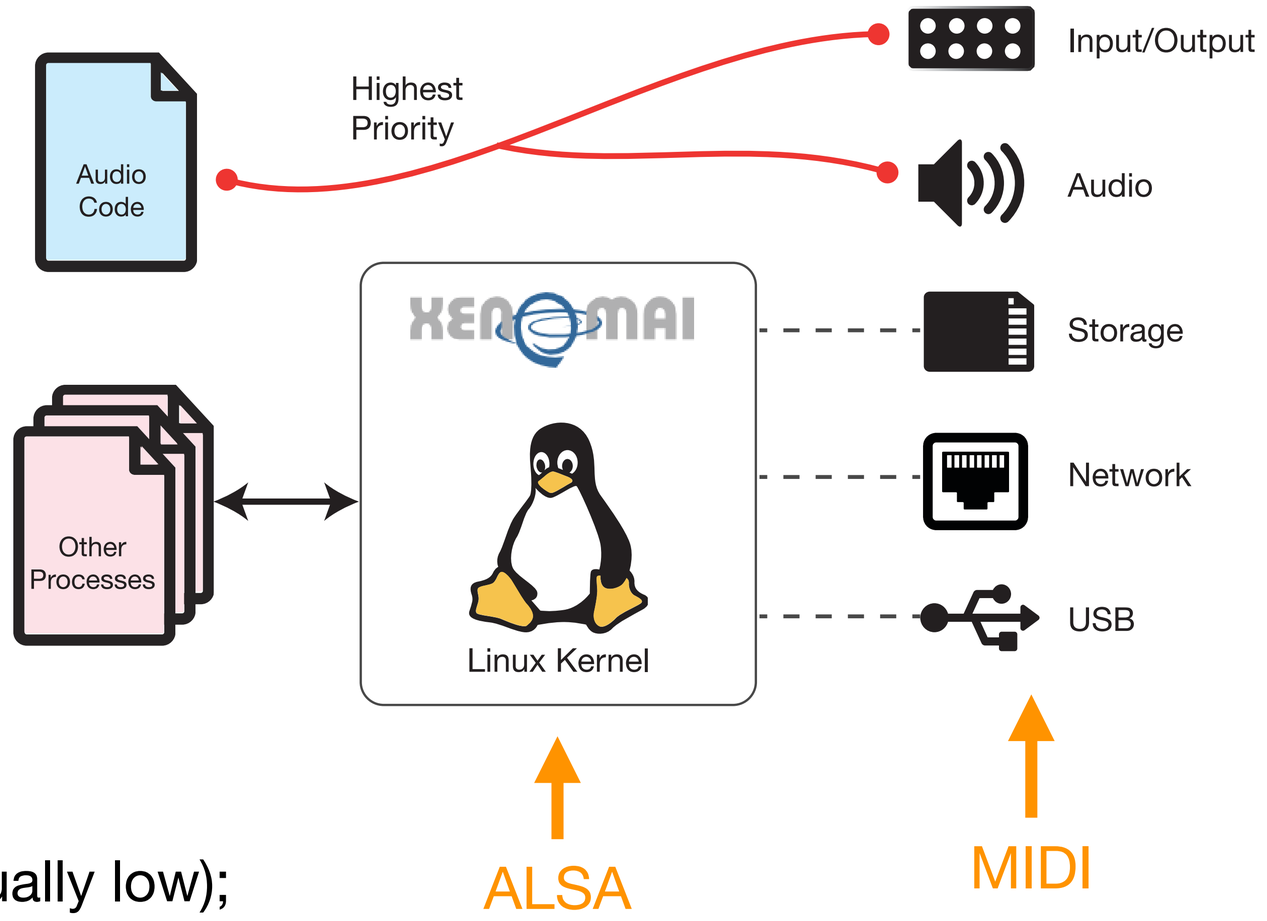
MIDI

- MIDI: Musical Instrument Digital Interface
  - Industry standard for connecting digital musical devices
  - First published in 1983, with minor updates since
  - Maintained by the MIDI Manufacturer's Association: http://midi.org
- The fundamental unit of MIDI is the message
  - A discrete packet of data, consisting of 1-3 bytes



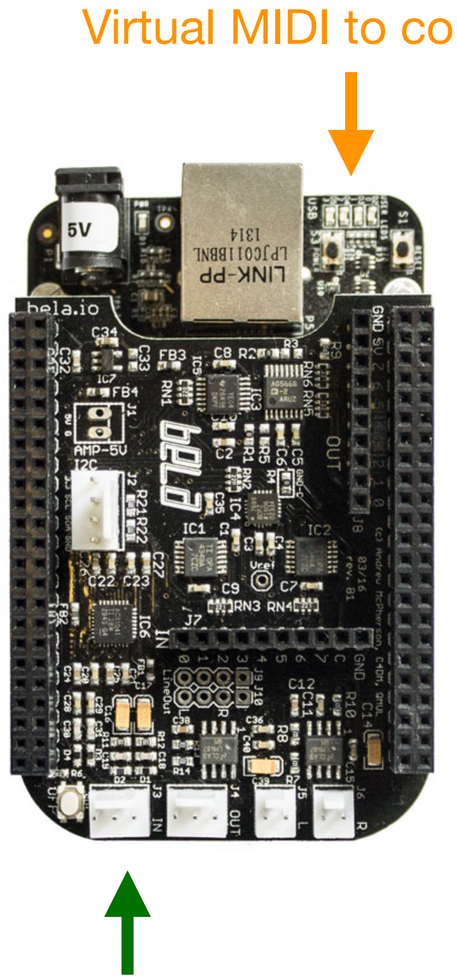
MIDI on Bela

- On Bela, MIDI uses ALSA (Advanced Linux Sound Architecture)
  - This is provided by the Linux kernel
- Advantage: compatibility
  - Any MIDI device that works on Linux works on Bela
  - All sources of MIDI data are treated equivalently (serial, USB, virtual ports)
- Disadvantage: real-time latency
  - ALSA cannot use the hard real-time priority of Bela's audio callback
  - We still need a way to pass data from the ALSA system to the Bela audio callback
  - Result: no latency guarantees (though usually low); latency could depend on other system load



MIDI on Bela: devices

- There are two common ways of using MIDI on Bela
  - Hardware MIDI devices
    - Connect these by USB to the host port on Bela
  - Bela as virtual MIDI device
    - Bela appears as a MIDI device to the host computer
    - Can exchange MIDI messages back and forth with the computer
- In your code, each MIDI device is specified by a device name (or port name)
  - The virtual device is usually `hw:0,0,0`
  - The first connected USB device is usually `hw:1,0,0`
  - To see connected devices, run `amidi -l` on the Bela console



MIDI on Bela: API (option 1)

- Bela provides an API that wraps the ALSA system
- To use it:
  1. Include `Midi.h`
  2. Make a global `Midi` variable
  3. Initialise the settings in `setup()`
  4. Read/write the messages in `render()`

```
#include <Bela.h>
#include <libraries/Midi/Midi.h>

Midi gMidi;                                    // MIDI object
const char* gMidiPort0 = "hw:1,0,0";           // Port name

bool setup(BelaContext *context, void *userData)
{
    // Initialise the MIDI device
    gMidi.readFrom(gMidiPort0);
    gMidi.writeTo(gMidiPort0);
    gMidi.enableParser(true);
    return true;
}

void render(BelaContext *context, void *userData)
{
    // At the beginning of each callback, look for available MIDI
    // messages that have come in since the last block
    while(gMidi.getParser()->numAvailableMessages() > 0) {
        MidiChannelMessage message;
        message = gMidi.getParser()->getNextChannelMessage();

        // Check message types and decide what to do with each one
        if(message.getType() == kmmNoteOn) {
            // Note on
        }
        // etc.
    }

    // Now calculate the audio for this block
}
```

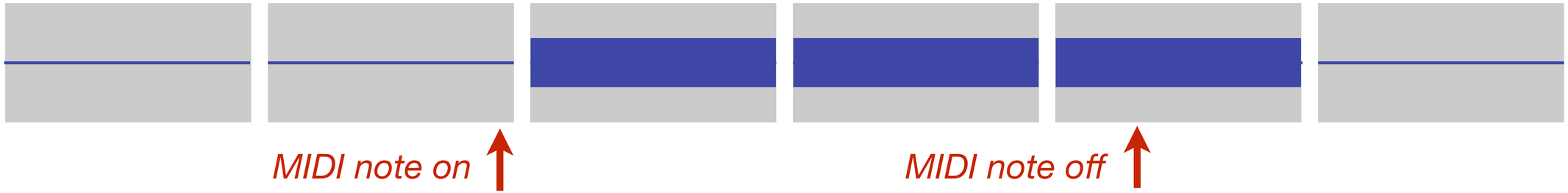
MIDI on Bela: API (option 2)

- You can also use a separate MIDI callback function
  - The system calls your function whenever MIDI data comes in
  - MIDI output is handled separately
- To use it:
  - In `setup()`, tell the `Midi` object where to find your callback
  - In the callback function, read the incoming message
  - Up to you to pass any required data to `render()`, e.g. using global variables

```
#include <Bela.h>
#include <libraries/Midi/Midi.h>

Midi gMidi;                                  // MIDI object
const char* gMidiPort0 = "hw:1,0,0";         // Port name

void midiEvent(MidiChannelMessage message, void *arg);

bool setup(BelaContext *context, void *userData)
{
    // Initialise the MIDI device
    gMidi.readFrom(gMidiPort0);
    gMidi.writeTo(gMidiPort0);
    gMidi.enableParser(true);
    gMidi.setParserCallback(midiEvent, (void *)gMidiPort0);
    return true;
}

void midiEvent(MidiChannelMessage message, void *arg)
{
    // Check message types and decide what to do with each one
    if(message.getType() == kmmNoteOn) {
       // Note on
    }
    else if(message.getType() == kmmNoteOff) {
       // Note off
    }
    // etc.
}
```

MIDI on Bela: other considerations

- Remember: MIDI is fundamentally asynchronous with audio
  - It comes from a different hardware device, via a different thread
    - When you read the messages in `render()`, you're seeing everything that happened since the last block
    - You won't necessarily know exactly which audio frame the message was received at



- MIDI messages have persistent effects
  - When a message comes in, we start doing something and usually continue doing it until we receive another message
  - Examples:
    - A Note On message starts some sound, which continues after the instant that the message comes in
    - Control Change or Pitch Wheel messages will affect current and future notes on that channel
  - This implies that we need to remember what happened before (e.g. with global variables)

MIDI note number

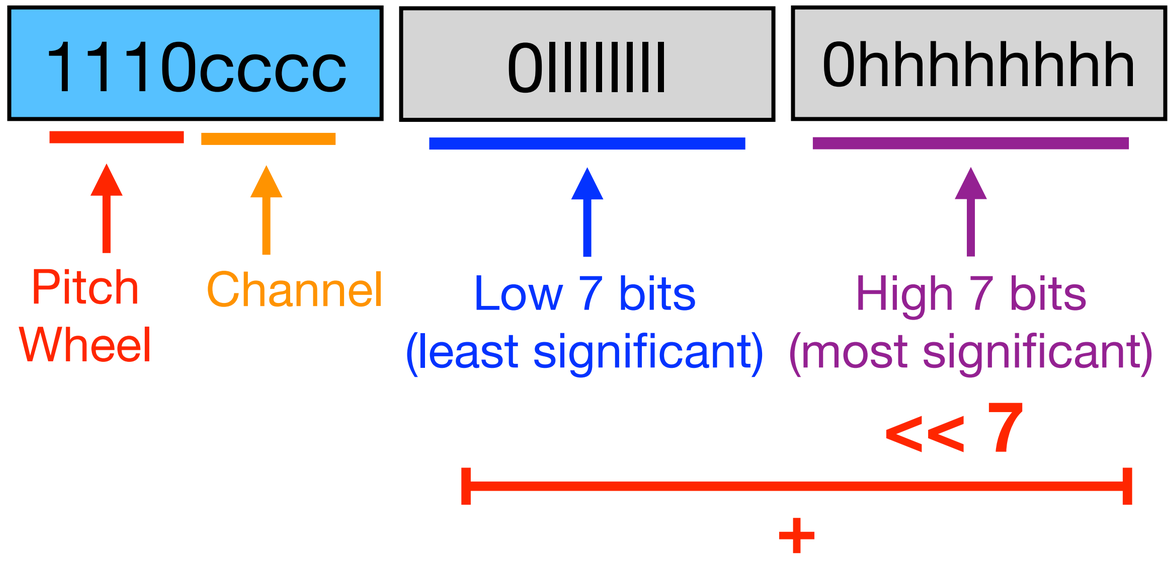
- MIDI note numbers uniquely indicate each key of the keyboard
  - Numbered sequentially from low to high
  - By definition, middle C is note number 60
    - It has frequency 261.63Hz
  - The A above middle C is note number 69
    - It has frequency 440Hz (a commonly used reference, sometimes called "A440")

- How should we write the formula to go from MIDI note number to frequency?
  - For a note number *N*, how do we calculate number of semitones above A440? *N* - 69
    - Number of octaves? (*N* - 69) / 12
  - If note 69 corresponds to $f_{ref} = 440\text{Hz}$ how do we calculate the frequency for note N?

$$f_{out} = 440 \cdot 2^{\frac{N-69}{12}}$$ *(MIDI note number to frequency)*

Implementing MIDI pitch wheel

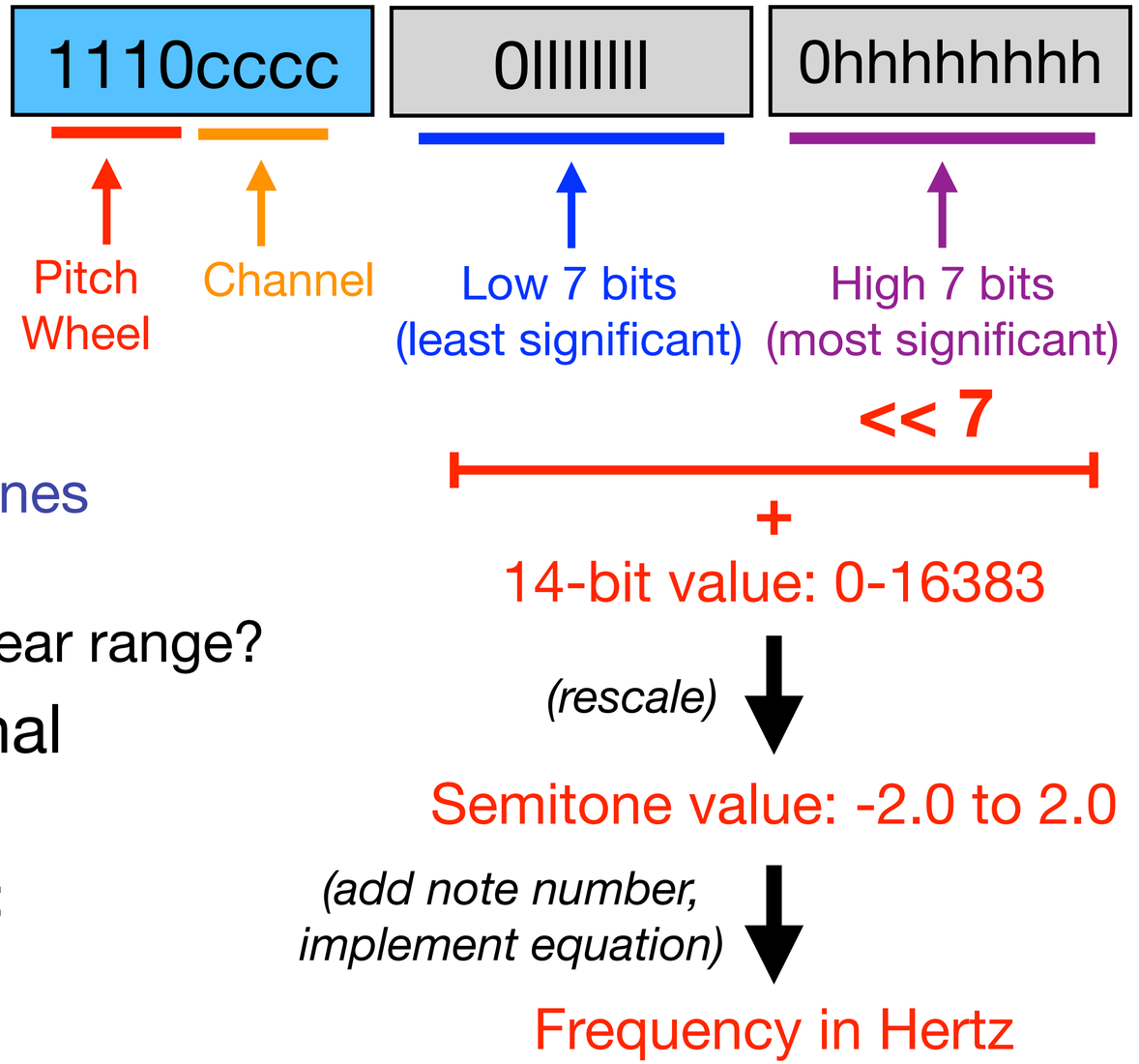
- Pitch Wheel messages create a relative change in frequency
  - A change compared to where that note number would normally be tuned
- To calculate the oscillator frequency, we always need both of:
  - The note number of the most recent Note On message
  - The value of the most recent Pitch Wheel message
  - We receive one message at a time, which means we have to remember the other one!
- Pitch Wheel messages are 14 bits (0 to 2^{14}-1), split over two data bytes
  - Our first task is to reconstruct the 14-bit value
  - Use the bit shift operator `<<` to shift the MSB left by 7 bits (i.e. multiply by 2^7)
  - Then add the two values together



Implementing MIDI pitch wheel

- Once we have the 14-bit value, convert it to a bend in semitones
  - The 14-bit value ranges from 0-16383
  - The centre value is 8192: this corresponds to no bend
  - A value of 0 corresponds to -2 semitones
  - A value of 16383 corresponds to +2 semitones
    - Though these ranges are adjustable
  - What function could we use to rescale a linear range?
- Finally, add the bend value to the original note number
  - Then convert the sum to a frequency using:

$$f_{out} = 440 \cdot 2^{\frac{N-69}{12}}$$

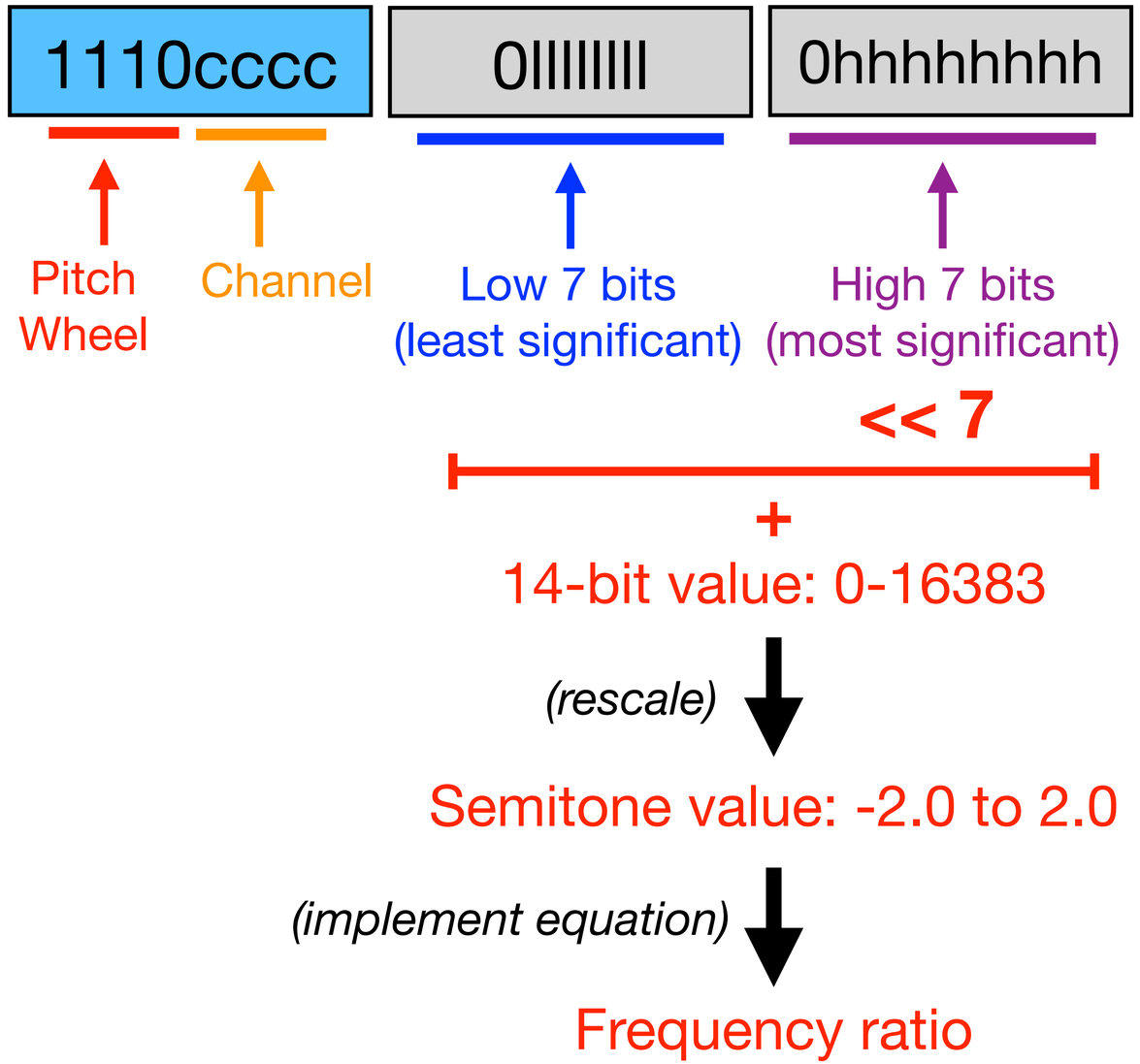


Implementing MIDI pitch wheel

- Alternatively: convert the pitch bend directly into a frequency ratio
  - Store the frequency f_{note} of the MIDI note elsewhere

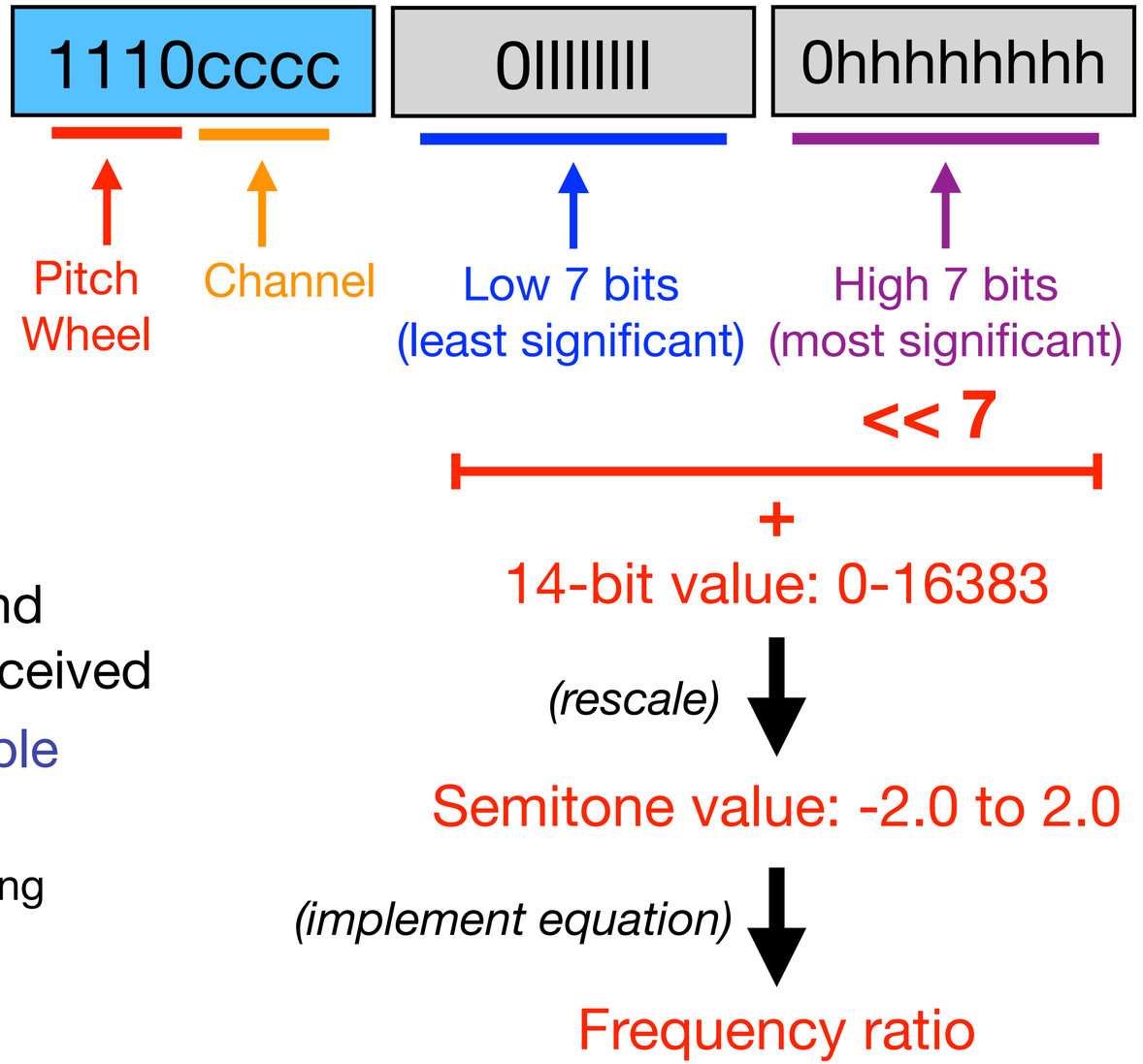
$$f_{out} = f_{note} \cdot 2^{\frac{N}{12}}$$

  - Multiply the two values together to get the final frequency
  - We'll take this approach for reasons we'll see later



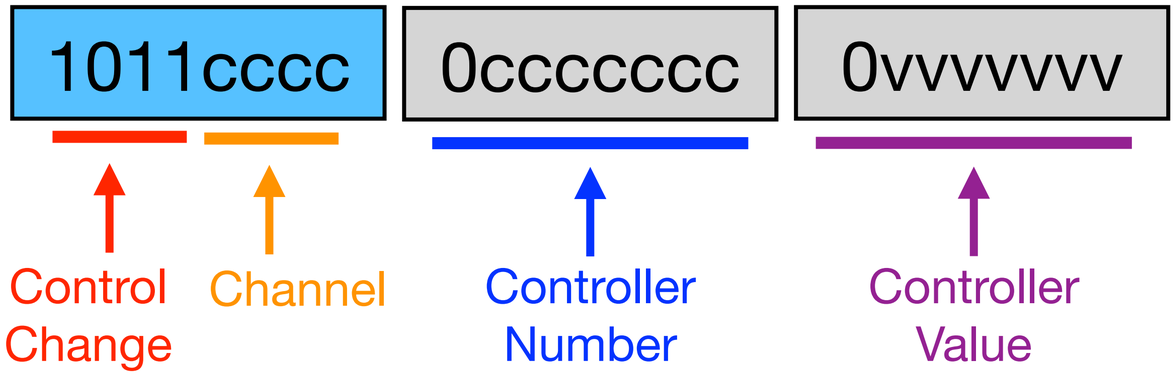
Implementing MIDI pitch wheel

- Task: using the midi-pitchwheel example, implement the pitch wheel
  - In `midiEvent()`, add a new `if()` statement looking for a message of type `kmmPitchBend`
  - Get the two data bytes from the message (see how it's done for `kmmNoteOn`)
    - Convert the two bytes to a single 14-bit number
  - Write a new function called `pitchWheel()` and call it when the `kmmPitchBend` message is received
  - Store the frequency ratio in a new global variable called `gPitchBendRatio`
    - In `render()`, calculate the final frequency by multiplying the note frequency by this value



MIDI control change messages

- Control Change
  - Status byte begins with 1011 (hexadecimal 0xB)
  - Indicates that a controller changed value
  - Controllers in MIDI might be dials, sliders, pedals, breath controllers, etc.
    - They produce a continuous value (0-127) which affects all notes on the channel
  - 2 data bytes: controller number, value
  - Controller numbers are standardised in the MIDI specification
    - e.g. CC1 = modulation wheel; CC7 = volume; CC11 = expression; CC64 = damper/sustain pedal
    - Controllers 120-127 are used for special functions on the channel, e.g. CC123 = all notes off
- Like all MIDI messages, the effect is persistent
  - When you receive a Control Change message, you'll need to keep track of its effect
  - Typically, rescale the range: from 0-127 to an appropriate range for the specific control



Multiple oscillators

- On analog synths, it's common to use 2 or more oscillators for a single voice
  - The oscillators are tuned nearly, but not exactly, to the same frequency
  - Slight detuning results in beat frequencies according to the mathematical relationship:

$$\sin(x) + \sin(y) = 2\sin(\frac{x+y}{2})\cos(\frac{x-y}{2})$$

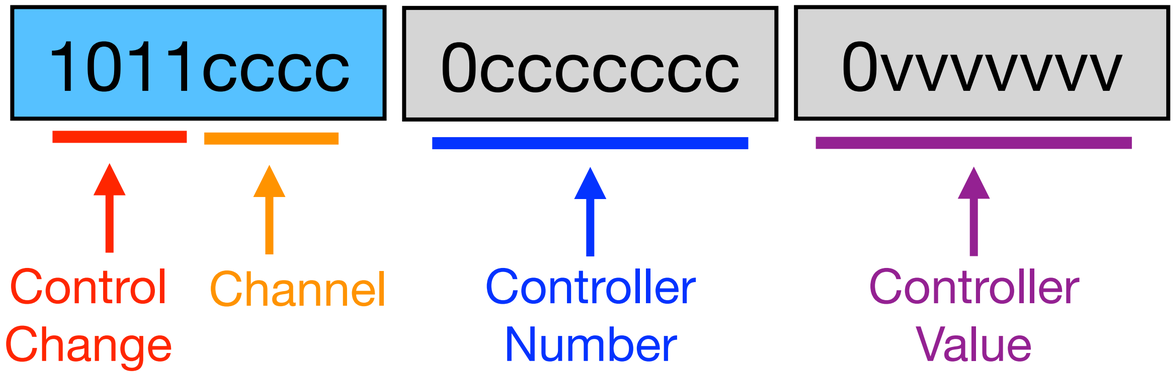
- We will use two controls to adjust the oscillator frequencies:
  - One for setting the centre frequency of both oscillators
    - Based on MIDI Note On messages, like before
  - One for changing the detuning (the slight difference between the oscillators)
    - We will implement this with a MIDI Control Change message
  - If the centre frequency is *f* and the detuning amount is *r*, we should have:

$$f_1 = (1+r)f \qquad f_2 = (1-r)f$$

    - In this case, *r* = 0 corresponds to no detuning (both oscillators the same frequency)

MIDI control change messages

- Task 1: in project midi-pitchwheel, implement two detuned oscillators
  - Replace the `Wavetable` object `gOscillator` with a two-element array
  - Initialise both oscillators in `setup()`
    - You don't have to recalculate the table
  - Declare a global variable `gDetuning` which holds the detuning amount
    - Try a default value of **0.002** for a good effect
  - In `render()`, set the frequency of both oscillators using the detuning amount
  - Mix both oscillator outputs together before applying the filter
- Task 2: use a MIDI control change to change the detuning ratio
  - Choose a controller number based on your keyboard's controls
    - Look in the console when the project is running and you adjust a controller
  - In `midiEvent()`, look for messages of type `kmmControlChange`
    - If the controller number matches, rescale the range to be between **0-0.01**

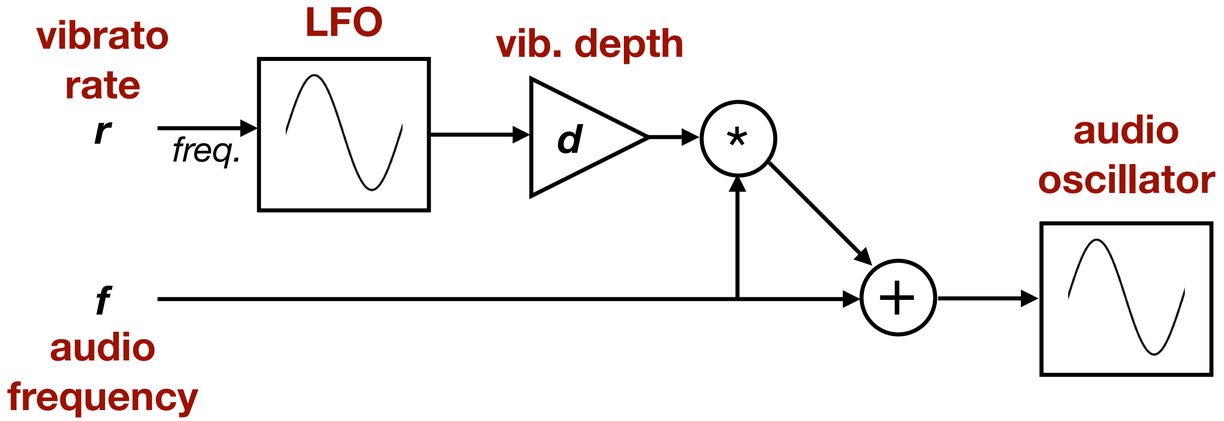


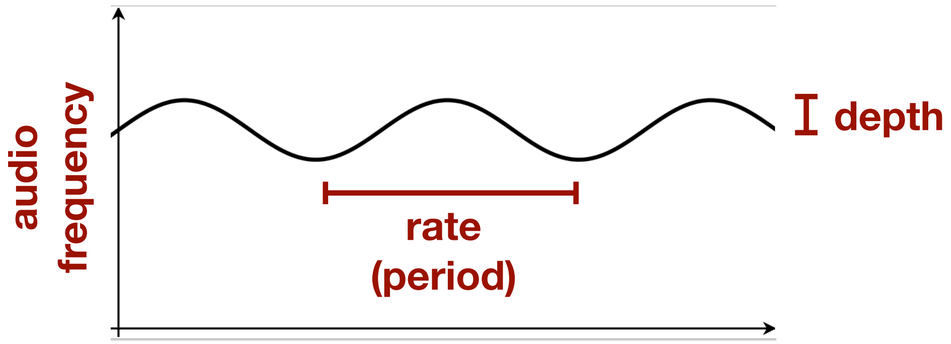
Modulation wheel

- In addition to the pitch wheel, most keyboards have a modulation wheel (“mod wheel”)
  - Introduced on the Minimoog in 1970 (before MIDI)
  - Unlike the pitch wheel which snaps back to centre, the modulation wheel stays in place
- The mod wheel can affect various parameters:
  - Filter cutoff frequency
  - Oscillator parameters (e.g. waveform)
  - Vibrato depth
    - Do this by change the amplitude of a low-frequency oscillator (LFO)
    - The LFO will then affect the oscillator frequency

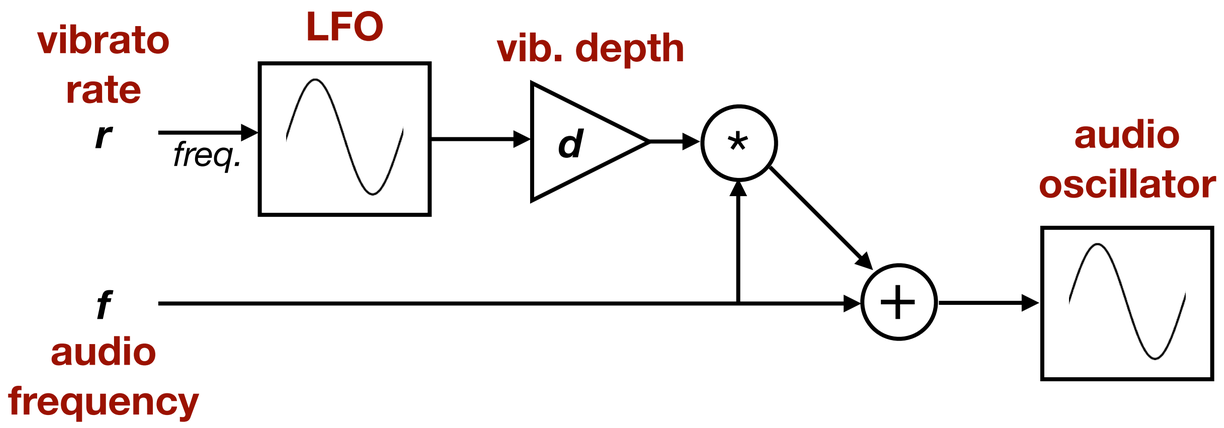
Low-frequency oscillators

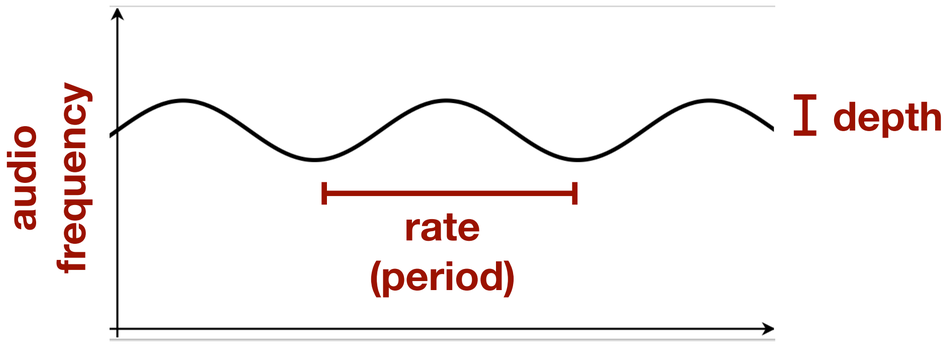
- We have seen oscillators (sine, sawtooth, wavetable) as audio sources
- In digital musical systems, oscillators are also often used as control signals
  - Typically these control signals will change more slowly than the audio waveform
  - Hence we refer to such oscillators as low-frequency oscillators of LFOs
- Example: Vibrato
  - Periodic modulation of frequency, typically changing at 3-6Hz (vibrato rate)
  - Typical amount of frequency variation ±0.5-3% (vibrato depth)
  - To implement: use an LFO to control the frequency of another (audio frequency) oscillator





Low-frequency oscillators

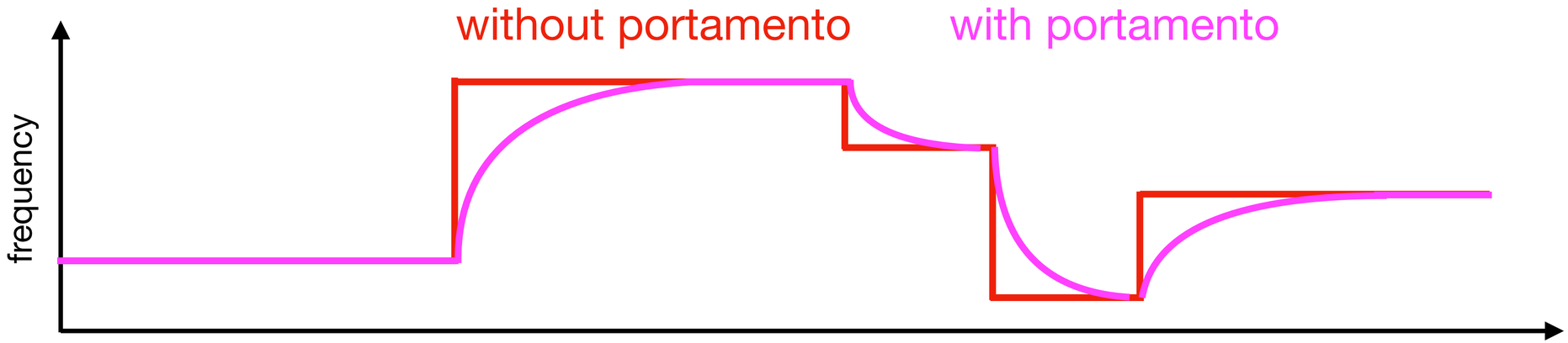




- Task 1: in midi-pitchwheel, use a `Wavetable` object to implement an LFO
  - In `setup()`, calculate a wavetable for a single sine wave (different than the current oscillators)
  - Set its default frequency to something low (e.g. 5Hz)
  - Use the output of the LFO to modulate the frequency of the main oscillators
- Task 2: make the vibrato depth controllable with the mod wheel
  - The mod wheel produces a MIDI control change message with a controller number of 1
  - Range of values is 0-127. Need to rescale to a suitable range of vibrato depth (e.g. 0-0.05)
  - Optionally add another MIDI controller to adjust vibrato speed (range 1-10Hz)

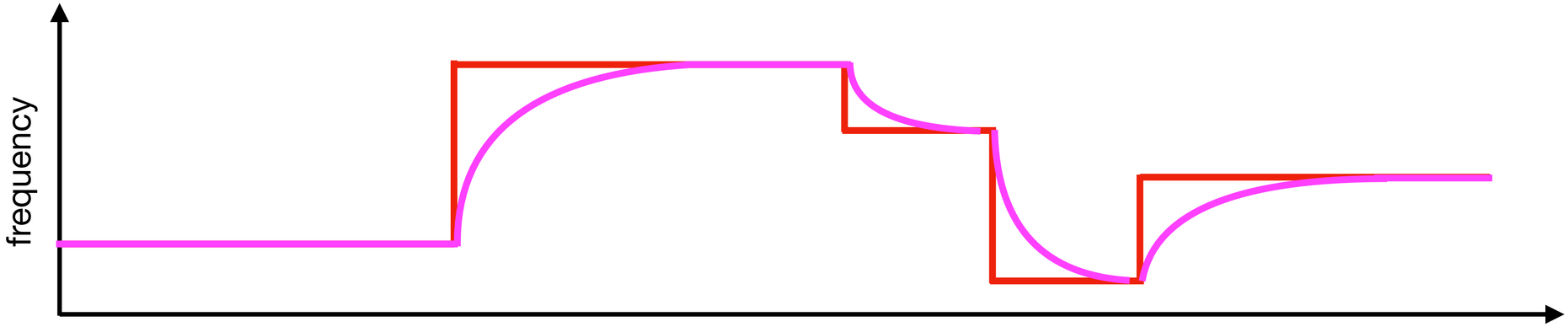
Portamento

- Portamento is a musical effect where the pitch moves gradually, rather than abruptly, from one note to the next:



- It is commonly found on analog synths, where the effect can be achieved by putting a low-pass filter on the pitch CV
  - Lower cutoff frequency → slower change in pitch → more pronounced portamento
- Equivalently, we can use an exponential segment to change frequency every time the MIDI note changes
  - Use the `ExponentialSegment` class from Lecture 14

Portamento implementation



- Task: in the midi-pitchwheel project, add portamento to the note frequency
  - Use the `ExponentialSegment` class (already included in the project)
  - Two possible approaches:
    1. Use `ExponentialSegment` to control the current centre frequency (not including pitch/mod wheels)
    2. Use `ExponentialSegment` to control the current (fractional) MIDI note number
  - Add a new MIDI control change message to adjust the portamento time
    - This will affect the time passed to the `ExponentialSegment::rampTo()`

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources