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 15: MIDI part 1

What you'll learn today:

Introduction to the MIDI protocol
Handling MIDI Note On and Note Off messages
Keeping track of multiple notes

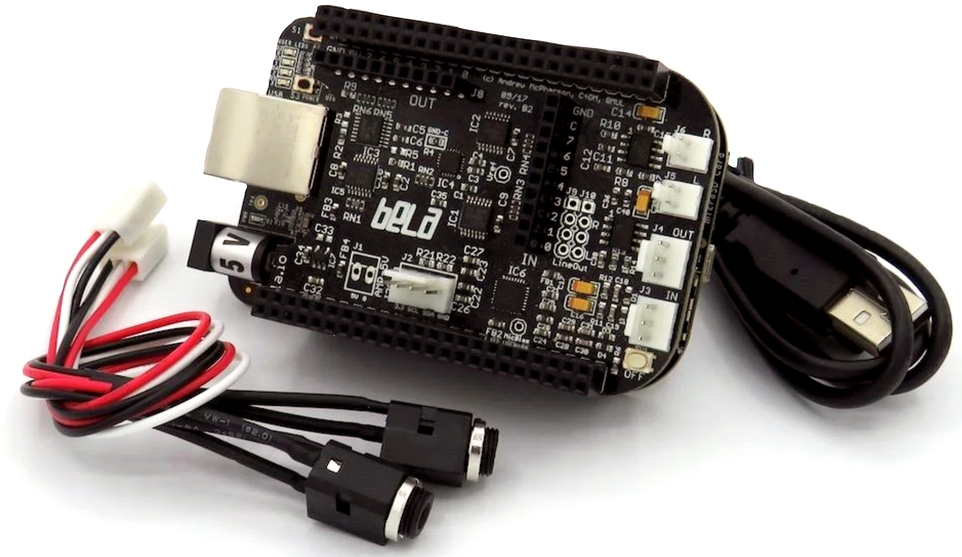
What you'll make today:

Monophonic MIDI synth with ADSR envelope

Companion materials:

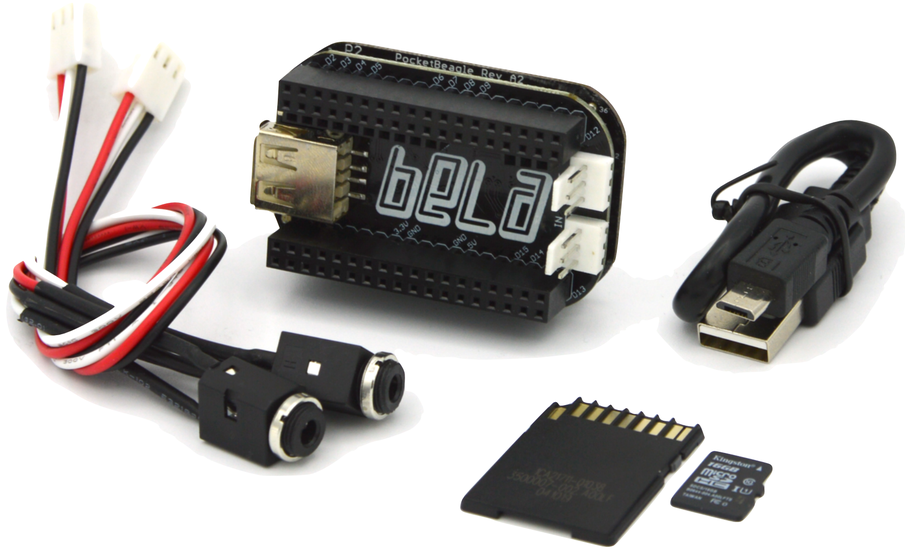
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

or



Bela Mini Starter Kit

[shop.bela.io]

Also needed for this lecture:

or virtual keyboard software:

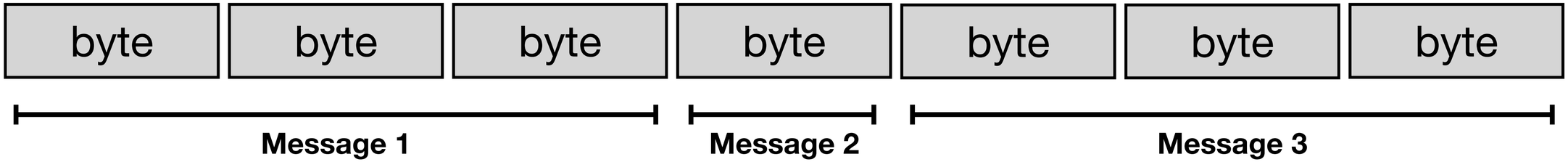
MIDIKeys (Mac)
vmpk (cross-platform)

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
- Several common MIDI transport layers
  - Serial: 31250 bps, 5-pin DIN cables, one cable for each direction
  - USB
  - Bluetooth
  - Virtual ports between software applications

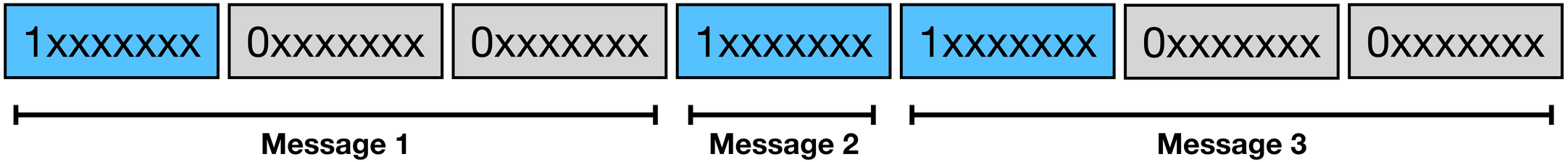
MIDI messages

- The fundamental unit of MIDI is the message
  - A discrete packet of data, consisting of 1-3 bytes



- The challenge is understanding which bytes group together
  - On a serial port, the contents of the byte are the only information we have
  - No external signals to tell us which bytes group with which others
- MIDI therefore uses the high bit of each byte to signal the start of a message
  - The first byte of each message has a 1 in the high bit; the others have a 0

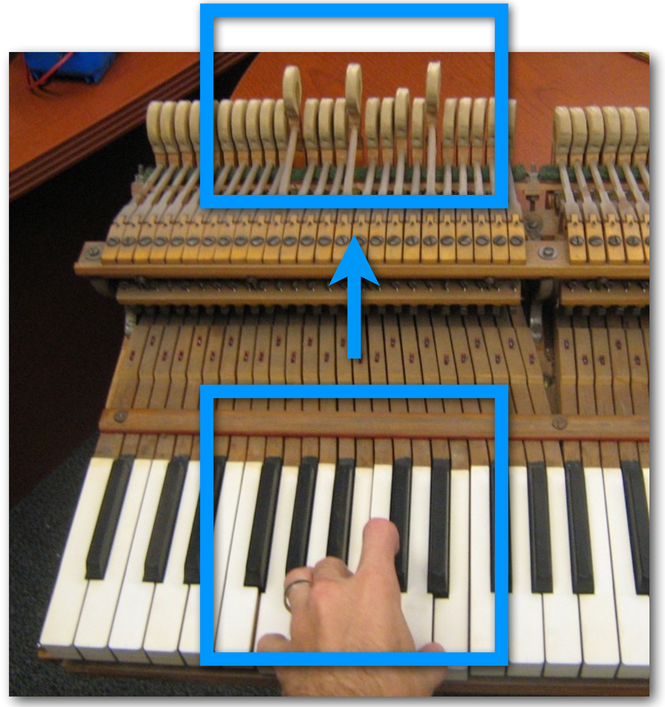
MIDI messages

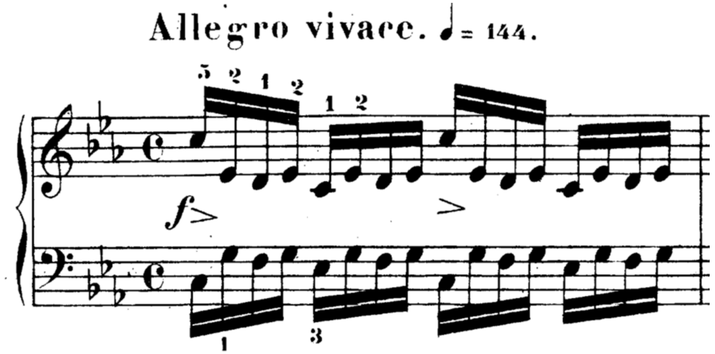


- The first byte of each message is called the status byte
  - The high bit is set to 1
  - The other 7 bits tell us what kind of message it is
- The remaining bytes are called data bytes
  - The high bit is set to 0
  - The data is held in the other 7 bits
  - For this reason, many MIDI messages have 7-bit resolution (values 0 to 127)
  - The number of data bytes is defined by the message type
- More details on MIDI messages:
  - https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message

The musical assumptions of MIDI

- MIDI was designed with keyboards and western music in mind
  - It makes many assumptions about how music is organised
  - Captures many salient aspects in very little information, at the cost of generality
- The fundamental unit of MIDI is the note
  - Notes have a discrete onset and a discrete release
    - Happens at a particular instant in time
    - We'll sometimes hear the word "trigger" used to describe note events
  - Note onsets have a velocity
    - Analogy to the speed of a key press
    - Often related to volume
  - Note releases can also have a velocity
    - Analogy to speed of key release
    - In practice, this field is rarely used





Note On	
Note #	Velocity

Note Off	
Note #	Velocity

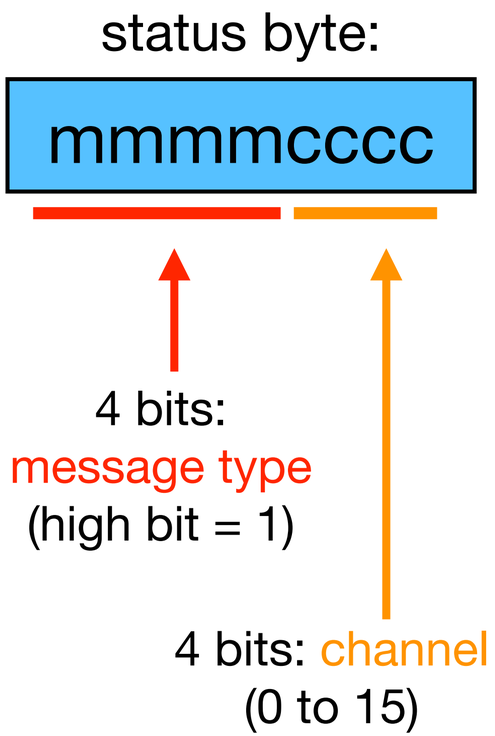
The musical assumptions of MIDI

- Notes are characterised by their note number
  - There are 128 note numbers (from 0 to 127)
  - Every key on the keyboard has a unique note number
    - This is agreed across all MIDI devices
  - By convention, middle C on the piano is note 60
    - That means the 88-key grand piano has note numbers 21 to 108
  - Notes are numbered by semitone
    - i.e. sequentially by piano key
    - There are 12 note numbers per octave

- There are other types of messages providing continuous control
  - Many of these are organised around the capabilities of electronic synths
- Collectively, these messages describe the piano and western staff notation
  - MIDI can be applied to many other situations, but be aware of its biases!

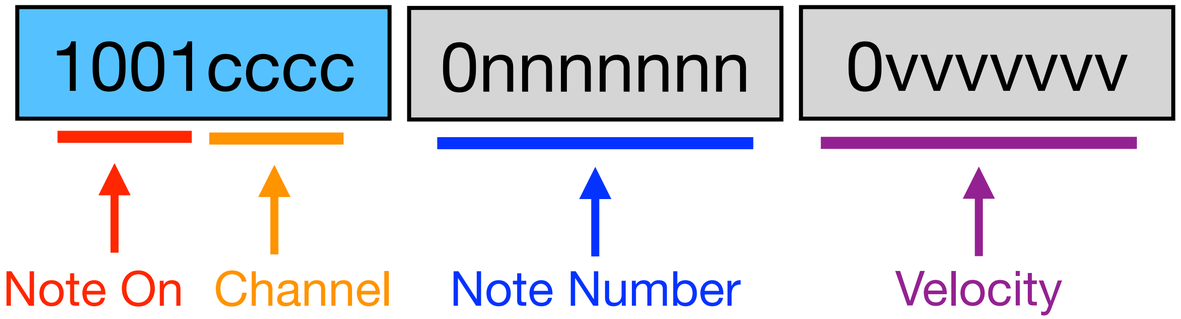
MIDI message types

- The MIDI 1.0 standard defines several types of messages
  - The status byte tells us the kind of message
  - Specifically, most message types are identified in the high 4 bits of the status byte
- MIDI messages can be sent on one of 16 channels
  - Different channels can be used to play different instruments at the same time, using a single MIDI cable
  - A single channel can have multiple notes, but other controls affect the whole channel
  - The channel is typically identified by the low 4 bits of the status byte
    - Values of 0 to 15 within the byte (2^4 possible values)
    - By convention these are known as channels 1 to 16

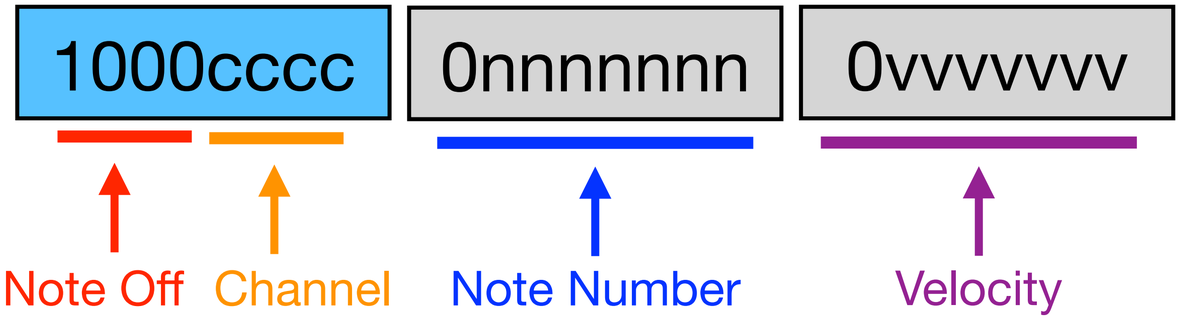


MIDI message types

- Note On
  - Status byte begins with 1001 (hexadecimal 0x9)
  - Indicates that a note started
    - e.g. a key was pressed
  - 2 data bytes: note number, velocity
    - In practice, usually related to frequency and amplitude
  - A Note On message with a velocity of 0 is the same as a Note Off message

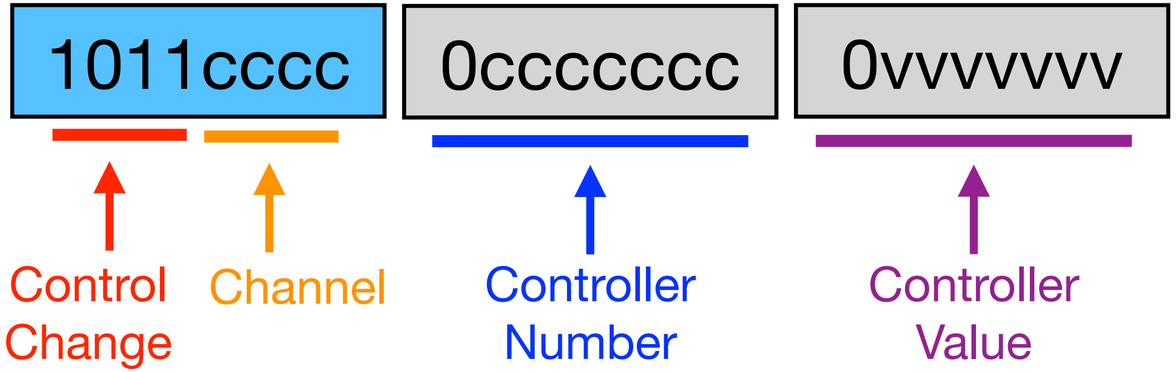


- Note Off
  - Status byte begins with 1000 (hexadecimal 0x8)
  - Indicates a note ended
    - e.g. a key was released
  - 2 data bytes: note number, velocity



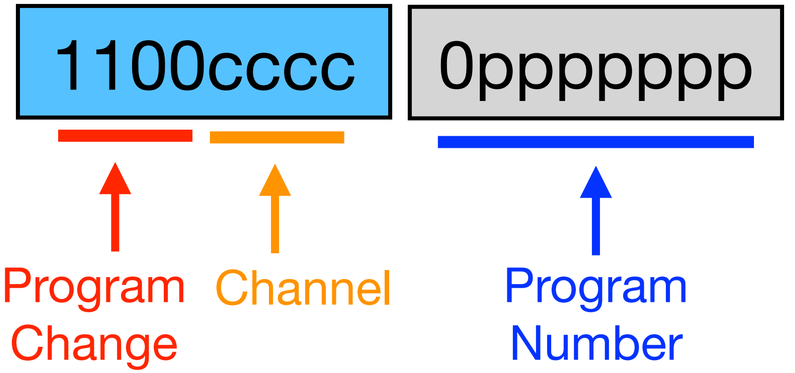
MIDI message types

- 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

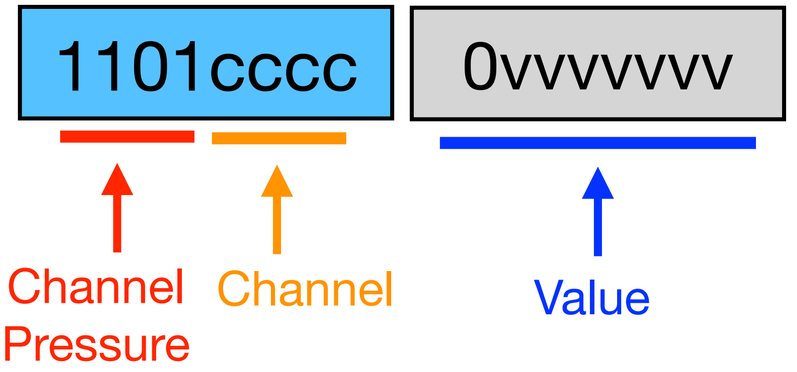


MIDI message types

- Program Change
  - Status byte begins with 1100 (hexadecimal 0xC)
  - Indicates that the synth should change program (preset or sound)
  - 1 data byte: program number

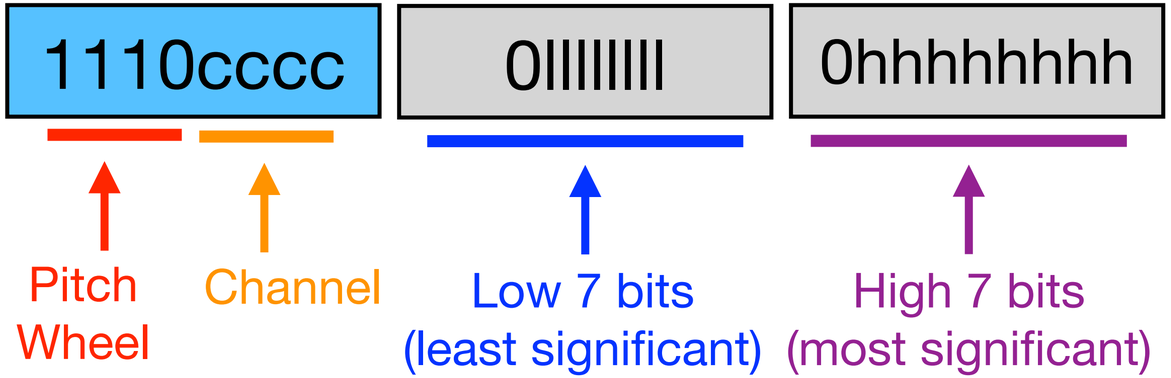


- Channel Pressure (aftertouch)
  - Status byte begins with 1101 (hexadecimal 0xD)
  - Some MIDI keyboards have a pressure sensor under the keys
    - Other types of MIDI controller could use different sensors for this purpose
  - This message reports the pressure value
    - Affects all notes on the channel
  - 1 data byte: pressure value



MIDI message types

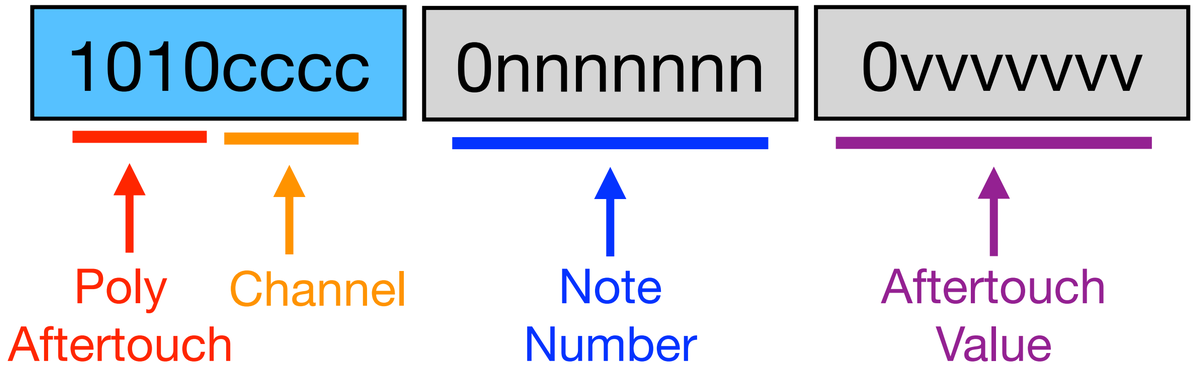
- Pitch Wheel
  - Status byte begins with 1110 (hexadecimal 0xE)
  - A control to bend the pitch of the notes up or down
  - First appeared in the Minimoog (1970)
  - The human ear is sensitive to frequency
    - We need more than 7 bits of resolution!
  - 2 data bytes: LSB and MSB
    - Each contains 7 bits, for a composite 14-bit value
    - 16384 (2^{14}) possible values
    - The centre value of the pitch wheel is 8192; higher or lower values bend the pitch up or down
  - By convention, the pitch wheel range is ±2 semitones (adjustable by other messages)



Minimoog (1970)

MIDI message types

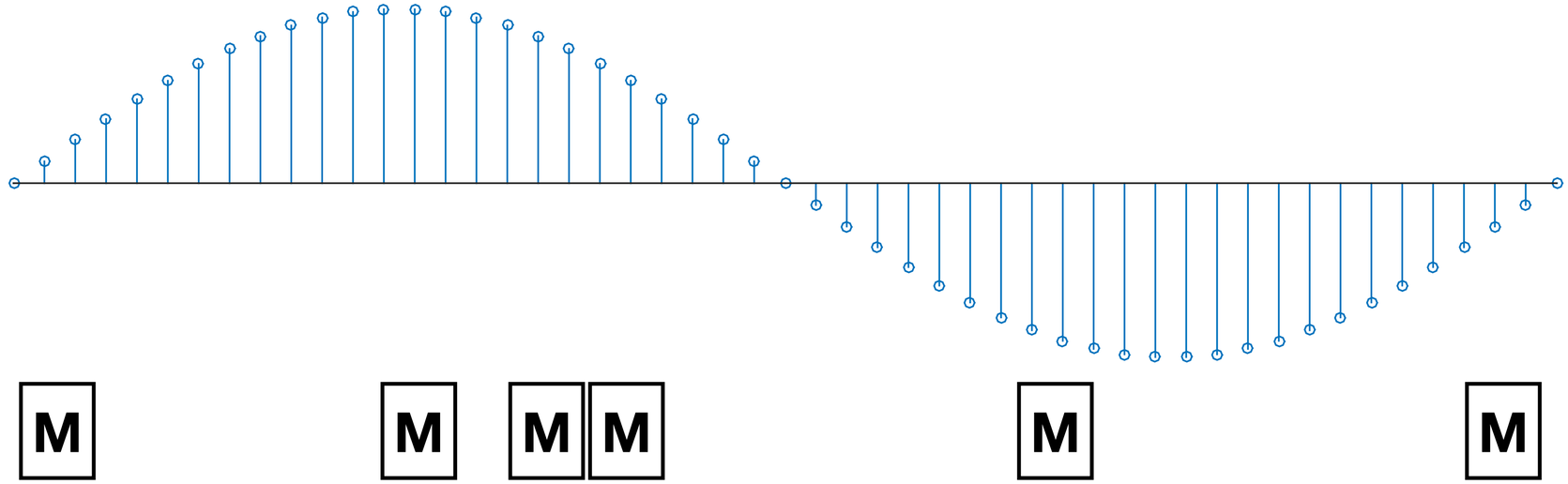
- Polyphonic Aftertouch
  - Status byte begins with 1010 (hexadecimal 0xA)
  - Like channel pressure, but a separate value for each key
  - Relatively few controllers support this
  - 2 data bytes: note number, value



- System Real-Time messages
  - Various messages beginning with 1111 (0xF) for controlling global settings of the device
- System Exclusive (SysEx)
  - A block of bytes beginning with 11110000 (0xF0) and ending with 11110111 (0xF7)
  - Used for manufacturer-specific messages, not supported by all devices
  - Variable length and meaning

MIDI and audio

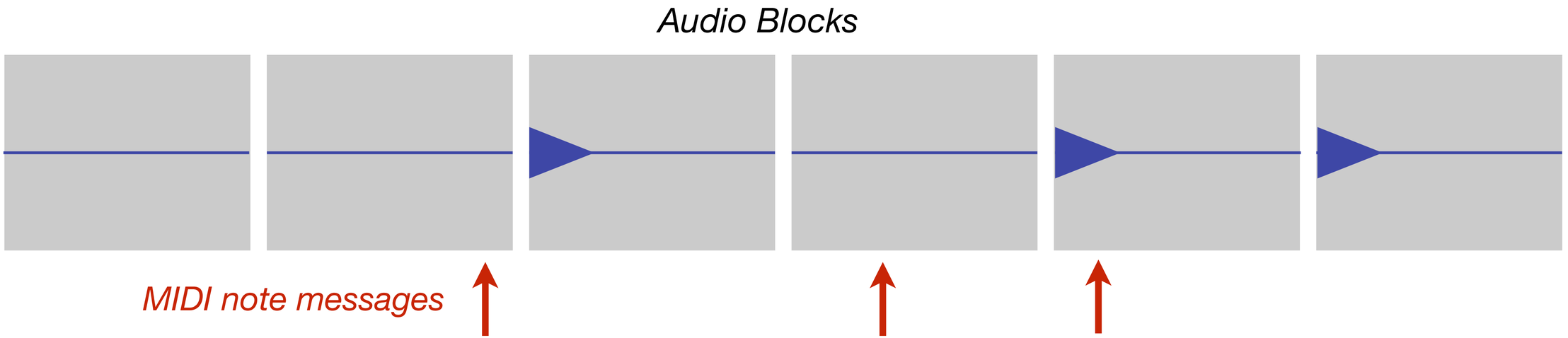
- Audio is a sampled signal
  - Has a regular sampling period, constant data rate, regular block size in real time



- By contrast, MIDI messages are sporadic events
  - Messages might arrive at any time (subject to overall limited bandwidth)
  - No regular sampling period, and no guarantee on alignment to audio frames
    - At minimum, we know that messages are sequential (one message finishes before the next begins)
- This sets up a familiar problem for real-time audio systems...

Jitter

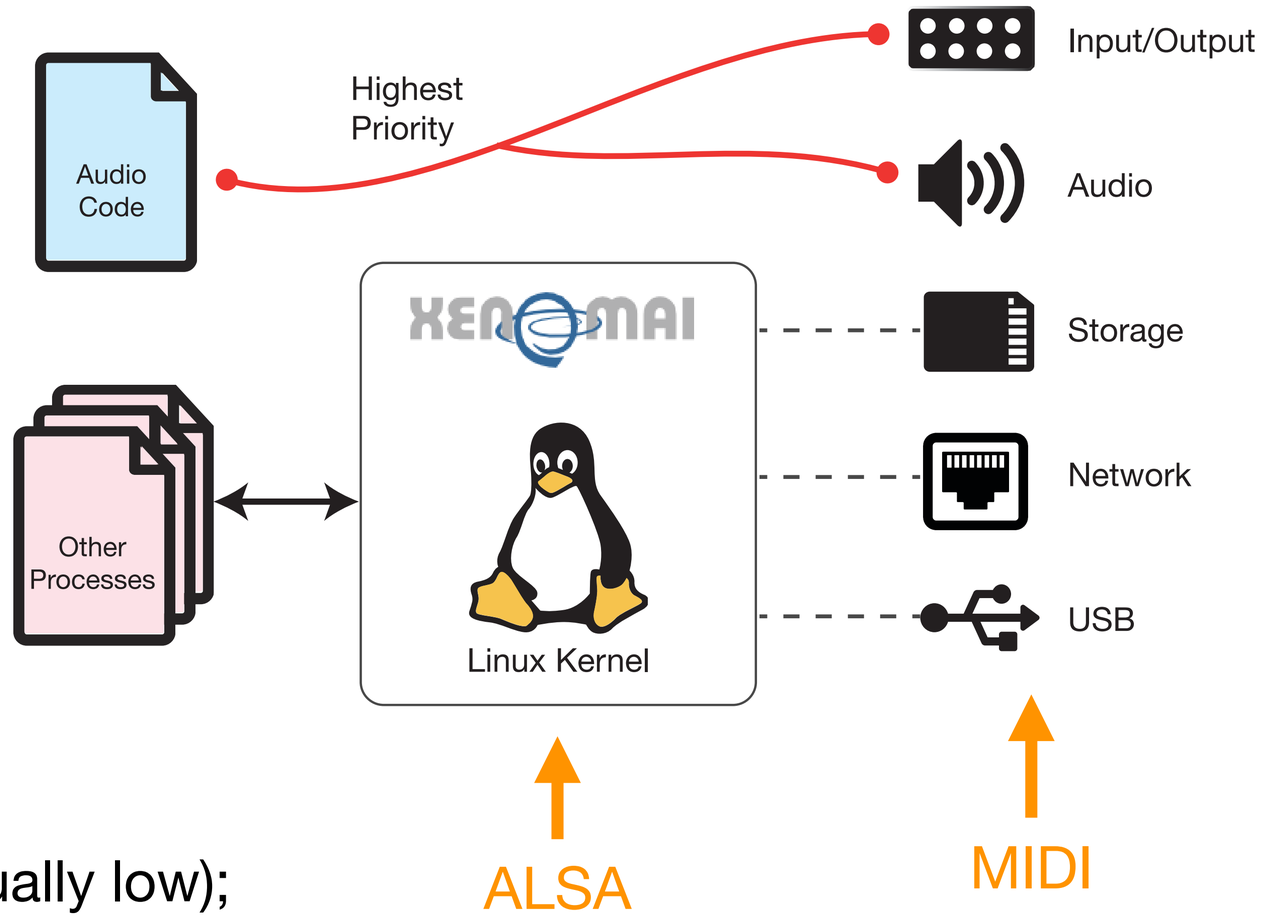
- Round-trip audio latency is usually constant
- MIDI-to-audio latency might vary randomly depending on the implementation
  - MIDI comes from a different hardware device, might be processed in a different thread
  - This variation in latency is known as jitter



- Unless we carefully plan otherwise, MIDI data processed at block intervals
  - Implication: larger block sizes mean more jitter
  - Possible alternative: strong timestamping of MIDI messages (requires hardware support)

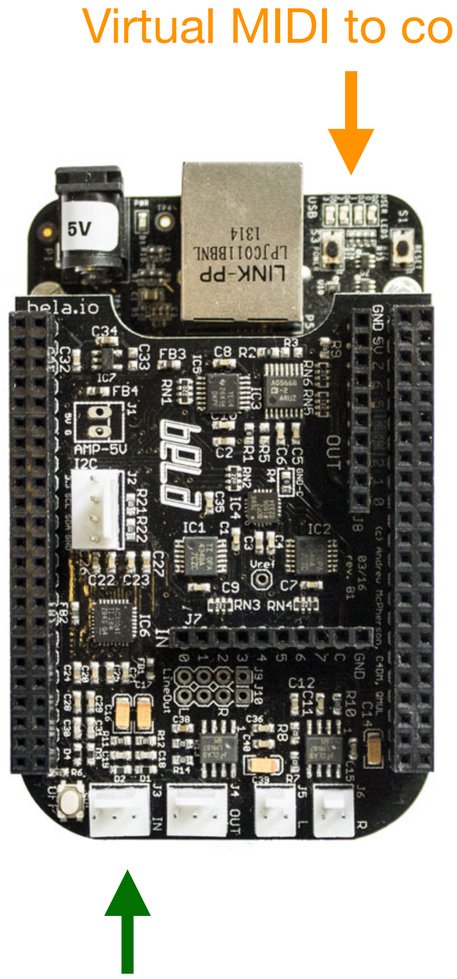
MIDI on Bela

- 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



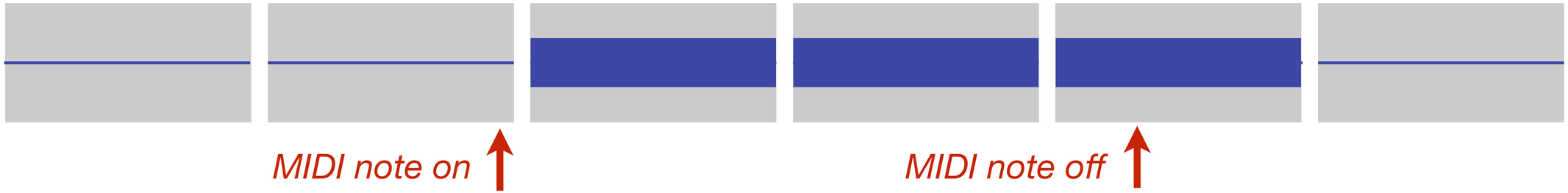
MIDI on Bela: API

- 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()`
- There is also an option to have a separate MIDI callback
  - The system calls your function whenever MIDI data comes in
  - See Bela example: `Communication/MIDI`

```
#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
}
```

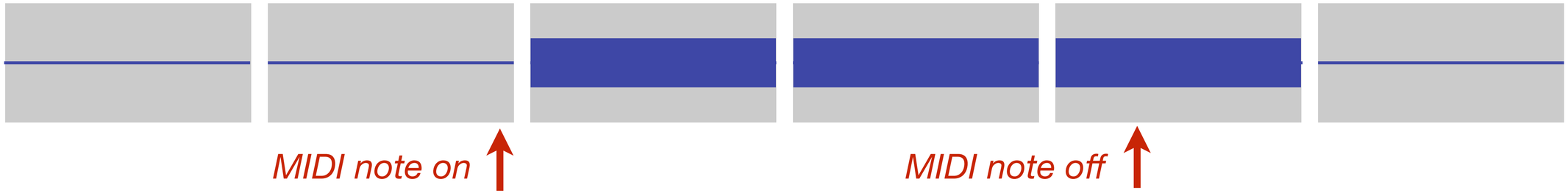
MIDI on Bela: other considerations

- 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)

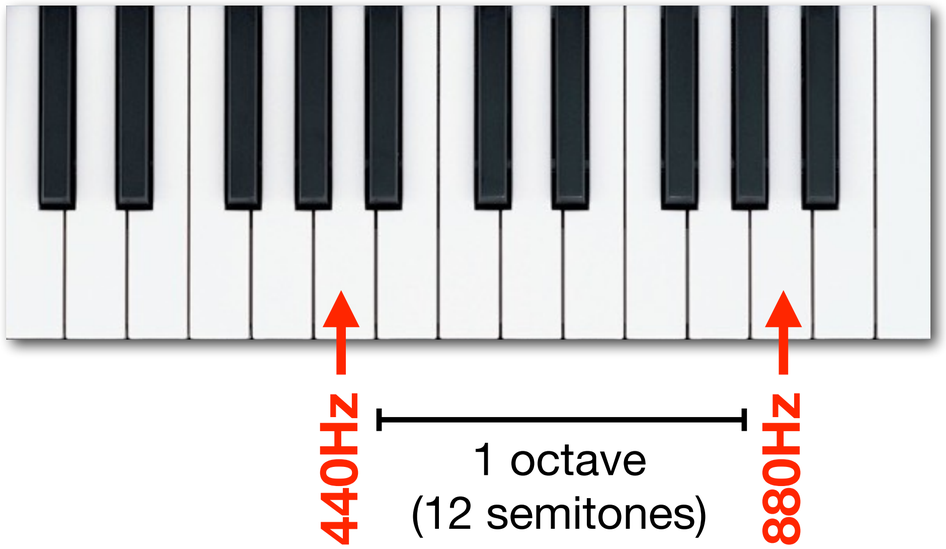
Making a MIDI synth

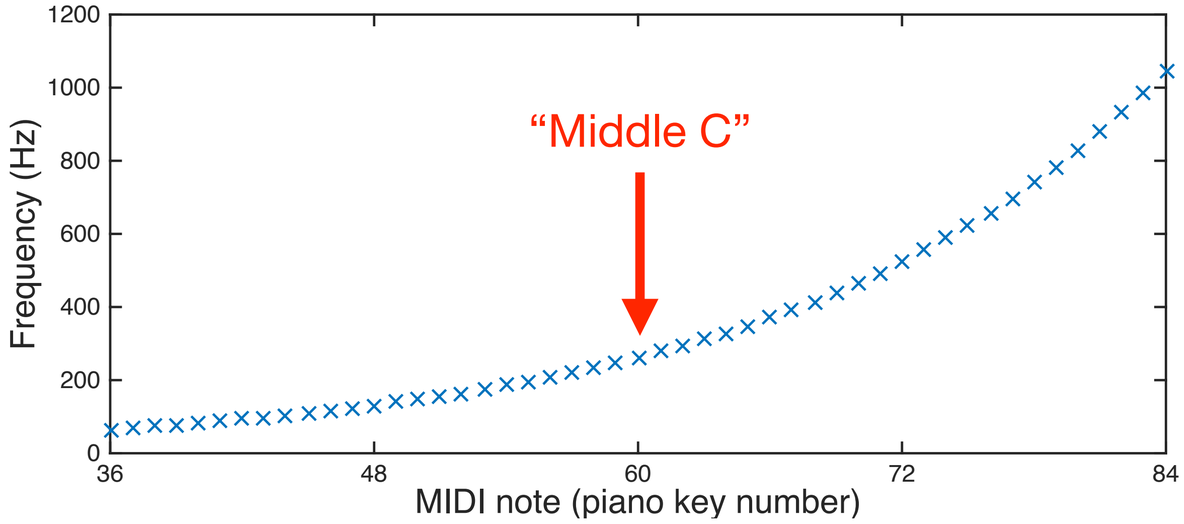


- When a Note On message arrives
  - Start generating sound for that note
    - If the sound has continuous sustain, keep generating it until we get a Note Off message
    - If the sound has a fixed length (e.g. percussion), generate it for the indicated length then stop
  - Convert the note number to a frequency
  - Optionally, set the amplitude (and/or other parameters) based on the velocity
    - Not all synths work this way: classic analog synths often do not have velocity control
- When a Note Off message arrives
  - Stop generating sound for that particular note
    - We'll look soon at how to handle multiple simultaneous notes

Review: frequency and pitch

- The perceived pitch of a sound is related to its frequency
- Pitch perception also operates on a logarithmic scale
  - Musical intervals are defined as frequency ratios
    - An octave is a 2:1 ratio; a perfect fifth is 3:2, a fourth is 4:3, etc.
  - To get from frequency to notes on a keyboard, we need a logarithmic calculation
  - However, by convention, this calculation is different from decibels





From musical pitch to frequency

- In audio synthesis, we often want to convert from pitch ("note") to frequency
  - We want a linearised scale: an increment of *x* is always the same musical interval
  - On analog synths, a common standard is 1V/octave
    - i.e. the control signal going up 1V makes the pitch go up by an octave (what happens to the frequency?)
- Given a reference frequency f_{ref} and an input *V* measured in octaves:
  - $f_{out} = f_{ref}2^{V}$ *(octaves to frequency)*
  - What if *V* was measured in semitones instead? (12 semitones to the octave)
    - How do I convert from semitones to octaves?
  - $f_{out} = f_{ref}2^{V/12}$ *(semitones to frequency)*

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)*

Building a basic MIDI synth

- Tasks: using the midi-sinetone example
  1. Implement MIDI note to frequency conversion for Note On messages

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

     - Sound can play all the time for now
     - Fill in the `noteOn()` function
  2. Mute the sound when a Note Off is received
     - Fill in the `noteOff()` function
  3. Implement velocity mapping to amplitude
     - Use a decibel scale where velocity 1-127 maps from -40dB to 0dB

$$A_{dB} = 20\log_{10}(A_{linear}) \quad \longleftrightarrow \quad A_{linear} = 10^{\frac{A_{dB}}{20}}$$

Monophony and polyphony

- There are two general approaches to musical synthesisers:

Monophonic

- One note at a time, no matter how many keys are pressed
- Only need one oscillator, one amplitude envelope, one filter etc.
- Early analog synths worked this way

Polyphonic

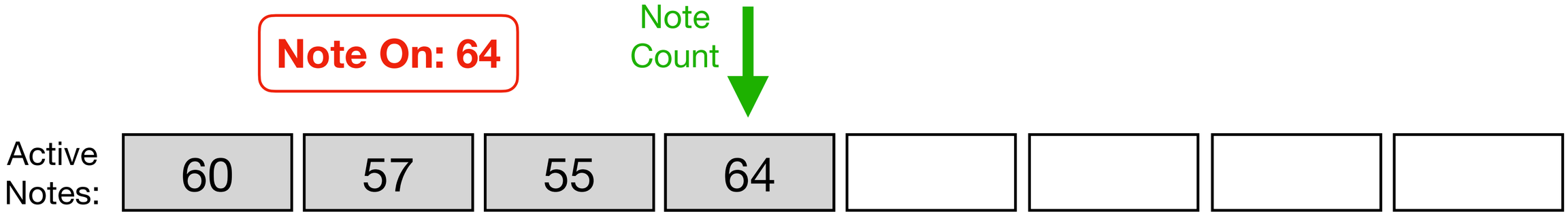
- Multiple notes at a time
- Closer to the familiar piano
- Each note has its own independent oscillators, filters, envelopes etc.
- There might be a maximum polyphony
- Some later analog synths and many digital ones work this way

Monophonic synths

- One note at a time seems easy: but which note?
  - The MIDI keyboard might still generate multiple notes
  - Typically, a monosynth uses the most recent note it receives
  - Our example has worked this way so far
- What happens when there are multiple notes active, and one note goes off?
  - In our simple example, the synth went silent: this isn't what we want
  - We would want it to revert to another note that was still held down
- Need a way to remember multiple key presses, in order of arrival
  - Play the most recent note, but remember the earlier ones
  - When the most recent note releases, go back to the one before
  - Only when all the notes release should the synth go silent

Remembering note order

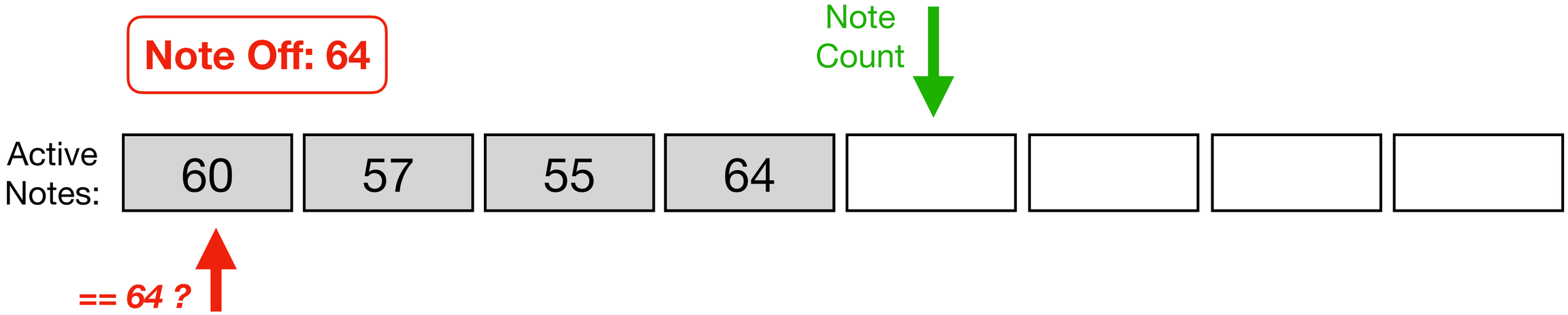
- We need two things to remember multiple key presses
  1. Array of currently active notes
     - This does not necessarily mean our synth is polyphonic: just that we remember which keys are held
     - Oldest notes occupy the earliest slots in the array
  2. Counter which keeps track of how many notes are in the array
     - Also doubles as a pointer to the first empty slot



- On receiving a Note On message:
  - Add the note number to the end of the array (at the location of the counter)
  - Increment the counter (which always points to the first empty slot)
  - Start the note as normal (set frequency, trigger envelope, etc.)

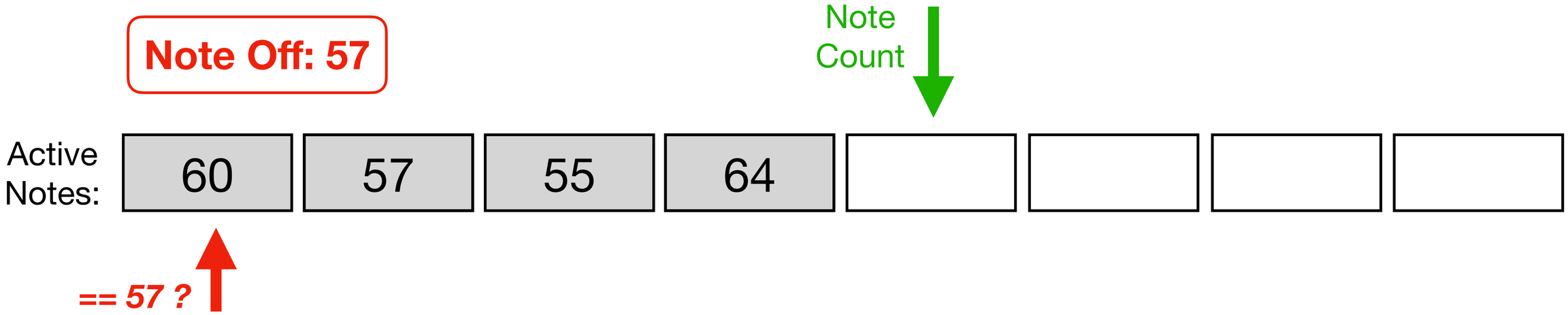
Remembering note order

- On receiving a Note Off message:
  - Step through the array and see if any of the elements match the note number
- If we find a match to the most recent note:
  - Remove it from the array, decreasing the count
  - Play the previous note (update oscillator frequency; might not retrigger the envelope)



Remembering note order

- On receiving a Note Off message:
  - Step through the array and see if any of the elements match the note number
- If we find a match to an older note:
  - Remove it from the array
  - Move back all the later elements, then decrease the count

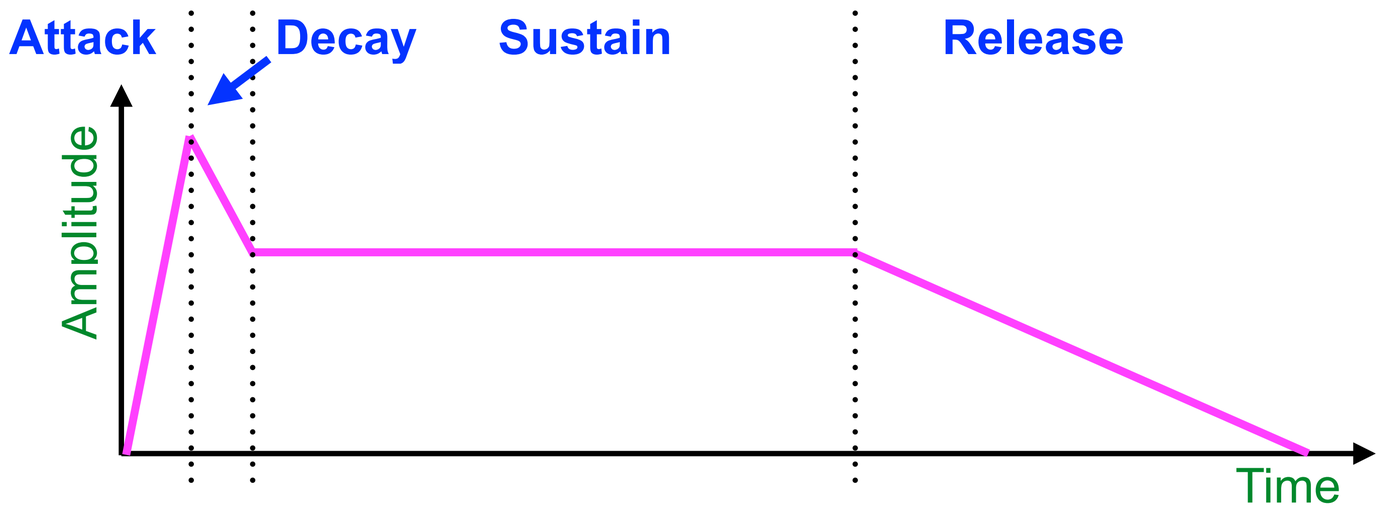


- If array becomes empty (count is 0), turn oscillator off

Note order task

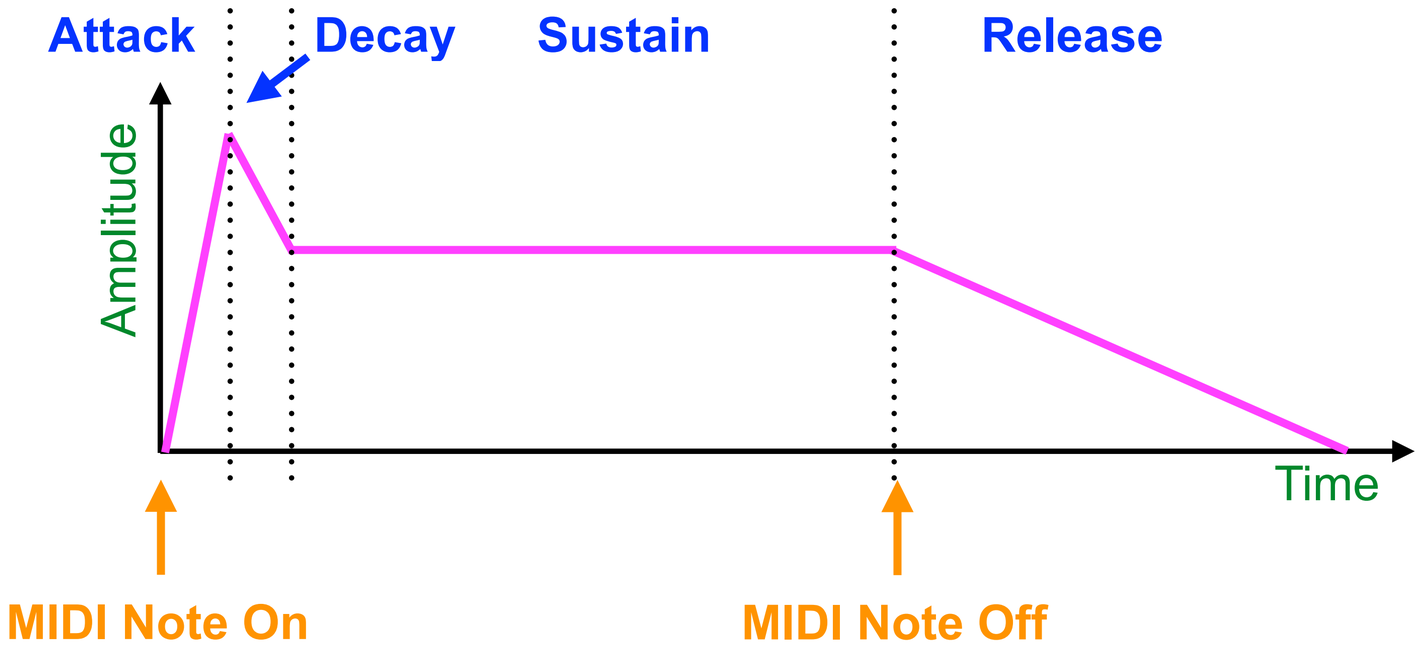
- Task: using the example midi-multinote
  - The code in `noteOn()` is already done for you, to add the most recent note to the array
  - Implement the code in `noteOff()` to keep track of multiple notes
  - When we get a note off message:
    - Check each element of the array to see if it matches the note number we just received
    - Remove that note from the array
    - If we removed the latest (last) element of the array, we will need to change the frequency
    - Move any later elements backward by one slot so the array doesn't contain any gaps
  - Variables are declared for you at the top of the file

ADSR envelopes



- Four phases (states) to an ADSR envelope:
  - Attack: level rises from 0 to maximum value (typically 1)
  - Decay: level drops from 1 to the sustain level (between 0 and 1)
  - Sustain: level remains constant at the sustain level
  - Release: level drops to 0 over a specified period of time
- The segments are often linear, can also be exponential

MIDI with ADSR



- Task: in project midi-adsr, use MIDI to control the ADSRs for amplitude/filter
  - All code to change is in `render.cpp`
  - Trigger the ADSRs in `noteOn()`
  - Release the ADSRs when the last note goes off in `noteOff()`
  - Process the ADSR values in `render()`, following Lecture 14

Gates and triggers

- In addition to control voltages (CVs), some synth modules are controlled by gates and triggers
  - Both are effectively digital signals, with only two states (on and off)
- A gate is a signal representing two states: open and closed
  - Usually, by convention, a positive voltage is open and 0V is closed
  - The ongoing value of the gate signal is what matters
- A trigger is a signal representing an instantaneous event
  - Used to begin envelopes, advance sequencers, and many other applications
- MIDI Note On messages might control both gates and triggers
  - The gate will indicate whether the synth is currently sounding
  - The trigger will create the attack phase of the ADSR envelope
- Task: in midi-adsr, make the ADSR trigger only on the first Note On received
  - Don’t retrigger the ADSR if a note is already sounding, just change the pitch

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources