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 13: State machines

What you'll learn today:

Structure and uses of state machines
Debouncing buttons
Hysteresis

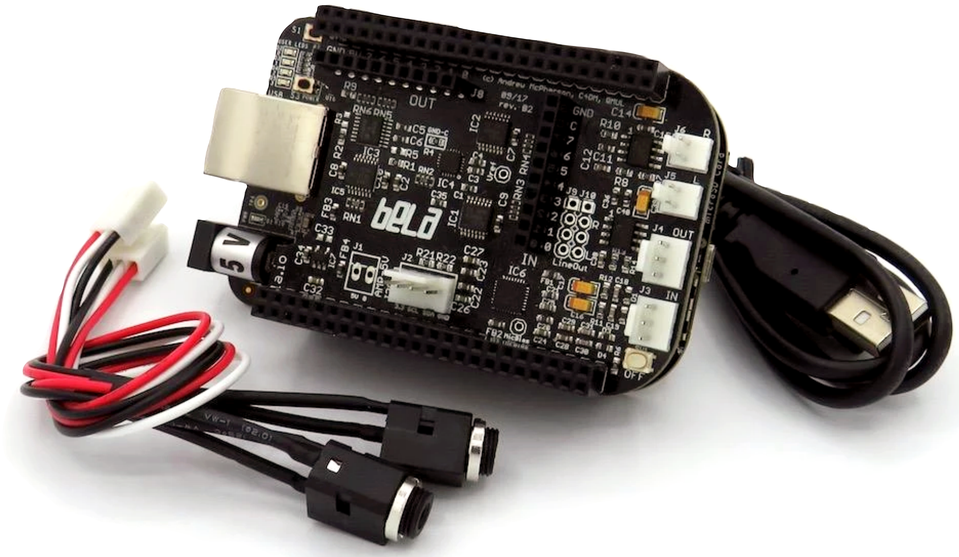
What you'll make today:

Metronome which marks the bar lines, button debouncer

Companion materials:

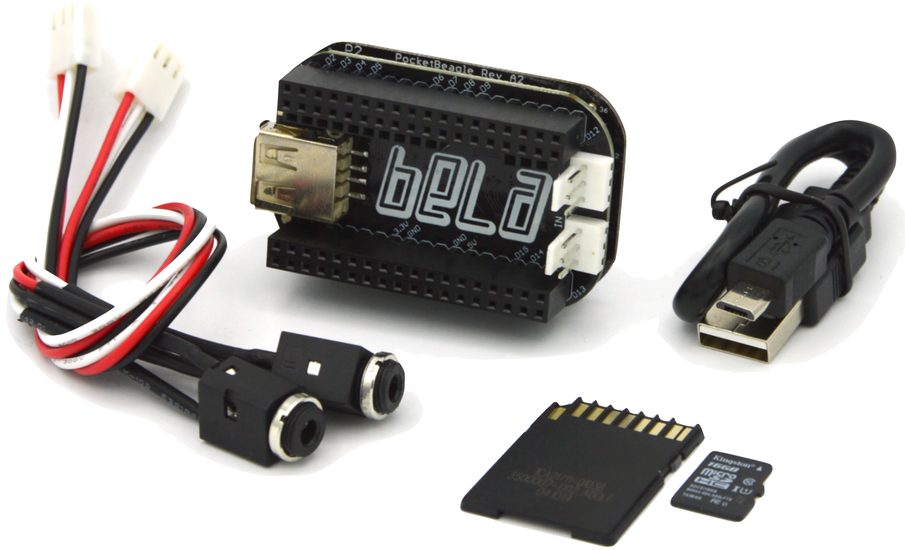
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

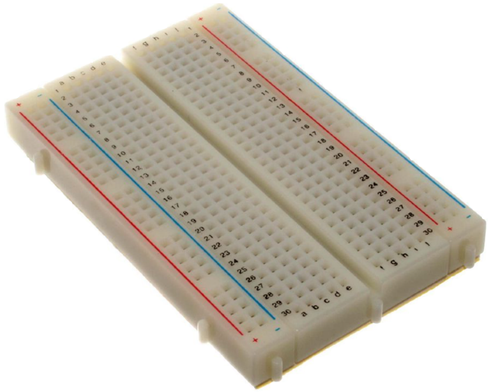
or

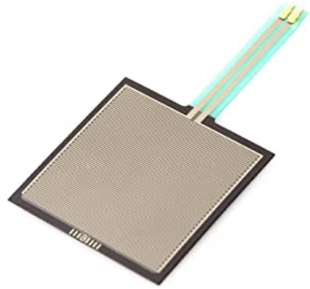


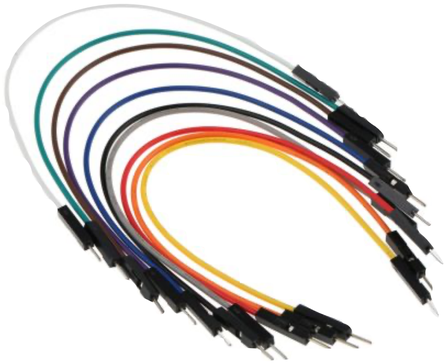
Bela Mini Starter Kit

[shop.bela.io]

Also needed for this lecture:

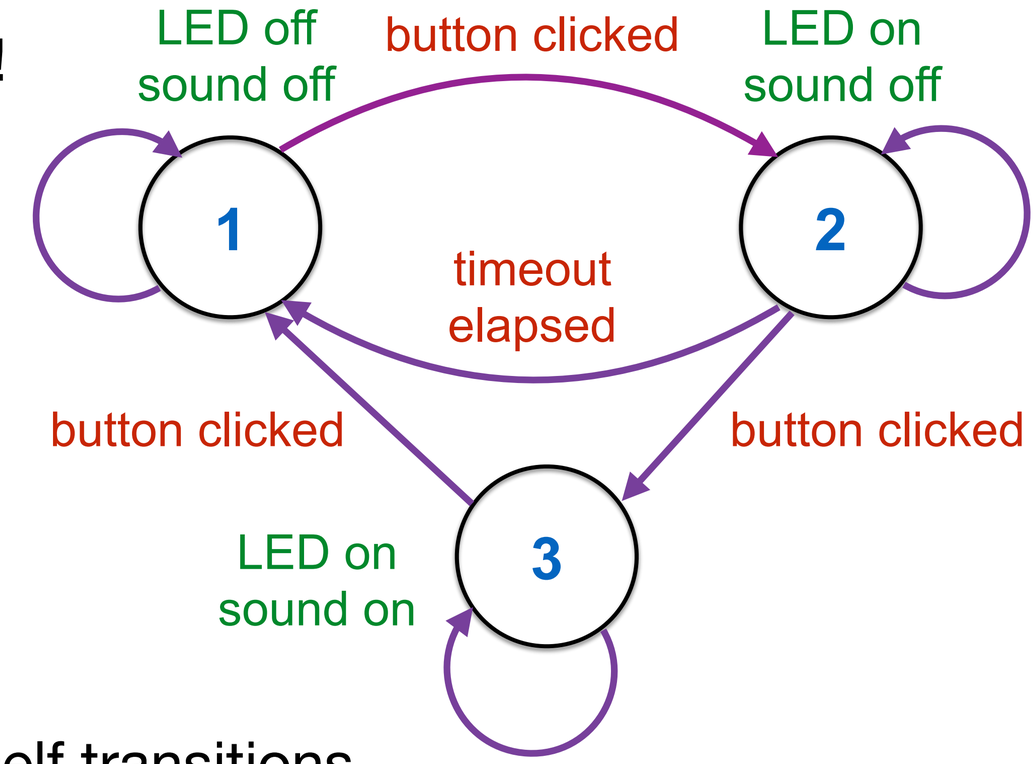






State machines

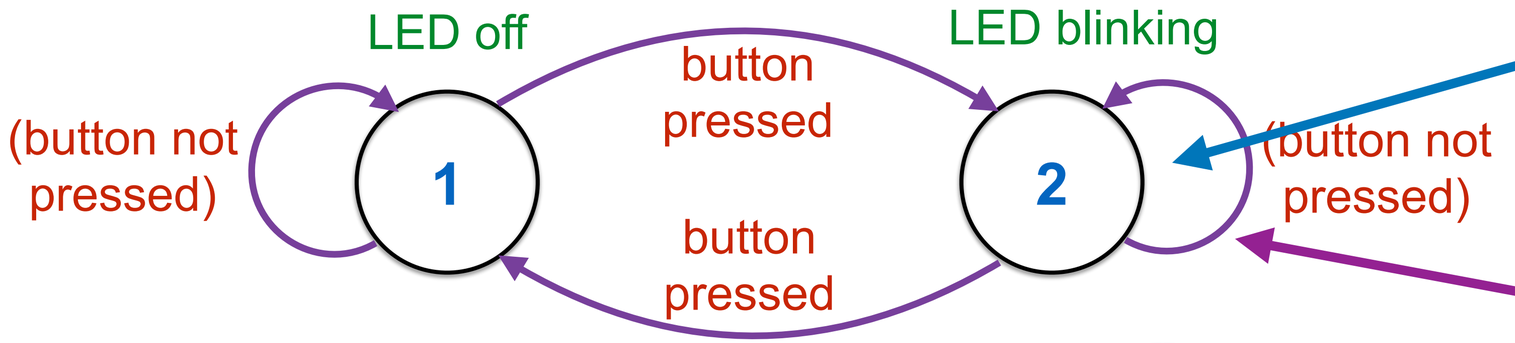
- A finite state machine (or FSM or state machine) is a simple mathematical model of computation
  - Lots of useful systems can be modelled this way!
- Four requirements:
  1. Set of **states** that the system can be in
  2. Set of **transitions** between those states
  3. Set of **inputs** which determine the transitions
  4. Set of **outputs** generated by the machine
- Only a finite number of states, and behaviour is deterministic
  - Transitions can be from one state to another, or self transitions



Simple state machine example

- Consider a simple LED blinker
- Push button to start blinking, push button again to stop:

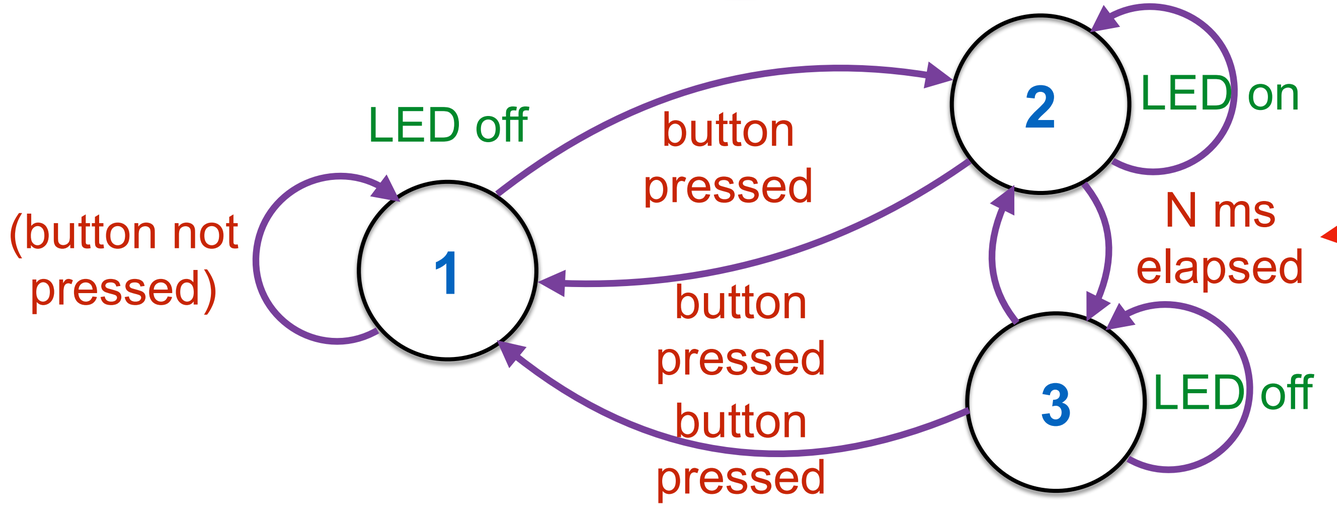
Version 1:



At any given time, the system is in either state 1 or state 2 (but not both)

Sometimes these self transitions will be omitted for clarity (they are still implied)

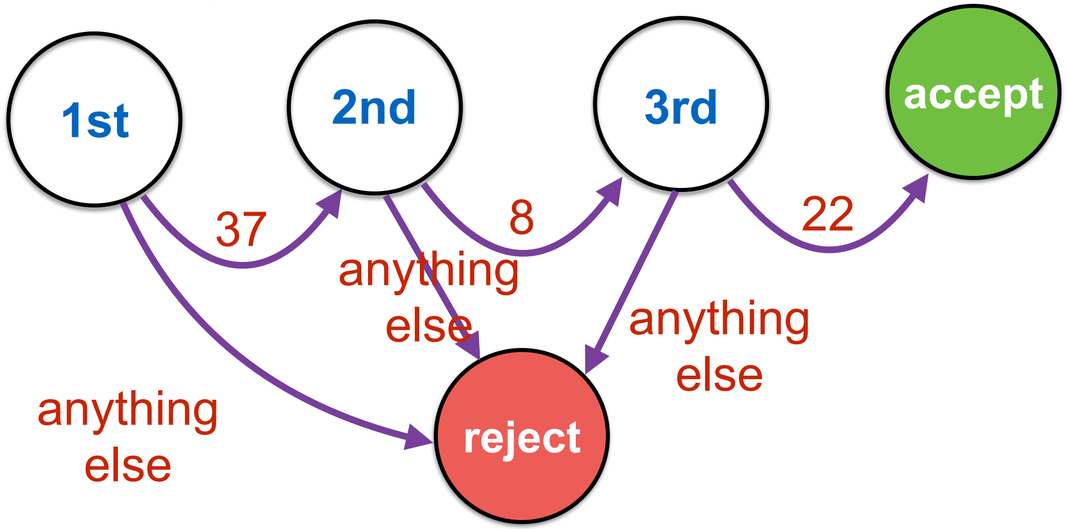
Version 2:
(equivalent)



Notice: time can be an input that causes a state transition

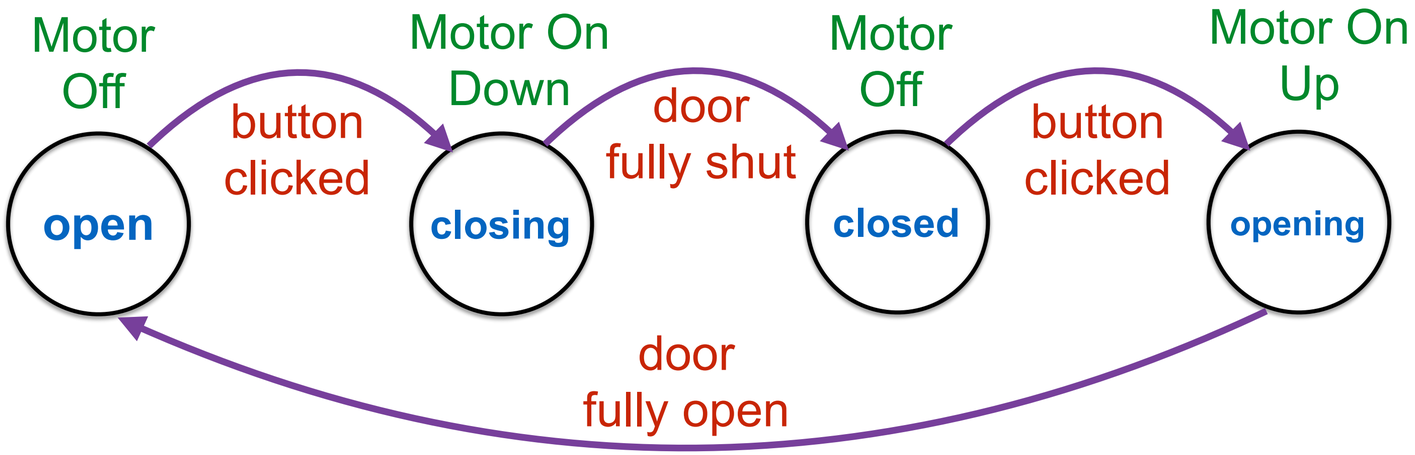
Types of state machine

- Finite state acceptors or classifiers
  - This subclass of state machine takes input only
  - Recognises sequences of inputs by reaching a particular state when a valid sequence is received
- Example: combination lock
  - States: whether we're awaiting the first, second or third number to be entered
  - Inputs: which numbers you turn to on the dial



Types of state machine

- Finite state transducers
  - Have both input and output
  - Most practical cases we work with will be these
- Example: garage door opener
  - States: door open, closing, closed, opening
  - Transitions: only certain possibilities
  - Inputs: button click, door sensor
  - Outputs: motor control

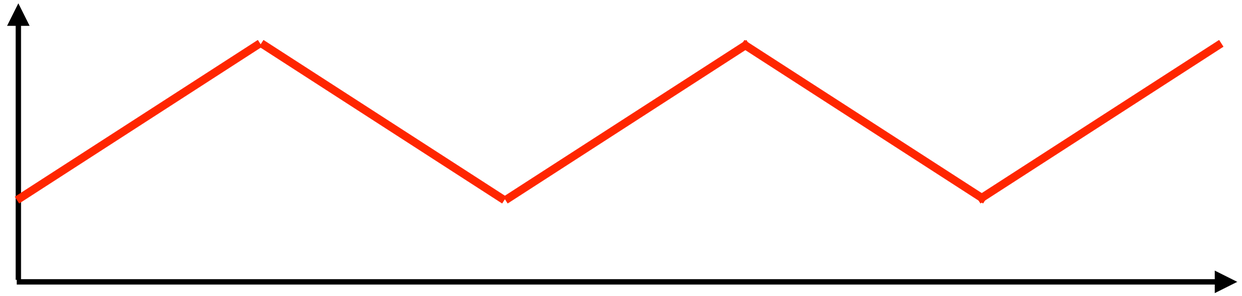


Generating outputs

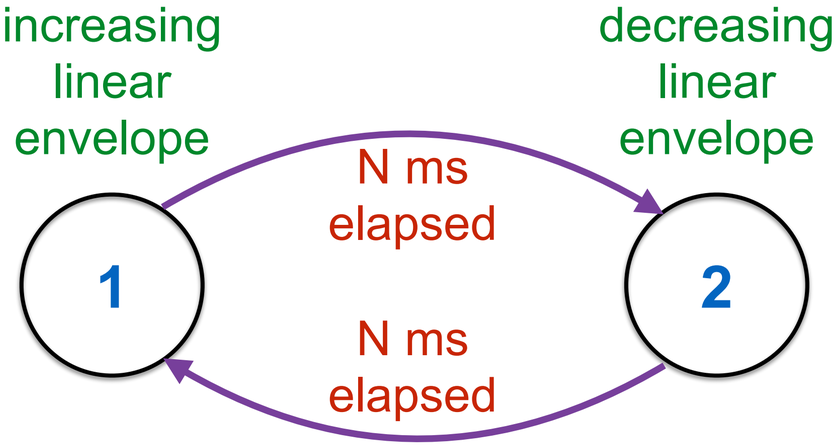
- When are the outputs of a state machine generated?
  - Whenever we enter a particular state (entry actions)
  - As long as we remain in a particular state (could think of this as a self-transition)
  - Whenever we leave a particular state (exit actions)
- Different models of state machines
  - Moore model: outputs depend only on the state
  - Mealy model: outputs depend on the state and its inputs
- We will treat this topic very informally!
  - Practical advice: choose whatever implementation is the clearest, with the least duplication of code
- Important: state machines are deterministic
  - Each set of input conditions should allow only one transition from any given state
  - Clearly specify the transition criteria (guards) in terms of all of the relevant inputs

State machine envelope example

- Back to our increasing and decreasing line segments:



- How many states?
  - 2: increasing and decreasing
- What determines the transitions?
  - Elapsed time (or reaching the desired level)
- Any other inputs?
  - Not in this example
- What are the outputs?
  - Linear envelopes going upwards and downwards



State machine implementation

Practically speaking, coding a state machine needs:

1. One or more variables to keep track of the state
   - If using a state machine in `render()` or across different functions, you will need global variables to remember the state
   - Could use a single variable to hold all possible states
     - e.g. assign a unique number to each state
     - the variable holds the number of whichever state we're in
   - Or might be conceptually clearer to use multiple variables
     - e.g. a boolean for whether you're in a particular state (on/off for example), plus another variable to say which of several "on" states you might be in
2. A set of `if()` statements to check the state
   - Check which state the system is in, using the variables above
   - Take actions and look for transitions accordingly

```
int gState = 0;
```

```
if(gState == 0) {
    // Actions and transitions
    // for state 0
}
else if(gState == 1) {
    // Actions and transitions
    // for state 1
}
// etc.
```

State machine implementation

- For clarity, it can be useful to name each state
  - As opposed to just giving each one a number
  - Define constants at the top of the code file to associate each name with a number
- C/C++ `enum` keyword is useful here
  - enumerate: define lots of (integer) constants at once
  - Unless otherwise specified, the constants are assigned numbers in increasing order
  - After you define the names, never use the numbers directly in your code
  - Here, the "k" prefix means constant: a coding style, not a requirement

```
enum {
    kStateOpen = 0,
    kStateClosing,  // = 1
    kStateClosed,   // = 2
    kStateOpening   // = 3
};

int gState = kStateOpen;

void myFunction() {
    if(gState == kStateOpen) {
        // ...
    }
    else if(gState == kStateClosing) {
        // ...
    }
    // etc.
}
```

State machine implementation

Practically speaking, coding a state machine needs:

3. Other `if()` statements using inputs to decide on transitions between states
   - Look at the inputs, decide which state to go to next
   - Self-transitions (no state change) are possible, and may even be the most frequent event
   - Remember, time can be one of your inputs
     - Can measured elapsed time by counting samples
     - Remember to reset the counter when needed (e.g. when changing state)

Changing state is easy!
Just need to assign a new value into the state variable, and next time something different will happen

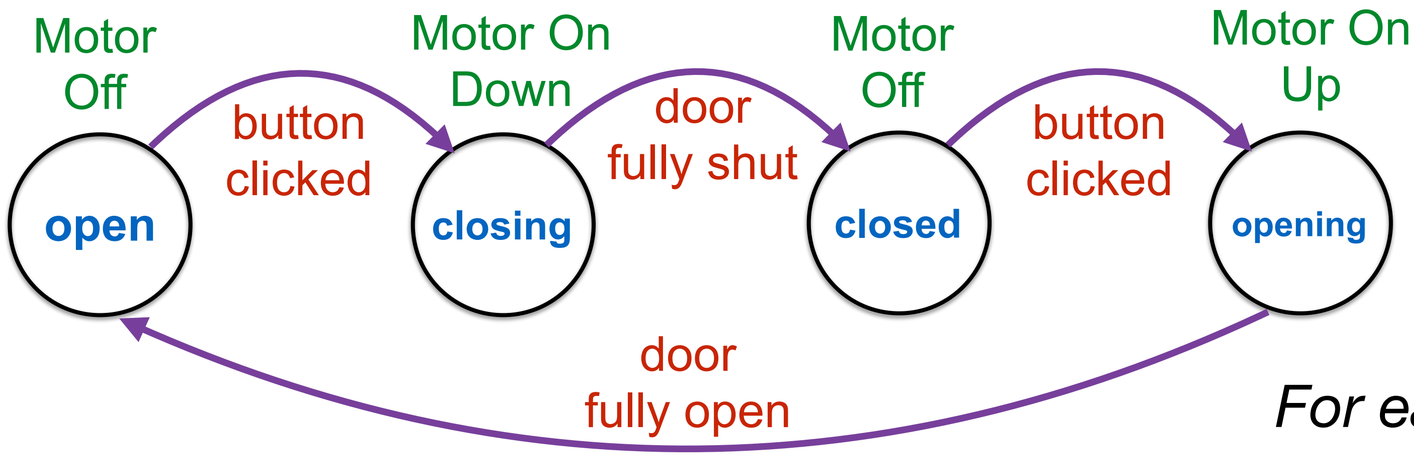
```
if(gState == kStateOpen) {
    motorOff(); // Action for this state
    if(buttonWasPushed) {
        gState = kStateClosing; // Transition
    }
}
else if(gState == kStateClosing) {
    motorDown(); // Action for this state
    if(doorHasClosed) {
        gState = kStateClosed;
    }
}
// etc.
```

State machine implementation

Practically speaking, coding a state machine needs:

4. Clear planning on when to check for transitions
   - In digital logic, state machine is often driven by a clock
   - Edges of the clock determine when machine checks its state and when it can transition
   - If using real-time audio, audio frame clock will work
     - Check for actions and transitions on each iteration of the `for()` loop in `render()`
     - Each frame, perform actions for a particular state, then decide what state to go to for the next frame
   - Also consider which actions should be performed once on entry to a state (or on transition, or exit), and which should happen every cycle we're in a state

State machine example



At the top of the file:

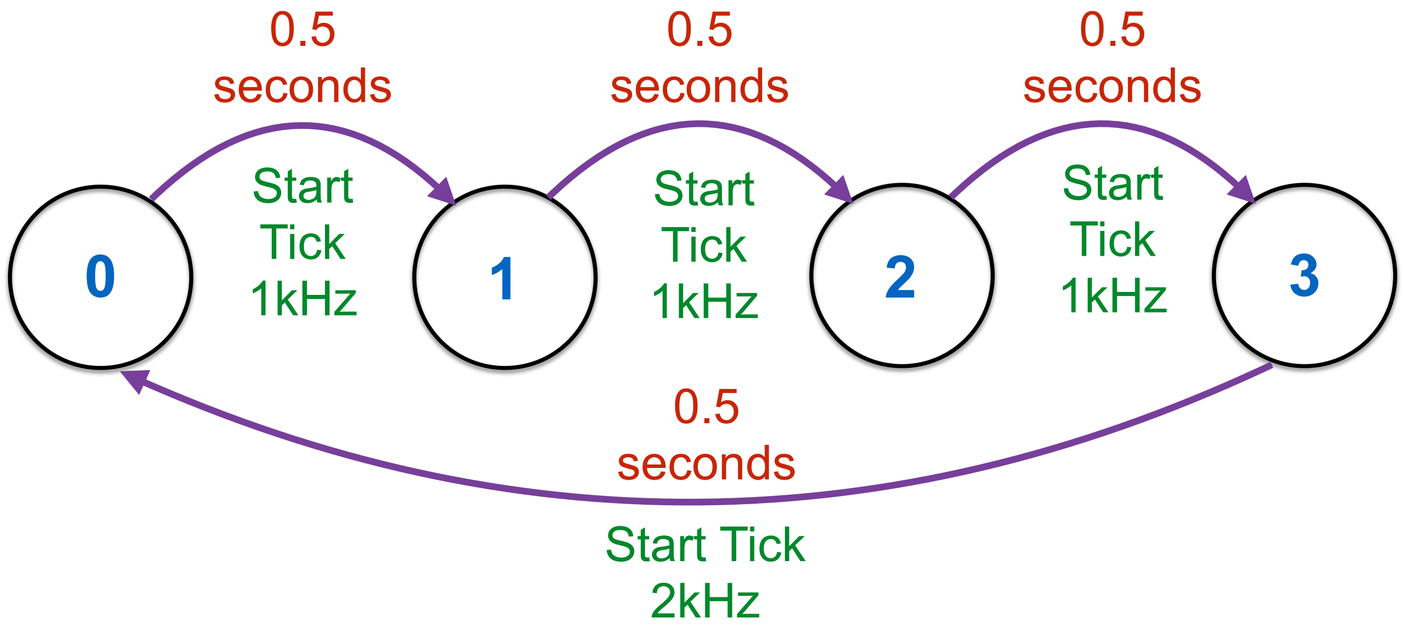
```
// Defining four possible states:
enum {
    kStateOpen = 0,
    kStateClosing,  // = 1
    kStateClosed,   // = 2
    kStateOpening   // = 3
};

// This tells us which state we're in:
int gState = kStateOpen;
```

For each iteration in real time:

```
if(gState == kStateOpen) {
    motorOff();       // Action for this state
    if(buttonWasPushed) {       // Check input
        gState = kStateClosing; // Transition
    }
}
else if(gState == kStateClosing) {
    motorDown();     // Action for this state
    if(doorHasClosed) {        // Check input
        gState = kStateClosed; // Transition
    }
}
// etc.
```

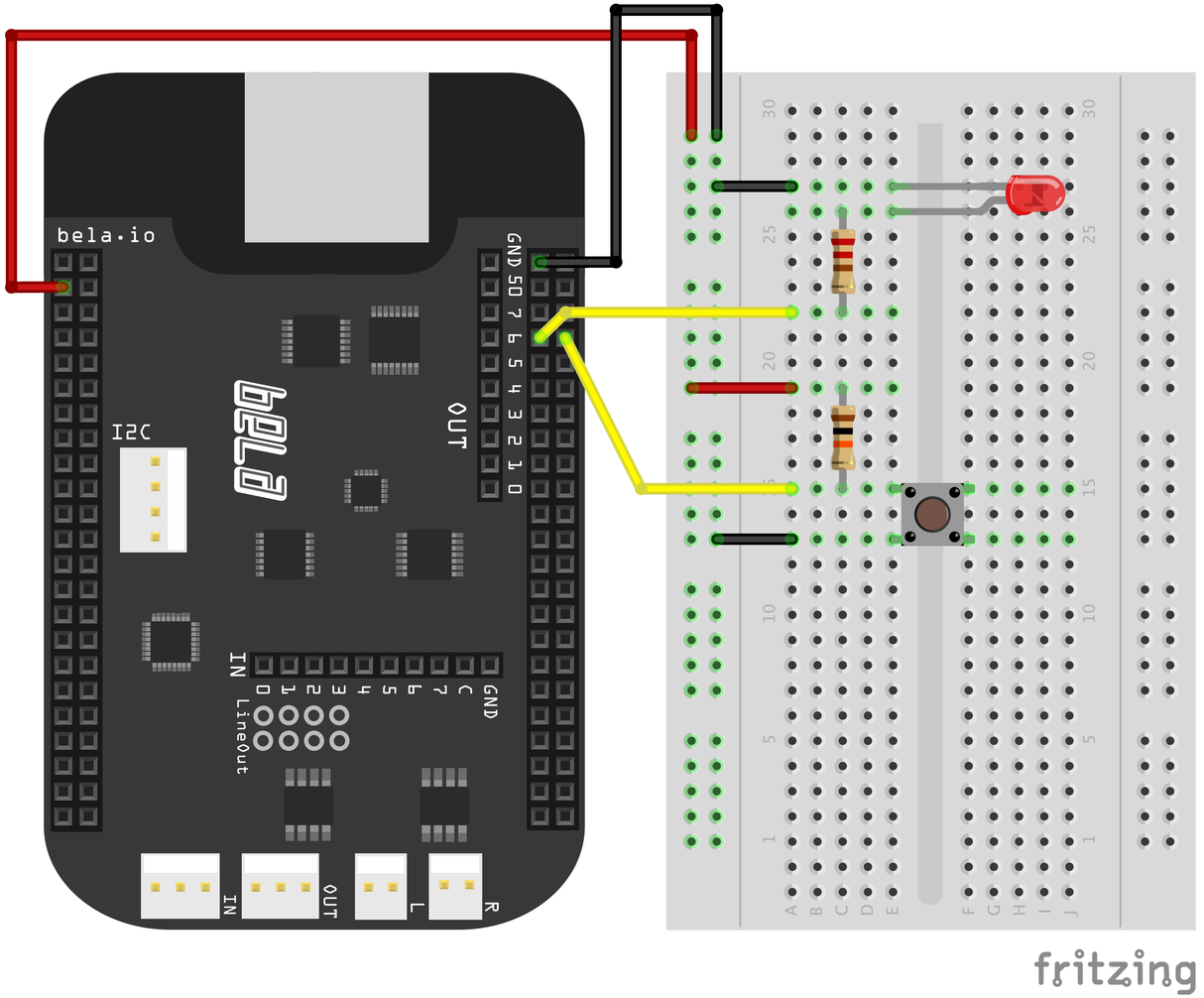
State machine task

- Task: using the metronome-envelope example
  - Make every fourth tick have twice the frequency
  - `gFrequency` = 2000 for tick 0; `gFrequency` = 1000 for ticks 1-3
  - Create a state variable to keep track of which of 4 states we're in (i.e. which tick in the cycle)
  - Reset the envelope and change frequency on entering the new state (i.e. on the transition)

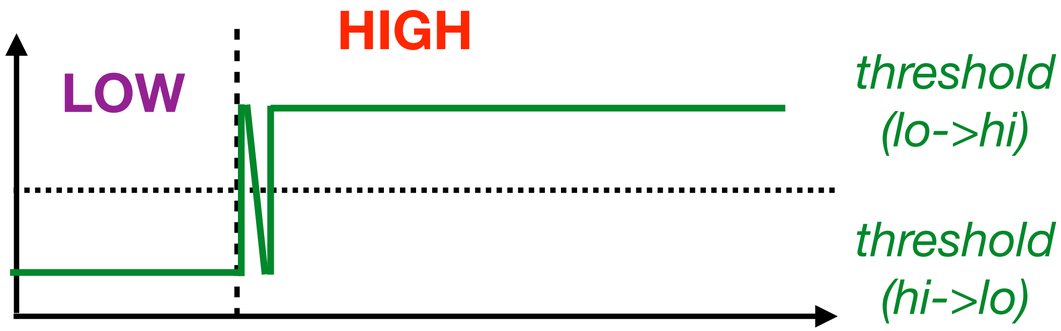


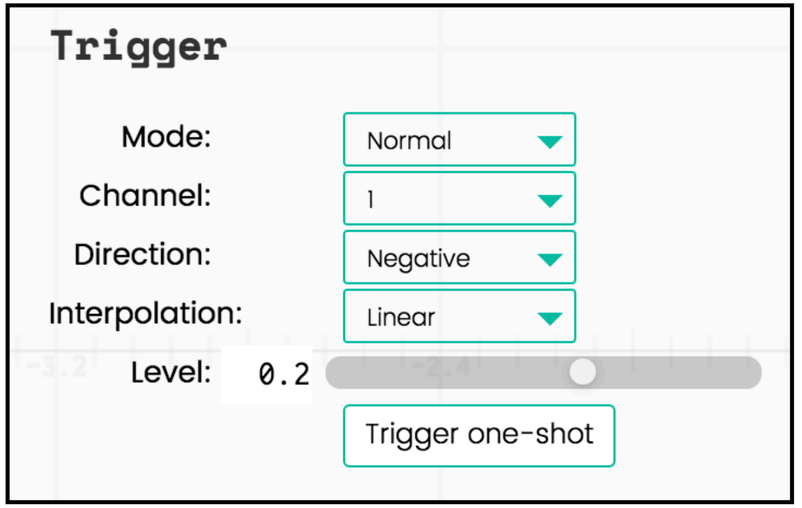
State machine task 2

- Task 2: using the metronome-envelope example
  - Add a button to start and stop the metronome
  - Add a 5th state to represent “stopped”
  - When the metronome is stopped, a click should always start at tick 0 (2kHz)
  - When the metronome is running, a click should go to the stopped state
  - Don’t have to immediately silence the sound, just don’t start another tick



Debouncing

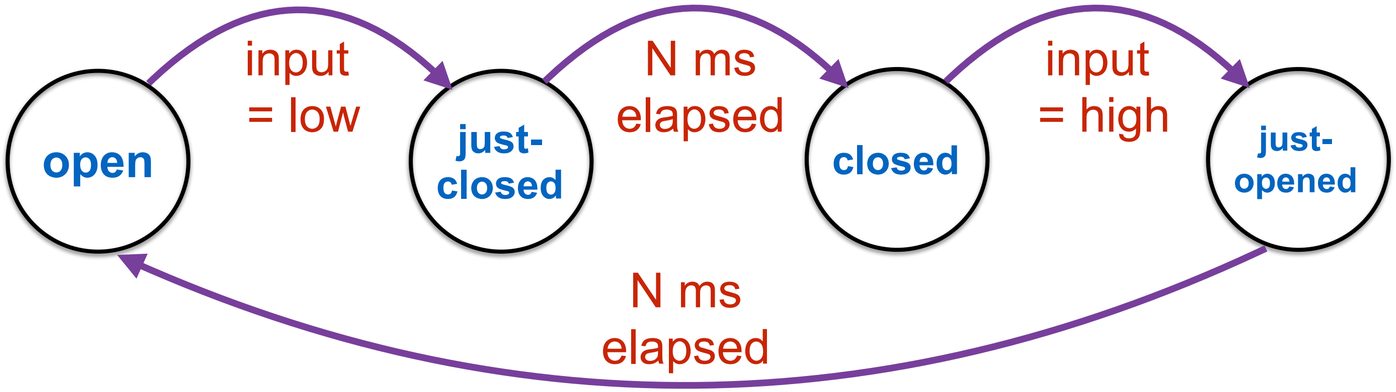




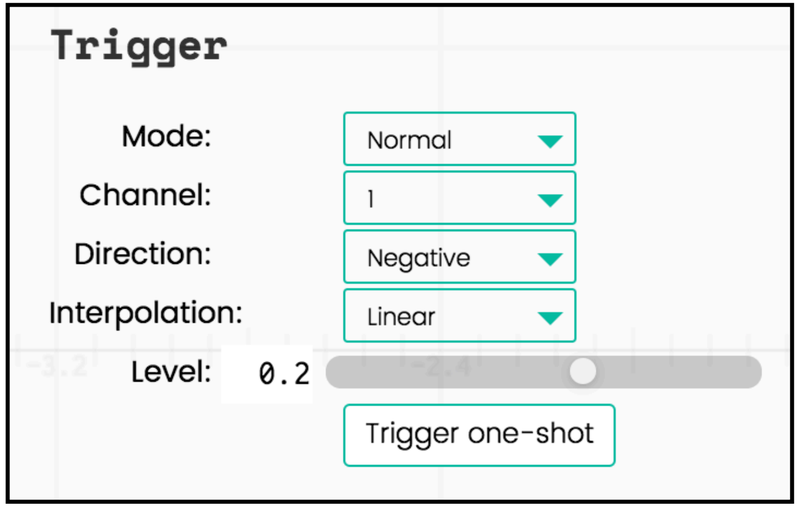
- A button is a spring-loaded set of plates which can physically bounce when pressed and released
  - This can cause errors in the signal
  - A process called debouncing compensates for false triggers
- Several approaches to debouncing
  - Simplest is a timer that ignores input from the switch for a window after a state change
- Task: run the scope-button example, look at the button signal on the Bela scope
  - Set to the trigger mode to normal, negative, threshold > 0 to see if you can spot the bouncing
  - How long does the bouncing last?

Debouncing state machine

- We can consider debouncing as a simple state machine:

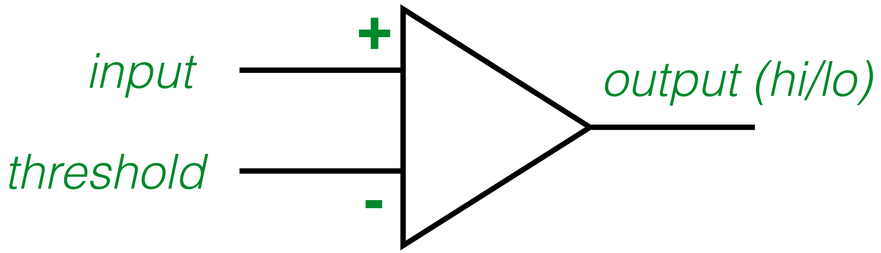


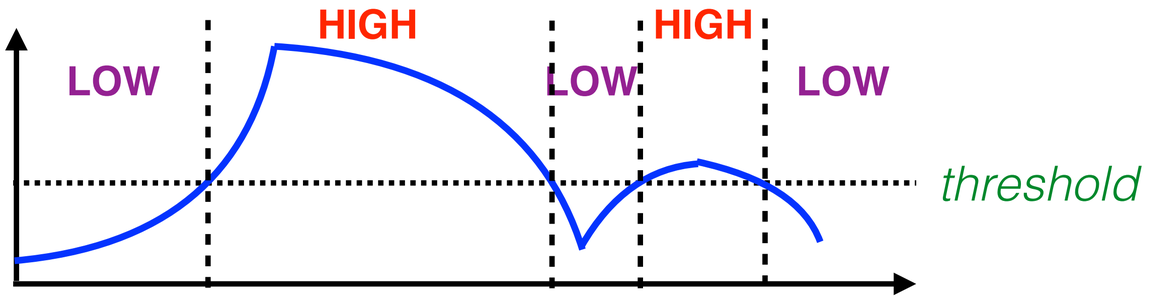
  - Four states: switch open; switch closed; plus two transient states
  - The transient states wait a short time before allowing the button to register another change
    - How long should the wait time be?
    - How do we handle the timer for the wait?
    - Remember what time we entered the state (or reset a counter)



- Task: using debounce example, implement a debouncing state machine
  - When entering the just-closed state, toggle an LED on or off
  - Toggling the LED will also need a second variable to keep track of whether it was on or off

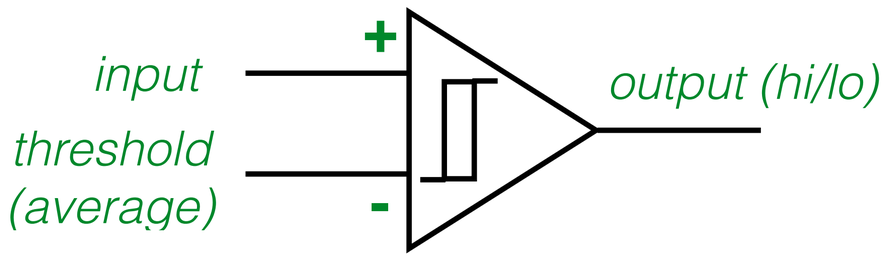
Comparator

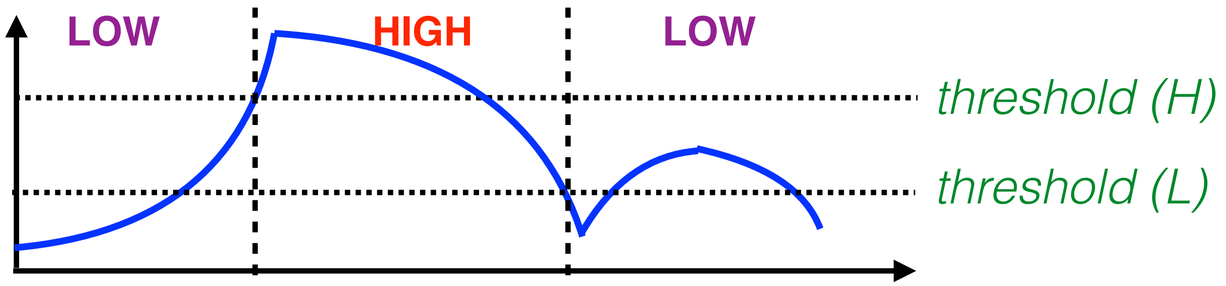




- A comparator is a simple analog component
  - Takes two input signals, outputs high or low depending on which one is bigger
  - In practice, we often compare an input signal to a fixed threshold
  - Note: continuous input, discrete output (only 2 states)
- In code, what kind of statement is this?
  - `if()`
  -
```
if(value > threshold)
  output = 1; // or run some action
else
  output = 0; // or run some other action
```

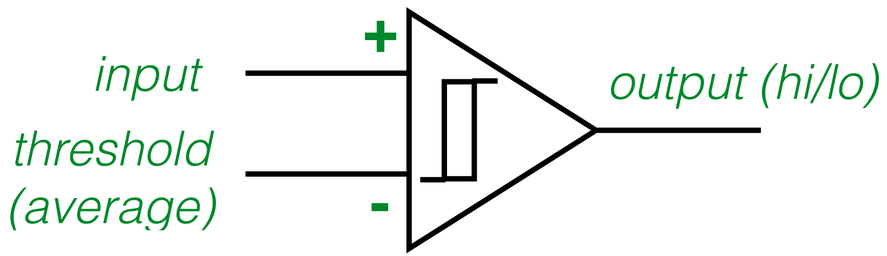
Comparator with hysteresis

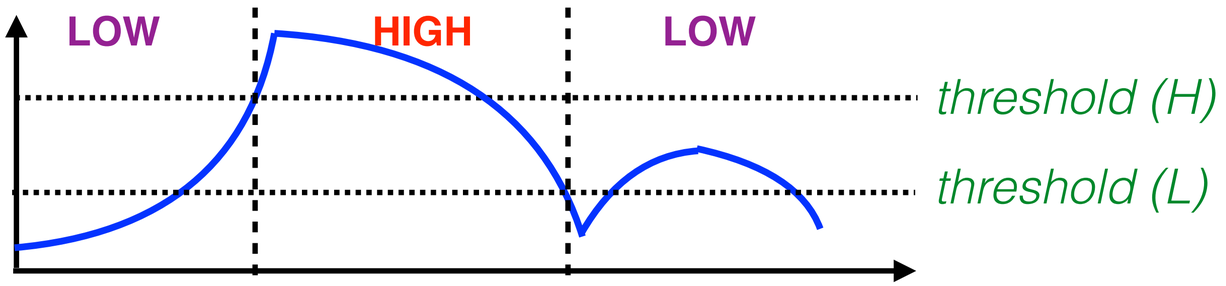




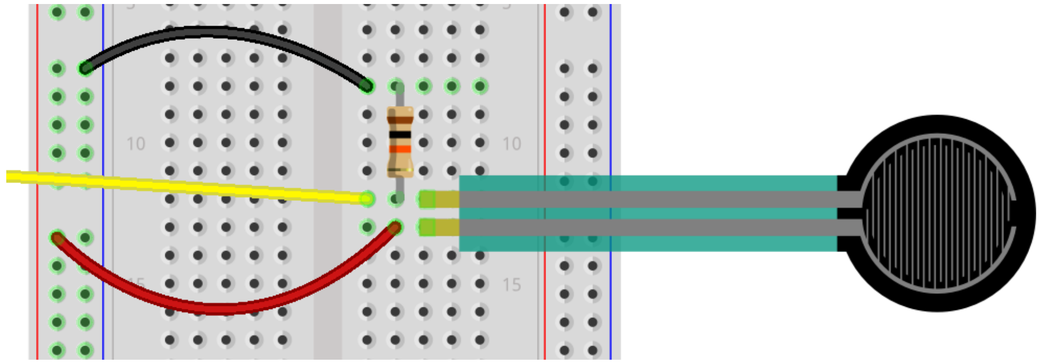
- Comparator with hysteresis (or Schmitt trigger)
  - Output depends on the input and the past state
  - Threshold changes depending on whether last output was high or low
    - Higher threshold for L→H transition, lower threshold for H→L
  - Hysteresis is useful for noise immunity
    - Small changes will not cause output to rapidly flip back and forth
- Digital implementation is a form of state machine
  - Two states: high and low
  - Transition conditions are different for each state

Hysteresis task





- Task: using project hysteresis-comparator
  - Implement a hysteresis comparator on analog input 0
  - Connect an FSR to the analog input
  - Make the thresholds 0.75 (L→H) and 0.25 (H→L)
  - Use an LED to indicate the output state
  - Optionally, print a message to the console when state changes
    - e.g. `rt_printf("LOW ---> HIGH");`



Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources