Introduction to Design (2)

Microcontrollers and Interfacing

Week 11

digital input, debouncing, interrupt-driven input

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

KUAS
KYOTO UNIVERSITY of ADVANCED SCIENCE

this week

digital input

- push-button switches
- pull-up resistors

debouncing

- in software
- in hardware

non-timer interrupts

- for digital input

digital input

configuring ditial pin as input: `void pinMode(`n`, INPUT)`

- pin becomes *high impedance* (high resistance)
- microcontroller does not generate a voltage on the pin
- microcontroller *senses* the voltage *applied to* the pin

reading the voltage on the pin: `int digitalRead(`n`)`

- returns HIGH or LOW

transition from LOW to HIGH is *somewhere* between GND (0 V) and V_{cc} (5 V)

- how do we find the exact voltages involved?

digital input

T_A = -40°C to 85°C, V_{CC} = 1.8V to 5.5V (unless otherwise noted)

Symbol	Parameter	Condition	Min.	Typ.	Max.	Units
V_{IL}	Input Low Voltage, except XTAL1 and $\overline{\text{RESET}}$ pin	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	-0.5 -0.5		$0.2V_{CC}$ (1) $0.3V_{CC}$ (1)	V
V_{IH}	Input High Voltage, except XTAL1 and $\overline{\text{RESET}}$ pins	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	$0.7V_{CC}$ (2) $0.6V_{CC}$ (2)		V_{CC} + 0.5 V_{CC} + 0.5	V
I_{IL}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin low (absolute value)			1	µA
I_{IH}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin high (absolute value)			1	µA

Notes: 1. "Max" means the highest value where the pin is guaranteed to be read as low
2. "Min" means the lowest value where the pin is guaranteed to be read as high

from this we can calculate

- the maximum voltage that is guaranteed to be LOW
- the minimum voltage that is guaranteed to be HIGH

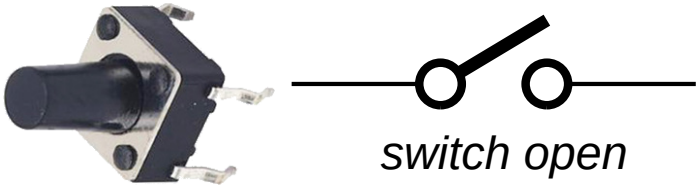
note: from the *leakage current* we can also calculate

- the effective input resistance of a digital I/O pin

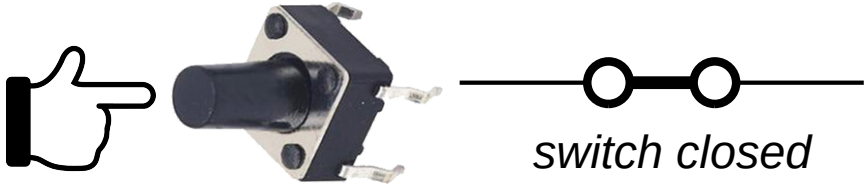
push-button switches

‘normally open’ push-button

- converts physical movement into a resistance change
- two contacts
- when button not pressed, contacts are open (not connected, infinite resistance)



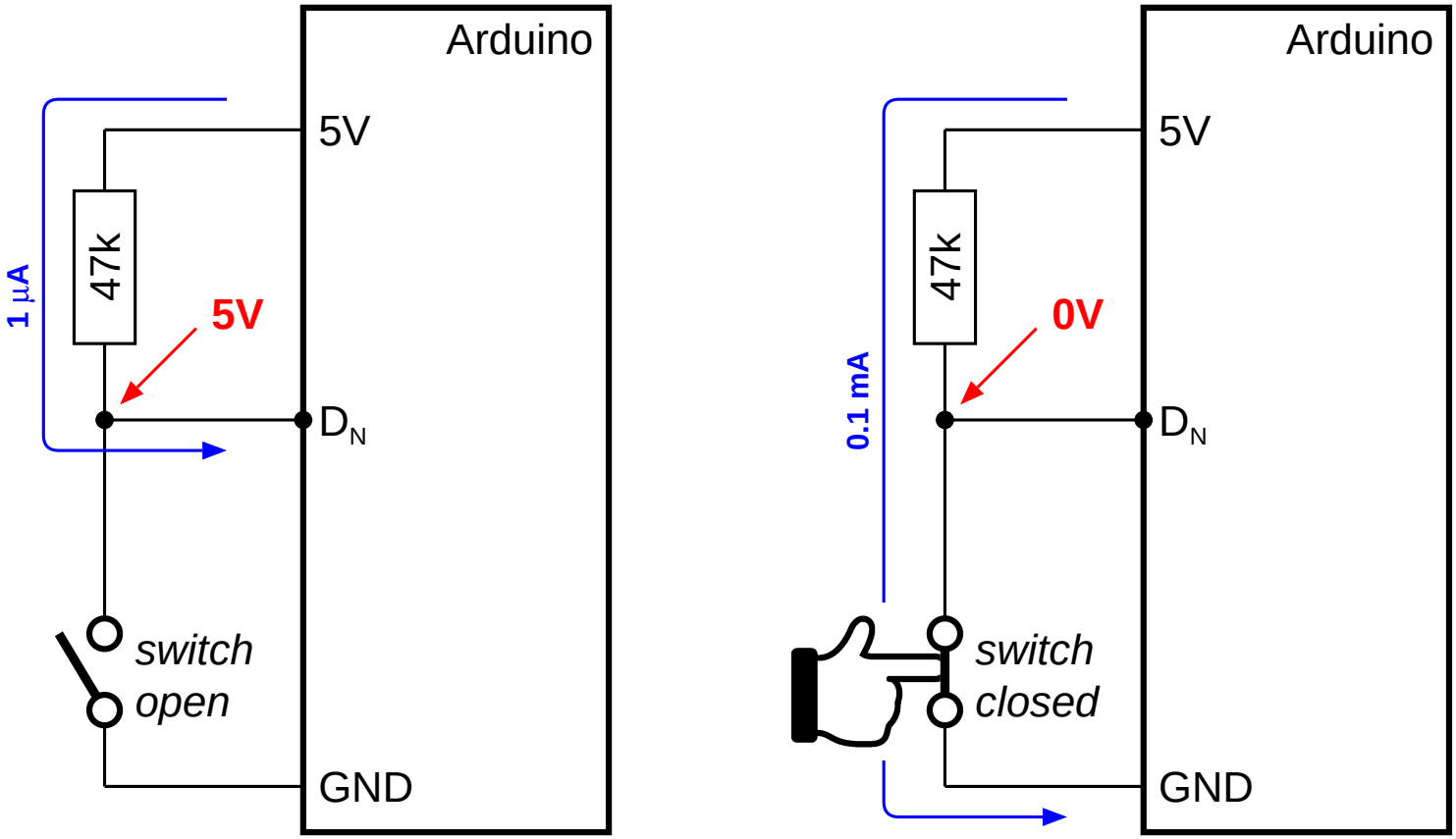
- when button pressed, contacts are closed (connected, zero resistance)



challenge: design a circuit that produces

- 5 V when a switch is open (not pressed), or
- 0 V when a switch is closed (pressed)

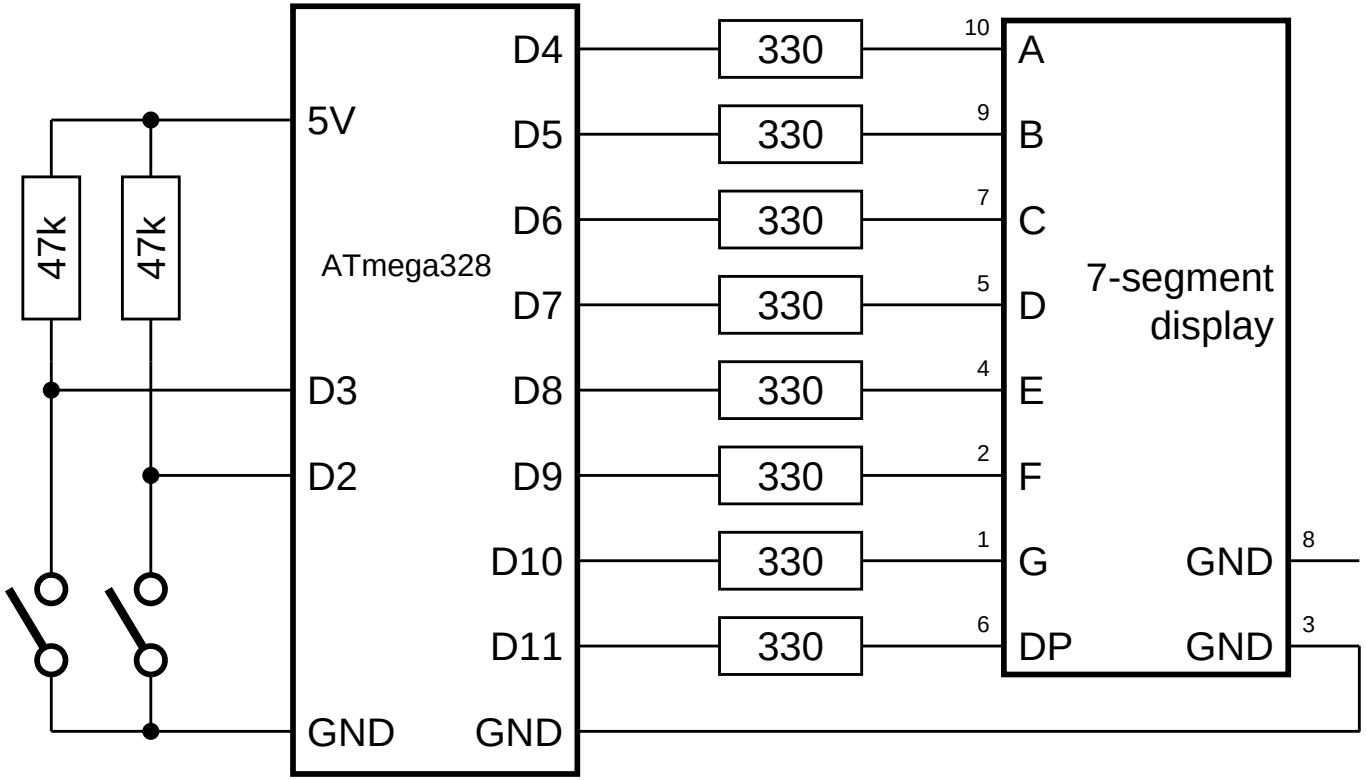
push-button switches



digital input circuit

two switches and seven-segment display can be connected at the same time

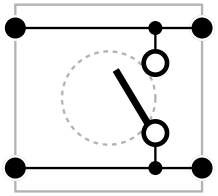
- set mode of digital pins 2 and 3 to `INPUT`
- pull-up resistors keep them `HIGH`
- pressing a switch pulls them `LOW`
- use `digitalRead()` to read the value (`HIGH` or `LOW`)



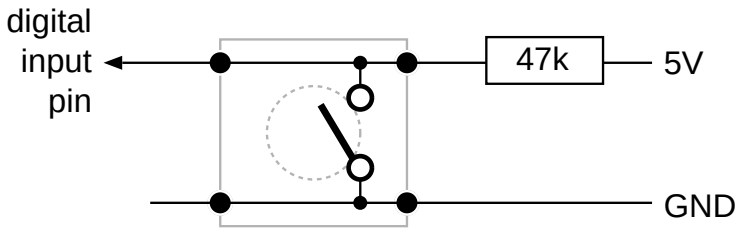
digital input breadboard layout

some switches have four terminals

usually these are arranged in pairs, internally connected from one side of the device to the other

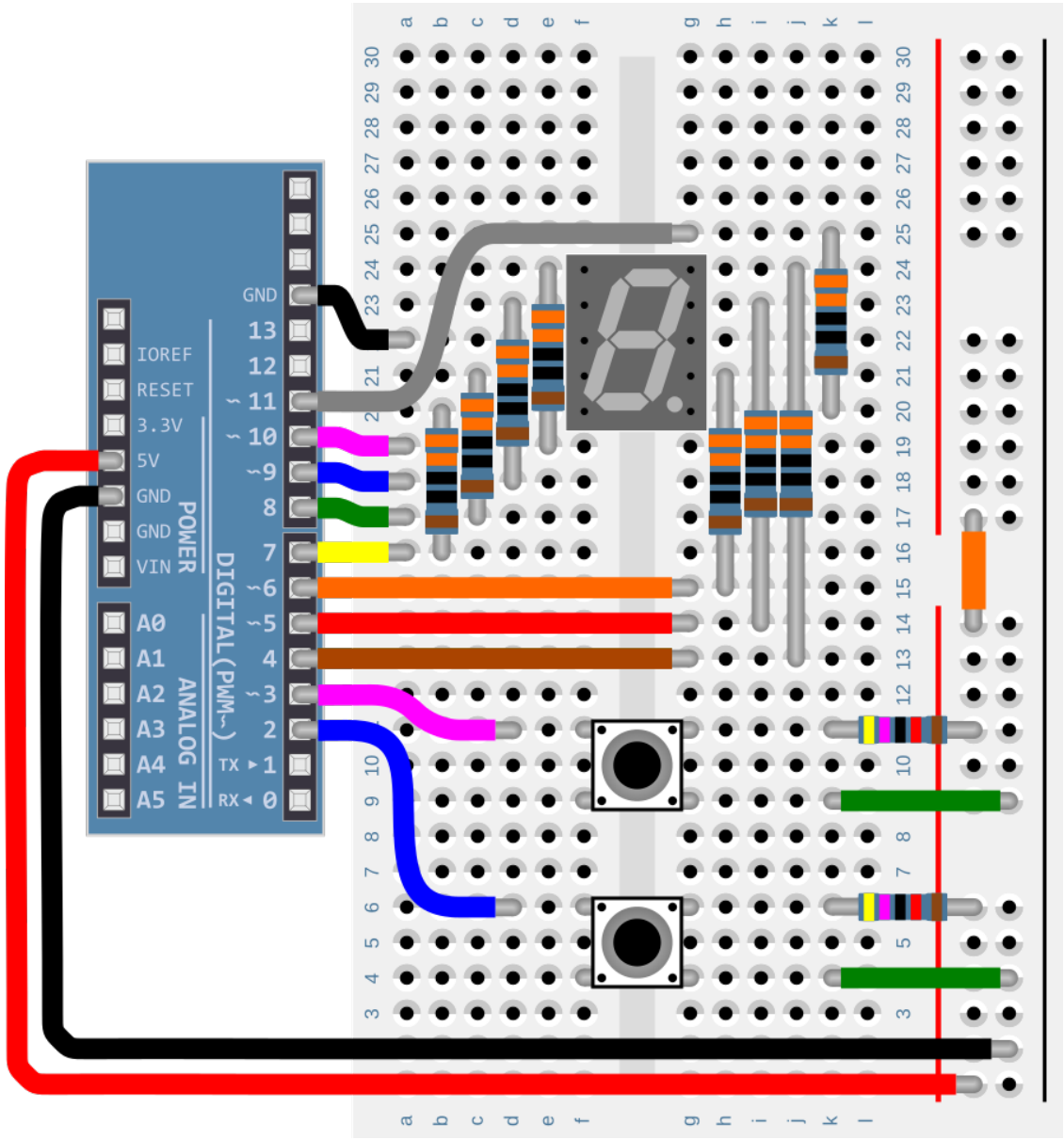


this is very convenient for our circuit



note: we have to connect the switches to digital pins 2 and 3 (which are the only pins that support interrupts)

to make room for them, move the 7-segment display connections up by two pins and modify the program accordingly



digital input software

pins connected to switches should be configured as `INPUT`s

```
void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
}
```

when a switch is pressed, the corresponding input changes from `HIGH` to `LOW`

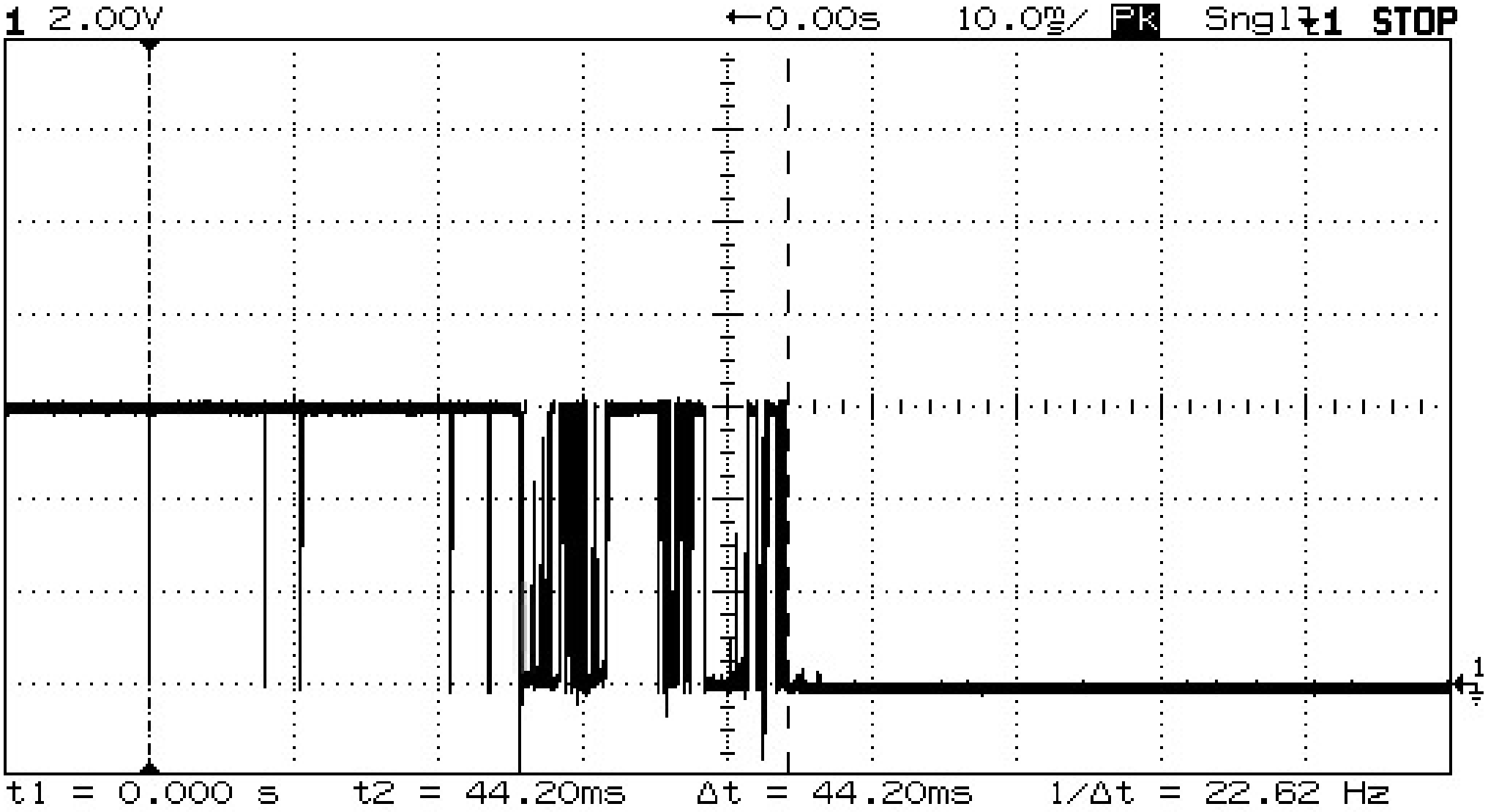
two consecutive `digitalRead()`s will therefore return different results

- the first is `HIGH` (switch open $\Rightarrow$ not pressed)
- the second `LOW` (switch closed $\Rightarrow$ pressed)

```
int oldState = HIGH;

void loop() {
  int newState = digitalRead(2);
  if (oldState == HIGH && newState == LOW) {
    // the switch was pressed
  }
  oldState = newState;
}
```

switch bounce



debouncing

when our circuit (sketch) is fast enough

- each transisition of the signal can be seen as a separate ‘button press’
- one physical press is counted many times

debouncing

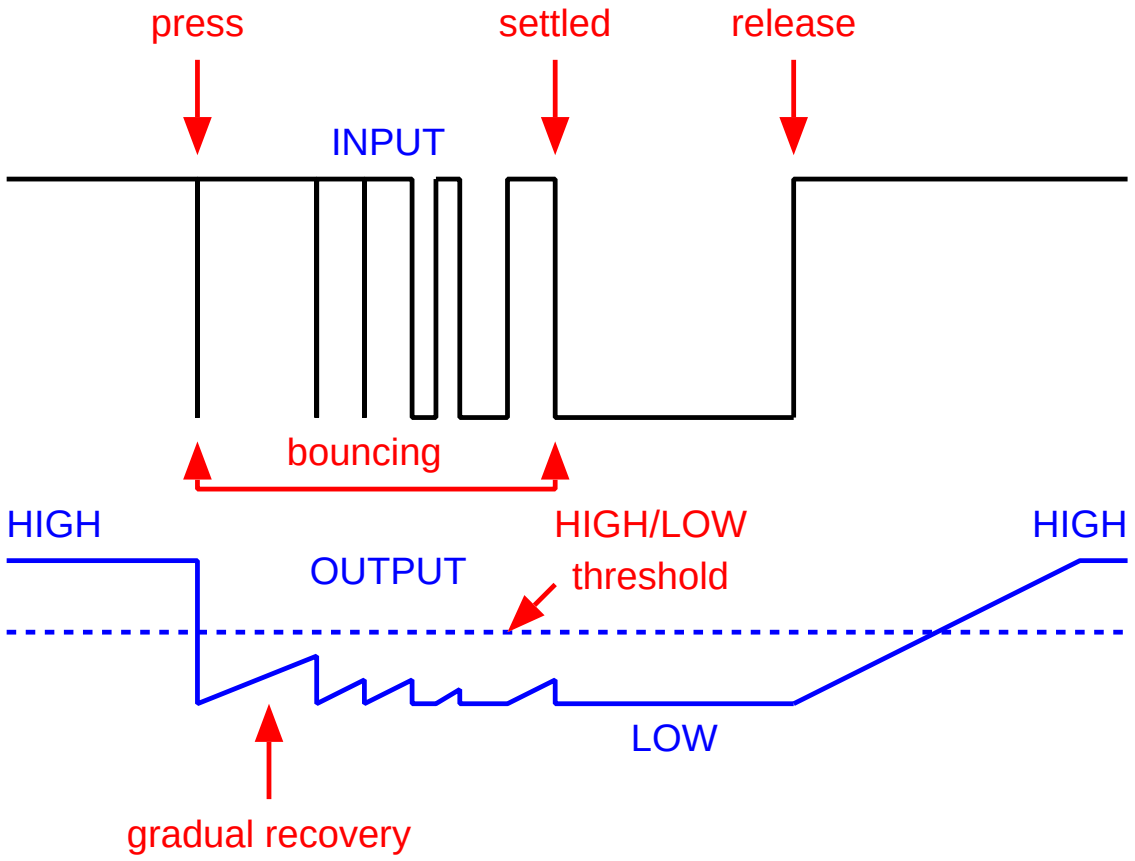
- attempts to elimintate the false ‘presses’
- each physical press is counted only once by software

can be done in hardware or in software

hardware debouncing

need a circuit that will

- react *quickly* to the initial press
- recover *slowly* from closed (`LOW`) to open (`HIGH`)
- delay recovery until all ‘bouncing’ has ceased

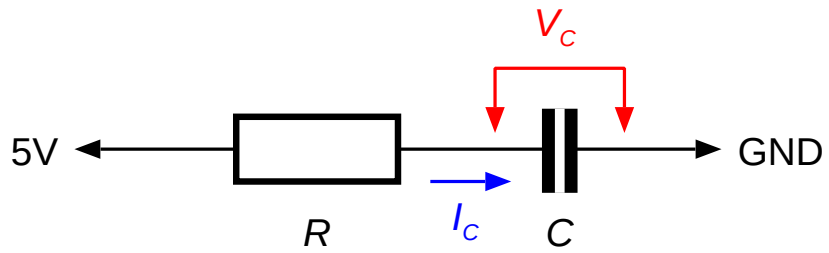


capacitors

time delays (or filtering high-frequency noise) often done with a capacitor

- stores charge
- voltage proportional to stored charge
- rate of charging (current)

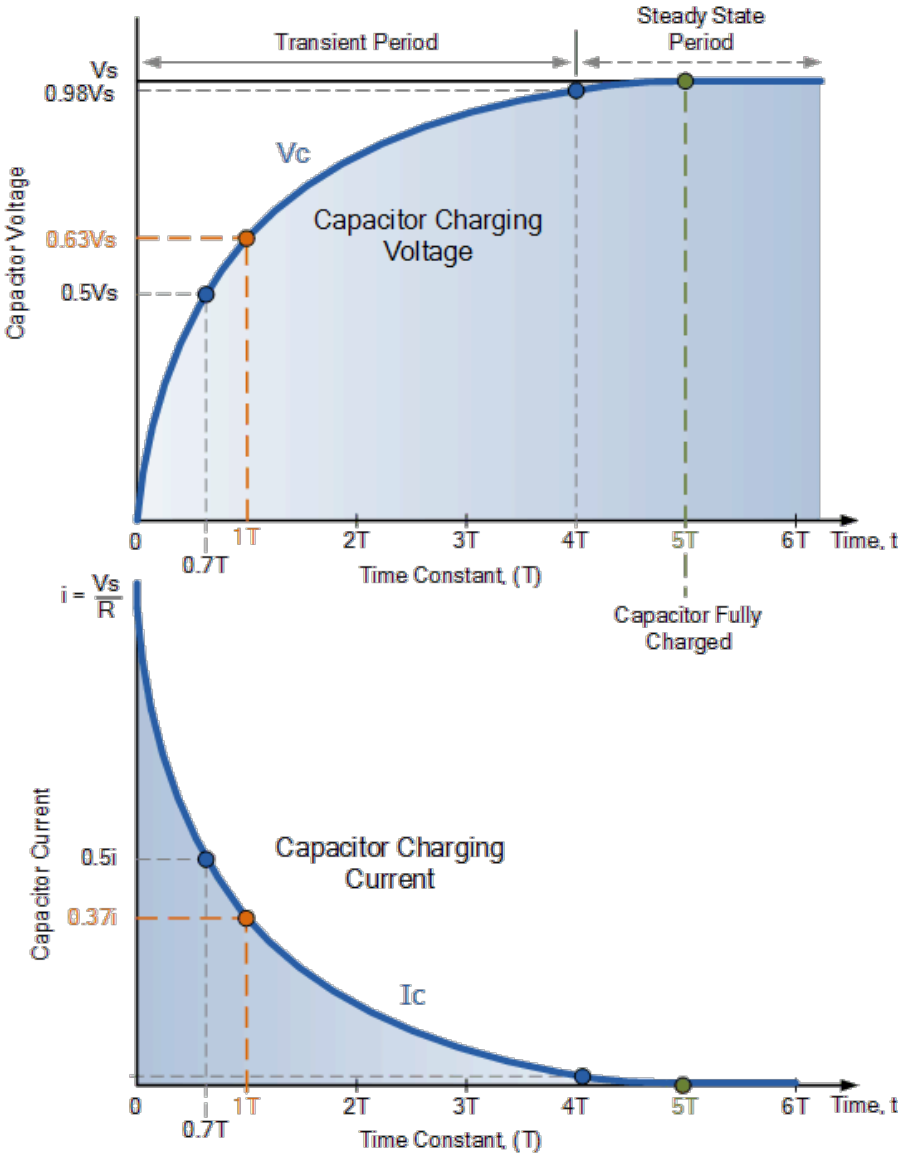
  inversely proportional to voltage



time constant $\tau = R \times C$

- time taken to reach 63% charge

$R \times C \times \ldots$	*condition reached*
$\tau \times 0.7$	50% of final voltage
$\tau \times 1.0$	63% of final voltage
$\tau \times 4.0$	98% of final voltage
$\tau \times 5.0$	‘fully’ charged



capacitor markings

2200 μF
polarised

+ −

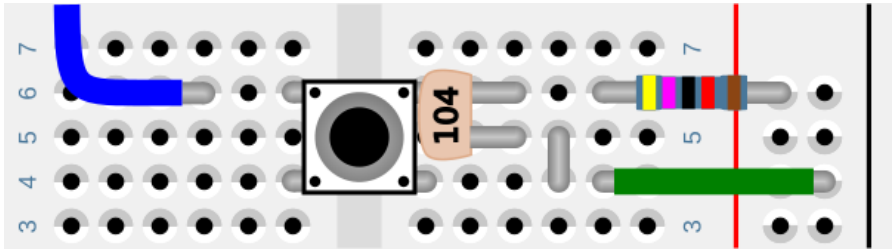
100 nF
(10×10^4 pF)

marking	*value*	
101	100 pF	100 pF
222	2200 pF	2.2 nF
103	10000 pF	10 nF
104	100000 pF	100 nF

hardware debouncing

connect a capacitor across the switch terminals

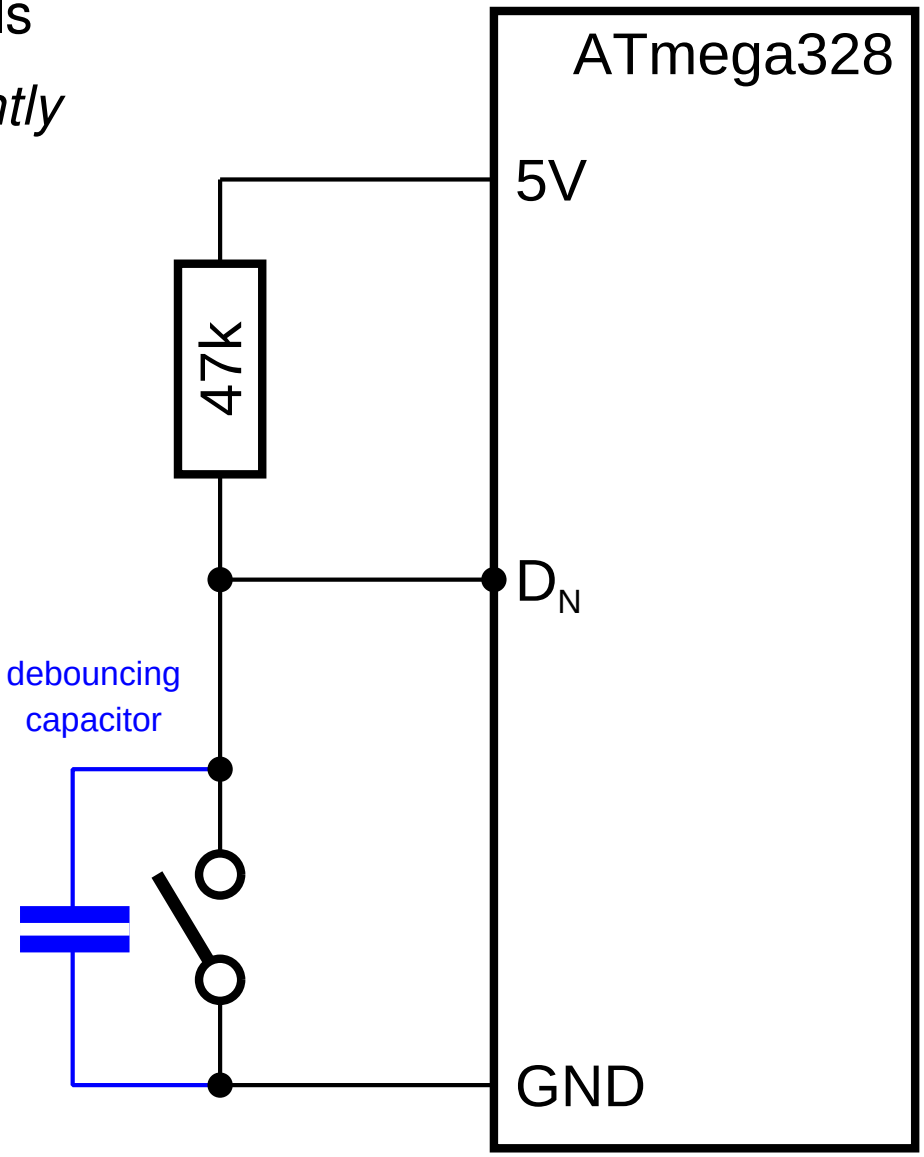
- switch closed: capacitor discharges *instantly* (through the switch)
- switch open: capacitor recharges *slowly* (through the pull-up resistor)



time constant $\tau = 47000 \times C$

- if longer than bouncing duration, then
- input remains `LOW` during bouncing

e.g: 100 nF gives $\tau = 5$ ms



software debouncing

introduce a delay after detecting a press

- wait a fixed amount of time (enough for the switch to 'settle')
  - must deal with worst possible behaviour $\Rightarrow$ delay may be noticeable
  - stops the program doing useful things while 'busy waiting'
  - does not deal with releasing the switch

better idea: simulate a capacitor using a counter

- when the switch is closed, the counter (capacitor) is reset (discharged)
- when the switch is open, the counter (capacitor) is incremented (recharging)
- button press occurs when counter (capacitor) changes from full to empty

using a counter

- can be performed in steps, one step each time `loop()` runs
- deals properly with releasing the switch

software debouncing

eliminate the need for physical capacitor

simple approach: add fixed delay

- must design for worst possible switch behaviour
- delay is long enough (many tens of milliseconds) to be noticeable

better approach: simulate the capacitor

- software timer represents the charge on the capacitor

when switch state is ‘open’ and input is LOW

- counter is reset to zero (simulated capacitor is discharged)
- switch state changes to ‘closed’

when switch state is ‘closed’ and input is HIGH

- counter increments each cycle (simulated capacitor charging)
- when counter reaches a threshold, switch state changes to ‘open’

pull-up resistors

pull-up resistors used very often

- can be provided externally, but
- *every* switch input requires one, so
- microcontroller can provide *internal* pull-up resistors

configure pin as `INPUT_PULLUP`

- enables internal pull-up resistor to 5 V

internal pull-up resistors are ≈20k Ω

- 0.25 mA flows out of the pin when it is pulled `LOW`

concurrency

sketch does only one thing

- read switch inputs, adjust counter, display digit

most of the time the switch is not pressed

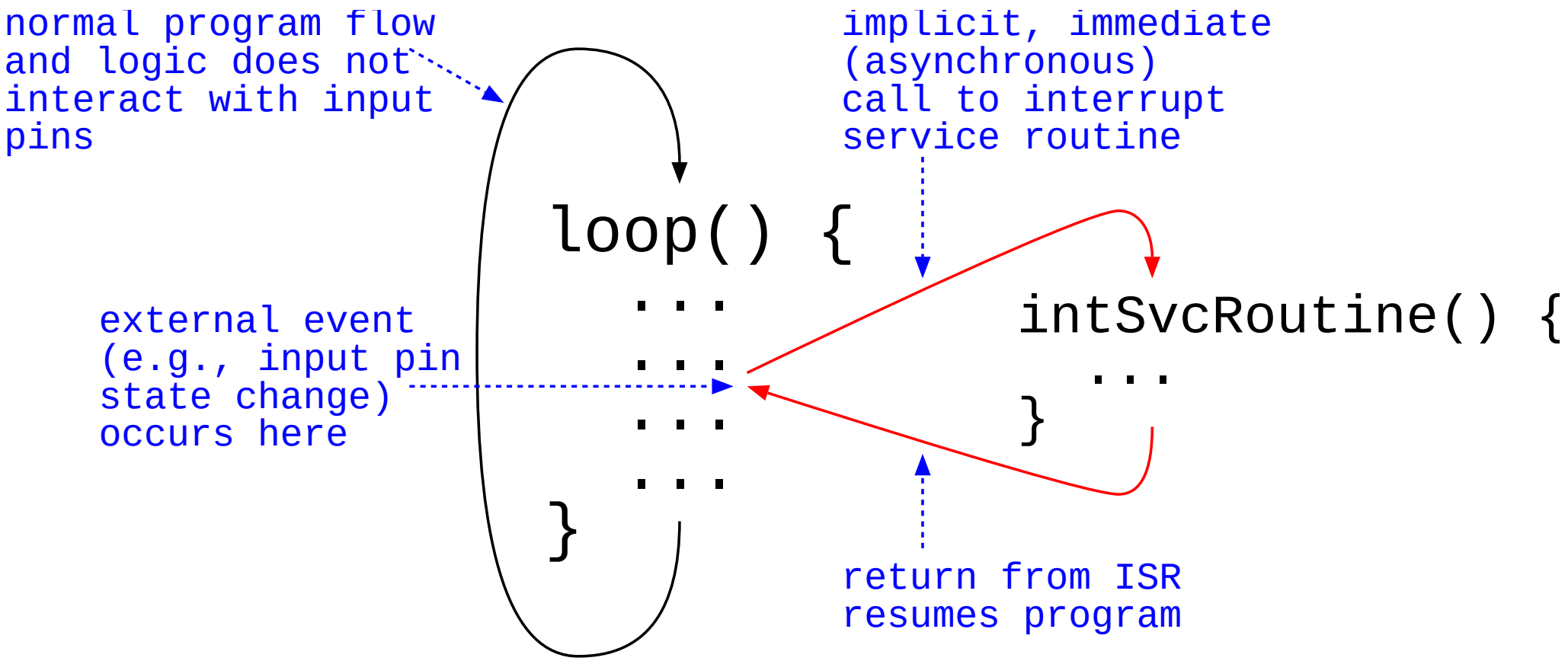
- most of the time, the sketch is doing nothing (very busily)

difficult to integrate other repetitive tasks

better design:

- perform repetitive tasks in `loop()`
- respond to events (e.g., switch presses) by *interrupting* normal tasks
  - interruption is *transparent* to normal tasks

interrupts



some services already provided by regular interrupts

- e.g., `millis()` which returns number of milliseconds elapsed since reset
  - counter incremented by a simple ISR every 1 ms

can also be triggered by input changes on pins 2 and 3

interrupts

two input pin interrupts associated with pin 2 or pin 3

interrupt handler installed in `setup()` using `attachInterrupt()`

- `digitalPinToInterrupt(`*`pin`*`)` tells you the interrupt number

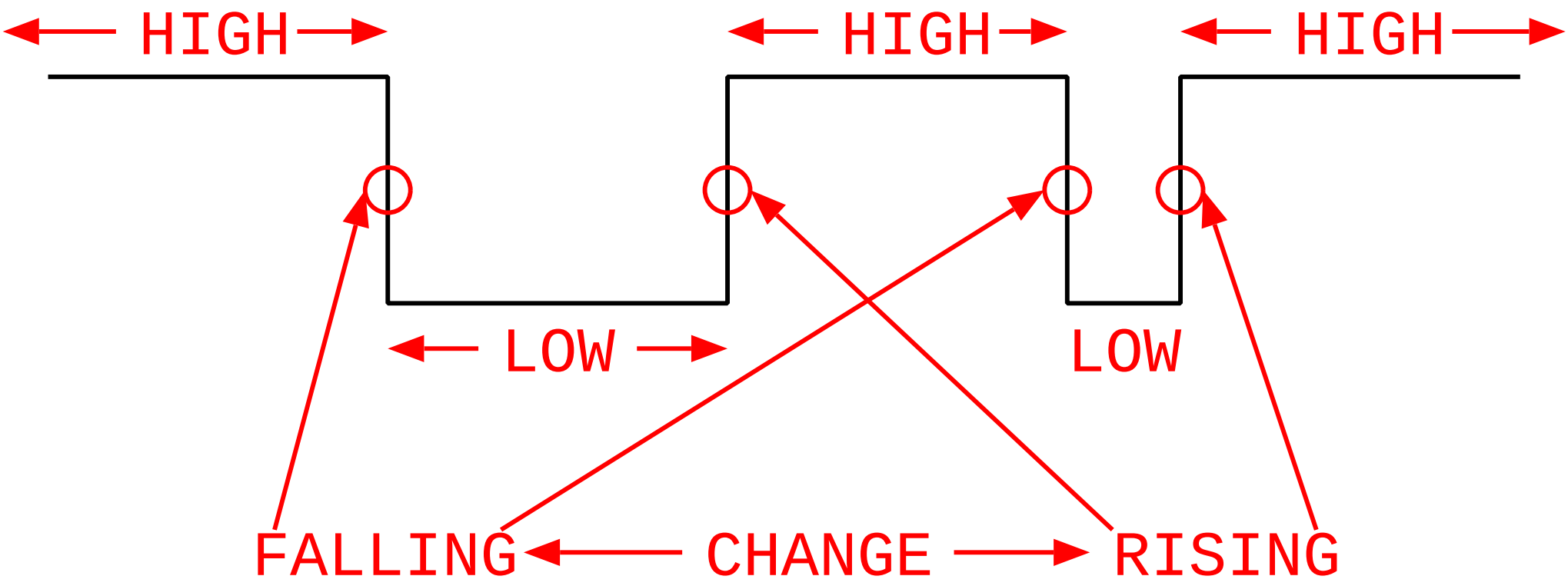
```
void setup() {
  pinMode(2, INPUT);
  attachInterrupt(digitalPinToInterrupt(2), myISR, CHANGE);
}

void myISR() {
  // ... deal with state change on pin 2 ...
}
```

`attachInterrupt()` has three arguments:

- the interrupt number
- the handler function (interrupt service routine)
- the state that triggers an interrupt: `HIGH`, `LOW`, `RISING`, `FALLING`, `CHANGE`

interrupt triggers



trigger name	*interrupts triggered...*
`LOW`	continuously while input is `LOW`
`HIGH`	continuously while input is `HIGH`
`RISING`	whenever input transitions from `LOW` to `HIGH`
`FALLING`	whenever input transitions from `HIGH` to `LOW`
`CHANGE`	whenever input changes state (rising or falling)

interrupt handlers (service routines)

interrupts are *disabled* while a handler is running

- handler must finished quickly
- handler cannot use `delay()`, `millis()`, serial I/O, etc.
  - all of these rely on interrupts of their own to work properly
- handler should not (normally) re-enable interrupts
  - risk of the handler being called repeatedly from within itself

if you need millisecond time in your ISR,

- update a global variable in `loop()` with the current `millis()`

non-linear, unpredictable control flow confuses the compiler

- global variables modified from inside an ISR *must* be declared `volatile`
  - warns the compiler that their value might change asynchronously, at any time

e.g:

```
volatile unsigned long eventCounter = 0;

void myISR() { eventCounter += 1; }
```

exercises

create and use digital inputs

- push buttons

control a counter with them

- displayed on the seven-segment display

debounce the buttons in hardware

- attach a parallel capacitor

debounce the buttons in software

- using a delay
- by simulating a capacitor

manage the buttons in the background using interrupts