

Timing and Analog Input

CSE 1302

What about fractions?

- Positional number systems work on both sides of the decimal point (radix point).

- If radix is r (n integer digits, m fractional digits):

$$\text{val} = a_{n-1} \cdot r^{n-1} + a_{n-2} \cdot r^{n-2} + \dots + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + a_{-m} \cdot r^{-m}$$

- e.g., $wx.yz_{16} = w \cdot 16 + x + y \cdot 16^{-1} + z \cdot 16^{-2}$
or $wx.yz_2 = w \cdot 2 + x + y \cdot 2^{-1} + z \cdot 2^{-2}$

Two kinds of numbers

- Integers – radix point is assumed to be at the far right end of the digits:
 - E.g. 01001110.
- Fixed point – radix point is at a given, fixed location:
 - E.g. 0100.1110
 - 0.1001110 is a common representation on digital signal processors

Q notation

- $Q_n.m$ means a number with $n+m$ bits (digits), n integer and m fractional. Sign bit is often in addition to this.
- E.g., $Q_{3.4}$ for 0100.1100, with value 4.75
- Q_m means a number with $m+1$ bits, m are fractional
- E.g., Q_3 notation would have 4 bits and the following values
 - $wxyz = w.xyz = w \cdot (-1) + x \cdot (1/2) + y \cdot (1/4) + z \cdot (1/8)$
 - range is now -1 to $+7/8$, with resolution $1/8$

Floating point representation

What about the reals? Use scientific notation.

In base 10: $x \cdot 10^y$ $0.32 \times 10^{-3} = 0.00032$

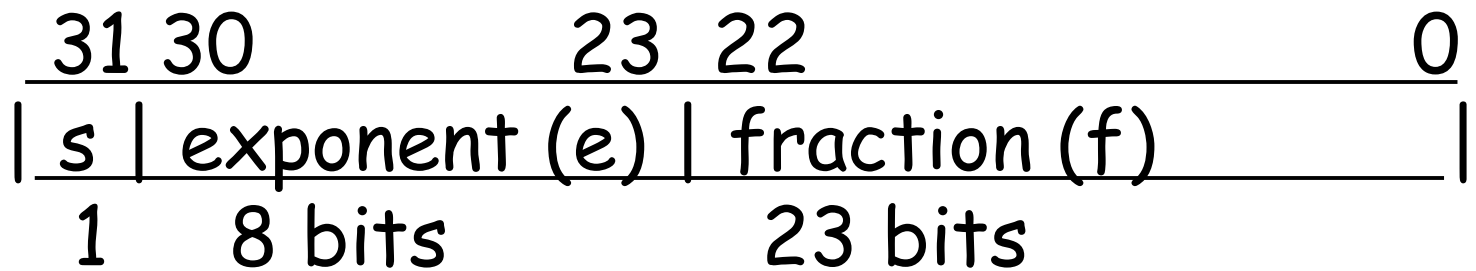
In base 2: $x \cdot 2^y$ called floating point

↑ ↑
| |
| | exponent
| |
| | mantissa

IEEE Floating Point

- Limited range of x and y (fixed # of bits) means we cannot represent every real number exactly
- IEEE std. 754 describes a standard form for floating point number representations
 - Single precision is 32 bits in size
 - Double precision is 64 bits in size

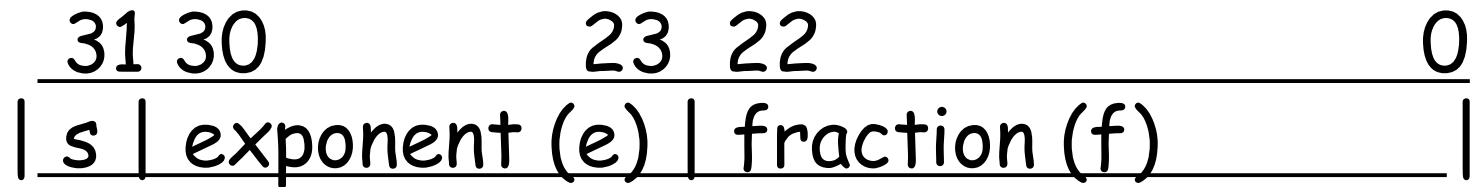
Single precision (32 bits)



$$\text{value} = (-1)^s \times 2^{e-127} \times \underline{1}.f$$

↑ hidden "1"

$$\text{range} = \pm 2 \times 10^{\pm 38}$$



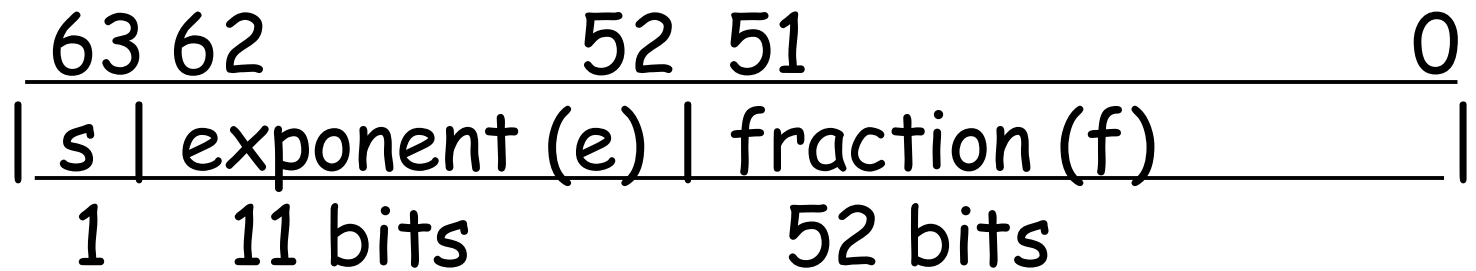
- $s = 0, e = 0, f = 0 \Rightarrow \text{value} = \text{zero}$
- $e = 255, f = 0 \Rightarrow \text{value} = (-1)^s \times \text{infinity}$
- $e = 255, f \neq 0 \Rightarrow \text{value} = \text{"not a number"}$ triggers exception
- $e = 0, f \neq 0 \Rightarrow \text{denormalized}$

$$\text{value} = (-1)^s \times 2^{-126} \times \underline{0}.f$$

$\uparrow$ hidden "0"

- Note use of sign-magnitude for entire number, and excess notation (excess 127) for exponent

Double precision (64 bits)



$$\text{value} = (-1)^s \times 2^{e-1023} \times \underline{1}.f$$

↑ hidden "1"

$$\text{range} = \pm 2 \times 10^{\pm 308}$$

$e = 0, f \neq 0 \Rightarrow$ denormalized

$$\text{value} = (-1)^s \times 2^{-1022} \times \underline{0}.f$$

Simple Timing

- Use `Thread.sleep()` in Java
 - Argument is integer number of milliseconds before the method returns

```
for (int i=0; i < numSeconds; i++) {  
    Thread.sleep(1000);  
    System.out.println(i + " seconds have elapsed");  
}
```

- Use `delay()` on Arduino
 - Same approach as in Java

Effects of Simple Timing

- What are possible issues with this code?

while (true)

wait for 1 second



do some work



output results



end while



Better Timing

- Use a free-running timer
 - `unsigned long millis()`
 - Returns # of milliseconds since reset
 - Rolls over to zero after about 50 days
- Now we can use delta time techniques

```
while (true)
```

```
    if (millis() > loopEndTime) then
```

```
        loopEndTime += deltaTime
```

```
        do some work
```

```
    end if
```

```
end while
```

Impact of Delta Timing

```
while (true)
```

```
  if (millis() > loopEndTime) then
```

```
    loopEndTime += deltaTime
```

```
    do some work
```

```
    output results
```

```
  end if
```

```
end while
```



Mind the Type

- `millis()` returns an unsigned long: 32 bits, 0 ... 4,294,967,295 ms
- Store it in anything smaller and the arithmetic breaks long before 50 days:
 - `int` on the Uno is 16 bits, max 32,767 — your timing dies after 32.7 seconds
 - unsigned `int`, max 65,535 — 65.5 seconds
 - `long` is signed — fine for 24.9 days, then negative
- Every variable that holds a time or an interval gets the same type:
 - `unsigned long loopEndTime;`
 - `const unsigned long GREEN_MS = 3041;`

What if work has delays?

```
while (true)
  if (millis() > loopEndTime) then
    loopEndTime += deltaTime
     do some work
  end if
end while
```

Especially if work takes longer than deltaTime!

Two Ways to Schedule the Next Event

Keep the frequency

```
loopEndTime += deltaTime;
```

- The next deadline is the scheduled one, not "now"
- If the work runs long, the next pass fires immediately and catches up
- Long-run rate stays exactly $1 / \text{deltaTime}$
- Use it when the beat matters: sampling, the stoplight

Keep the gap

```
loopEndTime = now + deltaTime;
```

- The next deadline is measured from "now"
- If the work runs long, everything after it slides later — the schedule drifts
- Guarantees at least deltaTime between events
- Use it when the pause matters and the rate does not

Both are delta timing: the action depends on the difference between two times, and neither one blocks.

Studio 3 shows you both. Pick one per timer, and know which one you picked.

Think Like a Finite-State Machine

```
while (true)
  ➡ if (millis() > loopEndTime) then
      loopEndTime += deltaTime
      do some work
    end if
  end while
```

Do some (but not all) of the work

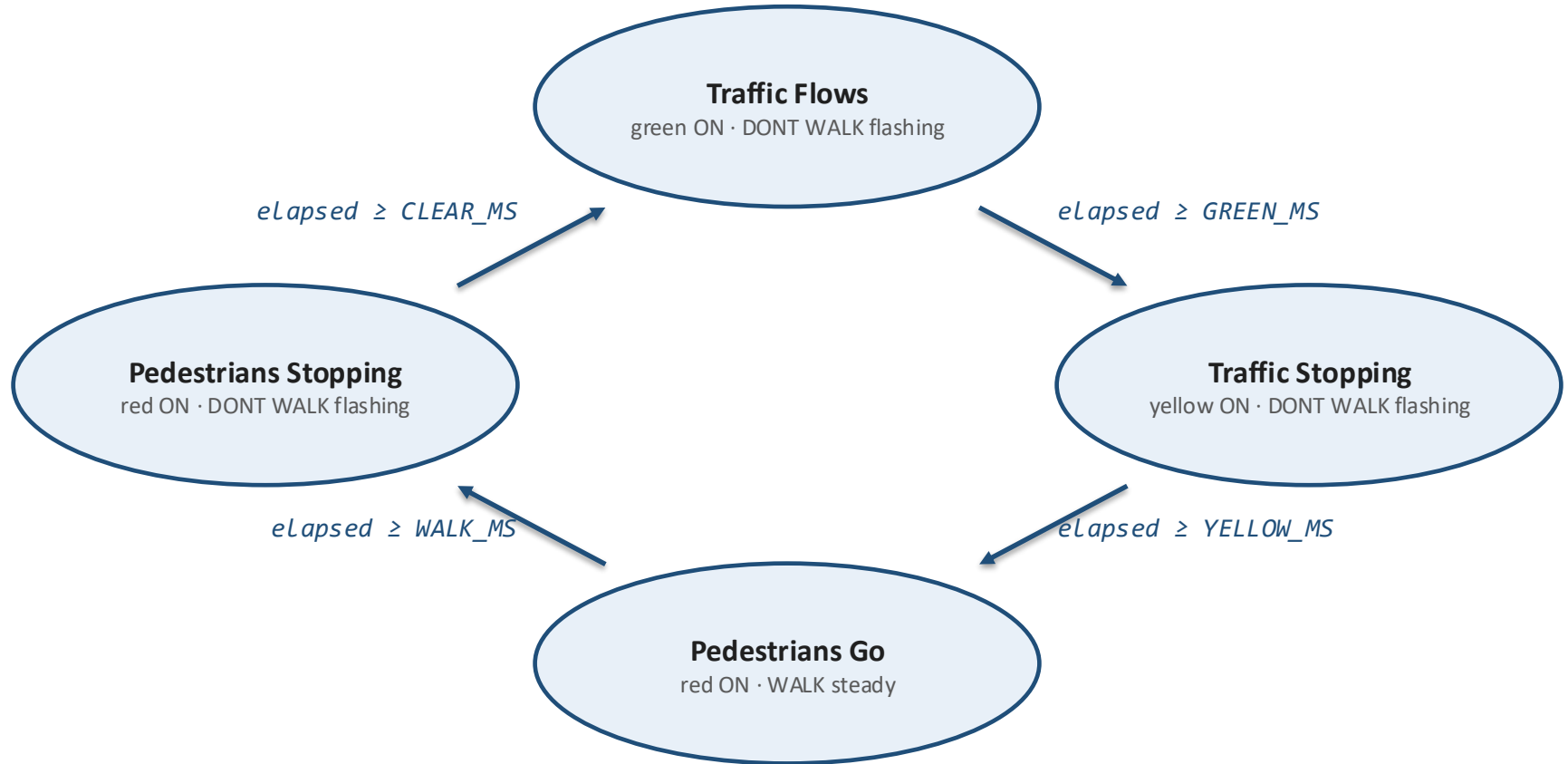
Remember “state” information (in one or more variables)

Inside delta time conditional if, add switch statement

Finite State Machine (FSM)

- Useful concept for today's studio software
- Used extensively in hardware and software systems design and analysis
- Explicitly enumerate (i.e., list) all of the “states” that our design can have, and articulate:
 - What happens (e.g., is output) in each state
 - What state is next under what conditions
- “States” represent what our design wishes to remember

FSM Diagram: The Studio 3 Intersection



Every transition is a time condition — that is exactly what a delta timer gives you.

One-way road, west → east. States hold what the system must remember; the labels in each bubble are its outputs.

One Intersection, Four States

State	Traffic light	Pedestrian light	Leaves when
TRAFFIC_FLOWS	Green	DONT WALK (flashing)	GREEN_MS elapsed
TRAFFIC_STOPPING	Yellow	DONT WALK (flashing)	YELLOW_MS elapsed
PEDESTRIANS_GO	Red	WALK (steady)	WALK_MS elapsed
PEDESTRIANS_STOPPING	Red	DONT WALK (flashing)	CLEAR_MS elapsed

The transition condition is the same shape in every state: has this state's interval elapsed?

Make at least one interval prime — 3041 ms, say. With delay()-based timing you would need the intervals to share a common factor with the flash rate. With delta timing they are simply independent.

Two Timers, One loop()

The state machine and the DONT WALK flasher run at different rates.

Give each its own accumulator, side by side. Never nest one inside the other.

```
const unsigned long FLASH_MS = 500;

unsigned long stateEndTime = 0;    // timer 1: state changes
unsigned long flashEndTime = 0;    // timer 2: DONT WALK blink
bool flashOn = false;

void loop() {
    unsigned long now = millis();

    if (now > stateEndTime) {      // timer 1
        state = nextState(state);
        stateEndTime += stateDuration(state);
        setTrafficLights(state);
    }

    if (now > flashEndTime) {      // timer 2
        flashEndTime += FLASH_MS;
        flashOn = !flashOn;
        updateWalkSignal(state, flashOn);
    }
}
```

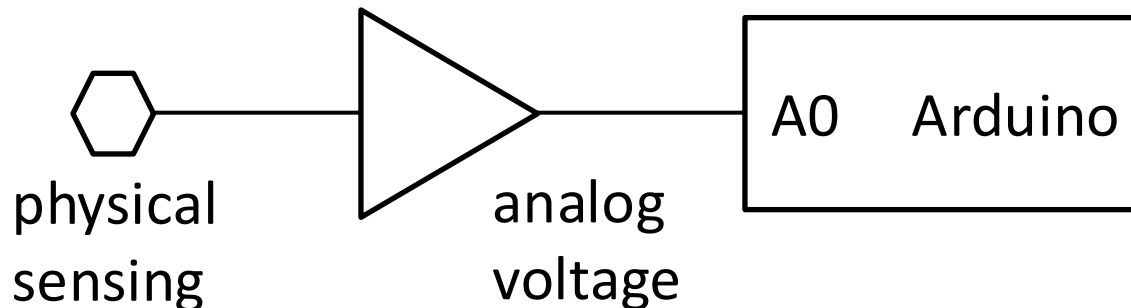
Neither if blocks; both stay on schedule. Assignment 3 has the same shape: sample at 4 Hz, chirp at a rate the temperature sets.

Quiz Time

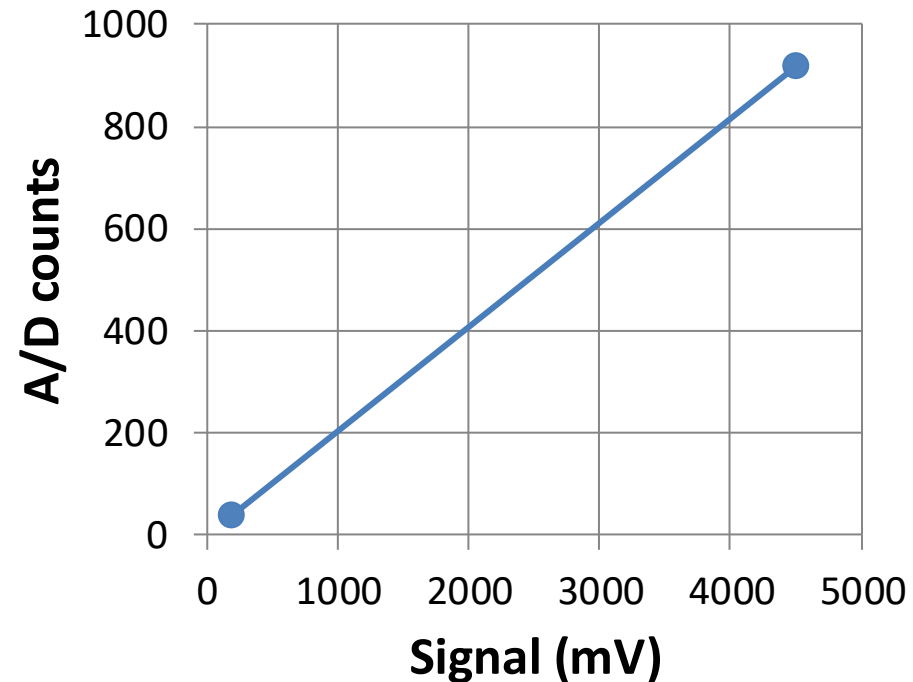
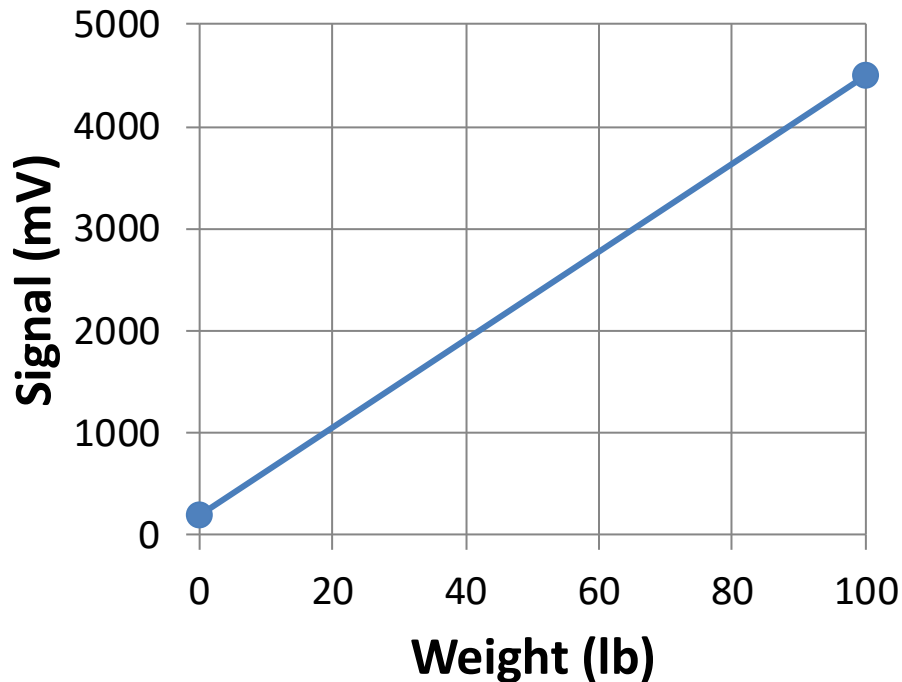
- Go to Canvas and answer the single question for Quiz 3A
- What data type does `millis()` return?
 - unsigned long
 - unsigned int
 - char
 - float

Analog to Digital Conversion

- Convert physical property to voltage signal
- A/D converter on Arduino converts voltage signal to digital representation
 - 10-bit A/D converter has range 0 to $2^{10} - 1$ (0 to 1023) for voltage range 0 to V_{REF}



Understanding Ranges



$$\text{signal} = m \times \text{weight} + b$$

$$\text{signal} = 43 \frac{\text{mV}}{\text{lb}} \times \text{weight} + 200 \text{mV}$$

$$\text{counts} = m \times \text{signal} + b$$

$$\text{counts} = 0.2 \frac{\text{cnt}}{\text{mV}} \times \text{signal} + 0$$

$$\text{counts} = 8.6 \frac{\text{cnt}}{\text{lb}} \times \text{weight} + 40$$

$$\text{weight} = 0.116 \frac{\text{lb}}{\text{cnt}} \times \text{counts} - 4.65$$

Choosing the Analog Reference

- `analogRead()` returns 0 ... 1023 counts spanning 0 V up to VREF
- Default VREF is 5 V: $5000 \text{ mV} / 1024 \text{ counts} \approx 4.9 \text{ mV per count}$
- `analogReference(INTERNAL);` in `setup()` makes $V_{\text{REF}} = 1.1 \text{ V}$
 - $1100 \text{ mV} / 1024 \text{ counts} \approx 1.1 \text{ mV per count}$
 - Same 10 bits, about 4.5× finer resolution — you simply cannot read above 1.1 V
- For the TMP36 in Assignment 3, 1.1 V works

Noisy Analog Signals



- Noise is ever present in analog signals
- For stable signal, quick fix is to average several readings

$$avg = \frac{1}{N} \sum_{1}^N A/D \text{ input}_i$$