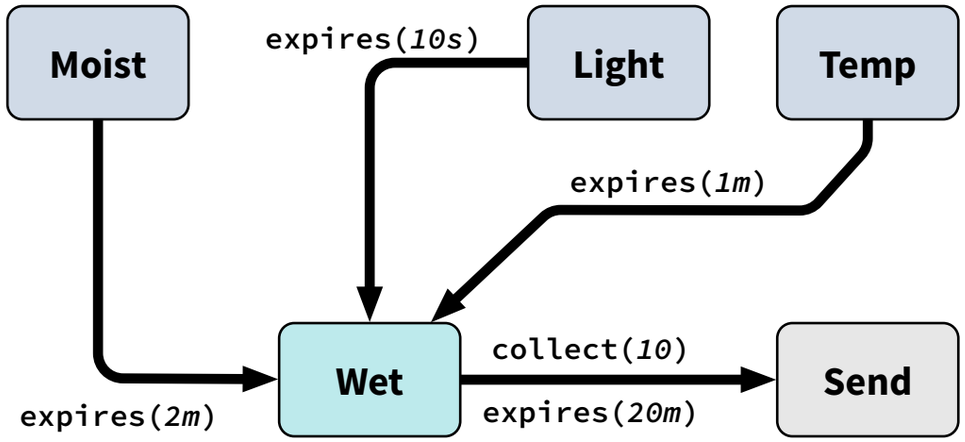
Northwestern
McCORMICK SCHOOL
OF ENGINEERING

# Timely Execution on Intermittently Powered Batteryless Sensors


**Josiah Hester**
Kevin Storer @ *UCI*
Jacob Sorber @ *Clemson*


SenSys'17—November 7, 2017


josiah@northwestern.edu
josiahhester.com


image: Wikimedia Commons, John Seb Barber

# Batteryless Sensing Applications

**Infrastructure**
Pipelines
Bridges
Roads

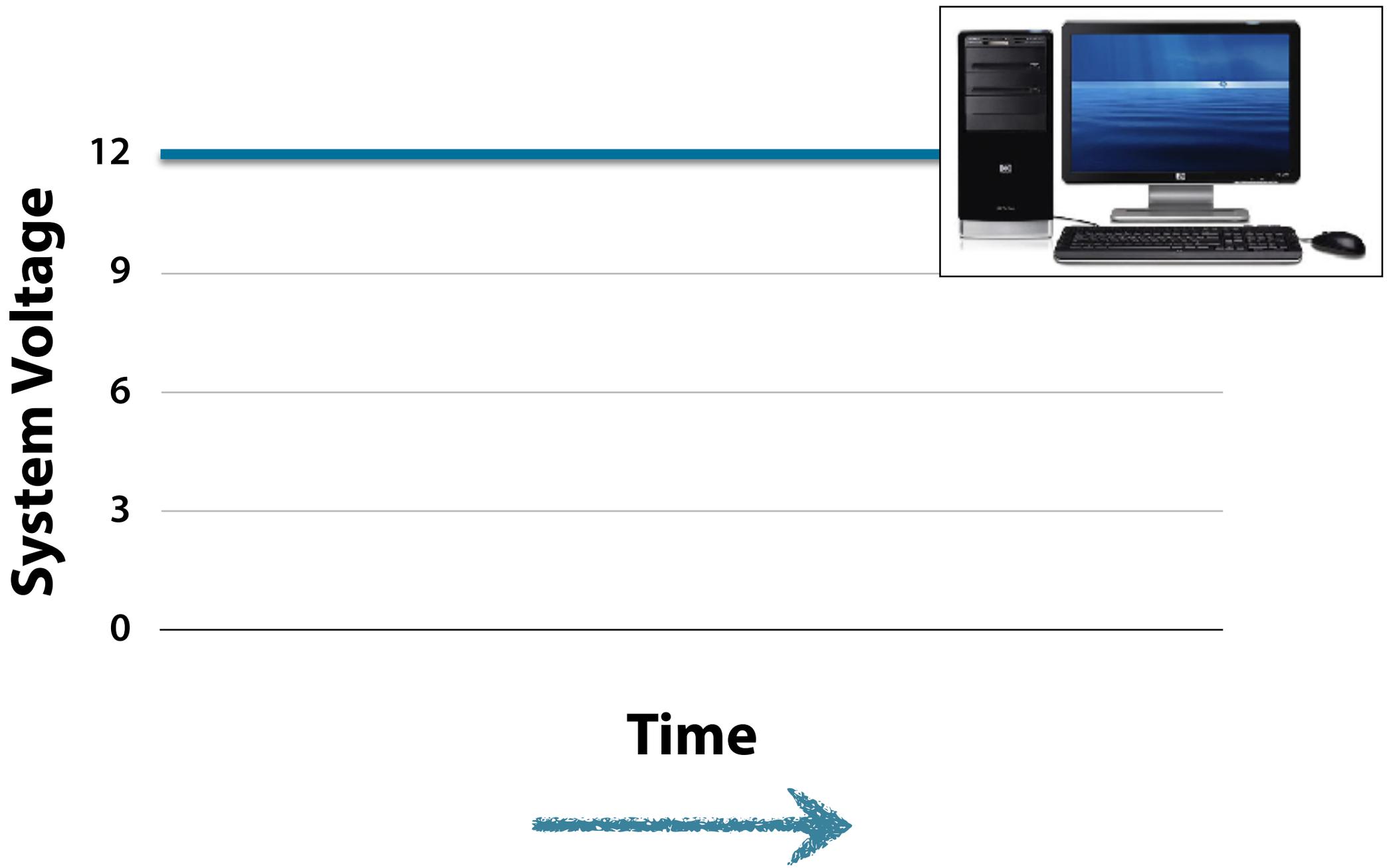
**Wearables**
Clothing
Jewelry
Implants

**Buildings**
Occupancy
Energy Monitor

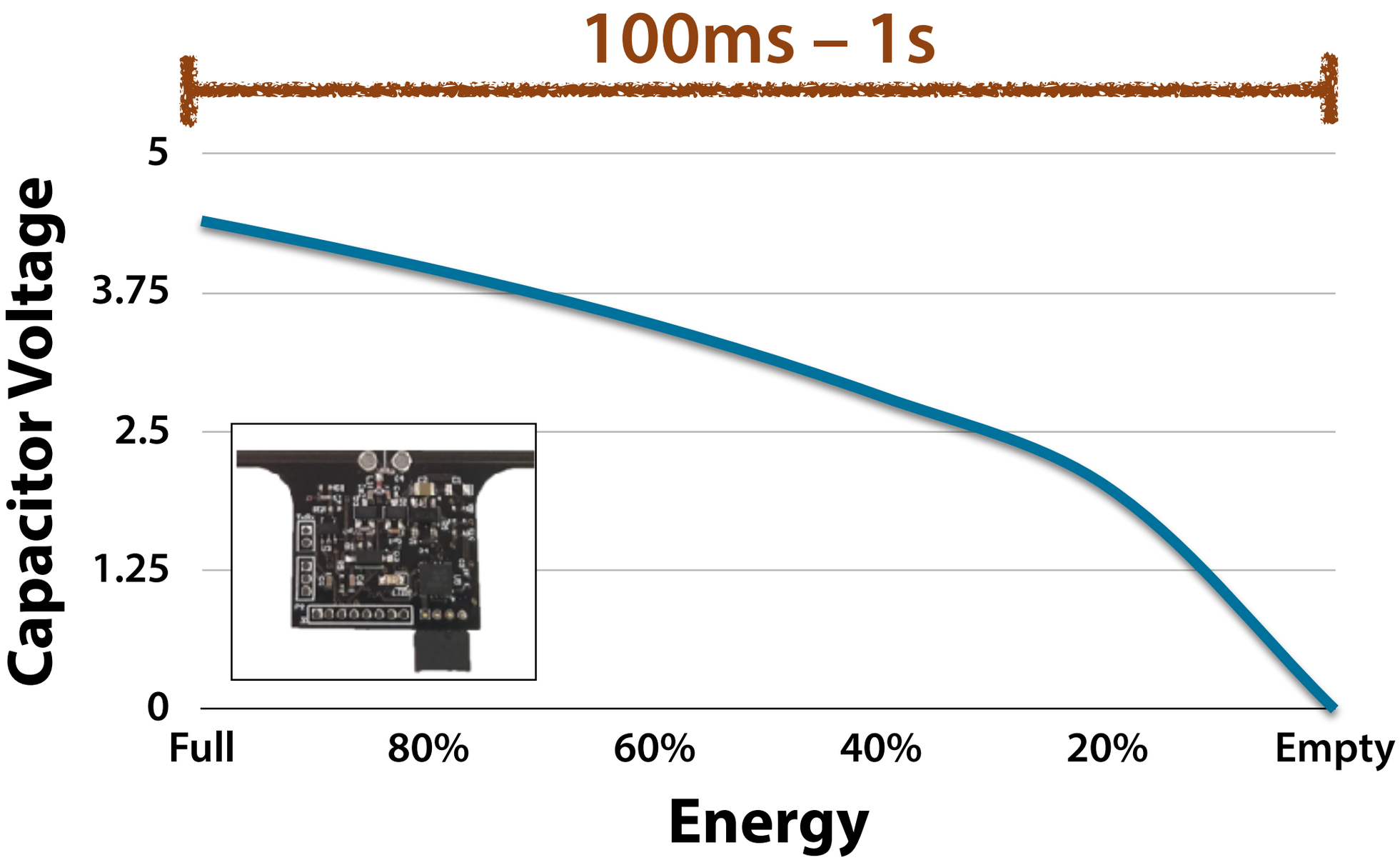
**Wildlife Tracking**
Small animal
Endangered

**Extreme Locales**
Deep Sea, High Altitude, Space

# Traditional Energy View



# RFID-Scale Energy



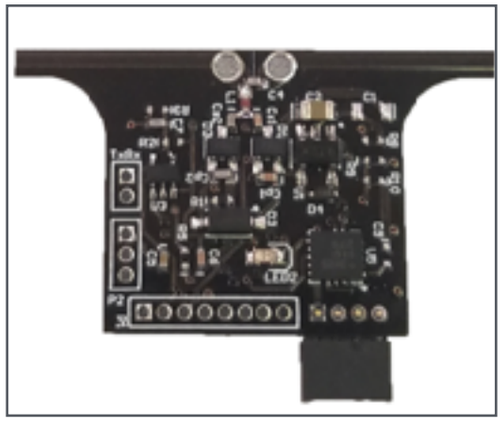
# Intermittent Computing

As small as possible

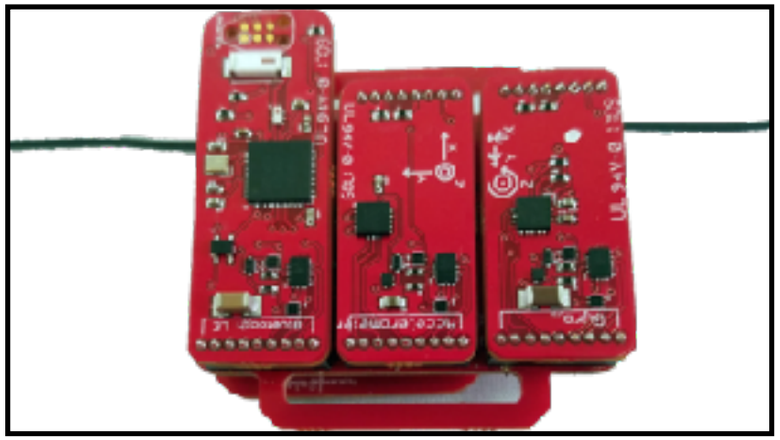
- Minimal energy storage (Cap)
- Harvest energy (RF, Solar)

Run when you can

- Frequent failures
- Erratic supply



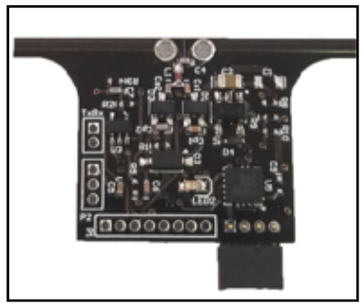
WISP



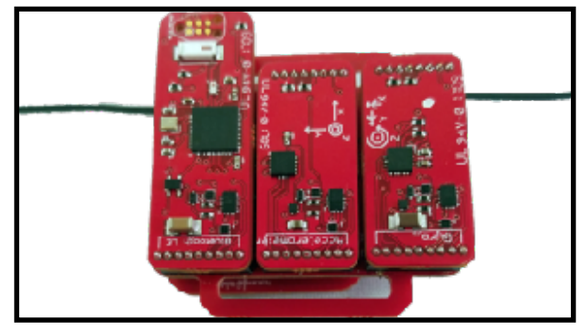
Flicker

# Research Space

## Platform and Hardware

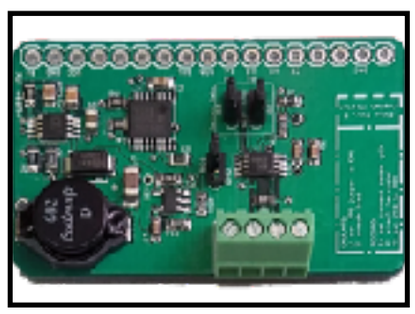


WISP

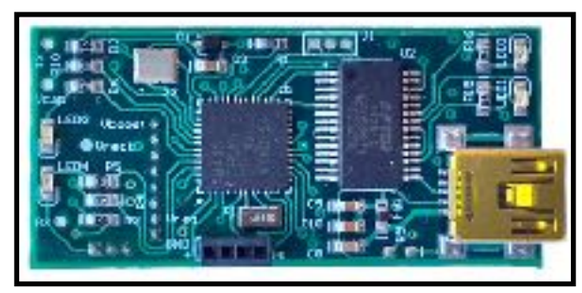


Flicker

## Testing and Debugging



Ekho



EDB

**Language and Runtime**

Dewdrop, Mementos, DINO
Hibernus, Chain, HarvOS…

**+**

**Mayfly**
**(This Work)**

# Intermittent Programming

Power failures make coding tough

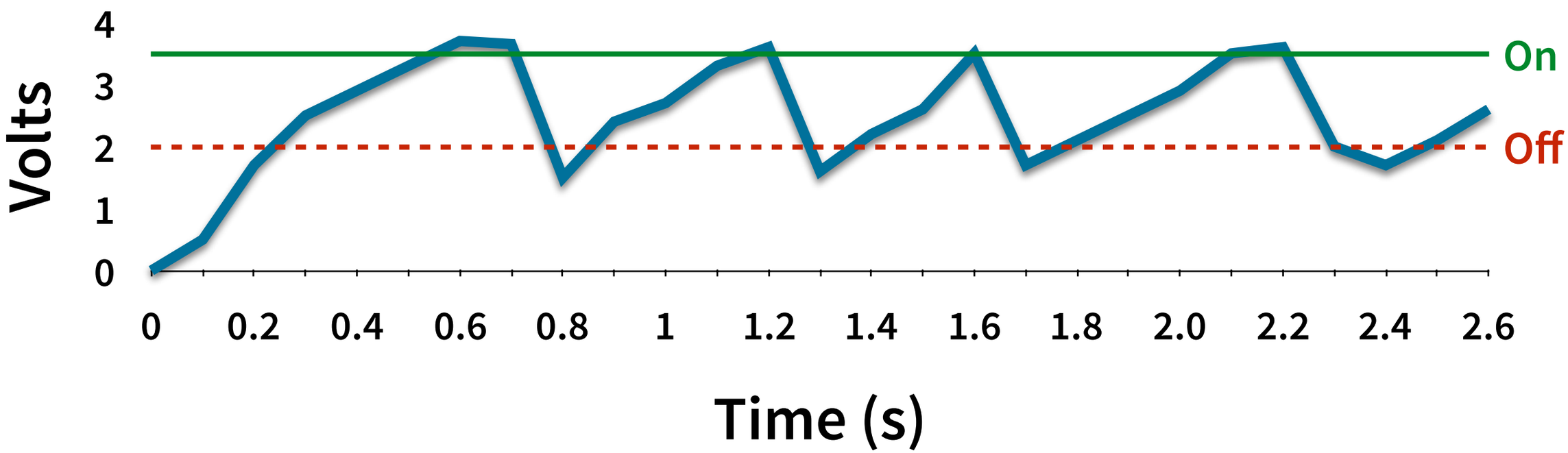
- limited functionality
- no knowledge of time
- no way to control data

We want sophisticated sensing.

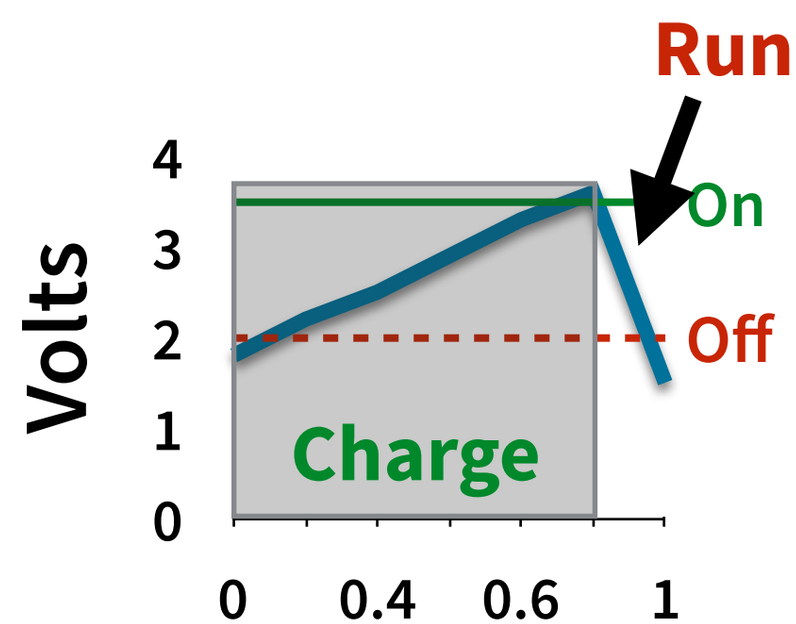
There are challenges…

# Intermittent Execution

## A typical energy harvesting voltage trace…



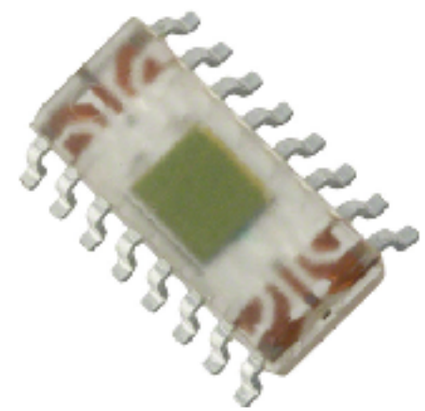

1. Energy harvesting
2. Execution of program
3. Power failure
4. RAM and clocks cleared



# Greenhouse Monitoring

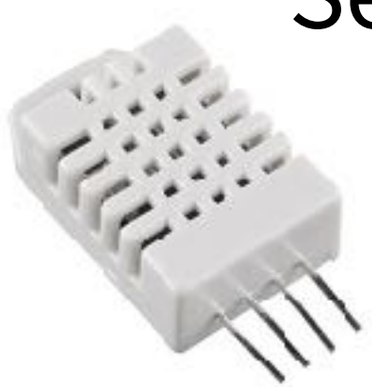
Computation


Solar Energy Harvesting



Energy Storage

Humidity and Wetness Sensors



# Intermittent Execution

```
NV int t,l,w,m

main() {
  while(1)
    t = temp()
    l = light()
    m = moist()
    w = wet(t,l)
    send(w)
    sleep()
}
```

**Source**

# Intermittent Execution

```
NV int t,l,w,m

main() {
  while(1)
    t = temp()
    l = light()
    m = moist()
    w = wet(t,l)
    send(w)
    sleep()
}
```

```
main()
  while(1)
    t = temp()
    l = light()
    m = moist()
    w = wet(t,l)
    send(w)
    sleep()
  while(1)
      ⋮

  <repeat>
```

```
main()
  while(1)
    t = temp()
```

Reboot ⋮

```
main()
  while(1)
    t = temp()
    l = light()
```

Reboot ⋮

```
main()
  while(1)
    t = temp()
```

**Programs may not complete**

# Checkpoints

```
NV int t,l,m,w

main() {
  while(1)
    t = temp()
    l = light()
    m = moist()
    w = wet(t,l)
    send(w)
    sleep()
}
```

**Source**

```
main()
  while(1)
    t = temp()
```

**Reboot**

```
main()
  while(1)
    t = temp()
    l = light()
```

**Reboot**

```
main()
  while(1)
    t = temp()
```

**Intermittent**

```
main()
  while(1)
    t = temp()
```

**Chk**

```
    t = temp()
    l = light()
    m = moist()
```

**Chk**

```
    w = wet(t,l)
```

**Reboot**

```
    w = wet(t,l)
    send(w)
    sleep()
```

**Reboot**

```
    w = wet(t,l)
    send(w)
    sleep()
```

**Checkpoints**

# State of the Art

Existing techniques handle checkpoints and transactions

- checkpointing saves progress
  Mementos, Hibernus, QuickRecall, HarvOS
- memory consistency
  DINO, Ratchet

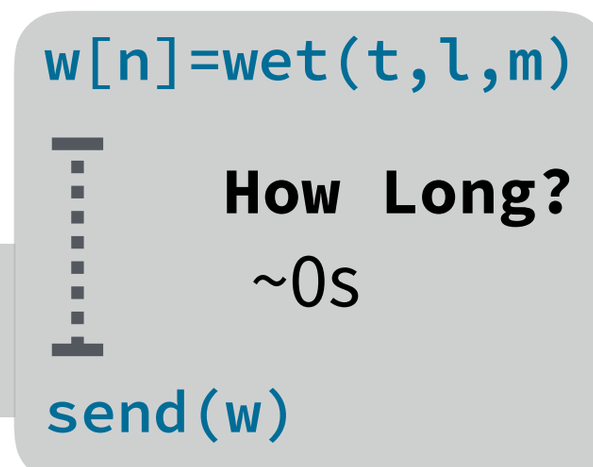
# Limitations

**1.** Current programming models ignore the relationship between time and data.

**2.** The code does not map to actual program behavior.

# Limitations

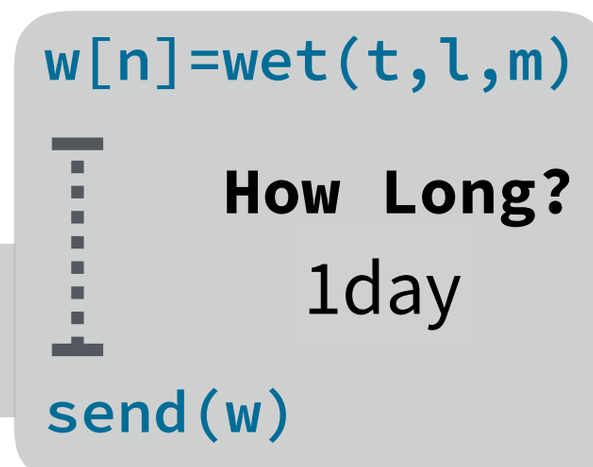
```
1. NV int w, t, l, m
2. main() {
3.   while(1)
4.     t = temp()
5.     l = light()
6.     m = moist()
7.     w=wet(t,l,m)
8.     send(w)
9.     sleep(1)
10. }
```




**Source** **Execution**

# Limitations

```
1. NV int w, t, l, m
2. main() {
3.   while(1)
4.     t = temp()
5.     l = light()
6.     m = moist()
7.     w=wet(t,l,m)
8.     send(w)
9.     sleep(1)
10. }
```




**Source** **Execution**

**How does the runtime manage time?**

# Limitations

**1.** Current programming models ignore the relationship between time and data.

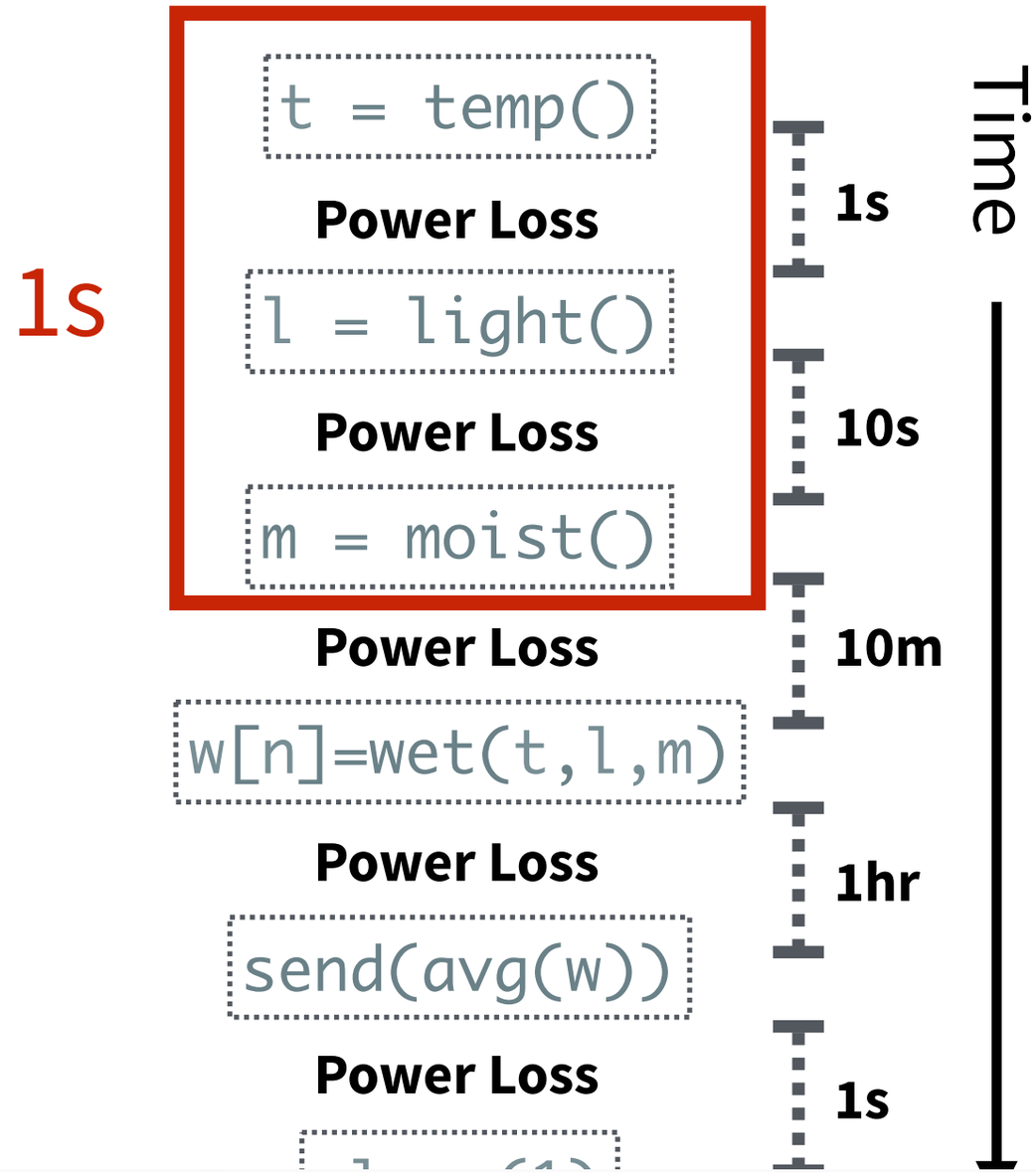
**2.** The code does not map to actual program behavior.

# Limitations

```
1. NV int w, t, l, m
2. main() {
3.   while(1)
4.     t = temp()
5.     l = light()
6.     m = moist()
7.     w=wet(t,l,m)
8.     send(w)
9.     sleep(1)
10. }
```

1s

**Source**




**Execution does not match intent**

# Limitations

```
1. NV int w, t, l, m
2. main() {
3.   while(1)
4.     t = temp()
5.     l = light()
6.     m = moist()
7.     w=wet(t,l,m)
8.     send(w)
9.     sleep(1)
10. }
```

**Source**

```
1. NV int t, l, m, w
2. NV STATE curr_S, next_S
3. const int[] expires = {1,10,20,5}
4. main() {
5.   while(1) {
6.       task_s()
7.       switch(next_S) {
8.             case T:
9.                   t = temp()
10.                  tt = timestamp()
11.                  next_S = L
12.                  break
13.            case L:
14.                  l = light()
15.                  tl = timestamp()
16.                  next_S = M
17.                  break
18.            case M:
19.                  if(tt < expires[0] &
20.                        tl < expires[1])
21.                        m = moist()
22.                        tm = timestamp()
23.                        next_S = W
24.                  else next_S = T
25.                  break
26.            case W:
27.                  if(m < expires[2])
28.                        w = wet(t,l,m)
29.                        wm = timestamp()
30.                        next_S = W
31.                  else next_S = M
32.                  break
33.            case S:
```

**Intermittence forces complexity**

```
42.      }
43.      task_e()
44.  }
45.}
```

# Mayfly

**1.** Coordination language–built on C

**2.** Relates time and data.

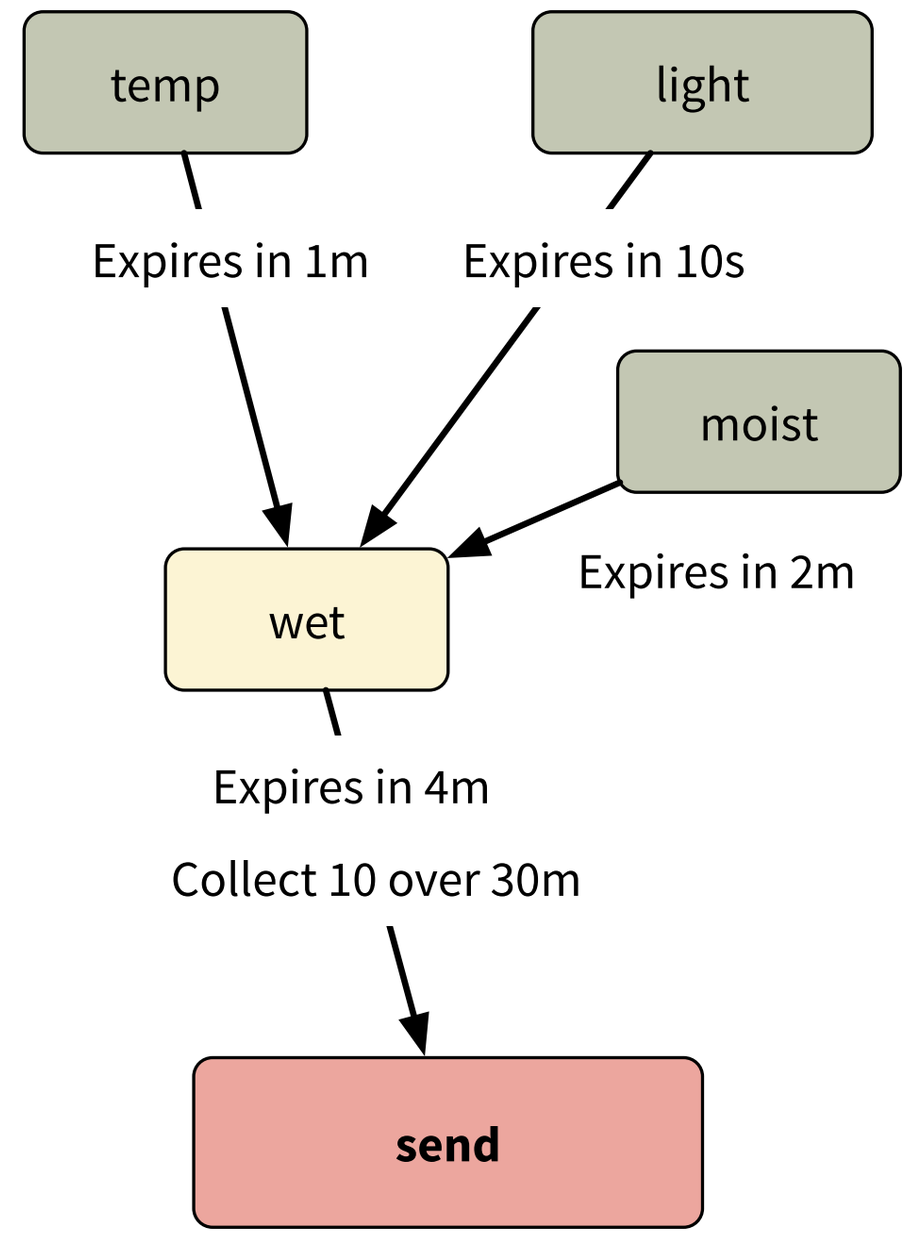
**3.** Code maps to program behavior.

# Syntax

```
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
```

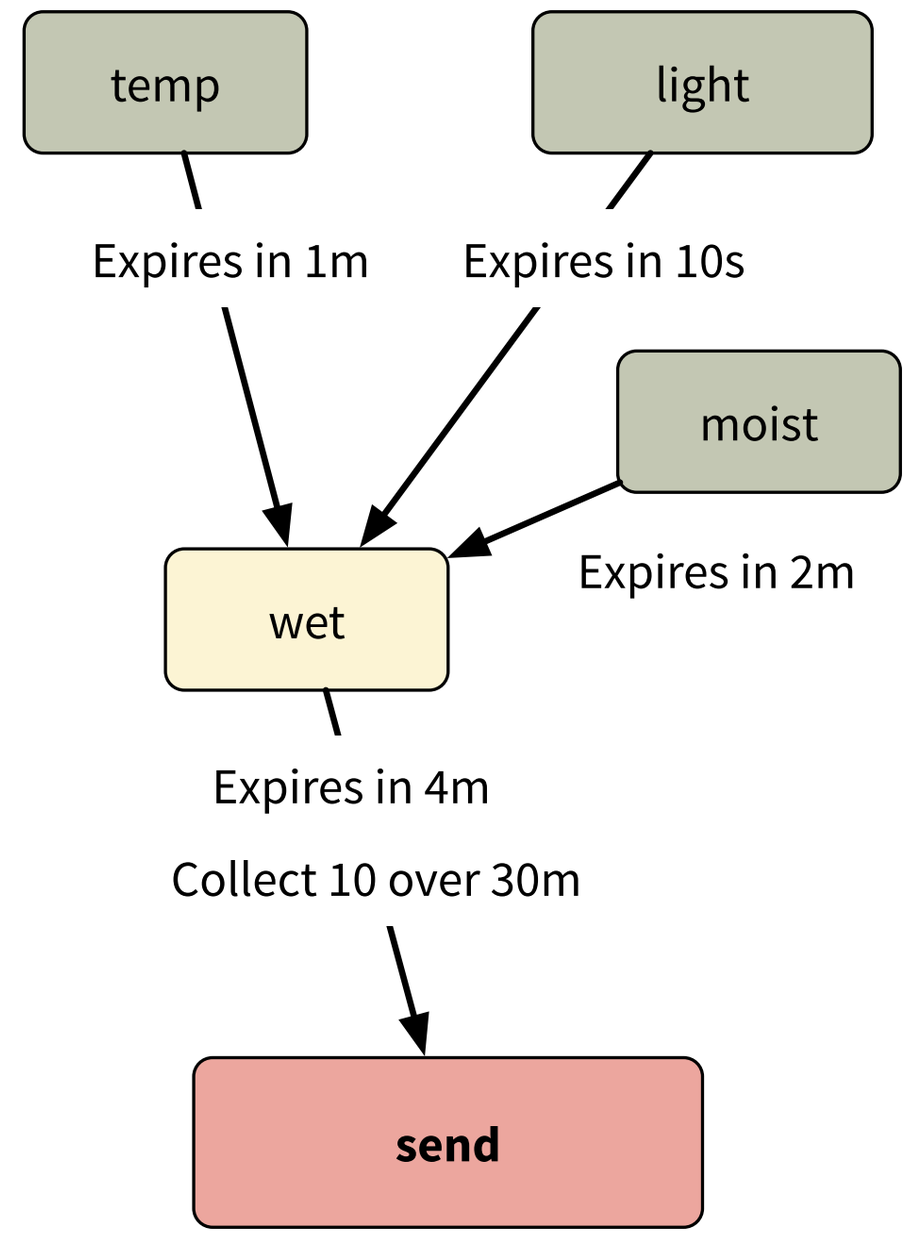


# Syntax

```
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
```

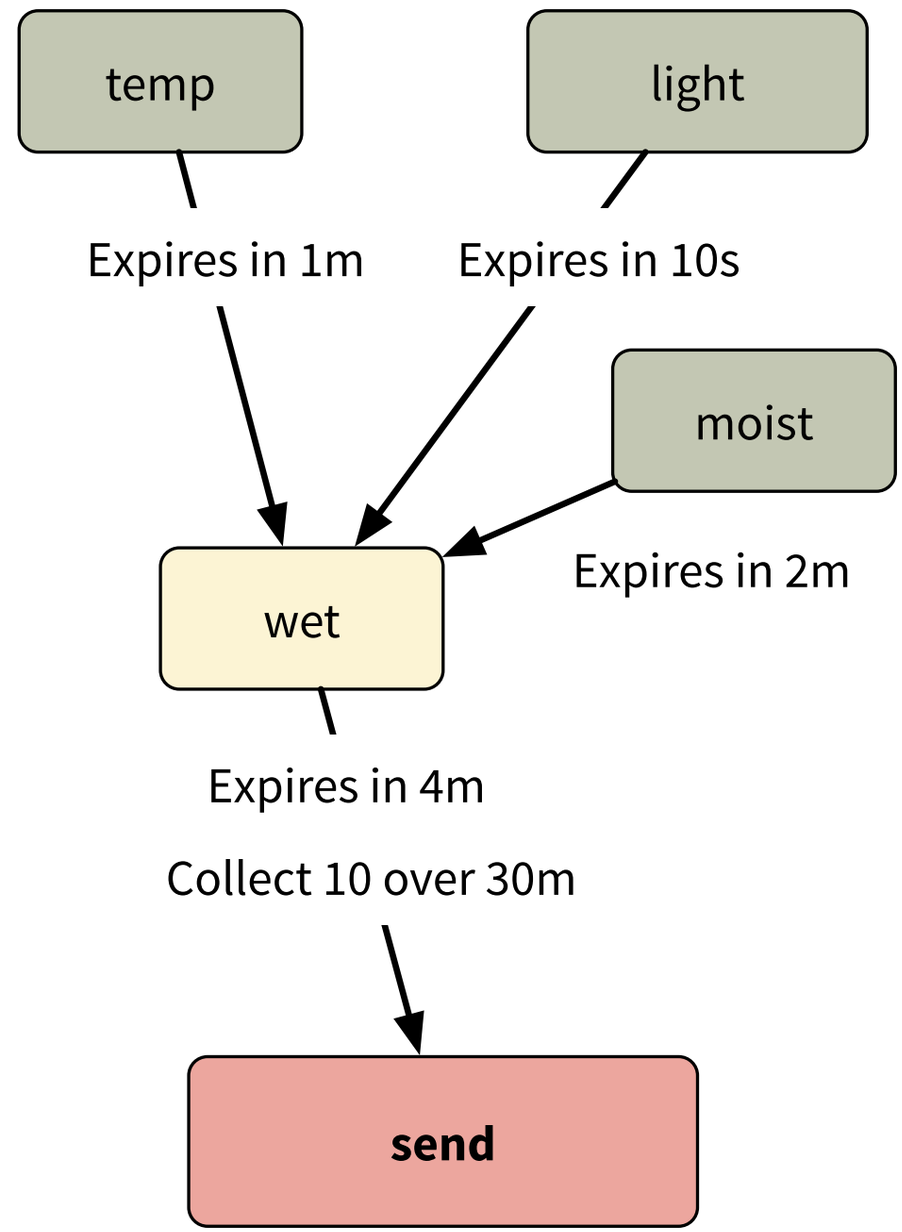


# Syntax

```
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
```

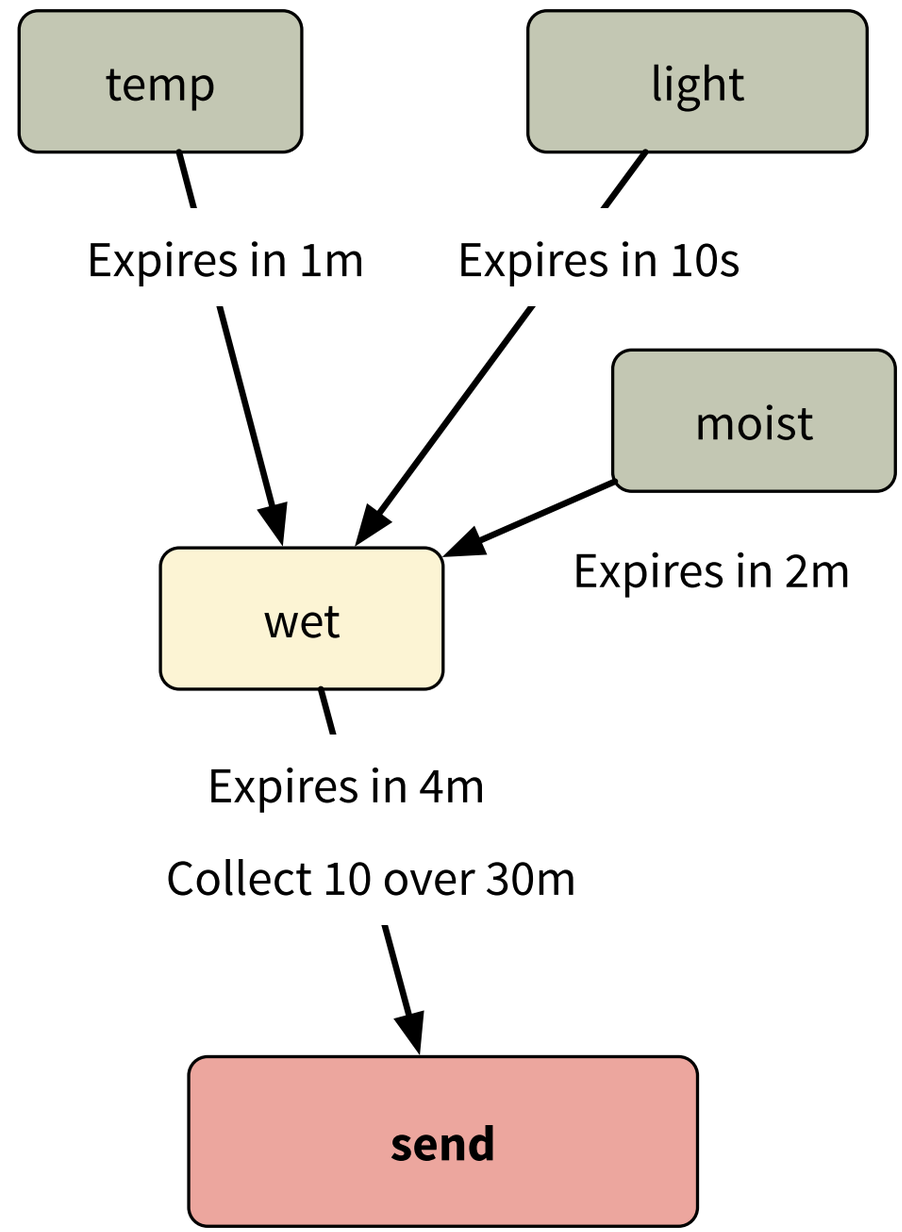


# Syntax

```
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
```



# Syntax

```
expires(time)
```

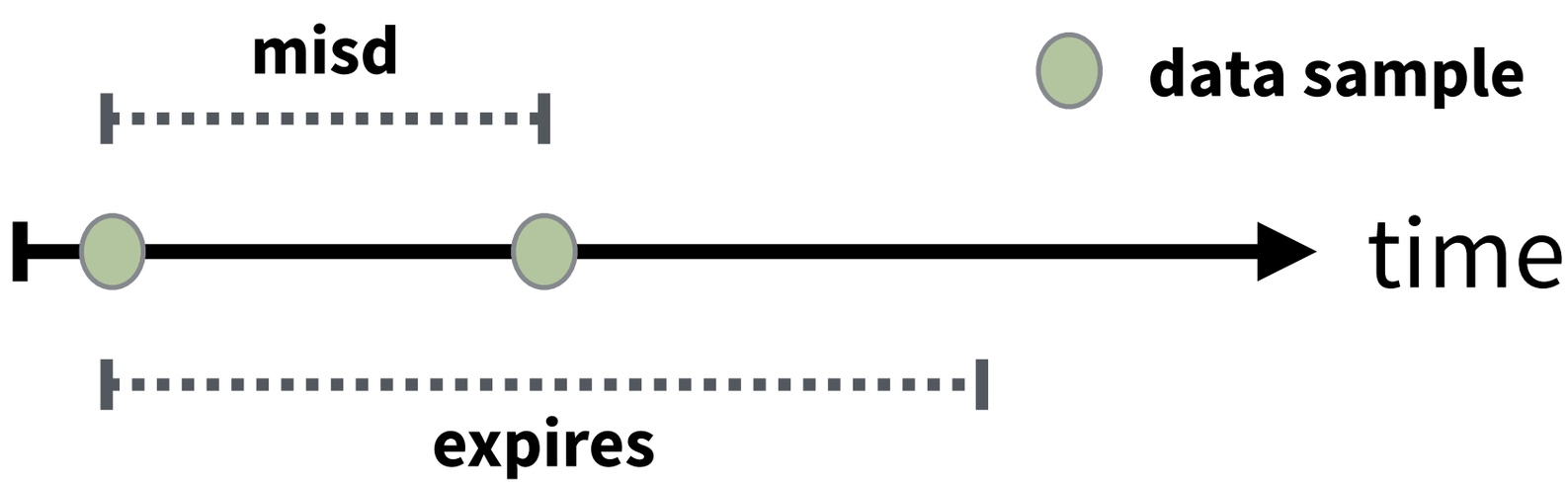
**Definition.** The amount of time data can sit on an edge.

# Syntax

`misd(time)`

*minimum-inter-sample-delay*

**Definition.** The amount of time before more data is useful.



# Syntax

```
collect(quantity)
```

**Definition.** Gather a set of data coming out of the task.

.

# Syntax

```
// Optional global policies
{: scheduling_policy(FINISH_FLOW); :}

// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int light, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge constraints
light -> wet {expires(10s)}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10)}
```

# Scheduling

**1.** Statically defined task schedule

**2.** Near zero energy overhead

**3.** Handles transactions, timekeeping

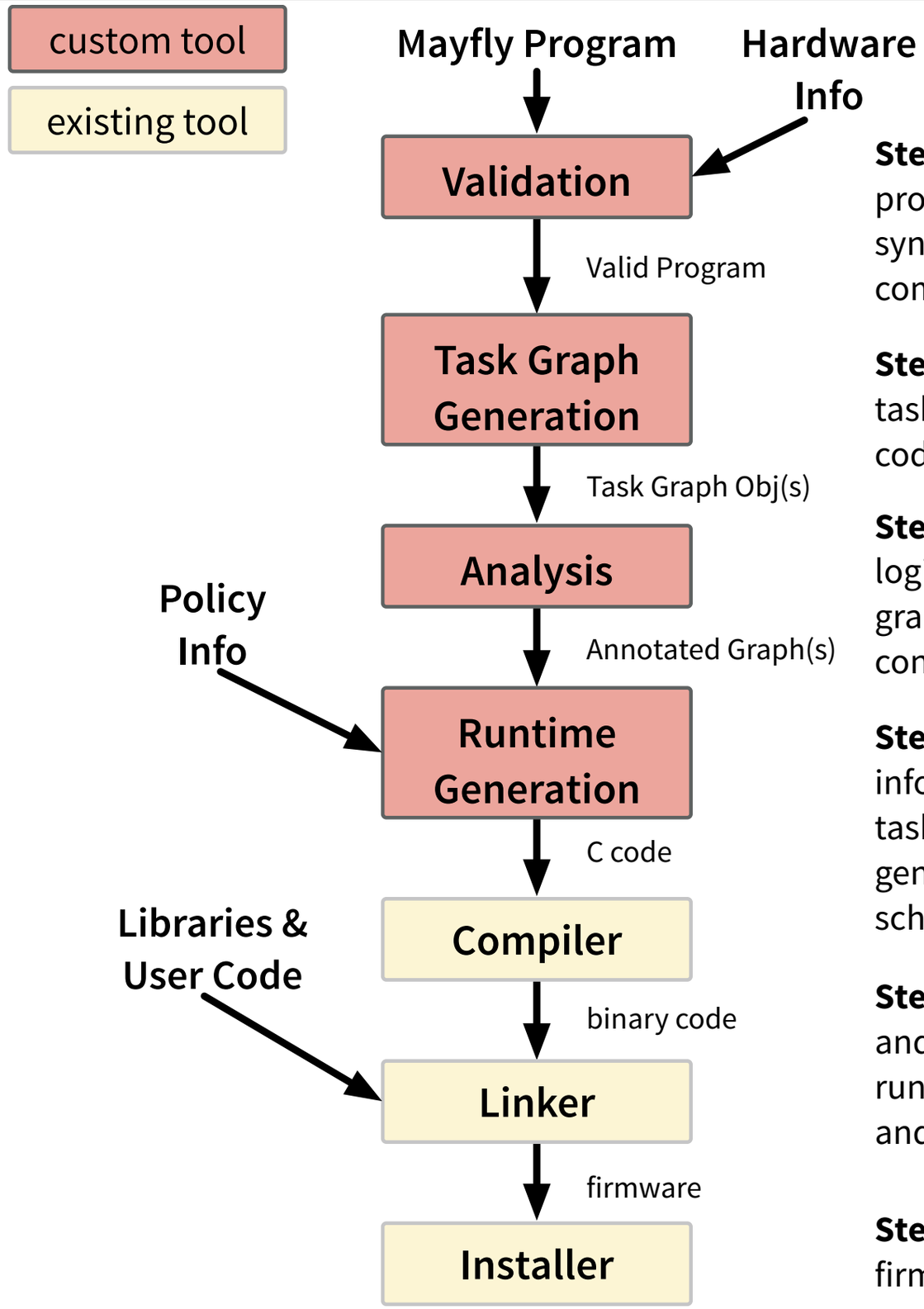
.

# Scheduling Loop

```
// Pre: Sort tasks by priority
while(1)
{
  // What's the next thing to do?
  t = next_task();
  // Maintain ordered stack
  if(constraints_satisfied())
   execute(t);
```

**Approximation is the intent**

# Compilation




**Step 1:** Parse program and verify syntax. Check for common errors.

**Step 2:** Generate task graphs from code.

**Step 3:** Check for logic errors. Annotate graph(s) with constraints.

**Step 4:** Take policy information and task graphs to generate sensing schedule.

**Step 5:** Compile and link with runtime libraries and user code.

**Step 6:** Install firmware on device.

# Implementation

**1.** Manage timekeeping

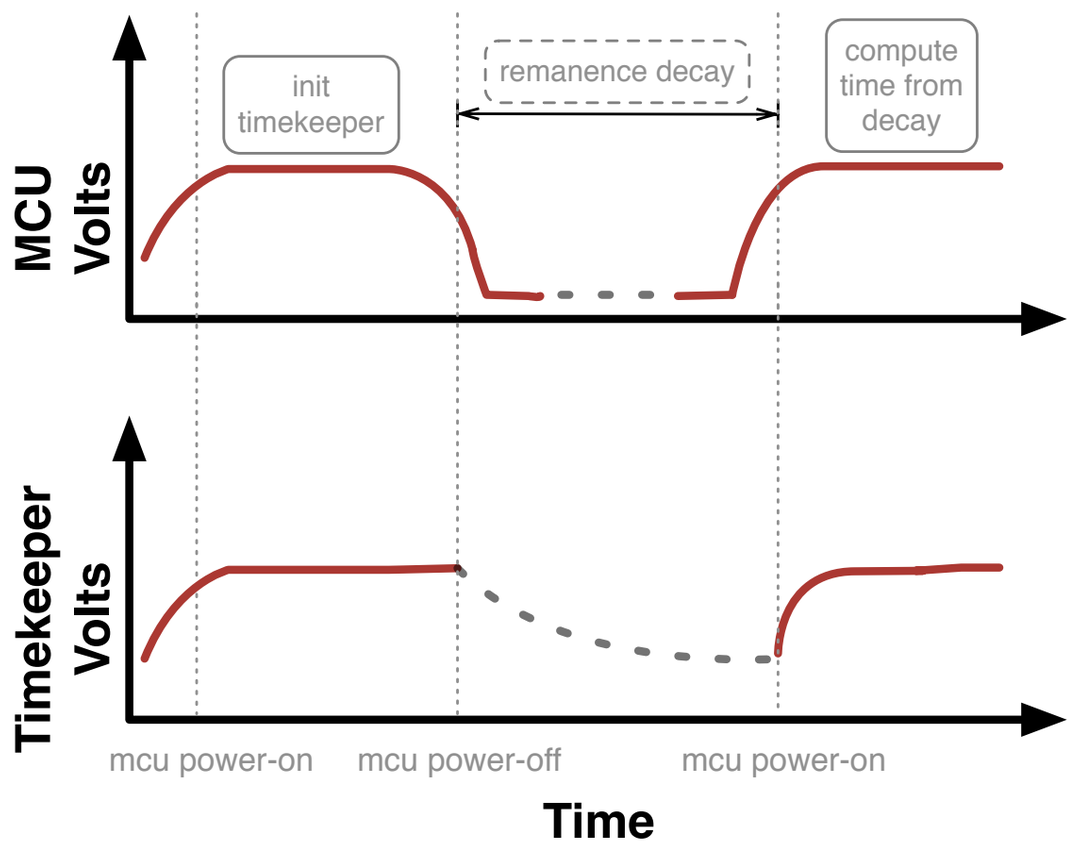
**2.** Manage execution state

**3.** Manage data on edges

# Timekeeping

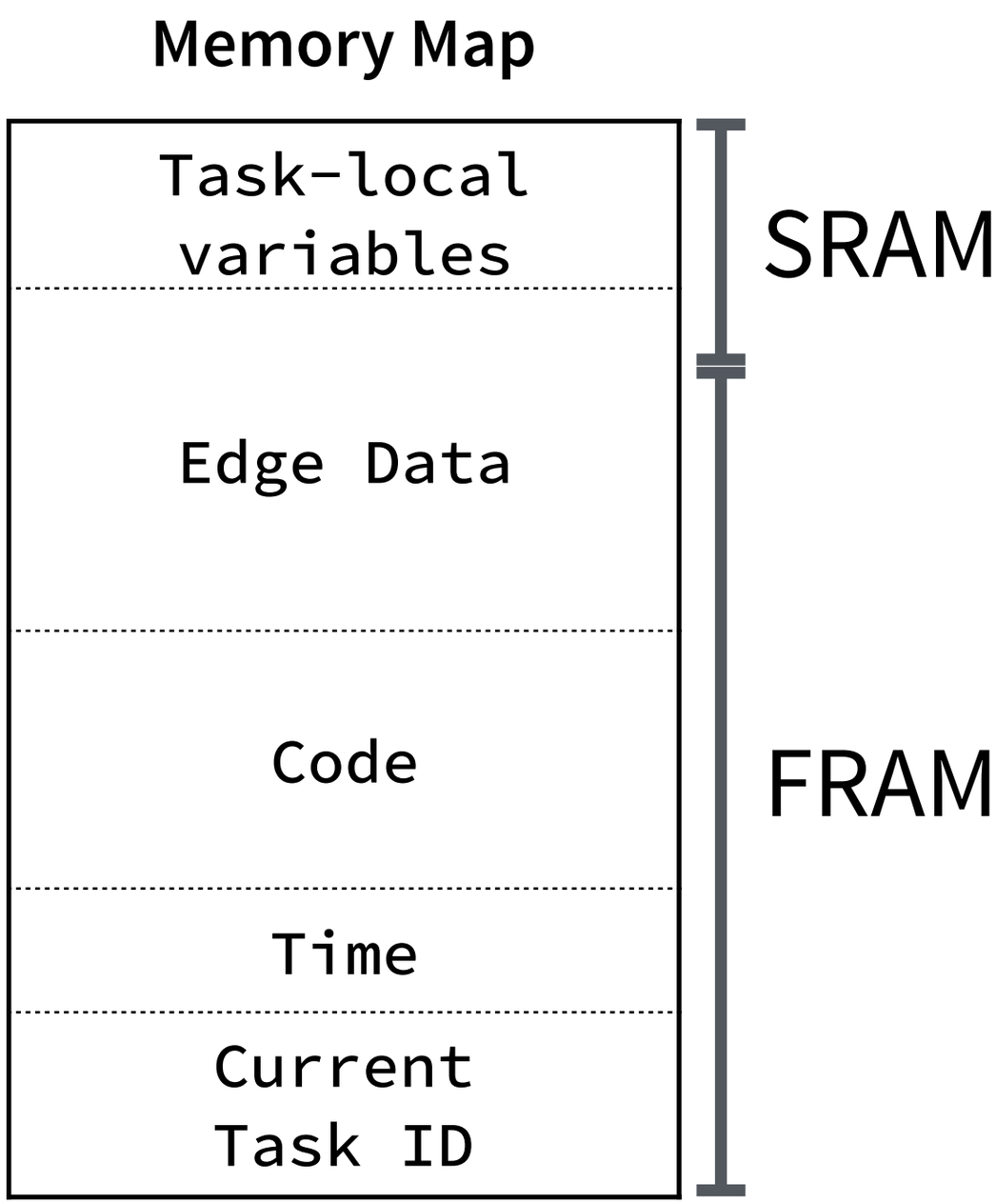
Capacitor or SRAM as Hourglass

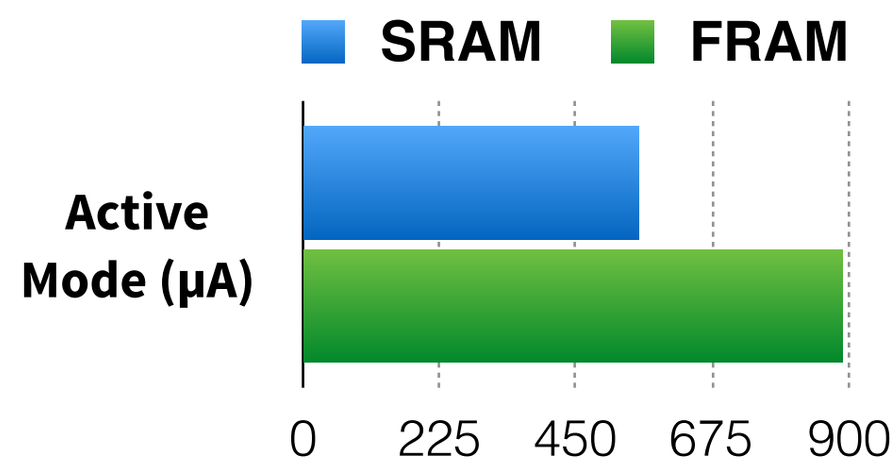
- timekeeping across power failures
- fine grained and reliable



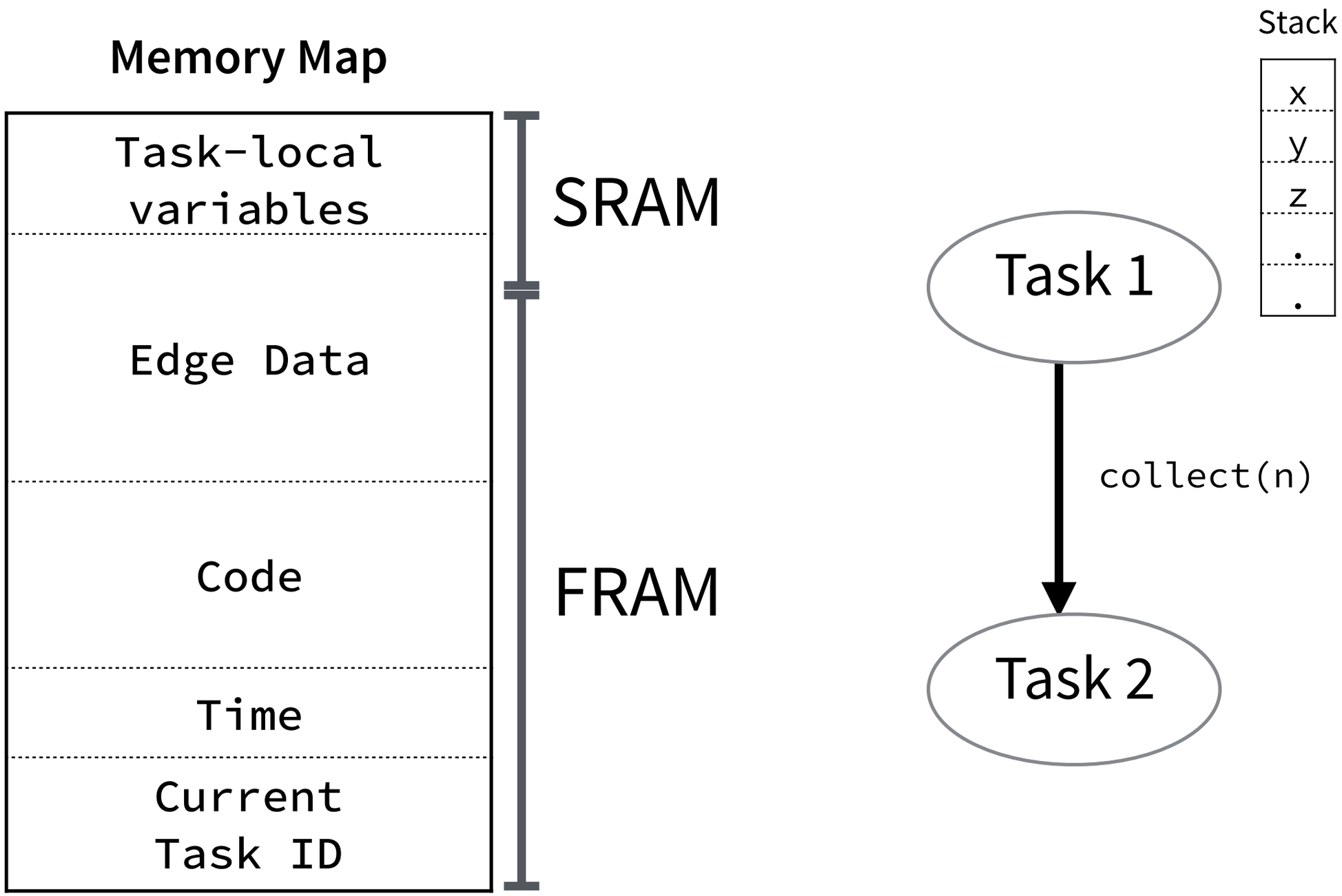

Persistent Clocks for Batteryless Sensing Devices, *Josiah Hester et al. ACM TECS, 2016*

# Memory Model

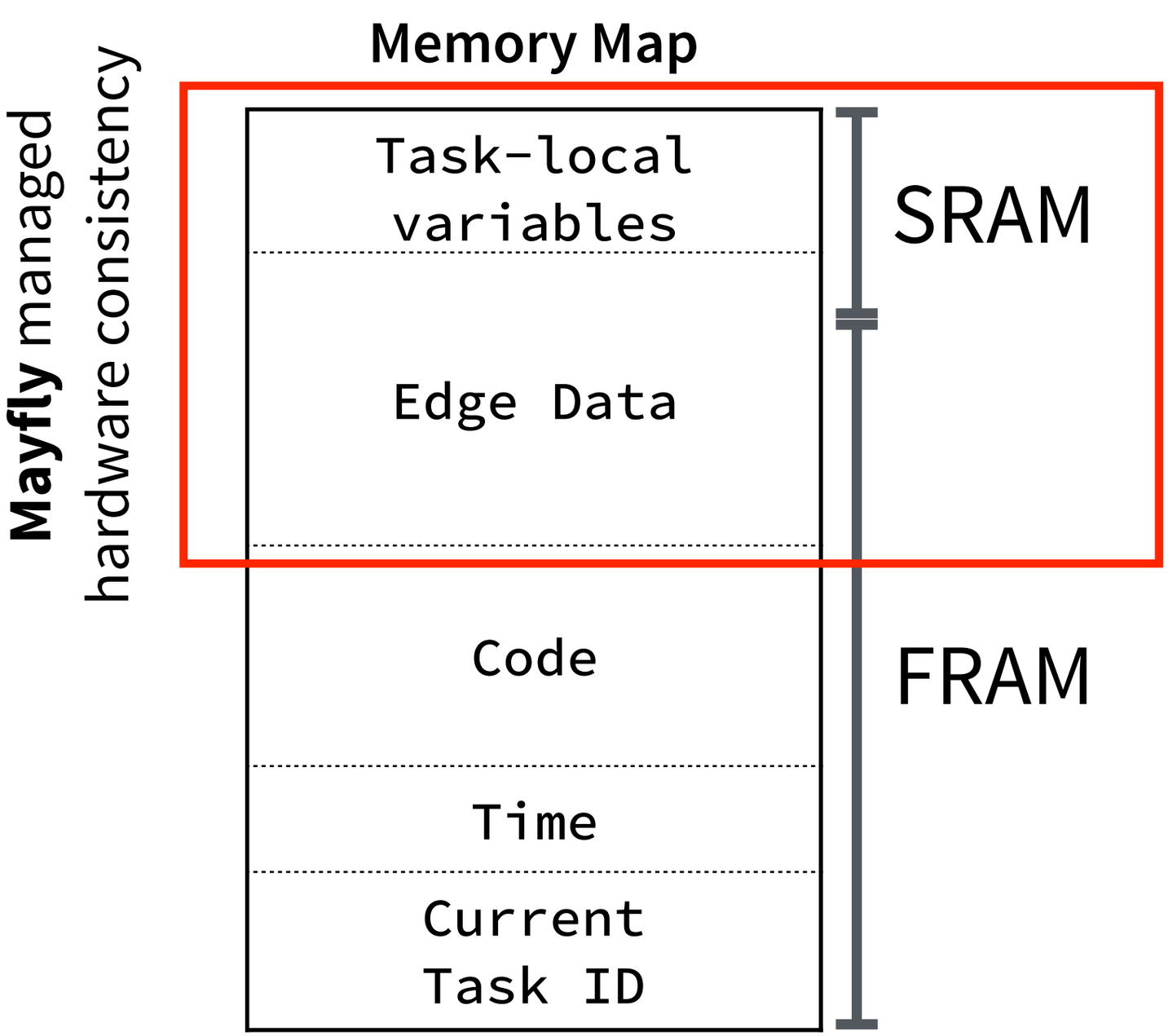


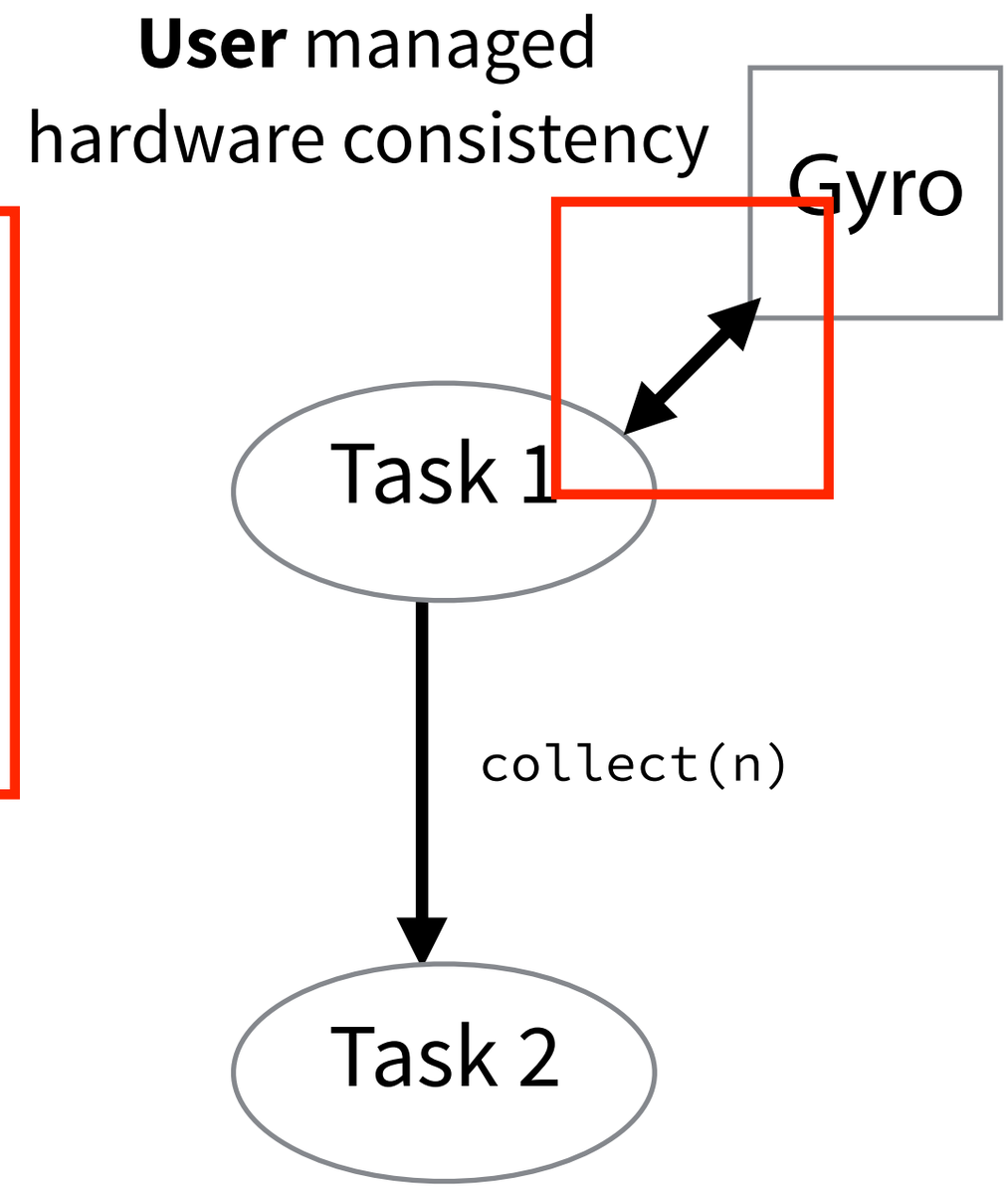



# Memory Model



# Memory Model



# Applications

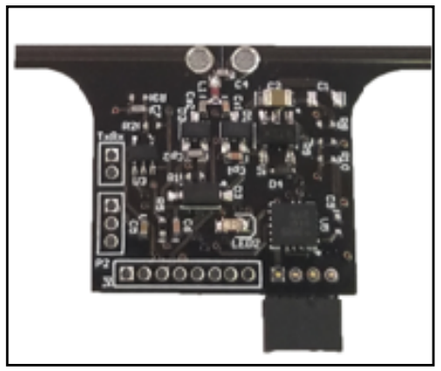
**Cold-chain Equipment Monitoring**
Logs temperature at a steady rate to monitor environment sensitive equipment, artifacts or people.

**Sit-to-stand Recognition**
Gathers acceleration readings and roughly classifies stand up and sit down movement.

**Time sensitive data.**

# Evaluation

**1.** Experimental Setup: WISP, RFID Reader



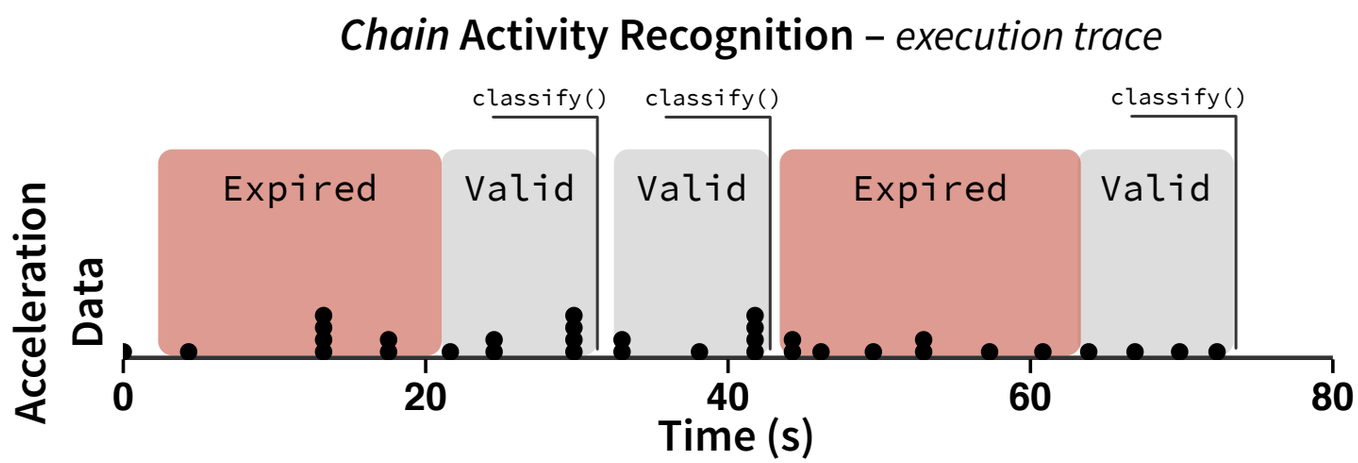
+

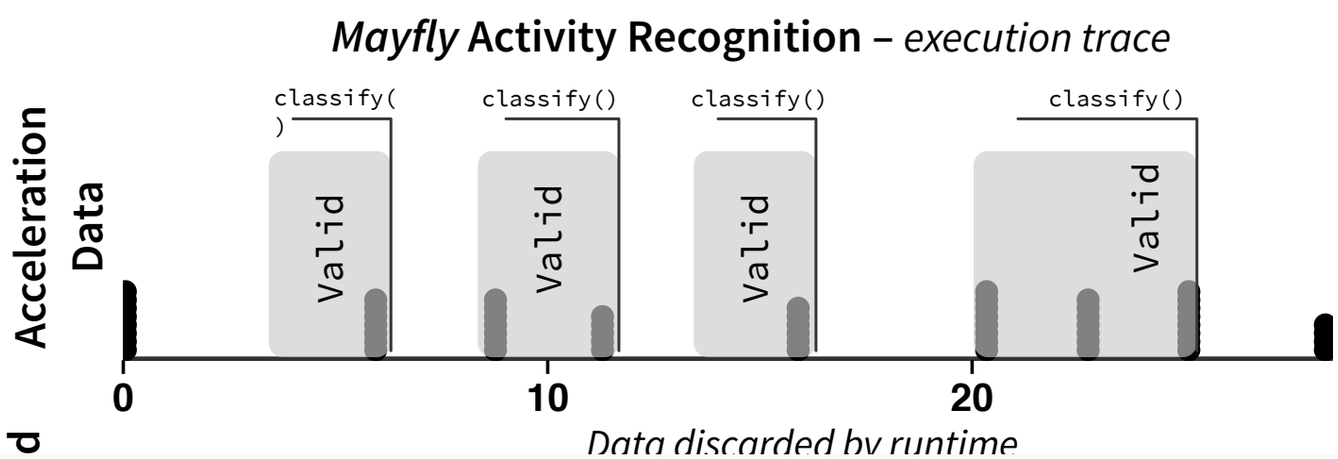
Custom Timekeeper PCB

**2.** Comparison against Chain + DINO

**3.** Experiments: *Data Utility, Memory, Overhead, Usability*

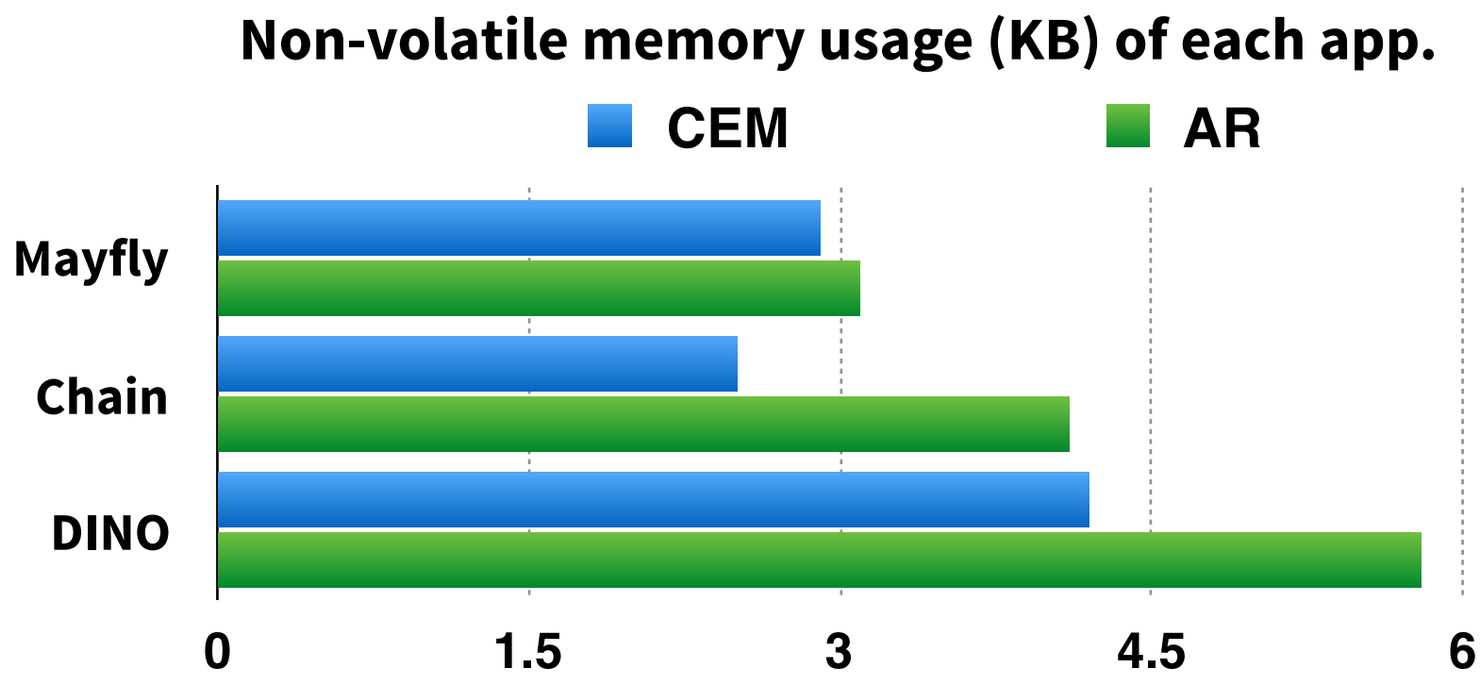
# Data Utility

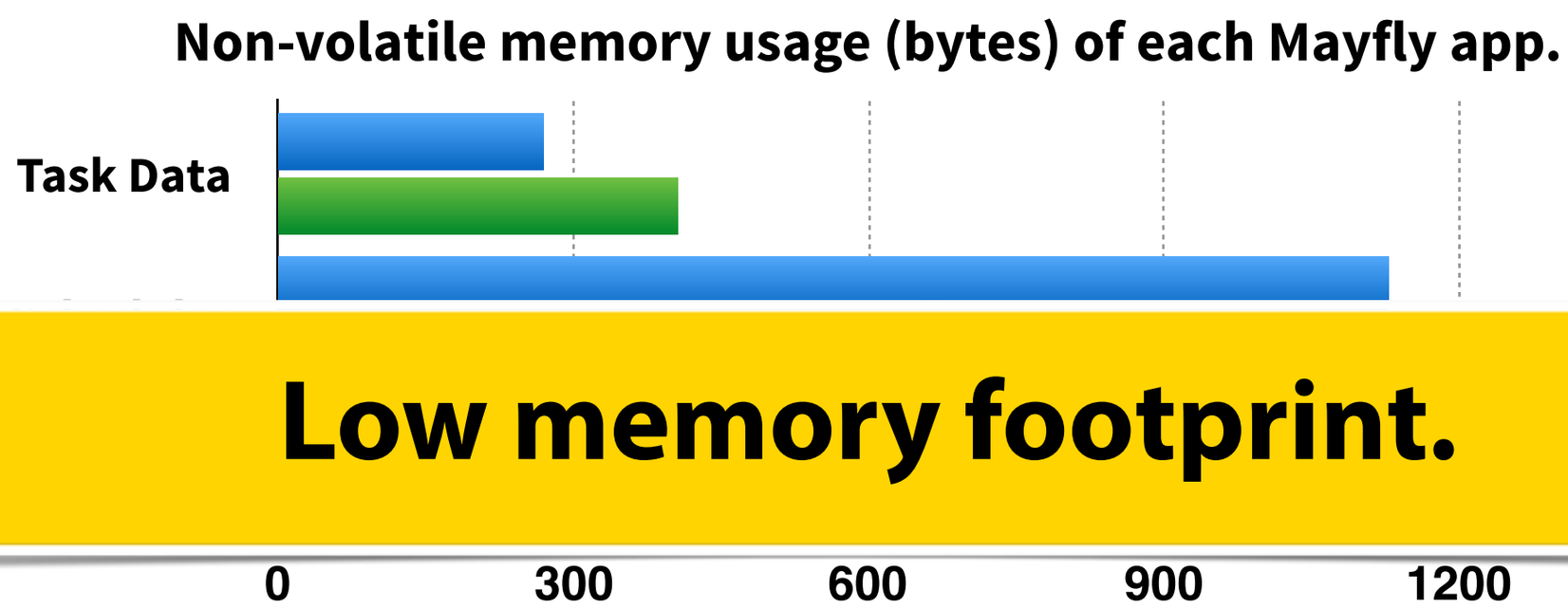



**Mayfly computes on useful data.**

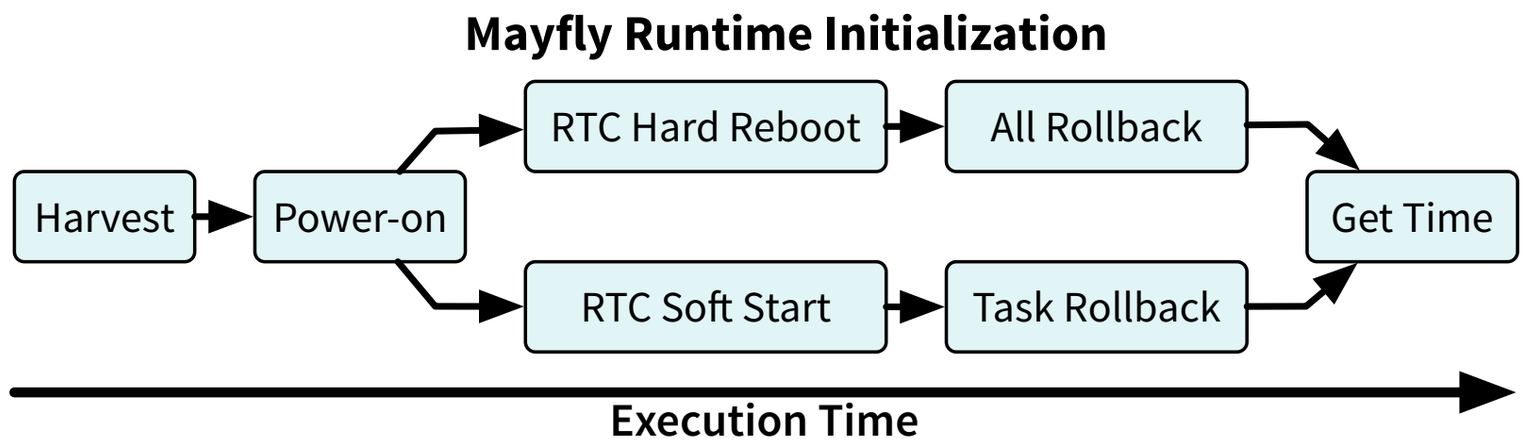
# Memory Footprint




**Low memory footprint.**

# Overhead




| Init function | Time |
|---|---|
| Power-on from brown-out | 1.0 ms |
| RTC hard reboot | 708.5 ms |
| RTC soft start | 144.6 µs |
| Rollback all Data | 23.6 µs |
| Single task rollback | 4.6 µs |
| Get time (seconds, minutes, hours) | 246.4 µs |
| Get time (seconds) | 80.6 µs |
| **Scheduler function** | |
| Process constraints for single task | 4.5 µs |
| Scheduler cant find task to execute | 56.3 µs |
| Task finished, commit results | 7.0 µs |

**Needs improvement**

# User Study

**1.** Reduce program writing time?

**2.** Increase understanding of intermittence?

**3.** Will programmers use Mayfly?

**11 participants, 2.5hrs, 3 coding challenges**

# Future Work

**1.** Triggered Tasks

**2.** Dynamic Scheduling

**3.** Lightweight Memory Versioning

# Mayfly

**1.** Coordination language–built on C

**2.** Relates time and data.

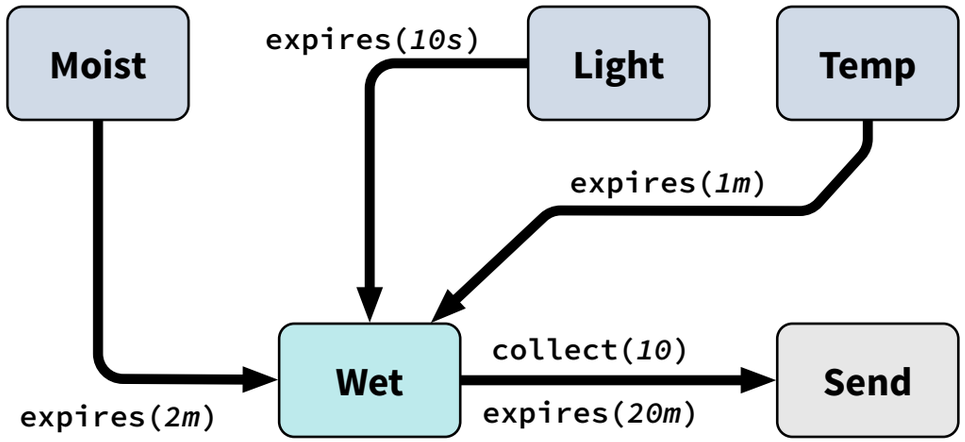
**3.** Code maps to program behavior.

**https://github.com/PERSISTLab/**

Northwestern
McCORMICK SCHOOL
OF ENGINEERING

# Timely Execution on Intermittently Powered Batteryless Sensors


**Josiah Hester**
Kevin Storer @ *UCI*
Jacob Sorber @ *Clemson*


SenSys'17—November 7, 2017

https://github.com/PERSISTLab/

josiah@northwestern.edu josiahhester.com