NVRAM in embedded systems

impact on software layers

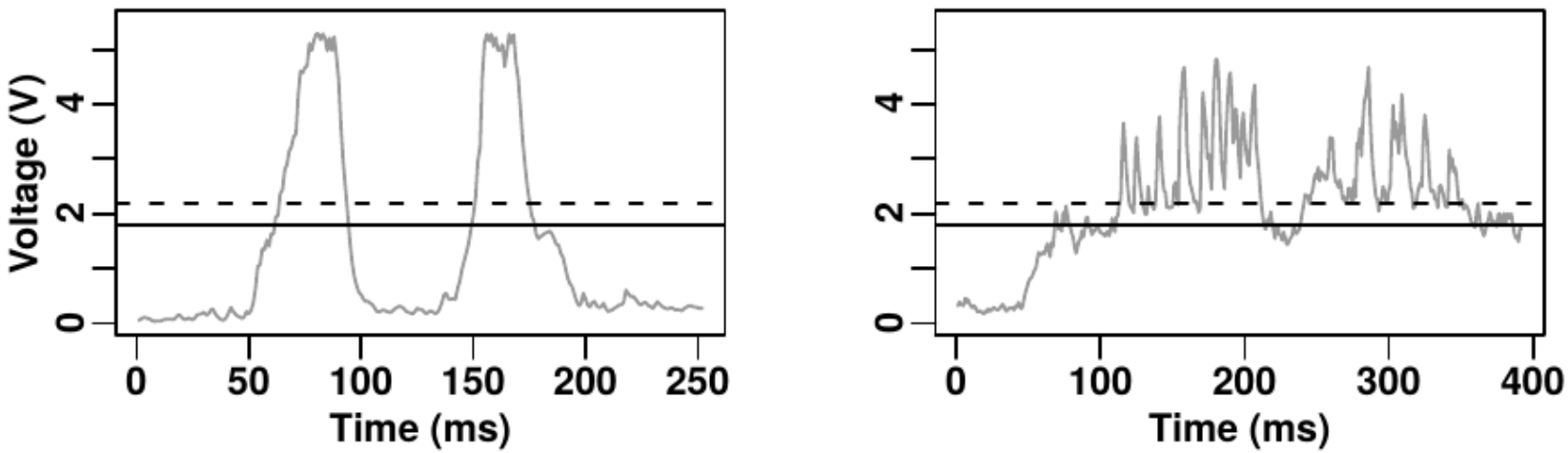
Gautier Berthou, Tristan Delizy, Kevin Marquet,
Tanguy Risset, Guillaume Salagnac

INSA Lyon, Citi Lab, INRIA/Socrate

Fréjus, March 25th 2018

Embedded systems specifics 1

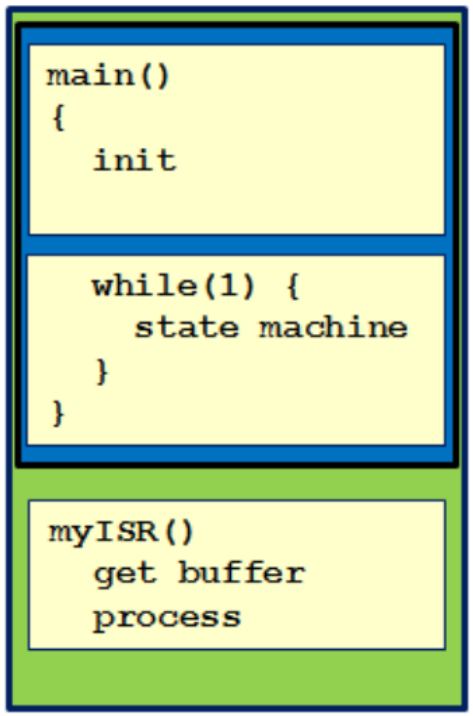
- Energy management
  - Extend lifetime
  - or expect power shortages



Energy consumption (static and dynamic) of NVRAM has an impact

Embedded systems specifics 2

- Specific programming model
  - Bare-metal
  - or dedicated operating system (Contiki, RIOT, FreeRTOS...)
  - or general-purpose OS (Linux)



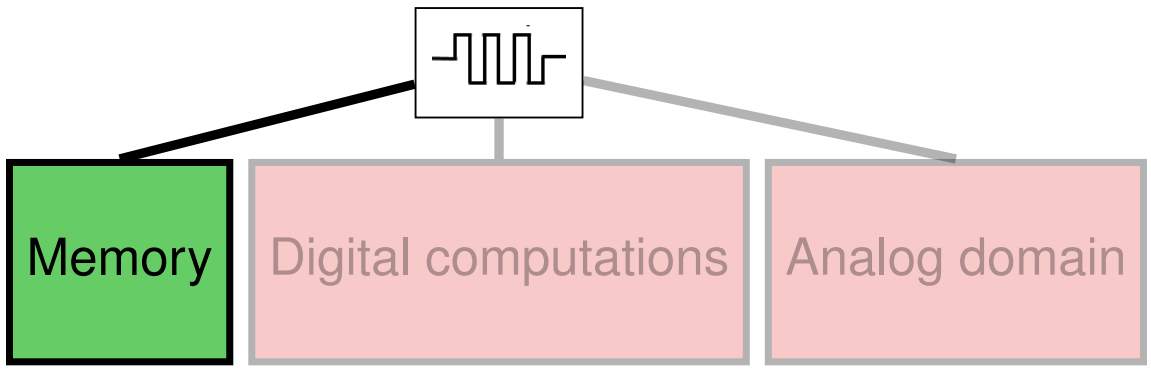
Persistence of NVRAM has an impact

Embedded systems specifics 3

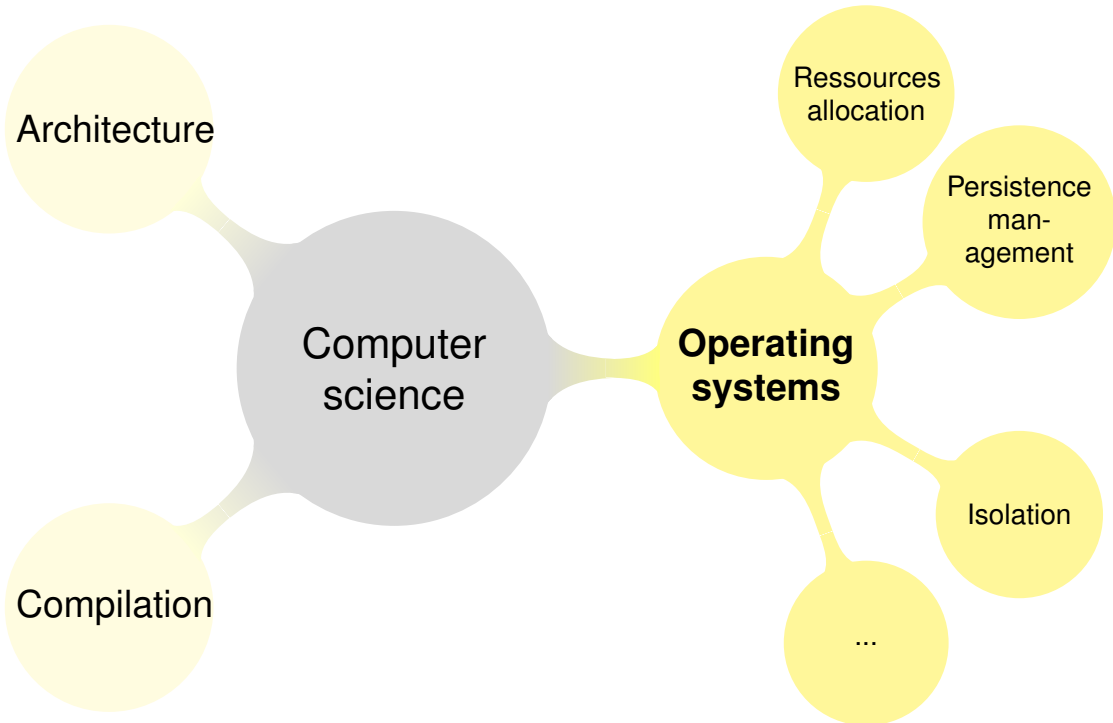
- Dedicated purpose
  - RFID tag, sensor, image processing...
  - Memory hierarchy may be chosen in this goal

Performances of NVRAM has an impact

Point of view: memristor as memory



Point of view: we design the OS



Outline

(achieved) Sytare OS persistence management

- Power not predictable

(in-progress) Dycton NVRAM management

- Normally-off computing

(just started) INRIA ZEP Multi-team project

- Hard + soft
- Predictable power *or* not

Outline

(achieved) Sytare OS persistence management

- Power not predictable

(in-progress) Dycton NVRAM management

- Normally-off computing

(just started) INRIA ZEP Multi-team project

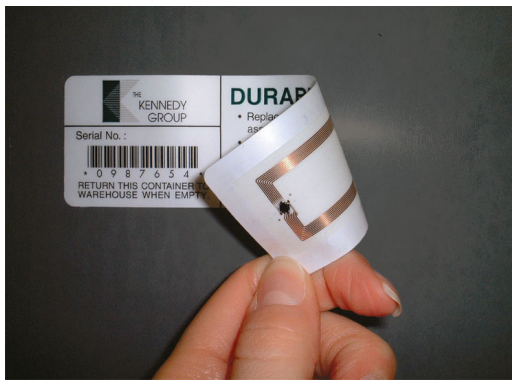
- Hard + soft
- Predictable power *or* not

Context: *Transiently Powered Systems (TPS)*

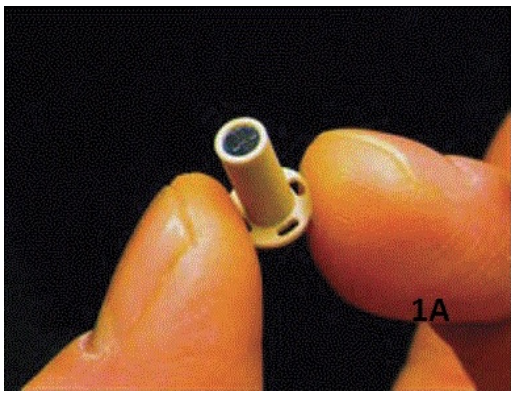
Communicating Tiny Things

- No battery ▶ must harvest power from the environment

smart cards



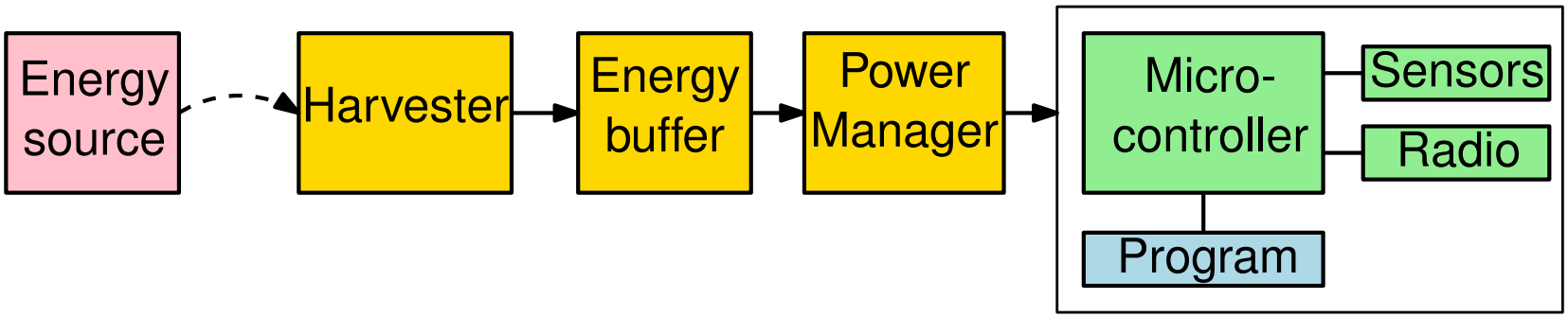
RFID tags

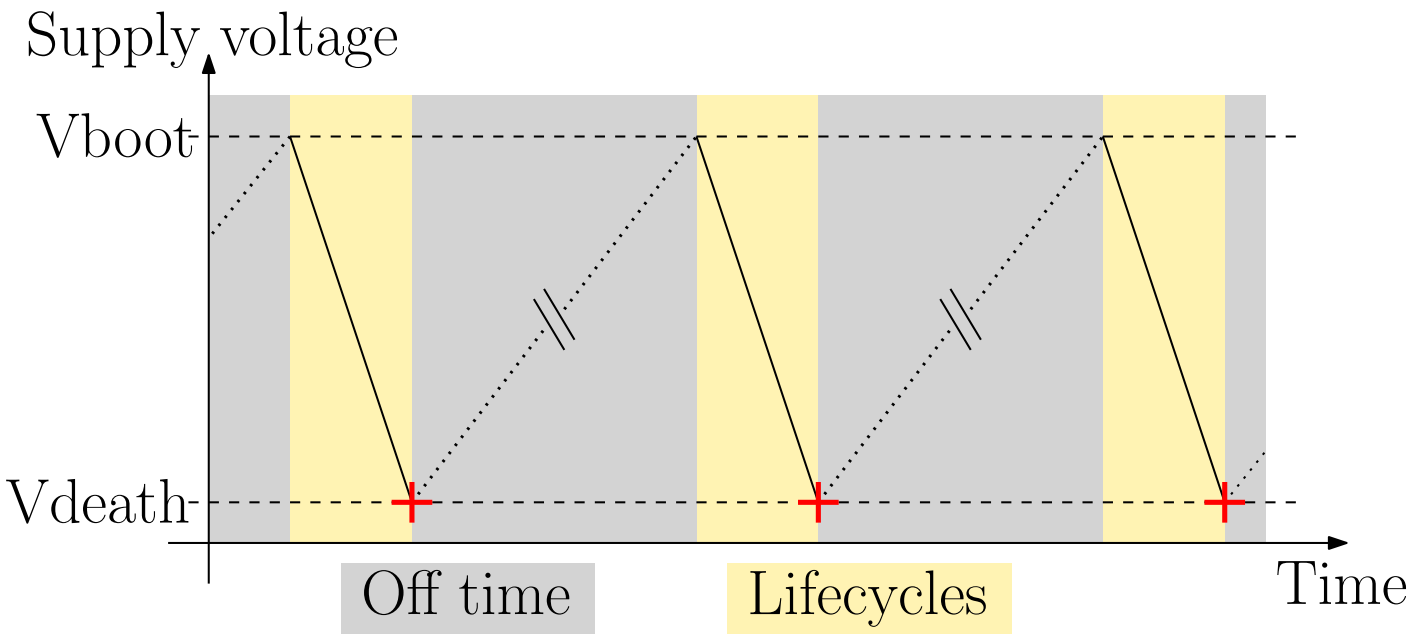


wearable sensors

- ▶ Wearable computing, home automation, environment monitoring, parking assistance, supply chain control...

Transient power = frequent power failures





Problem statement

Industrial Approach:

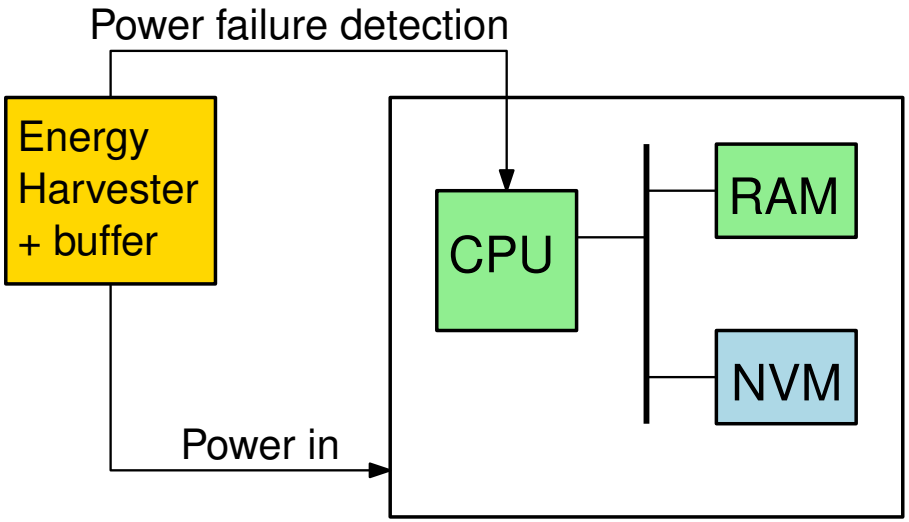
- Application software must run to completion in one lifecycle
- SW and HW are codesigned: one platform per application

How to run arbitrary code despite frequent, unexpected reboots?

Academic approach:

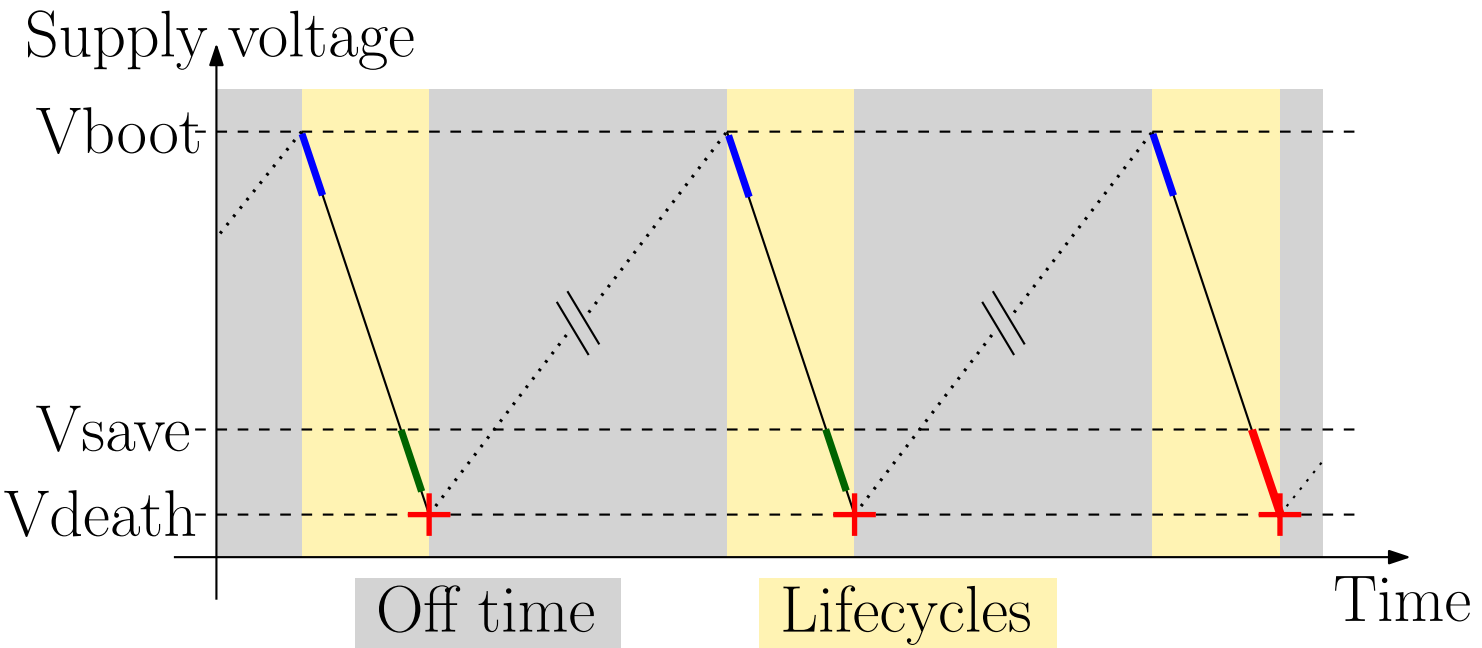
- Spread execution across multiple lifecycles

State of the art: program checkpointing



Program Checkpointing:

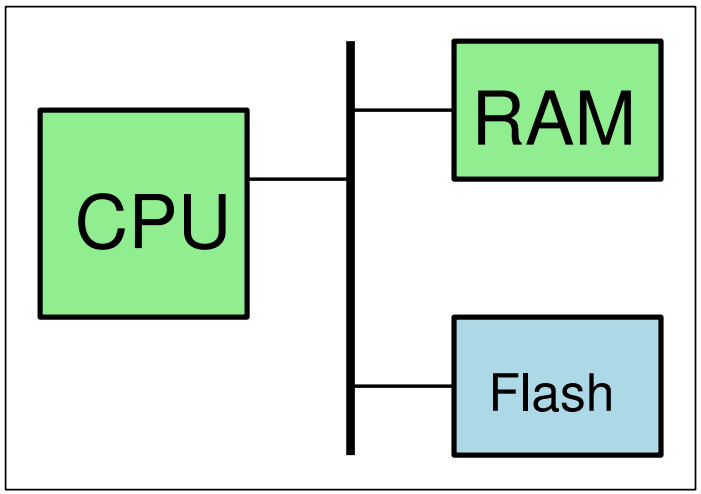
- Anticipate power failures
- Save program state to a non-volatile memory
- Restore state on next boot



Checkpointing for Transiently Powered Systems

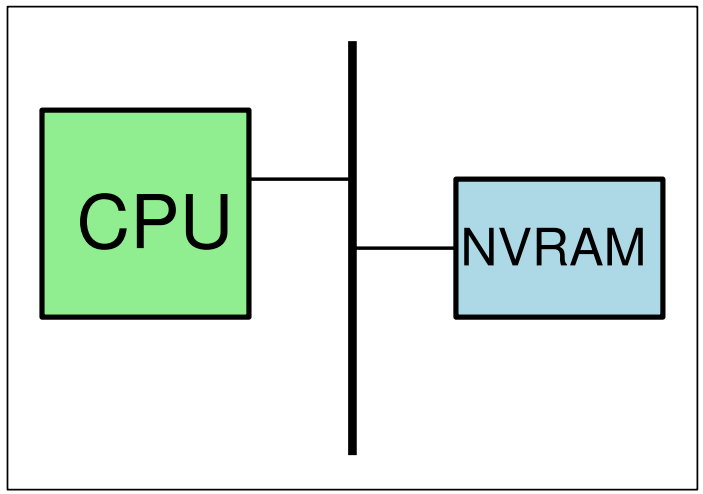
[Ransford *et al* ’13]

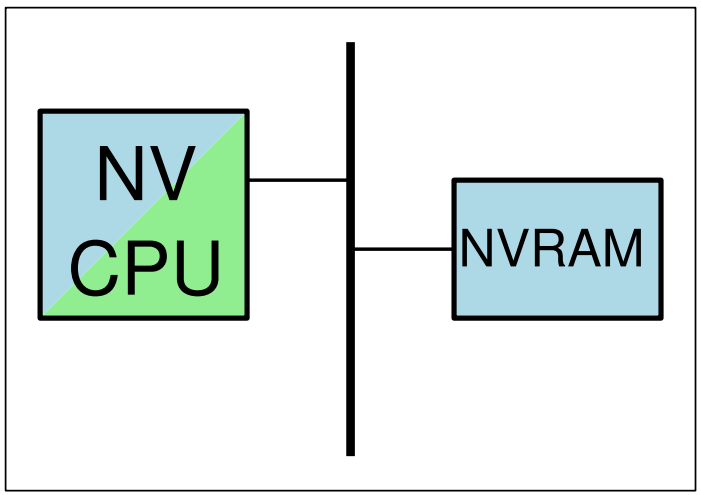
[Bhatti & Mottola ’16]



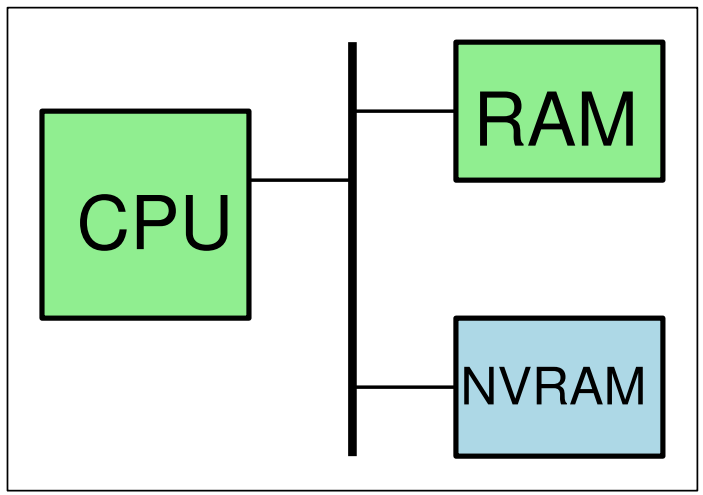
[Lucia & Ransford ’15]

[Jayakumar *et al* ’14]





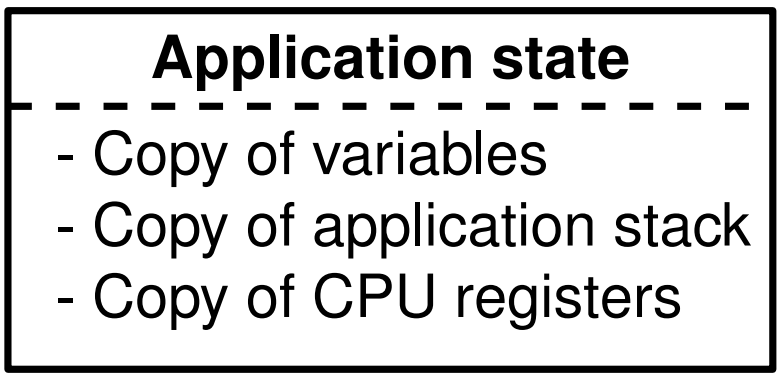
[Liu *et al* ’15]



[Balsamo *et al* ’15, ’16]

[Ait Aoudia *et al* ’14] (previous work)

Typical checkpoint structure



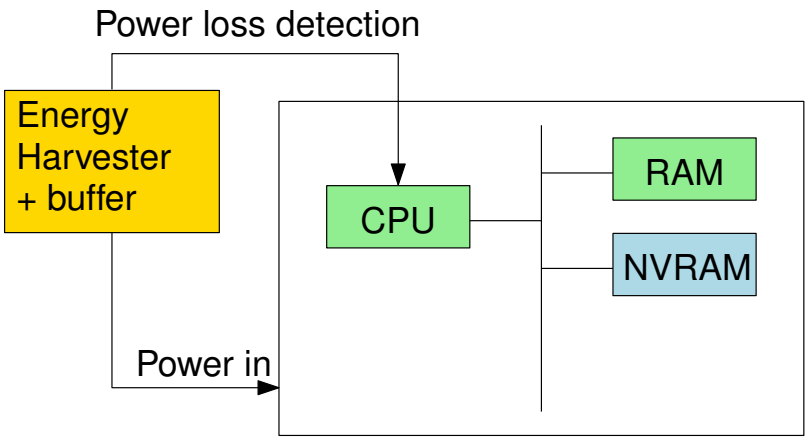
- Contains all relevant data to enable application persistence
- Stored in non-volatile memory

Checkpointing approaches

- Explicit code.
- Compile-time instrumentation:
  - User code explicitly defines code boundaries so that checkpoints are statically inserted. DINO [Lucia & Ransford '15] is an example.
  - Static analysis approach: Alpaca. [Maeng *et al* '17]
- Runtime checkpointing:
  - Pure hardware-based approach: with a non-volatile processor, perform light hardware checkpointing on write-back stage. [Liu & Jung '16]
  - Operating System based approach.

We focus here on an **Operating System** approach.

Making peripherals persistent, too



Non trivial initialization

- timing, polling, ordering constraints

Non trivial access

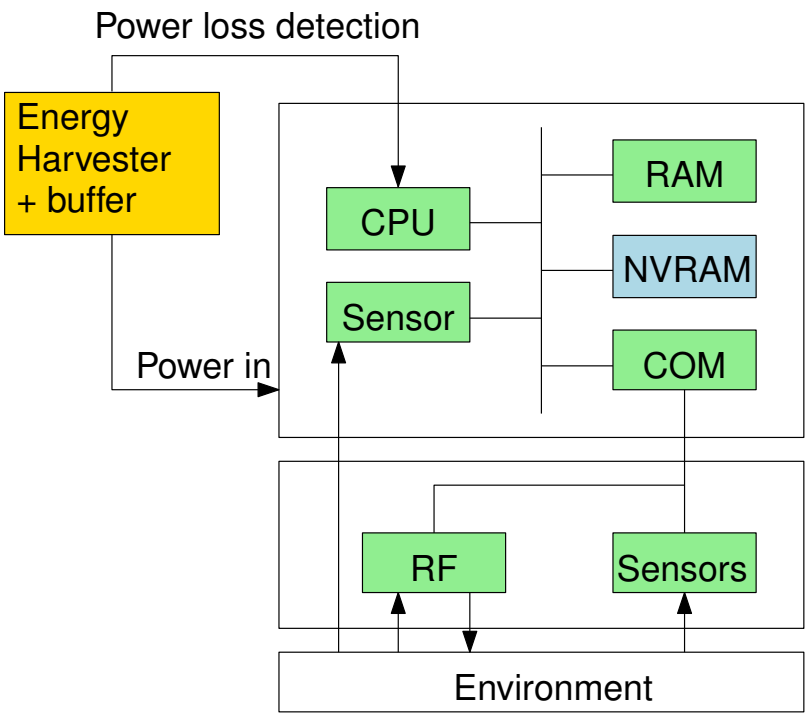
- not mapped in memory

Most peripherals do not support "resuming"

Interrupts carry volatile data

Program checkpointing is not enough

Making peripherals persistent, too



Non trivial initialization

- timing, polling, ordering constraints

Non trivial access

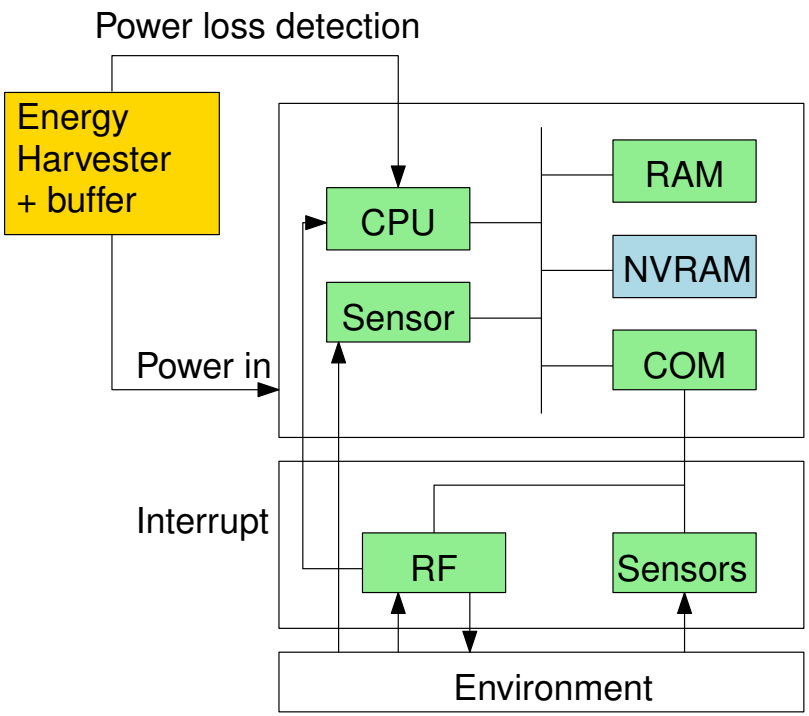
- not mapped in memory

Most peripherals do not support "resuming"

Interrupts carry volatile data

Program checkpointing is not enough

Making peripherals persistent, too



Non trivial initialization

- timing, polling, ordering constraints

Non trivial access

- not mapped in memory

Most peripherals do not support "resuming"

Interrupts carry volatile data

Program checkpointing is not enough

Approach

Run a small OS kernel that:

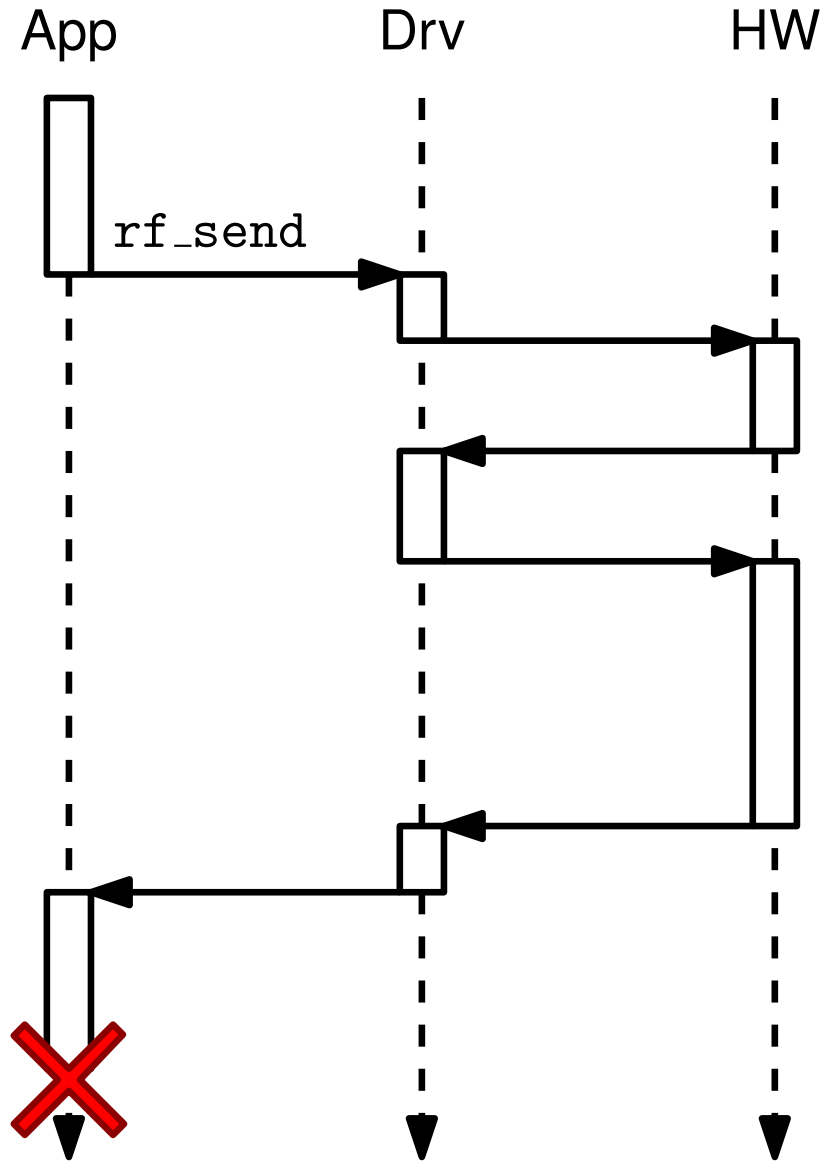
- Impose a **driver API**
- Design a **checkpointing** mechanism
- Specify an **interrupt management model**
- Define the notion of ***device context***
- Re-define the notion of ***system call***

The Peripheral State Volatility Problem

Application code

```
void main(void){
   sensor_init();
   rf_init(myconfig);

   for(;;){
     v = sensor_read();
     rf_send(v);
     ...
  }
}
```

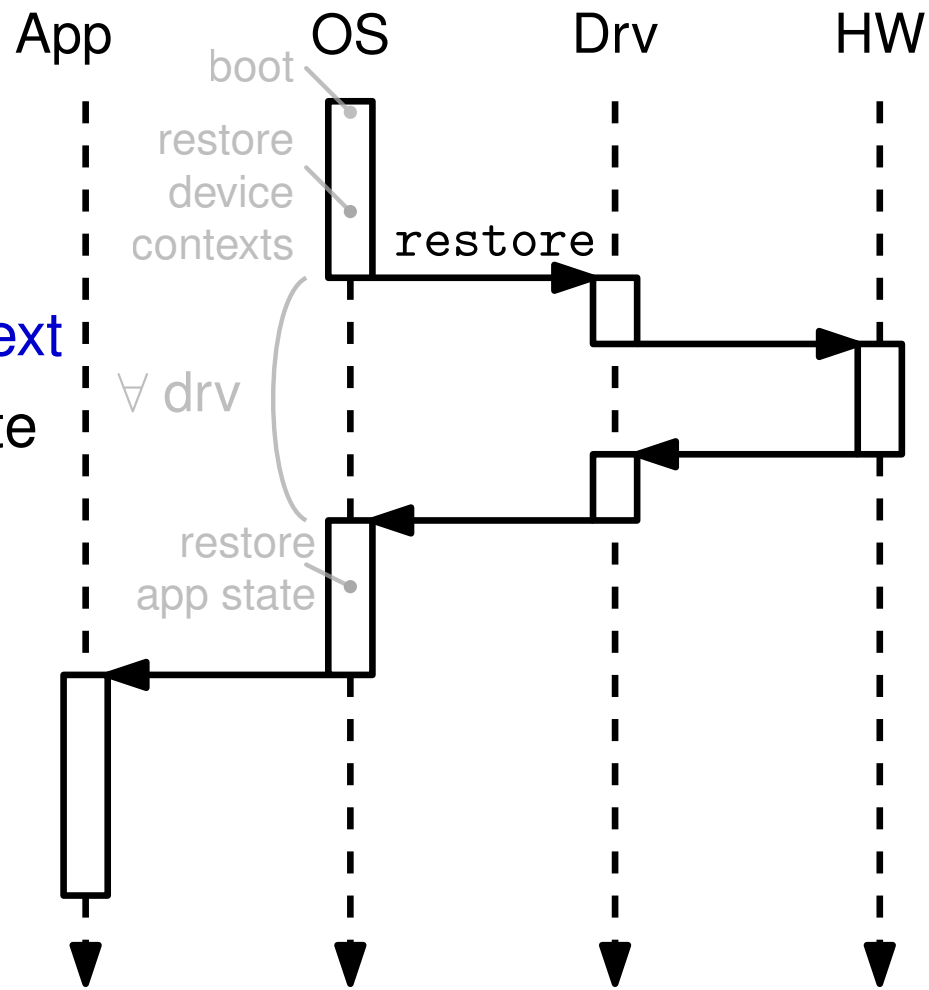


Restoring memory content will not restore device state

Our approach: distinct roles for OS and drivers

Each driver:

- Provides a `restore()` function
  `init()` + transitions to saved state
- Puts its variables into a device context
  that describes a “restore()-able” state



Operating System:

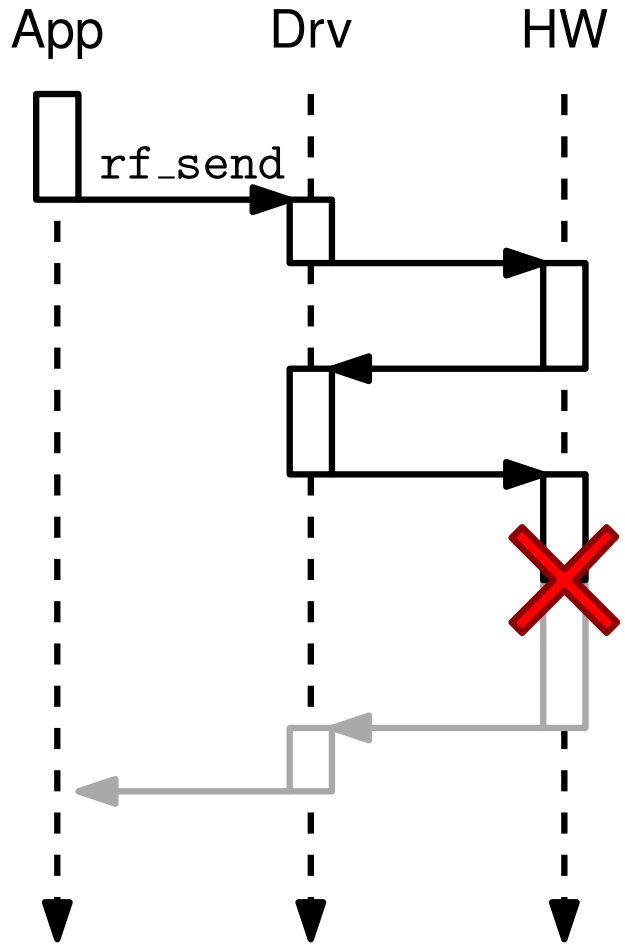
- Persists device contexts
- Calls every `restore()` function
- Persists application state

The Peripheral Access Atomicity Problem

Application code

```
void main(void){
   sensor_init();
   rf_init(myconfig);

   for(;;){
     v = sensor_read();
     rf_send(v);
     ...
  }
}
```



In most cases, resuming execution in the middle of a hardware access does not make sense

Our approach: make driver calls atomic

Encapsulate driver functions into OS wrappers.

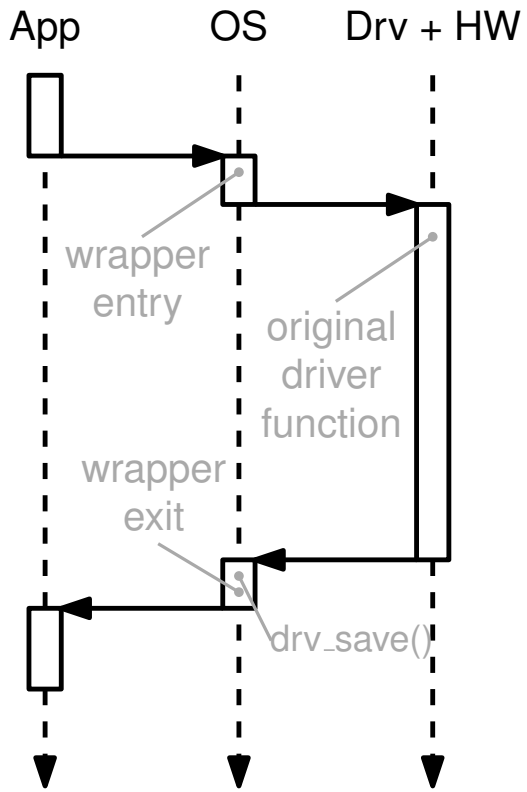
Each driver provides a `save()` function that copies device context into checkpoint image.

On wrapper entry:

- save arguments + function called
- switch to volatile stack

On wrapper exit:

- save device contexts
- clear arguments
- switch back to application stack



Interrupted driver calls are retried and not just resumed. Define the notion of syscall.

Our approach: make driver calls atomic

Encapsulate driver functions into OS wrappers.

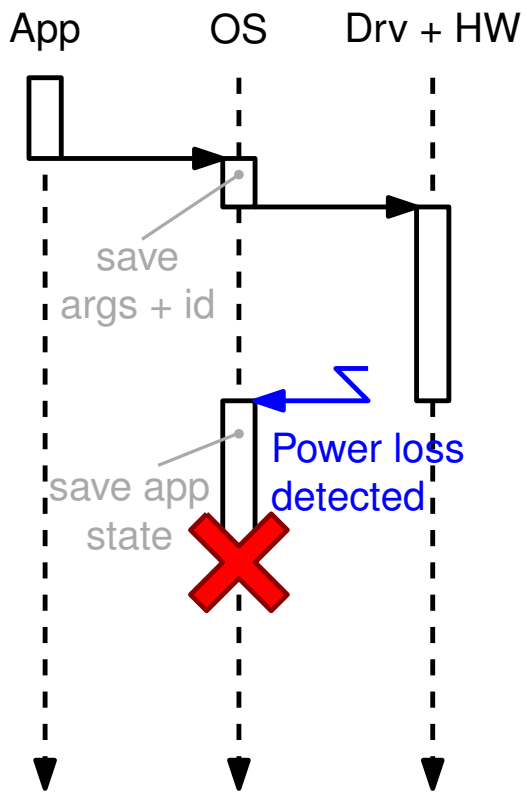
Each driver provides a `save()` function that copies device context into checkpoint image.

On wrapper entry:

- save arguments + function called
- switch to volatile stack

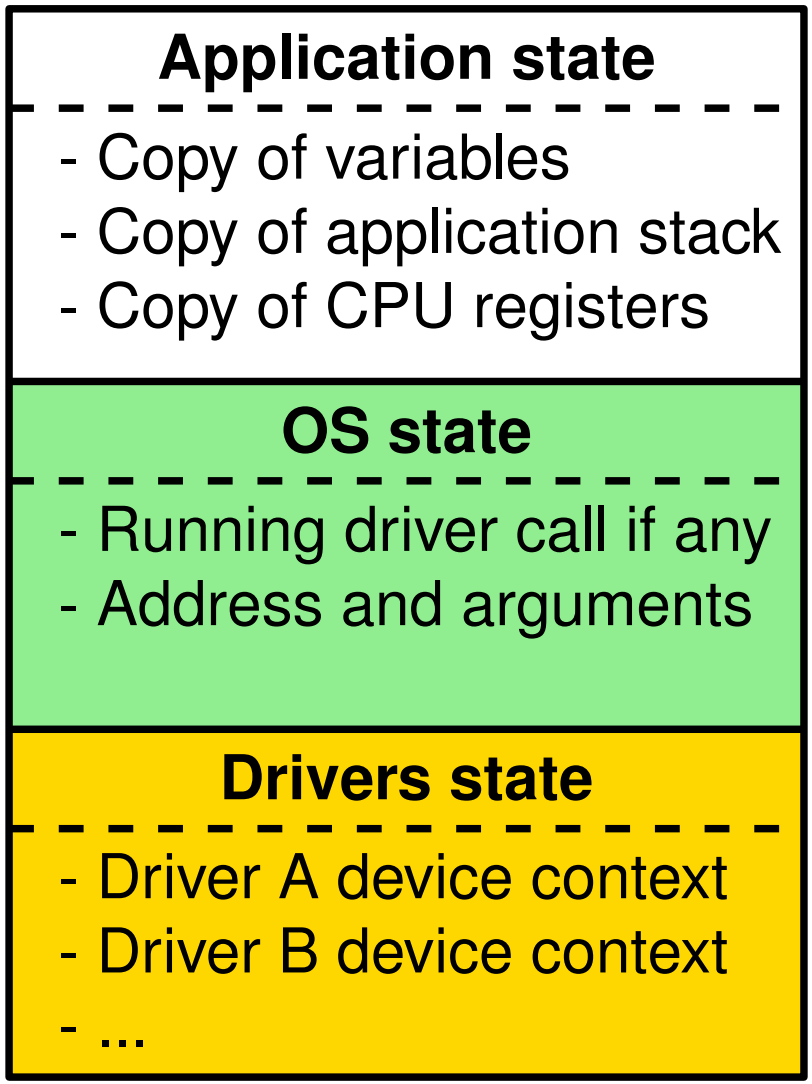
On wrapper exit:

- save device contexts
- clear arguments
- switch back to application stack



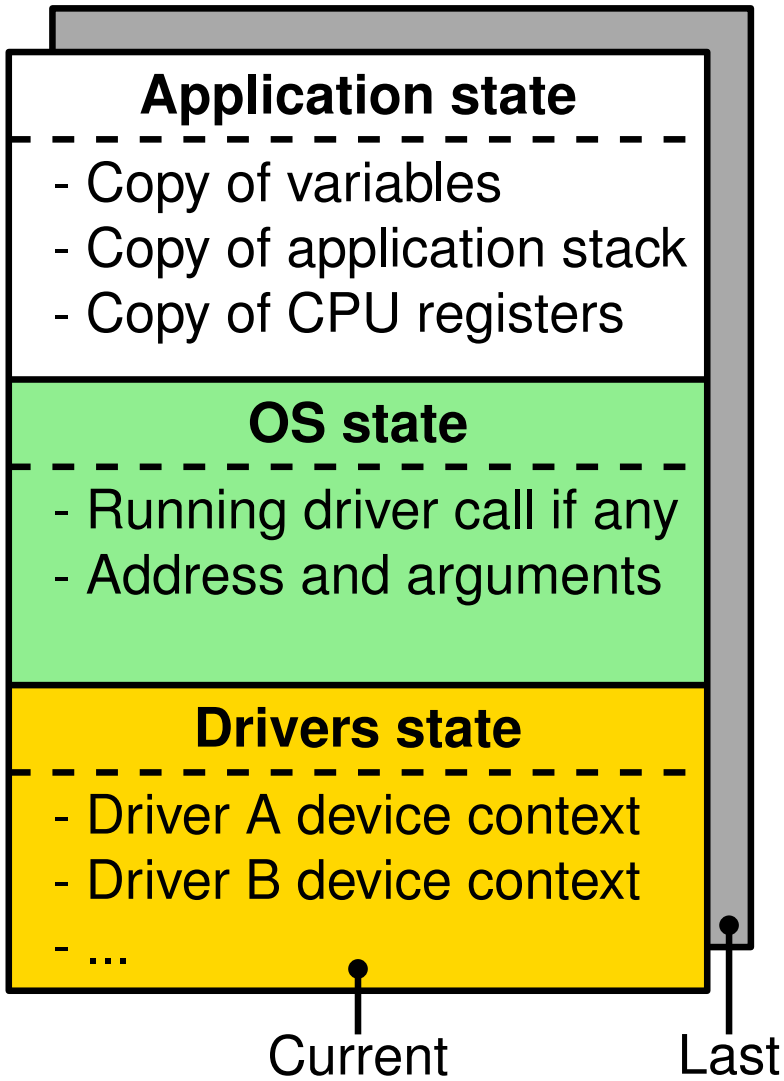
Interrupted driver calls are retried and not just resumed.

Checkpoint structure



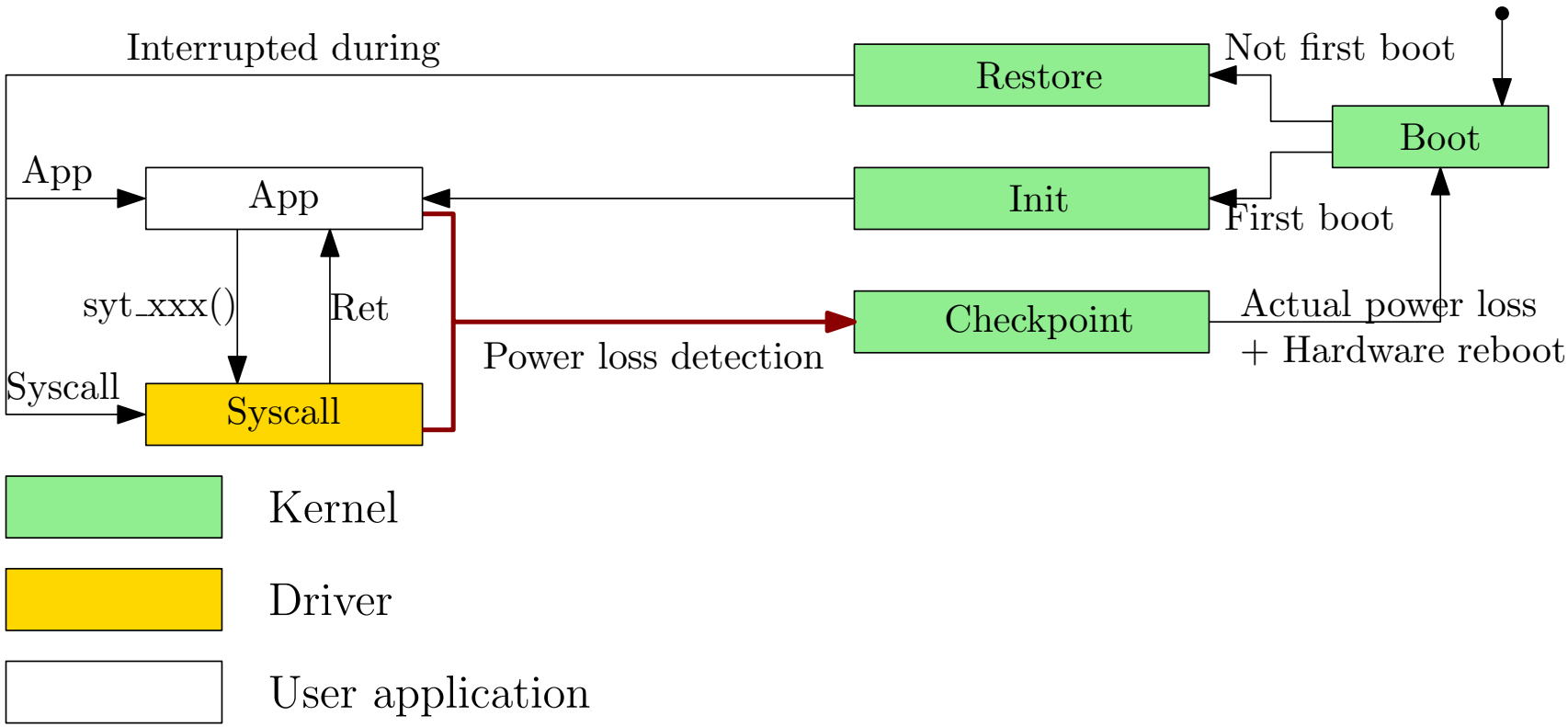
- Checkpoint contains all relevant data to enable application and peripheral persistence
- Checkpoint is stored in non-volatile memory

Checkpoint structure - Continued

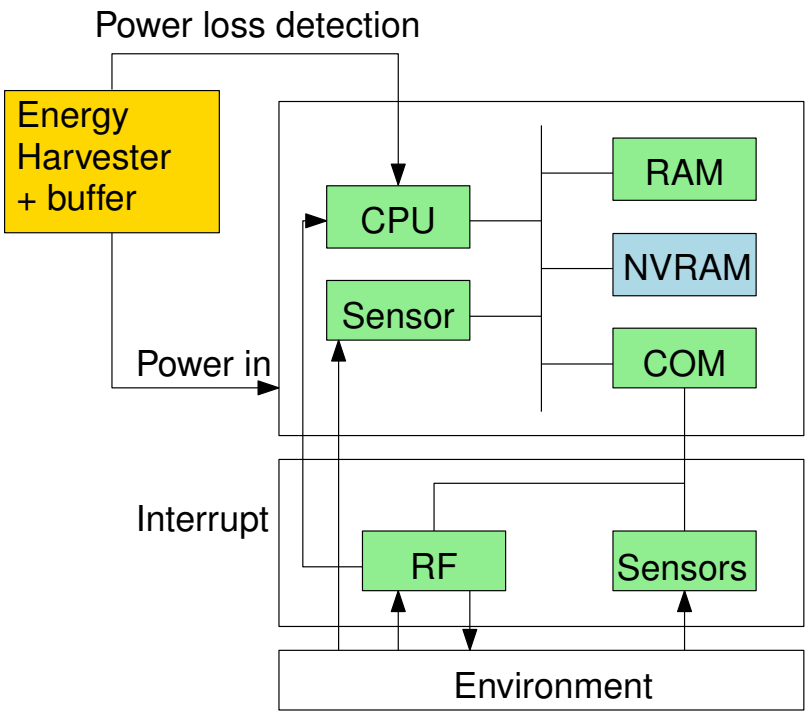


- Double buffered image stored in non-volatile memory.
- Last image contains the last stable state.
- Current image contains the progress made since last was built.
- On boot, states are restored using last image data.
- On power loss detection, both images are swapped atomically.

Operating System architecture



Pb: interrupts are volatile as well



- Embedded applications always use interrupts.
- User wants control over interrupt handling.
- Interrupts carry volatile data that will be lost upon power loss.
- Interrupt occurrence and data must be persisted.

Application code with interrupts

Example

```
ISR deviceA_interrupt()
{ ... }
ISR deviceB_interrupt()
{ ... }

void main(void){
  hardware_init();
  while(1) {
    ...
  }
}
```

Interrupt-related problems

Problems not specific to transiently-powered systems:

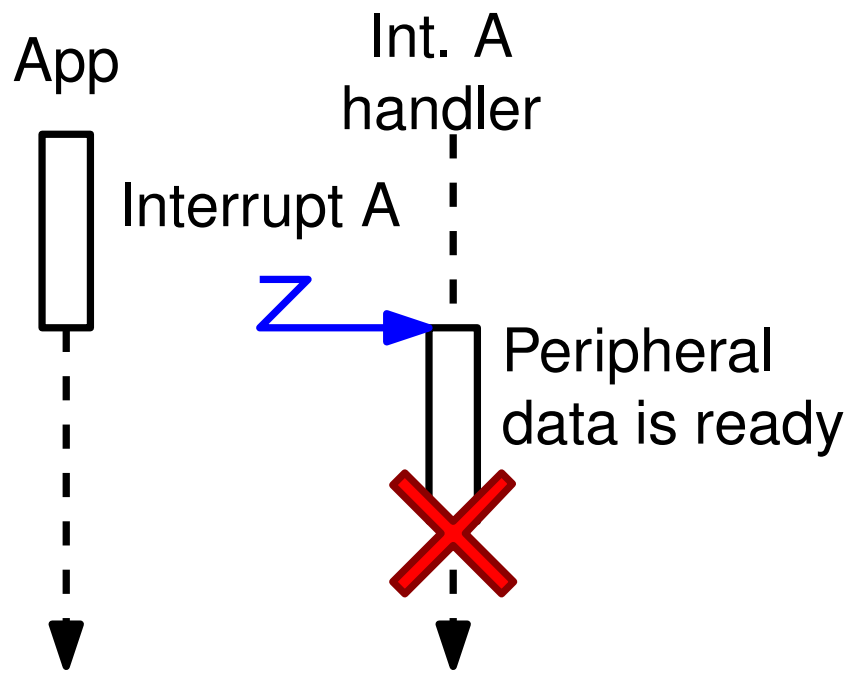
- Concurrency
- Race conditions
- Solution: critical sections with interrupts enabled.

Specific to transiently-powered systems:

- Interrupt data volatility

Interrupt data volatility

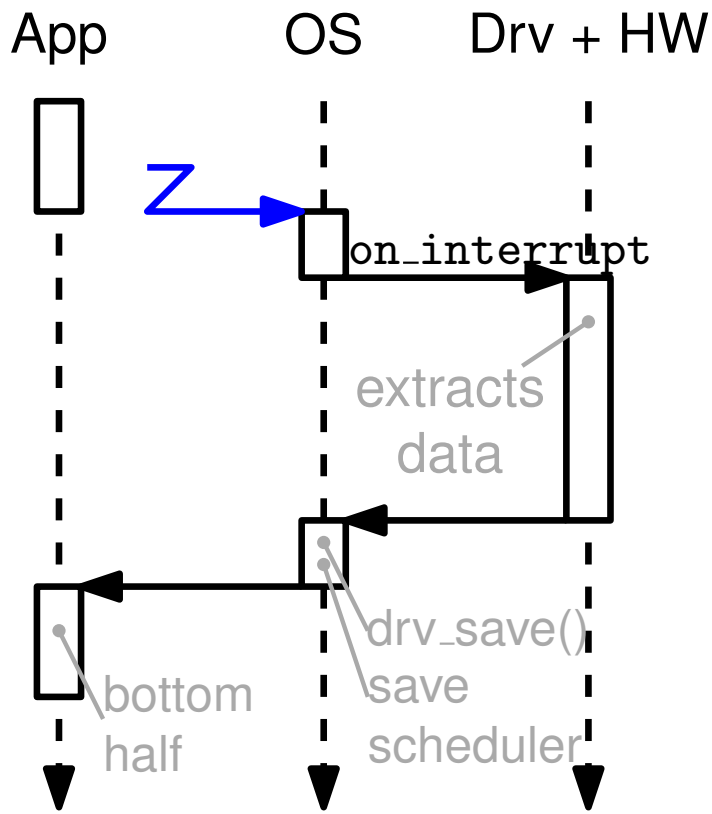
- Interrupt occurrence is volatile data.
- Peripheral data, *e.g.*, radio packet content, are also volatile data.



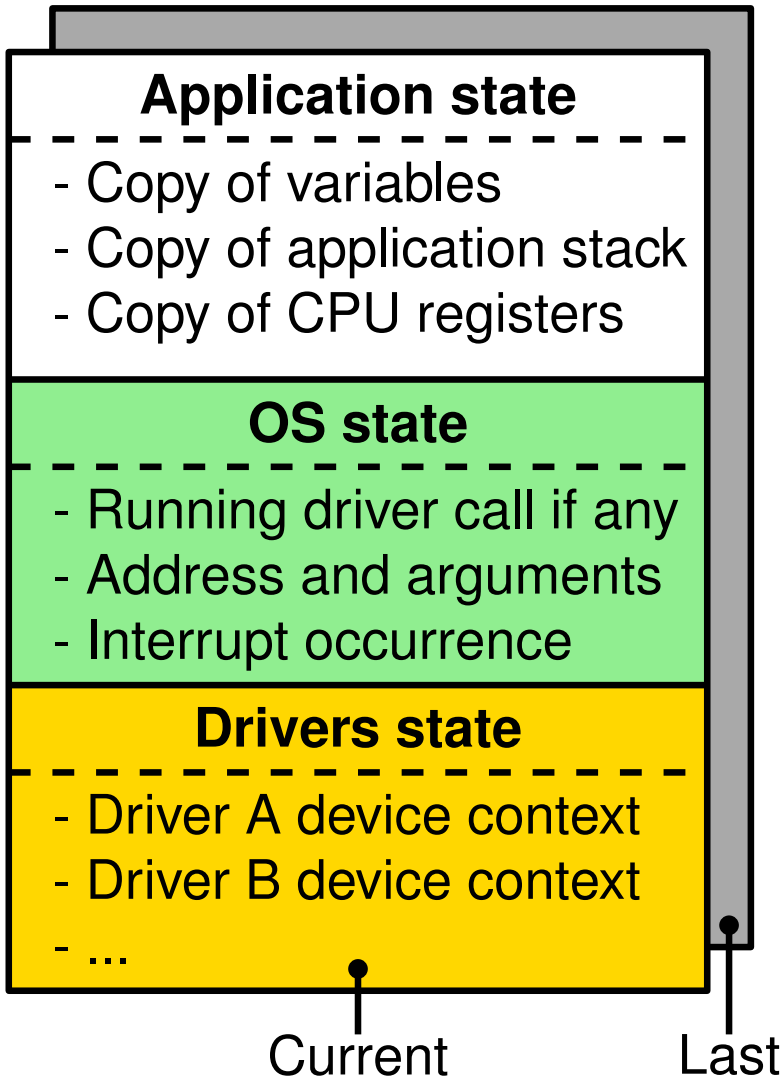
Our approach: extend solution to peripheral state volatility problem

OS-managed top halves and user-managed bottom halves

- Each driver provides an `on_interrupt()` routine.
- Each top half calls the `on_interrupt()` routine of relevant drivers.

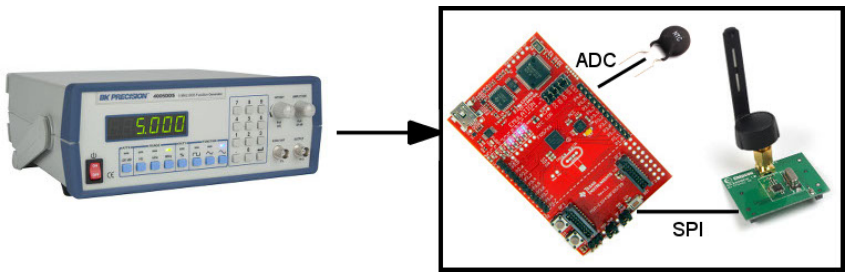


Checkpoint structure - Completed



- Information about interrupt occurrence are kept in the OS section of the checkpoint image.
- Data carried by interrupts are kept in the relevant device drivers. Ex: radio packet content is owned by the radio chipset driver.

Sytare Evaluation Setup



- MSP430FR5739: 16-bit CPU 24MHz, 1kB SRAM, 15kB FeRAM
- RF-chip: CC2500

Application

```
void main(void){
    syt_sensor_init();
    syt_rf_init(myconfig);

    for(;;){
      v = syt_sensor_read();
      compute();
      syt_rf_send(v);
      ...
   }
}
```

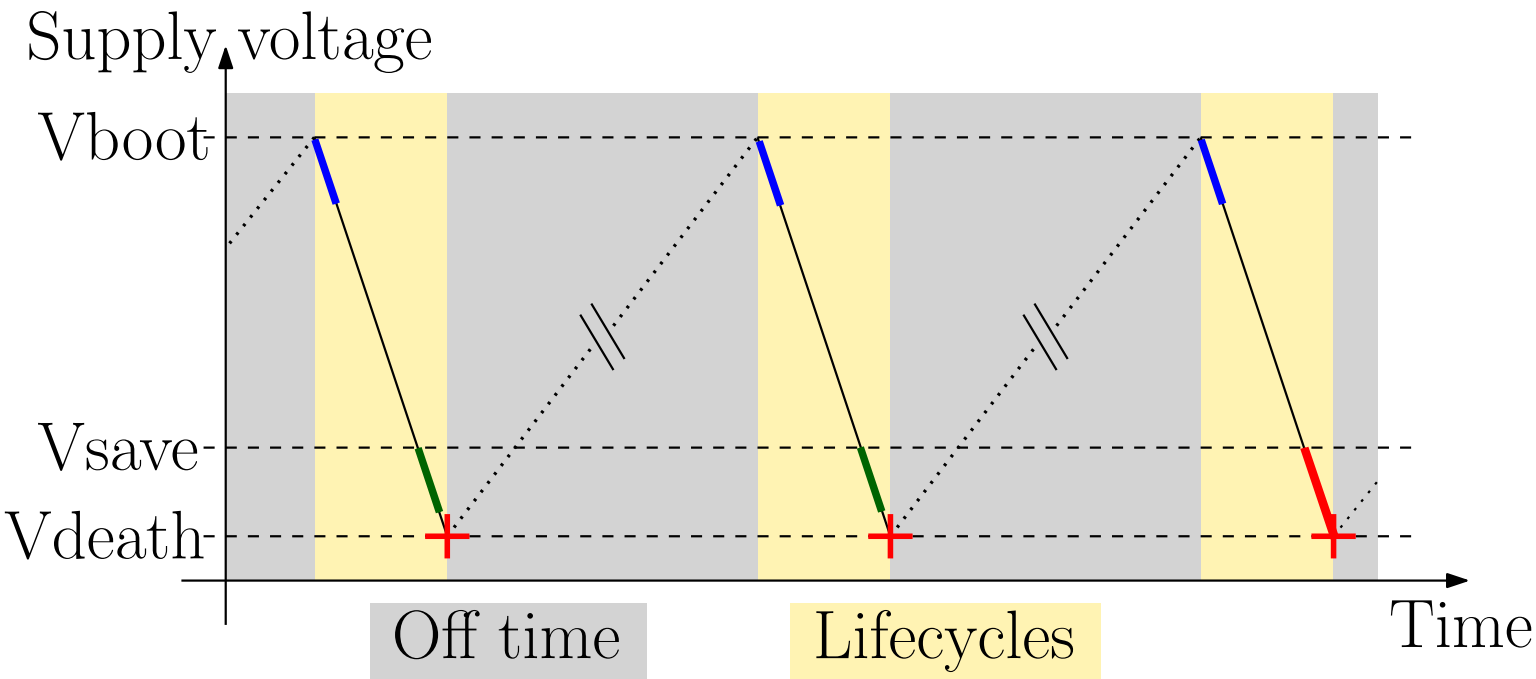
Evaluation methodology

Experimental setup

- Varying parameter: lifecycle duration

Ground-truth

- Same application without OS layer
- Stable supply without outage



Evaluation methodology

Performance metrics

- Duration of shortest usable lifecycle
- Temporal execution efficiency

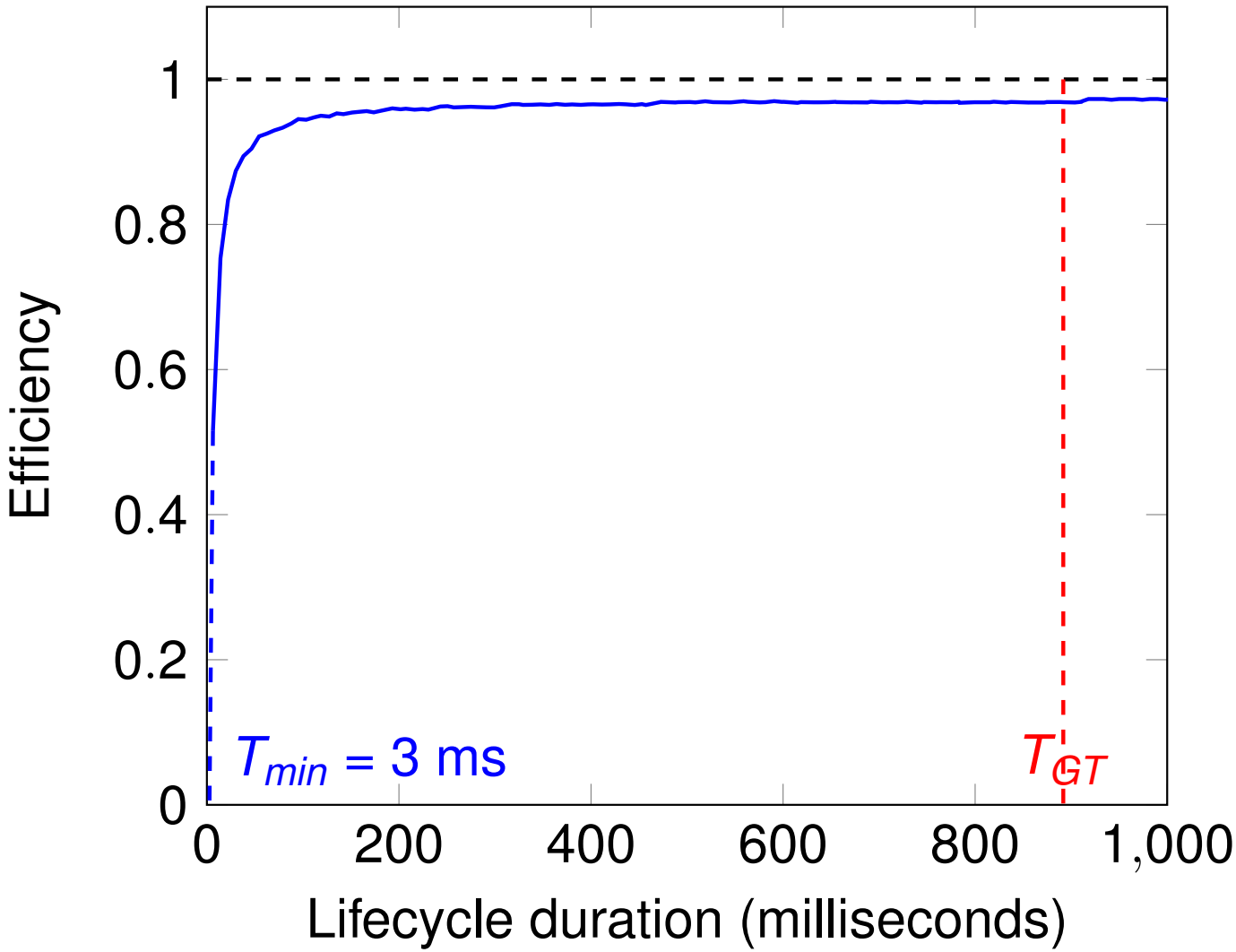
$$\text{Efficiency}(x) = \frac{T_{GT}}{T(x)}$$

x Lifecycle duration

T_{GT} Application runtime under ground-truth conditions

$T(x)$ Application runtime with OS layer when the platform is ON

Efficiency results



Results: Driver call temporal overhead

Driver calls are encapsulated into wrappers

Driver call	Overhead (%)
Led toggle	1263
ADC read	27
Radio sleep	137
Radio wake-up	8
Radio send	1

Conclusion

Peripheral State Persistence for Transiently Powered Systems

- Volatility: device contexts + `save()` and `restore()` methods
- Atomicity: retry VS resume

Project sources available at: `https://gitlab.inria.fr/citi-lab/sytare`

Perspectives:

- Persistence management to existing OS (RIOT, Contiki),
- Energy-based decision making,
- Support for multiprocessing,
- Design networking stacks and protocols,
- Formalisation of persistence-related problems.

Outline

(achieved) Sytare OS persistence management

- Power not predictable

(in-progress) Dycton NVRAM management

- normally-off

(just started) INRIA ZEP Multi-team project

- Hard + soft
- predictable *and* not predictable power

Dycton

Architecture:

- Several memory banks
  - slow and fast memories
  - ram + nvram
- Normally-off
- Quite powerful processor

Goal: save energy

Big difference: we choose when to switch off

- data allocation problem
- no peripheral persistence
- but persistence of some data: routing tables, radio channel config etc.

Outline

(achieved) Sytare OS persistence management

- Power not predictable

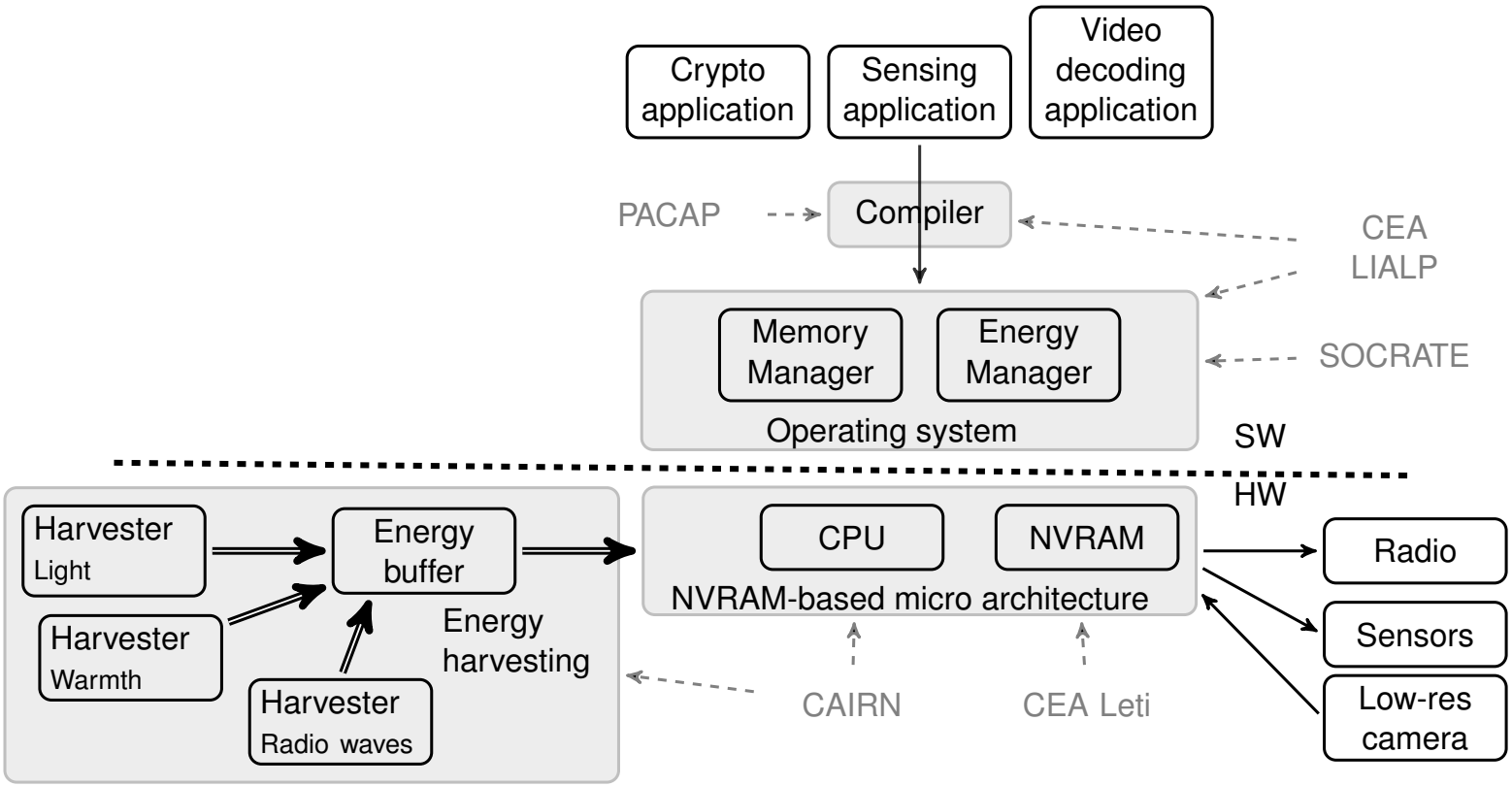
(in-progress) Dycton NVRAM management

- Normally-off

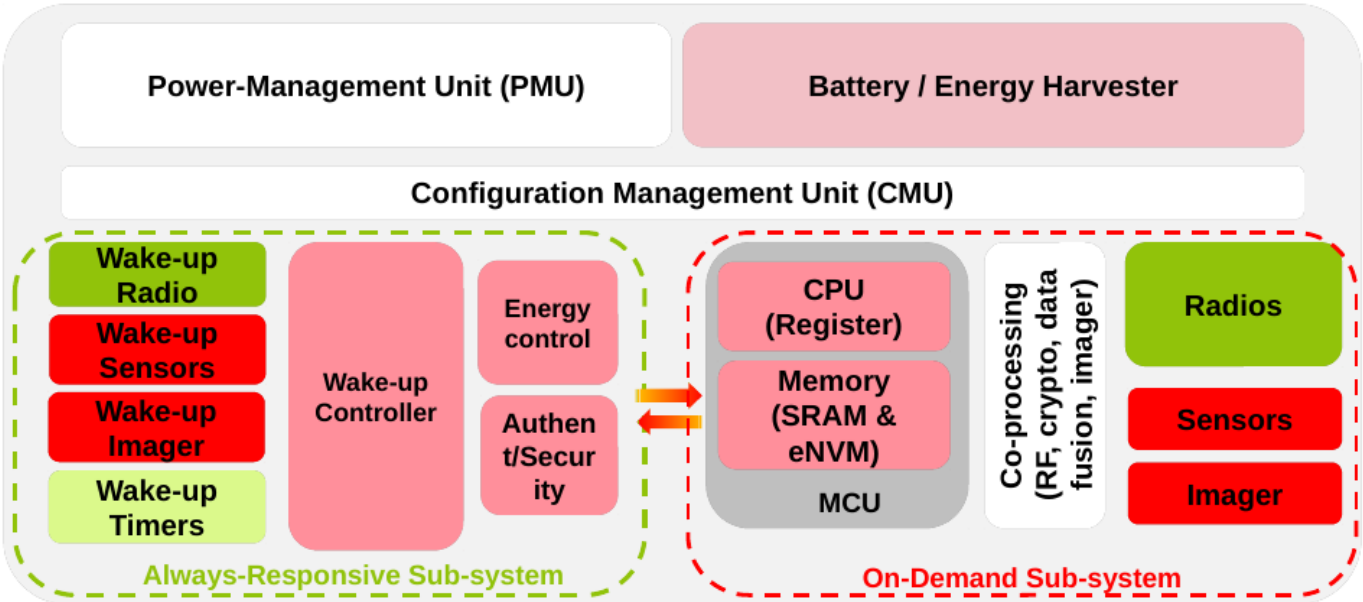
(just started) ZEP INRIA project

- Hard + soft
- Predictable *and* not predictable power

ZEP: ZEro-Power computing systems



Architecture



- Based on CEA Leti's L-IOT (RISC-V)
- $\longrightarrow$ Extension for nvram

Conclusion: parallel with high-end systems

- Programming model impacted
  - in multi-core processors: crashes are expected
  - example: transactions / lock

```
pthread_lock(&m);
...  e=malloc()...
...  tail=e...
pthread_unlock(&m);
```

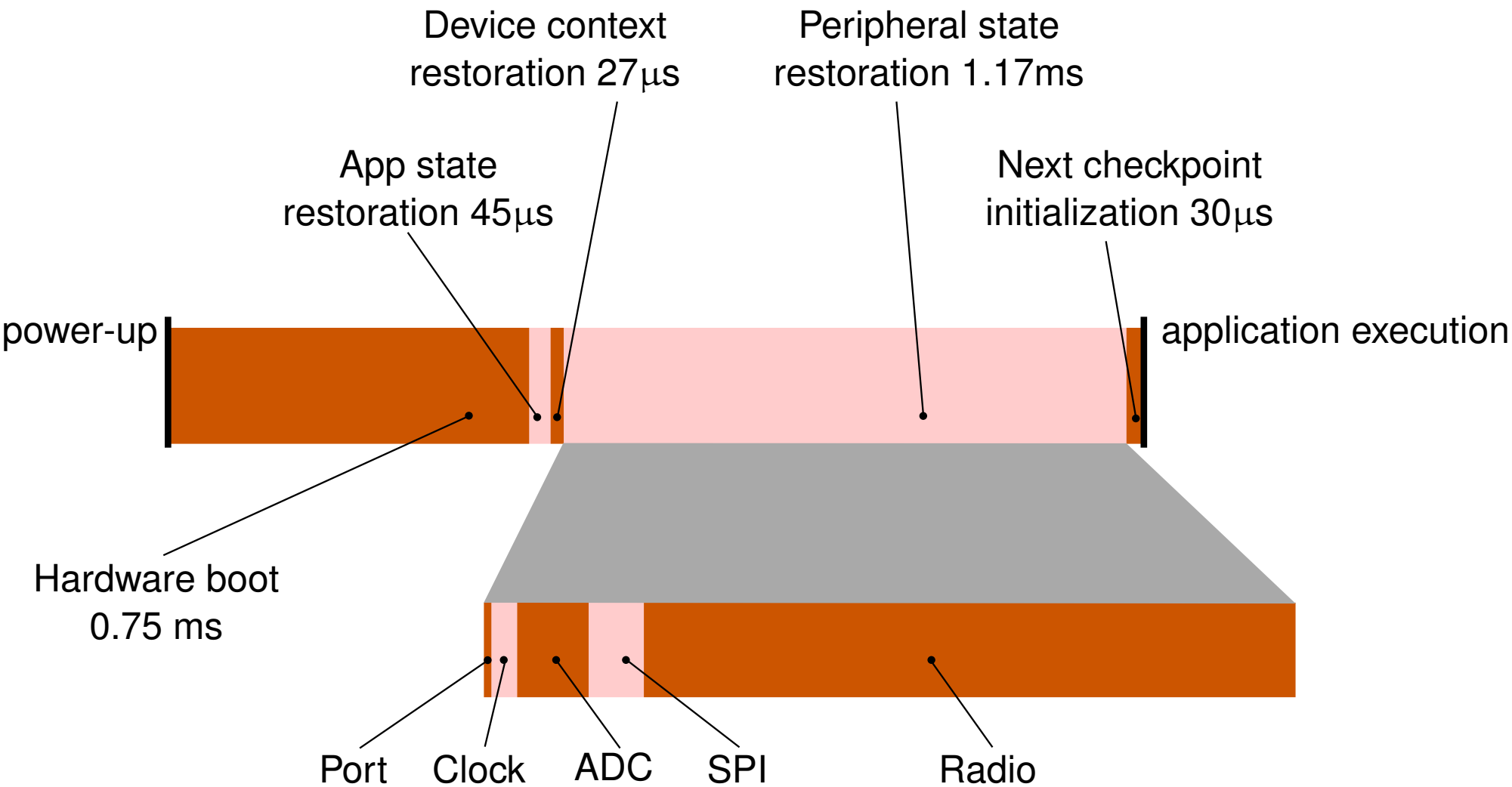
Conclusion: parallel with high-end systems

- Programming model impacted
  - in multi-core processors: crashes are expected
  - example: transactions / lock
- Memory hierarchy must be chosen
  - Other choices
- Peripherals'state persistence
  - Very few works
- Not studied
  - Explicit placement
  - Security issues

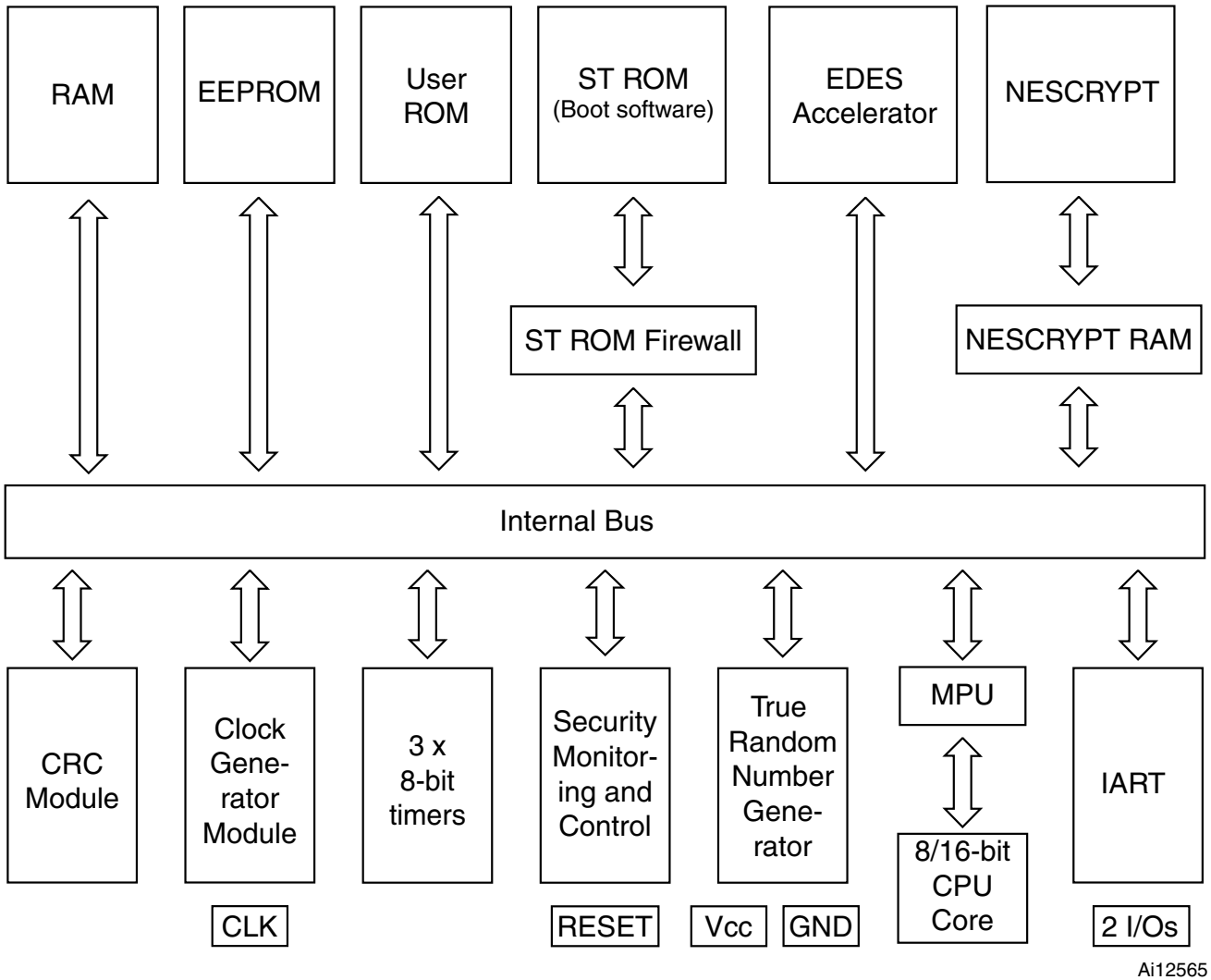
Thanks

Questions ?

System boot sequence



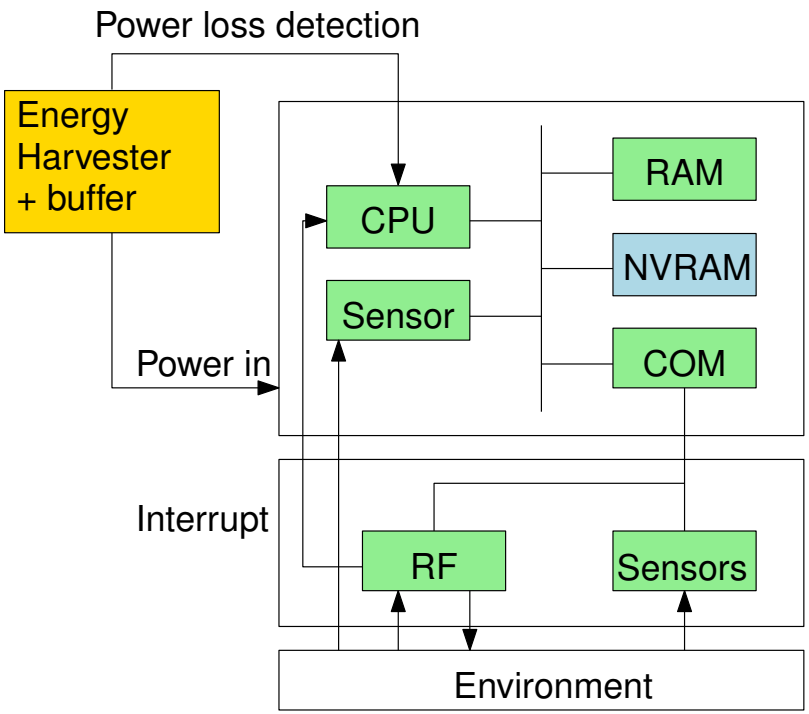
TPS example architecture : ST23ZL48 microcontroller



- 16-bits CPU (27MHz)
- 8kB RAM
- 300kB ROM
- 48kB EEPROM

`http://www.st.com/en/secure-mcus/st23zl48.html`

Pb: interrupts are volatile as well



- Embedded applications always use interrupts.
- User wants control over interrupt handling.
- Interrupts carry volatile data that will be lost upon power loss.
- Interrupt occurrence and data must be persisted.

Top halves and bottom halves

Interrupt handling is split into kernel-managed top halves and user-managed bottom halves.

- Enables hiding power loss from user, with power loss being handled by top halves.

Design choices: two axes

1. Bottom half nestedness? No
2. Allowance of hardware operations being called from bottom halves ? Yes

Interrupt-related problems

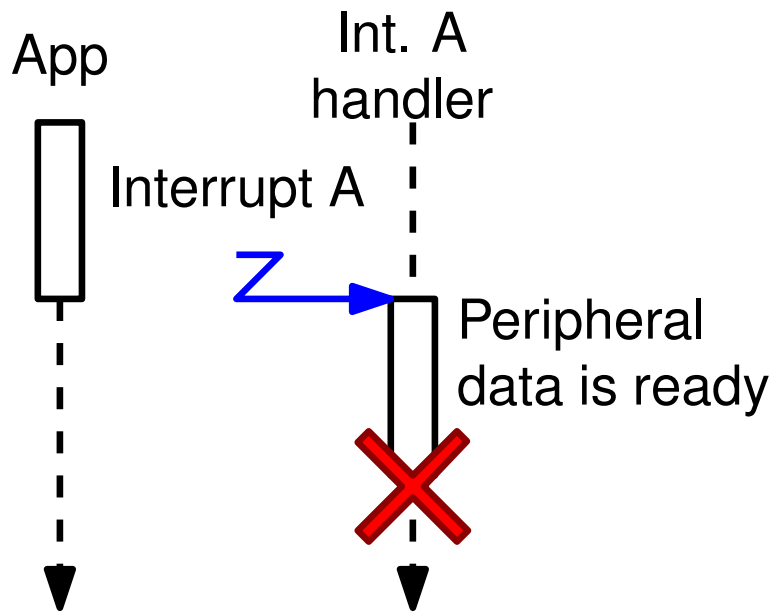
Problems not specific to transiently-powered systems:

- Data race conditions between user application and bottom halves.
- Peripheral access race conditions.
- Peripheral access atomicity for other interrupts than power loss detection interrupt.

Specific to transiently-powered systems:

- Interrupt data volatility.

Interrupt data volatility



Interrupt data volatility

What must be persisted

- OS scheduler data: pending bottom halves.
- User bottom half vector.
- Peripheral-specific data carried by interrupt: part of the device context.

Solution

- All top halves start with extracting peripheral-specific data to store them in the corresponding device context.
- All top halves then schedule their bottom halves.
- The device contexts, scheduler data and bottom half vector are part of the checkpointing image, which makes them persistent.

Data race conditions

Application code

```
static int x;

bottom_half() { ++x; }

main()
{
    x = 0;
    ...
    if(x == 4) {
        ...
    }
}
```

Data race conditions

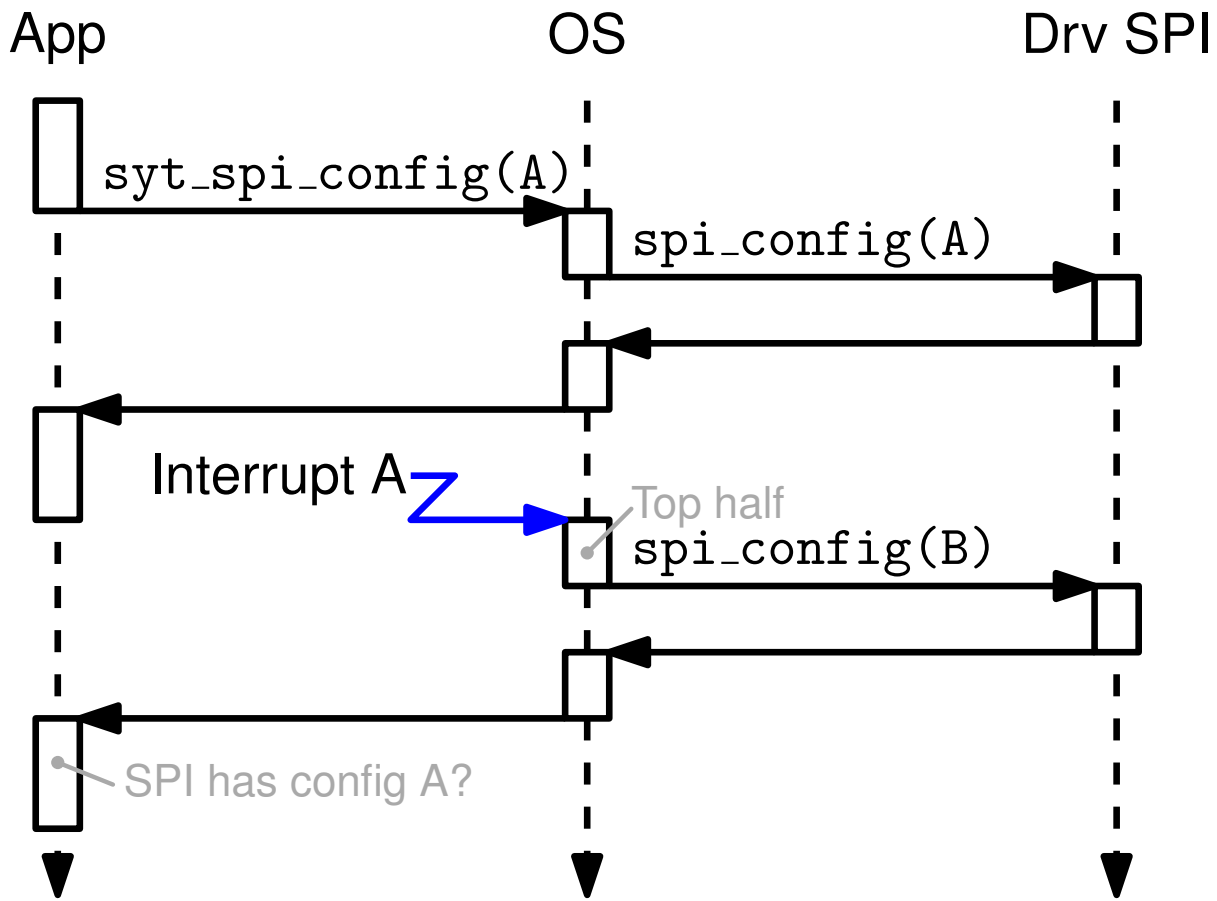
Problem

- Bottom halves may share data with user application: global variables.

Solution

- User-defined critical sections that disable bottom halves but keep interrupts enabled.
- Bottom halves are delayed until the end of the critical section.
- Interrupts are enabled, which makes the sytem reactive upon power loss detection.

Peripheral access race conditions



Peripheral access race conditions

Problem

- Interrupts might occur during a hardware access initiated by user.
- Top half might use the same peripheral and put it in an inconsistent state with respect to the application.

Solution

- Provide lock mechanism, accessible from the user, who indicates to the kernel which peripherals are locked.
- When an interrupt occurs, if the top half tries to use a locked peripheral, both the top and bottom halves are discarded.

Peripheral access atomicity

Problem

- Interrupts might occur during a hardware access initiated by user.

Solution

- Rerun syscall from the beginning when returning from interrupt.
- Makes the syscall management policy consistent with power loss detection occurring during syscall execution.
- Pessimistic approach that leads to time and energy overhead.

Operating System architecture - Completed