Synchronous Programming of Device Drivers for Global Resource Control in Embedded Systems

Nicolas BERTHIER

UNIVERSITÉ DE GRENOBLE

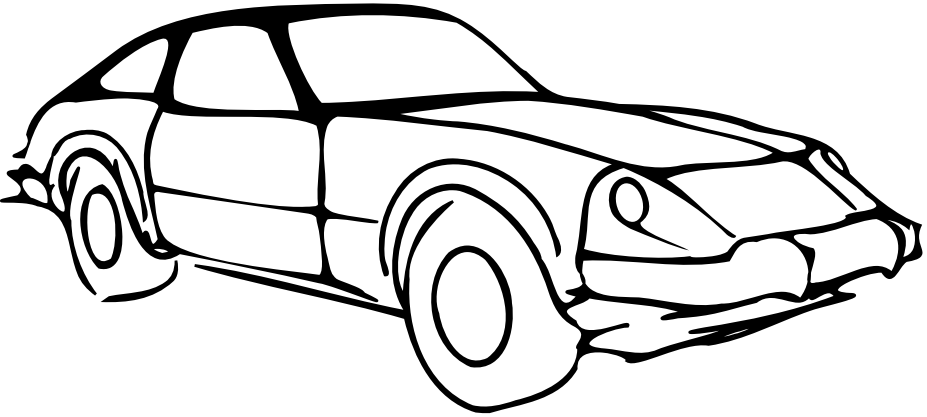
Verimag

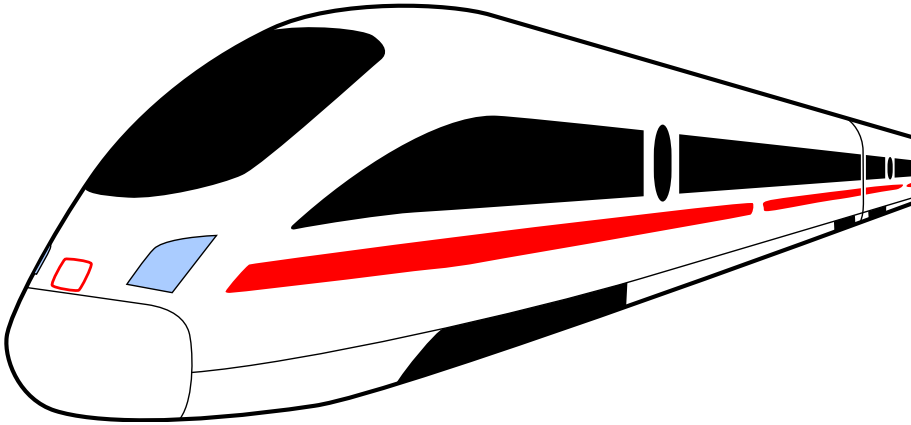
March 12, 2012

Jury

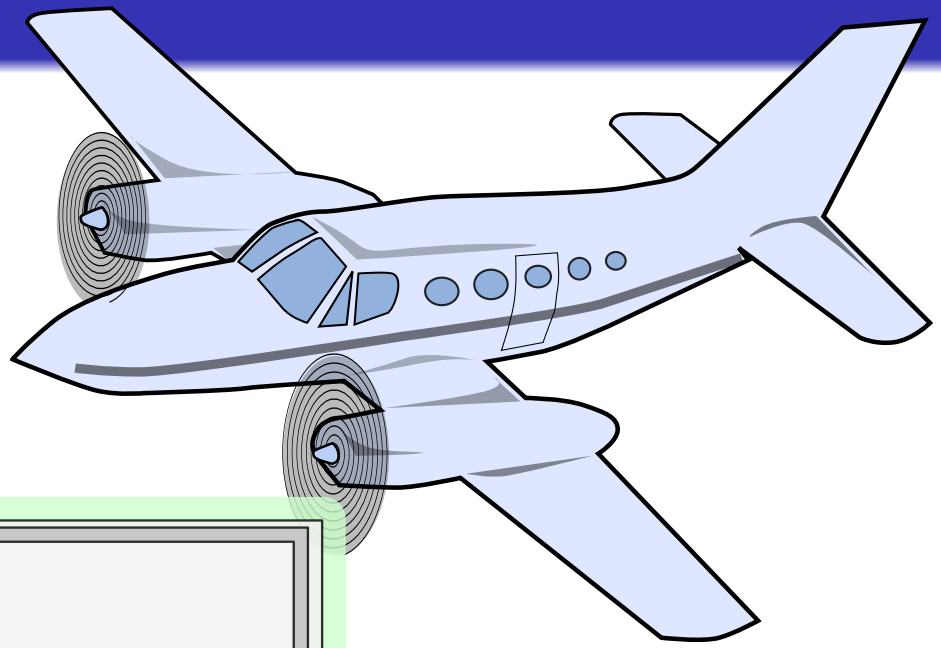
Gilles MULLER	INRIA	Reviewer
Éric FLEURY	ENS Lyon	Reviewer
Antoine FRABOULET	INSA Lyon	Examiner
Abdoulaye GAMATIÉ	CNRS	Examiner
Florence MARANINCHI	Grenoble INP	Advisor
Laurent MOUNIER	UJF	Advisor

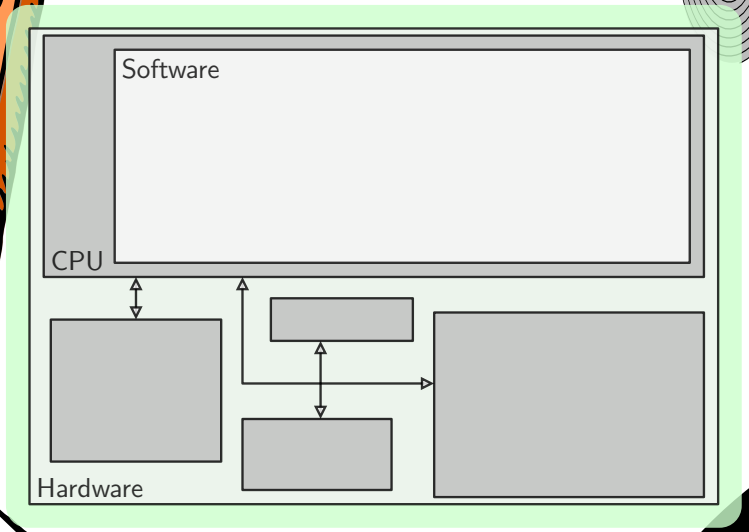
Embedded Systems

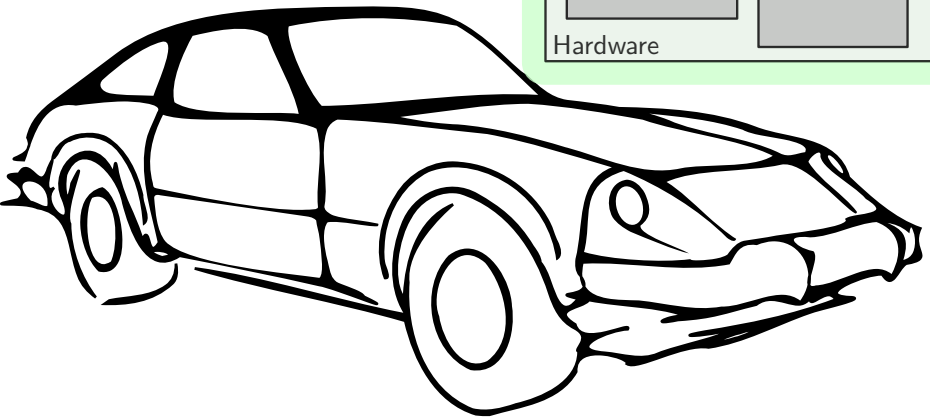


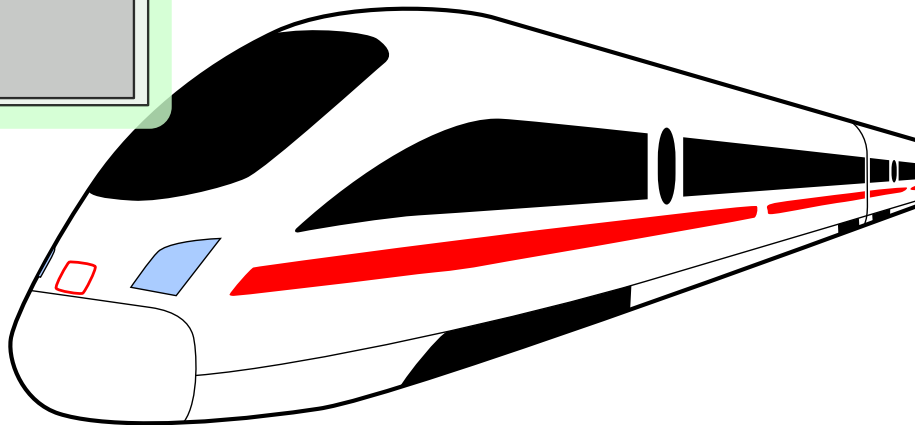


Embedded Systems

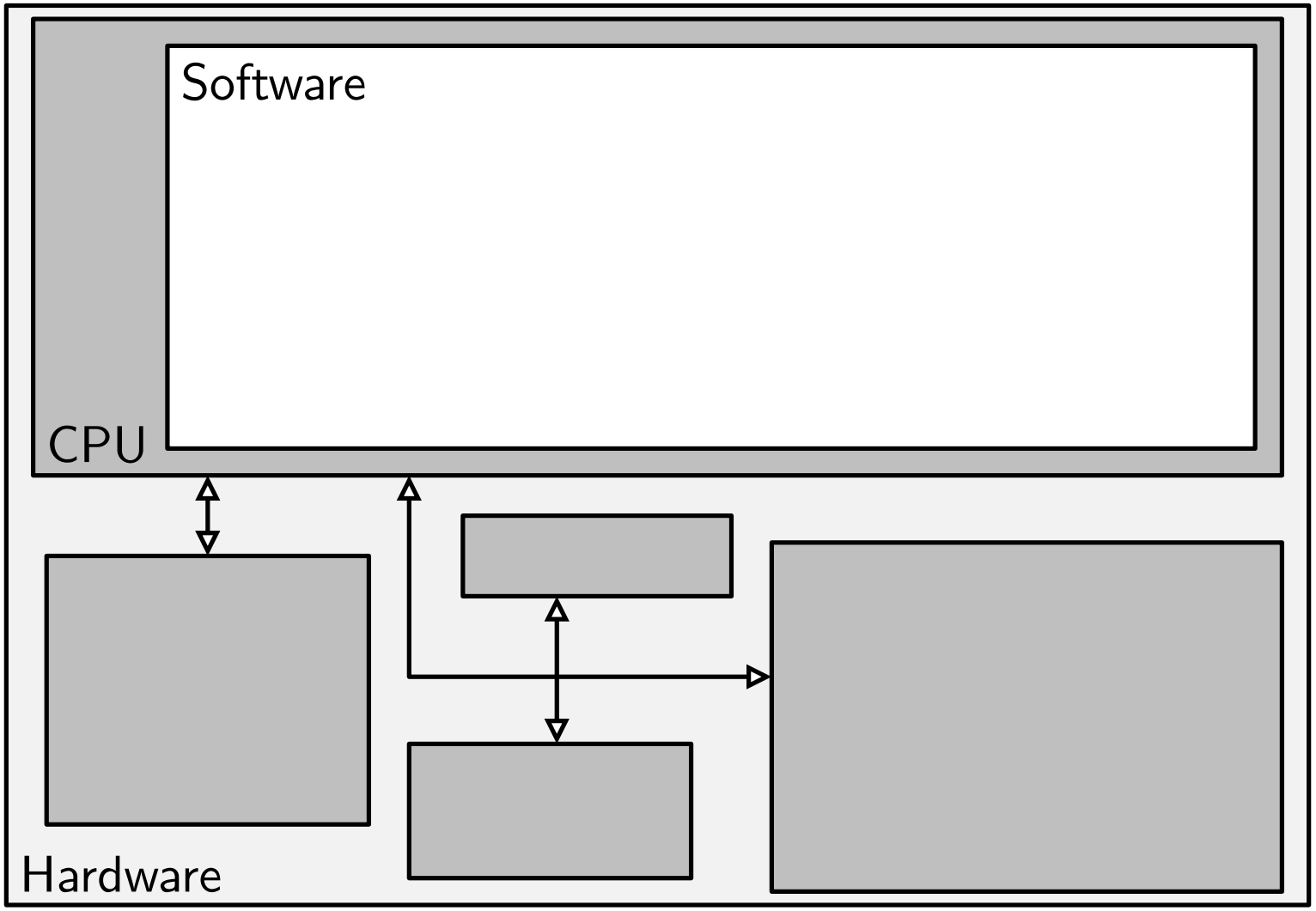




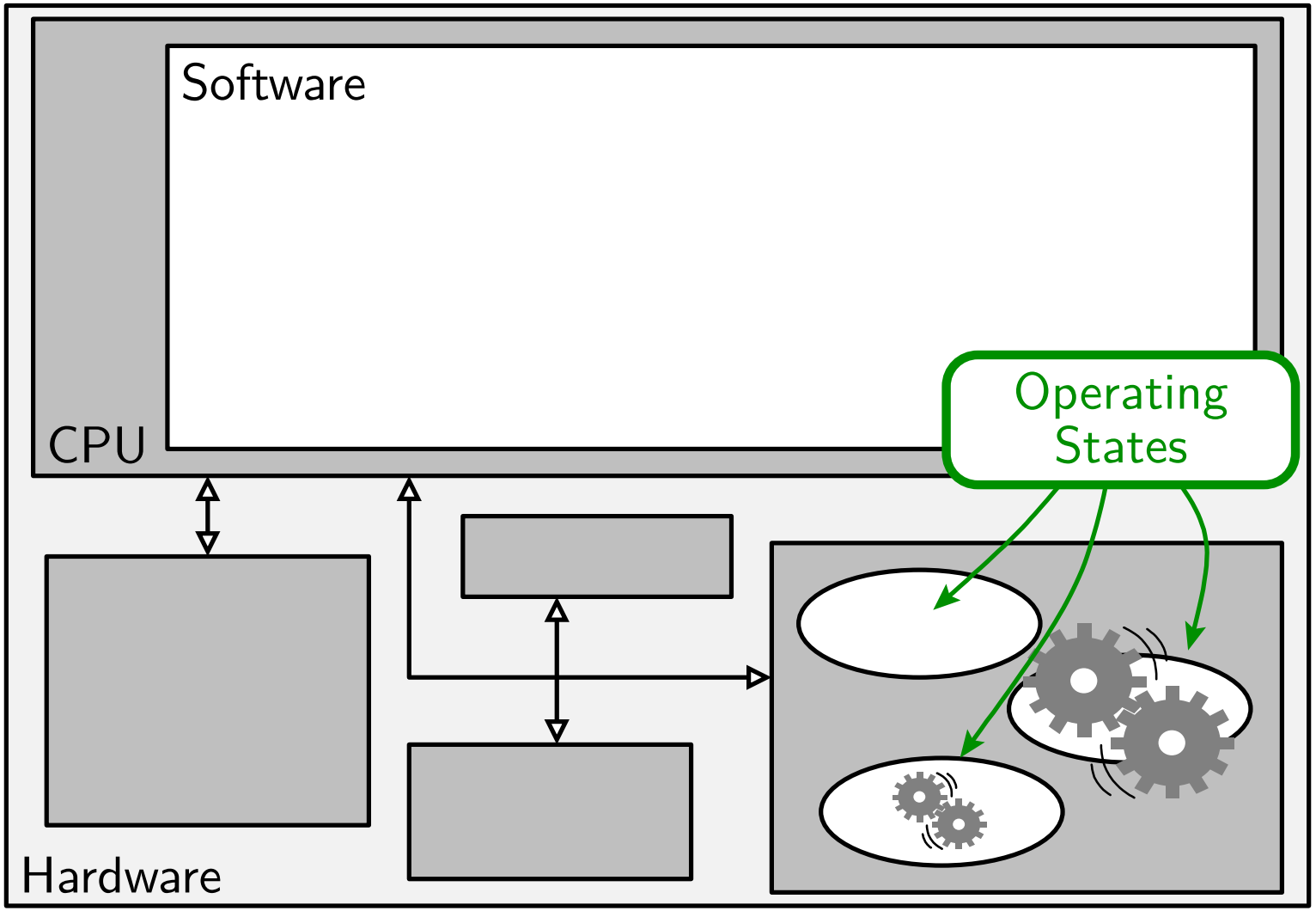




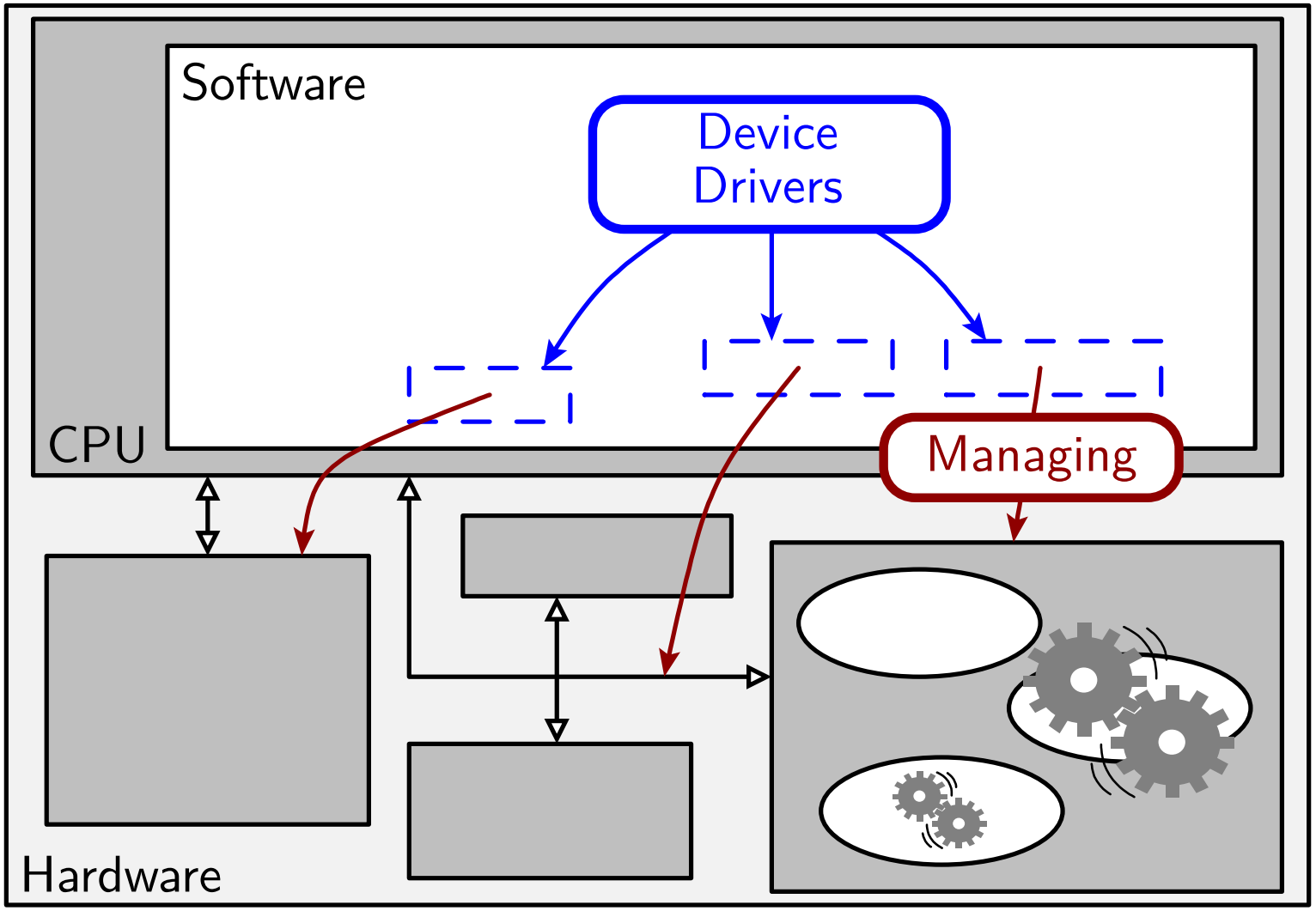
Resources & Device Drivers



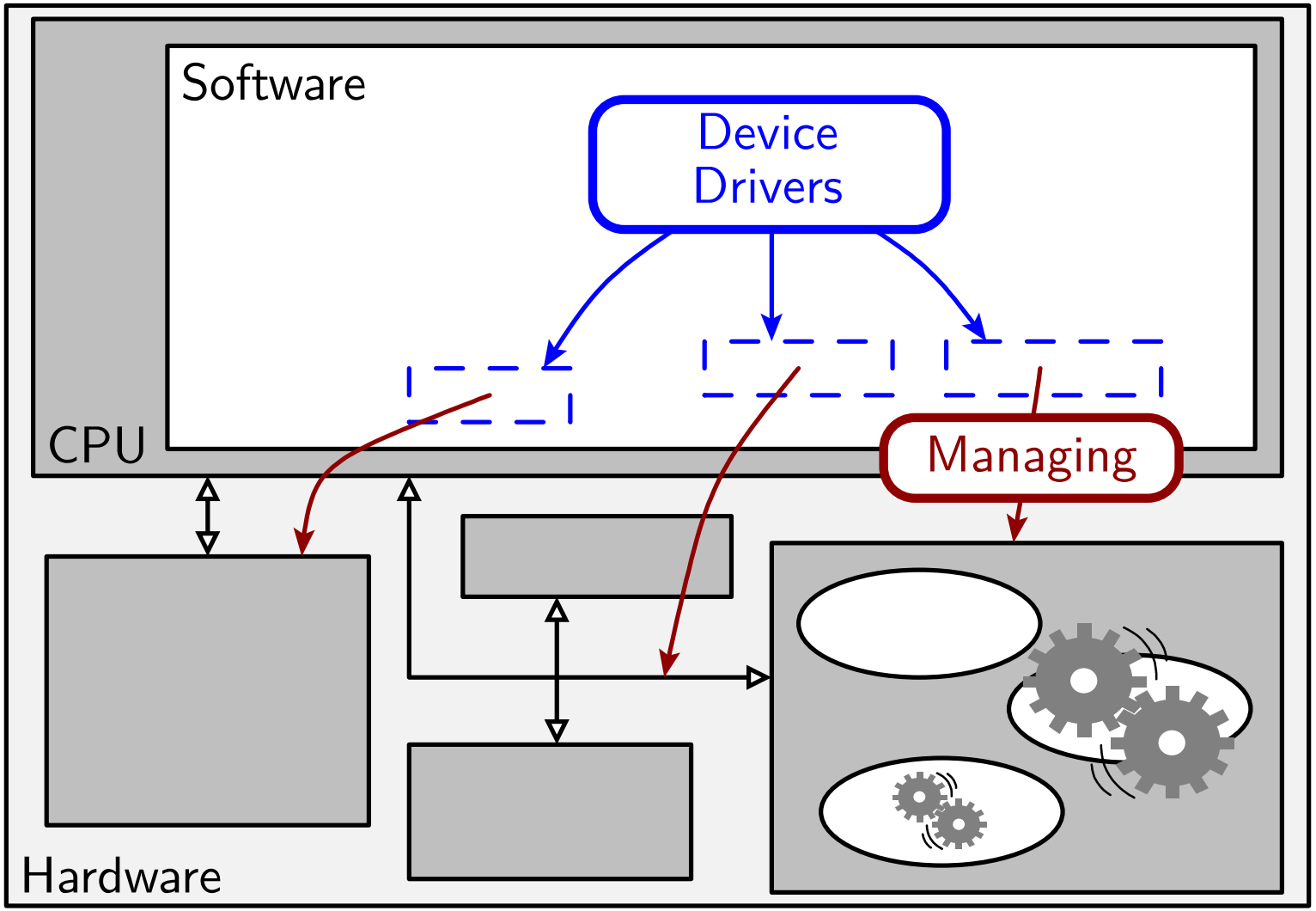
Resources & Device Drivers



Resources & Device Drivers

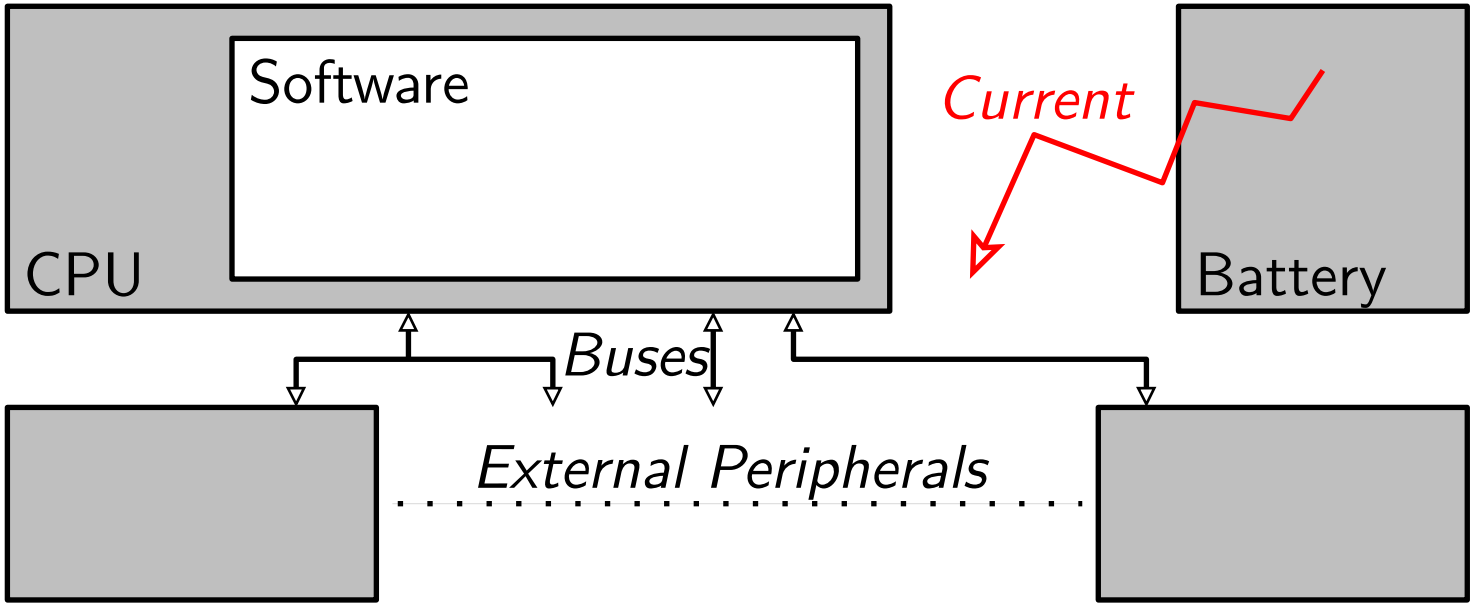


Resources & Device Drivers

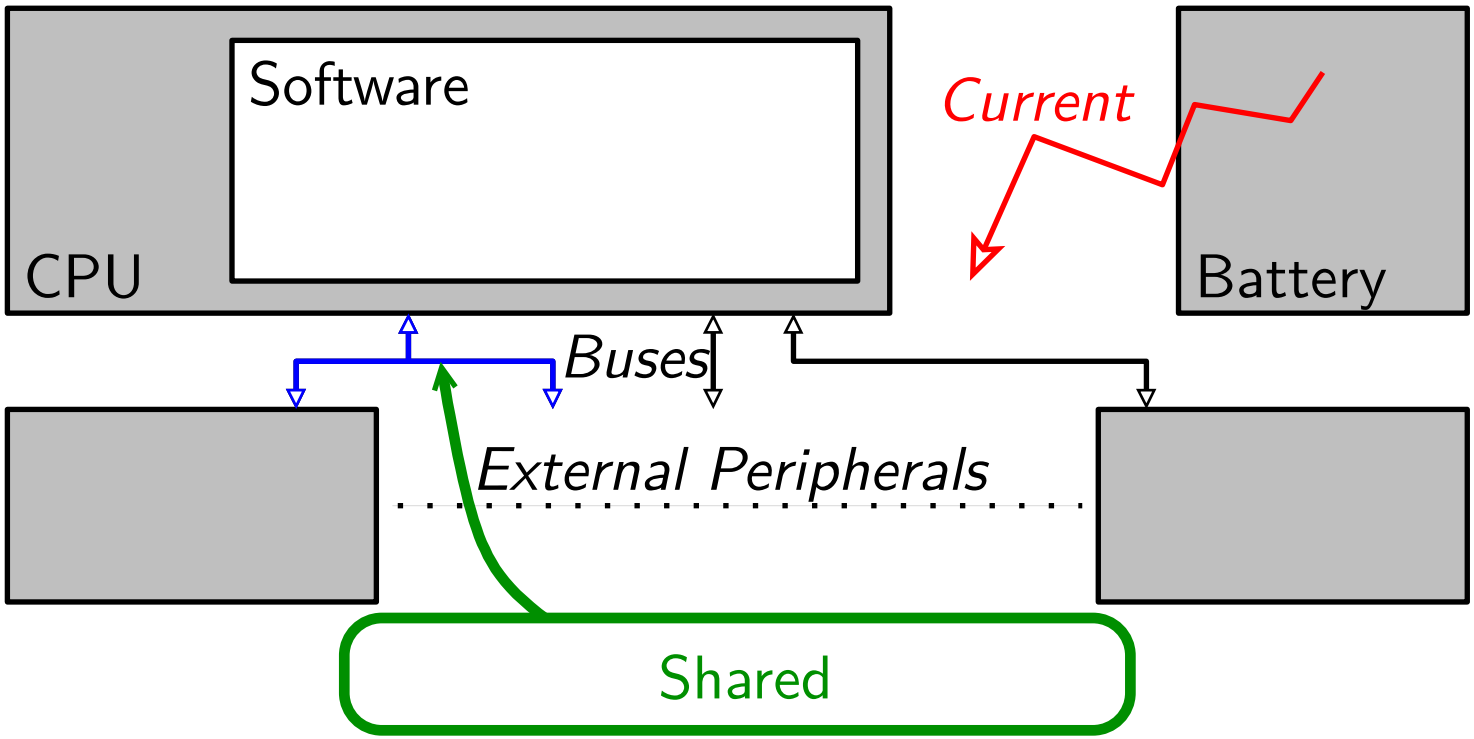


Local Control of Peripherals: One Device $\longleftrightarrow$ One Driver

Need for Global Resource Control



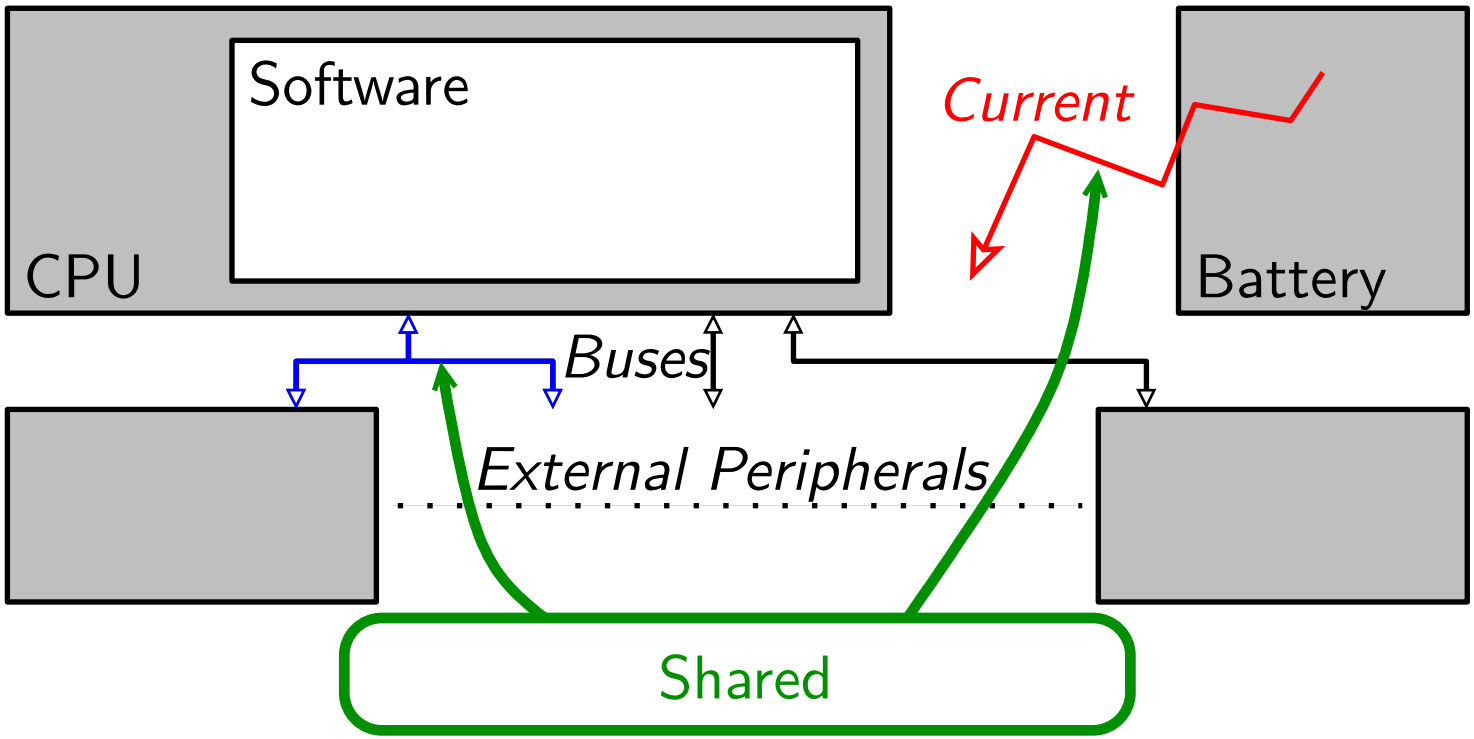
Need for Global Resource Control



Avoiding

- Conflicting Accesses to Shared Buses

Need for Global Resource Control



Avoiding

- Conflicting Accesses to Shared Buses
- Consumption Peaks

Proposal for Implementing Global Control

New Software Architecture

- Operating at the Level of Device Drivers
- Design Based on Synchronous Programming Principles

Proposal for Implementing Global Control

New Software Architecture

- Operating at the Level of Device Drivers
- Design Based on Synchronous Programming Principles

Choosing an Application Domain

- Proof-of-Concept Implementation

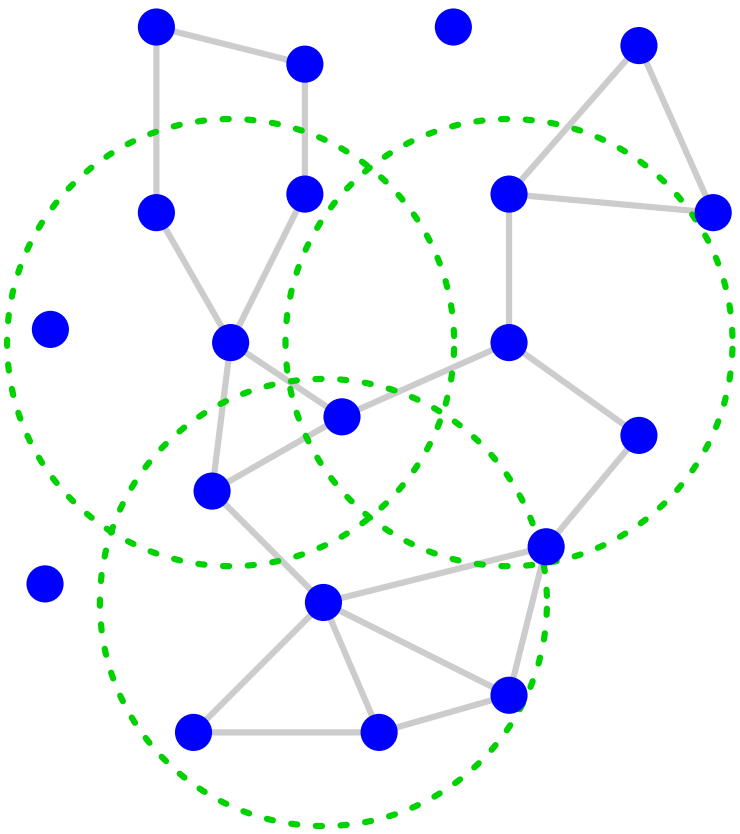
Outline

- Choosing an Application Domain and Outline of the Proposal
- Background
- Detailed Contribution
- Conclusion

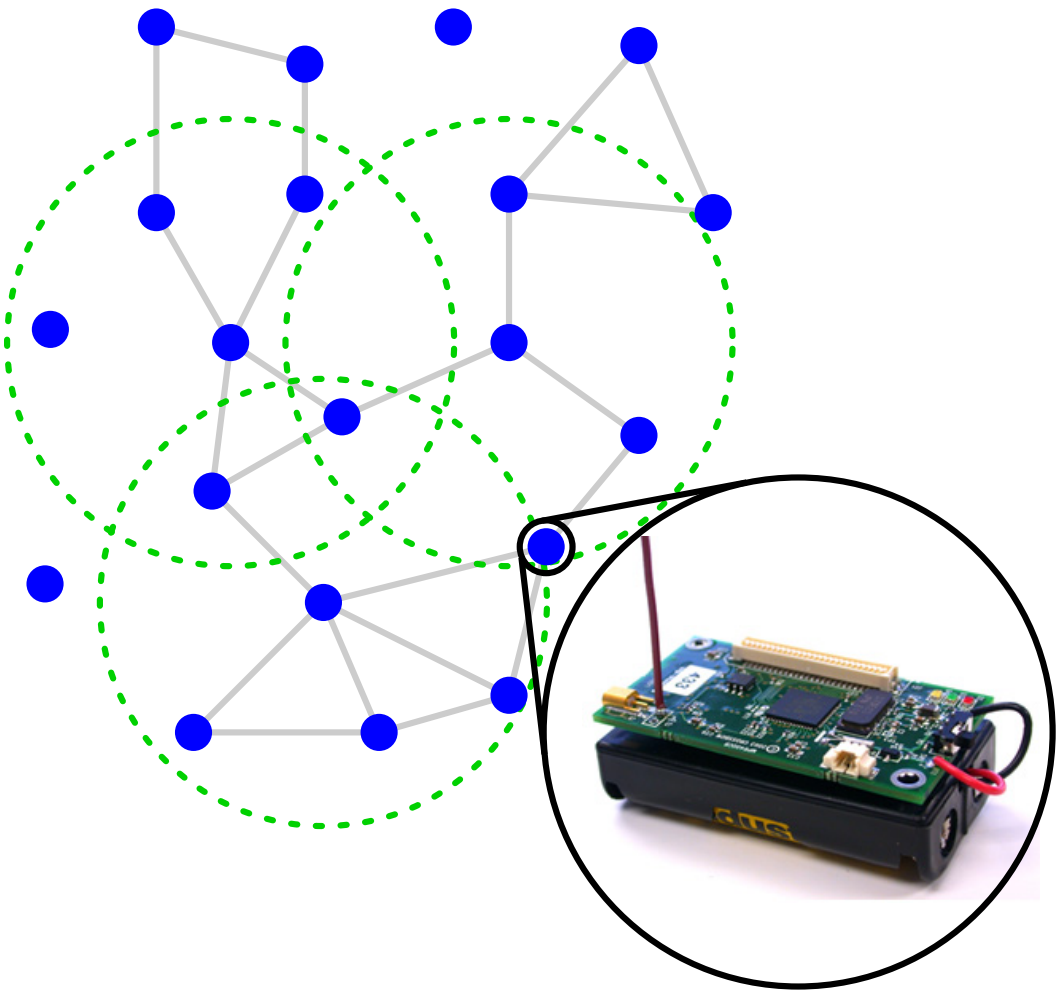
Outline

- Choosing an Application Domain and Outline of the Proposal
  - Wireless Sensor Nodes
  - Related Work
  - Constraints
  - Idea of the Solution
- Background
- Detailed Contribution
- Conclusion

Choosing an Application Domain: Wireless Sensor Nodes



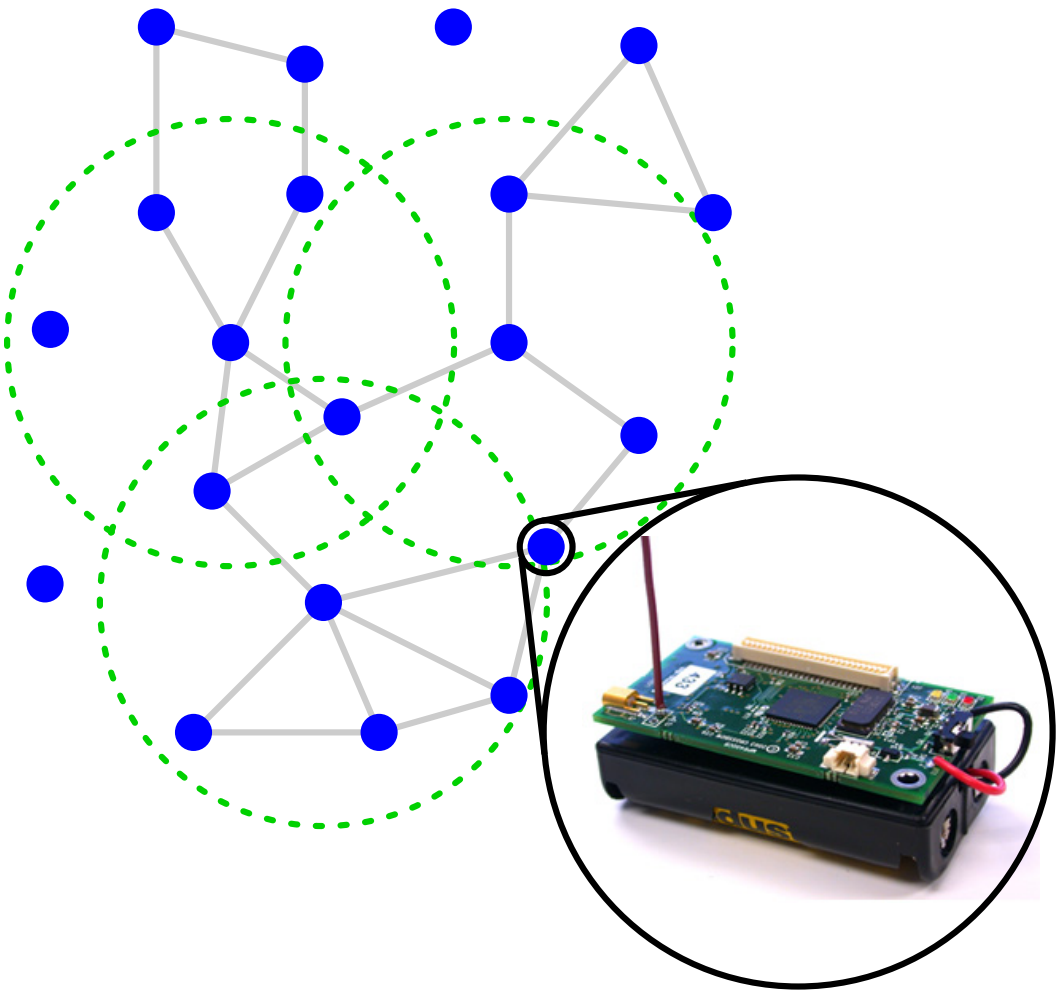
Choosing an Application Domain: Wireless Sensor Nodes



Components

- µ-Controller (µ-C)
- Radio Transceiver(s)
- Sensors
- Battery
- ...

Choosing an Application Domain: Wireless Sensor Nodes



Components

- µ-Controller (µ-C)
- Radio Transceiver(s)
- Sensors
- Battery
- ...

Constraints

- Slow Computations
- Low Amount of Memory
- Limited Energy
- ...

Choosing an Application Domain: Wireless Sensor Nodes

Interesting Characteristics

- Not Too Complex But Representative

Manageable Hardware/Software Complexity

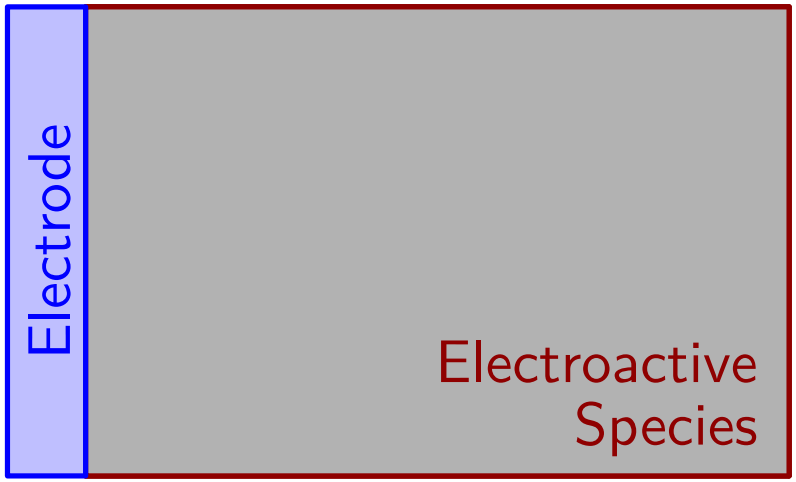
- Relatively Simple Hardware. . .
  - Compared to, *e.g.* a Multimedia System-on-Chip
- . . . But Not Too Much
  - Complex Peripheral Devices
  - Task Parallelism *e.g.*, Network Management, Application(s)

Generalizable Solution

- Generic Hardware

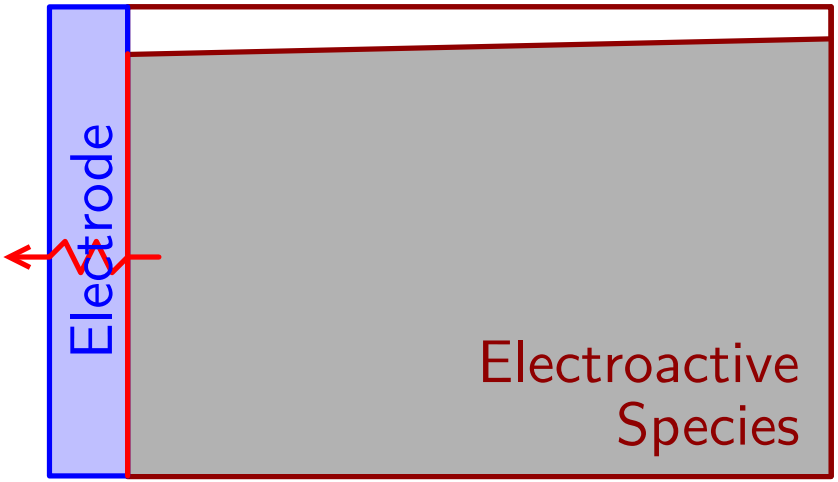
Peculiar Battery Behaviors and Consequences on Software

Fully Charged Battery



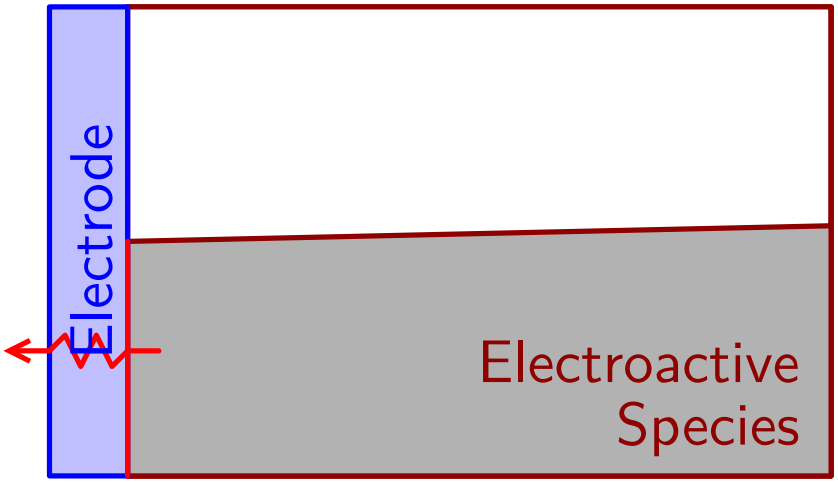
Peculiar Battery Behaviors and Consequences on Software

Slowly Discharging Battery



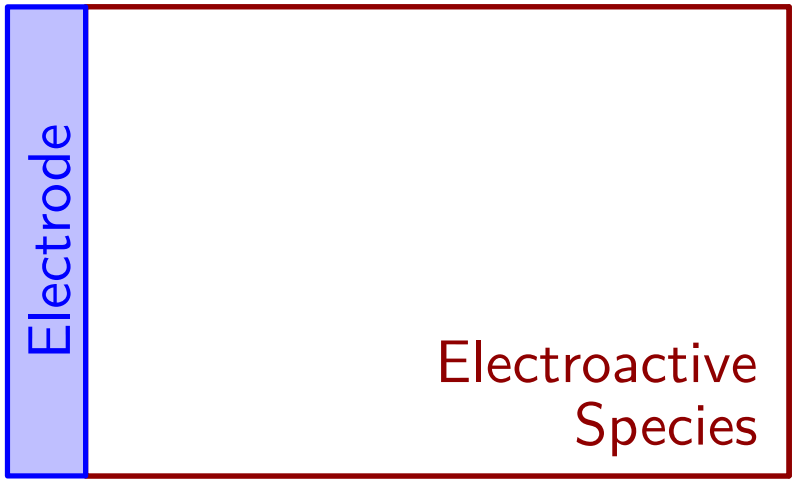
Peculiar Battery Behaviors and Consequences on Software

Slowly Discharging Battery



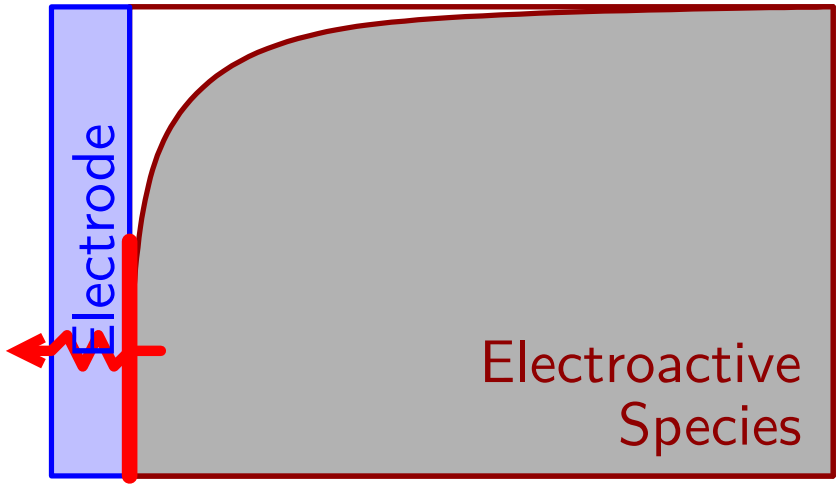
Peculiar Battery Behaviors and Consequences on Software

Fully Discharged Battery



Peculiar Battery Behaviors and Consequences on Software

Quickly Discharging Battery

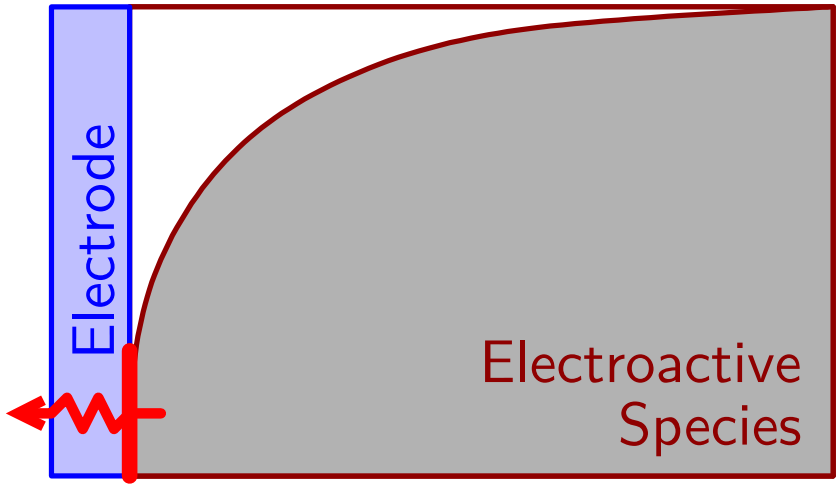


Rate Capacity Effect & Resistance Increase

- Software *Should* Avoid *Power Consumption Peaks*

Peculiar Battery Behaviors and Consequences on Software

Quickly Discharging Battery

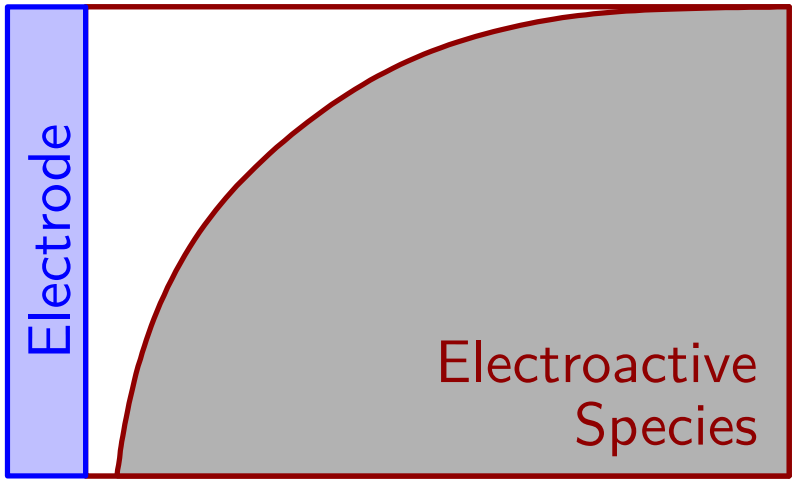


Rate Capacity Effect & Resistance Increase

- Software *Should* Avoid *Power Consumption Peaks*

Peculiar Battery Behaviors and Consequences on Software

Apparently Discharged Battery

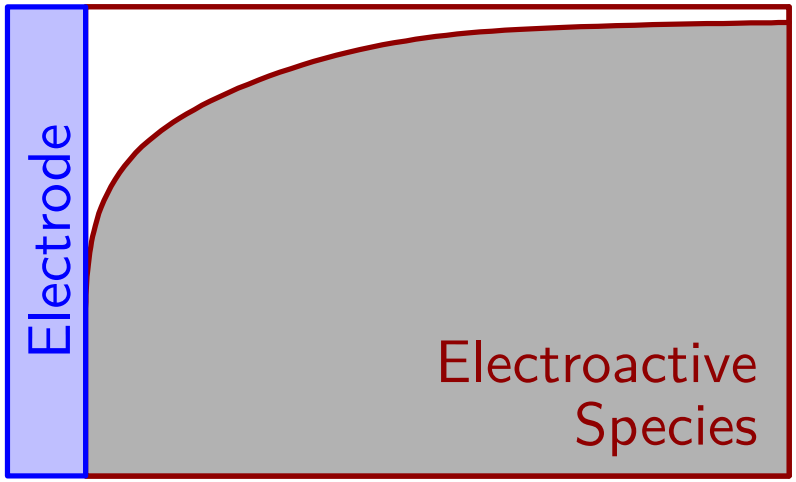


Rate Capacity Effect & Resistance Increase

- Software *Should* Avoid *Power Consumption Peaks*

Peculiar Battery Behaviors and Consequences on Software

Recovery After Idle Time

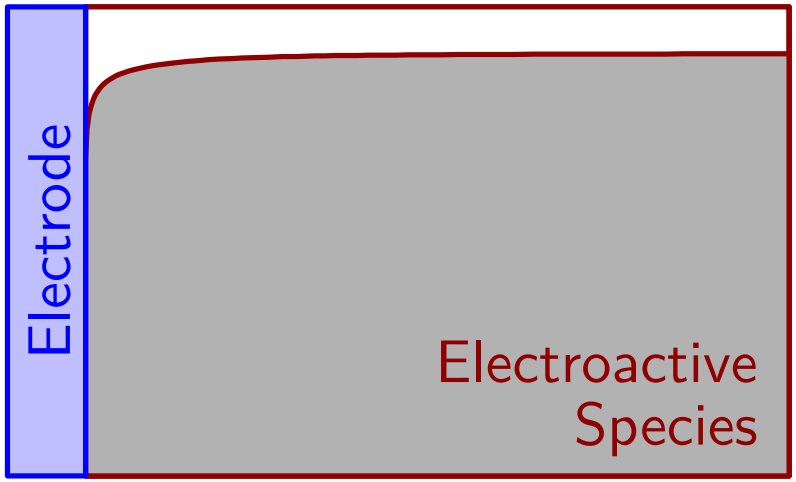


Rate Capacity Effect & Resistance Increase

- Software *Should* Avoid *Power Consumption Peaks*

Peculiar Battery Behaviors and Consequences on Software

Recovery After Idle Time



Rate Capacity Effect & Resistance Increase

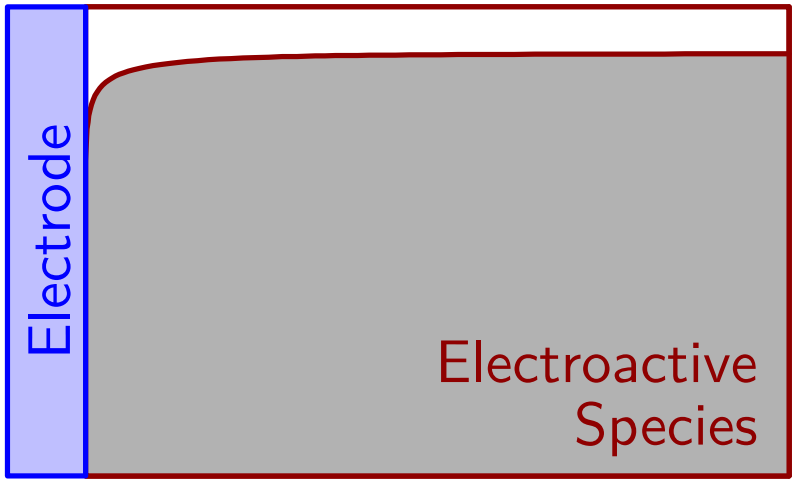
- Software *Should* Avoid *Power Consumption Peaks*

Recovery Effect

- Software *Should* Tune Pauses According to *Power Consumption Profile*

Peculiar Battery Behaviors and Consequences on Software

Recovery After Idle Time



Rate Capacity Effect & Resistance Increase

- Software *Should* Avoid *Power Consumption Peaks*

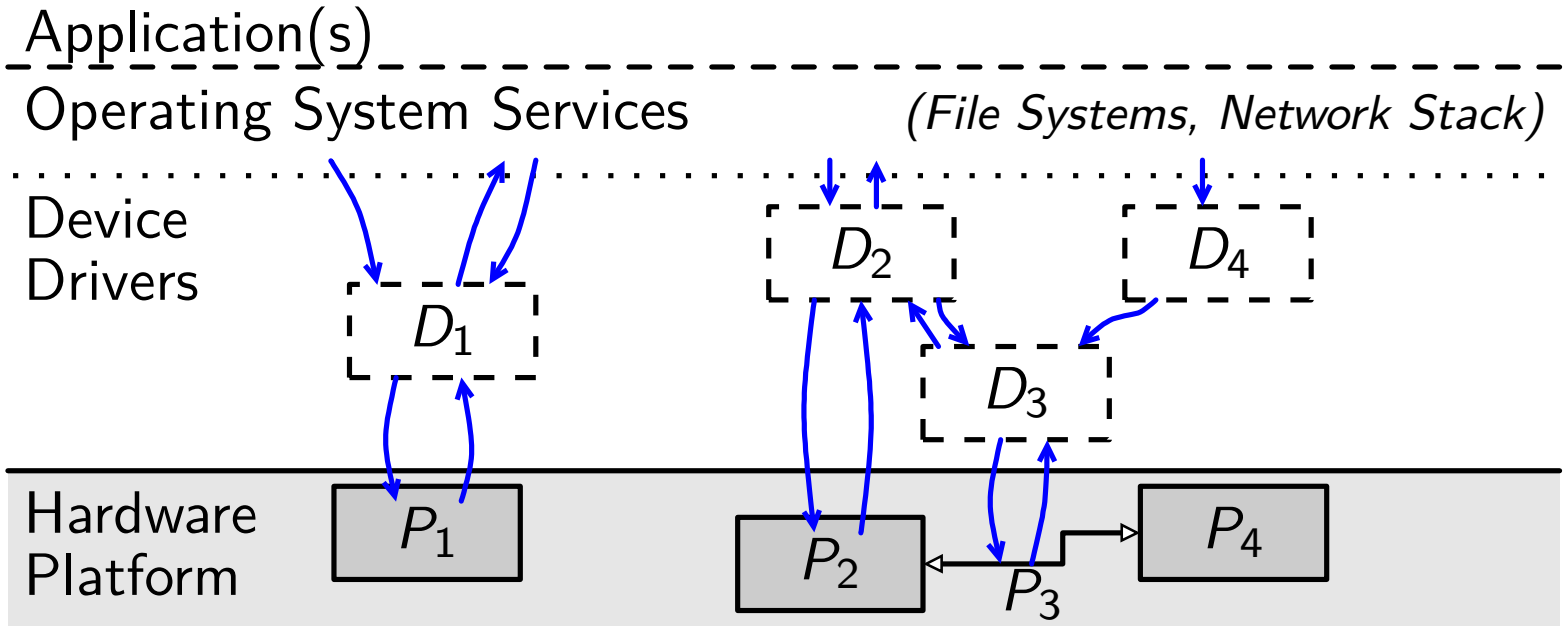
Recovery Effect

- Software *Should* Tune Pauses According to *Power Consumption Profile*

$\rightsquigarrow$ Complexity of the Software

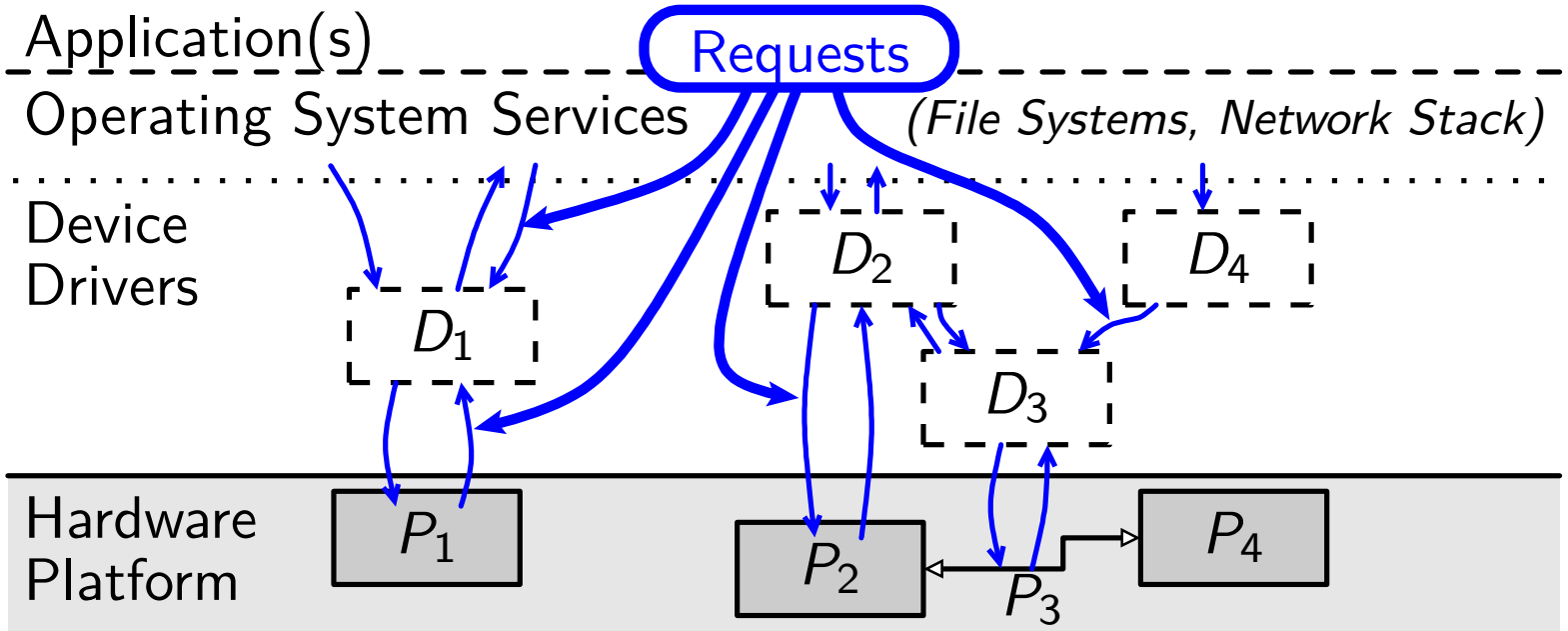
Usual Programming Practice

(with an Operating System)



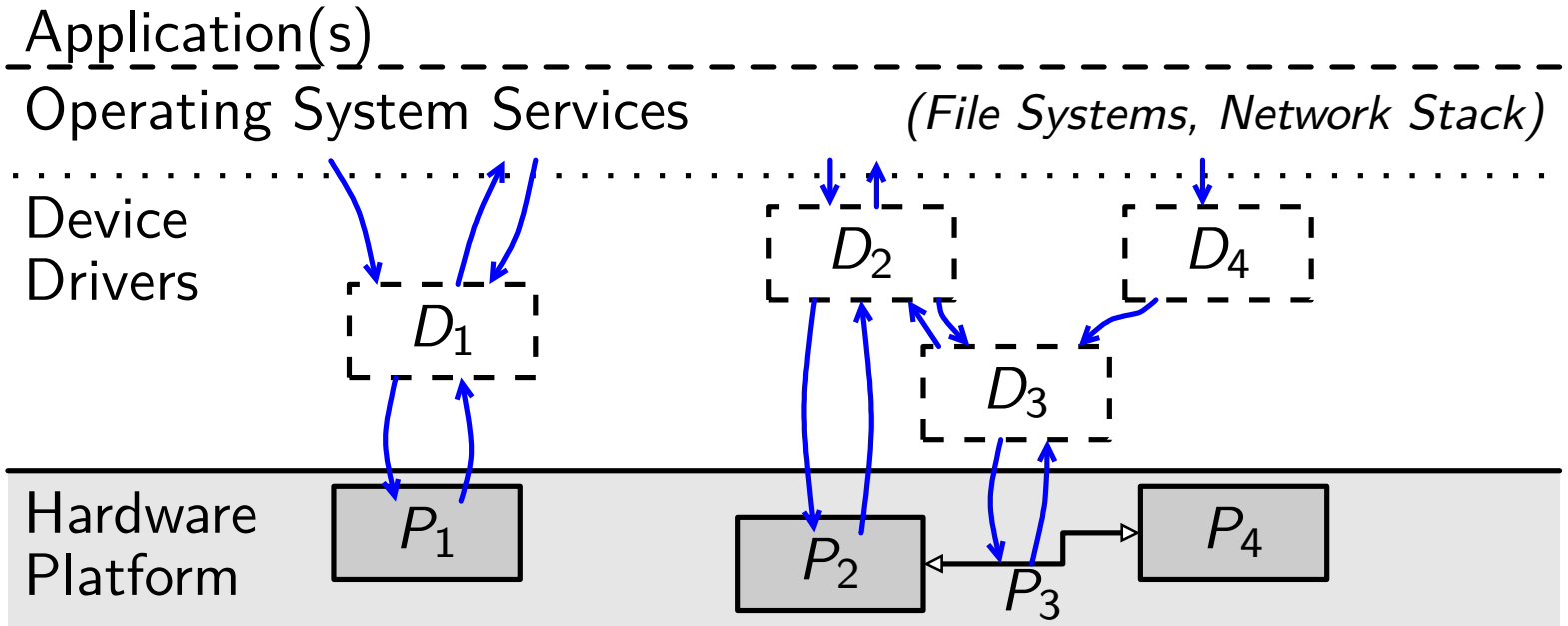
Usual Programming Practice

(with an Operating System)



Usual Programming Practice

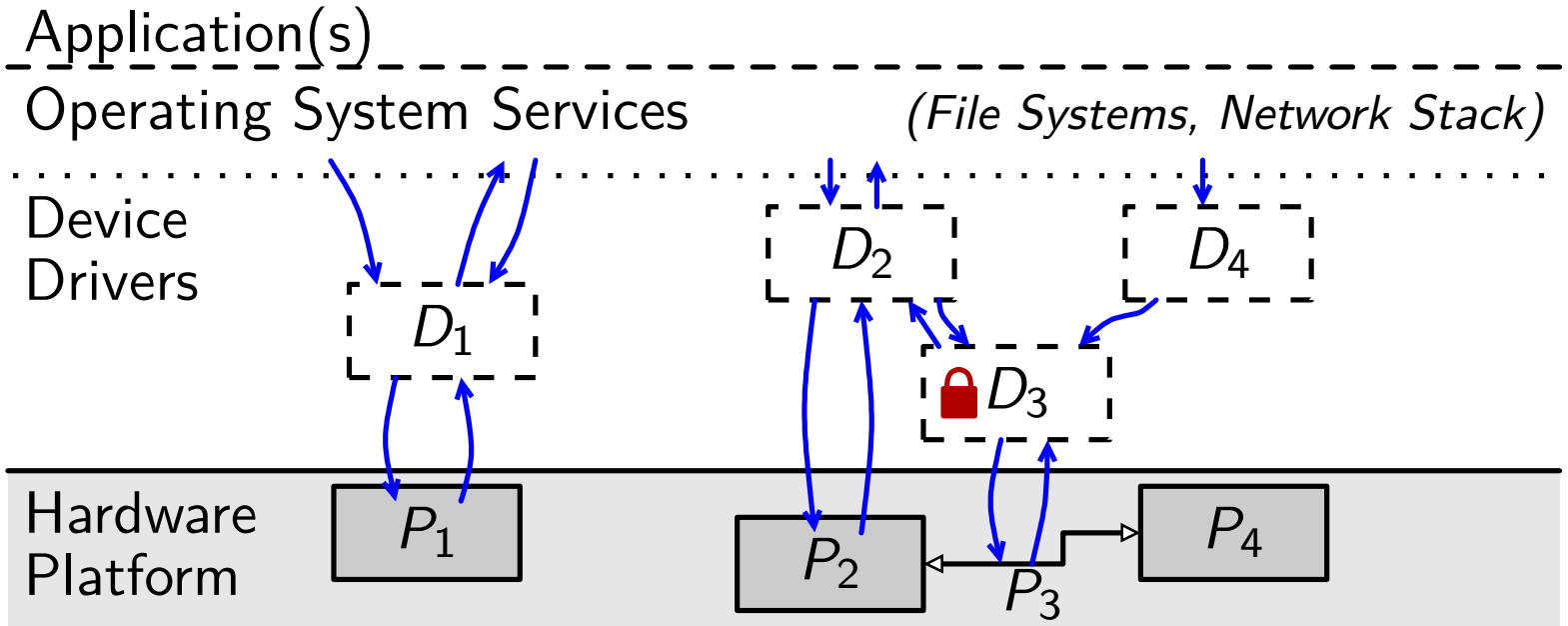
(with an Operating System)



- Device Drivers Designed Independently of Each Other
- Controlling Accesses to Shared Peripherals (*e.g.*, P_3)
  - Left to the Upper Layers

Usual Programming Practice

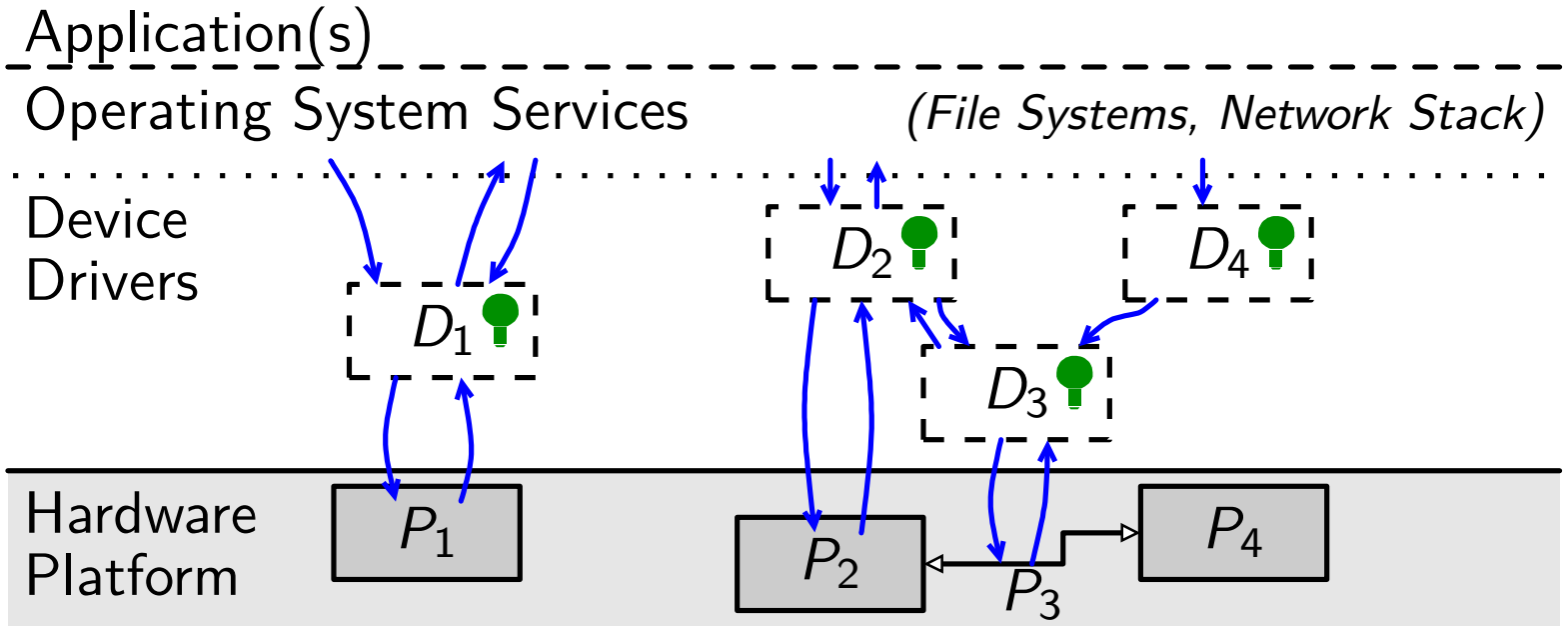
(with an Operating System)



- Device Drivers Designed Independently of Each Other
- Controlling Accesses to Shared Peripherals (*e.g.*, P_3)
  - Left to the Upper Layers
  - Transparent for the Applications (Locks)

Usual Programming Practice

(with an Operating System)

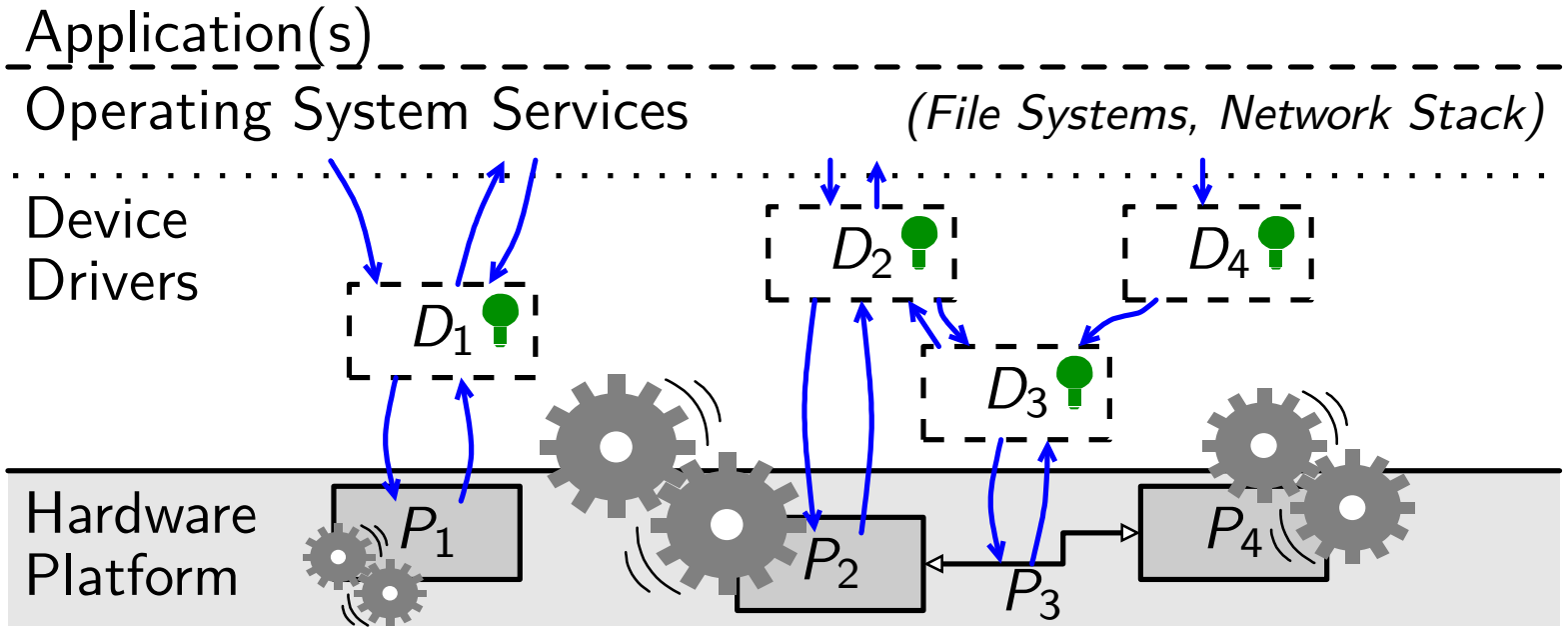


- Device Drivers Designed Independently of Each Other

$\rightsquigarrow$ Decentralized Knowledge on the State of the Hardware!

Usual Programming Practice

(with an Operating System)



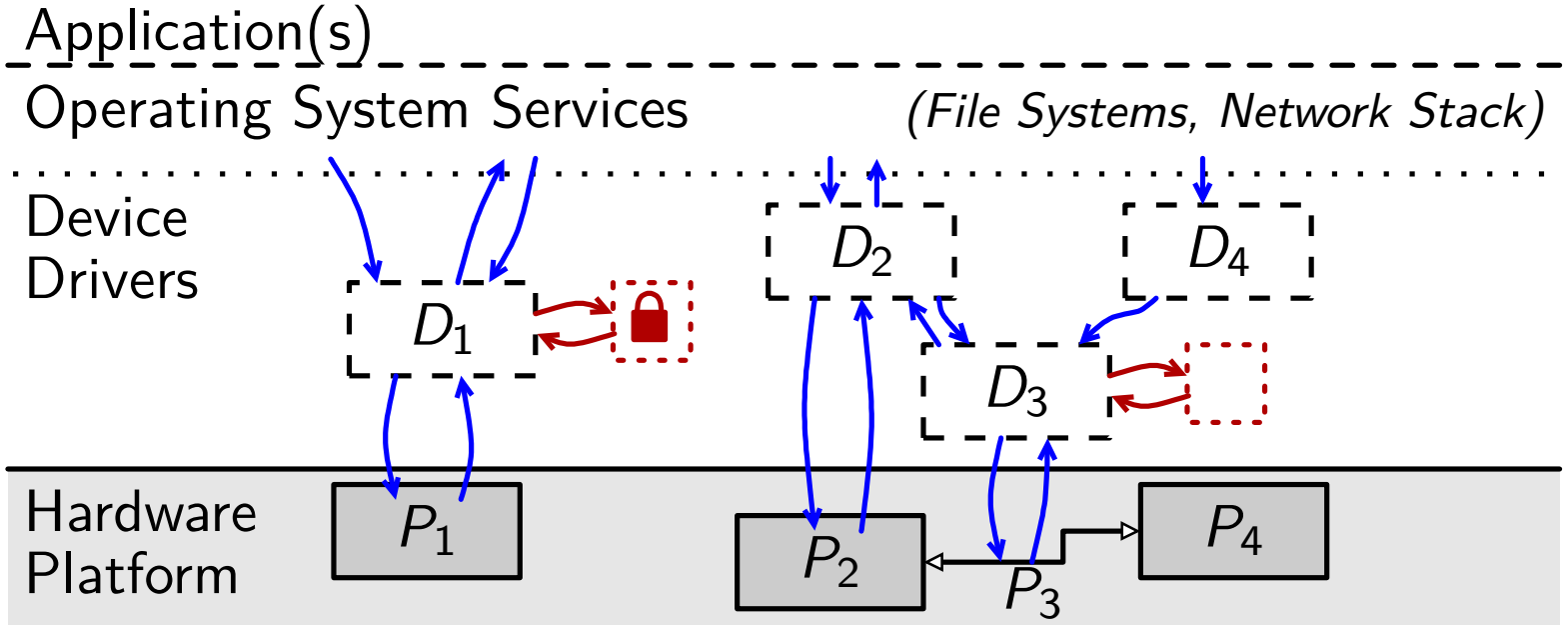
- Device Drivers Designed Independently of Each Other

$\rightsquigarrow$ Decentralized Knowledge on the State of the Hardware!

Avoiding Consumption Peaks?

Towards *Global Control*: ICEM

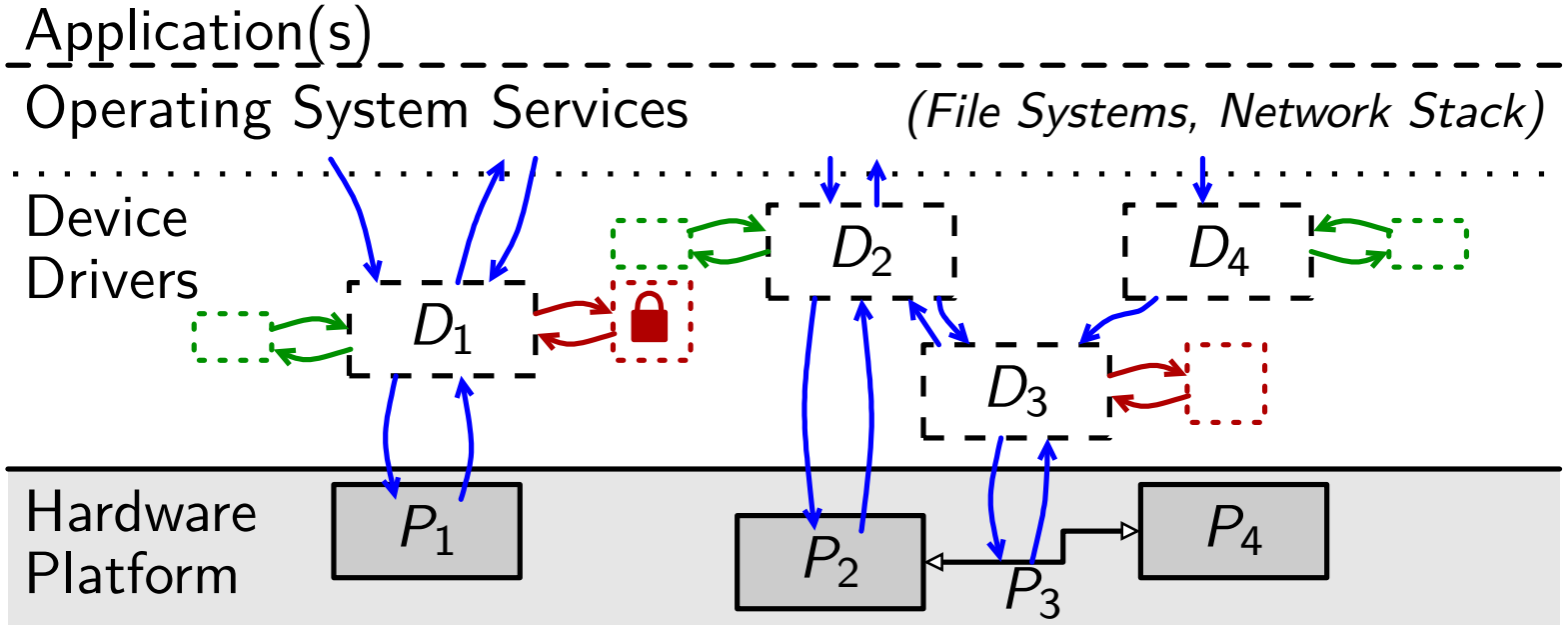
"*Integrating Concurrency Control and Energy Management in Device Drivers*" [Klues *et al.*, 2007]



- Dedicated Software Components
  - Each Arbiter Manages Concurrent Accesses to One Shared Device

Towards *Global Control*: ICEM

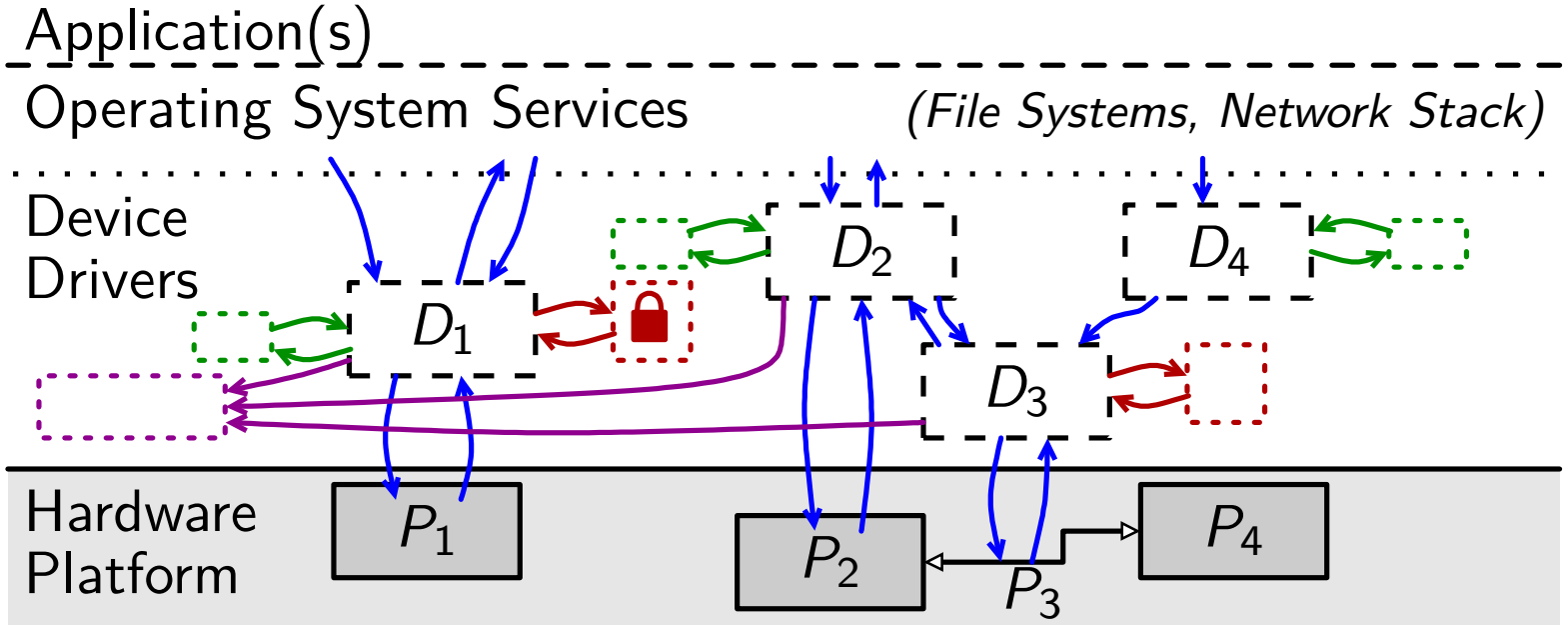
"*Integrating Concurrency Control and Energy Management in Device Drivers*" [Klues *et al.*, 2007]



- Dedicated Software Components
  - Each Arbiter Manages Concurrent Accesses to One Shared Device
  - Each Power Manager Controls the Operating State of One Device

Towards *Global Control*: ICEM

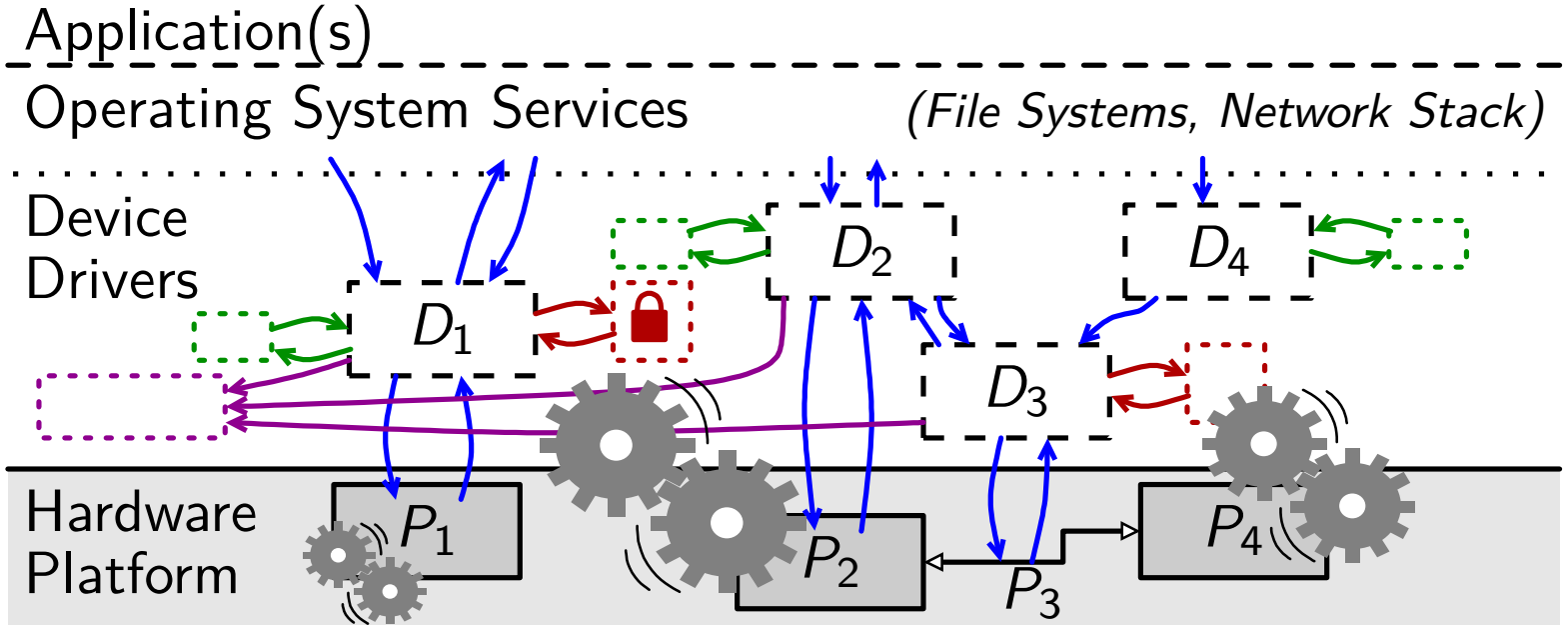
"*Integrating Concurrency Control and Energy Management in Device Drivers*" [Klues *et al.*, 2007]



- Dedicated Software Components
  - Each Arbiter Manages Concurrent Accesses to One Shared Device
  - Each Power Manager Controls the Operating State of One Device
  - One Global *Ad hoc* Component Manages the State of the µ-C

Towards *Global Control*: ICEM

"*Integrating Concurrency Control and Energy Management in Device Drivers*" [Klues *et al.*, 2007]



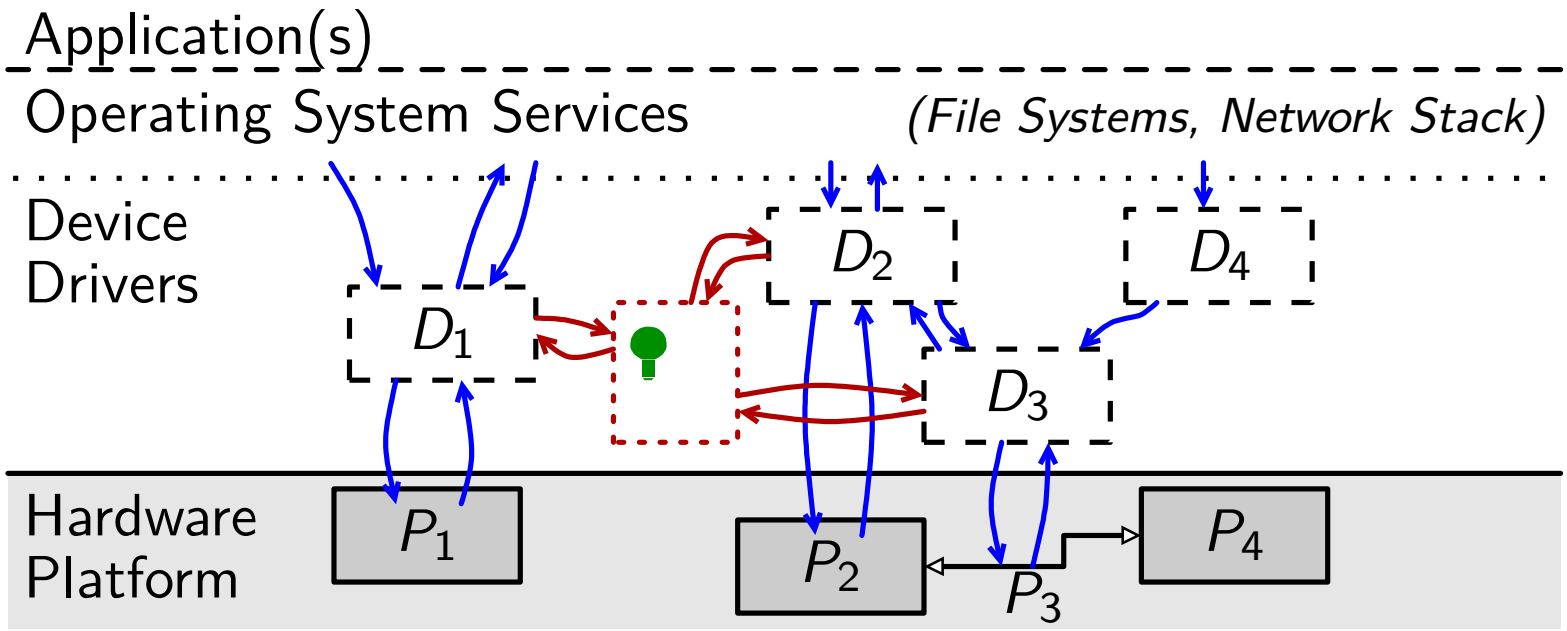
- Dedicated Software Components
  - Each Arbiter Manages Concurrent Accesses to One Shared Device
  - Each Power Manager Controls the Operating State of One Device
  - One Global *Ad hoc* Component Manages the State of the µ-C

Avoiding Consumption Peaks?

Existing Solution for *Global Control*

Multithreaded Device Manager

[Choi *et al.*, 2008]

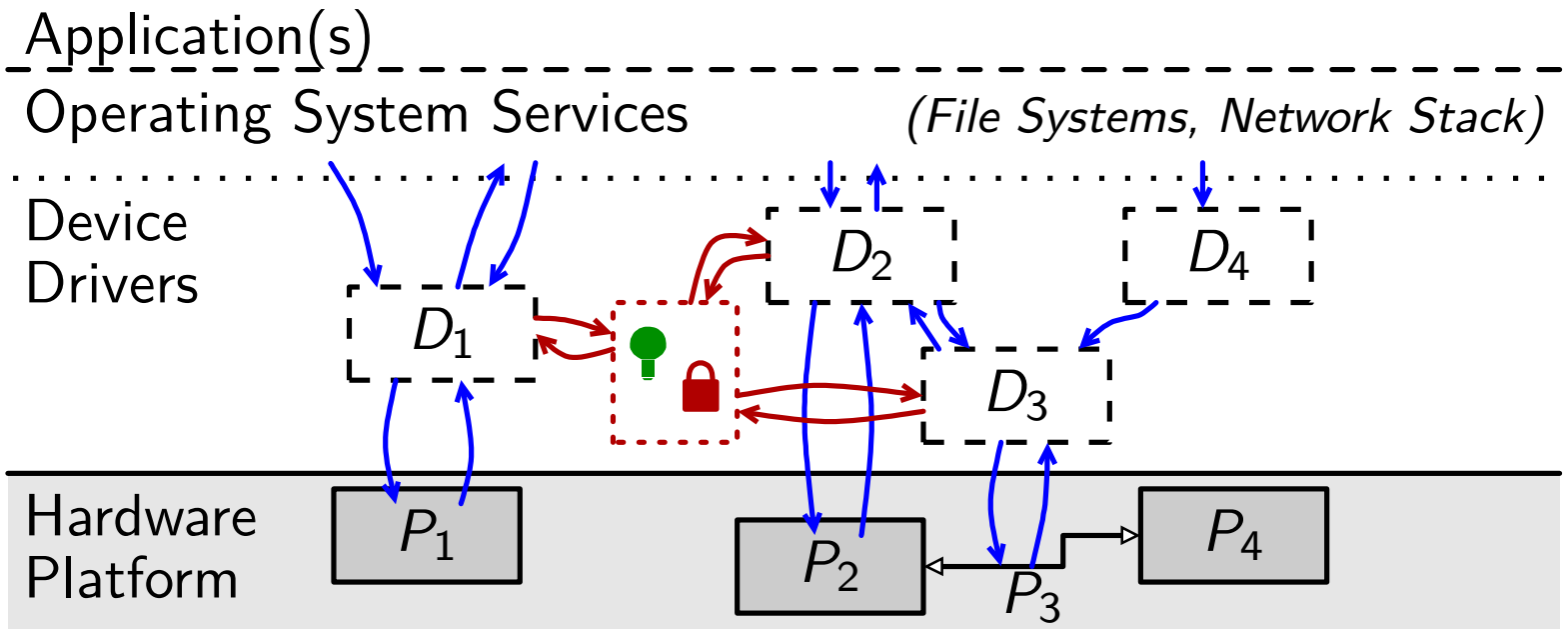


- Global Device Manager
  - Centralized Knowledge
  - Authorization to Change State

Existing Solution for *Global Control*

Multithreaded Device Manager

[Choi *et al.*, 2008]



- Global Device Manager
  - Centralized Knowledge
  - Authorization to Change State
- Use of Locks $\rightsquigarrow$ Deadlocks

Constraints on the Solution we Seek for Global Control

(Global) Control $\Rightarrow$ Refused Requests

Constraints on the Solution we Seek for Global Control

(Global) Control ⇒ Refused Requests

Summary of Related Work

ICEM: Not a Solution for Global Control

Choi *et al.*: Global Control with Unsatisfactory Architecture

Constraints on the Solution we Seek for Global Control

(Global) Control ⇒ Refused Requests

Summary of Related Work

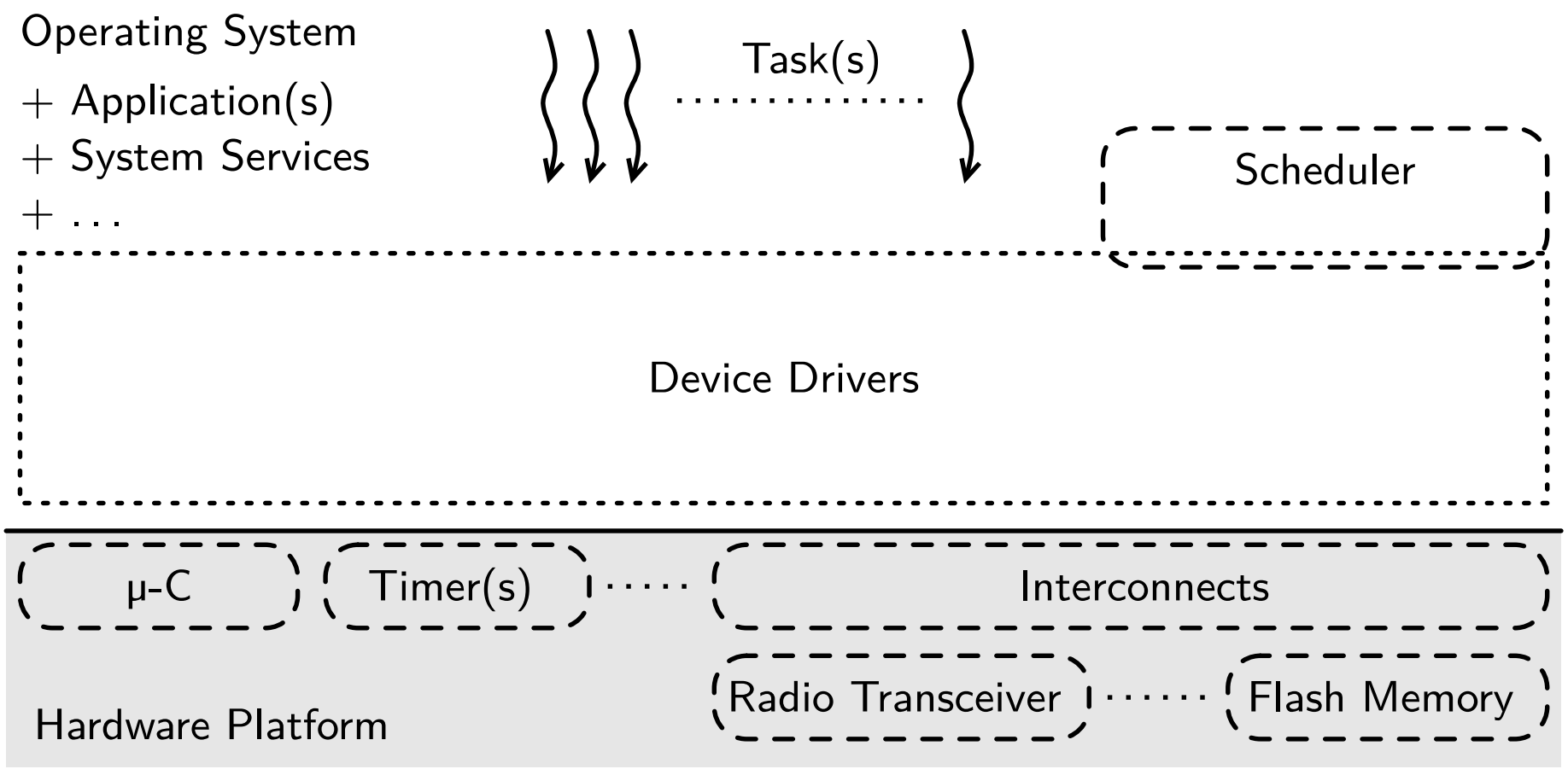
ICEM: Not a Solution for Global Control

Choi *et al.*: Global Control with Unsatisfactory Architecture

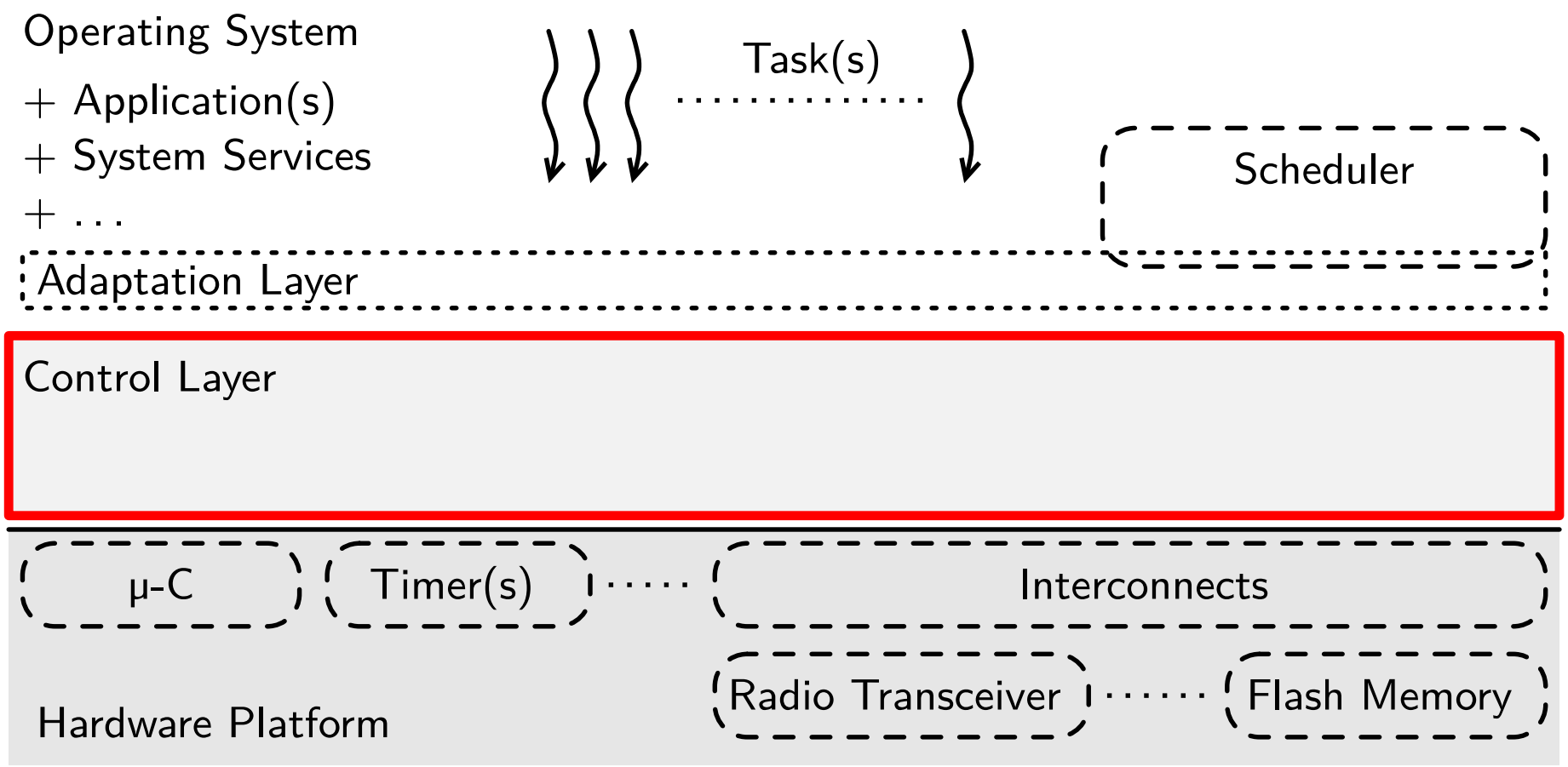
Objectives for the Solution

- Reusing Existing Software with Minimal Impact
  - *e.g.*, Protocol Stacks, Applications
- Reducing Cost/Overhead
- Guaranteeing Correctness
- Impact on Programming Model
  - Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

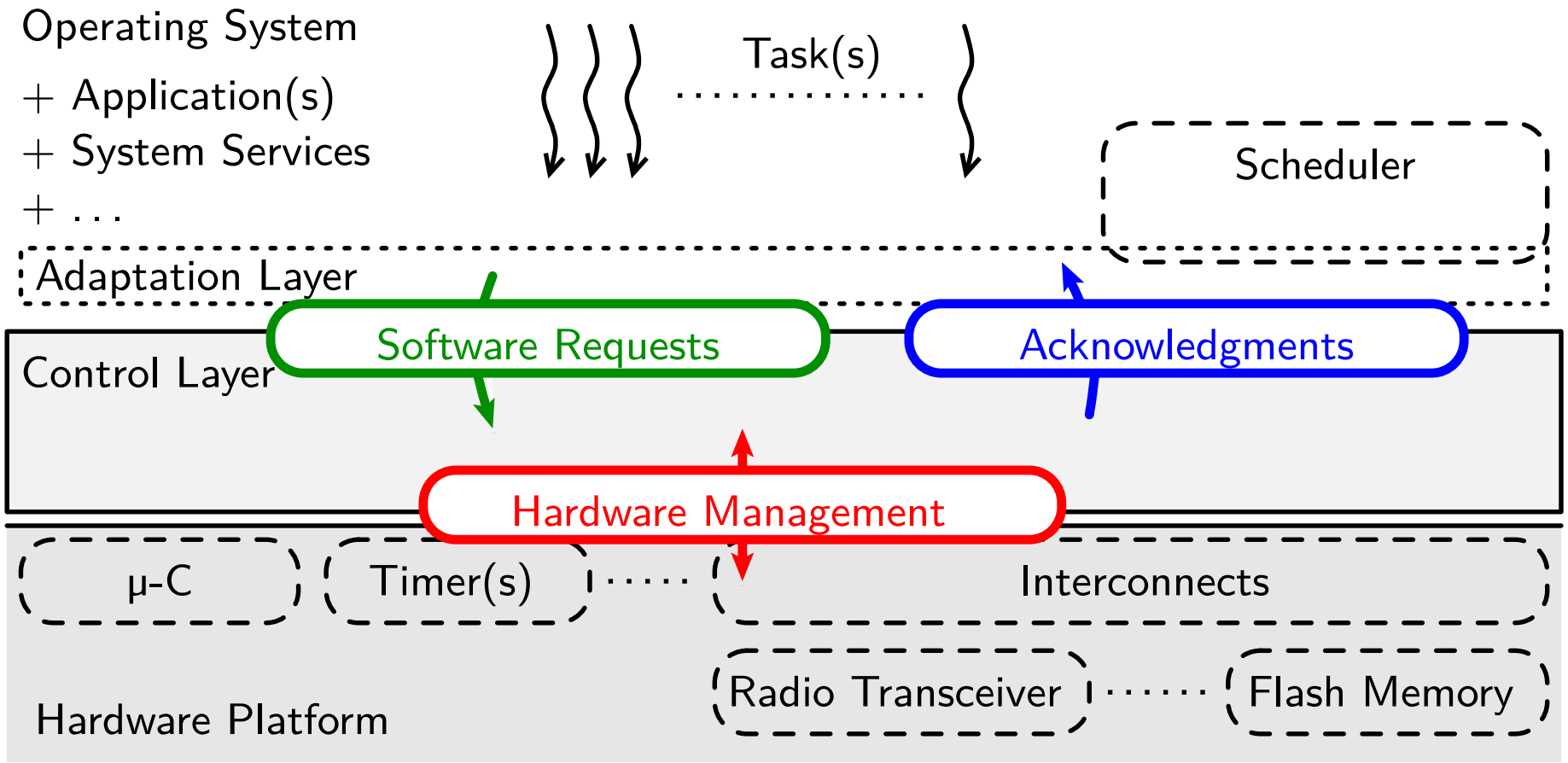
Idea of the Solution: Introducing the Control Layer



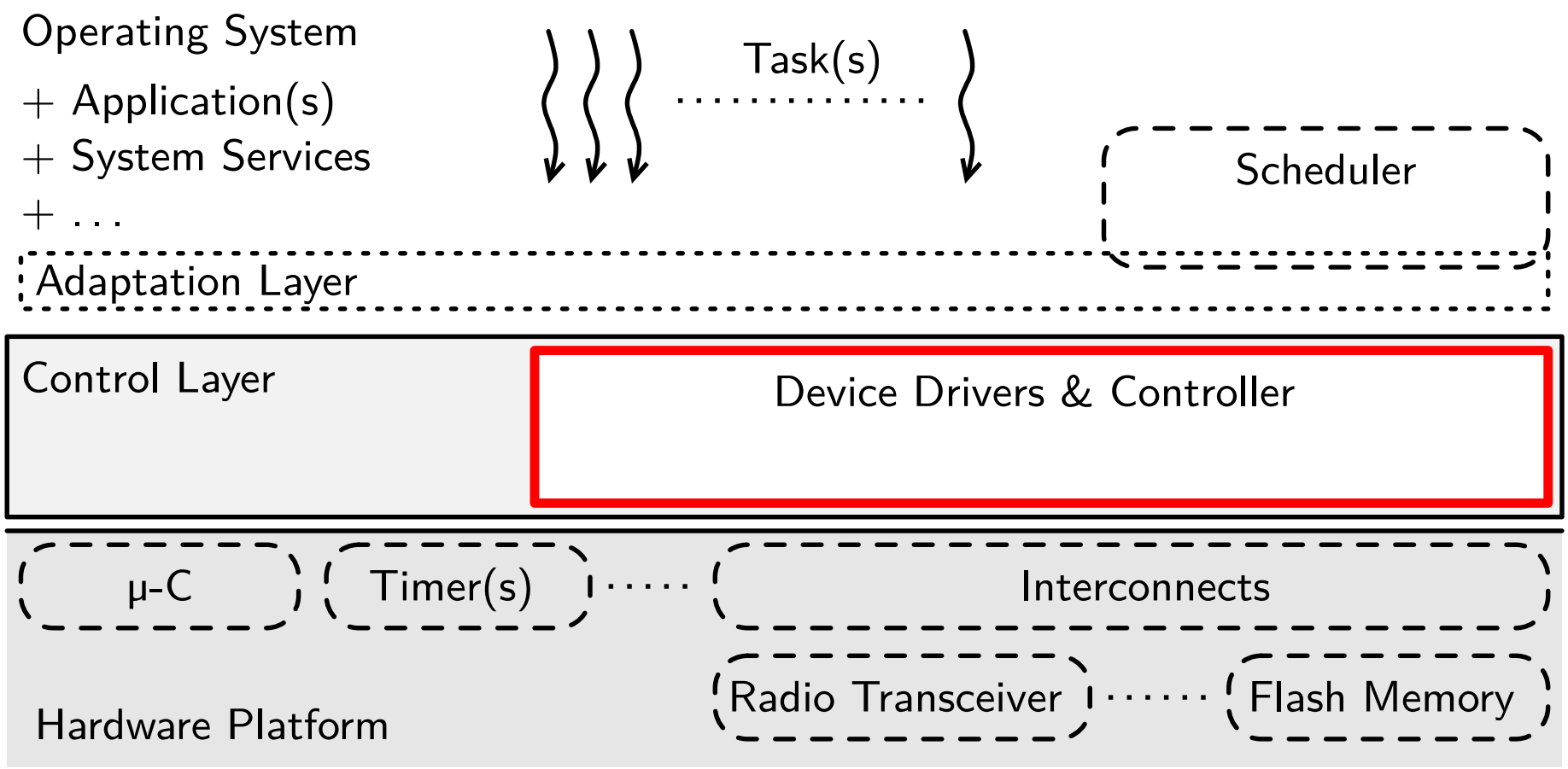
Idea of the Solution: Introducing the Control Layer



Idea of the Solution: Introducing the Control Layer



Idea of the Solution: Building the Control Layer



Design Principle

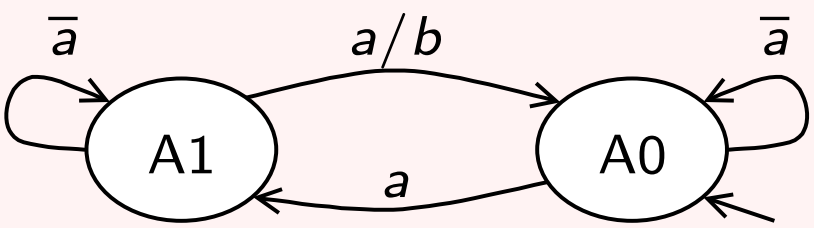
- Synchronous Programming
- Controller Synthesis Principle

Outline

- Choosing an Application Domain and Outline of the Proposal
- Background
  - Synchronous Programming
  - Controller Synthesis Principle
- Detailed Contribution
- Conclusion

Synchronous Programs as Boolean Mealy Machines

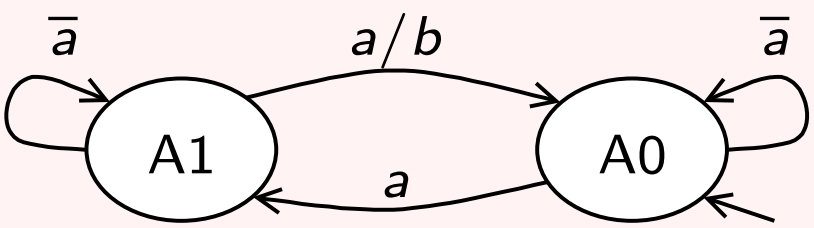
One Boolean Mealy Machine



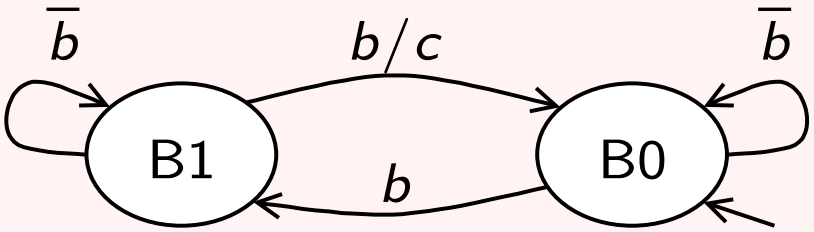
`Sa`: outputs one b every two a's

Synchronous Programs as Boolean Mealy Machines

Synchronous Product



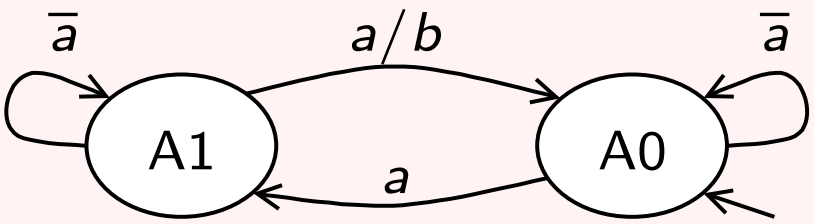
`Sa`: outputs one b every two a's



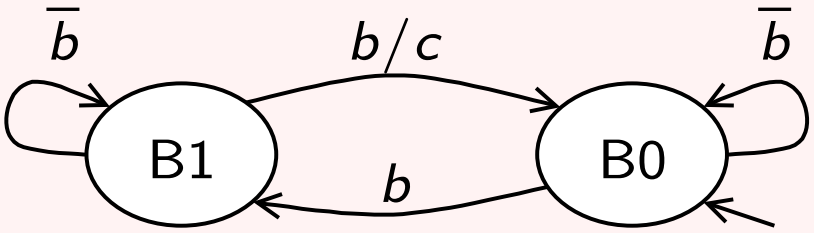
`Sb`: outputs one c every two b's

Synchronous Programs as Boolean Mealy Machines

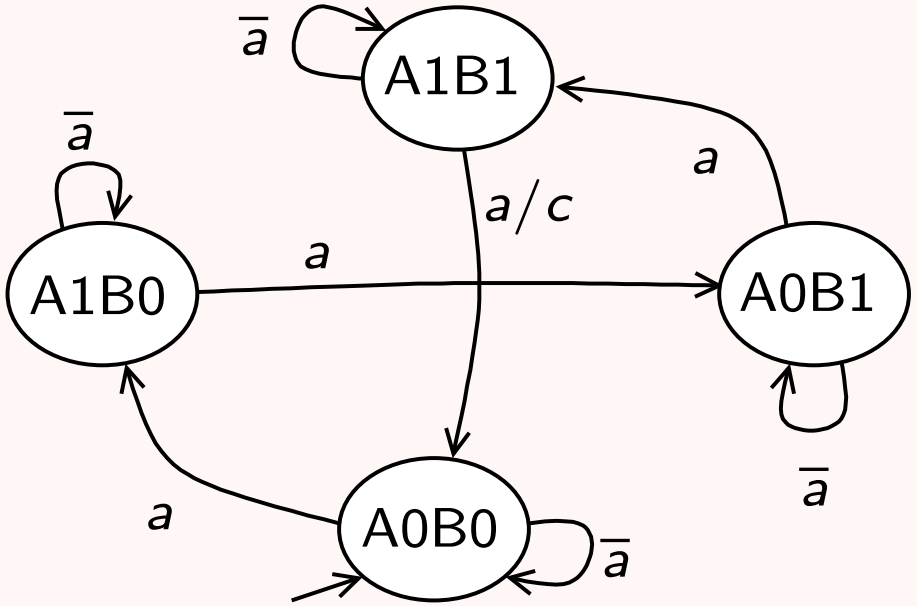
Synchronous Product



Sa: outputs one b every two a's



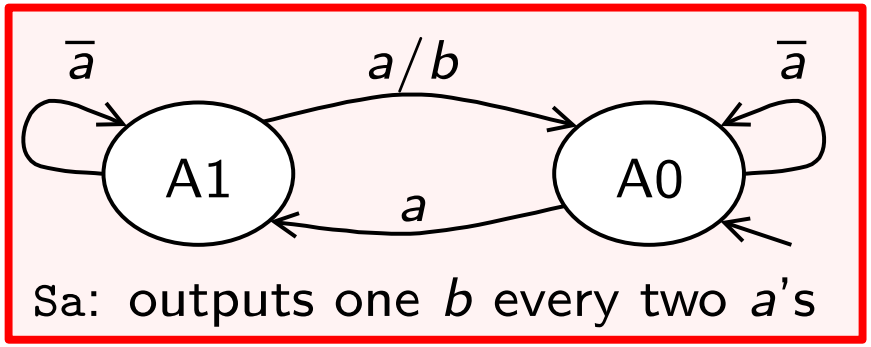
Sb: outputs one c every two b's



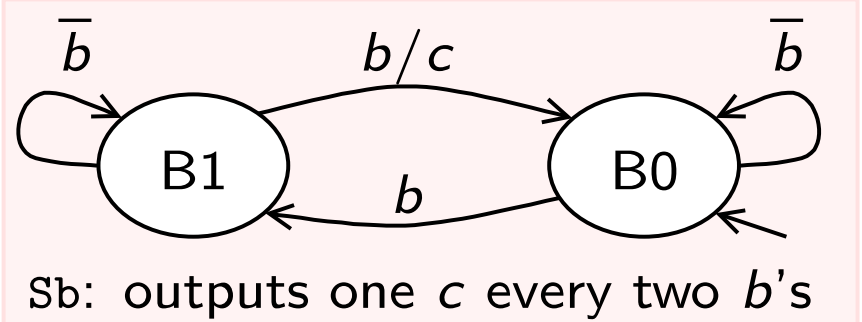
Sa$\times$Sb$\backslash b$: one c every four a's

Synchronous Programs as Boolean Mealy Machines

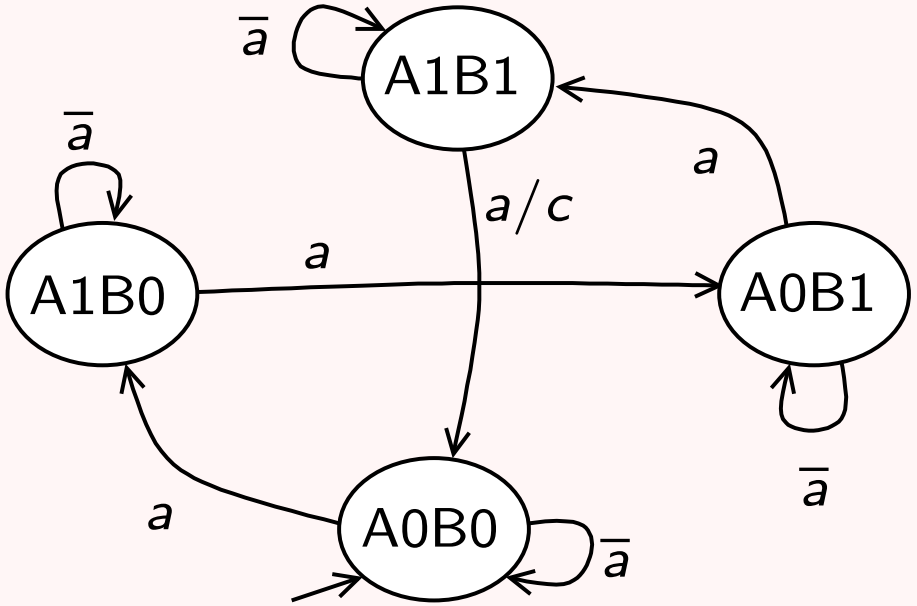
In Lustre



Sa: outputs one b every two a's



Sb: outputs one c every two b's

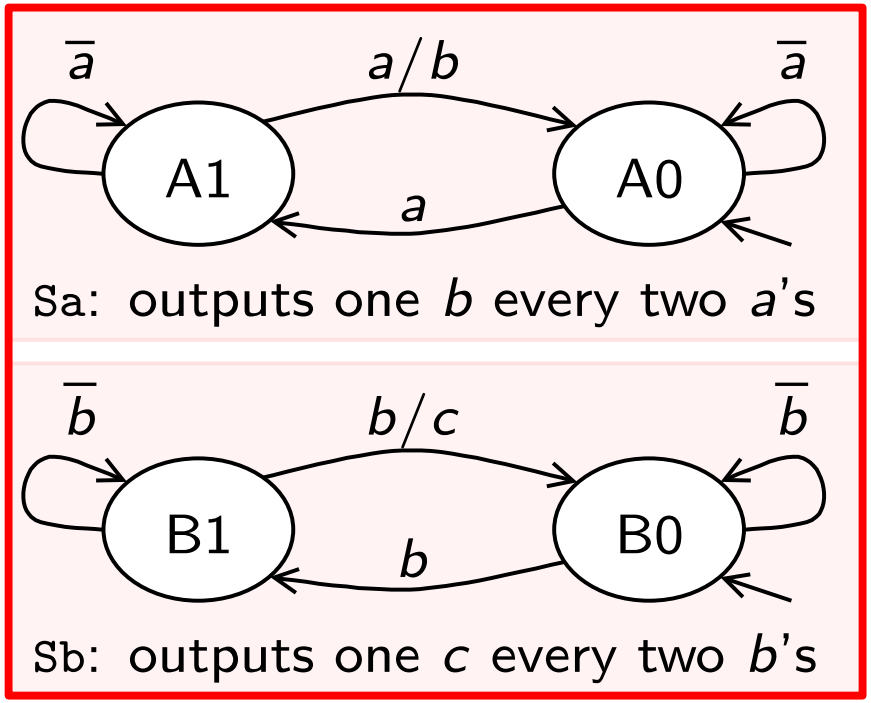


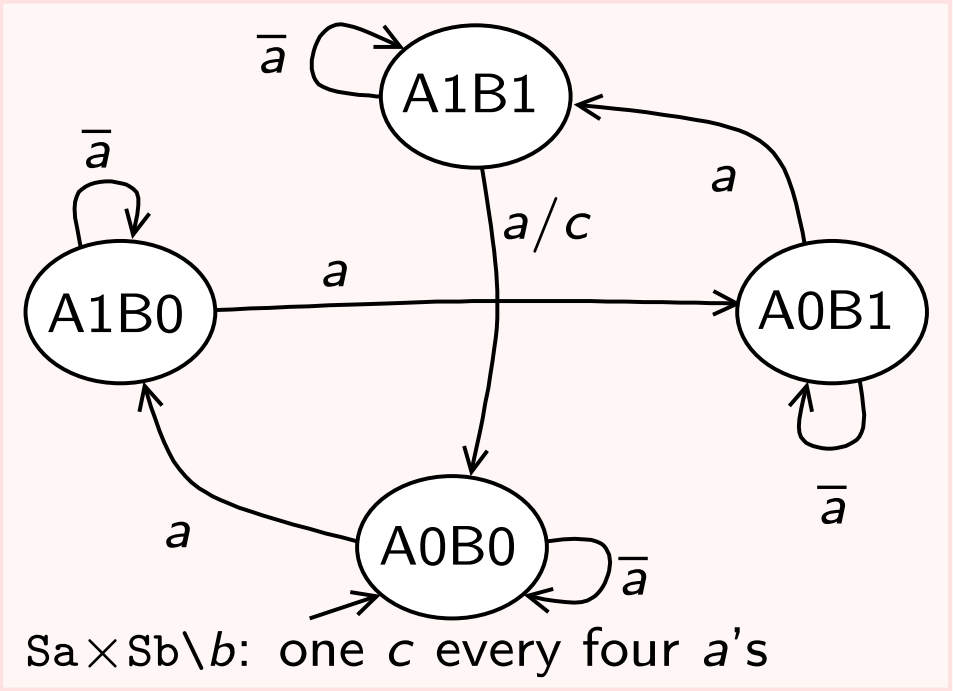
Sa$\times$Sb$\backslash b$: one c every four a's

```
node Sa (a: bool) returns (b: bool);
var inA0, inA1, m_A0, A0: bool;
let
  inA1 = not m_A0;      inA0 = m_A0;      b = a and inA1;

  A0 = not a and inA0 or a and inA1; m_A0 = true -> pre A0;

tel;
```

Synchronous Programs as Boolean Mealy Machines

In Lustre






```
node SE (a: bool) returns (c: bool);
var inA0, inA1, m_A0, A0, inB0, inB1, m_B0, B0, b: bool;
let
  inA1 = not m_A0;      inA0 = m_A0;      b = a and inA1;
  inB1 = not m_B0;      inB0 = m_B0;      c = b and inB1;
  A0 = not a and inA0 or a and inA1; m_A0 = true -> pre A0;
  B0 = not b and inB0 or b and inB1; m_B0 = true -> pre B0;
tel;
```

Synchronous Programs as C Code

The Compilation Produces a Reactive Kernel in C

```
bool M1, M2, INIT;                /* State Variables */
void init () { INIT = 1; } /* Initialization  */

void run_step (bool a) {
  bool L1, L2, L3, L4, L5, L6;
  L2 = INIT | M1;    L5 = INIT | M2;
  L4 = ~L5 &  a;     L1 = ~L2 &  L4;
  L6 =  L5 & ~a;     L3 =  L2 & ~L4;
  main_0_c (L1);
  M1 = L3 | L1;      M2 = L6 | L4;
  INIT = 0;
}
```

One Call to `run_step()` $\equiv$ One Transition in the Product

Controller Synthesis Principle

Principle

- Automata $A_1 \dots A_n$
- Property ϕ (*e.g.*, Avoiding Consumption Peaks)

$$A_1 \parallel A_2 \parallel \cdots \parallel A_n \qquad \phi$$

Controller Synthesis Principle

Principle

- Automata $A_1 \dots A_n$
- Property ϕ (*e.g.*, Avoiding Consumption Peaks)
- Provided $A_1 \dots A_n$ are made Controllable ($\rightsquigarrow A_1{}' \dots A_n{}'$)

$$A_1{}' \parallel A_2{}' \parallel \cdots \parallel A_n{}' \qquad \phi$$

Controller Synthesis Principle

Principle

- Automata $A_1 \dots A_n$
- Property ϕ (*e.g.*, Avoiding Consumption Peaks)
- Provided $A_1 \dots A_n$ are made Controllable ($\rightsquigarrow A_1{}' \dots A_n{}'$)

$\rightsquigarrow$ Controller C

$$A_1{}' \parallel A_2{}' \parallel \cdots \parallel A_n{}' \parallel C \models \phi$$

Controller Synthesis Principle

Principle

- Automata $A_1 \dots A_n$
- Property ϕ (*e.g.*, Avoiding Consumption Peaks)
- Provided $A_1 \dots A_n$ are made Controllable ($\rightsquigarrow A_1{}' \dots A_n{}'$)

$\rightsquigarrow$ Controller C

$$A_1{}' \parallel A_2{}' \parallel \cdots \parallel A_n{}' \parallel C \models \phi$$

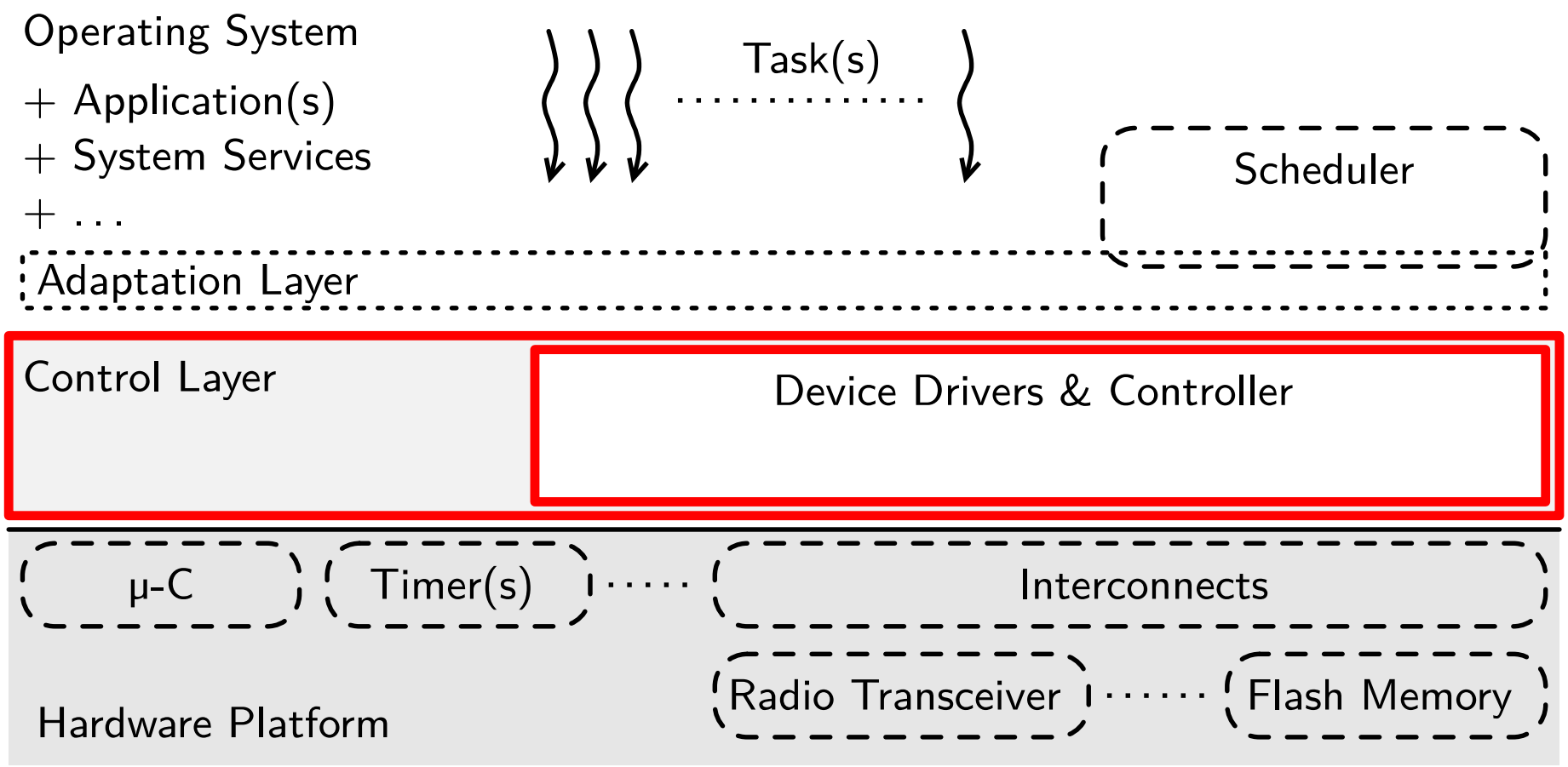
Available Tools

- For Synchronous Languages: SIGALI [Marchand, 1997]

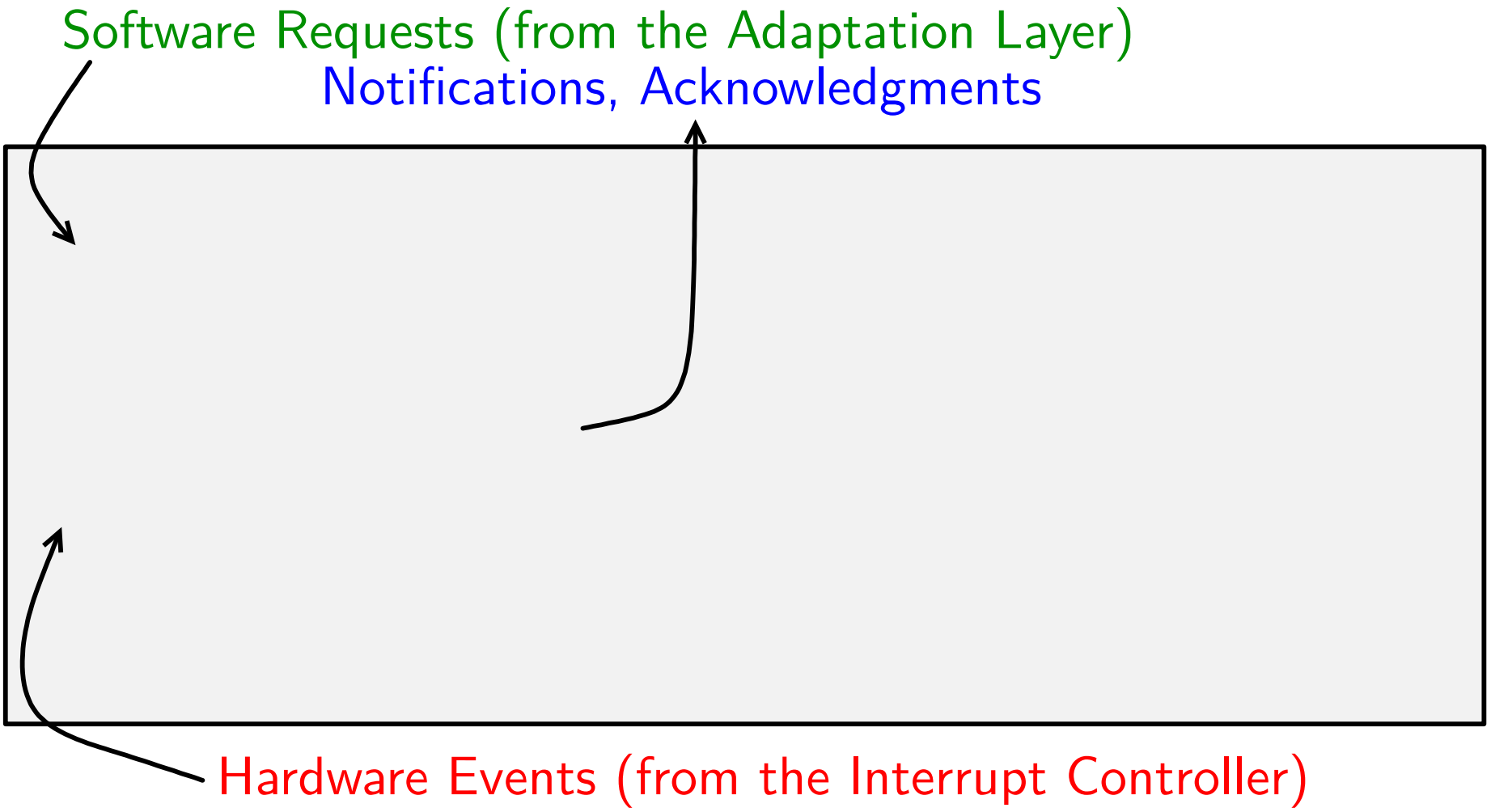
- In this Talk: Manual Design

Outline

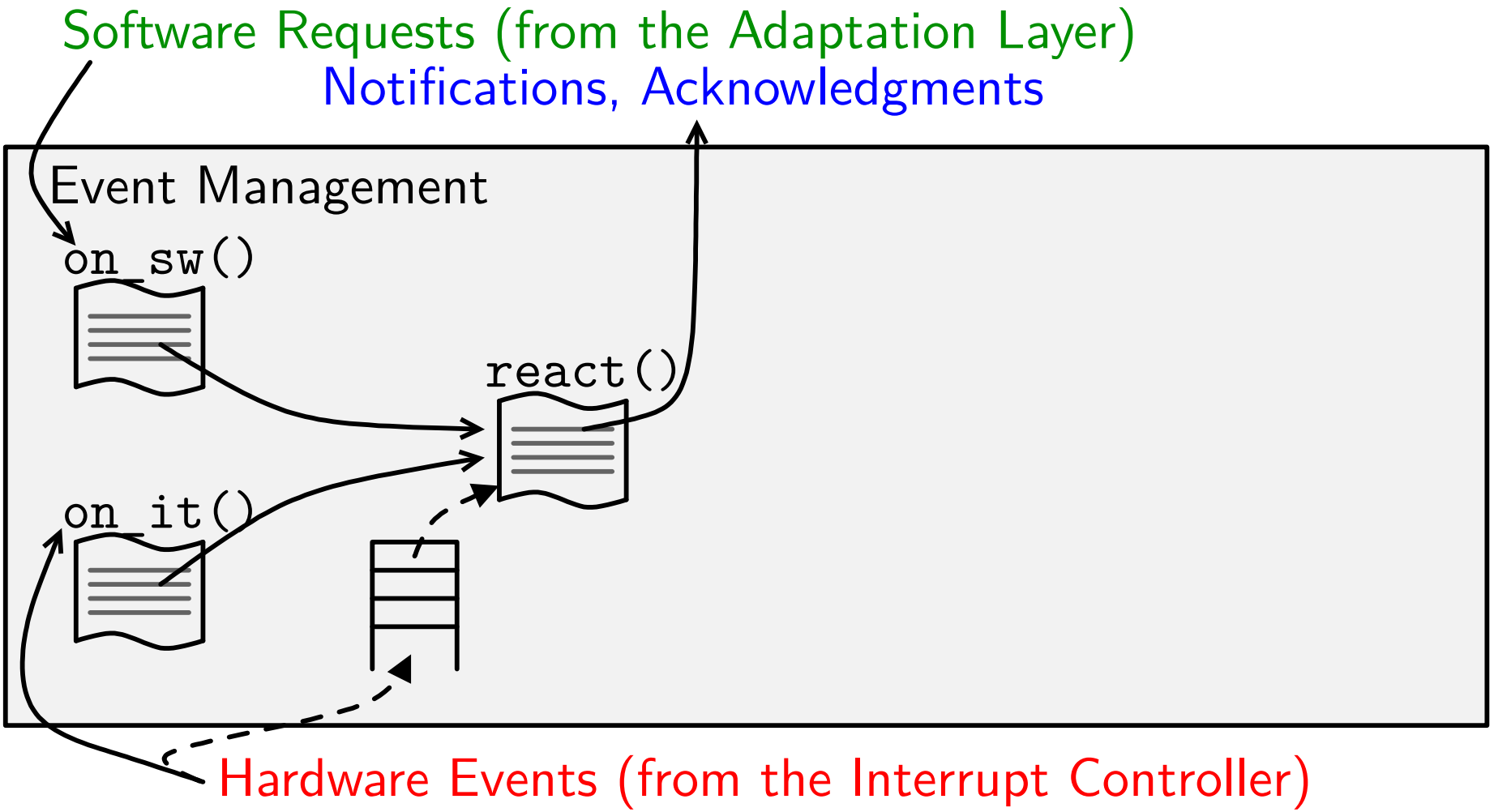
Idea of the Solution: Building the Control Layer



Architecture: Control Layer



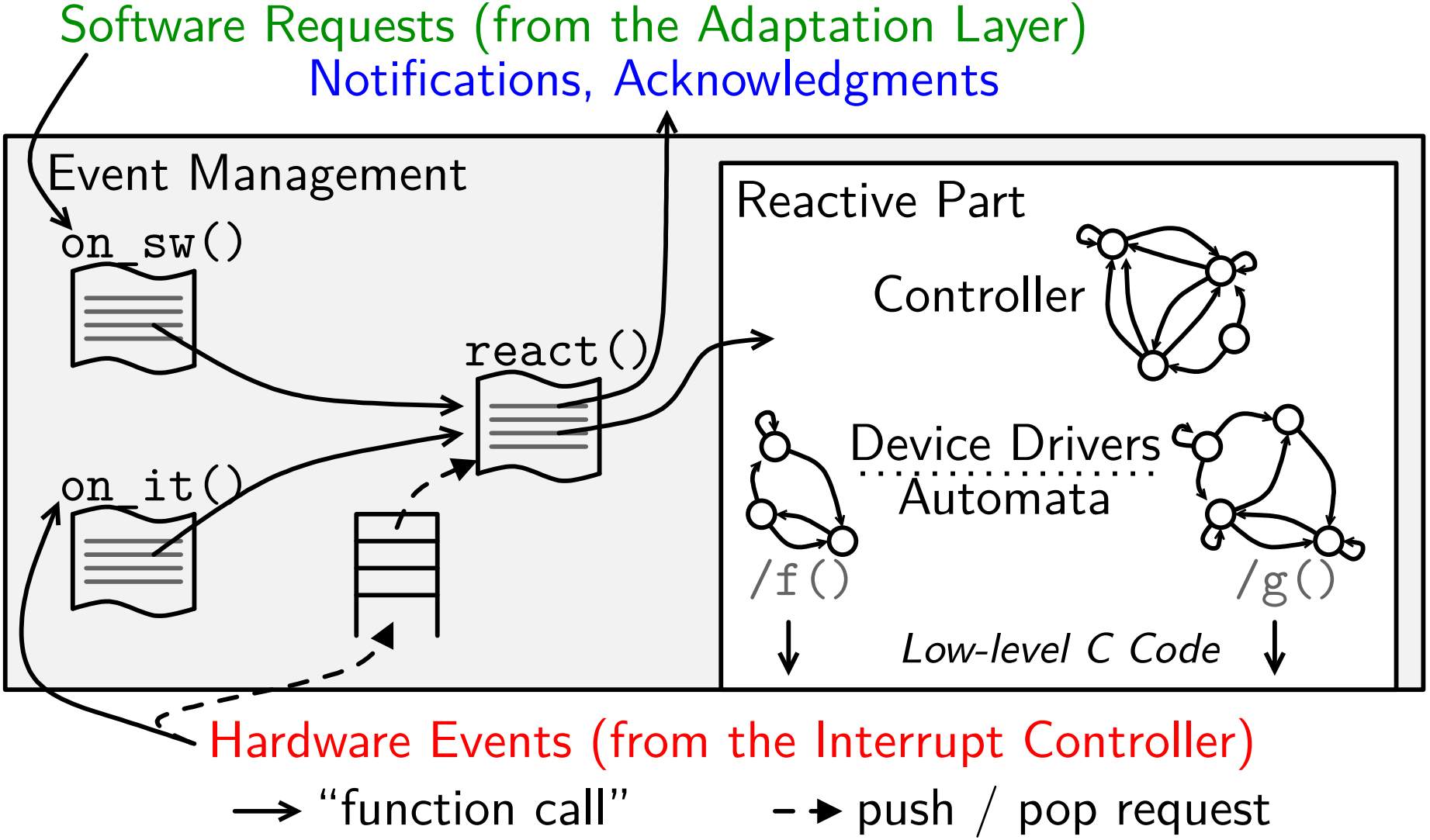
Architecture: Control Layer



→ "function call" - → push / pop request

`react()`: Combines Software and Hardware Requests to be Submitted to the Reactive Part

Architecture: Control Layer

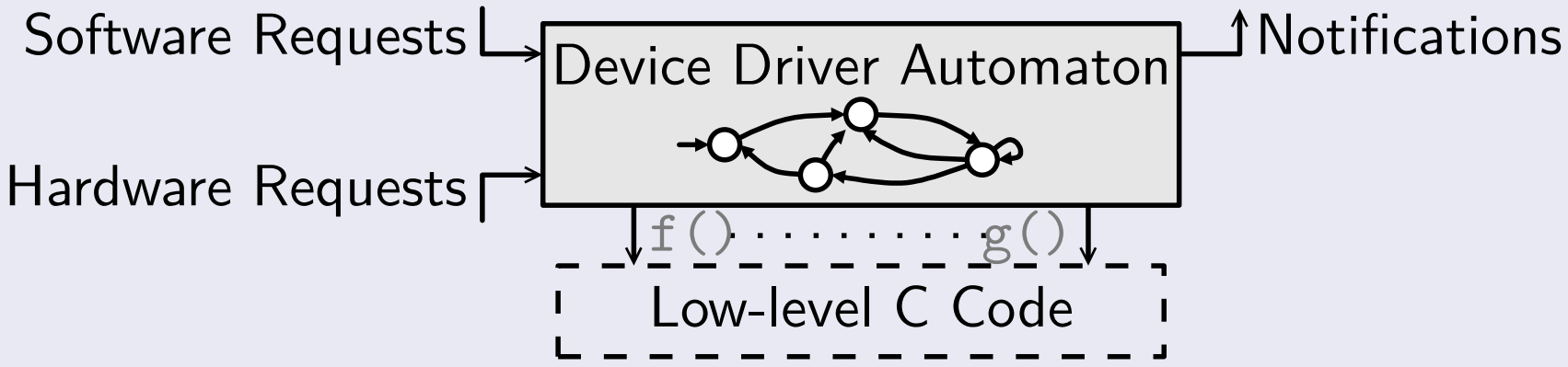


→ "function call" - → push / pop request

`react()`: Combines Software and Hardware Requests to be Submitted to the Reactive Part

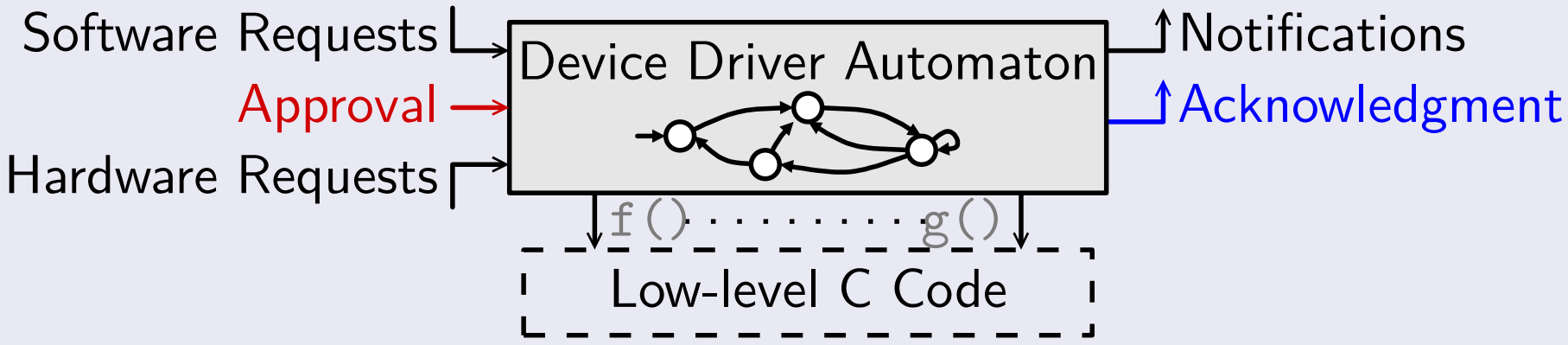
Automata Involved

One Driver Automaton per Device



Automata Involved

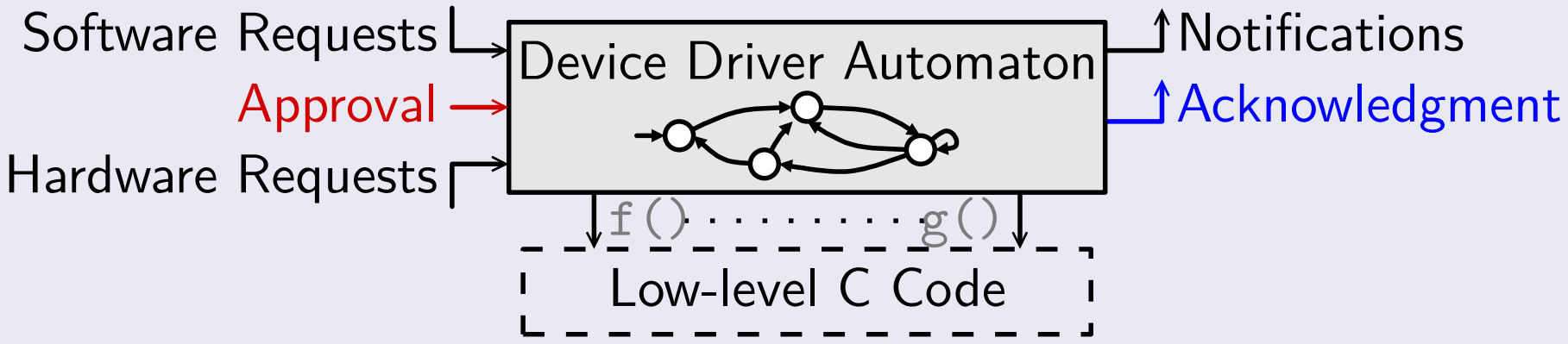
One Driver Automaton per Device



- Can be made *Controllable*

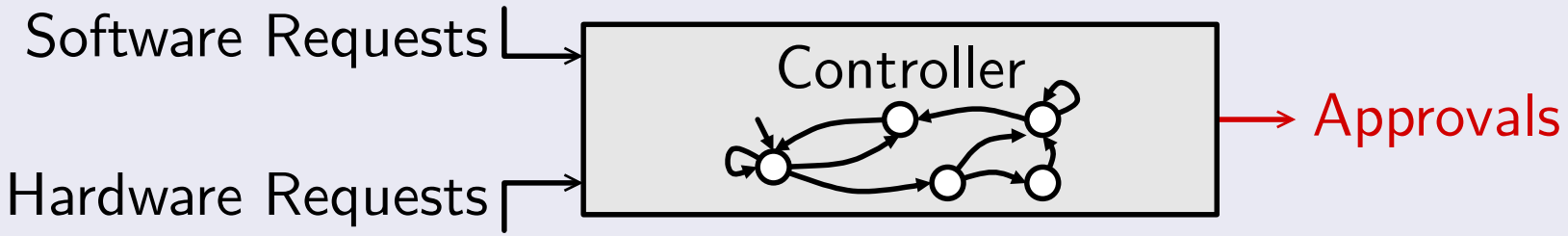
Automata Involved

One Driver Automaton per Device

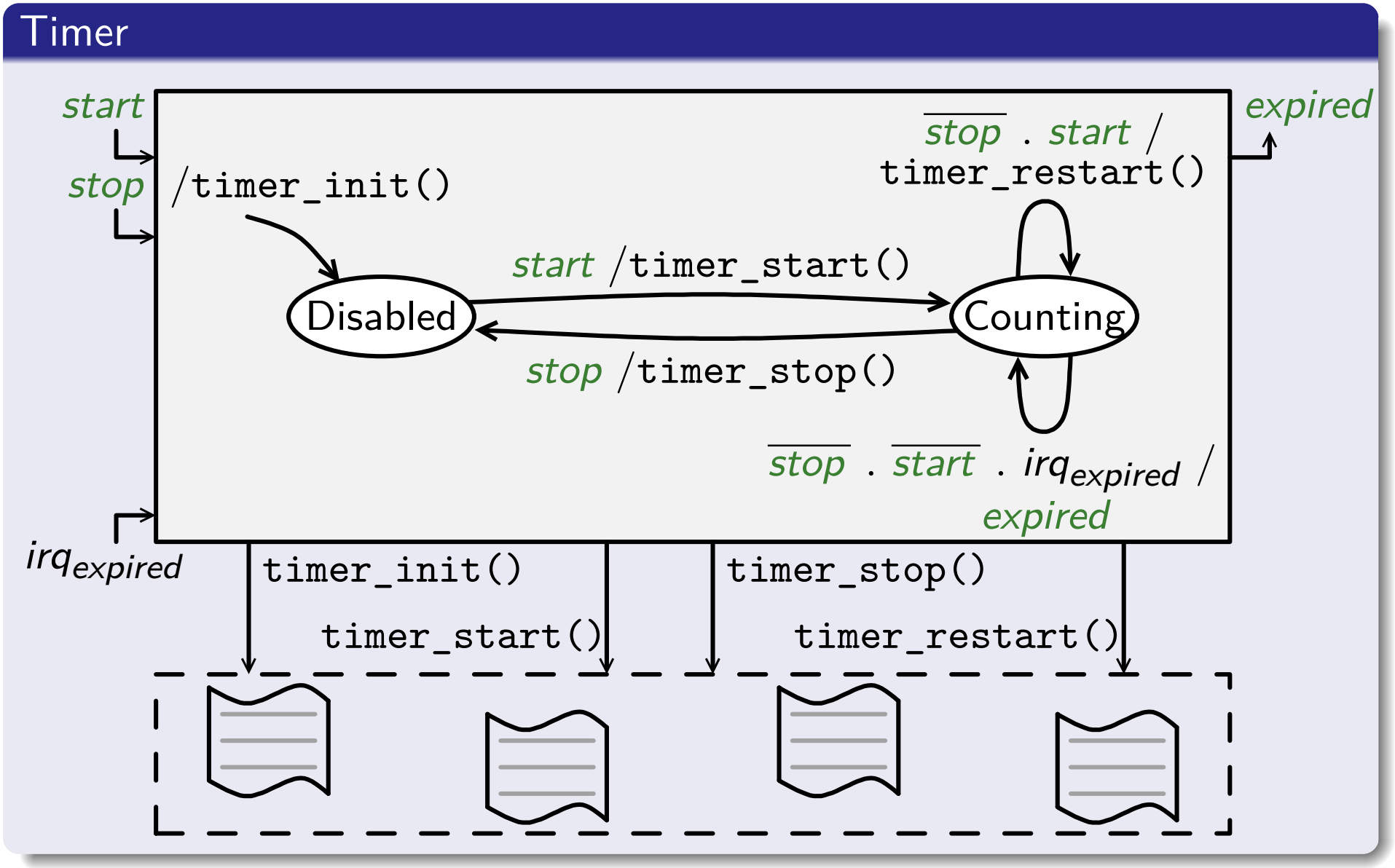


- Can be made *Controllable*

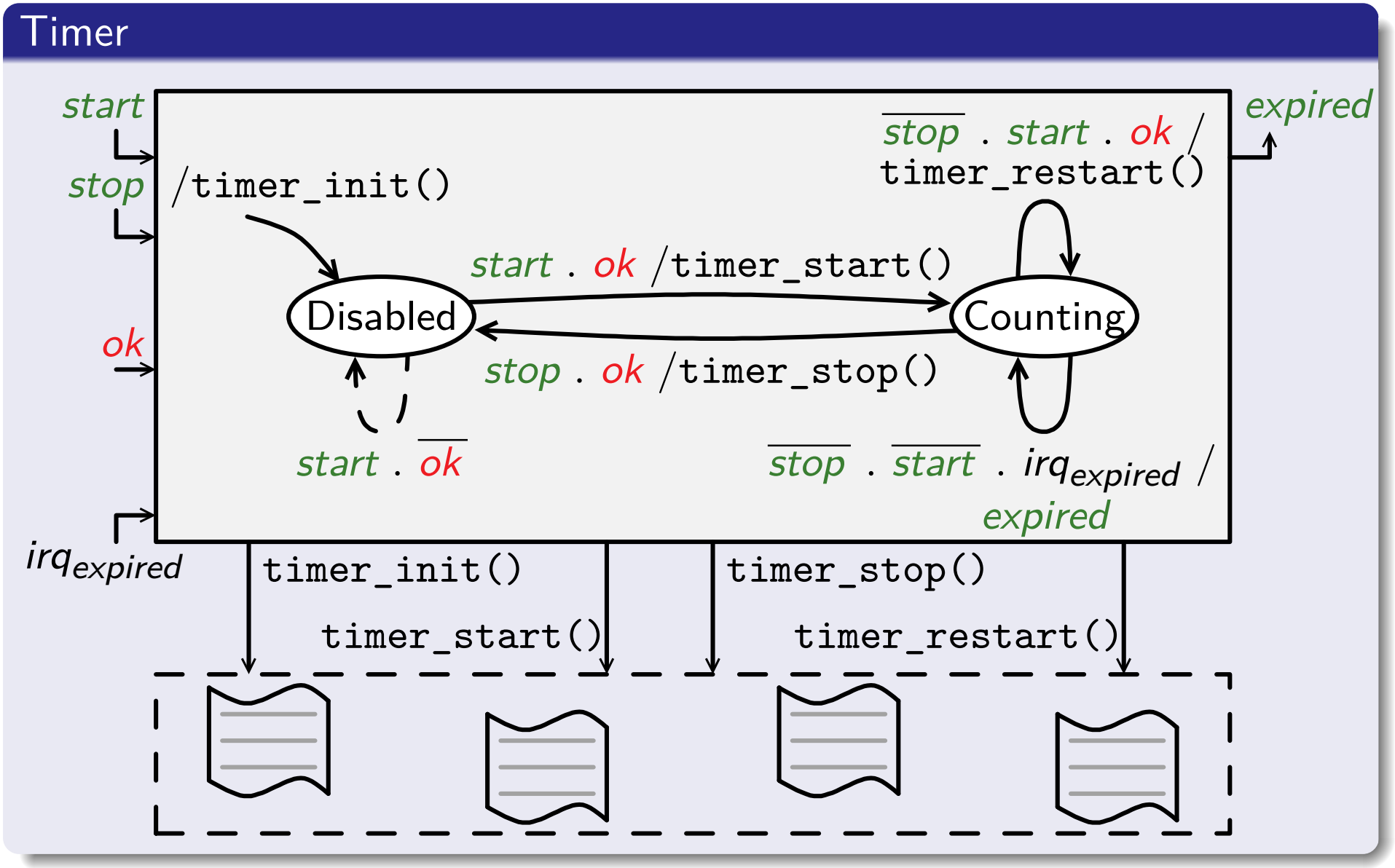
One Controller Makes Decisions



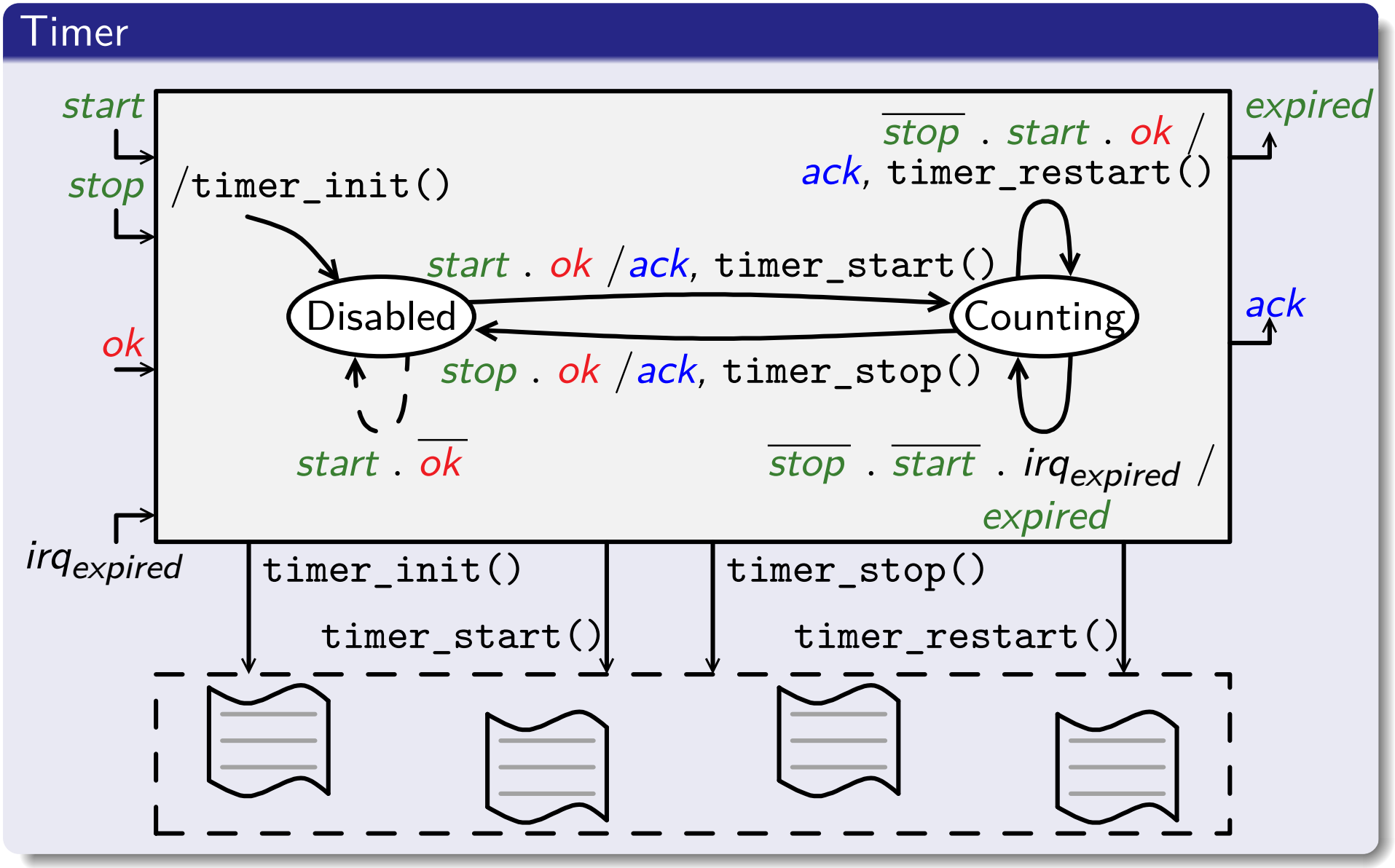
Example of Uncontrollable Driver Automaton



Example of Controllable Driver Automaton

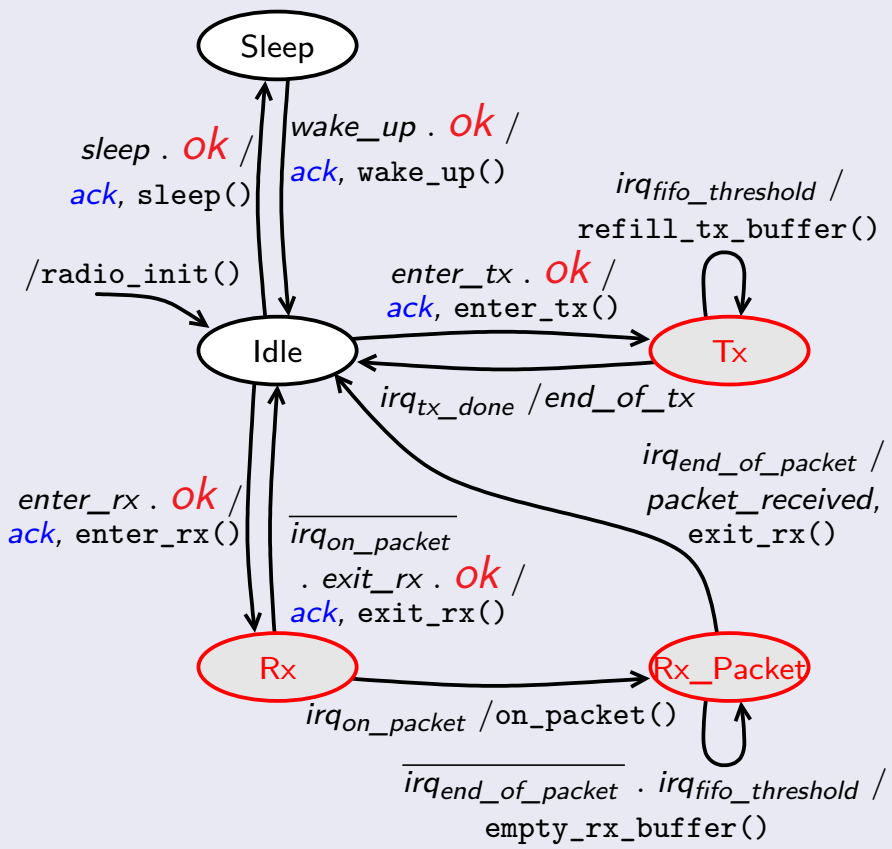


Example of Controllable Driver Automaton

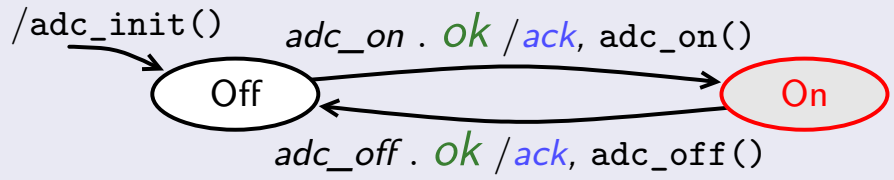


Exclusion of Energy-greedy States: Example

Radio Transceiver

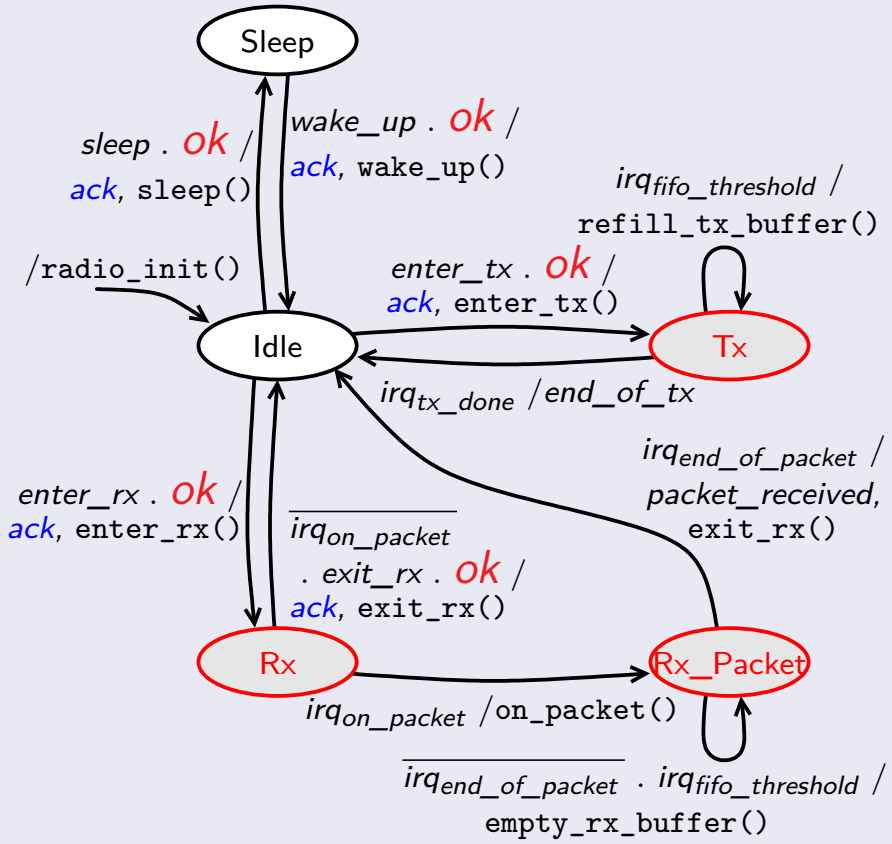


ADC

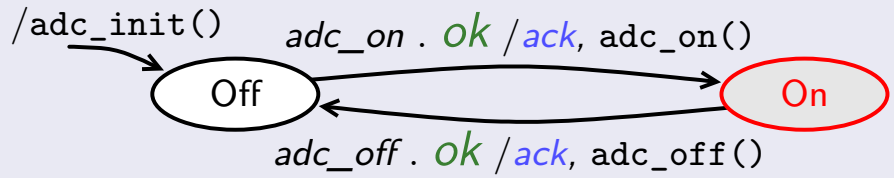


Exclusion of Energy-greedy States: Example

Radio Transceiver



ADC

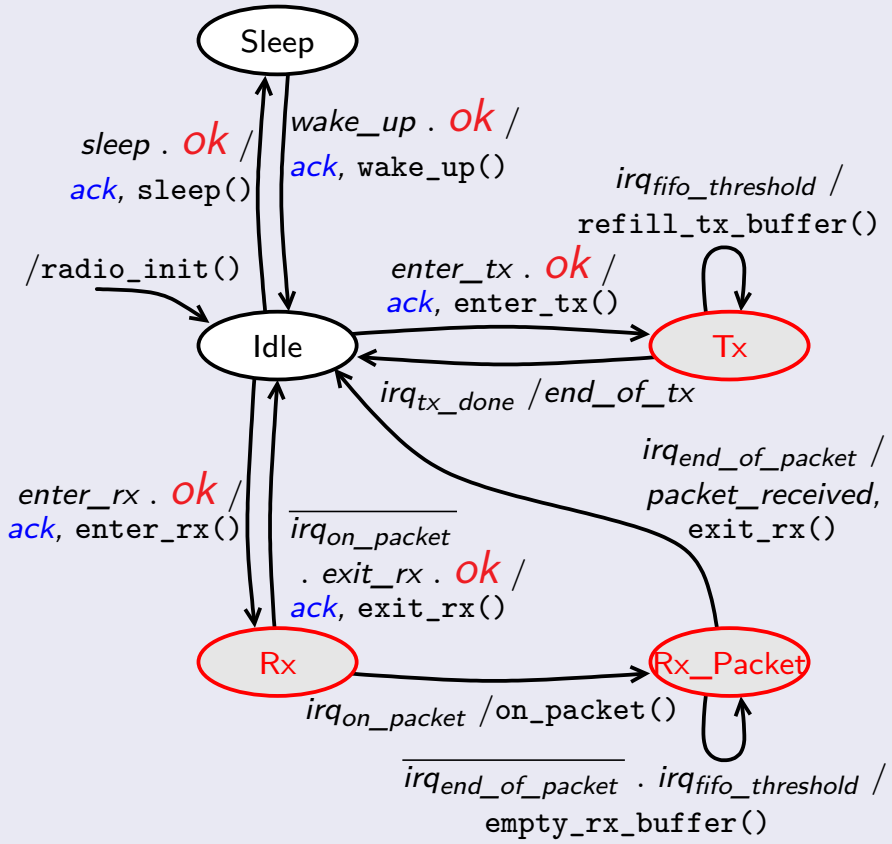


Guarantee:

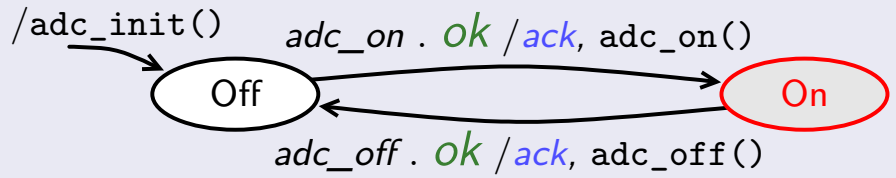
- States Tx×On, Rx×On and Rx_Packet×On are Unreachable

Exclusion of Energy-greedy States: Example

Radio Transceiver



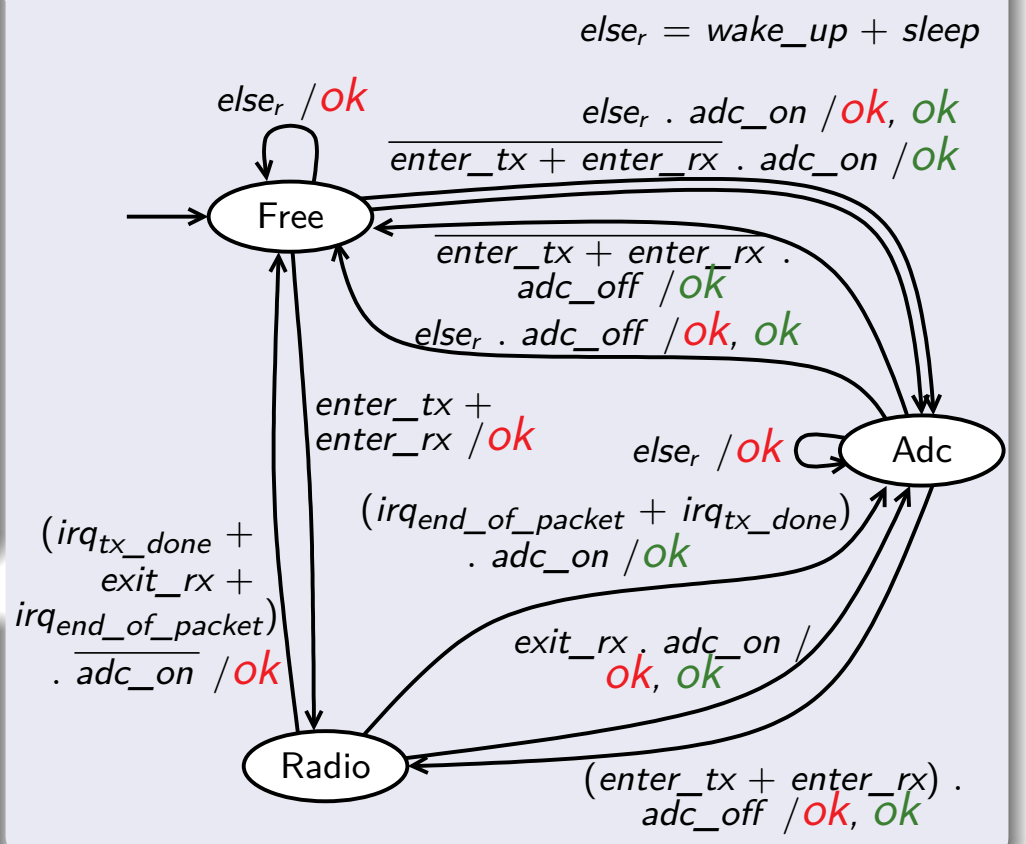
ADC



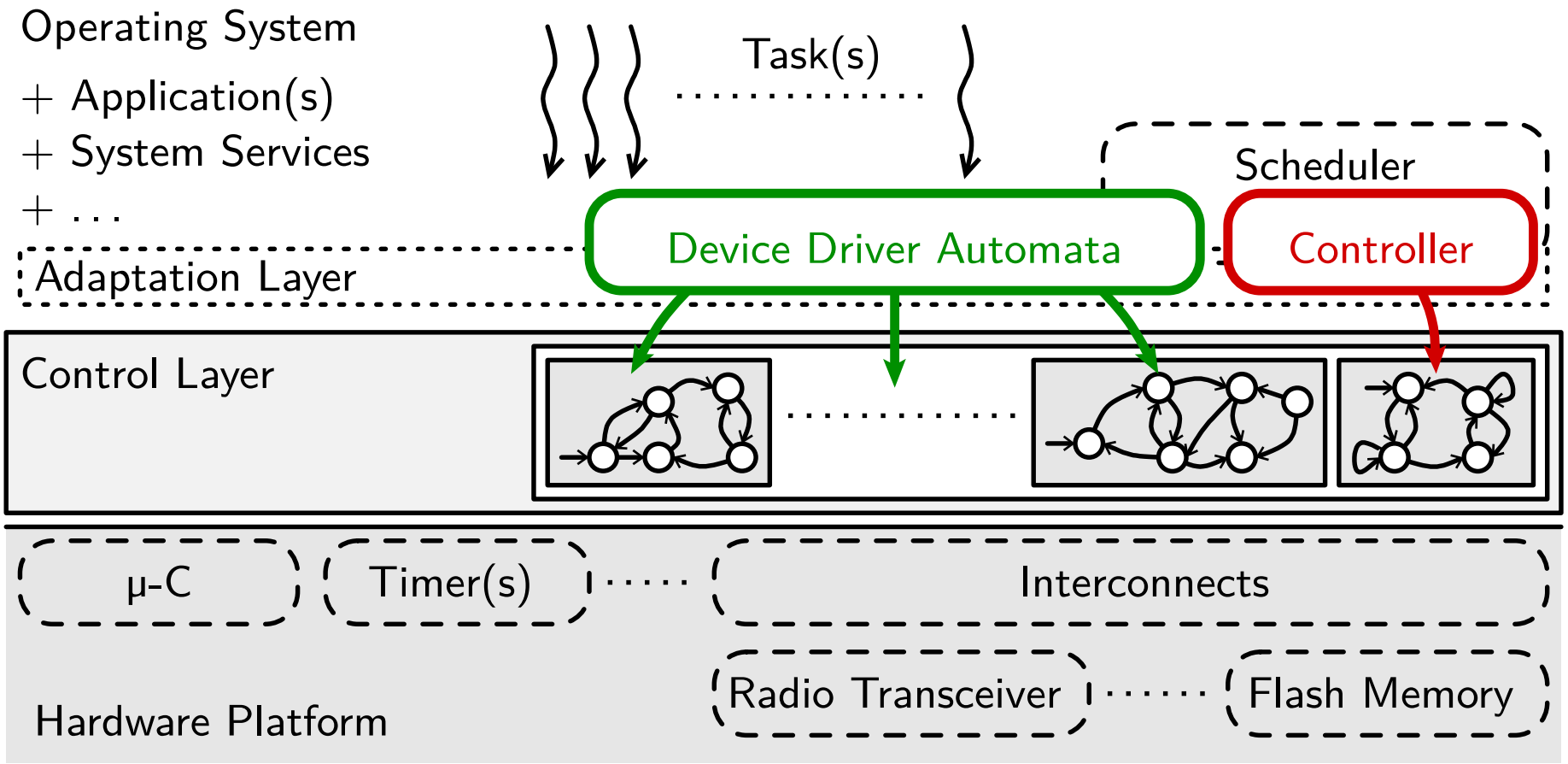
Guarantee:

- States Tx×On, Rx×On and Rx_Packet×On are Unreachable

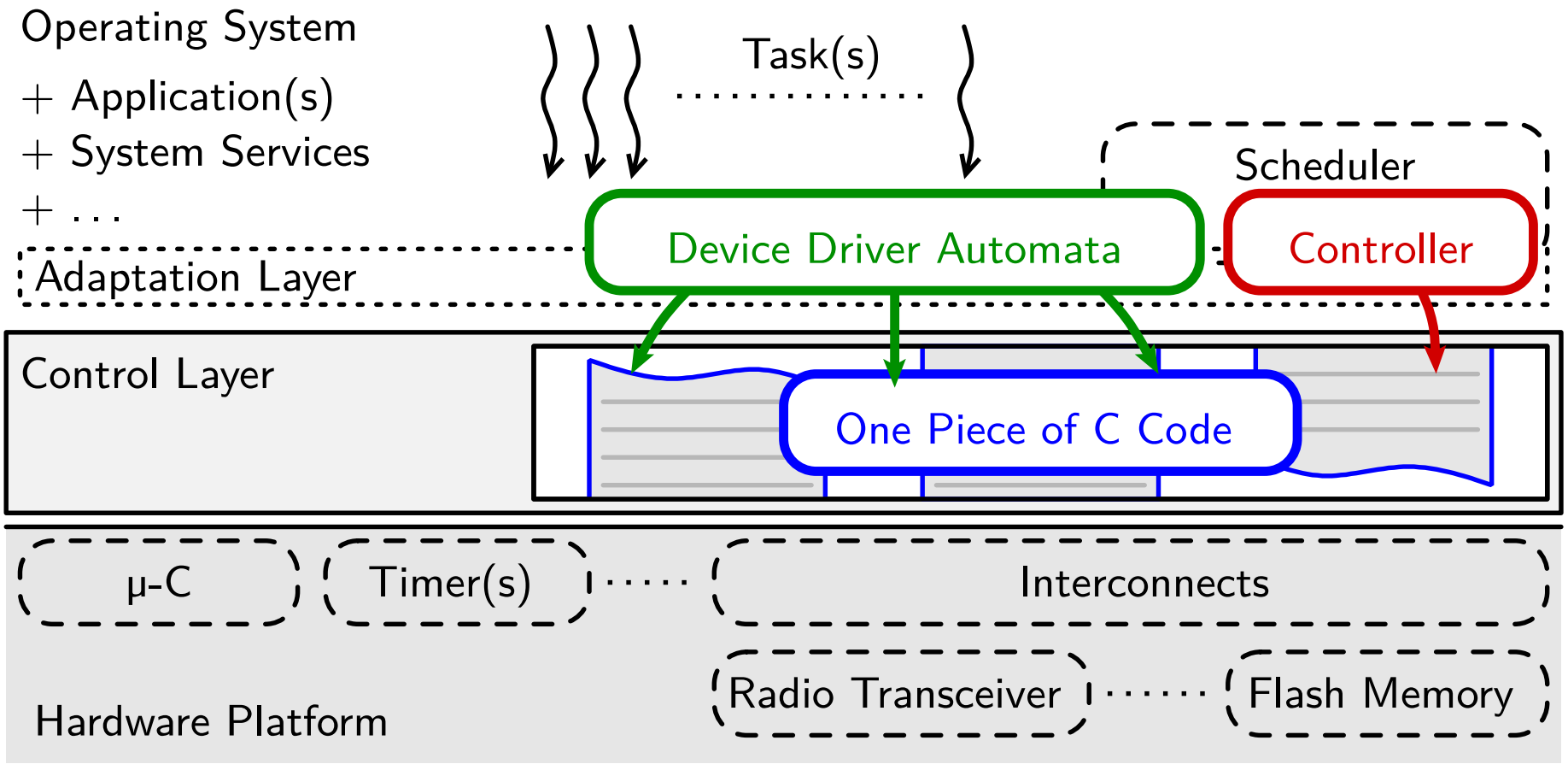
Controller



Putting it All Together



Putting it All Together



Outline

Programming with Boolean Mealy Machines

Implementing Control Parts of the Device Drivers

Synchronous Programming Directly in C [von Hanxleden, 2009]

```
extern int tick (void) {
  TICKSTART (MainTask); /* Main Task Specification. */

  FORK (Sa, SaTask);    /* Declaring SaTask.        */
  FORK (Sb, SbTask);    /* idem SbTask.             */
  FORKE (Parent);

 Sa:
 A0:                    /* Initial State.           */
  AWAIT (a);            /* Await Input 'a'.         */
  TRANS (A1);           /* Go to State A1.          */
 A1:
  AWAIT (a);
  EMIT (b);             /* Emit 'b'.                */
  TRANS (A0);           /* Go to State A0.          */

 Sb:
 B0:
  AWAIT (b);
  TRANS (B1);
 B1:
  ...
```

Programming with Boolean Mealy Machines

Implementing Control Parts of the Device Drivers

Synchronous Programming Directly in C [von Hanxleden, 2009]

```
extern int tick (void) {
  TICKSTART (MainTask); /* Main Task Specification. */

  FORK (Sa, SaTask);    /* Declaring SaTask.        */
  FORK (Sb, SbTask);    /* idem SbTask.             */
  FORKE (Parent);

 Sa:
 A0:                    /* Initial State.           */
  AWAIT (a);            /* Await Input 'a'.         */
  TRANS (A1);           /* Go to State A1.          */
 A1:
  AWAIT (a);
  EMIT (b);             /* Emit 'b'.                */
  TRANS (A0);           /* Go to State A0.          */

 Sb:
 B0:
  AWAIT (b);
  TRANS (B1);
 B1:
  ...
```

Scalability Issues

Programming with Boolean Mealy Machines *(cont'd)*

Implementing Control Parts of the Device Drivers

ARGOS [Maraninchi, 1990]

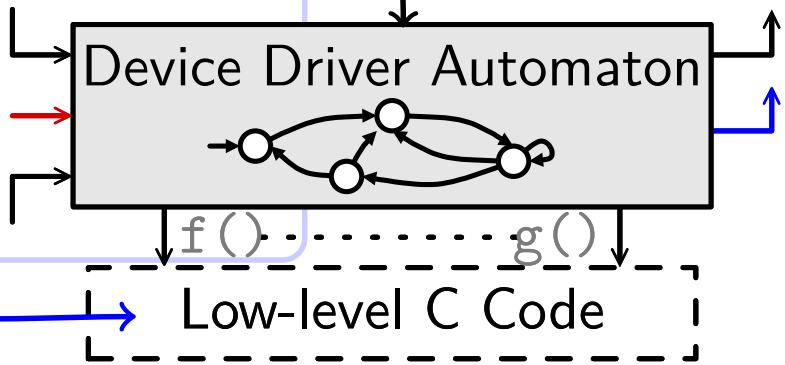
```
process SE (a) (c) {
  internal b {                    // Local Signal b
    automaton                     // Sa
      {
        init A0                   // Initial State
        A0                        // From A0...
          -> A1 with a;           // ... Go to A1 if 'a'
          +> A0;                  // ... Else Stay in A0
        A1
          -> A0 with a / b;
          +> A1;
      }
    ||                            // Parallel Composition
    automaton                     // Sb
      {
        init B0
        B0
          -> B1 with b;
          +> B0;
        B1
          -> B0 with b / c;
          +> B1;
      }
  }
}
```

$\rightsquigarrow$ LUSTRE Code

Synchronous Device Driver: Example

```
process tsl2550 (on, off, cfg, read0, read1, ok) (ack, sensing) {
  internal shutdown {
    automaton {
      init Off
      Off
        -> Sensing with on & ok / ack, on_code, sensing;
        +> Off;
      Sensing { hold_unless (shutdown)(sensing)
                || ident (~shutdown & cfg)(cfg_code) }
        -> Off with off & ok / ack, off_code, shutdown;
    }
  }
  || ident (read0) (read0_code)
  || ident (read1) (read1_code)
}
```

```
extern status_t read0_code (tsl2550_res_t *res) {
  i2c_write (TSL2550_I2C_ADDR, TSL2550_CMD_ADC0);
  *res = i2c_read (TSL2550_I2C_ADDR);
  if (*res & 0x80)
    {
      return -EFAIL;
    }
  *res &= 0x7f;
  return SUCCESS;
}
```

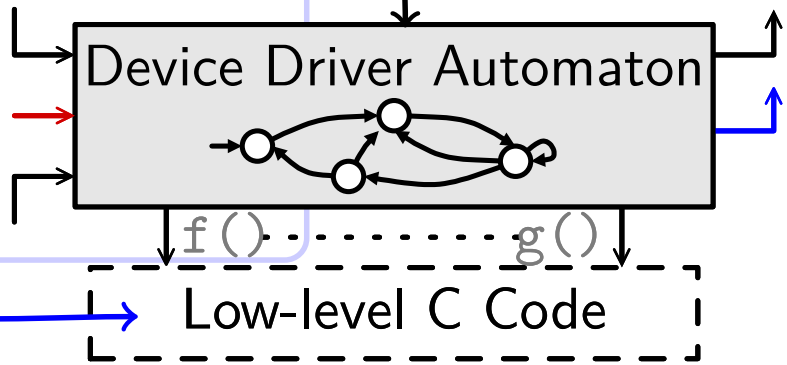


Synchronous Device Driver: Example

```
process tsl2550 (on, off, cfg, read0, read1, ok) (ack, sensing) {
  internal shutdown {
    automaton {
      init Off
      Off
        -> Sensing with on & ok / ack, on_code, sensing;
        +> Off;
      Sensing { hold_unless (shutdown)(sensing)
                || ident (~shutdown & cfg)(cfg_code) }
        -> Off with off & ok / ack, off_code, shutdown;
    }
  }
  || ident (read0) (read0_code)
  || ident (read1) (read1_code)
}
```

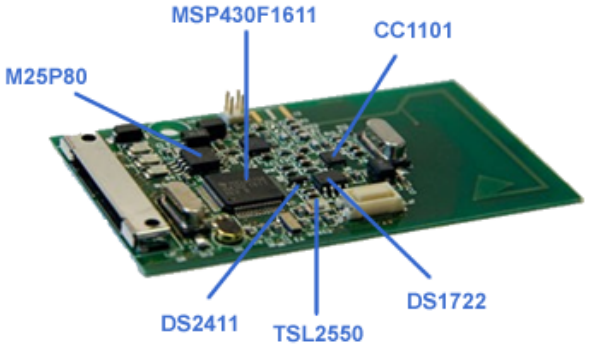
(Function Call)

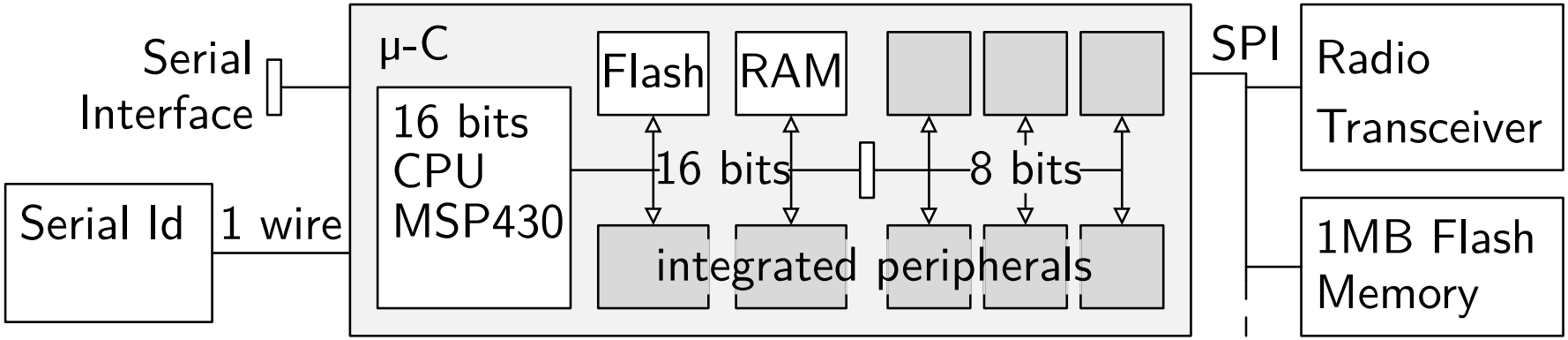
```
extern status_t read0_code (tsl2550_res_t *res) {
  i2c_write (TSL2550_I2C_ADDR, TSL2550_CMD_ADC0);
  *res = i2c_read (TSL2550_I2C_ADDR);
  if (*res & 0x80)
    {
      return -EFAIL;
    }
  *res &= 0x7f;
  return SUCCESS;
}
```



Proof-of-Concept Implementation

- Targeting Wsn430 Wireless Sensor Network Nodes
- Device Drivers & Controller Encoded in ARGOS
  - $\sim$ 10 Device Drivers
- Adapted Operating Systems:
  - Home-made Multithreaded
  - CONTIKI





Outline

Evaluation Methodology

Global Control $\Rightarrow$ Overhead

Evaluation Methodology

Global Control $\Rightarrow$ Overhead

Quantitative Evaluation

- We Cannot Compare with a Solution for *Global Control*
  - No Available Figures about *Global Device Manager* by [Choi *et al.*]

Evaluation Methodology

Global Control $\Rightarrow$ Overhead

Quantitative Evaluation

- We Cannot Compare with a Solution for *Global Control*
  - No Available Figures about *Global Device Manager* by [Choi *et al.*]
- Our Solution
  - Identifiable Overhead

$\rightsquigarrow$ Comparing Orders of Magnitude

Evaluation Methodology

Global Control $\Rightarrow$ Overhead

Quantitative Evaluation

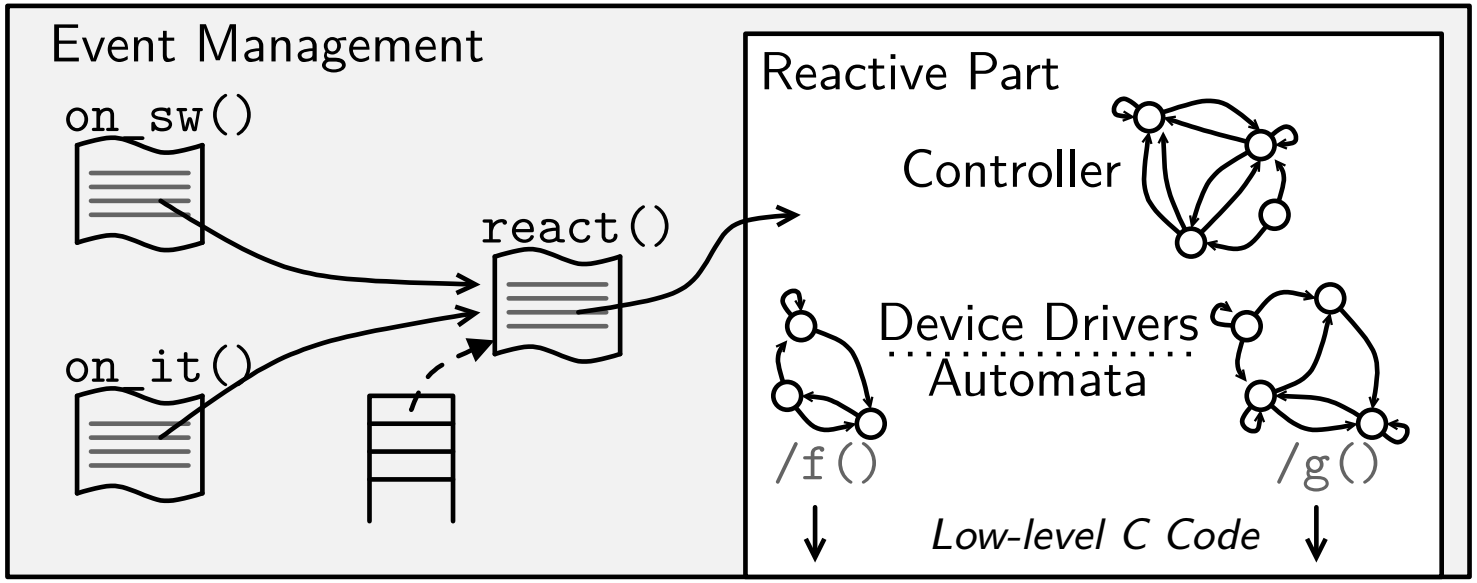
- We Cannot Compare with a Solution for *Global Control*
  - No Available Figures about *Global Device Manager* by [Choi *et al.*]
- Our Solution
  - Identifiable Overhead

$\rightsquigarrow$ Comparing Orders of Magnitude

Qualitative Evaluation

- Impact on Reused Software
- Extensibility

Quantitative Evaluation

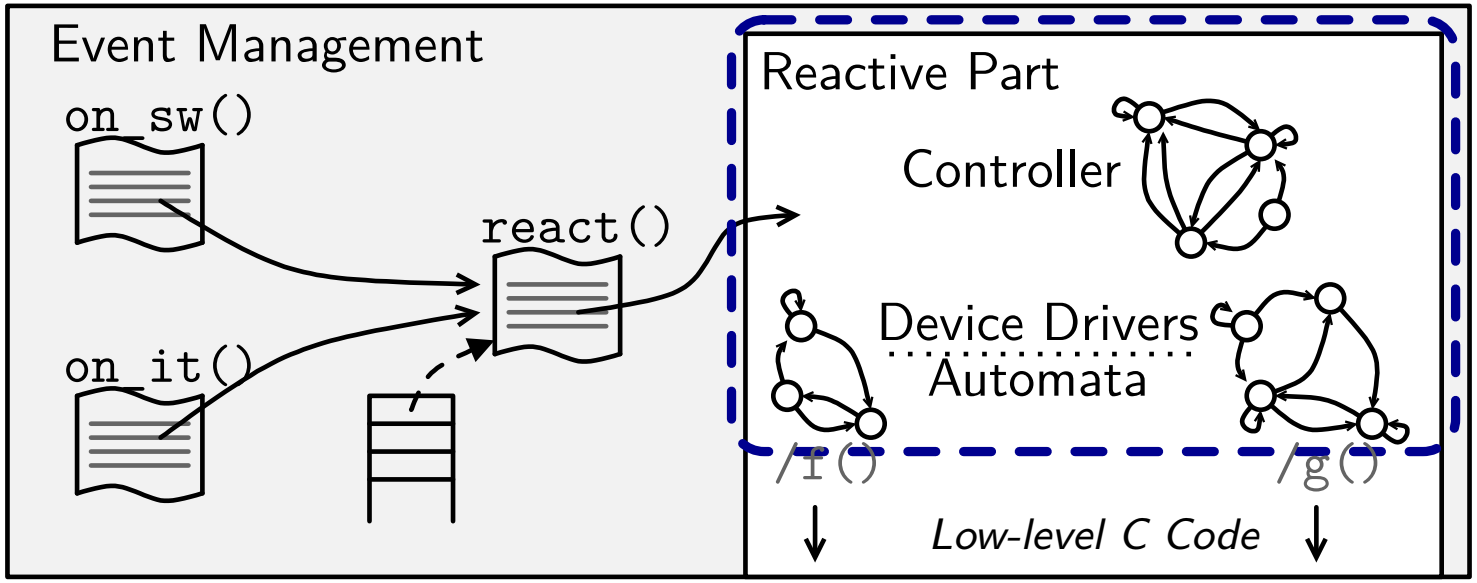


Memory Footprint Overhead Induced by the Control Layer

- Event Management: $\approx$ 1 KB
- Reactive Part without Low-level Code: $\approx$ 2 KB

- Typical Footprint of Whole Software Stack $\sim$ 25 KB

Quantitative Evaluation



One Transition of the Reactive Part without Low-level Code

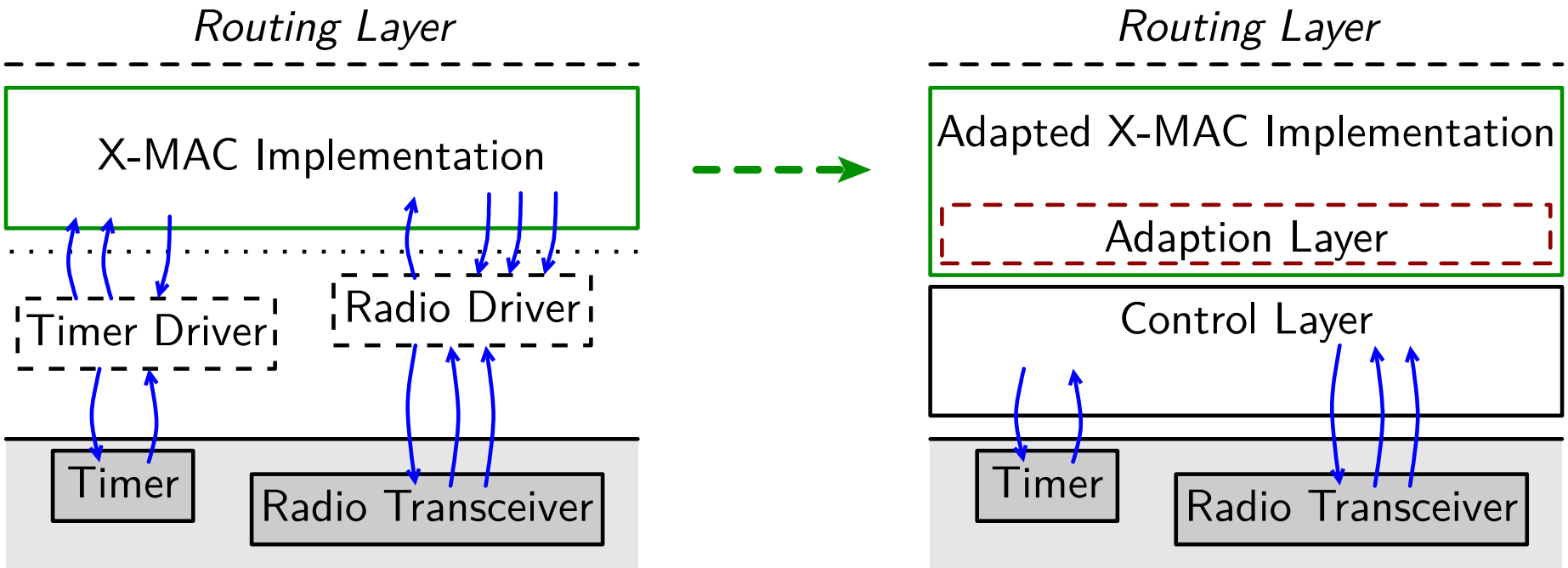
- ≈ 1,600 CPU Cycles (10 Device Drivers)

- ICEM's Solution for *Local* Control: ≈ 400 CPU Cycles per Arbiter

Evaluating the Impact on Reused Software

Setting up the Experiment

- Taking a Representative Piece of Existing Software
  - X-MAC Protocol Implementation
- Make it Use the Control Layer
  - Identify & Quantify Modifications

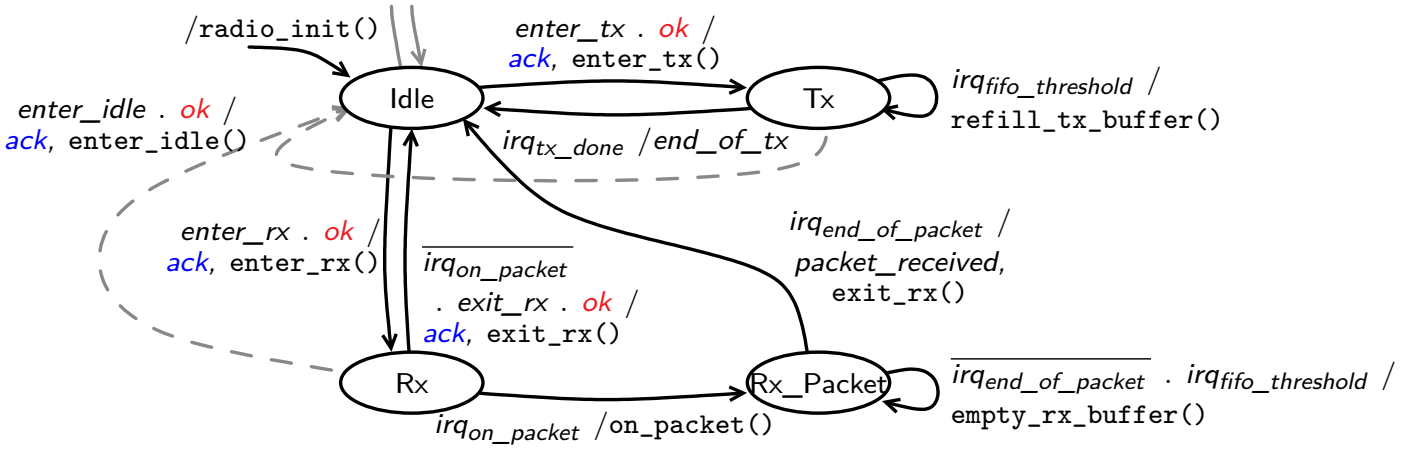


Evaluating the Impact on Reused Software *(cont'd)*

Example Function

[SensTools]

```
static uint16_t send_data (void) {
  /* Stop Preamble Timer:                                          */
  timerB_unset_alarm (ALARM_PREAMBLE);

  /* Goto Idle Operating Mode:                                     */
  cc1100_cmd_idle ();
  cc1100_cmd_flush_rx (); cc1100_cmd_flush_tx ();

  /* Start Transmission:                                           */
  cc1100_cmd_tx ();
  cc1100_fifo_put ((uint8_t*)(&txframe.length),
                   txframe.length+1);

  /* Setup Function to be Executed Upon End of Transmission:       */
  cc1100_gdo0_register_callback (send_done);
  return 0;
}
```

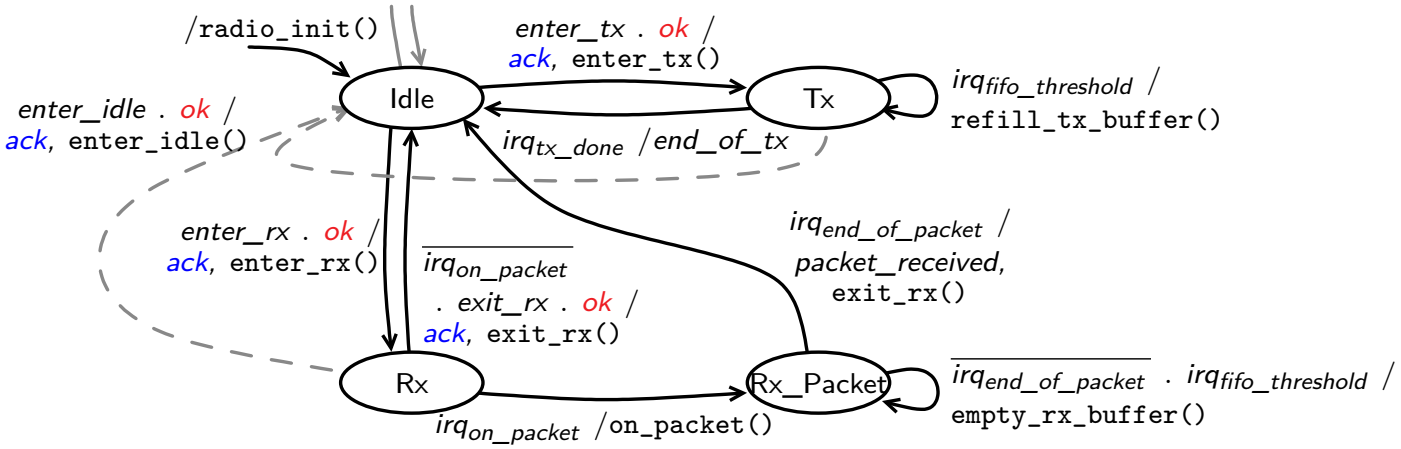


Evaluating the Impact on Reused Software *(cont'd)*

Example Function

[SensTools]

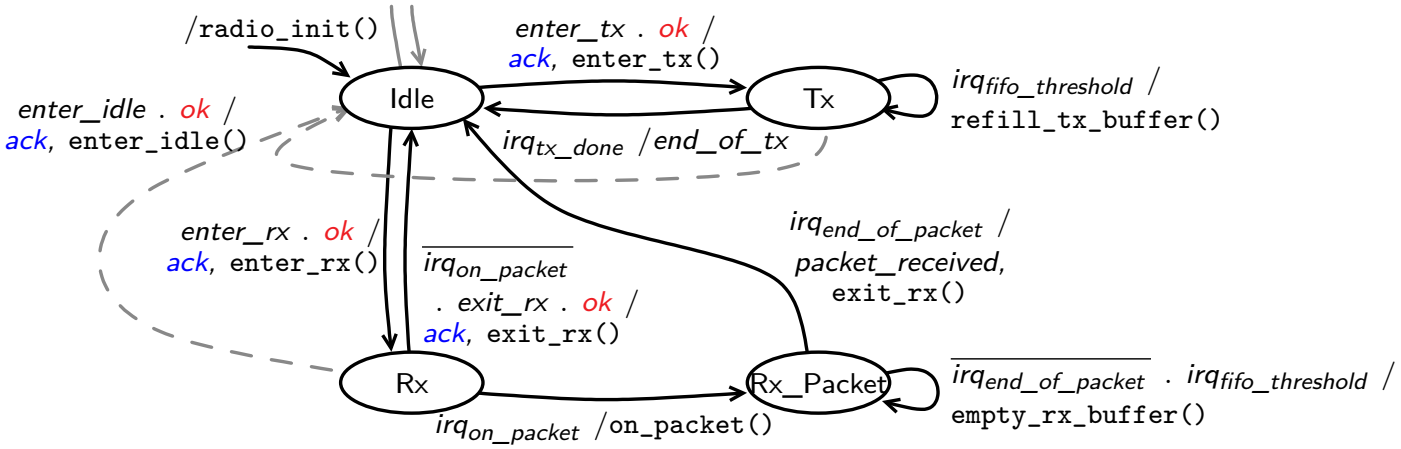
```
static uint16_t send_data (void) {
  /* Stop Preamble Timer:                                          */
  timerB_unset_alarm (ALARM_PREAMBLE);

  /* Goto Idle Operating Mode:                                     */
  cc1100_cmd_idle ();  ← enter_idle Command
  cc1100_cmd_flush_rx (); cc1100_cmd_flush_tx ();

  /* Start Transmission:                                           */
  cc1100_cmd_tx ();  ← enter_tx Command
  cc1100_fifo_put ((uint8_t*)(&txframe.length),
                   txframe.length+1);

  /* Setup Function to be Executed Upon End of Transmission:       */
  cc1100_gdo0_register_callback (send_done);  ← Register end_of_tx Notification Handler
  return 0;
}
```



Evaluating the Impact on Reused Software *(cont'd)*

Adapted Function

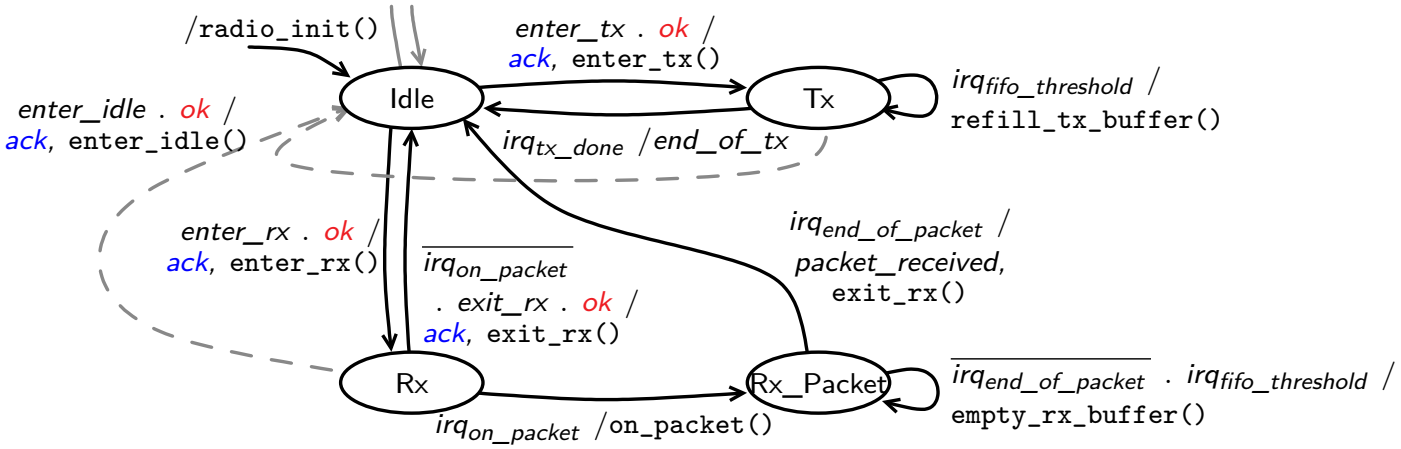
```
static uint16_t send_data (void) {
  /* Stop Preamble Timer:                                          */
  timerB_unset_alarm (ALARM_PREAMBLE);

  /* Goto Idle Operating Mode:                                     */
  if (radio_idle () != SUCCESS)
    if (error_cb) return error_cb ();

  /* Start Transmission:                                           */
  if (radio_send ((uint8_t*)(&txframe.length), txframe.length+1)
      != SUCCESS)
    if (error_cb) return error_cb ();

  /* Setup Function to be Executed Upon End of Transmission:       */
  radio_register_transmission_done_cb (send_done);
  return 0;
}
```



Evaluating the Impact on Reused Software *(cont'd)*

Adapted Function

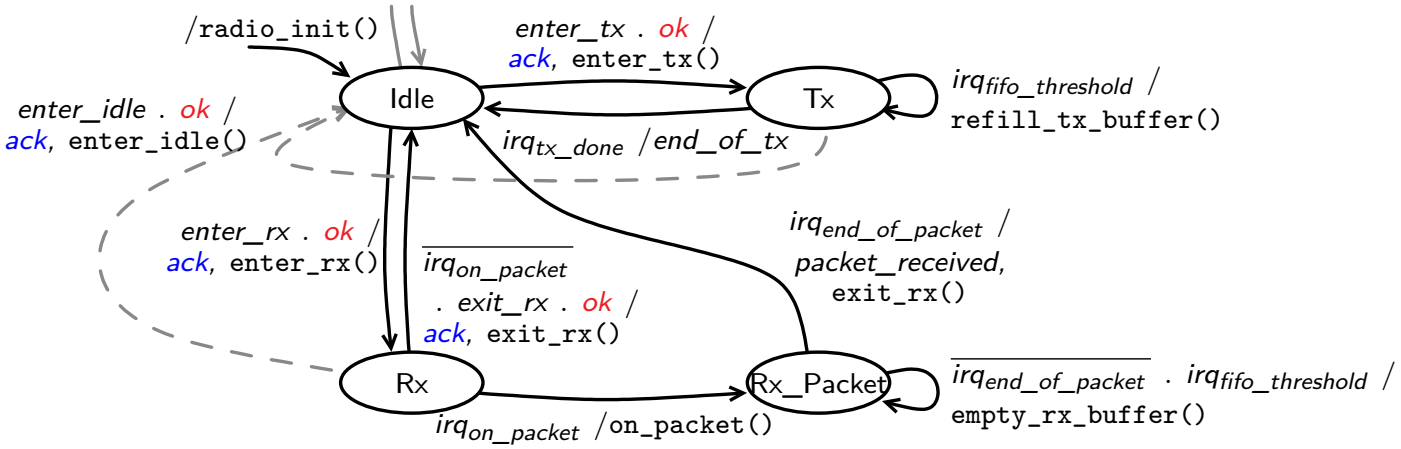
```
static uint16_t send_data (void) {
  /* Stop Preamble Timer:                                                    */
  timerB_unset_alarm (ALARM_PREAMBLE);

  /* Goto Idle Operating Mode:                                               */
  if (radio_idle () != SUCCESS)
    if (error_cb) return error_cb ();

  /* Start Transmission:                                                     */
  if (radio_send ((uint8_t*)(&txframe.length), txframe.length+1)
      != SUCCESS)
    if (error_cb) return error_cb ();

  /* Setup Function to be Executed Upon End of Transmission:                 */
  radio_register_transmission_done_cb (send_done);
  return 0;
}
```



Evaluating the Impact on Reused Software *(cont'd)*

A Function of the *Adaptation Layer*

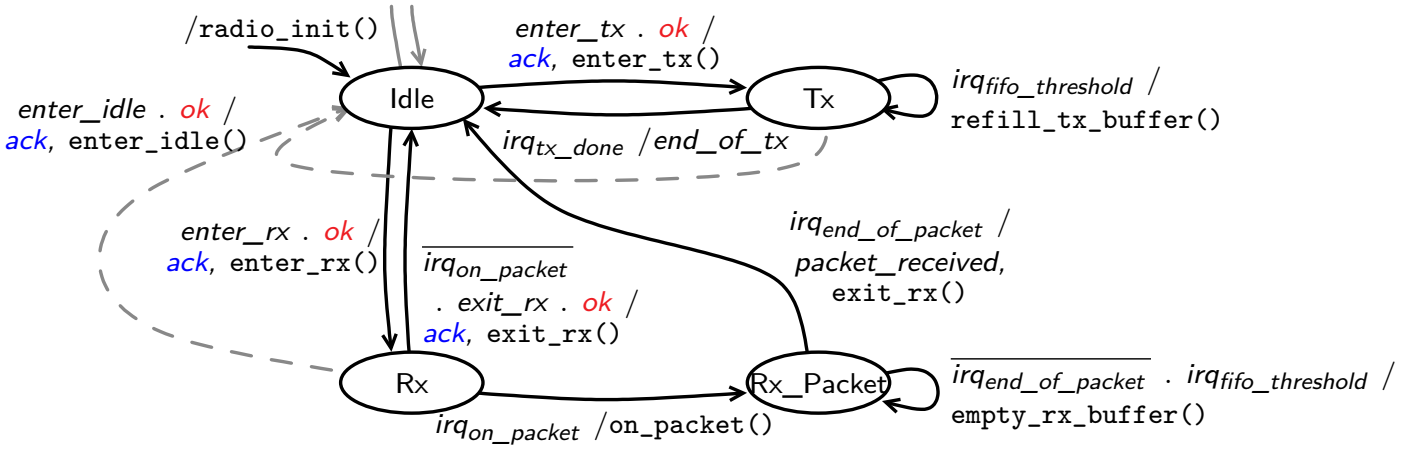
```
extern status_t radio_idle (void) {

  cl_outputs_v notifications;

  /* Emit Software Request to the Control Layer:                    */
  on_sw (cc1100_idle, & notifications);

  /* Test Acknowledgment:                                           */
  return cl_outputs_test (notifications, cc1100_ack)
         ? SUCCESS : -EFAIL;
}
```



Evaluating the Impact on Reused Software *(cont'd)*

A Function of the *Adaptation Layer*

```
extern status_t radio_idle (void) {

  cl_outputs_v notifications;

  /* Emit Software Request to the Control Layer:                    */
  on_sw (cc1100_idle, & notifications);

  /* Test Acknowledgment:                                           */
  return cl_outputs_test (notifications, cc1100_ack)
         ? SUCCESS : -EFAIL;
}
```



Evaluating the Impact on Reused Software *(cont'd)*

Generalization

Quantifying the Modifications

- 28 Commands $\rightsquigarrow$ Calls to 9 Functions of the Adaptation Layer
- Preserved 85% of the 700 Original Lines of Code

Evaluating the Impact on Reused Software *(cont'd)*

Generalization

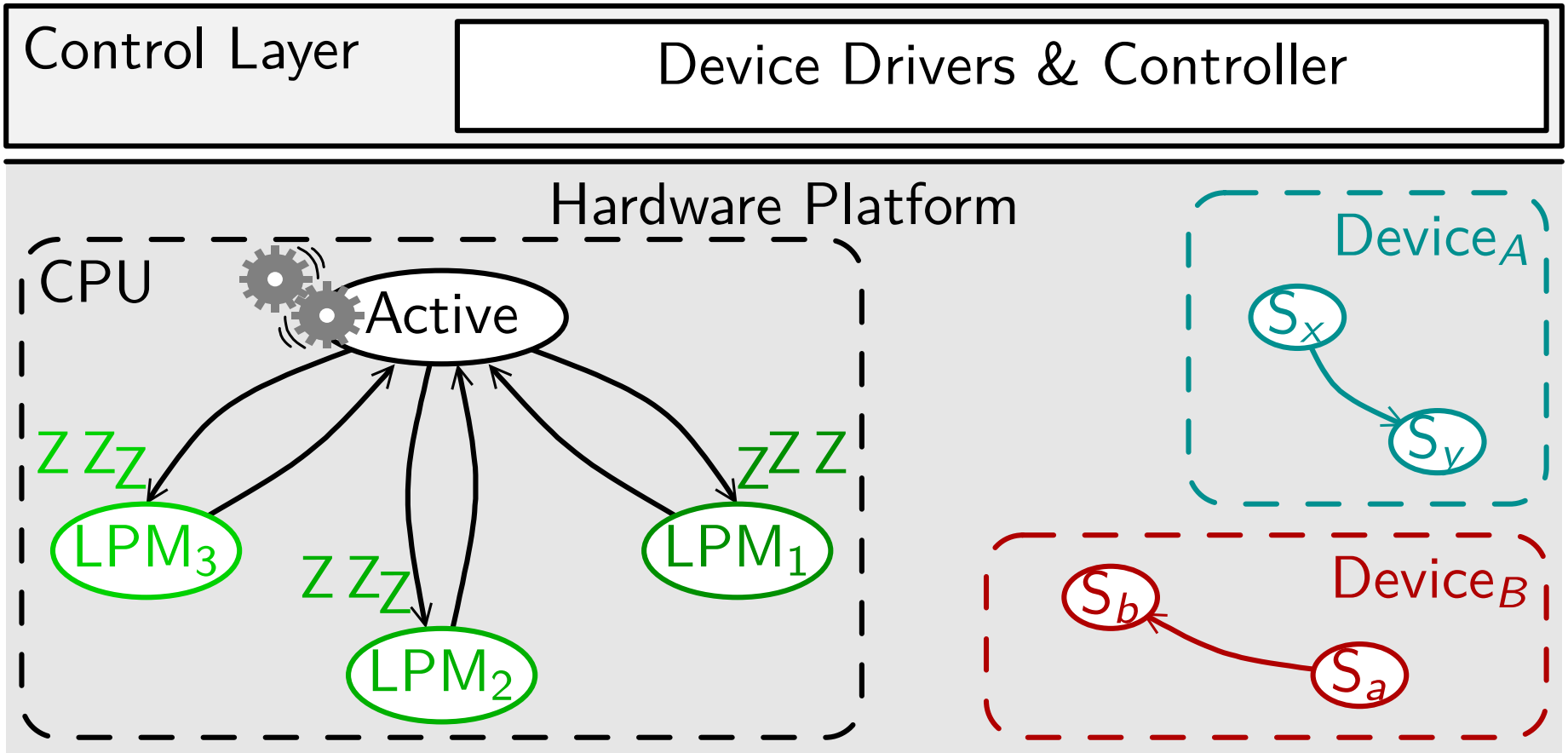
Quantifying the Modifications

- 28 Commands $\rightsquigarrow$ Calls to 9 Functions of the Adaptation Layer
- Preserved 85% of the 700 Original Lines of Code

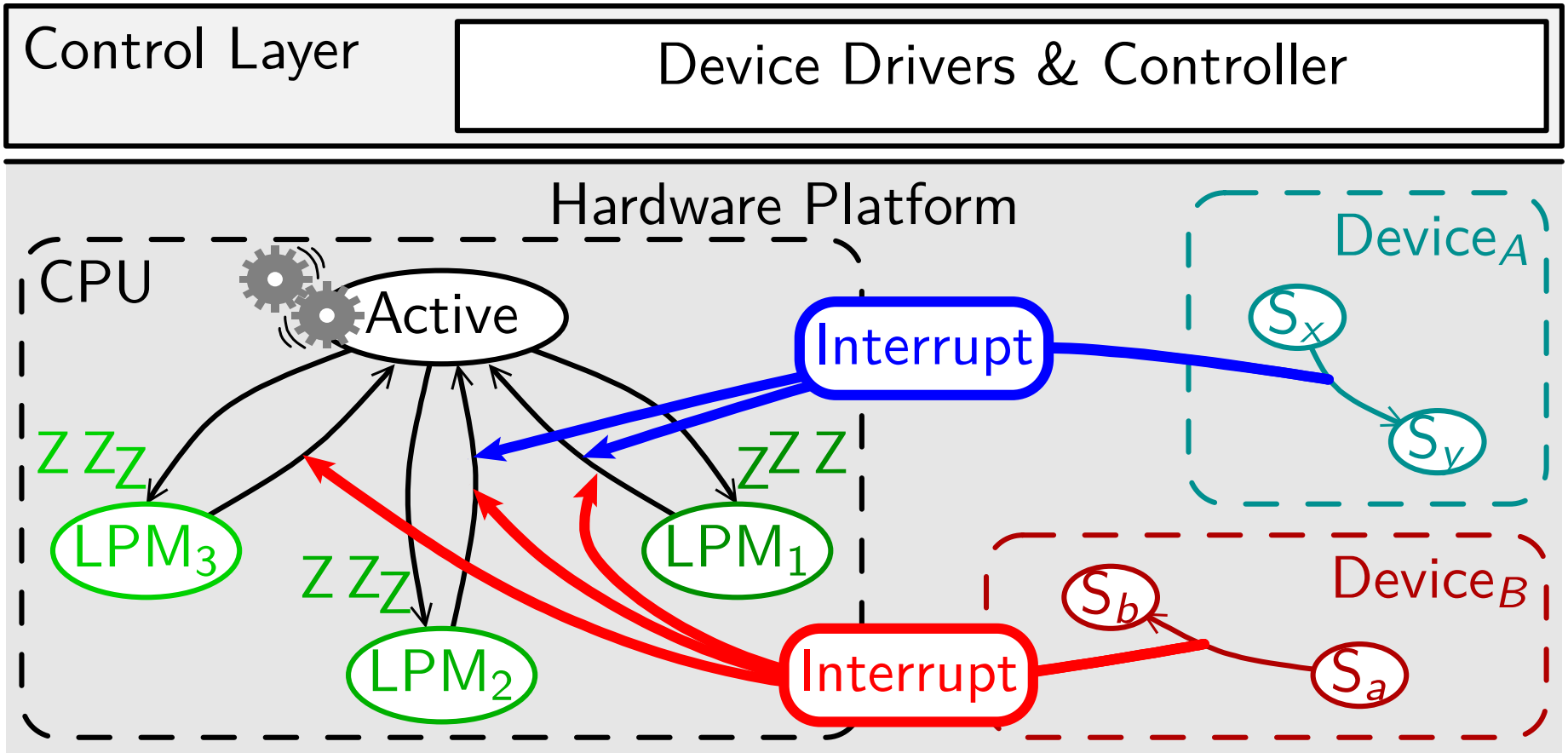
Methodology

- Identifying the *States* and *Transitions* of the Devices
- Representative Software $\rightsquigarrow$ Widely Applicable

Extensibility: Selecting the Best Low-Power Mode



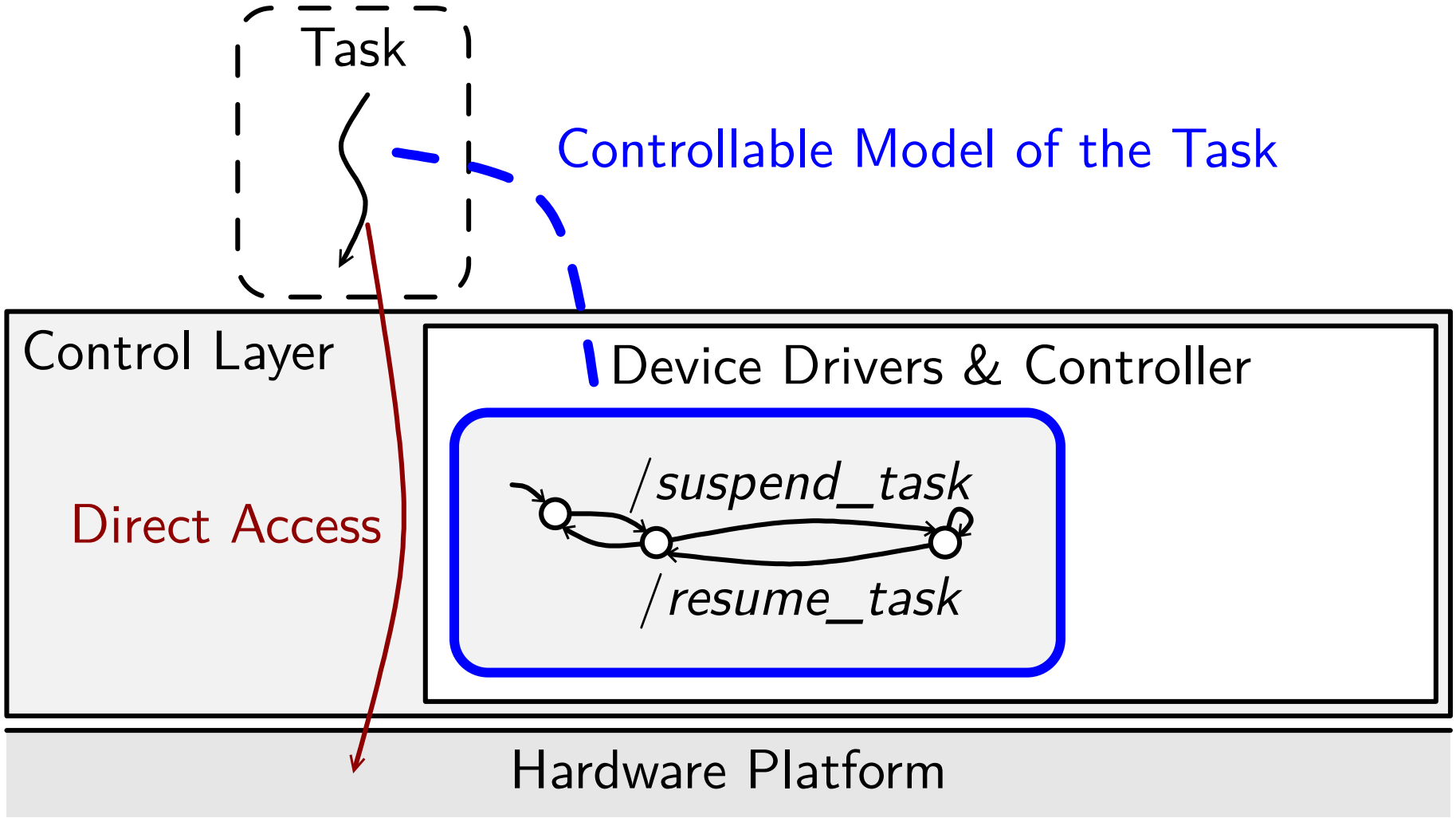
Extensibility: Selecting the Best Low-Power Mode



- Wake-up Guarantee (*e.g.*, State LPM_3 Should not be Reachable if $Device_B$ is not in State S_a)

$\rightsquigarrow$ Global Control Objective

Extensibility: Allowing Direct Access to the Devices



Outline

- Choosing an Application Domain and Outline of the Proposal
- Background
- Detailed Contribution
- Conclusion
  - Summary
  - Ongoing Work
  - Prospective

Summary

- New Software Architecture for *Global Control*

- Recall: *Global Control* $\Rightarrow$ Refused Requests

Recall: Objectives for the Solution

- Reusing Existing Software with Minimal Impact
- Reducing Cost/Overhead
- Guaranteeing Correctness
- Impact on Programming Model
  - Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

Summary

- New Software Architecture for *Global Control*

- Recall: *Global Control* ⇒ Refused Requests

Recall: Objectives for the Solution

- ✓ Reusing Existing Software with Minimal Impact
- Reducing Cost/Overhead
- Guaranteeing Correctness
- Impact on Programming Model
  - Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

Summary

- New Software Architecture for *Global Control*

- Recall: *Global Control* $\Rightarrow$ Refused Requests

Recall: Objectives for the Solution

- ✓ Reusing Existing Software with Minimal Impact
- ✓ Reducing Cost/Overhead
- Guaranteeing Correctness
- Impact on Programming Model
  - Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

Summary

- New Software Architecture for *Global Control*

- Recall: *Global Control* ⇒ Refused Requests

Recall: Objectives for the Solution

- ✓ Reusing Existing Software with Minimal Impact
- ✓ Reducing Cost/Overhead
- ✓ Guaranteeing Correctness
- Impact on Programming Model
  - Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

Summary

- New Software Architecture for *Global Control*

- Recall: *Global Control* ⇒ Refused Requests

Recall: Objectives for the Solution

- ✓ Reusing Existing Software with Minimal Impact
- ✓ Reducing Cost/Overhead
- ✓ Guaranteeing Correctness
- Impact on Programming Model
  - ✓ Avoiding Potential Deadlocks
  - Appropriate Diagnosis for Refused Requests

Summary

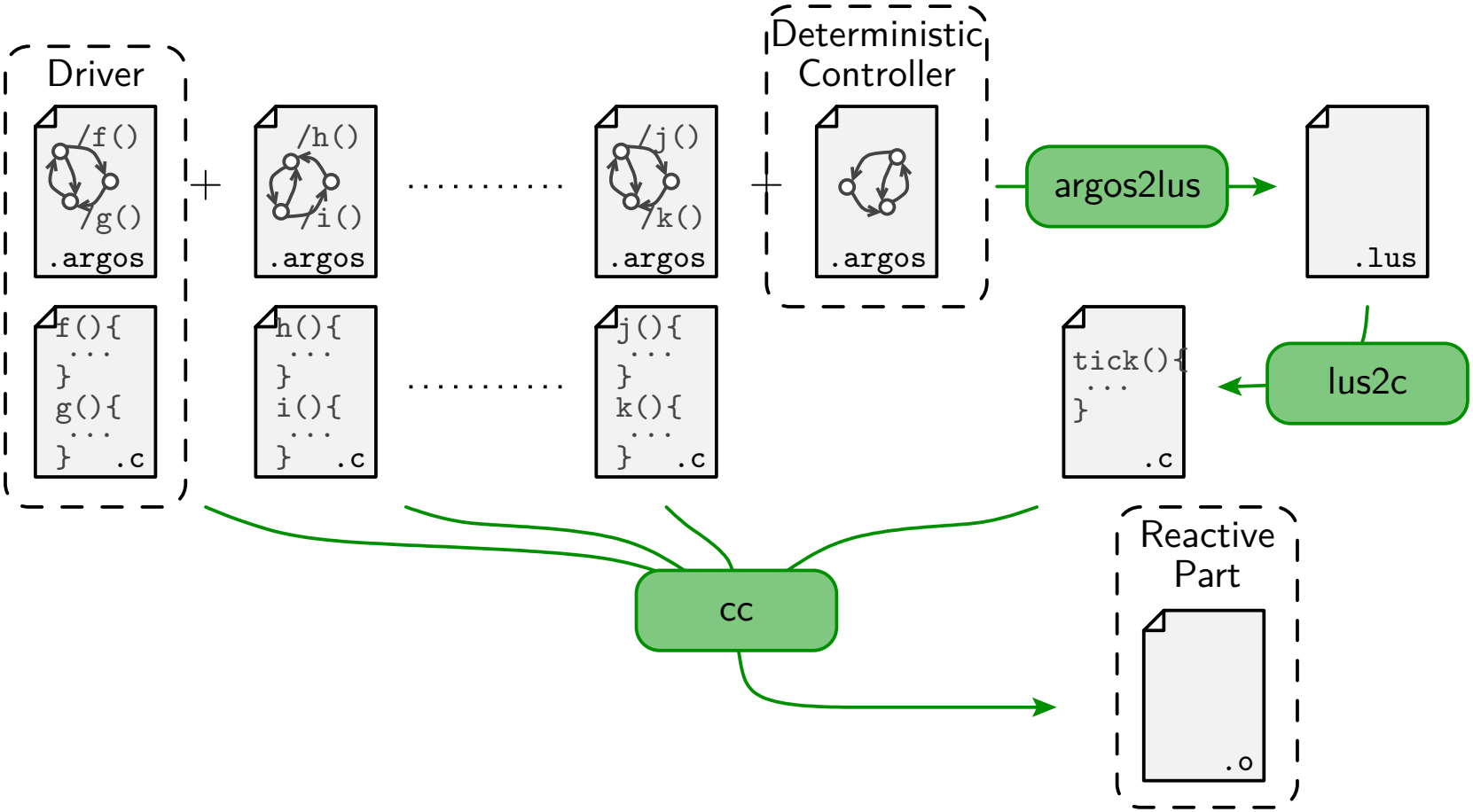
- New Software Architecture for *Global Control*

- Recall: *Global Control* $\Rightarrow$ Refused Requests

Recall: Objectives for the Solution

- ✓ Reusing Existing Software with Minimal Impact
- ✓ Reducing Cost/Overhead
- ✓ Guaranteeing Correctness
- ✓ Impact on Programming Model
  - ✓ Avoiding Potential Deadlocks
  - ✓ Appropriate Diagnosis for Refused Requests

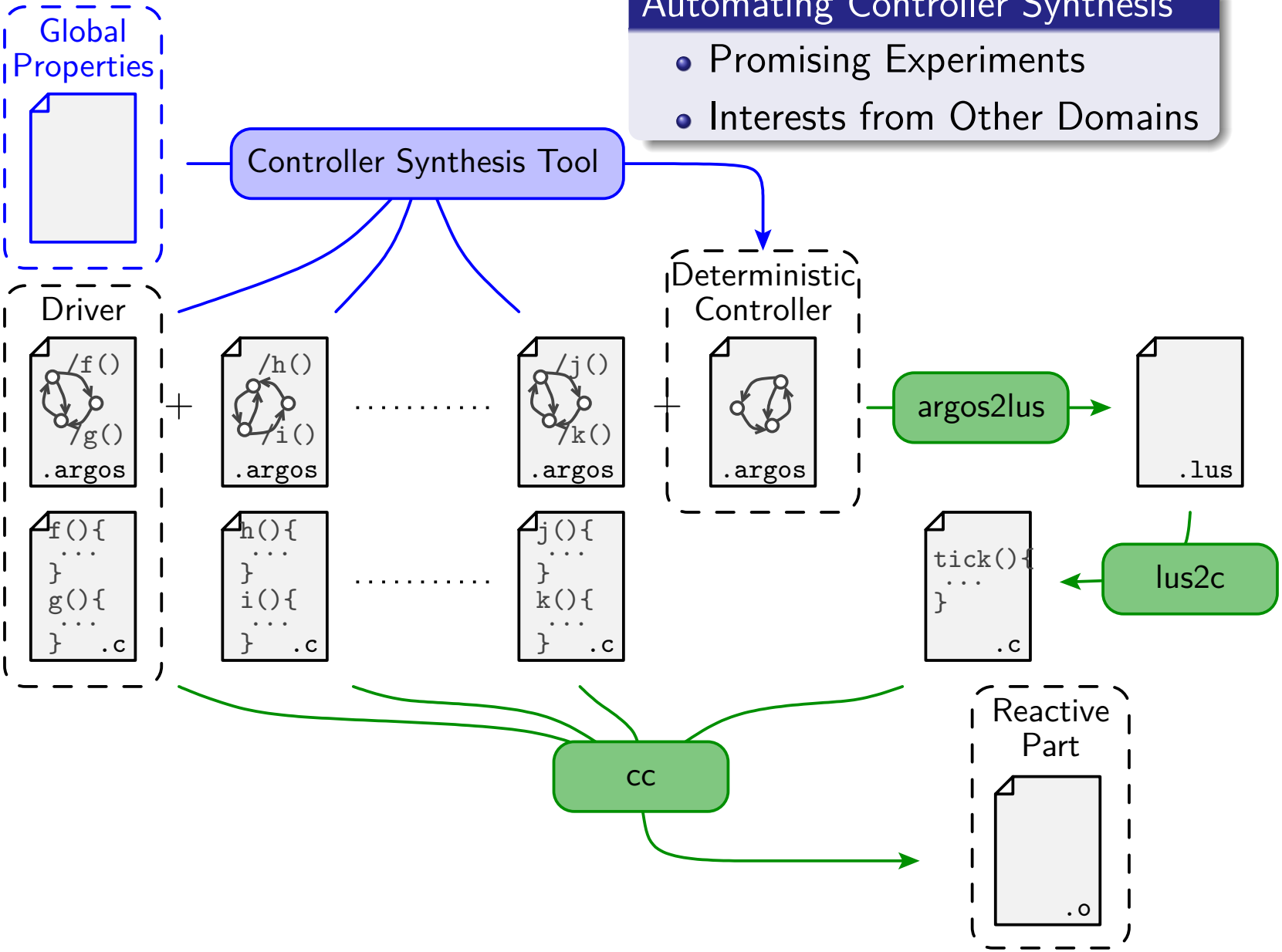
Ongoing Work: Ideal Tool-chain



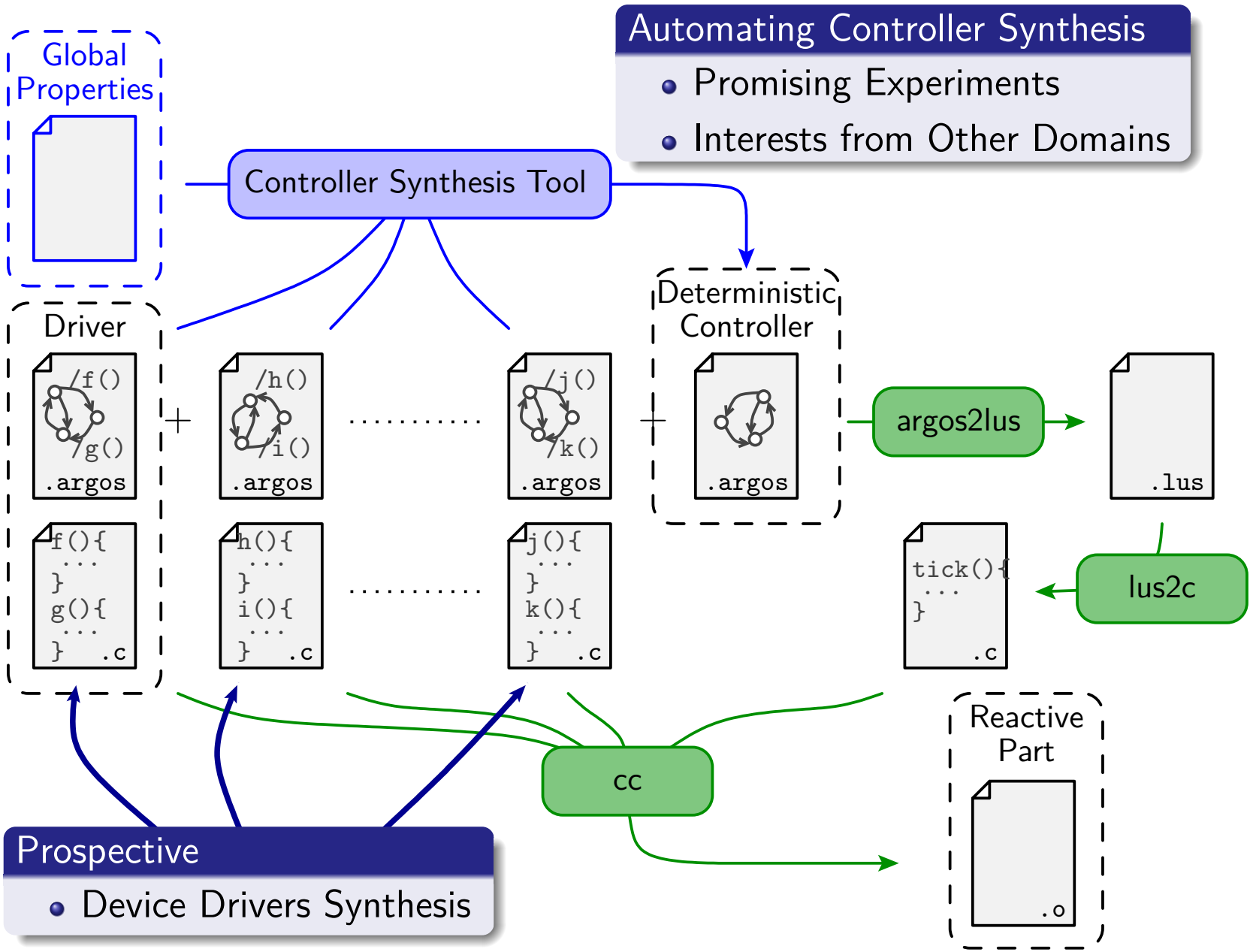
Ongoing Work: Ideal Tool-chain

Automating Controller Synthesis

- Promising Experiments
- Interests from Other Domains



Ongoing Work: Ideal Tool-chain



Automating Controller Synthesis

- Promising Experiments
- Interests from Other Domains

Prospective

- Device Drivers Synthesis

Ongoing Work: Impact on "Programming"

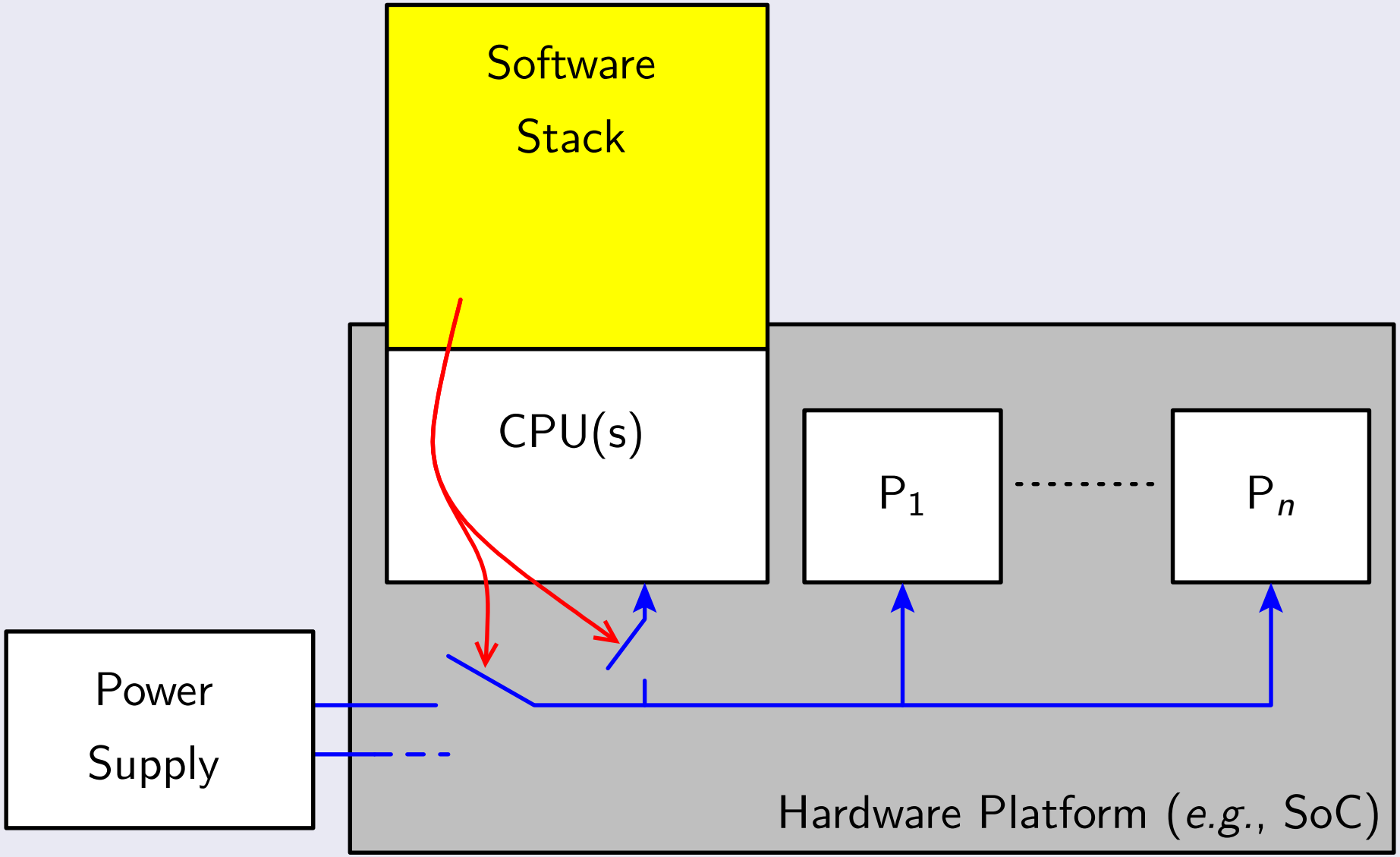
- Providing the Solution for Senslab Developers

Exploiting Diagnosis and Global Knowledge

- Network Protocols
  - Programming with *Refusals*
- Using the Models Now Available
  - Online Energy Estimation

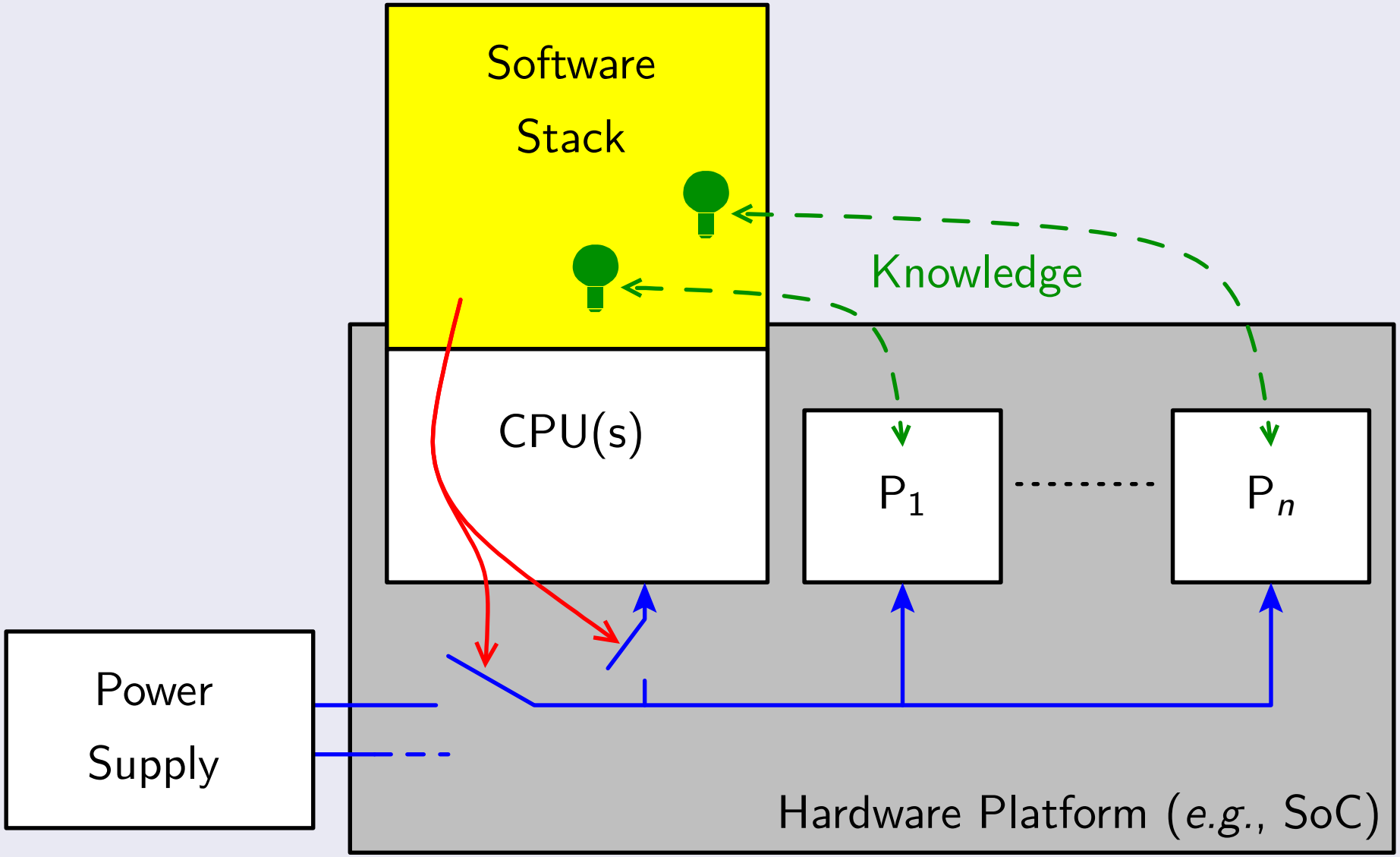
Generalization: Typical Use Case

The *Suspend Blockers* Case



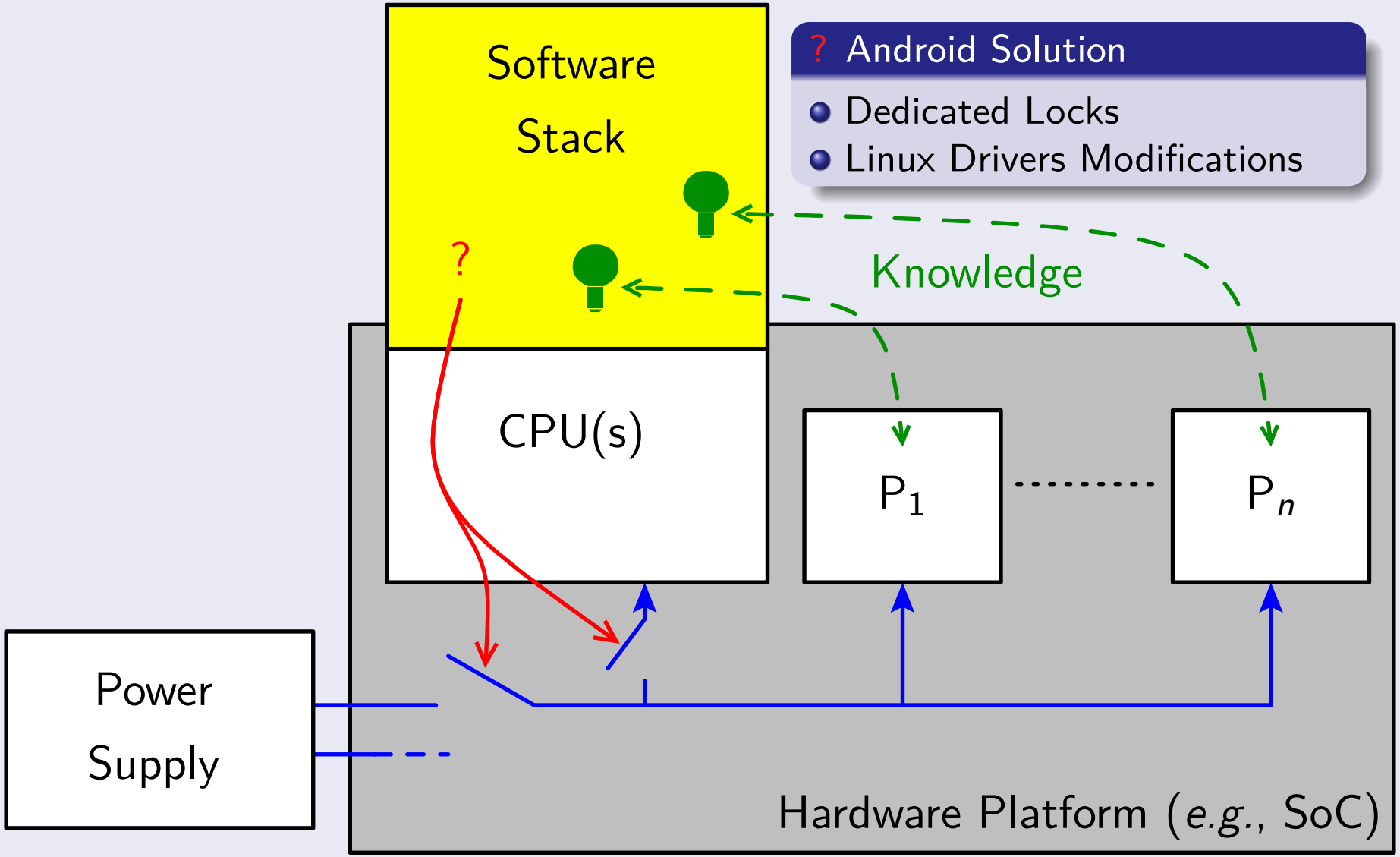
Generalization: Typical Use Case

The *Suspend Blockers* Case



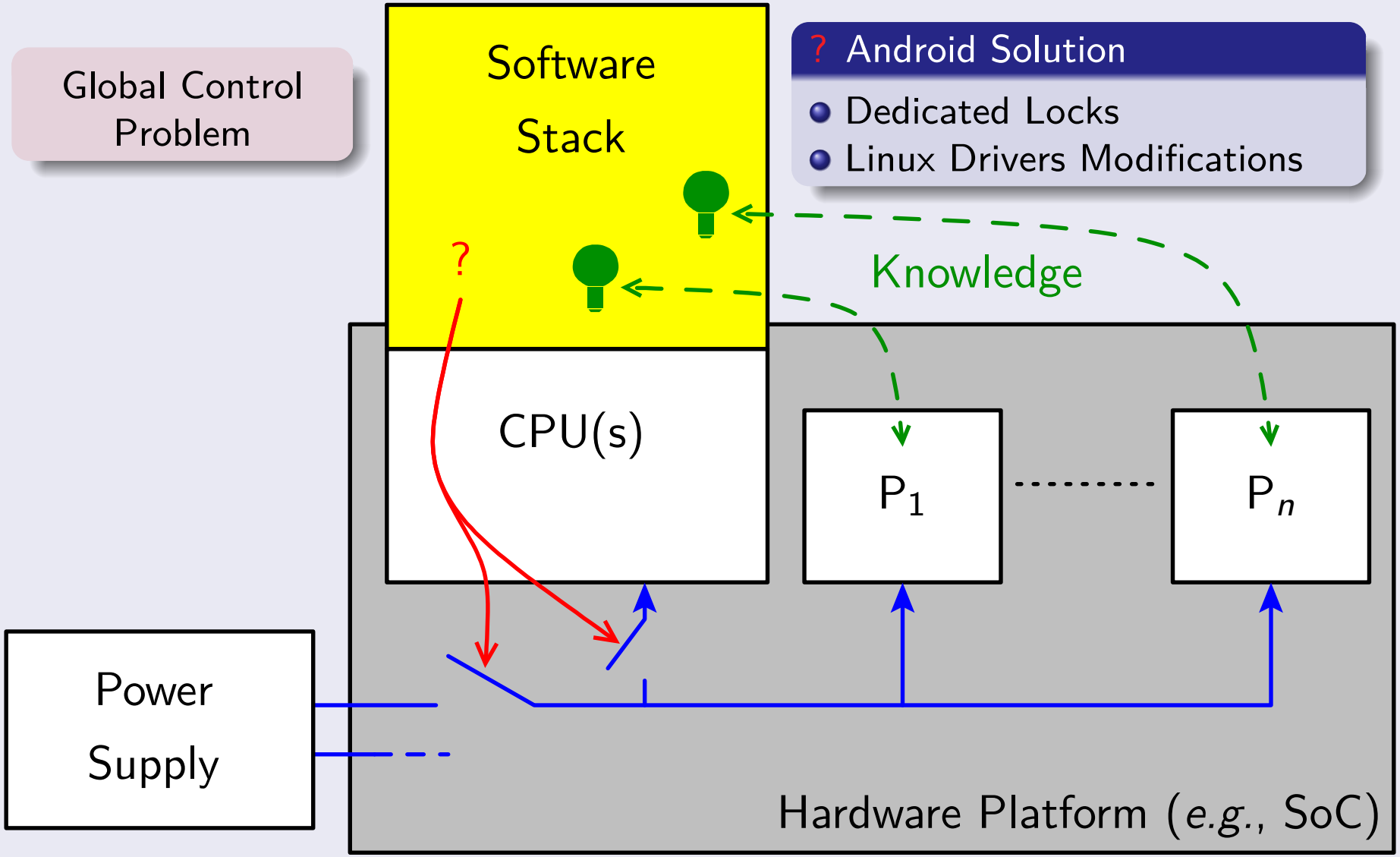
Generalization: Typical Use Case

The *Suspend Blockers* Case



Generalization: Typical Use Case

The *Suspend Blockers* Case



Thanks!

Publications

Nicolas Berthier, Florence Maraninchi, and Laurent Mounier.
Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems.
In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '11, pages 81–90, New York, NY, USA, 2011. ACM.

Nicolas Berthier, Florence Maraninchi, and Laurent Mounier.
Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems.
ACM Transactions on Embedded Computing Systems.
(Minor Revision).

Nicolas Berthier, Florence Maraninchi, and Laurent Mounier.
Global Platform Management by Using Synchronous Device Drivers in μ-Kernel-based Systems.
Fun Ideas and Thoughts Session — Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES), April 2011.

Outline

- Adaptation Layer
- Platform Control Server

Architecture: Adaptation Layer

Modified Part of the Operating System

- Simplified Device Drivers

Interacts with the Control Layer

- Emitting *Software Requests* to the Control Layer
  - Using `on_sw()`
- Receiving *Outputs* from the Control Layer
  - Notifications (Hardware Events, Acknowledgments)

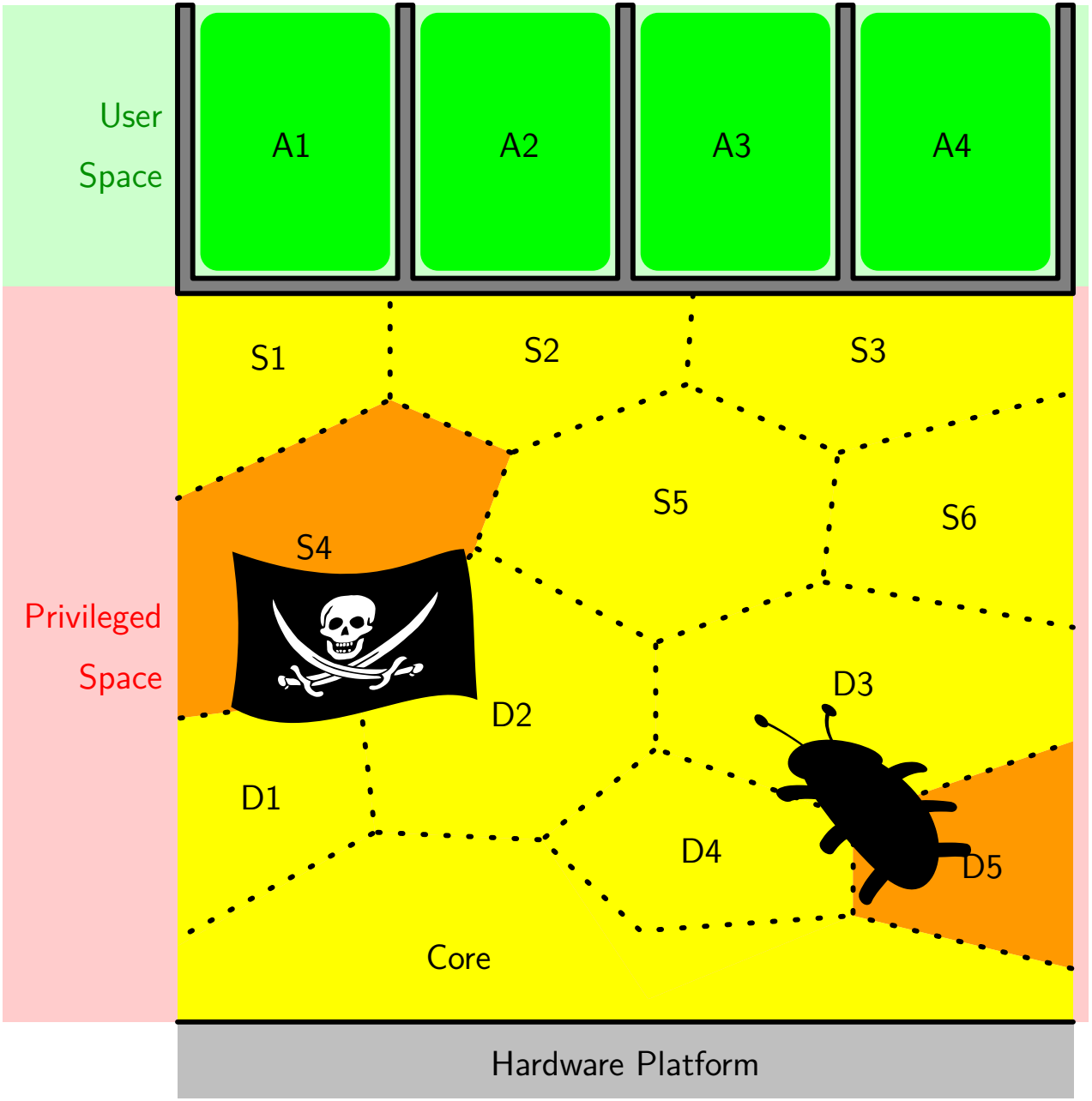
```
turn_adc_on ()
  /* Submit request 'adc_on' to the control layer.      */
  R = on_sw (adc_on);
  if (ack_a ∈ R) return success;
  /* Return an error code if request has been refused. */
  return error;
```

  - Virtual IRQs (Callbacks)

Outline

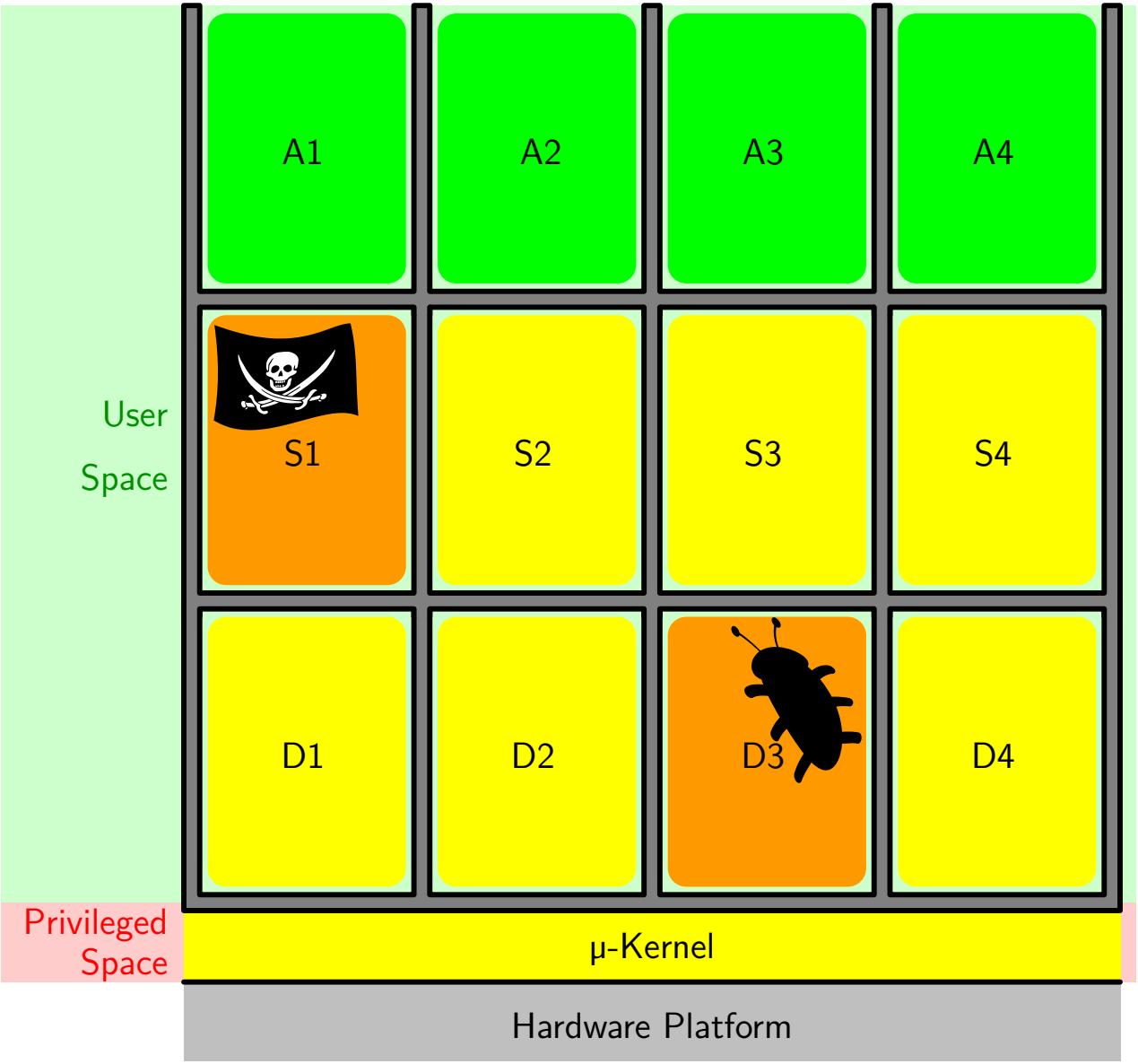
- Adaptation Layer
- Platform Control Server

Monolithic Kernel Systems



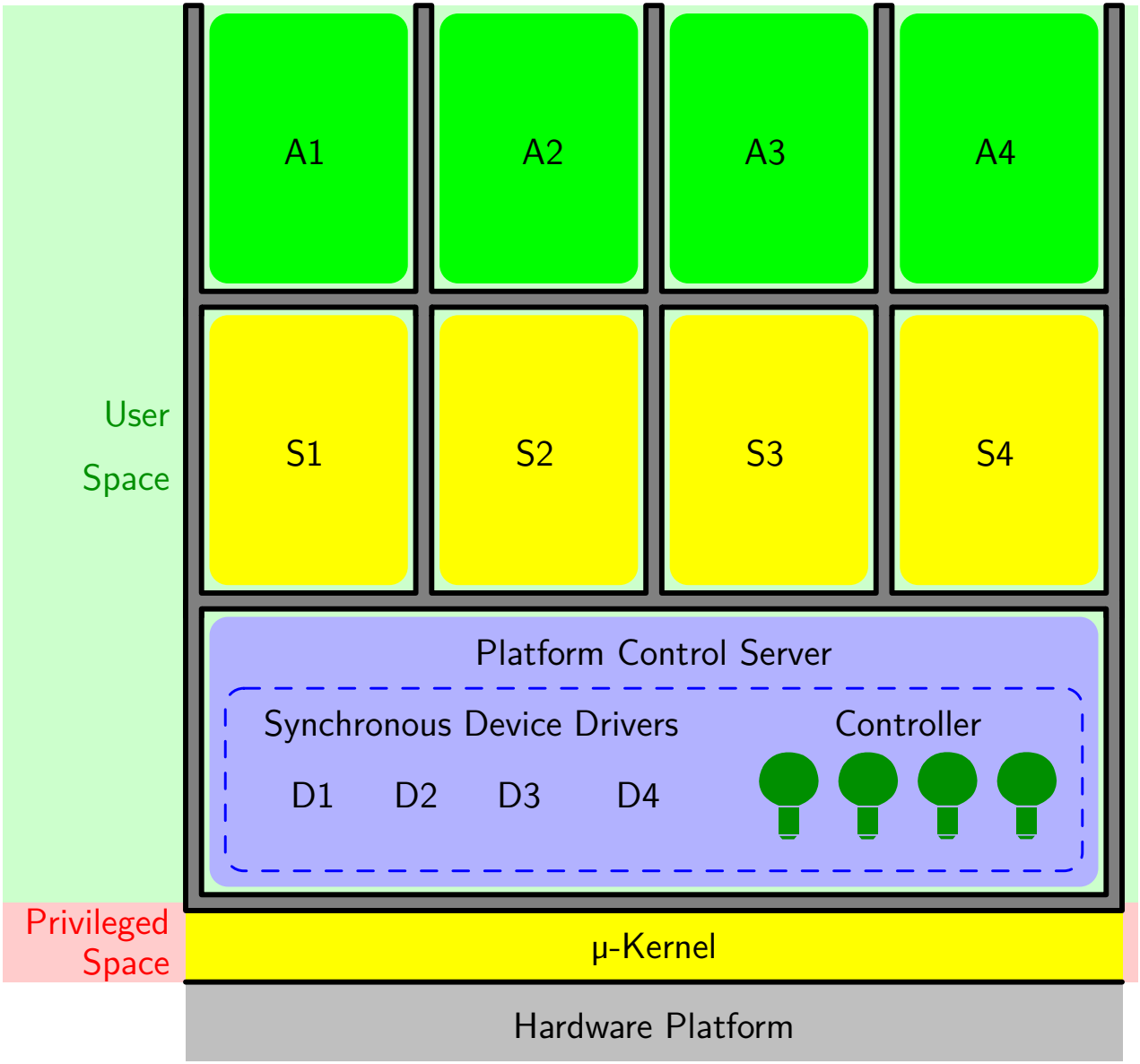
- Hardware Isolation
  - *User Task*-level
- "Huge" Amount of Privileged Code
- Examples: Linux, Windows' Kernel

µ-Kernel Systems



- Hardware Isolation
  - *Server*-level
- Neat Design
  - Architecture
  - Protocols
- "Small" Amount of Privileged Code
- Examples: Mach, L4

Proposal: Platform Control Server



- Neat Design

⇒ *Consistent View* of the Global State

- Possibly One Controller Making Decisions