Rage Against the State Machine: Type-Stated Hardware Peripherals for Increased Driver Correctness

Tyler Potyondy, Anthony Tarbinian, Leon Schuermann, Eric Mugnier, Adin Ackermann, Amit Levy, Pat Pannuto

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs Race Conditions

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs

Race Conditions

Memory Bugs

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs

Race Conditions

Memory Bugs

Concurrency Bugs

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs

Race Conditions

Memory Bugs

Device Protocol Bugs

Concurrency Bugs

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs


Race Conditions


Memory Bugs

Device Protocol Bugs


Concurrency Bugs

Kernel Device drivers...

- are >70% of OS code base
- written by many contributors
- source of many kernel exploits and faults

Logic Bugs


Race Conditions

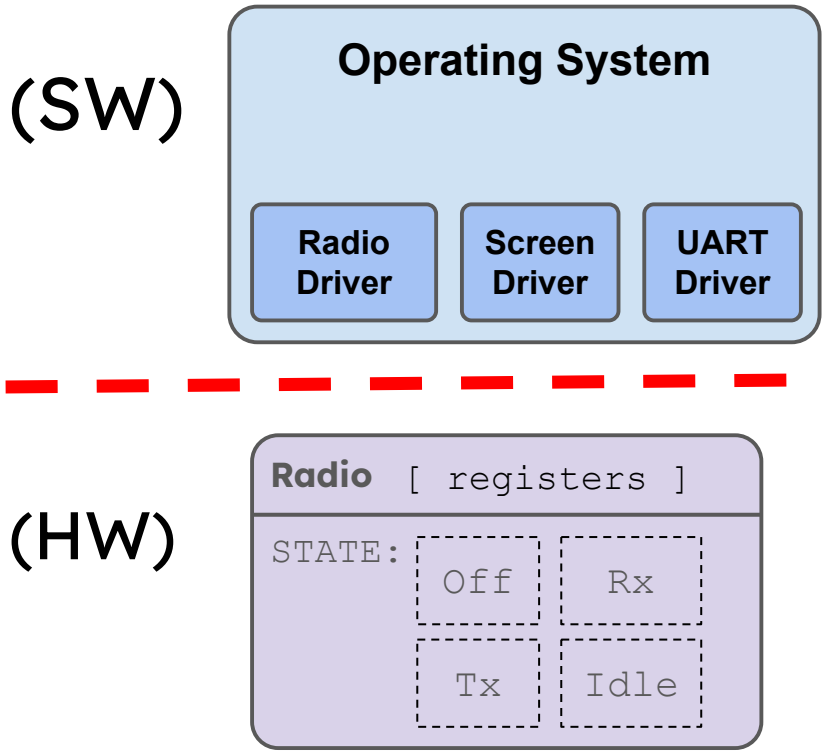

Memory Bugs

Device Protocol Bugs

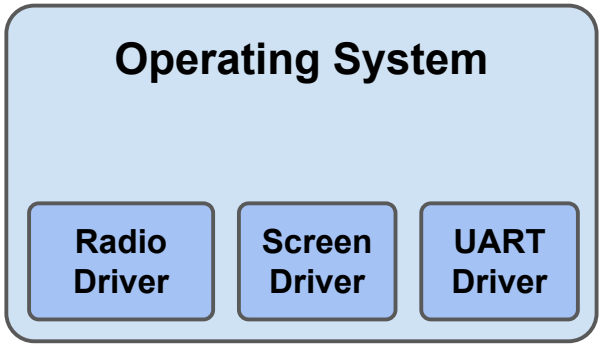

Concurrency Bugs

Device Protocol bugs have historically accounted for 38% of linux driver bugs!* **(highest driver bug class)**

*Ryzhyk2009

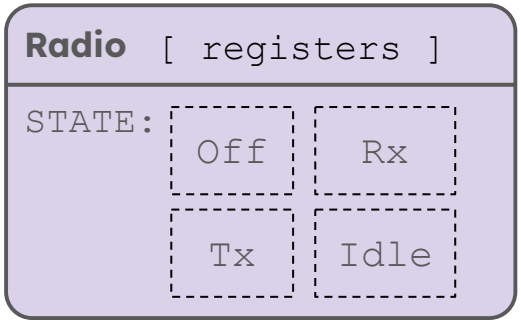

(SW)
Operating System
Radio Driver
Screen Driver
UART Driver
(HW)
Radio [registers]
STATE:
Off
Rx
Tx
Idle

(SW)



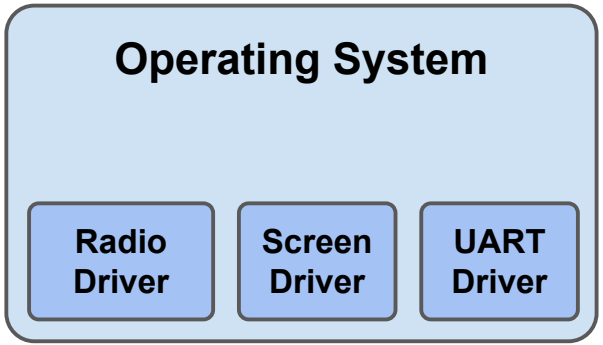
(Device Protocol)

(HW)



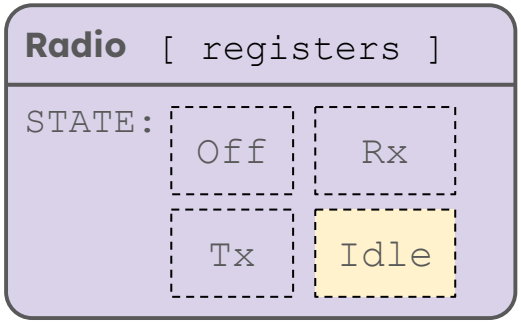
Must not interact with Radio registers when OFF.

(SW)



(Device Protocol)

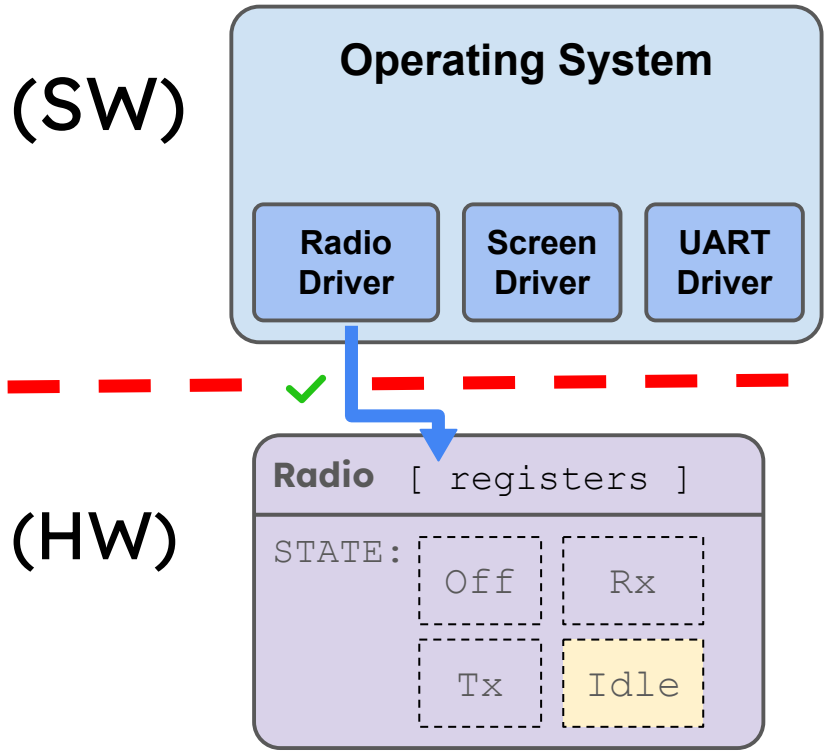
(HW)



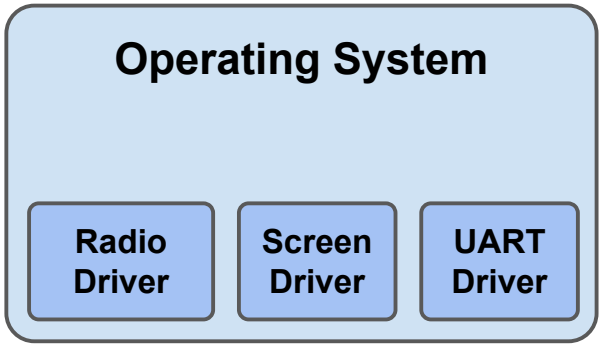
Must not interact with Radio registers when OFF.

(Device Protocol)

Must not interact with Radio registers when OFF.

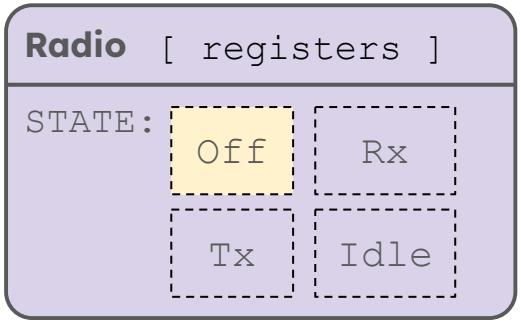


(SW)



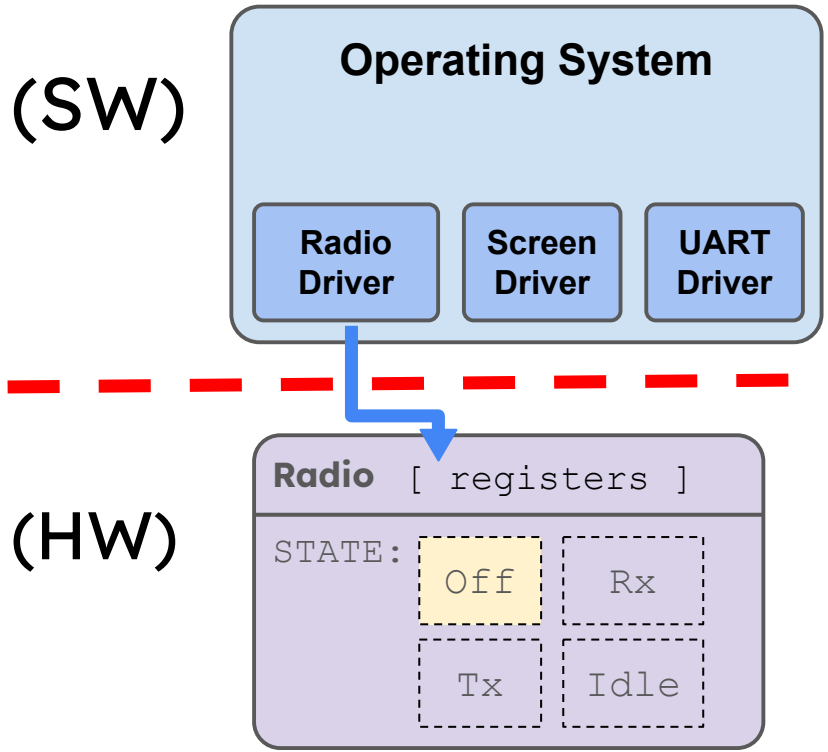
(Device Protocol)

(HW)

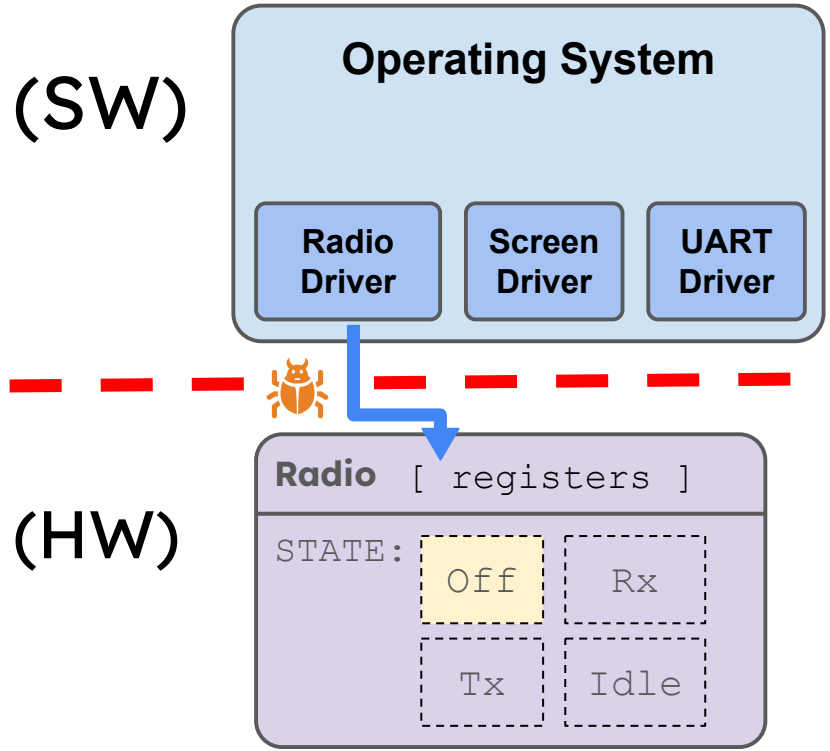


(Device Protocol)

Must not interact with Radio registers when OFF.



(SW)



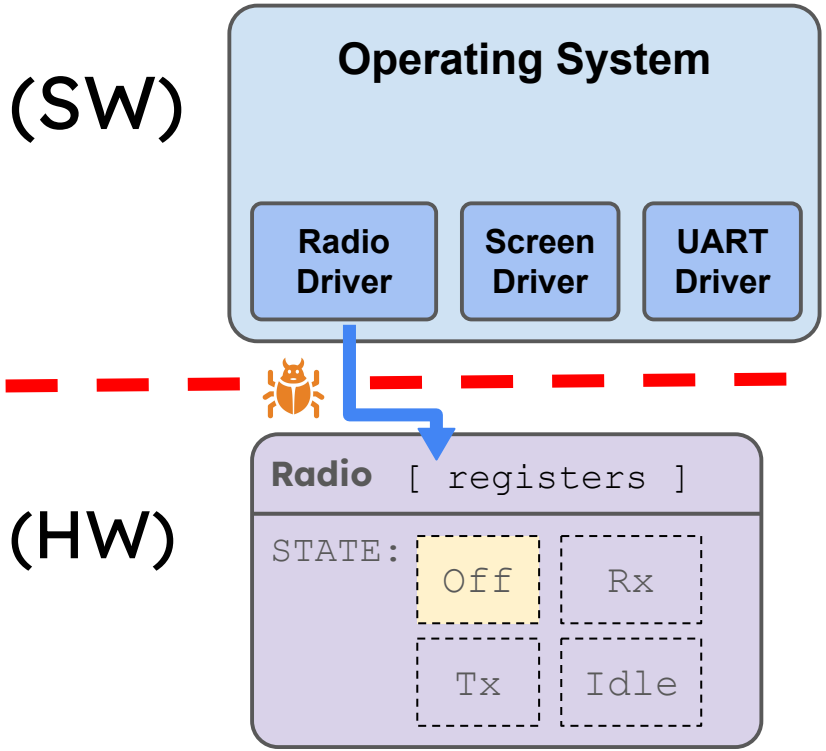
(HW)

(Device Protocol)

Must not interact with Radio registers when OFF.

Device Protocol Bug!

We present, **Abacus**, a framework that statically prevents device protocol violations.

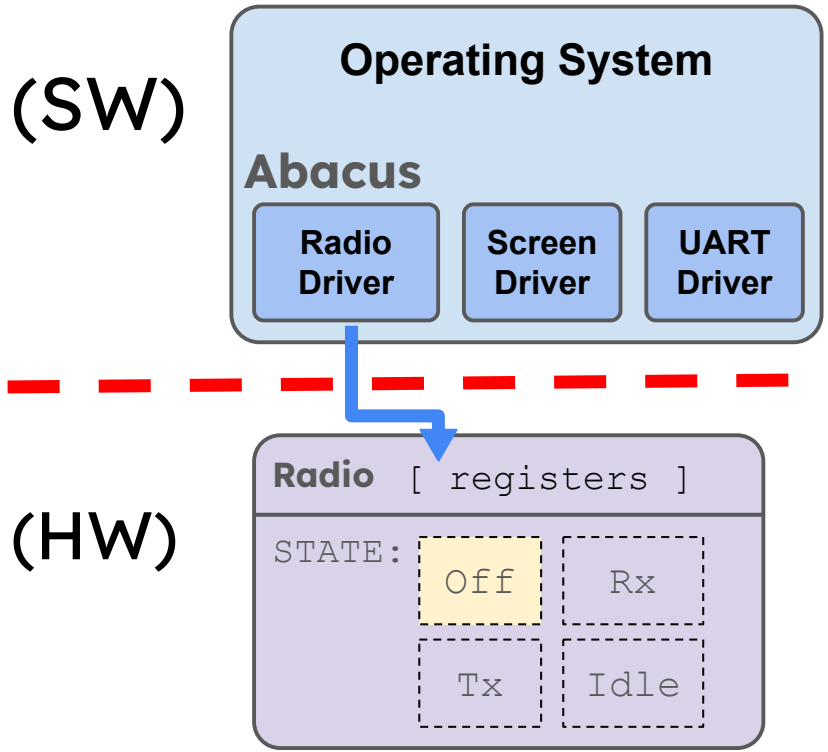


(Device Protocol)

Must not interact with Radio registers when OFF.

Device Protocol Bug!

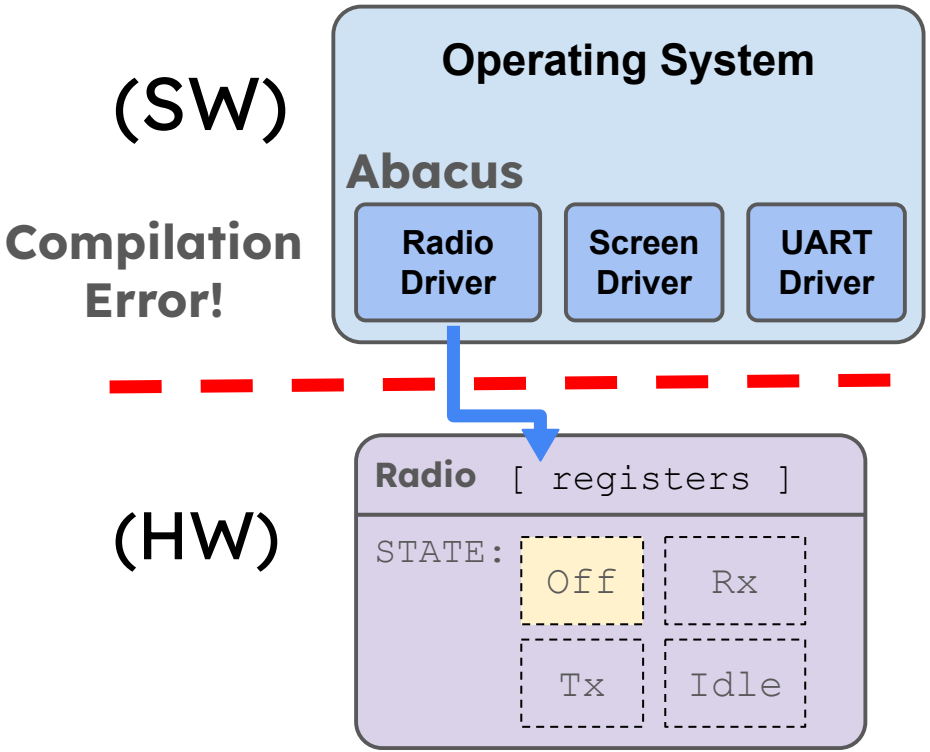
We present, **Abacus**, a framework that statically prevents device protocol violations.



(Device Protocol)

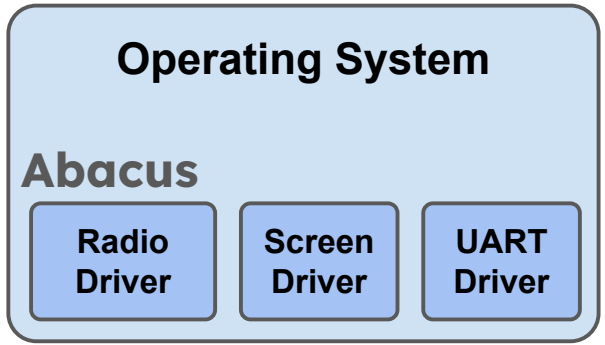
Must not interact with Radio registers when OFF.

We present, **Abacus**, a framework that statically prevents device protocol violations.



We present, **Abacus**, a framework that statically prevents device protocol violations.

(SW)

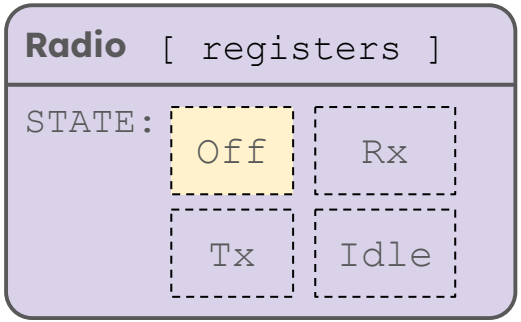


Compilation Error!

(Device Protocol)

Must not interact with Radio registers when OFF.

(HW)

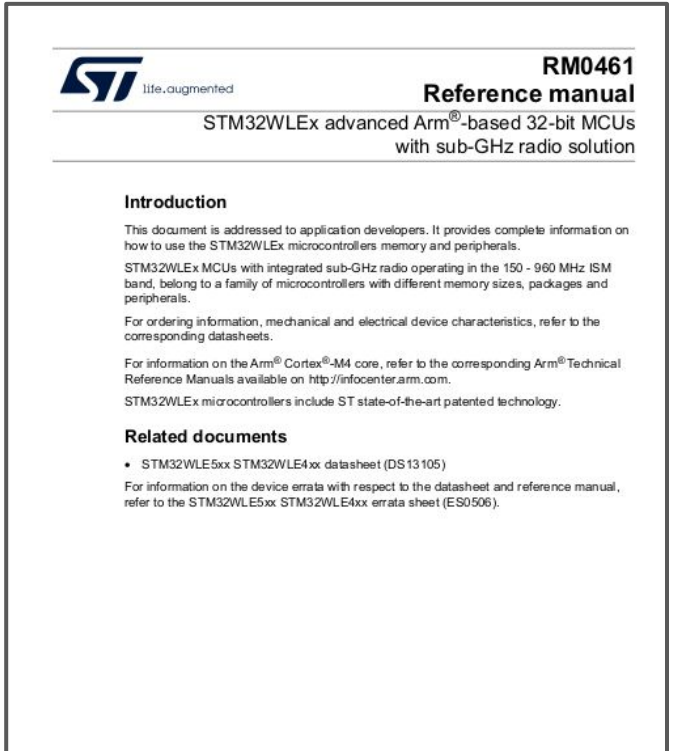


Outline

- Introducing device protocol violations
- **How do we build drivers today?**
- Our system - ***Abacus***
- Using ***Abacus*** to prevent device protocol bugs
- Evaluation & Closing Thoughts

Low-level Driver Development

Specification / reference manual



life.augmented

RM0461
Reference manual

STM32WLEx advanced Arm®-based 32-bit MCUs with sub-GHz radio solution

Introduction

This document is addressed to application developers. It provides complete information on how to use the STM32WLEx microcontrollers memory and peripherals.

STM32WLEx MCUs with integrated sub-GHz radio operating in the 150 - 960 MHz ISM band, belong to a family of microcontrollers with different memory sizes, packages and peripherals.

For ordering information, mechanical and electrical device characteristics, refer to the corresponding datasheets.

For information on the Arm® Cortex®-M4 core, refer to the corresponding Arm® Technical Reference Manuals available on http://infocenter.arm.com.

STM32WLEx microcontrollers include ST state-of-the-art patented technology.

Related documents

- STM32WLE5xx STM32WLE4xx datasheet (DS13105)

For information on the device errata with respect to the datasheet and reference manual, refer to the STM32WLE5xx STM32WLE4xx errata sheet (ES0506).

nRF52840

Product Specification

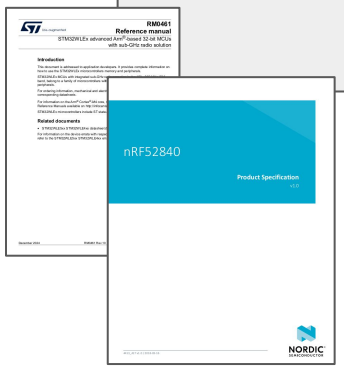
v1.0

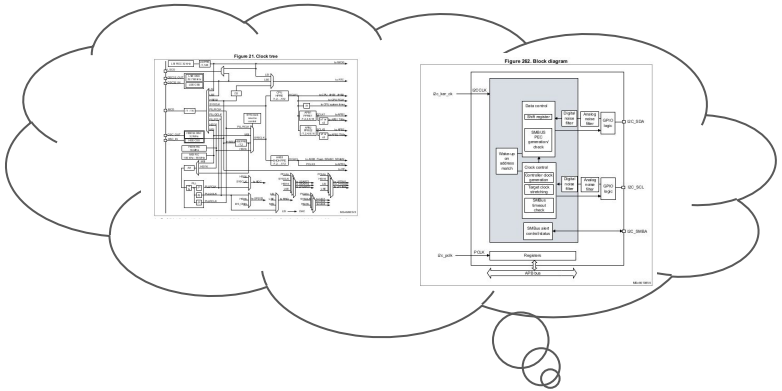
4413_417 v1.0 / 2018-03-16

NORDIC SEMICONDUCTOR

Low-level Driver Development

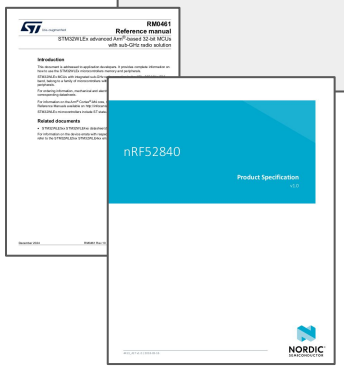
Specification / reference manual

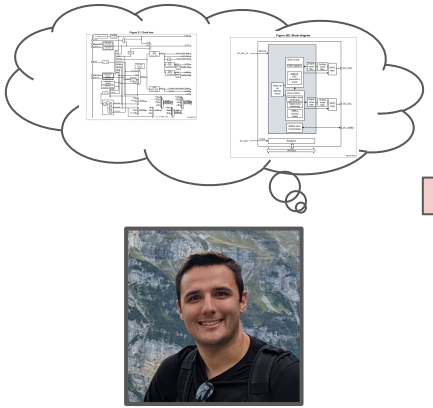




Low-level Driver Development

Specification / reference manual

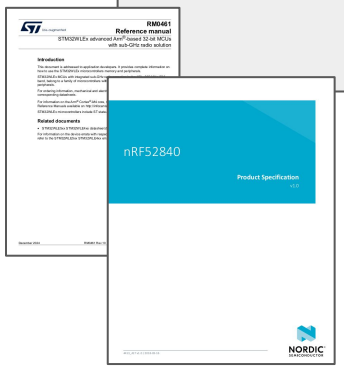


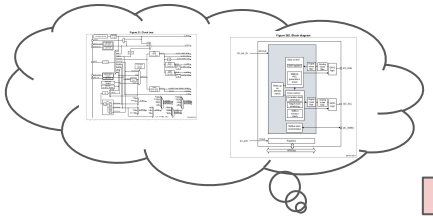


```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    slave_client: OptionalCell<&'a dyn hil::i2c::I2CHwSlaveClient>,
    buf: TakeCell<'static, [u8]>,
    slave_read_buf: TakeCell<'static, [u8]>,
}

/// I2C bus speed.
#[repr(u32)]
0 implementations
pub enum Speed {
    K100 = 0x01980000,
    K250 = 0x04000000,
    K400 = 0x06400000,
}

impl TWI<'_> {
    const fn new(registers: StaticRef<TwiRegisters>) -> Self {
        Self {
            registers,
            client: OptionalCell::empty(),
            slave_client: OptionalCell::empty(),
            buf: TakeCell::empty(),
            slave_read_buf: TakeCell::empty(),
        }
    }

    pub const fn new_twi0() -> Self {
        TWI::new(registers: INSTANCES[0])
    }

    pub const fn new_twi1() -> Self {
        TWI::new(registers: INSTANCES[1])
    }
```

Low-level Driver Development

Specification / reference manual



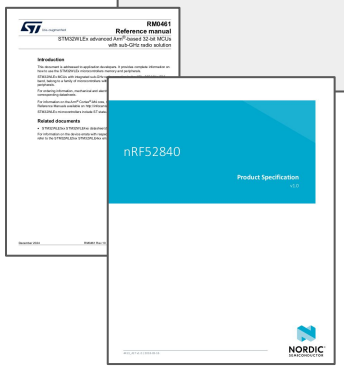


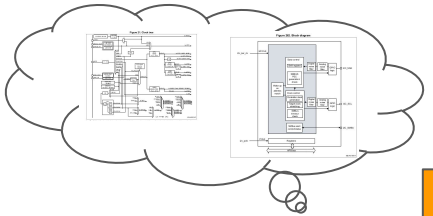
What can go wrong here?

```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    slave_client: OptionalCell<&'a dyn hil::i2c::I2CHwSlaveClient>,
    buf: TakeCell<'static, [u8]>,
    slave_read_buf: TakeCell<'static, [u8]>,
}

/// I2C bus speed.
#[repr(u32)]
0 implementations
pub enum Speed {
    K100 = 0x01980000,
    K250 = 0x04000000,
    K400 = 0x06400000,
}

impl TWI<'_> {
    const fn new(registers: StaticRef<TwiRegisters>) -> Self {
        Self {
            registers,
            client: OptionalCell::empty(),
            slave_client: OptionalCell::empty(),
            buf: TakeCell::empty(),
            slave_read_buf: TakeCell::empty(),
        }
    }

    pub const fn new_twi0() -> Self {
        TWI::new(registers: INSTANCES[0])
    }

    pub const fn new_twi1() -> Self {
        TWI::new(registers: INSTANCES[1])
    }
```

Low-level Driver Development

Specification / reference manual



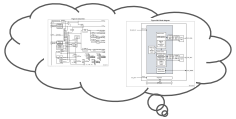


Challenging!

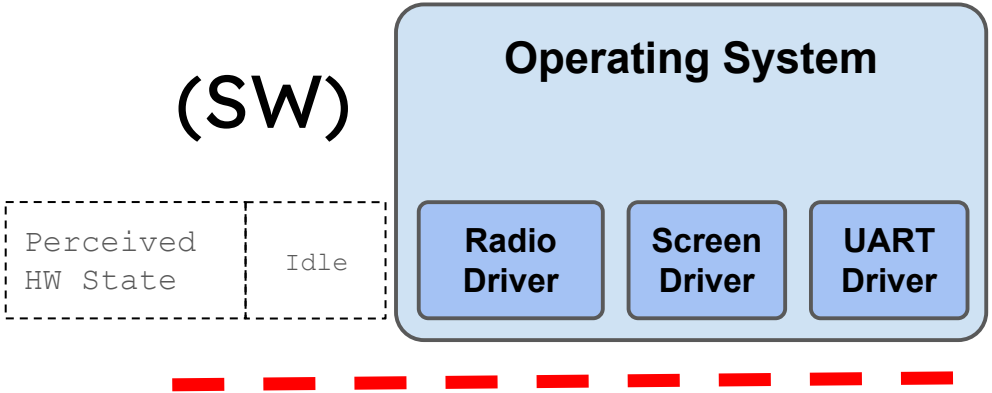
```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    slave_client: OptionalCell<&'a dyn hil::i2c::I2CHwSlaveClient>,
    buf: TakeCell<'static, [u8]>,
    slave_read_buf: TakeCell<'static, [u8]>,
}

/// I2C bus speed.
#[repr(u32)]
0 implementations
pub enum Speed {
    K100 = 0x01980000,
    K250 = 0x04000000,
    K400 = 0x06400000,
}

impl TWI<'_> {
    const fn new(registers: StaticRef<TwiRegisters>) -> Self {
        Self {
            registers,
            client: OptionalCell::empty(),
            slave_client: OptionalCell::empty(),
            buf: TakeCell::empty(),
            slave_read_buf: TakeCell::empty(),
        }
    }

    pub const fn new_twi0() -> Self {
        TWI::new(registers: INSTANCES[0])
    }

    pub const fn new_twi1() -> Self {
        TWI::new(registers: INSTANCES[1])
    }
```

What can go wrong here?

Why is this challenging?



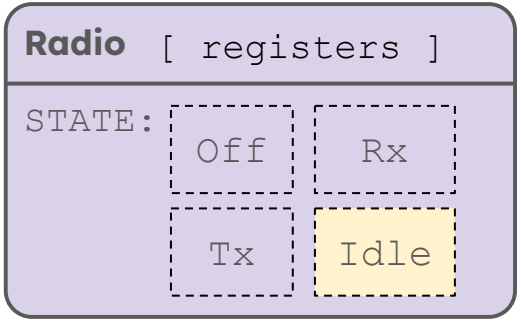
(SW)



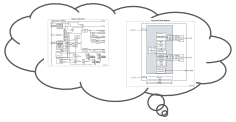
(Device Protocol)

Must not interact with Radio registers when OFF.

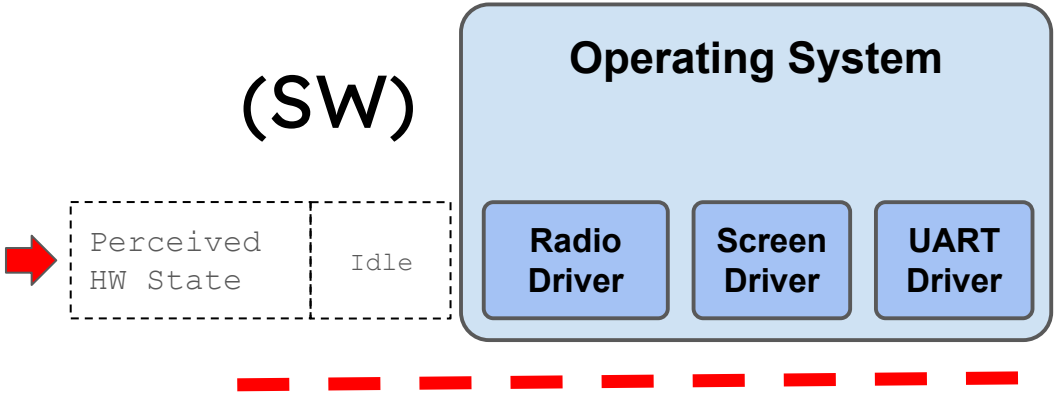
(HW)



Why is this challenging?



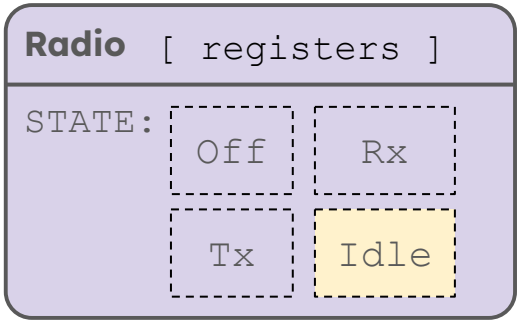
(SW)



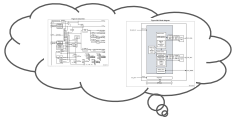
(Device Protocol)

Must not interact with Radio registers when OFF.

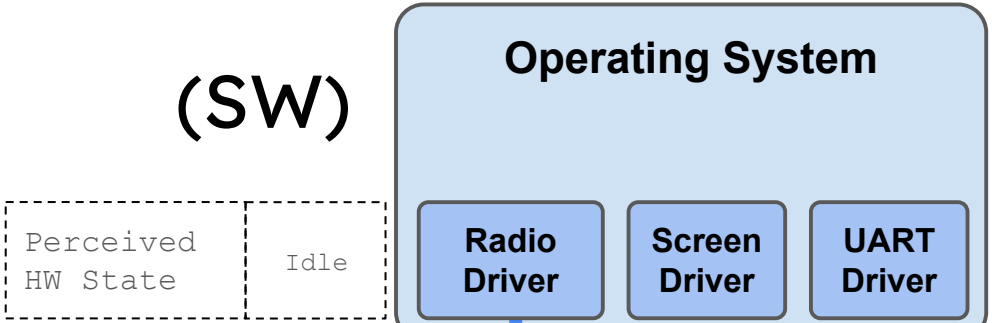
(HW)



Why is this challenging?

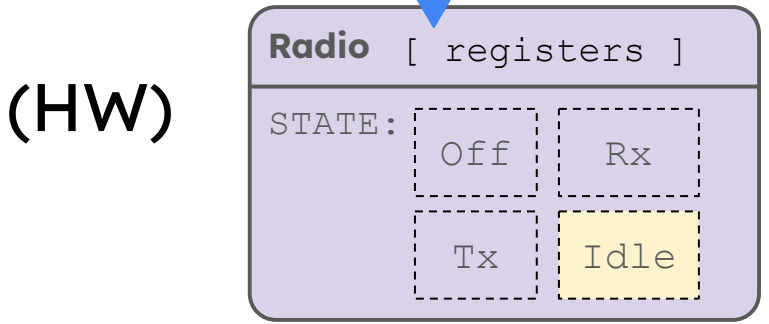


(SW)



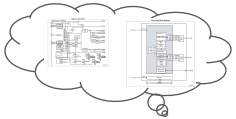
(Device Protocol)

(HW)

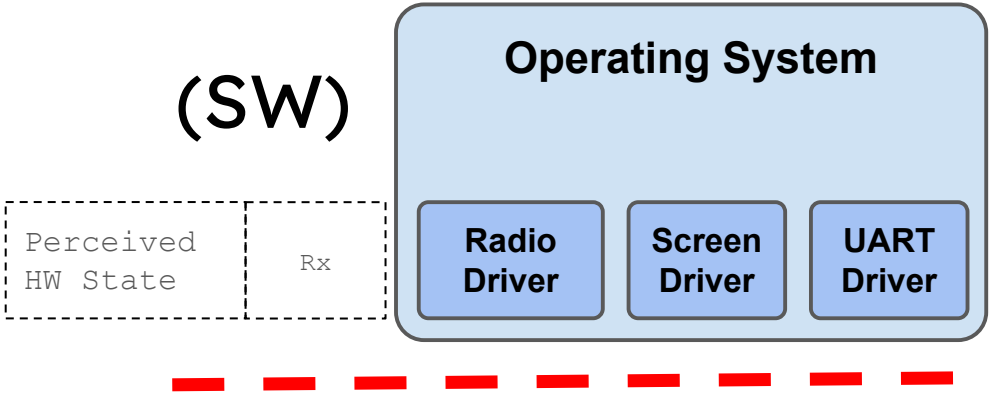


Must not interact with Radio registers when OFF.

Why is this challenging?



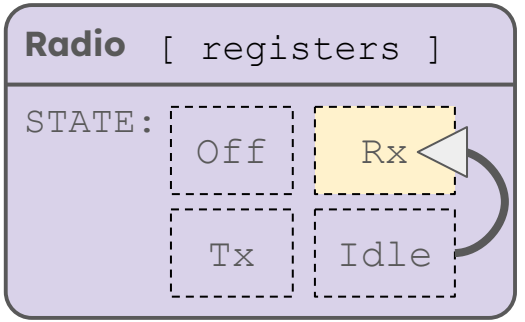
(SW)



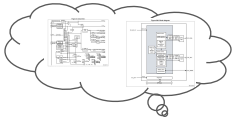
(Device Protocol)

Must not interact with Radio registers when OFF.

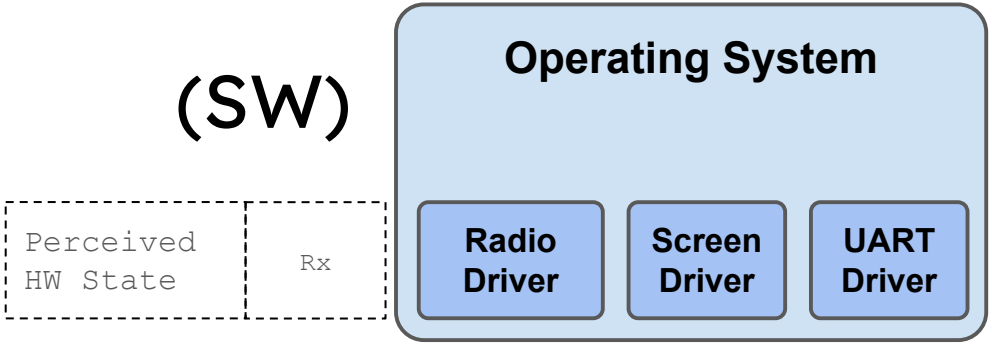
(HW)



Why is this challenging?



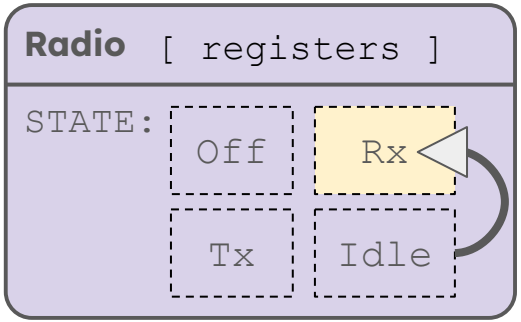
(SW)



(Device Protocol)

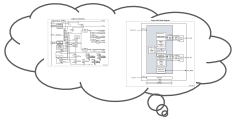
Must not interact with Radio registers when OFF.

(HW)

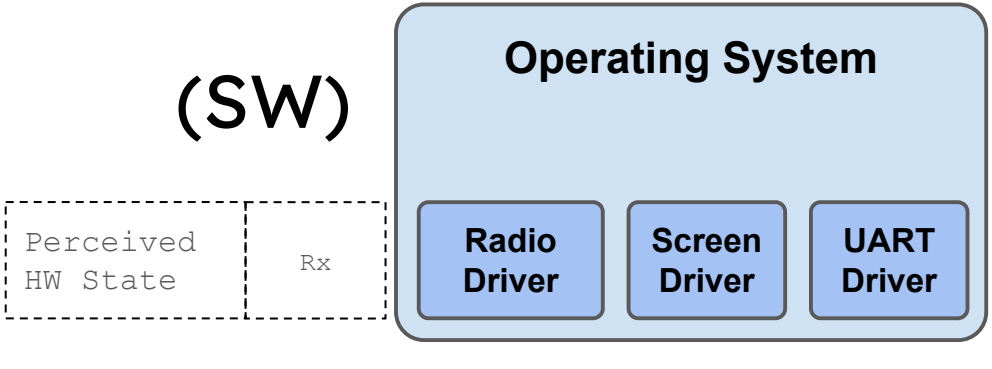


*Hardware configured to transition from RX to OFF after receive completed.

Why is this challenging?



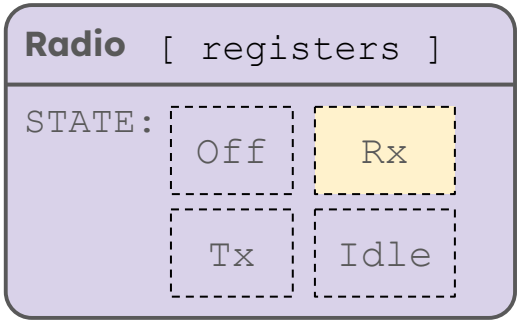
(SW)



(Device Protocol)

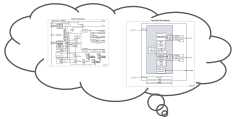
Must not interact with Radio registers when OFF.

(HW)

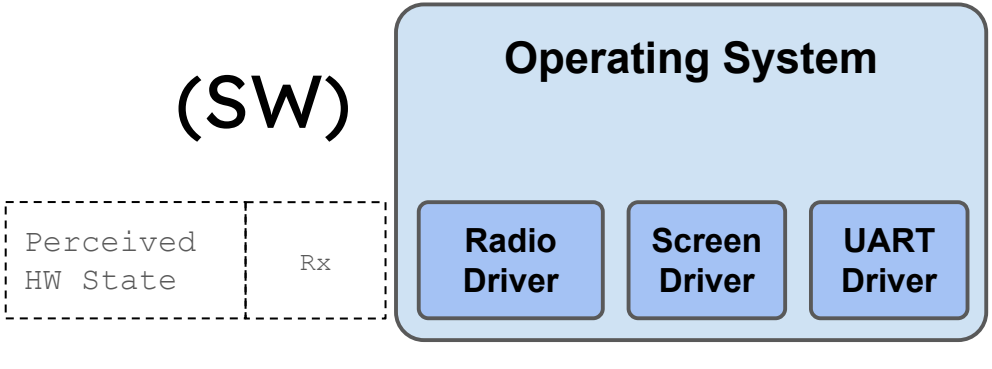


*Hardware configured to transition from `RX` to `OFF` after receive completed.

Why is this challenging?



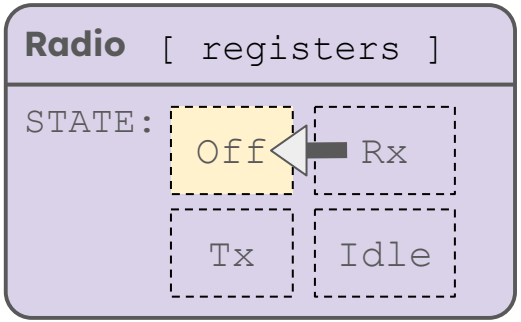
(SW)



(Device Protocol)

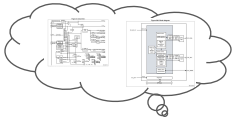
Must not interact with Radio registers when OFF.

(HW)

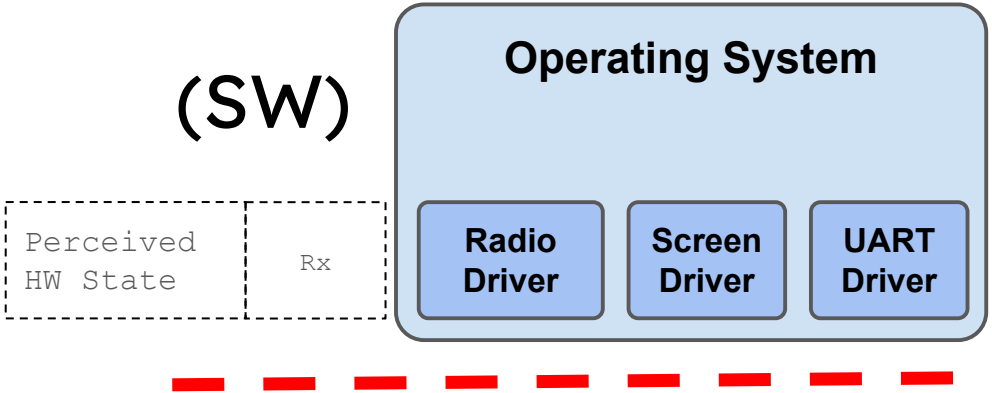


*Hardware configured to transition from `RX` to `OFF` after receive completed.

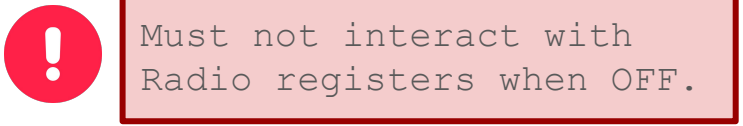
Why is this challenging?



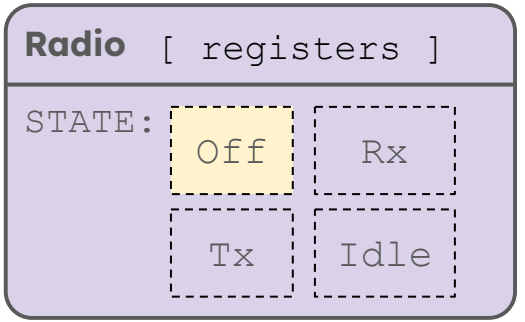
(SW)



(Device Protocol)

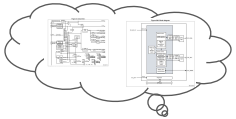


(HW)

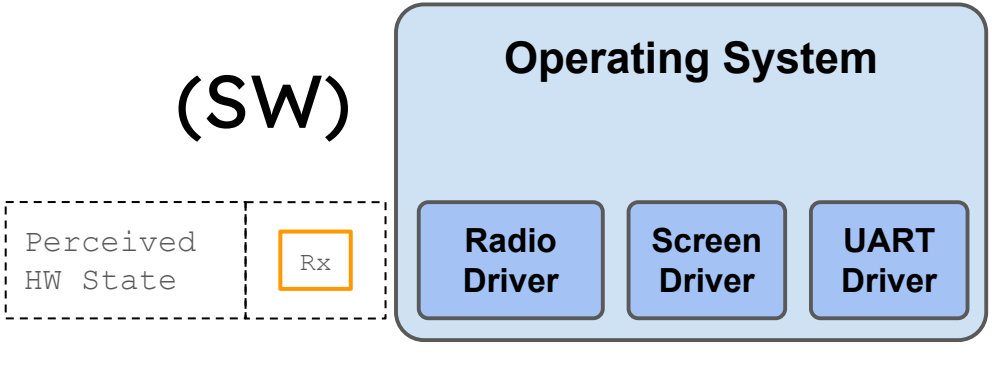


*Hardware configured to transition from `RX` to `OFF` after receive completed.

Why is this challenging?



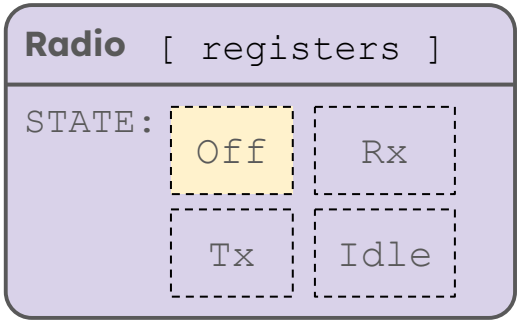
(SW)



(Device Protocol)

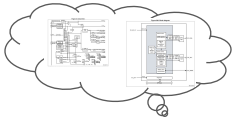
Must not interact with Radio registers when OFF.

(HW)

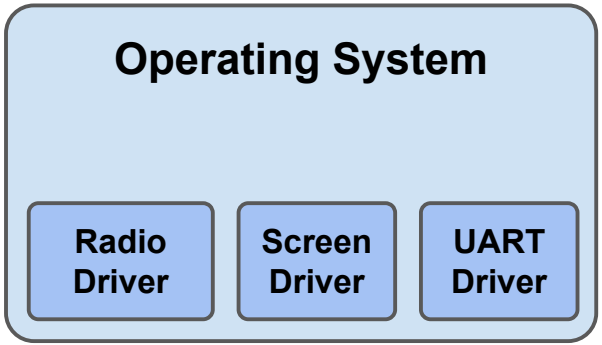


*Hardware configured to transition from `RX` to `OFF` after receive completed.

Why is this challenging?



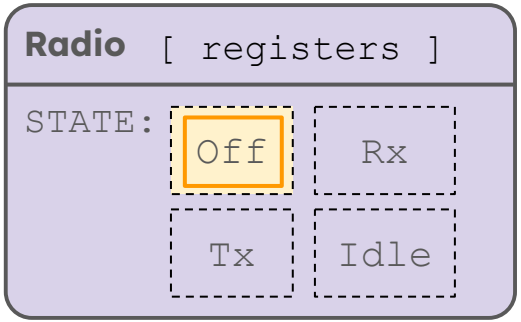
(SW)



Driver view of HW state is incorrect!

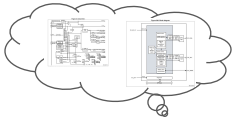
(Device Protocol)

(HW)



*Hardware configured to transition from `RX` to `OFF` after receive completed.

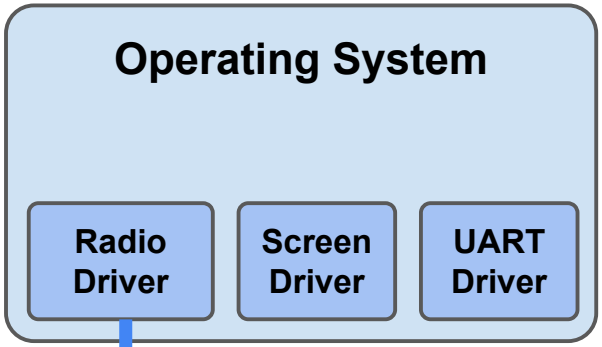
Why is this challenging?

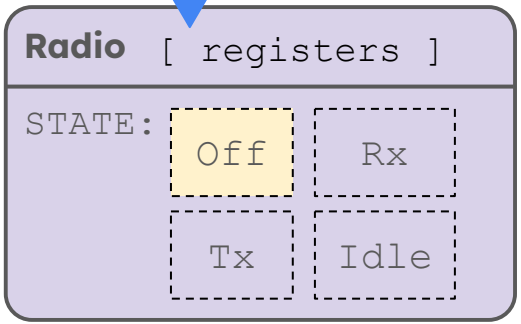


(SW)

Driver view of HW state is incorrect!

Perceived HW State	Rx



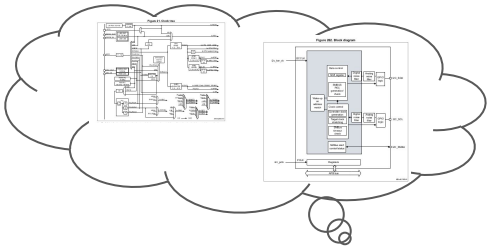


(HW)

(Device Protocol)

Must not interact with Radio registers when OFF.

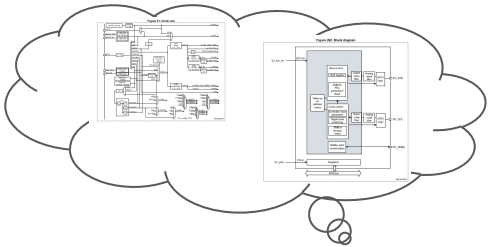
*Hardware configured to transition from `RX` to `OFF` after receive completed.



Challenging!

```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    slave_client: OptionalCell<&'a dyn hil::i2c::I2CHwSlaveClient>,
    buf: TakeCell<'static, [u8]>,
    slave_read_buf: TakeCell<'static, [u8]>,
}

/// I2C bus speed.
#[repr(u32)]
0 implementations
pub enum Speed {
    K100 = 0x01980000,
    K250 = 0x04000000,
    K400 = 0x06400000,
}

impl TWI<'_> {
    const fn new(registers: StaticRef<TwiRegisters>) -> Self {
        Self {
            registers,
            client: OptionalCell::empty(),
            slave_client: OptionalCell::empty(),
            buf: TakeCell::empty(),
            slave_read_buf: TakeCell::empty(),
        }
    }

    pub const fn new_twi0() -> Self {
        TWI::new(registers: INSTANCES[0])
    }

    pub const fn new_twi1() -> Self {
        TWI::new(registers: INSTANCES[1])
    }
```

Q: Can we enforce, <u>at compile time</u>, that the driver will <u>always</u> comply with the developer's hw mental model?



Verified!

Abacus

```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    slave_client: OptionalCell<&'a dyn hil::i2c::I2CHwSlaveClient>,
    buf: TakeCell<'static, [u8]>,
    slave_read_buf: TakeCell<'static, [u8]>,
}

/// I2C bus speed.
#[repr(u32)]
0 implementations
pub enum Speed {
    K100 = 0x01980000,
    K250 = 0x04000000,
    K400 = 0x06400000,
}

impl TWI<'_> {
    const fn new(registers: StaticRef<TwiRegisters>) -> Self {
        Self {
            registers,
            client: OptionalCell::empty(),
            slave_client: OptionalCell::empty(),
            buf: TakeCell::empty(),
            slave_read_buf: TakeCell::empty(),
        }
    }

    pub const fn new_twi0() -> Self {
        TWI::new(registers: INSTANCES[0])
    }

    pub const fn new_twi1() -> Self {
        TWI::new(registers: INSTANCES[1])
    }
```

Q: Can we enforce, <u>at compile time</u>, that the driver will <u>always</u> comply with the developer's hw mental model?

- software driver adheres to hardware's specification
- i.e., only performs operation (e.g. MMIO) valid for the given hw state

How might we prevent driver device protocol bugs?

Standard approaches for enforcing system properties (generally)...

Testing

(only proves the absence of tested bugs).

Formal Verification

(challenging; requires domain specific expertise).

How might we prevent driver device protocol bugs?

Standard approaches for enforcing system properties (generally)...

Testing

(only proves the absence of tested bugs).

TypeState Programming

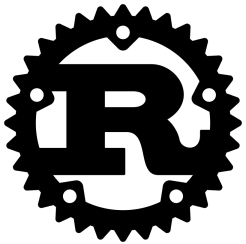
Formal Verification

(challenging; requires domain specific expertise).

We present, **Abacus**, a framework that statically (at compile time) prevents device protocol violations

- Model device protocols and hardware-initiated state transitions using **Abacus** typestates.
- Minimal to no overheads in runtime and code size.

TypeStates

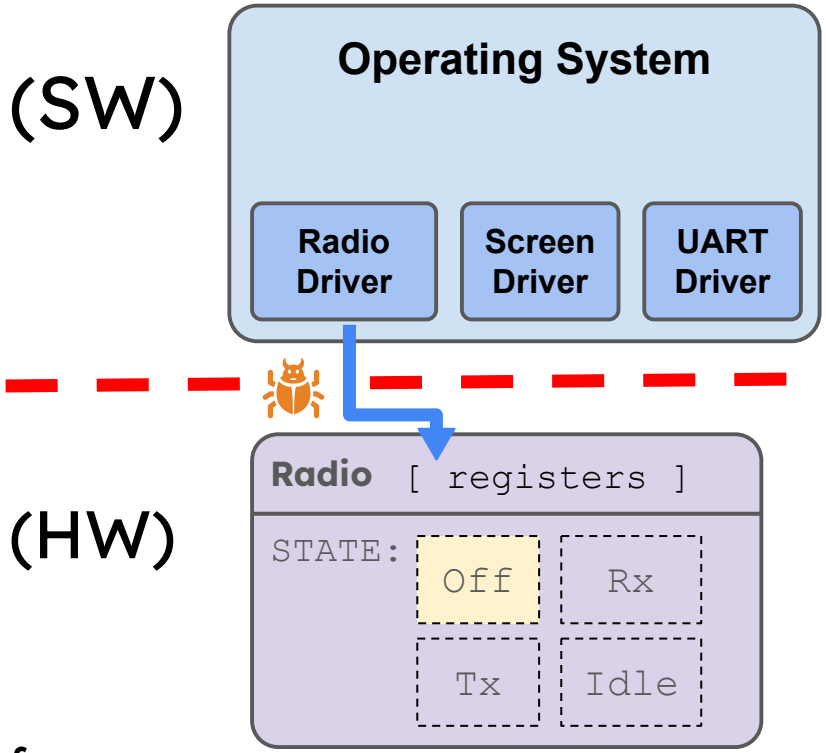


DSL

Outline

- Introducing device protocol violations
- How do we build drivers today?
- **Our system - *Abacus***
- Using ***Abacus*** to prevent device protocol bugs
- Evaluation & Closing Thoughts

Challenge: Out of the box typestates cannot model stateful device protocols!



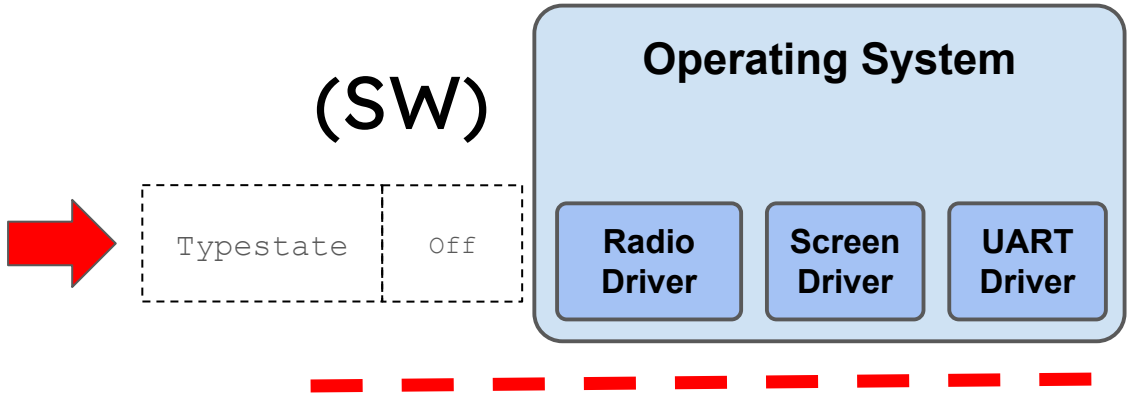
(Device Protocol)

Must not interact with Radio registers when OFF.

Device Protocol Bug!

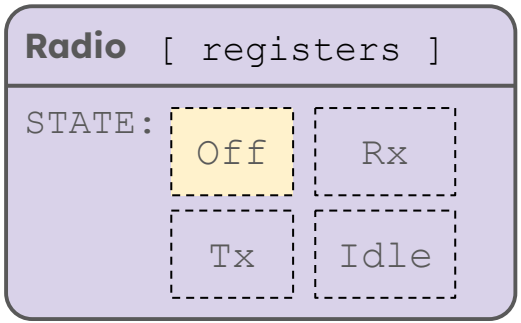
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



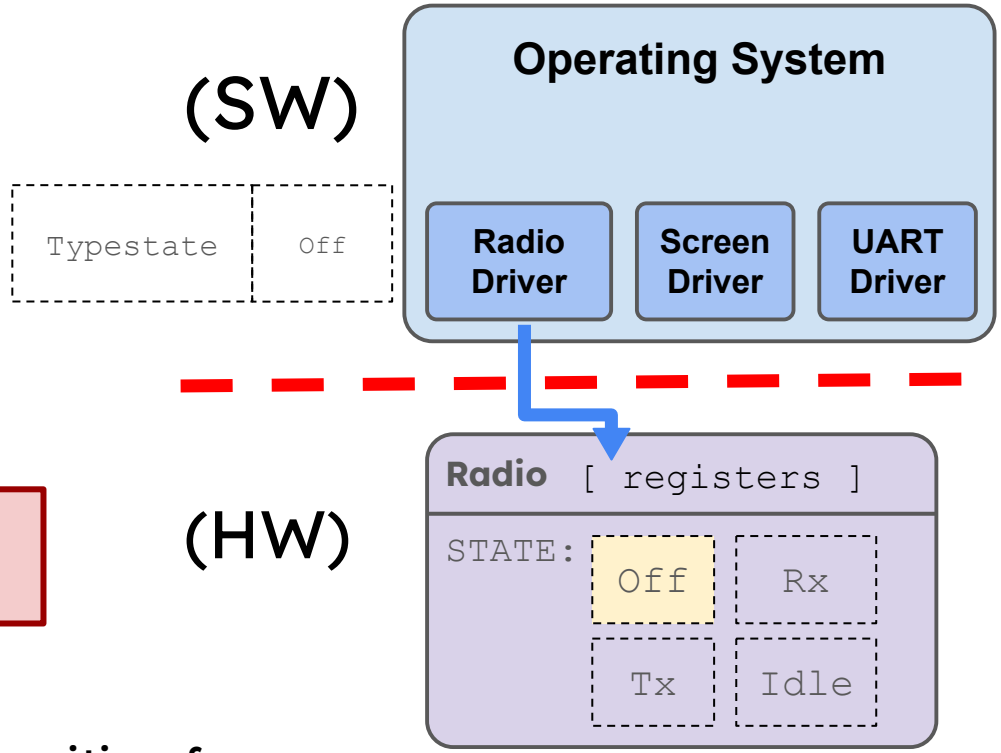
(Device Protocol)

Must not interact with Radio registers when OFF.



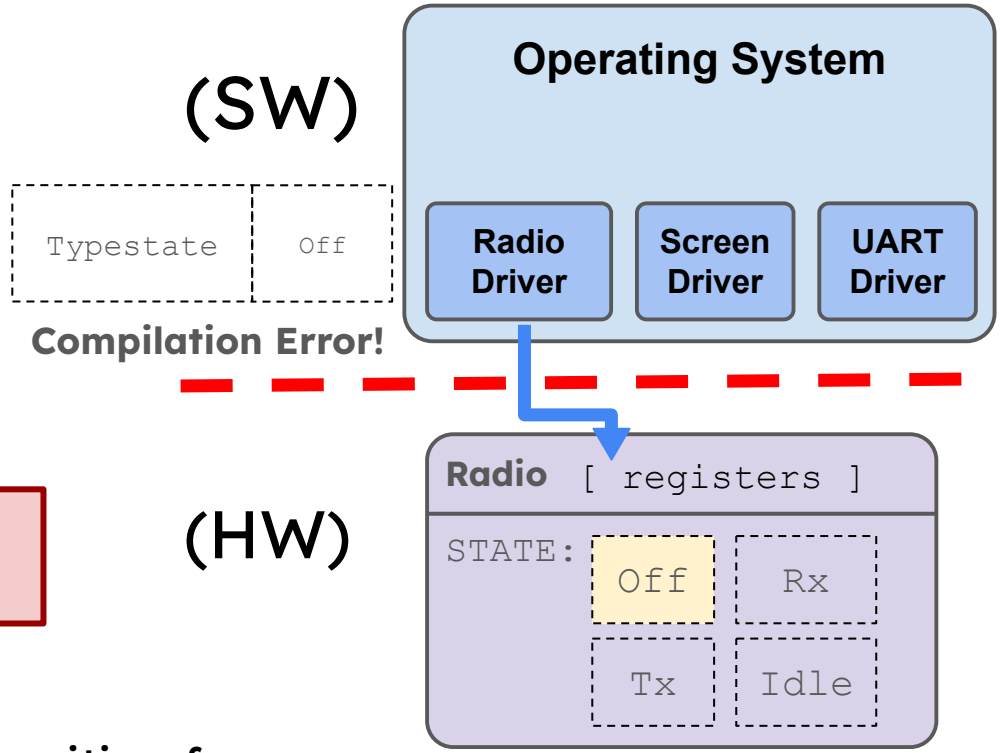
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



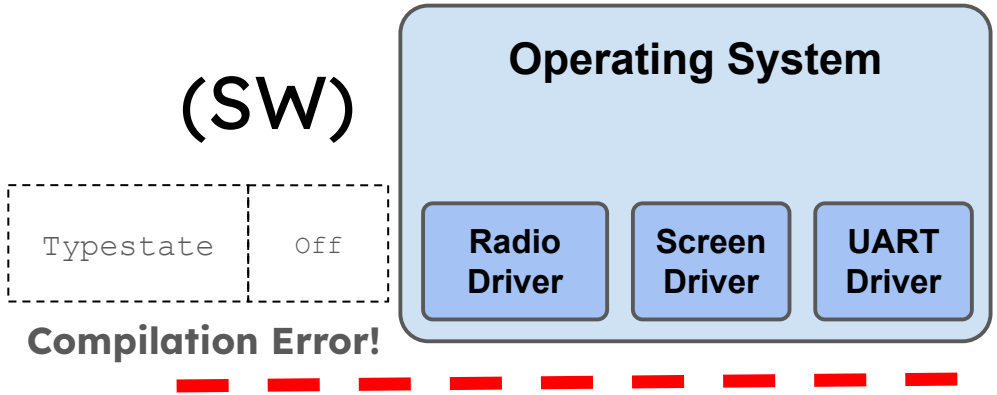
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



*Hardware configured to transition from `RX` to `OFF` after receive completed.

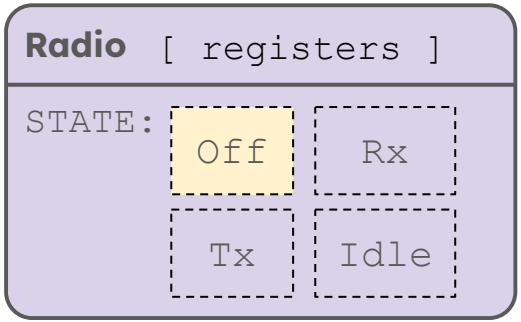
Challenge: Out of the box typestates cannot model stateful device protocols!



(Device Protocol)

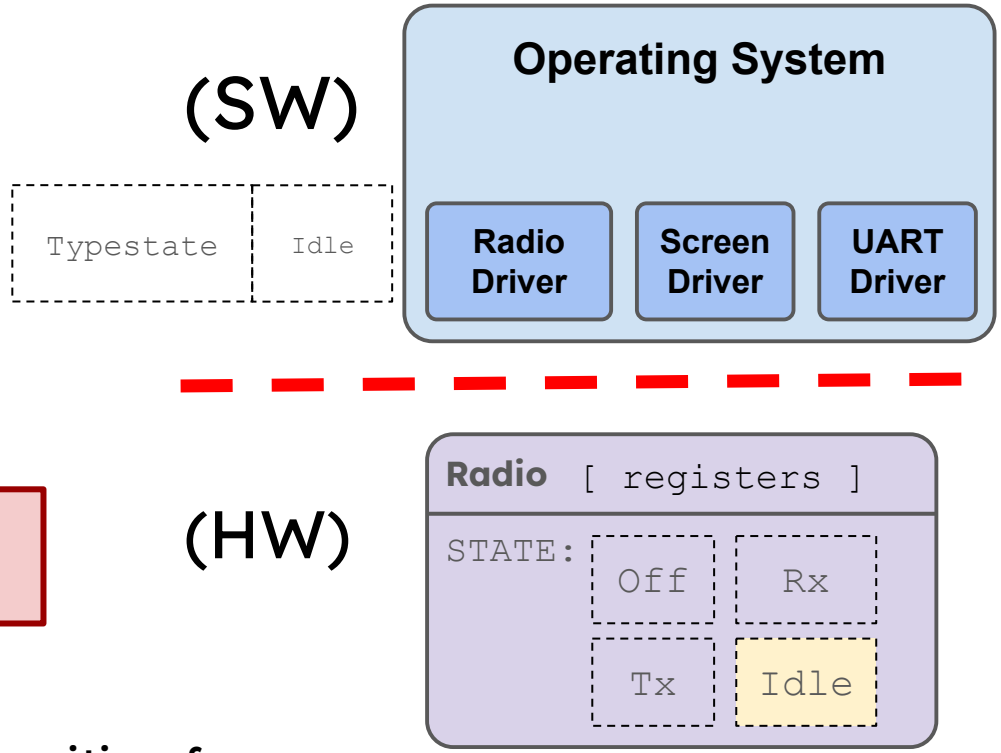
```
Must not interact with
Radio registers when OFF.
```

(HW)



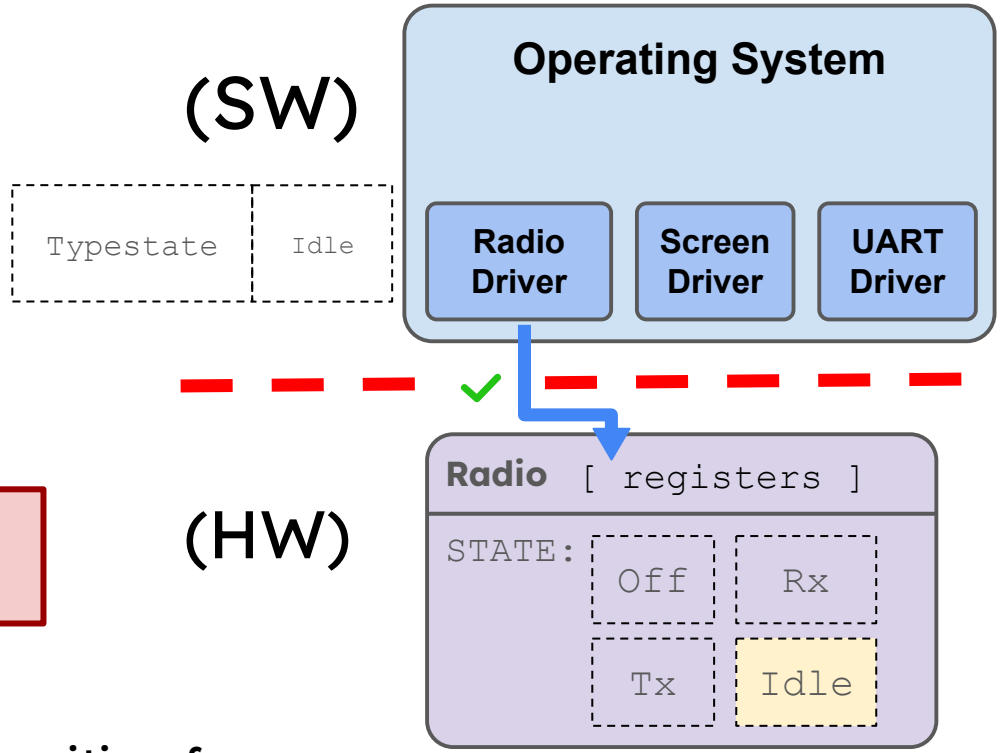
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



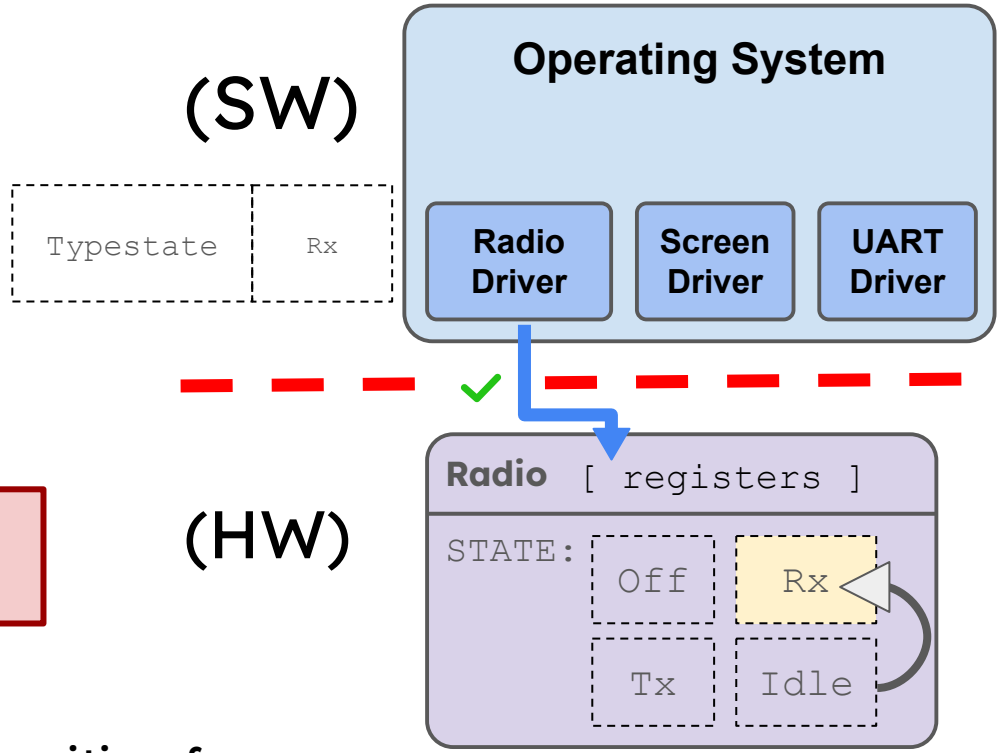
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



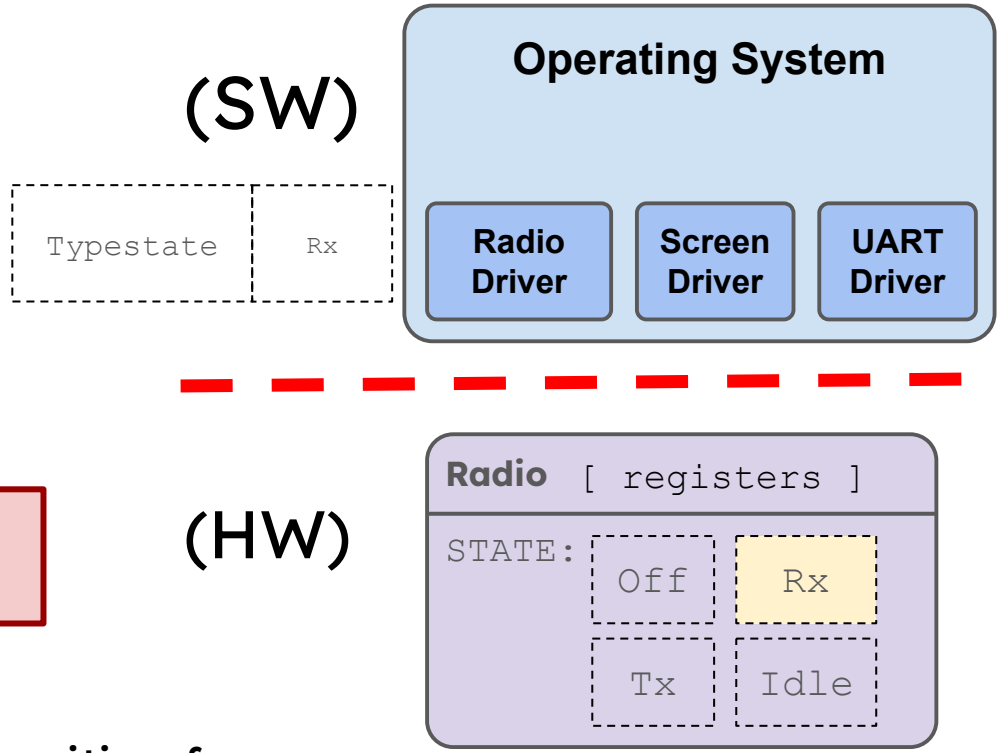
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



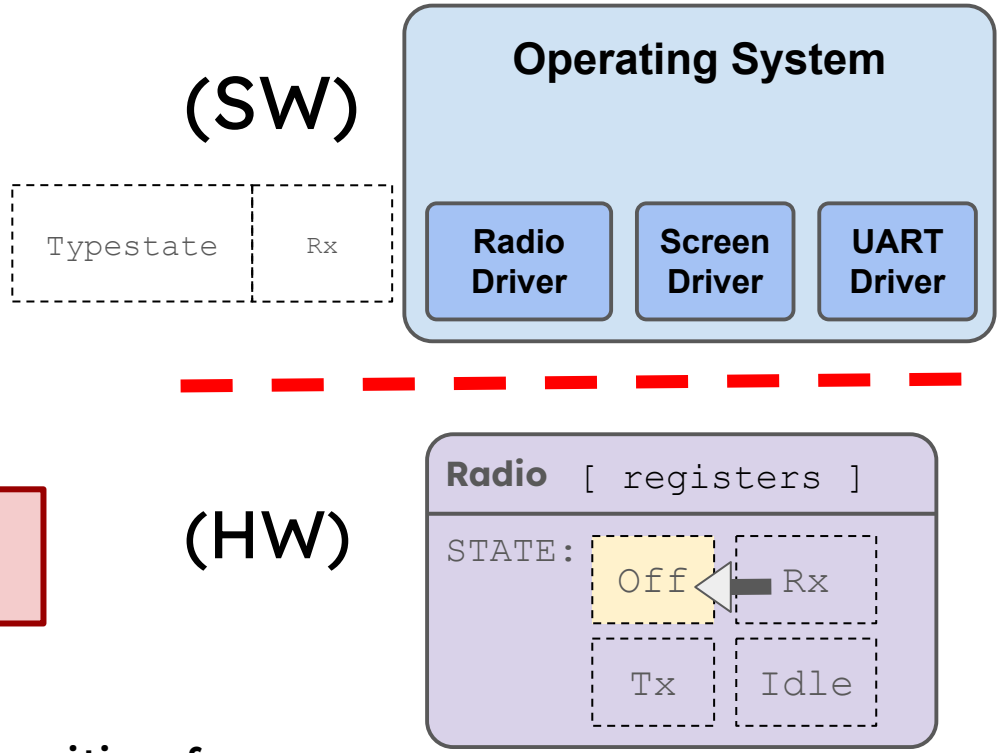
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



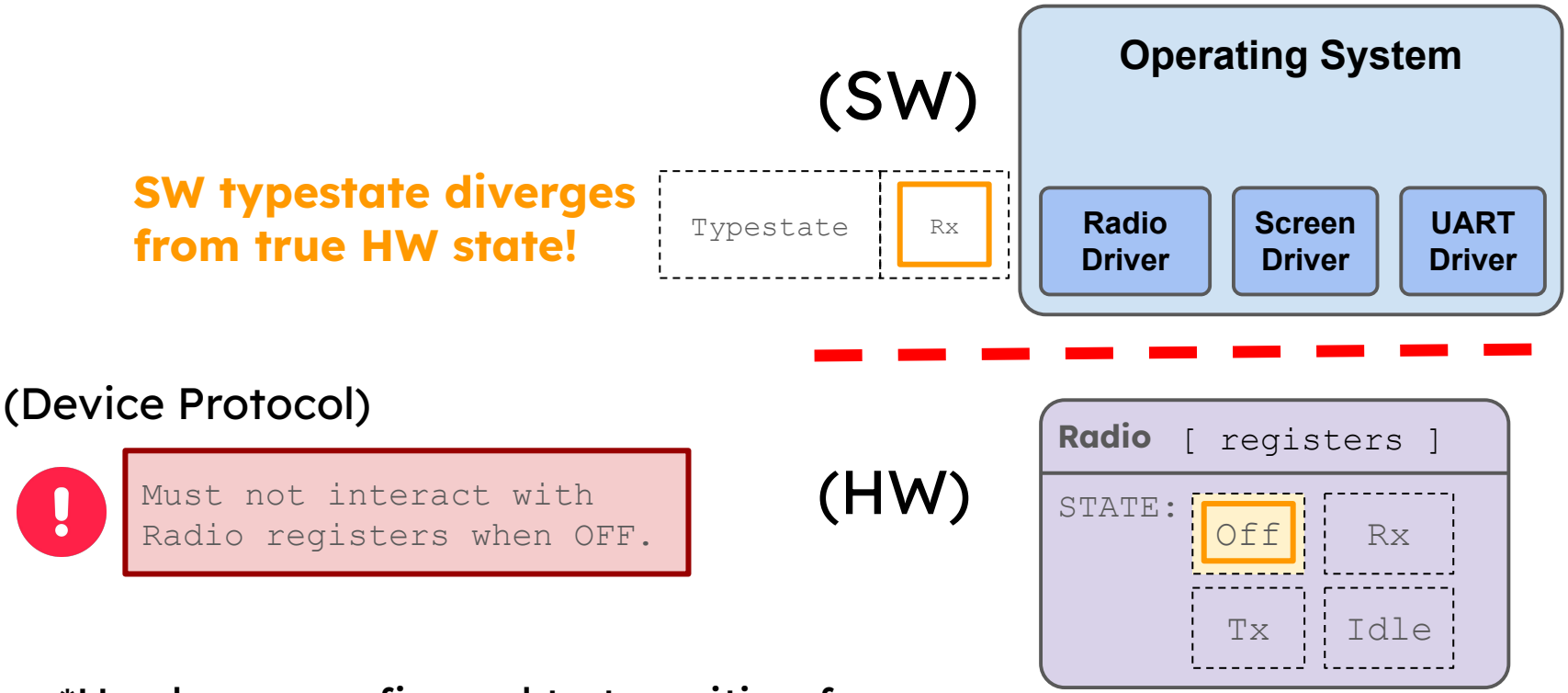
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



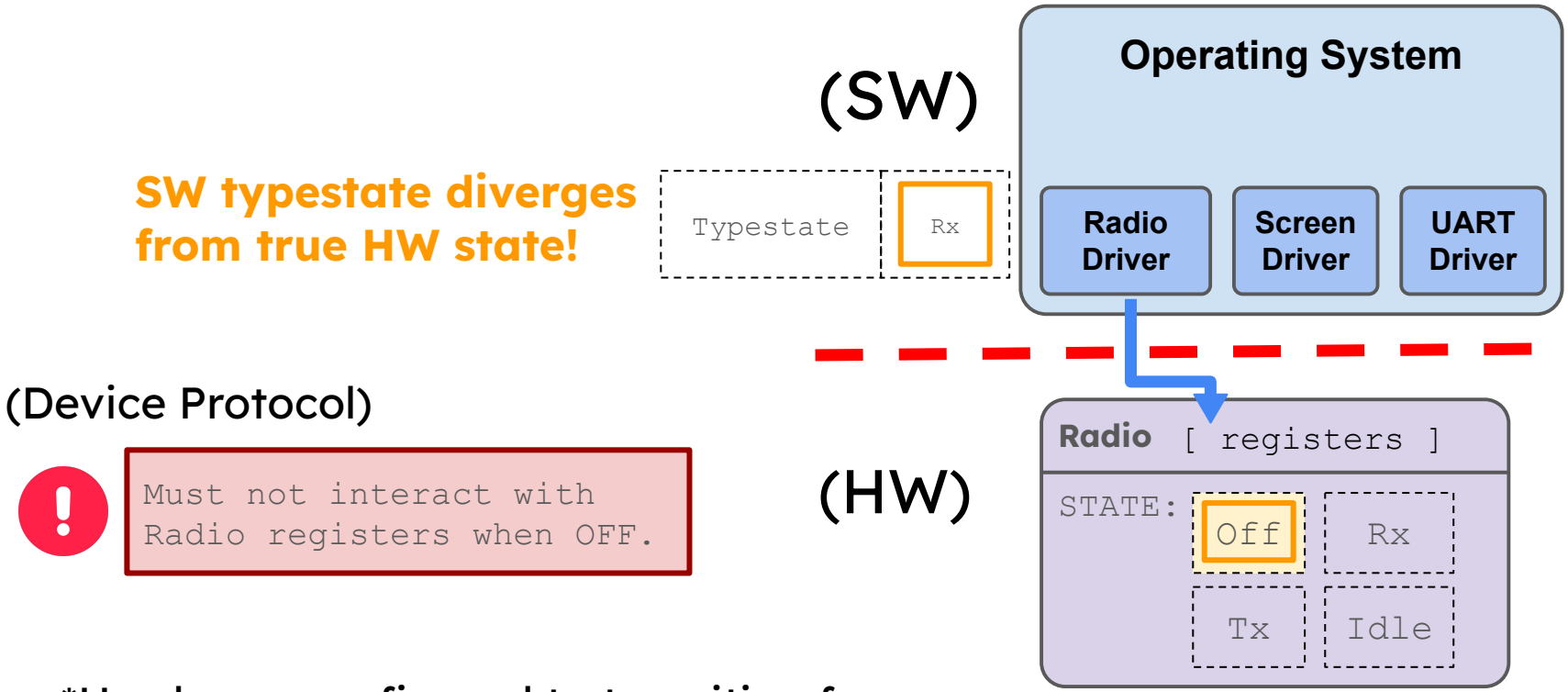
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



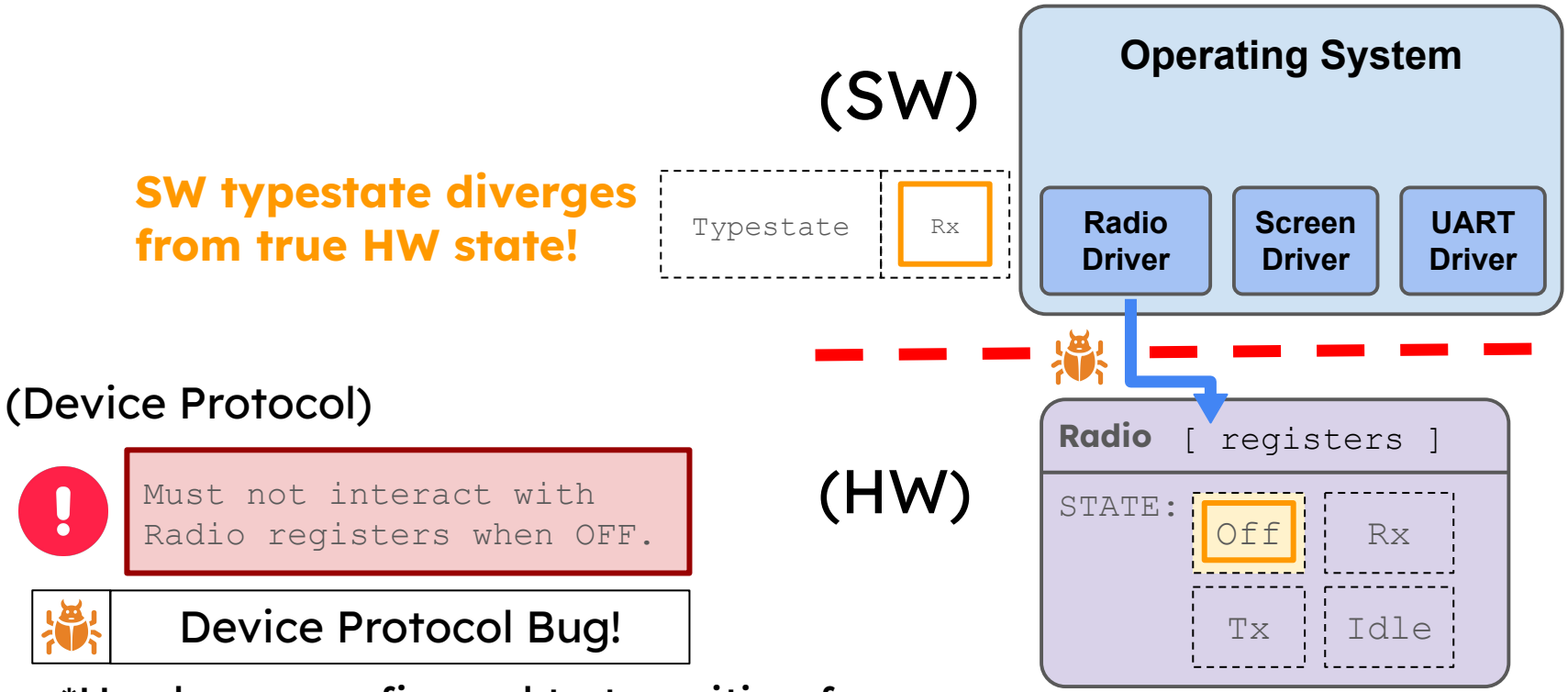
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



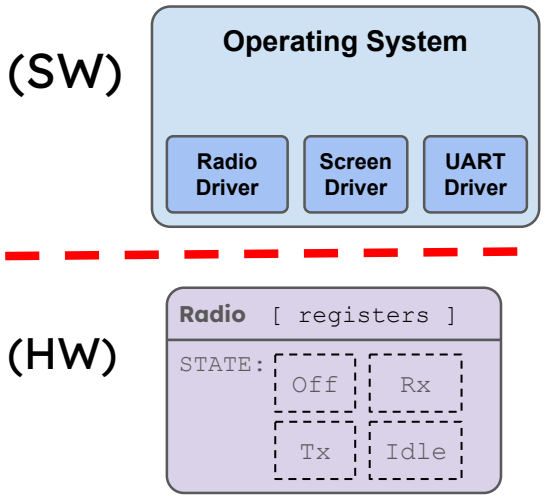
*Hardware configured to transition from `RX` to `OFF` after receive completed.

Challenge: Out of the box typestates cannot model stateful device protocols!



*Hardware configured to transition from `RX` to `OFF` after receive completed.

Abacus typestates model hardware-software concurrency

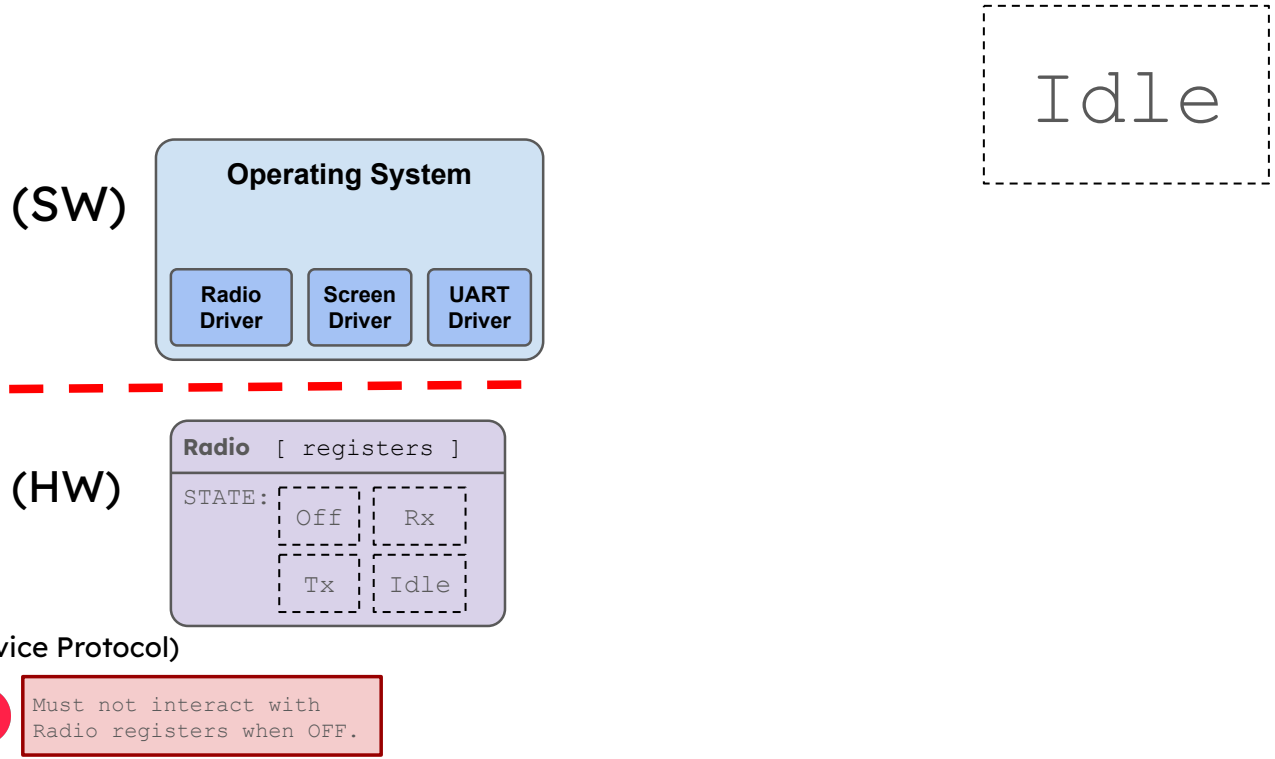


(Device Protocol)

Must not interact with Radio registers when OFF.

*Hardware configured to transition from `RX` to `OFF` after receive completed.

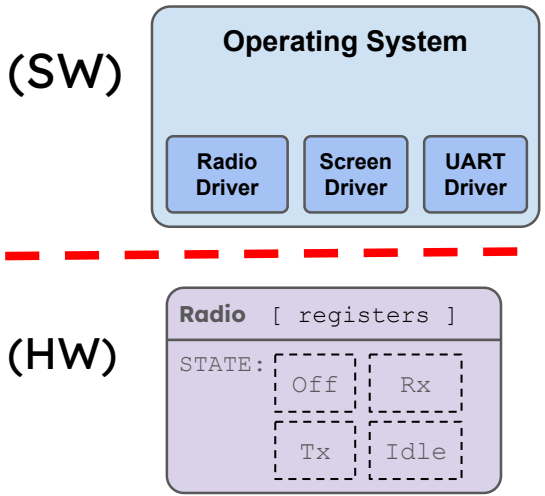
Abacus typestates model hardware-software concurrency



*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency



(Device Protocol)

Must not interact with Radio registers when OFF.

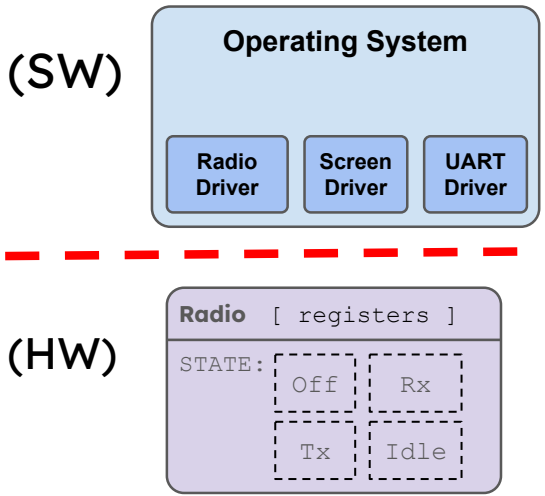
Idle

Rx

*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

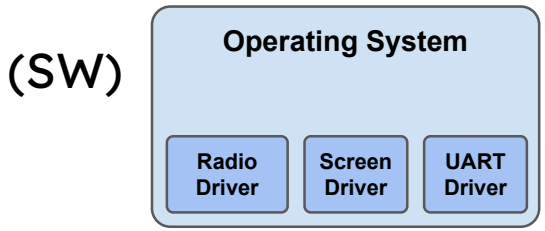
Abacus typestates model hardware-software concurrency



*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

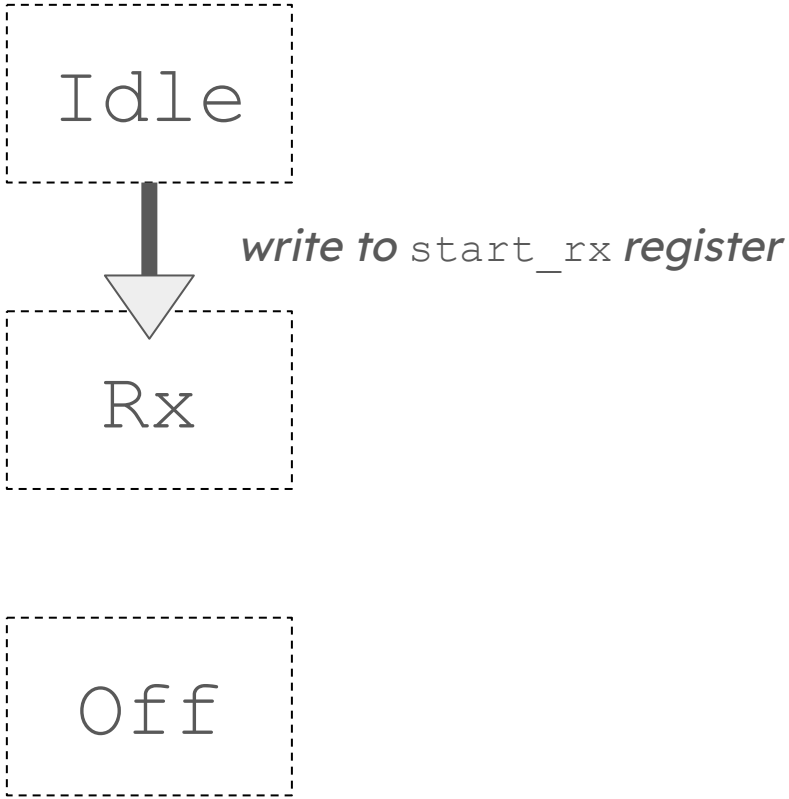
Abacus typestates model hardware-software concurrency



(Device Protocol)

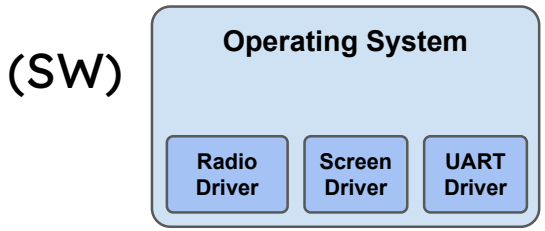
Must not interact with Radio registers when OFF.

*Hardware configured to transition from `RX` to `OFF` after receive completed.



(Developer Mental Model of HW Specification)

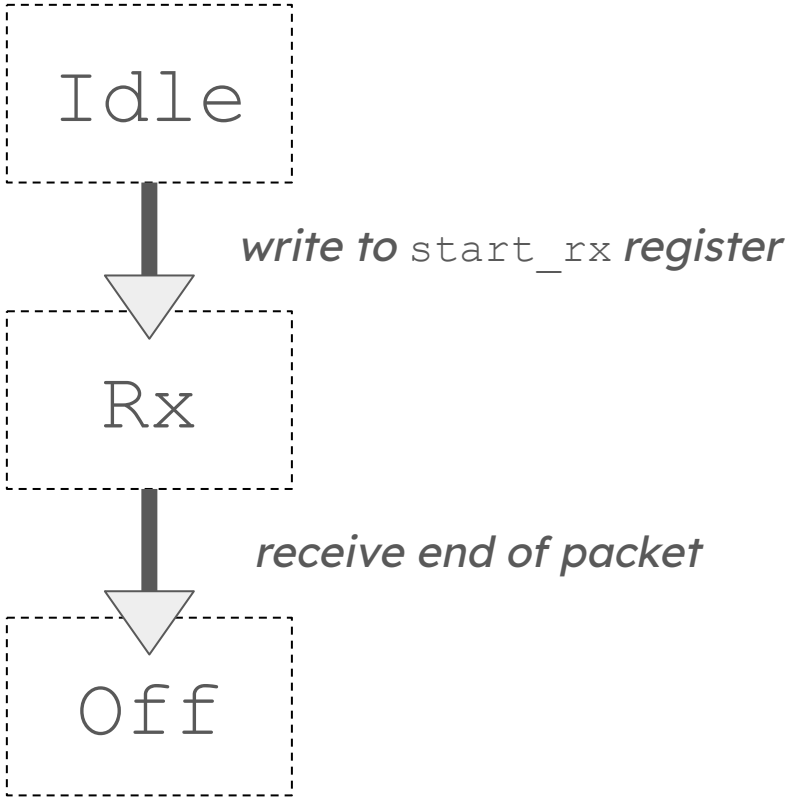
Abacus typestates model hardware-software concurrency



(Device Protocol)

Must not interact with Radio registers when OFF.

*Hardware configured to transition from `RX` to `OFF` after receive completed.

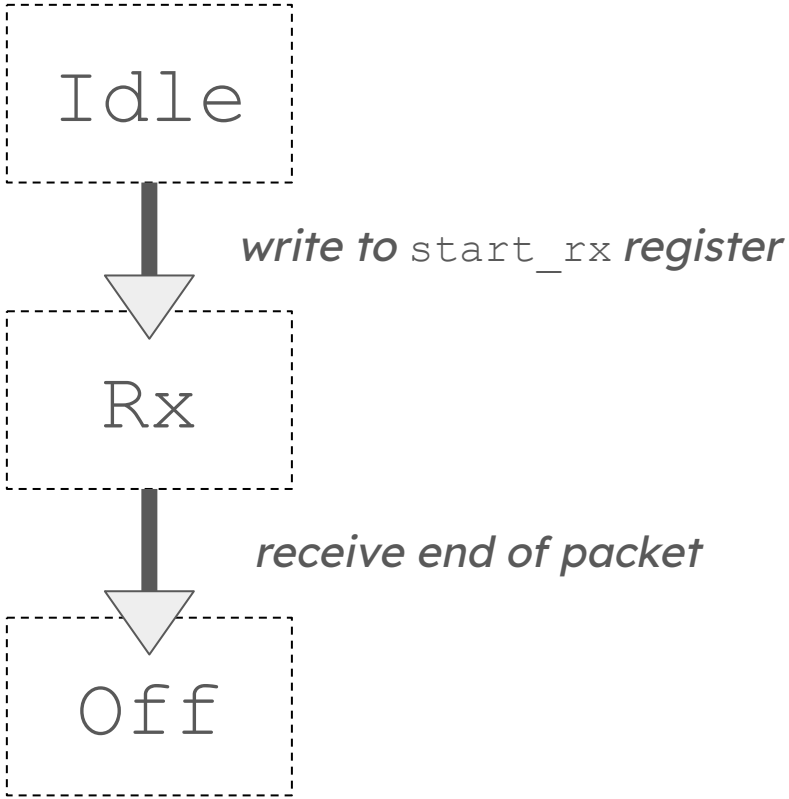


(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

There are two classes of hw state transitions:

- Software-initiated



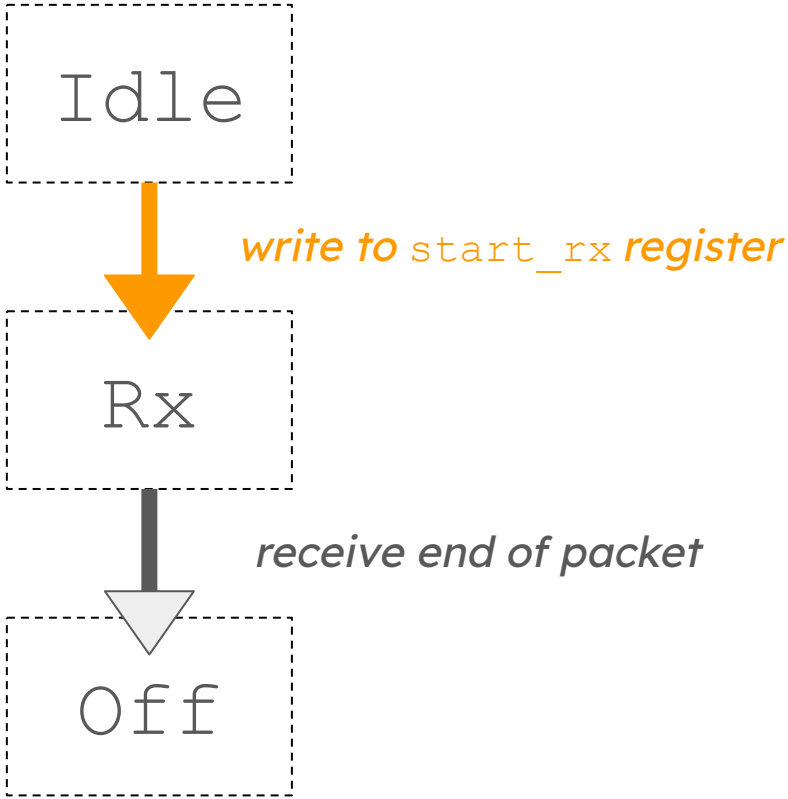
*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

There are two classes of hw state transitions:

- Software-initiated



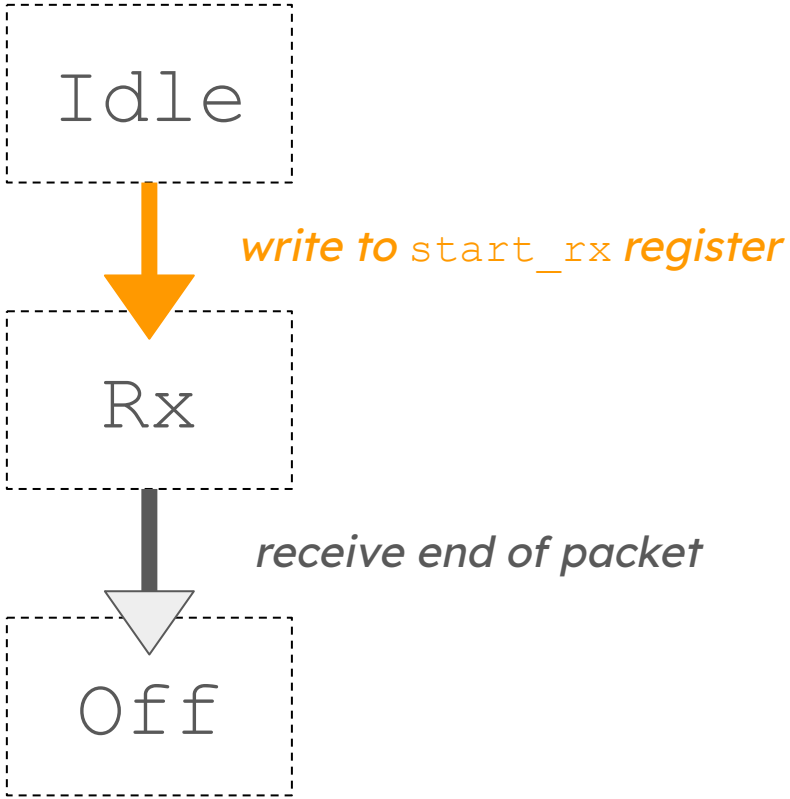
*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

There are two classes of hw state transitions:

- Software-initiated
- Hardware-initiated



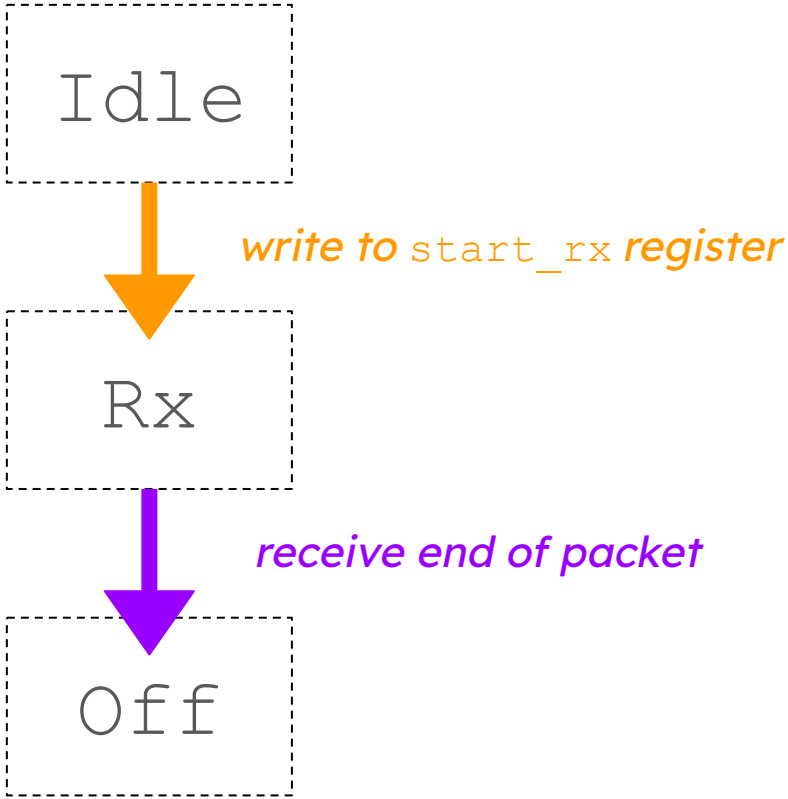
*Hardware configured to transition from `RX` to `OFF` after receive completed.

(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

There are two classes of hw state transitions:

- Software-initiated
- Hardware-initiated



*Hardware configured to transition from `RX` to `OFF` after receive completed.

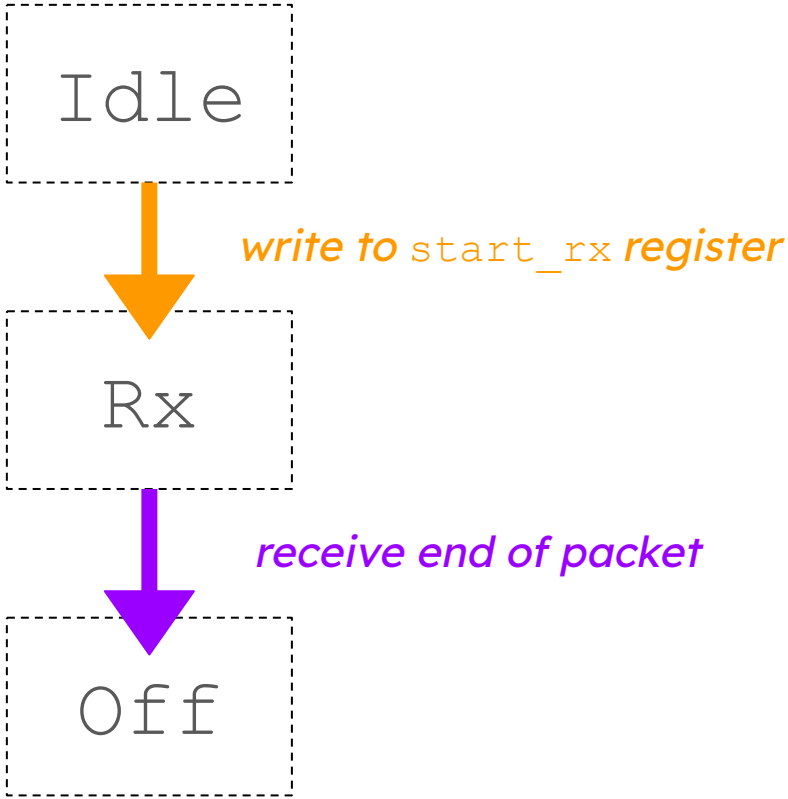
Abacus typestates model hardware-software concurrency

There are two classes of hw state transitions:

- Software-initiated
- Hardware-initiated

Categorize hardware states into two mutually exclusive families:

- stable state
- transient state

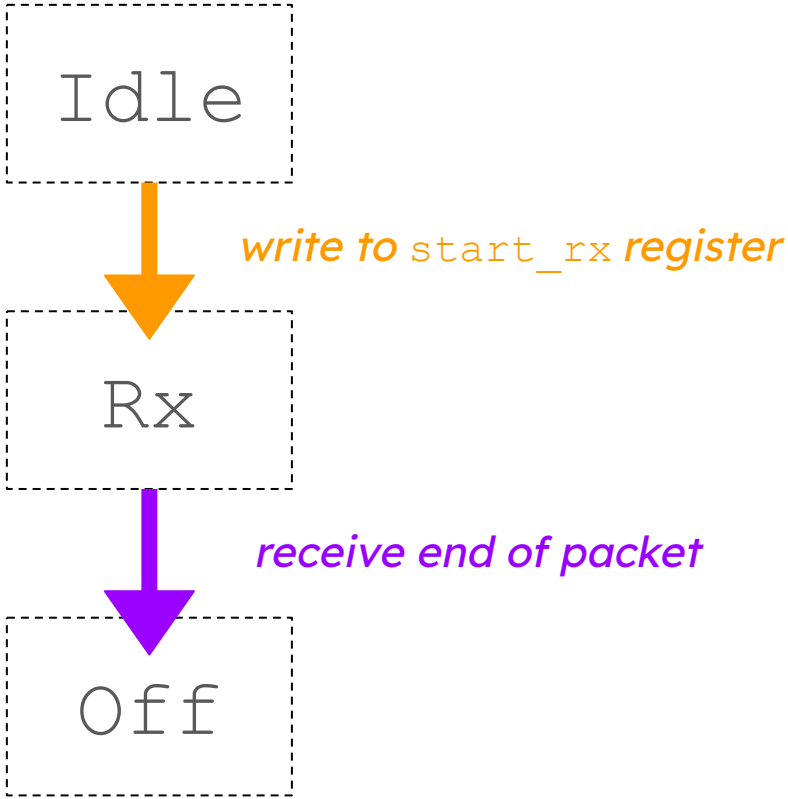


(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

Stable State

- Hw state that *<u>can only be</u>* exited with a software-initiated state transition.

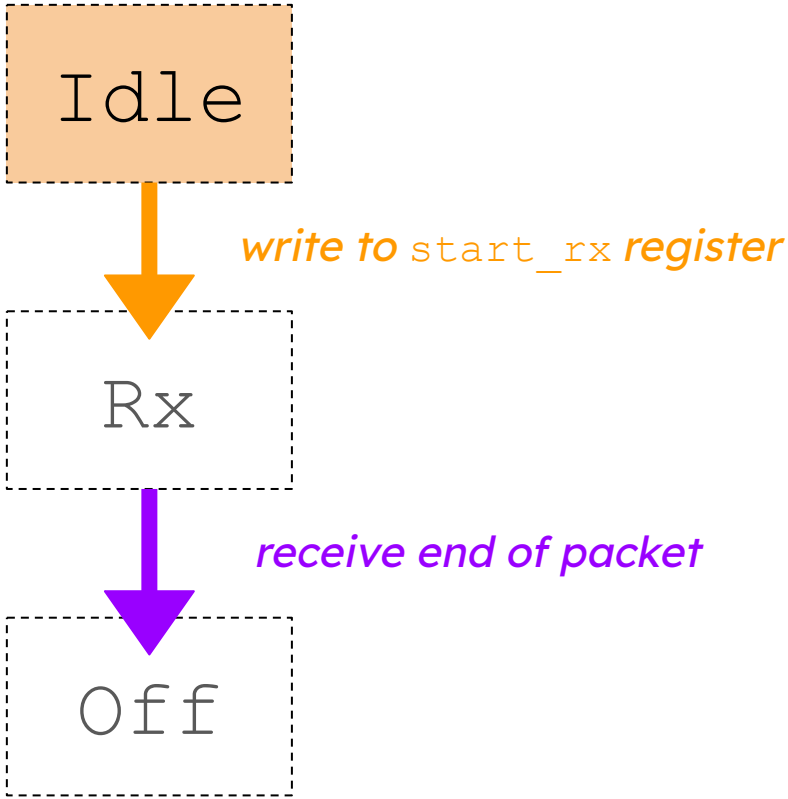


(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

Stable State

- Hw state that *<u>can only be</u>* exited with a software-initiated state transition.



(Developer Mental Model of HW Specification)

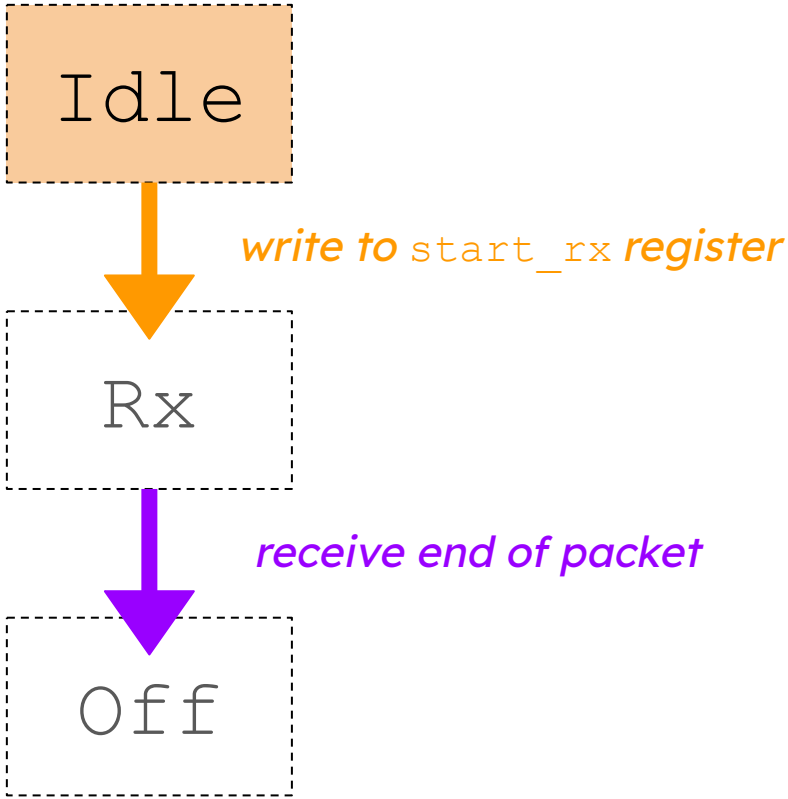
Abacus typestates model hardware-software concurrency

Stable State

- Hw state that *can only be* exited with a software-initiated state transition.

Transient State

- Hw state with *at least* one hw-initiated state transition.
- Transition from transient state without explicit software involvement.



(Developer Mental Model of HW Specification)

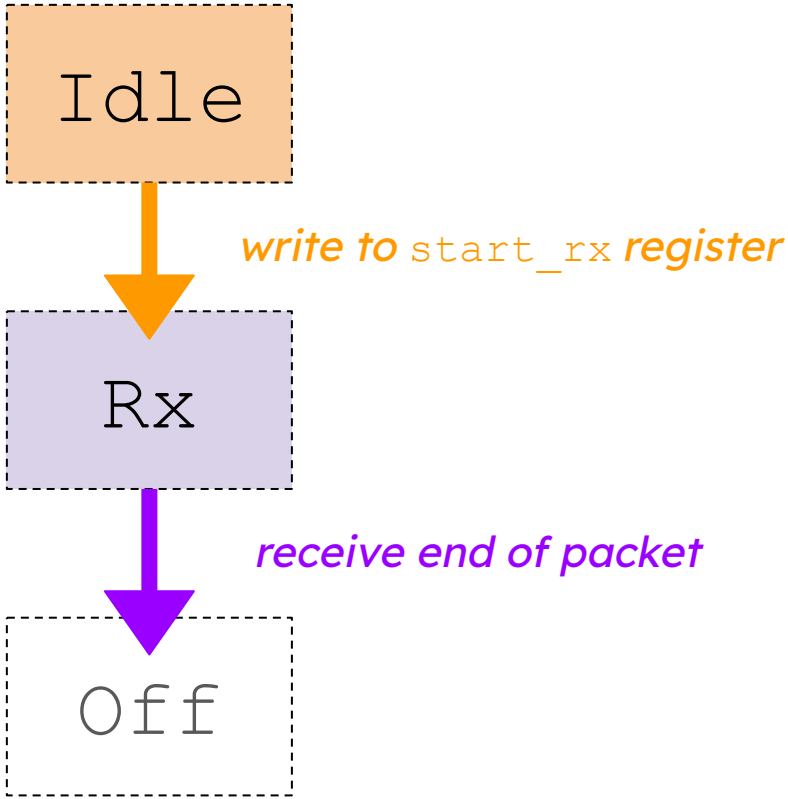
Abacus typestates model hardware-software concurrency

Stable State

- Hw state that *can only be* exited with a software-initiated state transition.

Transient State

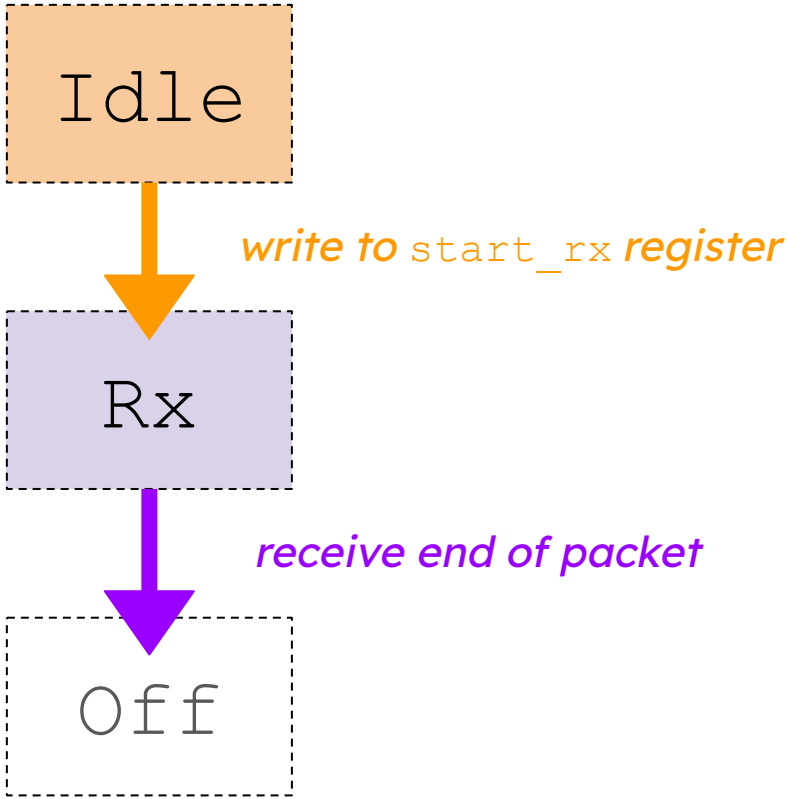
- Hw state with *at least* one hw-initiated state transition.
- Transition from transient state without explicit software involvement.



(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

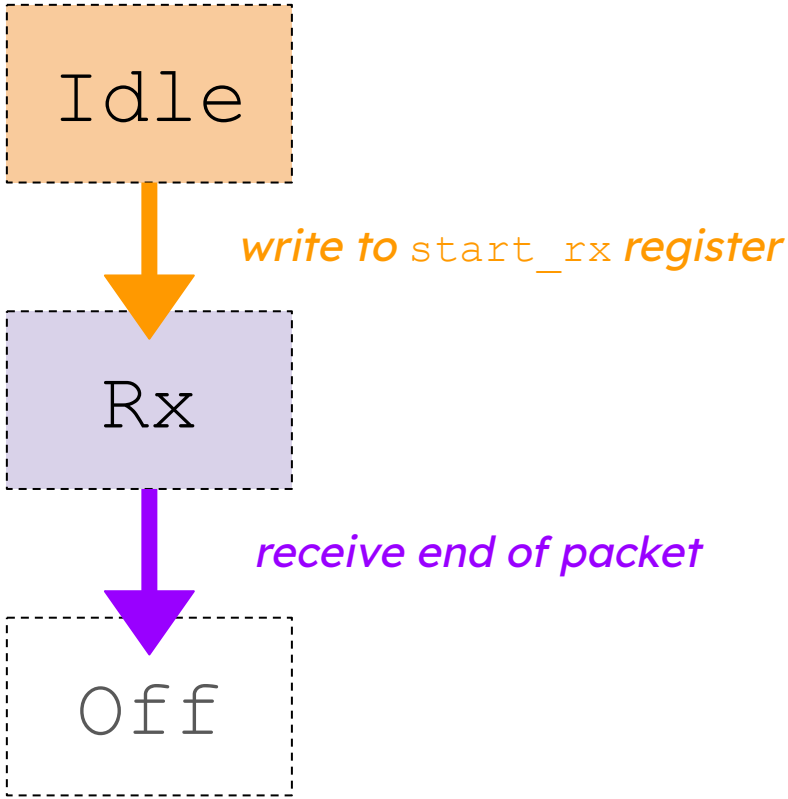
- Stable states can be modeled with out-of-the-box typestates.



(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

- Stable states can be modeled with out-of-the-box typestates.
- Transient states cause typestates to no longer accurately model hw.



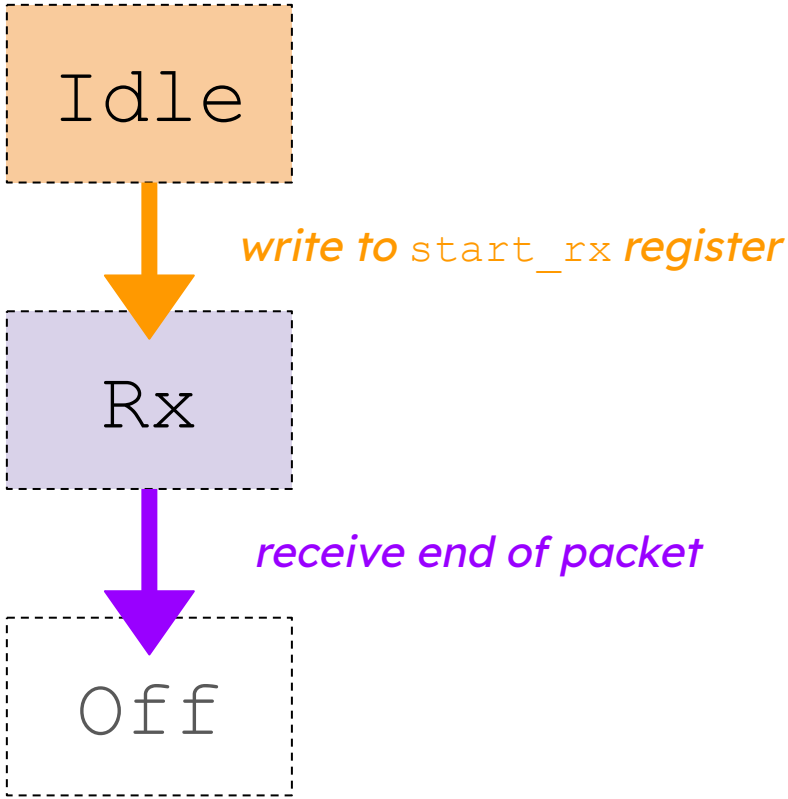
(Developer Mental Model of HW Specification)

Abacus typestates model hardware-software concurrency

- Stable states can be modeled with out-of-the-box typestates.
- Transient states cause typestates to no longer accurately model hw.

Abacus TypeStates

Standard Typestates + restrict transient state operations



(Developer Mental Model of HW Specification)

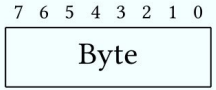
Outline

- Introducing device protocol violations
- How do we build drivers today?
- Our system - ***Abacus***
- **Using *Abacus* to prevent device protocol bugs**
- Evaluation & Closing Thoughts

Let’s build a UART driver…

DATA

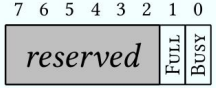
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly

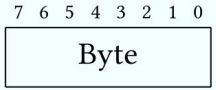


Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

Let's build a UART driver...

DATA

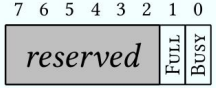
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly



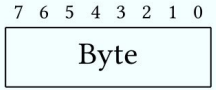
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

```
Push to queue by writing to data reg.
```

Let's build a UART driver…

DATA

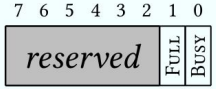
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly



Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

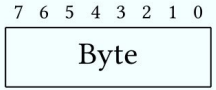
```
Push to queue by writing to data reg.
```

```
Transmit queue popped by hardware.
```

Let's build a UART driver…

DATA

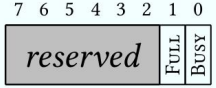
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly



Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

Push to queue by writing to data reg.

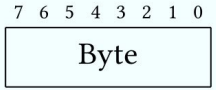
Transmit queue popped by hardware.

DATA must not be written when FULL is asserted.

Let's build a UART driver…

DATA

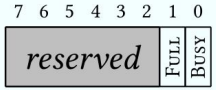
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly



Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

```
QueueReady
```

```
Push to queue by writing to data reg.
```

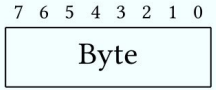
```
Transmit queue popped by hardware.
```

```
DATA must not be written when
FULL is asserted.
```

(Developer Mental Model of HW Specification)

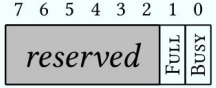
Let's build a UART driver…

DATA
WriteOnly



Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS
ReadOnly



Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

`Push to queue by writing to data reg.`

`Transmit queue popped by hardware.`

`DATA must not be written when FULL is asserted.`

`QueueReady`

`QueueMaybeFull`

(Developer Mental Model of HW Specification)

Let's build a UART driver…

DATA
WriteOnly

7	6	5	4	3	2	1	0
Byte							

Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS
ReadOnly

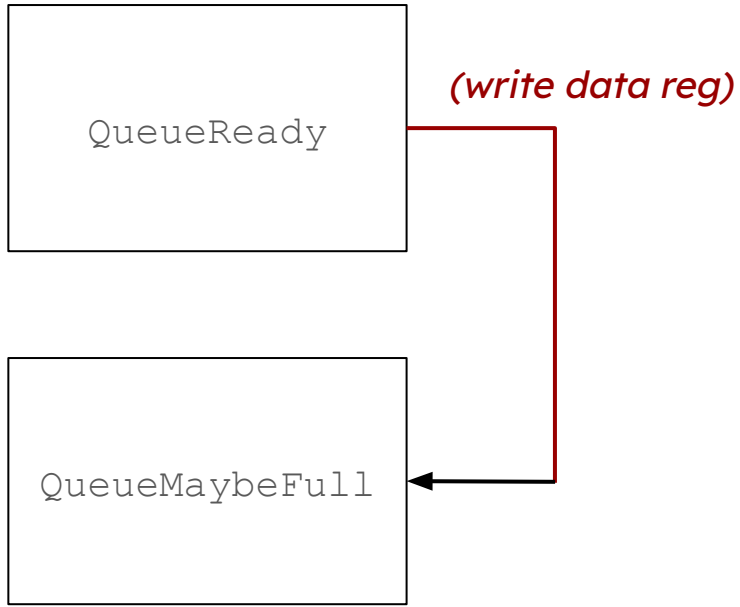
7	6	5	4	3	2	1	0
reserved						FULL	BUSY

Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

Push to queue by writing to data reg.

Transmit queue popped by hardware.

DATA must not be written when FULL is asserted.



(Developer Mental Model of HW Specification)

Let's build a UART driver…

DATA

WriteOnly

7	6	5	4	3	2	1	0
Byte							

Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

STATUS

ReadOnly

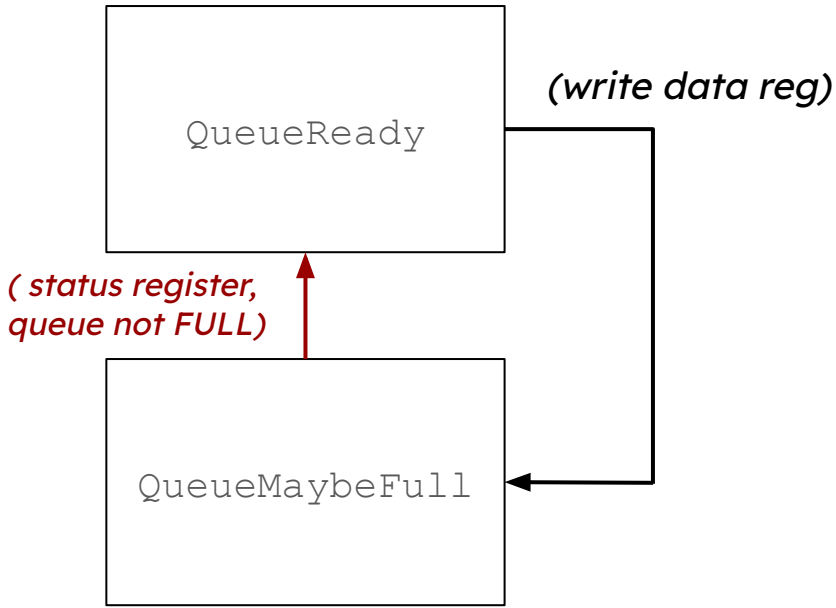
7	6	5	4	3	2	1	0
reserved						FULL	BUSY

Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

Push to queue by writing to data reg.

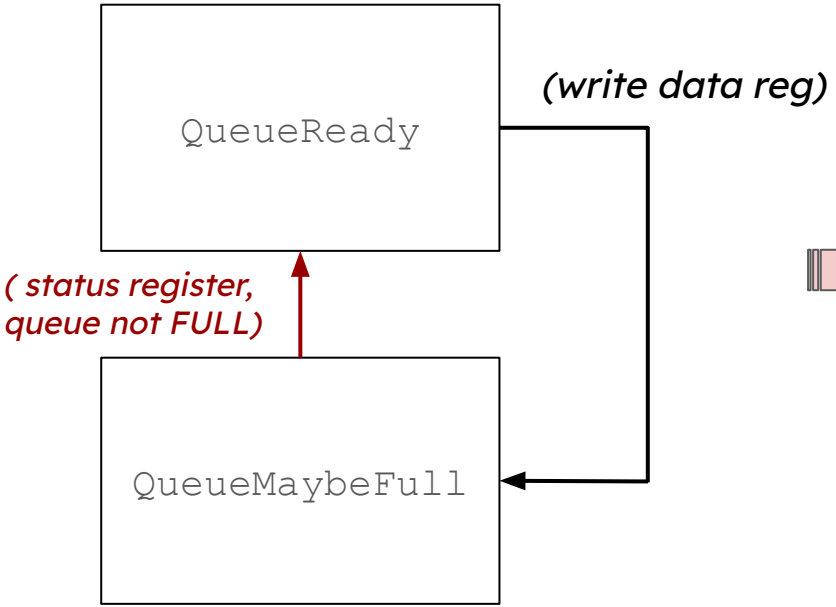
Transmit queue popped by hardware.

DATA must not be written when FULL is asserted.



(Developer Mental Model of HW Specification)

Let’s build a UART driver…

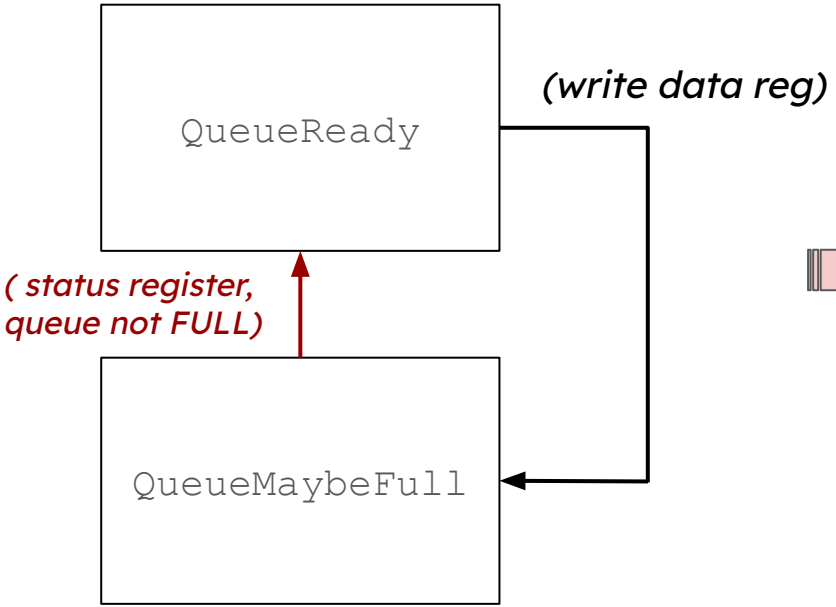



```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                  // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

(Implemented UART driver – based on our mental model)

(Developer Mental Model of HW Specification)

Let's build a UART driver…




```rust
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                  // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

(Implemented UART driver – based on our mental model)

Do you see the bug?

(Developer Mental Model of HW Specification)

Let’s build a UART driver…

```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                 // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

(Implemented UART driver – based on our mental model)

Recall…

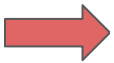
DATA must not be written when FULL is asserted.

Do you see the bug?

Let's build a UART driver…

Violate device protocol!

We assume that the hw transmit queue is NOT full when calling this function



```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                 // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

(Implemented UART driver – based on our mental model)

Recall…

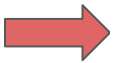
DATA must not be written when FULL is asserted.

Do you see the bug?

Abacus statically prevents this bug!

Violate device protocol!

We assume that the hw transmit queue is NOT full when calling this function



```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                 // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

(Implemented UART driver – based on our mental model)

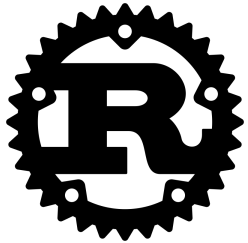
Recall…

DATA must not be written when FULL is asserted.

Do you see the bug?

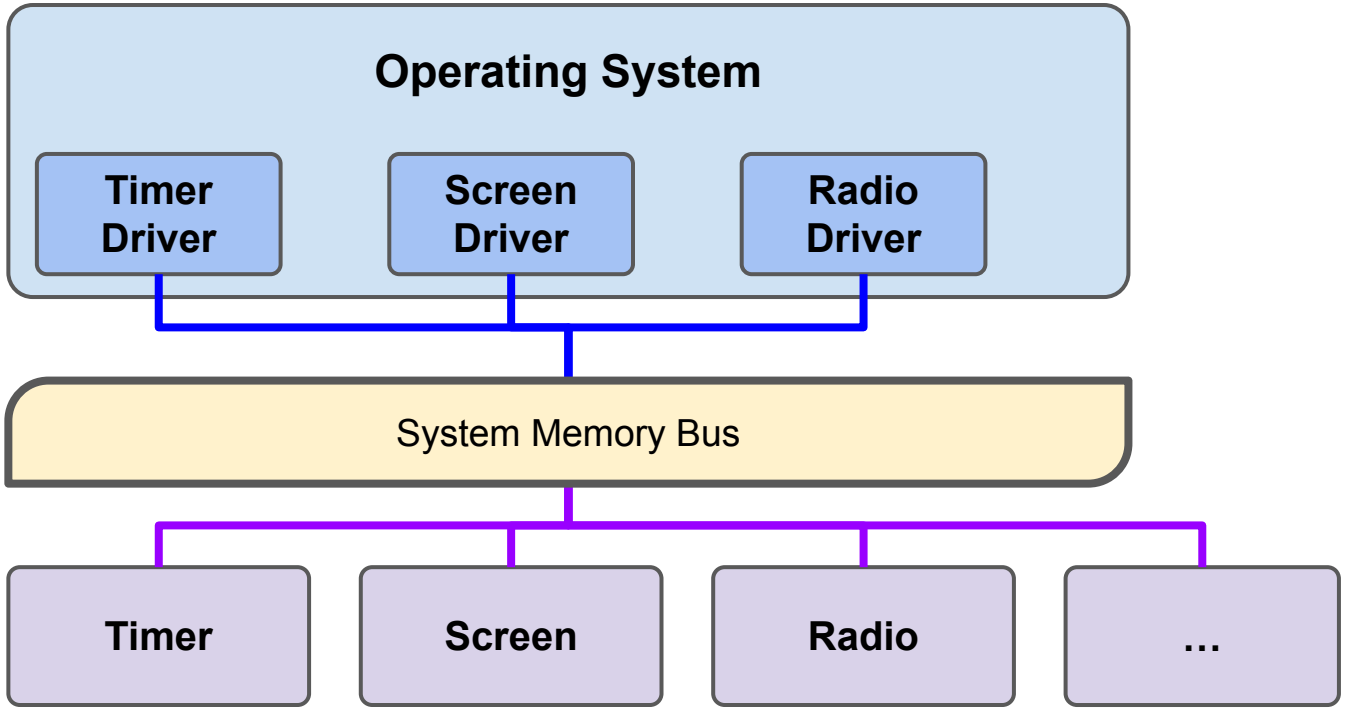
Abacus statically prevents this bug!

Abacus TypeStates



DSL (Abacus Framework)

Observation: Software talks to **hardware** through a "narrow waist" — (e.g. memory-mapped I/O, PCIe, QSPI)



(1) Add DSL Annotations

```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

Original (buggy) UART driver

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

Annotations for updated UART driver

Enforce device protocols by constraining MMIO using typestates.

(1) Add DSL Annotations

- Label states and mark transient states

```
1  struct UartRegisters {
2    data: RegisterWO<u8>,
3    status: RegisterRO<StatusReg>
4  }                 // ^^^^^^^^^^ helper for bitfields
5
6  fn transmit(reg: &UartRegisters, buf: &[u8]) {
7    for index in len(buf):
8      reg.data.write(buf[index])
9      // busy wait until queue has space
10     while (reg.status.read().is_set(StatusReg::FULL) {}
11 }
```

Original (buggy) UART driver

```
1  +#[abacus(states=[ QueueReady,
2  +                  QueueMaybeFull(*T*) ])]
3   struct UartRegisters {
4  +  #[attribute(SC(QueueReady, QueueMaybeFull))]
5     data: RegisterWO<u8>,
6     status: RegisterRO<u8, Status>,
7  }
```

Annotations for updated UART driver

Enforce device protocols by constraining MMIO using typestates.

(1) Add DSL Annotations

- Label states and mark transient states
- Add constraints to registers

```
1  struct UartRegisters {
2    data: RegisterWO<u8>,
3    status: RegisterRO<StatusReg>
4  }                 // ^^^^^^^^^^ helper for bitfields
5
6  fn transmit(reg: &UartRegisters, buf: &[u8]) {
7    for index in len(buf):
8      reg.data.write(buf[index])
9      // busy wait until queue has space
10     while (reg.status.read().is_set(StatusReg::FULL) {}
11 }
```

Original (buggy) UART driver

```
1  +#[abacus(states=[ QueueReady,
2  +                  QueueMaybeFull(*T*) ])]
3   struct UartRegisters {
4  +  #[attribute(SC(QueueReady, QueueMaybeFull))]
5     data: RegisterWO<u8>,
6     status: RegisterRO<u8, Status>,
7  }
```

Annotations for updated UART driver

Enforce device protocols by constraining MMIO using typestates.

(1) Add DSL Annotations

- Label states and mark transient states
- Add constraints to registers

```
1  struct UartRegisters {
2    data: RegisterWO<u8>,
3    status: RegisterRO<StatusReg>
4  }                 // ^^^^^^^^^^ helper for bitfields
5
6  fn transmit(reg: &UartRegisters, buf: &[u8]) {
7    for index in len(buf):
8      reg.data.write(buf[index])
9      // busy wait until queue has space
10     while (reg.status.read().is_set(StatusReg::FULL) {}
11 }
```

Original (buggy) UART driver

```
1  +#[abacus(states=[ QueueReady,
2  +                  QueueMaybeFull(*T*) ])]
3   struct UartRegisters {
4  +  #[attribute(SC(QueueReady, QueueMaybeFull))]
5     data: RegisterWO<u8>,
6     status: RegisterRO<u8, Status>,
7  }
```

Annotations for updated UART driver

Enforce device protocols by constraining MMIO using typestates.

(2) Abacus DSL auto generates Abacus typestates

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

unmodified

auto generated code

(2) Abacus DSL auto generates Abacus typestates

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: AbacusSCRegisterWO<S, u8>,
  status: RegisterRO<u8, Status>,
}
```

1. Modified MMIO register `struct`

unmodified

auto generated code

(2) Abacus DSL auto generates Abacus typestates

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: AbacusSCRegisterWO<S, u8>,
  status: RegisterRO<u8, Status>,
}
```

1. Modified MMIO register `struct`

unmodified

auto generated code

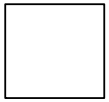
(2) Abacus DSL auto generates Abacus typestates

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: AbacusSCRegisterWO<S, u8>,
  status: RegisterRO<u8, Status>,
}
// Wrapper around state changing registers.
struct AbacusSCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}
```

1. Modified MMIO register `struct`
2. Wrap registers in typestate



unmodified

auto generated code

(2) Abacus DSL auto generates Abacus typestates

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
    data: RegisterWO<u8>,
    status: RegisterRO<u8, Status>,
}
```

DSL Annotations

1. Modified MMIO register `struct`
2. Wrap registers in typestate
3. Only define valid transitions

unmodified

auto generated code

```
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: AbacusSCRegisterWO<S, u8>,
  status: RegisterRO<u8, Status>,
}
// Wrapper around state changing registers.
struct AbacusSCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}
// This impl is only generated for S==QueueReady,
// which enforces the queue's device protocol.
impl UartRegisters<QueueReady> {
    fn write(self, val: u8) ->
      UartRegisters<QueueMaybeFull> {
        self.data.reg.write(val)
      }
}
```

Abacus statically eliminates the device protocol violation.

```
struct UartRegisters {
  data: RegisterWO<u8>,
  status: RegisterRO<StatusReg>
}                 // ^^^^^^^^^^ helper for bitfields

fn transmit(reg: &UartRegisters, buf: &[u8]) {
  for index in len(buf):
    reg.data.write(buf[index])
    // busy wait until queue has space
    while (reg.status.read().is_set(StatusReg::FULL) {}
}
```

Original (buggy) UART driver

DATA must not be written when FULL is asserted.

```
// Driver obj. holds hardware reference & driver-specific state
struct UartDriver {
-  registers: &UartRegisters,
+  registers: AbacusCell<UartStates>,
}

impl UartDriver {
   pub fn transmit(&self, buf: &[u8]) {
     for data in buf.iter() {
-      self.registers.data.write(data);
-      while self.registers.status.is_set(Status::FULL) {};
+      self.registers.map(|state| {
+        match state {
+          UartStates::QueueReady(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

Abacus statically eliminates the device protocol violation.

Recall…

```
// This impl is only generated for S==QueueReady,
// which enforces the queue's device protocol.
impl UartRegisters<QueueReady> {
    fn write(self, val: u8) ->
      UartRegisters<QueueMaybeFull> {
        self.data.reg.write(val)
      }
}
```

DATA must not be written when FULL is asserted.

```
// Driver obj. holds hardware reference & driver-specific state
struct UartDriver {
-  registers: &UartRegisters,
+  registers: AbacusCell<UartStates>,
}

impl UartDriver {
   pub fn transmit(&self, buf: &[u8]) {
     for data in buf.iter() {
-      self.registers.data.write(data);
-      while self.registers.status.is_set(Status::FULL) {};
+      self.registers.map(|state| {
+        match state {
+          UartStates::QueueReady(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

Abacus statically eliminates the device protocol violation.

Recall…

```
// This impl is only generated for S==QueueReady,
// which enforces the queue's device protocol.
impl UartRegisters<QueueReady> {
    fn write(self, val: u8) ->
      UartRegisters<QueueMaybeFull> {
        self.data.reg.write(val)
      }
}
```

DATA must not be written when FULL is asserted.

```
// Driver obj. holds hardware reference & driver-specific state
struct UartDriver {
-  registers: &UartRegisters,
+  registers: AbacusCell<UartStates>,
}

impl UartDriver {
   pub fn transmit(&self, buf: &[u8]) {
     for data in buf.iter() {
-      self.registers.data.write(data);
-      while self.registers.status.is_set(Status::FULL) {};
+      self.registers.map(|state| {
+        match state {
+          UartStates::QueueReady(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

Abacus statically eliminates the device protocol violation.

Recall…

```
// This impl is only generated for S==QueueReady,
// which enforces the queue's device protocol.
impl UartRegisters<QueueReady> {
    fn write(self, val: u8) ->
      UartRegisters<QueueMaybeFull> {
        self.data.reg.write(val)
      }
}
```

```
 impl SyncState for UartRegisters<QueueMaybeFull> {
   fn sync_state(self) -> UartStates
+    // Implemented by developers and trusted by Abacus.
+    {
+      if self.status.is_set(Status::FULL) {
+          UartState::MaybeFull(self)
+      } else {
+         unsafe { UartState::QueueReady(self.into()) }
+      }
 }
```

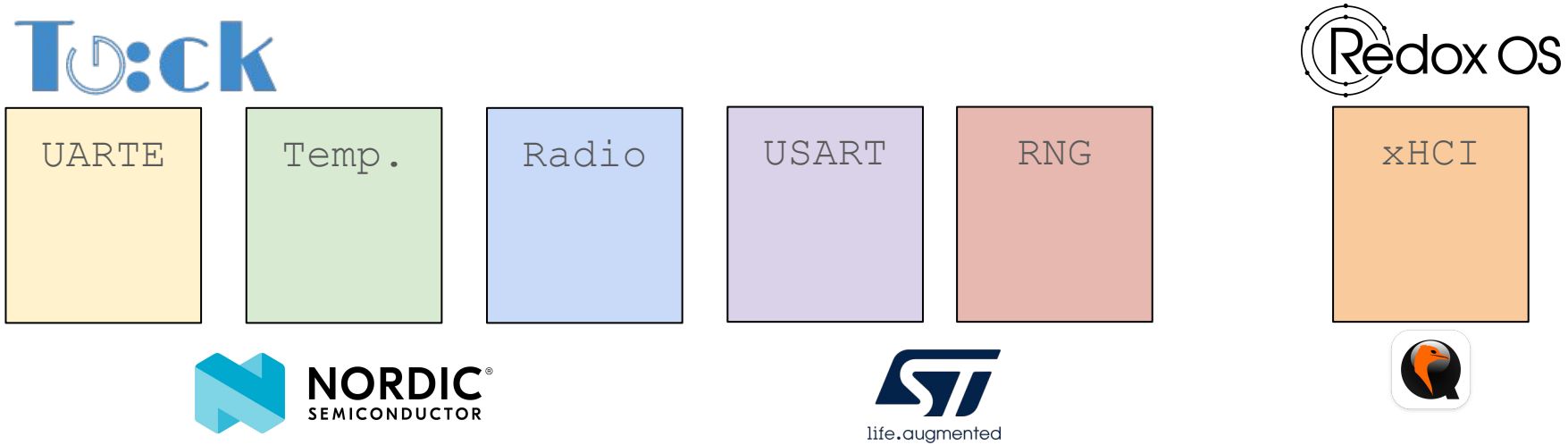
DATA must not be written when FULL is asserted.

```
1  // Driver obj. holds hardware reference & driver-specific state
2  struct UartDriver {
3  -  registers: &UartRegisters,
4  +  registers: AbacusCell<UartStates>,
5  }
6
7  impl UartDriver {
8     pub fn transmit(&self, buf: &[u8]) {
9       for data in buf.iter() {
10 -       self.registers.data.write(data);
11 -       while self.registers.status.is_set(Status::FULL) {};
12 +       self.registers.map(|state| {
13 +         match state {
14 +           UartStates::QueueReady(regs) => {
15 +             regs.data.write(data).sync_state()
16 +           }
17 +           UartStates::QueueMaybeFull(regs) => {
18 +             regs.sync_state() /* no regs.data.write() exists */
19 +           }}});
20 }}}
```

Outline

- Introducing device protocol violations
- How do we build drivers today?
- Our system - ***Abacus***
- Using ***Abacus*** to prevent device protocol bugs
- **Evaluation & Closing Thoughts**

(***Abacus***) Implementation with TockOS and RedoxOS



(Q1) DSL annotation overhead?
(Q2) Binary size overhead?
(Q3) Runtime performance overhead (cycles)?

Abacus DSL adds modest developer overhead.

Driver	Number of States	Abacus Annotations
nRF52 UARTE	5	43 (+)
nRF5x Temperature	2	4 (+)
nRF52 15.4 Radio	8	33 (+)
STM USART	5	45 (+)
STM TRNG	2	13 (+)
xHCI PortSC	5	14 (+)

Lines of Code of Abacus DSL annotations per driver.

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead?
(Q3) Runtime performance overhead (cpu cycles)?

Abacus DSL adds modest developer overhead.

Driver	Number of States	Abacus Annotations
nRF52 UARTE	5	43 (+)
nRF5x Temperature	2	4 (+)
nRF52 15.4 Radio	8	33 (+)
STM USART	5	45 (+)
STM TRNG	2	13 (+)
xHCI PortSC	5	14 (+)

Lines of Code of Abacus DSL annotations per driver.

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead?
(Q3) Runtime performance overhead (cpu cycles)?

Abacus adds zero binary size overhead.

Driver	Platform	Percent Diff
UARTE	Nrf52840	0.00%
Temperature Sensor	Nrf52840	0.00%
IEEE 802.15.4 Radio	Nrf52840	0.00%
TRNG	STM	0.00%
USART	STM	0.00%
xHCI	Redox	-0.33%

Driver binary size – Abacus vs unmodified driver.

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)?

Abacus adds zero binary size overhead.

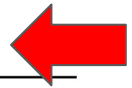
Driver	Platform	Percent Diff
UARTE	Nrf52840	0.00%
Temperature Sensor	Nrf52840	0.00%
IEEE 802.15.4 Radio	Nrf52840	0.00%
TRNG	STM	0.00%
USART	STM	0.00%
xHCI	Redox	-0.33%

Driver binary size – Abacus vs unmodified driver.

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)?

Abacus adds zero binary size overhead.

Driver	Platform	Percent Diff
UARTE	Nrf52840	0.00%
Temperature Sensor	Nrf52840	0.00%
IEEE 802.15.4 Radio	Nrf52840	0.00%
TRNG	STM	0.00%
USART	STM	0.00%
xHCI	Redox	-0.33%



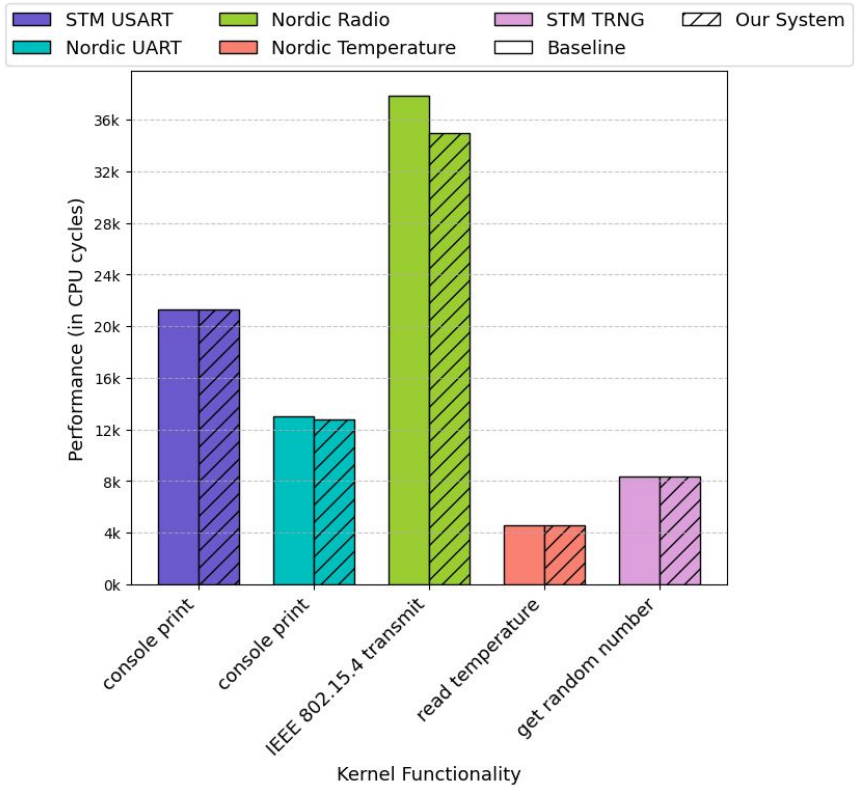
Driver binary size – Abacus vs unmodified driver.

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)?

Abacus adds minimal runtime overhead.

Function	Baseline (V. Cycles)	Abacus (V. Cycles)	Percent Diff
`attach_device`	638×10^9	544×10^9	-17.4%
`get_pls`	41.5×10^3	37.6×10^3	-9.45%

Microbenchmark of Redox xHCI driver methods measured as virtual cycles in QEMU x86_64.



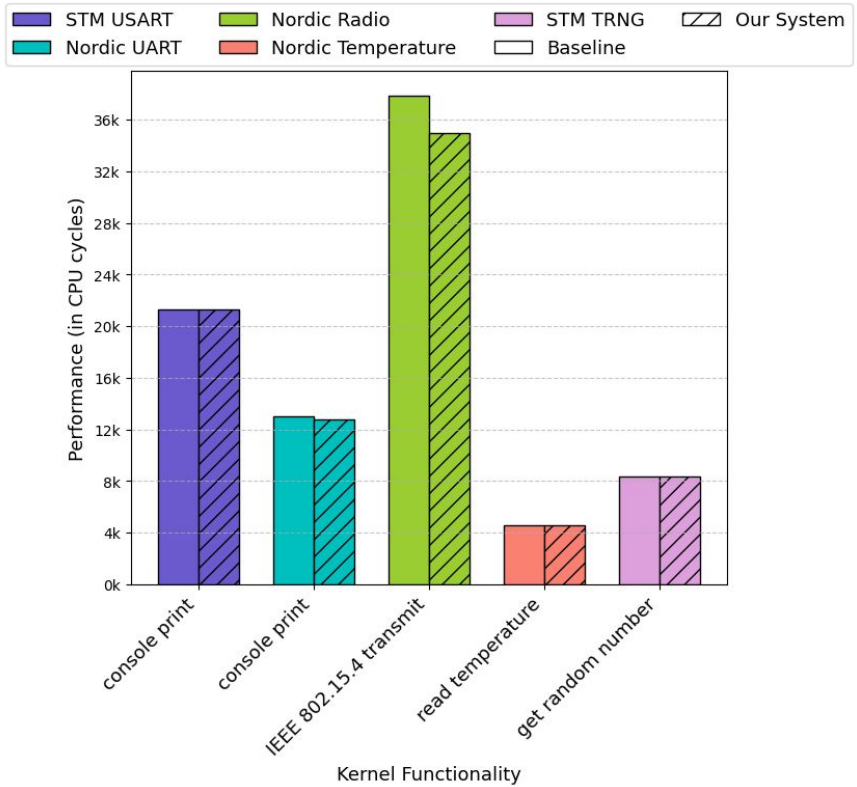
MacroBenchmark of Tock driver Performance (in CPU cycles).

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)? - minimal

Abacus adds minimal runtime overhead.

Function	Baseline (V. Cycles)	Abacus (V. Cycles)	Percent Diff
`attach_device`	638×10^9	544×10^9	-17.4%
`get_pls`	41.5×10^3	37.6×10^3	-9.45%

Microbenchmark of Redox xHCI driver methods measured as virtual cycles in QEMU x86_64.



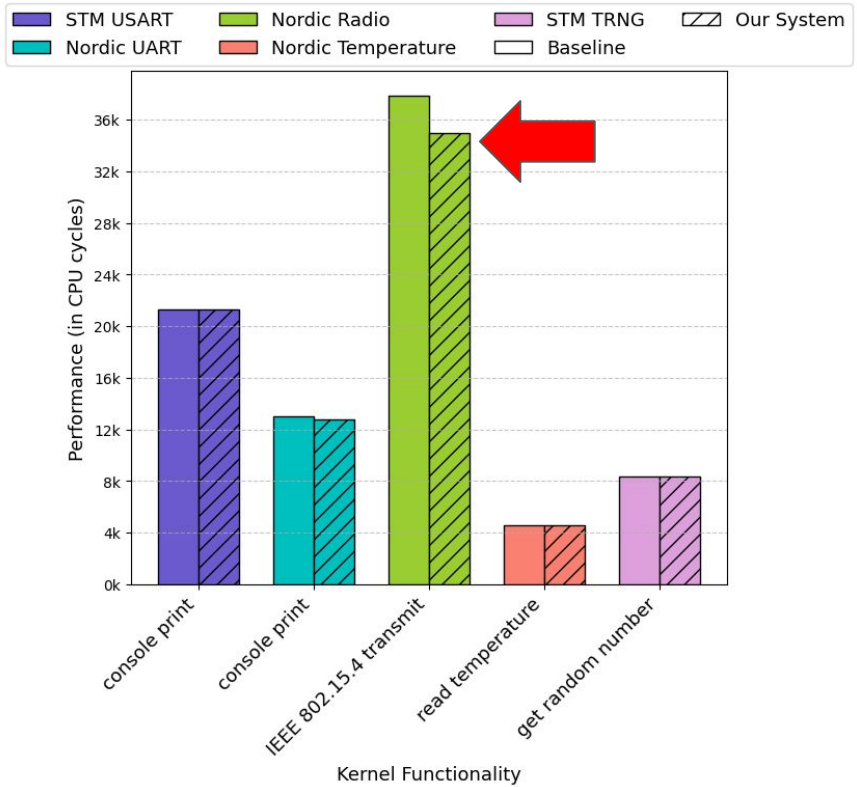
MacroBenchmark of Tock driver Performance (in CPU cycles).

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)? - minimal

Abacus adds minimal runtime overhead.

Function	Baseline (V. Cycles)	Abacus (V. Cycles)	Percent Diff
`attach_device`	638×10^9	544×10^9	-17.4%
`get_pls`	41.5×10^3	37.6×10^3	-9.45%

Microbenchmark of Redox xHCI driver methods measured as virtual cycles in QEMU x86_64.



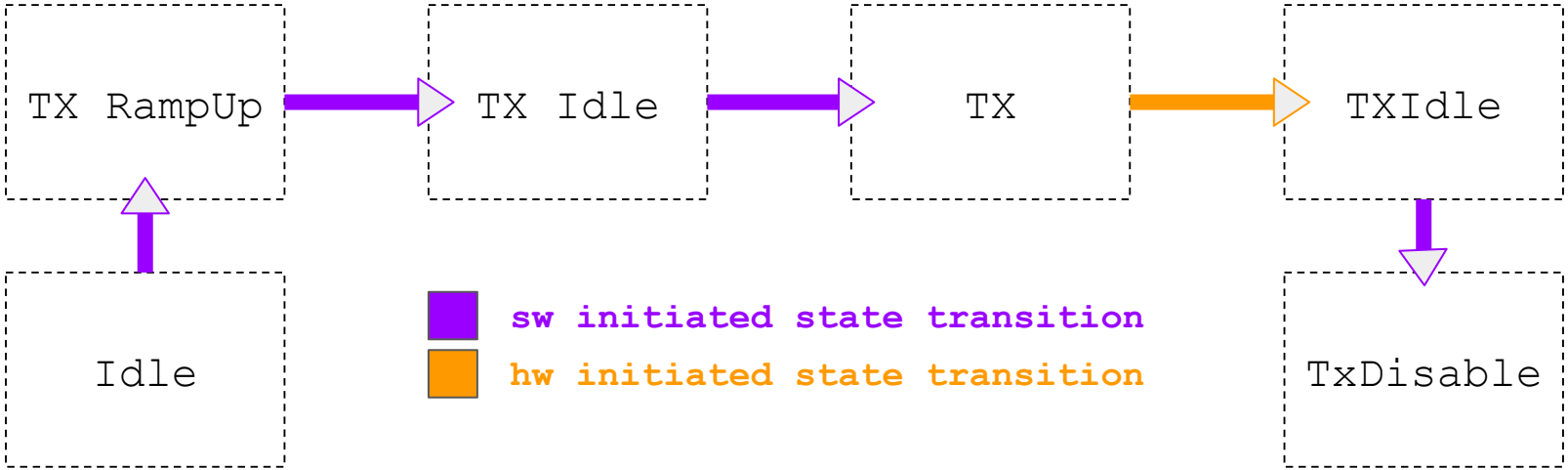
MacroBenchmark of Tock driver Performance (in CPU cycles).

(Q1) DSL annotation overhead? - modest
(Q2) Binary size overhead? - none
(Q3) Runtime performance overhead (cpu cycles)? - minimal

Case Study – NRF52 IEEE802.15.4 Radio Driver

IEEE802.15.4 radio provides HW features to allow for more efficient radio driver operation (e.g., hw shortcuts)

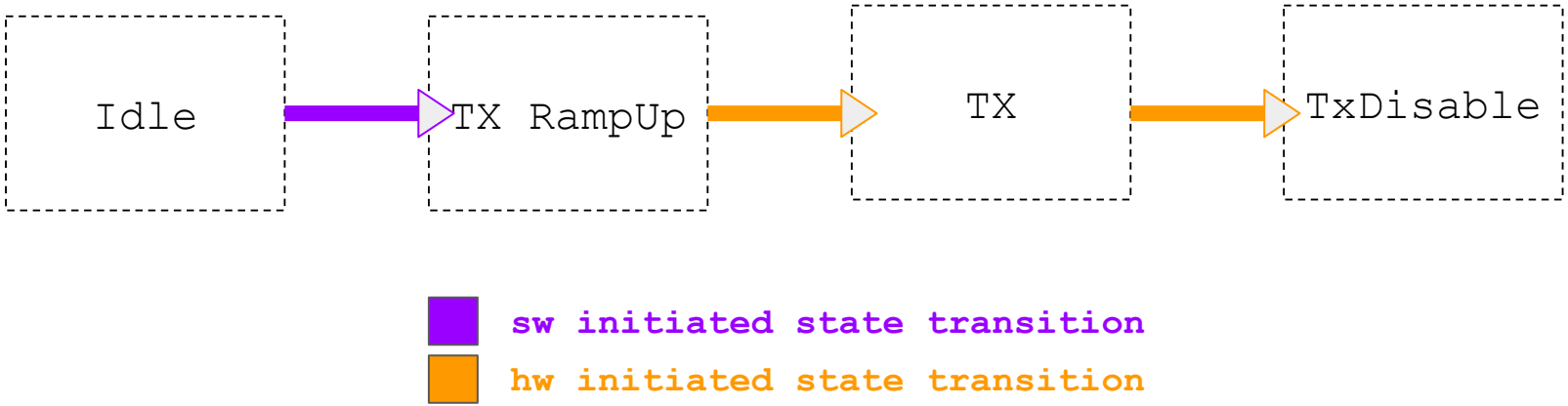
State Machine without HW shortcuts



Case Study – NRF52 IEEE802.15.4 Radio Driver

IEEE802.15.4 radio provides HW features to allow for more efficient radio driver operation (e.g., hw shortcuts)

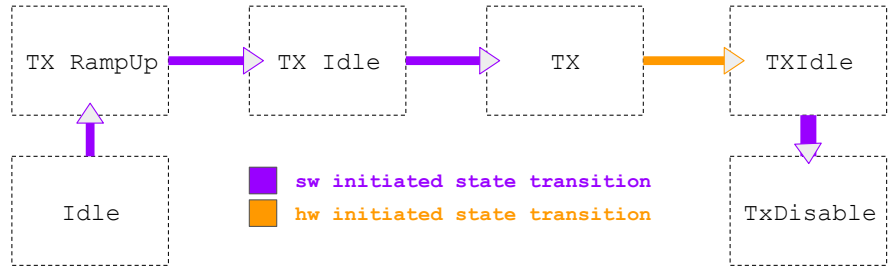
State Machine with *READY_START*, *END_DISABLE* HW shortcuts



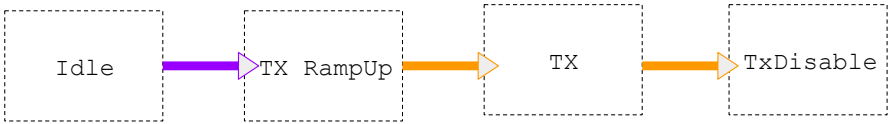
Case Study – NRF52 IEEE802.15.4 Radio Driver

IEEE802.15.4 radio provides HW features to allow for more efficient radio driver operation (e.g., hw shortcuts)

State Machine without HW shortcuts



State Machine with *READY_START*, *END_DISABLE* HW shortcuts

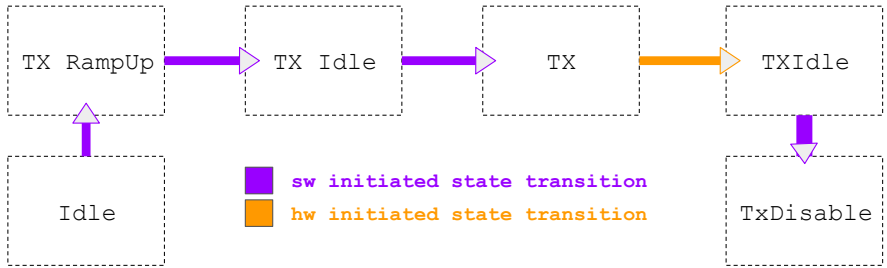


Case Study – NRF52 IEEE802.15.4 Radio Driver

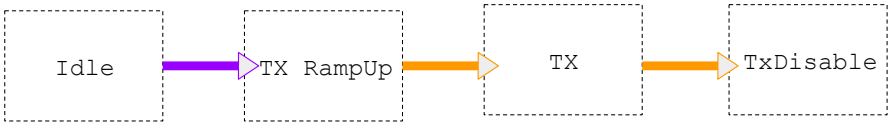
IEEE802.15.4 radio provides HW features to allow for more efficient radio driver operation (e.g., hw shortcuts)

Driver's do not use all available HW shortcuts!

State Machine without HW shortcuts



State Machine with *READY_START*, *END_DISABLE* HW shortcuts



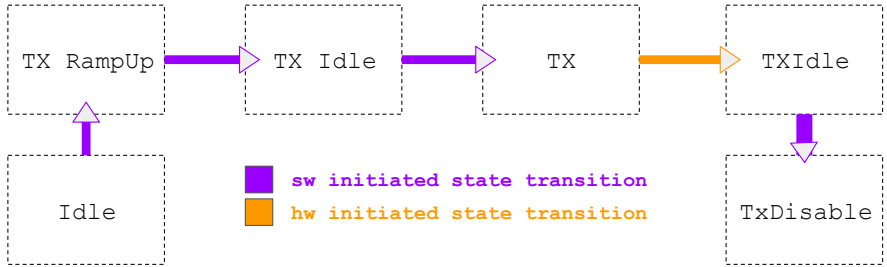
Case Study – NRF52 IEEE802.15.4 Radio Driver

IEEE802.15.4 radio provides HW features to allow for more efficient radio driver operation (e.g., hw shortcuts)

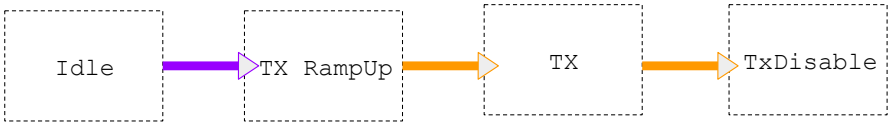
Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

State Machine without HW shortcuts



State Machine with *READY_START*, *END_DISABLE* HW shortcuts



Case Study – NRF52 IEEE802.15.4 Radio Driver

Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

1. Integrate Abacus into Tock's 15.4 driver.

Case Study – NRF52 IEEE802.15.4 Radio Driver

Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

1. Integrate Abacus into Tock's 15.4 driver.
2. Update Abacus DSL state machine to use all 4 transmit HW shortcuts.

Case Study – NRF52 IEEE802.15.4 Radio Driver

Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

1. Integrate Abacus into Tock's 15.4 driver.
2. Update Abacus DSL state machine to use all 4 transmit HW shortcuts.
3. Compiler identifies all invalid transitions.

Case Study – NRF52 IEEE802.15.4 Radio Driver

Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

1. Integrate Abacus into Tock's 15.4 driver.
2. Update Abacus DSL state machine to use all 4 transmit HW shortcuts.
3. Compiler identifies all invalid transitions.

Updated and working driver in ~2 hours!

50% decrease in driver interrupts; 8% runtime improvement

Case Study – NRF52 IEEE802.15.4 Radio Driver

Tock *(2/4) TX HW shortcuts*

Driver's do not use all available HW shortcuts!

1. Integrate Abacus into Tock's 15.4 driver.
2. Update Abacus DSL state machine to use all 4 transmit HW shortcuts.
3. Compiler identifies all invalid transitions.

Updated and working driver in ~2 hours!

50% decrease in driver interrupts; 8% runtime improvement

Abacus allows driver developers to easily use complex, hardware features that may otherwise be "too hard to get right"

Abacus statically prevents device protocol violations using typestates.

Imposes minimal to no code size and runtime overheads.

Source code: https://github.com/abacus-rs/abacus-registers

Rust crate: https://crates.io/crates/abacus-registers

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Generated by the DSL (simplified for exposition)
enum UartStates {
  QueueReady(UartRegisters<QueueReady>),
  MaybeFull(UartRegisters<QueueMaybeFull>),
}
```

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Generated by the DSL (simplified for exposition)
enum UartStates {
  QueueReady(UartRegisters<QueueReady>),
  MaybeFull(UartRegisters<QueueMaybeFull>),
}

// Necessary interfaces are referenced by the DSL, which
// triggers compiler hints for missing impls if-needed.
impl SyncState for UartRegisters<QueueMaybeFull> {
  fn sync_state(self) -> UartStates
+   // Implemented by developers and trusted by Abacus.
+   {
+     if self.status.is_set(Status::FULL) {
+         UartState::MaybeFull(self)
+     } else {
+        unsafe { UartState::QueueReady(self.into()) }
+     }
}
```

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Generated by the DSL (simplified for exposition)
enum UartStates {
  QueueReady(UartRegisters<QueueReady>),
  MaybeFull(UartRegisters<QueueMaybeFull>),
}

// Necessary interfaces are referenced by the DSL, which
// triggers compiler hints for missing impls if-needed.
impl SyncState for UartRegisters<QueueMaybeFull> {
  fn sync_state(self) -> UartStates
+   // Implemented by developers and trusted by Abacus.
+   {
+     if self.status.is_set(Status::FULL) {
+         UartState::MaybeFull(self)
+     } else {
+        unsafe { UartState::QueueReady(self.into()) }
+     }
}
```

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Generated by the DSL (simplified for exposition)
enum UartStates {
  QueueReady(UartRegisters<QueueReady>),
  MaybeFull(UartRegisters<QueueMaybeFull>),
}

// Necessary interfaces are referenced by the DSL, which
// triggers compiler hints for missing impls if-needed.
impl SyncState for UartRegisters<QueueMaybeFull> {
  fn sync_state(self) -> UartStates
+   // Implemented by developers and trusted by Abacus.
+   {
+     if self.status.is_set(Status::FULL) {
+         UartState::MaybeFull(self)
+     } else {
+        unsafe { UartState::QueueReady(self.into()) }
+     }
}
```

developer implemented

auto generated code

(3) Implement Resync. Mechanism for Transient States

```
+#[abacus(states=[ QueueReady,
+                  QueueMaybeFull(*T*) ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady, QueueMaybeFull))]
   data: RegisterWO<u8>,
   status: RegisterRO<u8, Status>,
}
```

DSL Annotations

```
// Generated by the DSL (simplified for exposition)
enum UartStates {
  QueueReady(UartRegisters<QueueReady>),
  MaybeFull(UartRegisters<QueueMaybeFull>),
}

// Necessary interfaces are referenced by the DSL, which
// triggers compiler hints for missing impls if-needed.
impl SyncState for UartRegisters<QueueMaybeFull> {
  fn sync_state(self) -> UartStates
+   // Implemented by developers and trusted by Abacus.
+   {
+     if self.status.is_set(Status::FULL) {
+         UartState::MaybeFull(self)
+     } else {
+         unsafe { UartState::QueueReady(self.into()) }
+     }
}
```

developer implemented

auto generated code

A TypeStated Queue

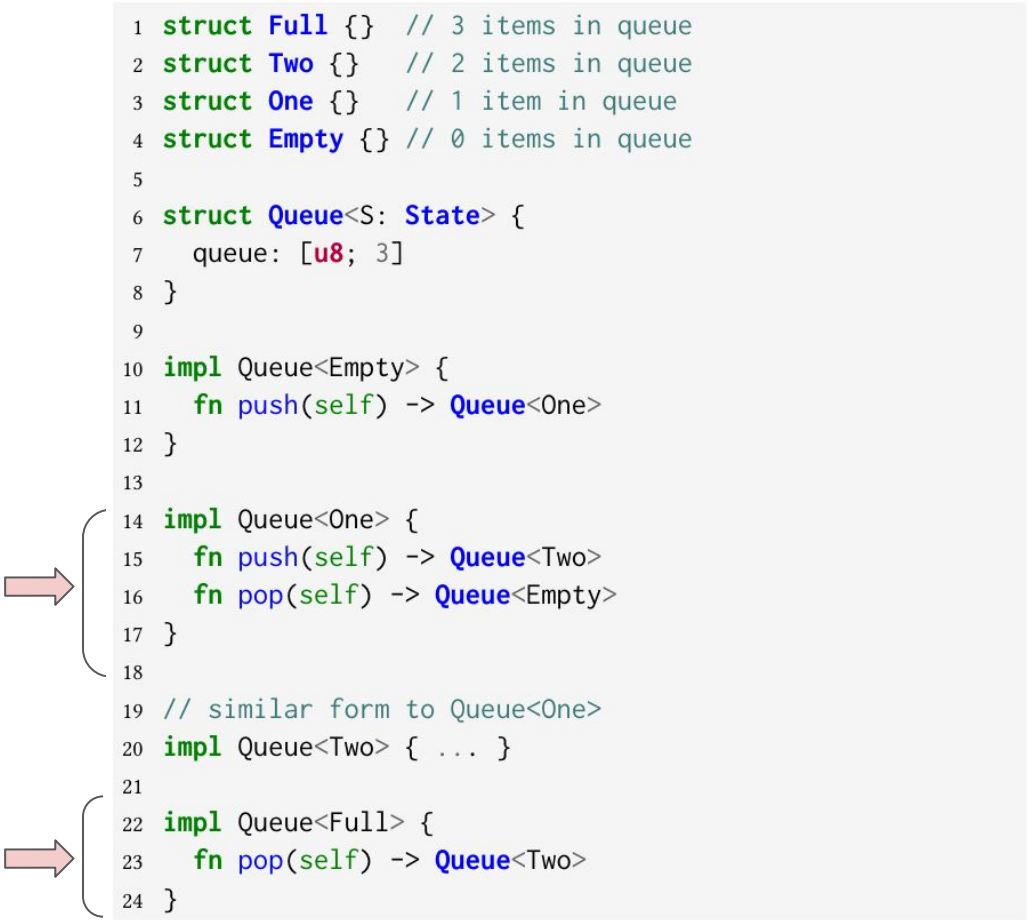
```
1 struct Full {}  // 3 items in queue
2 struct Two {}   // 2 items in queue
3 struct One {}   // 1 item in queue
4 struct Empty {} // 0 items in queue
5
6 struct Queue<S: State> {
7   queue: [u8; 3]
8 }
```

- Encode system properties into the type-system.

(Using typestates to *statically enforce a correct implementation for a queue* of size 3)

A TypeStated Queue

- Encode system properties into the type-system.
- Define valid operations as functions on respective type.

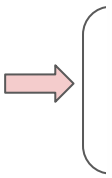


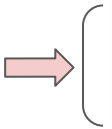
```rust
struct Full {}  // 3 items in queue
struct Two {}   // 2 items in queue
struct One {}   // 1 item in queue
struct Empty {} // 0 items in queue

struct Queue<S: State> {
  queue: [u8; 3]
}

impl Queue<Empty> {
  fn push(self) -> Queue<One>
}

impl Queue<One> {
  fn push(self) -> Queue<Two>
  fn pop(self) -> Queue<Empty>
}

// similar form to Queue<One>
impl Queue<Two> { ... }

impl Queue<Full> {
  fn pop(self) -> Queue<Two>
}
```

(Using typestates to *statically enforce a correct implementation for a queue* of size 3)

A TypeStated Queue

- Encode system properties into the type-system.
- Define valid operations as functions on respective type.
- Incorrect usages result in a compilation error!





```rust
struct Full {}  // 3 items in queue
struct Two {}   // 2 items in queue
struct One {}   // 1 item in queue
struct Empty {} // 0 items in queue

struct Queue<S: State> {
  queue: [u8; 3]
}

impl Queue<Empty> {
  fn push(self) -> Queue<One>
}

impl Queue<One> {
  fn push(self) -> Queue<Two>
  fn pop(self) -> Queue<Empty>
}

// similar form to Queue<One>
impl Queue<Two> { ... }

impl Queue<Full> {
  fn pop(self) -> Queue<Two>
}
```

(Using typestates to *statically enforce a correct implementation for a queue* of size 3)

A TypeStated Queue

Recall from hw spec…

> The UART transmits whenever queue is non-empty and pops entries once sent.

Out-of-the-box typestates cannot model this state transition!

(Using typestates to *statically enforce a correct implementation for a queue* of size 3)

```
1  struct Full {}  // 3 items in queue
2  struct Two {}   // 2 items in queue
3  struct One {}   // 1 item in queue
4  struct Empty {} // 0 items in queue
5
6  struct Queue<S: State> {
7    queue: [u8; 3]
8  }
9
10 impl Queue<Empty> {
11   fn push(self) -> Queue<One>
12 }
13
14 impl Queue<One> {
15   fn push(self) -> Queue<Two>
16   fn pop(self) -> Queue<Empty>
17 }
18
19 // similar form to Queue<One>
20 impl Queue<Two> { ... }
21
22 impl Queue<Full> {
23   fn pop(self) -> Queue<Two>
24 }
```