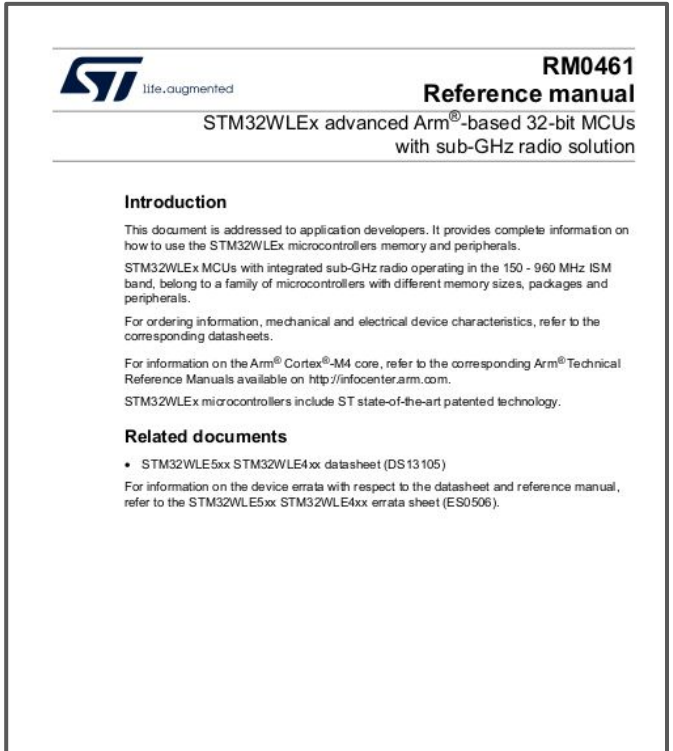
Just because you can doesn't mean you should…

TypeStates for Increased Driver Correctness

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

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

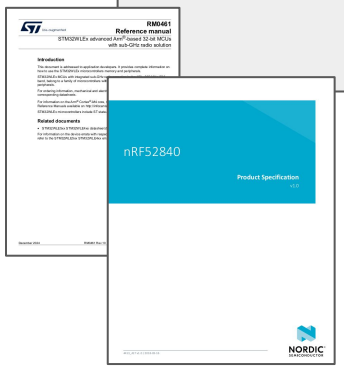
v1.0

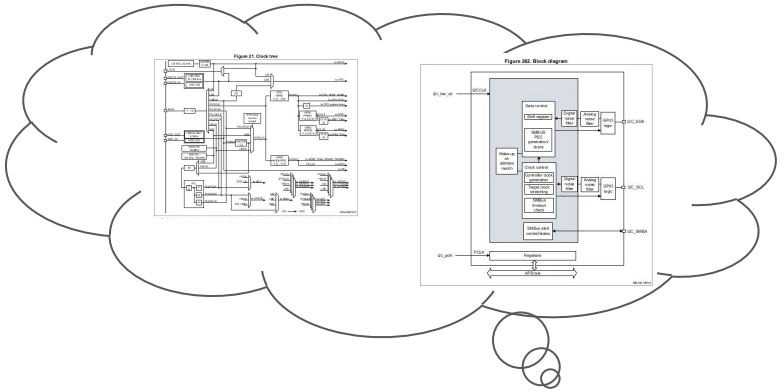
4413_417 v1.0 / 2018-03-16

NORDIC SEMICONDUCTOR

Low-level Driver Development

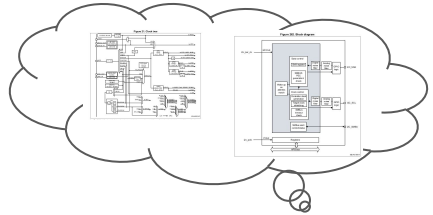
Specification / reference manual





Low-level Driver Development

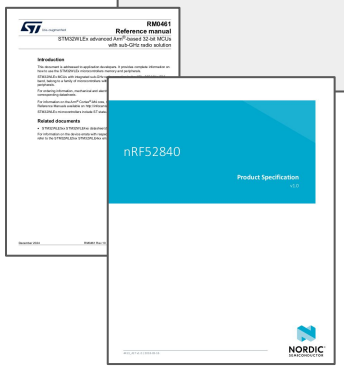
Specification / reference manual

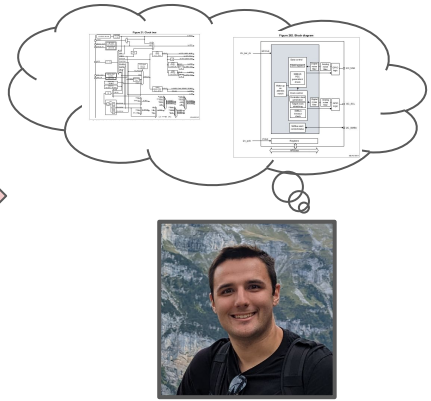


```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

Low-level Driver Development

Specification / reference manual



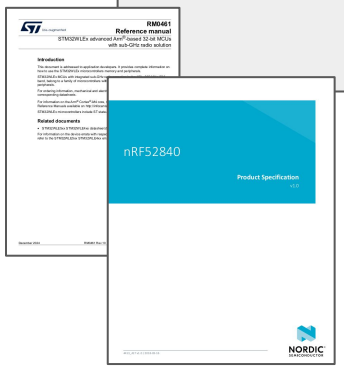


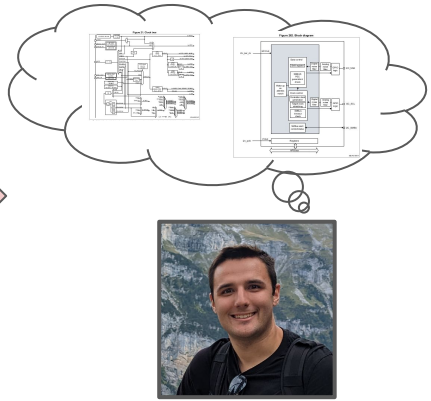
What can go wrong here? :)

```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<'a> TWI<'a> {
    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])
    }
```

Low-level Driver Development

Specification / reference manual



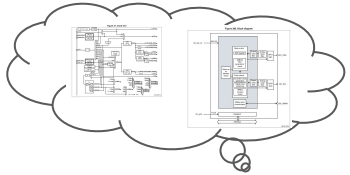


Challenging!

```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<'a> TWI<'a> {
    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])
    }
```

What can go wrong here? :)

Why is this challenging?



Challenging!

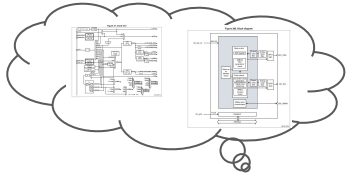
```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

Implemented driver must “comply” with the hw specification.

a. Validity of a given MMIO operation depends on the current hardware state.

b. Hardware may transition the hardware state without input from the driver.

Why is this important?

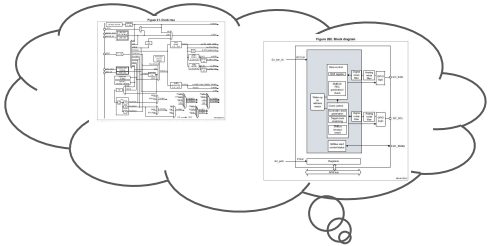


Challenging!

```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<'a> TWI<'a> {
    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])
    }
```

Driver MMIO operations that violate and do not comply with the hw spec…

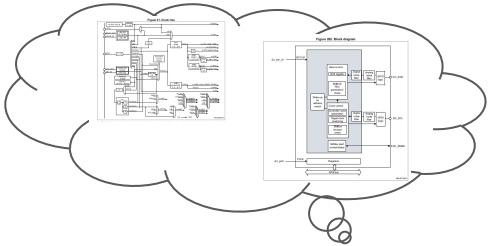
- may result in a buggy driver.
- at worst, may cause systematic failures (e.g. hanging the system's bus).



Challenging!

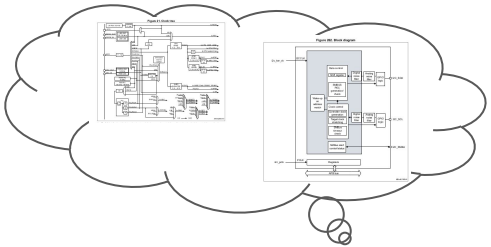
```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

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




```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<'a> TWI<'a> {
    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])
    }
```

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



Verified!

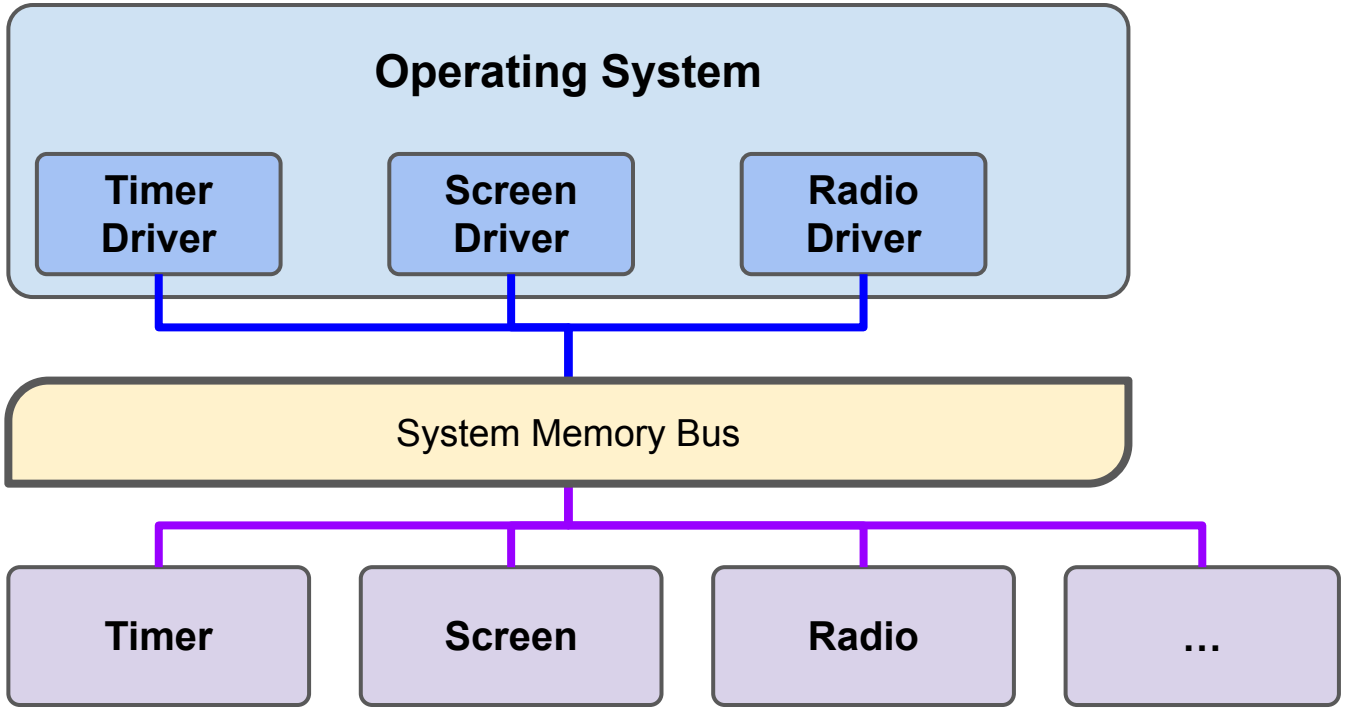
```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

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

*We introduce **<u>state safety</u>**—*

- software driver adheres to hardware's specification

Key Insight: Software talks to **hardware** through a "narrow waist" — memory-mapped I/O



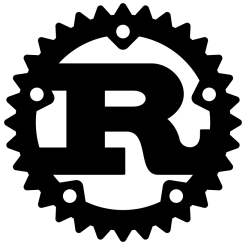
We present a framework that statically (compile-time) guarantees state safety

***State safety*—**

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

Achieve state safety enforcement with minimal-to-no overheads in runtime and code size

TypeStates



DSL

Outline

- Introducing **state safety**
- **How do we build drivers today?**
- TypeState programming
- Our System
- Evaluation & Closing Thoughts

Let’s build a UART driver…

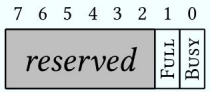
DATA

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



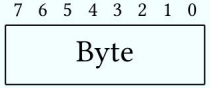
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

(Hypothetical UART Hardware Specification)

Let’s build a UART driver…

DATA

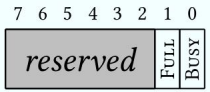
WriteOnly



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

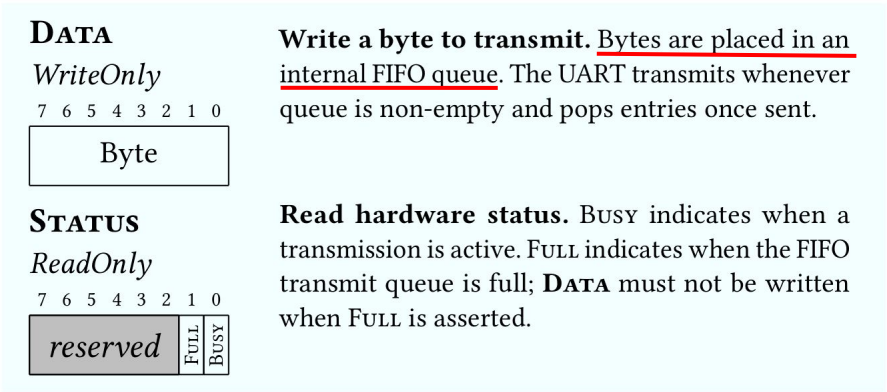
ReadOnly



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.

(Hypothetical UART Hardware 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.

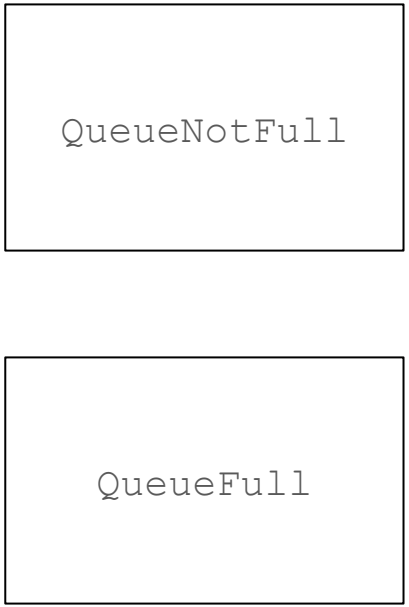
STATUS

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.

(Hypothetical UART Hardware Specification)



(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.

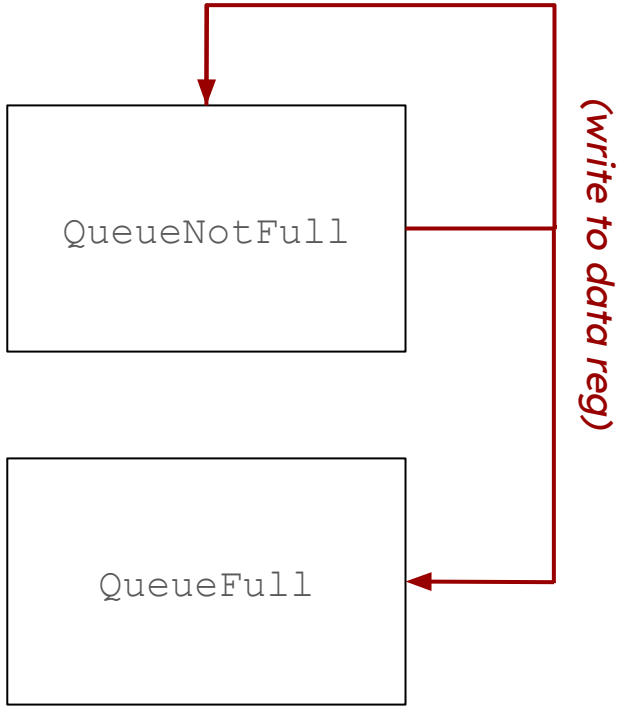
STATUS

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.

(Hypothetical UART Hardware Specification)



(Developer Mental Model of HW Specification)

Let's build a UART driver…

DATA

WriteOnly

7	6	5	4	3	2	1	0
Byte							

STATUS

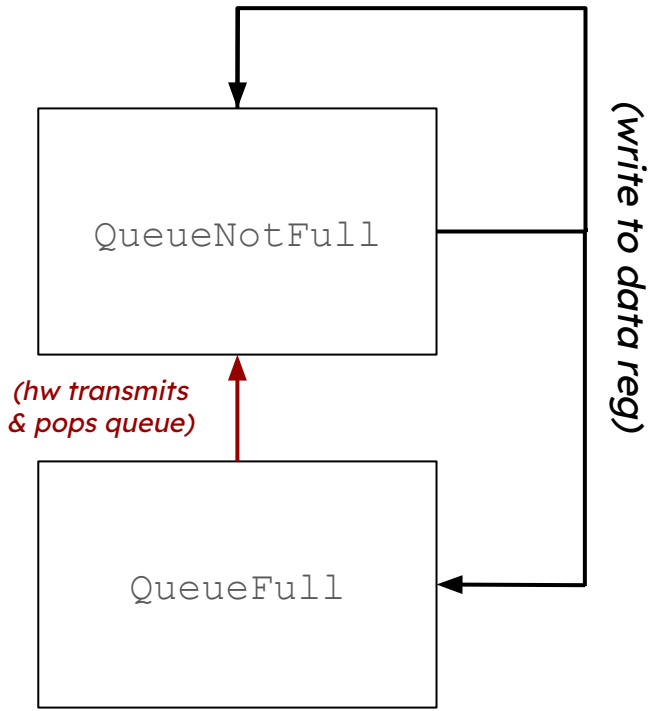
ReadOnly

7	6	5	4	3	2	1	0
reserved						FULL	BUSY

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.

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.

(Hypothetical UART Hardware Specification)



(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.

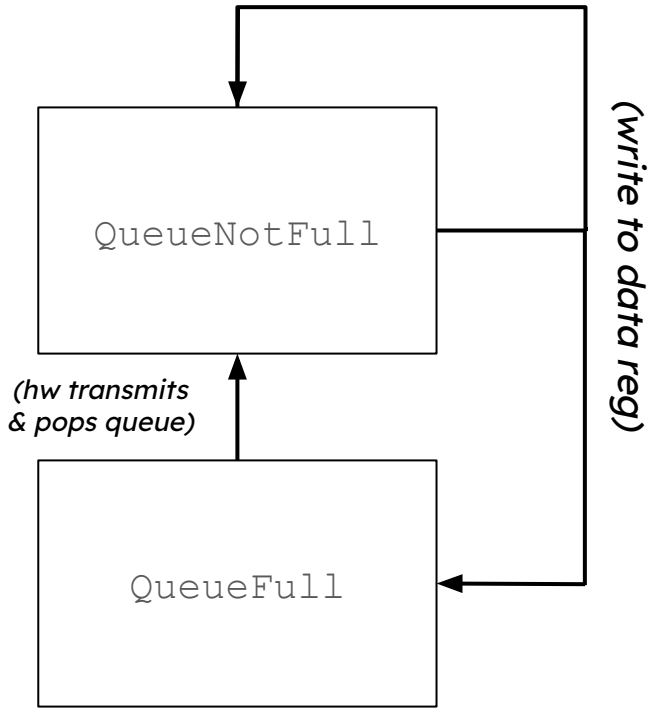
STATUS

ReadOnly

7	6	5	4	3	2	FULL	BUSY
reserved							

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.

(Hypothetical UART Hardware Specification)



(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

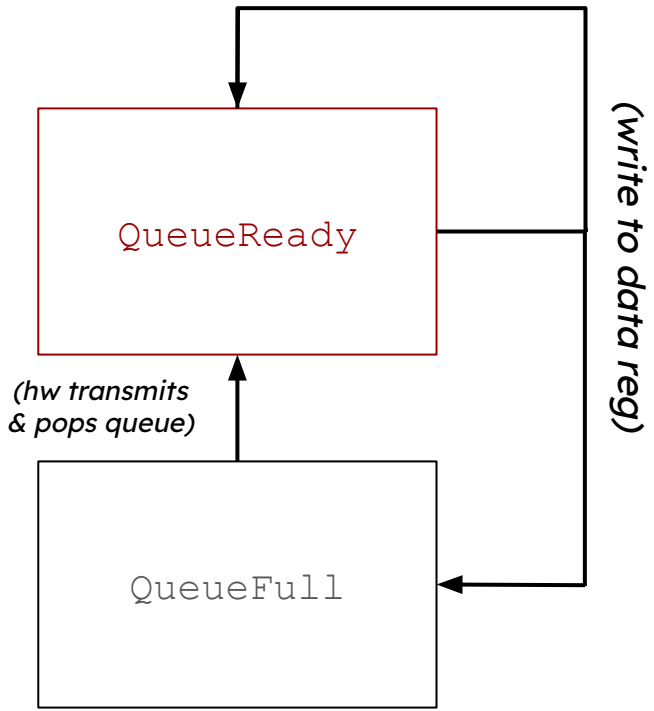
ReadOnly

7 6 5 4 3 2

reserved | FULL | BUSY

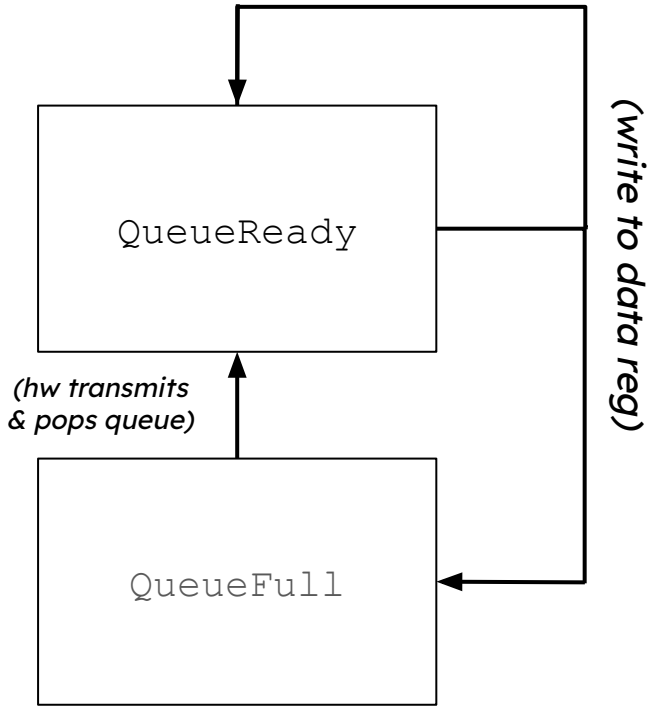
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

(Hypothetical UART Hardware Specification)



(Developer Mental Model of HW Specification)

Let’s build a UART driver…

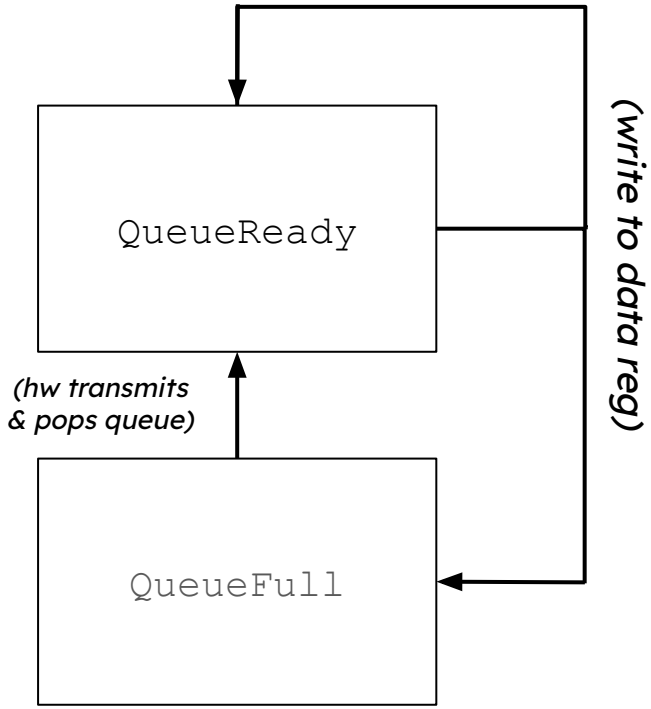


(Developer Mental Model of HW Specification)

```
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)

Let's build a UART driver...



(Developer Mental Model of HW Specification)

```
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 }
```

(Implemented UART driver – based on our mental model)

Do you see the bug?

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…

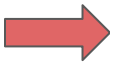
DATA must not be written when FULL is asserted.

Do you see the bug?

Let's build a UART driver…

Violate state safety!

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



```
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 }
```

(Implemented UART driver – based on our mental model)

Recall…

DATA must not be written when FULL is asserted.

Do you see the bug?

How might we prevent this bug?

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 this bug?

Standard approaches for enforcing system properties (generally)…

Testing

(only proves the absence of tested bugs).

TypeState Programming – our approach!

Formal Verification

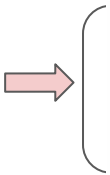
(challenging; requires domain specific expertise).

Outline

- Introducing **state safety**
- How do we build drivers today?
- **TypeState programming**
- Our System
- Evaluation & Closing Thoughts

A TypeStated Queue

- Encode system properties into the type-system.

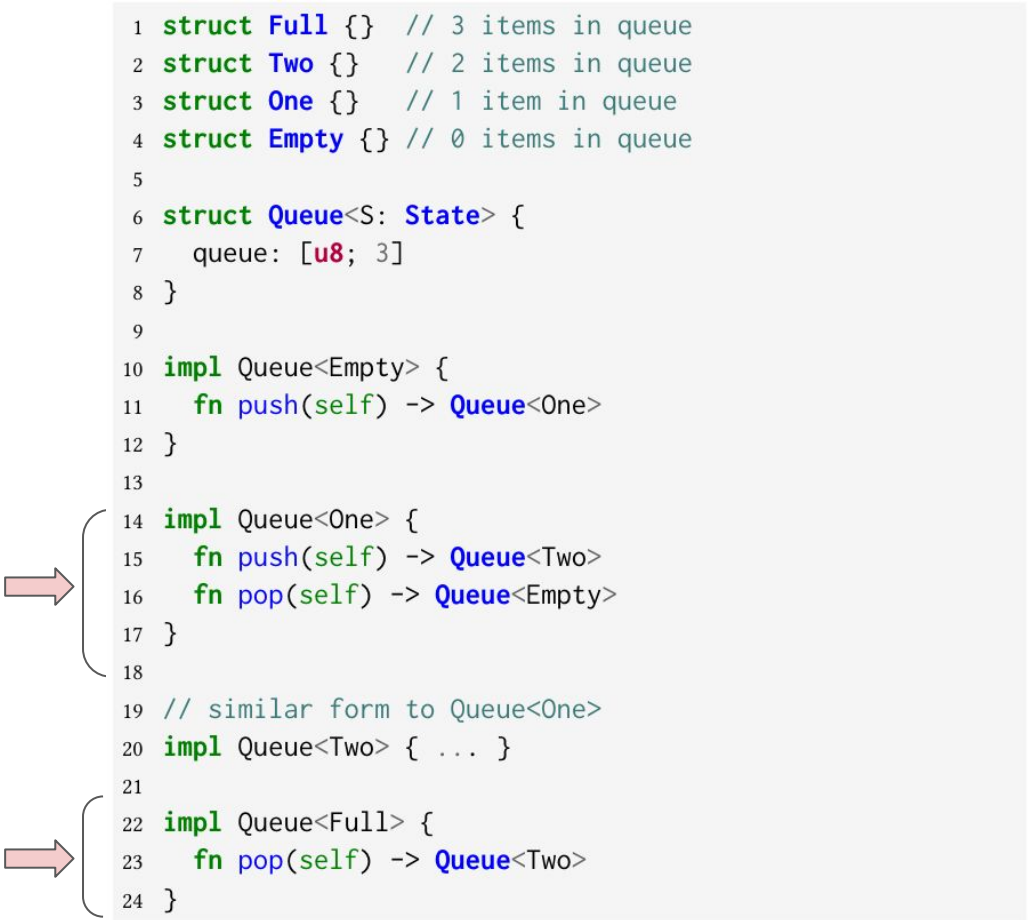


```
struct Full {}  // 3 items in queue
struct Two {}   // 2 items in queue
struct One {}   // 1 item in queue
struct Empty {} // 0 items in queue

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

(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.

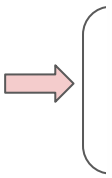


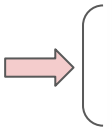
```
struct Full {}  // 3 items in queue
struct Two {}   // 2 items in queue
struct One {}   // 1 item in queue
struct Empty {} // 0 items in queue

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!

```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)

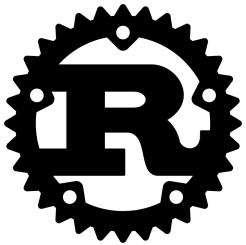
Outline

- Introducing **state safety**
- How do we build drivers today?
- TypeState programming
- **Our System**
- Evaluation & Closing Thoughts

We present a framework that statically (at compile time) guarantees hardware state safety

- Achieve state safety enforcement with minimal to no overheads in runtime and code size.

TypeStates

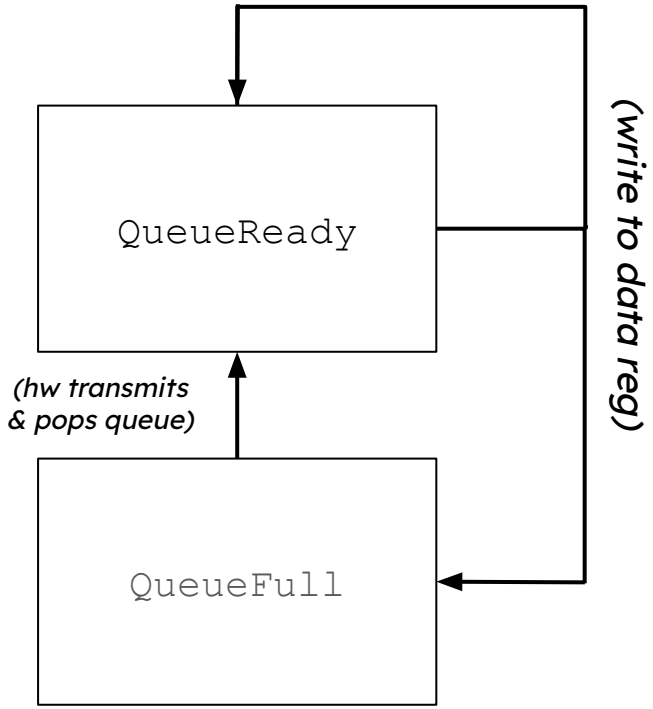


DSL

- **Primary contribution:** Introduce a principled approach to faithfully model asynchronous hardware using type-states.

How to faithfully model asynchronous HW?

We observe… there are two classes of hw state transitions

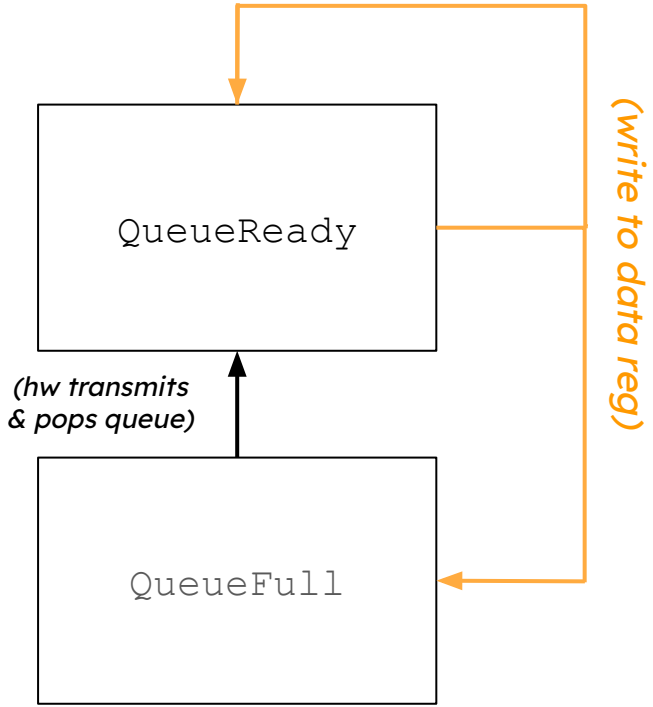


(Developer Mental Model of HW Specification)

How to faithfully model asynchronous HW?

We observe… there are two classes of hw state transitions

- Software-initiated

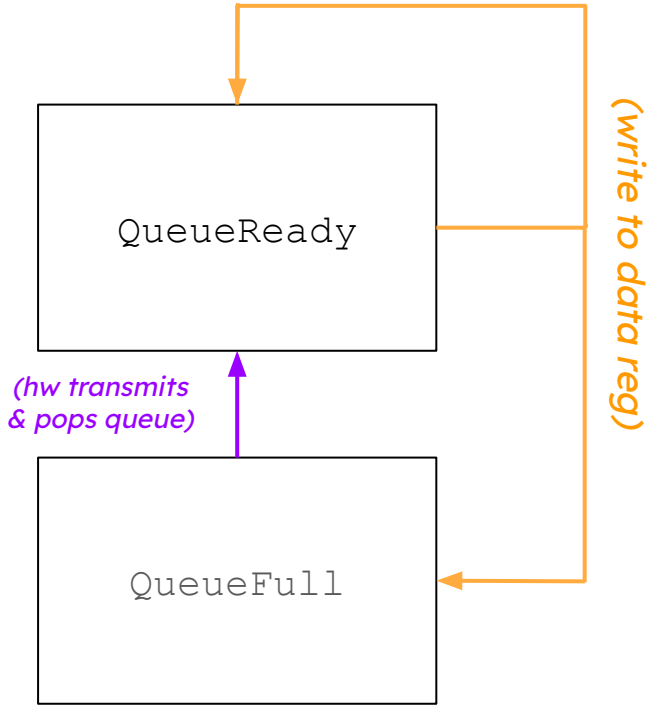


(Developer Mental Model of HW Specification)

How to faithfully model asynchronous HW?

We observe… there are two classes of hw state transitions

- Software-initiated
- Hardware-initiated



(Developer Mental Model of HW Specification)

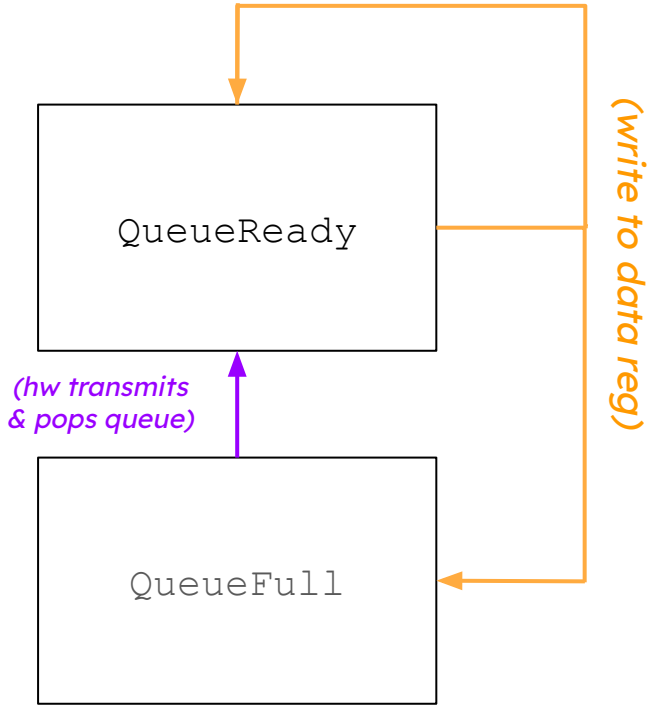
How to faithfully model asynchronous HW?

We observe… there are two classes of hw state transitions

- Software-initiated
- Hardware-initiated

Categorize hardware states into two mutually exclusive families

- transient state
- stable state

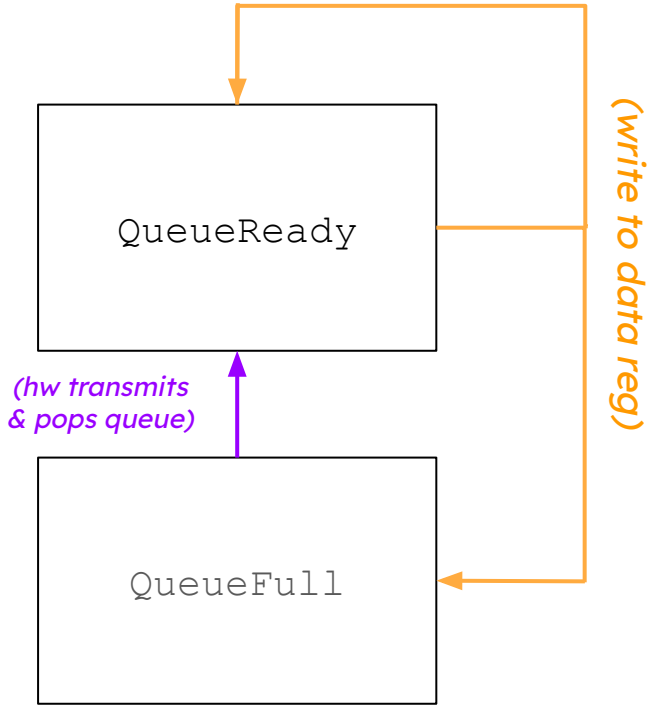


(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

Stable State

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

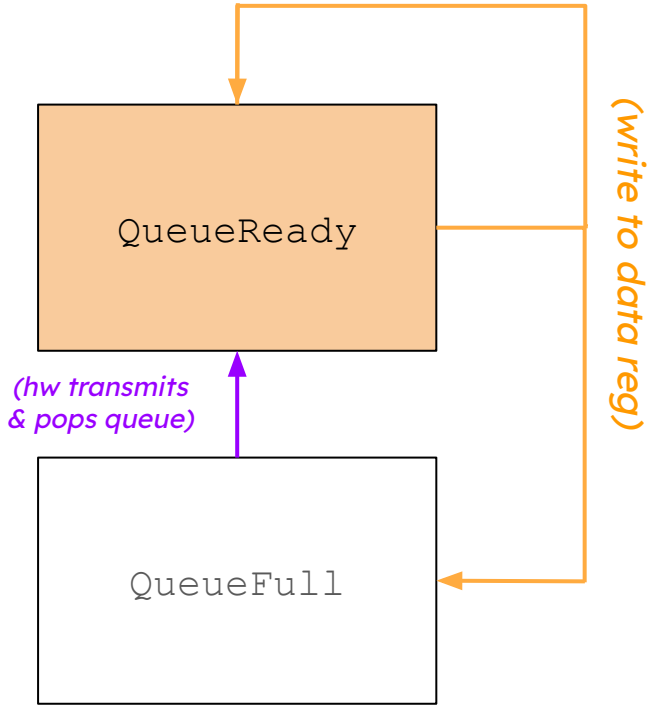


(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

Stable State

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



(Developer Mental Model of HW Specification)

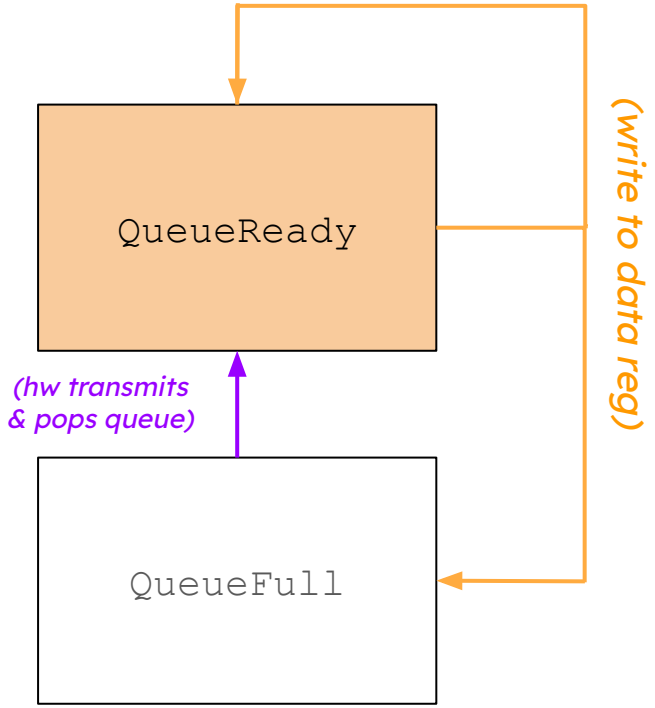
Faithfully Modelling Asynchronous HW

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)

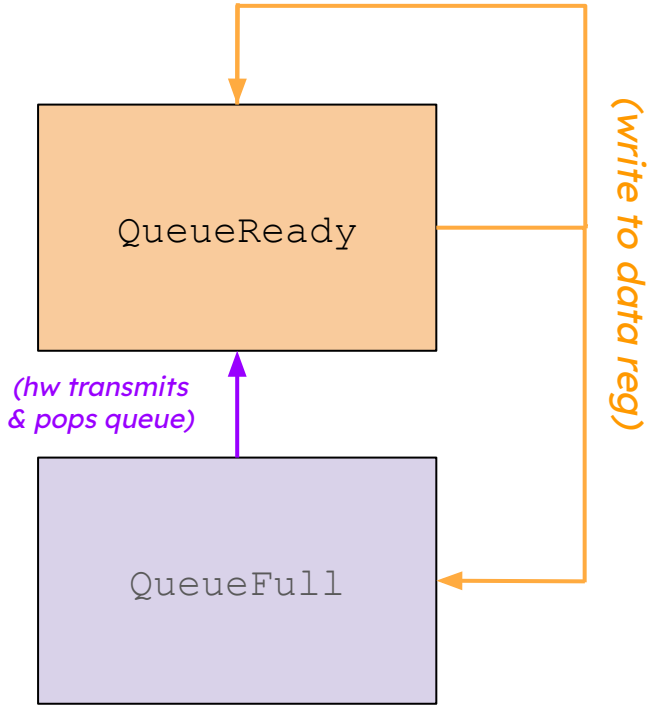
Faithfully Modelling Asynchronous HW

Stable State

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

Transient State

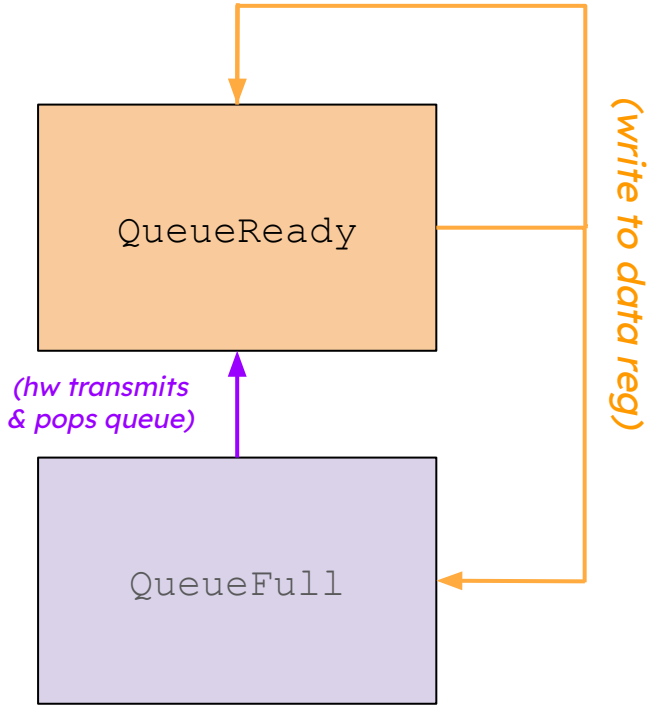
- Hw state with *at least* one hw-initiated state transition.
- Transition from transient state without explicit software involvement.



(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

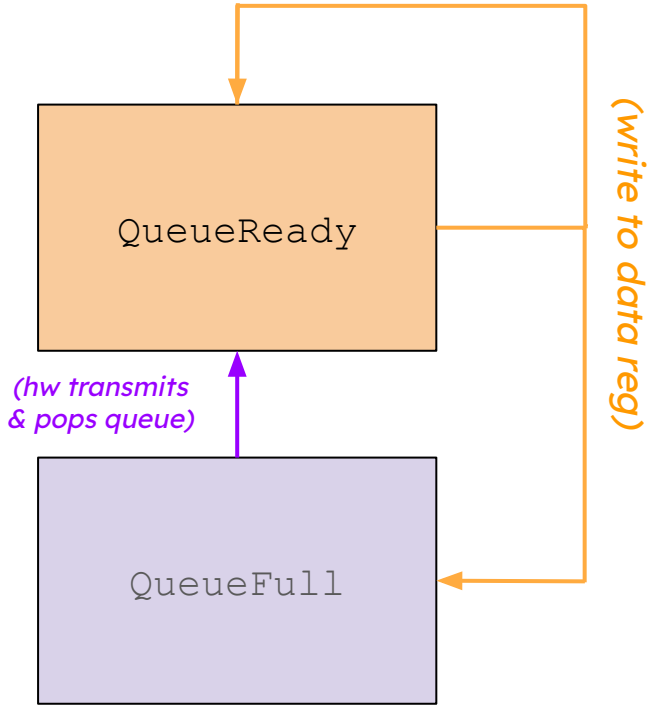
- Stable states can be modeled with out-of-the-box typestates.



(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

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

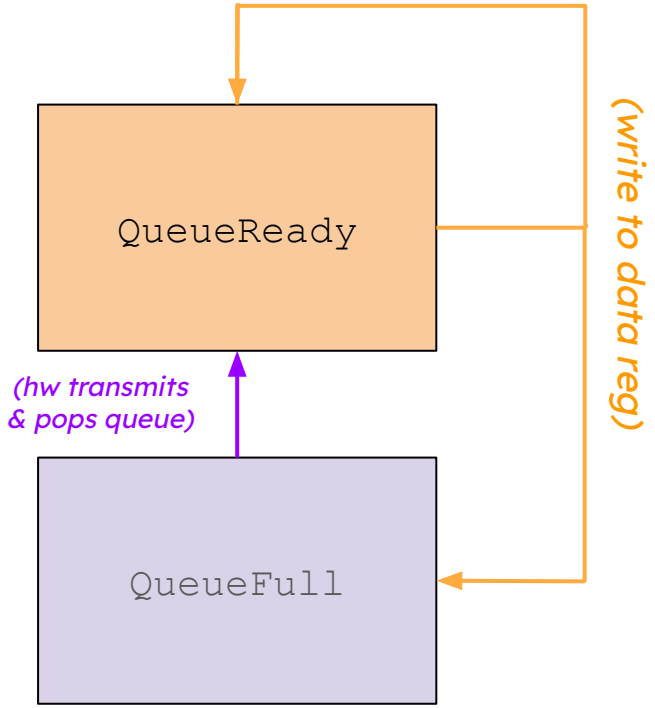


(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

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

Typestates + *re-synchronization mechanism*



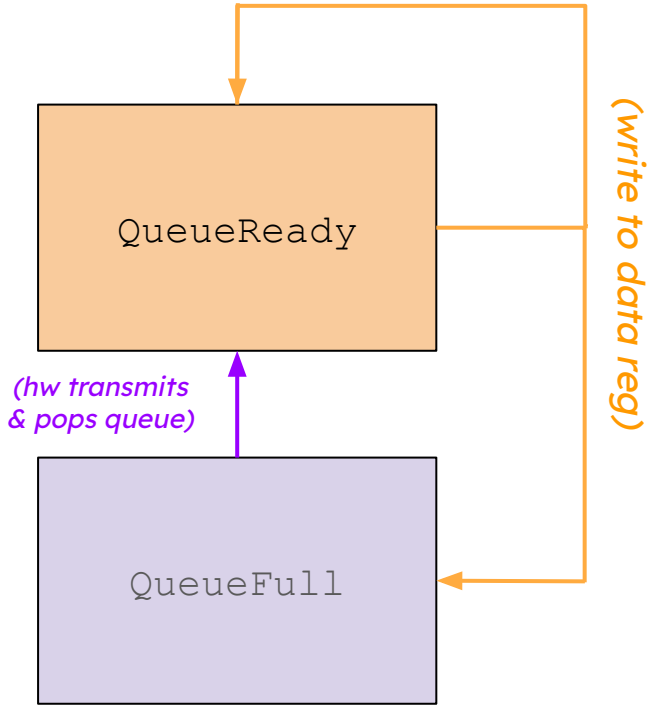
(Developer Mental Model of HW Specification)

Faithfully Modelling Asynchronous HW

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

Typestates +
re-synchronization mechanism

Careful: Transient states have potential for TOCTOU bugs!



(Developer Mental Model of HW Specification)

Framework and DSL

Annotate MMIO register struct using DSL

(transient state)

```
1  #[dsl(states=[  QueueReady<Idle>,
2                  QueueReady<Busy>(*T*),
3                  QueueMaybeFull(*T*)     ])]
4  struct UartRegisters {
5    #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
6    data: RegisterWO<u8>,
7    // No attributes are required for `Status`
8    status: RegisterRO<StatusReg>,
9    #[attribute(SCReg(Any, QueueReady<Idle>))]
10   flush: RegisterWO<u8>,
11   #[attribute(QueueReady<Idle>)]
12   config: RegisterRW<u8>,
13 }
```

Framework and DSL

Annotate MMIO register struct using DSL

(transient state)

(State transitioning operation)

```
1  #[dsl(states=[  QueueReady<Idle>,
2                  QueueReady<Busy>(*T*),
3                  QueueMaybeFull(*T*)     ])]
4  struct UartRegisters {
5    #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
6    data: RegisterWO<u8>,
7    // No attributes are required for `Status`
8    status: RegisterRO<StatusReg>,
9    #[attribute(SCReg(Any, QueueReady<Idle>))]
10   flush: RegisterWO<u8>,
11   #[attribute(QueueReady<Idle>)]
12   config: RegisterRW<u8>,
13 }
```

Framework and DSL

Annotate MMIO register struct using DSL

(transient state)

```
1  #[dsl(states=[  QueueReady<Idle>,
2                  QueueReady<Busy>(*T*),
3                  QueueMaybeFull(*T*)     ])]
4  struct UartRegisters {
5    #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
6    data: RegisterWO<u8>,
7    // No attributes are required for `Status`
8    status: RegisterRO<StatusReg>,
9    #[attribute(SCReg(Any, QueueReady<Idle>))]
10   flush: RegisterWO<u8>,
11   #[attribute(QueueReady<Idle>)]
12   config: RegisterRW<u8>,
13 }
```

(State transitioning operation)

(register only written to when in QueueReady<Idle> state)

Framework and DSL

Annotate MMIO register struct using DSL

(transient state)

```
#[dsl(states=[  QueueReady<Idle>,
                QueueReady<Busy>(*T*),
                QueueMaybeFull(*T*)     ])]
struct UartRegisters {
  #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
  data: RegisterWO<u8>,
  // No attributes are required for `Status`
  status: RegisterRO<StatusReg>,
  #[attribute(SCReg(Any, QueueReady<Idle>))]
  flush: RegisterWO<u8>,
  #[attribute(QueueReady<Idle>)]
  config: RegisterRW<u8>,
}
```

(register only written to when in QueReady<Idle> state)

Framework and DSL

Annotate MMIO register struct using DSL

Generate needed typestates and state transitions

(transient state)

```
#[dsl(states=[  QueueReady<Idle>,
                QueueReady<Busy>(*T*),
                QueueMaybeFull(*T*)     ])]
struct UartRegisters {
  #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
  data: RegisterWO<u8>,
  // No attributes are required for `Status`
  status: RegisterRO<StatusReg>,
  #[attribute(SCReg(Any, QueueReady<Idle>))]
  flush: RegisterWO<u8>,
  #[attribute(QueueReady<Idle>)]
  config: RegisterRW<u8>,
}
```

(register only written to when in QueReady<Idle> state)

Framework and DSL

Generate needed typestates and state transitions

Typestates have repetitive code patterns

Our DSL generates 1000s of lines of code per driver

TypeStated wrapper types for registers

State changes only defined for valid hw states

```rust
// Auto generated from macro //
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: RegisterRO<StatusReg>,
  flush: SCRegisterWO<S, u8>,
  config: RegisterRW<N, S, u8>,
}

// Wrapper around State Changing Registers
struct SCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}

// Wrapper around constrained MMIO.
struct RegisterRW<N, S: State, T> {
  reg: RegisterRW<T>,
  associated_name: PhantomData<N>,
  associated_state: PhantomData<S>,
}

impl <T> RegisterRW<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}

impl <S: State> UartRegisters<S>
{
  fn queue_flush(self) -> UartRegisters<QueueReady<Idle>>
}

impl <S: State> UartRegisters<QueueReady<S>> {
  fn write_data(self, data: u8)
    -> UartRegisters<QueueMaybeFull>;
}
```

Framework and DSL

Annotate MMIO register struct using DSL

Generate needed typestates and state transitions

(transient state)

```
#[abacus(states=[  QueueReady<Idle>,
                   QueueReady<Busy>(*T*),
                   QueueMaybeFull(*T*)     ])]
struct UartRegisters {
  #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
  data: RegisterWO<u8>,
  // No attributes are required for `Status`
  status: RegisterRO<StatusReg>,
  #[attribute(SCReg(Any, QueueReady<Idle>))]
  flush: RegisterWO<u8>,
  #[attribute(QueueReady<Idle>)]
  config: RegisterRW<u8>,
}
```

(register only written to when in QueReady<Idle> state)

```
// Auto generated from macro //
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: RegisterRO<StatusReg>,
  flush: SCRegisterWO<S, u8>,
  config: RegisterRW<N, S, u8>,
}

// Wrapper around State Changing Registers
struct SCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}

// Wrapper around constrained MMIO.
struct RegisterRW<N, S: State, T> {
  reg: RegisterRW<T>,
  associated_name: PhantomData<N>,
  associated_state: PhantomData<S>,
}

impl <T> RegisterRW<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}

impl <S: State> UartRegisters<S>
{
  fn queue_flush(self) -> UartRegisters<QueueReady<Idle>>
}

impl <S: State> UartRegisters<QueueReady<S>> {
  fn write_data(self, data: u8)
    -> UartRegisters<QueueMaybeFull>;
}
```

TypeStated wrapper types for registers

State changes only defined for valid hw states

Framework and DSL

Annotate MMIO register struct using DSL

Generate needed typestates and state transitions

Implement driver using typestates

(transient state)

```
#[abacus(states=[  QueueReady<Idle>,
                   QueueReady<Busy>(*T*),
                   QueueMaybeFull(*T*)     ])]
struct UartRegisters {
  #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
  data: RegisterWO<u8>,
  // No attributes are required for `Status`
  status: RegisterRO<StatusReg>,
  #[attribute(SCReg(Any, QueueReady<Idle>))]
  flush: RegisterWO<u8>,
  #[attribute(QueueReady<Idle>)]
  config: RegisterRW<u8>,
}
```

(register only written to when in QueReady<Idle> state)

```
// Auto generated from macro //
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: RegisterRO<StatusReg>,
  flush: SCRegisterWO<S, u8>,
  config: RegisterRW<N, S, u8>,
}

// Wrapper around State Changing Registers
struct SCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}

// Wrapper around constrained MMIO.
struct RegisterRW<N, S: State, T> {
  reg: RegisterRW<T>,
  associated_name: PhantomData<N>,
  associated_state: PhantomData<S>,
}

impl <T> RegisterRW<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}

impl <S: State> UartRegisters<S>
{
  fn queue_flush(self) -> UartRegisters<QueueReady<Idle>>
}

impl <S: State> UartRegisters<QueueReady<S>> {
  fn write_data(self, data: u8)
    -> UartRegisters<QueueMaybeFull>;
}
```

TypeStated wrapper types for registers

State changes only defined for valid hw states

Framework and DSL

Our system statically guarantees state safety!

Annotate MMIO register struct using DSL

Generate needed typestates and state transitions

Implement driver using typestates

(transient state)

```
#[abacus(states=[  QueueReady<Idle>,
                   QueueReady<Busy>(*T*),
                   QueueMaybeFull(*T*)     ])]
struct UartRegisters {
  #[attribute(SCReg(QueueReady<Any>, QueueMaybeFull))]
  data: RegisterWO<u8>,
  // No attributes are required for `Status`
  status: RegisterRO<StatusReg>,
  #[attribute(SCReg(Any, QueueReady<Idle>))]
  flush: RegisterWO<u8>,
  #[attribute(QueueReady<Idle>)]
  config: RegisterRW<u8>,
}
```

(register only written to when in QueReady<Idle> state)

```
// Auto generated from macro //
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: RegisterRO<StatusReg>,
  flush: SCRegisterWO<S, u8>,
  config: RegisterRW<N, S, u8>,
}

// Wrapper around State Changing Registers
struct SCRegisterWO<S: State, T> {
  reg: RegisterWO<T>,
  associated_state: PhantomData<S>,
}

// Wrapper around constrained MMIO.
struct RegisterRW<N, S: State, T> {
  reg: RegisterRW<T>,
  associated_name: PhantomData<N>,
  associated_state: PhantomData<S>,
}

impl <T> RegisterRW<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}

impl <S: State> UartRegisters<S>
{
  fn queue_flush(self) -> UartRegisters<QueueReady<Idle>>
}

impl <S: State> UartRegisters<QueueReady<S>> {
  fn write_data(self, data: u8)
    -> UartRegisters<QueueMaybeFull>;
}
```

TypeStated wrapper types for registers

State changes only defined for valid hw states

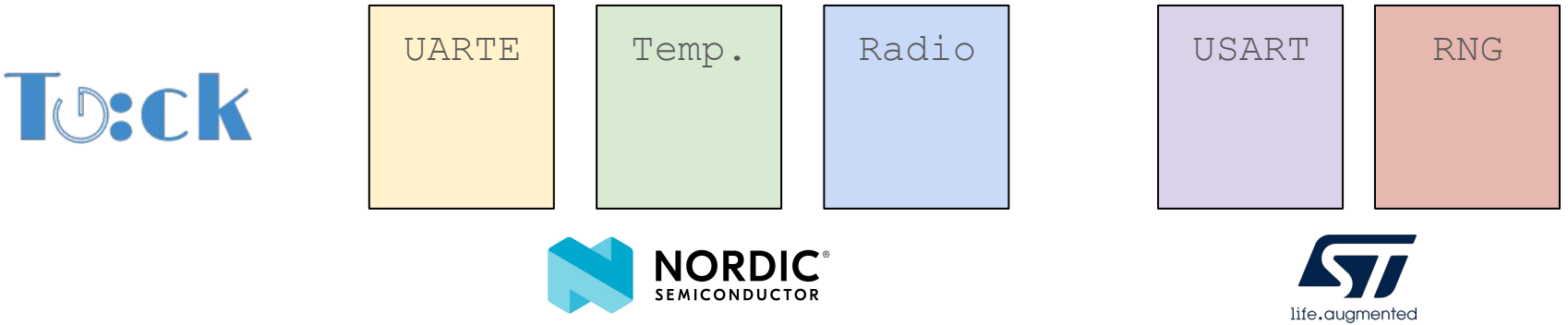
Outline

- Introducing **state safety**
- How do we build drivers today?
- TypeState programming
- Our System
- **Evaluation & Closing Thoughts**

Implementation with TockOS

TockOS is…

- an embedded operating system written in Rust.
- the firmware for OpenTitan root-of-trust.
- used by Google and Microsoft for security modules in millions of devices.
- used in safety critical applications.



Evaluation - Code Size

Our system adds no code size overhead.

- The DSL procedural macros generate 1000s of lines of code for each driver.
- By using Rust zero sized types, our typestates are elided by the compiler.

Driver	Platform	Binary Size (B)	Diff (B)	Percent Diff
Baseline	Nrf52840	218594	–	–
UART	Nrf52840	218594	+0	0.00%
Temperature Sensor	Nrf52840	218594	+0	0.00%
IEEE 802.15.4 Radio	Nrf52840	218602	+8	0.00%
Baseline	STM	107482	–	–
TRNG	STM	107490	+8	0.00%
UART	STM	107490	+8	0.00%

Table 1. Code size of total kernel binary image for a baseline kernel image and kernel integrating our system into drivers.

Microbenchmarks

Our system adds minor runtime overhead.

- Cases in which our system outperforms are due to incidental reimplementation improvements.

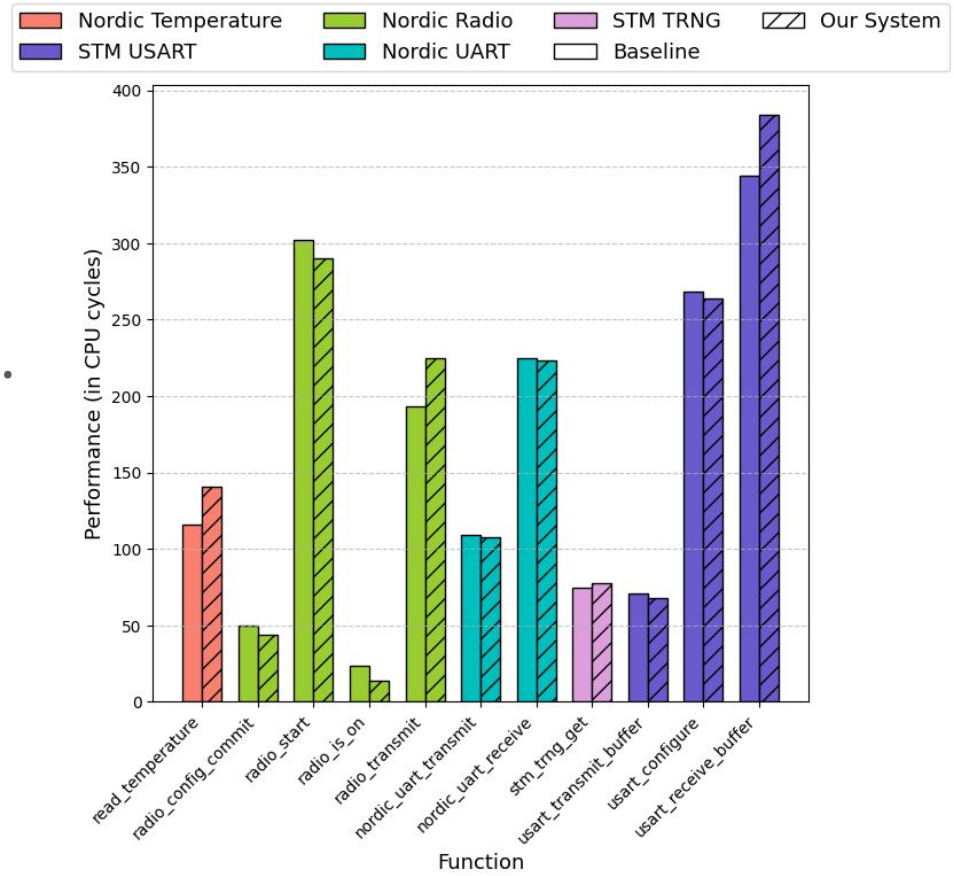
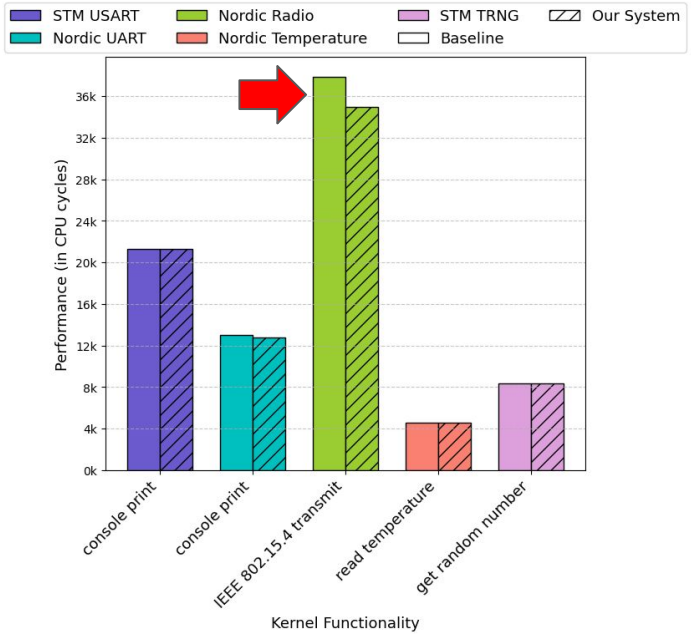


Figure 1. MicroBenchmarks of TockOS HAL method performance for integrated drivers and the baseline.

Macrobenchmarks

How minor is this “minor overhead”?

- Negligible overheads from driver’s usage of typestates within context of kernel’s operation.

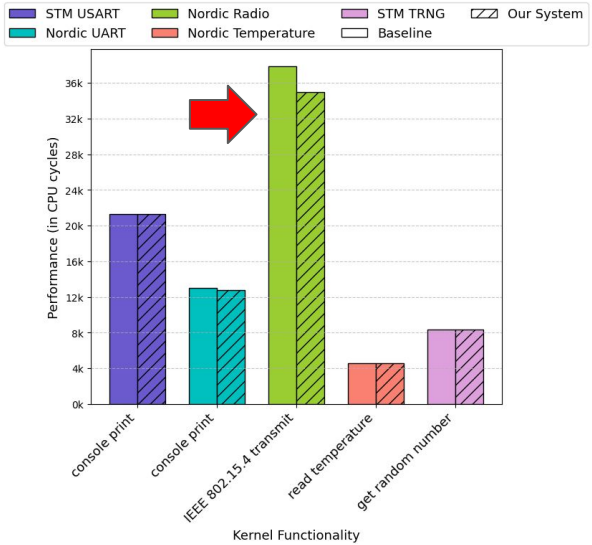


Functionality	Driver	Platform	Percent Difference
Console Print	UARTE	Nrf52840	**-1.67%**
Read Temperature	Temperature Sensor	Nrf52840	**+1.23%**
Transmit Packet	IEEE 802.15.4 Radio	Nrf52840	**-7.76%**
Console Print	USART	STM	**+0.14%**
Get Random Number	TRNG	STM	**+0.13%**

Figure 3. MacroBenchmark Performance (in CPU cycles) demonstrating our system/baseline driver runtime when performing kernel functionality that is underpinned by the device driver.

Case Study – NRF52 IEEE802.15.4 Driver

- NRF52 15.4 Radio provides hardware performance features that are challenging to use correctly (highly asynchronous).
- Easily add these performance features (require 1 hour) to the radio driver using our system.
- Yield 7.76% performance improvement.



Our system enables the usage of asynchronous hw features that may otherwise be prohibitively complex.

Conclusion

Our system statically enforces state safety and can faithfully model asynchronous hardware using typestates.

Conclusion

Our system statically enforces state safety and can faithfully model asynchronous hardware using typestates.

Imposes minimal to no code size and runtime overheads.

Conclusion

Our system statically enforces state safety and can faithfully model asynchronous hardware using typestates.

Imposes minimal to no code size and runtime overheads.

Future work.

- Async typestates for other applications beyond device drivers?
- Ability to track hardware state => track power usage in software.

Framework and DSL

Implement driver using typestates

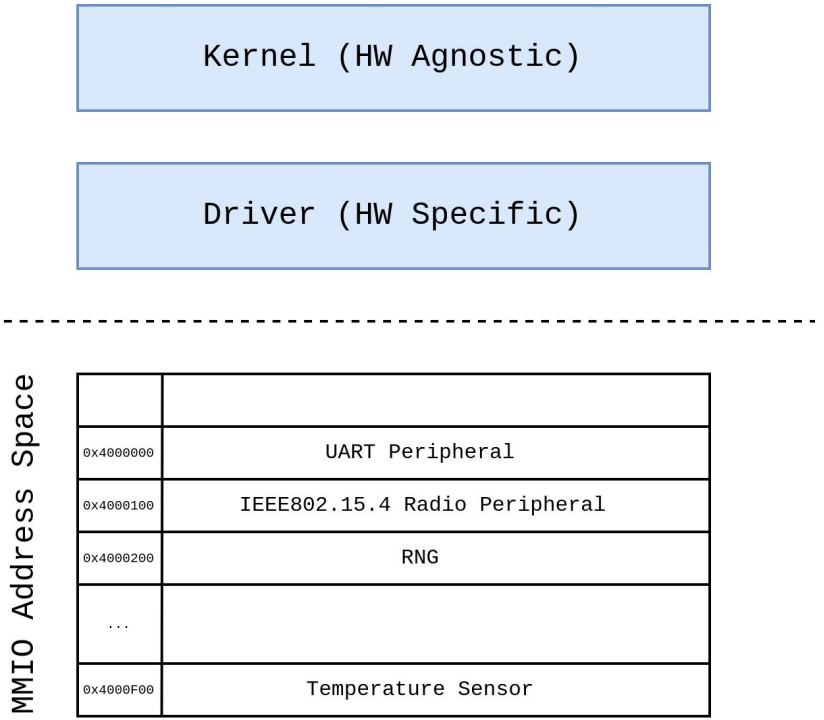
```
fn transmit(mut reg: QueueFeasibleStates, buf: &[u8])
-> QueueFeasibleStates {
  for &byte in buf.iter() {
    reg = loop {
      match reg {
        QueueFeasibleStates::QueueReadyIdle(r) => {
          break r.write_data(byte).sync_state(),
        }
        QueueFeasibleStates::QueueReadyBusy(r) => {
          break r.write_data(byte).sync_state(),
        }
        QueueFeasibleStates::MaybeFull(r) => {
         reg = r.sync_state(),
        }
  }}}
  reg
}
```

**Divergent State Synchronization Mechanism*

Only state safe operations defined -> typestates statically guarantee state safety!

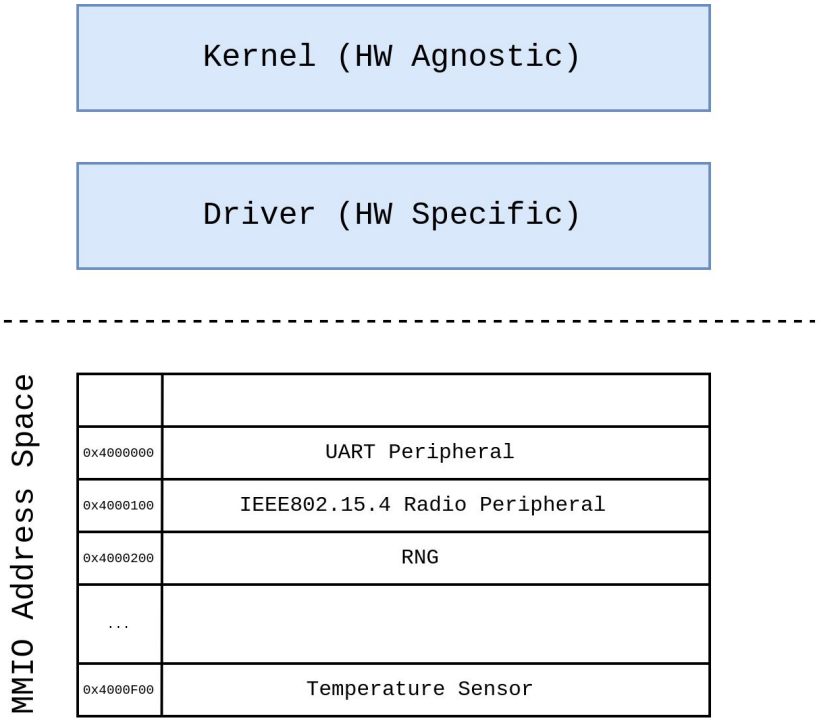
Framework and DSL

- *CHALLENGE: How can typestates be properly passed to the driver's execution context?*
- Solution (1): Driver specific typestates passed to the hw agnostic kernel.
- Solution (2): Driver stores/fetches the typestate when crossing the kernel/driver boundary.



Framework and DSL

- *CHALLENGE: How can typestates be properly passed to the driver's execution context?*
- Solution (1): Driver specific typestates passed to the hw agnostic kernel.
- Solution (2): Driver stores/fetches the typestate when crossing the kernel/driver boundary.



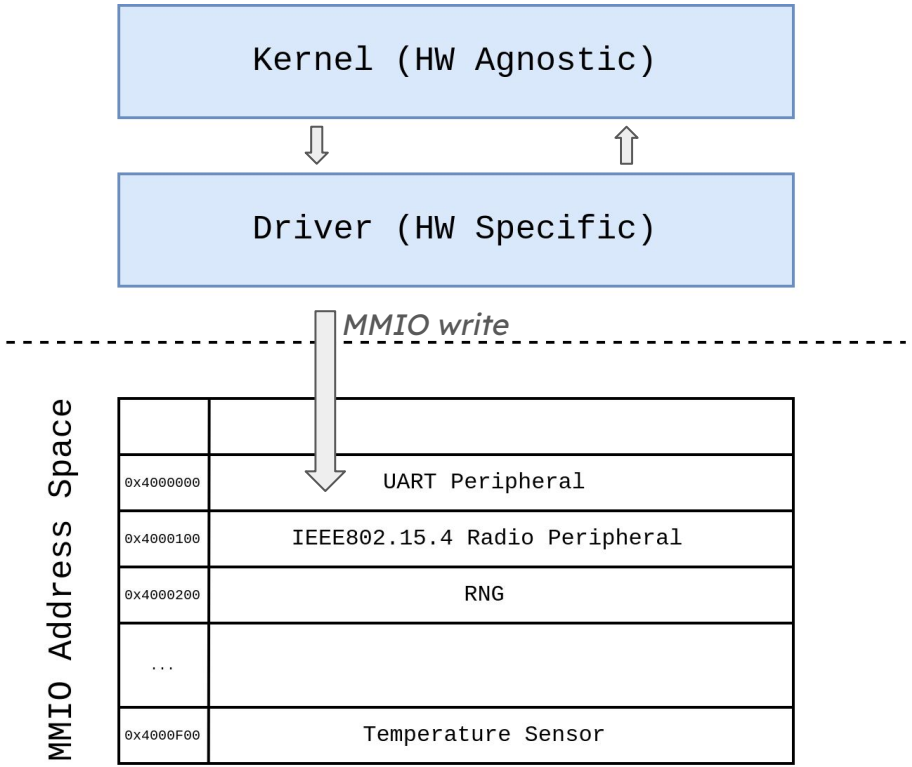
We evaluate that for ARM Cortex-M4, this state store/fetch causes a 21 cycle overhead.

Implementation with TockOS - Procedural Macros

- Procedural Macros are a powerful Rust macro/code generation library.
- We use Rust Procedural Macros to parse the framework DSL and generate the 1000s of lines needed for representing state safety as typestates.
- Our proc-macro generates Tock specific bindings that integrate with Tock's abstraction for MMIO registers.

Low-level Software Drivers

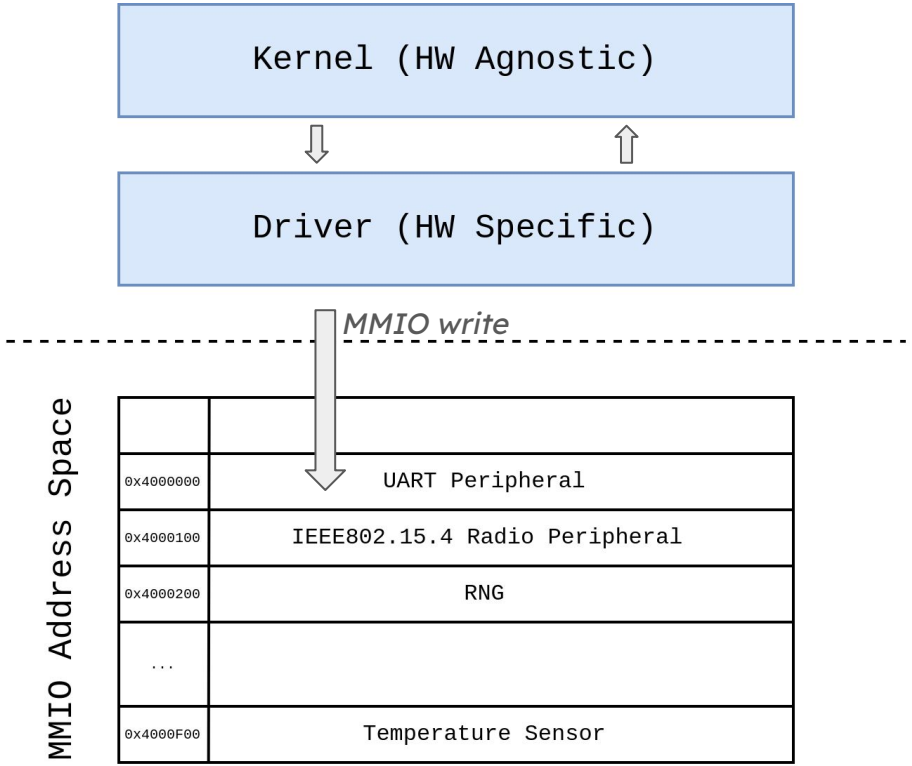
- Low-level software bridges the kernel and hardware.
- MMIO is permissive in what a driver *CAN* do.
- Hardware is opinionated about what a driver *SHOULD* do.



Low-level Software Drivers

- Low-level software bridges the kernel and hardware.
- MMIO is permissive in what a driver *CAN* do.
- Hardware is opinionated about what a driver *SHOULD* do.

An interface requiring <u>many</u> invariants to be upheld while enforcing <u>minimal restrictions</u> has a high likelihood of bugs!



State Safety

We introduce state safety to express when software adheres to hardware's specification, and only performs MMIO operations valid for the given state.

- Authoring state safe driver is dependent on the hardware's dynamic state.
- Modern hardware is increasingly capable and may transition states without input from software.

State Safety

We introduce state safety to express when software adheres to hardware's specification, and only performs MMIO operations valid for the given state.

- Authoring state safe driver is dependent on the hardware's dynamic state.
- Modern hardware is increasingly capable and may transition states without input from software.

- Violating state safety… 
  - may compromise the individual driver's correctness.
  - at worst, may result in systematic failures.