Just because you can doesn't mean you should…

TypeStates for Increased Driver Correctness

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

Hello!

- PhD Student at UC San Diego
- Involved with Tock since 2023 (~2.5 years)
- My research centers around making systems secure-by-default

(Last weekend in the Sierra's!)

Rust will fix our problems!

Buffer overflows

Use-After-Free

Data Races

Uninitialized Accesses

Rust will fix our problems!

~~Buffer overflows~~

~~Use-After-Free~~

Device Protocol Bugs

FFI Bugs

~~Data Races~~

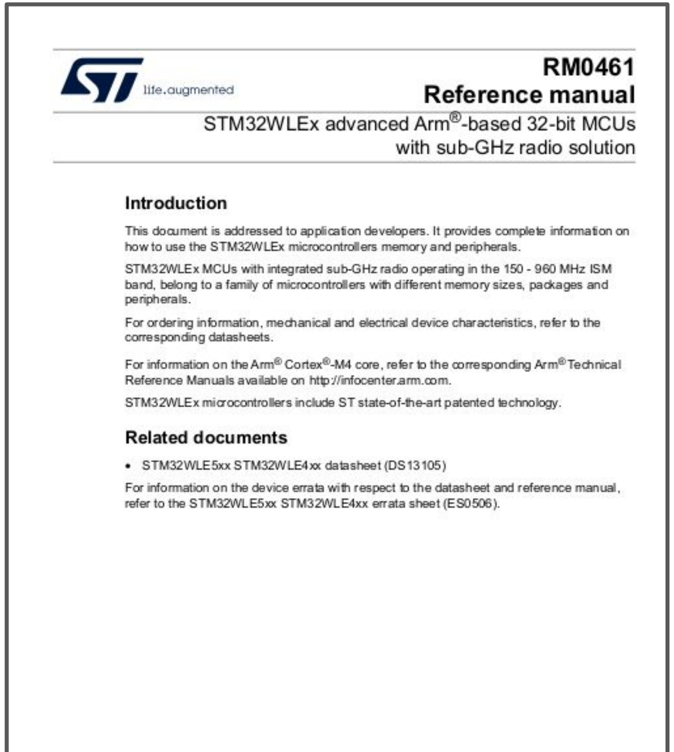
~~Uninitialized Accesses~~

What is a device protocol violation?

When software issues commands to hardware that violate the hardware specification

Low-level Driver Development

Specification / reference manual



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).

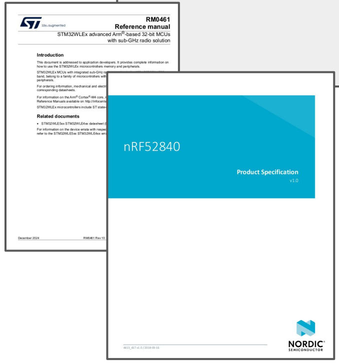
nRF52840

Product Specification
v1.0

4413_417 v1.0 / 2018-03-16

Low-level Driver Development

Specification /
reference manual


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 ra
band, belong to a family of microcontrollers with
peripherals.

For ordering information, mechanical and electr
corresponding datasheets.

For information on the Arm® Cortex®-M4 core,
Reference Manuals available on http://infocente

STM32WLEx microcontrollers include ST state-

Related documents

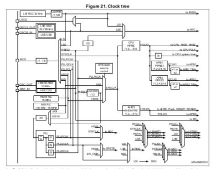
- STM32WLE5xx STM32WLE4xx datasheet (

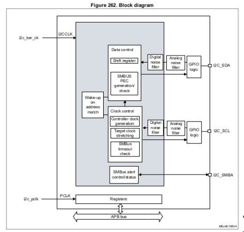
For information on the device errata with respec
refer to the STM32WLE5xx STM32WLE4xx err

December 2024 RM0461 Rev 10

nRF52840

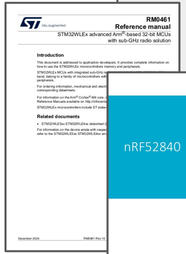
Product Specification
v1.0





Low-level Driver Development

Specification / reference manual


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 ra

band, belong to a family of microcontrollers with

peripherals.

For ordering information, mechanical and electr

corresponding datasheets.

For information on the Arm® Cortex®-M4 core,

Reference Manuals available on http://infocente

STM32WLEx microcontrollers include ST state-

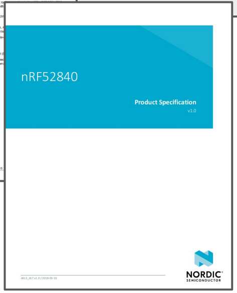
Related documents

- STM32WLE5xx STM32WLE4xx datasheet (

For information on the device errata with respec

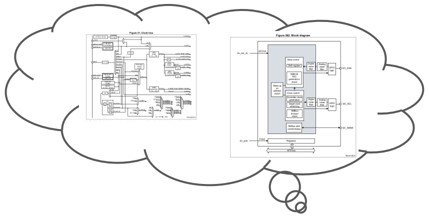
refer to the STM32WLE5xx STM32WLE4xx erra

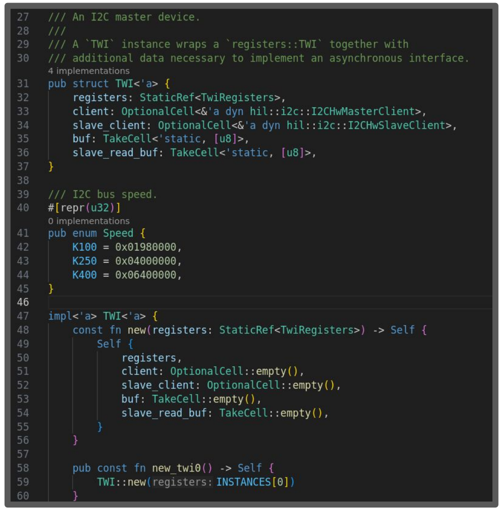
December 2024 RM0461 Rev 10


nRF52840

Product Specification

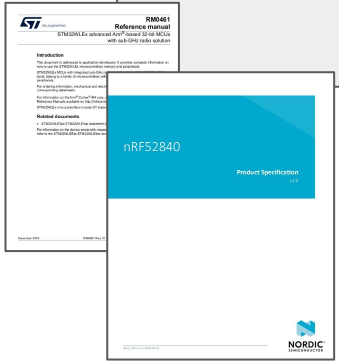
v1.0




```
/// 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

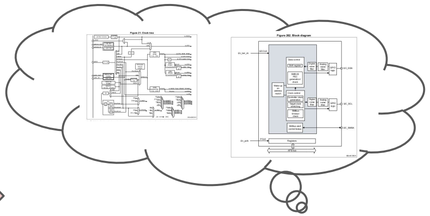
Specification / reference manual


RM0461
Reference manual
STM32WLEx advanced Arm®-based 32-bit MCUs with sub-GHz radio solution

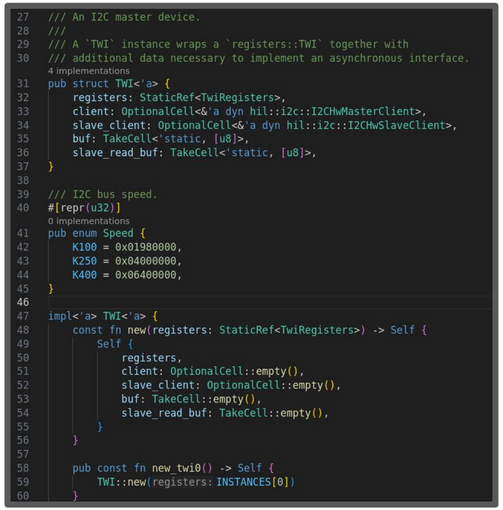
Introduction

Related documents

nRF52840
Product Specification
v1.0



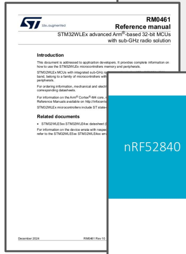
What can go wrong here? :)



```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

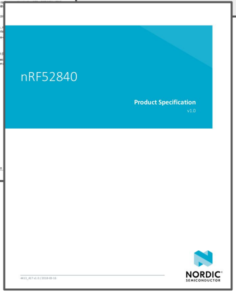

RM0461
Reference manual
STM32WLEx advanced Arm®-based 32-bit MCUs with sub-GHz radio solution

Introduction

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

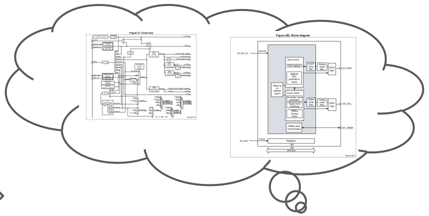
Related documents

December 2024 RM0461 Rev 10


nRF52840

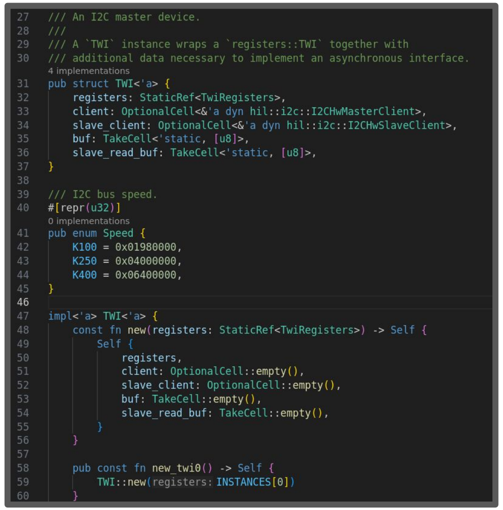
Product Specification
v1.0

4413_417 v1.0 / 2018-03-16



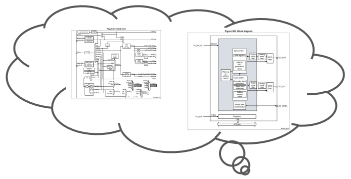
What can go wrong here? :)

Challenging!

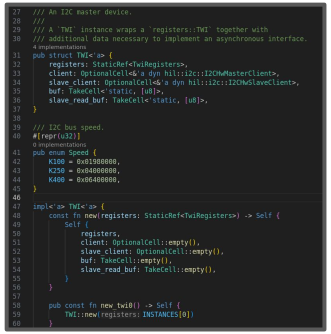


```
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

Why is this challenging?

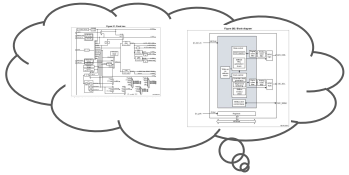


Challenging!

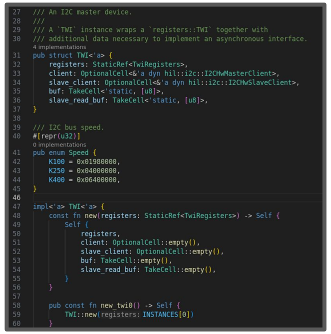


```rust
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

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



Challenging!



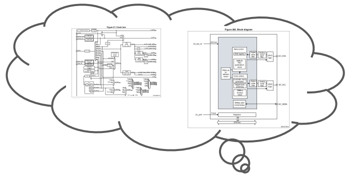
```
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

Why is this challenging?

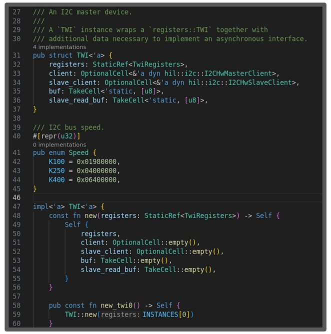
Validity of a given MMIO operation depends on the current hardware state.

➕

Modern hardware may transition the hardware state without input from the driver.



Challenging!



```
/// 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])
    }
```

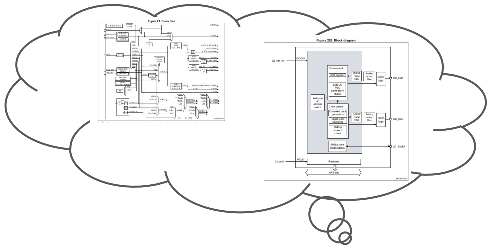
Why is this challenging?

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

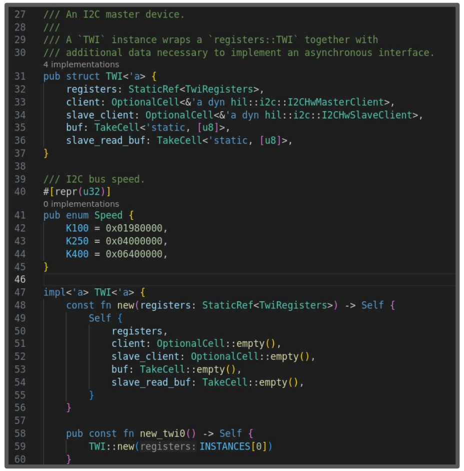
➕

Modern hardware may transition the hardware state without input from the driver.

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

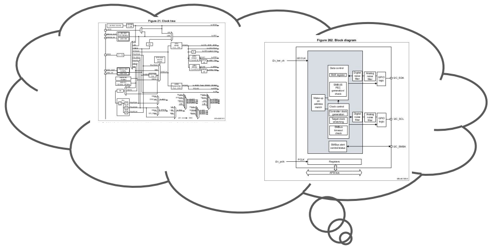


Challenging!

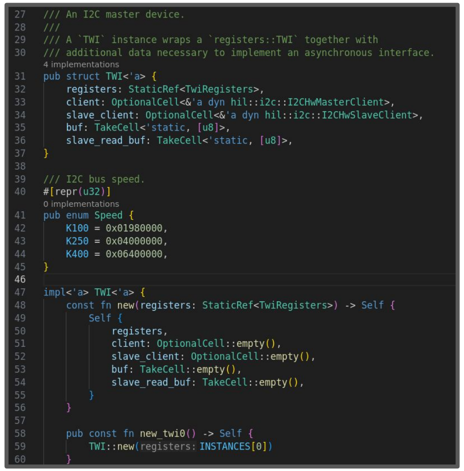


```
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,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

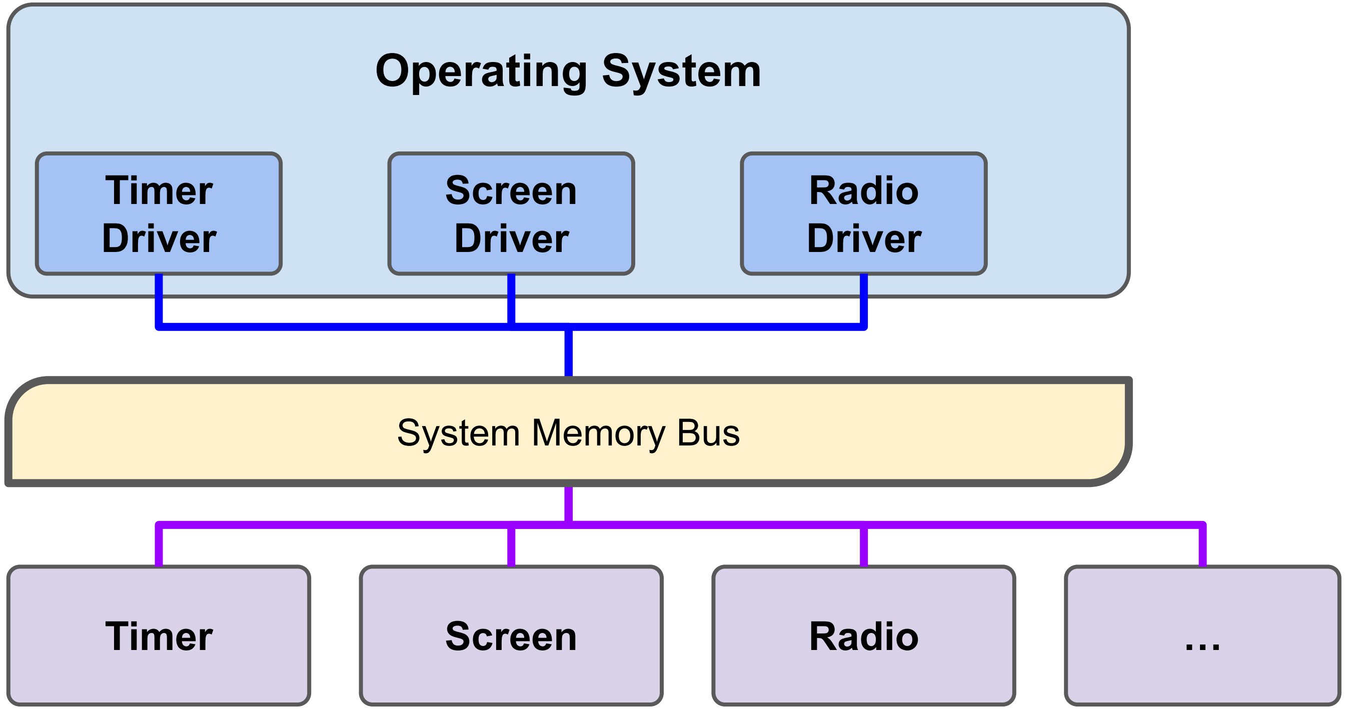


```
/// An I2C master device.
///
/// A `TWI` instance wraps a `registers::TWI` together with
/// additional data necessary to implement an asynchronous interface.
4 implementations
pub struct TWI<'a> {
    registers: StaticRef<TwiRegisters>,
    client: OptionalCell<&'a dyn hil::i2c::I2CHwMasterClient>,
    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, at compile time, that the implemented driver will always comply with the developer’s hw mental model?

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) prevents device protocol violations

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

Statically eliminate device protocol violations with minimal-to-no overheads in runtime and code size.

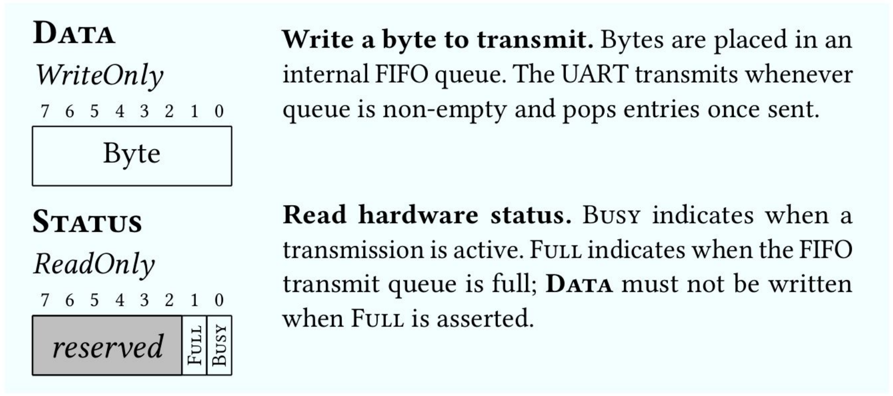
TypeStates

DSL

Outline

- Introducing device protocol violations
- **How do we build drivers today?**
- TypeState programming
- Our System
- Evaluation & Closing Thoughts

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

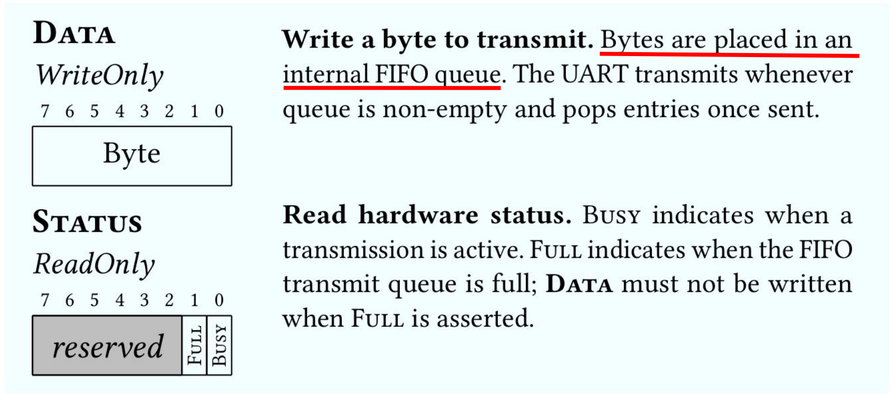
7 6 5 4 3 2 1 0

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)

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.

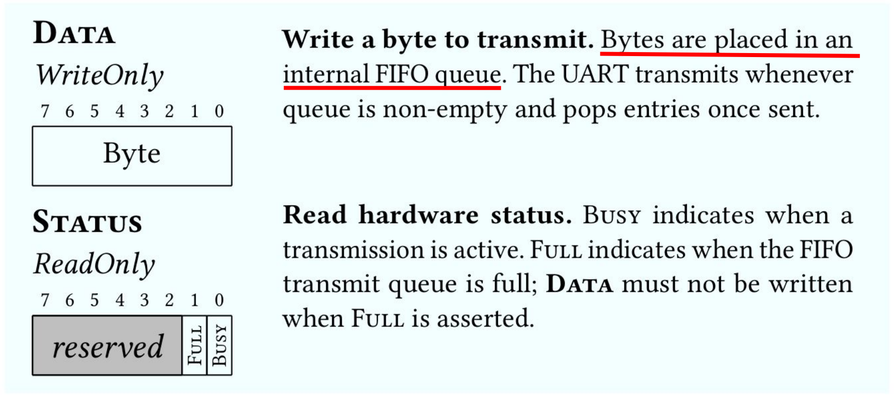
Status
ReadOnly

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)

Let's build a UART driver…



DATA
WriteOnly

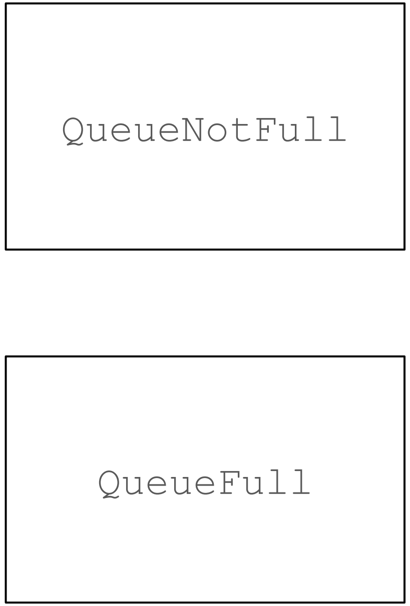
7 6 5 4 3 2 1 0
Byte

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

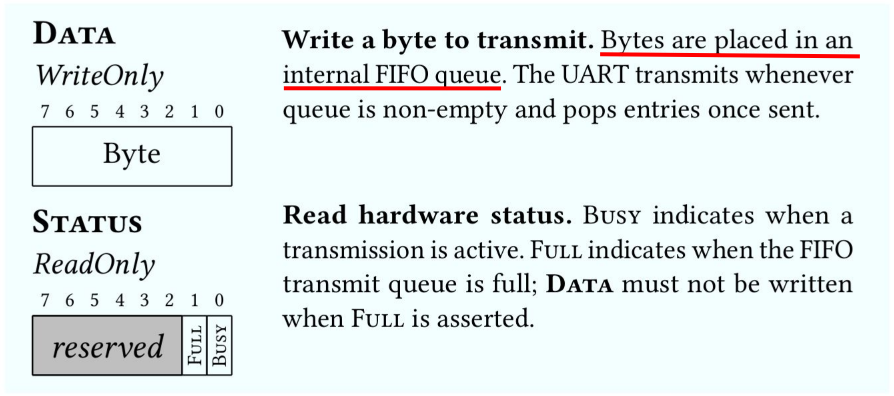
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.



(Hypothetical UART Hardware Specification)

(Developer Mental Model of HW Specification)

Let’s build a UART driver…



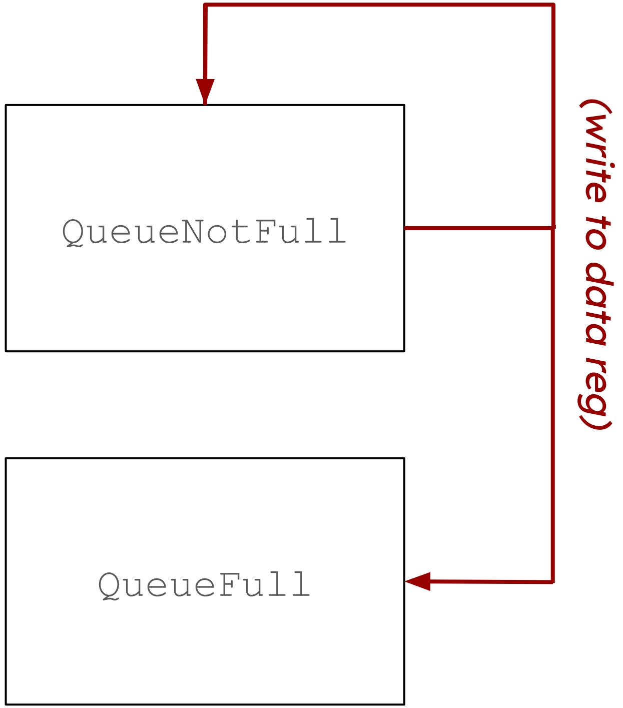
DATA
WriteOnly
7 6 5 4 3 2 1 0
Byte

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

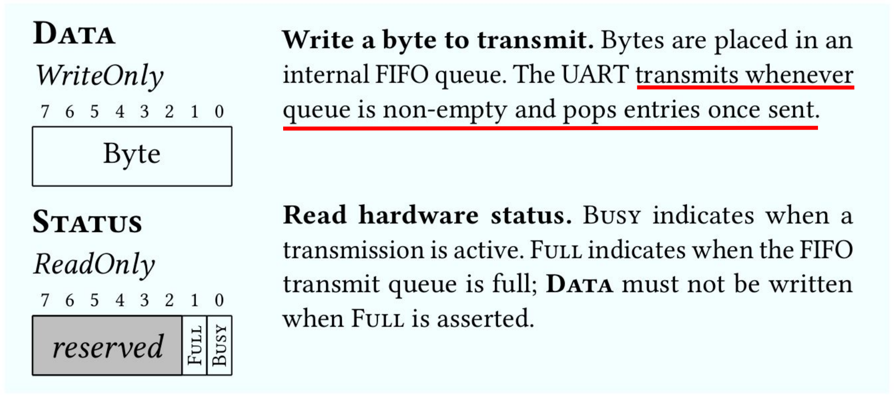
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

(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							

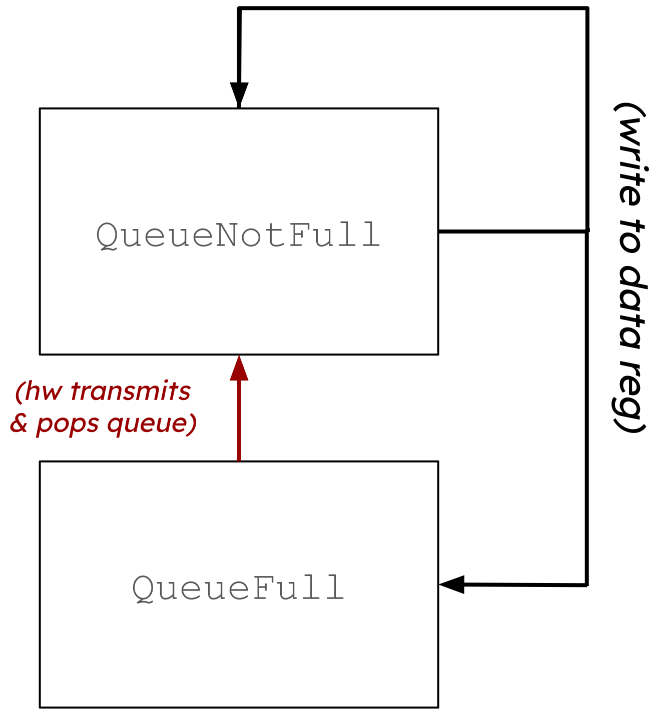
Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

Status
ReadOnly

7	6	5	4	3	2	1	0
reserved						Full	Busy

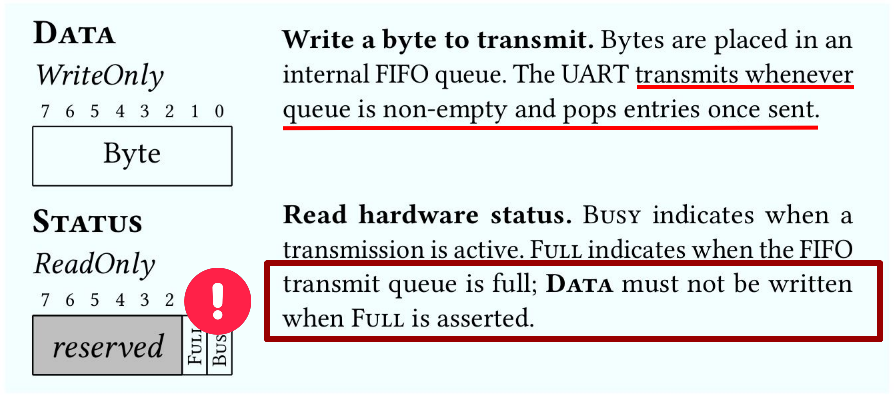
Read hardware status. Busy indicates when a transmission is active. Full indicates when the FIFO transmit queue is full; **Data** must not be written when Full is asserted.

(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							

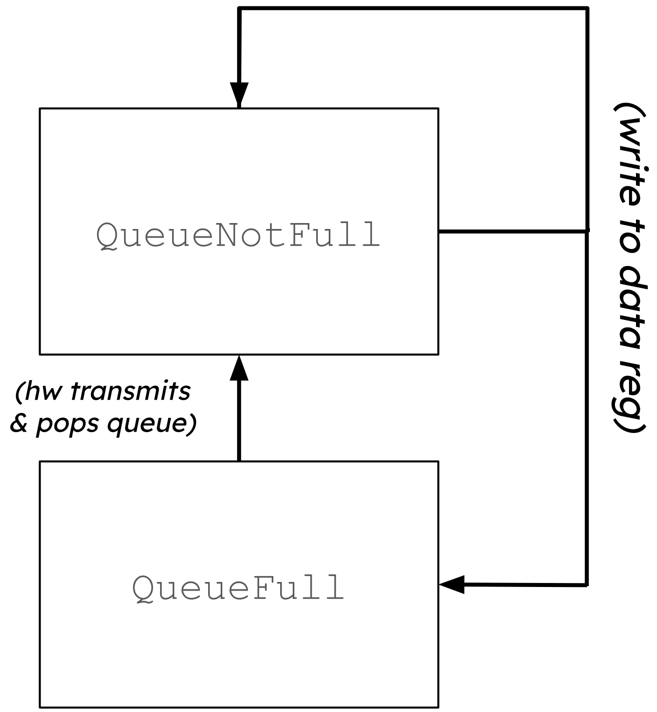
Write a byte to transmit. Bytes are placed in an internal FIFO queue. The UART transmits whenever queue is non-empty and pops entries once sent.

Status
ReadOnly

7	6	5	4	3	2	1	0
reserved						Full	Busy

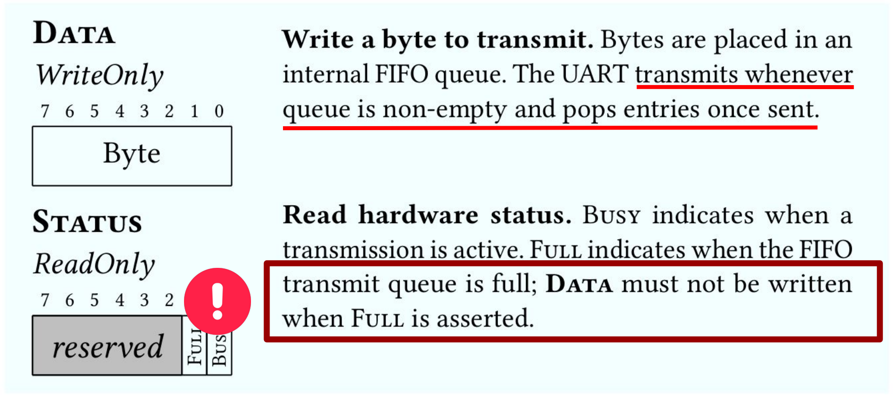
Read hardware status. Busy indicates when a transmission is active. Full indicates when the FIFO transmit queue is full; **Data** must not be written when Full is asserted.

(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

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.

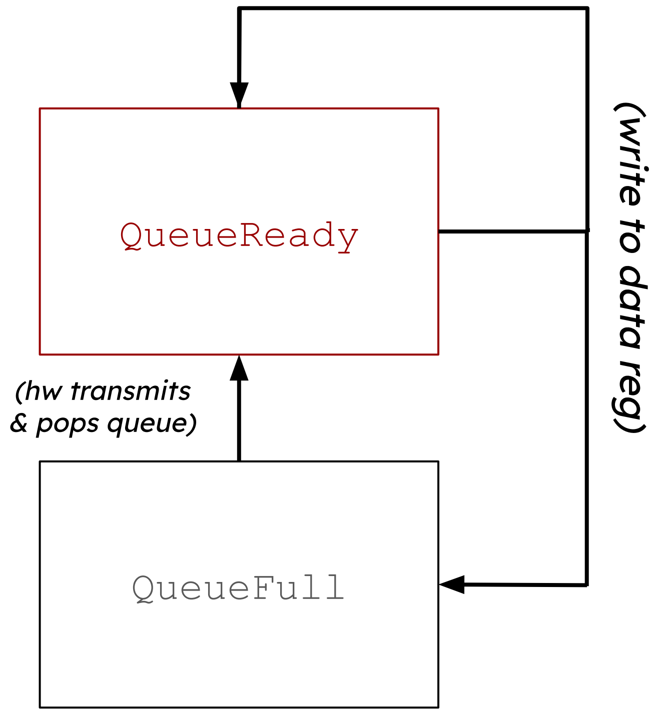
STATUS
ReadOnly

7 6 5 4 3 2

reserved FULL BUSY

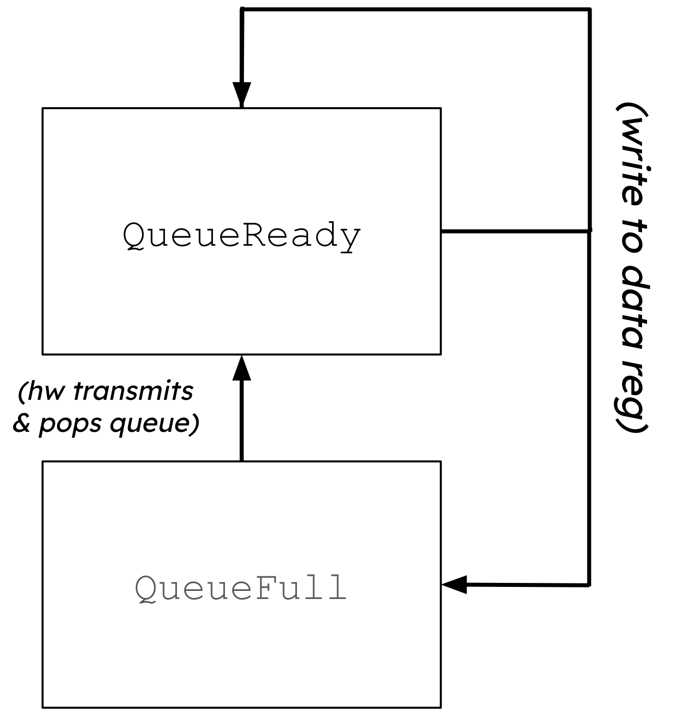
Read hardware status. BUSY indicates when a transmission is active. FULL indicates when the FIFO transmit queue is full; **DATA** must not be written when FULL is asserted.

(Hypothetical UART Hardware Specification)

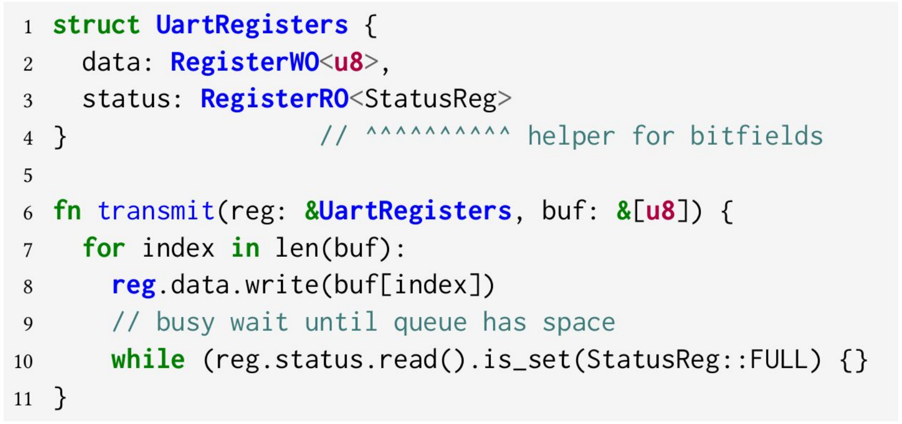


(Developer Mental Model of HW Specification)

Let’s build a UART driver…



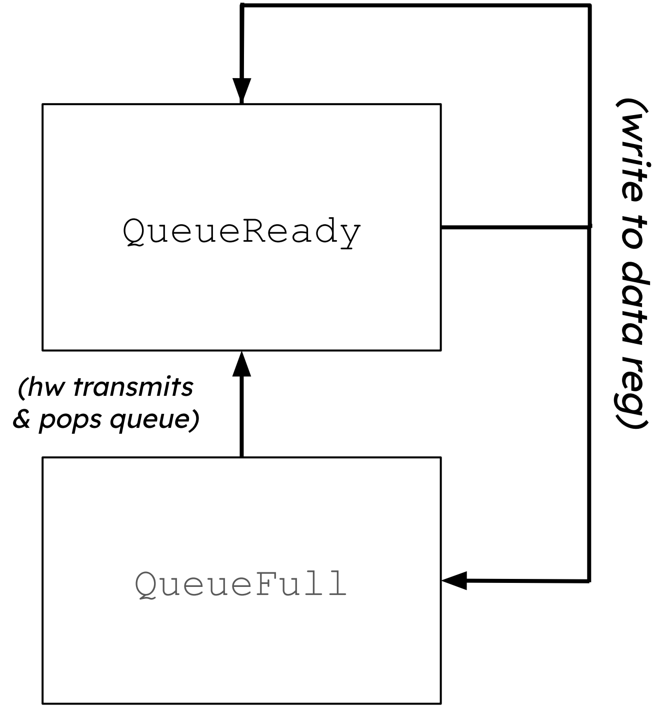
(Developer Mental Model of HW Specification)



```
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)

```
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?

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?

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

Formal Verification

(challenging; requires domain specific expertise).

Outline

- Introducing device protocol violations
- How do we build drivers today?
- **TypeState programming**
- Our System
- Evaluation & Closing Thoughts

A TypeStated Queue

- Encode system properties into the type-system.

```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]
}
```

(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)

```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>
}
```

Outline

- Introducing device protocol violations
- How do we build drivers today?
- TypeState programming
- **Our System**
- Evaluation & Closing Thoughts

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

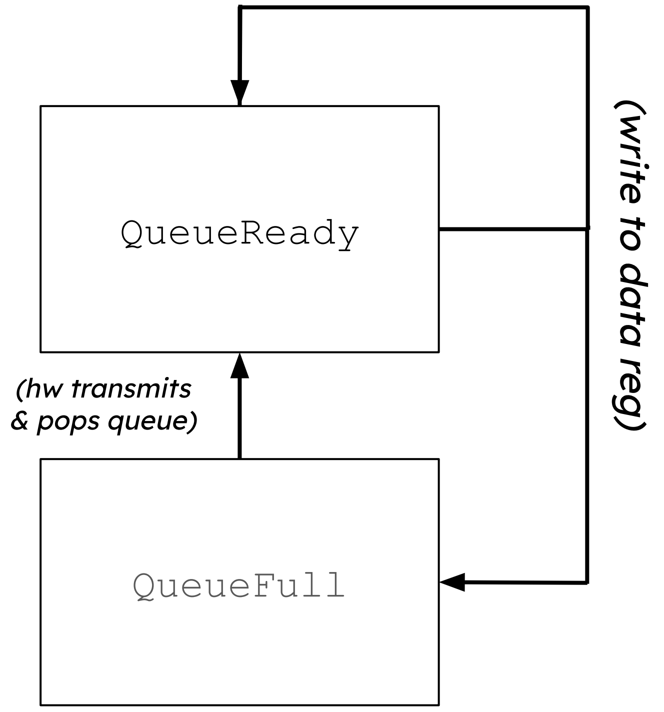
- Achieve device protocol enforcement with <u>minimal to no overheads in runtime and code size.</u>

TypeStates

DSL

- **Primary contribution:** Introduce a refinement to type-states and principled approach to <u>model hardware-software concurrency</u> using type-states.

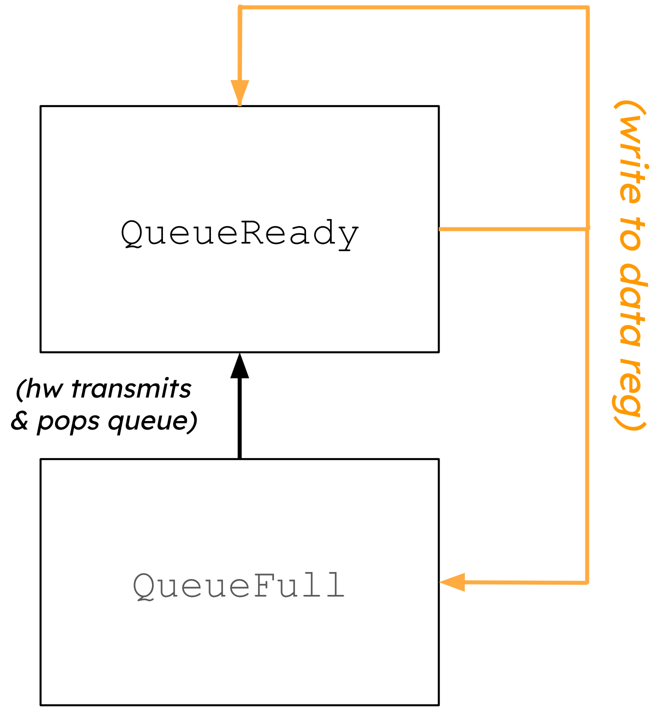
We observe… there are two classes of hw state transitions



(Developer Mental Model of HW Specification)

We observe… there are two classes of hw state transitions

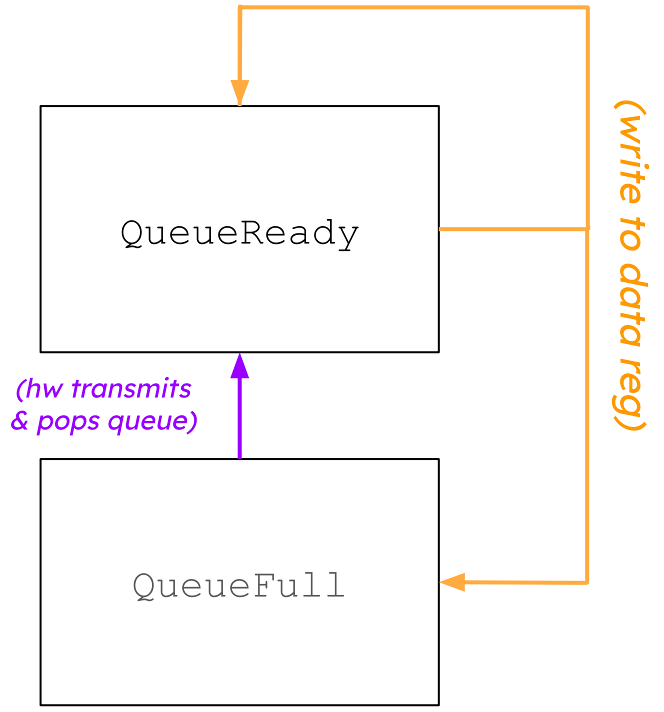
- Software-initiated



(Developer Mental Model of HW Specification)

We observe... there are two classes of hw state transitions

- Software-initiated
- Hardware-initiated



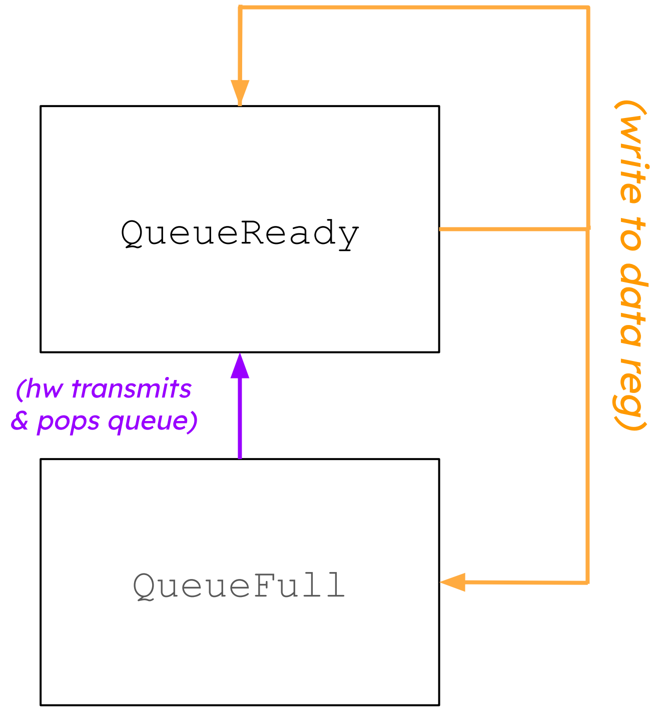
(Developer Mental Model of HW Specification)

We observe... there are two classes of hw state transitions

- Software-initiated
- Hardware-initiated

Categorize hardware states into two mutually exclusive families

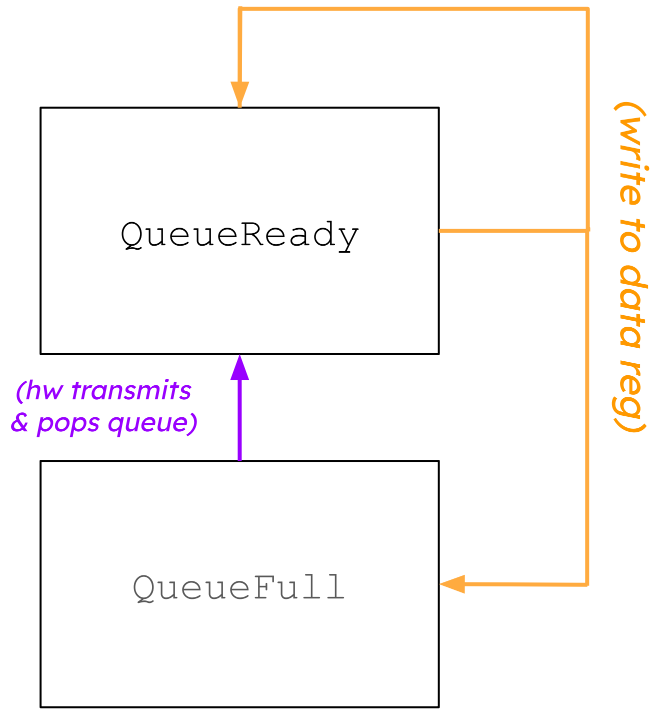
- transient state
- stable state



(Developer Mental Model of HW Specification)

Stable State

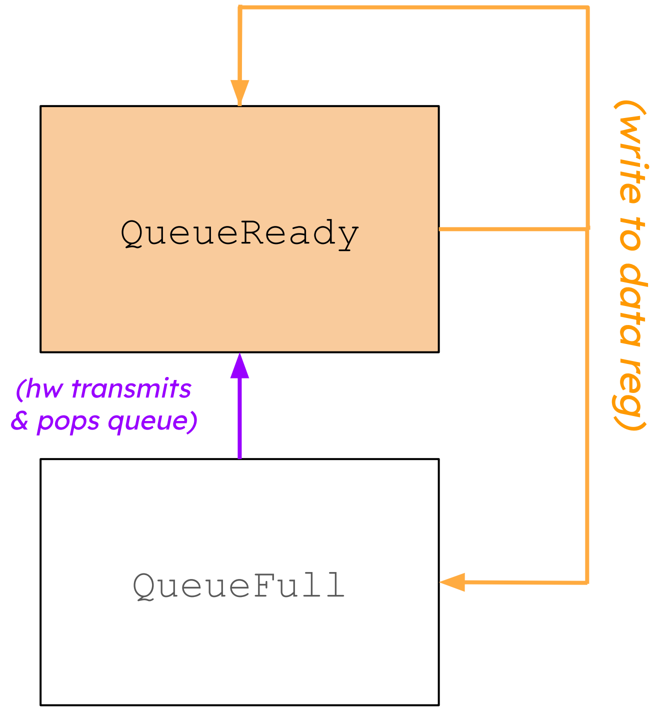
- Hw state that *can only be* exited with a software-initiated state transition.



(Developer Mental Model of HW Specification)

Stable State

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



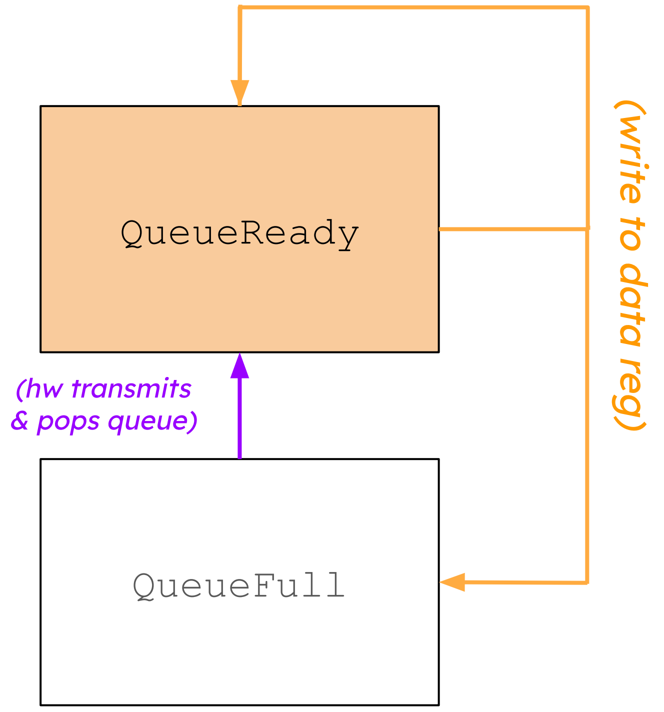
(Developer Mental Model of HW Specification)

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.



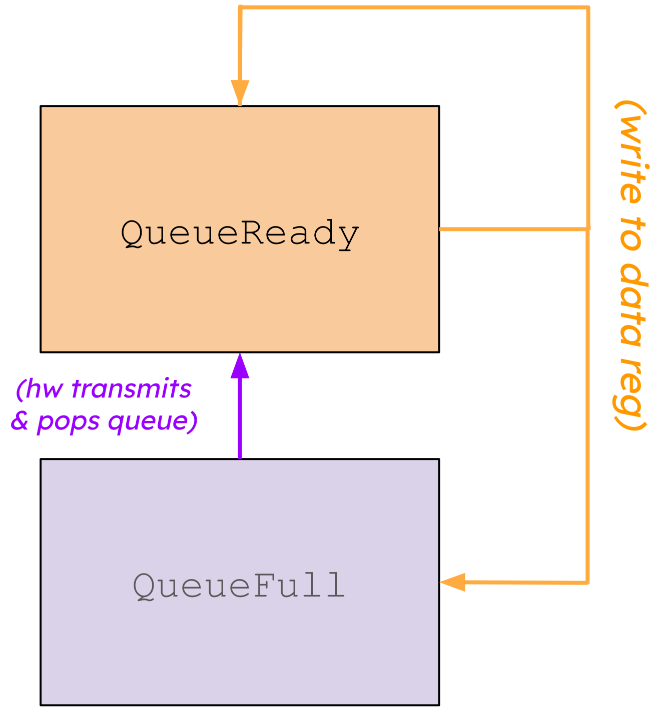
(Developer Mental Model of HW Specification)

Stable State

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

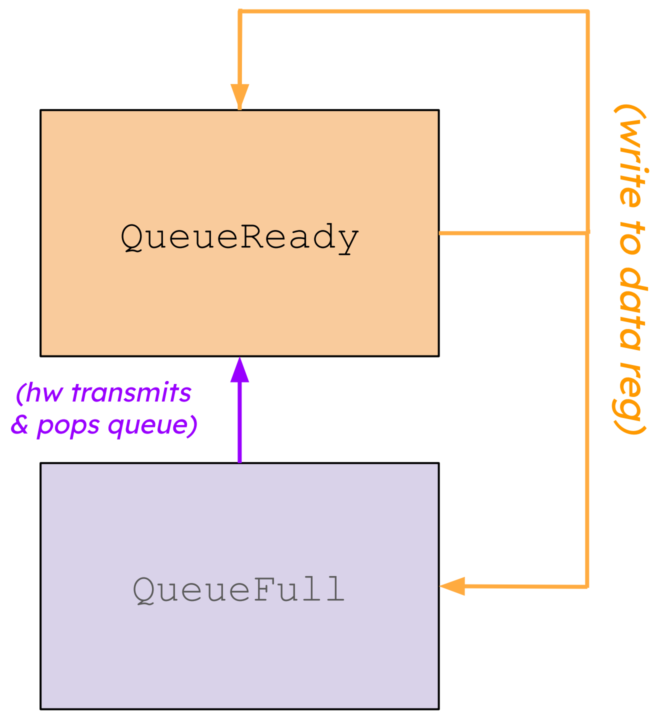
Transient State

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



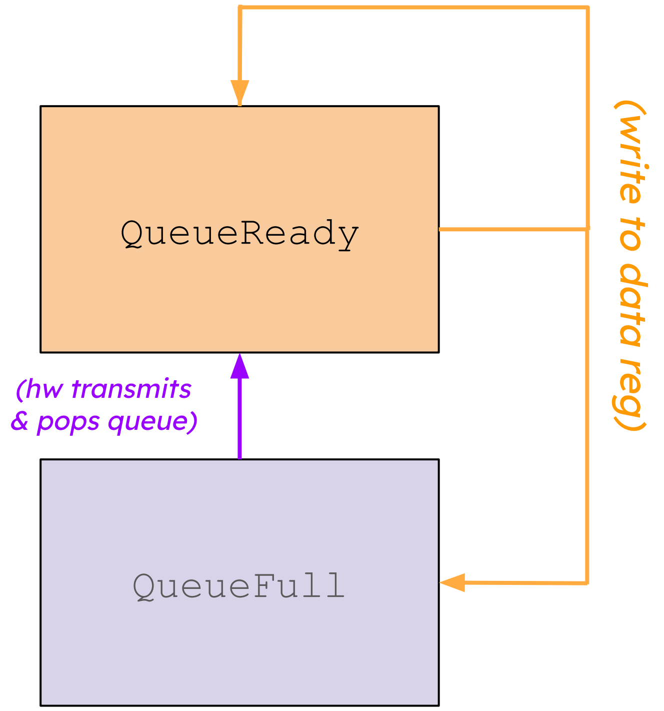
(Developer Mental Model of HW Specification)

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



(Developer Mental Model of HW Specification)

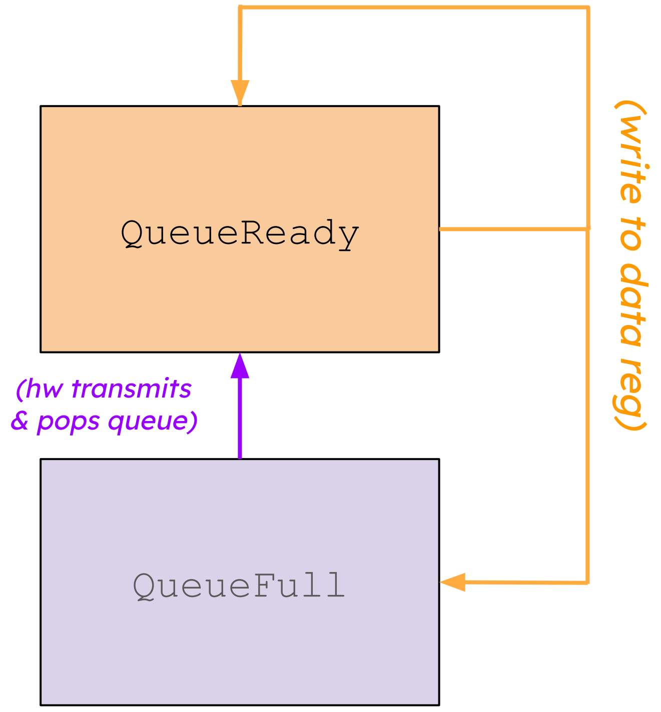
- Stable states can be modeled with out-of-the-box typestates.
- Transient states cause typestates to no longer accurately model hw (violate static invariance).



(Developer Mental Model of HW Specification)

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

Typestates + restrict transient state operations & re-synchronization mechanism

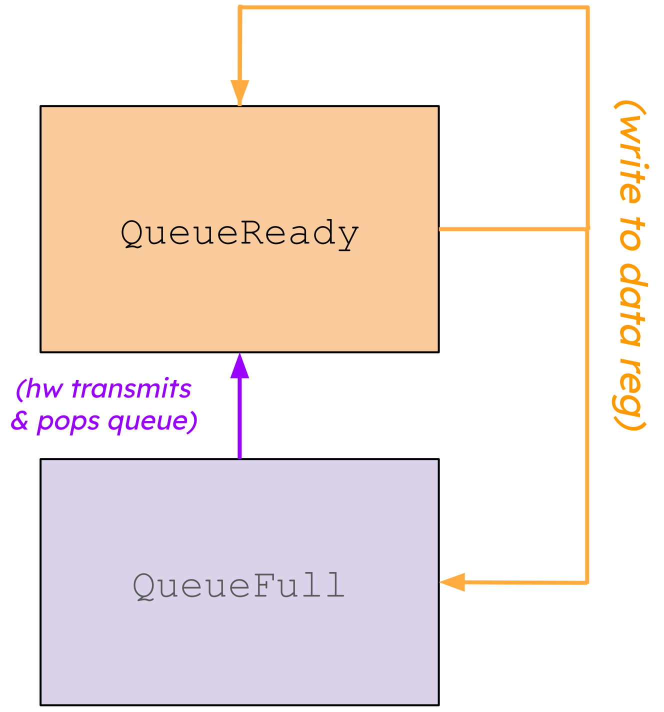


(Developer Mental Model of HW Specification)

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

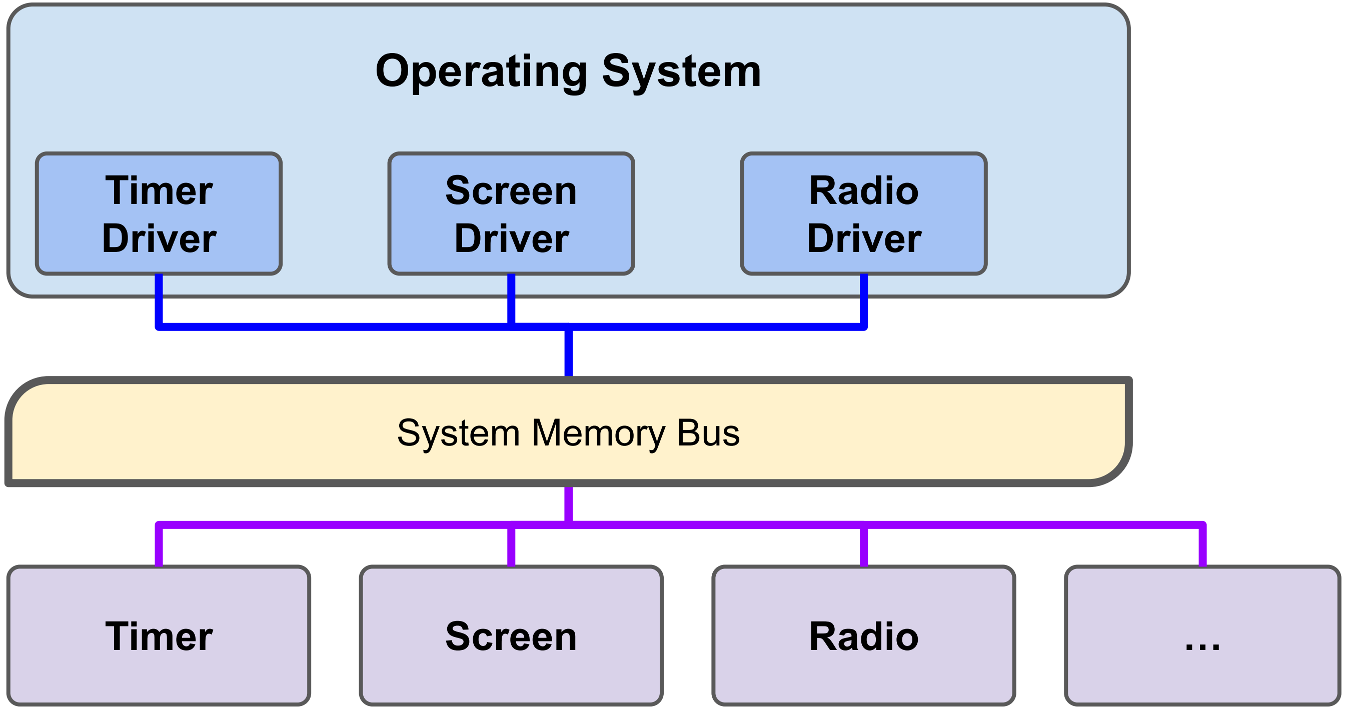
Typestates + restrict transient state operations & re-synchronization mechanism

```
Careful: Transient states have
potential for TOCTOU bugs!
```



(Developer Mental Model of HW Specification)

(Recall) Key Insight: Software talks to hardware through a “narrow waist” — memory-mapped I/O

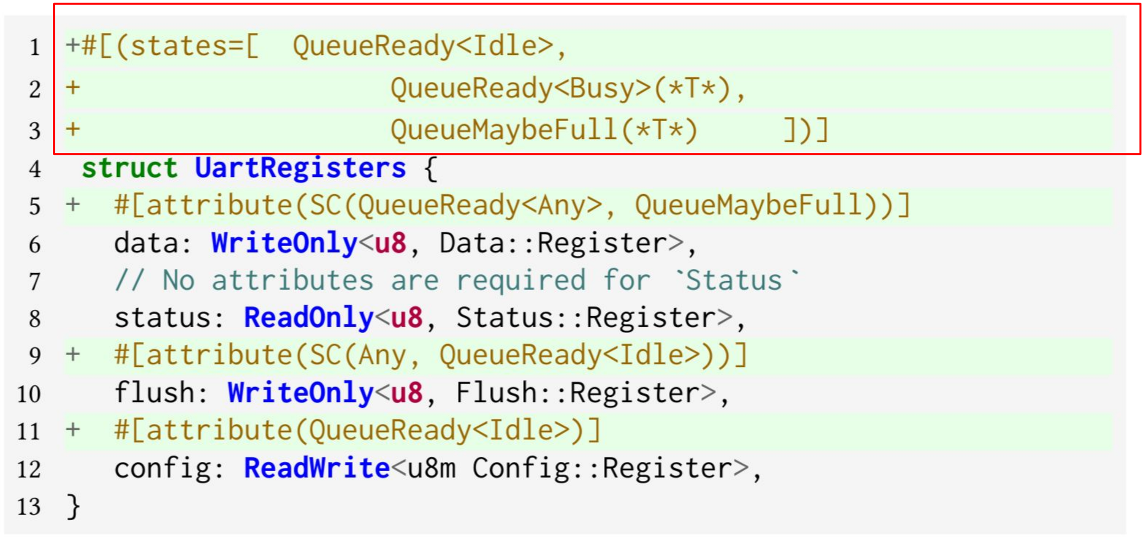


```
+#[(states=[  QueueReady<Idle>,
+                   QueueReady<Busy>(*T*),
+                   QueueMaybeFull(*T*)     ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady<Any>, QueueMaybeFull))]
   data: WriteOnly<u8, Data::Register>,
   // No attributes are required for `Status`
   status: ReadOnly<u8, Status::Register>,
+  #[attribute(SC(Any, QueueReady<Idle>))]
   flush: WriteOnly<u8, Flush::Register>,
+  #[attribute(QueueReady<Idle>)]
   config: ReadWrite<u8m Config::Register>,
}
```

Annotations for updated UART driver

Enforce device protocols by constraining MMIO using type-states.

1. Label states and mark transient states

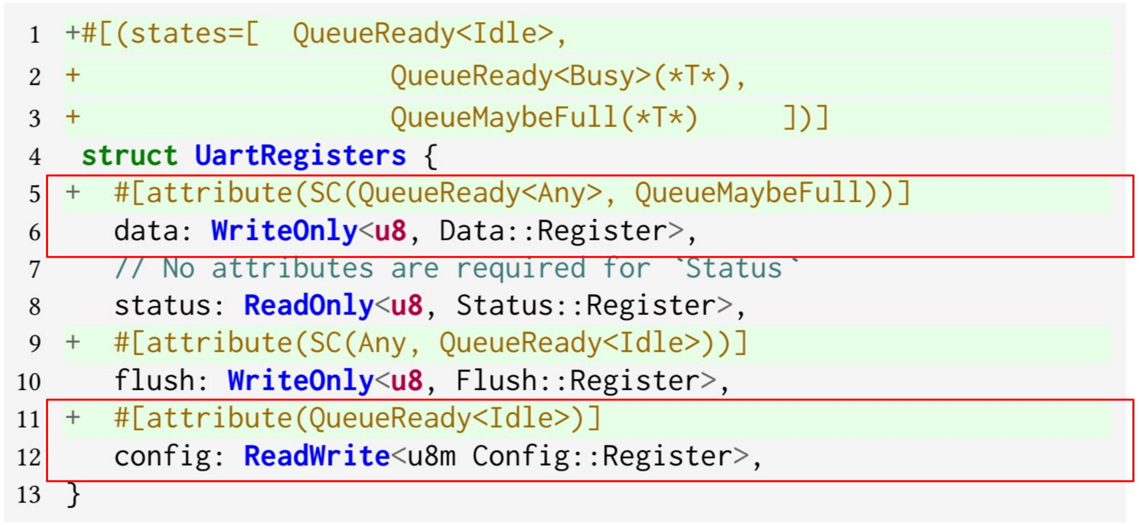


```
+#[(states=[  QueueReady<Idle>,
+                   QueueReady<Busy>(*T*),
+                   QueueMaybeFull(*T*)     ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady<Any>, QueueMaybeFull))]
   data: WriteOnly<u8, Data::Register>,
   // No attributes are required for `Status`
   status: ReadOnly<u8, Status::Register>,
+  #[attribute(SC(Any, QueueReady<Idle>))]
   flush: WriteOnly<u8, Flush::Register>,
+  #[attribute(QueueReady<Idle>)]
   config: ReadWrite<u8m Config::Register>,
}
```

Annotations for updated UART driver.

Enforce device protocols by constraining MMIO using type-states.

1. Label states and mark transient states
2. Add constraints to registers

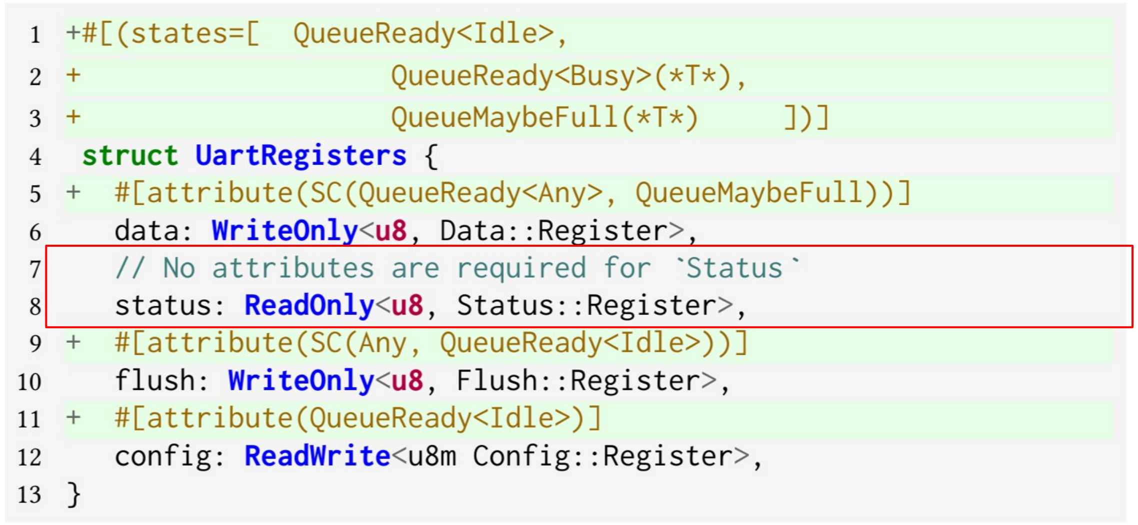


```
+#[(states=[  QueueReady<Idle>,
+                   QueueReady<Busy>(*T*),
+                   QueueMaybeFull(*T*)     ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady<Any>, QueueMaybeFull))]
   data: WriteOnly<u8, Data::Register>,
   // No attributes are required for `Status`
   status: ReadOnly<u8, Status::Register>,
+  #[attribute(SC(Any, QueueReady<Idle>))]
   flush: WriteOnly<u8, Flush::Register>,
+  #[attribute(QueueReady<Idle>)]
   config: ReadWrite<u8m Config::Register>,
}
```

Annotations for updated UART driver.

Enforce device protocols by constraining MMIO using type-states.

1. Label states and mark transient states
2. Add constraints to registers



```
+#[(states=[  QueueReady<Idle>,
+                   QueueReady<Busy>(*T*),
+                   QueueMaybeFull(*T*)     ])]
 struct UartRegisters {
+  #[attribute(SC(QueueReady<Any>, QueueMaybeFull))]
   data: WriteOnly<u8, Data::Register>,
   // No attributes are required for `Status`
   status: ReadOnly<u8, Status::Register>,
+  #[attribute(SC(Any, QueueReady<Idle>))]
   flush: WriteOnly<u8, Flush::Register>,
+  #[attribute(QueueReady<Idle>)]
   config: ReadWrite<u8m Config::Register>,
}
```

Annotations for updated UART driver.

Enforce device protocols by constraining MMIO using type-states.

DSL / proc-macro autogenerates type-states

```rust
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: ReadOnly<u8, Status>,
  reset: SCRegisterWO<S, u8>,
  config: RWRegister<N, S, u8>,
}

// Wrapper around state changing registers.
struct SCRegisterWO<S: State, T> {
  reg: WriteOnly<T>,
  associated_state: PhantomData<S>,
}

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

// This impl is only generated for S==QueueReady<Idle>,
// which enforces config's device protocol invariants.
impl <T> RWRegister<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

DSL / proc-macro autogenerates type-states

1. Modified MMIO register `struct`

```rust
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: ReadOnly<u8, Status>,
  reset: SCRegisterWO<S, u8>,
  config: RWRegister<N, S, u8>,
}

// Wrapper around state changing registers.
struct SCRegisterWO<S: State, T> {
  reg: WriteOnly<T>,
  associated_state: PhantomData<S>,
}

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

// This impl is only generated for S==QueueReady<Idle>,
// which enforces config's device protocol invariants.
impl <T> RWRegister<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

DSL / proc-macro autogenerates type-states

1. Modified MMIO register `struct`

```rust
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: ReadOnly<u8, Status>,
  reset: SCRegisterWO<S, u8>,
  config: RWRegister<N, S, u8>,
}

// Wrapper around state changing registers.
struct SCRegisterWO<S: State, T> {
  reg: WriteOnly<T>,
  associated_state: PhantomData<S>,
}

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

// This impl is only generated for S==QueueReady<Idle>,
// which enforces config's device protocol invariants.
impl <T> RWRegister<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

DSL / proc-macro autogenerates type-states

1. Modified MMIO register `struct`
2. Wrap tock registers in type-state

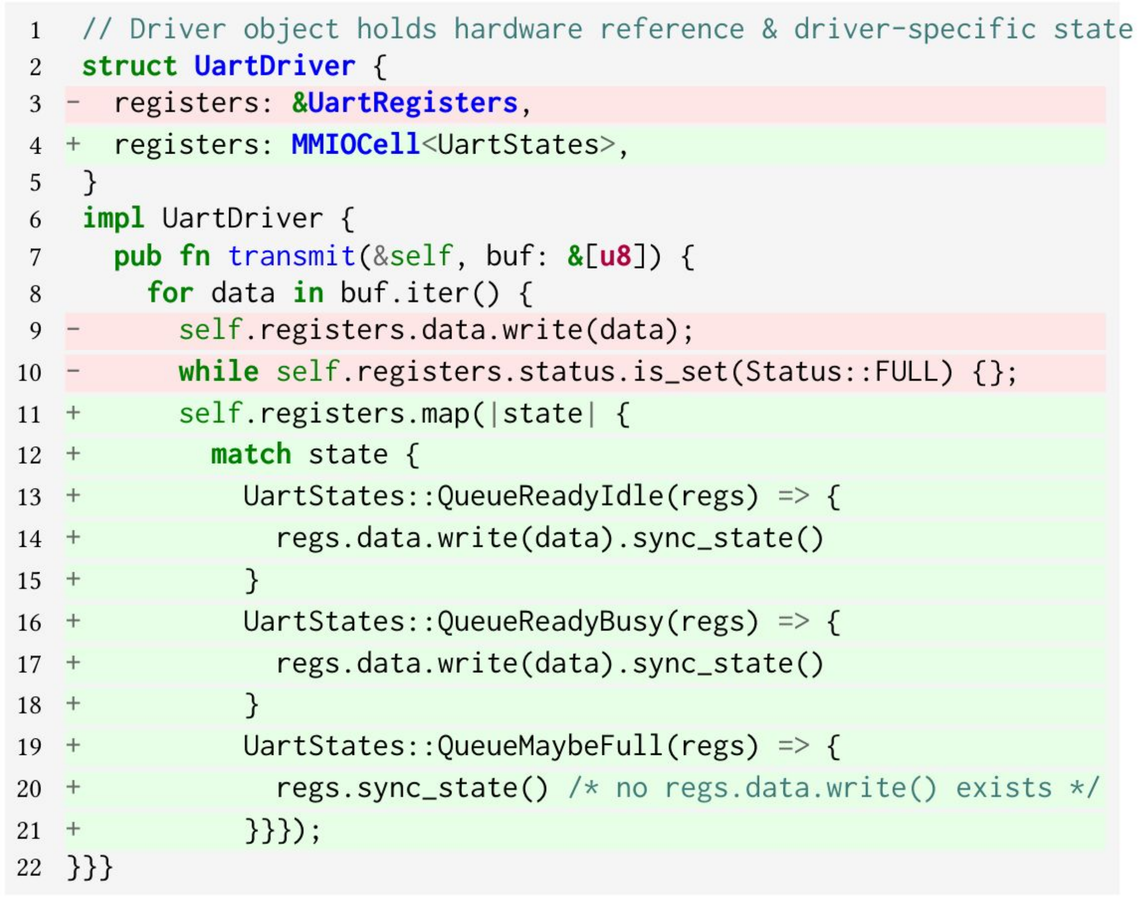
```rust
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: ReadOnly<u8, Status>,
  reset: SCRegisterWO<S, u8>,
  config: RWRegister<N, S, u8>,
}

// Wrapper around state changing registers.
struct SCRegisterWO<S: State, T> {
  reg: WriteOnly<T>,
  associated_state: PhantomData<S>,
}

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

// This impl is only generated for S==QueueReady<Idle>,
// which enforces config's device protocol invariants.
impl <T> RWRegister<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

DSL / proc-macro autogenerates type-states

1. Modified MMIO register `struct`
2. Wrap tock registers in type-state
3. Only define valid transitions

```rust
// Hardware object made generic over device state.
struct UartRegisters<S: State> {
  data: SCRegisterWO<S, u8>,
  status: ReadOnly<u8, Status>,
  reset: SCRegisterWO<S, u8>,
  config: RWRegister<N, S, u8>,
}

// Wrapper around state changing registers.
struct SCRegisterWO<S: State, T> {
  reg: WriteOnly<T>,
  associated_state: PhantomData<S>,
}

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

// This impl is only generated for S==QueueReady<Idle>,
// which enforces config's device protocol invariants.
impl <T> RWRegister<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

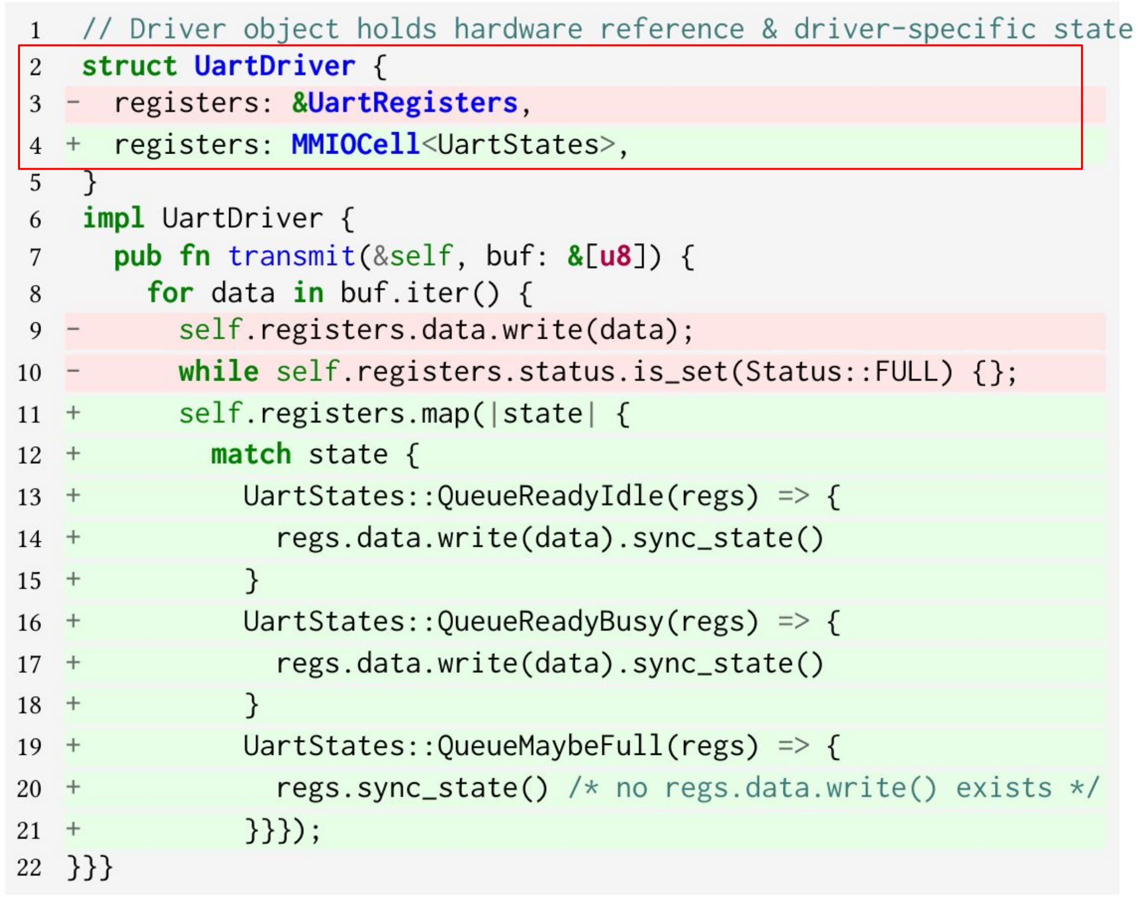
Updated transmit



```diff
 // Driver object holds hardware reference & driver-specific state
 struct UartDriver {
-  registers: &UartRegisters,
+  registers: MMIOCell<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::QueueReadyIdle(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueReadyBusy(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

No longer possible for the transmit method to attempt to write DATA when the queue is possibly full!

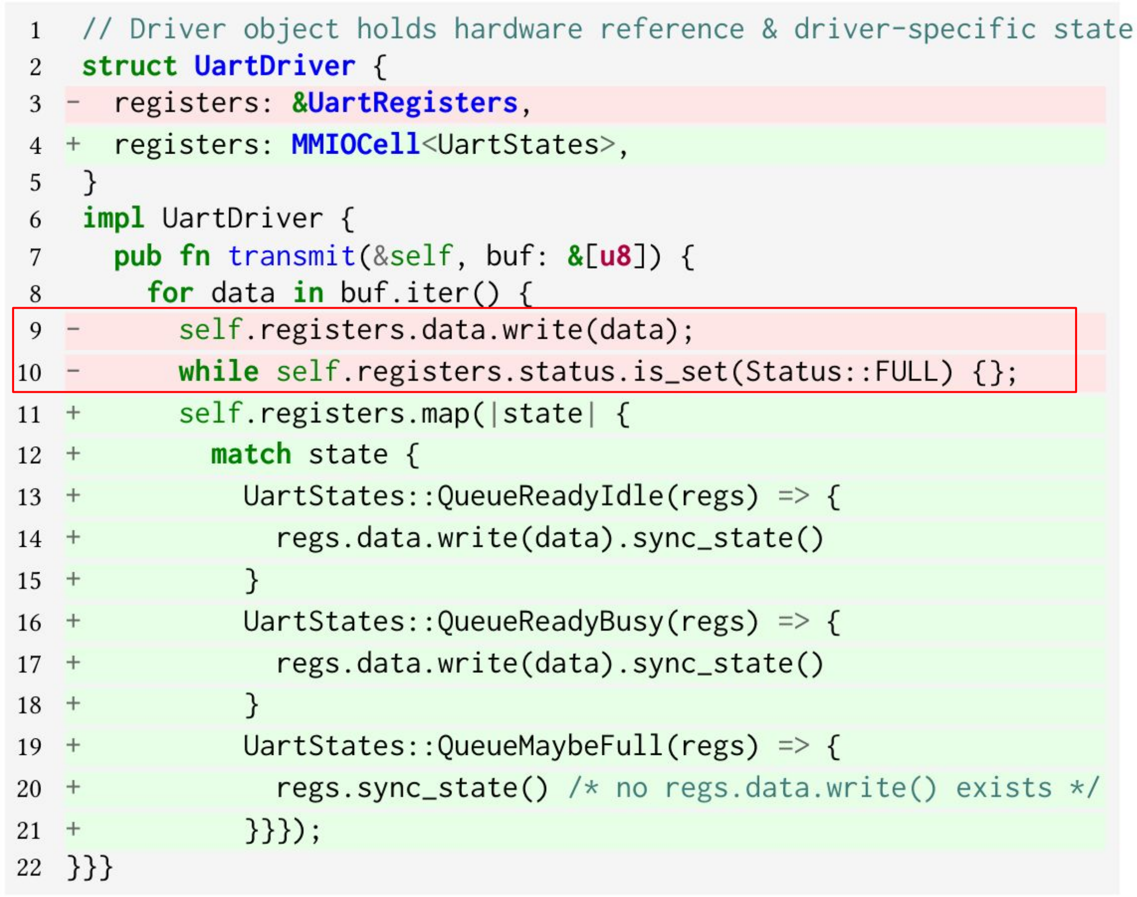
Updated transmit



```
  // Driver object holds hardware reference & driver-specific state
  struct UartDriver {
-   registers: &UartRegisters,
+   registers: MMIOCell<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::QueueReadyIdle(regs) => {
+             regs.data.write(data).sync_state()
+           }
+           UartStates::QueueReadyBusy(regs) => {
+             regs.data.write(data).sync_state()
+           }
+           UartStates::QueueMaybeFull(regs) => {
+             regs.sync_state() /* no regs.data.write() exists */
+           }}});
}}}
```

No longer possible for the transmit method to attempt to write DATA when the queue is possibly full!

Updated transmit



```
 // Driver object holds hardware reference & driver-specific state
 struct UartDriver {
-  registers: &UartRegisters,
+  registers: MMIOCell<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::QueueReadyIdle(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueReadyBusy(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

No longer possible for the transmit method to attempt to write DATA when the queue is possibly full!

Updated transmit

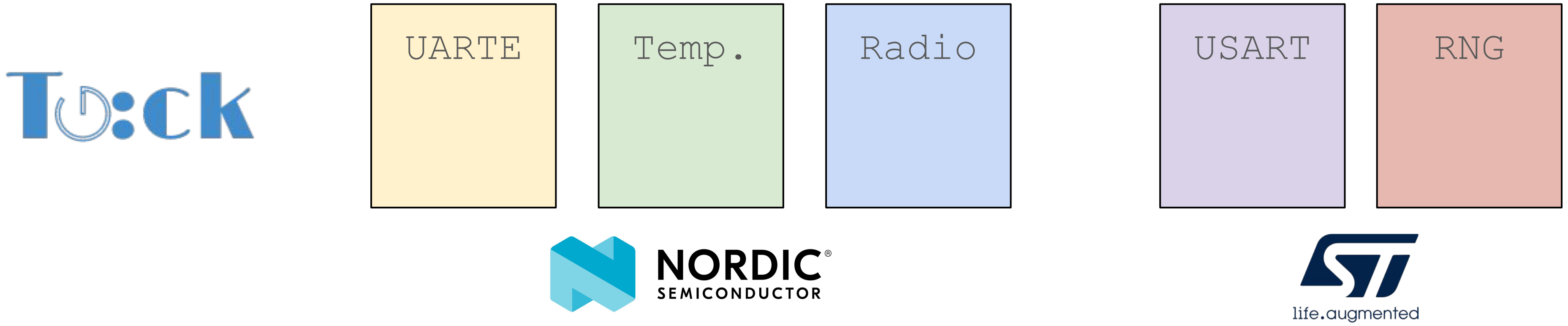
```
 // Driver object holds hardware reference & driver-specific state
 struct UartDriver {
-  registers: &UartRegisters,
+  registers: MMIOCell<UartStates>,
 }
 impl UartDriver {
   pub fn transmit(&self, buf: &[u8]) {
     for data in buf.iter() {
-      self.registers.data.write(data);
-      while self.registers.status.is_set(Status::FULL) {};
+      self.registers.map(|state| {
+        match state {
+          UartStates::QueueReadyIdle(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueReadyBusy(regs) => {
+            regs.data.write(data).sync_state()
+          }
+          UartStates::QueueMaybeFull(regs) => {
+            regs.sync_state() /* no regs.data.write() exists */
+          }}});
}}}
```

No longer possible for the transmit method to attempt to write DATA when the queue is possibly full!

Outline

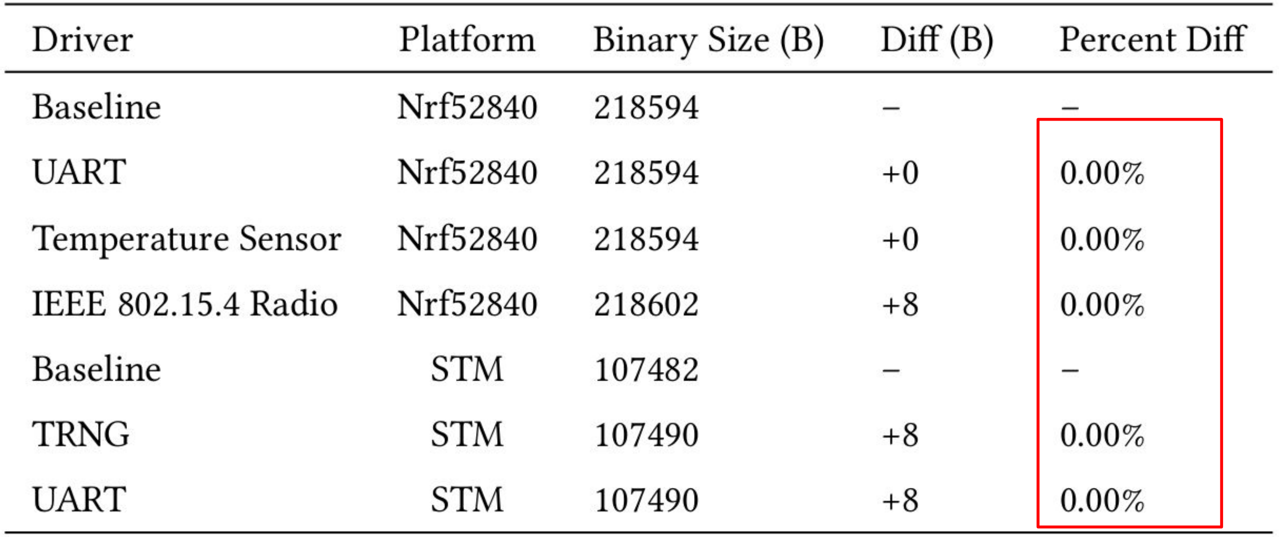
- Introducing device protocol violations
- How do we build drivers today?
- TypeState programming
- Our System
- **Evaluation & Closing Thoughts**

Implementation with TockOS



What’s the catch…

- Code size?
- Developer effort?
- Runtime performance?

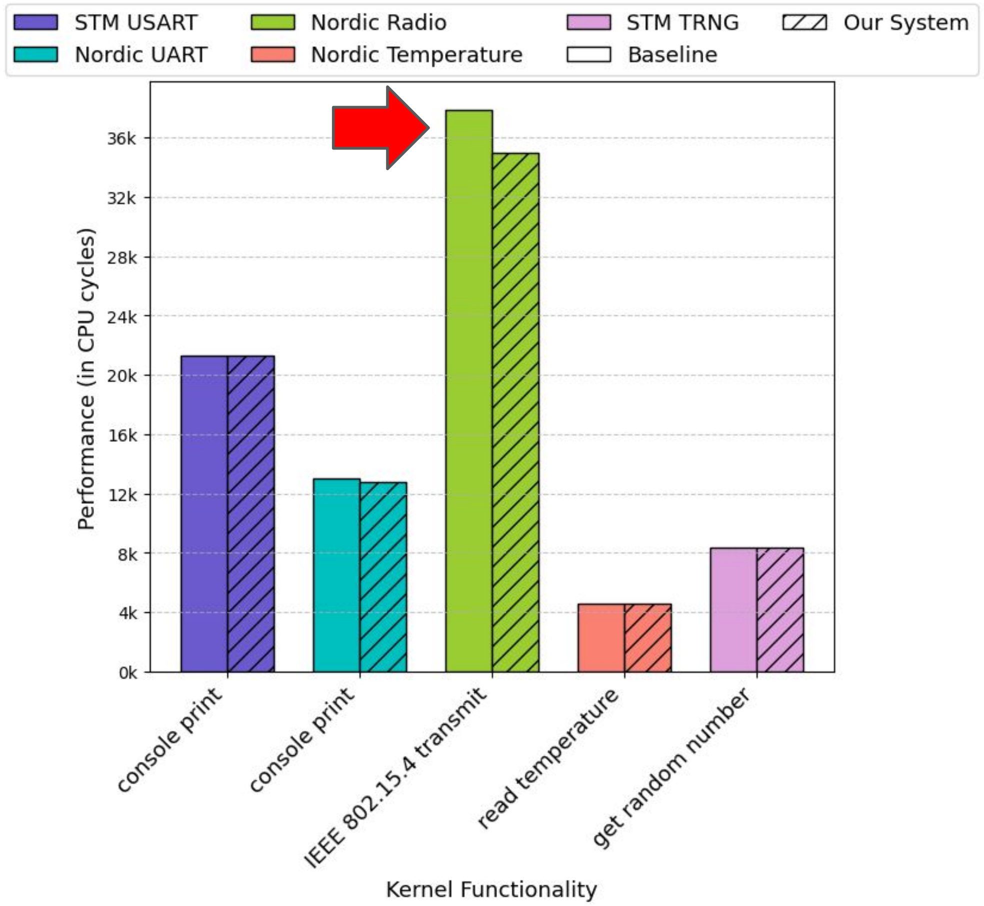


Driver	Platform	Binary Size (B)	Diff (B)	Percent Diff
Baseline	Nrf52840	218594	–	–
UART	Nrf52840	218594	+0	0.00%
Temperature Sensor	Nrf52840	218594	+0	0.00%
IEEE 802.15.4 Radio	Nrf52840	218602	+8	0.00%
Baseline	STM	107482	–	–
TRNG	STM	107490	+8	0.00%
UART	STM	107490	+8	0.00%

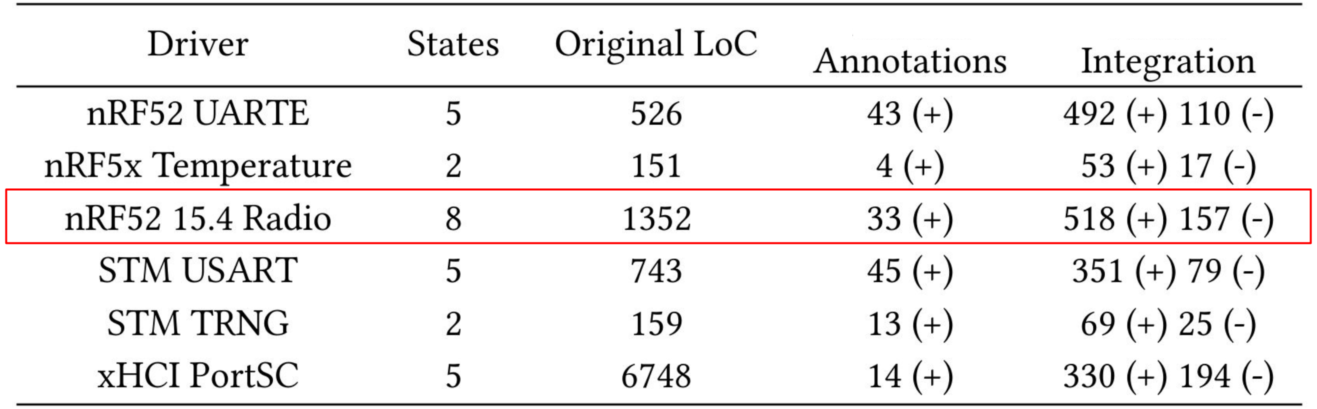
Code size of total kernel binary image for a baseline kernel image and kernel integrating our system into drivers.

Our system adds no code size overhead!

Our system adds negligible runtime overheads.



MacroBenchmark Performance (in CPU cycles).



Driver	States	Original LoC	Annotations	Integration
nRF52 UARTE	5	526	43 (+)	492 (+) 110 (-)
nRF5x Temperature	2	151	4 (+)	53 (+) 17 (-)
nRF52 15.4 Radio	8	1352	33 (+)	518 (+) 157 (-)
STM USART	5	743	45 (+)	351 (+) 79 (-)
STM TRNG	2	159	13 (+)	69 (+) 25 (-)
xHCI PortSC	5	6748	14 (+)	330 (+) 194 (-)

Our system adds some developer overheads

(improving the usability is ongoing!)

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD
released by Google

July 2023

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD
released by Google

July 2023

`Add SW ACKs to radio driver`

August 2023

3 possible HW "shortcuts" to enable faster radio TX

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD released by Google

July 2023

`Add SW ACKs to radio driver`

August 2023

3 possible HW "shortcuts" to enable faster radio TX

Unable to get all 3 working (~2 weeks of development)

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD
released by Google

July 2023

Add SW ACKs to radio driver

August 2023

Integrate our system into 15.4 driver

March 2025

Unable to get all 3 shortcuts working (~2 weeks of development)

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD released by Google

July 2023

Add SW ACKs to radio driver

August 2023

Integrate our system into 15.4 driver

March 2025

Unable to get all 3 shortcuts working (~2 weeks of development)

- Updated our state machine in DSL to use all 3 TX HW shortcuts

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD released by Google

July 2023

Add SW ACKs to radio driver

August 2023

Integrate our system into 15.4 driver

March 2025

Unable to get all 3 shortcuts working (~2 weeks of development)

- Updated our state machine in DSL to use all 3 TX HW shortcuts
- Compiler identifies sections of driver that must be updated (errors)

Case Study – NRF52 IEEE802.15.4 Driver

Tock
OPENTHREAD released by Google

July 2023

Add SW ACKs to radio driver

August 2023

Integrate our system into 15.4 driver

March 2025

Unable to get all 3 shortcuts working (~2 weeks of development)

- Updated our state machine in DSL to use all 3 TX HW shortcuts
- Compiler identifies sections of driver that must be updated (errors)

Updated and working driver in ~2 hours!

Case Study – NRF52 IEEE802.15.4 Driver

Tock

OPENTHREAD released by Google

July 2023

Add SW ACKs to radio driver

August 2023

Integrate our system into 15.4 driver

March 2025

Unable to get all 3 shortcuts working (~2 weeks of development)

- Updated our state machine in DSL to use all 3 TX HW shortcuts
- Compiler identifies sections of driver that must be updated (errors)

Updated and working driver in ~2 hours!

50% decrease in driver interrupts; 8% runtime improvement

Our system statically prevents device protocol violations using typestates.

Imposes minimal to no code size and runtime overheads.

Crate coming soon!

(will be a counterpart to tock-registers)