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

Tyler Potyondy
UC San Diego
La Jolla, California, USA

Anthony Tarbinian
UC San Diego
La Jolla, California, USA

Leon Schuermann
Princeton University
Princeton, New Jersey, USA

Eric Mugnier
UC San Diego
La Jolla, California, USA

Adin Ackerman
UC San Diego
La Jolla, California, USA

Amit Levy
Princeton University
Princeton, New Jersey, USA

Pat Pannuto
UC San Diego
La Jolla, California, USA

Abstract

Hardware provides driver authors both a strict specification of the operations a driver is allowed to do, and a highly permissive interface full of operations a driver can do. Authoring drivers that adhere to the provided hardware *device protocol* is challenged by dynamic definitions of what a driver should do based on the hardware's state. This is further complicated by increasingly capable hardware which may transition between states concurrently and independently from the software driver.

We present Abacus, a framework that statically prevents drivers from violating device protocols. Abacus refines type-states to model hardware-software concurrency and presents a formalization of hardware states into two families: stable and transient states. The Abacus framework provides a domain specific language for developers to encode device protocol invariants in tens of lines of code, and, using the generated Abacus type-states, statically prevents device protocol bugs. We demonstrate the Abacus framework's practicality by integrating it into drivers in two Rust OSes. We find that Abacus imposes minimal to no overhead in code-size and runtime performance, statically detects device protocol violations, and enables the usage of hardware features that would otherwise be prohibitively complex.

CCS Concepts: • **Software and its engineering → Operating systems**; **Domain specific languages**.

Keywords: Device Driver Bugs; Type-State Programming

ASPLOS '26, March 21–26, 2026, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790207

ACM Reference Format:
Tyler Potyondy, Anthony Tarbinian, Leon Schuermann, Eric Mugnier, Adin Ackerman, Amit Levy, and Pat Pannuto. 2026. Rage Against the State Machine: Type-Stated Hardware Peripherals for Increased Driver Correctness. In *Proceedings of ASPLOS '26, March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790207

1 Introduction

Low-level device driver code is notoriously difficult to get correct. Sitting at the boundary between hardware and software, it requires developers to understand both the intricacies of the hardware they are trying to support, and of the software environment these drivers are used in. At the same time, it is critical to get right: low-level driver bugs at best endanger the system's availability, and at worst can result in critical safety violations. Finally, drivers make up a disproportionate percentage of a modern OS's code base—70% or more—and are often written by a wide array of contributors. Given this premise, it is no surprise that driver code is a common culprit for operating system faults and exploits [26].

The recent emergence and widespread adoption of new, safe systems programming languages—chief among them Rust—promise to make building such low-level systems safer: they eliminate entire classes of bugs like out-of-bounds accesses or data races by construction, and have shown significant impact in real-world settings [45]. However, rewriting everything in Rust will not solve all issues. In particular, for device drivers to be correct, we need more than mere memory and type safety: we require that drivers adhere to their hardware's specification and interface contract. With this paper, we present a solution to this problem: we introduce Abacus, a Rust framework that statically prevents drivers from violating hardware specification invariants.

Specifically, Abacus addresses *device protocol violations*, bugs where software issues commands to hardware that violate the hardware specification [38]. Consequences of these bugs can range from relatively benign misbehavior

Device Protocol Invariant	C-like MMIO	Typed Registers	Traditional Type-State	Abacus Type-State
Bitfield Modify	●	●	●	●
Write-1 to Clear	○	●	●	●
WriteOnly Restriction	–	●	●	●
ReadOnly Restriction	–	●	●	●
State transition on write	–	–	●	●
State transition on read	–	–	●	●
State-specific register access	–	–	○	●
State-specific bitfield access	–	–	○	●

Table 1. Common device protocol invariants and support for their enforcement across MMIO interfaces. In strongly-typed languages, libraries such as `tock-registers` constrain registers and bitfields to legal accesses [47]. Traditional type-states can track changes made by software, but can enforce state-specific access rules *only* if hardware cannot change state on its own. Abacus gracefully handles joint custody, where hardware transitions state independent of (and possibly without notifying) software.

(e.g., ignoring commands or dropping packets [16]), to crippling (e.g., powering off a peripheral with an active DMA transaction can hang the system bus which then requires a watchdog reset to recover [2]), all the way to catastrophic failures (e.g., igniting batteries [29]).

Unfortunately, there are basically no guardrails for driver code. Devices expose an unconstrained interface to hardware through mechanisms such as memory-mapped input-output (MMIO) operations. In contrast to this permissive interface, device datasheets—PDFs written in natural language—provide strict specifications to the defined and likewise valid hardware operations that a driver may perform. Further complicating matters, these constraints change dynamically and are dependent on the current state of hardware. Table 1 gives a summary of common device protocol invariants.

Device protocol violations[1] plague drivers in real systems. An analysis of Linux's USB, IEEE 1394, and PCI drivers [38] finds that device protocol violations account for 38% of their patched bugs—the highest percentage of all types of bugs. Performing a similar analysis using two popular Rust OSes, Tock [24] and Redox [35], we find 21 instances of patched device protocol violations—evidence that device protocol violations still remain a problem in systems written in Rust.

Traditional approaches towards driver correctness have not been able to address these issues in practice. Functional testing or fuzzing provides only low-confidence into whether a given driver implementation is correct and handles all edge-cases—it is necessarily incomplete and cannot statically guarantee correctness. Conventional formal verification approaches provide robust correctness guarantees, but require highly specific expertise and require prohibitive time and effort to prove the correctness of practical systems [11, 21]. Type-state programming represents a reasonable trade-off: it is more rigorous than testing but also more easily specified than conventional verification methodologies [22].

When applying "standard" type-state programming to driver development we run into issues: although type-states can maintain correctness within a concurrent software-software context [50], traditional type-state programming cannot usefully encapsulate the hardware interface due to the control that hardware exerts upon the state machine. Namely, modern hardware performs implicit state transitions concurrently and independently from software to yield increased runtime performance, parallelism, and power savings [20].

Abacus introduces a principled approach to model device protocols using type-states and maintain correctness despite hardware-initiated state transitions. Our approach provides a refinement to type-state programming and a formalization of hardware states into two families: *stable states*, which only software can transition out of, and *transient states*, which can be exited by software or hardware.

2 Abacus Solves Real-World Problems

To contextualize and motivate this work, we start with three examples of correctness and performance deficiencies in the implementations of Ethernet, UART, and a wireless radio in modern, "safe", Rust-based OSes (Tock and Redox).

Protocol Errors are Often Subtle. Redox had a device protocol violation in the Intel 82599 10 Gb Ethernet Controller [36]. This hardware features receive filters, but states in the datasheet that *"before the receive filters are updated/-modified the* `RXCTRL.RXEN` *bit should be set to* `0b`*"* [15]. One code path, `set_promisc`, failed to observe this invariant. The resulting *undefined behavior* is easy to miss during device bringup[2]—packets will still be received, just *perhaps* not *all* of the packets one would expect to see in 'promiscuous mode'.

This bug persisted in Redox for over four years [13]. With Abacus, attempting to write to the `FCTRL` register while `RXCTRL.RXEN==1` would be rejected at compile-time.

Even "Easy" Things are Hard. Universal Asynchronous Receiver/Transmitter (UART) is a ubiquitous, de facto standard, with hardware designs dating back to the 1970s [44]. UARTs are usually the simplest communication interface and are readily available on nearly every MCU. Despite this, six of the eighteen patched device protocol violations we studied in Tock are UART related.

[1] We use the Ryzhyk et al. [38] definition of device protocol violations: "putting the device into an incorrect state, misinterpreting device state, incorrectly parsing or generating data exchanged with the device, issuing a sequence of commands to the device that violates the device-protocol, specifying incorrect timeout values for device operations, and endianness violations."

[2] This bug was copied into the Redox repository in 2020, "Move ixgbed in-tree" (`93d5ce`) and exists in the June 2019 "Initial commit" (`79e1bc`) of github.com/ackxolotl/ixgbed; we were unable to trace provenance further.

UART operates at speeds much slower than CPU processors. Thus, UART peripherals often provide hardware transmit/receive queues; these queues specify device protocol invariants and perform implicit hardware transitions. Failing to discern the hardware state in Tock UART drivers resulted in: kernel crashes due to overfilling a hardware transmit queue [23], race conditions and data loss due to assuming the hardware receive queue is empty [16], and complete peripheral failures due to performing operations before the hardware returned to a valid state [3]. Abacus prevents all of these bugs by construction.

We Don't Even Try the Really Hard Stuff. Hardware peripherals are shipping with increasingly capable "DMA engines"—i.e., Nordic's Programmable Peripheral Interconnect [43], Microchip's Sleepwalking [28], TI's µDMA [14], etc.—these are effectively co-processors that jointly control peripheral devices. In principle, these "hardware shortcuts" ease tight timing requirements and improve runtime performance as the CPU can remain suspended longer.

In practice, reasoning about device protocol correctness under this joint-ownership model can be prohibitively difficult, and can result in hardware shortcuts not being used. Consider 15.4[3] drivers for the Nordic nRF52 family of chips. Although the nRF52 15.4 radio provides four hardware shortcuts to allow for more efficient radio transmit operation, we find that many radio drivers do not use all available shortcuts. Tock uses two of the four transmit shortcuts, where it 'has to' in order to transmit acknowledgements fast enough. The `nrf-rs` embedded-hal states, *"shortcuts will be kept off by default and only be temporarily enabled within blocking functions."*[4] RIOT-OS uses only two of the simplest shortcuts.[5]

With Abacus, advanced hardware features previously deemed "too hard to get right" are not just tractable, but are surprisingly accessible. In Section 7.2, we walk through a case study that details adopting Tock's 15.4 driver to use additional hardware shortcuts in just two (expert developer) hours. This new driver reduced radio interrupts by 50% and decreased the runtime overhead to transmit by 8%.

2.1 Paper Roadmap

To start, we introduce a minimalist UART peripheral which we use to explain the challenges with writing drivers today (Section 3), the limitations of current type-state paradigms for the hardware–software interface (Section 4), how Abacus refines type-state programming to permit joint custody with hardware (Section 5), and finally the complete Abacus framework (Section 6). We then show how Abacus integrates with existing OS code and handles more complex features such as DMA and interrupts (Section 7), which includes a case study of applying Abacus to complex hardware (i.e. nRF52 15.4 radio mentioned above). We close with an evaluation of the Abacus framework and show that it exhibits near-zero overhead in real world code (Section 8).

3 How Do We Write Drivers Today?

We illustrate the process of—and challenges with—writing a device driver by considering a simple example: a UART peripheral featuring a hardware-backed transmit queue, inspired by the 16550 UART.

A Brief Primer on MMIO. From the host CPU's perspective, hardware peripherals behave as asynchronous co-processors that communicate via a hardware-software abstraction (e.g. MMIO provides a shared memory abstraction).[6] In the case of MMIO, host reads or writes to memory addresses do not point to traditional storage, but are instead backed by registers in the peripheral state machine. Obeying the hardware's interaction semantics across this shared memory abstraction are unique to each peripheral, and the onus of adherence is wholly on the driver.

Step 1: Extract Mental Model from Specification

To develop a device driver, developers first consult a peripheral's reference manual. This document provides an abstract description of peripheral function and a description of its software interface. Figure 1a excerpts the MMIO interface relevant for transmitting data through our UART. This UART has an internal, limited-depth FIFO queue that transmits whenever it is non-empty. **Data** can be written to insert a byte into this queue, but should only be written when the queue has space available. To check if the queue is full, a driver must read the Full field from **Status**.

From the hardware description, developers build a mental model of the peripheral's behavior, such as the state machine in Figure 1b. Conceptually, this UART is in one of two states: Either there is space available in the transmit queue (`QueueReady`) or the queue is full (`QueueFull`).[7] Developers must take care to only enqueue new data when the peripheral is in the `QueueReady` state.

Step 2: Implement the Software State Machine

Next, a developer will implement a driver based on their mental model of the peripheral's behavior and invariants. Listing 1 presents an implementation of a synchronous UART driver that exposes a function to transmit an array of bytes

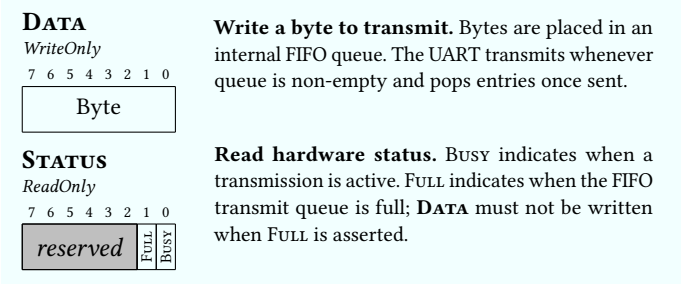
[3] IEEE 802.15.4 is a popular IoT wireless standard with largely software-driven MACs that imposes tight timing constraints on sending, receiving, and acknowledging packets [10]—an ideal use case for hardware shortcuts.

[4] github.com/nrf-rs/nrf-hal/blob/445688/nrf-hal-common/src/ieee802154.rs#L200

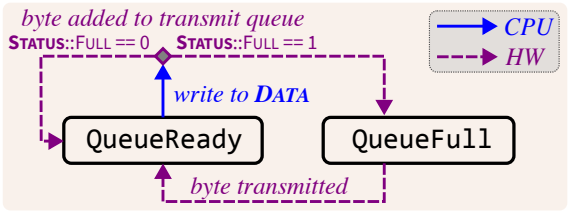
[5] github.com/RIOT-OS/RIOT/blob/5136d2/cpu/nrf5x_common/radio/nrfble/nrfble.c#L50

[6] Other hardware-software abstractions include PCIe, x86 ports, or buses such as QSPI. For the purposes of this example, we will focus on MMIO as the hardware-software communication abstraction.

[7] Note that these are not states one would see in the underlying HDL source. The hardware implementation will have states such as `IDLE`, `START_BIT`, `TX_DATA`, etc. Similarly, an implementation would not have a literal **Status** register, but something more like `assign full_out=(fifo_head+1)==fifo_tail`.



(a) Manufacturer Hardware Specification.



(b) Developer mental model of hardware specification.

Figure 1. Initial UART Hardware Peripheral. Example of a manufacturer hardware specification for UART transmission, and the mental model of hardware a developer might form using this 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 }
```

Listing 1. UART Driver `transmit` Implementation. Based on the developer's mental model from Figure 1b.

over the UART interface. The `transmit` function maps the state machine to the driver, which enqueues a byte for transmission by writing to **DATA** and then busy-waits by polling the FULL field in **STATUS**. A developer will then test various uses, verifying that the function does indeed transmit data, including very long messages certain to exceed the hardware queue, which busy-waits as-needed—Done!

Steps 3, 4, 5, 6… Discover and Fix Bug (Repeat)

Although this driver appears to abide by the specified hardware invariants, there is a subtle bug. The `transmit` function assumes that the transmit queue is empty when this function is called, a problematic assumption that may not be true across a software reset or in the context of an interrupt.

Indeed, a variation of this bug exists in the current (35a1d8) `panic!` handler implementation for several Tock boards (e.g., hail/src/io.rs#L48-52). At 48 MHz CPU and 115,200 baud, if there are more than 60 bytes remaining in a UART DMA transaction when a `panic!` occurs, the 200,000 cycle delay (kernel/src/debug.rs#L166-L169) will not be enough.

Takeaway: HW/SW Interface is Too Permissive

Nothing in the hardware–software interface itself indicates when a driver's interactions are non-conformant. The MMIO abstraction models peripheral interactions as memory read and write operations, with no restrictions on whether any read or write (with a particular value) is valid at a given time. This is in contrast to the peripheral's specification, which restricts conformant driver implementations to performing a limited set of MMIO interactions, which change dynamically based on the peripheral's state.

The hardware-software boundary is, however, a useful 'narrow-waist'—in the case of MMIO and many other hardware-software interface abstractions, e.g. Peripheral Component Interconnect Express (PCIe), all hardware accesses go through a `struct UartRegisters` reference. If we can refine this type to capture peripheral state, we can impose state-based constraints on the hardware-software interface, and thus restrict software to only perform valid hardware-software interactions.

4 Applying Type-States To Drivers

Type-state programming is a promising candidate to encapsulate hardware-software interactions and statically eliminate device protocol violations. Here, we provide a primer on type-state programming, look at how it integrates with modern safety-oriented languages, and then examine why contemporary type-state paradigms struggle to capture hardware–software interactions.

4.1 Background: Type-state Programming

Type-state programming encodes states and high-level system properties into a type system [1]. States are modeled as types that define only the operations which are valid for the given state. Executing an operation will change the type to reflect the new, resultant state. This enables static enforcement through the compiler's type checker in strongly typed languages. Because only the set of valid operations for a given state is defined for the corresponding type, any invalid operation will be undefined and result in a compilation error.

Type-state programming has a long and rich body of work ranging from the introduction of the concept [46], through their application and enforcement using static analysis tools [9], to the eventual native support for type-states in strongly- and affine-typed languages [1, 6]. As one such language, Rust provides out-of-the-box support for type-state programming, and, moreover, can represent type-states

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

Listing 2. Type-stated Queue. While similar to the UART FIFO, this assumes that software knows the depth of the queue and that software will be responsible for all transitions.

through *owned, zero-sized types* (ZSTs) which provide a zero-cost abstraction with no runtime artifact [48]. State transitions *consume* the current state (through Rust's move semantics) and return the new state.

4.2 Example: A Type-Stated Fixed-Depth Queue

We illustrate type-state development in Listing 2 with a state machine that describes a fixed-depth queue. Lines 1–4 introduce *marker-types* that represent the set of states the modeled state machine can be in. Then, the `Queue` is made generic over these `State`-types (line 6). This enables a `Queue` to have different methods available based on the state that it is in. For example, `Queue<Empty>` cannot remove a non-existent element and thus does not have a `pop()` method (lines 10–12). Similarly, a full queue cannot accept an element and thus does not feature a `push()` method (lines 22-24).

4.3 Hardware Custody Breaks Type-State Invariants

Listing 2's `Queue` statically prohibits *overrun* or *underrun* conditions. As the bug in our driver in Listing 1 is a potential queue overrun, type-states seem like a promising approach to eliminate these types of bugs. Unfortunately, we cannot directly apply Listing 2's approach to the UART driver of Listing 1. Because hardware performs implicit state transitions that occur in parallel to the software driver, e.g. popping the UART transmit queue, our type-state must properly model the state machine in a concurrent context.

Type-states provide compile-time correctness by assuming a *static invariance* property—the object type must reflect the modeled runtime state [46]. If this property is violated, type-state guarantees become moot. In the presence of concurrency, there is a heightened risk of violating static invariance and producing a type-state error as the true state may not be reflected across all concurrent executions.

Prior work provides a mechanism to maintain type-state correctness in a concurrent context through runtime checks [50]. This mechanism relies upon the assumption that state transitioning operations can be delayed; when a given context attempts to transition from a state A to a state B, the operation is routed through a control layer which delays the operation if there exists an ambiguous global state that may violate static invariance. While this works for concurrent software-software contexts, it fails when applied to hardware-software concurrency as hardware-initiated state transitions exist below a software control plane and cannot be delayed pending runtime checks.

4.4 Takeaway: Type-state Programming Cannot Model Hardware Initiated State Transitions

Existing approaches do not allow type-states to readily model device protocols, as hardware is the *primary custodian* of the device state machine. Software drivers are afforded only *partial custody*, which means that decisions are sometimes made without notification or consent of type-stated software—violating static invariance.

5 Abacus Concurrent Hardware Type-States

In this section, we present how Abacus refines type-state programming to better model hardware-software concurrency, to preserve a refined static invariance, and to thus prevent device protocol violations.

The Naïve Strawman. Existing type-states, of course, can *correctly* model hardware—most simply by defining one `Any` state that the device is assumed to always transition to. This is, however, not a *useful* abstraction. For instance, while hardware can transition our UART from `QueueFull`→`QueueReady`, hardware *cannot* cause the device to leave the `QueueReady` state, only a software action (writing **Data**) can do that. Abacus understands the *reachability* of hardware-initiated transitions, which allows a more expressive interface.

5.1 Stable and Transient States

Abacus's key observation for refinement of type-states is that device state transitions are either exclusively software-initiated or hardware-initiated. We define two mutually exclusive families of device states. A *stable state* can only be exited by a software-initiated state transition. Conversely, a *transient state* is any state with at least one hardware-initiated outbound state transition. In other words, a device can transition out of a transient state without explicit software involvement (including without notification).

For the UART model in Figure 1b, `QueueReady` is a stable state; the transmit queue can never be filled without software inserting a new word of data. `QueueFull` is a transient state; the peripheral transmits enqueued words without software involvement, and it can remove an element from the queue at any time. Thus, eventually, and not correlated with any MMIO operation issued by the software driver, UART will transition from the `QueueFull` state to the `QueueReady` state.

5.2 Controlled Type-State Divergence

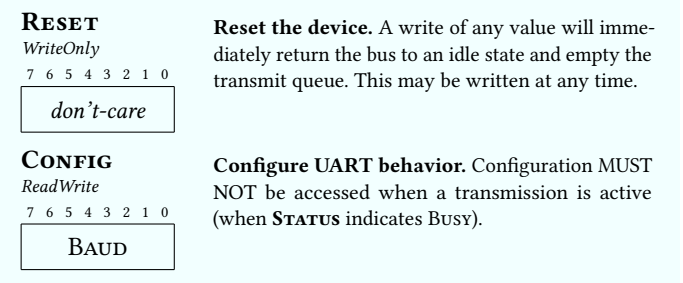
When hardware-initiated state transitions occur, the software driver's type-state will diverge from the true device state and violate traditional static invariance. Transient states by definition have at least one hardware-initiated transition and are the source of this divergence. Transient states are non-deterministic and may undergo a state transition at any point in time; from a software perspective, a device believed to be in a transient state may be in that state or any hardware-reachable state (i.e., states reachable by hardware-initiated transitions). Although we cannot prevent this divergence, we can refine our type-states to account for it.

Normally, violating static invariance nullifies compile-time guarantees because the misrepresenting type-state misidentifies the operations that are valid for the true device state. However, many operations are valid across multiple states—e.g., it is valid to read the example's **Status** from any state. To preserve device protocol correctness while maximizing expressiveness, Abacus limits the operations a software driver may initiate to the intersection of operations valid in all reachable states.

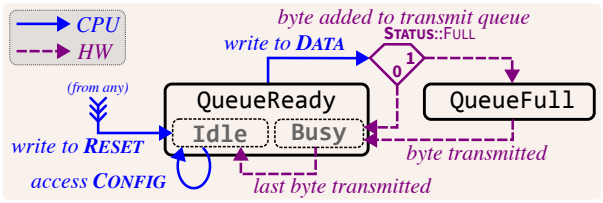
Stable states and the transient state refinement are an application of existing theory to reason about concurrent programs; namely, Rely-Guarantee logic [17]. Jones addresses interfering concurrent programs (i.e., access to shared resources) and explicitly models the non-determinism of each concurrent component through component specific "rely-conditions" (assumptions the component can make) and "guarantee-conditions" (assumptions the component promises to uphold). Abacus uses type-states and the Rust type system to encode and enforce "rely-conditions" that specify the operations valid for a given hardware state; Abacus type-states correctly modeling the underlying hardware represent "guarantee-conditions". Together, stable states, by definition, and transient states, through our transient state refinement, uphold this "guarantee-condition" and allow Abacus type-states to provide compile-time device protocol enforcement across hardware-software concurrency.

5.3 Resolving Divergence with Re-Synchronization

Eventually, software needs to sync back up with the true device state. One option is for software to force the device into a consistent state—e.g., by performing an explicit software interaction, such as "reset", that is valid for the given transient state and returns the device to a known, stable state. More commonly, however, software can simply ask the device about its current state. Abacus provides an explicit re-synchronization mechanism that uses information exposed by the peripheral to transition its software model out of a transient state. As such synchronization depends on introspecting hardware state dynamically, it is ultimately hardware-specific. Abacus thus requires drivers to implement trusted hooks (Rust `traits`) that can determine, e.g., through interrupts or a status register, the current hardware state.



(a) Additional Manufacturer Hardware Specifications.



(b) Developer model integrating extended hardware specification.

Figure 2. Extensions to the UART Specification. On top of the base specification from Figure 1a, this adds **Reset** to cancel in-flight operations and **Config** for baud rate.

6 The Abacus Framework

In this section, we show how the Abacus framework applies our refined type-states to code and how we can use Abacus type-states to statically prevent device protocol violations.

A More Advanced UART. Before diving in, we extend the capability of our running example as shown in Figure 2. In particular, we add the **Reset** capability proposed earlier to cancel active transactions. We also add a configuration register for the baud rate (transmission speed), which should only be accessed when the device is idle. To account for the **Config** access restriction, we update the model to add `Idle` and `Busy` substates to the `QueueReady` state. For more advanced peripherals, especially those with many states, substates can (greatly) ease reasoning, hence their use in our exemplar here, rather than another top-level state.

6.1 Requirements for Correctness

To ensure that the conditions of the Abacus refinement are upheld when we integrate Abacus hardware type-states into driver code we must (1) label hardware states as either transient or stable, (2) ensure the operations defined for a transient type-state are valid across all reachable states, and (3) provide a resynchronization mechanism.

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

Listing 3. Abacus annotations for updated UART. Annotations are lightweight and integrate easily with hardware definitions while enforcing Abacus refinement conditions.

6.2 A DSL for Ergonomic and Accessible Correctness

The Abacus framework uses a domain-specific language (DSL) to ease developer burden for upholding the refinement's conditions. This DSL allows developers to specify a peripheral's device protocol and state machine in tens of lines of annotations and then autogenerates the needed Abacus type-states.

Abacus is designed to easily adapt driver code by annotating peripheral definitions with information about the hardware state machine. The annotations are compatible with existing type-based enforcement of other device protocol invariants (e.g., a WriteOnly wrapper type for registers). Hooking this 'lowest-level' definition of hardware allows Abacus to enforce device protocol invariants for all hardware accesses. We describe the design of the Abacus DSL by example, breaking down Listing 3 to show how Abacus integrates with the UART device object.

6.2.1 Defining Device States. The annotations on lines 1-3 enumerate the device states and specify that `QueueReady<Busy>` and `QueueMaybeFull` are transient states. This satisfies the refinement condition of labeling all hardware states as either transient or stable.

Notice that Figure 2b's `QueueFull` has been updated to `QueueMaybeFull` to capture software's inherent uncertainty of hardware state. Anecdotally, we have observed that just the exercise of expressing the developer device model in a form that must type-check frequently results in an iterative refinement of device models, sometimes exposing bugs in drivers we had written prior to even conceiving of Abacus.

6.2.2 Restricting Operations and Reachability. The Abacus DSL supports two primary classes of hardware register annotations: *standard constraints* and *state changing constraints*. Line 11 shows a standard constraint, which enforces the **Config** access restriction. A state change attribute, `SC([from],[to])`, both defines when access is valid (`[from]`) and the effect of access (transition to `[to]`). Using this syntax, line 5 specifies that writing to the **Data** register

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

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

// Wrapper around constrained MMIO.
struct AbacusRegisterRW<N, S: State, T> {
  reg: RegisterRW<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> AbacusRegisterRW<Config, QueueReady<Idle>, T> {
    fn read(&self) -> T {..}
    fn write(&self, T) {..}
}
```

Listing 4. UART Driver with Abacus Type-States. This is autogenerated code and represents a snippet of the expanded form of the annotated driver from Listing 3. Note that line 4 passes through from the original source unmodified, as there are no restrictions on access to **Status**.

is valid from any `QueueReady` substate and that doing so causes a transition to the `QueueMaybeFull` state. Similarly, line 9 specifies that writing to **Reset** is valid in any state and transitions the device to `QueueReady<Idle>`.

In particular, we highlight that Abacus's concept of expressing valid operations based on reachability from transient states is fundamental. If a hypothetical device protocol depends upon a transient state, a software driver, irrespective of Abacus, cannot guarantee the absence of a time-of-check-time-of-use (TOCTOU) bug, since the hardware by definition operates concurrently and without constraint from software.

Aside: Bit Fields? To keep exposition readable, our UART only has register-level restrictions. Often, device protocols require finer-grained control—e.g., only allowing certain fields (bits) of a register to be accessed in a given state. The declaration of the `Status` type (not shown) that is used on line 8 and defines the fields for the `UartRegisters::status` member can be similarly annotated with Abacus.

6.3 Abacus Type-States

Abacus type-states and state transition methods follow a standard type-state paradigm—i.e., they use Rust's single ownership and move semantics to model state transitions. Listing 4 shows an example of Abacus type-states. This listing is a portion of the code generated by the DSL using the annotations in Listing 3. In principle, Abacus type-states can be authored manually. In practice, the types and methods are

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

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

Listing 5. Synchronization Interface. Support code and interfaces to synchronize hardware and software state are generated by the DSL, but the device-specific nature of implementations demands that developers author method bodies.

verbose and subtle, and the correctness guarantees provided by Abacus rely on the correctness of these definitions.

Device protocol restrictions are enforced with wrapper types that are generic over the current device state. For **Config**, line 6's `AbacusRegisterRW` type restricts access by restricting the `impl` on line 24 to only define methods which allow MMIO accesses when the device is `Idle`. Notably, the Abacus framework allows for registers that are valid for all states, such as **Status**, to remain unconstrained and unaffected by the type-state.

The Abacus enforcement design does presume that hardware-software interface abstractions, e.g. MMIO, are encapsulated behind a type (e.g., `RegisterRW` in our examples), such that there are `read/write()` (or equivalent) methods to intercede on. Generally, some kind of wrapper is necessary to express MMIO or PCIe accesses soundly, since the underlying mechanism does not follow the semantics expected by the abstract machine of major systems languages (without qualifiers such as `volatile`). In practice, we could not find any Rust code in major use where MMIO or PCIe accesses did not use a wrapper type amenable to Abacus.

6.4 Synchronizing Software to Hardware State

When hardware transitions the state machine, Abacus requires a synchronization mechanism to reconcile state divergence. Reconciliation logic is highly device specific; software may get state-specific interrupts, generic interrupts, or (as in our UART) need to poll a register. To provide device specific flexibility, we do not auto-generate synchronization functions, and instead, the developer is required to implement the `trait SyncState` for all transient states. An example is shown in Listing 5.

Abacus trusts `SyncState` authors to properly enforce and uphold device protocols. The DSL generates the `SyncState` interface, and failing to implement `SyncState` for a transient state will result in a compilation error. These DSL-demanded interfaces form a complete resynchronization mechanism,

```
// Driver obj. holds hardware reference & driver-specific state
struct UartDriver {
-  registers: &UartRegisters, // type from Listing 1
+  registers: AbacusCell<UartStates>, // type from Listing 5
}
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 */
+          }});
}}}
```

Listing 6. Fixing Listing 1's Bug with Abacus. With Abacus, it is no longer possible for the transmit method to attempt to write **Data** when the queue is possibly full.

which fulfills the third and final condition of the Abacus type-state refinement laid out in Section 6.1.

6.5 An Abacus-Based UART Driver

Now that we have our well-defined peripheral object built in Listings 3 and 5, we can look at a complete example of a UART driver with Abacus type-states and transitions.

One Last Detail. Abacus type-states require move semantics to model state transitions. However, a common pattern in real Rust OSes is to define drivers with methods that use `&self` (preventing a move as the object members are behind a shared reference). As an ergonomic aid, we provide a custom `Cell` type, `AbacusCell`, which uses interior mutability to resolve this tension. For details on `Cells` and interior mutability concerns in Rust, see Appendix A.

6.5.1 Adapting a Driver to Abacus.

Listing 6 demonstrates the changes needed to integrate Abacus into our UART driver. Rather than the explicit, manual register flag check on line 9, Abacus uses a match pattern followed by calling `sync_state`. In practice, `sync_state` serves to introspect the new hardware state (using the register bitflag) and encodes this into the updated type-state.

This example prioritizes simplicity and a minimal diff from the original `transmit` approach as an aid to explain Abacus clearly. A more practical implementation would avoid the repeated `map` and lazily `sync_state`.

6.6 Applying Abacus to Other Drivers and Hardware

The above UART example demonstrates how the Abacus framework and DSL is encoded, type-states are autogenerated, and finally how a developer writes a driver using Abacus. Although ideal for conveying these concepts, a UART hardware peripheral does not fully capture the complexity of many hardware peripherals and their accompanying device protocols (e.g. a wireless radio or GPU). Here we describe how Abacus can be applied to other, complex drivers and also the device protocol violations Abacus cannot prevent. Furthermore, Section 7.2 presents a case study and associated Listing 7 for a more complex driver—the NRF52 wireless radio driver.

Abacus and Complex Hardware. Recall, Abacus is a principled approach to model stateful device protocols using type states and ensures correctness despite hardware-initiated state transitions. The complexity of a driver's device protocol increases with the number of valid states and state transitions. Using the Abacus framework, we can enforce stateful device protocols so long as the device protocol is representable as a hardware state machine. The effort to implement a driver using Abacus scales relative to the number of states and transitions. Practically, this manifests in more DSL annotations to encode the device protocol and increased verbosity when implementing the driver—Abacus requires developers to explicitly consider and handle all possible/reachable hardware states when accessing registers that may have asynchronously undergone a transition.

Finally, the Abacus approach applies across any hardware-software boundary mechanism. Our current implementation (Section 7) works for devices that interact with software over MMIO as well as PCIe. Other mechanisms, e.g. x86 ports or buses such as QSPI would need to be implemented separately, though the same approach applies.

Limitations. The Abacus framework is unable to model device protocols that specify an operation is valid after some time interval (e.g. 20ms) **and do not** offer a hardware related indication (e.g. an interrupt or a status register). Furthermore, Abacus's usability is limited by the need to express the device protocol as a hardware state machine. In the case of some arbitrarily complex hardware, a state machine representation may result in a state space explosion; this would render Abacus intractable due to challenges encoding the state machine representation into the Abacus DSL.

7 Implementation

Section 6 shows how the Abacus DSL allows developers to enforce device protocols through Abacus' hardware type-states in driver code. The Section 6 UART is a theoretical example that lacks the complexities of "real" hardware (e.g. interrupts, DMA, etc.). In this section, we show Abacus can be used in real and challenging hardware drivers and how the Abacus DSL integrates with existing OS abstractions.

7.1 OS Integration

TockOS. Tock is an embedded OS that provides process isolation and the ability to run mutually distrustful applications [24]. It is used in security-critical (e.g., Root-of-Trust [5, 30]) and safety-critical (e.g., automotive [32]) settings. This, and its ability to run multiple applications make adherence to device protocols especially important: no application should allow drivers or hardware to diverge into dangerous, undefined, and potentially unrecoverable states.

We select five Tock drivers to integrate with Abacus. These include three drivers for the Nordic Semiconductor nRF52840 (temperature sensor, IEEE 802.15.4 radio, and UARTE) and two drivers for the STMicroelectronics STM32F429ZI (true random number generator, USART).

We integrate the Abacus framework DSL into each driver's MMIO register `struct` and refactor the driver to use Abacus type-states. Table 2 depicts the lines of code required per driver to annotate the MMIO struct and to refactor each driver to use Abacus. In porting these five drivers, Abacus detected a device protocol violation in the nRF52840 UARTE driver (attempted write to a disabled peripheral).

RedoxOS. RedoxOS [35] is a microkernel written in Rust. As a microkernel, Redox drivers run as user-space applications and are designed to have minimal opportunity to harm the system [34]. Device protocol violations have the potential to violate Redox's microkernel threat model; although a driver memory bug may not harm the system (due to address space isolation etc.), a device protocol violation may result in a system wide fault. We demonstrate how Abacus can be integrated into Redox drivers by adapting the eXtensible Host Controller Interface (xHCI) for Universal Serial Bus (USB) driver. More specifically, we use Abacus to constrain the Port Status and Control (PortSC) registers in xHCI.

The PortSC register functionality is closely tied to to the hardware state and is responsible for facilitating USB device connections. Furthermore, PortSC registers have a high number of transient states (e.g. a connected port can be unplugged at any moment). Table 2 depicts the lines of code required to annotate the xHCI port registers PCIe `struct` and to refactor the driver to use Abacus.

We discover one potential device protocol violation in xHCI; the driver fails to confirm that hardware is not in the `Resetting` state before reading the PortSC `PLS` state bits. This violates the device protocol as the `PLS` state bits are undefined when in the `Resetting` state.

Abacus Crate. Abacus is integrated into TockOS and RedoxOS as a Rust crate. The Abacus crate uses `tock-registers` as a dependency and provides the Abacus

framework using: a Rust procedural macro[8] to implement the DSL, `trait`s to enforce requirements for the Abacus refinement, and an `AbacusCell` implementation. This crate is platform agnostic and provides a type-stated hardware-software interface for Rust systems.

7.2 Real-World Problems - Revisited

Section 2 presents three real world examples (Redox Ethernet, Tock UART, and a 15.4 radio) of correctness and performance deficiencies that Abacus can statically prevent. We first present a case study of how Abacus is integrated into the Tock 15.4 radio driver and enables safe and straightforward use of hardware previously deemed "too hard to get right." Following this, we detail how Abacus can statically prevent the mentioned Redox Ethernet device protocol violation and Tock UART bugs.

Case Study - Nordic 15.4 Radio. Recall, the nRF52 15.4 radio provides runtime configurable hardware shortcuts that are complex to use correctly and under-utilized in the majority of embedded OS drivers. We first integrated Abacus into the existing Tock 15.4 driver. This involved reading the hardware datasheet, creating a mental model of the hardware state machine, and then annotating the driver's MMIO register struct using the Abacus DSL. Next, we updated the driver implementation to use Abacus type-states; Abacus wraps the register MMIO struct with the `AbacusCell` and `AbacusEnum`, and requires register operations occur within the `AbacusCell` closure. Listing 7 demonstrates how the 15.4 transmit function is updated to use Abacus. Notice, the non-Abacus driver attempts to track the state machine through the `self.state` field (line 3). This is no longer required as Abacus maintains an accurate model of the hardware state automatically.

Next, we updated the driver to use the full suite of hardware shortcuts. This involved updating our DSL annotations to reflect the new hardware state machine (due to the added hardware shortcuts). After updating the DSL, the Rust compiler produced compilation errors for instances that either did not consider each possible hardware state (i.e., detecting an incomplete match statement) or attempts to perform an invalid operation for the current hardware state. *Using Abacus, we successfully updated the Tock 15.4 radio driver to use the full suite of hardware shortcuts in under two hours.* This was possible because Abacus statically eliminates device protocol violations and prevents the software driver from mistaking the current hardware state—a task that becomes difficult with hardware shortcuts. Using the full suite of hardware shortcuts decreased the radio driver interrupts by 50% and provided an 8% decrease to the driver's transmit runtime overhead. *Abacus enables the usage of hardware features that may otherwise be prohibitively challenging to use correctly.*

[8]Rust procedural macros are a powerful tool for code generation [37]. Procedural macros allow developers to annotate Rust types with attributes that the macro will parse and then use as tokens for code generation.

```
1  fn transmit(&self, buf: &'static mut [u8], frame_len: usize,
2          ) -> Result<(), ErrorCode> {
3  -   if self.state.get() == RadioState::OFF {
4  -       return Err(ErrorCode::OFF);
5  -   } else if self.busy() {
6  -       return Err(ErrorCode::BUSY);
7          /*  OTHER STATES ELIDED FOR BREVITY  */
8  +   self.registers.map(|registers| {
9  +     match registers {
10 +       Nrf52RadioStore::RxIdle(reg) => {
11 +         Ok(reg.into_ccastart().into());
12 +       }
13 +       Nrf52RadioStore::Transient(reg) => {
14 +         // Check if we are already transmitting.
15 +         if self.busy() {
16 +             return Err((reg.into(), ErrorCode::BUSY));
17 +         }
18 +         let reg = reg.into_stop();
19 +         let reg_res = reg.sync_state();
20 +         if let Nrf52RadioStore::RxIdle(reg) = reg_res{
21 +             Ok(reg.into_ccastart().into());
22 +         } else {
23 +             Err(ErrorCode::BUSY)
24 +       /*  OTHER STATES ELIDED FOR BREVITY  */
```

Listing 7. Diff between the original Tock 15.4 driver and Abacus integrated 15.4 driver.

Static Bug Prevention - Redox Ethernet and Tock UART. The Ethernet and UART bugs detailed in Section 2 arise due to drivers performing operations that are undefined for the current hardware state (e.g. Redox Ethernet driver writing the `FCTRL` register while in the enabled state). Abacus can statically prevent such bugs through the use of Abacus type-states that model the hardware state. In the case of the Ethernet bug, the `FCTRL` register would be annotated in the Abacus framework and constrained to only allow writes when not in the `Enabled` state; this would statically catch the Ethernet bug as the invalid write would not be defined for the `Enabled` Abacus type-state. The UART bugs can be eliminated in a similar manner as these bugs are due to inaccurately discerning the hardware state and performing operations invalid for the current hardware state.

8 Evaluation

Abacus statically enforces device protocols in drivers while being usable across multiple OSes and hardware manufacturers. For Abacus to be practical, it must achieve device protocol enforcement while imposing minimal overheads in code-size and runtime. In this section, we evaluate Abacus' overheads with respect to code size, runtime performance, and developer burden.

Hardware Platforms

We evaluate Abacus on multiple platforms: evaluations using Tock use a Nordic nRF52840DK SoC for Nordic

Driver	States	Original LoC	Abacus Annotations	Abacus 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 (-)

Table 2. Lines of Code (LoC) required to integrate Abacus into an existing driver. Abacus Annotations refer to MMIO DSL annotations and Abacus Integration refers to required driver implementation changes.

Semiconductor-specific drivers, whereas evaluations of STM32 drivers on Tock use an STM32F429ZI. Finally, we perform evaluations of the Abacus-port of the Redox xHCI driver using a QEMU x86_64 emulated PC and a virtual keyboard device.

Developer Overhead: Lines of Code

Developers annotate register structs with the Abacus DSL and update drivers to use the generated Abacus type-states—incurring developer overheads. Table 2 shows the changes required, in lines of code (LoC), to update Tock and Redox drivers to use Abacus. The 15.4 driver—the most complex in terms of number of states—requires annotating 33 LoC using the Abacus DSL and the highest number of driver implementation changes (+518/-157).

Code Size

Table 3 presents the Abacus code size overhead in the five evaluated Tock drivers and xHCI Redox driver.

We evaluate the Tock drivers using the standard Tock build system, kernel image size reporting, and comparing binary size between the baseline kernel and kernel including Abacus integrated drivers. Across the evaluated Tock drivers, Abacus introduces *at worst, an 8 Byte overhead in the context of a kernel on the order of a 100 kB.* This is accomplished through the use of ZSTs that allow the compiler to elide all Abacus infrastructure from the compiled binary.

We measure the binary size of the compiled Redox xHCI driver crate (with and without Abacus). The Abacus xHCI driver is 7.5KB smaller (0.33%) than the standard driver; this occurs due to the Abacus integration being able to elide many repeated runtime checks. For instance, the xHCI driver polls the same hardware flag multiple times within nested function calls. Using Abacus, the type-stated register is passed to these nested function calls and eliminates these redundant checks due to Abacus modeling the state.

Runtime Microbenchmarks

For each Tock driver, we benchmark the performance (in CPU cycles) to execute the driver's methods. Figure 3 presents these microbenchmarks and demonstrates Abacus imposing in most cases moderate overheads. However, we notice in some cases that Abacus performs nearly equivalent or even better than the baseline—we attribute this to removing redundant checks and incidental implementation changes. At worst, Abacus imposes just a 40 cycle overhead, to the STM32 USART driver's `receive` method.

We benchmark Redox xHCI methods to determine the Abacus overheads using QEMU. We find that Abacus does not impose overheads. In the case of `attach_device`, Abacus eliminates redundant hardware state checks and results in the Abacus driver outperforming the standard driver.[9]

Driver	Platform	Binary Size	Diff	Percent Diff
Baseline	Nrf52840	218KB	–	–
UARTE	Nrf52840	218KB	+0	0.00%
Temperature Sensor	Nrf52840	218KB	+0	0.00%
IEEE 802.15.4 Radio	Nrf52840	218KB	+8B	0.00%
Baseline	STM	107KB	–	–
TRNG	STM	107KB	+8B	0.00%
USART	STM	107KB	+8B	0.00%
xHCI	Redox	2248KB	-7.5KB	-0.33%

Table 3. Code Size of total Tock kernel binary image for a baseline kernel and kernel integrating Abacus into UARTE, Temperature Sensor, 15.4 Radio, TRNG, USART drivers. xHCI code size baseline is the binary size of the xHCI driver.

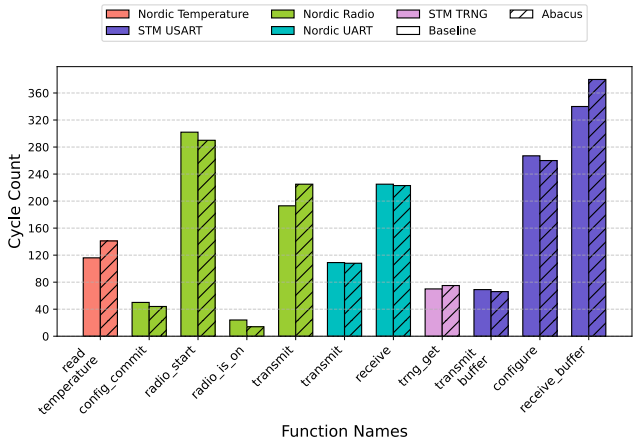


Figure 3. Microbenchmarks of TockOS driver method performance for Abacus integration versus baseline.

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

Table 4. Microbenchmark of Redox xHCI driver methods measured as virtual cycles in QEMU x86_64.

[9] Note that while the measured QEMU cycles might not represent an accurate value compared to real hardware, the relative difference is still informative.

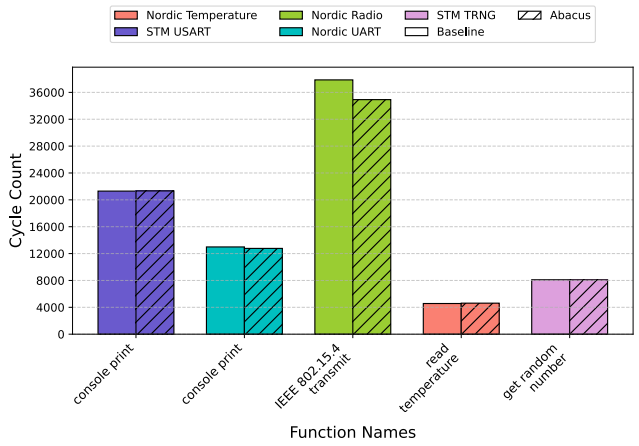


Figure 4. Macrobenchmark Performance (in CPU cycles) demonstrating the Abacus/baseline driver runtime for TockOS functionality implemented by the driver.

Runtime Macrobenchmarks. The microbenchmarks demonstrate that Abacus does not impose an overhead in Redox and imposes some overheads in Tock. We now show, using macrobenchmarks, that these Tock/Abacus overheads are insignificant when considering the total execution time (in CPU cycles) with respect to the kernel's operation. We show this by measuring the total time (in cycles) from when the kernel receives a syscall requesting a hardware operation (e.g. transmit a packet over the radio) to the time in which the kernel enqueues a completion notification for the application. Figure 4 demonstrates this performance measurement for each Tock driver. We see that Abacus imposes minor overheads in 3 of the benchmarked drivers; this overhead is most pronounced in the temperature driver, but still only amounts to a 1.2% overhead. The temperature driver is a very simple driver, and as such, the constant Abacus overhead represents a larger, but still insignificant, proportion to the overall driver's performance. We attribute the Nordic UART driver performance improvement to the Abacus implementation being able to avoid redundant checks. The substantial Nordic 15.4 driver improvement is due to the hardware shortcuts implemented using Abacus (detailed in Section 7.2).

9 Related Work

There is a rich history of approaches to improve driver correctness, ranging from testing, formal verification and DSLs which are complementary to Abacus.

Driver Fuzzing. PrIntFuzz [25], DevFuzz [49], and Fuzzware [41] all leverage fuzz testing combined with hardware simulations and static analysis to improve driver test coverage. These approaches have successfully discovered hundreds of bugs in Linux, FreeBSD, and Windows, providing improvements to driver correctness. However, they are limited by the inherent incompleteness of testing.

Formal Methods for drivers. Driver formal methods can broadly be divided into two categories: hardware-software co-verification approaches and driver synthesis or verification. Hardware-software co-verification [12, 39, 42] introduces specifications and methods to connect the verification completed during hardware design to the specifications of the driver code. On the other hand, driver synthesis and driver verification focus only on the correctness of the driver code itself. Ryzhyk et al. [40] leverages synthesis to generate correct drivers from a specification of what the driver should do. Chen et al. [4] and Pohjola et al. [33] leverage model checking to confirm that the driver implementations match the developer-provided specification. While these approaches can provide strong correctness guarantees, they are either limited to a small set of drivers or require significant developer effort to specify driver behavior.

DSLs for Device Protocols. Work such as DEVIL [27] and NDL [7] implements DSLs to improve driver correctness by allowing developers to specify device protocols, similar to Abacus. From these high-level implementations, they generate low-level code while checking safety properties. However, none of these works take into account the hardware state, rendering device protocols enforcement incomplete.

Type-States. Other works such as SquirrelFS [22] and Zhang and Wang have leveraged type-states to improve correctness in systems software. In SquirrelFS, they combine type-states with persistent memory to provide crash consistency for filesystems. SquirrelFS models persistent memory hardware using type-states, but does not consider hardware-initiated state transitions as persistent memory hardware operations are synchronous. Abacus type-states can model both synchronous and concurrent hardware. Zhang and Wang implement type-states in concurrent contexts; however, their state machine only models the code itself and, as a result, is applicable only to software concurrency. In contrast, Abacus maintains type-state correctness across contexts in which hardware has primary custody of the state machine, supporting hardware-software concurrency.

Rely-Guarantee Logic. Owicki and Gries first presented a methodology to formally reason about concurrent programs; this approach, however, required globally reasoning about each concurrent component's assertions. Rely-Guarantee (R/G) Logic improved upon Owicki and Gries to enable modular specifications localized to a given component [17]. More recent works have proposed: new methods to further localize R/G logic to avoid globally specifying shared state [8], applying R/G logic to concurrent garbage collection [19], and extending R/G logic to real-time scheduling [18]. Abacus employees the concepts of R/G logic with type-states

and is the first system to use R/G logic in the context of hardware-software concurrency.

10 Discussion and Future Work

Developer Mental Model Reliance. Abacus assumes the developer provided annotations that specify the device protocol are correct. This relies upon developers correctly interpreting the hardware specification and correctly translating it to the Abacus DSL—mistakes may result in Abacus *enforcing* device protocol violations! Hardware-software co-verification approaches may be used and integrated into Abacus to autogenerate the trusted annotations.

Limited to Rust. Abacus requires the host driver to be implemented in Rust. Although Abacus is implemented in Rust (procedural macros, `AbacusCell`, etc.), the Abacus refinement and Abacus hardware type-states are not Rust specific and could be constructed with any strongly typed language.

Abacus Type-States—Other Uses. We have open sourced and released the Abacus framework as a Rust crate. Beyond device protocols, the ability to model hardware using type-states provides a useful building block for other properties. For instance, power draw is related to the current hardware state. Modeling and tracking hardware states through the type system enables an OS to potentially track, among other things, instantaneous power draw entirely in software.

11 Conclusion

Correct drivers are hard to write because of the significant gap between the interface that hardware provides to software and the specification that hardware expects software to uphold through this interface. Instead of bridging this gap with cautious reasoning by developers, the Abacus framework captures the hardware specifications as a state machine. With the introduction of *stable* and *transient* state classification, it becomes possible to relax type-state's traditional single-owner semantics to one of joint custody with hardware. From this, Abacus can provide a no-overhead abstraction of hardware that enables correct-by-construction driver code. In sum, we find that modern, strongly-typed languages afford opportunities for *pragmatic verification* and allow us to eliminate device protocol violations in OS drivers.

12 Acknowledgments

We thank our shepherd David Cock and the anonymous reviewers for their insights and helpful feedback. This work was supported by the National Science Foundation under grant numbers 2303639, 2443589, 2534083.

References

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

[2] AMD Xilinx. [n. d.]. Zynq UltraScale+ MPSoC Restart solution. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841820/Zynq+UltraScale+MPSoC+Restart+solution. Accessed: Apr 2025.

[3] Brad Campbell. 2018. sam4l: usart: spi: fix callback location. 111bf988a9bad21cbc5420dba20408a01e01b676. Accessed: Aug 2025.

[4] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101

[5] Chips Alliance. [n. d.]. Caliptra - Support for VeeR EL2 with User Mode and Physical Memory Protection in Tock embedded OS. https://www.chipsalliance.org/news/caliptra-support-for-veer/. Accessed: Apr 2025.

[6] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2020. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 14 (Nov. 2020), 82 pages. https://doi.org/10.1145/3417516

[7] Christopher L. Conway and Stephen A. Edwards. 2004. NDL: a domain-specific language for device drivers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Washington, DC, USA) *(LCTES '04)*. Association for Computing Machinery, New York, NY, USA, 30–36. https://doi.org/10.1145/997163.997169

[8] Xinyu Feng. 2009. Local rely-guarantee reasoning. *SIGPLAN Not.* 44, 1 (Jan. 2009), 315–327. https://doi.org/10.1145/1594834.1480922

[9] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[10] Thread Group. 2025. What is Thread? https://www.threadgroup.org/What-is-Thread/Overview. Accessed: August 2025.

[11] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 653–669.

[12] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (Dec. 2018), 24 pages. https://doi.org/10.1145/3282444

[13] Ramla Ijaz. 2024. bug fixes [sic]. https://gitlab.redox-os.org/redox-os/drivers/-/merge_requests/180. Accessed: Aug 2025.

[14] Texas Instruments. 2020. UDMA. https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/07_00_00_11/exports/docs/pdk_jacinto_07_00_00/docs/userguide/modules/udma.html. Accessed: August 2025.

[15] Intel Corporation 2024. *Intel 82599 10 Gigabit Ethernet Controller: Datasheet*. Intel Corporation. Revision 3.5.

[16] Jon Flatley. 2023. Fix RX race condition in lowrisc UART. ab7cbd0d0bc59fd83077991a901d715116e5e896. Accessed: Aug 2025.

[17] C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. https://doi.org/10.1145/69575.69577

[18] Cliff B. Jones and Alan Burns. 2023. Extending rely-guarantee thinking to handle real-time scheduling. *Form. Methods Syst. Des.* 62, 1–3 (Nov. 2023), 119–140. https://doi.org/10.1007/s10703-023-00441-y

[19] Cliff B. Jones and Nisansala Yatapanage. 2019. Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. *Form. Asp. Comput.* 31, 3 (June 2019), 353–374. https://doi.org/10.1007/s00165-019-00482-3

[20] Hyung-Sin Kim, Michael P. Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. 2018. System Architecture Directions for Post-SoC/32-bit Networked Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (Shenzhen, China) *(SenSys '18)*. Association for Computing Machinery, New York, NY, USA, 264–277. https://doi.org/10.1145/3274783.3274839

[21] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[22] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. 2024. SquirrelFS: using the Rust compiler to check file-system crash consistency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 387–404. https://www.usenix.org/conference/osdi24/presentation/leblanc

[23] Leon Schuermann. 2023. litex/uart: fix TX race condition [sic]. d43d5b1c7fd7c78dac1a9724b5c59871df37196e. Accessed: Aug 2025.

[24] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[25] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/3533767.3534226

[26] Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martinez Garcia, and Stefan Mangard. 2025. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In *34th USENIX Security Symposium: USENIX Security 2025*. USENIX Association.

[27] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. 2000. Devil: An IDL for Hardware Programming. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/osdi-2000/devil-idl-hardware-programming

[28] Microchip. 2018. What is SleepWalking? How it Helps to Reduce Power Consumption. https://ww1.microchip.com/downloads/en/DeviceDoc/90003183A.pdf. Accessed: August 2025.

[29] Charlie Miller. 2011. Battery Firmware Hacking: Inside the innards of a Smart Battery. https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf. Accessed: August 20, 2025.

[30] Nazmus Sakibb. [n. d.]. Understanding the Microsoft Pluton security processor. https://techcommunity.microsoft.com/blog/windows-itpro-blog/understanding-the-microsoft-pluton-security-processor/4370413. Accessed: Apr 2025.

[31] Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Inf.* 6, 4 (Dec. 1976), 319–340. https://doi.org/10.1007/BF00268134

[32] Oxidos Automotive. [n. d.]. Rust-based secure ecosystem for safety critical automotive ECUs. https://oxidos.io/. Accessed: Apr 2025.

[33] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems* (Koblenz, Germany) *(PLOS '23)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3623759.3624544

[34] Redox OS Project. 2025. The Redox Operating System: System Design. https://doc.redox-os.org/book/microkernels.html. Accessed: August 18, 2025.

[35] Redox OS Project. 2025. Redox OS. https://www.redox-os.org. Accessed: August 1, 2025.

[36] Ramla Ijaz. 2023. Invalid/ Missing register writes in ixgbe driver [sic]. https://gitlab.redox-os.org/redox-os/drivers/-/issues/38. Accessed: Aug 2025.

[37] Rust Foundation. [n. d.]. The Rust Reference: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html. Accessed: Apr 2025.

[38] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 275–288. https://doi.org/10.1145/1519065.1519095

[39] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij, and Gernot Heiser. 2011. Improved device driver reliability through hardware verification reuse. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1950365.1950383

[40] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 661–676. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ryzhyk

[41] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1239–1256. https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[42] Daniel Schwyn, Zikai Liu, and Timothy Roscoe. 2025. Efeu: generating efficient, verified, hybrid hardware/software drivers for I2C devices. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 76–93. https://doi.org/10.1145/3689031.3696093

[43] Nordic Semiconductor. 2024. PPI - Programmable peripheral interconnect. https://docs.nordicsemi.com/bundle/ps_nrf52840/page/ppi.html. Accessed: August 2025.

[44] IEEE Spectrum. 2017. Chip Hall of Fame: Western Digital WD1402A UART. https://spectrum.ieee.org/chip-hall-of-fame-western-digital-wd1402a-uart. Accessed: August 20, 2025.

[45] Jeff Vander Stoep and Alex Rebert. 2024. Eliminating Memory Safety Vulnerabilities at the Source. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html. Accessed: August 20, 2025.

[46] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[47] Tock Operating System. 2025. Tock Registers Crate. https://crates.io/crates/tock-registers. Accessed: August 2025.
[48] The Rustonomicon. [n. d.]. Exotically Sized Types. https://doc.rust-lang.org/nomicon/exotic-sizes.html. Accessed: Apr 2025.
[49] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DevFuzz: Automatic Device Model-Guided Device Driver Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. 3246–3261. https://doi.org/10.1109/SP46215.2023.10179293
[50] Lu Zhang and Chao Wang. 2014. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2610384.2610405

A The `AbacusCell` Type

The AbacusCell leverages interior mutability to allow moving type-stated registers within `&self` driver methods. Our UART driver includes an `AbacusCell` field (which in turn contains our type-stated `UartRegister` **`struct`** wrapped in our `StateEnum`). The design is inspired by a series of similar custom `Cell` types provided by Tock OS's `tock-cells` crate that address similar interior mutability needs.

To interact with hardware registers, drivers must use the `AbacusCell` `map` method which moves the `StateEnum` into the provided closure. In practice, this `map` method 'takes' the `StateEnum` from the `AbacusCell` type; any attempt by other threads to call this `AbacusCell` `map` will fail. Finally, the `map` method requires the `StateEnum` be returned from the closure and replaces the `StateEnum` within the `AbacusCell`.

In short, the `AbacusCell` allows for interacting with hardware across multiple threads (e.g. through a shared reference to the driver object) while preserving Abacus' requirement for moving type-stated register `structs`. Moreover, this serves a dual benefit of preventing race conditions between threads that perform register operations and state transitions.

B Artifact Appendix

B.1 Abstract

The Abacus framework encodes device protocol state machines and constraints into the Abacus DSL and statically prevents device protocol violations using type-state programming. This artifact consists of three repositories: our research prototype integrated into a RedoxOS xHCI driver, our research prototype integrated into 5 TockOS drivers across two hardware manufacturers, and a Rust crate (`abacus-registers`) that provides the Abacus framework as a standalone library.

B.2 Description

B.2.1 How to access.

- RedoxOS xHCI Driver Integration
- TockOS Driver Integration
- Abacus Rust Crate: `abacus-registers`