|galois|

# Verifying seL4 Device Drivers with CN

Gus O'Hanley

Sep 2 2026

## Why bother?

- We expect new drivers to be verified, using seL4 Device Driver Framework (SDDF)
- But there are many legacy drivers written in C
  - such as libethdrivers
- In our work with Dornerworks, we verified an Intel i210 driver
  - https://github.com/seL4/util_libs/blob/master/libethdrivers/src/plat/pc99/intel.c
- Same approach can be used for other device drivers, helping industry deploy verified drivers with seL4

## How *should* we verify a driver?

- Have a specification for the device
  - For any commercial device, unlikely
- Have a specification for what the driver should do
  - “work”
- Prove that the driver controls the device so that that happens
  - Difficult without either of the above

## How *can* we verify a driver?

- What should the driver do?
  - Program the device in the manner described in the datasheet
- Does it do that?
  - We need something we can compare with our program

## How *do* we verify a driver?

- Create a model of the device
  - Read the device datasheet
    - Example interactions
    - Register descriptions
  - Look at the manufacturer's drivers
    - Often help clarify ambiguity
- Prove that the driver correctly configures and uses the device
  - Fix bugs along the way

## The tool: CN is a type system for C

- CN is a first-order separation logic refinement type system for C
- If CN verify accepts a function spec, there is no undefined behavior in that function
  - If the precondition holds and the specs of all called functions are accurate
- CN *never guesses*, annotations for all choices
  - No surprises
- based on the cerberus C semantics
  - However, no support for concurrency or parallelism

## Relevant CN semantics

- Logical functions
- Predicates
  - Can express memory ownership, do not need to
- Specifications
  - Use predicates and logical functions
  - Describe the behavior of a C function

## The device: Intel i210 NIC

- An ethernet card
- Many, many features, modes, and configurable behaviors.
- We specify what's necessary for the simplest usage
  - 1 RX queue
  - 1 TX queue
  - No split packets
  - Legacy descriptor formats
  - **Configuration necessary to enforce this**

## We have one predicate for the I210

`I210(pointer p)`

- Owns the registers `*p`
- Owns whatever the values of those registers require
  - For us, TX/RX queues
- Owns whatever *those* require
  - For us, buffers described by a certain range of TX/RX descriptors

## Problem: The i210 is not actually part of the program

- The device communicates over MMIO and DMA
- The program controls when MMIO happens
- The device controls what is read and when DMA happens

## Solution: We don't know the specific device state

We package all semantics into a relation `HardwareStep(I, O)`

`O` is in the reflexive and transitive closure of all actions the hardware can take from `I`

"`I` can turn into `O`"

We only specify outputs up to `HardwareStep`

## Reading and writing can be CN functions

```
I210Read(s, r)
```

```
I210Write(s, r, v)
```

These compute the resulting hardware state after a read or write.

They can fail, indicating an unsupported or unspecified result.

## Those are logical functions, we still need specs

So we need

- `RegWriteRequires`
- `RegWriteEnsures`
- `RegReadRequires` (In our model, nothing)
- `RegReadEnsures` (In our model, nothing)
- `HardwareStepEnsures`
- NOT `HardwareStepRequires`, `I210()` and `Reg*Requires` are designed to ensure this

# Actually doing writing must be a C function

Solution: we make C functions `reg_read` and `reg_write` that do the operation

```
spec reg_read(pointer reg_base, u32 offset);
    requires take I_state = I210(reg_base);
             offset_to_register(offset) != NothingReg{};
    ensures take O_state = I210(reg_base);
            let result = I210Read(I_state, offset);
            return == result.value;
            result.state == O_state;
```

## We still need to allow the hardware to act

Solution: we use C functions `reg_read_step` and `reg_write_step` that do the operation and then a hardware step

```
spec reg_read_step(pointer reg_base, u32 offset);
    requires take I_state = I210(reg_base);
             offset_to_register(offset) != NothingReg{};
    ensures take O_state = I210(reg_base);
            let result = I210Read(I_state, offset);
            return == result.value;
            I210HardwareStep(result.state, O_state);
            take ens = HardwareStepEnsures(reg_base, result.state, O_state);
```

## We don’t need to trust these functions

Their meaning is completely constrained by `I210`, `I210Read`, `I210Write`, and `HardwareStep`.

```
void definition_of_HardwareStepEnsures(i210 *p)
/*@
requires take I = I210(p);
ensures take O = I210(p); HardwareStep(I, O);
  take E = HardwareStepEnsures(p, I, O);
@*/
{ /* nothing! */ }
```

## We must trust our interpretation of the datasheet

- We also use the example interactions and programmer guide sections to produce test traces
  - These test traces are just functions that use the model
  - But we consider them part of the definition of the model
- We can write hardware steps for single actions (RX alone, TX alone, etc) and then verify that our `HardwareStep` relation is defined by them
  - Easier to be confident in each alone

## We must trust our model, but we can check it

`hw_read_ok`, `hw_write_ok`: update a runtime model of the hardware with the operation, returning whether this is a possible execution in our abstract model

## To summarize

- We model any MMIO device as
  `predicate DeviceState Device(pointer p)`
- Memory reads and writes as explicit CN functions paired with implementing C functions
- A hardware step relation expresses semantics and all nondeterminism
- Derived properties of these
- A monitor that checks the hardware against the semantics

## Results and Next steps

- Verified safety of all i210 device driver functions
  - (sddf notify specs exist but are trusted)
  - proofs run in 2 hours without caching, very parallelizable
  - Have not verified full functional correctness
- Found 2 bugs in the driver
  - one low severity
  - one medium severity
- Transitioning this approach to Dornerworks, so this approach can be scaled to other drivers in industrial use
- The same approach can be used for Rust drivers (Microkit)!
  - instead of CN use Verus

## Those two bugs

- The driver never checks if the TX queue is ready
  - “Low”? It does check if RX is ready, and it’s likely TX would complete before RX if it started before
- The driver does not check if a device reset is complete correctly
  - You need to check a bit in a different register, it checked the bit that starts the reset
  - “Medium”? It waits 3ms so it’s probably fine, and…
  - The freebsd driver checks in a different way and if it isn’t ready in 10ms it just continues, so it’s probably fine