Lecture 16: Embedded OS

CE346 – Microcontroller System Design

Branden Ghena – Fall 2025

Northwestern

Administrivia

- See you all tomorrow for project check-ins

- Last quiz this coming Tuesday
  - I promise to grade Quiz 3 this weekend. Sorry about the delay

Today's Goals

- Introduce embedded OS concerns and how they are different from general-purpose computing, particularly in security.
- Describe Tock Operating System goals and design
- Introduce Rust programming language
- Explore example OS device driver stackup from Tock
- Sidebar: I promise not to quiz you on specific Tock details
  - Although I might ask you about embedded OS in general

General Tock resources

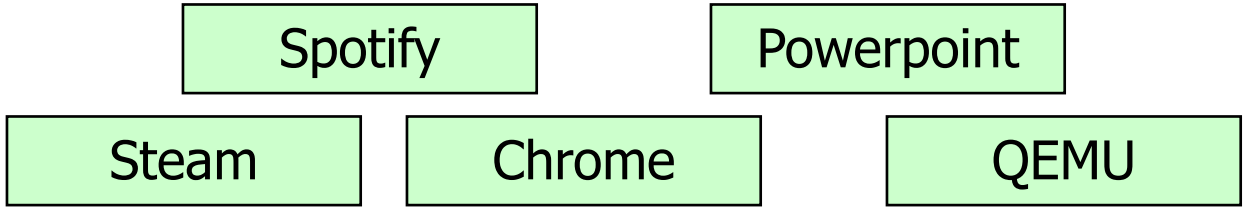
- https://github.com/tock/tock
- https://www.tockos.org/
- **"Multiprogramming a 64 kB Computer Safely and Efficiently"** Levy et al. 2017. Symposium on Operating Systems Principles. https://brandenghena.com/projects/tock/levy17multiprogramming.pdf
- **"Tock: From Research to Securing 10 Million Computers"** Schuermann et al. 2025. Symposium on Operating System Principles. https://brandenghena.com/projects/tock/2025-sosp-tock-decade.pdf

Outline

- **Embedded Operating Systems**
- Embedded Security Challenges
- Tock Overview
- Rust Programming
- Tock Driver Design

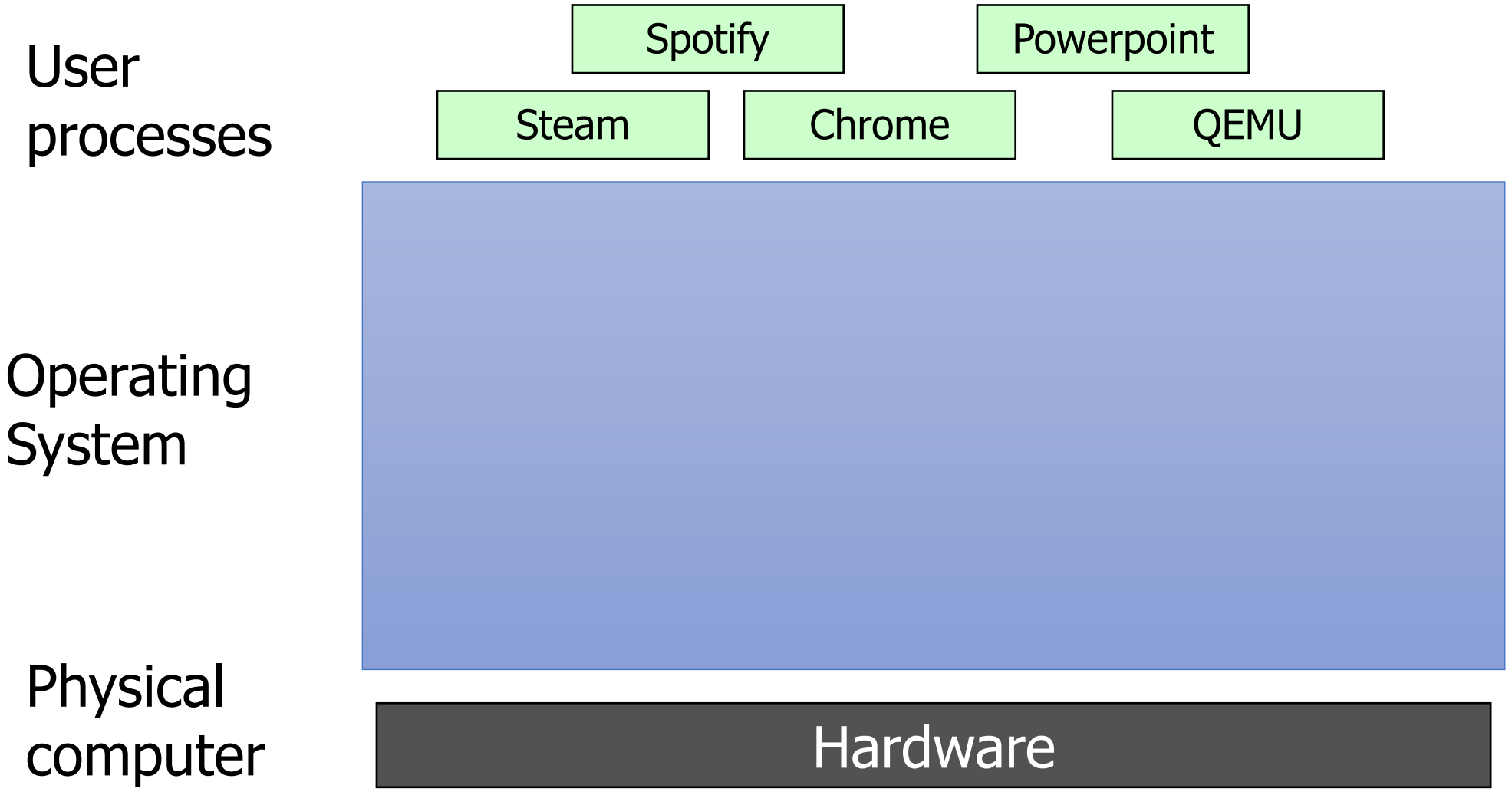
What is an operating system?

User processes

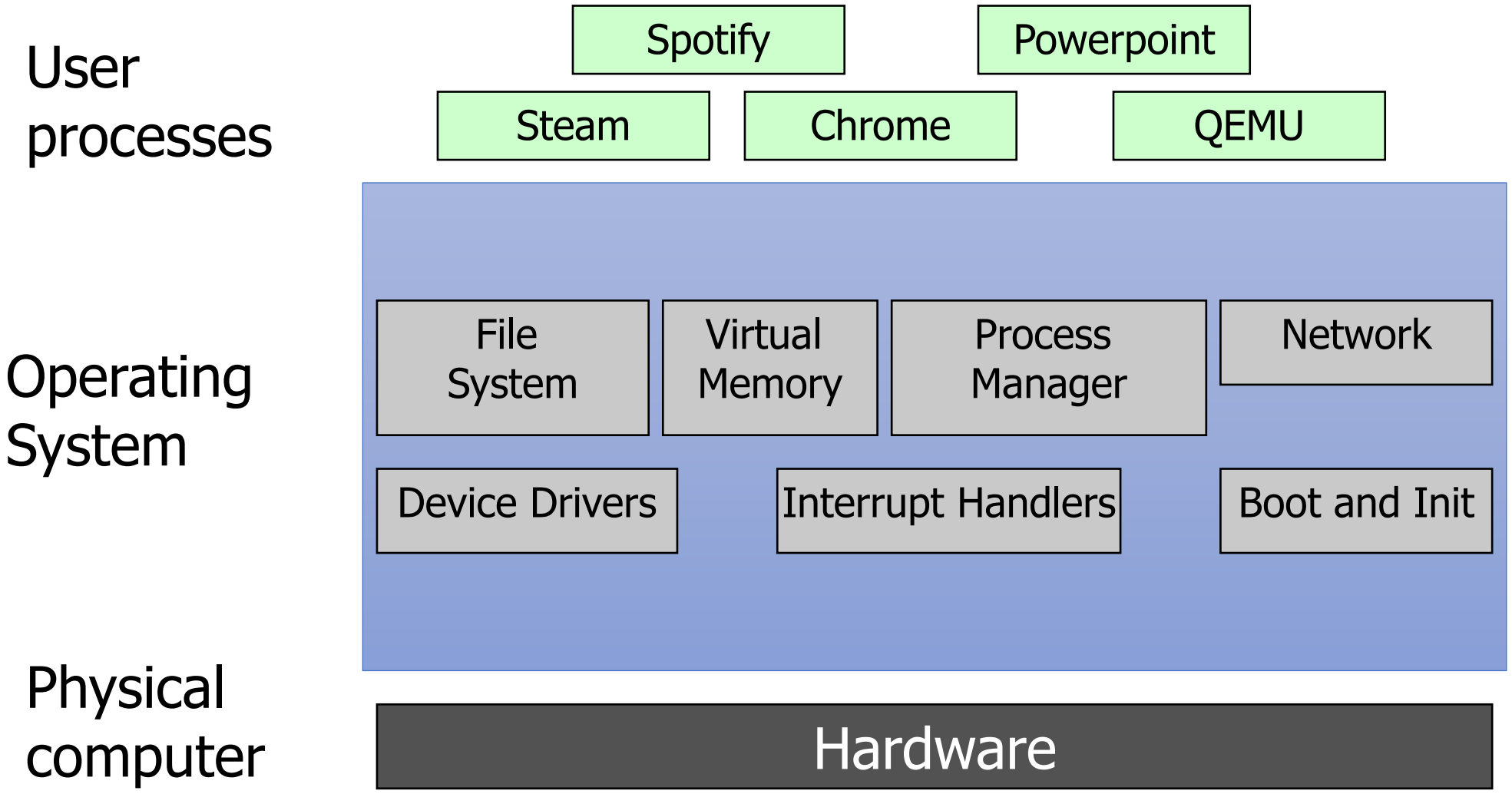


Physical computer

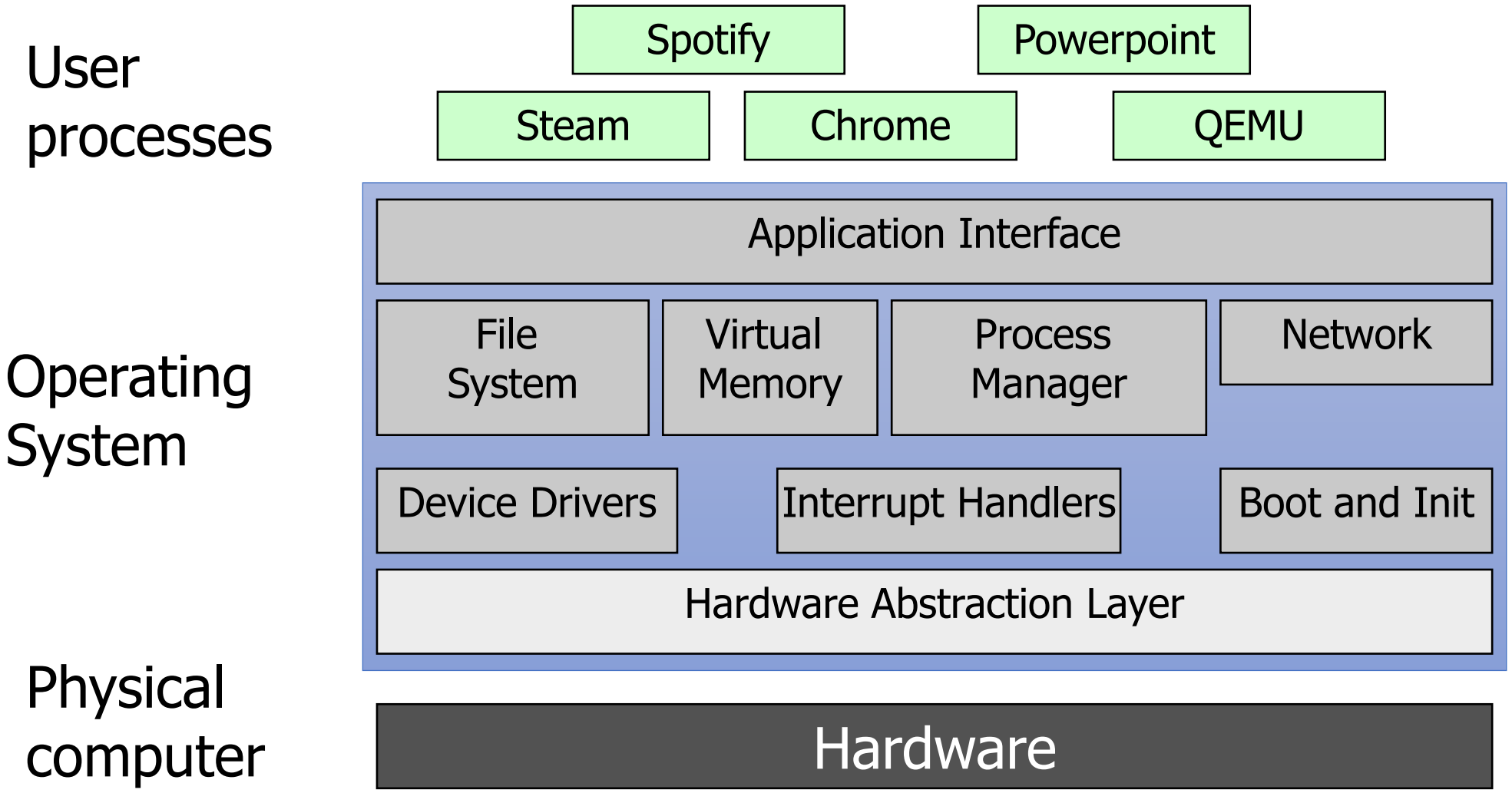
What is an operating system?



What is an operating system?



What is an operating system?



Operating systems goals

- OSes make advancing technology available to rapidly evolving applications. They do so with two major goals:

  1. Provide **abstractions** to applications to enable hardware compatibility
     - Why: allow reuse of common features, avoid low-level details
     - Challenges: What are the correct abstractions?

  2. Manage **sharing of resources** across many applications
     - Why: protect applications, enforce fair access
     - Challenges: What are the mechanisms and what are the policies?

- Good operating systems do these quickly, efficiently, and securely

How does this differ from what we've been doing?

- Often our application and driver logic were coordinated and in some cased co-mingled
  - We didn't consider "sharing" of resources between multiple stakeholders

- We did abstract away interfaces to hardware
  - But we didn't consider how to do so in a way that allowed hardware to be interchangeable

Userspace versus kernel distinction

- Kernel has full control over the system
  - Manages resources
- Userspace has partial control (whatever the kernel allows it)
  - Uses resources granted to achieve some goal
- When userspace wants access to something, it needs to request it from the kernel
  - Uses “system calls” to do so
  - Trigger a software interrupt, with some values to describe what it wants
    - Kernel takes action (or denies action) and returns back to application

Linux system calls

- Example system calls
  - https://man7.org/linux/man-pages/man2/syscalls.2.html

Number	***Name***	***Description***
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Embedded needs its own operating systems

- Can’t use normal Linux!
  - Too little memory and processing in embedded hardware
  - Too much concern about power in embedded applications
  - Microcontrollers don’t have the necessary hardware features (virtual memory)

- Important: Linux is *never* going to be the solution
  - Capabilities of general computers are orders of magnitude better
  - Embedded systems are gaining more capabilities
  - But new lower-power, lower-cost systems keep emerging too

Linux is useful for many edge devices though

- Raspberry Pi is a great example

- If your device is powerful enough to run Linux, maybe that's a good idea!
  - Essentially gives you a normal computer platform to run stuff on
  - Easier to do big things: Webcam -> Python -> OpenCV

- But there will always be smaller, cheaper, less capable devices where Linux doesn't make sense
  - Or even bigger, capable systems that need to be precisely controlled

A variety of embedded OSes exist with different designs

- TinyOS (2000-2012-ish) was designed to support extremely low-capability devices with 2 KB of RAM total

- FreeRTOS (2003-today) was designed to real-time tasks with a very small kernel, around 8 source files

Zephyr (2016-today)

- Linux Foundation embedded OS project
  - Developed in C
  - Lots of industry buy-in

- Features (in their words)
  - Extensive suite of kernel services
    - Multi-threading, memory allocation, IPC, power management, etc.
  - Multiple scheduler choices
  - Highly configurable and modular
  - Cross architecture

Outline

- Embedded Operating Systems
- **Embedded Security Challenges**
- Tock Overview
- Rust Programming
- Tock Driver Design

Two needs not met by traditional embedded OSes

1. Security
   - Protect the core platform from applications

2. Multiprogramming
   - Run multiple, unrelated applications (securely)

Embedded devices are a weak link

- Custom, application-specific code written in C
  - Limited code-reuse
  - Low testing coverage

- All code on the system is trusted
  - No isolation: any code can directly access hardware registers
  - Little distinction between “kernel” and “application”

Weak devices provide network entry points

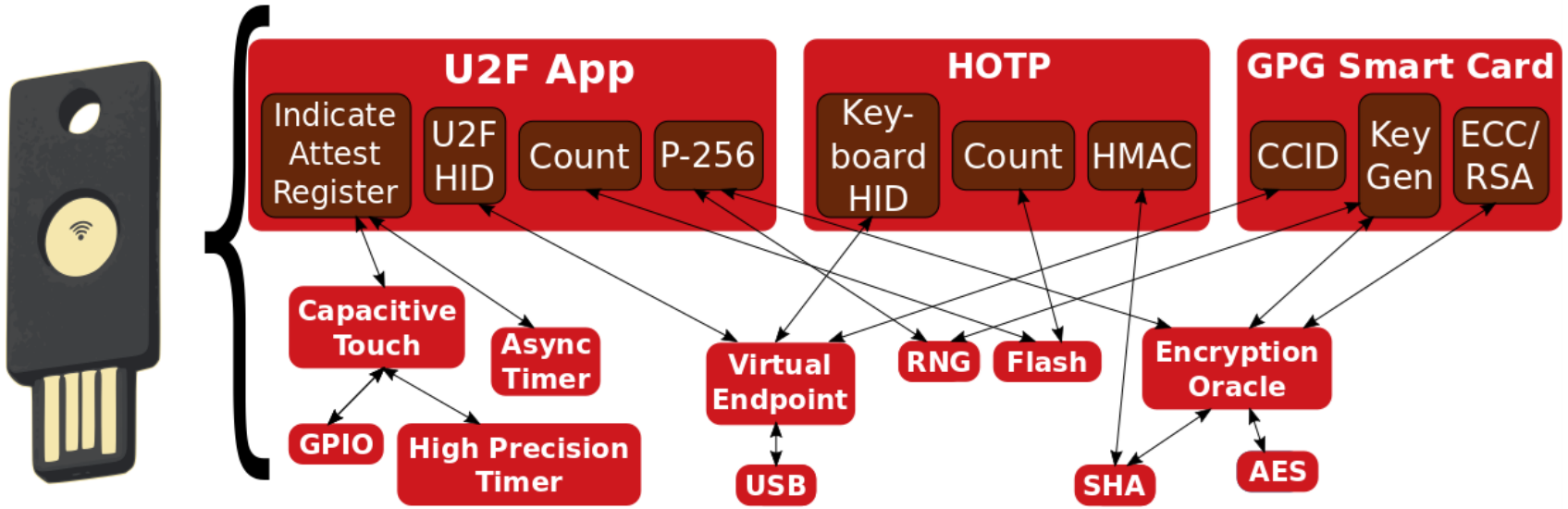
- IoT devices can be used as a network entry point
  - Step one: hack the device
  - Step two: use the device to access information on the private network

- Example: casino high-rollers database obtained through an IoT fish tank thermostat
  - https://interestingengineering.com/a-casinos-database-was-hacked-through-a-smart-fish-tank-thermometer

- See also: Mirai botnet

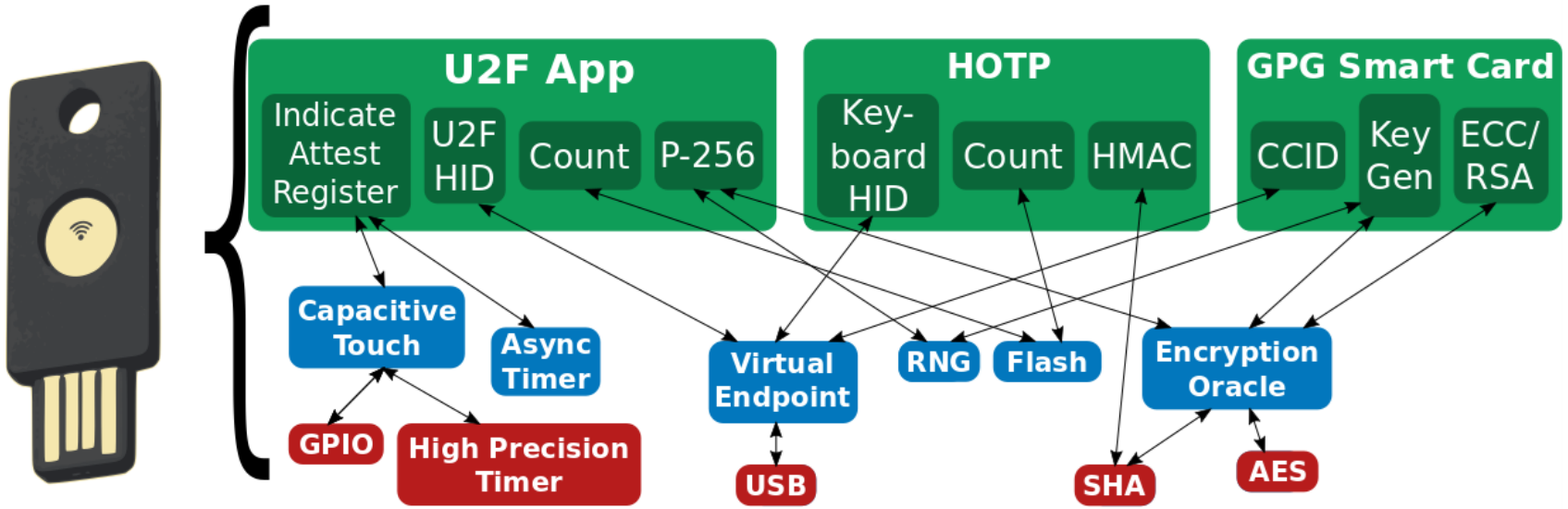
Modern systems increasingly need support for multiprogramming

- Example: USB authentication key
  - Universal Second Factor (U2F)
  - HMAC One-time Password (HOTP)
  - GPG Key (GNU Privacy Guard)



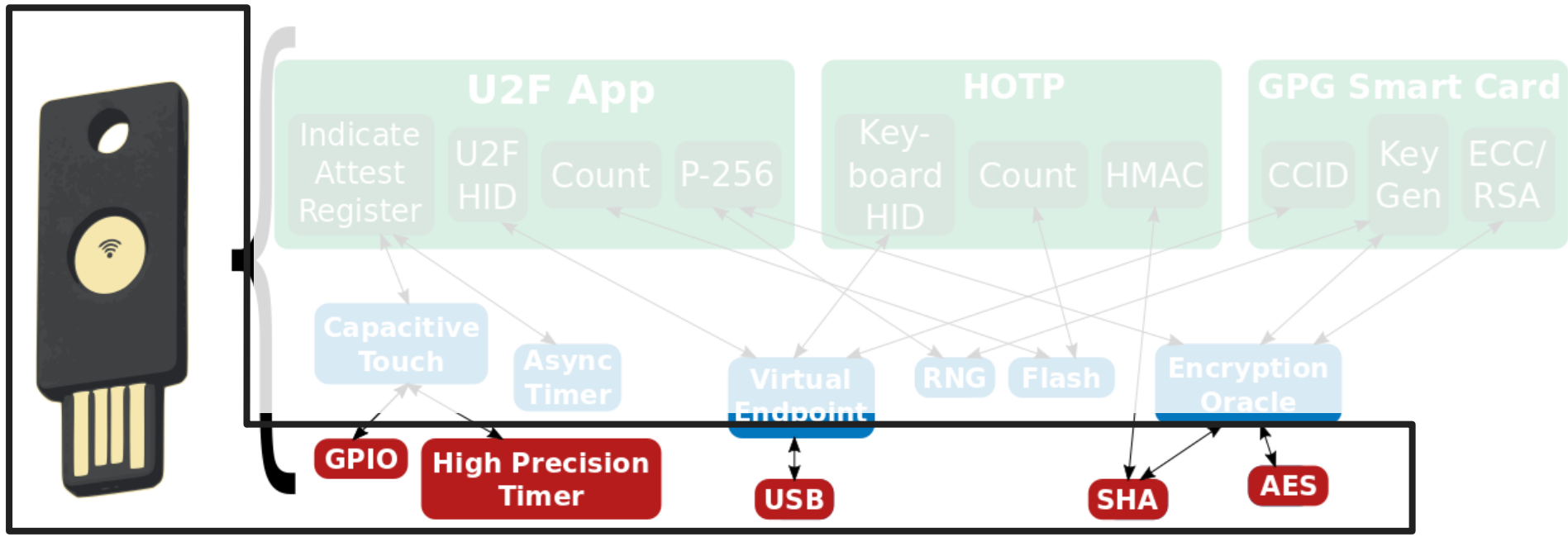
Security layering is desirable

- Different domains with different expectations
  - Applications, services, and platform



Platform layer

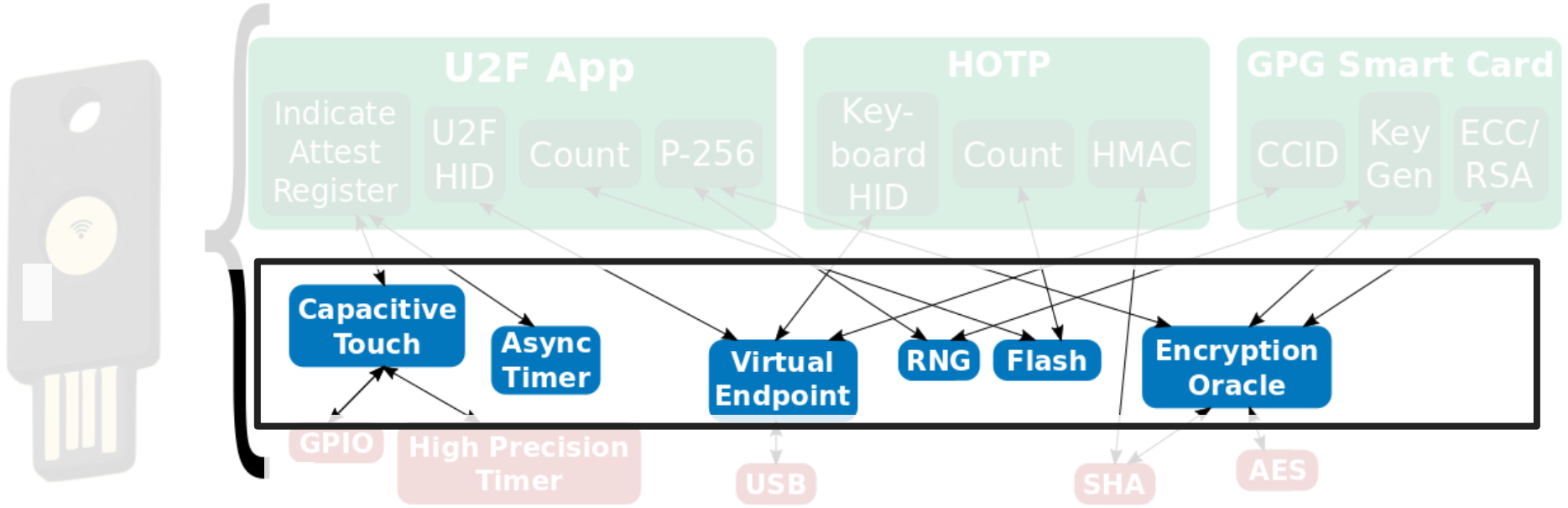
- Core kernel plus microcontroller-specific code
  - ~10 developers
  - Trusted compute base



Goal: possible to correctly and safely extend

Services layer

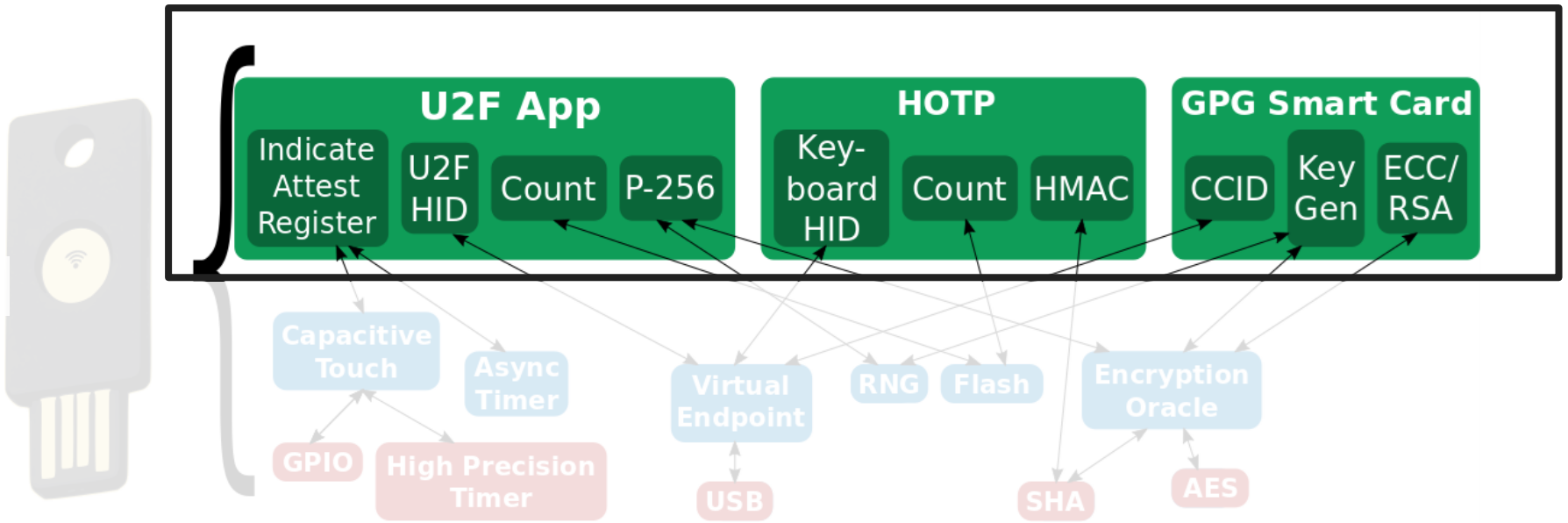
- Device drivers, networking, libraries
  - ~100 developers
  - Auditable, but possibly still buggy



Goal: protect kernel from safety violations

Applications layer

- End-user functionality
  - ~1000 developers
  - Third-party applications, potentially malicious



Goal: end-users can install 3rd-party apps

Break + Question

- What are challenges for updating IoT device firmware?

Break + Question

- What are challenges for updating IoT device firmware?

1. Manufacturer needs to create update (still exist AND care)
   - Software is usually closed source, so others can't update it

2. Update needs to get to the device
   - Either the device needs an internet connection to check
   - Or consumer needs to be aware of update and have a mechanism to upload it

Outline

- Embedded Operating Systems
- Embedded Security Challenges
- **Tock Overview**
- Rust Programming
- Tock Driver Design

Tock

- Embedded OS designed for secure multiprogramming
  - Kernel written in Rust programming language
  - Applications written in any language
  - Runs on multiple hardware platforms

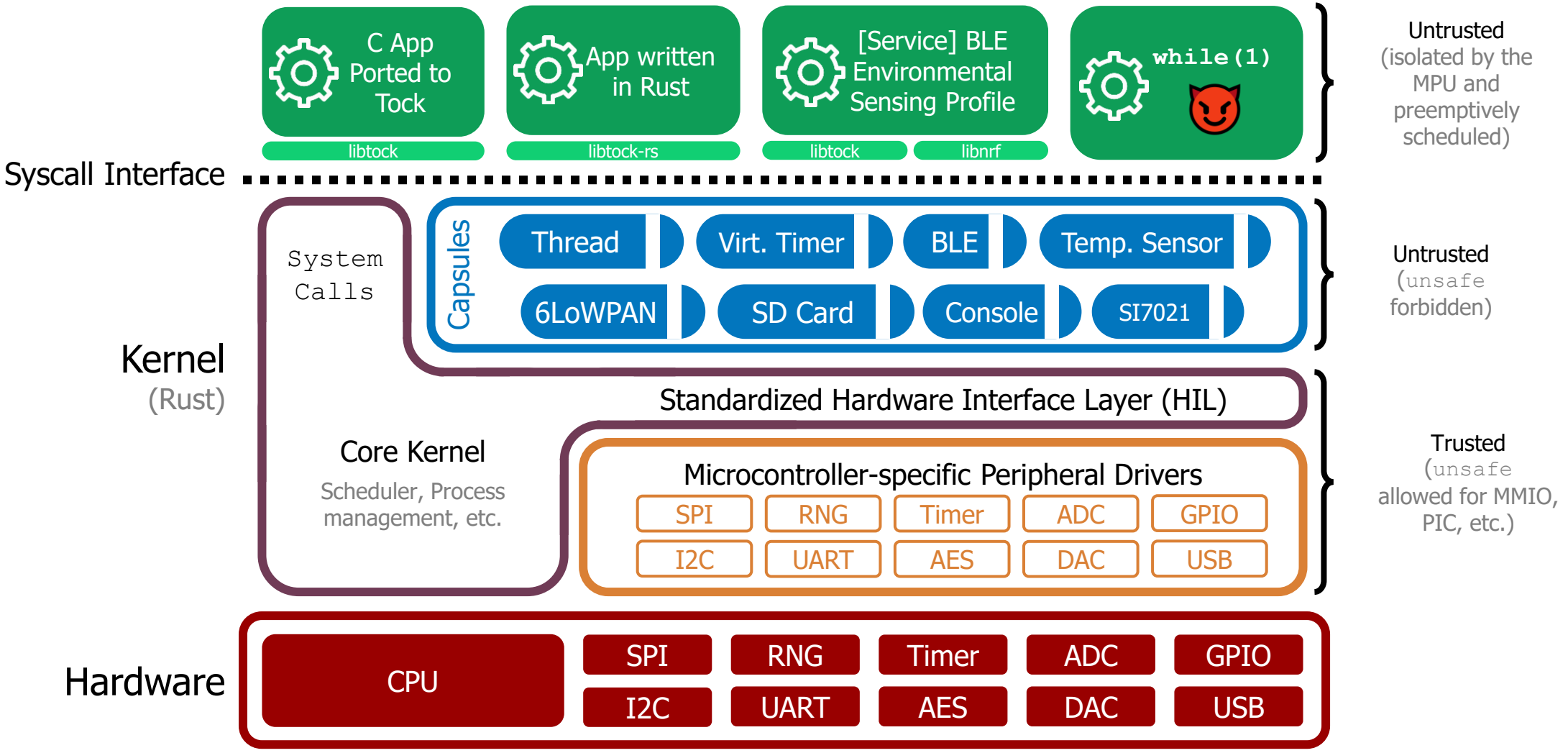
- Open-source research project
  - 2015 collaboration between Stanford, UC Berkeley, and Michigan
    - Since expanded to 6-7 universities (Princeton, UCSD, **Northwestern**)
    - Plus several companies (Google, Western Digital)

- https://github.com/tock/tock
- https://tockos.org/

Tock Features

- A **preemptive** embedded OS (runs on microcontrollers)
  - ARM Cortex-M and RISC-V supported now
- Has separate **kernel** and **user space**
  - Most embedded OS have the one piece software philosophy
  - Runs untrusted apps in user space
  - Uses memory protection on applications (hardware support required)
- Kernel (and drivers) written in **Rust**
- Apps written in C/C++ or Rust (any language is possible)

Tock software organization



Tock's isolation mechanisms

Totally untrusted

Trusted for liveness, not safety

Processes

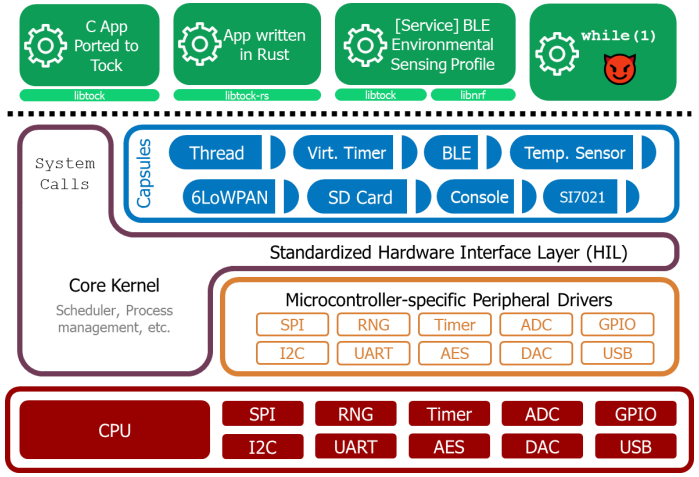
- Standalone executable in any language
  - C, C++, Rust, Lua
- Isolation enforced at runtime
- Higher overhead
- Applications

Capsules

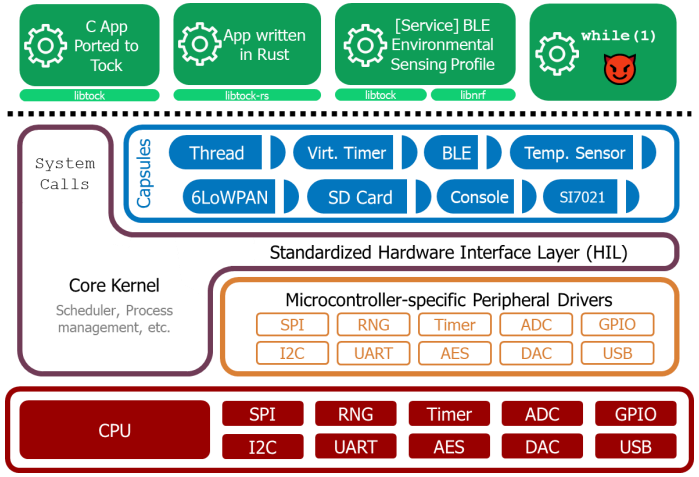
- Rust code linked into kernel
- Isolation enforced at compile-time
- Lower overhead
- Used for device drivers, protocols, timers...

What does the Tock core kernel include?

- Managing Processes
  - Loading, stopping
- Scheduling
  - Determining which process is running at which time
  - Handling hardware events
- Memory Management
  - Mostly giving chunks to processes
  - Also memory protections
- Syscalls
  - Connecting processes to device drivers



Device drivers

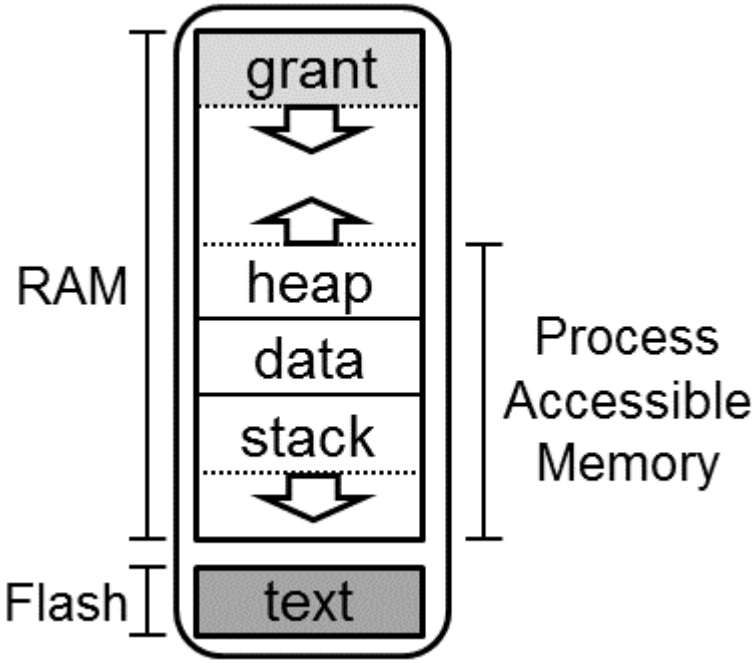


- Much larger percentage of the code in Tock

- Split into groups
  - **Capsules** are microcontroller-agnostic (above the core kernel)
    - Syscall drivers connect syscalls to specific devices
    - Control for generic systems: GPIO, LED, Timers, etc.
    - External devices: connected over buses
      - I2C: Temperature, Pressure, Humidity, Light, IMU
      - SPI: Screen, Memory
  - **Chip drivers** are microcontroller-specific (below the core kernel)
    - Implement various peripherals within those microcontrollers

Userspace applications

- Written and compiled entirely separately from the kernel
  - Load the kernel into flash
  - Individually write each application to flash as as well
- Written in any language
  - C is the major one, Rust also big
  - C++ and Lua exist
- System calls to the kernel to make device requests



How do applications access devices?

- System calls are used to access devices
  - All device interactions are non-blocking (start now, finish later)

- Three generic syscalls
  - Command – takes a 32-bit numerical argument
  - Allow – takes a pointer to a buffer to read/write
  - Subscribe – takes a pointer to a function to callback

- First two arguments to all syscalls
  1. Driver number (the driver it wants to interact with)
  2. Minor number (driver-specific identifier)

Example: console capsule (driver number: 1)

- Command syscall values
  1. Transmit from buffer, argument is length
  2. Receive into buffer, argument is length
  3. Cancel request

- Allow syscall values
  1. Pointer to buffer for sending
  2. Pointer to buffer for receiving

- Subscribe syscall values
  1. Send complete handler
  2. Receive complete handler

Writing bytes to console:

1. uint8_t buffer[20] = {…data…}
2. Allow(1, 1, buffer)
3. Command(1, 1, 20)

Event-driven programming

- Tock adopts an event-driven model to avoid concurrency issues
  - Single core system, so the source of concurrency is interrupts
  - Exposed to applications through “subscribe” callbacks

- Callbacks never occur during normal operation
  - Even if application is descheduled due to timeslice

- Additional syscall: yield – no arguments
  - Application pauses until a callback is ready for it
  - Once one or more events are ready
    - Call the callback handlers, one at a time
    - Return from yield statement

Yield example

```
void sensor_callback (int value) {
  printf("Got sensor reading %i\n", value);
}


int main () {
  sensor_register(sensor_callback); // calls allow
  sensor_sample(); // calls command
  yield();
}
```

Real Tock applications

- Practically, these syscalls are wrapped in library code rather than exposed raw like that
  - Except for yielding

- Libtock-C design (C userspace)
  - Libtock – async libraries
  - Libtocksync – sync libraries (built using async libraries)

Example Tock application

- https://github.com/tock/libtock-c/blob/master/examples/tests/microbit_v2/main.c

Outline

- Embedded Operating Systems
- Embedded Security Challenges
- Tock Overview
- **Rust Programming**
- Tock Driver Design

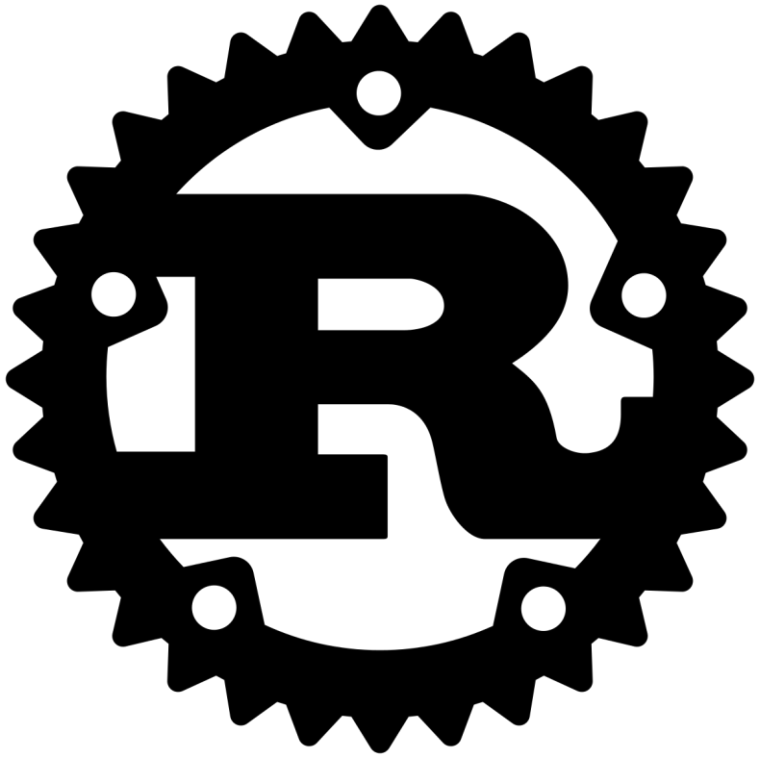
Tock threat model

- Kernel security
  - Secrets **may not be accessed** by applications or capsules.
  - Applications and capsules **may not modify kernel data except through exposed APIs.**

- In normal operating systems, drivers are run in kernel mode
  - Full access to memory and hardware on the system

- So how do we do better than that without moving drivers into userspace applications

Tock solution: use language features



- Tock uses the Rust programming language for the kernel

- Systems language
  - Represents how hardware actually interacts
  - Strong type system and memory safety
  - Runtime behavior similar to C

- Result: capsules *cannot* access memory they do not own
  - Cannot access application secrets
  - Cannot modify other kernel structures (even by accident)

Basic Rust syntax

```
let a: i32 = 1;
```

- Creates a variable a which is a 32-bit signed integer. Sets value to 1

```
let mut b: u8 = 0;
```

- Creates a variable b which is an 8-bit unsigned integer. Sets value to 0
- Mutable variables can be modifed later in code

```
fn add2(x: f64, y: f64) -> f64 {
    // implicit return (no semicolon)
    x + y
}
```

- Function that takes two double arguments and returns a double

Rust Structs and Impl

```rust
// Define a struct named 'Point' with fields 'x' and 'y'
struct Point {
    x: f64,
    y: f64,
}

// Implementation block for the 'Point' struct
impl Point {
    // Define a method that calculates the distance between two points
    fn distance_to(&self, other: &Point) -> f64 {
        // Calculate the distance using the Euclidean distance formula
        ((self.x - other.x).powi(2) + (self.y - other.y).powi(2)).sqrt()
    }
}
```

Later in code:

```rust
let distance = point1.distance_to(&point2);
```

What problems is Rust trying to solve?

- Memory lifetime
  - Use-after-free
  - Common example: pass a reference into a function, then free it
- Data races
  - Multiple access to data and at least one modifies it
  - Problems we were solving with locks
- Generally: have the compiler handle for you whatever it can

Rust has a strong notion of “ownership”

- Ownership is a notion of managing memory and sharing.

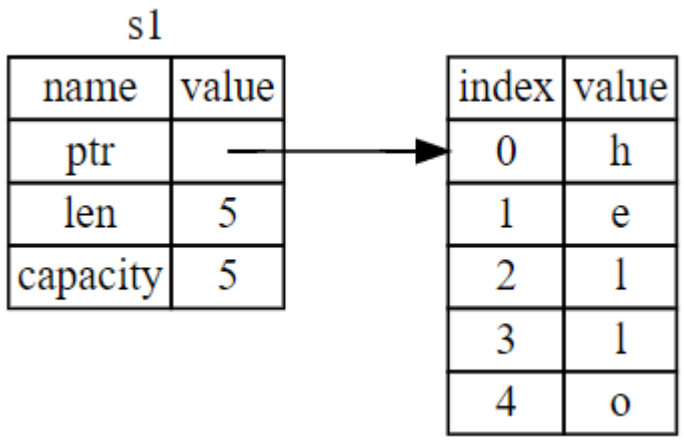
- Rust ownership rules
  - Each value has a variable that’s called its owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value is dropped.

- Last one is straightforward:
  - Values on the stack go away at the end of the function (or any block { })
  - This *lifetime* works just like C or C++

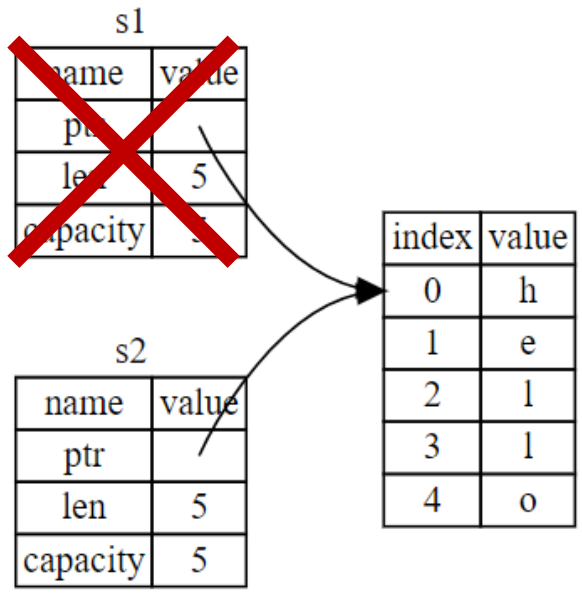
Example: ownership with strings

- Rust ownership rules
  - Each value has a variable that's called its owner.
  - There can only be one owner at a time.

```
let s1 =
String::from("hello");
```




```
let s1 =
String::from("hello");
let s2 = s1;
```



Ownership is transferred through function calls

```
fn main() {
  let s = String::from("hello"); // s comes into scope
  takes_ownership(s); // s's value moves into the function...

  // s is no longer valid here

} // Here, s goes out of scope. But because s's value
  // was moved, nothing special happens.


fn takes_ownership(some_string: String) {
  // some_string comes into scope
  println!("{}", some_string);

} // Here, some_string goes out of scope and `drop` is called.
  // The backing memory is freed.
```

My feelings about writing code in Rust

- Three steps of Rust acceptance

1. You are confused about little weird syntax things.

2. You are frustrated by so many compilation errors.

3. You realize your code mostly works if it compiles.

Tock downside: all drivers *must* be written in Rust

- Why hasn't *every* OS taken the "language support" path?
  - Primarily, it wasn't really available. C or C++ were the only real options.

- Also, the vast majority of developers know C but not Rust
  - And the vast majority of existing code is in C and not Rust

- One of Tock's major challenges is that things like networking stacks need to be re-written in Rust
  - BLE, 802.15.4, WiFi

Sidebar: the future of Rust

- Rust is still growing in popularity but still fairly young
  - Many people are excited to have a safe alternative to C
  - But there is a steep learning curve to using Rust
  - And a vast body of existing code already in C

- How dominant would it have to be to switch our curriculum?
  - Unclear. We've taught C for decades.

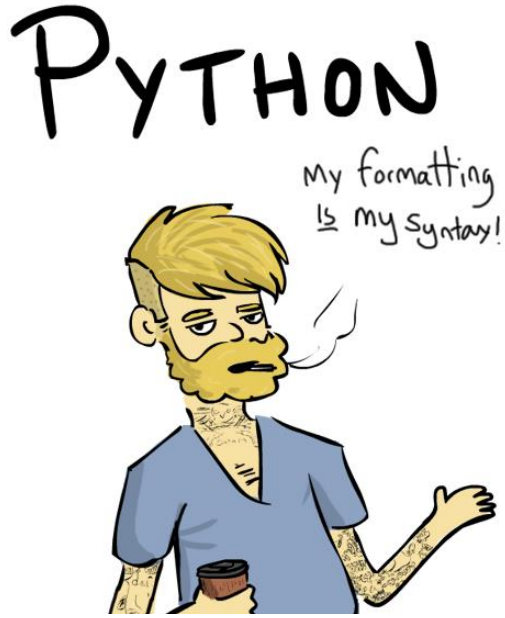
  - Certainly considering it for CS211, but no immediate plan to change
    - Still can't *stop* teaching C (too useful)
    - Rust in 5 weeks at the end of CS211 would be much harder than C++

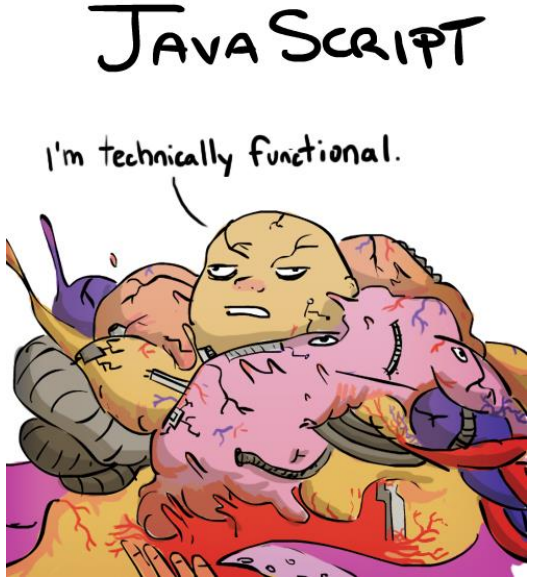
Learning Rust

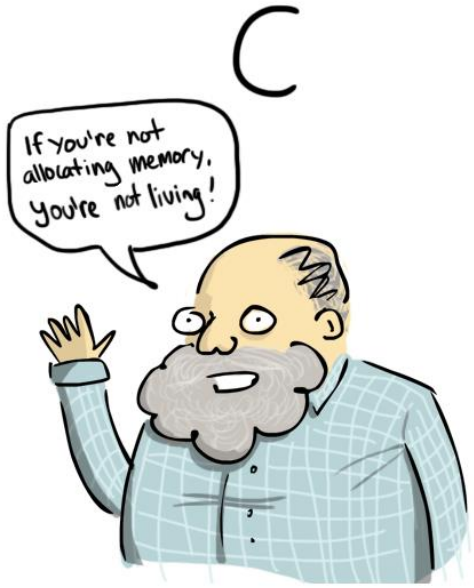
- Slowly creeping into the Systems curriculum at Northwestern

- You'll likely need to learn it on your own
  - https://www.rust-lang.org/learn
  - https://doc.rust-lang.org/book/
  - https://serokell.io/blog/learn-rust
  - https://learnxinyminutes.com/docs/rust/ (5 minute overview)

Break + Programming languages personified







https://leftoversalad.com/c/015_programmingpeople/

Outline

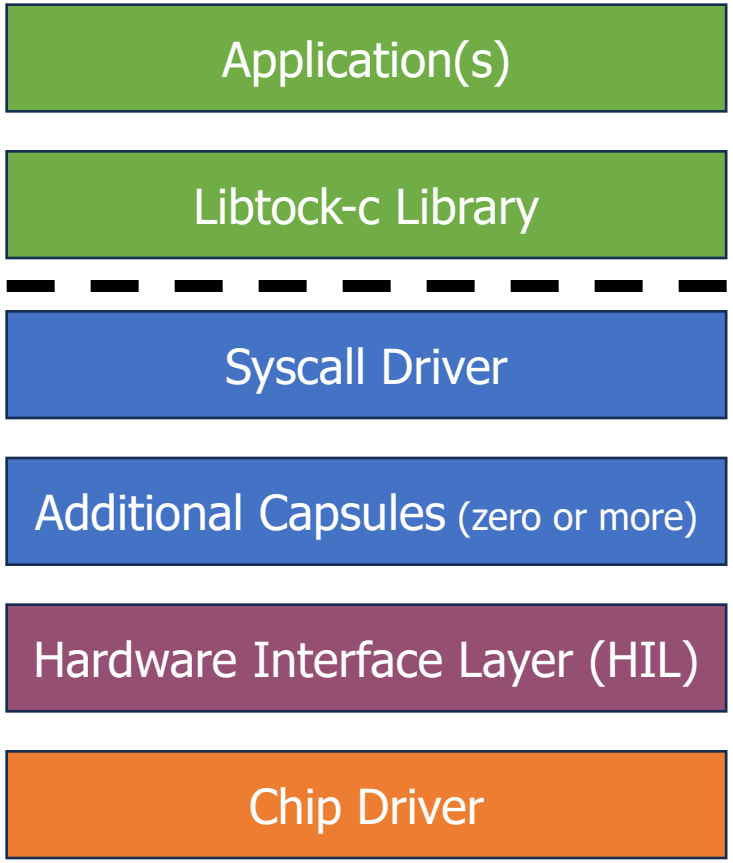
- Embedded Operating Systems
- Embedded Security Challenges
- Tock Overview
- Rust Programming
- **Tock Driver Design**

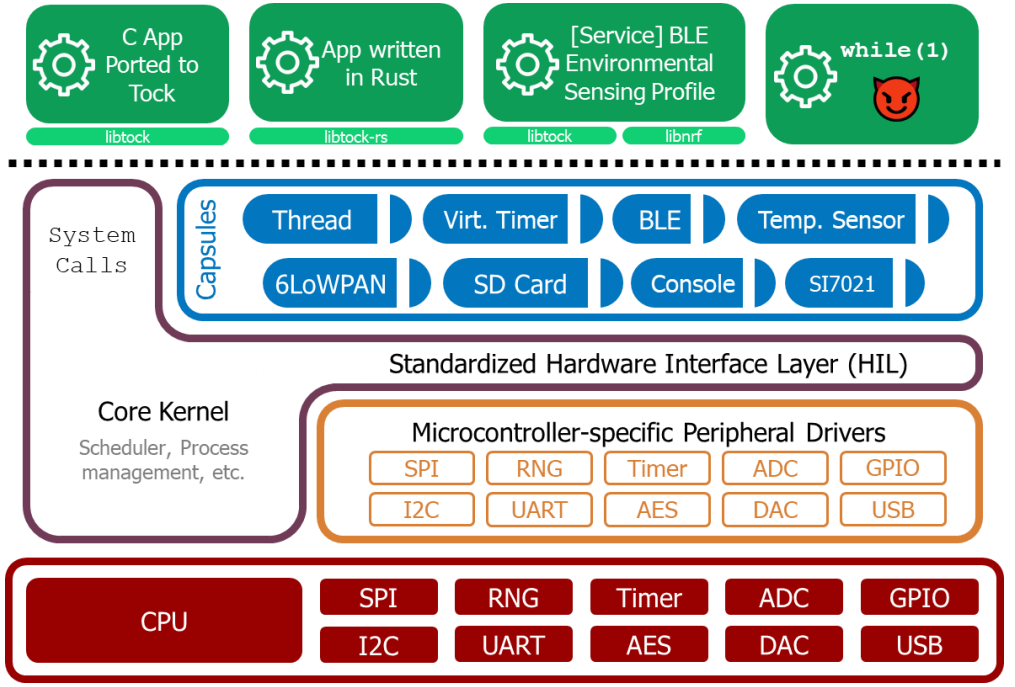
Driver “stack”

- Driver code does a lot of work internally
- Externally provides a simpler abstract interface to the device

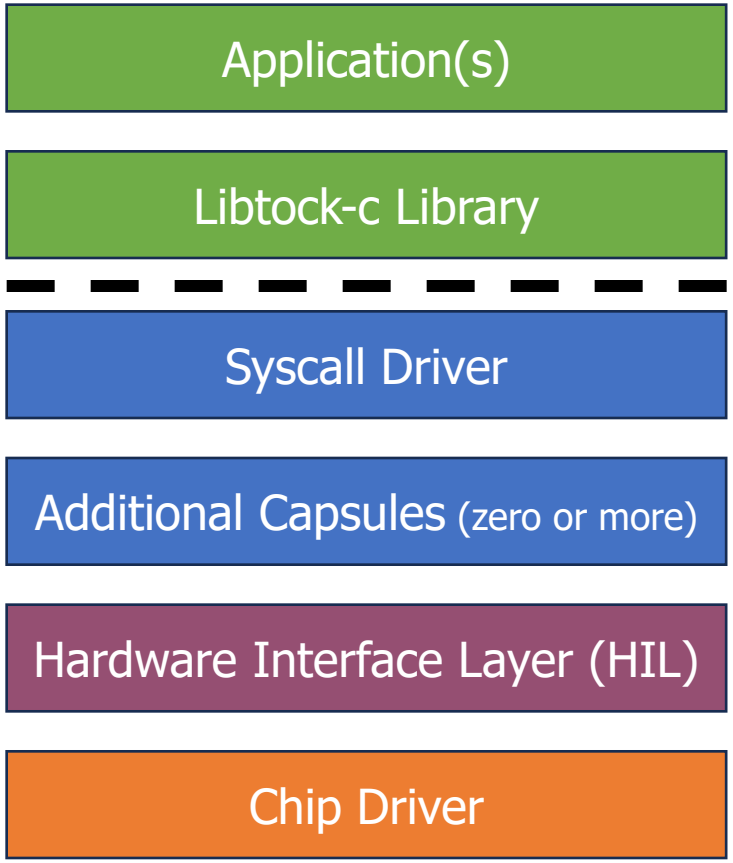
- Some drivers build on top of other drivers
  - Snake game
  - Builds on top of LED Matrix driver
  - Builds on top of GPIO and Timers

Typical software stack in Tock





Layer connections

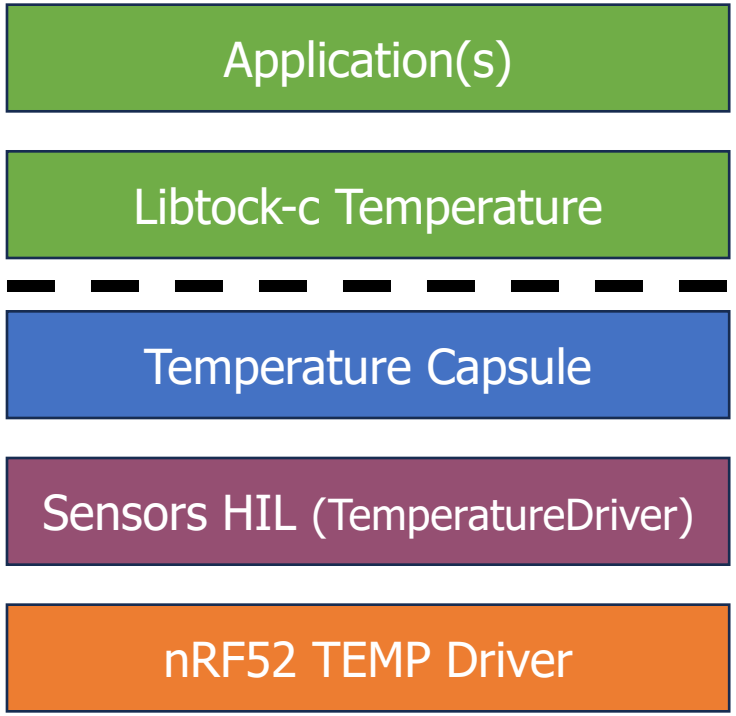


- Each layer can communication with the one above and below it
- Within the kernel each layer
  - Has a reference to the thing below and can call functions on it directly
  - Has a "client" reference to thing above
    - Allows responding upwards when action is complete
    - Usually one function: callback()

Example: Tock temperature stack on Microbit

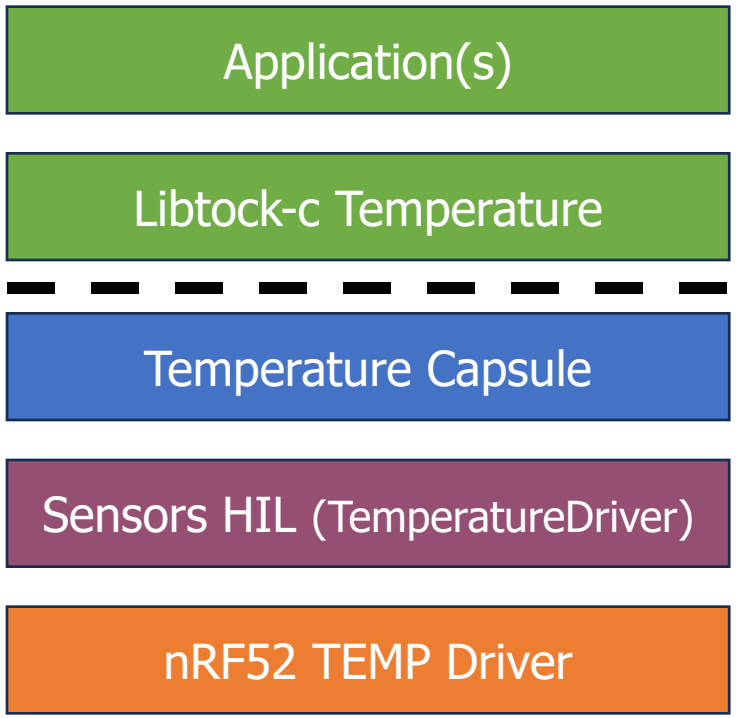
- Let's revisit our favorite temperature sensor
  - Degrees Celsius in 0.25 degree increments

- Register interface
  - TASKS_START
    - Start a measurement
  - INTENSET
    - Enable interrupts
  - EVENTS_DATARDY
    - Check if temperature value is ready
  - TEMP
    - Read temperature value



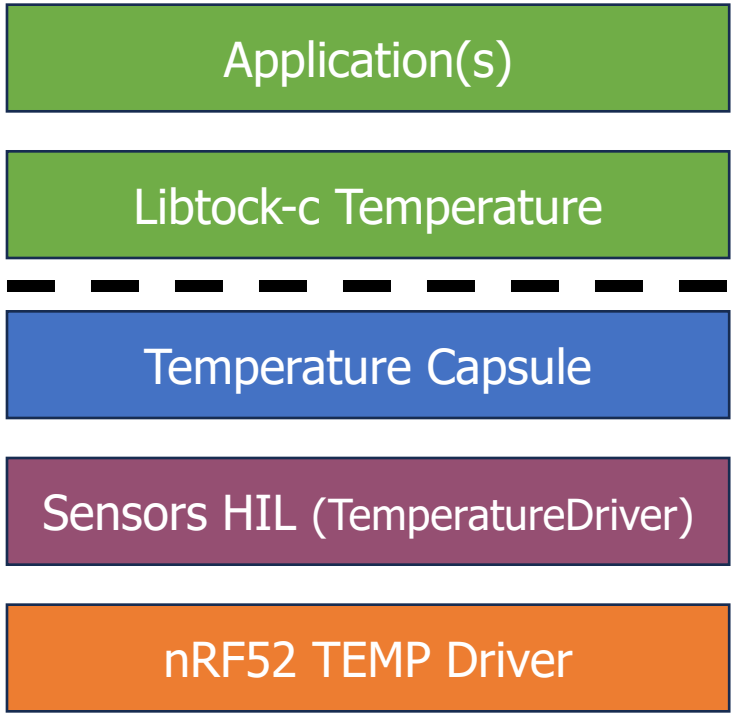
Temperature chip driver

- https://github.com/tock/tock/blob/master/chips/nrf5x/src/temperature.rs
- Accesses registers to interact with Temperature peripheral
- Demonstrates Tock register interface
- Implements the TemperatureDriver HIL



Senors Hardware Interface Layer

- Rust "trait" that a struct can implement
  - Allows shared behavior by multiple underlying implementations
- TemperatureDriver
  - set_client(&TemperatureClient)
  - read_temperature()
    -> Result<(), ErrorCode>
- TemperatureClient
  - callback(Result<i32, ErrorCode>)
- https://github.com/tock/tock/blob/master/kernel/src/hil/sensors.rs

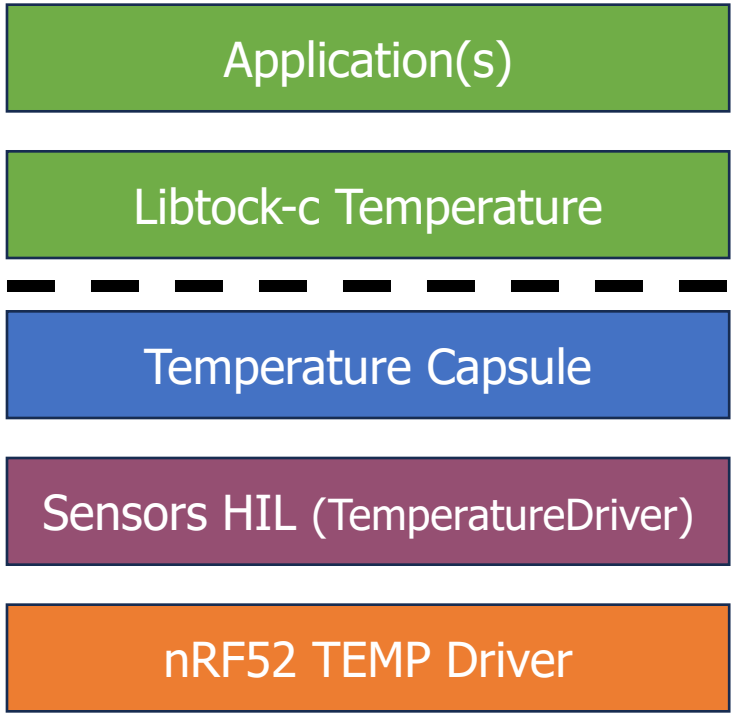


Temperature capsule

- https://github.com/tock/tock/blob/master/capsules/extra/src/temperature.rs

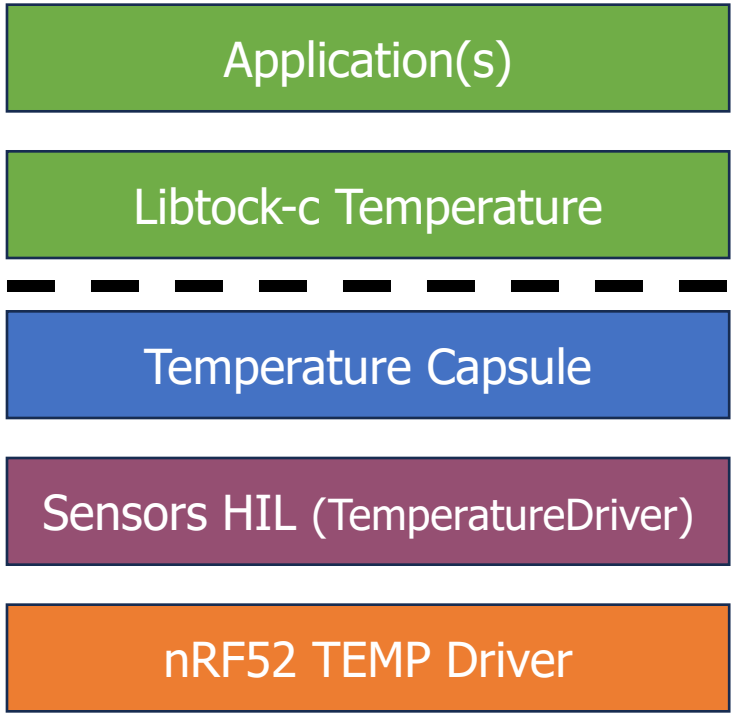
- Implements Syscall driver
  - Commands
    - 0 – check if this exists
    - 1 – read temperature
  - Subscribe
    - Get callbacks with temperature values

- Tracks “busy” status of sensor
  - But allows multiple applications to be notified when value is ready

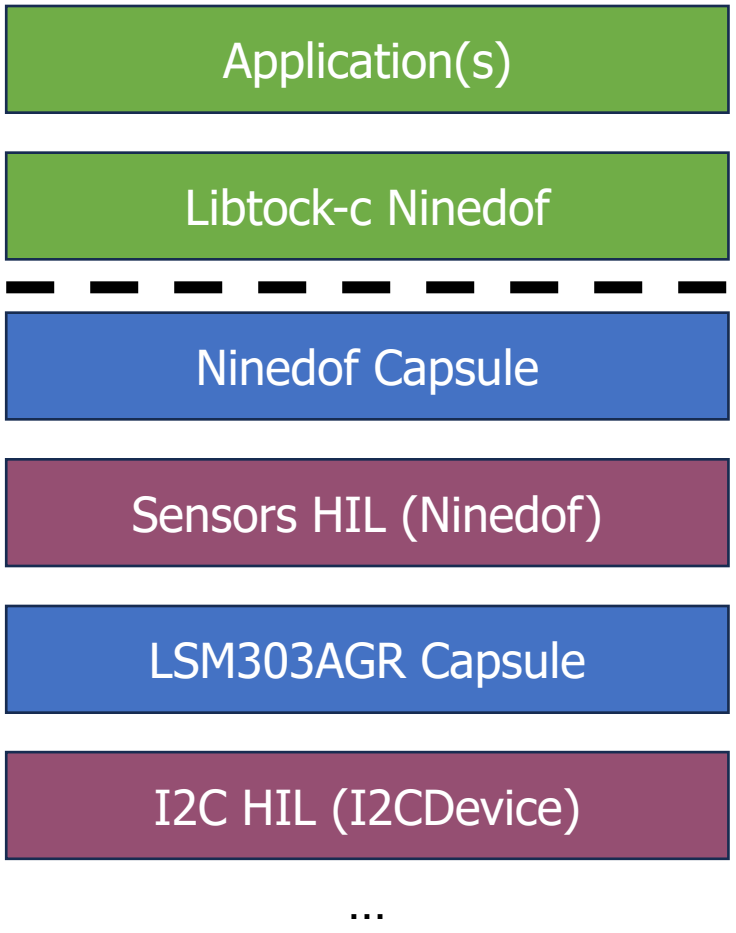


Libtock-c Temperature Library

- https://github.com/tock/libtock-c/blob/master/libtock/sensors/temperature.c
- libtock_temperature_read(callback)
- Actual syscalls in a separate file
  - https://github.com/tock/libtock-c/blob/master/libtock/sensors/syscalls/temperature_syscalls.c



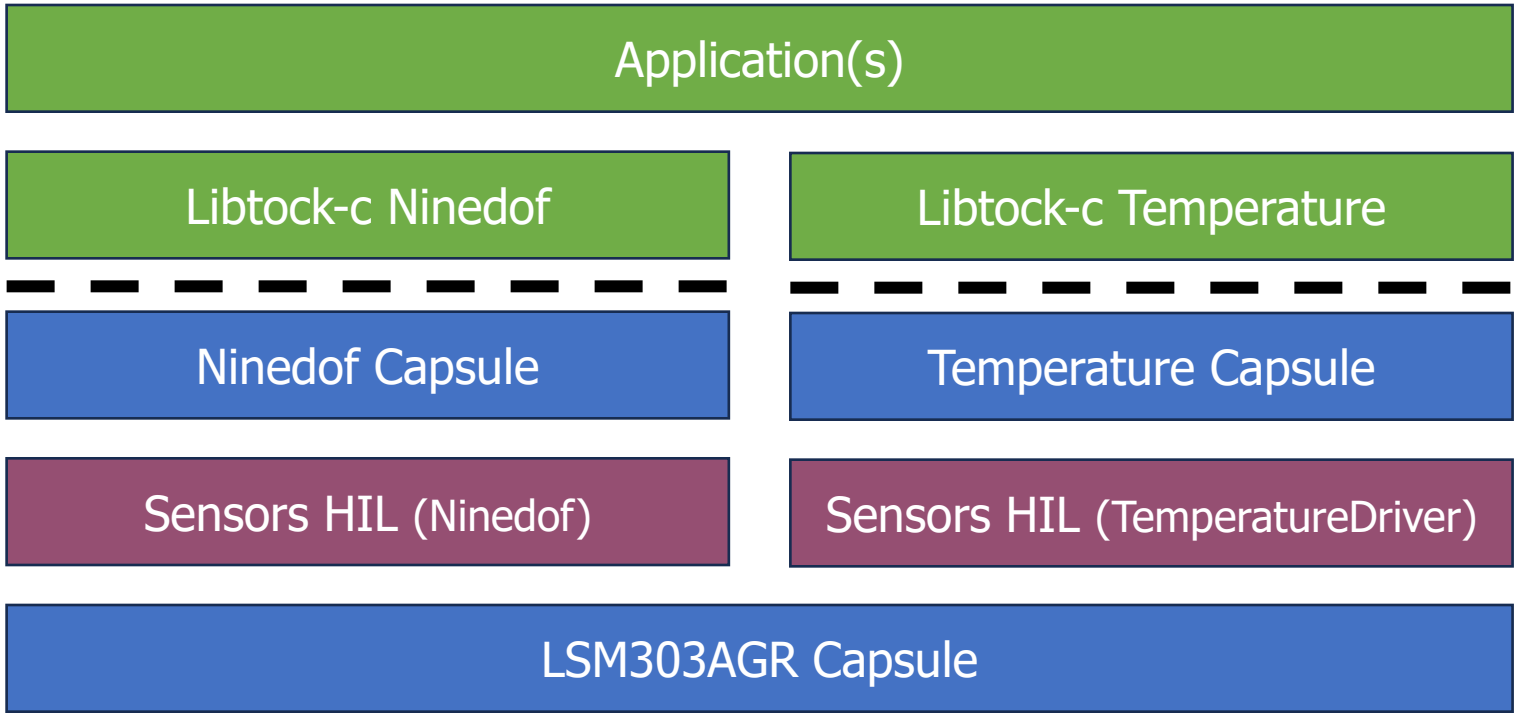
Example: Microbit Inertial Measurement Unit (IMU)



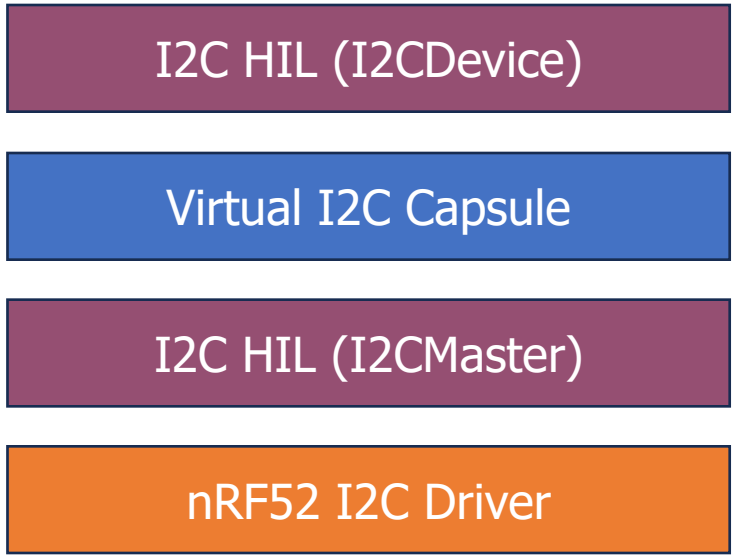
- Capsule for LSM303AGR chip
  - Uses generic I2C capsule below it
  - Provides interface for Ninedof above it
- Allows LSM303AGR driver to be reused on multiple boards
  - Each which could have different microcontrollers

Full LSM303AGR stack

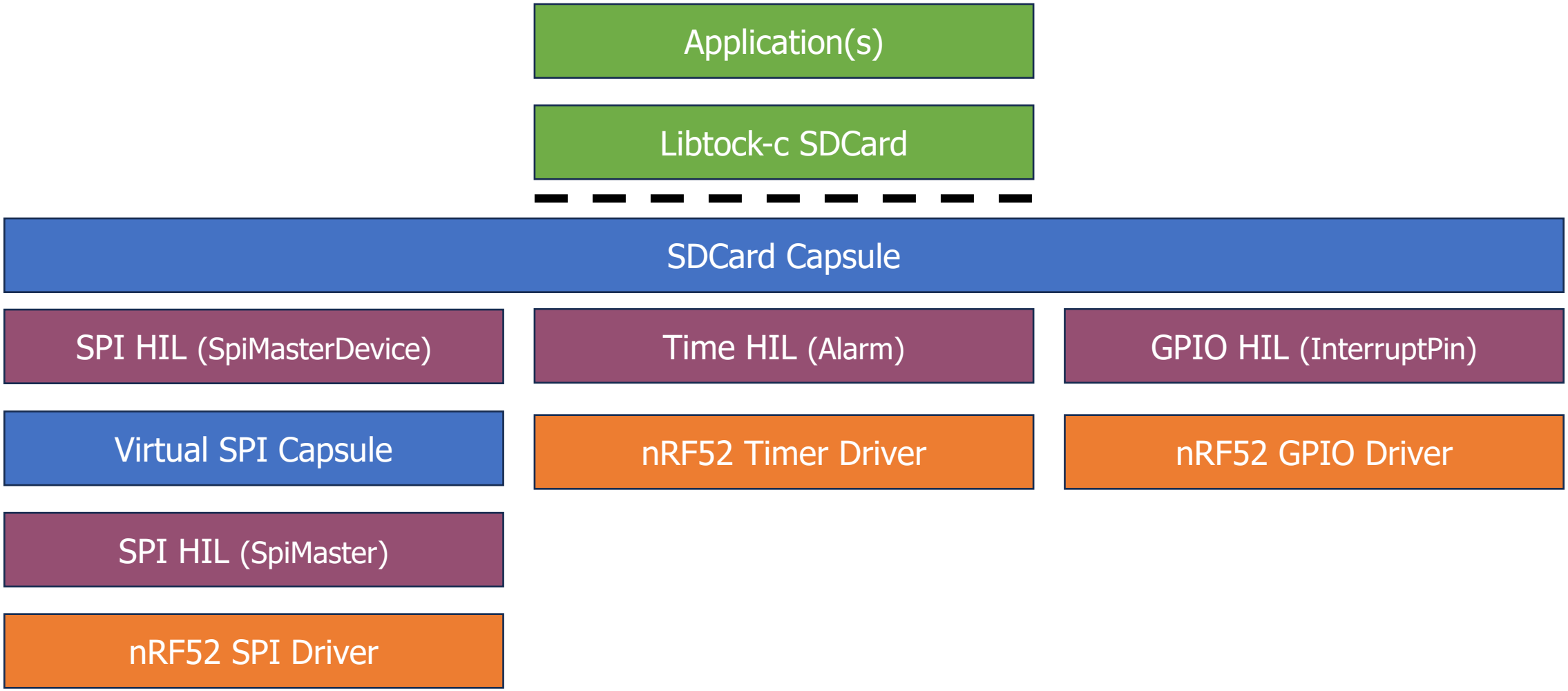
- LSM303AGR provides two interfaces
  - Ninedof
  - Temperature!



- Beneath it gets a bit messy
  - I2CDevice
    - Concept of a device with an address
  - I2CMaster
    - Implementation



Capsules can have multiple drivers beneath them too



General Tock resources

- https://github.com/tock/tock
- https://www.tockos.org/
- **"Multiprogramming a 64 kB Computer Safely and Efficiently"** Levy et al. 2017. Symposium on Operating Systems Principles. https://brandenghena.com/projects/tock/levy17multiprogramming.pdf
- **"Tock: From Research to Securing 10 Million Computers"** Schuermann et al. 2025. Symposium on Operating System Principles. https://brandenghena.com/projects/tock/2025-sosp-tock-decade.pdf

Outline

- Embedded Operating Systems
- Embedded Security Challenges
- Tock Overview
- Rust Programming
- Tock Driver Design