Connected Embedded Rust

A Survey of system software and network stacks in Rust

Elena Frank
Freie Universität Berlin

Internet of Things & Security Seminar, 04.07.2023

Outline

Embedded Rust
- Rust Programming Language
- Hardware Abstraction
- Asynchronous Code

Survey of System Software
- Concurrency Frameworks
- Tock
- Hubris

Networking stacks
- smoltcp
- Hubris’ usage of smoltcp
- Tock’s Network Stack

Summary

Outline

Rust Programming Language

- Popular Alternative to C for low-level Systems Programming
- Focus on Reliability and Performance
- Compiler enforced Memory, Thread and Type Safety

Ownership and Mutability

C:

```
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Ownership and Mutability

C:

```c
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Rust:

```rust
struct Item {
  n: u8
}

fn inc_if_zero(var: &Item) {

  if var.n == 0 {
    var.n += 1;
  }
}

fn foo() {
  let var = Item { n: 0 };

  let ref1 = &var;
  let ref2 = &var;

  thread::spawn(move || inc_if_zero(ref1));
  thread::spawn(move || inc_if_zero(ref2));

}
```

Ownership and Mutability

C:

```c
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Rust:

```rust
struct Item {
  n: u8
}

fn inc_if_zero(var: &Item) {

  if var.n == 0 {
    var.n += 1;
  }
}

fn foo() {
  let var = Item { n: 0 };

  let ref1 = &var;
  let ref2 = &var;

  thread::spawn(move || inc_if_zero(ref1));
  thread::spawn(move || inc_if_zero(ref2));

}
```

```
error[E0594]: cannot assign to ‘item.n‘, which is behind a ‘&‘ reference
 -> src/main.rs:7:9
  |
5 | fn increase_if_zero(var: &Item) {
  |                          --- help: consider changing this to be a mutable reference: ‘&mut Item‘
6 |     if var.n == 0 {
7 |         var.n += 1;
  |         ^^^^^^^^^^^ ‘var‘ is a ‘&‘ reference, so the data it refers to cannot be written
```

Ownership and Mutability

C:

```c
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Rust:

```rust
struct Item {
  n: u8
}

fn inc_if_zero(var: &mut Item) {

  if var.n == 0 {
    var.n += 1;
  }
}

fn foo() {
  let mut var = Item { n: 0 };

  let ref1 = &mut var;
  let ref2 = &mut var;

  thread::spawn(move || inc_if_zero(ref1));
  thread::spawn(move || inc_if_zero(ref2));

}
```

Ownership and Mutability

C:

```c
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Rust:

```rust
struct Item {
  n: u8
}

fn inc_if_zero(var: &mut Item) {

  if var.n == 0 {
    var.n += 1;
  }
}

fn foo() {
  let mut var = Item { n: 0 };

  let ref1 = &mut var;
  let ref2 = &mut var;

  thread::spawn(move || inc_if_zero(ref1));
  thread::spawn(move || inc_if_zero(ref2));

}
```

```
error[E0499]: cannot borrow ‘var‘ as mutable more than once at a time
  -> src/main.rs:17:16
     |
16   |     let ref1 = &mut var;
     |                ------ first mutable borrow occurs here
17   |     let ref2 = &mut var;
     |               ^^^^^^^^^ second mutable borrow occurs here
```

Ownership and Mutability

C:

```c
struct item {
    int n;
};

void inc_if_zero(struct item *var) {

  if (var->n == 0) {
    var->n += 1;
  }
}

void foo(void) {
  struct item var = { n: 0 };

  struct item *ref1 = &var;
  struct item *ref2 = &var;

  create_thread(inc_if_zero, ref1);
  create_thread(inc_if_zero, ref2);
  return;
}
```

Rust:

```rust
struct Item {
  n: u8
}

fn inc_if_zero(var: Mutex<Item>) {
  let mut var = var.lock().unwrap());
  if var.n == 0 {
    var.n += 1;
  }
}

fn foo() {
  let var = Mutex::new(Item { n: 0 });

  let ref1 = &var;
  let ref2 = &var;

  thread::spawn(move || inc_if_zero(ref1));
  thread::spawn(move || inc_if_zero(ref2));

}
```

Generics and Traits

```
pub trait Write <Word> {

    type Error;

    fn write(&mut self, word: Word) -> Result<(), Self::Error>;
    fn flush(&mut self) -> Result<(), Self::Error>;
}


fn write_all<S>(
  serial: &mut S,
  buffer: &[u8]
) -> Result<(), S::Error>
where
    S: Write<u8>
{
    for &byte in buffer {
        block!(serial.write(byte))?;
    }

    Ok(())
}
```

[5]

Outline

Micro-architecture Layer:

- Processor-specific logic
- Processor calls, timer management, interrupt handling, ...
- Example: ARM Cortex-M Processors

Micro-architecture Layer:

- Processor-specific logic
- Processor calls, timer management, interrupt handling, ...
- Example: ARM Cortex-M Processors

Peripheral Access Layer:

- Provide access to a chips memory-mapped registers and periphery
- GPIO, SPI, I2C, UART, ..
- Example: Nordic Conductor nRF52805 SoC

Hardware Abstraction Layers

Micro-architecture Layer:

- Processor-specific logic
- Processor calls, timer management, interrupt handling, ...
- Example: ARM Cortex-M Processors

Peripheral Access Layer:

- Provide access to a chips memory-mapped registers and periphery
- GPIO, SPI, I2C, UART, ..
- Example: Nordic Conductor nRF52805 SoC

Board Layer:

- Configure underlying layers for one concrete Board
- Abstracts Hardware details, configures periphery &assigns drivers, ...
- Example: STM32 Nucleo-32 Board

Outline

Challenge:

- Embedded Systems do a lot of I/O
- I/O is asynchronous => result not immediately available
- Program execution should continue with other work in the meantime

Challenge:

- Embedded Systems do a lot of I/O
- I/O is asynchronous => result not immediately available
- Program execution should continue with other work in the meantime

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```
pub trait Future {
  type Output;
  fn poll(...) -> Poll<Self::Output>;
}
```

Challenge:

- Embedded Systems do a lot of I/O
- I/O is asynchronous => result not immediately available
- Program execution should continue with other work in the meantime

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```
pub trait Future {
  type Output;
  fn poll(...) -> Poll<Self::Output>;
}
```

- The *await* keyword allows blocking the current execution until a Future is ready
- Functions that use await themselves become a Future and must be annotated as *async*

Outline

- Embedded platforms for apps with concurrent units of execution

Embassy:

```
#[embassy::task]
async fn blink(pin: AnyPin) {

  let mut led = Output::new(pin,Level::Low,
    OutputDrive::Standard);

  loop {
    led.set_high();
    Timer::after(Duration::from_millis(150)).await;
    led.set_low();
    Timer::after(Duration::from_millis(150)).await;
  }
}
.
#[embassy::main]
async fn main(spawner: Spawner, p: Peripherals) {

  spawner.spawn(blink(p.P0_13.degrade())).unwrap()

  loop { }
}
```

[2]

RTIC:

```
#[app(...)]
mod app {

  #[shared]
  struct Shared {}

  #[local]
  struct Local {
    led: PA5<Output<PushPull»,
  }

  #[init]
  fn init(_: init::Context) -> (Shared, Local) {
    // Setup RCC and timer
    ...
    // Setup LED
    let led = ...

    blink::spawn().ok();
    (Shared {}, Local { led })
  }

  #[task(local = [led])]
  async fn blink(cx: blink::Context) {
    // blink led
    ...
  }
}
```

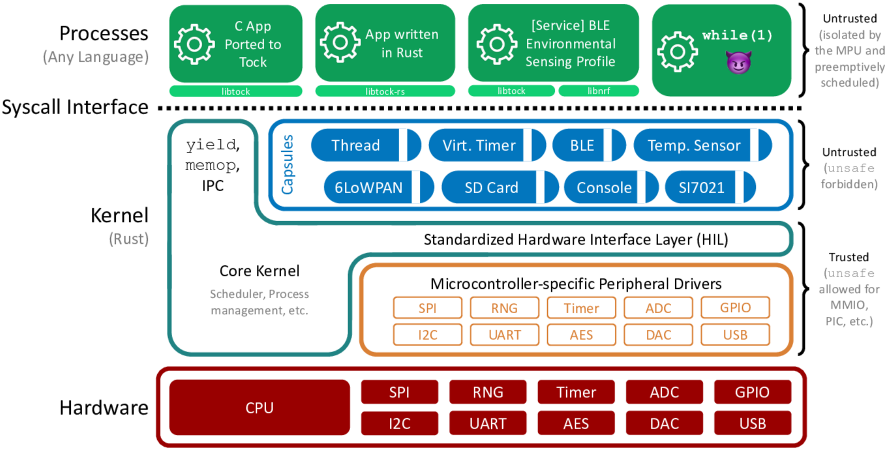
[3]

Outline

Tock Operating System

- Goal: enable secure multi-programming on embedded system
- Monolithic kernel: units sand-boxed via Rust type system
- Supports dynamic loading of processes and memory management
- IPC implemented as Client-Server communication
- Own Hardware Interface-Layer

Tock Architecture



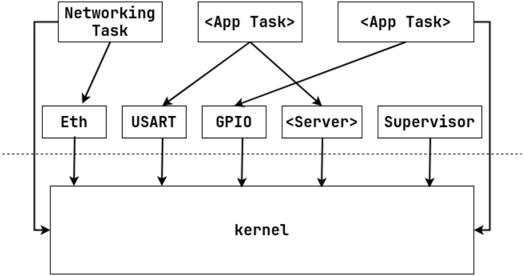
[4]

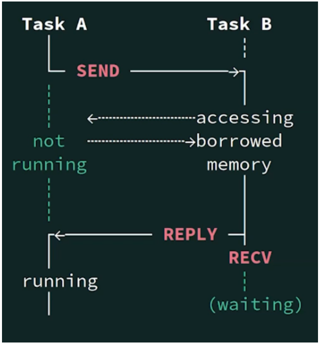
Outline

Hubris Operating System

- Very small hubris kernel + collection of tasks
- Tasks: Separately-compiled Programs & Drivers
  - Unprivileged processor mode
  - Isolated in memory
  - No dynamic creation of tasks; fixed priorities
- "Aggressively static" for all allocatable or routable resources
- Synchronous IPC: Sender is blocked until receiver responded

Hubris Architecture





[1]

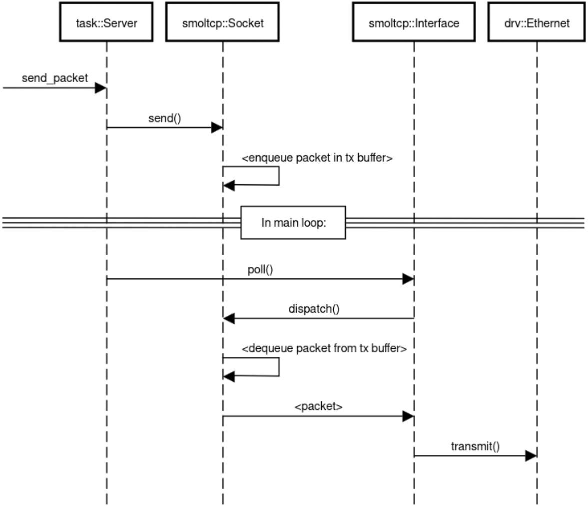
Outline

smoltcp

- Sockets
  - Supported: raw Sockets, ICMP, UDP and TCP
  - Implementation of socket logic: buffering, packet construction, ...
  - Sans-IO: no actual I/O done by the socket
- *Device* trait
  - Methods for interacting with the actual device
  - User-provided
- Interfaces
  - Supported: Ethernet, IP, IEEE 802.15.4 + 6LoWPAN
  - Takes care of physical addressing, neighbour discovery, sending of control packets
  - Implemented as state-machine

Outline

Sending a packet in Hubris



Outline

Tock’s Network Stack

```
IPv6 over ethernet:      Non-Thread 15.4:   Thread Stack:                                                Encapsulation Libraries
+-------------------+-------------------+------------------------------+
|                         Application                                  |-------------------\
----------------------------------------+-------------+---+------------+                    \
|TCP Send| UDP Send |TCP Send| UDP Send |  | TCP Send |   | UDP Send |--\                     v
+--------+----------+--------+----------+  +----------+   +----------+   \                 +-------------+  +-------------+
|     IP Send       |     IP Send       |  |          IP Send          |   \       ----->  | UDP Packet |  | TCP Packet |
|                   |                   |  +---------------------------+    \     /        +-------------+  +-------------+
|                   |                   |                              |     \   /         +------------+
|                   |                   |                              |      -+-------->  | IP Packet |
|                   |                   |       THREAD                 |       /           +------------+
| IP Send calls eth | IP Send calls 15.4|                   <--------|------>              +-------------------------+
| 6lowpan libs with | 6lowpan libs with |                              |       \  ------->  | 6lowpan compress_Packet |
| default values    | default values    |                              |        \          +-------------------------+
|                   |                   |                              |         \         +-------------------------+
|                   |                   +                  +-----------|           ------>  | 6lowpan fragment_Packet |
|                   |                   |                  | 15.4 Send |                   +-------------------------+
|-------------------|-------------------+----------------------------+
|     ethernet      |          IEEE 802.15.4 Link Layer                |
+-------------------+--------------------------------------------------+
```

[4]

Summary

- Rust offers powerful primitives for building secure and reliable low-level systems
- The type system can be leveraged in very different designs

[1] Cliff L. Biffle. *On Hubris and Humility: developing an OS for robustness in Rust*. https://talks.osfc.io/osfc2021/talk/JTWYEH/. Nov. 2021.

[2] *Embassy*. https://github.com/embassy-rs/embassy. June 2023.

[3] *Real-Time Interrupt-driven Concurrency (RTIC)*. https://github.com/rtic-rs/rtic. June 2023.

[4] *Tock*. https://github.com/tock/tock. June 2023.

[5] *embedded-hal*. https://crates.io/crates/embedded-hal. v0.2.7.