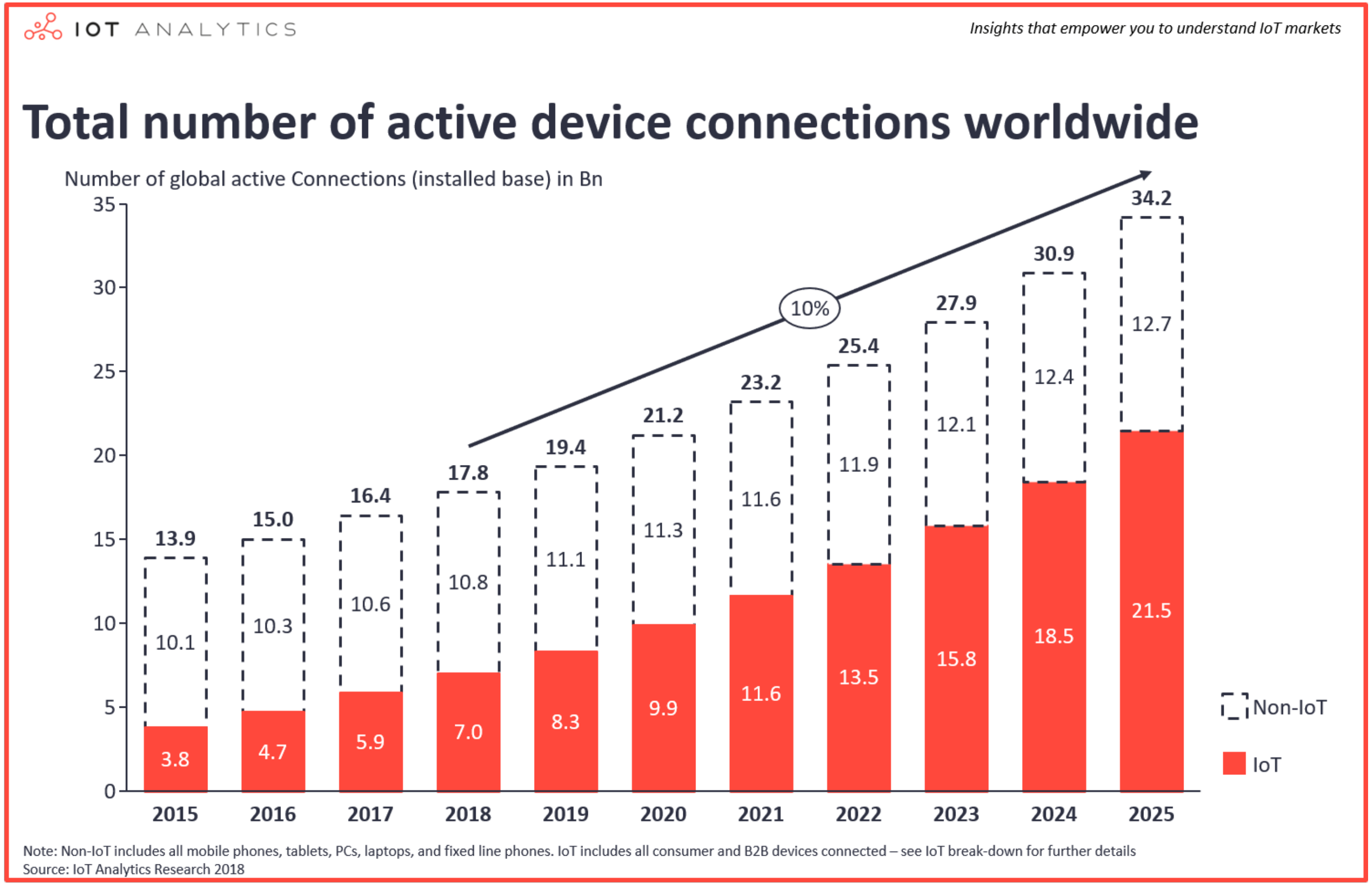
Rust RTIC@Grepit 2021

Per Lindgren PhD, Professor
per.lindgren@ltu.se
Luleå University of Technology
per.lindgren@grepit.se
Director of RnD Grepit AB

Andreas Lundqvist
andreas.lundqvist@grepit.se
COO Grepit AB

Grand challenge - Short Term

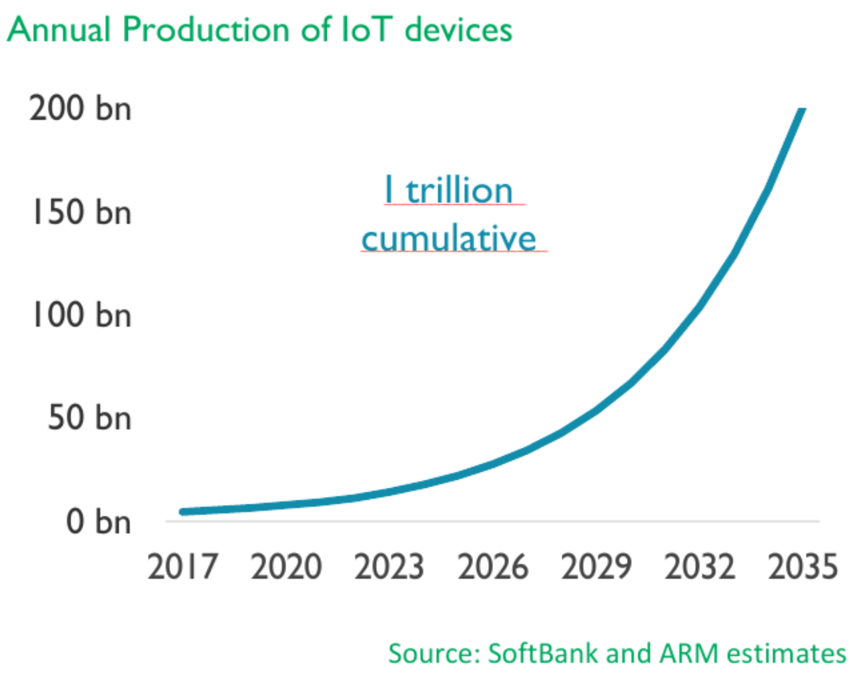


Grand challenge – Long Term

By the year 2035, spending on IoT hardware and services will reach **a trillion dollars per annum**.

This level of investment supports our view that **a trillion IoT devices** will be produced within the next twenty years.

Source: ARM



Grand challenge - Design

Distribution of spending on IoT systems in 2035

IoT Services 65%	IT services	45%	Systems integration (design, procurement, project management) Data centre hosting (renting out servers and storage space) Device lifecycle management (provisioning, updating, decommissioning) Analytics software (sold under software-as-a-service contracts)
	Telecoms services	15%	Carrier networks (mobile, wireline) Internet Service Providers
	Financial services	5%	Financing; payments processing
IoT Hardware 35%	Installation	10%	Installing devices on-site
	Distribution	5%	Transporting components to assemblers, devices to end users
	Assembly	5%	Assembling components into modules, modules into devices
	Components	15%	Semiconductor chips, analog components, circuit boards

Source: ARM estimates

Grand challenge - Design

- Robustness, Reliability & **Security**
- Efficiency (CPU, Memory, Power, Bandwidth)
- Cost (Design & Maintainance)

Rust RTIC

The Rust language

- Performance
- Reliability
- Productivity

RTIC model

- Efficient Scheduling
- Safe Concurrency
- Easy to program

Rust RTIC

- **IoT devices**
- **Robotics**
- **Automotive**

The Rust language

- Performance
  - Typically on par or better than C/C++
  - “Zero-cost” abstractions (code & memory OH optimized out)
- Reliability
  - Memory safety (compile time checking + built in assertions)
  - Well defined behavior (unlike C/C++ with lots of UD)
- Productivity
  - Best in class ecosystem with version handling etc.

  Most loved language among developers since its release!

Rust Memory Safety

- Builds on linear type theory
  - Philip Wadler, “Linear Types can change the World!”, 1990
  - Each value **must** be used once (and is consumed when used)
- More permissive, *affine* type system
  - Each value **may** be used once, and is consumed when used or goes out of scope
  - Each value (v:T) has a **single owner**
    Accesses by reference under restrictions (Rust aliasing rules)
    - multiple `& v:T` references may co-exist, or
    - only a single (unique) `& mut v:T` reference is allowed

Rust Memory Safety

- The `borrow checker` enforces the aliasing rules.
- Rust ensures all references to be live and point to valid data. (Similar to C++ RAII but fully enforced at compile time.)

- In effect, Rust programs passing compilation are memory safe, unless for code **explicitly** marked `unsafe`.
- `unsafe` still implies all Rust invariants besides allowing:
  – *raw* pointer dereferencing, and
  – calling other `unsafe` code.

Rust Memory Safety

- What about Out Of Memory (OOM)?
  - OOM is not a concept of the Rust language!
  - The Rust standard library (std) has a default `allocator`:
    - System will `panic` on OOM, which is “sound”
(does not break memory safety).
    - However, it breaks reliability
(up to the user/OS to deal with).
    - Recovering may be very hard/impossible and thus a no-go for reliable safety critical applications.

Rust Memory Safety

- What about Stack Memory Overruns (SMO)?
  - SMO is not a concept of the Rust language!
  - A Rust run-time system may protect stack frames, and catch overruns (does not break “soundness”)
  - However, it breaks reliability (hard to recover, similar to OOM)
- In essence:
  - *Memory Safe by Construction*
  - *OOM and SMO handled by sacrificing reliability*

  (Not perfect, but far better than C/C++)

And now, the problem with Threads...

... well might look cute but ...

The problem with Threads...

- Deceivingly simple, it is very easy to make mistakes ...
- ... forget to lock (race-condition)
- ... forget to unlock (live/dead-locks)
- ... cyclic resource dependencies (dead-locks)
- Complexity
  - Huge APIs, see e.g., http://man7.org/linux/man-pages/man7/pthreads.7.html
  - What does locking a Mutex really mean?
    - Depends on OS, Scheduling Policy, how the Mutex was created, etc.
    - ... in the end who knows what the cat brought in ...

RTIC Model
A “Thread free” solution

- Shared Resources
- Concurrent Tasks
  - Run-to-end semantics
  - Resources can be locked *only* in LIFO order (nested critical sections)

```
Resources {
    shared : T,
}
...

#[task(resources = [shared])]
fn task1(mut cx: task1::Context) {
    cx.resources.shared.lock(|shared| {
        // shared is of type &mut T
        *shared = ...;
    });
}
```

Background
Stack Resource Policy (SRP)

- Resurces are
  - Accessed as named critical sections (lock on entry/unlock on exit)
  - Restricts concurrency to ensure unique ownership

- Tasks are
  - sequences of operations with run-to-completion semantics
  - only allowed to “claim” resources in a nested fashion

    Yes, the Last In First Out (LIFO) is indeed a stack,
    hence the name **Stack** Resource policy

    This is what makes SPR unique!

Background
SRP Key features

- Preemtive, static and dynamic (e.g., Eearliest Deadline First), scheduling of single-core systems with shared resources

- Race- and deadlock-free execution

- Bounded priority inversion

  *a task **t** is blocked only by the **single** longest critical section for any resource with a ceiling higher than the priority of **t***

- Memory efficient (executes on a single shared stack)

- Established theory for response time, overall schedulability ant total stack analysis

Background
SRP Requirements

- SRP requires static analysis of the set of Tasks & Resources
  - Ensure LIFO ordering of resource access
  - Compute the (static) ceiling for each resource

    the static priority ceiling for a resource ***r*** *is computed as*
    the maximum priority for any task ***t*** *that access* ***r***

SRP Is Well Known but...

- The programmer is used to *threads*, so could we translate a thread model to SRP?

  This is however problematic:

  - threads can typically be created/destroyed on the fly
  - lots of synchronization primitives, mutex, semaphores, conditional variables, etc.

  Without a *model* of the program it is not easy or even possible

- In pratice SRP based scheduling is **not** that common, an existing solution is the OSEK SLOTH-kernel, building on the AUTOSAR Task/Resource model

What is cortex-m-rtic?

- Single-core scheduler for the Cortex-M family of MCUs
- Strong guarantees to:
  - Race free execution (property of SRP)
    unbreakable: ensured by the design resources are accissible only when claimed
  - Deadlock free execution (property of SRP)
- Integrated with the cortex-m ecosystem
  - cortex-m, cortex-m-rt, etc.
  - svd2rust generated peripheral access
  - embedded-hal implementations and support crates
  - cargo

What is cortex-m-rtic?

- Other cortex-m-rtic properties
  - Memory and CPU Efficient execution with predicable overhead
    - Tasks/interrupts scheduled directly by the hardware (NVIC) zero memory and CPU overhead
    - Entering/exiting critical sections requires just a few machine instructions and a single byte stack memory for each nesting
    - Message passing using lock-free zero cost abstractions
- Other SRP properties
  - Bounded priority inversion
  - Single stack execution
  - Methods to response time analysis, overall schedulability, total stack usage

Under the hood...

- The **app** procedural macro
  - analyses the set of tasks and resources
  - computes resource ceiling values
  - generates glue code for scheduling and resource management

Resources and Priorities

- Assign task priories inverse to deadlines (DM scheduling)
- Locks are always wait free (never cause a context switch)
- Lock/unlock only a few machine instructions
  (single write to HW register or Cortex M3 and above)
- Locks are optimized out where possible
  (so you can write your code generic to priority assignments, and still have zero-cost access)

Supporting tools

- cargo-call-stack
  - static call graph reconstruction and stack estimation
- cargo-klee (experimental)
  - symbolic execution for Rust programs to prove
    - programs to be free of panic!
    - Input/Output equivalence between implementations
    - verify properties, e.g., safety, liveness, partial correctness

Total memory safety

- Recall Out Of Memory (OOM) and Stack Overflow is not covered by the Rust language/model
  - Heapless is a library for dynamic memory backed by static allocation (memory safe and panic free)
  - Cargo-call-stack gives worst case stack behavior per task (total stack usage can be bounded)
  - Used together to obtain “total memory safety”

Everything in Rust RTIC?

- May not be possible
  - Auto generated code (e.g., MATLAB)
  - Black box software components
  - Lack of pre-certified software components
  - Or simply, too high effort

- Luckily Rust provides excellent FFI support
  - Build system integration
  - Zero-cost (no added overhead)
  - By careful design memory safety remains
    (even improves memory safety of the exteral code base)

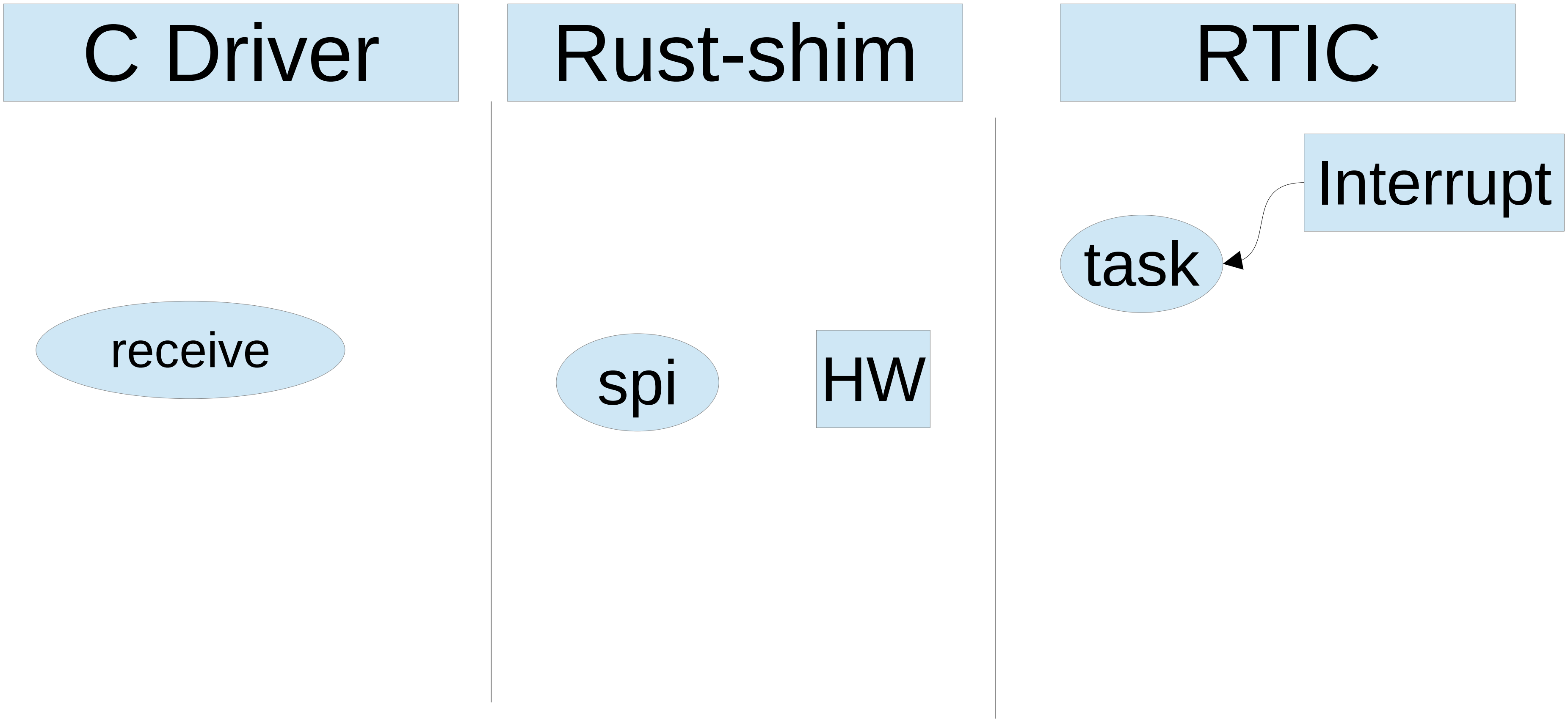
Legacy code integration

- FFI not restricted to C/C++, any lang with compatible ABI possible
- Tooling
  - rust-bindgen, cbindgen
    - Automatically generates Rust FFI bindings to/from C (and some C++)
    - RAW/unsafe interface to external code
  - Rust ships with an LLVM based toolchain
    - LLVM Link time optimization possible (LTO)
    - LLVM tools typically work out the box
      - Sanitizers, etc.

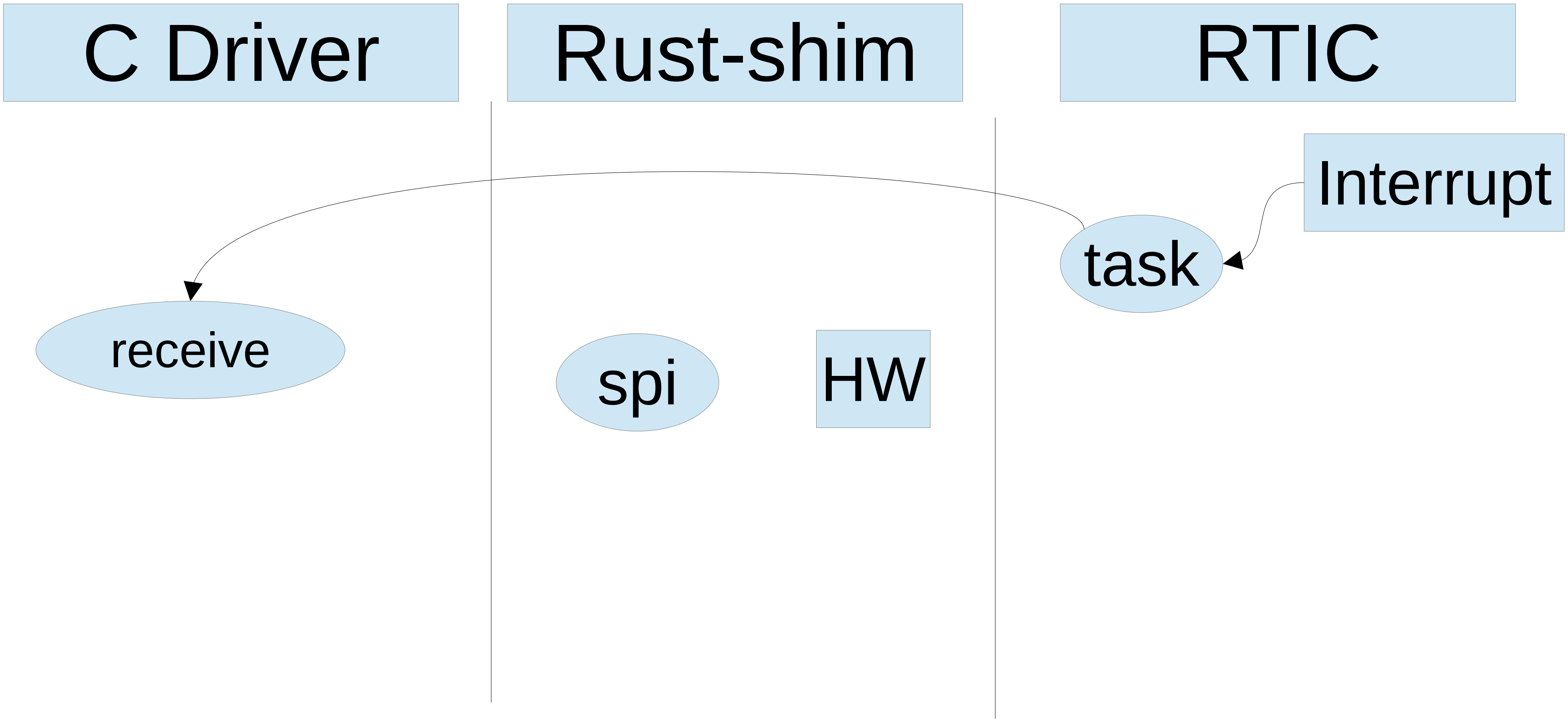
Memory safe
Legacy code integration

- External code stateless
  - Rust/RTIC has ownership of (memory) resources
    (External code passed reference to locked resource)
- External code stateful
  - Wrap component into RTIC resource to ensure safe state access
- External code stateful and “self scheduled” (e.g., capturing interrupts)
  - External code needs to be trusted
  - Allows for pre-certified software components (e.g., radio driver)

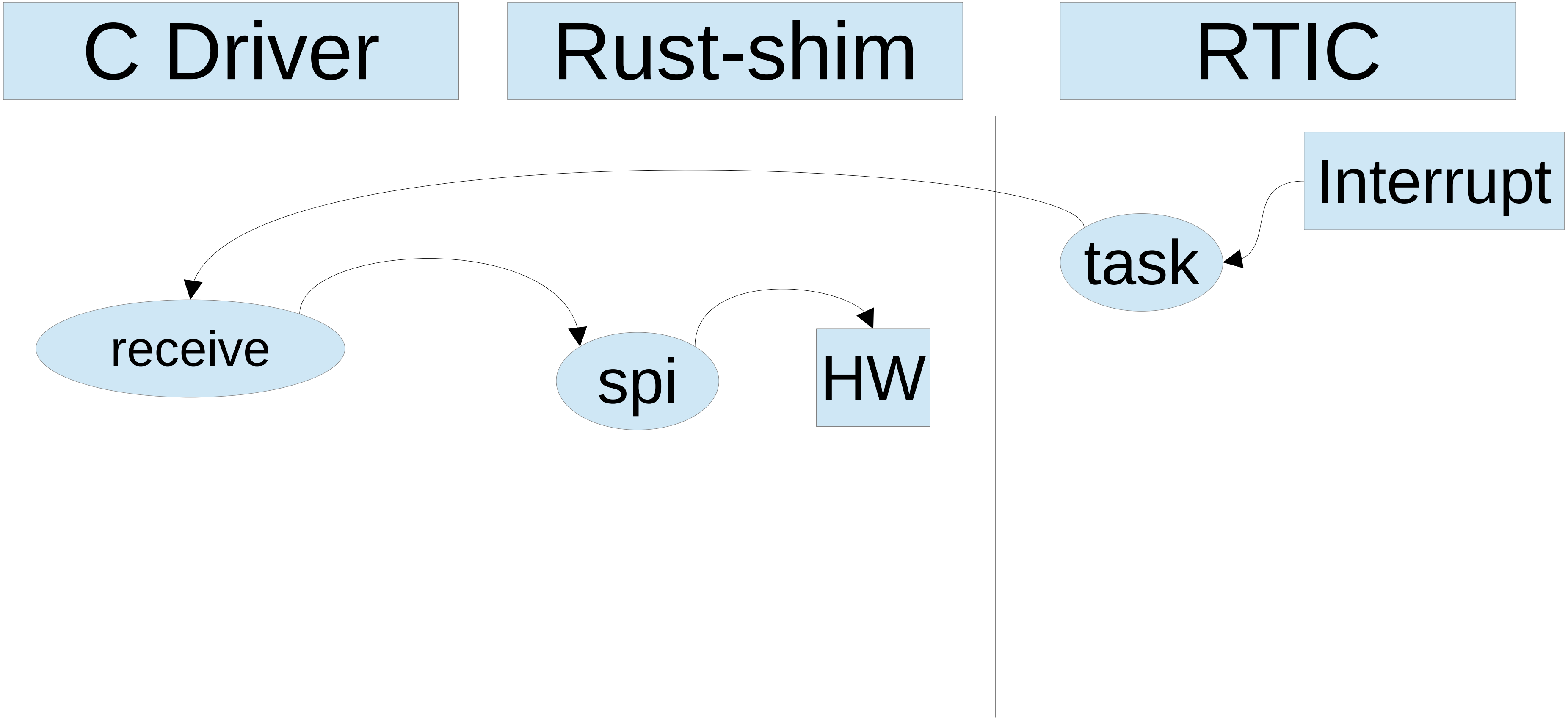
Example



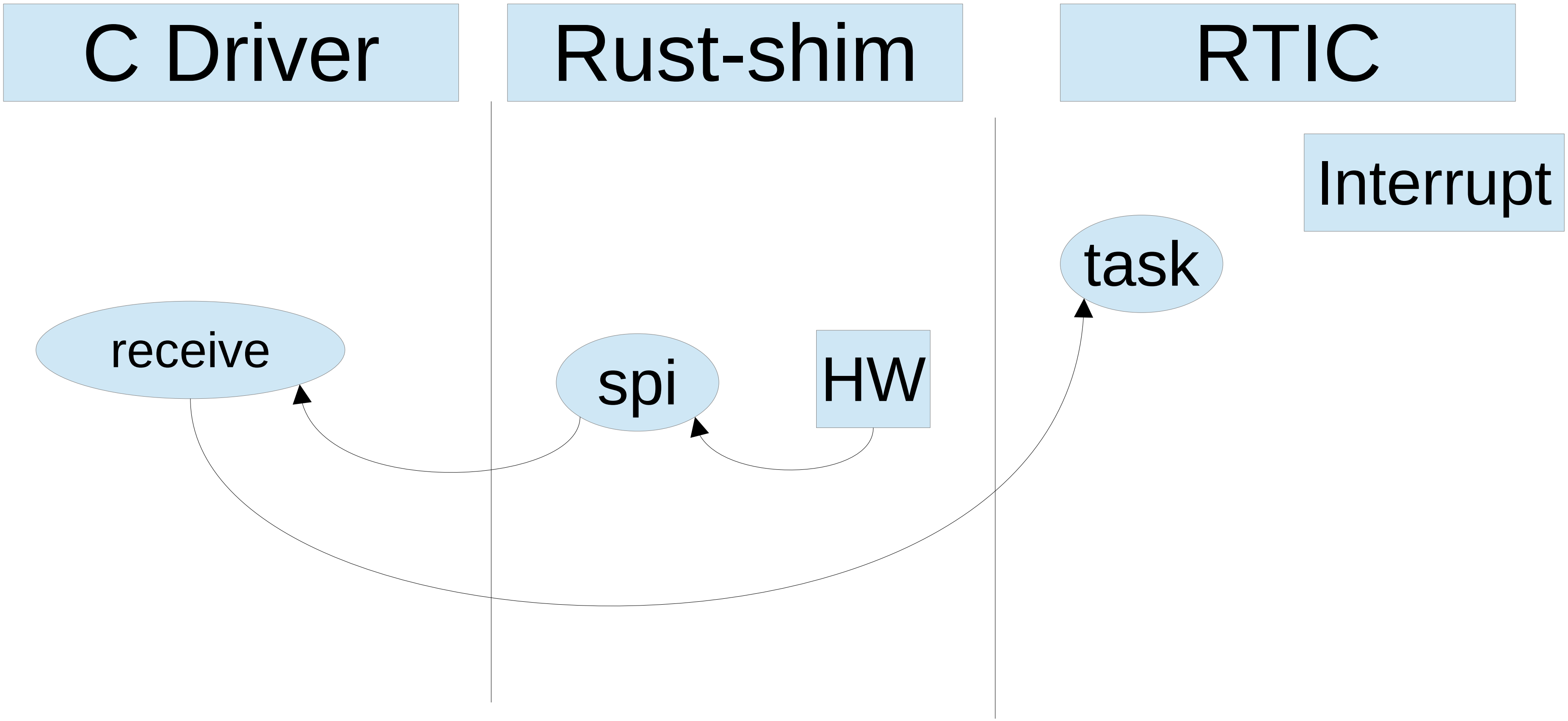
Example



Example



Example



Rust RTIC – In Production@Grepit

- Products on the market since 2017
  - 2017 GemPen - Gemstone Analysis Pen
  - 2018 Nexus - data acqusition system
  - 2019 X-Ray Geocore scanner (OreExplore)
  - 2020 Zpark - Electrical Vehicle Charger

Grepit competences and services

- Competences
  - Hardware (ASIC/FPGA/PCB, assembly etc.)
  - Sofware (Rust/C/C++/Python/Web services etc.)
- Grepit provides expert consulting services
- Customer development projects
- Complete system deliveries

Rust RTIC – In Production@Grepit

- Demo of Rust RTIC running on Zpark
  - LoRa Radio protocol implementation in C
  - Radio module implemented in Rust RTIC integrating C code

Rust RTIC – Related references

- RTIC: A Zero-Cost Abstraction for Memory Safe Concurrency
  https://www.youtube.com/watch?v=rYXy8dXYTNg
- RTIC: Real Time Interrupt driven Concurrency
  https://www.youtube.com/watch?v=saNdh0m_qHc
- An Overview of the Embedded Rust Ecosystem
  https://www.youtube.com/watch?v=vLYit_HHPaY
- Considering Rust
  https://www.youtube.com/watch?v=DnT-LUQgc7s

Rust RTIC

Open source project

- https://crates.io/crates/cortex-m-rtic
  Download/publishing

- https://rtic.rs

  Docs/book

- https://github.com/rtic-rs
  github development organization/team