Securely bringing OPENTHREAD released by Google
to an embedded operating system

Samir Rashid

My background

Aviation software (C++/Python)

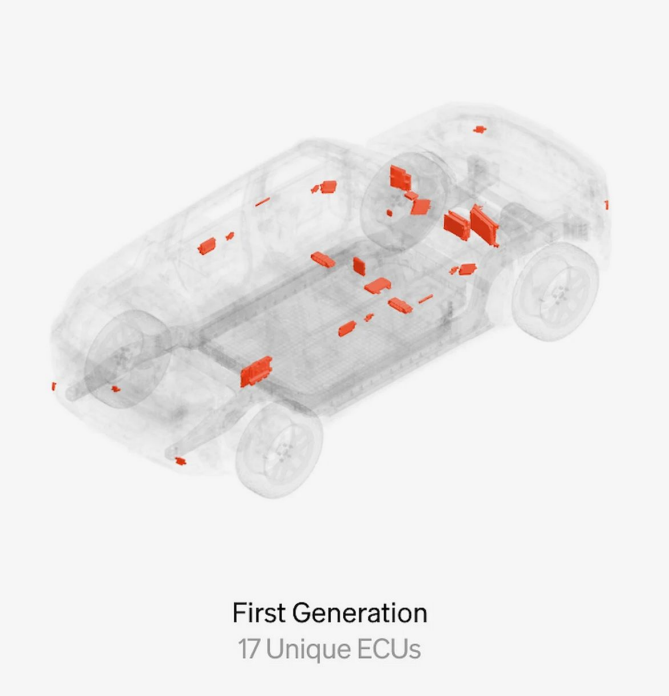
Operating systems (C/Rust)

Formal verification

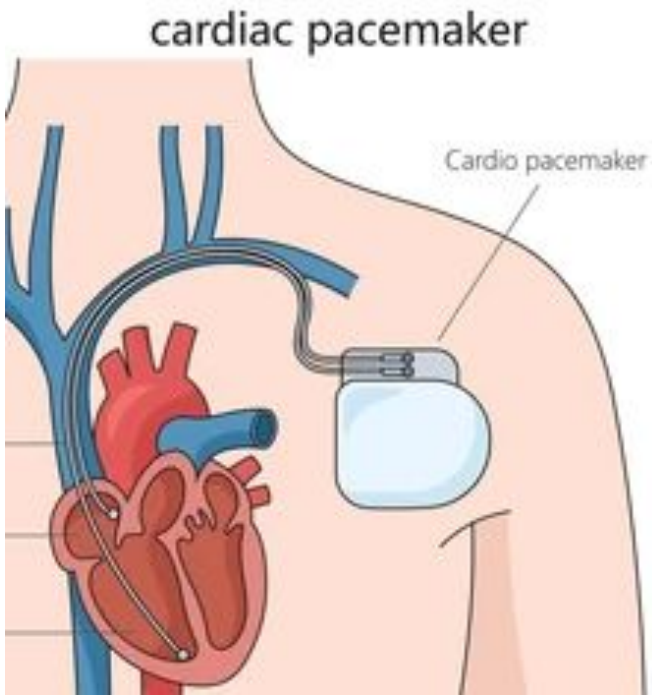
Ethernet Router

Viasat: Networking Driver

Motivation



First Generation
17 Unique ECUs



NEURALINK

Embedded devices are ubiquitous and insecure.

Embedded devices **can** be ubiquitous and **secure.**

Outline

Background

1. What is Tock OS
2. What is Thread

A Deep Dive into bringing OpenThead to Tock

3. Architecture Design
4. Implementation Challenges

Outline

Background

1. **What is Tock OS**
2. What is Thread

A Deep Dive into bringing OpenThead to Tock

3. Architecture Design
4. Implementation Challenges

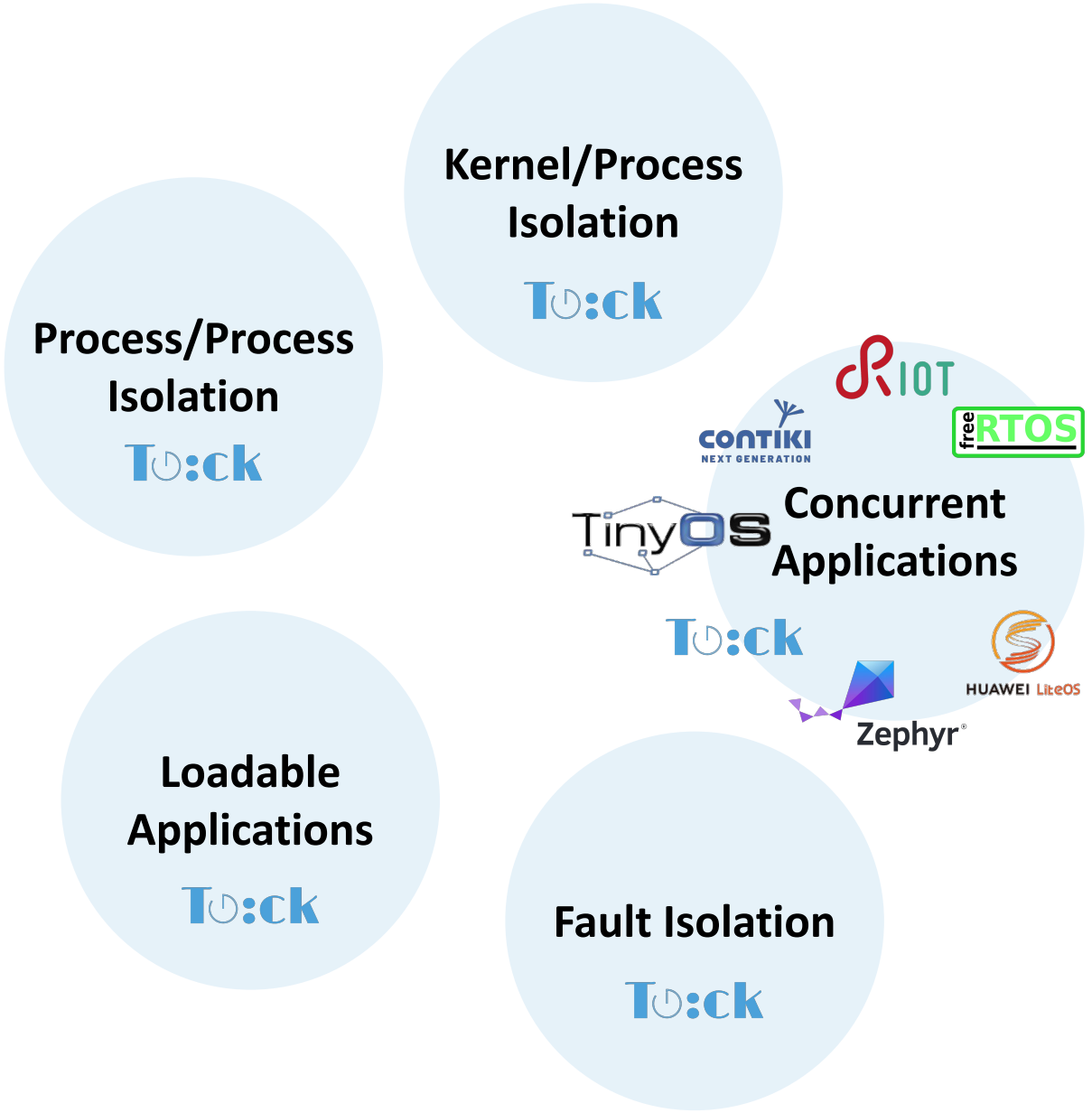
Embedded Operating Systems

Embedded OS Characteristics

(configured by default)

General purpose operating systems (e.g. Linux):

- Abstract and virtualize resources across processes.
- Isolate a process' memory r/w and faults to the given process.



What is Tock?

An embedded operating system designed for running
multiple concurrent, mutually distrustful applications
on low-memory and low-power microcontrollers.

TockOS

- A **preemptive** embedded OS (runs on MCUs)
  - Cortex-M
  - RISC-V
- Uses memory protection (**MPU** required)
- Has separate **kernel** and **user space**
  - most embedded OS have the one piece software philosophy
- Runs **untrusted apps** in user space
  - Apps written in C/C++ or Rust (any language that can be compiled)
- Kernel (and drivers) written in **Rust**

Functional Components

Applications

- User space processes
- Any language
- Independent executable

`System calls (syscall)`

Capsules

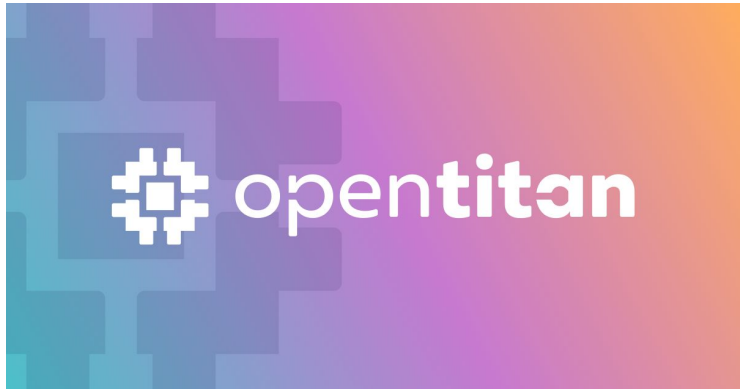
- Hardware independent drivers
- Inside the kernel
- Safe rust

`Hardware Interface Layer (HIL)`

HIL

- Abstraction over hardware dependent drivers
- Unsafe rust

Used by



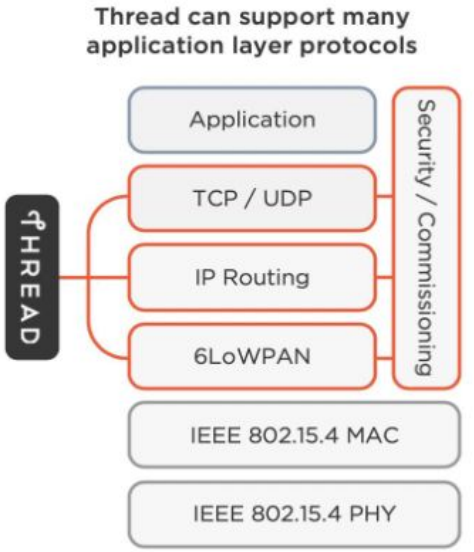
Outline

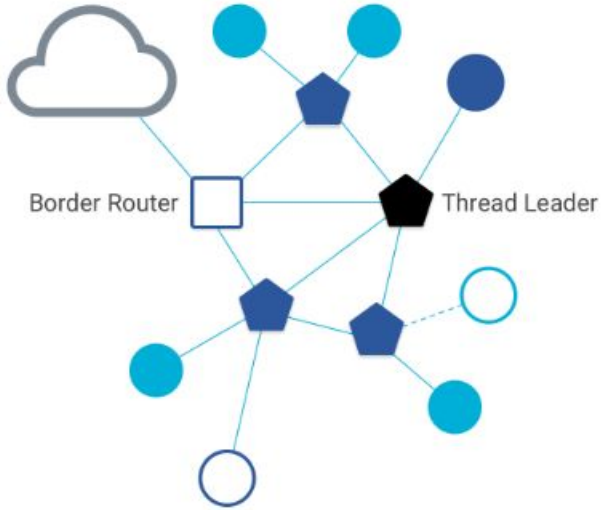
Background

1. What is Tock OS
2. **What is Thread**

What is Thread

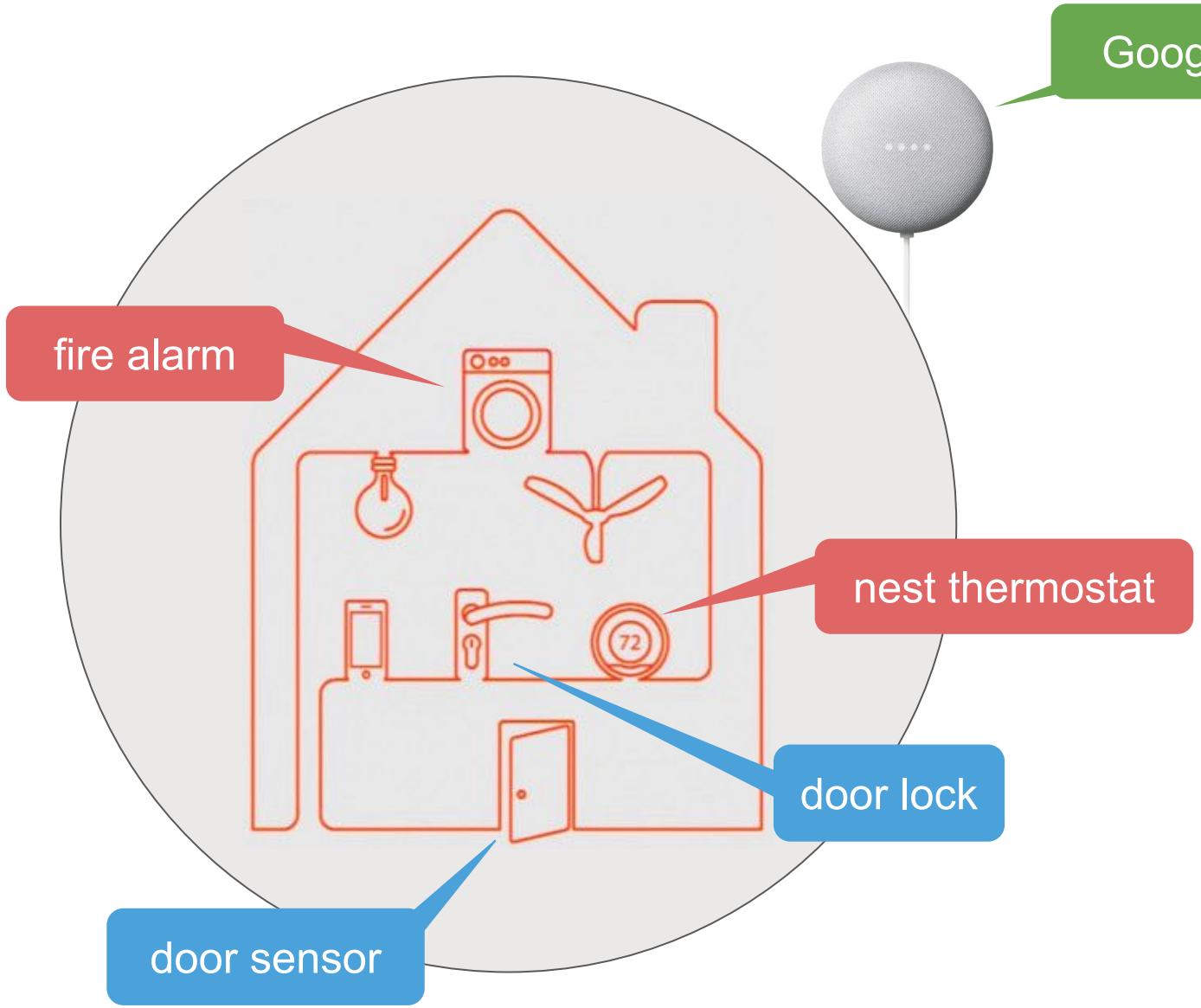
- Thread builds a networking layer on top of IEEE 802.15.4.
- Utilizes IPv6 routing for each node, allowing for interoperability with existing networks.
- Combines star and mesh topologies to provide a robust network.





[1] Thread Group - Thread 1.3.0 Webinar; 9-29-22

What is Thread?



Border Router

Router Node

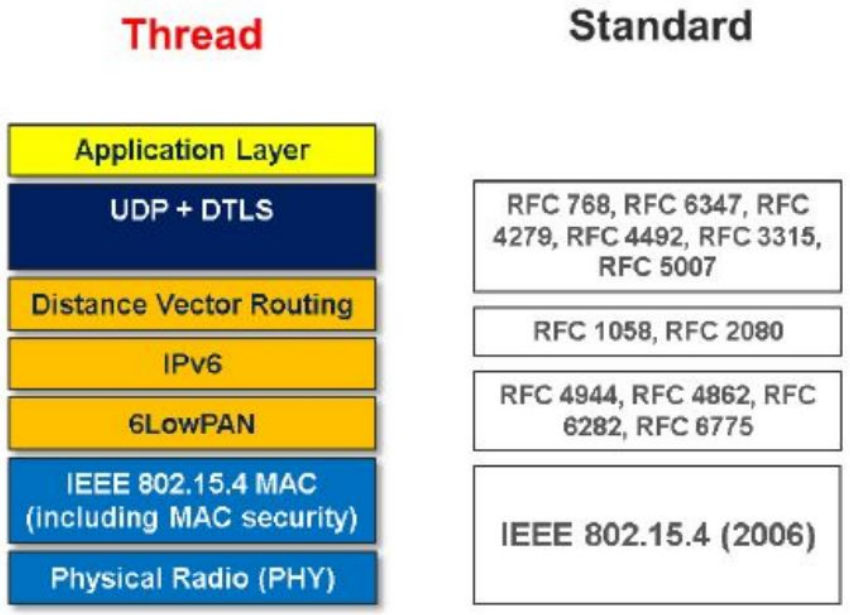
Child Node

(Image of House) https://www.the-ambient.com

Introduction to Thread

Why Thread instead of other IoT technologies?

- Simple network installation, start up, and operation
- Secure joining, and encrypted and secure communications
- Range. Mesh networking provides sufficient range to cover installation
- No single point of failure, and Self-healing
- Low power
- Build on existing standards and silicon
- Standards based and Royalty free
    - Thread Group Membership required

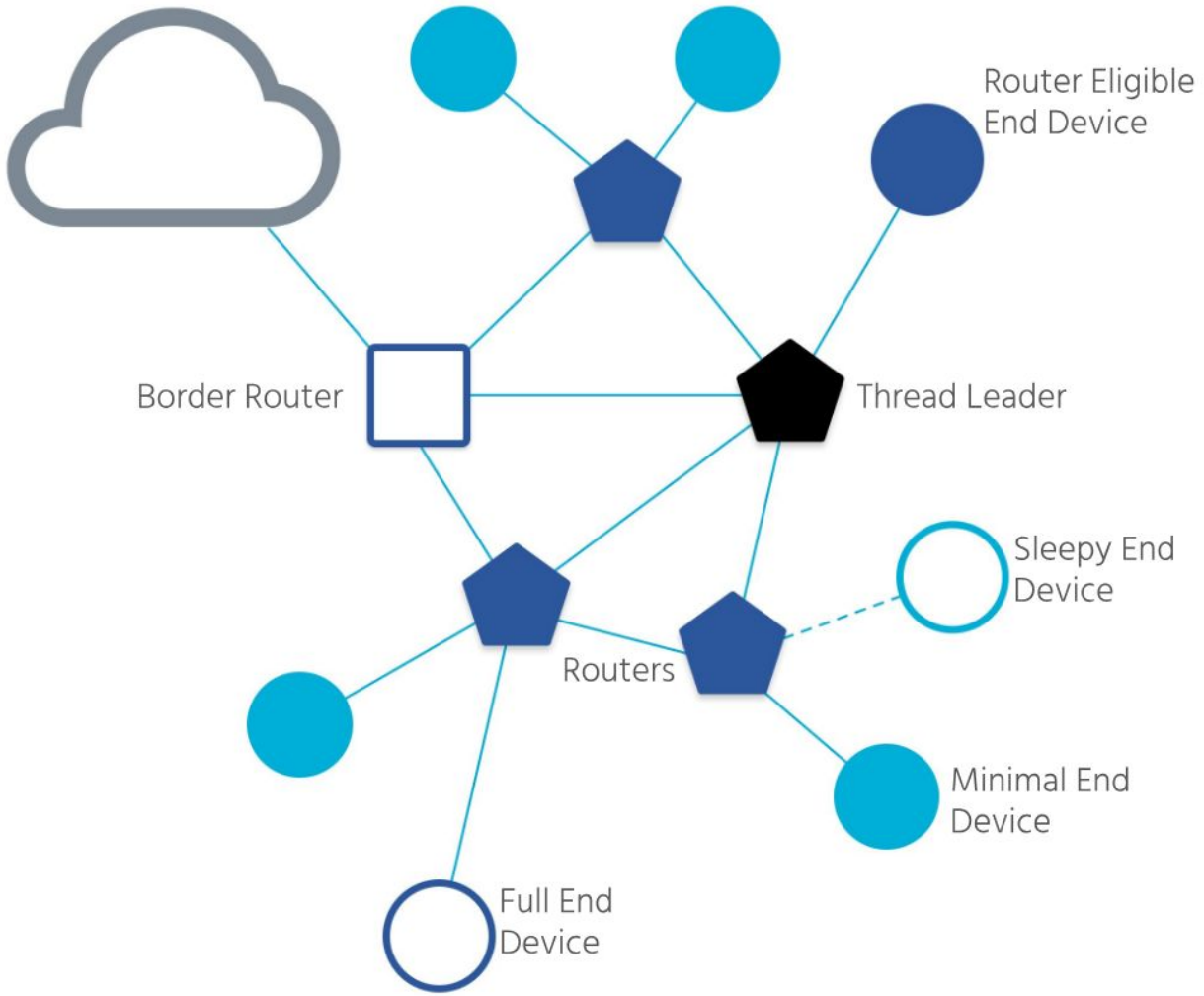


Device Types and Roles

- **Full Thread Device**
  - Border Routers
  - Routers
  - Router-eligible End Device (REED)
  - Full End Device
- **Minimal Thread Device**
  - Minimal End Device
  - Sleepy End Devices

Device roles are determined by the stack using Mesh Link Establishment messages with REEDs promoted to Routers as needed

All nodes have Link-Local and Mesh-Local IPv6 addresses. If a Border Router is present a Global Address may be configured using SLAAC or DHCPv6



Outline

Background

1. What is Tock OS
2. What is Thread

A Deep Dive into bringing OpenThead to Tock

3. Architecture Design

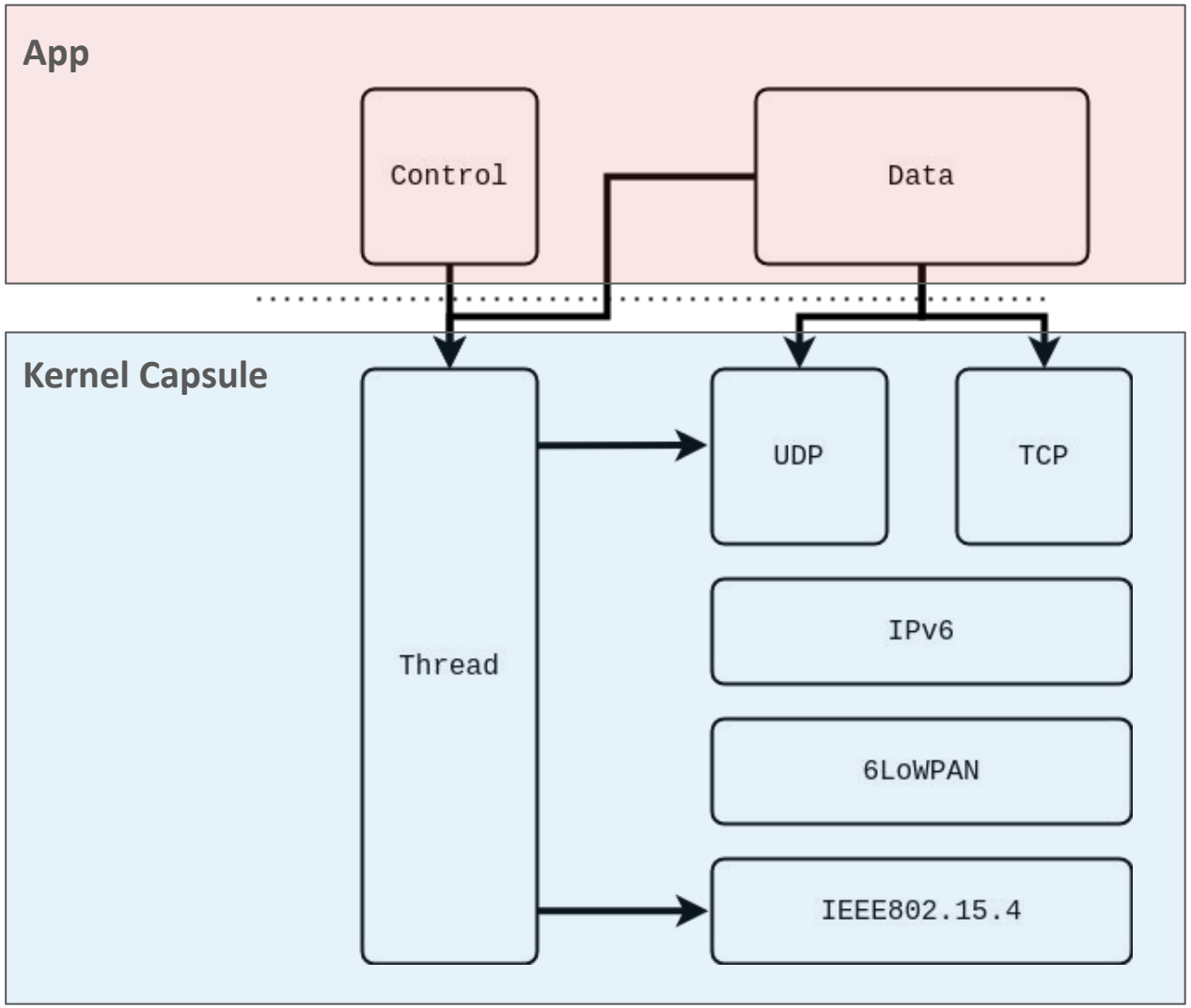
4. Implementation Challenges

Naive Idea

- Implement Thread in kernel

Issue:

- Reimplement whole spec?!



Tock and OpenThread

OpenThread: *de facto* open source Thread implementation that is actively maintained and contributed to.

Can we avoid reimplementing the Thread standard and instead just use this?

Tock and OpenThread Challenges

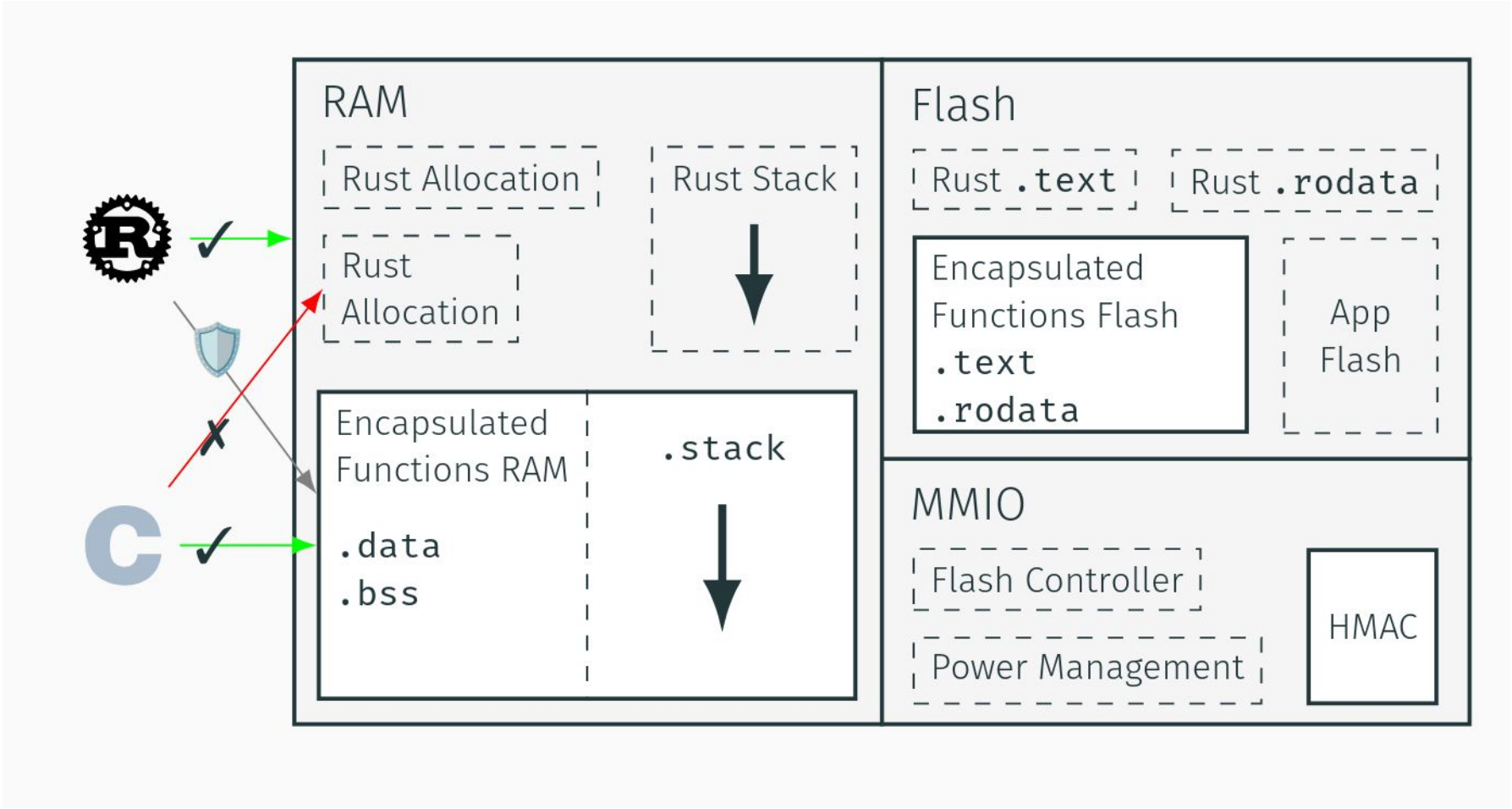
1. OpenThread is a large C library (foreign function interface with Rust kernel?).

Tock and OpenThread Challenges

1. OpenThread is a large C library (foreign function interface with Rust kernel?).

2. Tock's threat model excludes the use of external dependencies.

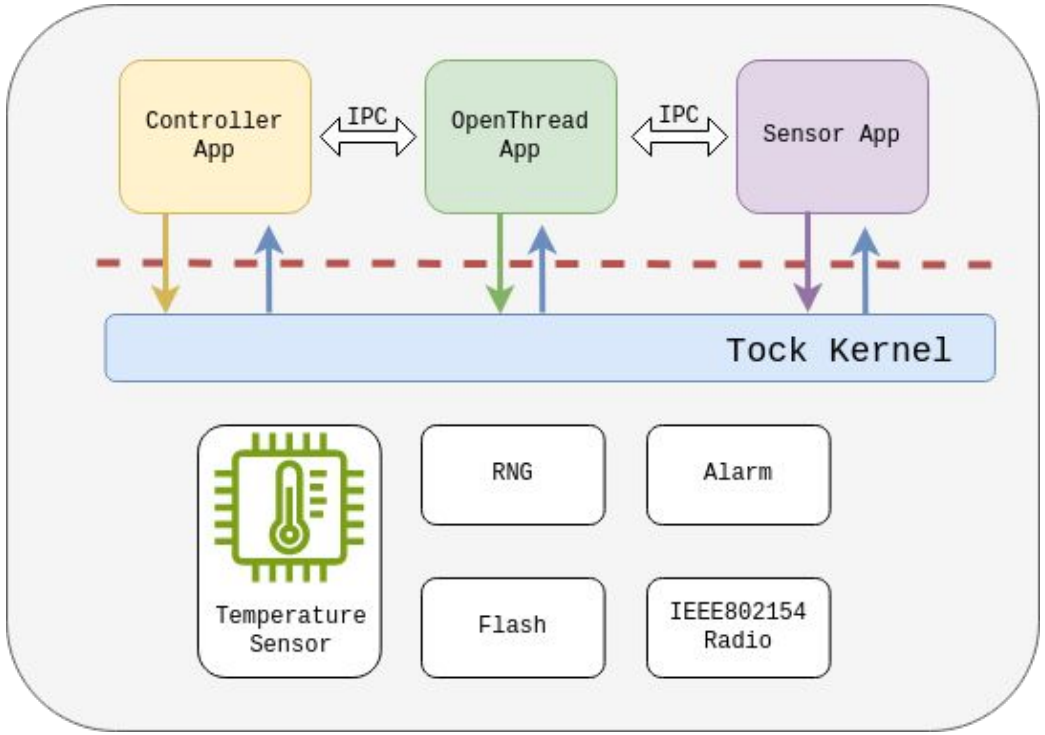
Idea: Hardware Isolate C/C++ in Kernel



Tock and OpenThread

Run OpenThread as a userspace application.

- OpenThread requires access to a radio, alarm, RNG, and flash
- Applications use OpenThread via IPC



Tock and OpenThread Challenges

1. OpenThread is a large C library (foreign function interface with Rust kernel?).

2. Tock's threat model excludes the use of external dependencies.

Tock and OpenThread Challenges

1. OpenThread is a large C library (foreign function interface with Rust kernel?).

2. Tock's threat model excludes the use of external dependencies.

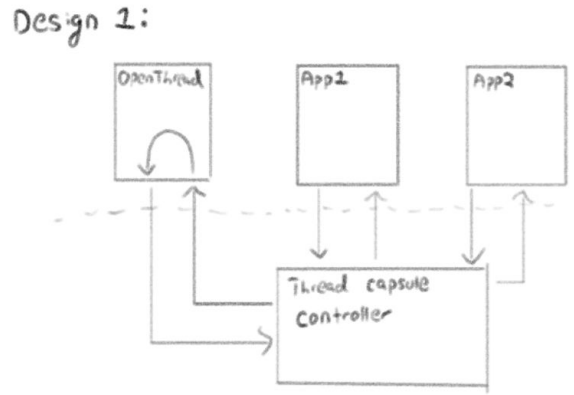
Tock and OpenThread Challenges

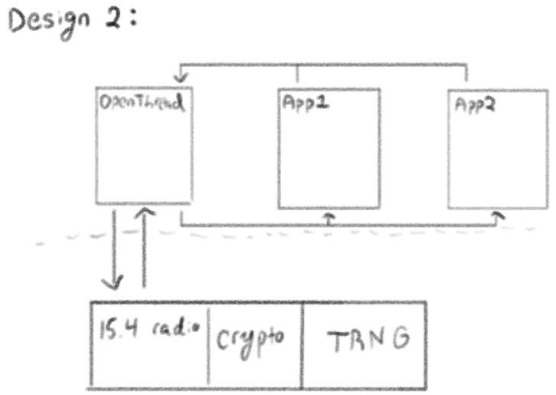
1. OpenThread is a large C library (foreign function interface with Rust kernel?).

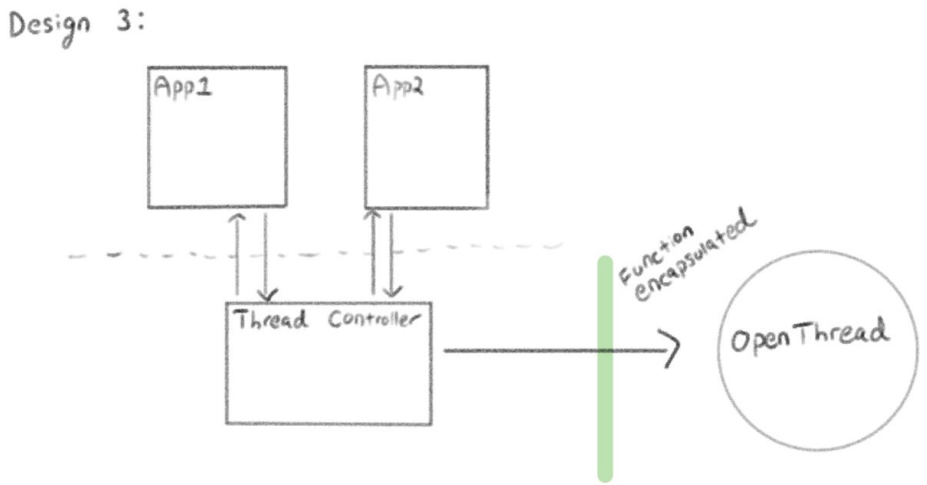
2. Tock's threat model excludes the use of external dependencies.

Tradeoffs

1. Rewrite it in Rust :)
   a. bug prone
   b. very inefficient to reimplement
2. Hardware isolated FFI (Encapsulated Functions)
   a. experimental, risky as only shown for crypto
   b. putting in kernel must be cooperative - cannot break isolation but bugs can hang kernel CPU or otherwise deadlock resources
3. **Userspace** (microkernel)
   a. several ways to design the IPC interface, or to add a dummy driver which interfaces with the app, but ultimately each design is isomorphic
   b. syscall and IPC overhead
   c. no need to trust openthread code







Stakeholders

We knew there are three pillars we need to address and convince the maintainers we uphold:

- Security
- Performance
- Efficient resource usage (flash)

Pause before we dive into code

We are going to start high level and go deep through
userspace ⟶ drivers ⟶ OS ⟶ architecture ⟶ registers

Outline

Background

1. What is Tock OS
2. What is Thread

A Deep Dive into bringing OpenThead to Tock

3. Architecture Design
4. **Implementation Challenges**

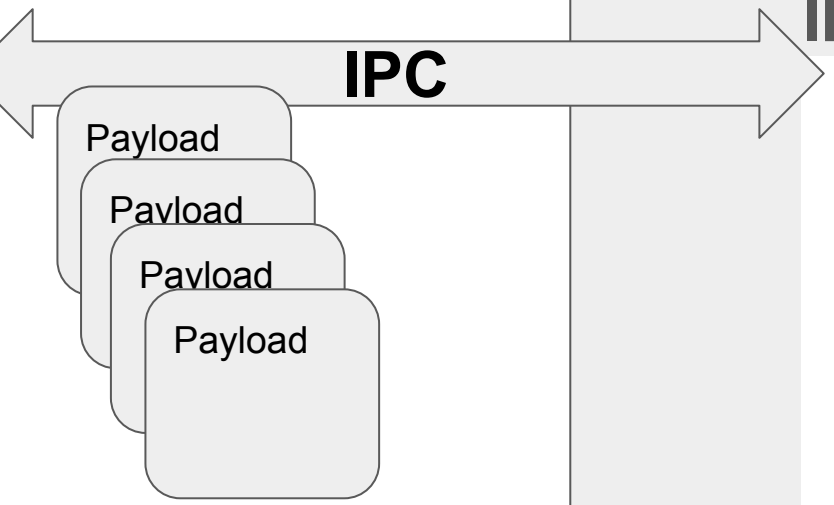
User Interface

OpenThread Process

Register IPC Service

```
// Register this application as an IPC service under its name:
ipc_register_service_callback("org.tockos.thread-tutorial.openthread",
                              openthread_ipc_callback,
                              NULL);
```

IPC Callback



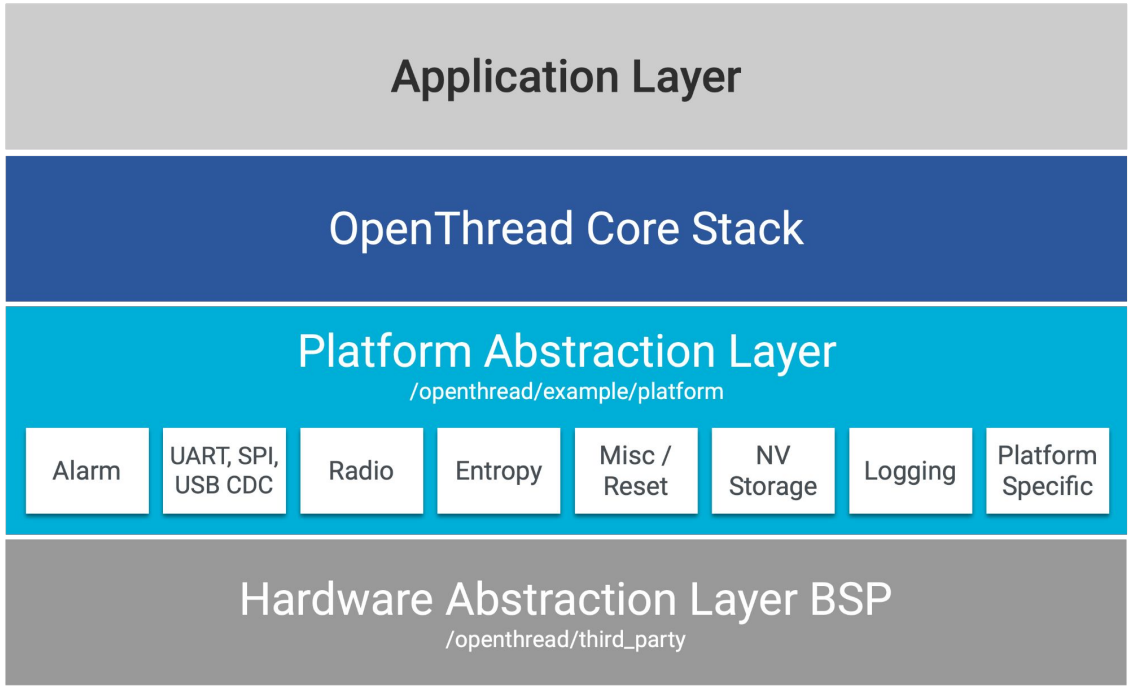

```
static void openthread_ipc_callback(int pid, int len, int buf,
                                    __attribute__((unused)) void* ud) {
  // Bounds check
  if (len < ((int) sizeof(prior_global_temperature_setpoint))) {
    ...
  }
  // Logic
  { ... }

  // Copy the current temperature into it.
  memcpy((void*) buf, &global_temperature_setpoint, sizeof(global_temperature_setpoint));

  // Notify the client that the temperature has changed.
  ipc_notify_client(pid);
}
```

Platform files

- Need to implement their headers
- Go from high level function such as modifying radio state to the exact registers to write



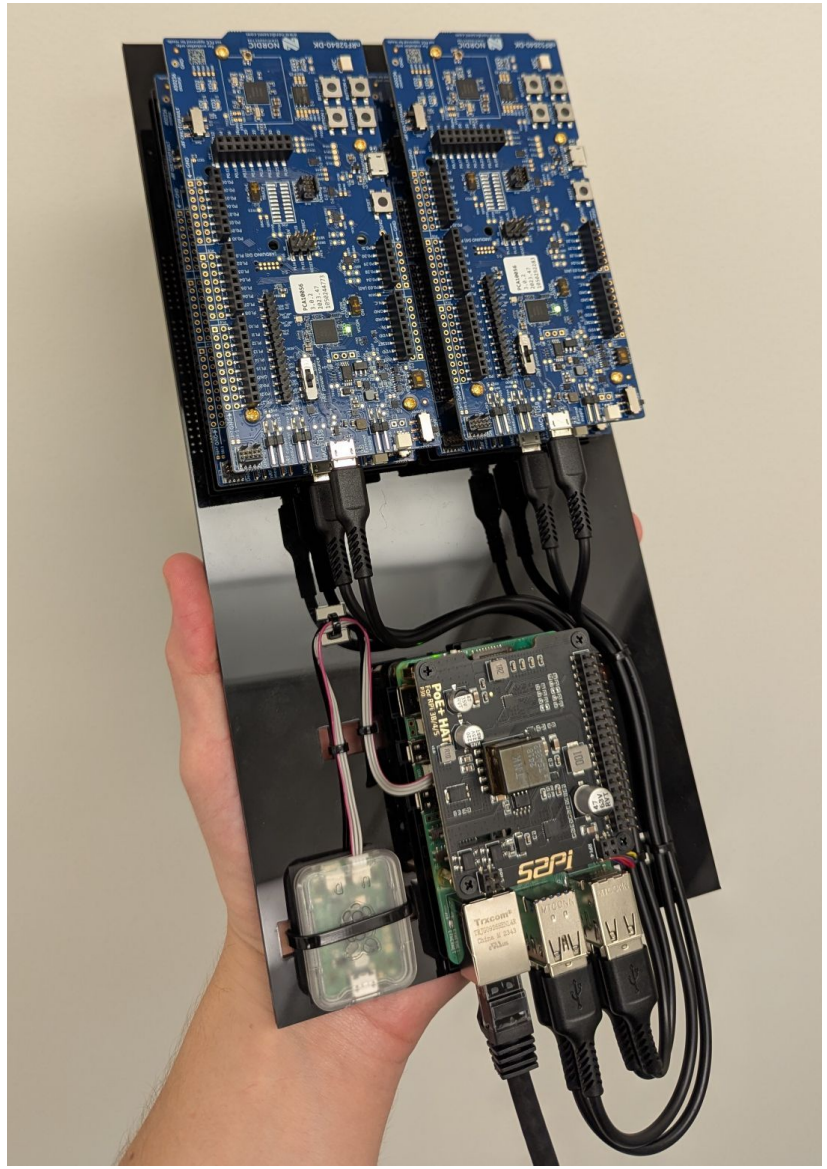
Example

```
/**
 * Fetches the value of a setting.
 *
 * Fetches the value of the setting identified
 * by @p aKey and write it to the memory pointed to by aValue.
 * It then writes the length to the integer pointed to by
 * @p aValueLength. The initial value of @p aValueLength is the
 * maximum number of bytes to be written to @p aValue.
 *
 * Can be used to check for the existence of
 * a key without fetching the value by setting @p aValue and
 * @p aValueLength to NULL. You can also check the length of
 * the setting without fetching it by setting only aValue
 * to NULL.
 *
 * Note that the underlying storage implementation is not
 * required to maintain the order of settings with multiple
 * values. The order of such values MAY change after ANY
 * write operation to the store.
 *
 * @param[in]      aInstance     The OpenThread instance structure.
 * @param[in]      aKey          The key associated with the requested setting.
 * @param[in]      aIndex        The index of the specific item to get.
 * @param[out]     aValue        A pointer to where the value of the setting should be written. May be set to NULL if
 *                               just testing for the presence or length of a setting.
 * @param[in,out]  aValueLength  A pointer to the length of the value. When called, this pointer should point to an
 *                               integer containing the maximum value size that can be written to @p aValue. At return,
 *                               the actual length of the setting is written. This may be set to NULL if performing
 *                               a presence check.
 *
 * @retval OT_ERROR_NONE             The given setting was found and fetched successfully.
 * @retval OT_ERROR_NOT_FOUND        The given setting was not found in the setting store.
 * @retval OT_ERROR_NOT_IMPLEMENTED  This function is not implemented on this platform.
 */
otError otPlatSettingsGet(otInstance *aInstance, uint16_t aKey, int aIndex, uint8_t *aValue, uint16_t *aValueLength);
```

It works…?

Test joining mesh net with reference implementation

- misbehavior after a few minutes of testing



Timer architecture diagram, layers

crashes, timer?

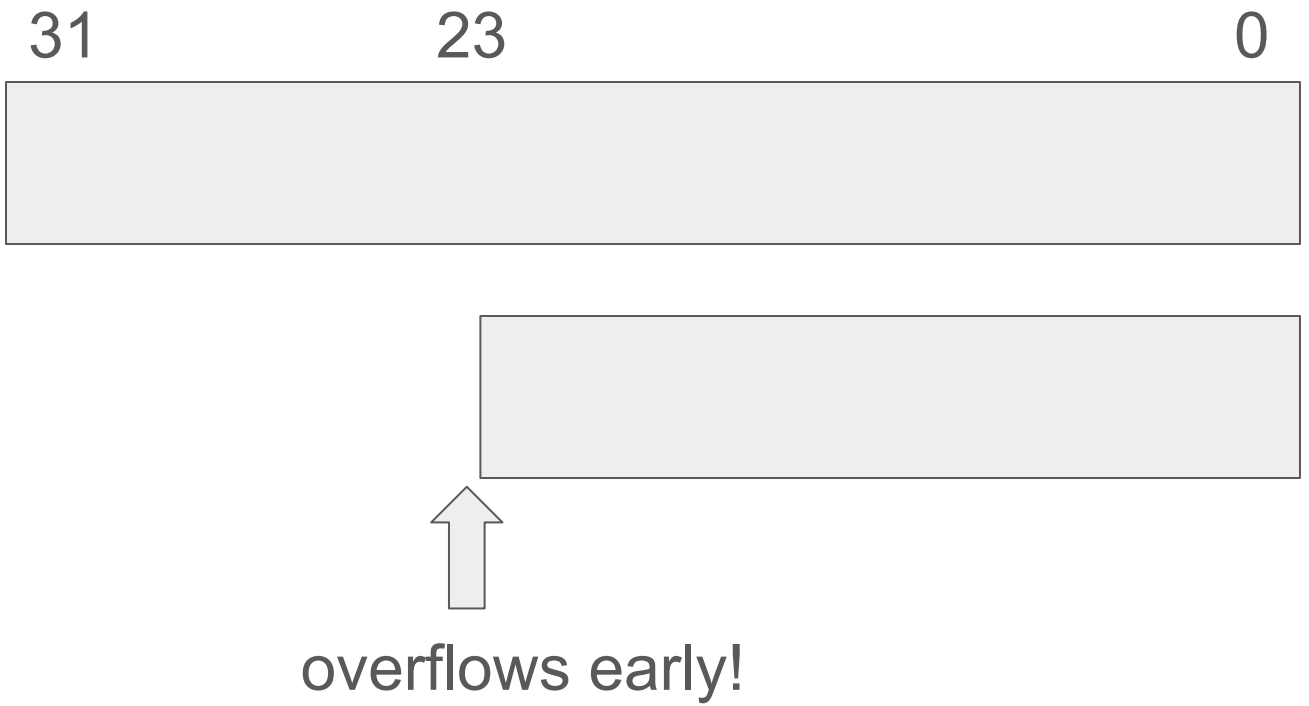
1. crash at 30 sec due to 24 bit overflow (what is prescaler, precision and energy tradeoff, interrupt at wrong time)
2. crash at 7 min due to 32 bit overflow (incorrect)
3. crash at [7min, inf) by bounding error manually
4. no crashes ever by formal verification

Bug 1

crash at 30 sec due to 24 bit overflow (what is prescaler, precision and energy tradeoff, interrupt at wrong time)

Github issues

The client should not assume anything about the underlying clock used by an implementation other than that it is running at sufficient frequency to deliver at least **millisecond granularity** and that it is a 32-bit clock (i.e. it will **wrap at 2^32** clock ticks).



Kernel hack fix

Must conform to the kernel ABI

- timers haven't changed since v1

Must be

- hardware agnostic
- architecture agnostic
- no regressions in user programs

» Kernel Time HIL

TRD: 105
Working Group: Kernel
Type: Documentary
Status: Draft
Obsoletes: 101
Author: Guillaume Endignoux, Amit Levy, Philip Levis, and Jett Rink
Draft-Created: 2021/07/23
Draft-Modified: 2021/07/23
Draft-Version: 1.0
Draft-Discuss: Github PR

Abstract

This document describes the hardware independent layer interface (HIL) for time in the Tock operating system kernel. It describes the Rust traits and other definitions for this service as well as the reasoning behind them. This document is in full compliance with TRD1.

1 Introduction

Microcontrollers provide a variety of hardware controllers that keep track of time. The Tock kernel organizes these various types of controllers into two broad categories: alarms and timers. Alarms continuously increment a clock and can fire an event when the clock reaches a specific value. Timers can fire an event after a certain number of clock ticks have elapsed.

Bug 2

crash at 7 min due to 32 bit overflow (incorrect)

Timers work by virtualizing a linked list

```
static void root_insert(libtock_alarm_ticks_t* alarm) {
  if (root == NULL) {
    root       = alarm;
    root->next = NULL;
    root->prev = NULL;
    return;
  }

  libtock_alarm_ticks_t** cur = &root;
  libtock_alarm_ticks_t* prev = NULL;
  while (*cur != NULL) {
    uint32_t cur_expiration = (*cur)->reference + (*cur)->dt;
    uint32_t new_expiration = alarm->reference + alarm->dt;
    if (!within_range(alarm->reference, cur_expiration, new_expiration)) {
      // insert before
      libtock_alarm_ticks_t* tmp = *cur;
      *cur        = alarm;
      alarm->next = tmp;
      alarm->prev = prev;
      tmp->prev   = alarm;
      return;
    }
    prev = *cur;
    cur  = &prev->next;
  }
  // didn't return, so prev points to the last in the list
  prev->next  = alarm;
  alarm->prev = prev;
  alarm->next = NULL;

}
```

2 cases

```
int libtock_alarm_in_ms(uint32_t ms, libtock_alarm_callback cb, void* opaque, libtock_alarm_t* alarm) {
  uint32_t now;
  int ret = libtock_alarm_command_read(&now);
  if (ret != RETURNCODE_SUCCESS) return ret;

  // If `dt_ms` is longer than the time that an alarm can count up to, then `timer_in` will
  // schedule multiple alarms to reach the full length. We calculate the number of full overflows
  // and the remainder ticks to reach the target length of time. The overflows use the
  // `overflow_callback` for each intermediate overflow.
  const uint32_t max_ticks_in_ms = ticks_to_ms(MAX_TICKS);
  if (ms > max_ticks_in_ms) {
    // overflows_left is the number of intermediate alarms that need to be scheduled to reach the target
    // dt_ms. After the alarm in this block is scheduled, we have this many overflows left (hence the reason
    // for subtracting by one). Subtracting by one is safe and will not underflow because we have already
    // checked that dt_ms > max_ticks_in_ms.
    alarm->overflows_left  = (ms / max_ticks_in_ms) - 1;
    alarm->remaining_ticks = ms_to_ticks(ms % max_ticks_in_ms);
    alarm->user_data       = opaque;
    alarm->callback        = cb;

    return libtock_alarm_at(now, MAX_TICKS, (libtock_alarm_callback)overflow_callback, (void*)(alarm),
                            &(alarm->alarm));
  } else {
    // No overflows needed
    return libtock_alarm_at(now, ms_to_ticks(ms), cb, opaque, &(alarm->alarm));
  }
}
```

timer interrupt

```
static void alarm_upcall(__attribute__ ((unused)) int   kernel_now,
                         __attribute__ ((unused)) int   scheduled,
                         __attribute__ ((unused)) int   unused2,
                         __attribute__ ((unused)) void* opaque) {
  for (libtock_alarm_ticks_t* alarm = root_peek(); alarm != NULL; alarm = root_peek()) {
    uint32_t now;
    libtock_alarm_command_read(&now);
    // has the alarm not expired yet? (distance from `now` has to be larger or
    // equal to distance from current clock value.
    if (alarm->dt > now - alarm->reference) {
      libtock_alarm_command_set_absolute(alarm->reference, alarm->dt);
      break;
    } else {
      root_pop();

      if (alarm->callback) {
        uint32_t expiration = alarm->reference + alarm->dt;
        alarm->callback(now, expiration, alarm->ud);
      }
    }
  }
}
```

on overflow

```
// The intermediate callback that handles overflows of alarm. This is used by
// `timer_in` to handle cases where `alarm_at` would normally be unable to
// support the whole timer length. `alarm_at` can only keep track of up to 2^32
// ticks.
static void overflow_callback(__attribute__ ((unused)) uint32_t now,
                              uint32_t                          last_timer_fire_time,
                              void*                             overflow_ud) {
  libtock_alarm_t* tock_timer = (libtock_alarm_t*)overflow_ud;

  if (tock_timer->overflows_left == 0) {
    // no overflows left, schedule last alarm with original callback
    libtock_alarm_at(last_timer_fire_time,
                     tock_timer->remaining_ticks,
                     tock_timer->callback,
                     tock_timer->user_data,
                     &(tock_timer->alarm));
  } else {
    // schedule next intermediate alarm that will overflow
    tock_timer->overflows_left--;

    libtock_alarm_at(last_timer_fire_time,
                     MAX_TICKS,
                     (libtock_alarm_callback) overflow_callback,
                     (void*) tock_timer,
                     &(tock_timer->alarm));
  }
}
```

Bug 3

crash at [7min, inf) by bounding error provably

Correctness is hard!

Lots of informal specification in comments

Deal with interleaved execution, race conditions, data structure corruption, worst case execution bounding

```
// General Notes:
// The TRD states that ref is needed to disambiguate case of very near
// and very far alarm.
//
// Case that is hard to reason about (and motivates need for ref)
// Consider the following |--------X-O--------| where O is now
// and X is the tick value the arm is to fire at. For this case,
// We do not know if a short alarm was set for a after now or if
// this is an alarm very far in the future.
//
// If this is the case of the "long timer", ref + dt  will be a
// "large value" and subsequently the value will be within the
// range of [ref, ref + dt)
// Example of this: |-----X-O---(REF)--| where X is (ref + dt)
//                  |-----X-(REF)-O----| where X is (ref + dt)
//
// In the case of the "short timer", ref + dt will be a "small value"
// and the value will not be within this range:
// Example of this: |----(REF)-X-O-----| where X is (ref + dt)
//
// The question remains, could this be simplied or done in a better way?
// It seems this could just be checked by seeing if REF < REF + DT to
// determine if it is a long vs short alarm (this probably is missing
// some edge cases. this might be interesting once we prove the correctness
// of this to see if a "simpler" calculation like this is also true. if
// it is not, we can then perhaps say "this is an example of what seems
// to be a correct solution, but fails to consider x,y,z edge cases etc".
// if it ends up being simpler, we can show that we found this simpler
// way that is logically equivalent but more easily understandable. Both
// seem valuable).

// Preconditions:
// 1. Timer is running
// 2. Ref <= now (reference is in the past)
// Things to prove:
// 1. enable alarm
// 2. If no alarm => set new alarm
//    If alarm => cancel previous and replace with new alarm
```

Proved upper bound of timer error

```
// Note the following code is mathematically equivalent to the following
// more readable conversion.
// ```
//   uint32_t seconds          = ms / 1000;
//   uint32_t leftover_millis = ms % 1000;
//   uint32_t millihertz       = frequency / 1000; // ticks per millisecond
//   return (seconds * frequency) + (leftover_millis * millihertz);
// ```
```

Math proof

```
// We can check the logical correctness of this conversion by doing dimensional
// analysis of the units.
// For the first half, `seconds` (sec) * `frequency` (ticks per second) = a quantity in ticks.
// And for the second half, `leftover_millis` (milliseconds) * `millihertz` (ticks per milliseco
// = a quantity in ticks.
// Thus, we know the calculation is logically correct as the units cancel out to return ticks.
//
// Although these are logically equivalent, this human readable implementation is
// faulty. I will prove this by (1.) showing an upper bound on the error of this conversion,
// and then (2.) explaining why this commented human readable implementation is
// equivalent to the true implementation.
```

```
static uint32_t ms_to_ticks(uint32_t ms) {
  uint32_t frequency;
  libtock_alarm_command_get_frequency(&frequency);

  // Note the following code is mathematically equivalent to the following
  // more readable conversion.
  // ```
  //   uint32_t seconds         = ms / 1000;
  //   uint32_t leftover_millis = ms % 1000;
  //   uint32_t millihertz      = frequency / 1000; // ticks per millisecond
  //   return (seconds * frequency) + (leftover_millis * millihertz);
  // ```
  // We can check the logical correctness of this conversion by doing dimensional
  // analysis of the units.
  // For the first half, `seconds` (sec) * `frequency` (ticks per second) = a quantity in ticks.
  // And for the second half, `leftover_millis` (milliseconds) * `millihertz` (ticks per millisecond)
  // = a quantity in ticks.
  // Thus, we know the calculation is logically correct as the units cancel out to return ticks.
  //
  // Although these are logically equivalent, this human readable implementation is
  // faulty. I will prove this by (1.) showing an upper bound on the error of this conversion,
  // and then (2.) explaining why this commented human readable implementation is
  // equivalent to the true implementation.
  //
  // Part 1:
  // The splitting of the input milliseconds has no error.
  uint32_t seconds         = ms / 1000;
  uint32_t leftover_millis = ms % 1000;
  // In other words, `seconds` + `leftover_millis` is always == `ms`.

  // The first half of the output converts the seconds component to ticks with no error.
  // This multiplication has no error because the board frequency is given in Hertz (ticks per second).
  uint64_t ticks = (uint64_t) seconds * frequency;

  // Now that we have converted the full seconds part of `ms` to ticks, we must
  // convert the `leftover_millis`.
  // The only error introduced is by the division step. Division loses
  // all fractional components, so you lose 3 decimal significant figures by doing the division.
  // This does not matter as the remainder lost to integer division is a fraction of a
  // millisecond. It is especially important that this conversion is lower than the
  // true value of the conversion. In summary, the error in `ms_to_ticks` is from losing
  // significant figures, but this loss is less than 1 millisecond of error.
  //
  // Note that the division happens after the multiplication. Let us take a look at a
  // concrete example of a worst case conversion.
  // For a board with an oscillator frequency of 32,768Hz (NRF52840dk), observe the
  // error introduced by arithmetic in the "human readable" implementation of this
  // same line of the code:
  //   uint32_t millihertz      = frequency / 1000; // ticks per millisecond
  //   ticks += leftover_millis * millihertz;
  //
  //   // The worst case is when the remainder `leftover_millis` is `frequency` - 1
  //   ticks += 32767 * millihertz;
  // But what is the value of millihertz here? It is `32768 / 1000` = `32`! This is
  // 2.4% error, has only 2 significant figures, and, worst of all, the division causes the
  // denominator to be smaller than it should be. Dividing by a smaller number causes the
  // entire value of `ticks` to be an overestimate. This is a critical correctness issue.
  // Let's trace an example execution of calculating overflows in `timer_in`. The code uses
  // this line to calculate how many overflows are needed.
  //     const uint32_t max_ticks_in_ms = ticks_to_ms(MAX_TICKS);
  // If `ticks_to_ms` is an overestimate, then a timer set for slightly longer than the
  // true value of `MAX_VALUE` in ms will cause the check for `length of timer` <
  // `max_ticks_in_ms` to be true. This incorrect check will set an overflowed timer and fire
  // too early.
  uint32_t milliseconds_per_second = 1000;
  ticks += ((uint64_t) leftover_millis * frequency) / milliseconds_per_second;

  // Part 2:
  // The human readable and actual implementation are logically equivalent.
  // The only difference is the location of the division by 1000.
  //   leftover_millis * millihertz = leftover_millis * (frequency / 1000)
  //   = leftover_millis * frequency / 1000 = (leftover_millis * frequency) / 1000
  // Thus, by a series of equivalences, these are doing the same calculation.

  return ticks;
}
```

4 TIMER_WIDTH_TICKS + Δ

incorrect logic:

num_overflows = length_ms / overflow_ms;

4 TIMER_WIDTH_TICKS + Δ

incorrect logic:

num_overflows = length_ms / overflow_ms;

```
-    tv->tv_usec = (remainder * 1000) / (frequency / 1000);
+    tv->tv_usec = ((remainder * 1000) / frequency) * 1000;
```

Bug 4 onwards

Master’s thesis:
Working on automated proof of the timer code to a formal model

Recap

Userspace	timer virtualization, millisecond api, bound error
>	virtualized resources in PAL
Driver	API changes (not described),
	virtualizes app timers (not described)
Kernel	hide timer width
Register	use prescaler hack (not described)

Result: it works! Taught a class using Thread.

This Work - *Tock and Wireless Networking*

Bridging Tock's default isolation and robustness to provide a by default secure, robust, and updatable IoT/wireless sensor network platform.

Q&A

We ported a Thread networking stack to Tock OS without introducing any dependencies or changing the threat model.

Required making

- (virtualized) PAL
- writing an IPC interface
- fixing and adding features to Tock
- debugging the Thread spec and OpenThread implementation
- dealing with bugs reaching all the way into the hardware manual

References

Userspace Thread source tree
TRD105: Time - The Tock Book Time ABI specification

Communications App - The Tock Book
IPC - The Tock Book
IPC Tutorial - Tock Book

OSFC Tock OS verification
https://github.com/orgs/openthread/discussions/9872 ask maintainers

OpenThread Tutorial IPC Code
Original Networking wg Thread porting planning documentation

Done in half the time?

constraints mean the project could not have been done faster

We are not the judges - it is the maintainers. So to upstream, they must be satisfied. Ultimately, they will maintain the code forever.

The solution must be the best solution on

- Security
- Performance
- Efficient resource usage (flash, like how timers are)

OR justify and prove that it is good enough

Half the time

- Security
  - Make its process privileged or run in kernel
    i. can arbitrarily corrupt Rust kernel guarantees
  - Nordic Semi supports bare metal, so we can just steal their implementations and allow OpenThread to write to registers instead of going through kernel APIs
  - compromised security options already exist

Backup Slides

Profiling overhead

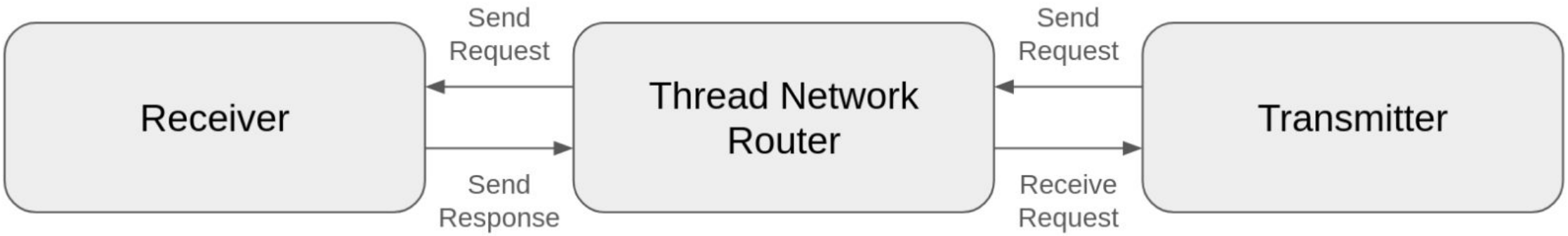


Figure 5: Round Trip Time

Github issues

Proposal 1:

I propose we concatenate the prescaler and alarm buffers.

Denied as it isn’t general to boards which actually do not have 32 bit timers. These are common on low power boards.

6.22 RTC — Real-time counter

The Real-time counter (RTC) module provides a generic, low power timer on the low-frequency clock source (LFCLK).

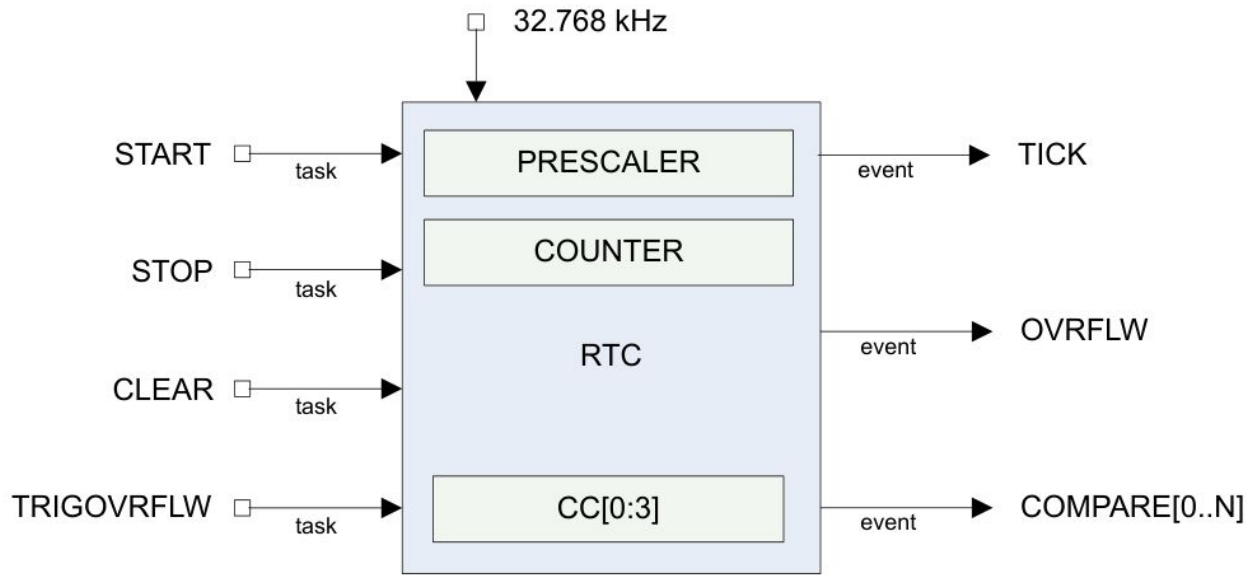


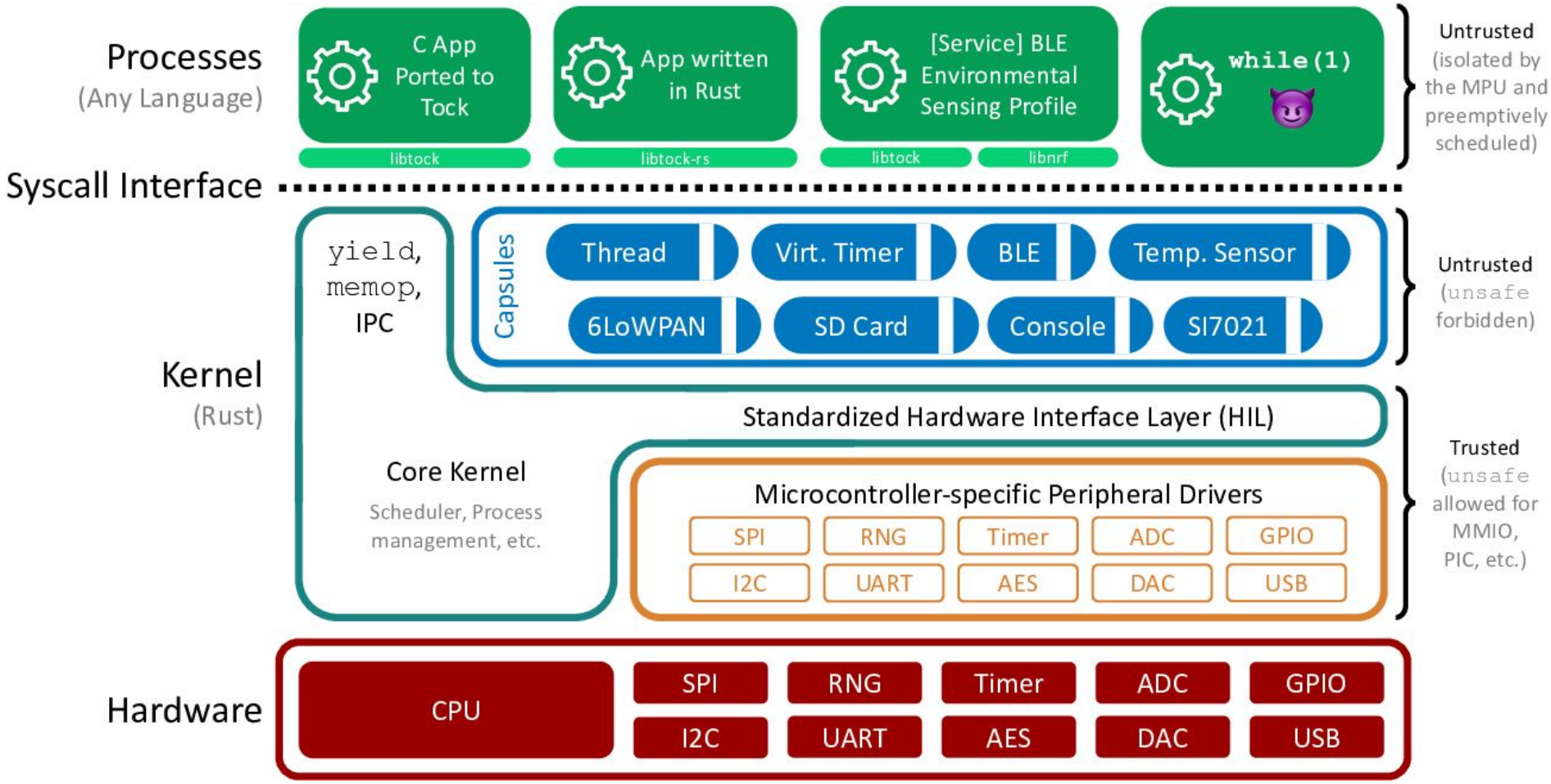
Figure 131: RTC block schematic

The RTC module features a 24-bit COUNTER, a 12-bit (1/X) prescaler, capture/compare registers, and a tick event generator for low power, tickless RTOS implementation.

Tock architecture slides

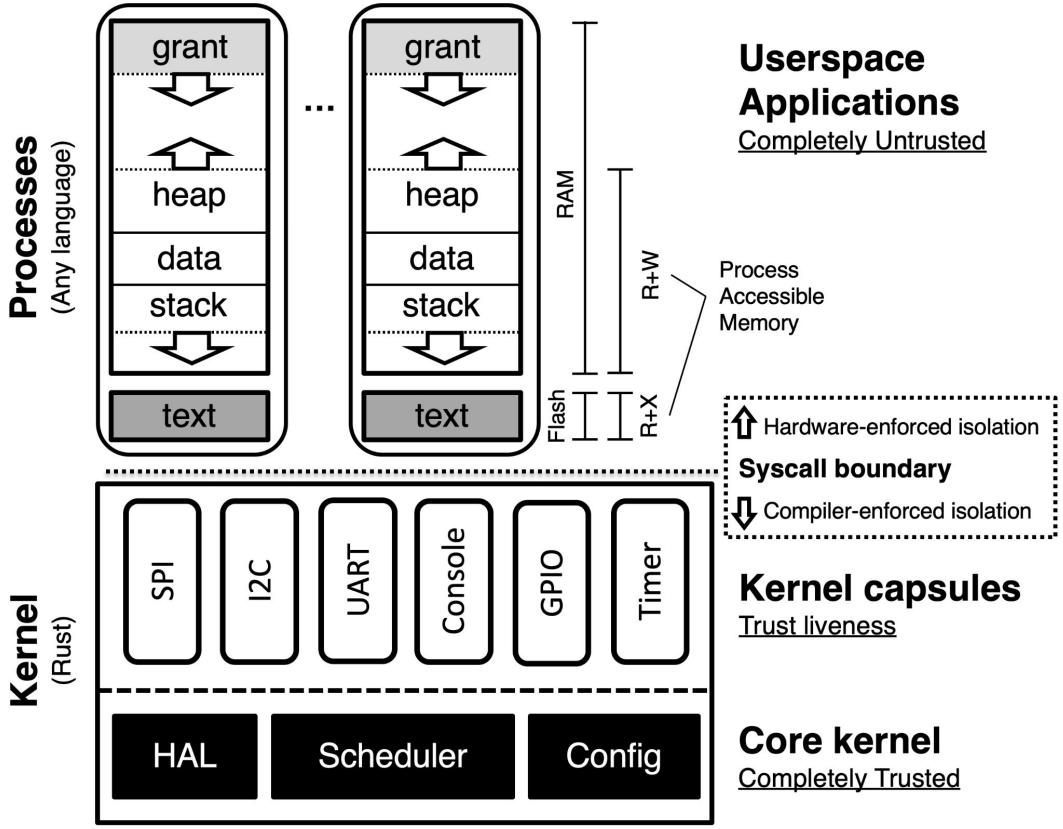
https://godsped.com/files/osfc24/slides.html?f=5#11

Architecture



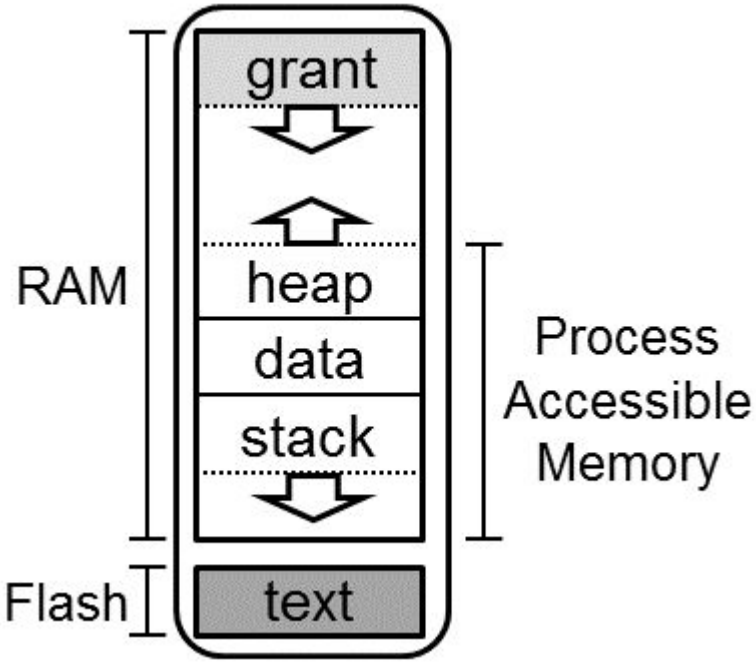
Kernel

- Non preemptive
  - Capsules (drivers) run to completion
  - Asynchronous API
- Does not allocate dynamic memory
  - Only static buffers
  - No out of memory errors in kernel
- Grants
  - Memory for capsules in user processes
  - Allocated at the start of a process (if possible)



Application (Process)

- Standalone executable
  - *compiled without TockOS kernel*
- Memory Protection
  - *MPU Regions*
- Can (*seg*)fault
- Relocatable code
  - if supported by the compiler
- IPC
  - service discovery
- Preemptible by the kernel



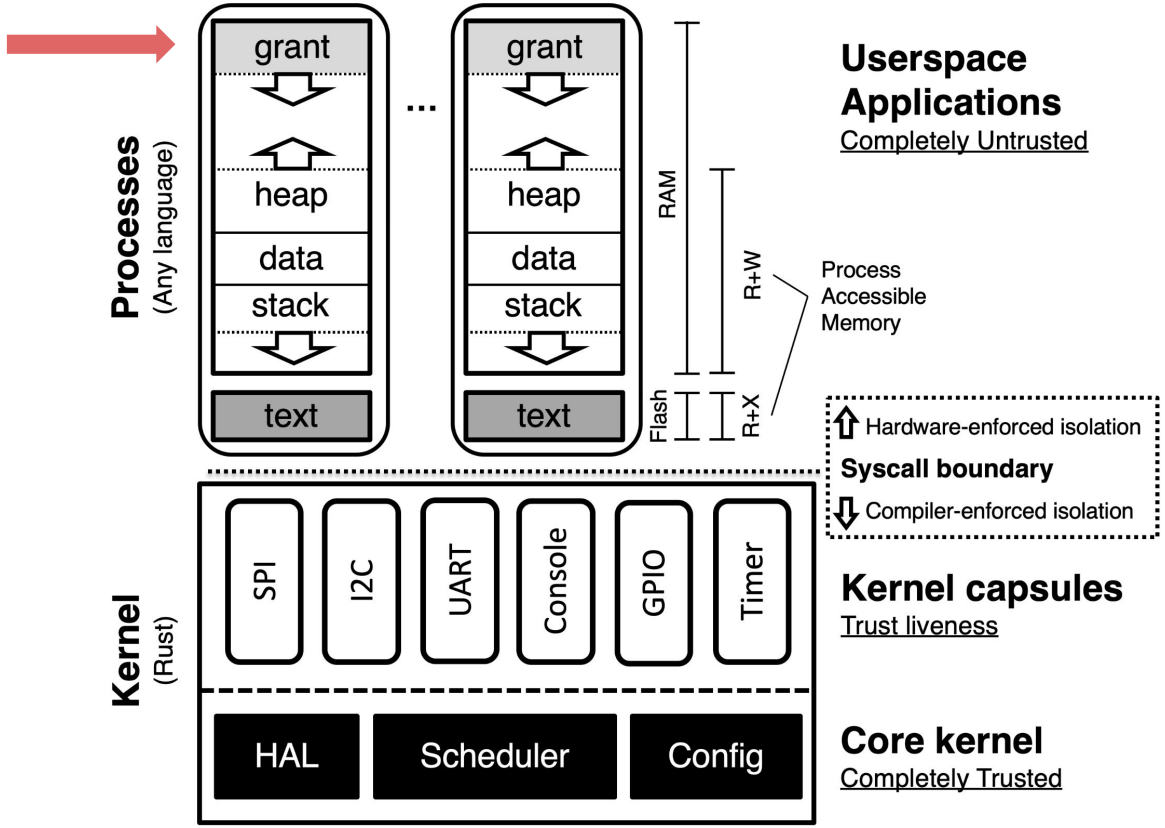
Grant Region

Microcontrollers are severely memory constrained.

Dynamic memory in the kernel has the potential to crash the kernel if memory is exhausted.

Grants are Tock's solution to this issue:

- Static memory used in kernel.
- Kernel can dynamically allocate memory in the application's grant as needed.



Key Takeaway: Grant memory exhaustion will only fault process, not the kernel.

Userland libraries

Libtock-C

- Stable (recommended)
- newlib
  - libc
  - libm
- libc+
  - *Lua53*
  - *LittlevGL*
- RV32: no suitable PIC support (as of May 2024)

Libtock-RS

- not stable yet
- core
  - active development
- No suitable PIC support in LLVM (as of May 2024)

Tockloader

- Manage Tock OS Application
- Uses TAB files
- Written in Python
- Built-in support for several boards
- *Small App Store*

Encapsulated Functions

- Another nice approach

IoT Technologies

Define up to Application Layer

- Zigbee
  - Zigbee Cluster Library to define “the Thing”
- Bluetooth Low Energy
  - GATT characteristics define “the Thing”
- Bluetooth Mesh
  - BT Mesh Models
- Z-Wave
  - Implements its own Transport and Application layers

Generally require an Application Gateway to translate messages to/from technology format

Define up to Transport Layer

- Wi-Fi
  - Ubiquitous, IP based
- 6LoWPAN
  - IP based
  - Compressed IPv6 over 802.15.4 (for this talk)
- Thread
  - Builds on 6LoWPAN, adding Routing, Security and Commissioning, and Certification programs

IP based Transport Layer technologies allow for multiple Applications to interoperate and co-exist on the same network