Embedded Operating Systems

Lecture 10

Embedded Operating Systems

usually called RTOS

- The purpose of an operating system
  - Abstractions
  - System calls
- Embedded Operating Systems
  - Real Time
- Tock OS

Operating System

the purpose of and OS

Bibliography

for this section

Andrew Tanenbaum, *Modern Operating Systems (4th edition)*

- Chapter 1 - *Memory Management*
  - Subchapter 1 - *Introduction*
    - Subchapter 1.1 - *What is an operating system?*
    - Subchapter 1.6 - *System calls*
    - Subchapter 1.7 - *Operating system structure*

Operating System

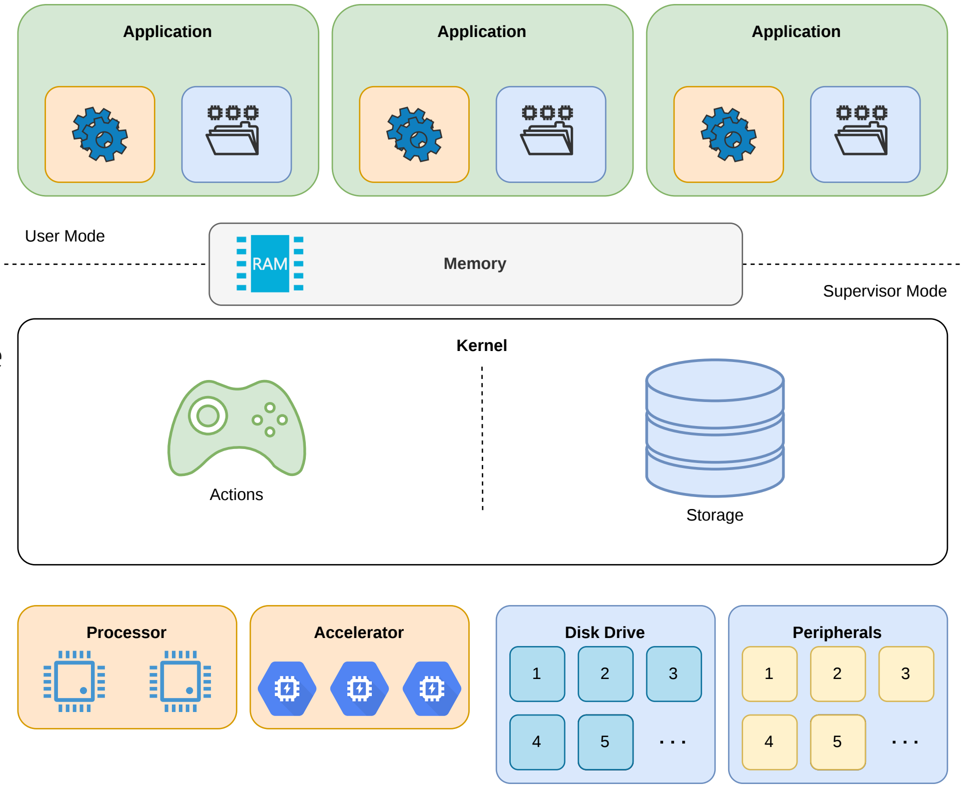
the main role

Allow Portability

- provides a hardware independent API
- applications should run on any hardware

Resources Management and Isolation

- allow applications to access resources
- prevent applications from accessing hardware directly
- isolate applications



Desktop and Server Operating Systems

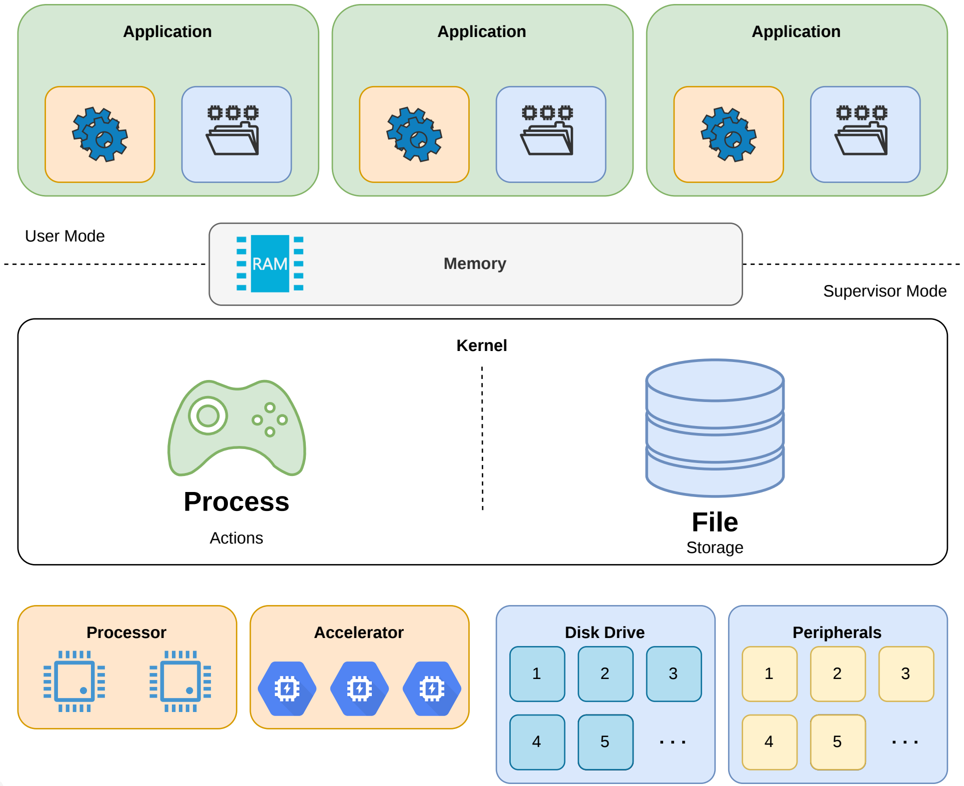
abstractions

Actions

- **process** and **threads**
- use the *Processor* and *Accelerators* (GPU, Neural Engine, etc)

Data

- everything is a file
- peripherals are viewed as files (*POSIX*)
  - `/sys/class/gpio/gpio5/direction`
  - `/sys/class/gpio/gpio5/value`



Embedded Operating Systems

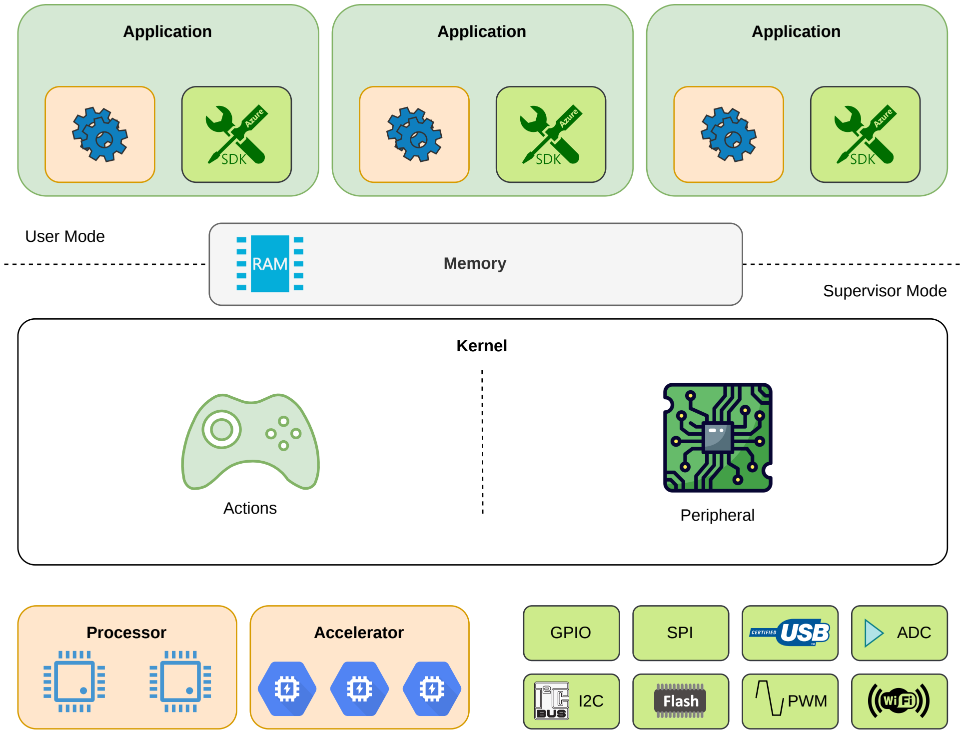
Actions

- **process** or **threads**
- use the *Processor* and *Accelerators* (Crypto Engines, Neural Engine, etc)

Peripheral

- provide a hardware independent API
- prevent processes from accessing the peripheral

usually the applications and the kernel are compiled together into a **single binary**



Scheduling Type

could a process stop the whole system?

Preemptive

- processes can be suspended by the scheduler
- a misbehaving process cannot stop the system

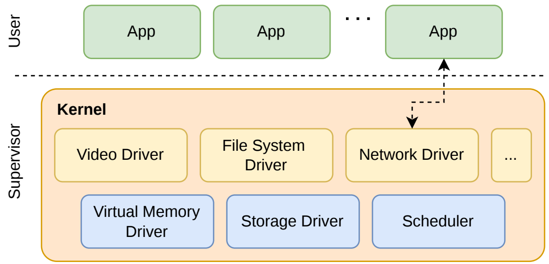
Cooperative

- processes **cannot be suspended** by the kernel
- a misbehaving process **can stop** the system

Kernel Types

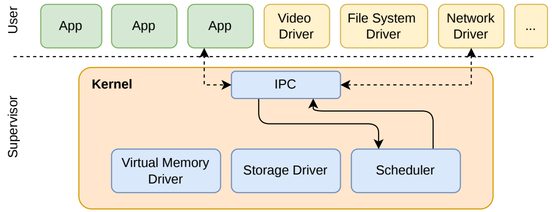
from the **kernel and drivers** point of view

Monolothic



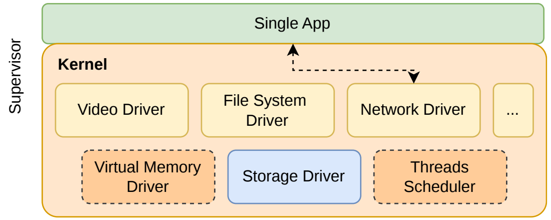
- all drivers in the kernel
- Windows, Linux, MacOS

Microkernel



- all drivers are applications
- Minix

Unikernel



- the kernel is bundled with all the drivers and one single application
- Unikraft/Linux
- Most of the microcontroller RTOSes

System Call

the OS API

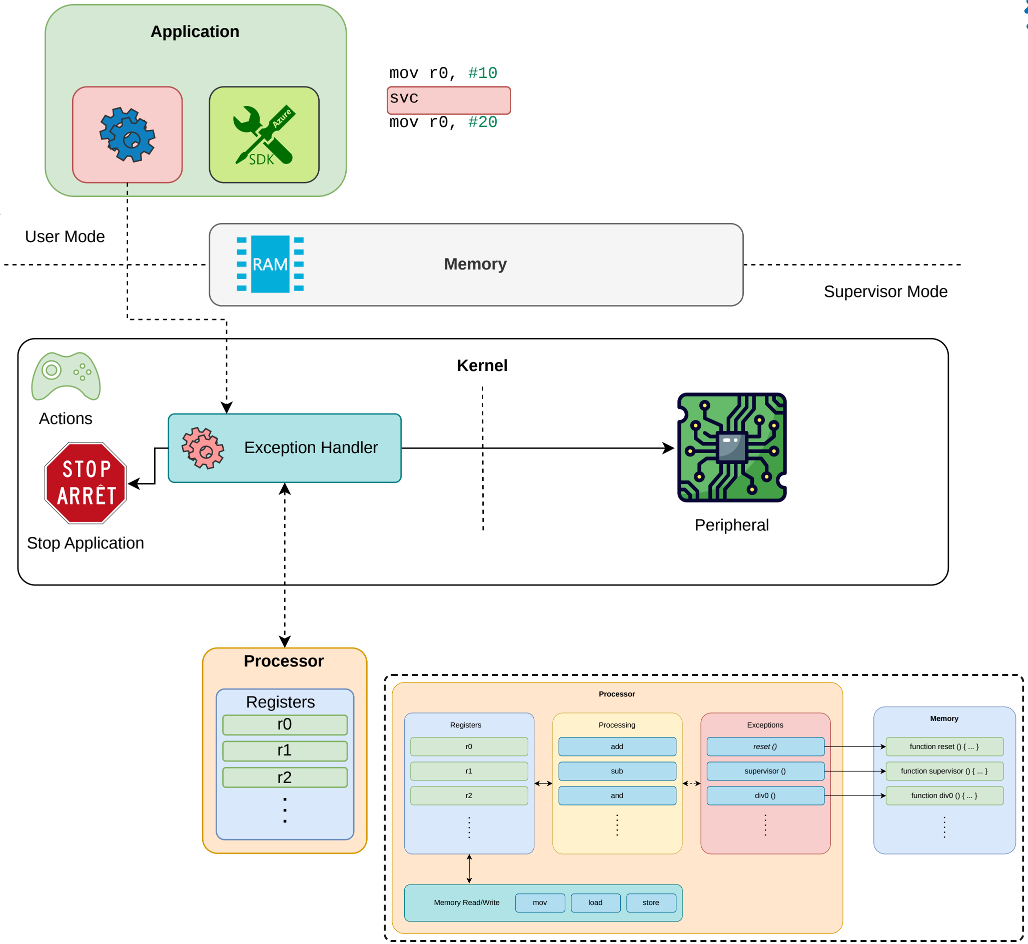
accessing a peripheral can be **performed** only **by the OS**

The application:

1. puts values in the registers
2. triggers an exception
   - `svc` instruction for ARM

The OS:

1. looks at the registers and determines what the required action is
2. performs the action
3. puts the return values into the registers



Embedded Operating Systems

aka Real-Time Operating Systems (RTOS)

Bibliography

for this section

Alexandru Radovici, Ioana Culic, *Getting Started with Secure Embedded Systems*

- Chapter 2 - *Embedded systems software development*

Embedded Operating Systems

- small OSes that run on microcontrollers
- most of the times called *Real Time OS* (*RTOS*)
- applications are similar to *threads* (are considered friendly)
- the whole system is compiled into a single binary
- similar to frameworks

Real Time?

upper bound

- **real time** means **performing** an action **always** in a **deterministic** amount of **time**
- the amount of time can be large
- **low latency** means that the amount if time must be small

The industry often uses real time interchangeably low latency.

Most Used

OS	Owner	Description
FreeRTOS	Amazon	Oldest RTOS, heavily used in the industry.
SafeRTOS	High Integrity Systems	Certified for functional safety, based on FreeRTOS.
Zephyr	Linux Foundation	*Linux'es little brother*, has an API inspired by Linux, is getting traction.

Tock OS

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

Bibliography

for this section

Alexandru Radovici, Ioana Culic, *Getting Started with Secure Embedded Systems*

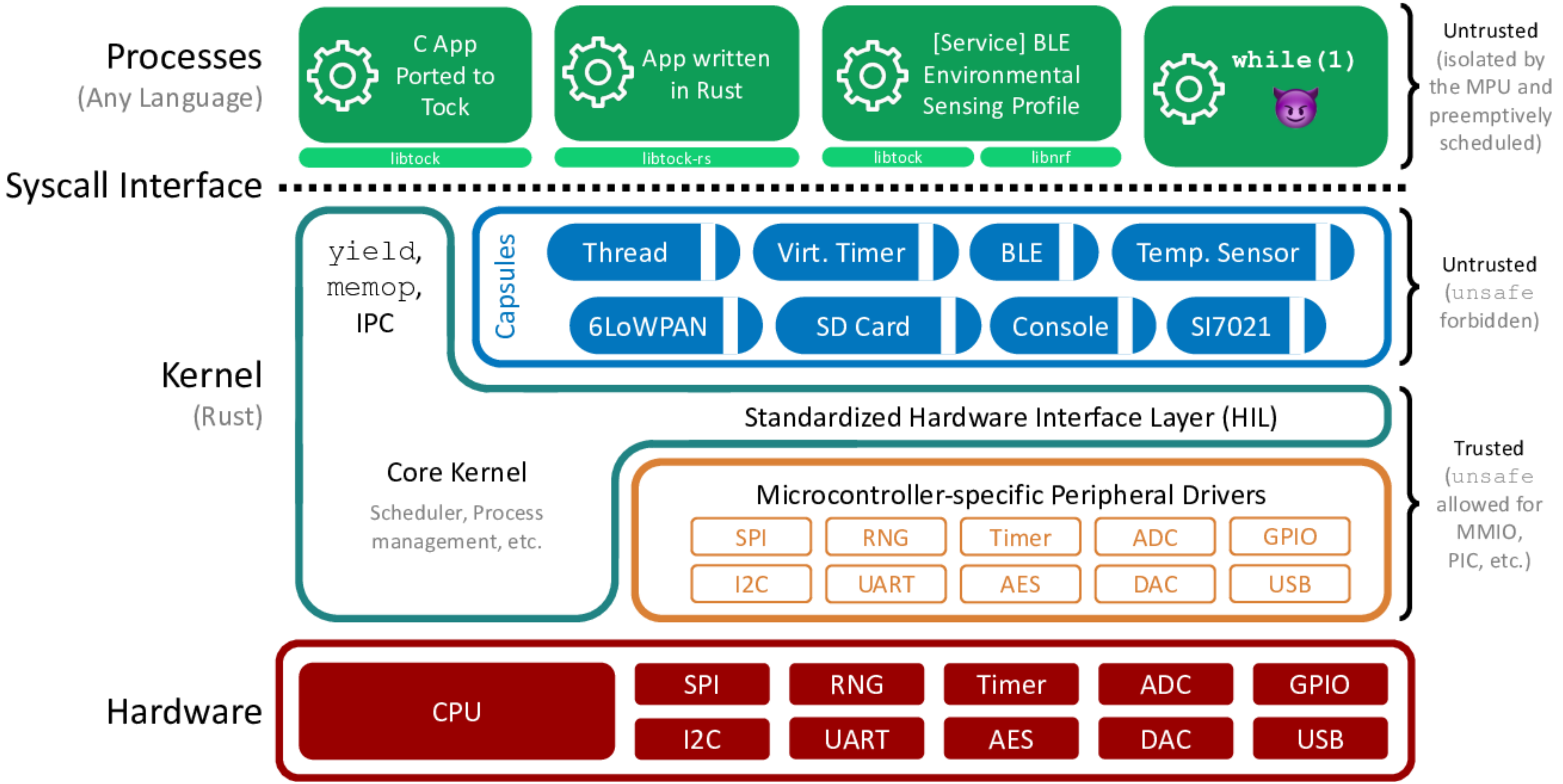
- Chapter 3 - *The Tock system architecture*

Tock OS

an embedded operating systems that works like a desktop or server one

- 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
- **Hybrid** architecture
- Kernel (and drivers) written in Rust
- Apps written in C/C++ or Rust (any language that can be compiled)

The Stack

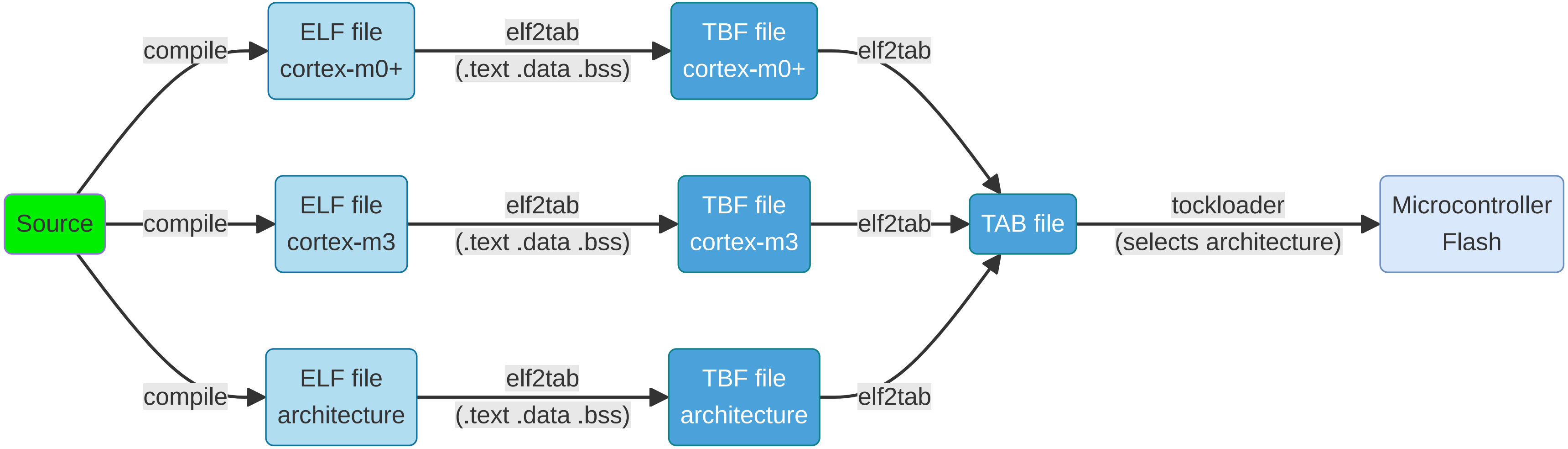


Processes
(Any Language)
C App Ported to Tock
App written in Rust
[Service] BLE Environmental Sensing Profile
while(1)
libtock
libtock-rs
libtock
libnrf
Untrusted (isolated by the MPU and preemptively scheduled)
Syscall Interface
Kernel
(Rust)
yield, memop, IPC
Capsules
Thread
Virt. Timer
BLE
Temp. Sensor
6LoWPAN
SD Card
Console
SI7021
Untrusted (unsafe forbidden)
Standardized Hardware Interface Layer (HIL)
Core Kernel
Scheduler, Process management, etc.
Microcontroller-specific Peripheral Drivers
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB
Trusted (unsafe allowed for MMIO, PIC, etc.)
Hardware
CPU
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB

Processes

separate binaries

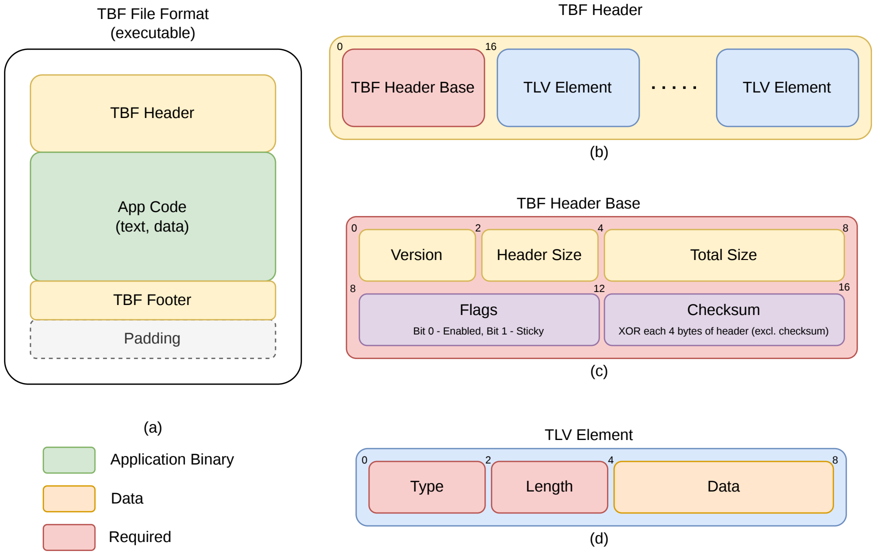
- compiled separately from the kernel
- written in any language that compiles (C, Rust, ...)
- saved into the *Tock Binary Format* (*TBF*) / *Tock Application Bundle* (*TAB*)

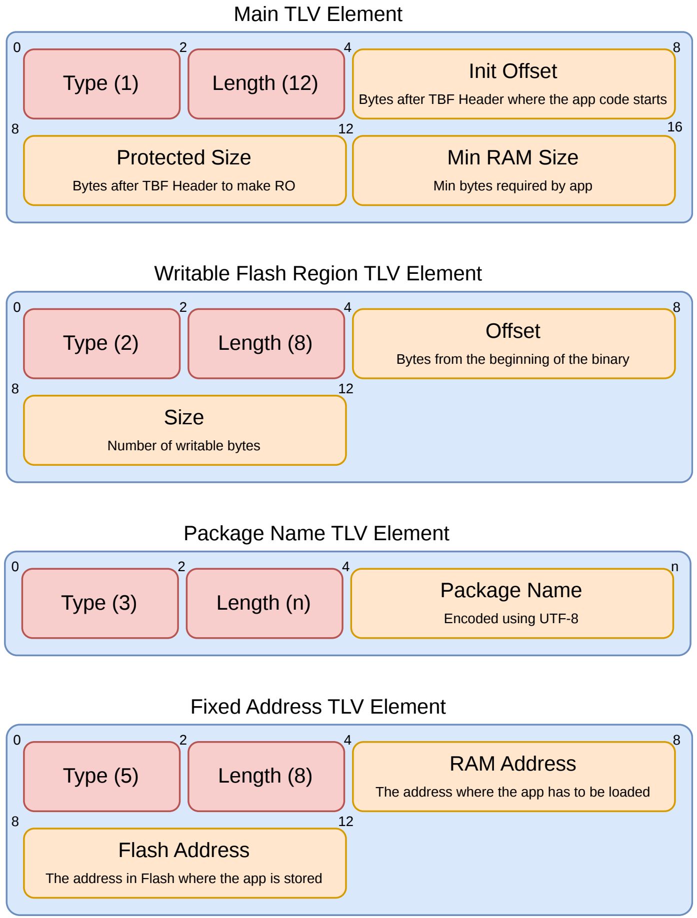


Tock Binary Format

stores

- **headers** about how to load the application
- the **binary code** and **data**
- **credential** footers





Memory Layout

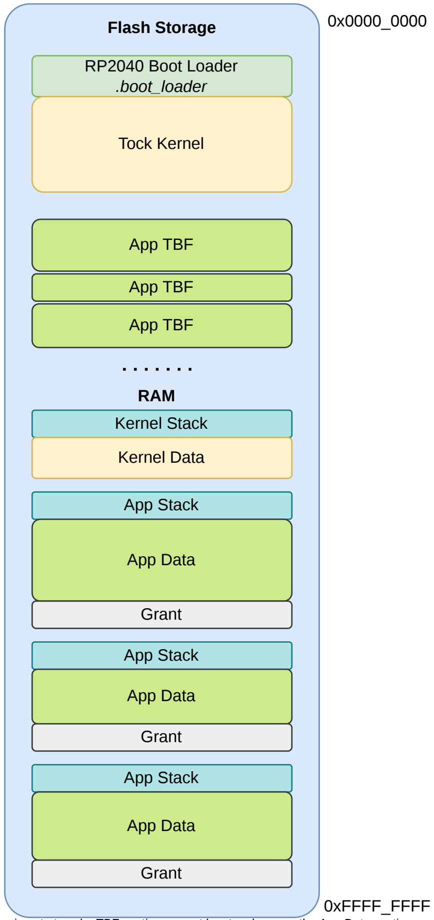
for the RP2040

Kernel

- is written in flash separated from the apps
- loads each app at boot

Applications

- each application TBF is written to the flash separately
- each application has a separate
  - *stack* in RAM
  - *grant* section where the kernel stores data about the app
  - *data* section in RAM



* drawing is not at scale, TBF sections are at least as large as the App Data sections

Memory Layout

for the RP2040 at runtime

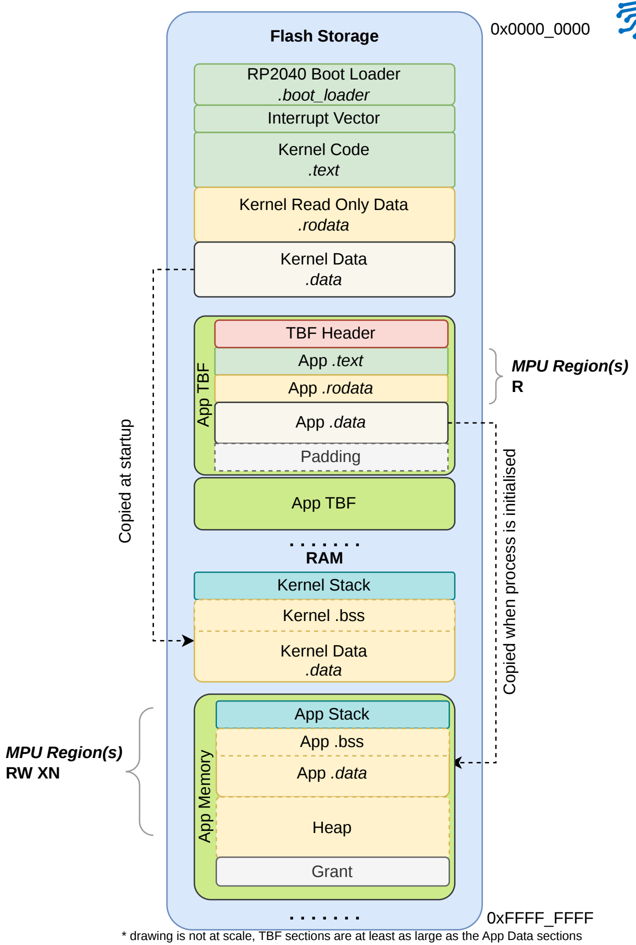
Kernel

- sets up the MPU every time it switches to a process

Applications

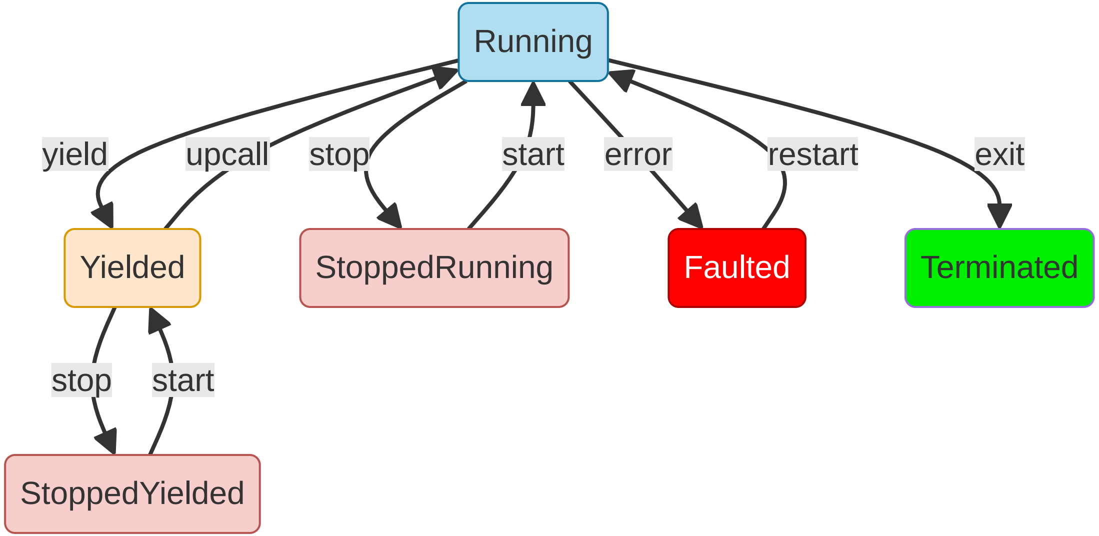
- can read and execute its code
- can read and write its *stack* and *data*
- can read and write the *allocated heap*

Applications are **not allowed** to access the **kernel's memory** or **the peripherals**.



Process States

- Tock runs only on *single core*
- *Running* state means the process is ready to run
- *Yielded* means the process waits for an event (*upcall*)
- *start* and *stop* are user commands
- a process is stopped only if the user asked it



Application API

libraries

Tock provides two libraries:

- `libtock-c` that is fully supported
- `libtock-rs` that is in development ⚠ [1]

---

1. Due to a Rust compiler issue, Rust applications are not relocatable. This means that developers have to know at compile time the load addresses for Flash and RAM. ↩

Example Application (C)

```c
#include <libtock-sync/services/alarm.h>
#include <libtock/interface/led.h>

int main(void) {
  // Ask the kernel how many LEDs are on this board.
  int num_leds;
  int err = libtock_led_count(&num_leds);
  if (err < 0) return err;

  // Blink the LEDs in a binary count pattern and scale
  // to the number of LEDs on the board.
  for (int count = 0; ; count++) {
    for (int i = 0; i < num_leds; i++) {
      if (count & (1 << i)) {
        libtock_led_on(i);
      } else {
        libtock_led_off(i);
      }
    }

    // This delay uses an underlying alarm in the kernel.
    libtocksync_alarm_delay_ms(250);
  }
}
```

Example Application (Rust)

```rust
//! A simple libtock-rs example. Just blinks all the LEDs.

#![no_main]
#![no_std]

use libtock::alarm::{Alarm, Milliseconds};
use libtock::leds::Leds;
use libtock::runtime::{set_main, stack_size};

set_main! {main}
stack_size! {0x200}

fn main() {
    if let Ok(leds_count) = Leds::count() {
        loop {
            for led_index in 0..leds_count {
                let _ = Leds::toggle(led_index as u32);
            }
            Alarm::sleep_for(Milliseconds(250)).unwrap();
        }
    }
}
```

Faults

similar to segfaults

- the kernel and apps can fault
- a detailed debug message can be displayed
- due to MPU usage Tock apps fault on:
  - trying to access memory outside its data (includes peripheral access)
  - stack overflow
  - trying to perform privileged operations

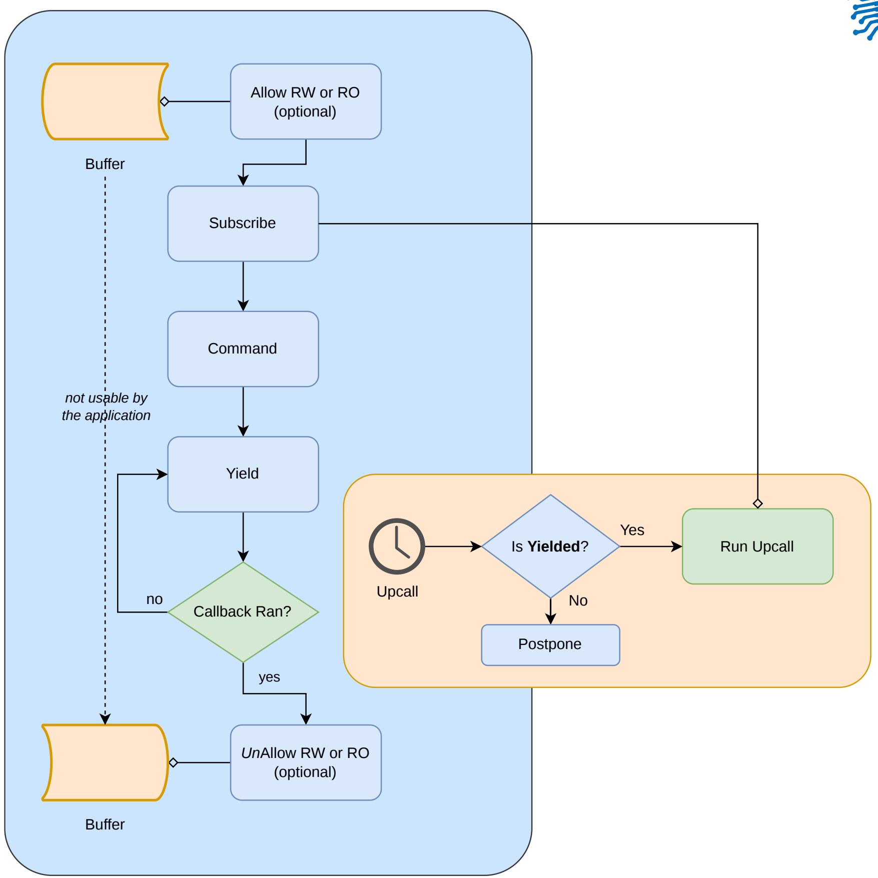
```
---| Fault Status |---
Data Access Violation:              true
Forced Hard Fault:                  true
Faulting Memory Address:            0x00000000
Fault Status Register (CFSR):       0x00000082
Hard Fault Status Register (HFSR):  0x40000000

---| App Status |---
App: crash_dummy   -   [Fault]
 Events Queued: 0   Syscall Count: 0   Dropped Callback Count
 Restart Count: 0
 Last Syscall: None

 ╔═══════════╤══════════════════════════════════════════╗
 ║  Address  │ Region Name    Used | Allocated (bytes)  ║
 ╚0x20006000═╪══════════════════════════════════════════╝
             │ ▼ Grant         948 |    948
  0x20005C4C ┼───────────────────────────────────────────
             │ Unused
  0x200049F0 ┼───────────────────────────────────────────
             │ ▲ Heap            0 |   4700               S
  0x200049F0 ┼─────────────────────────────────────────── R
             │ Data            496 |    496               A
  0x20004800 ┼─────────────────────────────────────────── M
             │ ▼ Stack          72 |   2048
  0x200047B8 ┼───────────────────────────────────────────
             │ Unused
  0x20004000 ┴───────────────────────────────────────────
```

System Calls

0. Yield
1. Subscribe
2. Command
3. ReadWriteAllow
4. ReadOnlyAllow
5. Memop
6. Exit
7. UserspaceReadableAllow



5: Memop

Memop expands the memory segment available to the process, allows the process to retrieve pointers to its allocated memory space, provides a mechanism for the process to tell the kernel where its stack and heap start, and other operations involving process memory.

```
memop(op_type: u32, argument: u32) -> [[ VARIES ]] as u32
```

Arguments

- `op_type` : An integer indicating whether this is a `brk` (0), a `sbrk` (1), or another memop call.
- `argument` : The argument to `brk` , `sbrk` , or other call.

Each memop operation is specific and details of each call can be found in the memop syscall documentation.

Return

- Dependent on the particular *memop* call.

6: Exit

The process signals the kernel that it has no more work to do and can be stopped or that it asks the kernel to restart it.

```
tock_exit(completion_code: u32)
tock_restart(completion_code: u32)
```

Return

None

2: Command

Command instructs the driver to perform a specific action.

```
command(driver: u32, command_number: u32, argument1: u32, argument2: u32) -> CommandReturn
```

Arguments

- `driver` : integer specifying which driver to use
- `command_number` : the requested command.
- `argument1` : a command-specific argument
- `argument2` : a command-specific argument

One Tock convention with the *Command* system call is that command number 0 will always return a value of 0 or greater if the driver is present.

Return

- three `u32` numbers
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `command_number` .
  - Other return codes based on the specific driver.

1: Subscribe

Subscribe assigns upcall functions to be executed in response to various events.

```
subscribe(driver: u32, subscribe_number: u32, upcall: u32, userdata: u32) -> Result<Upcall, (Upcall, ErrorCode)>
```

Arguments

- `driver` : integer specifying which driver to use
- `subscribe_number` : event number
- `upcall` : function's pointer to call upon event

```
void upcall(int arg1, int arg2, int arg3, void* userdata)
```

- `userdata` : value that will be passed back, usually a pointer

Return

- The previously registered upcall or `TOCK_NULL_UPCALL`
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `subscribe_number` .

0: Yield

Yield transitions the current process from the Running to the Yielded state.

```
// waits for the next upcall
// The process will not execute again until another upcall re-schedules the
// process.
yield()

// does not wait for the next upcall
// If a process has no enqueued upcalls, the
// process immediately re-enters the Running state.
yield_no_wait()
```

Return

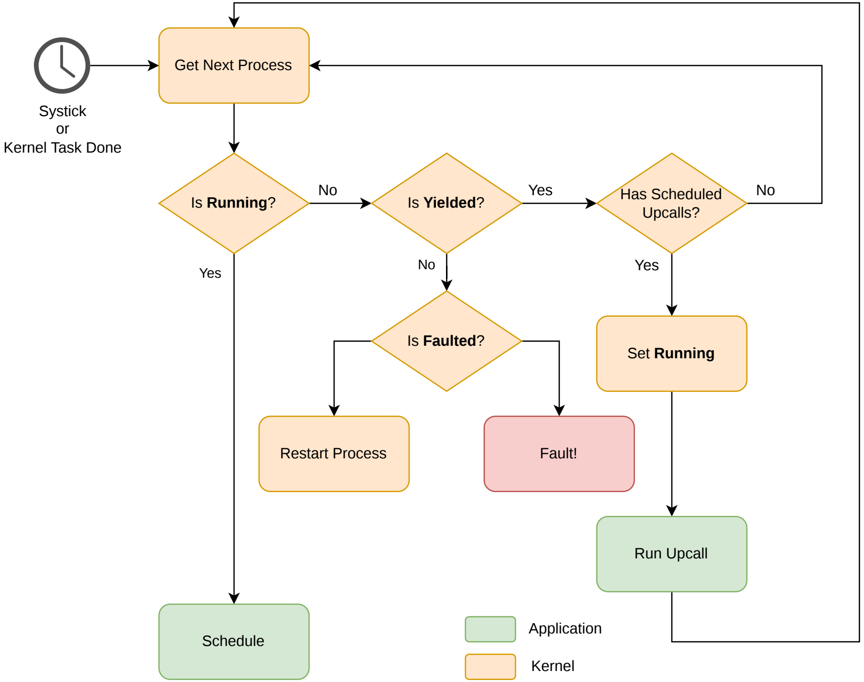
yield: None

yield_no_wait:

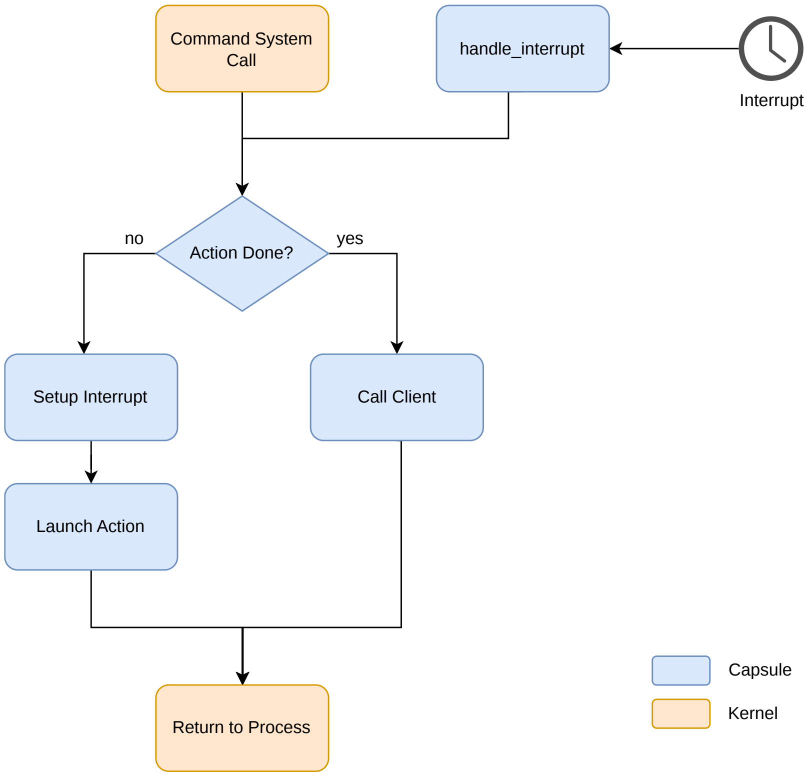
- 1 - *upcall* ran
- 0 - there was no queued *upcall* function to execute

Scheduler

using command, subscribe and yield



how the scheduler works



how drivers work

3 and 4: AllowRead(Write/Only)

Allow shares memory buffers between the kernel and application.

```
allow_readwrite(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSlice,
allow_readonly(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSlice,
```

Arguments

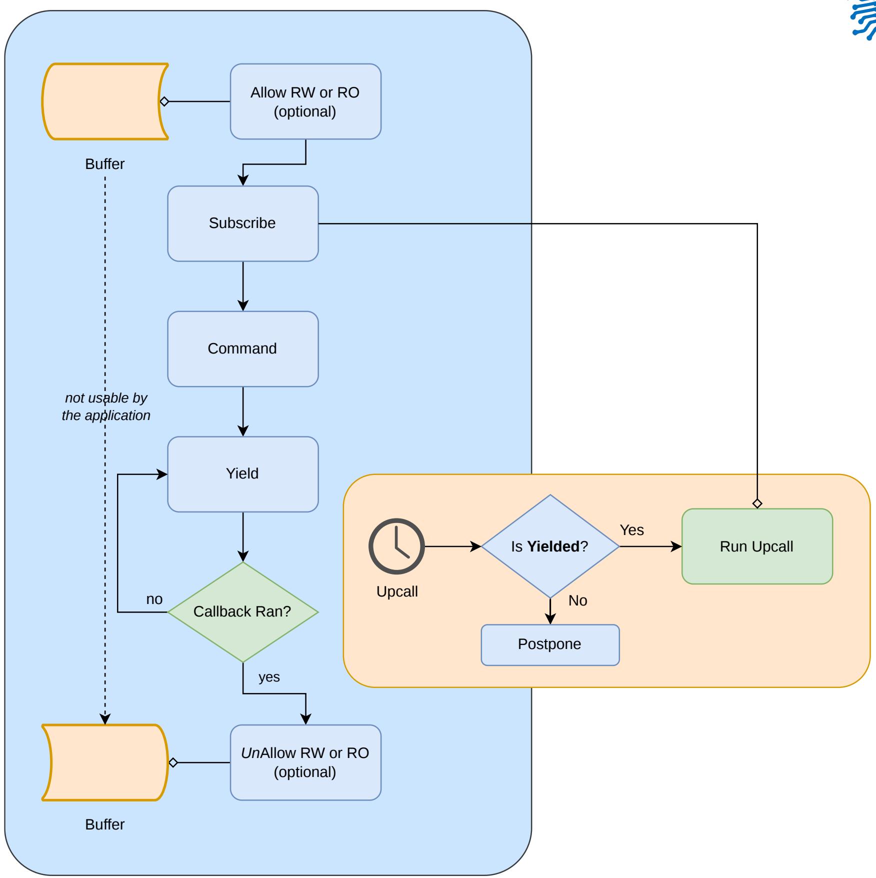
- `driver` : integer specifying which driver to use
- `allow_number` : driver-specific integer specifying the purpose of this buffer
- `pointer` : pointer to the buffer in the process memory space
  - null pointer revokes a previously shared buffer
- `size` : the length of the buffer

Return

- The previous allowed buffer or NULL
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `allow_number` .
  - `INVAL` the buffer referred to by `pointer` and `size` lies completely or partially outside of the processes addressable RAM.

System Call Pattern

1. *allow*: if data exchange is required, share a buffer with a driver
2. *subscribe* to the *action done* event
3. send a *command* to ask the driver to start performing an action
4. *yield* to wait for the *action done* event
   - *the kernel calls a callback*
   - verify if the expected event was triggered, if not *yield*
5. *unallow*: get the buffer back from the driver



Conclusion

we discussed about

- The purpose of an operating system
  - Abstractions
  - System calls
- Embedded Operating Systems
  - Real Time
- Tock OS