Tock Embedded OS Tutorial

SenSys 2017

Welcome to the Tock OS Training!

Please make sure you have completed all of the tutorial pre-requisites. If you prefer, you can download a virtual machine image with all the pre-requisites already installed.

https://github.com/helena-project/tock/tree/master/doc/courses/sensys/README.md

aka

http://bit.do/tock

Tock

A secure operating system for microcontrollers

- Kernel components in Rust
- Type-safe API for safe driver development
- Hardware isolated processes for application code

Use cases

- Security applications (e.g. authentication keys)
- Sensor networks
- Programmable wearables
- PC/phone peripherals
- Home/industrial automation
- Flight control

TockOS Stack

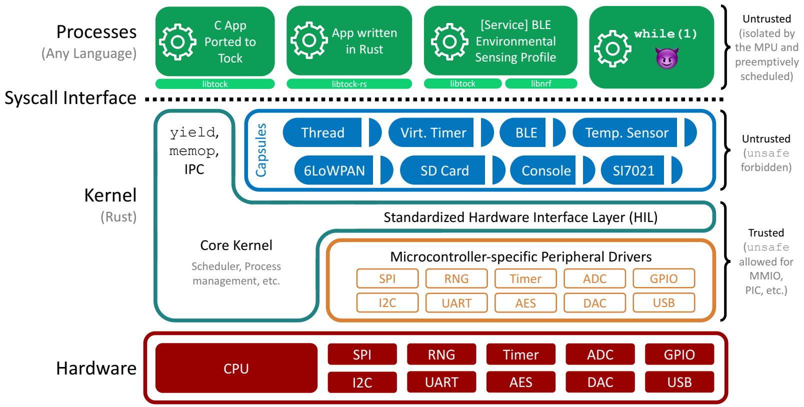

Figure 1

State of Tock

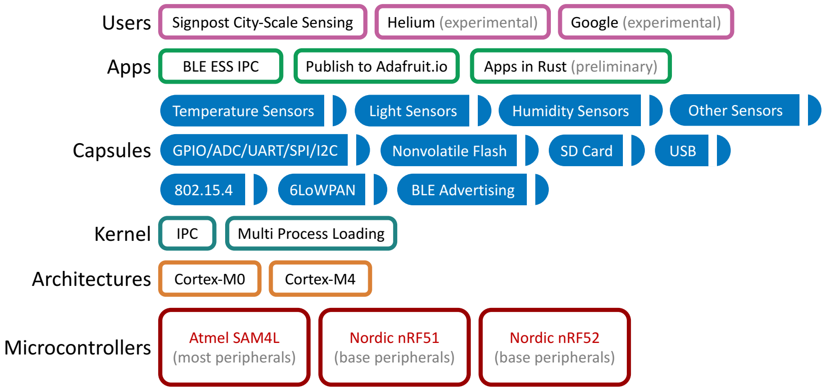

Figure 2

Tock 1.0 (Coming very soon)

- Stabilizes the initial syscall interface
  - Docs: https://github.com/helena-project/tock/tree/master/doc/syscalls
- Enable apps to be portable and independent of kernel
- Punts on stabilizing the internal kernel interfaces

Agenda Today

1. Intro to hardware, tools and development environment
2. Write an end-to-end Bluetooth Low Energy environment sensing application
3. Add functionality to the Tock kernel
    - Write some Rust!

Part 1: Hardware, tools, and development environment

Hail

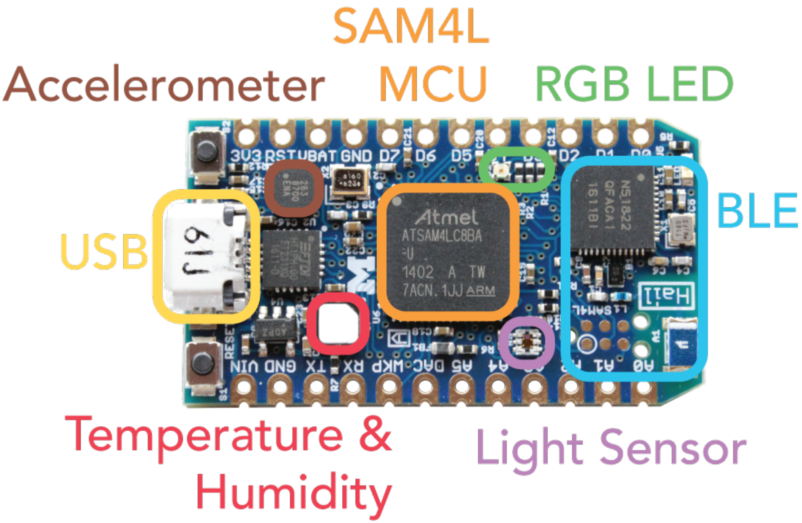

Figure 3

We need the Hails back at the end of the tutorial

But you can take one home with you! Purchase here:

https://tockos.org/hardware

Put in “SENSYS17” for $5 off, and “2600 Hearst Ave, Berkeley CA 94709” as the address for local pickup.

Binaries on-board in flash

- `0x00000`: **Bootloader**: Interact with Tockloader; load code
- `0x10000`: **Kernel**
- `0x30000`: **Processes**: Packed back-to-back

Tools

- `make`
- Rust/Cargo/Xargo (Rust code → LLVM)
- `arm-none-eabi` (LLVM → Cortex-M)
- `tockloader` to interact with Hail and the bootloader

Tools: `tockloader`

Write a binary to a particular address in flash

```
$ tockloader flash --address 0x10000 \
    target/thumbv7em-none-eabi/release/hail.bin
```

Program a process in Tock Binary Format[1]:

```
$ tockloader install myapp.tab
```

Restart the board and connect to the debug console:

```
$ tockloader listen
```

[1]TBFs are relocatable process binaries prefixed with headers like the package name. `.tab` is a tarball of TBFs for different architectures as well as a metadata file for `tockloader`.

Check your understanding

Turn to the person next to you:

1. What kinds of binaries exist on a Tock board? Hint: There are three, and only two can be programmed using `tockloader`.
2. What steps would you follow to program a process onto Hail? What about to replace the kernel?

Answers

1. The three binaries are the serial bootloader, the kernel, and a series of processes. The bootloader can be used to load the kernel and processes, but cannot replace itself.
2. To install a process, simply run `tockloader install` in the app directory. To load the kernel, use `tockloader flash` and specify the binary and the address: `tockloader flash --address 0x10000 hail.bin`.

Hands-on: Set-up development environment

3. Compile and flash the kernel
4. (Optional) Familiarize yourself with `tockloader` commands
   - `uninstall`
   - `list`
   - `erase-apps`
5. (Optional) Add some other apps from the repo, like `blink` and `sensors`

- Head to http://bit.do/tock2 to get started!
- (https://github.com/helena-project/tock/blob/master/doc/courses/sensys/environment.md)

Part 2: User space

System calls

Tock supports five syscalls that applications use to interact with the kernel.

Call	Target	Description
command	Capsule	Invoke an operation on a capsule
allow	Capsule	Share memory with a capsule
subscribe	Capsule	Register an upcall
memop	Core	Modify memory break
yield	Core	Block until next upcall is ready

C System Calls: `command` & `allow`

```
// Start an operation
int command(u32 driver, u32 command, int arg1, int arg2);

// Share memory with the kernel
int allow(u32 driver, u32 allow, void* ptr, size_t size);
```

C System Calls: `subscribe`

```
// Callback function type
typedef void (sub_cb)(int, int, int, void* userdata);

// Register a callback with the kernel
int subscribe(u32 driver,
              u32 subscribe,
              sub_cb cb,
              void* userdata);
```

C System Calls: `yield` & `yield_for`

```c
// Block until next callback
void yield(void);

// Block until a specific callback
void yield_for(bool *cond) {
  while (!*cond) {
    yield();
  }
}
```

Example: printing to the debug console

```
#define DRIVER_NUM_CONSOLE 0x0001

bool done = false;

static void putstr_cb(int x, int y, int z, void* ud) {
  done = true;
}

int putnstr(const char *str, size_t len) {
  allow(DRIVER_NUM_CONSOLE, 1, str, len);
  subscribe(DRIVER_NUM_CONSOLE, 1, putstr_cb, NULL);
  command(DRIVER_NUM_CONSOLE, 1, len, 0);
  yield_for(&done);

  return SUCCESS;
}
```

Inter Process Communication (IPC)

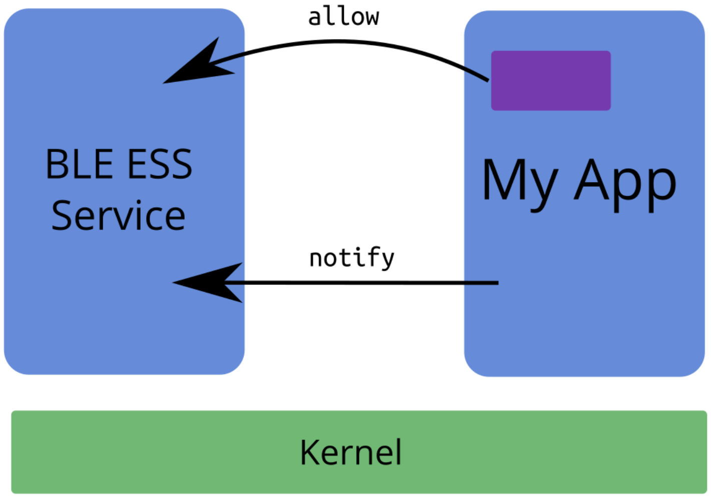

Figure 4

Tock Inter Process Communication Overview

Servers

- Register as an IPC service
- Call `notify` to trigger callback in connected client
- Receive a callback when a client calls `notify`

Clients

- Discover IPC services by application name
- Able to share a buffer with a connected service
- Call `notify` to trigger callback in connected service
- Receive a callback when service calls `notify`

Client Inter Process Communication API

```c
// Discover IPC service by name
int ipc_discover(const char* pkg_name);

// Share memory slice with IPC service
int ipc_share(int pid, void* base, int len);

// Register for callback on server `notify`
int ipc_register_client_cb(int pid, subscribe_cb cb,
                           void* userdata);

// Trigger callback in service
int ipc_notify_svc(int pid);
```

Check your understanding

Turn to the person next to you:

1. How does a process perform a blocking operation? Can you draw the flow of operations when a process calls `delay_ms(1000)`?
2. Which functions would a client call to interact with an IPC service that provides a UART console? What does the design of the console service look like?

Answers

1. A blocking operation starts with setting up a callback (using the `subscribe` syscall), then is initiated with a `command` syscall, and the process then blocks until the callback is called. For `delay_ms(1000)`, the application first registers a timer done callback, then calls the correct timer command with the value `1000`, then calls `yield()` which will return when the timer callback is triggered after 1000 ms.
2. First the client would call `ipc_discover()` to find the ID of the console service. Then, the client would call `ipc_share()` to share a buffer with the service, fill in the buffer with the string it wants to print to the console, and call `ipc_notify_svc()` to invoke the service to actually print the string. If the client wants to know when the string has been printed, it should `ipc_register_client_cb()` before notifying the service to get a callback.

   The console service is relatively simple. It first has to register a callback to receive notifications from clients. When the callback triggers, it uses the buffer shared by the client and prints the contents to the console.

Hands-on: Write a BLE environment sensing application

3. Get an application running on Hail
4. Print “Hello World” every second
5. Extend your app to sample on-board sensors
6. Extend your app to report through the `ble-env-sense` service

- Head to http://bit.do/tock3 to get started!
- (https://github.com/helena-project/tock/blob/master/doc/courses/sensys/application.md)

Part 3: The kernel

Trusted Computing Base (`unsafe` allowed)

- Hardware Abstraction Layer
- Board configuration
- Event & Process scheduler
- Rust `core` library
- Core Tock primitives

```
kernel/
chips/
```

Capsules (`unsafe` not allowed)

- Virtualization
- Peripheral drivers
- Communication protocols (IP, USB, etc)
- Application logic

`capsules/`

Constraints

Small isolation units

Breaking a monolithic component into smaller ones should have low/no cost

Avoid memory exhaustion in the kernel

No heap. Everything is allocated statically.

Low communication overhead

Communicating between components as cheap as an internal function call. Ideally inlined.

Event-driven execution model

```
pub fn main<P, C>(platform: &P, chip: &mut C,
                  processes: &mut [Process]) {
    loop {
        chip.service_pending_interrupts();
        for (i, p) in processes.iter_mut().enumerate() {
            sched::do_process(platform, chip, process);
        }

        if !chip.has_pending_interrupts() {
            chip.prepare_for_sleep();
            support::wfi();
        }
    }
}
```

Event-driven execution model

```rust
fn service_pending_interrupts(&mut self) {
    while let Some(interrupt) = get_interrupt() {
        match interrupt {
            ASTALARM => ast::AST.handle_interrupt(),
            USART0 => usart::USART0.handle_interrupt(),
            USART1 => usart::USART1.handle_interrupt(),
            USART2 => usart::USART2.handle_interrupt(),
            ...
        }
    }
}
```

Event-driven execution model

```
impl Ast {
    pub fn handle_interrupt(&self) {
        self.clear_alarm();
        self.callback.get().map(|cb| { cb.fired(); });
    }
}
impl time::Client for MuxAlarm {
    fn fired(&self) {
        for cur in self.virtual_alarms.iter() {
            if cur.should_fire() {
                cur.armed.set(false);
                self.enabled.set(self.enabled.get() - 1);
                cur.fired();
            }
        }
    }
}
```

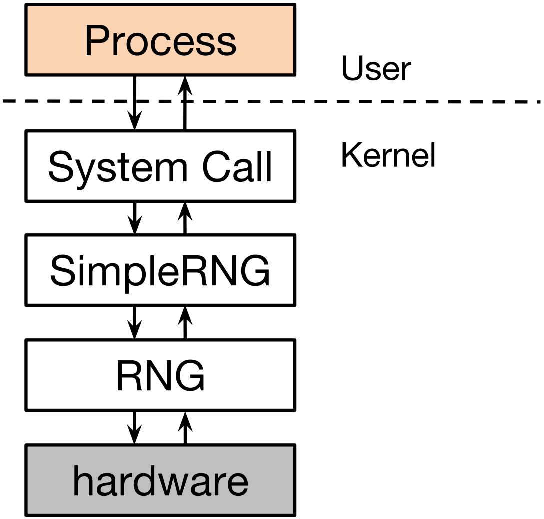

Figure 5: Capsules reference each other directly, assisting inlining

Check your understanding

Turn to the person next to you:

1. What are Tock kernel components called?
2. Is the kernel scheduled cooperatively or preemptively? What happens if a capsule performs a very long computation?
3. How is a hardware interrupt handled in the kernel?

Answers

1. Tock kernel components are called “capsules”
2. The kernel is scheduled cooperatively by capsules calling methods on each other. If a capsule performs a very long computation it might prevent other capsules from running or cause them to miss events.
3. Hardware interrupts are scheduled to run when capsules next yield. If a process is running when a hardware event happens, the hardware event will be immediately handled.

Hands-on: Write and add a capsule to the kernel

4. Read the Hail boot sequence in `boards/hail/src/main.rs`
5. Write a new capsule that prints “Hello World” to the debug console.
6. Extend your capsule to print “Hello World” every second
7. Extend your capsule to print light readings every second
8. Extra credit

- Head to http://bit.do/tock4 to get started!
- (https://github.com/helena-project/tock/blob/master/doc/courses/sensys/capsule.md)

We need the Hails back!

But you can take one home with you! Purchase here:

https://tockos.org/hardware

Put in “SENSYS17” for $5 off, and “2600 Hearst Ave, Berkeley CA 94709” as the address for local pickup.

imix & Hail Comparison

	imix	Hail
Microcontroller	Sam4l	Sam4l
Sensors		
‣ Accelerometer	□	□
‣ Temperature/Humidity	□	□
‣ Light	□	□
‣ Accelerometer	□	□
Radios		
‣ BLE	□	□
‣ 802.15.4	□	
Other Features		
‣ Buttons	1 user, 1 reset	1 user, 1 reset
‣ LEDs	3	1 blue, 1 RGB
‣ Hardware RNG	□	
‣ USB Host	□	pins only
‣ Independent Power Domains	□	
Programming	USB or JTAG	USB or JTAG
Form Factor	Custom, Arduino Headers	Particle Photon
Size	2.45” x 4“	0.8” x 1.44“
Price	$100	$60

Stay in touch!

https://www.tockos.org

https://github.com/helena-project/tock

tock-dev@googlegroups.com

#tock on Freenode

Quick Survey!

- https://goo.gl/???