

Tock Embedded OS Tutorial

Octopi Kickoff 2018

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/alevy/tock-chalmers/tree/master/doc/courses/chalmers/README.md>
aka

<https://goo.gl/Fztjgv>

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

Part 1: Hardware, tools and development environment

TI LaunchXL CC26x2



- ▶ Bluetooth/802.15.14 System on a Chip
- ▶ ARM Cortex-M4 (with a Cortex-M0 for the radio, but we can't access it)
- ▶ 80KB RAM, 352KB Flash

Binaries on-board

Bootloader

Kernel

Processes

Tools

- ▶ `make` (just instrumenting cargo and the bootloader tool)
- ▶ Rust (nightly for `asm!`, compiling core, etc)
- ▶ `tockloader` to interact with the board

Tools

Program the kernel

```
$ make program
```

Program a process in Tock Binary Format¹:

```
$ make APP=../app/app.tbf program-app
```

Connect to the debug console:

```
$ tockloader listen
```

¹TBFs are relocatable process binaries prefixed with headers like the package name.

Check your understanding

1. What kinds of binaries exist on a Tock board? Hint: There are three, and only two can be programmed using `tockloader`.
2. Can you point to the chip on the LaunchXL that runs the Tock kernel? How about the processes?
3. What steps would you follow to program a processes onto LaunchXL? What about to replace the kernel?

Hands-on: Set-up development environment

1. Compile and flash the kernel

- ▶ Head to `<https://goo.gl/4VBvnd>` to get started!
- ▶ (<https://github.com/alevy/tock-chalmers/blob/master/doc/courses/chalmers/environment.md>)

Part 2: User space

System calls

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

```
int command(uint32_t driver, uint32_t command,  
            int arg1, int arg2);  
  
int allow(uint32_t driver, uint32_t allow, void* ptr,  
          size_t size);
```

C System Calls: subscribe

```
typedef void (subscribe_cb)(int, int, int,  
                             void* userdata);  
  
int subscribe(uint32_t driver, uint32_t subscribe,  
              subscribe_cb cb, void* userdata);
```

C System Calls: yield & yield_for

```
void yield(void);
```

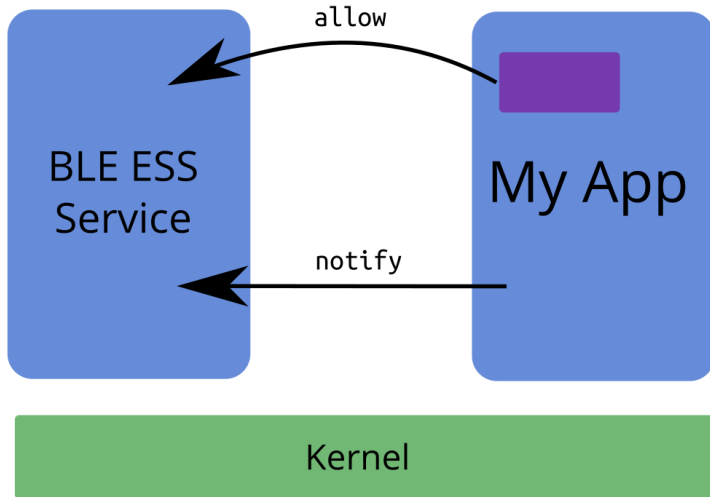
```
void yield_for(bool *cond) {  
    while (!*cond) {  
        yield();  
    }  
}
```

Example: printing to the debug console

```
static void putstr_cb(int _x, int _y, int _z, void* ud) {  
    putstr_data_t* data = (putstr_data_t*)ud;  
    data->done = true;  
}
```

```
int putnstr(const char *str, size_t len) {  
    putstr_data_t data;  
    data.buf = str;  
    data.done = false;  
  
    allow(DRIVER_NUM_CONSOLE, 1, str, len);  
    subscribe(DRIVER_NUM_CONSOLE, 1, putstr_cb, &data);  
    command(DRIVER_NUM_CONSOLE, 1, len, 0);  
    yield_for(&data.done);  
    return ret;  
}
```


Inter Process Communication (IPC)



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`

Inter Process Communication API

```
// discover IPC service by name  
// returns error code or PID for service  
int ipc_discover(const char* pkg_name);  
  
// shares memory slice at address 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);  
  
// trigger callback in a client  
int ipc_notify_client(int pid);
```

Check your understanding

1. How does a process perform a blocking operation? Can you draw the flow of operations when a process calls `delay_ms(1000)`?
2. How would you write an IPC service to print to the console? Which functions would the client need to call?

Hands-on: Write a simple applicaiton

1. Get an application running on LaunchXL
 2. Print “Hello World” every second
 3. Toggle an LED every second every second
- ▶ Head to <<https://goo.gl/hoxAP6>> to get started!
 - ▶ (github.com/alevy/tock-chalmers/blob/master/doc/courses/chalmers/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

```
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();
            }
        }
    }
}
```

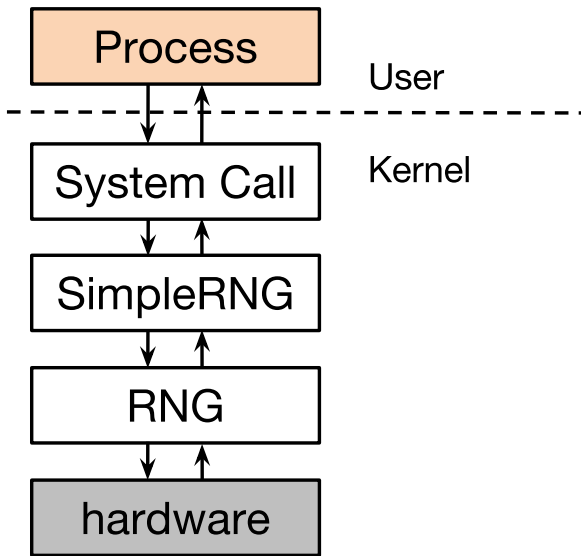


Figure 1: Capsules reference each other directly, assisting inlining

The mutable aliases problem

```
enum NumOrPointer {  
    Num(u32),  
    Pointer(&mut u32)  
}  
  
// n.b. will not compile  
let external : &mut NumOrPointer;  
match external {  
    Pointer(internal) => {  
        // This would violate safety and  
        // write to memory at 0xdeadbeef  
        *external = Num(0xdeadbeef);  
        *internal = 12345; // Kaboom  
    },  
    ...  
}
```

Interior mutability to the rescue

Type	Copy-only	Mutual exclusion	Opt.	Mem Opt.
Cell	✓	✗	✓	✓
VolatileCell	✓	✗	✗	✓
TakeCell	✗	✓	✗	✓
MapCell	✗	✓	✓	✗

```

pub struct Fxos8700cq<`a> {
    i2c: &`a I2CDevice,
    state: Cell<State>,
    buffer: TakeCell<`static, [u8]>,
    callback:
        Cell<Option<&`a hil::ninedof::NineDofClient>>,
}

impl<`a> I2CClient for Fxos8700cq<`a> {
    fn cmd_complete(&self, buf: &`static mut [u8]) { ... }
}

impl<`a> hil::ninedof::NineDof for Fxos8700cq<`a> {
    fn read_accelerometer(&self) -> ReturnCode { ... }
}

pub trait NineDofClient {
    fn callback(&self, x: usize, y: usize, z: usize);
}

```


Check your understanding

1. What is a `VolatileCell`? Can you find some uses of `VolatileCell`, and do you understand why they are needed?
Hint: look inside `chips/sam4l/src`.
2. What is a `TakeCell`? When is a `TakeCell` preferable to a standard `Cell`?

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

1. Read the Hail boot sequence in `boards/hail/src/main.rs`
2. Write a new capsule that prints “Hello World” to the debug console.
3. Extend your capsule to print “Hello World” every second
 - ▶ Head to <https://goo.gl/FXw6xg> to get started!
 - ▶ (github.com/alevy/tock-chalmers/blob/master/doc/courses/chalmers/capsule.md)

<https://www.tockos.org>

<https://github.com/tock/tock>

tock-dev@googlegroups.com

#tock on Freenode