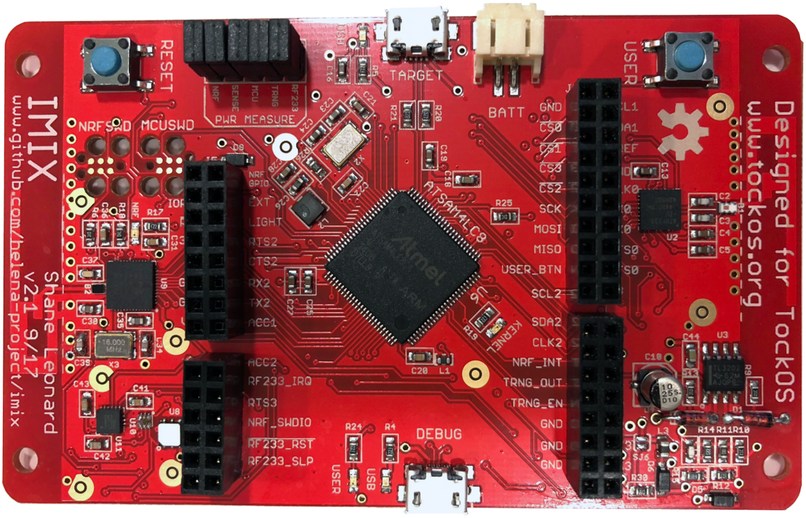
Tock Embedded OS Tutorial

SenSys 2018

Agenda Today

09:30-10:40	Intro to Tock, Development Environment & Hardware
10:40-11:00	Coffee break
11:00-12:00	Hardware setup and installing apps
12:00-13:30	Lunch
13:30-15:20	Find and fix a real world bug
15:20-15:40	Coffee break
15:40-17:30	Choose your own adventure

Part 1: Hardware, tools, and development environment


RESET
PWR MEASURE
NRF
SENSE
MCU
TRNG
RF233
TARGET
BATT
USER
IMIX
www.github.com/helena-project/imix
Shane Leonard
v2.1 9/17
NRFSWD
MCUSWD
Designed for TockOS
www.tockos.org
ATSAM4LC8
Atmel
KERNEL
16.000 MHz
EXT
LIGHT
RTS2
CTS2
RX2
TX2
ACC1
ACC2
RF233_IRQ
RTS3
NRF_SWDIO
RF233_RST
RF233_SLP
GND
CS0
CS1
CS3
CS2
SCK
MOSI
MISO
USER_BTN
SCL2
SDA2
CLK2
NRF_INT
TRNG_OUT
TRNG_EN
GND
GND
GND
DEBUG
USER
USB
USB

imix

- Atmel SAM4L, Cortex-M4, 64 kB RAM, 256 kB flash
- Nordic NRF51 Bluetooth SoC
- 802.15.4 radio (6lowpan)
- Temperature, humidity, and light sensors
- 2 USBs (target USB + FTDI serial USB)
- 2 LEDs, 1 “user” button

Binaries on-board in flash

- `0x00000`: **Bootloader**: Interact with Tockloader; load code
- `0x10000`: **Kernel**
- `0x40000`: **Processes**: Packed back-to-back

Tools

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

Tools: `tockloader`

Write a binary to a particular address in flash

```
$ tockloader flash --address 0x10000 \
    target/thumbv7em-none-eabi/release/imix.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 imix? 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. Use `tockloader`:
   - `tockloader install app.tab`
   - `tockloader flash --address 0x10000 imix-kernel.bin`

Hands-on: Set-up development environment

3. Compile and program 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://j2x.us/tock1 to get started!
- (https://github.com/tock/tock/blob/tutorial-sensys-2018/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`

```c
// 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`

```c
// 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;
}
```

Hands-on: Write a simple application

3. Get an application running on imix
4. Print “Hello World” every second
5. Extend your app to sample on-board sensors

- Head to http://j2x.us/tock2 to get started!
- (https://github.com/tock/tock/blob/tutorial-sensys-2018/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

```
arch/
chips/
kernel/
```

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

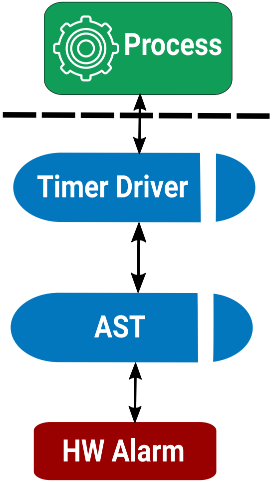
```rust
pub fn main<P, C>(platform: &P, chip: &mut C,
                  processes: &mut [Process]) {
    loop {
        chip.service_pending_interrupts();
        for (i, p) in processes.iter_mut().enumerate() {
            sched::do_process(platform, chip, process);
        }

        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();
            }
        }
    }
}
```


Process
Timer Driver
AST
HW Alarm

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 imix boot sequence in `boards/imix/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://j2x.us/tock3 to get started!
- (https://github.com/tock/tock/blob/tutorial-sensys-2018/doc/courses/sensys/capsule.md)

Stay in touch!

https://www.tockos.org

https://github.com/tock/tock

tock-dev@googlegroups.com