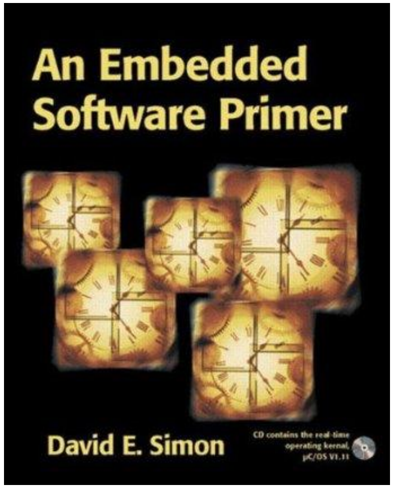
CESE4030
Embedded Systems Laboratory

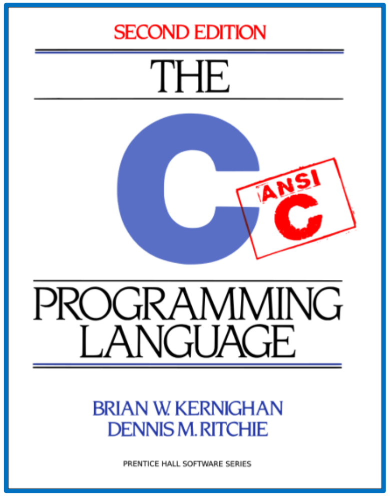
Embedded Programming

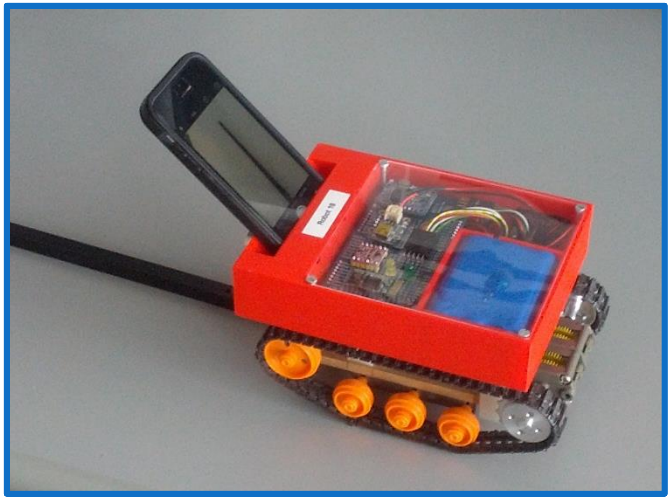
Embedded Software

CSE2425

- 2nd year BSc course
- Fast forward (10:1)





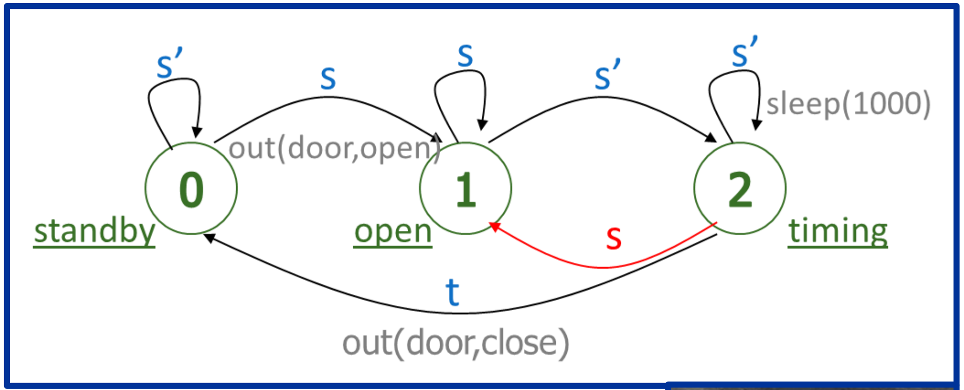


Embedded Programming

- More difficult than “classical” programming
  - Interaction with hardware
  - Real-time issues (timing)
  - Concurrency (multiple threads, scheduling, deadlock)
  - Event-driven programming (interrupts)

- FSMs to the rescue
  - modelling tool
  - programming paradigm ⬅

Programming State Machines



- Finite State Machines
  - prime design pattern in embedded systems
- Transitions initiated by events
  - interrupts (timers, user input, …)
  - polling
- Actions
  - output
  - modifying system state (e.g., writing to global variables)

Running example

- See Wikipedia: Automata-based programming[1]
- Consider a program in C that reads a text from the standard input stream, line by line, and prints the first word of each line. Words are delimited by spaces.

[1] https://en.wikipedia.org/wiki/Automata-based_programming

Exercise (5 min)

Code

- Consider a program in C that reads a text from the standard input stream, line by line, and prints the first word of each line. Words are delimited by spaces.

Ad-hoc solution

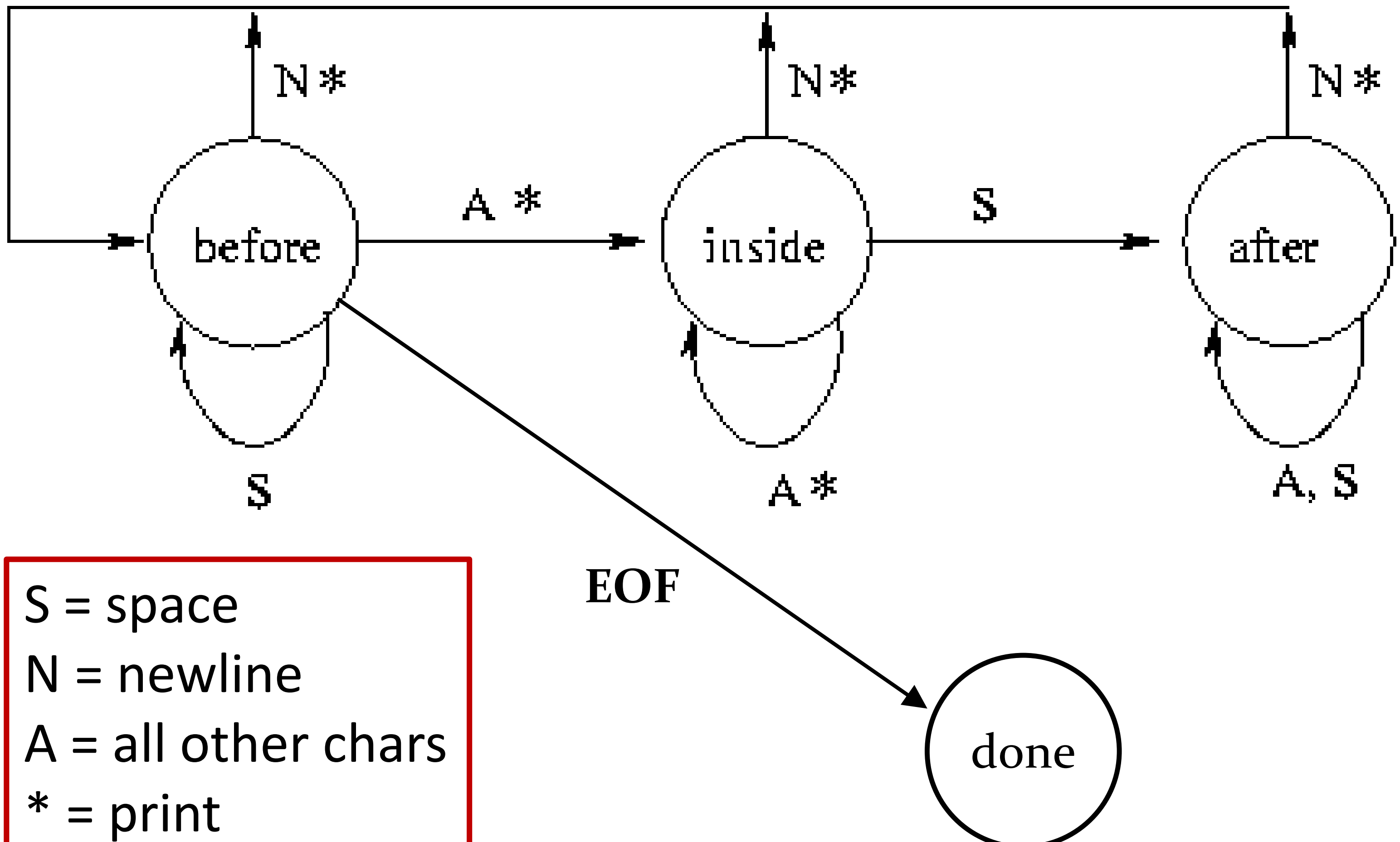
```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    do {
        do
            c = getchar();
        while(c == ' ');
        while(!isspace(c) && c != '\n` && c != EOF) {
            putchar(c);
            c = getchar();
        }
        putchar('\n');
        while(c != '\n` && c != EOF)
            c = getchar();
    } while(c != EOF);
    return 0;
}
```

skip leading spaces

print word

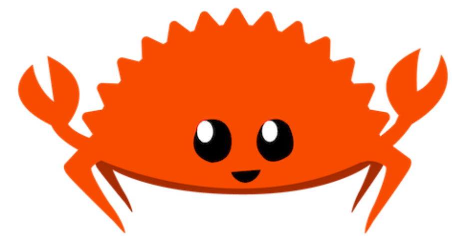
skip trailing chars

FSM



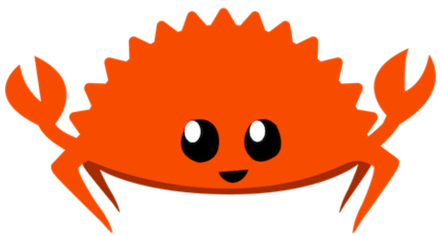
S = space
N = newline
A = all other chars
* = print

FSM-based solution



```
use crate::get_char;

enum State {Before, Inside, After}

pub fn main() -> io::Result<()> {
    let mut inp = File::open("input.txt")?;

    let mut state = State::Before;

    while let Some(c) = get_char(&mut inp)? {
        match state {
            State::Before => {
                if c != ' ' {
                    print!("{c}");
                    if c != '\n' {
                        state = State::Inside;
                    }
                }
            }
            State::Inside => {
```

FSM-based solution



```
                State::Inside => {
                    if c != ' ' {
                        print!("{c}");
                    } else if c == '\n' {
                        println!();
                        state = State::Before;
                    } else
                        state = State::After;
                }
                State::After => {
                    if c == '\n' {
                        println!();
                        state = State::Before;
                    }
                }
            }
        }
    Ok(())
}
```

does not scale to large FSMs

Refactored solution

```
pub trait State {
    // we say here that to be called a state, a type
    // must have a `step` function that takes a character, and
    // returns a new state.
    fn step(&self, c: char) -> &dyn State;
}
    ⋮
pub fn main() -> io::Result<()> {
    let mut inp = File::open("input.txt")?;
    let mut state: &dyn State = &Before;
    while let Some(c) = get_char(&mut inp)? {
        state = state.step(c);
    }
    Ok(())
}
```

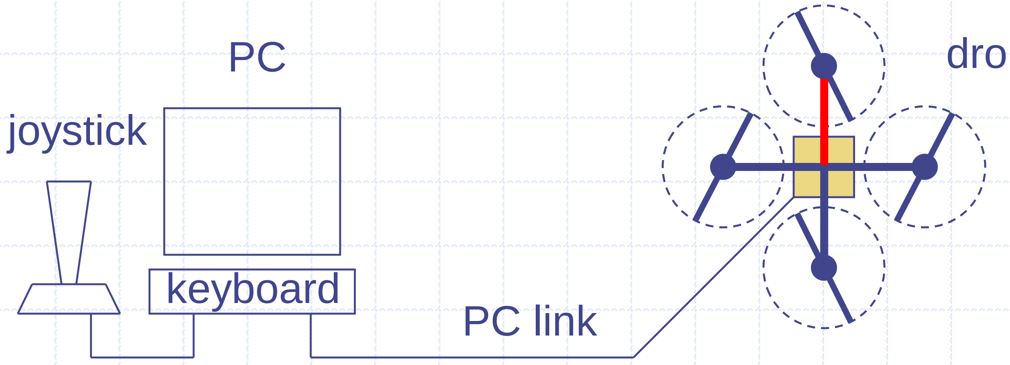
Refactored solution

```
// we define a type "Before" which has this property that
// it is a state, and we implement its `step` function.
pub struct Before;
impl State for Before {
    fn step(&self, c: char) -> &dyn State {
        if c != ' ' {
            print!("{c}");
            if c != '\n' {
                return &Inside;
            }
        }
        self
    }
}

pub struct Inside;
impl State for Inside {
    fn step(&self, c: char) -> &dyn State {
```

What’s in the assignment?

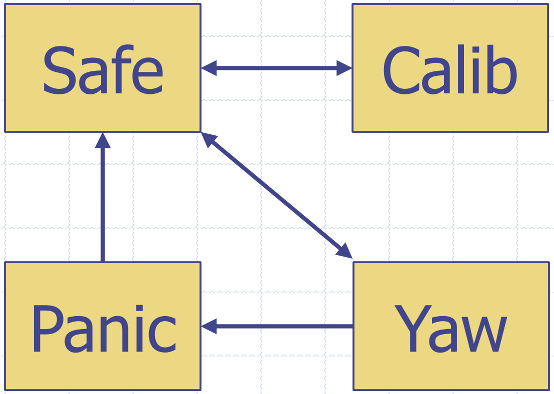
BACK TO QUADCOPTERS

Quadrupel: FSM

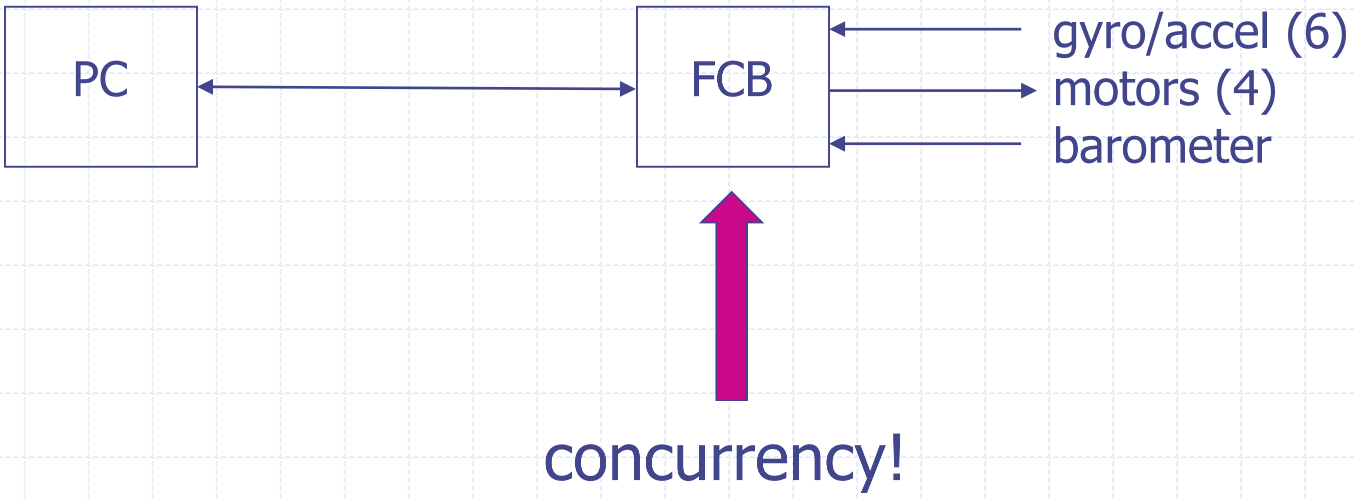


From the assignment

- Safe
- Panic
- Calibrate
- Yaw
- …



Quadrupel: FSM + control loop



Software Architecture Survey

- Round-Robin (polling)
- Round-Robin (with interrupts)
- [Function-Queue Scheduling]
- Real-Time OS

- Motivates added value of RTOS
- At the same time demonstrates you don't always need to throw a full-fledged RTOS at your problem!

Round-Robin

```
void   main(void)
{
        while (TRUE) {
                !! poll device A
                !! service if needed

                ..

                !! poll device Z
                !! service if needed
        }
}
```

- polling: response time slow and stochastic
- fragile architecture

Round-Robin with Interrupts

```
void   isr_deviceA(void)
{
        !! service immediate needs + assert flag A
}
..

void   main(void)
{
        while (TRUE) {
                !! poll device flag A
                !! service A if set and reset flag A
                ..
}
```

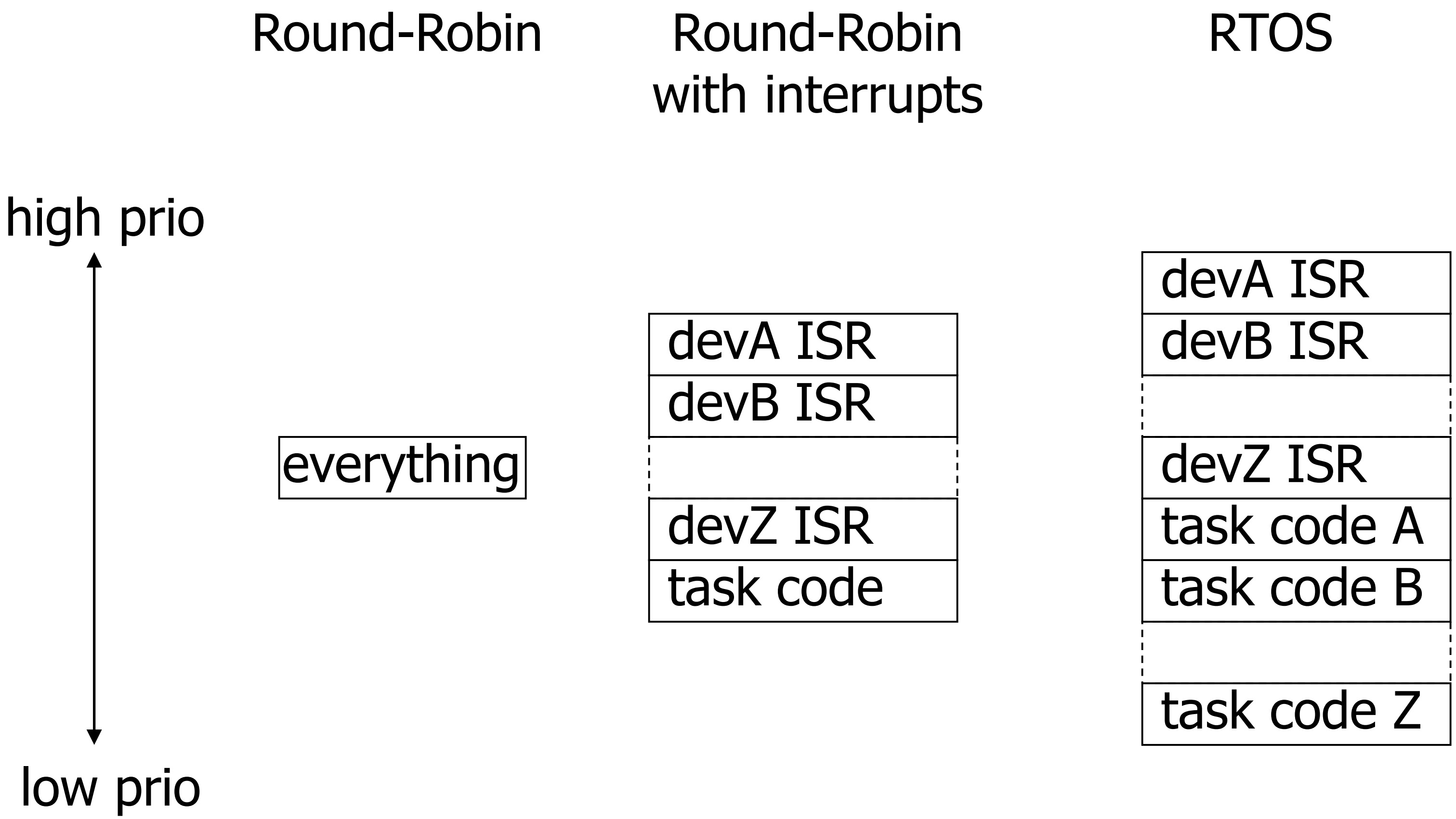
- ISR (interrupt vs. polling!): much better response time
- main still slow (i.e., lower priority than ISRs)

Real-Time OS

```
void   isr_deviceA(void)
{
        !! service immediate needs + set signal A
}
..

void   taskA(void)
{
        !! wait for signal A
        !! service A
}
..
```

- includes task preemption by offering thread scheduling
- stable response times, even under code modifications

Architecture Overview



What’s in the template?

BACK TO QUADCOPTERS

Gitlab & friends

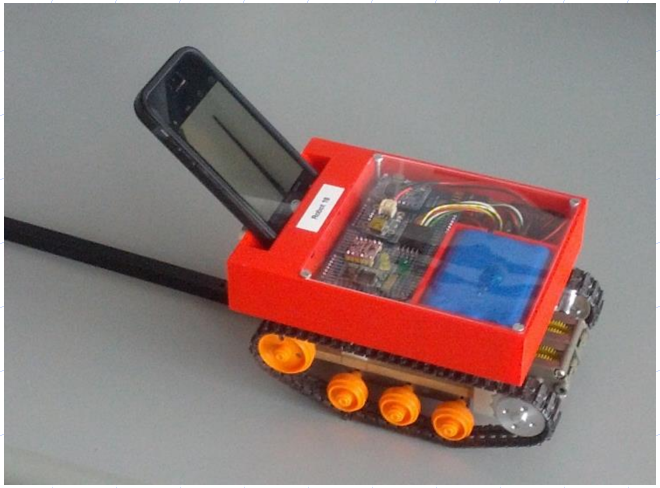
Computer and Embedded Systems Engineering / Embedded Systems Lab

- template-project
- documentation

System Architecture (today!)

- Functional decomposition

- Who does what?
- Interfaces



Communication protocol (lab 1)

- PC -> Drone (send)
  - periodic: pilot control
  - ad hoc: mode changing, param tuning
- Drone -> PC (receive)
  - periodic: telemetry (for visualization)
  - ad hoc: logging (for post-mortem analysis)
- Dependable, robust to data loss
  - header synch

Design your protocol (today!)

- Packet layout
  - start/stop byte(s)
  - header, footer?
  - fixed/variable length
- Message types
  - values (sizes)
  - frequency

BW + processing constraints?!

Before you go

Safety first:

- goggles
- common sense