

# CS4140

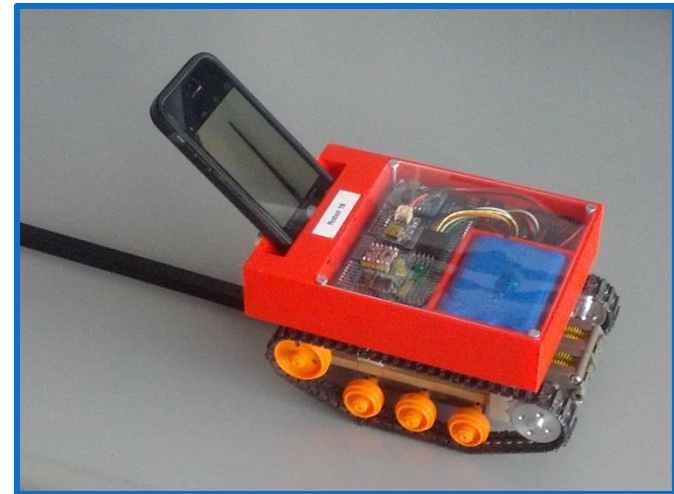
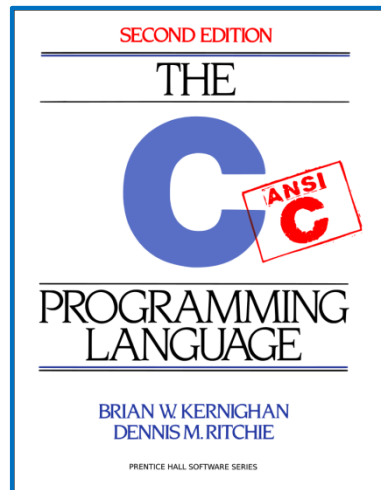
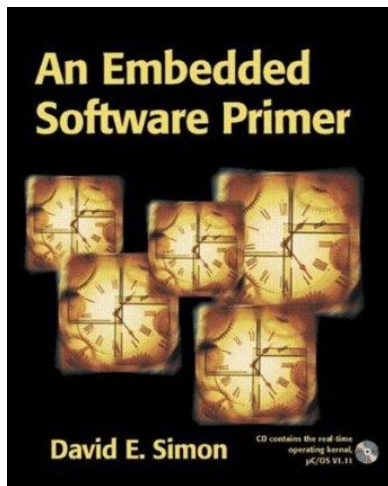
## Embedded Systems Laboratory

### Embedded Programming

# Embedded Software

TI2726-B

- 2<sup>nd</sup> 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)
  - Need to understand underlying RTOS principles
  - Event-driven programming (interrupts)
- Lots of (novice) errors (hence the crisis)



# Embedded Programming Example

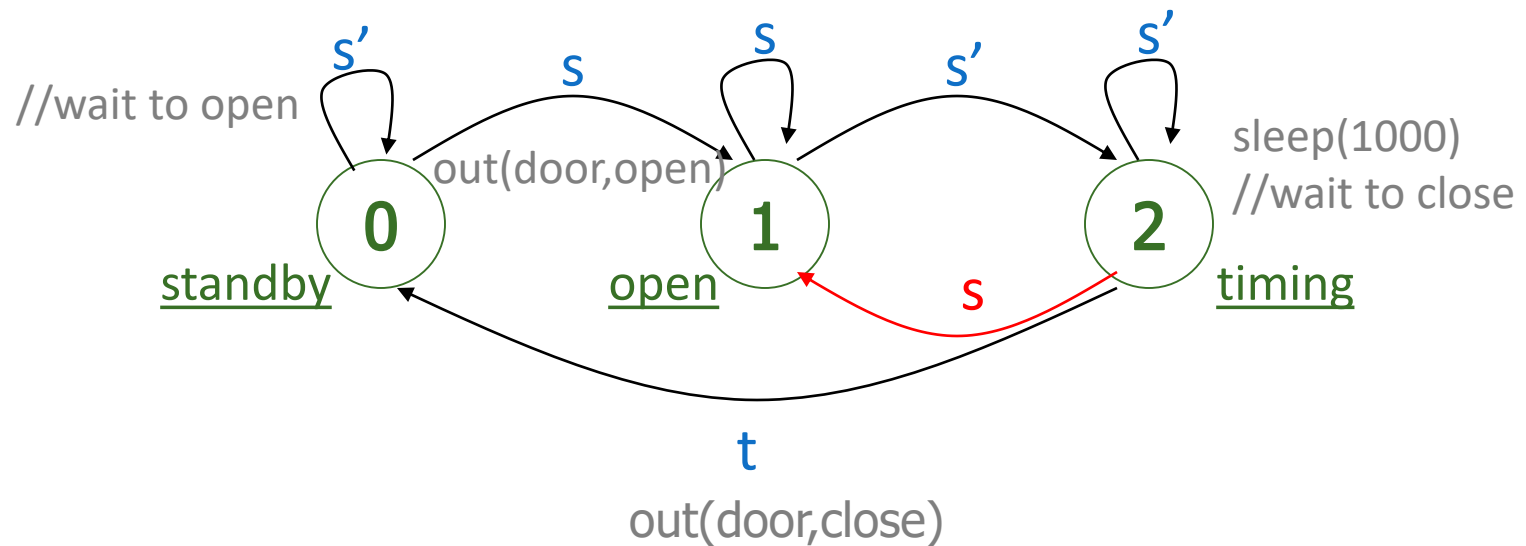
- Automatic sliding gate task (thread):

```
for (;;) {  
    // wait to open  
    while (inp(sensor) != 1) ;  
    out(door,OPEN) ;  
    // wait to close  
    while (inp(sensor) == 1) ;  
    sleep(1000) ;  
    // close after timeout  
    out(door,CLOSE) ;  
}
```

- Any issues with this code?



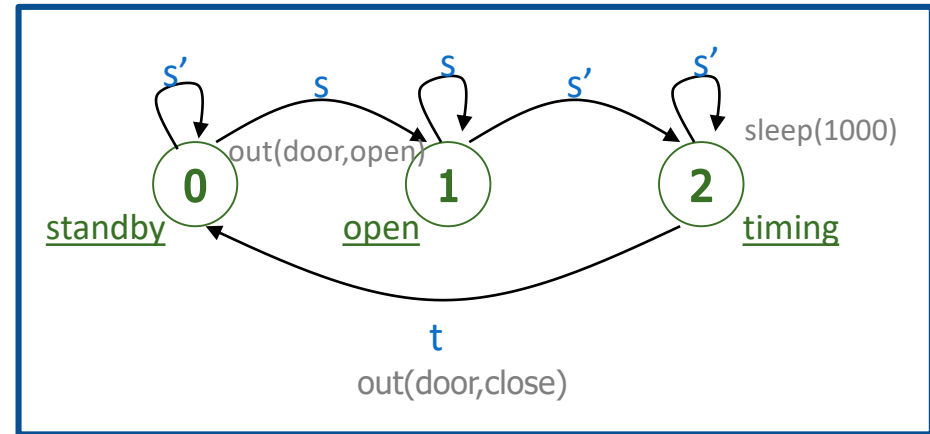
# Specification: Finite State Machine



- Red arc missing from the specification
- Door can slam in your face!

# 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**<sup>1</sup>
- 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.

<sup>1</sup>[https://en.wikipedia.org/wiki/Automata-based\\_programming](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

- too many loops
- duplicate EOF corner casing

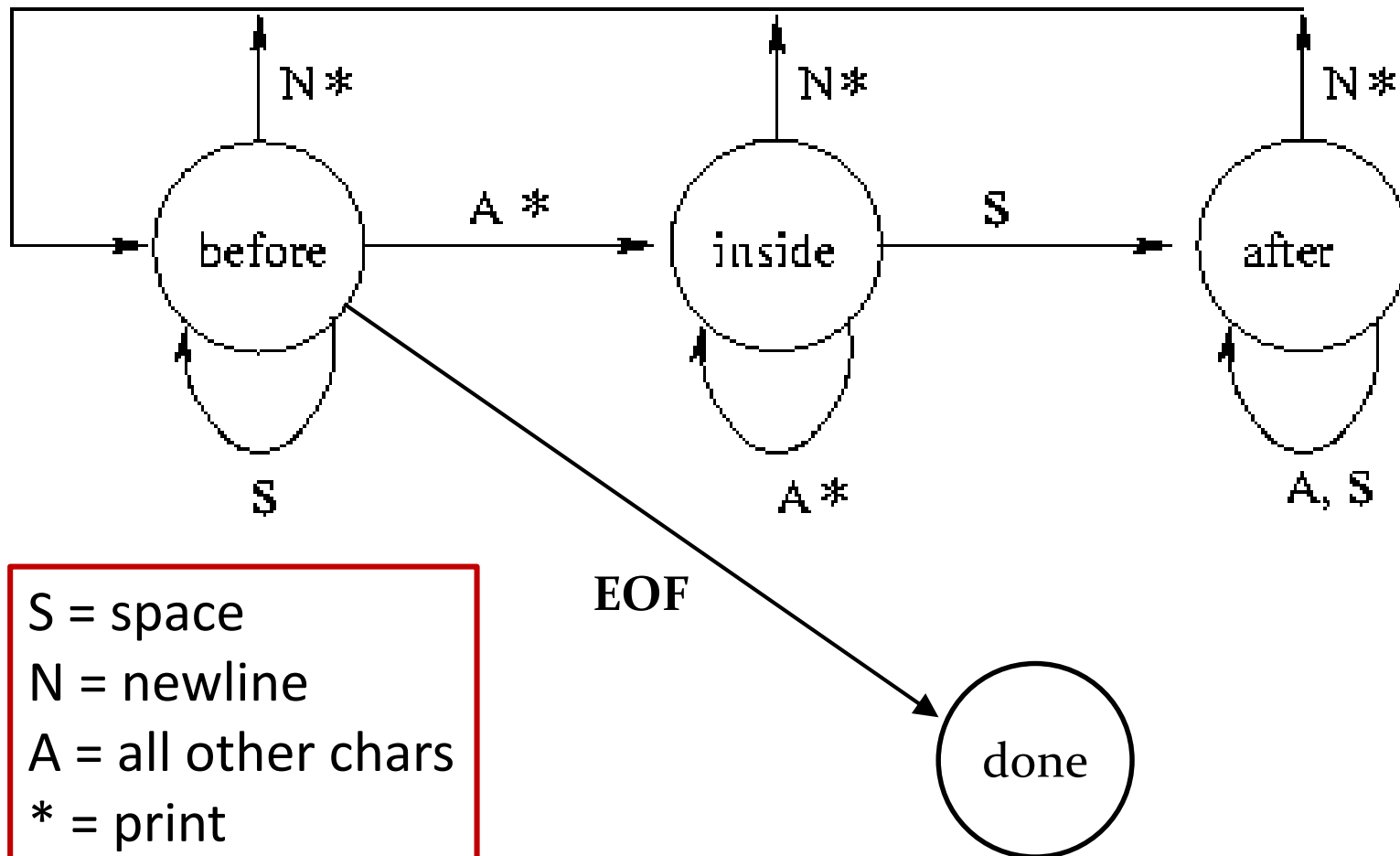
```
1. #include <stdio.h>
2. #include <ctype.h>
3. int main(void)
4. {
5.     int c;
6.     do {
7.         do
8.             c = getchar();
9.             while(c == ' ');
10.            while(!isspace(c) && c != '\n' && c != EOF) {
11.                putchar(c);
12.                c = getchar();
13.            }
14.            putchar('\n');
15.            while(c != '\n' && c != EOF)
16.                c = getchar();
17.        } while(c != EOF);
18.    return 0;
19. }
```

skip  
leading  
spaces

print  
word

skip  
trailing  
chars

# FSM



# FSM-based solution

```
1. int main(void)
2. {
3.     enum states {
4.         before, inside, after
5.     } state;
6.     int c;
7.     state = before;
8.     while((c = getchar()) != EOF) {
9.         switch(state) {
10.            case before:
11.                if(c != ' ') {
12.                    putchar(c);
13.                    if(c != '\n')
14.                        state = inside;
15.                }
16.                break;
17.            case inside:
```

- 1 loop
- 1 case for EOF checking

# FSM-based solution

```
17.     case inside:
18.         if(!isspace(c))
19.             putchar(c);
20.         else if(c == '\\n') {
21.             putchar('\\n');
22.             state = before;
23.         } else
24.             state = after;
25.         break;
26.     case after:
27.         if(c == '\\n') {
28.             putchar('\\n');
29.             state = before;
30.         }
31.         break;
32.     default:
33.         fprintf(stderr, "unknown state %p\\n", state);
34.         abort();
```

defensive programming!

# Refactored solution

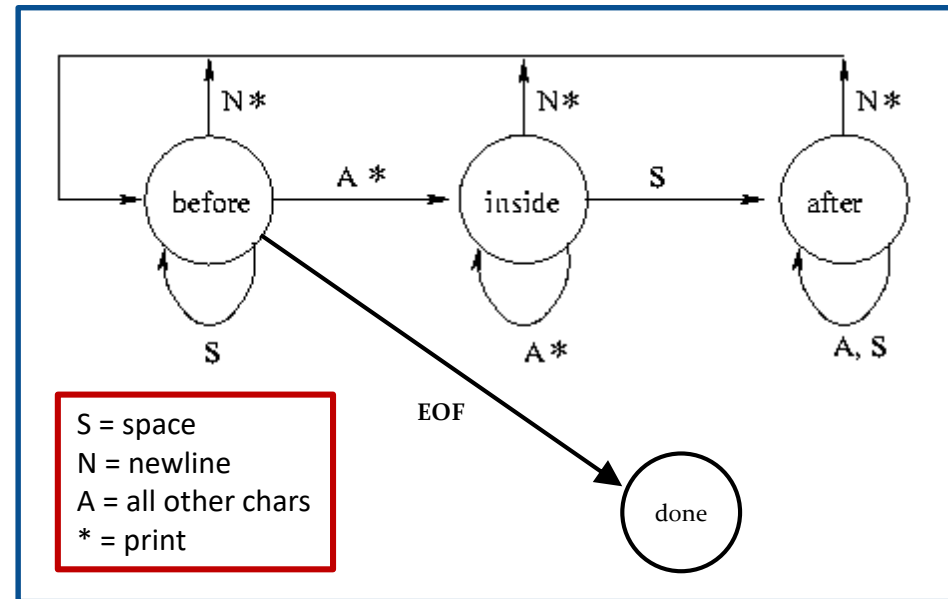
```
1. enum states { before, inside, after };
2. void step(enum states *state, int c)
3. {
4.     switch(*state) {
5.         case before: ... *state = inside; ...
6.         case inside: ... *state = after; ...
7.         case after: ... *state = before; ...
8.         default: fprintf(stderr, "unknown state %p\n", state);
9.     }
10.}
11.int main(void)
12.{
13.    int c;
14.    enum states state = before;
15.    while((c = getchar()) != EOF) {
16.        step(&state, c);
17.    }
18.    return 0;
```

- lifted loop

# FSM: table-based solution

- Transition:
  - action
  - next state

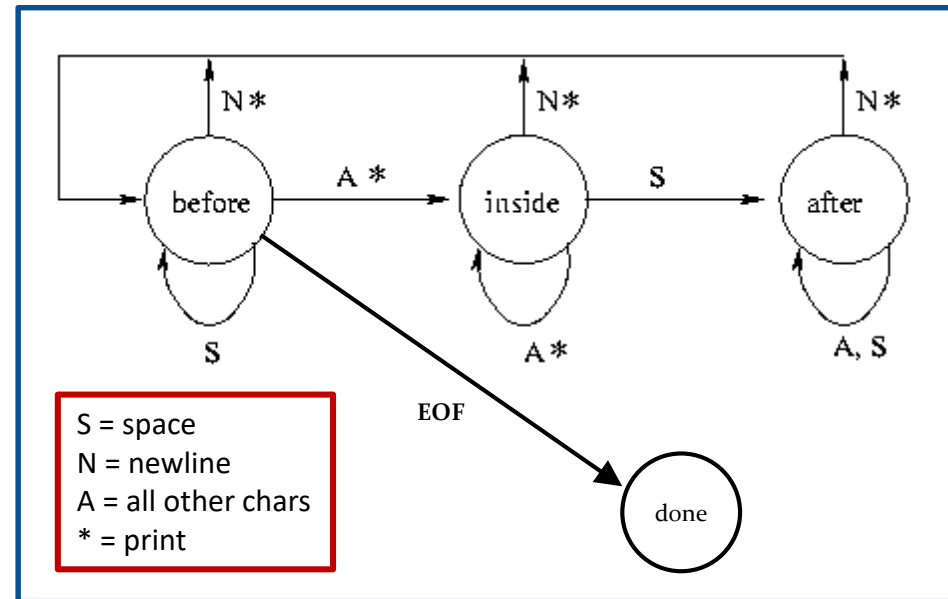
```
1. int main(void)
2. {
3.     int c;
4.     states state = before;
5.     while((c = getchar()) != EOF) {
6.         edges edge = lookup(state, c);
7.         edge.action();
8.         state = edge.next;
9.     }
10.    return 0;
11.}
```



# FSM: table-based solution

- Transition:
  - action
  - next state

```
1. int main(void)
2. {
3.     int c;
4.     states state = before;
5.     while((c = getchar()) != EOF) {
6.         edges *edge = &lookup[state, c];
7.         edge->action(c);
8.         state = edge->next;
9.     }
10.    return 0;
11.}
```





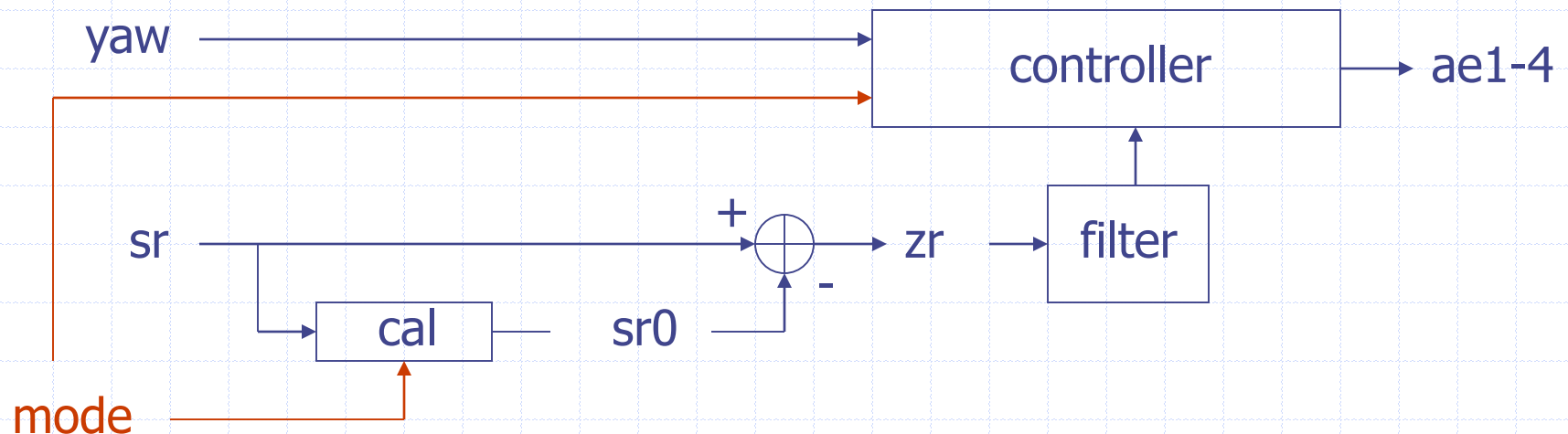
What's in the assignment?

# **BACK TO QUADCOPTERS**

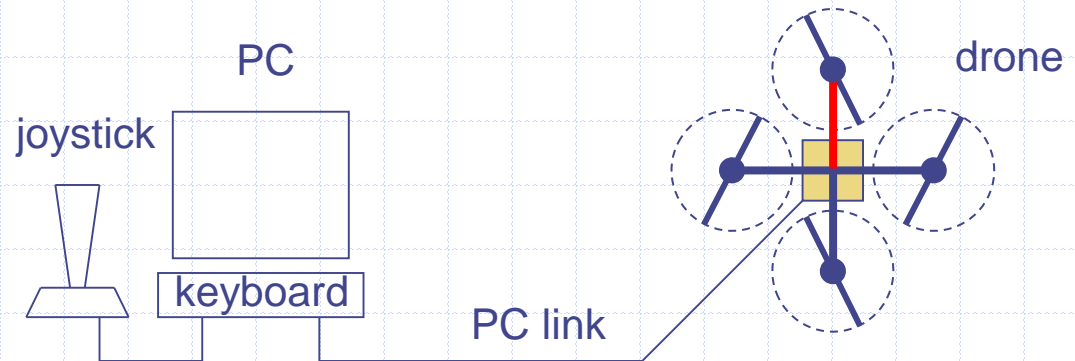


# Controller Modes

- controller mode: manual
- controller model: calibrate
- controller mode: control (yaw, pitch, roll)

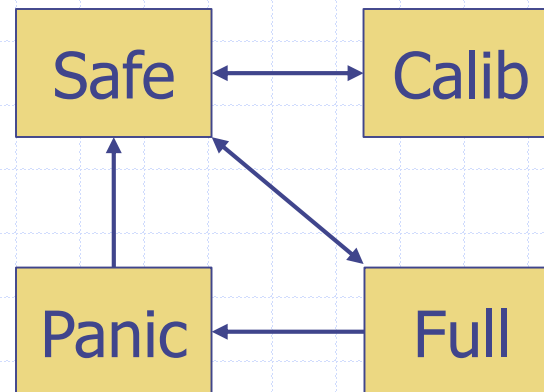


# Quadrupel: FSM

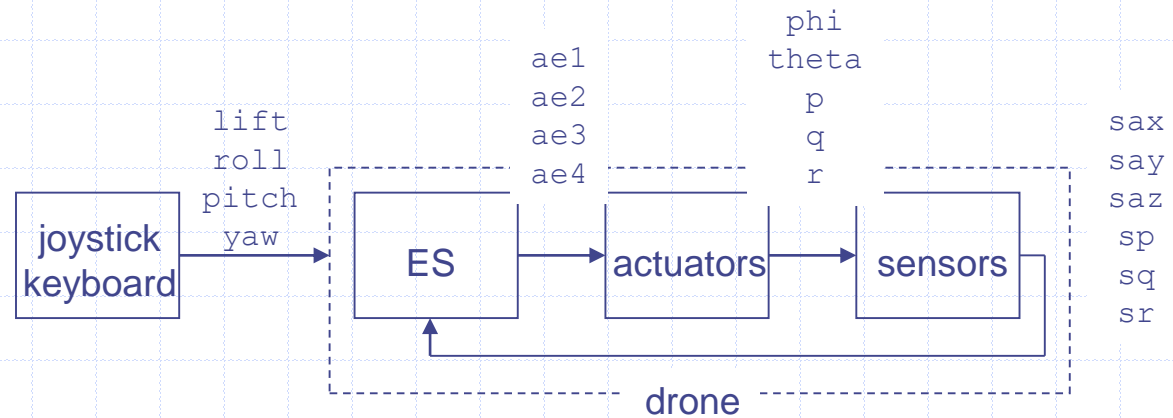


From the assignment

- Safe
- Panic
- Calibrate
- Full control
- ...



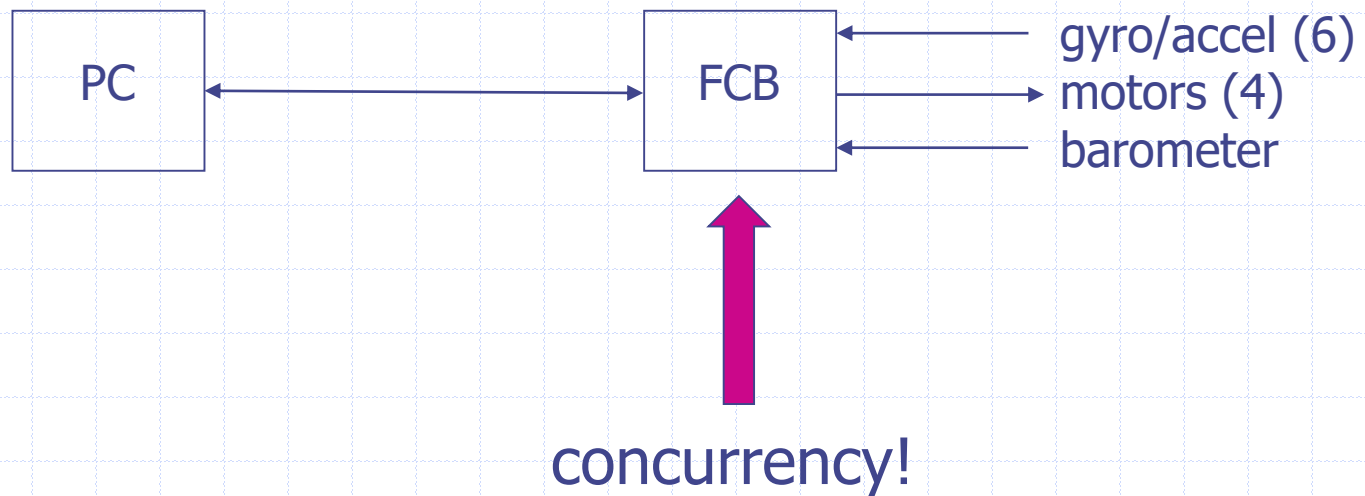
# Quadrapel: Control Loop



## Loop

- Read sensors
- Compare with set points
- Set motor values

# Quadrapel: FSM + control loop



# 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?!

# System Architecture (today!)

- ◆ Functional decomposition
- ◆ Who does what?
- ◆ Interfaces



# Software Architecture Survey

- ◆ Round-Robin (no interrupts)
  - ◆ 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

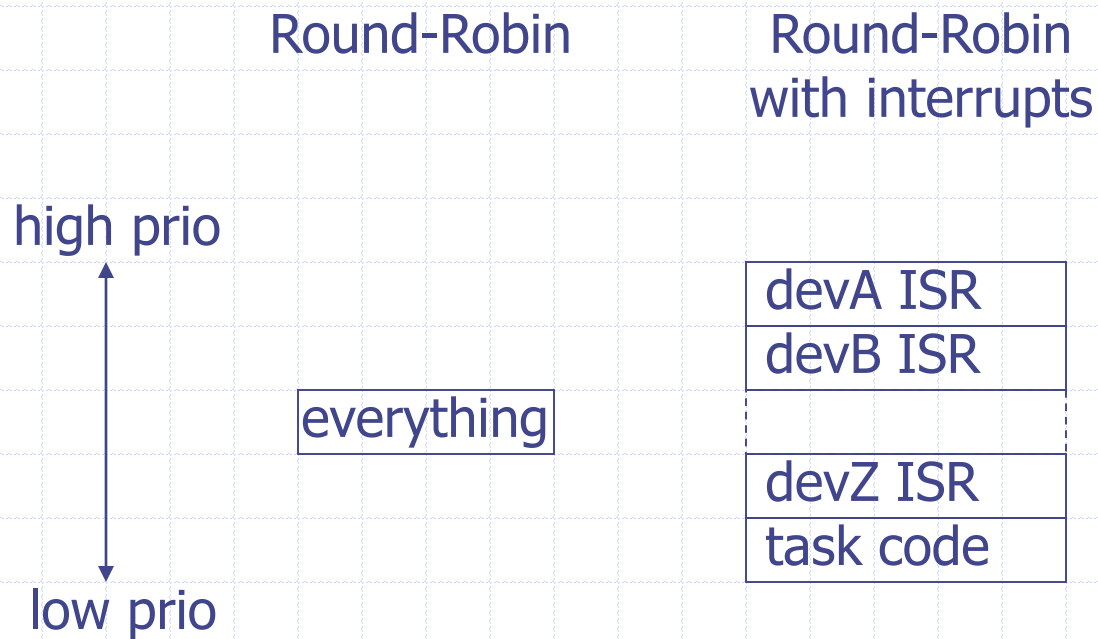
```
void    isr_deviceA(void)
{
    !! service immediate needs + assert flag A
}
..

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

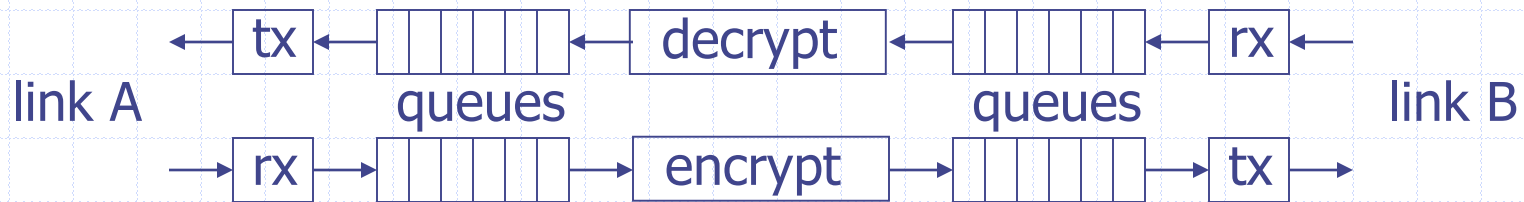
- ◆ ISR (interrupt vs. polling!): much better response time
- ◆ main still slow (i.e., lower priority than ISRs)

# RR versus RR+I

- ◆ Interrupt feature introduces priority mechanism



# Example: Data Bridge



- ◆ IRQs on char rx and tx devices (UART)
- ◆ rx ISR reads UART and queues char
- ◆ tx ISR simply asserts ready flag
- ◆ main reads queues, decrypt/encrypts, writes queues, writes char to UART & de-asserts flag (critical section!)
- ◆ architecture can sustain data bursts

# RR with Interrupts: Evaluation

- ◆ simple, and often appropriate (e.g., data bridge)
- ◆ main loop still suffers from stochastic response times
- ◆ interrupt feature has even aggravated this problem: fast ISR response at the expense of even slower main task (ISRs preempt main task because of their higher priority)
- ◆ this rules out RR+I for apps with CPU hogs
- ◆ moving workload into ISR is usually not a good idea as this will affect response times of other ISRs

# Function-Queue Scheduling

```
void    isr_deviceA(void)
{
    !! service immediate needs + queue A() at prio A
}
..

void    main(void)
{
    while (TRUE) {
        !! get function from queue + call it
    }
}

void    function_A(void) { !! service A }
..
```

# Function-Queue Sched: Evaluation

- ◆ task priorities no longer hardwired in the code (cf. RR architectures) but made flexible in terms of data
- ◆ each task can have its own priority
- ◆ response time of task  $T$  drops dramatically:  
from  $\sum_{i \in \text{all} \setminus T} t_i$  (RR) to  $\max_{i \in \text{all} \setminus T} t_i$  (FQS)
- ◆ still sometimes not good enough: need *preemption* at the *task* level, just like ISRs preempt tasks (in FQS a function must first finish execution before a context switch can be made)

# Real-Time OS

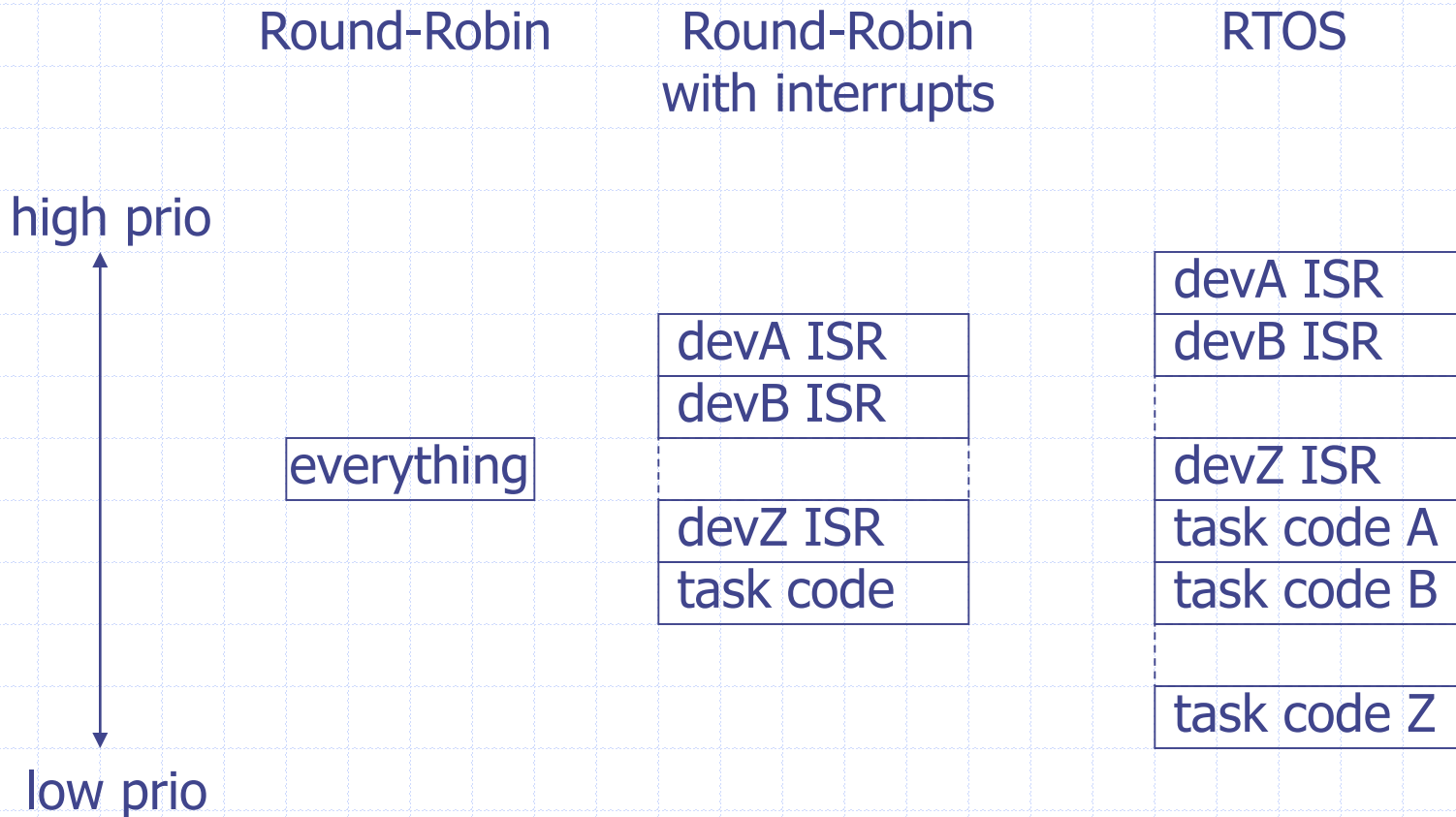
```
void    isr_deviceA(void)
{
    !! service immediate needs + set signal A
}
..

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

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



# Performance Comparison



# RTOS: Primary Motivation

- ◆ Task switching with *priority preemption*
- ◆ Additional services (semaphores, timers, queues, ..)
- ◆ Better code!
  - Having interrupt facilities, one doesn't always need to throw a full-fledged RTOS at a problem
  - However, in vast majority of the cases the code becomes (1) cleaner, (2) much more readable by another programmer, (3) less buggy, (4) more efficient
- ◆ The price: negligible run-time overhead and small footprint

# Interrupts are evil



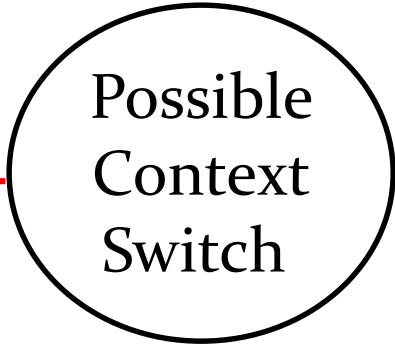
- Concurrent execution
- Shared data problem

# Shared-Data Problem?

```
void    isr_read_temps(void)
{
    iTemp[0] = peripherals[..];
    iTemp[1] = peripherals[..];
}
```

```
void    main(void)
{
    ...
    while (TRUE) {
        tmp0 = iTemp[0];
        tmp1 = iTemp[1];
        if (tmp0 != tmp1)
            panic();
    }
}
```

NOT ATOMIC!



Possible  
Context  
Switch

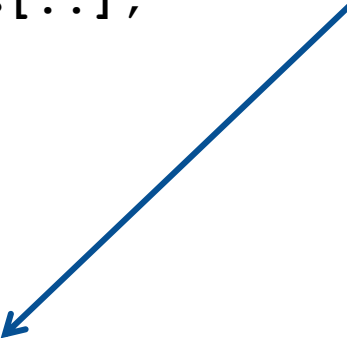
# Finding this bug...

- Can be very tricky
  - The bug does not occur always!
- Frequency depends on
  - The frequency of interrupts
  - Length of the critical section
- Problem can be difficult to reproduce
- Advise: double check the access on data used by ISR!

# Solving the Data-Sharing Problem?

```
void    isr_read_temps(void)
{
    iTemp[0] = peripherals[..];
    iTemp[1] = peripherals[..];
}

void    main(void)
{
    ...
    while (TRUE) {
        if (iTemp[0] != iTemp[1])
            panic();
    }
}
```



```
MOVE R1, (iTemp[0])
MOVE R2, (iTemp[1])
SUBTRACT R1,R2
JCOND ZERO, TEMP_OK
...
...
TEMP_OK:
...
```

# Solution #1

- Disable interrupts for the ISRs that share the data

```
...  
while (TRUE) {  
    !! DISABLE INT  
    tmp0 = iTemp[0];  
    tmp1 = iTemp[1];  
    !! ENABLE INT  
    if (tmp0 != tmp1)  
        panic();  
}
```

The critical section is now atomic

# Atomic & critical section

- A part of a program is atomic if it cannot be interrupted
  - Interrupts and program code share data
- *atomic* can also refer to mutual exclusion
  - Two pieces of code sharing data
  - They can be interrupted
- The instructions that must be atomic = *critical section*



# Be careful!

```
static int iSeconds, iMinutes;

void interrupt vUpdateTime(void)
{
    ++iSeconds;
    if (iSeconds>=60) {
        iSeconds=0;
        ++iMinutes;
    }
}

long lSeconds(void)
{
    disable();
    return (iMinutes*60+iSeconds);
    enable();
}
```

too little, too late 😞


# Function calls and enable()

- enable() can be a source of bugs!

```
void function1 ()
{
    ...
    // enter critical section
    disable();
    ...
    temp = f2 ();
    ...
    // exit critical section
    enable();
    ...
}
```

```
int f2 ()
{
    ...
    disable();
    ...
    enable();
    ...
}
```

should test if  
this is fine

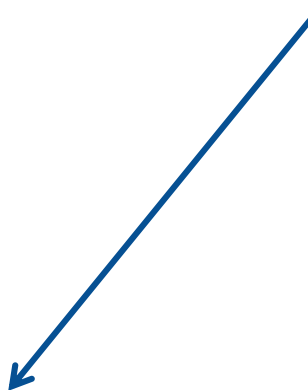


# More on shared-data...

```
static long int lSecondsToday;
```

```
void interrupt vUpdateTime()  
{  
    ...  
    ++lSecondsToday;  
    ...  
}
```

```
long lGetSeconds()  
{  
    return (lSecondsToday);  
}
```



```
MOVE R1, (lSecondsToday)  
MOVE R2, (lSecondsToday+1)  
...  
RETURN
```

# Any issues here?

```
static long int lSecondsToday;
```

```
void interrupt vUpdateTime()  
{  
    ++lSecondsToday;  
}
```

```
long lGetSeconds()  
{  
    long lReturn;
```

```
    lReturn = lSecondsToday;  
    while (lReturn!=lSecondsToday)  
        lReturn = lSecondsToday;
```

```
    return (lReturn);
```

```
}
```

} ingenious code  
without interrupts

# Any issues here?

```
volatile static long int lSecondsToday;
```

```
void interrupt vUpdateTime()  
{  
    ++lSecondsToday;  
}
```

```
long lGetSeconds()  
{  
    long lReturn;  
  
    lReturn = lSecondsToday;  
    while (lReturn != lSecondsToday)  
        lReturn = lSecondsToday;  
  
    return (lReturn);  
}
```

Otherwise compiler  
might optimize this  
code!

# Interrupt Latency

- Quick response to IRQ may be needed
- Depends on previous rules:
  - The longest period of time in which interrupts are disabled
  - The time taken for the higher priority interrupts
  - Overhead operations on the processor (finish, stop, etc.)
  - Context save/restore in interrupt routine
  - The work load of the interrupt itself
- worst-case latency =  $t_{\text{maxdisabled}} + t_{\text{higher prio ISRs}} + t_{\text{myISR}} + \text{context switches}$