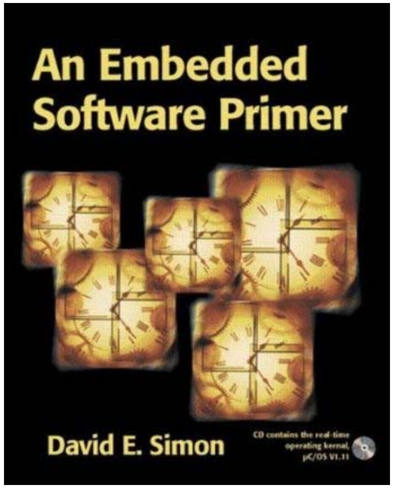
In4073
Embedded Real-Time Systems

Embedded Programming

Embedded Software

TI2726-B

- 2^{nd} 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)

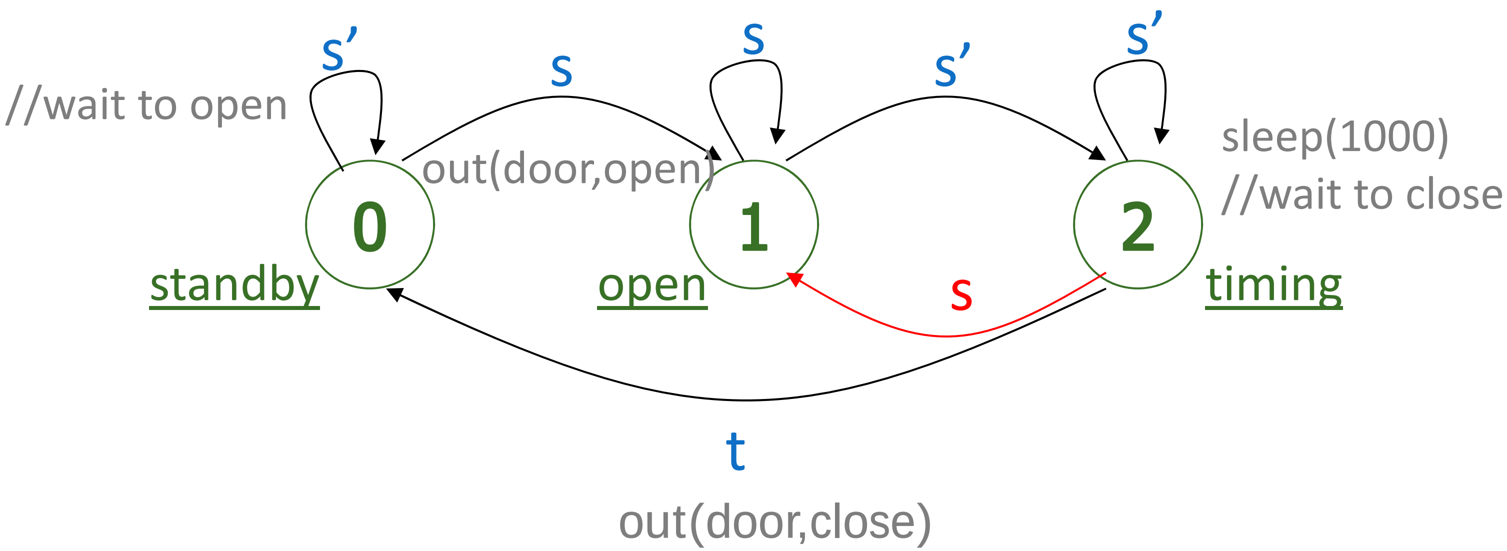
Embedded Programming Example

- 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);
}
```

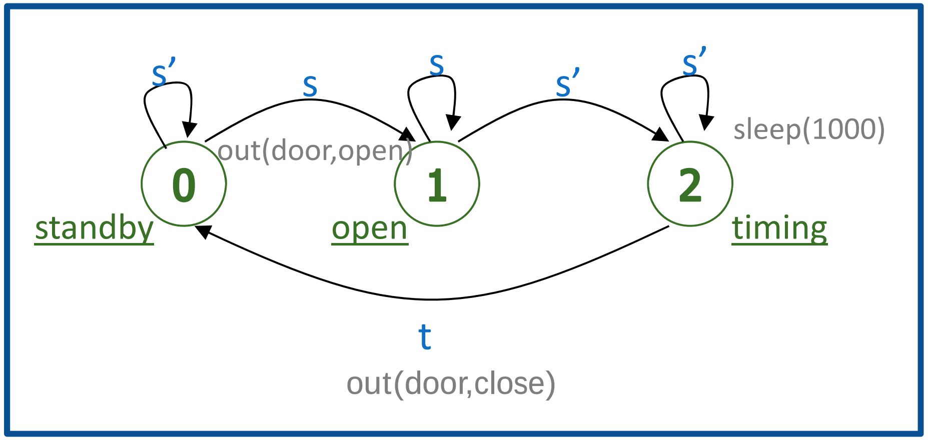
- Any issues with this code?

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[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

- too many loops
- duplicate EOF corner casing

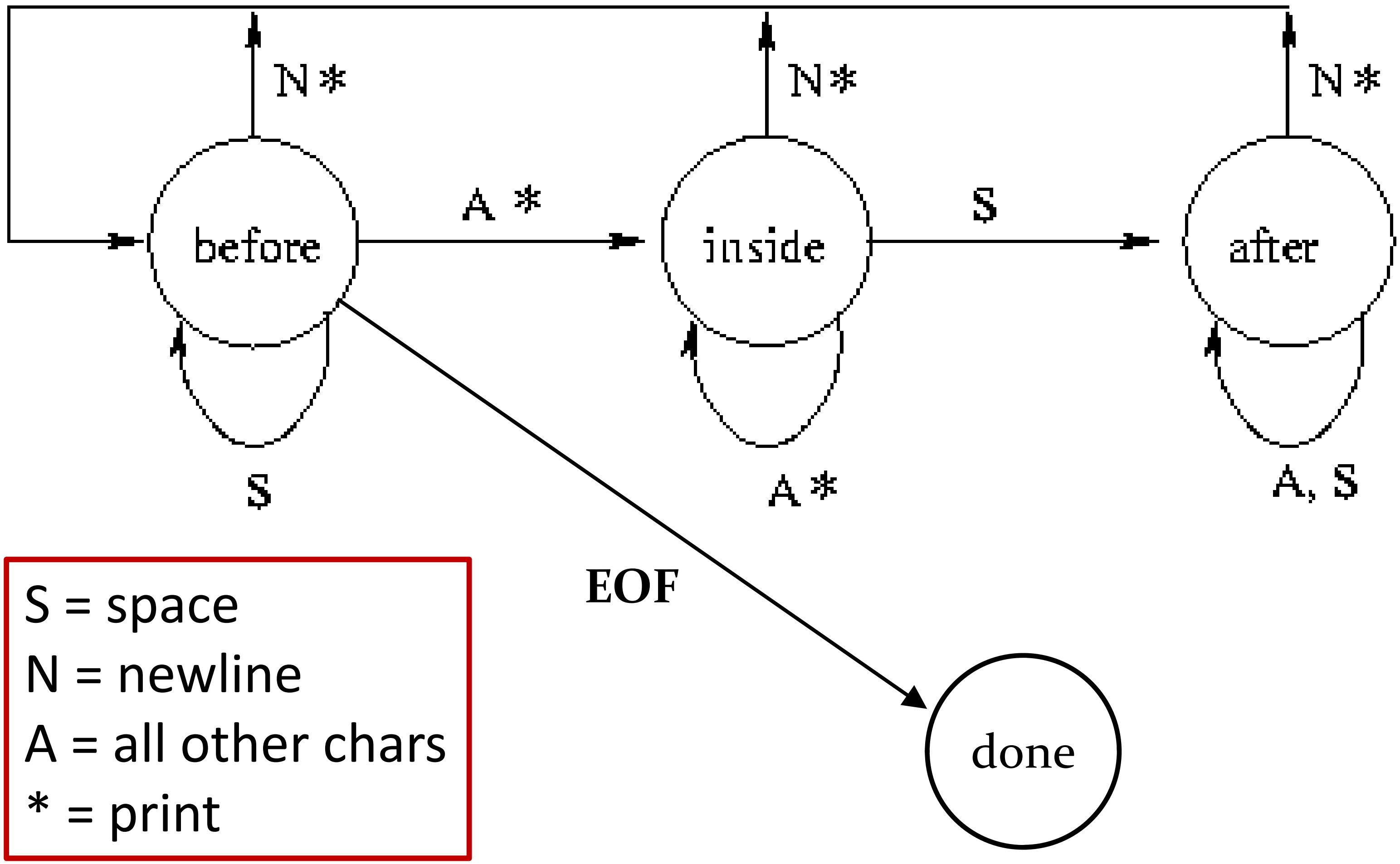
```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    do {
        do
            c = getchar();
        while(c == ' ');
        while(!isspace(c) && c != '\n‘ && c != EOF) {
            putchar(c);
            c = getchar();
        }
        putchar('\n');
        while(c != '\n‘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

```
int main(void)
{
    enum states {
        before, inside, after
    } state;
    int c;
    state = before;
    while((c = getchar()) != EOF) {
        switch(state) {
            case before:
                if(c != ' ') {
                    putchar(c);
                    if(c != '\n')
                        state = inside;
                }
                break;
            case inside:
```

- 1 loop
- 1 case for EOF checking

FSM-based solution

```
        case inside:
            if(!isspace(c))
                putchar(c);
            else if(c == '\n') {
                putchar('\n');
                state = before;
            } else
                state = after;
            break;
        case after:
            if(c == '\n') {
                putchar('\n');
                state = before;
            }
            break;
        default:
            fprintf(stderr, "unknown state %p\n", state);
            abort();
```

defensive programming!

Refactored solution

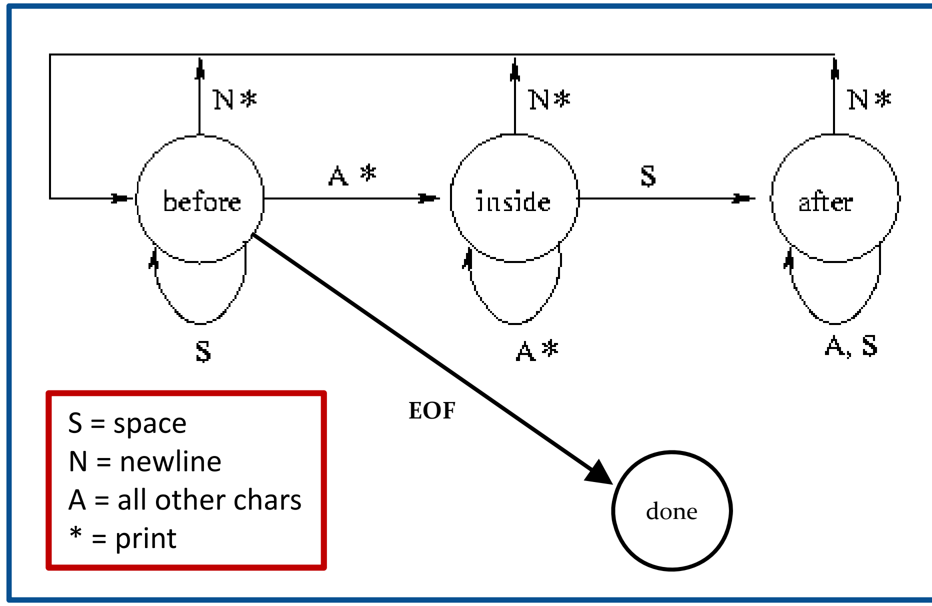
- lifted loop

```
enum states { before, inside, after };
void step(enum states *state, int c)
{
    switch(*state) {
        case before: ... *state = inside; ...
        case inside: ... *state = after; ...
        case after:  ... *state = before; ...
    }
}
int main(void)
{
    int c;
    enum states state = before;
    while((c = getchar()) != EOF) {
        step(&state, c);
    }
    return 0;
}
```

FSM: table-based solution

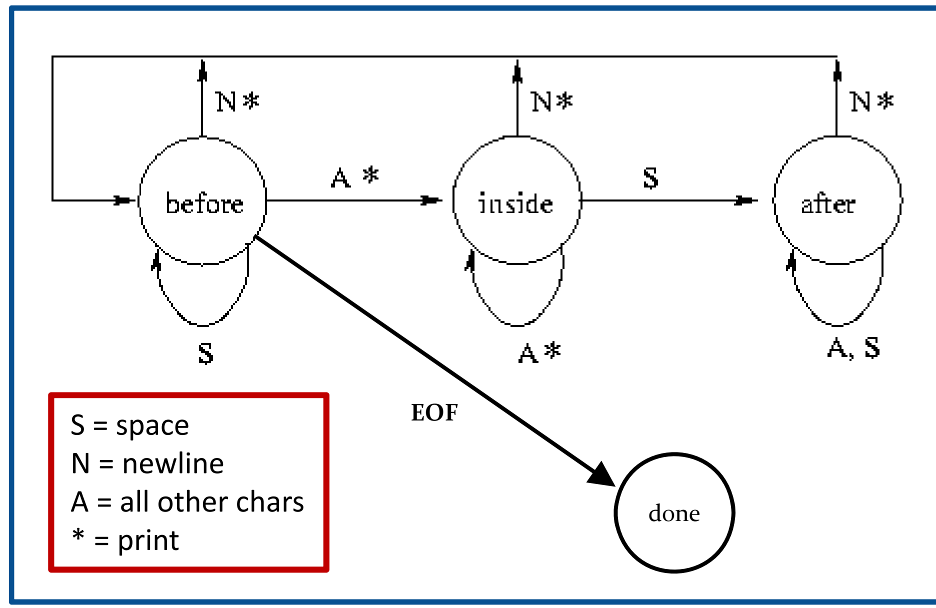
- Transition:
  - action
  - next state

```
int main(void)
{
    int c;
    states state = before;
    while((c = getchar()) != EOF) {
       edges edge = lookup(state, c);
        edge.action();
        state = edge.next;
    }
    return 0;
}
```



FSM: table-based solution

- Transition:
  - action
  - next state



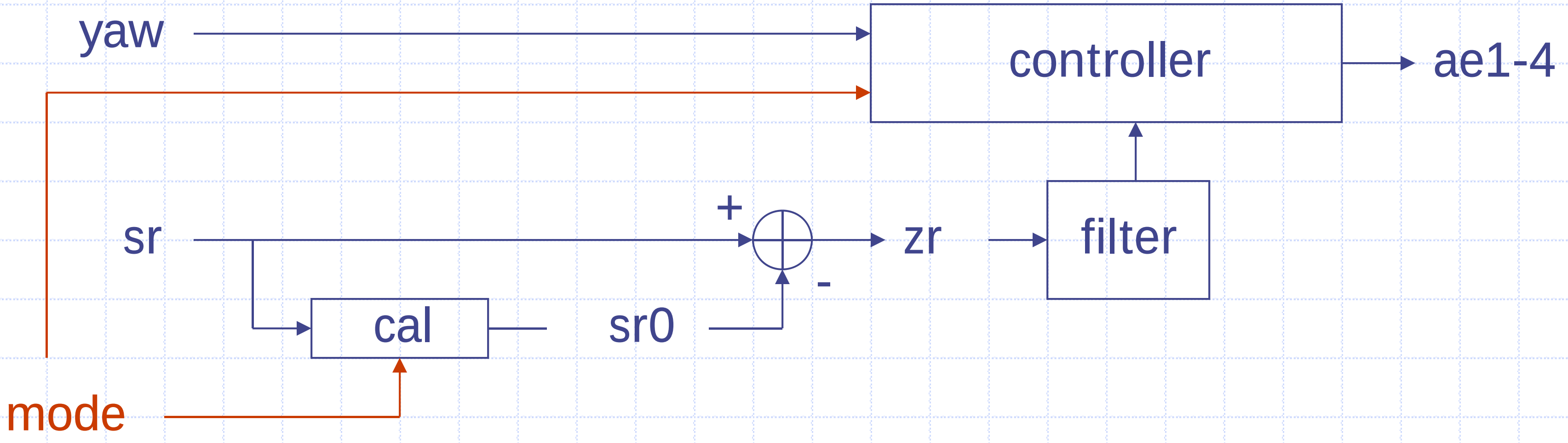

```c
int main(void)
{
      int c;
      states state = before;
      while((c = getchar()) != EOF) {
         edges *edge = &lookup[state, c];
         edge->action(c);
         state = edge->next;
      }
      return 0;
}
```

What’s in the assignment?

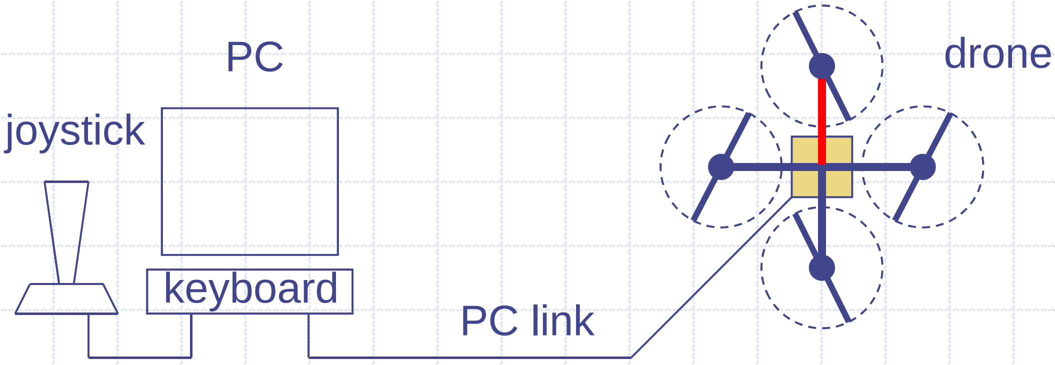
BACK TO QUADCOPTERS

Controller Modes

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

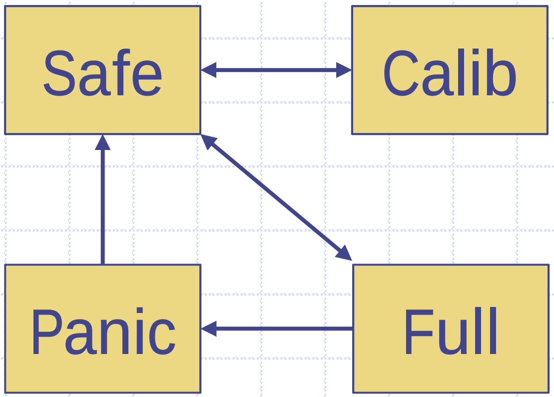


Quadrupel: FSM

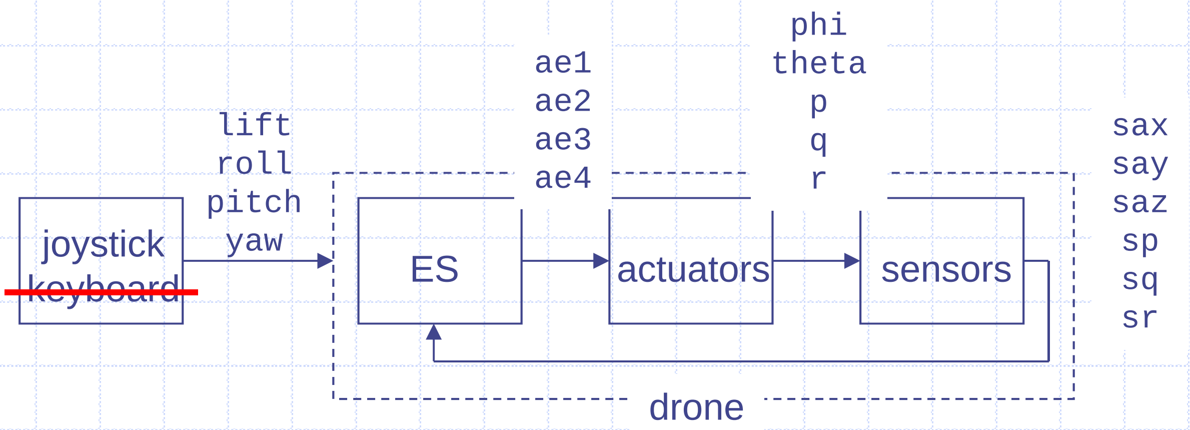


From the assignment

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



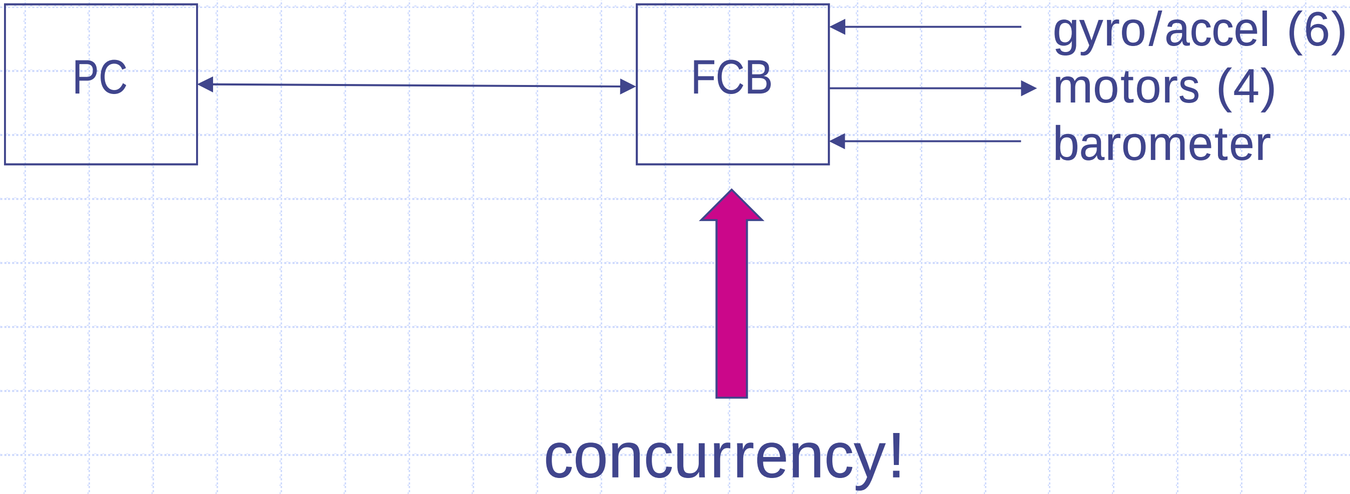
Quadrupel: Control Loop



Loop

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

Quadrupel: 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

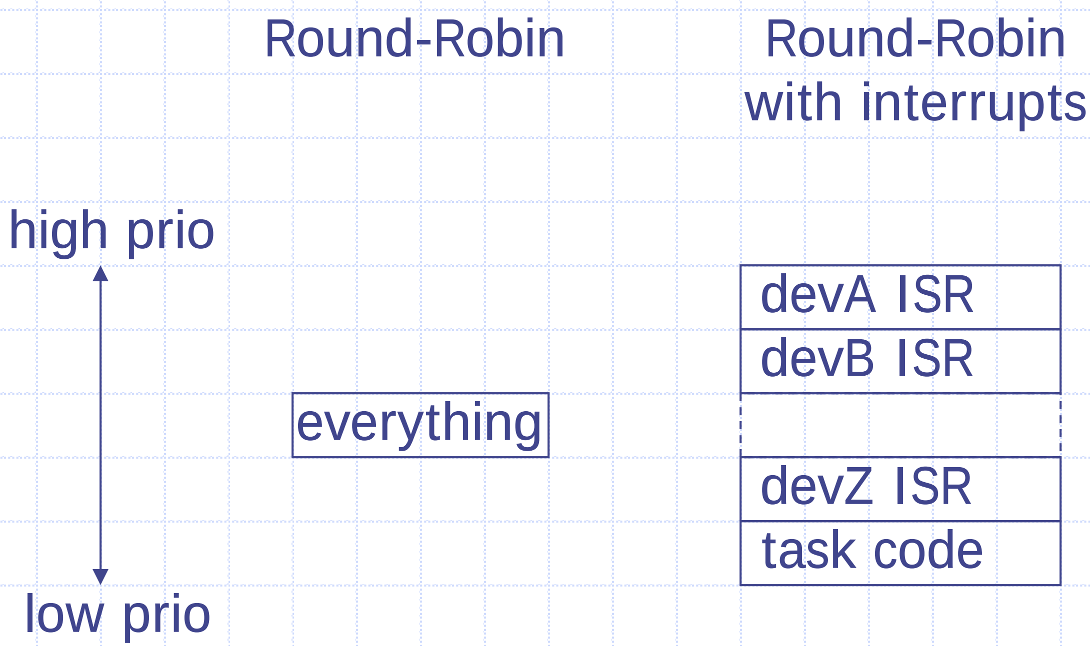
```
void   isr_deviceA(void)
{
       !! service immediate needs + assert flag A
}
..

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

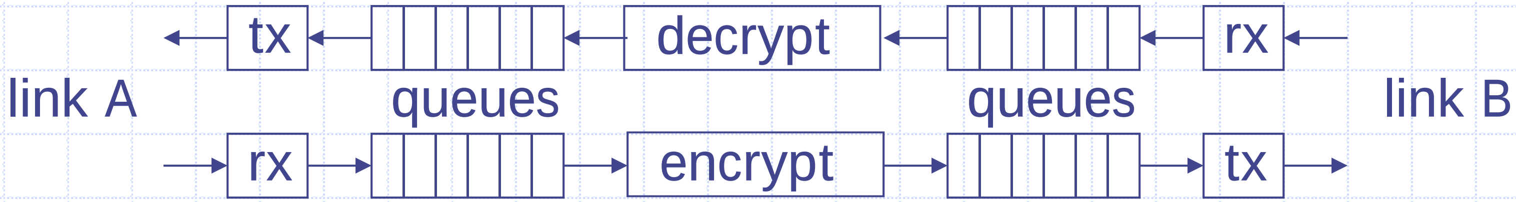
- ISR (interrupt vs. polling!): much better response time
- main still slow (i.e., lower priority then ISRs)

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 $\Sigma_{i \in all \setminus T}$ t_i (RR) to max $_{i \in all \setminus T}$ 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)

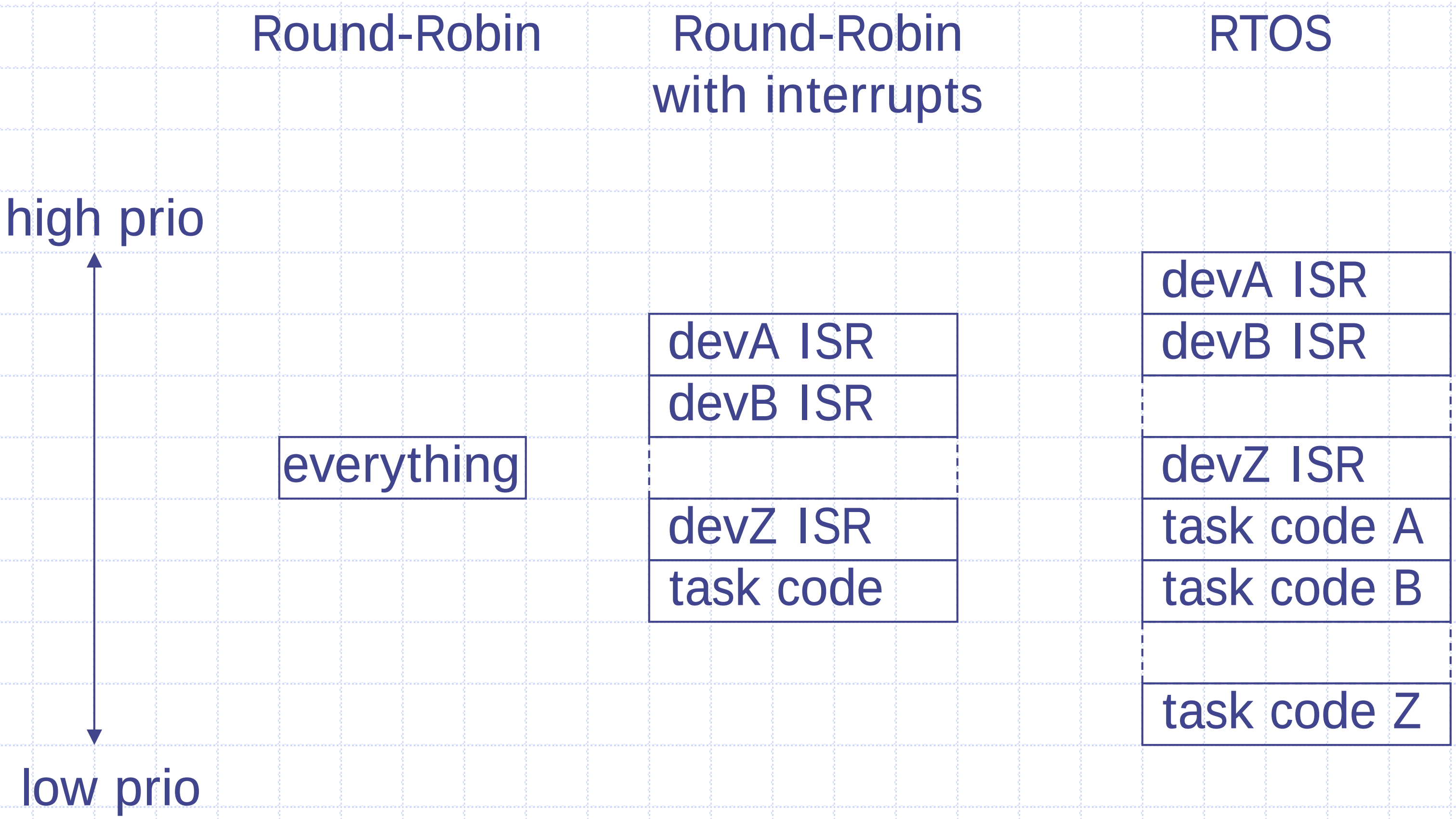
Real-Time OS

```
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

Book: page 92 ...

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])
SUBSTRACT 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_maxdisabled + t_higher prio ISRs + t_myISR + context switches