In4073
Embedded Real-Time Systems

Embedded Programming
Embedded Software

TI2726-B

- 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)
  - 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):

```c
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 programming¹

- 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.

¹https://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.
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. }

Ad-hoc solution

- too many loops
- duplicate EOF corner casing
S = space
N = newline
A = all other chars
* = print
EOF
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 != '
')
14.                         state = inside;
15.                 } 
16.                 break;
17.             case inside:
18.         }
19.     }
20. }

• 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;
  } else
    state = after;
break;
default:
  fprintf(stderr, "unknown state %p\n", state);
  abort();
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.     }  
9. }  
10. int main(void)  
11. {  
12.     int c;  
13.     enum states state = before;  
14.     while((c = getchar()) != EOF) {  
15.         step(&state, c);  
16.     }  
17.     return 0;  
18. }
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

```c
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
- …
Quadrupel: Control Loop

Loop
- Read sensors
- Compare with set points
- Set motor values
Quadrupel: FSM + control loop

PC → FCB

gyro/accel (6)
motors (4)
barometer

concurrency!
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

```c
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

```c
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

Round-Robin
Round-Robin with interrupts

high prio

devA ISR
devB ISR
devZ ISR
task code

everything

low prio
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/encryts, 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

```c
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:
  
  $$\sum_{i \in \text{all} \setminus T} t_i \text{ (RR)} \quad \text{to} \quad \max_{i \in \text{all} \setminus T} t_i \text{ (FQS)}$$
- 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

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

```c
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

<table>
<thead>
<tr>
<th>Round-Robin</th>
<th>Round-Robin with interrupts</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>high prio</td>
<td>devA ISR</td>
<td>devA ISR</td>
</tr>
<tr>
<td></td>
<td>devB ISR</td>
<td>devB ISR</td>
</tr>
<tr>
<td></td>
<td>devZ ISR</td>
<td>devZ ISR</td>
</tr>
<tr>
<td>everything</td>
<td></td>
<td>task code Z</td>
</tr>
<tr>
<td>low prio</td>
<td>devZ ISR</td>
<td>task code B</td>
</tr>
<tr>
<td></td>
<td>task code A</td>
<td>task code Z</td>
</tr>
<tr>
<td></td>
<td>task code Z</td>
<td></td>
</tr>
</tbody>
</table>
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
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();
    }
}
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?

```c
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
static int iSeconds, iMinutes;

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

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

- enable() can be a source of bugs!

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

int f2 ()
{
    ...
    disable();
    ...
    enable();
    ...
    return; // should test if this is fine
}
static long int lSecondsToday;

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

long lGetSeconds()
{
    return (lSecondsToday);
}
static long int lSecondsToday;

void interrupt vUpdateTime()
{
    ++lSecondsToday;
}

long lGetSeconds()
{
    long lReturn;

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

    return (lReturn);
}
Any issues here?

```c
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_ISR + t_myISR + context switches