Asynchronism

Why?

How do you handle your communication flow?

Do you have times when you check certain communication?

Is certain communication interrupting you? – at any time?

Do you assign “importance levels” to your communication channels/sources?
Asynchronism

CPU

STM32L476 Discovery

... running its sequence of machine instructions.

CPU

STM32L476 Discovery

... running its sequence of machine instructions.

How to interact with all the other devices inside the MCU?

Asynchronism

STM32L476 Discovery

Multiplexed 24bit Σ-Δ-DAConverter with stereo power amp

9 axis motion sensor (underneath display):
- 3 axis accelerometer
- 3 axis gyroscope
- 3 axis magnetometer

Current meter to MCU 60 nA... 50 mA

Headphone jack

USB OTG

Microphone

Asynchronism

STM32L476 Discovery

STM32L476 Discovery

CPU

Reset

STM32L476 Discovery

STM32L476 Discovery

Power

Over current

LCD

Debugger state

User LEDs

OTG LEDs

Joystick

Asynchronism
All external devices need to be “checked” by asking for their status.
This should usually happen (semi-) regularly.

This will lead to a loop of polling requests.

- Maximal latencies can be calculated straightforward.
- Simplicity of design (with small number of devices).
- Fastest option with small number of devices (like: one).
- All devices will need to wait their turn
  ... even if this device is the only one with new data!
- The “main” program transforms into one large loop which can
  be hard to handle in terms of scalable program design.
- Events or data can be missed.
Interrupts

- One or multiple lines wired directly into the sequencer
- Required for: Pre-emptive scheduling, timer-driven actions, transient hardware interactions, ...
- Usually preceded by an external logic ("interrupt controller") which accumulates and encodes all external requests.

On interrupt (if unmasked):
- CPU stops normal sequencer flow.
- Lookup of interrupt handler’s address
- Current IP and state pushed onto stack.
- IP set to interrupt handler.

Asynchronism

Interrupt processing

- Program
- Stack
- Interrupt handler

- Code
- Stack
- Interrupt handler

- Code
- Stack
- Interrupt handler
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
... or run some time
critical code...
Remove local variables

Program

Stack

Code

PC

Registers

Local variables

FP

SP

Global variables

Local variables

FP

SP
We successfully interrupted a sequence of operations ... and now the trick to get to the other side.
Asynchronism

Interrupt processing

The CPU hardware (!) did that, before anything was changed

Push registers
Declare local variables
Run handler code
... do some I/O ...
... or run some time
... critical code ...

Remove local variables
Interrupt processing

Interrupt handler

Program

Stack

Code

- Flags
  - User
  - Stack
  - Control
  - Critical
  - Parameters
  - Local variables
  - Global variables

- FP
  - Return address
  - Context
  - Parameters
  - Local variables

- PC

Push registers
Declare local variables
Run handler code
... or run some time critical code ...
Remove local variables
Pop registers

Return from interrupt

LR is loaded with a special value
Interrupt processing

Interrupt handler

Clear interrupt flag (Adjust priorities) (Re-enable interrupt)

Push other registers
Declare local variables
Run handler code
... do some I/O ...
... or run some time critical code ...

Remove local variables
Pop other registers
Interrupt handler

**Things to consider**

- Interrupt handler code can be interrupted as well.
- Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- Can you overrun a stack with interrupt handlers?

Interrupt handler code can be interrupted as well.

Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?

Can you overrun a stack with interrupt handlers?

Can we have one of those?
Asynchronism

Multiple programs

If we can execute interrupt handler code “concurrently” to our “main” program:

- Can we then also have multiple “main” programs?
Asynchronism

Context switch

Dispatcher

Process 1

Push registers
Declare local variables
Store SP to PCB 1

Process 2

Load SP from PCB 2

Dispatcher

Process 1

Push registers
Declare local variables
Store SP to PCB 1

Process 2

Remove local variables

Dispatcher
Anything else could go wrong?

- If there is neither communication nor contention between concurrent parts... all is easy... and boring.
- What happens if concurrent programs share data?
Asynchronism

Shared variables

Atomic load & store operations

-Assumption 1: every individual base memory cell (word) load and store access is atomic
-Assumption 2: there is no atomic combined load-store access

G : Natural := 0; -- assumed to be mapped on a 1-word cell in memory

task body P1 is
begin
G := 1
G := G + G;
end P1;

G := G + G;
end P2;

end P3;

What is the value of G?

Mutual exclusion ... or the lack thereof

Count : Integer := 0;

task body Enter is
begin
for i := 1 .. 100 loop
Count := Count + 1;
end loop;
end Enter;

end P1;

G := G + G;
end P2;

end P3;

What is the value in memory cell G after all three programs complete?

What is the value in memory cell G after both programs complete?

This is terrible!

Nobody is their right mind would analyse a program like that.

... are we missing something?

... is there an elegant way out?
Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

for_enter:
  cmp r1, #100
  bgt end_for_enter

for_leave:
  cmp r1, #100
  bgt end_for_leave

end_for_enter:
  for_leave:
      cmp r1, #100
      bgt end_for_leave

ldr r4, =Count
mov r1, #1

What is the value at address Count after both programs complete?
```assembly
for_enter:
  cmp r1, #100
  bgt end_for_leave
  mov r1, #1
  for_leave:
  str r1, [r4], #1

Critical section
```

```assembly
fail_lock_enter:
  cmp r4, [r4]
  bne fail_lock_leave
  ldrex r0, [r3]
  cmp r0, #0
  bne fail_lock_leave
  str rex r5, [r4]
  strex r5, [r4]
  dmb

Critical section
```

```assembly
fail_lock_leave:
  cmp r0, [r3]
  bne fail_lock_enter
  add r1, r1, #1
  b for_leave
```

Critical section

```assembly
Critical section
```

Any context switch needs to clear reservations.
Asynchronism

Mutual exclusion: atomic test-and-set operation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag;
begin
loop
[L := C; C := 1];
exit when L = 0;
------ change process
end loop;
------ critical_section_i;
C := 0;
end loop;
end Pi;

end for_enter:

fail_lock_leave:

for_leave:

end for_leave:

end

end

Asynchronism

Mutual exclusion: atomic exchange operation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin
loop
[L := C; C := Temp];
exit when L = 0;
------ change process
end loop;
------ critical_section_i;
L := 1; C := 0;
end loop;
end Pi;

end for_enter:

fail_lock_leave:

for_leave:

end for_leave:

end

end

Does that work?

Does that work?

Does that work?

Does that work?
Asynchronism

Mutual exclusion: atomic exchange operation

```plaintext
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin
loop
[Temp := L; L := C; C := Temp];
exit when L = 0;
------ change process
end loop;
L := 1; C := 0;
end loop;
end Pi;
```

Mutual exclusion: memory cell reservation

```plaintext
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin
loop
L := 1; C := 0;
end loop;
end Pi;
```

Does that work?

Mutual exclusion!, No deadlock!, No global live-lock!

Individual starvation possible! Busy waiting loops!
Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable \( S \) operating as a flag to indicate synchronization conditions
- an atomic operation \( P \) on \( S \) — for ‘passeren’ (Dutch for ‘pass’): \( P(S): [\text{as soon as } S > 0 \text{ then } S := S - 1] \) — this is a potentially delaying operation
- an atomic operation \( V \) on \( S \) — for ‘vrygeven’ (Dutch for ‘to release’): \( V(S): [S := S + 1] \)

> then the variable \( S \) is called a Semaphore.

Types of semaphores:

- **Binary semaphores**: restricted to \([0, 1]\) or [False, True] resp. Multiple \( V \) (signal) calls have the same effect than a single call.
- Atomic hardware operations support binary semaphores.
- Binary semaphores are sufficient to create all other semaphore forms.
- **General semaphores** (counting semaphores): non-negative number; (range limited by the system) \( P \) and \( V \) increment and decrement the semaphore by one.
- **Quantity semaphores**: The increment (and decrement) value for the semaphore is specified as a parameter with \( P \) and \( V \).

> All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e. ‘1’.
Semaphore: word 0x00000000

1. ldr r3, =Semaphore

2. `...`

3. wait (Semaphore)

4. `...`

5. Critical section

6. signal (Semaphore)

7. `...`

8. Any context switch needs to clear reservations:

9. wait_1:
   - ldr r0, [r3]
   - cmp r0, 0
   - beq wait_1; if Semaphore = 0
   - sub r0, #1; dec Semaphore
   - strex r1, r0, [r3]; try update
   - cmp r1, 0
   - bne wait_1; if touched
   - dmb; sync memory

10. `...`

11. Critical section

12. signal (Semaphore)

13. `...`

14. wait_2:
   - ldr r0, [r3]
   - cmp r0, 0
   - beq wait_2; if Semaphore = 0
   - sub r0, #1; dec Semaphore
   - strex r1, r0, [r3]; try update
   - cmp r1, 0
   - bne wait_2; if touched
   - dmb; sync memory

15. `...`

16. Critical section

17. signal (Semaphore)

18. `...`

19. Critical section

20. Critical section

21. ...
Semaphores

```plaintext
S : Semaphore := 1;

task body Pi is
begin
  loop
    ---- non_critical_section_i;
    wait (S);
  end loop;
end Pi;

task body Pj is
begin
  loop
    ---- non_critical_section_j;
    wait (S);
  end loop;
end Pj;
```

**G** Works too?!
Interrupts & Exceptions
- Concept
- Hardware/Software interaction
- Recursive interrupts

Concurrency & Synchronization
- Race conditions
- Synchronization
- Passing data

Mutual exclusion!, No global live-lock!
Works for any dynamic number of processes.
Individual starvation possible!
Deadlock possible!

Asynchronism

Summary

1. Interrupts & Exceptions
   - Concept
   - Hardware/Software interaction
   - Recursive interrupts

2. Concurrency & Synchronization
   - Race conditions
   - Synchronization
   - Passing data