Communication: Principles & Patterns

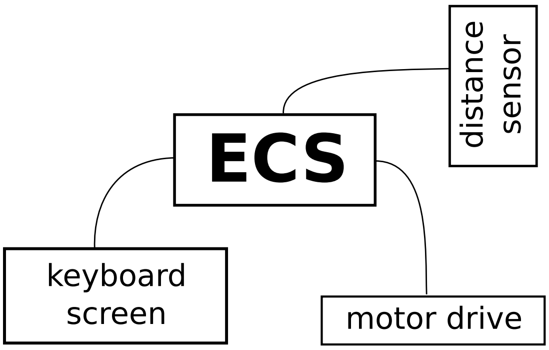
Herman Bruyninckx
Eindhoven University of Technology / KU Leuven
`http://people.mech.kuleuven.be/~bruyninc/`

Embedded Motion Control
June 3, 2015

Overview

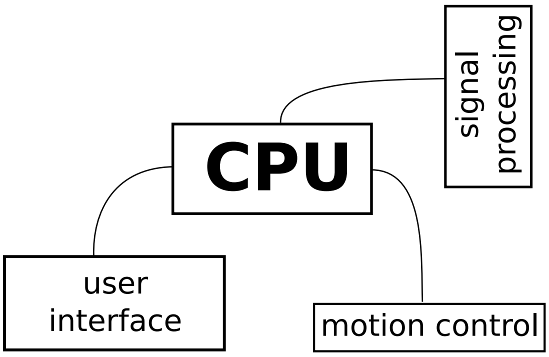
- Problem sketch
- Communication "stacks": OSI & Ethernet
- Hardware–hardware synchronization (data bus protocol)
- Hardware–software synchronization (Interrupt Service Routine)
- Collocated software–software synchronization (shared memory)
- Non-collocated software–software synchronization (message passing)

Problem 1: hardware schema



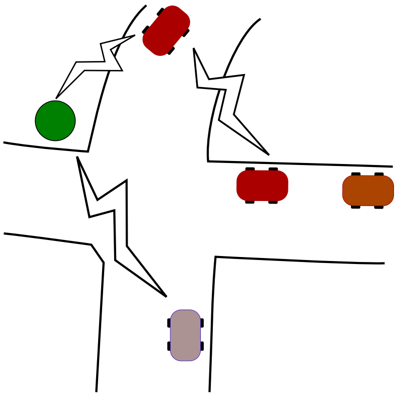
- How to read in sensor information?
- How to write out motor signals?
- How to interact with operator?
- . . .

Problem 2: software schema



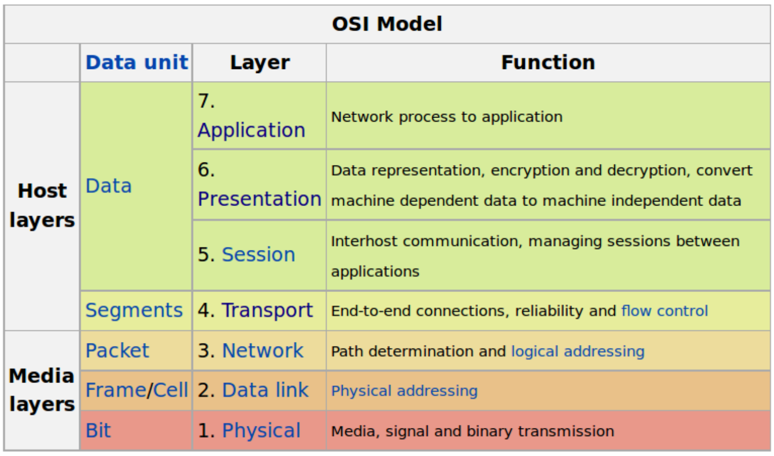
- How to coordinate the **execution** of the signal processing and the motor controller?
- What software to execute when operator pushes a button?
- ...

Problem 3: system-to-system schema



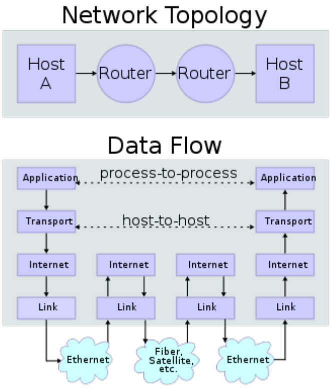
- How to get a **message** from one system to the other?
- What software to execute when a message is received by the communication hardware?
- ...

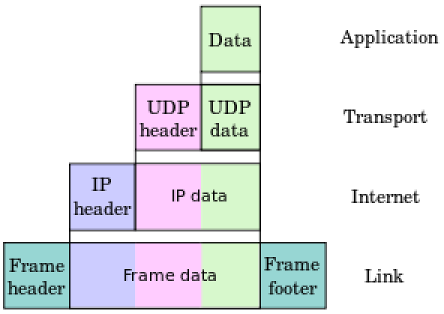
Communication “stacks”: OSI



OSI Model			
	Data unit	**Layer**	**Function**
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication, managing sessions between applications
	Segments	4. Transport	End-to-end connections, reliability and flow control
Media layers	Packet	3. Network	Path determination and logical addressing
	Frame/Cell	2. Data link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

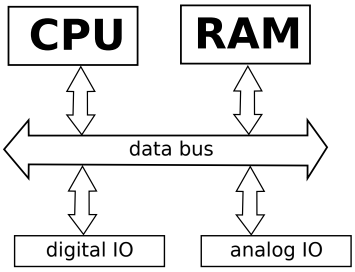
Communication "stacks": Internet





Dozens of protocols, e.g., **EtherCat** for hard-realtime control.

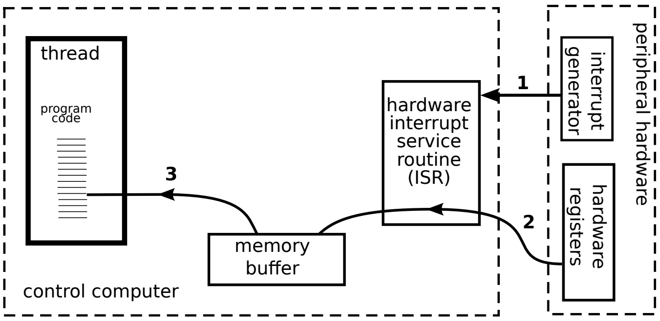
HW–HW synchronization —Data bus protocols—



- All "rectangles" are **electronic registers**
- The **hardware bus clock** triggers
  - when they can **change value**
  - when which register can **use the bus**

⇒ **one copy** of **consistent** data at a time

HW–SW synchronization —Interrupt Service Routine (ISR)—



- **hardware** *pre-empts* operating system software

⇒ **only consistent** data copies at all times!

- `http://en.wikipedia.org/wiki/Interrupt_handler`

HW–SW synchronization
—Hardware support—

- *Turn off* interrupts **while** processing one ISR.
- *Test-and-set*: to read/write "register word" **atomically**. (Available on most CPUs.)
- *Compare-and-swap*: to switch **pointers to buffers** atomically. (Available on *more and more* CPUs.)
- *Direct Memory Access* on bus: bus stops CPU to copy data from one place to another.

  (Stalls CPU! Bad for realtime, good for throughput. . .)

HW–SW synchronization
—Hardware support—

- *Turn off* interrupts **while** processing one ISR.
- *Test-and-set*: to read/write "register word" **atomically**. (Available on most CPUs.)
- *Compare-and-swap*: to switch **pointers to buffers** atomically. (Available on *more and more* CPUs.)
- *Direct Memory Access* on bus: bus stops CPU to copy data from one place to another.

  (Stalls CPU! Bad for realtime, good for throughput. . .)

Do: Keep ISR **short**! **Don't**: block in ISR!

Collocated SW–SW synchronization
—Shared memory—

Operating system support for synchronisation:

- *Mutex* ("*mut*ual *ex*clusion"):
  - synchronisation for shared access to data structures in memory
  - mutual exclusion is only **indirect**, i.e., via **code** fragments.
  - mutex has "owner", enforcable by OS.
- *Semaphore* for distinct memory spaces
- **Condition variable**!!! (see later)
- *Spin-lock* (only for inside *kernel*. . .)
- **Lock-free** data exchange.

See `http://people.mech.kuleuven.be/~bruyninc/ecs/AsynchronousSynchronization.pdf` for more details.

Condition variable

Condition variable has been introduced for two reasons:

1. It allows to make a task **sleep** until a certain application-defined **logical criterium** is satisfied.
2. It allows to make a task sleep **within a critical section**. (Unlike a semaphore.)

Condition variable

Condition variable has been introduced for two reasons:

1. It allows to make a task **sleep** until a certain application-defined **logical criterium** is satisfied.
2. It allows to make a task sleep **within a critical section**. (Unlike a semaphore.)

This is in fact two times the same reason, because the critical section is needed to evaluate the application-defined logical criterium atomically.

Condition variable (2)

CV = combination of:

1. **mutex lock**
2. **boolean expression** as logical “wake-up criterium”
3. **signal** that other tasks can fire to wake up the task blocked in the condition variable, so that it can re-check its boolean expression.

```
int sem_wait(sem_t *sem)
{
   pthread_mutex_lock(&sem->mutex);
   while (sem->count == 0) pthread_cond_wait(&sem->cond, &sem->mutex);
   sem->count--;
   pthread_mutex_unlock(&sem->mutex);
   return(0);
}
```

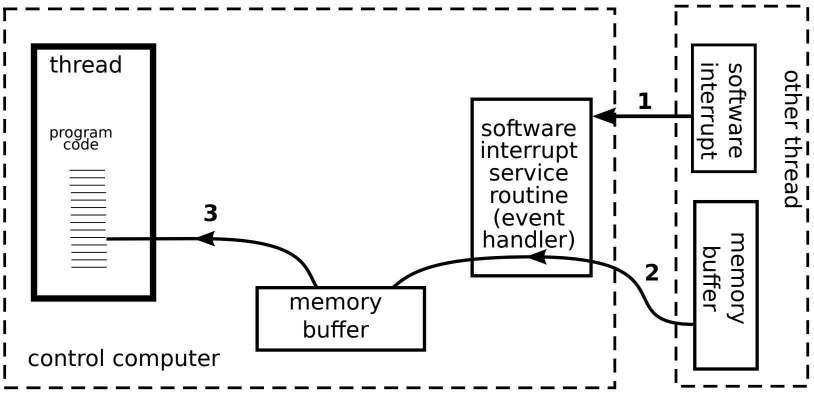
Condition variable (3)

Most important feature of CV: link to **logical condition** checking!

The lock allows to check the boolean expression *atomically* in a critical section, and to wait for the signal within that critical section.

It's the operating system's responsibility to release the mutex behind the back of the task, when it goes to sleep in the wait, and to take it again when the task is woken up by the signal.

Non-collocated synchronization —System-to-system message passing—

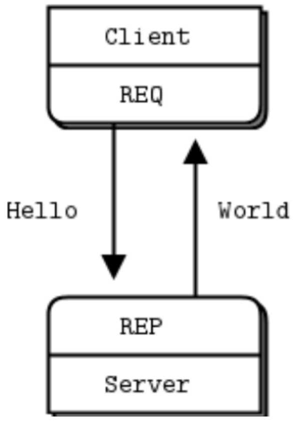


Message passing (2)

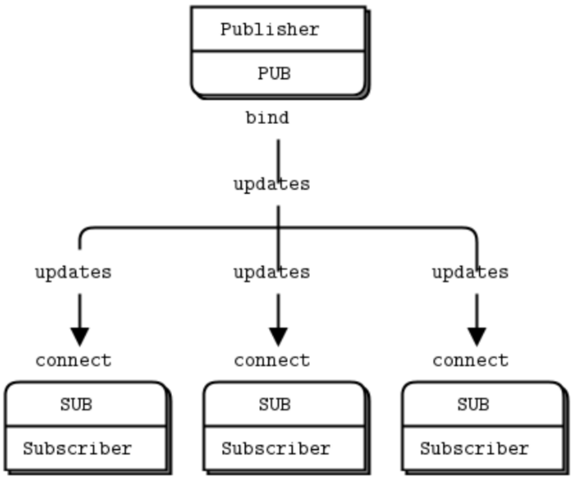
- Same **mechanism** as ISR generated by **hardware**.
- More variety in **policies**:
  - *protocols*: TCP, UDP, RTP, http, FTP,. . .
  - *buffering*: FIFO, circular, LIFO, lockless,. . .
  - *synchronisation*:
    - processes *wait* for each other (Concurrent Sequential Processes, "CSP")
    - *"high water – low water"* buffer overflow coordination
  - *Quality of Service* monitoring: heartbeat, bandwidth adaptation, message dropping,. . .
  - *security*: https, ssh,. . .
  - . . .

⇒ most often **handling** of message interrupt is **split** over *ISR* (blocking!) and *device driver* (non-blocking).

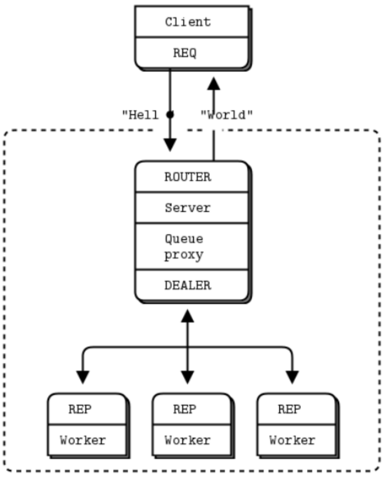
Communication Patterns
—Request–Reply—



Communication Patterns (2)
—Publish–Subscribe—



Communication Patterns (3)
—Router–Dealer—



Communication Patterns (4)

Lots of other protocols needed:

- service discovery
- broker
- tracing/logging
- . . .

Conclusions

- “communication” is a **very mature** subject

⇒ use one of the many, many **libraries**!
For example: ZeroMQ.

⇒ outsource it to specialists. . .

- most important in *your* system design: which **communication patterns** do I need?

⇒ don’t forget about **data models**, i.e., **what** has to be communicated;?

- don’t forget **shared memory**!
(“blackboard architecture”, “lockfree data exchange”, “zero-copy communication”,. . .)