

# Communication and synchronization of asynchronous activities

**Herman Bruyninckx**

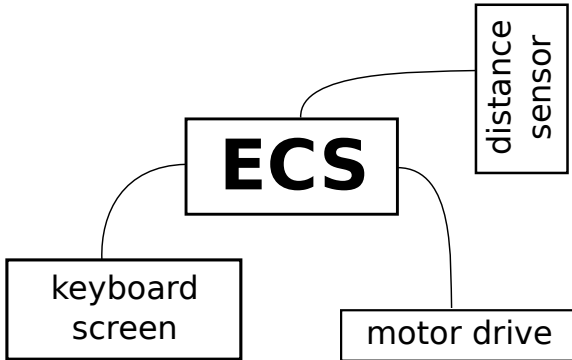
—  
Department of Mechanical Engineering  
K.U.Leuven, Belgium  
—

March 9, 2011

# Overview

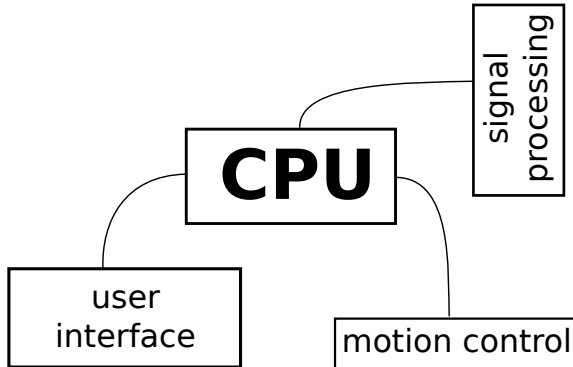
- ▶ **Given:** “high-level” hardware and software schemas
- ▶ **Required:** synchronisation:
  - ▶ Hardware–hardware synchronization  
(data bus protocol)
  - ▶ Hardware–software synchronization  
(Interrupt Service Routine)
  - ▶ Collocated software–software synchronization  
(shared memory)
  - ▶ Non-collocated software–software synchronization  
(message passing)

# Problem 1: hardware schema



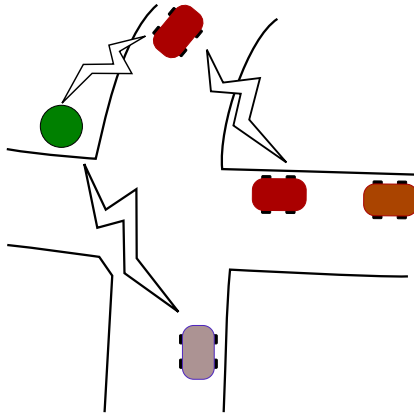
- ▶ How to read in sensor information?
- ▶ How to write out motor signals?
- ▶ How to interact with operator?
- ▶ ...

## 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?
- ▶ ...

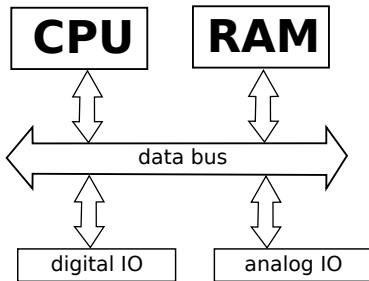
# Problem 3: system-to-system schema



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

# HW–HW synchronization

## —Data bus protocols—

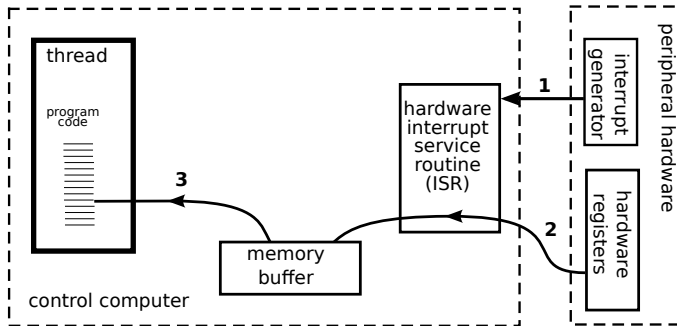


- ▶ All “rectangles” are **electronic registers**
- ▶ The bus clock defines (“coordinates”)
  - ▶ when they can **change value**
  - ▶ when which register can **use the bus**

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

# HW–SW synchronization

## —Interrupt Service Routine (ISR)—



# 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 *few* 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 *few* 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*:
  - ▶ 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*!!!
- ▶ *Spin-lock* (only for inside *kernel*...)
- ▶ *Lock-free* data exchange.

See <http://people.mech.kuleuven.be/~bruyinck/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 (cont'd)

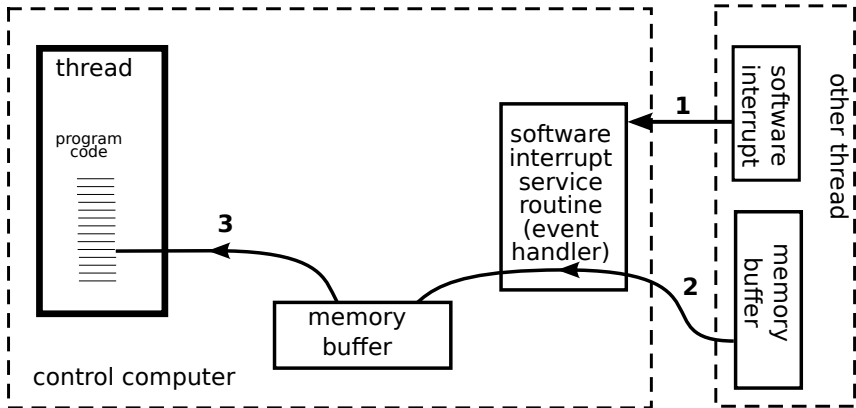
Most important feature of CV: link to **logical condition** checking!

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. . .
- ▶ . . . or **synchronous** event handling!

# Message passing (2)

- ▶ Same **mechanism** as ISR. . .
- ▶ . . . or **synchronous** event handling!
  
- ▶ More variety in **policies**:
  - ▶ Buffering (FIFO, circular, LIFO, lockless, . . . )
  - ▶ Synchronous messages  
(Concurrent Sequential Processes)
  - ▶ . . .

# Conclusions

Synchronisation is:

- ▶ **difficult**
- ▶ a major source of **indeterminism & logical errors**
- ▶ too often **non-configurable**

# Conclusions

Synchronisation is:

- ▶ **difficult**
- ▶ a major source of **indeterminism & logical errors**
- ▶ too often **non-configurable**

So:

- ▶ use **“middleware”, “frameworks”!**  
Bad news: not many good ones exist yet, certainly not for hard realtime...  
Good news: OrocOS.org, ROS.org, AUTOSAR.org
- ▶ **separate** “Five C’s”!
- ▶ **separate** library design and system architecture!