Architecture : Programming Input-Ouput

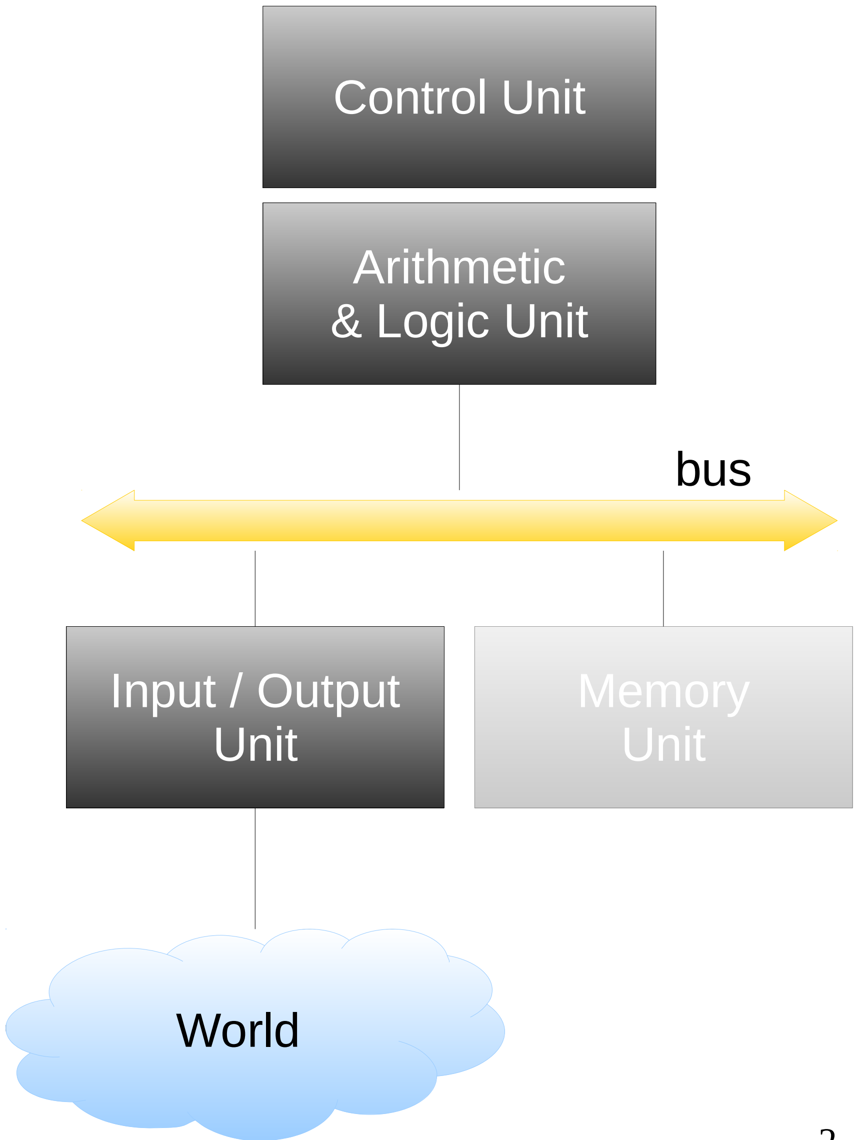
H. Cassé <casse@irit.fr>

Some schemes are gracefully provided wikipedia and openclipart under open license.

UNIVERSITE TOULOUSE III PAUL SABATIER
Université de Toulouse
Faculté Sciences et Ingénierie

Focusing on Input/Output

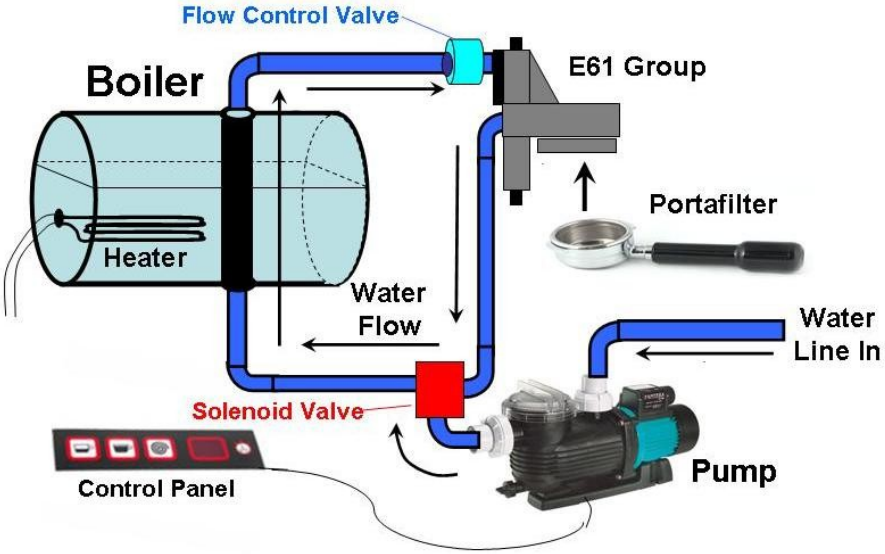
- Laptop / Desktop
  - input – keyboard, mouse
  - output – screen, printer
  - storage – hard disk, USB key
  - etwork
- embedded system
  - sensor – light, noise, speed, rotation, etc
  - actuator – motor (servo, linear, switch, ...)

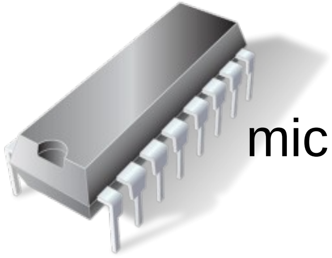


Home Appliance

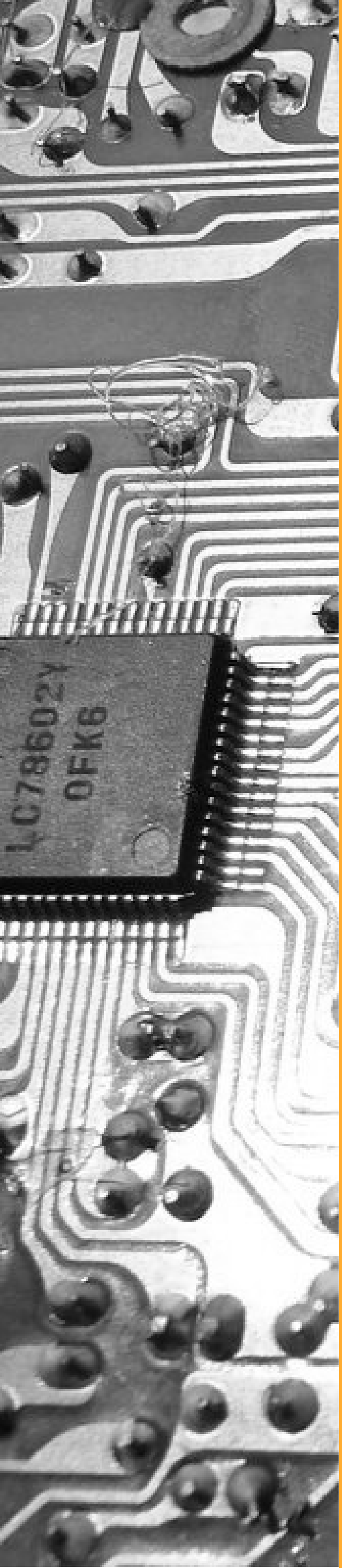
- characteristics
  - heterogeneous applications
  - general public
  - hidden
  - specific sensor/actuator
- constraints
  - mass production → cost
  - reliability
- examples
  - washing machine, oven, alarm clock, coffee machine, ...

Espresso Coffee Machine





microcontroler



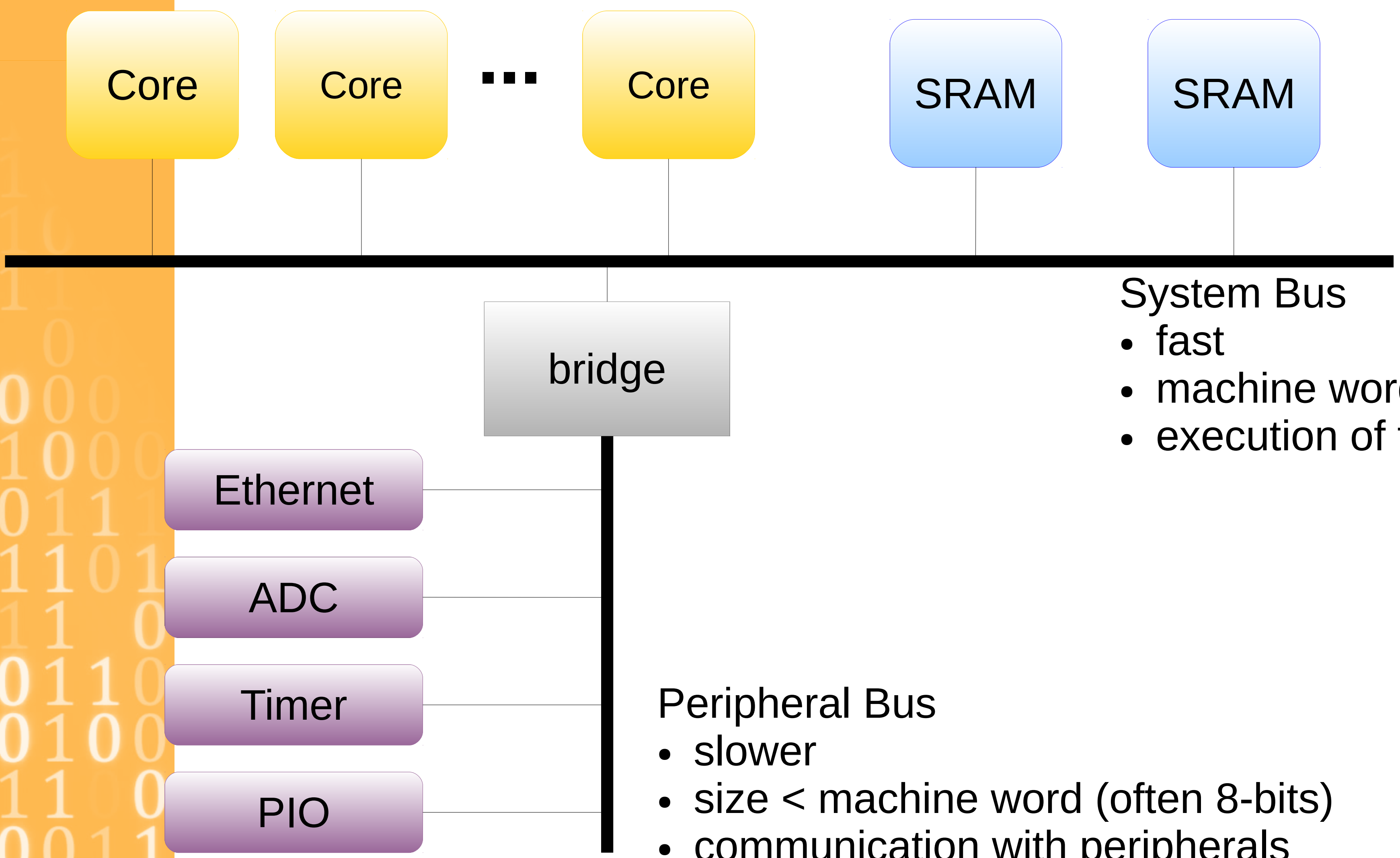
Overview

- Introduction
- **The microcontroler**
- I/O Management
- Using interrupts
- Sensors & Actuators

The microcontroller

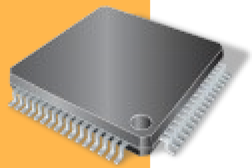
- on one chip
  - 1-n cores (computation units)
  - memories
    - Static RAM
    - ROM memories (flash)
  - drivers for peripherals
    - communication (serial, ethernet, wireless, SPI, I2C, CAN etc)
    - hardware driver (parallel IO, PWM, ADC, PWM, …)
    - timers

Organization



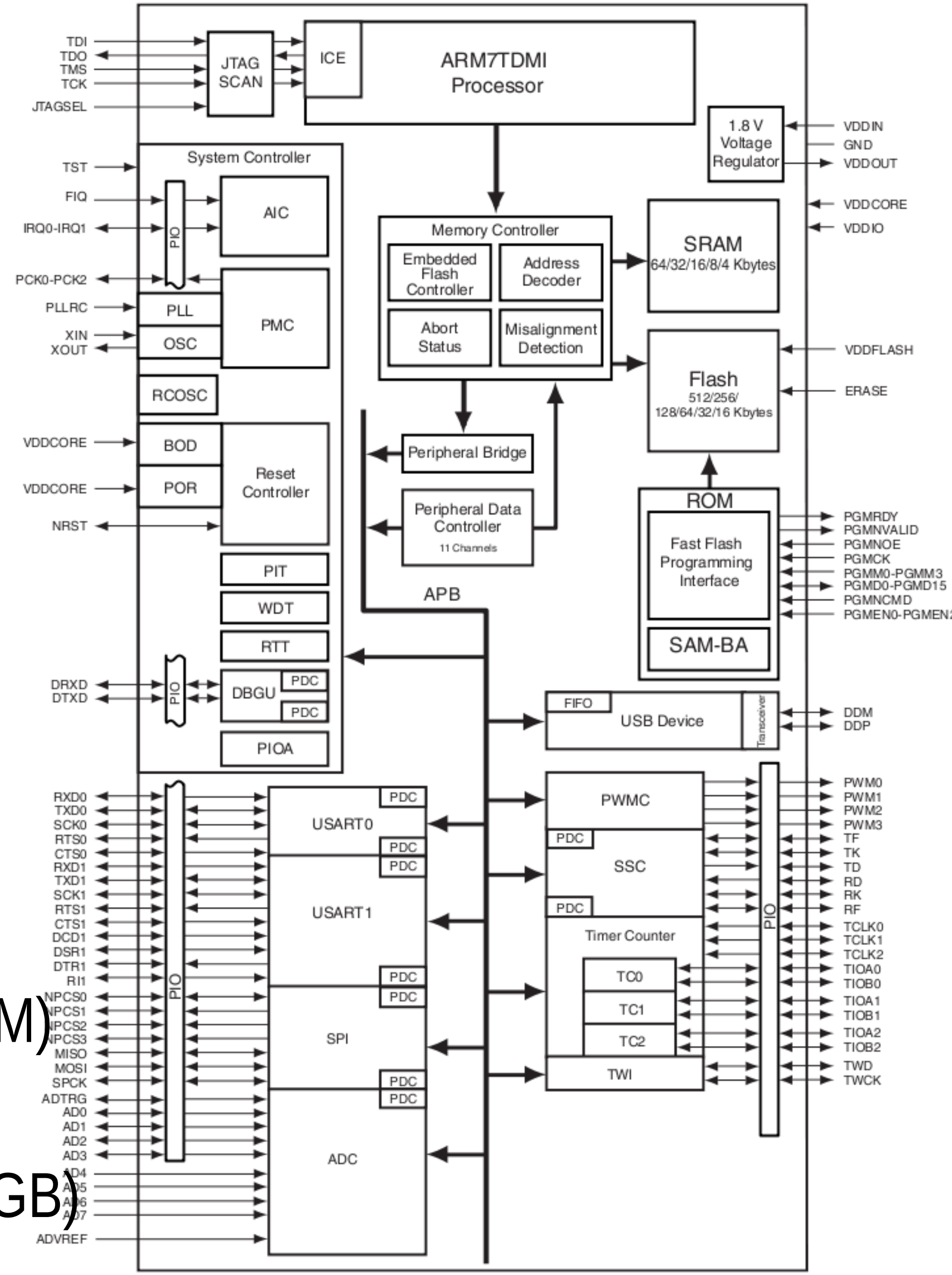
System Bus
- fast
- machine word width
- execution of the program

Peripheral Bus
- slower
- size < machine word (often 8-bits)
- communication with peripherals



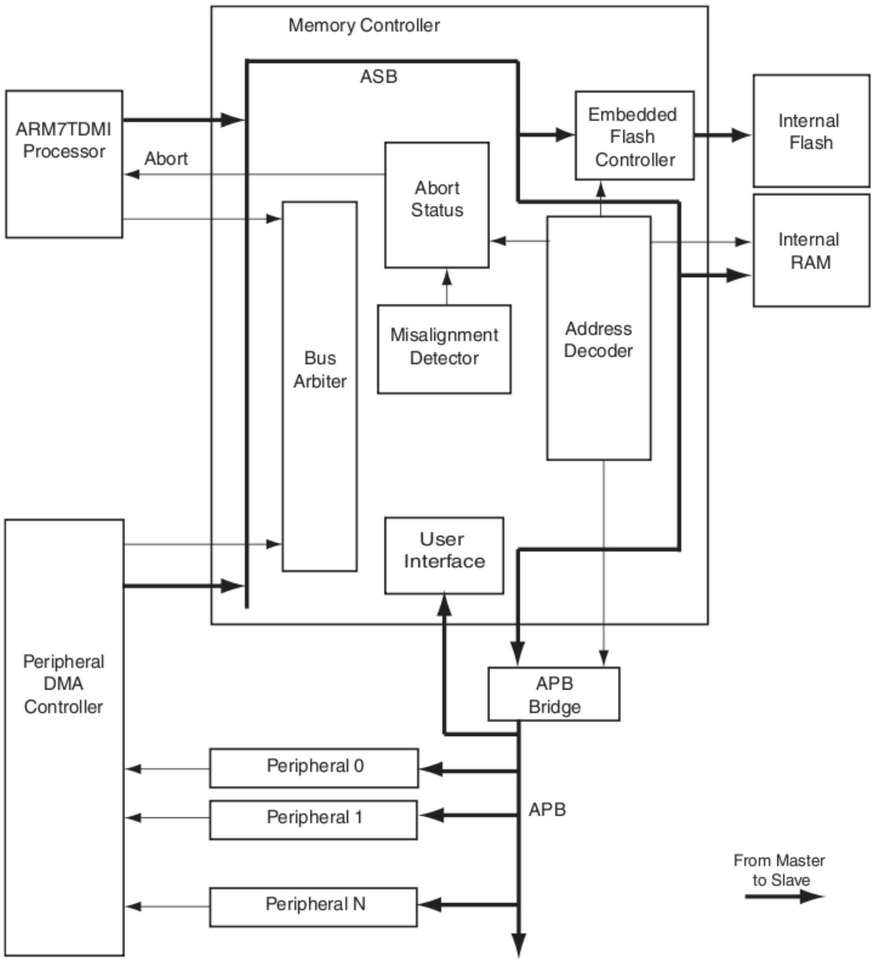
AT9SAM7S

- constructor – ATMEL
- 1 ARM v5T core
  (today laptops 4-8 cores)
- 32-bits
- 48 MHz
  (today i686 2-3 GHz)
- RISC instruction set
- 4-64 KB SRAM
  (today desktop 4-8 GB RAM)
- 16-256 KB flash
  (common USB stick – > 4 GB)

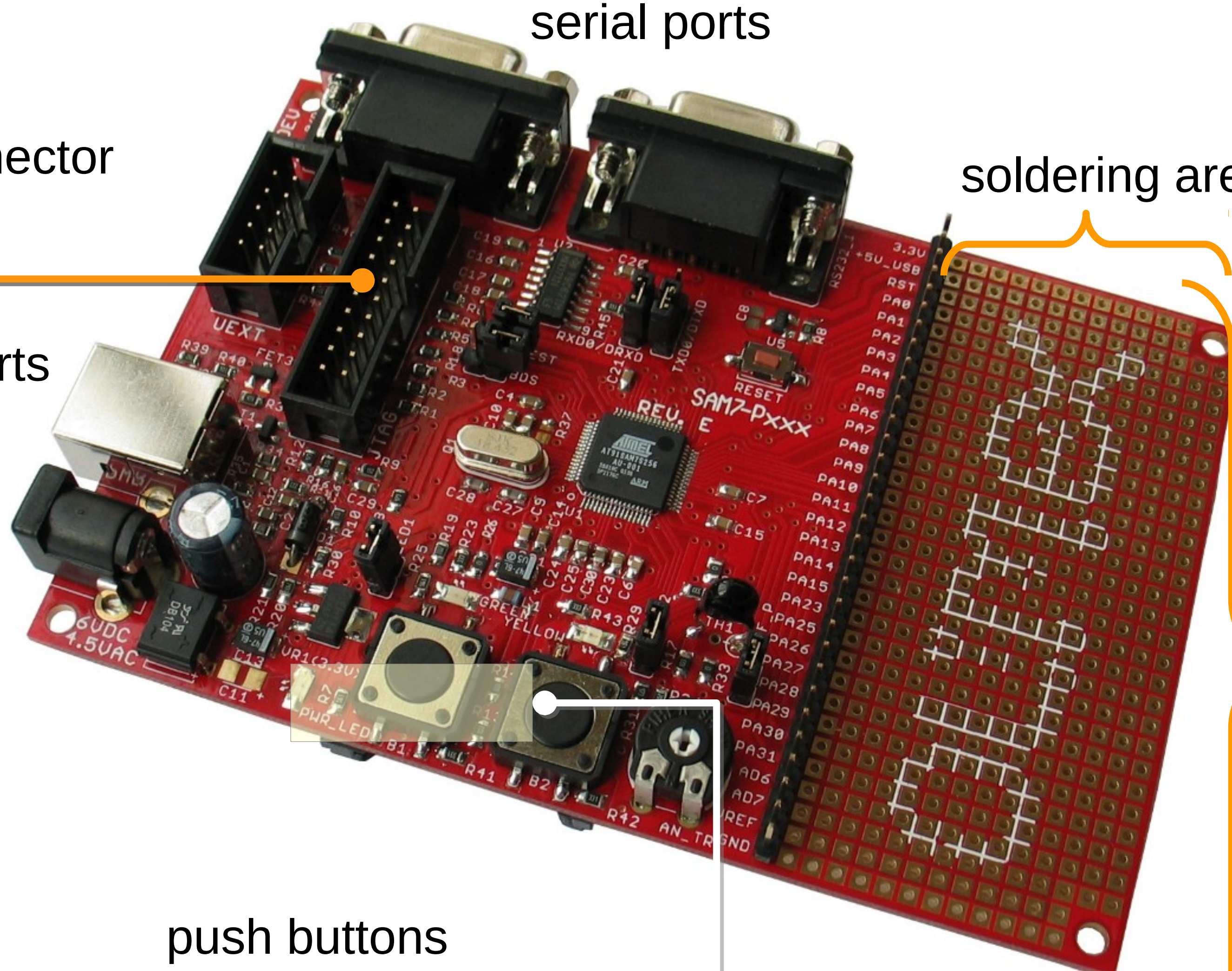


AMBA Bus

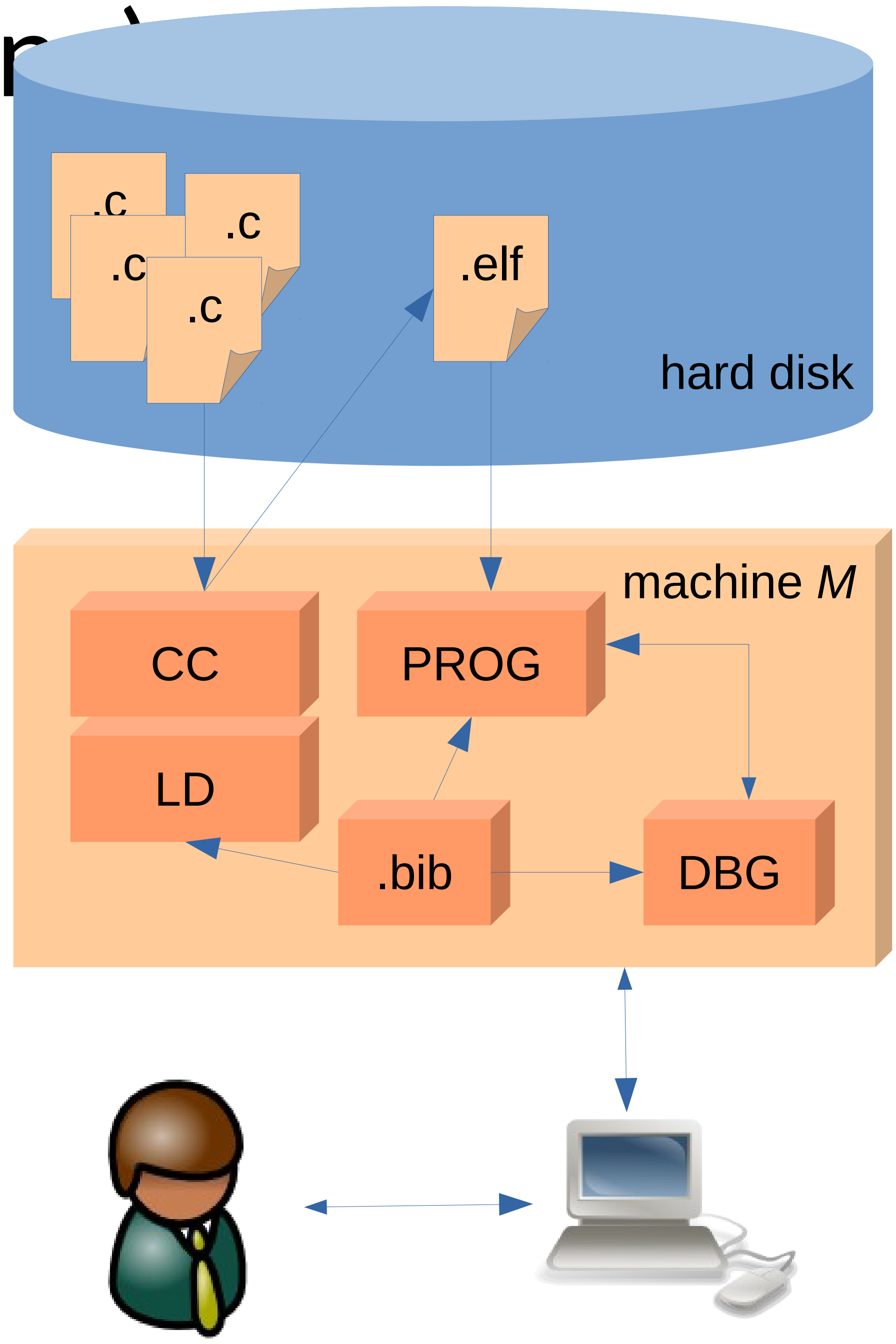
- Amba System Bus
  - masters: core + DMA
  - slaves: flash, RAM
- Amba Peripheral Bus
  - 1 master: APB bridge
  - slave: peripherals



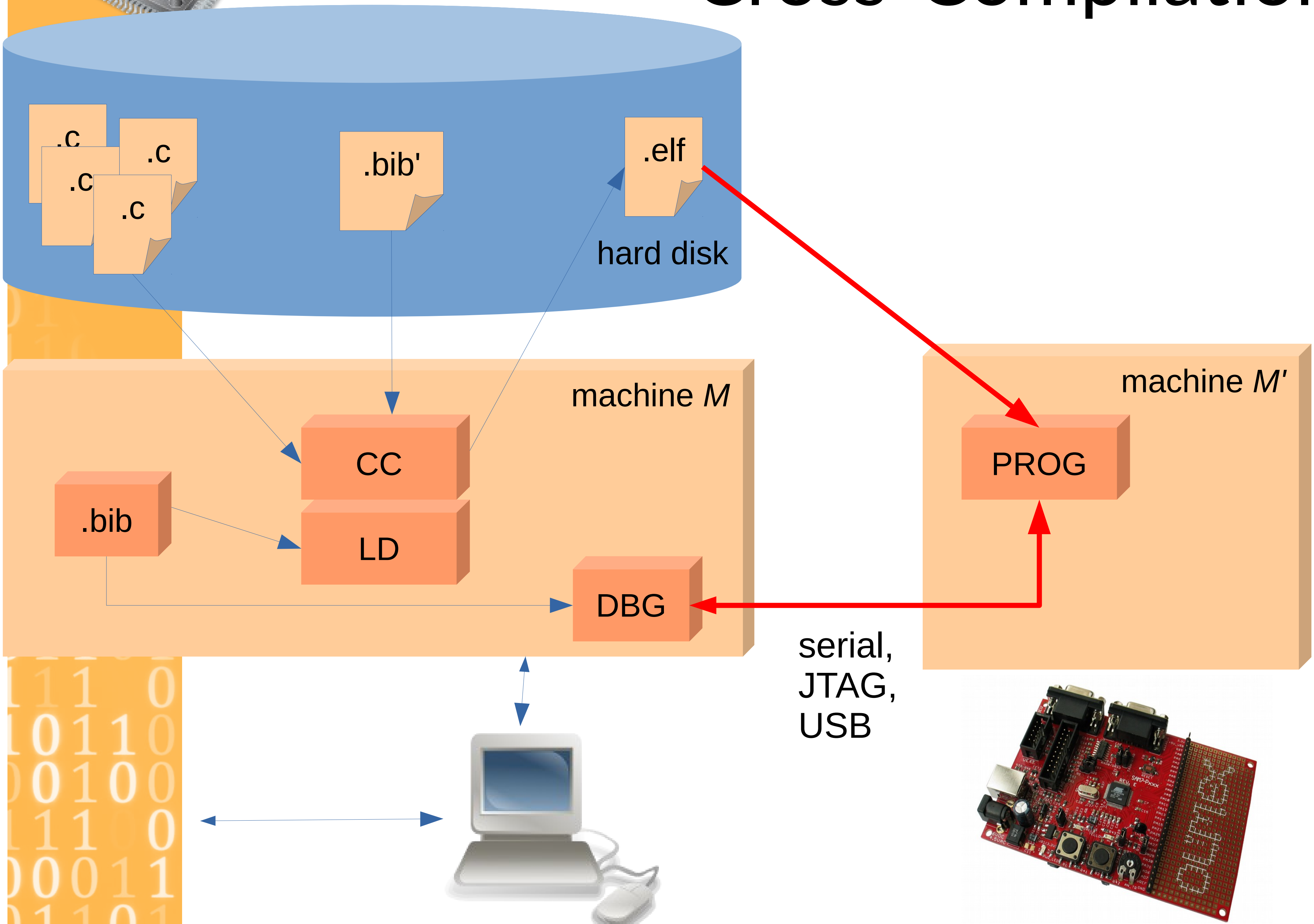
AT91SAM7S on an Olimex Board



Compilation on a Desktop (host machine)

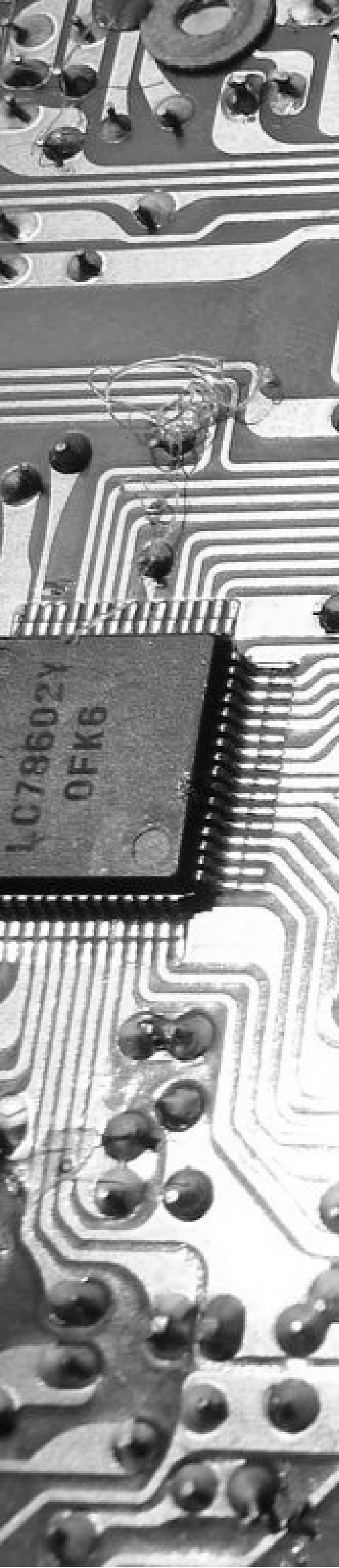


Cross-Compilation



Operating System

- bare metal (no OS)
  - too expensive
    - memory space
    - computation power
    - money
  - reliability → complete control of the hardware
  - no need for usual facilities of the OS: no standard hardware
- thin layer of OS / library
  - memory management libraries
  - network stacks
  - application organization: scheduler, cooperative tasks, pre-emptive tasks (thread)

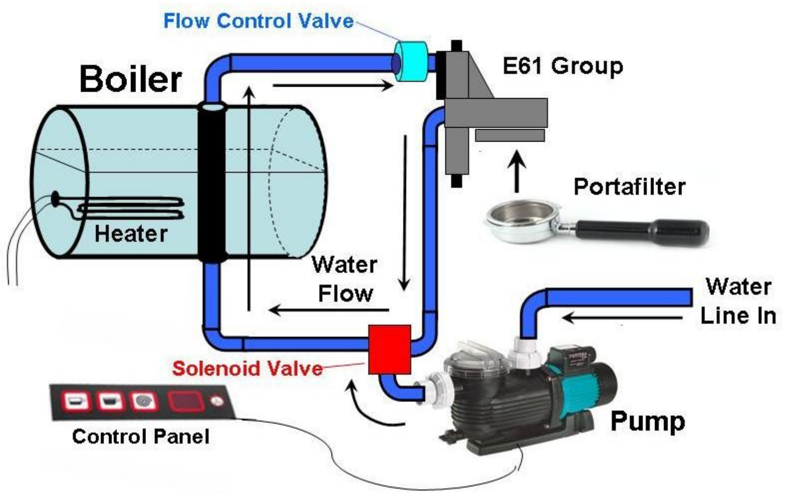


Overview

- Introduction
- The microcontroler
- **I/O Management**
- Using interrupts
- Sensors & Actuators

Peripherals

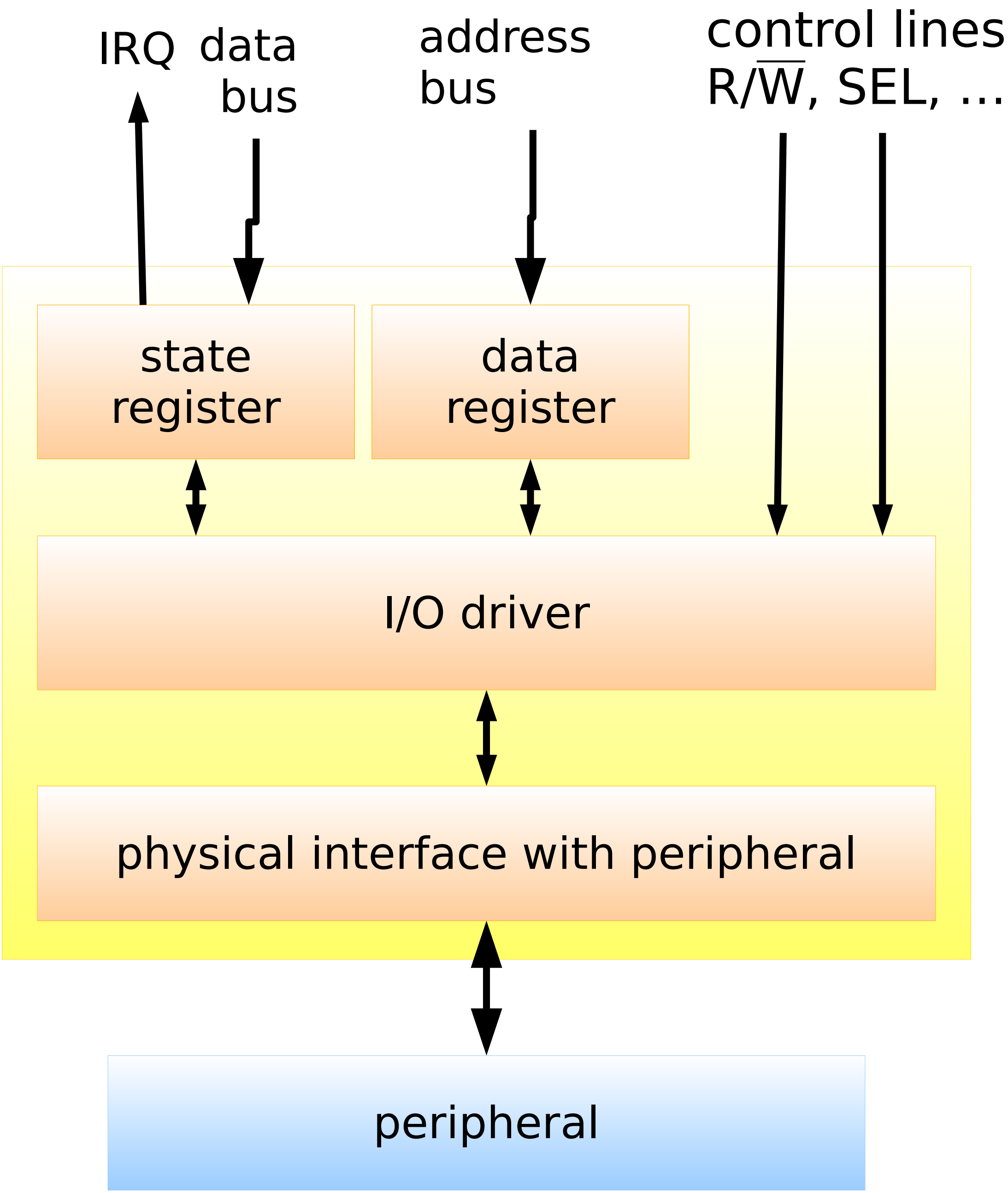
- definition
  - “an ancillary device used to put information into and get information out of the computer”, Laplante, Philip A. (Dec 21, 2000). Dictionary of Computer Science, Engineering and Technology. CRC Press
- examples
  - input – push button, sensors (thermistor), USB plug, etc
  - output – LED, motor, servo-motor,
- access path for the processor
  - through the bus (System Bus – Bridge – Peripheral Bus)
  - through a peripheral controller



input peripheral
- push buttons – short/long coffee, start/stop
- thermistor for boiler
- pressure gauge

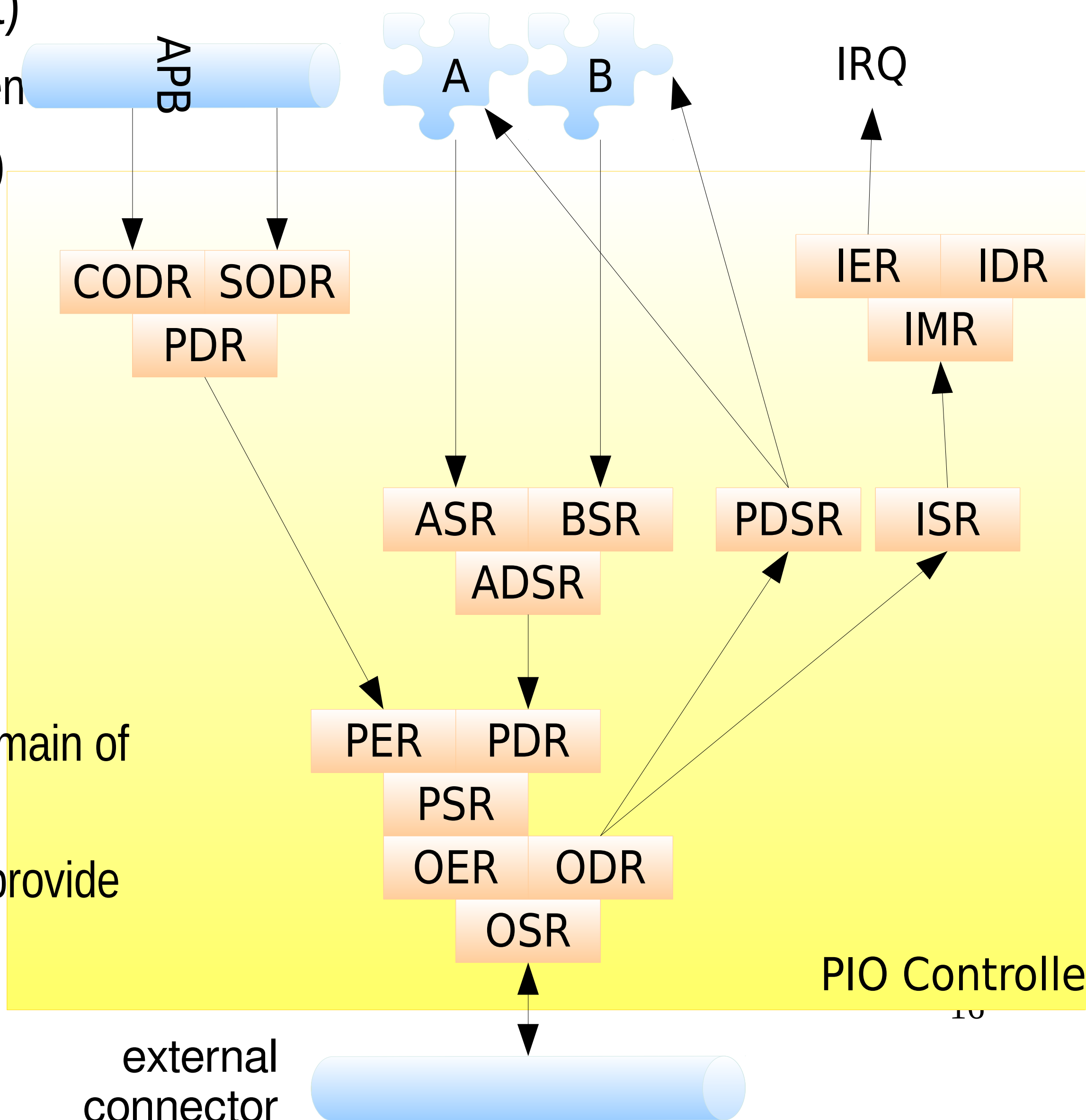
output peripheral
- indication LEDs (ready, working, on/off)
- flow control valve
- solenoid valve
- pump
- heater for boiler

Peripheral Controller

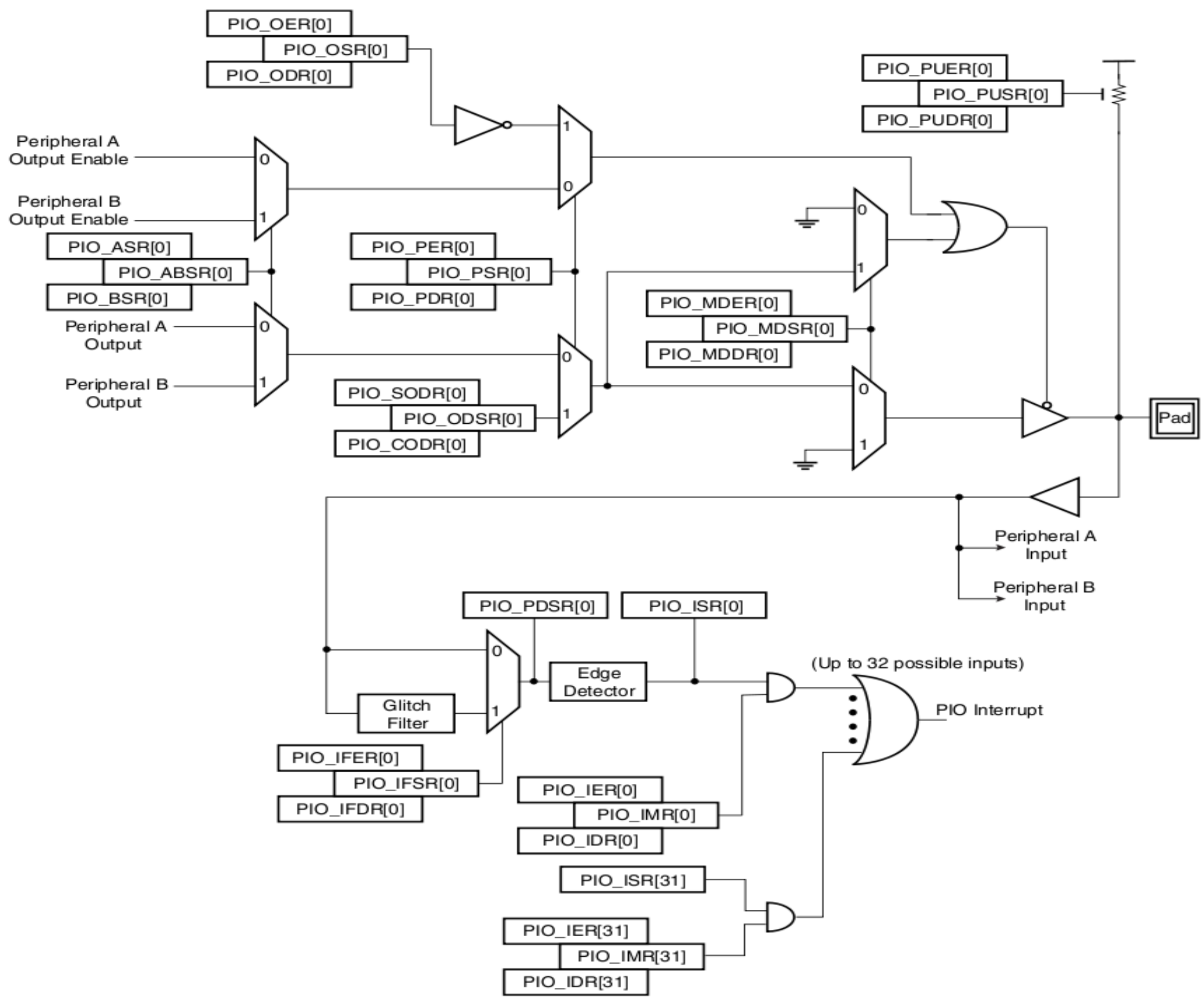


PIO of AT91SAMS

- PIO (Parallel Input Output)
  - 32 bidirectional independen
  - 2-state connection (0 / 5V)
- multiplexed
  - processor direct
  - périphérique A
  - périphérique B
- why multiplexing?
  - not enough pins
  - enlarge the application domain of the microprocessor
  - does my micro-controller provide enough I/O pins?



From manual



PIO_OER[0]
PIO_OSR[0]
PIO_ODR[0]
PIO_PUER[0]
PIO_PUSR[0]
PIO_PUDR[0]
Peripheral A Output Enable
Peripheral B Output Enable
PIO_ASR[0]
PIO_ABSR[0]
PIO_BSR[0]
PIO_PER[0]
PIO_PSR[0]
PIO_PDR[0]
PIO_MDER[0]
PIO_MDSR[0]
PIO_MDDR[0]
Peripheral A Output
Peripheral B Output
PIO_SODR[0]
PIO_ODSR[0]
PIO_CODR[0]
Pad
Peripheral A Input
Peripheral B Input
PIO_PDSR[0]
PIO_ISR[0]
Edge Detector
Glitch Filter
(Up to 32 possible inputs)
PIO Interrupt
PIO_IFER[0]
PIO_IFSR[0]
PIO_IFDR[0]
PIO_IER[0]
PIO_IMR[0]
PIO_IDR[0]
PIO_ISR[31]
PIO_IER[31]
PIO_IMR[31]
PIO_IDR[31]

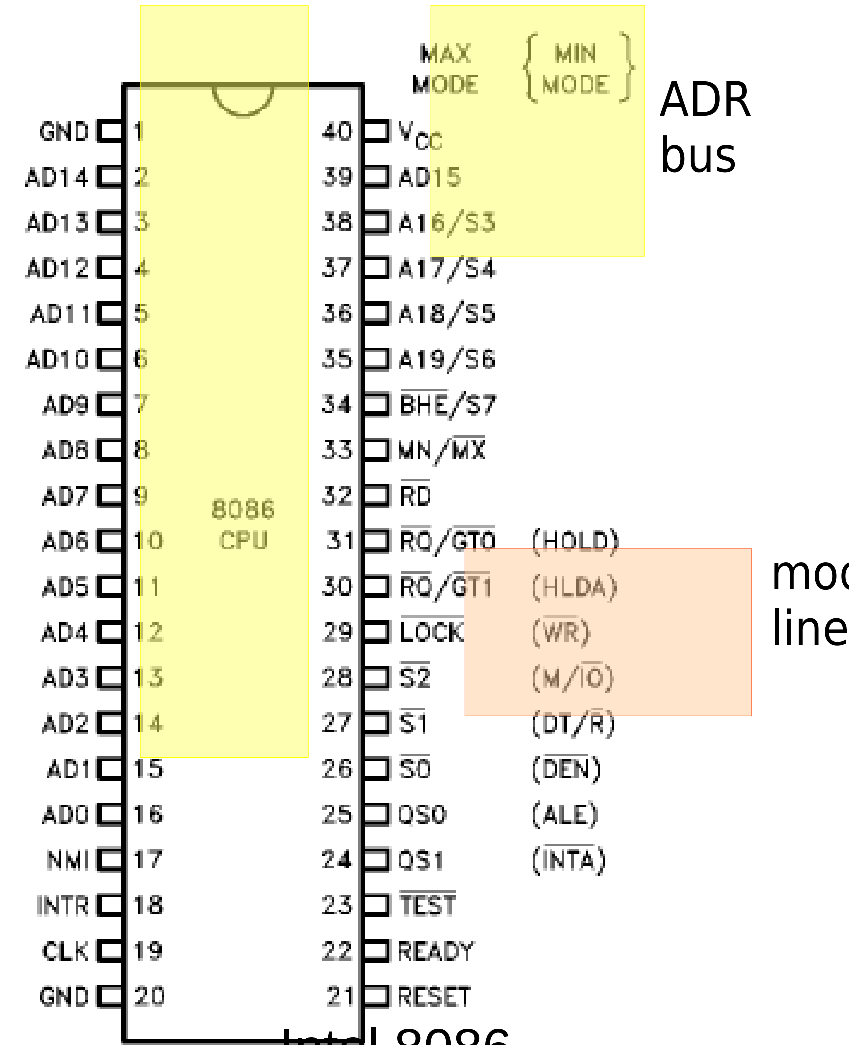
How to access hardware I/O registers?

- solution 1: special machine instruction
  - input machine register, I/O register
  - output I/O register, machine register
  - I/O register = number
  - I/O special bus or ADR/DATA bus with a special line
  - less and less used → lack of flexibility
- solution 2: mapped in memory
  - allocation d'une partie de l'espace d'adressage
  - nécessite d'avoir un gros espace d'adressage
  - processeur dédié aux entrées-sortie (allocation dans les adresses basses pour un accès facilité)
  - ajout d'un décodeur du bus d'adresse dans le contrôleur

How to access hardware I/O registers?

- solution 1: special machine instruction
  - `input` *machine register*, *I/O register*
  - `output` *I/O register*, *machine register*
  - I/O register = number
  - I/O separated bus or ADR/DATA bus with a special line
  - less and less used
    → lack of flexibility

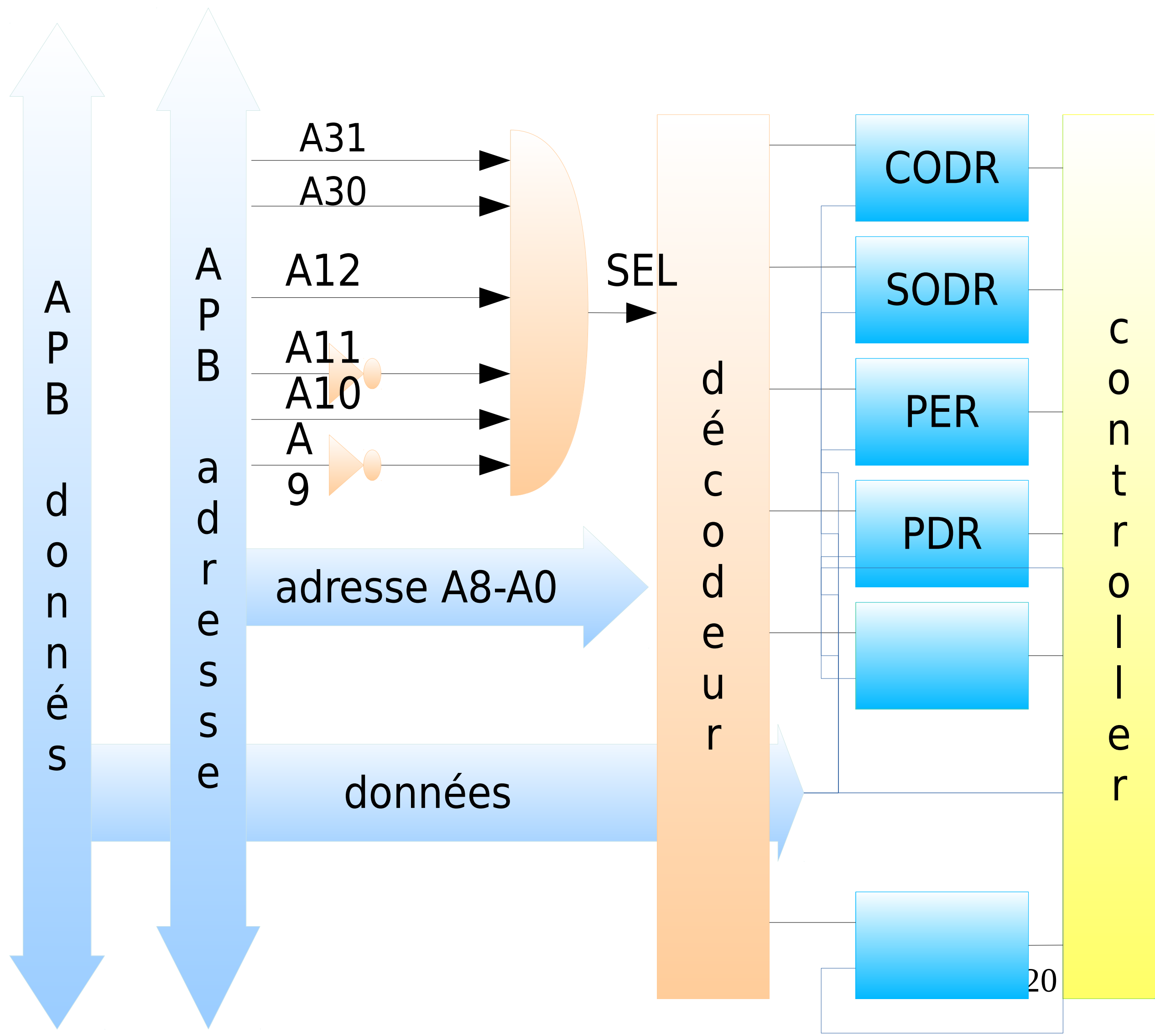
S2	S1	S0	
0	0	0	IT ACK
0	0	1	Read IO
0	1	0	Write IO
0	1	1	Halt
1	0	0	Read code
1	0	1	Read MEM
1	1	0	Write MEM
1	1	1	



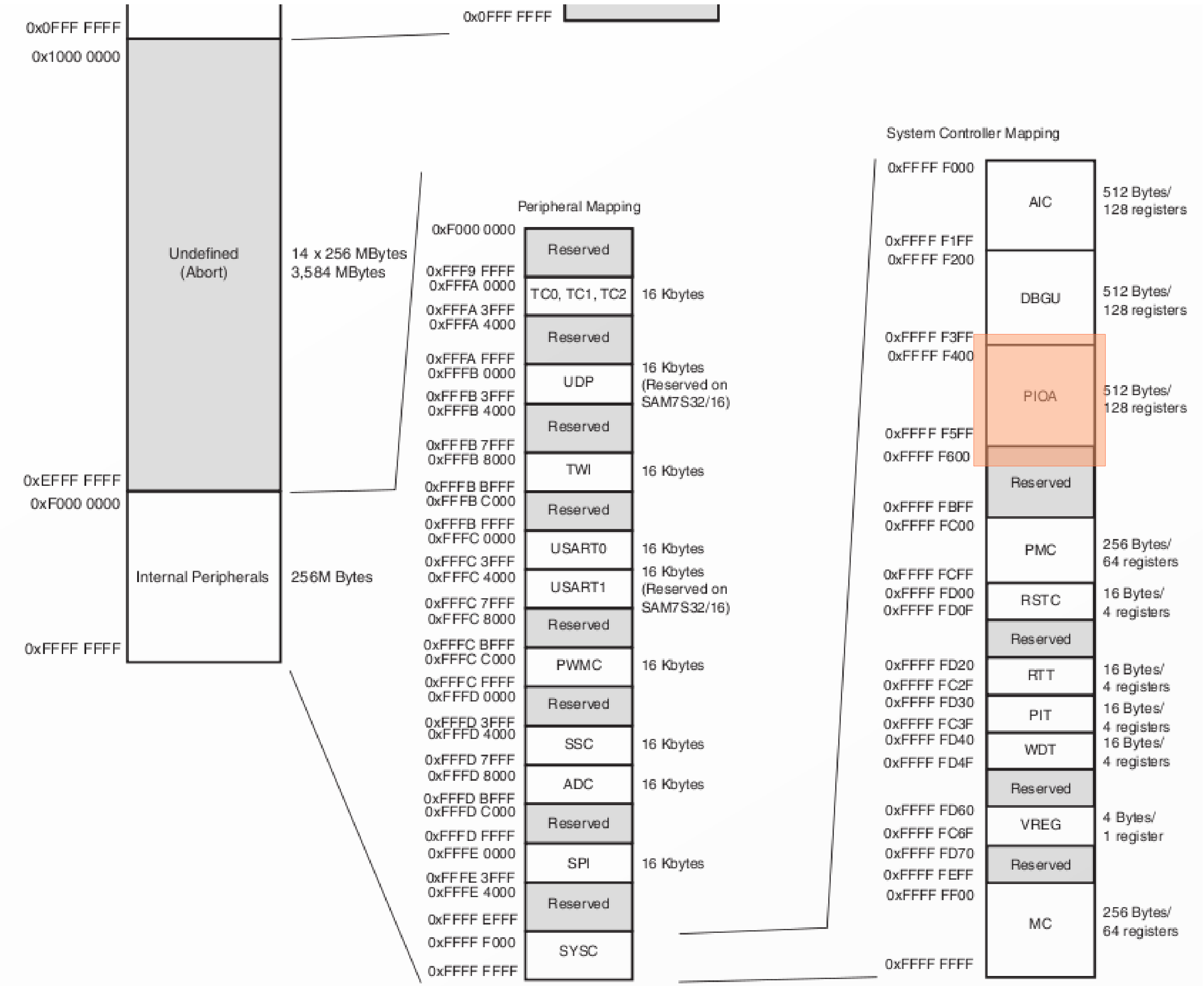
Intel 8086

How to access hardware registers?

- solution 2 : mapped in memory
  - peripheral = memory
  - dedicated address space
  - requires a big address space
  - I/O access by simple load/store instructions



AT91SAMS Memory Map



PIO I/O Registers

- base offset
  0xFFFF F400
  - depends on the micro-controller
  - independent of the core
- for each I/O register
  - offset relative to base address
  - functionality description
  - size (in bits/bytes)
  - type – read/write, read-only, write-only
  - from
    « PIO A – User Interface »
    AT91SAM ARM-based Flash MCU, Atmel

Offset	Register	Name	Access	Reset
0x0000	PIO Enable Register	PIO_PER	Write-only	–
0x0004	PIO Disable Register	PIO_PDR	Write-only	–
0x0008	PIO Status Register	PIO_PSR	Read-only	(1)
0x000C	Reserved			
0x0010	Output Enable Register	PIO_OER	Write-only	–
0x0014	Output Disable Register	PIO_ODR	Write-only	–
0x0018	Output Status Register	PIO_OSR	Read-only	0x0000 0000
0x001C	Reserved			
0x0020	Glitch Input Filter Enable Register	PIO_IFER	Write-only	–
0x0024	Glitch Input Filter Disable Register	PIO_IFDR	Write-only	–
0x0028	Glitch Input Filter Status Register	PIO_IFSR	Read-only	0x0000 0000
0x002C	Reserved			
0x0030	Set Output Data Register	PIO_SODR	Write-only	–
0x0034	Clear Output Data Register	PIO_CODR	Write-only	
0x0038	Output Data Status Register	PIO_ODSR	Read-only or[2] Read-write	–
0x003C	Pin Data Status Register	PIO_PDSR	Read-only	(3)
0x0040	Interrupt Enable Register	PIO_IER	Write-only	–
0x0044	Interrupt Disable Register	PIO_IDR	Write-only	–
0x0048	Interrupt Mask Register	PIO_IMR	Read-only	0x00000000
0x004C	Interrupt Status Register[4]	PIO_ISR	Read-only	0x00000000
0x0050	Multi-driver Enable Register	PIO_MDER	Write-only	–
0x0054	Multi-driver Disable Register	PIO_MDDR	Write-only	–
0x0058	Multi-driver Status Register	PIO_MDSR	Read-only	0x00000000
0x005C	Reserved			
0x0060	Pull-up Disable Register	PIO_PUDR	Write-only	–
0x0064	Pull-up Enable Register	PIO_PUER	Write-only	–
0x0068	Pad Pull-up Status Register	PIO_PUSR	Read-only	0x00000000
0x006C	Reserved			

How to program them?

- In C! C has been designed for this!
- setting an I/O register = writing to the right memory
- how to get the right address?

```
#include <stdint.h>
#define PIO_BASE   0xfffff400
#define PIO_SODR   *(uint32_t *)(PIO_BASE + 0x30)
```

- how to avoid unexpected optimization from the compiler?
  - this memory is not volatile!

```
#define PIO_SODR   *(volatile uint32_t *)(PIO_BASE
+ 0x30)
```

- set of definitions PIO_xxx
  - API for peripheral PIO
  - useful to put them in a header file, `PIO.h`

PIO Driving

- two phases
  - initialize the controller/peripheral
  - use it
- PIO specials
  - each register drive the set of pins: 1 pin / 1 bit
  - 0 = non-used, 1 = action
  - registers Cxx, xDx ⇒ clear/disable
  - registers Sxx, xEx ⇒ set/enable

Programming in C (reminder)

mask building

- 1 << *i* = set to 1 the bit *i*
- ~(1 << *i*) = all bits to 1 except bit *i*
- (1 << *i*) | (1 << *j*) = bit *i* and *j* to 1

bit changing

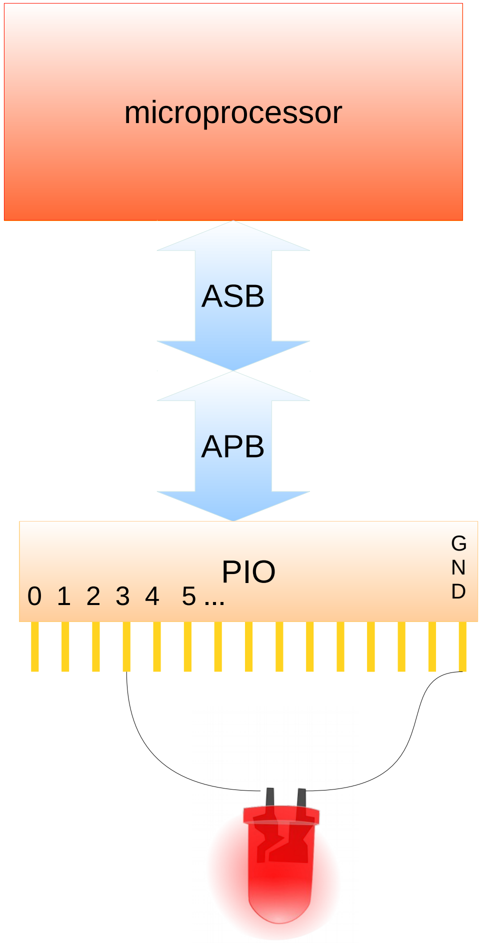
- *v* | (1<< *i*) : *v* with bit *i* to 1
- *v* & ~(1 << *i*) : *v* with bit *i* to 0

bit test

- v & (1 << i) : true if bit i of v is 1
- !(v & (1 << i)) : true if bit i of v is 0

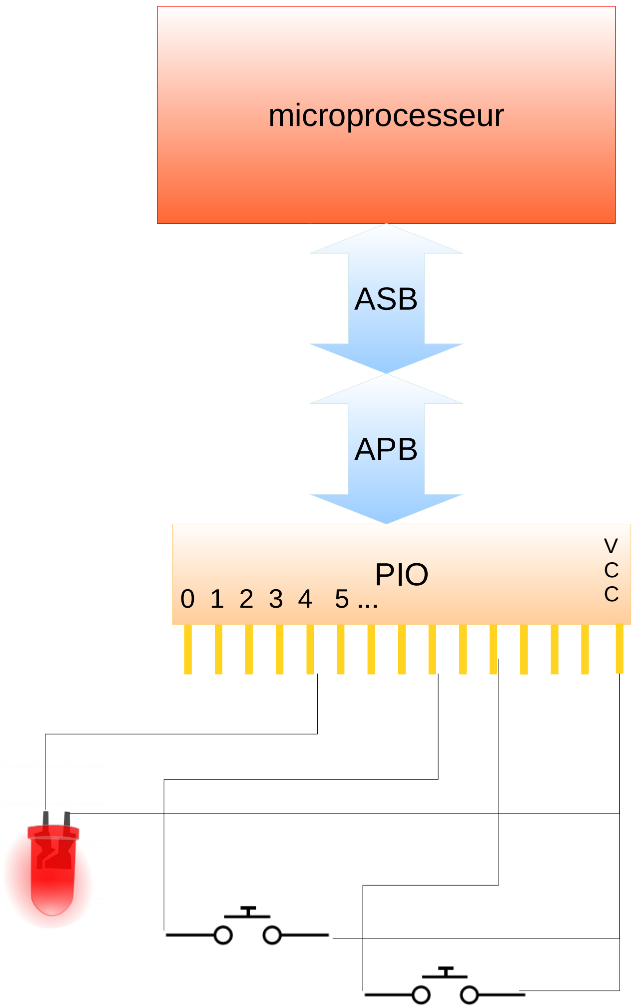
Example: blinking LED

```
#define LED1            (1 << 3)
#define PIO_BASE        0xffff f400
#define PIO_SODR \
    *(volatile uint32_t *)(PIO_BASE + 0x30)
    ...
int main(void) {
    int i, s = 0;

    /* initialization */
    PIO_PER = LED1;    /* controlled by core */
    PIO_OER = LED1;    /* configured output */

    /* main endless loop */
    while(1) {
        if(s)
            PIO_CODR = LED1;   /* switch on */
        else
            PIO_SODR = LED1;   /* switch off */
        s =!s ;
        /* waiting loop */
        for(i = 0 ; i < 100000 ; i++) ;
    }
}
```



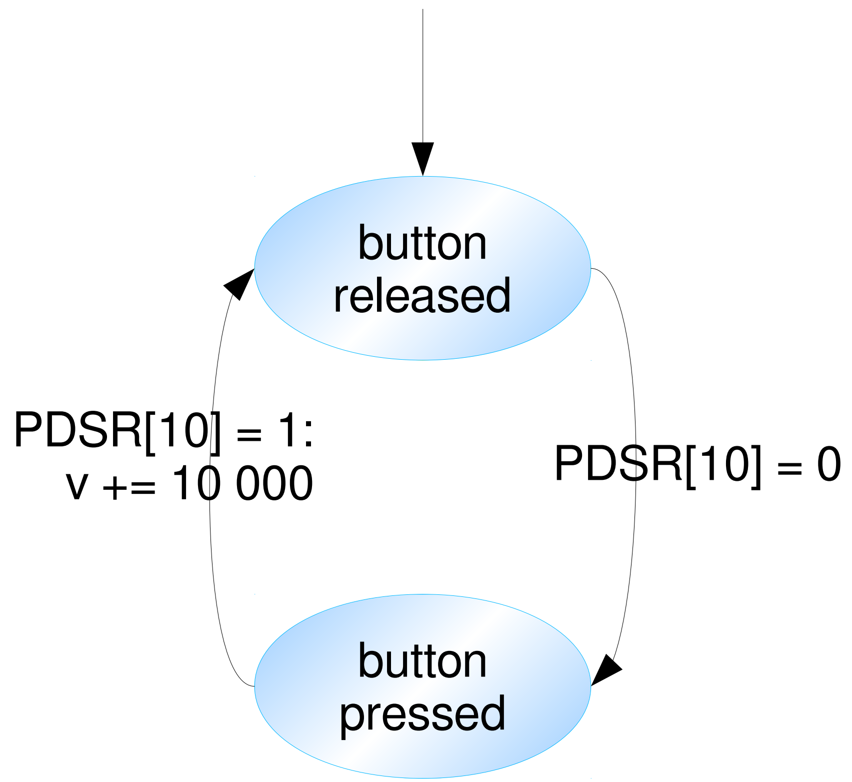
Improvement: speed up/down with push button

```c
#define PUSH1      (1 << 10)
#define PUSH2      (1 << 11)
    ...

int main(void) {
    int i, s = 0, v = 100000 ;
    int p1 = 0, p2 = 0;

    /* initialization */
    PIO_PER = LED1 | PUSH1 | PUSH2;
    PIO_OER = LED1;
    PIO_ODR = PUSH1 | PUSH2 ;

    /* main loop */
    while(1) {

        /* LED management */
        if(s) PIO_CODR = LED1 ;
        else PIO_SODR = LED1 ;
        s =!s ;
```



Improvement: speed up/down with push button

```
        /* waiting loop */
        for(i = 0 ; i < v; i++) ;

        /* manage button 1 */
        if(p1 && (PIO_PDSR & PUSH1)) {
            p1 = 0 ;
            v += 10000 ;
        }
        else if(!p1 && !(PIO_PDSR & PUSH1))
            p1 = 1 ;

        /* manage button 2 */
        if(p2 && (PIO_PDSR & PUSH2)) {
            p2 = 0 ;
            v -= 10000 ;
        }
        else if(!p2 && !(PIO_PDSR & PUSH2))
            p2 = 1 ;
    }
}
```



p1 = automaton state
signal = PDSR bits
state change =
modification of p1

Beware of Optimizing Compiler

```
uint32_t *PDSR = (uint32_t *)0xFFFFF400 ;

/* waiting loop */
while(*PDSR & (1 << 3))
    ;
```

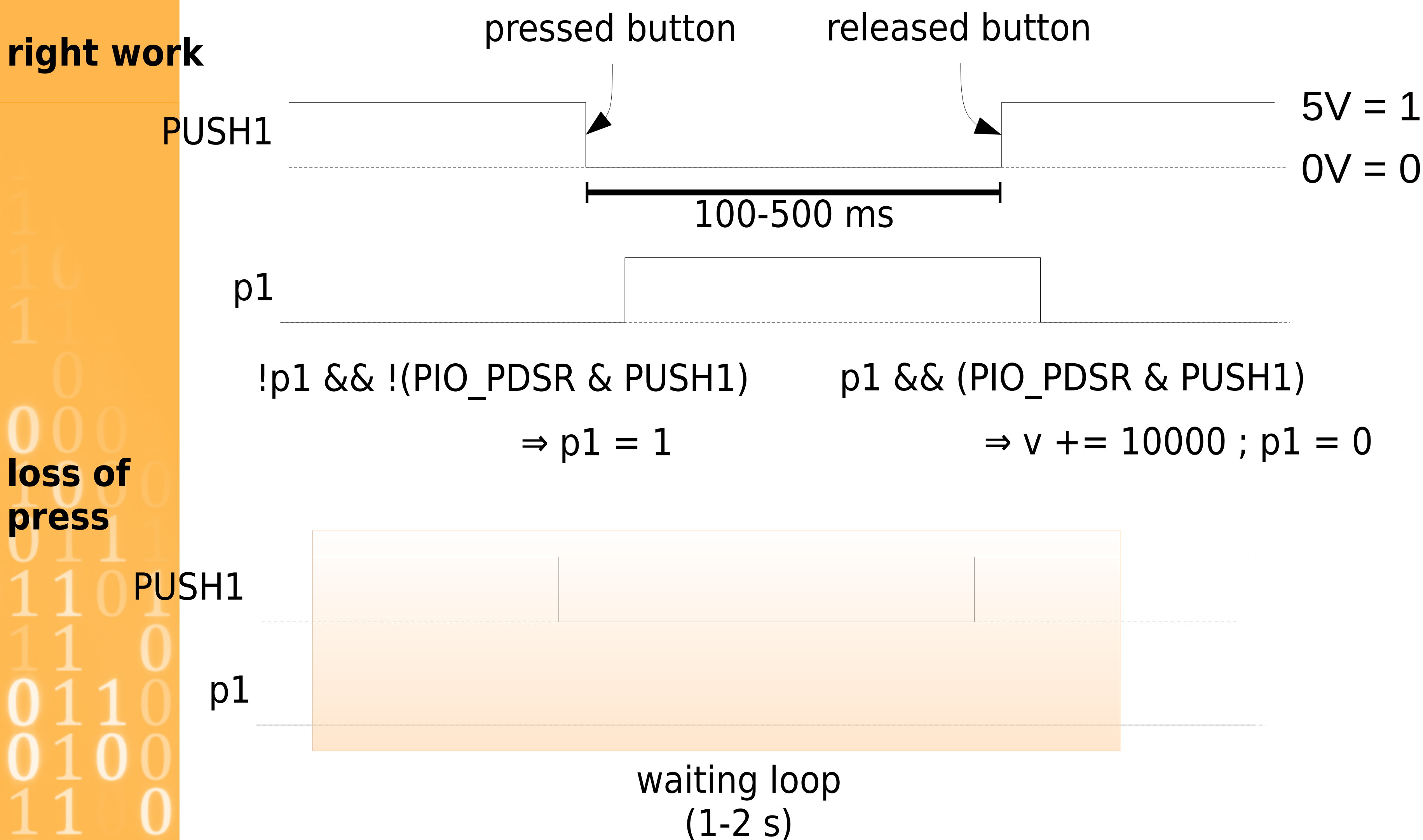
from the compiler point of view, the condition is condition is constant!

optimization

```
uint32_t *PDSR = (uint32_t *)0xffffF400 ;

/* waiting loop */
uint32_t tmp = *PDSR & (1 << 3) ;
while(tmp)
    ;
```

`volatile` is required here!

Loss of Button Press

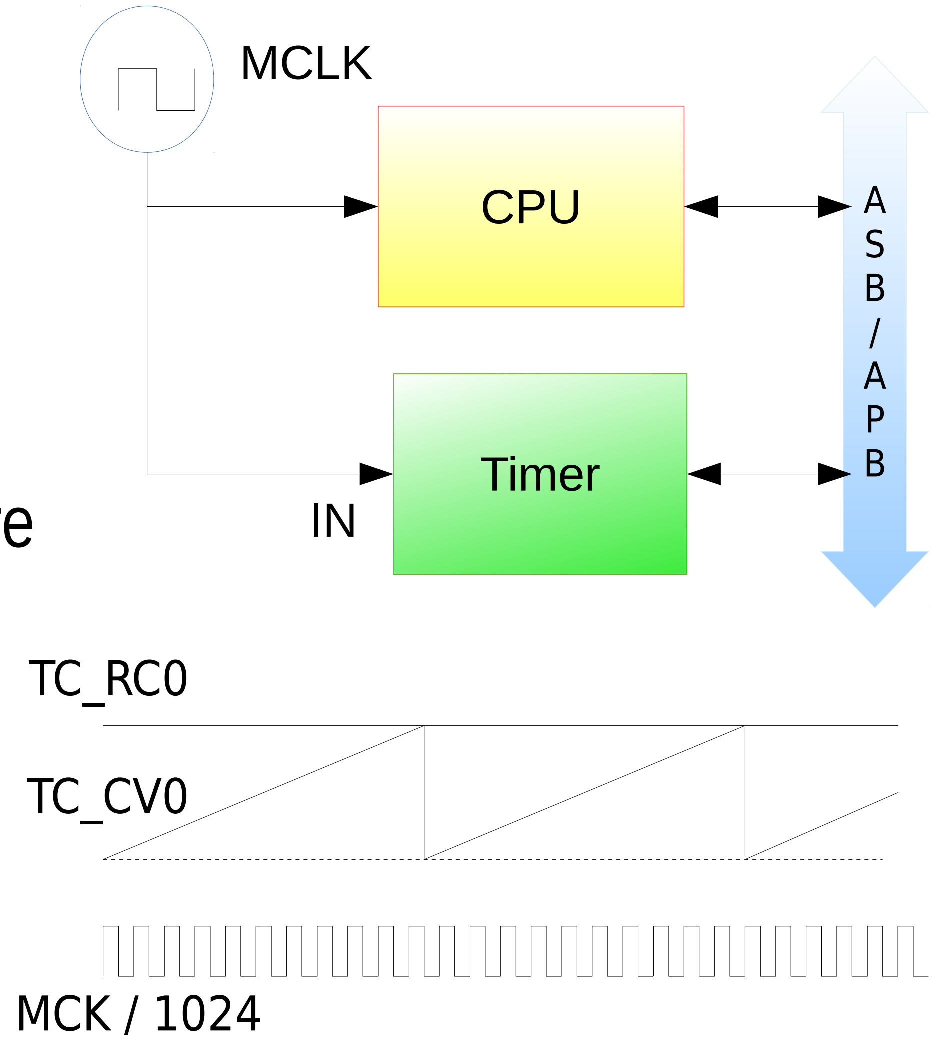


Improvement

```c
#define PUSH1      (1 << 10)
#define PUSH2      (1 << 11)
    ...

int main(void) {
    int i, s = 0, v = 100000 ;
    int p1 = 0, p2 = 0;

    /* initialization */
    PIO_PER = LED1 | PUSH1 | PUSH2;
    PIO_OER = LED1;
    PIO_ODR = PUSH1 | PUSH2 ;

    /* main loop */
    while(1) {

        /* LED management */
        if(s) PIO_CODR = LED1 ;
        else PIO_SODR = LED1 ;
        s =!s ;

        /* waiting loop */
        for(i = 0 ; i < v; i++) {

            /* manage button 1 */
            if(p1 && (PIO_PDSR & PUSH1)) {
                p1 = 0 ;
                v += 10000 ;
            }
            else if(!p1 && !(PIO_PDSR & PUSH1)
                p1 = 1 ;

            /* manage button 2 */
            if(p2 && (PIO_PDSR & PUSH2)) {
                p2 = 0 ;
                v -= 10000 ;
            }
            else if(!p2 && !(PIO_PDSR & PUSH2)
                p2 = 1 ;
        }
    }
}
```

New Problem: the delay waiting is dependent on the code!

- timer
  - counter (16-bits on AT91SAM7S)
  - incremented at a rate depending on MCK
  - independent of the core
  - MCK – Master Clock ~48 MHZ
  - divided by to count longer delays



Using a Timer

- CCR – command
  - CLKDIS – disable bit
  - CLKEN – enable bit
  - SWTRG – reset bit
- CMR – configure
  - CLOCK1-5 – divider 2, 8, 32, 128, 1024
  - CPCTRG – reset when RC = CV
- RC – register
- SR – status register
  - CPCS – bit set when RC = CV

```
/* initialisation */
TC0_CCR = TC_CLKDIS;   /* disabled */
TC0_CMR = TC_CLOCK5
           /* divided by 1024 */
       |  TC_CPCTRG;
           /* retsart when CV = RC */
TC0_RC = 46875;        /* delay = 1s */
TC0_CCR = TC_CLKEN;    /* enabled */
TC0_CCR = TC_SWTRG;    /* reset */
```

48 MHz ⇔ 1 s
48,000,000 Hz ⇔ 1 s

/ 1024

46,875 Hz ⇔ 1 s

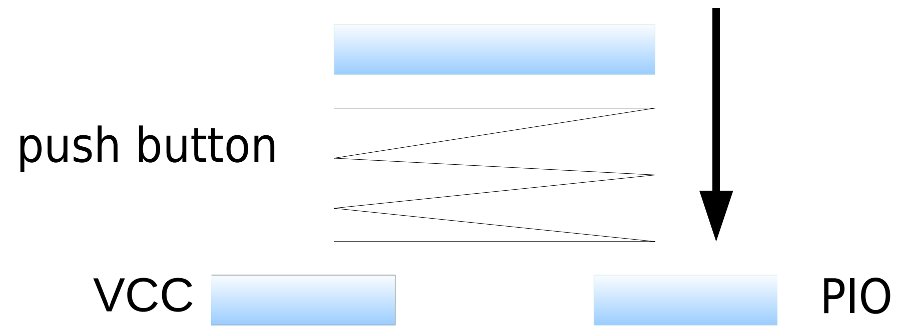
```
/* waiting the end of count */
while(!(TC0_SR  & TC_CPCS)) ;
```

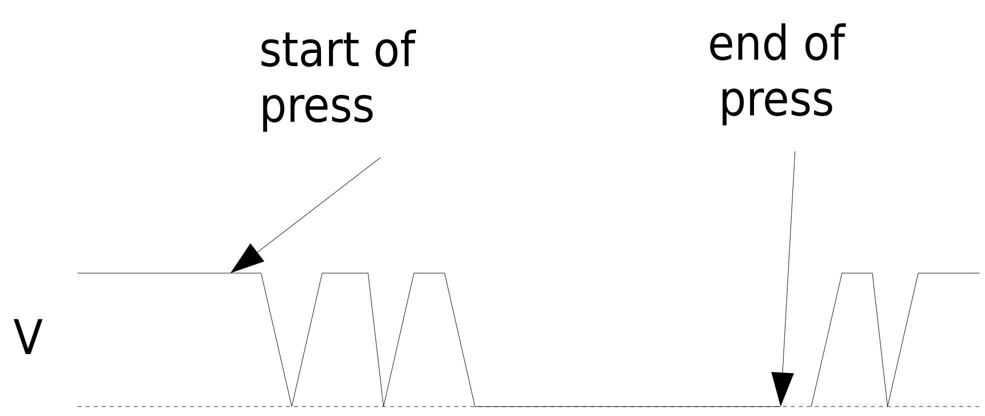
Improvement: Blinking with Timer

```c
    ...

int main(void) {
    int i, s = 0;
    int p1 = 0, p2 = 0;

    /* initialization */
    PIO_PER = LED1 | PUSH1 | PUSH2;
    PIO_OER = LED1;
    PIO_ODR = PUSH1 | PUSH2;

    /* timer initialization */
    TC0_CCR = TC_CLKDIS;
    TC0_CMR = TIMER_CLOCK5 | SWTRG;
    TC0_RC = 46875;
    TC0_CCR = TC_CLKEN;
    TC0_CCR = TC_SWTRG;

    /* main loop */
    while(1) {
        /* LED management */
        if(TC0_SR  & TC_CPCS) {
            if(s) PIO_CODR = LED1 ;
            else PIO_SODR = LED1 ;
            s =!s ;
        }

        /* button 1 management */
        if(p1 && (PIO_PDSR & PUSH1))
            { p1 = 0 ; TC0_RC += 1000; }
        else if(!p1 && !(PIO_PDSR & PUSH1))
            p1 = 1 ;

        /* button 2 management */
        if(p2 && (PIO_PDSR & PUSH2))
            { p2 = 0 ; TC0_RC -= 1000; }
        else if(!p2 && !(PIO_PDSR & PUSH2))
            p2 = 1 ;
        }
}
```

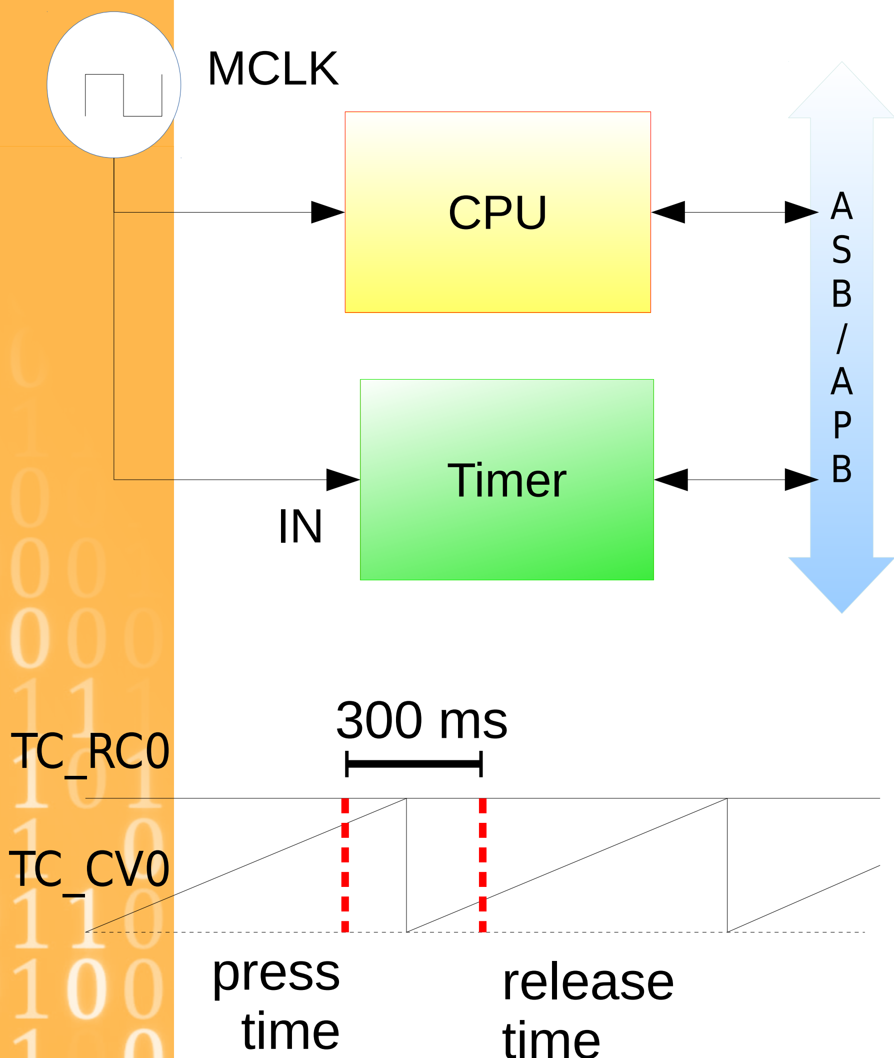
Problem: push button bounces

```c
/* 100ms = 1s / 10 = MCK / 1024 / 10 */
#define ONE_SECOND      46875
#define PUSH_TIME       ONE_SECOND / 10
    ...
int time_press, time_release;
    ...
/* button 1 management */
if(p1 && (PIO_PDSR & PUSH1)) {
    time_release = TC_CV0 ;


    if(time_release – time_press
    >= PUSH_TIME)
        { p1 = 0 ; TC_RC0 += 1000 ; }
}
else if(!p1 && !(PIO_PDSR & PUSH1)) {
        p1 = 1;
        time_press = TC_CV0;
    }
    ...
```





Problem: push button bounces



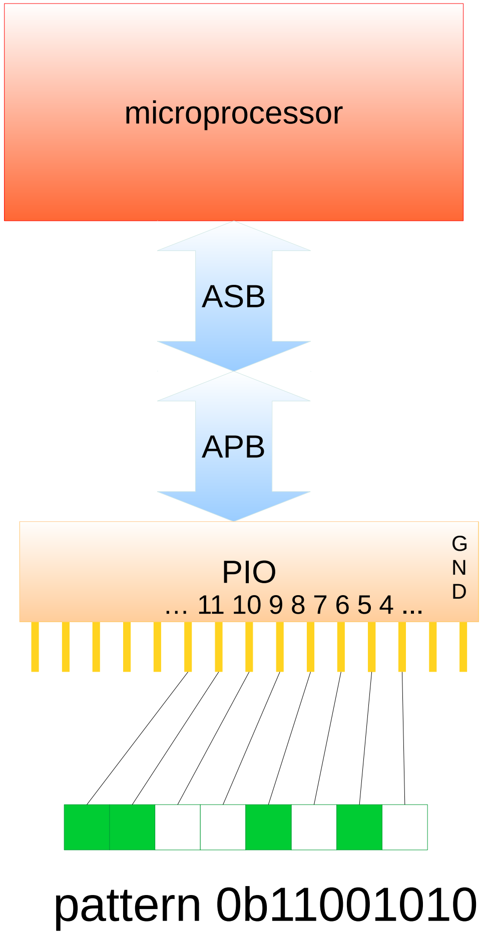

```
/* 100ms = 1s / 10 = MCK / 1024 / 10 */
#define ONE_SECOND      46875
#define PUSH_TIME       ONE_SECOND / 10
    ...
int time_press, time_release;
    ...
/* button 1 management */
if(p1 && (PIO_PDSR & PUSH1)) {
    time_release = TC_CV0 ;
    if(time_release < time_press)
        time_release += ONE_SECOND;
    if(time_release – time_press
    >= PUSH_TIME)
        { p1 = 0 ; TC_RC0 += 1000 ; }
}
else if(!p1 && !(PIO_PDSR & PUSH1)) {
        p1 = 1;
        time_press = TC_CV0;
    }
    ...
```

Exercise (1a)

PIO pins 4 to 11 (8 pins) are connected to LED to display a pattern 0b11001010 that we want to shift to achieve a visual effect (scheme on the right).

The current state of the pattern is stored in global variable pattern.

1) Initialize the PIO in main().
2) Write a function display() that displays the pattern on the PIO.



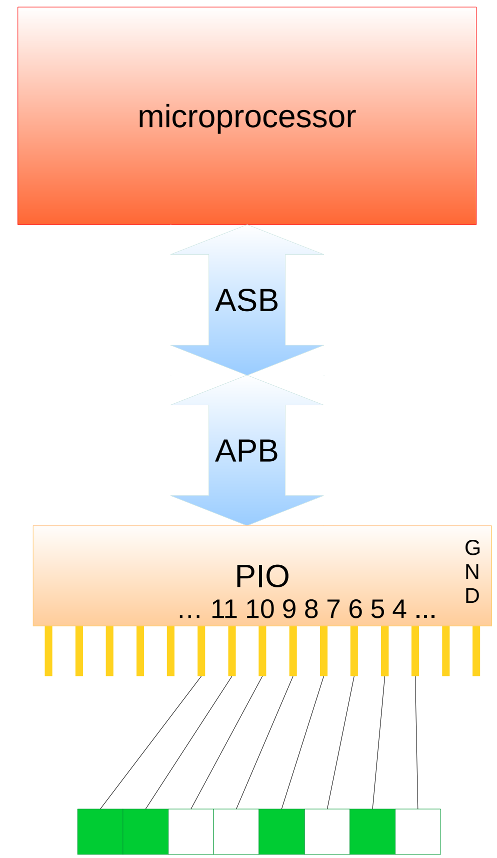

```
char pattern = 0b11001010;
void display(void) { … }
int main(void) { … }
```

Exercise (1b)

3) We consider that pin 20 and 21 are connected to push buttons.

- Pushing on 20 make the pattern to shift left.
- Pushing on 21 make the pattern to shift to right.

Implement the endless loop in main() that manages the push buttons.



pattern 0b11001010

```
char pattern = 0b11001010;
void display(void) { … }
int main(void) { … }
```

Exercise (2a)

The USART device is a universal serial port controller (like RS-232C for example). The port est bidirectional: this enables the connection of a terminal (keyboard + screen) and to perform text input/output. Serial connections are often used for debugging purpose.

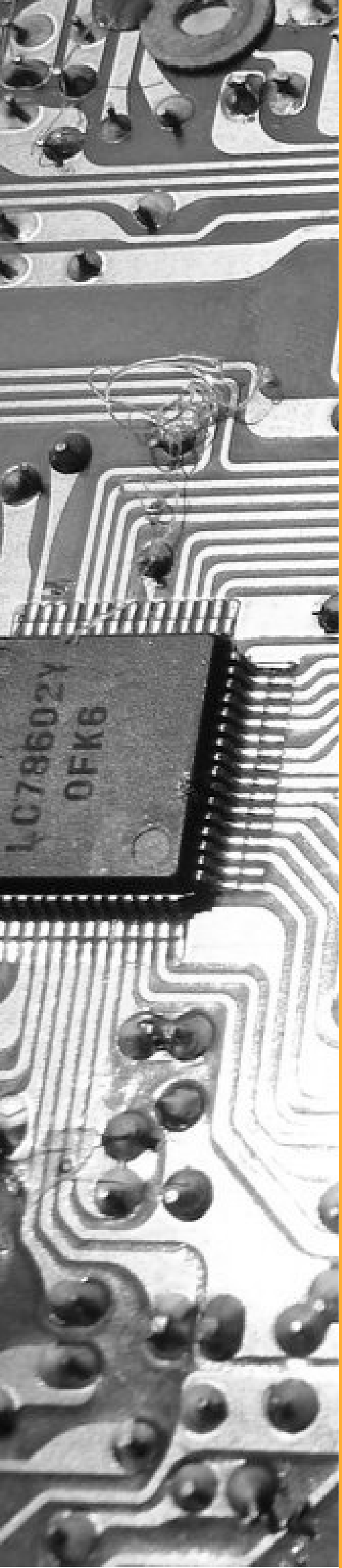
Here, we are not interested by the initialization of the USART that is relatively simple but long. We focus on the input/output functions of the device. To send a character, we have to wait for the previous character to be sent (**ARM_US_TXRDY** bit of **US0_CSR** is set) and to write the character to the register **US0_THR**. To receive a character, we have to wait for one to be available (**ARM_US_RXRDY** bit of register **US0_CSR**) and to get it from **US0_RHR**.

1) Write the functions **void usart_putc(char c)** et **void usart_puts(char *s)** that send, respectively, a character and a string of character to the serial port.

Exercise (2b)

2) Write the functions **char serial_getc(void)** and **void serial_gets(char *s)** that allow to receive, respectively, a character and a character string (ended by '\n') from the serial port.

3) Use the functions previously defined to read two integers from the serial port, to compute the sum and to display the result on the serial port. To convert the character string to integer, you can use the function **int atoi(char *s)**.



Overview

- Introduction
- The microcontroler
- I/O Management
- **Using interrupts**
- Sensors & Actuators

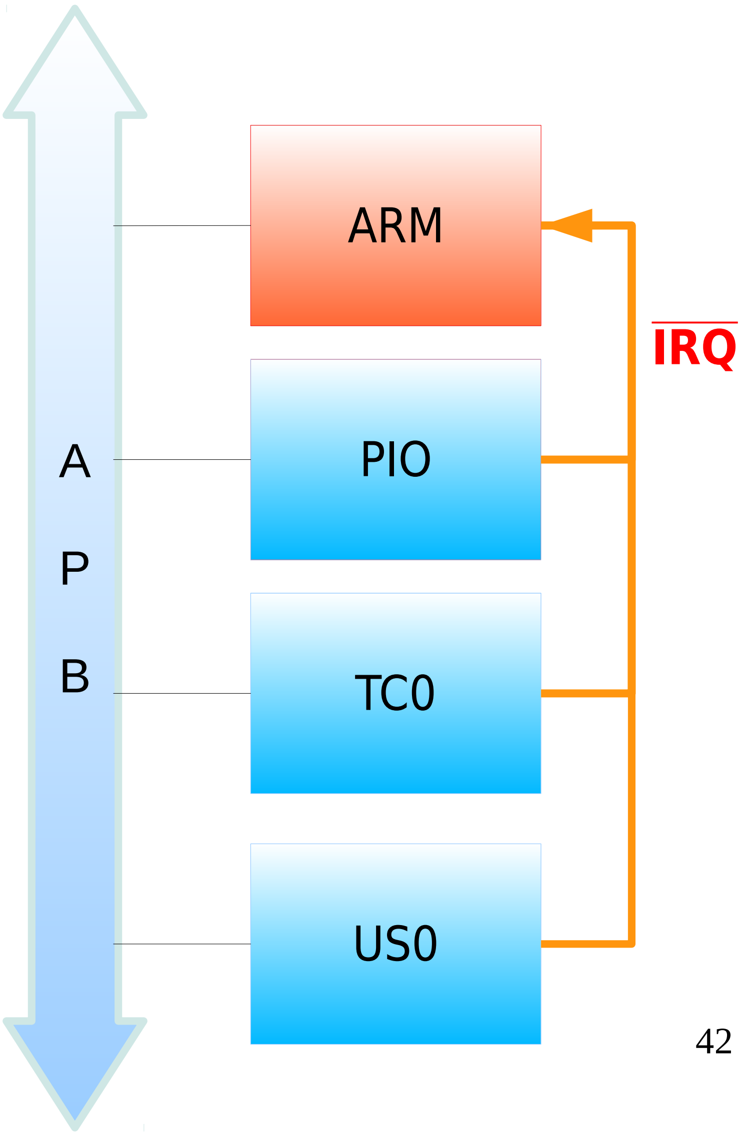
Polling Approach

```c
/* state variable declaration */

…

int main(void) {

    /* initialization */

    while(1) {

        if(event1)
            action1();

        if(event2)
            action2();

        if(event3)
            action3();

        …
    }
}
```

- pros
  - easy to implement
  - easy to understand
  - deterministic – order of actions is always the same – good behaviour for real-time
- cons
  - minimal reaction time = loop body duration – reducing frequency → increase this duration
  - active loop = energy consumption
  - main loop may be come very complex

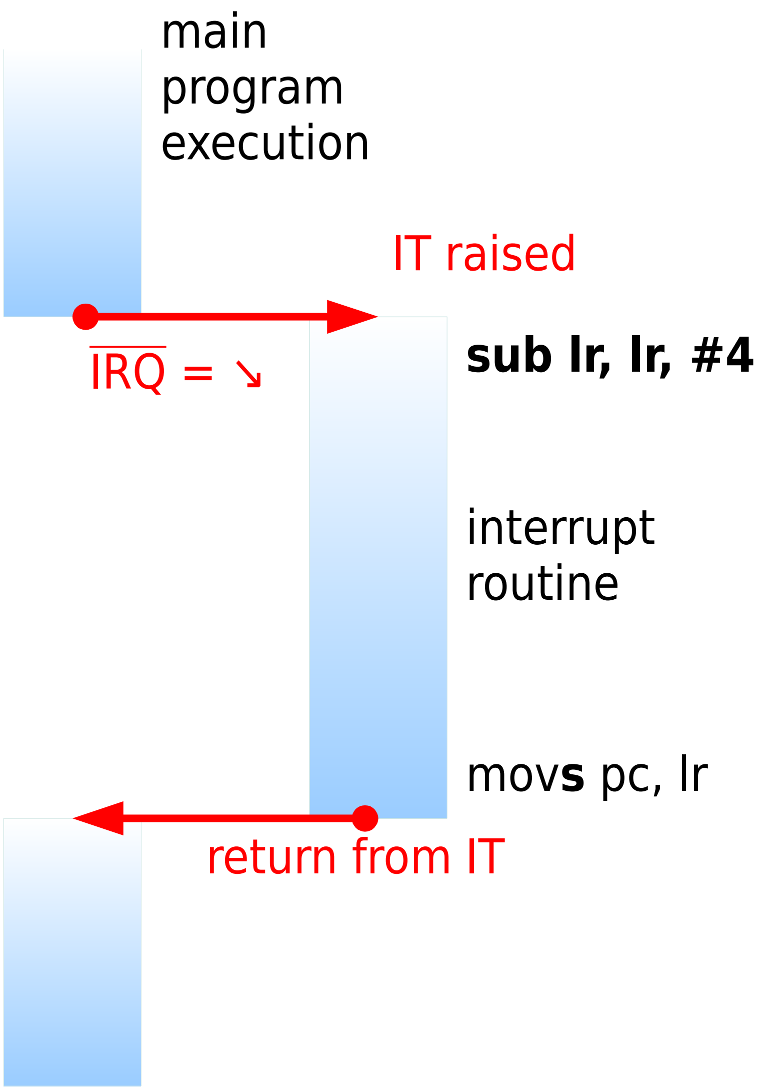
Interrupts (IT)

- principles
  - asynchronous management on input/output
  - on event
    - current program stopped
    - execution of an interrupt routine
  - interrupt transparent for the main program
- realisation of the interrupt
  - current context (registers) is saved
  - selection of an interrupt routine
  - interrupt routine is executed
  - at routine end, main program context is reloaded
  - main program goes on (possibly not noticing the interrupt)



On ARM

- 2 IT lines
  - IRQ – normal interrupt
  - FIQ – interrupt requiring fast processing
- sequence
  - processor mode is changed to IRQ / FIQ
  - sp, lr, sr are specific to the new mode
  - lr ← return address + 4 (pipeline)
  - spsr ← cpsr (state register saving)
  - pc, cpsr ← 0x18/0x1C, bit I/F of sr are set
  - execution of the interrupt
  - on return, pc, cpsr ← lr, spsr (state register restoring and return to main)



ARM Exception Table

- exception =
  - stops the main program flow to execute a routine
  - example : error, interrupt
- ARM exception table
  - located at memory start (address 0x0000 0000)
  - table entry =

    1 instruction ⇒ B *exception_routine*
  - very simple implementation inside the microcontroler
    1 exception call ~ 1 sub-program call

Address	Meaning
0x 0000 0000	Reset
0x 0000 0004	Undefined instruction
0x 0000 0008	Software interrupt
0x 0000 000C	Prefetch Abort
0x 0000 0010	Data Abort
0x 0000 0014	Reserved
0x 0000 0018	IRQ
0x 0000 001C	FIQ

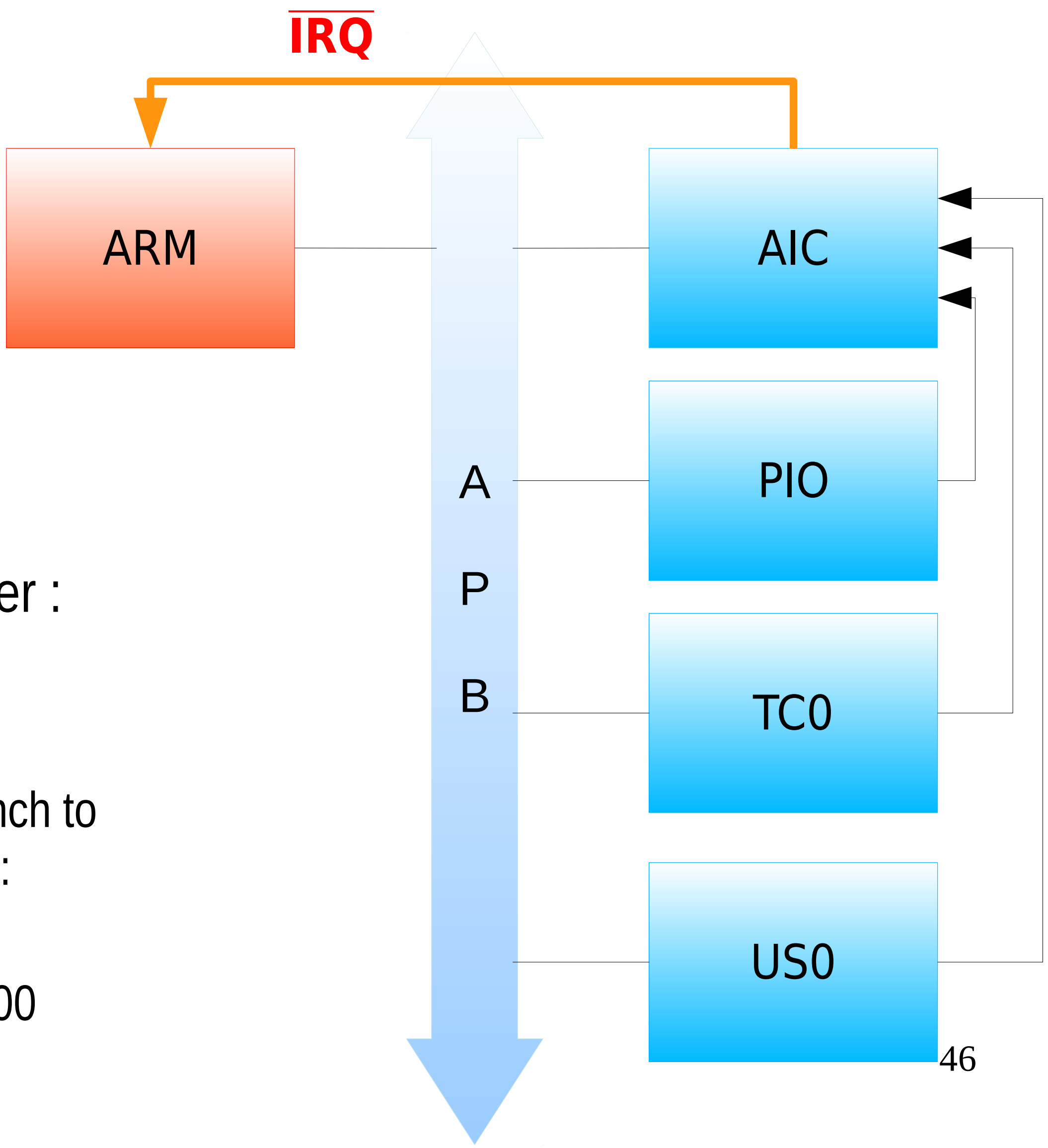
Which Device Caused the Interrupt?

- solution 1: polling
  - look at the state of each peripheral controller
  - beware: long processing ⇒ may block following ITs
  - for example, FIQ (Fast IQ) ⇒ require a processing as fast as possible to not block next FIQs!
  - time cost is as important as the number of peripherals

```
void it_handler(void) {
    if(PIO_ISR & (PUSH1  | PUSH2))
        PIO_handler();
    else if(TC_SR0 & CPCS)
        TC0_handler() ;
    else if(US0_CSR & (TXRDY | TXRDY))
        USART_handler() ;
    ...
}
```

AIC (Advanced Interrupt Controller)

- multiplexer of ITs (until 32 ITs)
- provide an IT vector for each peripheral
- re-routed to IRQ/FIQ line
- handles priorities between interrupts
- manage the typr of IT signal
- fast way to select the IT handler :
  - register AIC_IVR = @handler (located at 0xFFFF F100)
  - IRQ/FIQ handler = indirect branch to AIC_IVR from 0x0000 0018/1C:
  - LDR PC, [PC, #-0xF20] 0x18 + 8 – 0xF20 = 0xFFFFF100



How to program AIC?

- initialization phase
  - find the AIC IT number from the microcontroler manual
  - mask the IT (avoid any spurious IT)
  - set the handler vector and mode
  - purge remaining IT
  - unmask IT (re-enable the IT management)
- IT handler
  - usual handler
  - before leaving, signal the AIC about the end
- for each IT i,
  - AIC_SMRi = IT mode
    - bits 2-0 : priority
    - bits 6-5 : signal type (00 low level, 01 descending front, 10 high level, 11 ascending front)
  - AIC_SVRi = handler address
- other registers (1 bit / IT)
  - AIC_ISR – enabled ITs
  - AIC_IPR – pending ITs
  - AIC_IECR – for enabling ITs
  - AIC_IDCR – for disabling ITs
  - AIC_ICCR – to clear pending ITs
  - AIC_EOICR – read to signal end of IT

Example: with PIO and Timer

- PIO
  - only 1 IT for all pins
  - configure PUSH1, PUSH2 to raise ITs
- timer
  - one IT for each timer
  - different sources of ITs
  - example: CV = RC

```
#define ID_PIOA    2
#define ID_TC0     12

void initialize(void) {
    /* initialization of PIO */
    ...
    /* initialization of timer */
    ...
    /* IT masking */
    AIC_IDCR =
        (1 << ID_PIOA) | (1 << ID_TC0);

    /* set vector and mode */
    SMR2 = 0x00;
    SVR2 = PIOA_handler;
    SMR12 = 0x12;
    SVR12 = TC0_handler;

    /* in case of inconsistent state */
    AIC_EOICR = 0;
    AIC_ICCR = ID_PIOA | ID_TC0;

    /* activation */
    AIC_IECR = ID_PIOA | ID_TC0;
}
```

Example: continued

```
int s = 0;
int p1 = 0, p2 = 0 ;

void PIOA_handler(void)
__attribute__ ((interrupt("IRQ"))) {
    int v ;

    /* button 1 management */
    if(p1 && (PIO_PDSR & PUSH1))
        { p1 = 0 ; TC_RC0 += 1000 ; }
    else if(!p1 && !(PIO_PDSR & PUSH1))
        p1 = 1 ;

    /* button 2 management */
    if(p2 && (PIO_PDSR & PUSH2))
        { p2 = 0 ; v -= 10000 ; }
    else if(!p2 && !(PIO_PDSR & PUSH2))
        p2 = 1 ;

    /* acknowledge PIO and AIC */
    v = PIO_ISR ;
    V = AIC_EOICR ;
}
```

```
void TC0_handler(void)
__attribute__ ((interrupt("IRQ"))) {
    int v ;

    /* LED manegement */
    if(s) PIO_CODR = LED1 ;
    else PIO_SODR = LED1 ;
    s =!s ;

    /* acknowledge TC0 and AIC */
    v = TC_SR0 ;
    V = AIC_EOICR ;
}

int main(void) {
    initialiser() ;
    while(1) ;
}
```

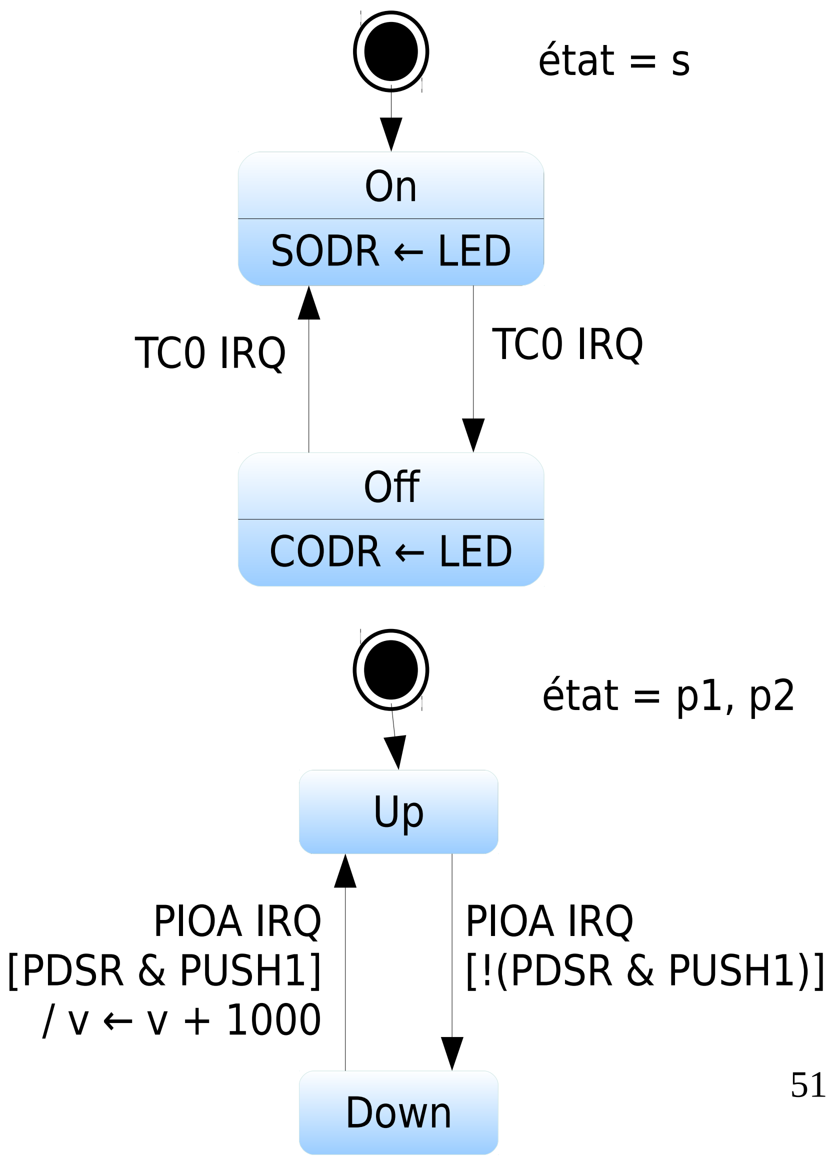
Programming with ITs (1)

- principles
  - IT execution time must be short to avoid to slowdown the reactivity of the system!
- solution
  - IT analyses the state of the peripherals
  - IT sends a message to the main program using global variables
  - the main program polls the main variables
  - when the global variables are set, the associated processing is performed
- main program / interrupt synchronisation
  - easy as only one executes at a time
- synchronization variable
  - as simple as a single boolean
  - as complex as a message box

```
int LED_update = 0 ;
void TC0_handler(void)
__attribute__ ((interrupt("IRQ"))) {
	int v ;
	LED_update = 1 ;
	v = PIO_ISR ;
	AIC_EOICR = 0 ;
}

int main() {
	initialiser() ;

	while(1) {
		if(LED_update) {
			if(s) PIO_CODR = LED1 ;
			else PIO_SODR = LED1 ;
			s =!s ;
			afficher_LED = 0 ;
		}
	}
}
```

ITs with Automata

- state =
  - global variables
- events
  - IRQ/FIQ – peripheral IT
  - condition: polling of the peripheral
- actions
  - performed inside the IT handler
  - do not forget to acknowledge peripheral and AIC!



Exercise (3a)

We will implement the serial port writing using the interrupts. To get a benefit, we will use a circular queue to store characters that needs to be sent. This circular queue is defined as below:

```
#define USART_SIZE  256
char usart_buf[USART_SIZE] ;
int usart_head, usart_tail, empty = 1 ;
```

When the queue is empty, `usart_head` = `usart_tail` and `empty` = 1.

1) Write the function `void putc(char c)` that (a) either add *c* to the queue if the USART is working, (b) or send the character to the USART.

Exercise (3b)

2) Write the USART handler that sends character to the USART if the queue is not empty. The interrupt must return as soon as the character is sent to not block other interrupts.

3) Write the function `void puts(char *s)`.

4) Write the initialization of USART and AIC:

USART configuration:

- ID_US0 = 6
- US0_IER = TXRDY
  enable USART IT “ready to transmit”
- USO_IDR ← TXRDY
  disable USART IT “ready to transmit”

Debugging with IT (1)

- beware when you use of a debugger
  - reading the memory is not a passive action!
  - memory visualization ⇒ effect on a peripheral controller
- example
  - reading the AIC_EOICR → acknowledge the AIC that the interrupt is ended!
  - set the AIC in debugging mode
    - AIC_DCR ← AIC_MPROT
    - now a write to AIC_EOICR is required to acknowledge the end of the interrupt

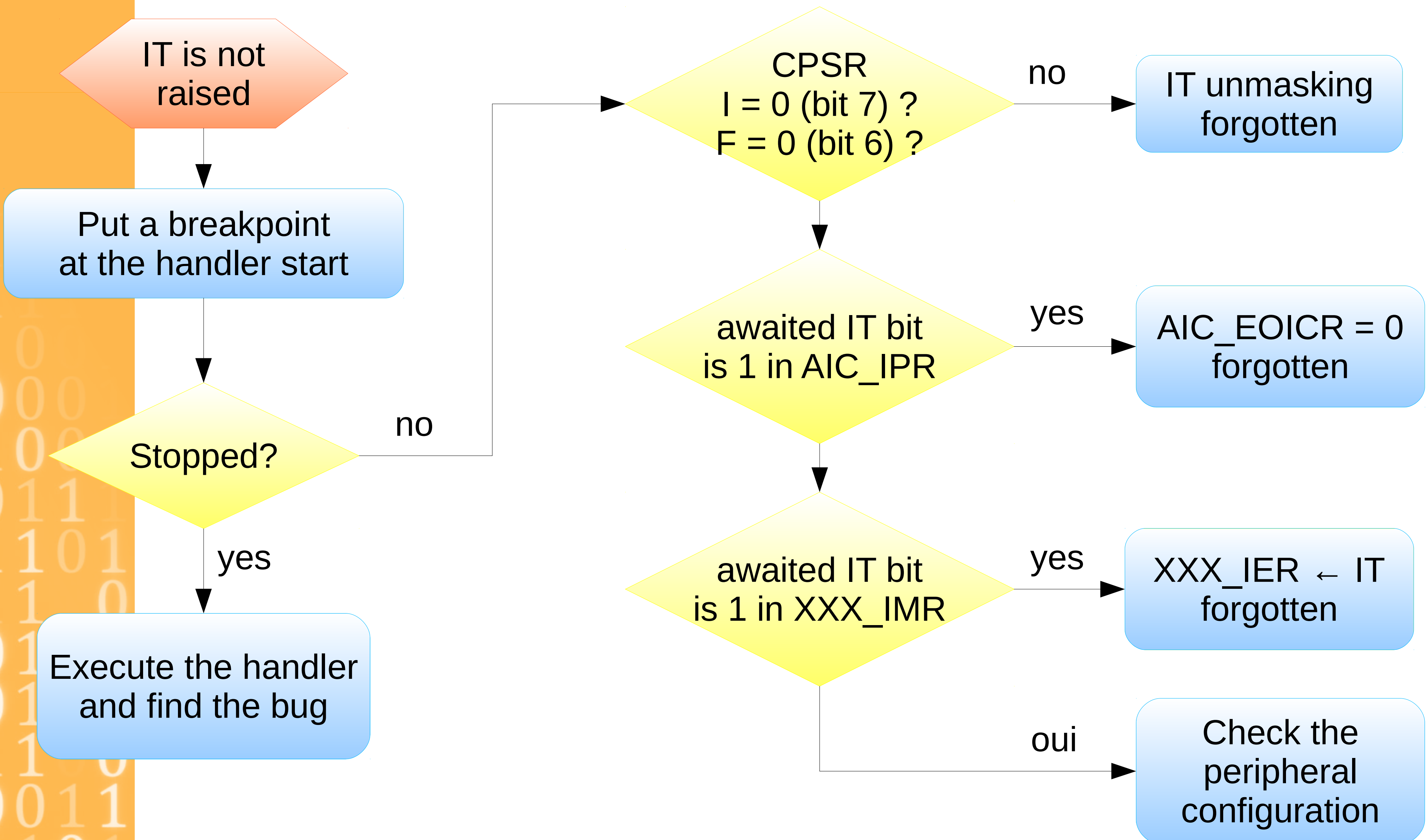
Debugging with ITs (2)

- observability problem
  - debug session inside a IT is long (from the microcontroler point of view)
  - raising and processing an IT takes very little time in reality

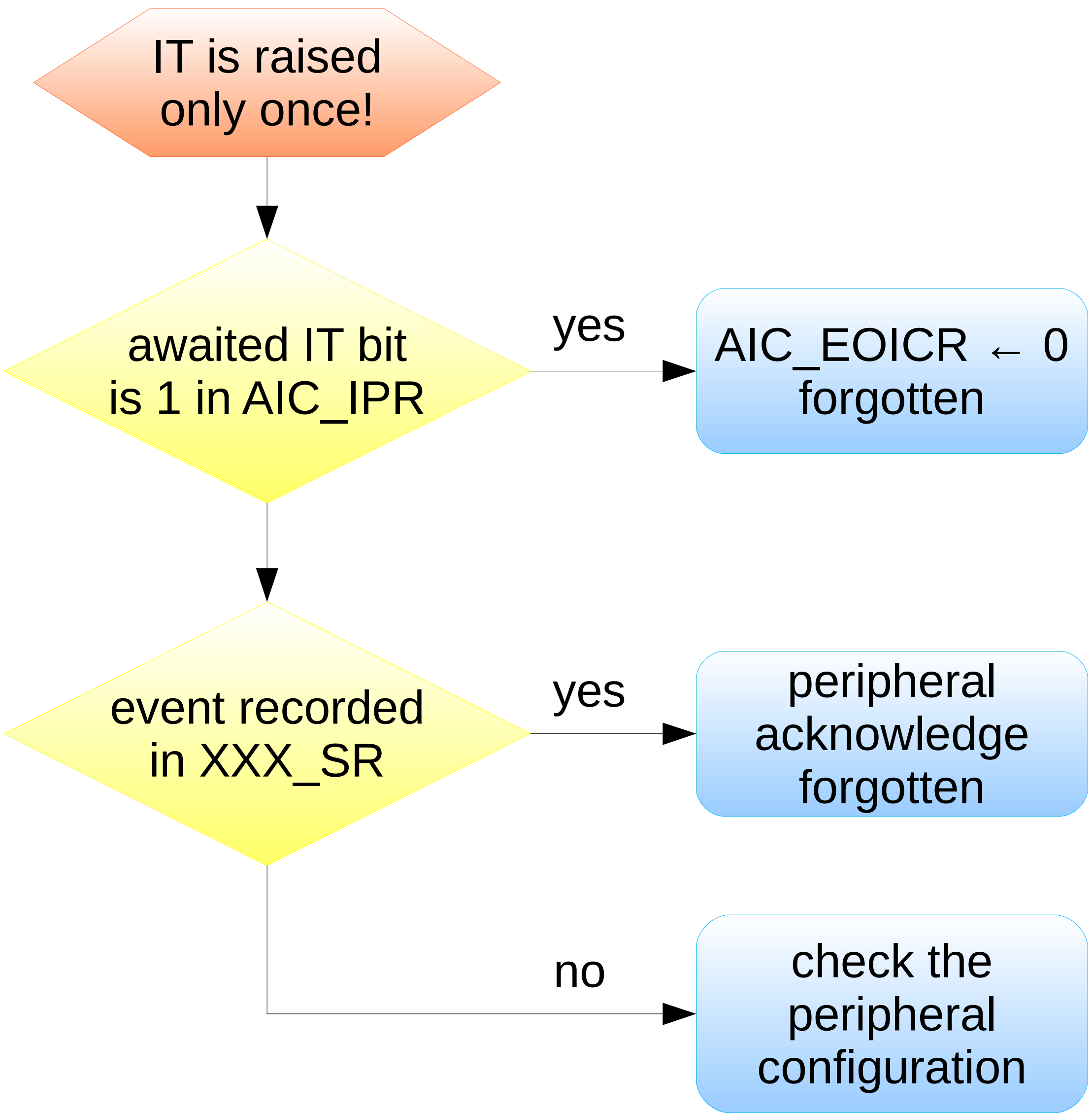
    Example: several times per second for the timer
  - we have not control on external effects

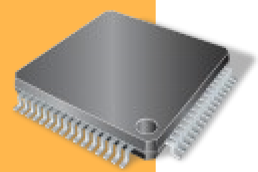
  ⇒ usually, it is not possible to “stop the world” and debug in real-time
- solutions
  - debugging IT = observation of IT alone followed by the shut-down of the system
  - tracing the events of the interrupt inside a buffer ⇒ transmitted using any flow input (like USART) for post-crash/bug examination
  - using an hardware probe to observe the microcontroler bus activity and to store information to external storage (may be expensive)

Recipe to Debug IT (1)

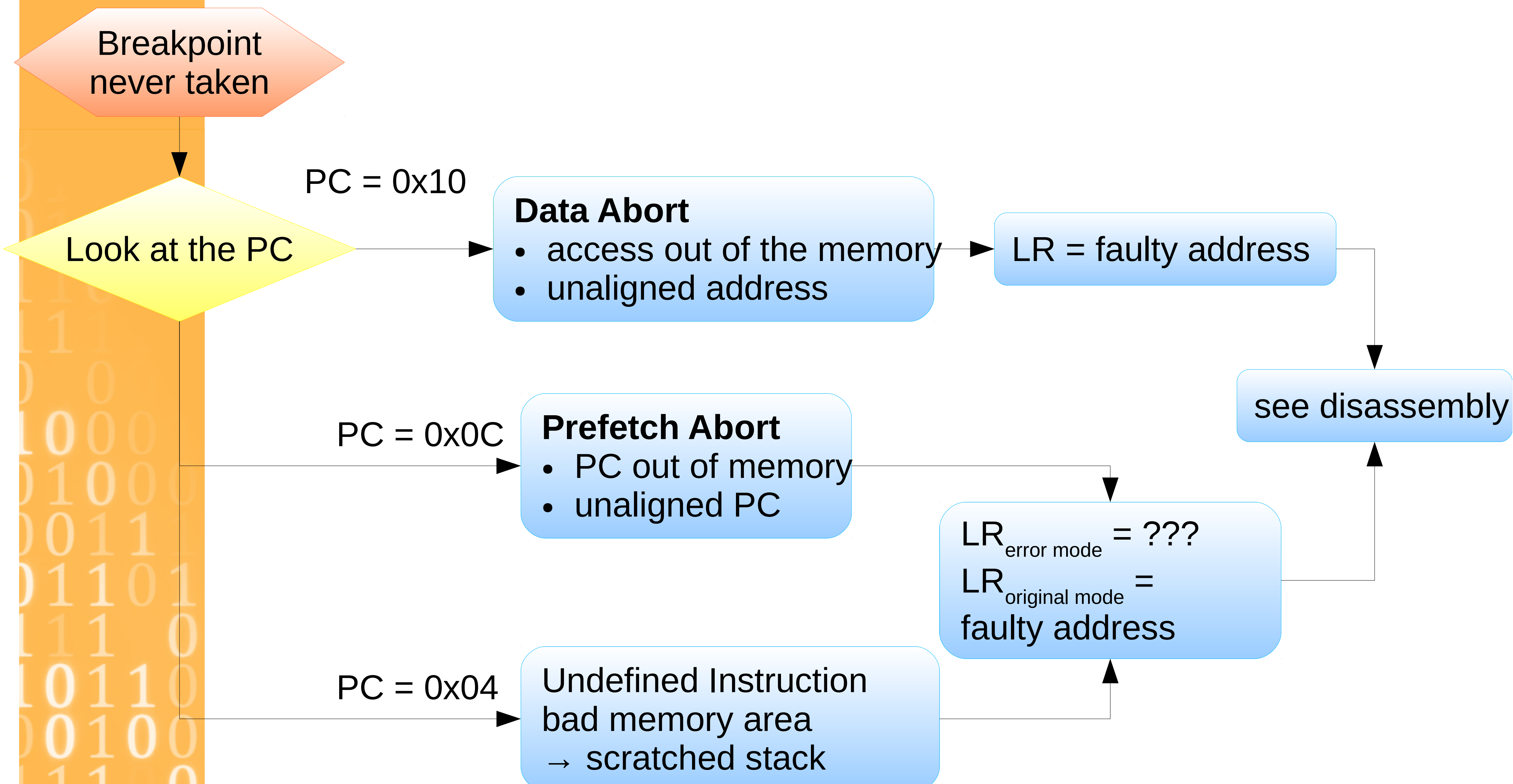


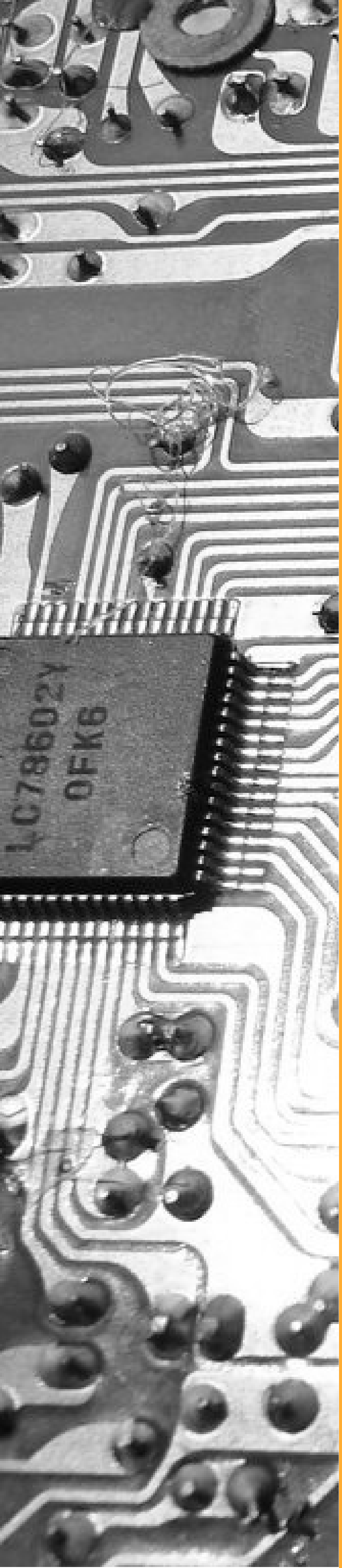
Recipe to Debug IT (2)





Recipe to Debug IT (3)



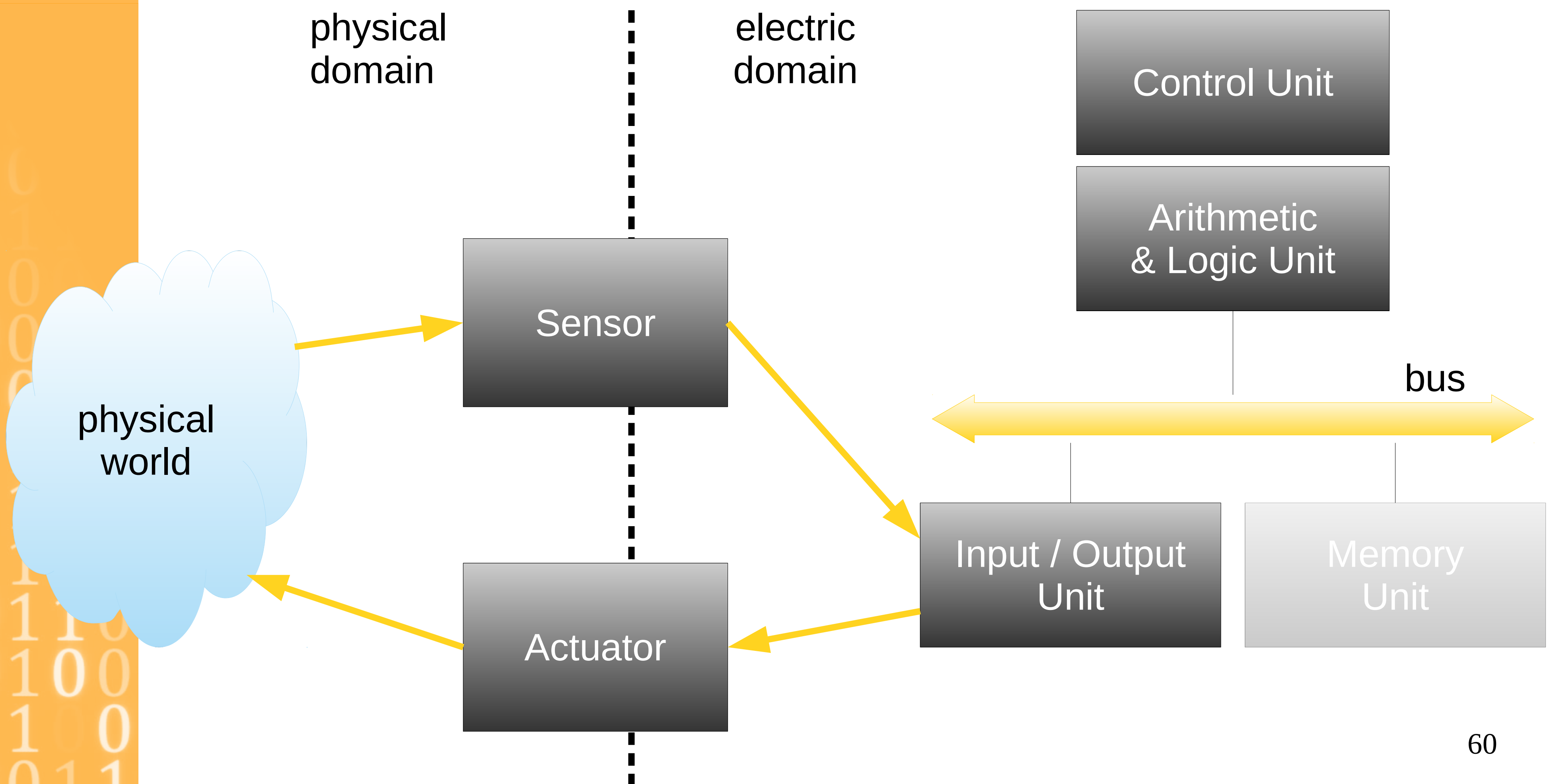


Overview

- Introduction
- The microcontroler
- I/O Management
- Using interrupts
- **Sensors & Actuators**

Transducers

Definition: transducer
A transducer is a device that converts energy from one form to another.
Wikipedia



More about sensors

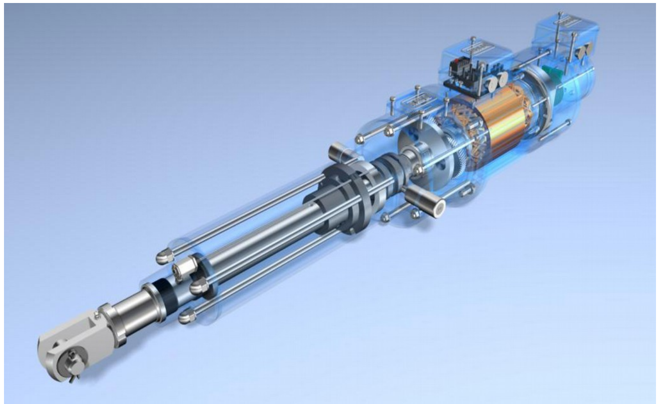
- Example
  - IR receiver: IR light
  - sun cell: light
  - thermistor:
  - temperature
  - pressure
  - sound
  - speed
  - etc
- physical domain
  - maximum, minimum
  - resolution, precision
  - error (%)
  - linearity
- electric domain
  - maximum, minimum
- working domain

  temperature, pressure, etc

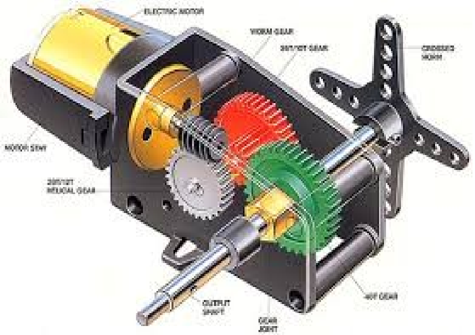
More about actuators

- physical domain
  - maximum, minimum
  - precision, resolution
- behaviour
  - update time
  - linear
  - hysteresis
- working domain
  - temperature
  - pressure
  - etc

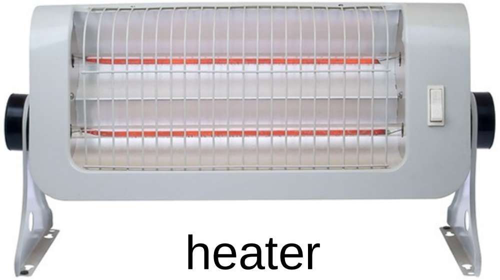
linear motor



quadcopter motor



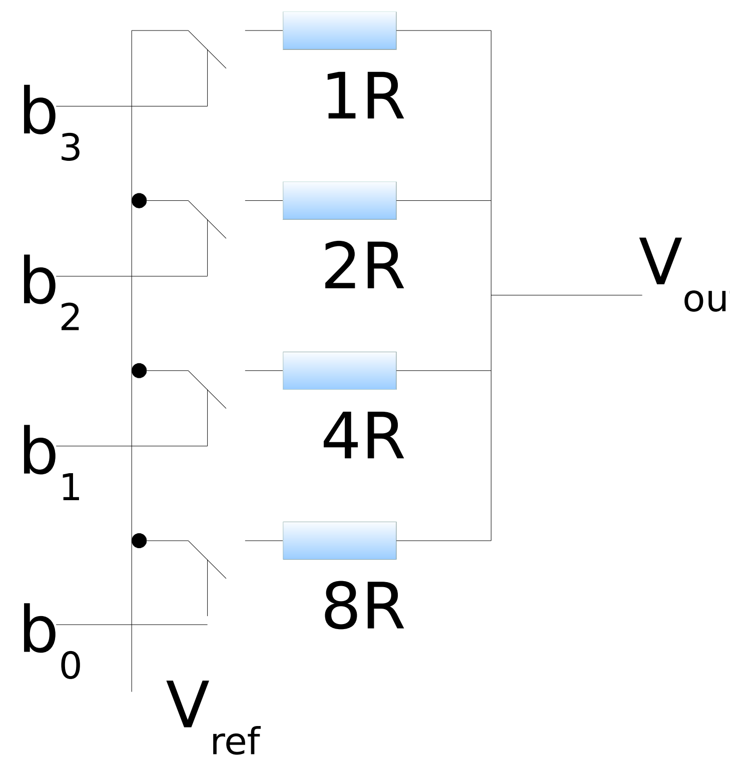
servo-motor



heater

DAC
(Digital-Analogic Converter)

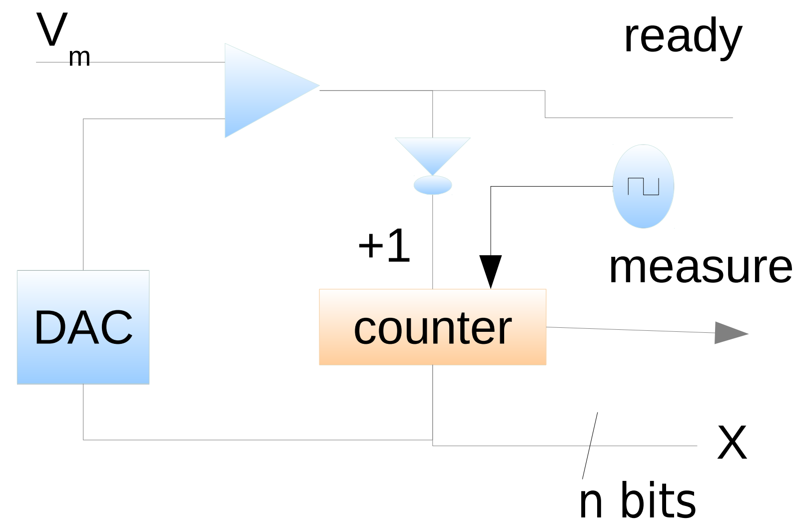
- basic actuator controller
  - input: number $(b_3\ b_2\ b_1\ b_0)_2$
  - output: voltage $V_{out} \in [0, V_{ref}]$
- resistor network
  - $v_i = (b_3\ b_2\ b_1\ b_0)_2$
  - R = sum of b_i / R_i
  - $R_i = 1/(2^{3-i} \times R_i)$
- properties
  - sensitivity: $q = V_{ref} / 2^n$
  - short response time
    transistor time + resistor time
  - very cheap: n bits →
    n transistors + n resistors
  - limited: 12 bits → R × 2048

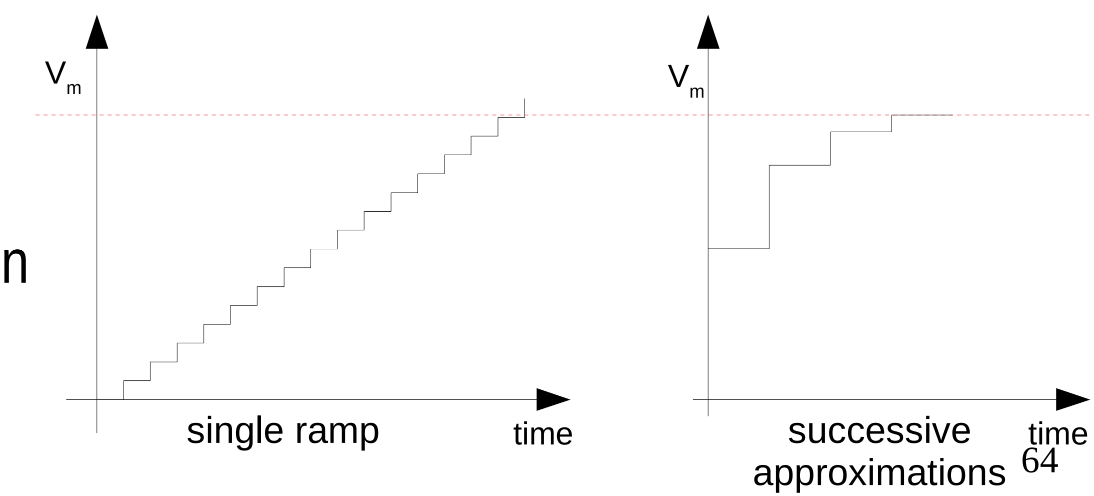


$$V_{out} = V_{ref} \times \left(\frac{b_3}{1R} + \frac{b_2}{2R} + \frac{b_1}{4R} + \frac{b_0}{8R} \right)$$

ADC
(Analogic-Digital Converter)

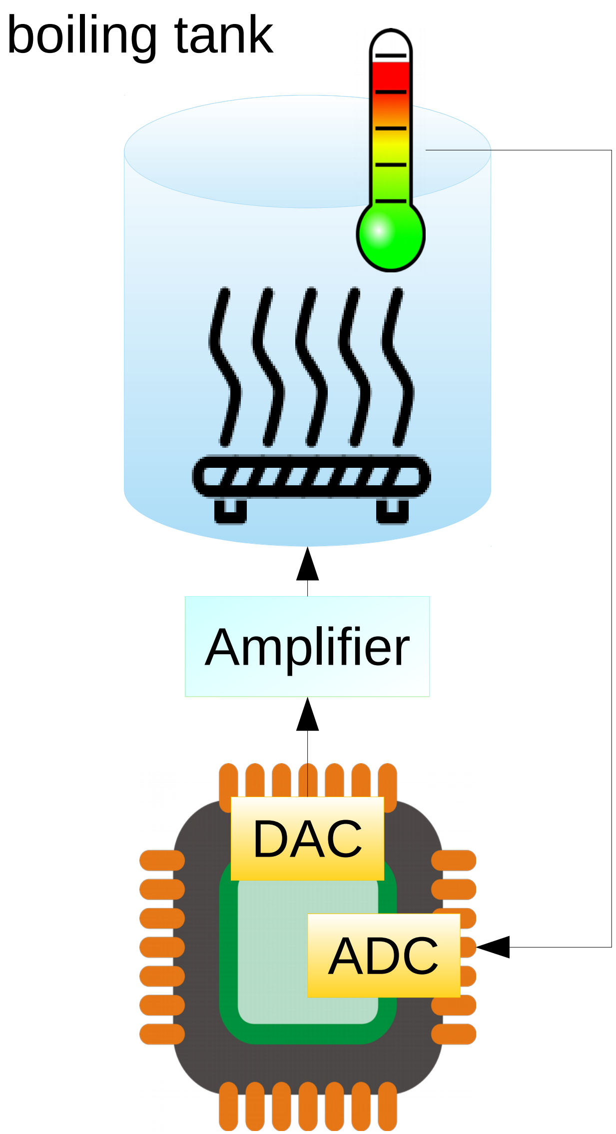
- ADC
  - controller for actuators
  - input: voltage V_m
  - output: number X n-bit
- performances
  - clock required
  - measurement time: $n \times (t_{DAC} + t_{comparator} + t_{counter})$
  - resolution depends on time





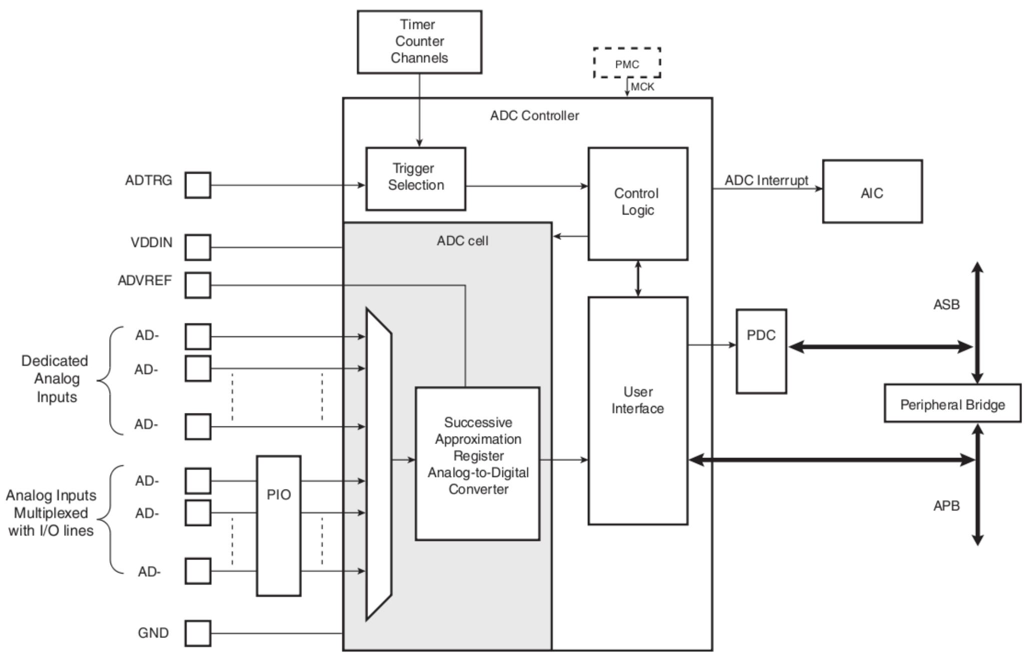
reading 15 for 4 bits

The Boiler of espresso machine with DAC




```
#define FREQUENCY  5
#define DELAY_MS   1000/FREQUENCY
#define REF_TEMP   90

void main(void) {

    /* peripherals initialization */
    …

    /* control loop */
    while(1) {
        float y = read_ADC();
        float e = REF_TEMP – y;
        write_DAC(e);
        wait(DELAY_MS);
    }
}
```

AT91SAM7S: successive approximations



- interrupt ID_ADC = 4
- base address = 0xFFFD 8000
- 8 input channels
  (PIO multiplexed)
- 1 converter
- resolution: 8 or 10 bits
- maximum frequency
  - 10-bit → 384 K samples/s
  - 8-bit → 583 K samples/s
- driving
  - polling
  - interrupts
  - triggered by timer

AT91SAM7S: Driving the ADC

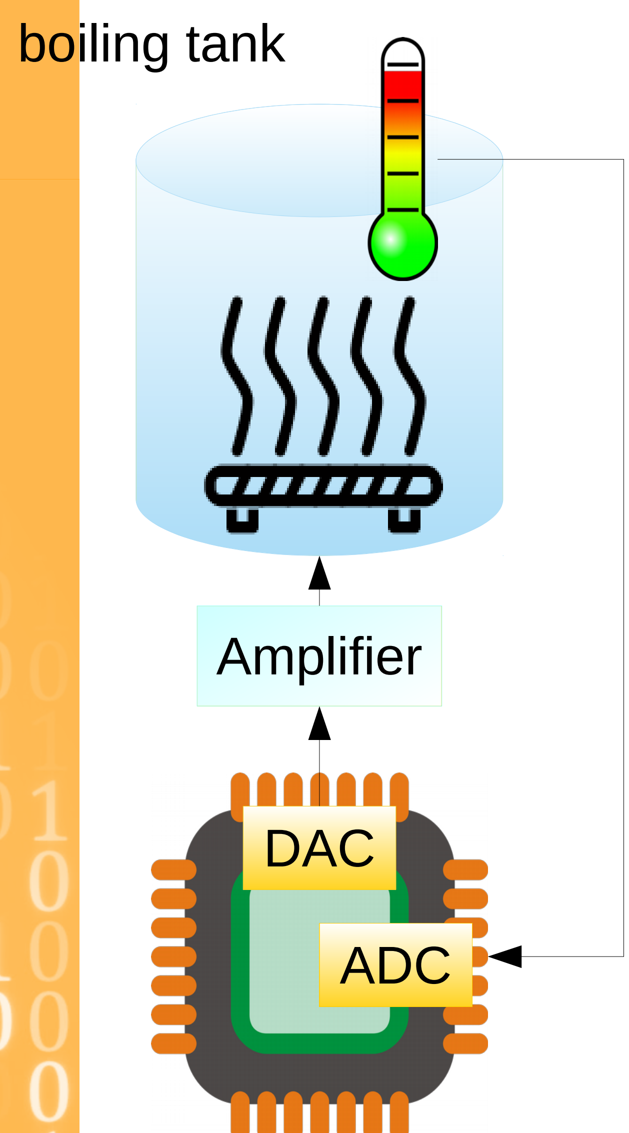
Polling approach

```
/* thermistor on AD5 */

/* initialization */
ADC_MR = 0x3f00;   /* 10 bits */
    /* frequency = MCK / (0x3F + 1) * 2 */
ADC_CHER = 1 << 5;
    /* channel 5 activation */

/* polling conversion */
ADC_CR = ADC_START;    /* start up */

/* wait for conversion end */
while(!(ADC_SR & (1 << 5)));

/* read the sample */
uint32_t v = ADC_CDR5;

/* conversion in degree */
float t = (float)temp * a + b;
/* a, b linearity coefficients */
```

Interrupt approach

```
/* AIC initialization */
AIC_IDCR = 1 << ID_ADC ;
AIC_SMR[ID_ADC] = 0 ;
AIC_SVR[ID_ADC] = adc_irq ;
AIC_IECR = 1 << ID_ADC ;

/* IT on channel 5 end */
ADC_IER = 1 << 5;

/* startup */
ADC_CR = ADC_START;

/* IT handler */
void adc_irq(void)
__attribute__ ((interrupt("IRQ"))) {
    uint32_t v = ADC_CDR5;
    /* sample processing */
    AIC_EOICR = 0 ;
}
```

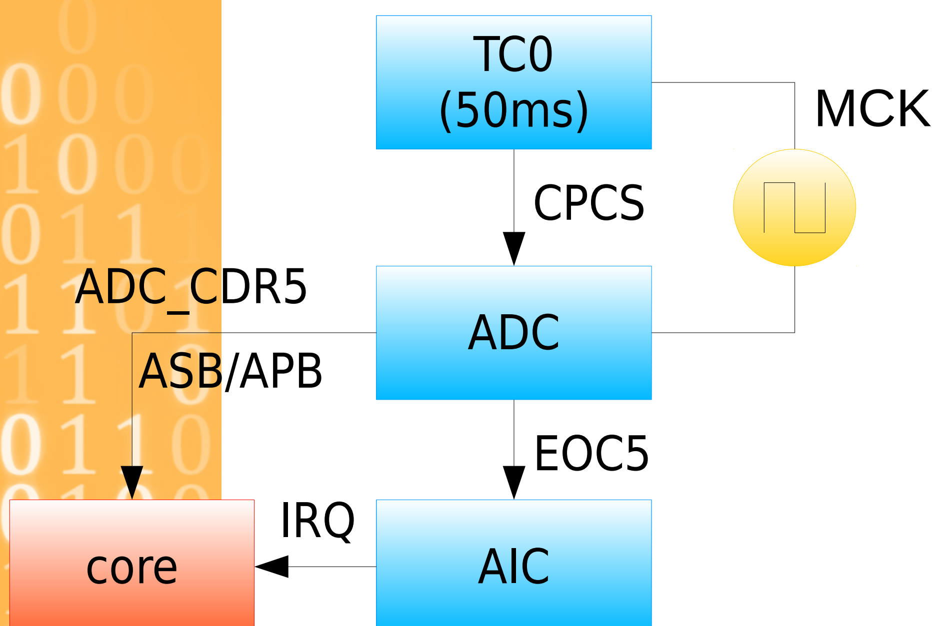
Issue 1: error in measure → oversampling + average




```
#define FREQUENCY  5
#define OVER_SAMP  10
#define DELAY_MS   1000/(FREQUENCY * OVER_SAMP)
#define REF_TEMP   90

void main(void) {
    float sum = 0;
    int cnt = 0;

    /* peripherals initialization */
    …

    /* control loop */
    while(1) {
        sum += read_ADC(); cnt++;
        if(cnt == OVER_SAMP) {
            float y = sum / OVER_SAMP;
            float e = REF_TEMP – y;
            write_DAC(e);
        }
        wait(DELAY_MS);
    }
}
```

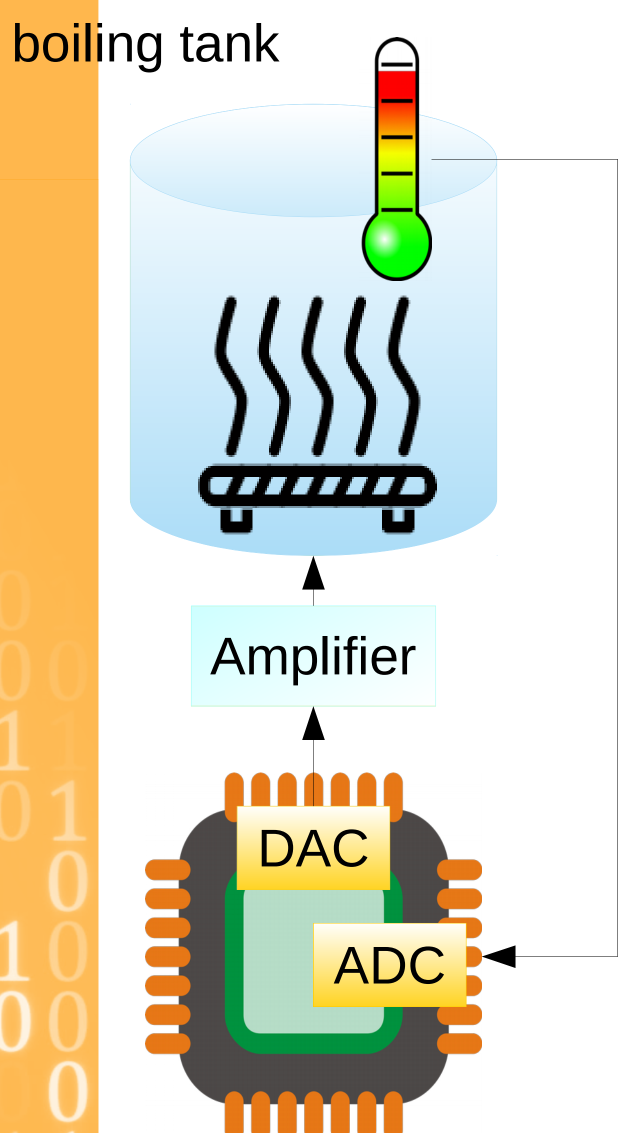
Triggering ADC from Timer

- improve periodicity precision
  - triggered by the timer
  - IT to process the result
  - exactly the period whatever the processing time!




```
/* IT initialization */
AIC_IDCR = 1 << ID_ADC ;
AIC_SMR[ID_ADC] = 0 ;
AIC_SVR[ID_ADC] = irq_aic ;
AIC_EICR = 1 << ID_ADC ;
ADC_IER = 1 << 5 ;

/* ADC initialization */
ADC_MR = 0x3f00
    | ADC_TRGEN      /* extern trigger */
    | ADC_TRGSEL_0; /* timer 0 */
ADC_CHER = 1 << 5;

/* initialisation du timer */
TC0_CCR = TC_CLKDIS ;
TC0_CMR = TC_CLOCK5 | TC_WAVE;
TC0_RC ← MCK / 1024 / 20; /* 50 ms */
TC0_CCR ← TC_CLKEN;
TC0_CCR ← TC_SWTRG;
```

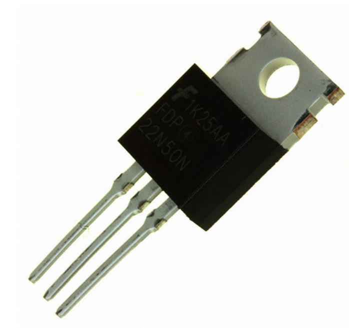
Issue 2: different physical quantities

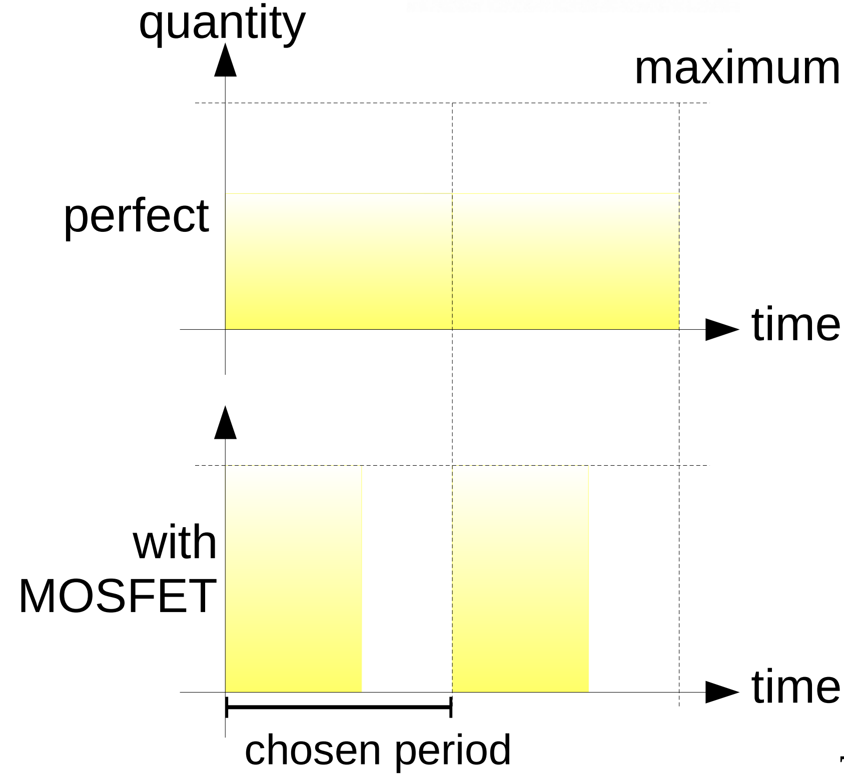



```
#define FREQUENCY  5
#define OVER_SAMP  10
#define DELAY_MS   1000/(FREQUENCY * OVER_SAMP)
#define REF_TEMP   90

void main(void) {
    float sum = 0;
    int cnt = 0;

    /* peripherals initialization */
    …

    /* control loop */
    while(1) {
        sum += read_ADC(); cnt++;
        if(cnt == OVER_SAMP) {
            float y = sum / OVER_SAMP;
            float e = REF_TEMP – y;
            write_DAC(to_volt(e));
        }
        wait(DELAY_MS);
    }
}
```

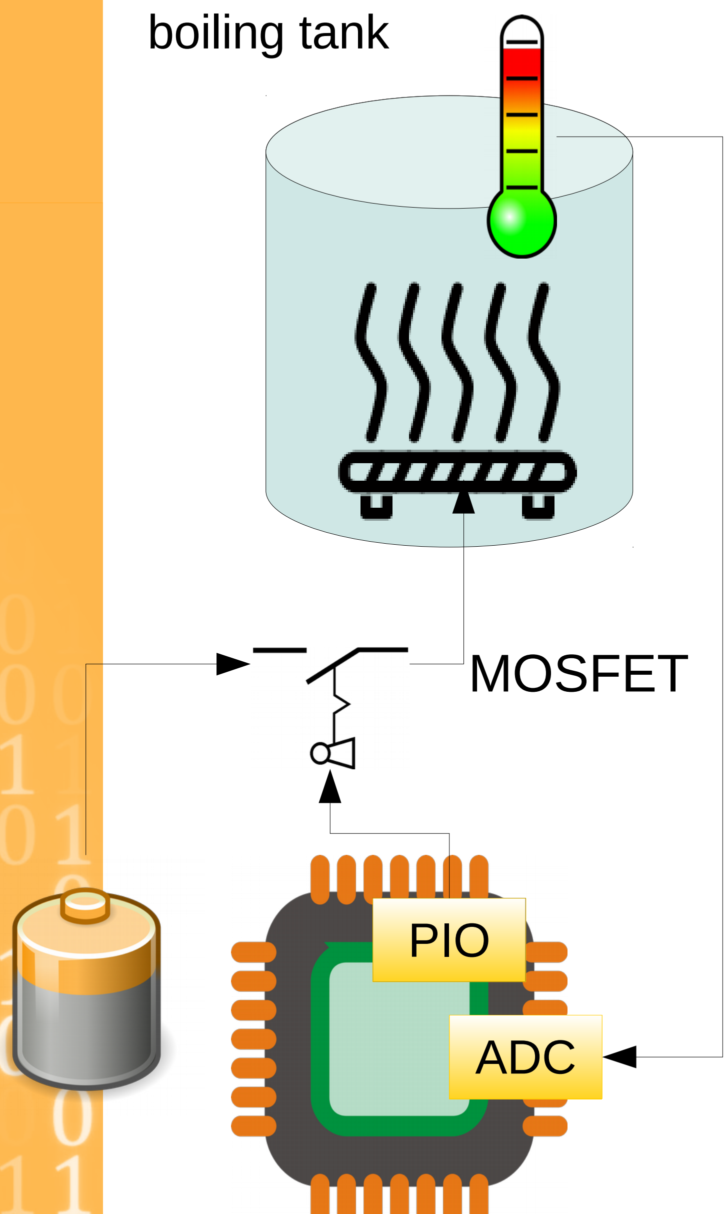
Problem: electrical domain

- power domain
  - microprocessor – 1.8-5V
  - actuator – 220V and more
- solution – Power transistor (MOSFET)
  - controlled in microelectronic domain
  - delivering actuator domain current
  - problem: on/off work
- solution
  - exploit inertia of physical effects
  - power = time × effect
  - integrate over time





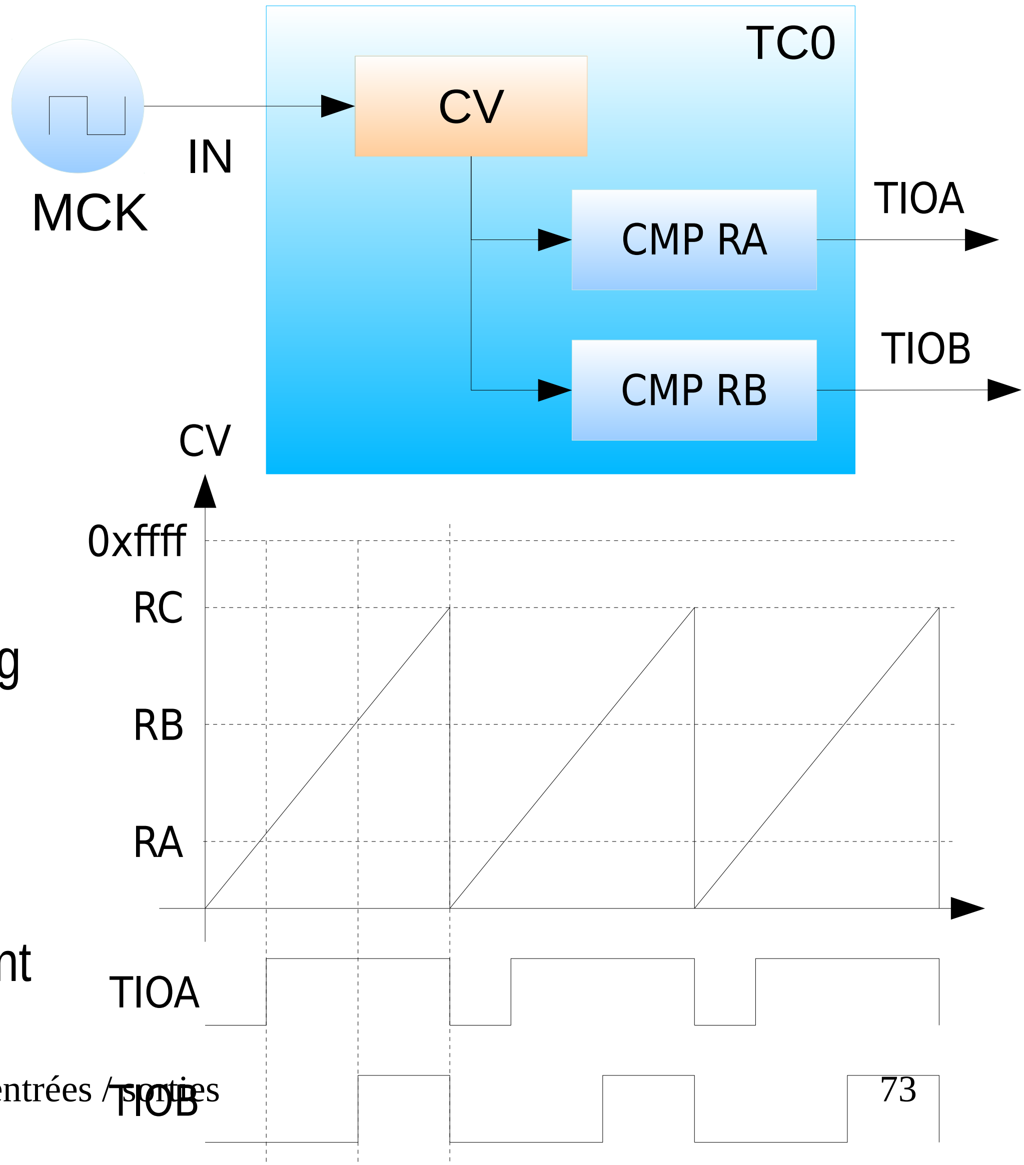
Issue 3: MOSFET Approach




```
…
void main(void) {
    …

    /* control loop */
    while(1) {
        float e = REF_TEMP – readADC();
        float u = to_time(e);
        setPIO(HEATER, 1);
        wait(u);
        setPIO(HEATER, 0);
        wait(DELAY_MS – u);
    }
}
```

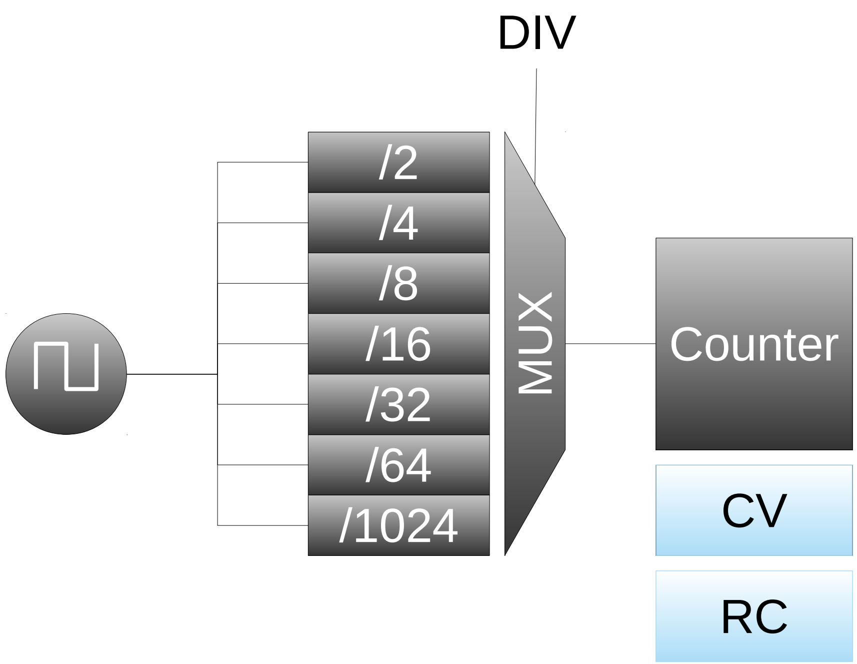
Timer in PWM mode (Pulse Width Modulation)

- timer configuration
  - duration = motor period
  - TIOA = 1 when RA ≤ VC
  - TIOB = 1 when RB ≤ VC
  - 1 timer ⇒ 2 controlled motors
- options
  - reset if VC = RC
  - ascending-descending counting (pulse centring)
  - reset on external event (force feedback)
  - comparison effect: front ↗, front ↘, inversion



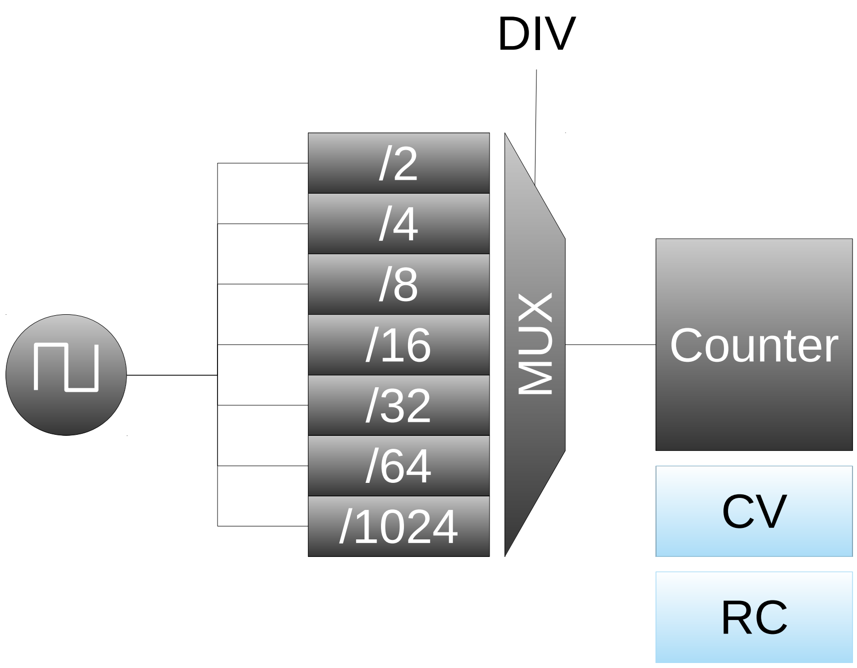
Configuring the timer

- MCK Hz = 1s
  - CV – 16-bit → max 2^{16} → max time = 2^{16}/MCK
  - MCK = 48MHz
  - max time = 1.36 ms
- divide by 1024
  - max time = 2^{16}/(MCK / DIV)
  - max time = 1.39s



Choosing the divider

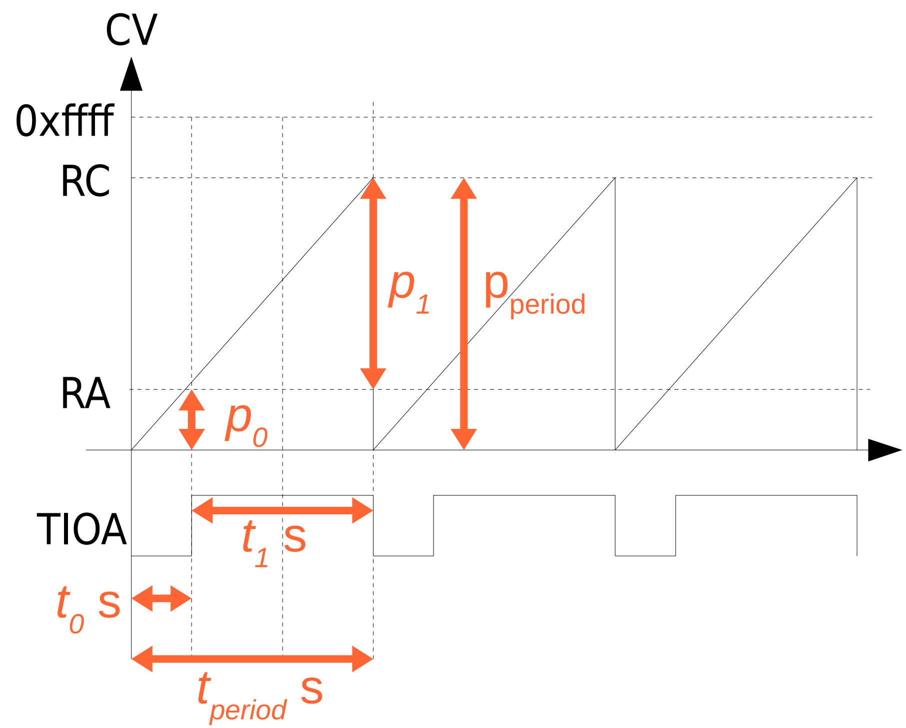
- best DIV
  - for period t
  - in pulses p
- maximum precision
  - $p = 2^{16}$
  - $d = MCK \times t / p$
  - best DIV $\geq$ d
- example
  - t = 20ms
  - $d = 48MHZ \times 0.02 / 2^{16} = 14{,}65$
  - DIV = 16



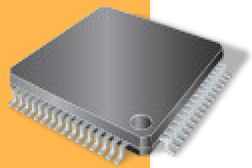
$$\frac{t \quad s}{1 \quad s} = \frac{p \quad \text{pulses}}{MCK / \mathrm{DIV}}$$

Computing the time in pulses

- time → pulses
  - p_{period} = MCK × t_{period} / DIV
- example: t_{period} = 20ms
  - p_{period} = 48MHZ × 0.02 / 16 = 60,000 pulses
  - RC ← 60000
- generating PWM
  - PIOA = 1 if RA ≥ CV
  - t_1 – time PIOA1 = 1
  - p_1 = MCK × t_1 / DIV
  - p_0 = p_{period} – p_1



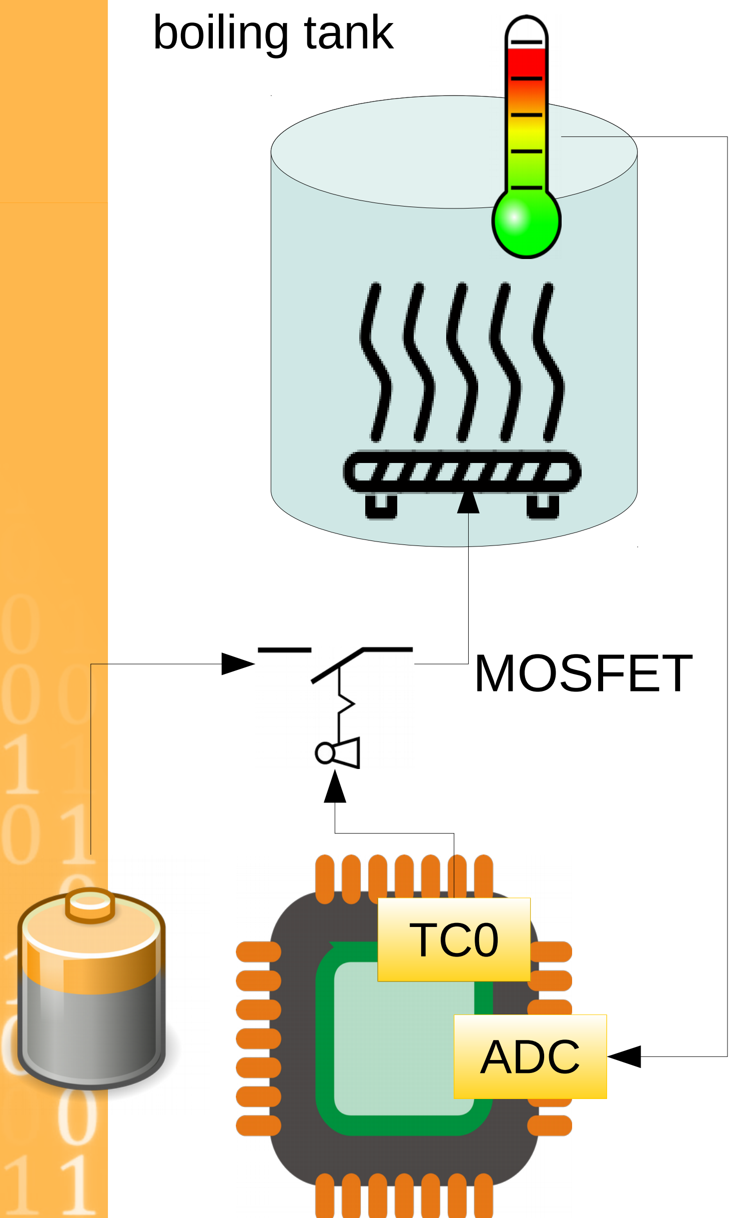
- example: t_1 = 5ms
  - p_1 = 48MHz × 0.015 / 16 = 15,000
  - p_0 = 60,000 – 15,000 = 45,000
  - RA ← 45,000



Programming the Timer in PWM

```
/* PIO initialization –  output on PA0 / PWM0 / TIOA0 */
PIO_BSR = 1 << 0 ; PIO_PDR = 1 << 0 ; PIO_OER = 1 << 0 ;

/* TC0 initialization: d = 16 (clock 3) */
TC0_CCR = TC_CLKDIS ;
TC0_CMR =
      TIMER_CLOCK2      /* / 16 */
   |  TC_WAVSEL_UP_CMP  /* ascending, reset on CV = RC */
   |  TC_WAVE           /* PWM mode */
   |  TC_ACPA_SET       /* TIOA = 1 on CV = RA */
   |  TC_ACPC_CLEAR ;   /* TIOA = 0 on CV = RC */
TC0_RC = 60000;
TC0_CCR = TC_CLKEN | TC_SWTRG ;

/* RA setting with v ∈ [0, 20[ ms (v ms = v / 1000 s) */
TC0_RA = 60000 - v * MCK / 32 / 1000 ;
```

Exercise: PWM



Program the boiler tank using PWM with TC0.

1) Write function init_boil() that initialize PIO and TC0.
   - t_{period} = 100ms

2) Write function set_boil(int t) that program the boiler tank to reach temperature t considering:
   - target temperature is 95°
   - ratio of PWM up (at 1) = t/95

     → pulse width = t × p_{period} / 95

3) Rewrite the main function to use TC0 PWM.