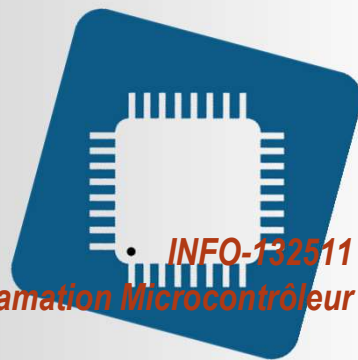


UNIVERSITÉ  
La Rochelle



• **INFO-132511**

## Objets Connectés : Programation Microcontrôleur

- Avant de passer au protocoles réseau :
  - Interruption
  - Mode de veille

1

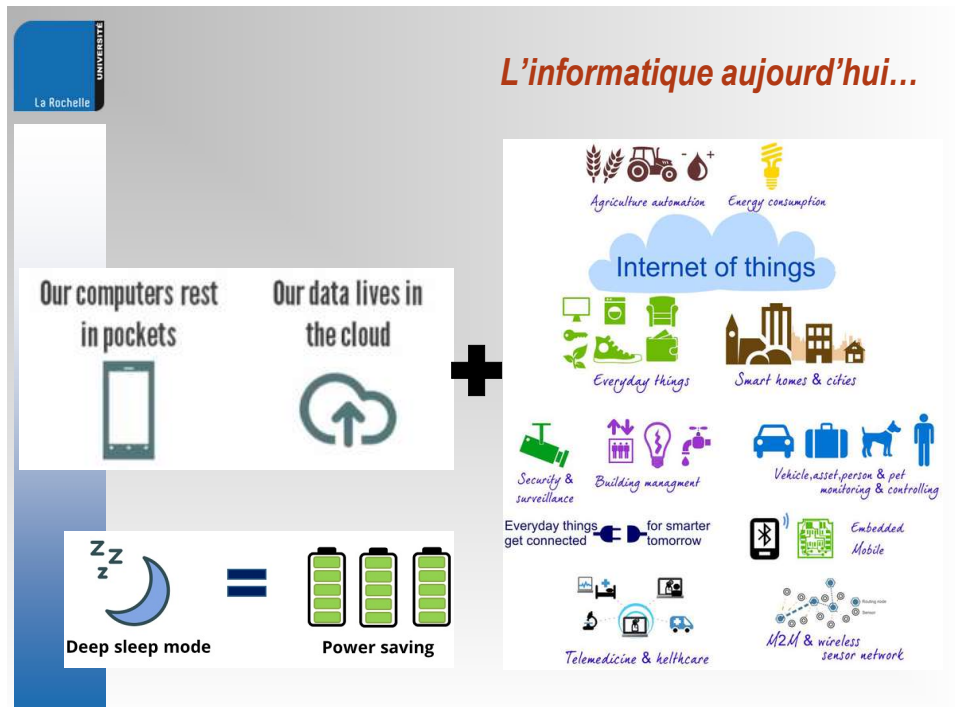
UNIVERSITÉ  
La Rochelle

## Miniaturisation, équipements nomades...



The diagram illustrates the trend of miniaturization in computing equipment. It shows a progression from a large, bulky mainframe computer (top left) to a compact laptop (top right), and from a large, rugged mobile phone (bottom left) to a sleek, modern smartphone (bottom right). Blue arrows indicate the direction of this technological evolution.

2



3

- UNIVERSITÉ  
La Rochelle
- ## Exploiter les performances « basse consommation » des microcontrôleurs
- Changement de technologie :
    - Vacuum Tubes → Transistor → Integrated Circuits
    - TTL → CMOS
  - Conséquences :
    - Miniaturisation
    - Baisse de la consommation
      - Tension d'alimentation (Vcc) : 5V → 3,3V → 2,2V
      - Consommation proportionnelle à la fréquence (CMOS)
  - **Evidemment, il faut que la programmation suive...**
    - Ne plus utiliser des boucles pour attendre. Utiliser les timers intégrés et mettre le CPU en veille
    - Ne plus faire de *polling* pour détecter des changements d'état. Utiliser des interruptions (source des interruptions : timers, événements externes) pour sortir le CPU de sa veille.
    - Le *framework* Arduino n'est pas forcément adapté !

4

## Exemple : Alimentation d'un PC (Personal Computer)

- Alimentation du IBM PC AT (1981) : 64W sur du 5V et 12V
- Standard ATX : 60W sur du 3.3V  
Au total (3.3V, 5V, 12V) environ 300W
- Sur un PC actuel : Maximum de puissance sur le rail 12V.  
La carte mère et le GPU convertissent le 12V dans les tensions nécessaires.

480 Watt ATX12V 1.3 PSU		
Voltage	Maximum current	Maximum wattage
+3.3 volts	34.0 amps	112.2 watts
+5 volts	35.0 amps	175 watts (200 watts maximum combined +5 and +3.3)
+12 volts	28.0 amps	336 watts
5 volts standby	2.0 amps	10 watts
-12 volts	1 amps	12 watts

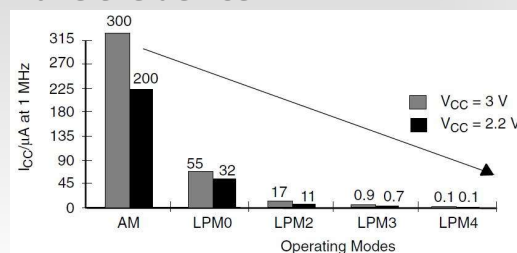
- Mac Pro (2013) : inactif 43 W, actif 205 W
- Raspberry Pi 4 (2019) : 15W

5

## Mode de veille des microcontrôleurs

- MSP430 de T.I. (un des précurseurs...)

- Active Mode (AM): CPU, all clocks, all modules active ( $\approx 300 \mu\text{A}$ )
- LPM3: On coupe tous les signaux d'horloge interne (pas de fonctionnement CPU) sauf l'horloge qui fait fonctionner le timer ( $\approx 1 \mu\text{A}$ )
- LPM4: On coupe toute les horloges. Seule la RAM est alimentée ( $\approx 0.1 \mu\text{A}$ ). **Only off-chip interrupts can wake the device**



To go further, drop core voltage (RAM retention > 1.8V)  
<50 nA pin leakage

6

UNIVERSITÉ

La Rochelle

## Les modes de veille de l'ESP32

**Power Requirement**

- Operating Voltage: 2.2V to 3.6V
- On-board 3.3V 600mA regulator
- 5  $\mu$ A during Sleep Mode
- 250mA during RF transmissions

esp-idf

deep\_sleep\_gpio

deep\_sleep\_timer

deep\_sleep\_ulp (ultra low power coprocessor)

7

UNIVERSITÉ

La Rochelle

Mode actif / active Mode		Mode modem/ modem Mode	
	<b>Allumés</b> Wifi Bluetooth Radio Noyau ESP32 Processeur ULP Périphériques RTC <b>Eteints</b>		<b>Allumés</b> Noyau ESP32 Processeur ULP RTC <b>Eteints</b> Wifi Bluetooth Radio Périphériques
Consommation : 160-260mA		Consommation : 3-20mA	
Light Sleep		Deep Sleep	
	<b>Allumés</b> noyau ESP32 processeur ULP <b>En pause</b> noyau et mémoire <b>Eteints</b> wifi bluetooth Radio Périphériques		<b>Allumés</b> RTC Processeur ULP <b>Eteints</b> Wifi Bluetooth Radio Périphériques Noyau ESP32
Consommation : 3-20mA		Consommation : 10 $\mu$ A	

esp\_light\_sleep\_start();

esp\_deep\_sleep\_start();

8

4

Table 5: Power Consumption by Power Modes		
Power mode	Comment	Power consumption
Active mode (RF working)	Wi-Fi Tx packet 13 dBm ~ 21 dBm	160 ~ 260 mA
	Wi-Fi / BT Tx packet 0 dBm	120 mA
	Wi-Fi / BT Rx and listening	80 ~ 90 mA
	Association sleep pattern (by light-sleep)	0.9 mA@DTIM3, 1.2 mA@DTIM1
Modem-sleep mode	The CPU is powered on.	Max speed: 20 mA Normal: 5 ~ 10 mA Slow speed: 3 mA
Light-sleep mode	-	0.8 mA
Deep-sleep mode	The ULP-coprocessor is powered on.	0.5 mA
	ULP sensor-monitored pattern	25 $\mu$ A @1% duty
	RTC timer + RTC memories	20 $\mu$ A
Hibernate mode	RTC timer only	2.5 $\mu$ A

	ESP32 Deepsleep	D21 (Ard. Zero) Suspend	STM32-L4 Stop-2
Power	7 $\mu$ A	150 $\mu$ A	2 $\mu$ A
CPU	aus	standby	standby
Memory	aus	standby	standby
Interrupts aktiv	😞	😞😞	😞
RTC mit Wakeup	😞	😞	😞
Timer aktiv	😞	😞	😞
Wakeup Delay	250 ms	< 100 $\mu$ s	5 $\mu$ s
ULP-Prozessor	😞	-	-

9

### Fonction delay = boucle d'attente

- Fonction delay, définie dans wiring.c :

```

void delay(unsigned long ms)
{
    uint32_t start = micros();
    while (ms > 0) {
        yield();
        while ( ms > 0 && (micros() - start) >= 1000) {
            ms--;
            start += 1000;
        }
    }
}

```
- Et yield() est défini dans hooks.c.

```

static void __empty() {
    // Empty
}
void yield(void) __attribute__ ((weak, alias("__empty")));

```

10

**Boucles d'attente active**

- Boucle d'attente :
- Ex : inversion d'état de la LED toute les secondes.
  - 100000 tours de boucles pour attendre, environ 10 cycles par tour,
  - Inversion d'état de la LED : opération logique avec le port de sortie, environ 3 cycles
  - Efficacité :  $3 / ((100000 * 10) + 3) = 0.00000125$  soit 0,99999875 % de la puissance électrique est consommée pour attendre !
- Idem pour la prise en compte des entrées : Interrogation en continu...
- Difficulté d'effectuer une autre tâche en même temps

```
for(int i = 0 ; i < 100000 ; i++)
    volatile void nop();
```


Polling

11

**Le concept "interruption"**

- [wikipedia] Arrêt temporaire de l'exécution normale d'un programme informatique par le microprocesseur afin d'exécuter un autre programme (appelé routine d'interruption ou ISR Interrupt Service Routine).
- Différentes sources d'interruptions
- Eventuellement des priorités
- Les occurrences ne sont pas prévisibles (difficile à debugger/simuler)
- Les interruptions constituent une tâche épisodique (ne pas se substituer au prog. Principal)

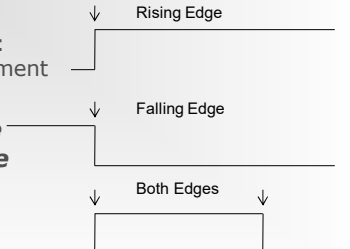
12




## Sources des Interruptions

200ms entre deux frappes touches.  
 240MHz ->  $48 \cdot 10^6$  cycles  
 Ou alors mise en veille !

- Saisie (de la part d'un humain = lent, possibilité de mise en veille entre deux actions)
- Sources possibles d'interruption :
  - Timer (3 comparateurs par core pour l'ESP32)
  - Réception ou fin de transmission
  - Performance monitoring (interne)
  - Changement d'état GPIO
  - Fin de conversion AD
- Remarques :
  - Déclenchement sur front (Edge Sensitive):  
 Déclenchement sur un front actif (changement de 0 à 1 ou de 1 à 0)
  - ***The ESP32 offers only edge sensitive interrupts, and only one edge may be detected at a time.***




13



## Caractéristiques des interruptions

- Hiérarchie des interruptions (c'est selon les uP...)
  - Les interruptions sont dites **masquables** si on peut décider de ne pas les prendre en compte (désactivation par un bit de contrôle individuel)
  - Plusieurs sources peuvent déclencher la même interruption (par ex. chaque bit d'un port 8 bits peut provoquer la même interruption). Dans ce cas (**partage d'interruption**), un drapeau (flag) est positionné (set) dans une registre dédié pour identifier la source
  - Une interruption non traitée est mémorisée. Une seconde interruption de même source alors que la première n'est pas traitée est perdue...


14



## Interrupt Service Routines

- Une ISR n'est pas prédictible
- Une ISR doit être courte et d'exécution rapide
  - On ne fera donc aucun calcul compliqué et aucun appel à des fonctions longues comme un affichage sur un écran LCD... **et pas d'écriture vers la liaison série !**
  - Si le programme principal exploite les interruptions, il ne faut évidemment pas désactiver trop souvent les interruptions
- On peut communiquer avec l'ISR par des variables globales.
- ESP32 = multicore... Echange d'information entre la routine d'interruption et le programme principal
  - Protection par une section critique
  - Entrer dans la section critique
  - Modifier les valeurs
  - Sortir de la section critique

15



## Programmer avec des interruptions

- Avec instructions assembleur
  - On place le vecteur d'interruption (adresse de l'ISR) à la bonne place (dans un tableau de vecteurs d'interruption)
  - Instructions pour valider/dévalider : DINT, EINT
- Mais on programme en langage C ou C-like !
- Utilisation de commandes intrinsèques (intrinsic)
  - *The `_DINT()` and `_EINT()` operations are intrinsic functions defined in the IAR C Compiler.*

```
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  ...
}
```
- Ou appel d'une fonction utilitaire du framework reliant la routine à la source d'interruption
 

```
attachInterrupt(numero, ISR, mode);
```

16



## Interruption avec le framework Arduino et l'ESP32

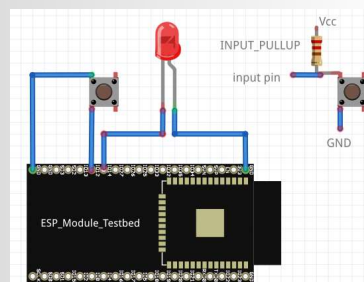
- `attachInterrupt(numero, ISR, mode);`
- Cette fonction prend 3 arguments :
  - le numéro d'interruption externe (attention limité selon plateforme)
  - la fonction à appeler quand l'interruption survient
  - la condition selon laquelle l'interruption survient :  
 RISING : l'interruption est déclenchée quand la broche concernée passe de LOW à HIGH, ou FALLING : de HIGH à LOW.
- ISR est la fonction desservant cette interruption (Interrupt Service Routine). Toujours même prototype : une fonction qui ne renvoie rien et qui ne prend aucun argument :  

```
void monTraitementISR(void) { ... }
```
- `detachInterrupt(...)` permet de déconnecter l'ISR de la source d'interruption. Son prototype est le suivant :
- `detachInterrupt(numero);`

17

## Exemple

- A faire : Un bouton allume la LED.
- `digitalPinToInterrupt(interruptPin)` permet de s'affranchir de la plateforme



```
byte ledPin = 14; // LED pin
byte interruptPin = 12; // pin that is attached to interrupt
volatile byte state = LOW; // hold the state of LED

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink,
    CHANGE);
}
```

18

## Suite du code

```
byte ledPin = 14; // LED pin
byte interruptPin = 12; // pin that is attached to interrupt
volatile byte state = LOW; // hold the state of LED

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink,
  CHANGE);
}

void loop()
{ }

/* interrupt function toggle the LED */
void blink()
{
  state = !state;
  digitalWrite(ledPin, state);
}
```

19

## Autre possibilité...

```
byte ledPin = 14; // LED pin
byte interruptPin = 12; // pin that is attached to interrupt
volatile byte state = LOW; // hold the state of LED

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink,
  CHANGE);
}

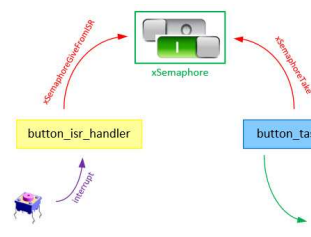
void loop()
{
  digitalWrite(ledPin, state);
}

/* interrupt function toggle the LED */
void blink()
{
  state = !state;
}
```

20

## Les interruption peuvent survenir à n'importe quel moment !

- Echange d'informations entre la routine d'interruption et le programme principal = variables globales
- Si nécessaire, protection par une section critique
  - Entrer dans la section critique, modifier les valeurs, sortir de la section critique
- Utilisation de ressources comme les Mutex ou les Semaphores



```

if (interruptCounter > 0) {
    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
    portEXIT_CRITICAL(&timerMux);

    // Interrupt handling code
}
    
```

21

## Comment déclarer une routine d'interruption

- Une ISR est une fonction.
- Il faut associer l'adresse de la fonction au vecteur d'interruption

MSP430  
Commande C et  
typage  
intrinsèques

```

#pragma vector=TIMER1_VECTOR
#pragma vector=TIMER0_VECTOR
__interrupt void Timer_A (void)
{
    ...
}
    
```

ESP32  
Typage  
intrinsèque et  
fonction utilitaires  
du framework

```

void IRAM_ATTR myISR(){
    ...
}

attachInterrupt(0, myISR, RISING);
    
```

22

## Résumons

- Sauf cas particulier, ne plus utiliser le polling pour détecter des actions extérieures
- Mais :
  - Difficile à formaliser
  - Difficile à debugger
- Ne pas faire :
  - Declaration de « grosses » variables dans l'ISR  
-> *Stack overflow*
  - Calcul longs ou transmission série dans l'ISR

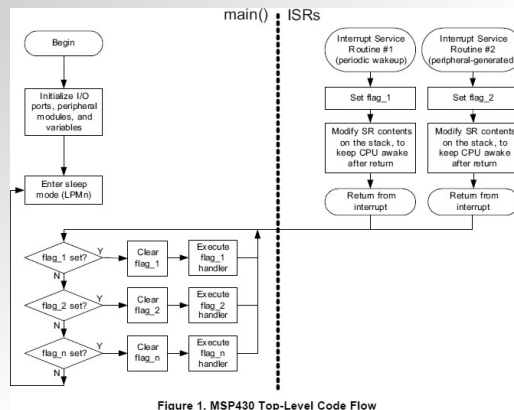


Figure 1. MSP430 Top-Level Code Flow

23

## Le mode interruptif autorise la mise en veille

- Si des interruptions sont possible, on peut éviter l'attente active
- Comment mettre le processeur en veille :
  - Assembleur : En positionnant certains bits dans certains registres
  - Langage C ou C-like : fonction intrinsèques  

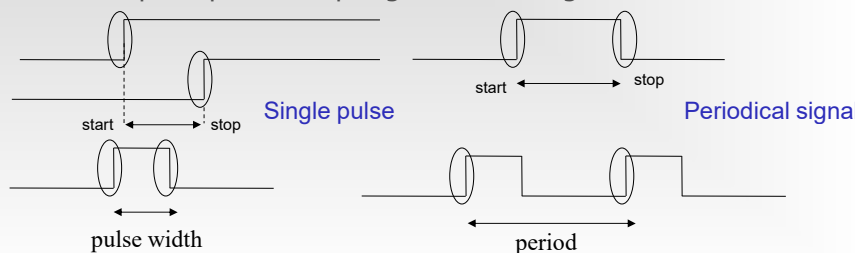
```
void __low_power_mode_3(void);
```
  - Fonction(s) du framework  

```
esp_deep_sleep_start();
```
- Prévoir un réveil possible : si l'exécution de code est inhibé (CPU=HALT), seule une interruption peut faire sortir le microcontrôleur de veille (issue d'un composant périphérique actif ou d'un évènement extérieur)

24

## Utilisations possibles d'un Timer

- Un **timer** est un périphérique matériel permettant de compter des événements. Si ces événements sont des tops d'horloge, on détermine une **durée**.
  - Un timer est généralement un circuit indépendant de l'ALU, celui-ci n'est pas mobilisé.
  - Souvent, un timer peut être déclenché/arrêté par un événement extérieur (un niveau sur une broche)
  - Le timer peut éventuellement piloter directement une broche pour par exemple générer un signal



25

## Timers pour l'ESP32

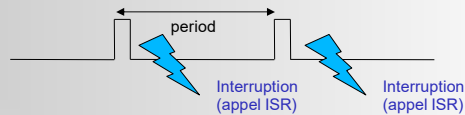
- *ESP32 chip contains two hardware timer groups, each containing two general-purpose hardware timers.*
- *They are all 64-bit generic timers based on 16-bit prescalers and 64-bit auto-reload-capable up/down counters.*
- Une dizaine de fonctions utilitaires dans le portage du framework Wiring/Arduino

26

## Utilisations possibles d'un Timer

- Sans relation avec "l'extérieur", un Timer permet d'exécuter des routines à périodes fixes, ou à un moment précis

- PWM
- Audio

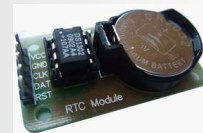


- Attention :  
ne pas confondre Timer et RTC

Une RTC (Real Time Clock) est souvent une horloge hardware indépendante avec une batterie de backup.

Un timer s'arrête si l'alimentation est coupée

L'ESP32 dispose d'une RTC, mais pas de batterie de backup



27

## Timers et horloges

- Principe pour économiser de l'énergie : désactiver des fonctions internes et ralentir l'horloge
- Il faut savoir quelle horloge est utilisée par le timer (éventuellement configurable)

Car arrêt complet de certaines horloges dans certains modes de veille

- **Divisions de fréquence intermédiaires possibles (prescaling)**

- ESP32 : Classiquement, l'horloge faisant évoluer le timer est cadencée à 80 Mhz. Si prescaling avec une division de 80, le « pas » ou *tick* du timer est la microseconde
- MSP430 : Avec un Quartz et le bon prescaling, on peut avoir un *tick* d'une seconde.

- définir le mode de comptage (UP, DOWN ou autre)

28

## Fonctions utilitaire pour le timer (specifique ESP32)

- Dans l'ordre :
  - Créer le timer (parametres : num, pré-division, sens)
    - `timer = timerBegin(0, 80, true);`
- Attacher la routine d'interruption au timer
  - `timerAttachInterrupt(timer, &onTimer, true);`
  - Remarquer la différence de syntaxe avec `AttachInterrupt`
- Positionner la valeur de déclenchement
  - `timerAlarmWrite(timer, 1000000, true);`
  - **Exprimé en microseconde uniquement si la prédivision est 80**
  - Combien d'alarmes par timer ?
- Lancer le comptage
  - `timerAlarmEnable(timer);`

29

## Les fonctions de mise en veille : light sleep

- Light Sleep
 

```
void setup()
{
    Serial.begin(115200);
    Serial.println("setup");
}
void loop()
{
    esp_sleep_enable_timer_wakeup(5000000); //5 seconds
    int ret = esp_light_sleep_start();      // bloquant !
    Serial.print("light_sleep:");
    Serial.println(ret);
}
```
- Attention, quelques bug (derègle la RTC, ...)

30

## Les fonctions de mise en veille : deep sleep

```
#define uS_TO_S_FACTOR 1000000 /* Conversion factor fo
                                micro seconds to seconds */
#define TIME_TO_SLEEP 3        /* Time ESP32 will go to
                                sleep (in seconds) */

void setup()
{
    pinMode(LED_BUILTIN,OUTPUT);
    delay(500);
    digitalWrite(LED_BUILTIN,HIGH);
    delay(3000);
    digitalWrite(BUILTIN,LOW);
    esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP *
                                uS_TO_S_FACTOR);

    esp_deep_sleep_start();
}

void loop()
{ }
```

- La sortie de veille deep sleep est assimilé à un redémarrage

31

## La sortie de veille

- Fonctions identiques, peu importe le mode (light sleep ou deep sleep)
- Le framework propose :
  - Un fonction pour le reveil via le timer (interne)  
**esp\_sleep\_enable\_timer\_wakeup(microsec)**
  - Une fonction si le reveil est provoqué par un GPIO unique
  - Une fonction si le reveil peut-être provoqué par des sources multiples

you can enable the wake up if the specified pin (*gpio\_num*) changes its status (*level*).

You can only use the pins [with RTC function](#) (0, 2, 4, 12-15, 25-27, 32-39) and the possible levels are 0 (= low) or 1 (high). If, for example, you want to wake up the chip from *deep sleep* if **pin 4** has a **low level**, you'll write:

```
esp_sleep_enable_ext0_wakeup(4, 0);
```

32



## La sortie de veille

Si possibilités multiples de sortie de veille :

`esp_sleep_enable_ext1_wakeup(bitmask, mode)`

This function accepts two arguments:

- A bitmask of the GPIO numbers that will cause the wake up;
- Mode: the logic to wake up the ESP32. It can be:
  - `ESP_EXT1_WAKEUP_ALL_LOW`: wake up when all GPIOs go low;
  - `ESP_EXT1_WAKEUP_ANY_HIGH`: wake up if any of the GPIOs go high.

Pour savoir exactement quell(s) GPIO a (ont) causé(s) le reveil :

`uint64_t esp_sleep_get_ext1_wakeup_status()`

On recupere un masque binaire indiquant le(les) GPIO(s) incriminé(s), ou 0 si c'est une autre raison.

33

## Sortie de veille

- Comme la sortie d'un *deep sleep* agit comme un reset, il existe un moyen pour savoir si reset ou sortie de veille

`esp_sleep_wakeup_cause_t esp_sleep_get_wakeup_cause()`

```
56 typedef enum {
57     ESP_SLEEP_WAKEUP_UNDEFINED,    //!< In case of deep sleep, reset was
58     ESP_SLEEP_WAKEUP_EXT0,        //!< Wakeup caused by external signal
59     ESP_SLEEP_WAKEUP_EXT1,        //!< Wakeup caused by external signal
60     ESP_SLEEP_WAKEUP_TIMER,       //!< Wakeup caused by timer
61     ESP_SLEEP_WAKEUP_TOUCHPAD,    //!< Wakeup caused by touchpad
62     ESP_SLEEP_WAKEUP_ULP,         //!< Wakeup caused by ULP program
63 } esp_sleep_wakeup_cause_t;
```

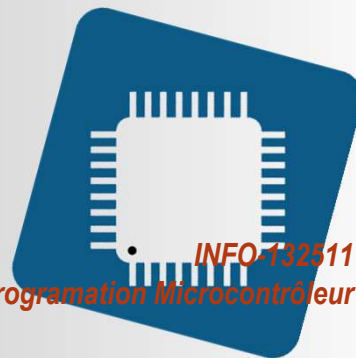
`uint64_t esp_sleep_get_ext1_wakeup_status()`

34

## *Développer des applications économes en énergie*

- Laisser le plus souvent le processeur en veille (donc savoir gérer des connexions réseau non persistantes)
- Alimenter électriquement les périphériques (interne et externes) que si nécessaire
- Utiliser au maximum les périphériques intégrés (concevoir des systèmes sans « glue logic »)
- Programmation :
  - Limiter les calculs. Par exemple, utiliser des tables de correspondance
  - Pas d'attente active (pooling) -> interruptions

35



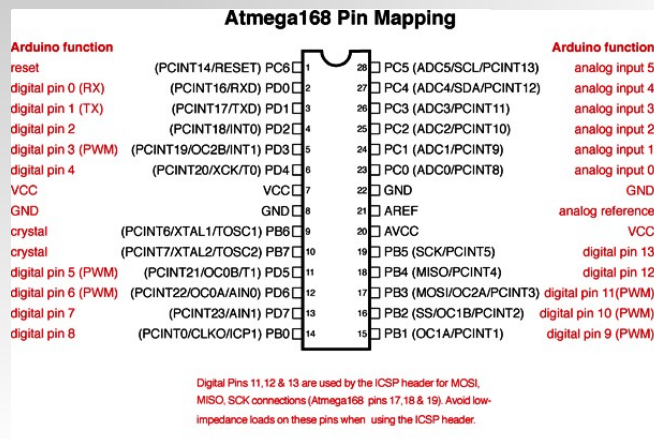
*Objets Connectés : Programation Microcontrôleur*

Microprocesseur :  
Direct I/O

36

## Direct I/O Access

- Sur la plupart des "petits" microcontrôleurs :
  - Access direct aux registres de configuration, aux registres de sortie (R/W) et d'entrée (Read-Only)
  - Regroupement des GPIO par 8 (contrôle via un octet)



37

## Utilisation de GPIO faisant partie d'un même PORT

- Ex. configurer les broches 3, 5 et 7 de l'Arduino (PD3, PD5 et PD7) comme des sorties, mettre la broche 3 à l'état haut

Avec le framework

```
pinMode(3, OUTPUT);
pinMode(5, OUTPUT);
pinMode(7, OUTPUT);
```

Sachant que :

```
void pinMode(byte pin, byte mode)
{
  byte bit =
    digitalPinToBitMask(pin);
  byte port =
    digitalPinToPort(pin);
  ...
}
```

Ecrire :

```
digitalWrite(3, HIGH);
```

Sans framework

```
DDRD = 0b10101000;
```

or

```
DDRD = 0xA8;
```

or

```
DDRD |= 1<<PD7 | 1<<PD5 | 1<<PD3;
```

```
PORTD = PORTD | 0x04;
```

38

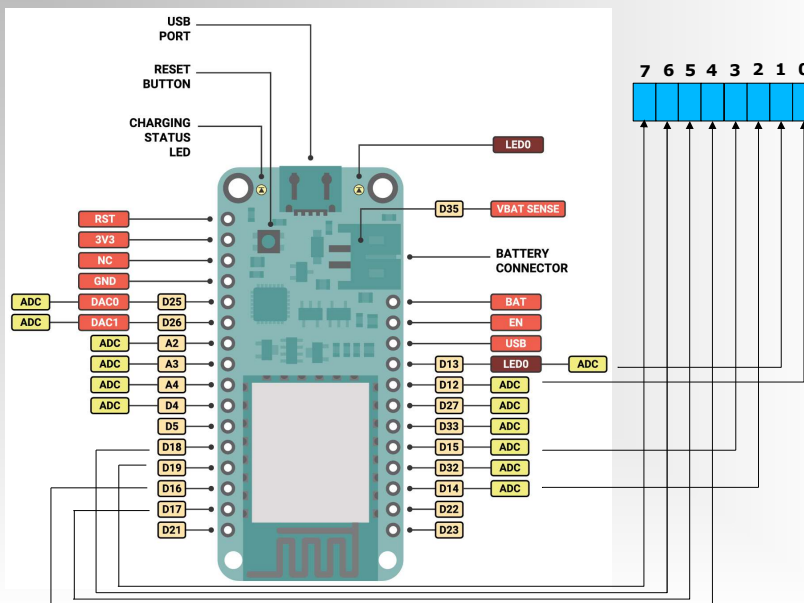
## Accès direct aux registres avec ESP32

- Registres de 32 bits
- Des GPIO avec des restrictions :
  - GPIO 34, 35, 36 et 39 : uniquement des entrées
  - GPIO 6,7,8,9,10 et 11 : utilisés pour communiquer avec la mémoire Flash interne (ne pas utiliser)
  - GPIO 22 et 23 : Liaisons I2C, donc ne pas utiliser si périphériques I2C
  - GPIO 25 et 26 : Sortie analogique (DAC=PWM hardware)
- Pour faciliter le développement (portage de code développé avec des processeurs disposant d'I/O sur des port 8 bits) :
  - Utiliser les GPIO 12 à 19 (8 bits consécutifs = 1 « octet »)
  - Accès direct aux registres :


```
REG_WRITE(GPIO_ENABLE_REG, BIT15);
REG_WRITE(GPIO_OUT_REG, BIT15);
```

39

## Raccordement des GPIO 12 à 19



40



## Lecture / Ecriture directe de registres pour l'ESP32

- Comment mettre à 1 un seul bit d'un Port (bit 3)
 


```
PORTD = PORTD | 0x04;

val = REG_READ(GPIO_OUT_REG, BIT15);
REG_WRITE(GPIO_OUT_REG, val | BIT15);

Ou bien
Val = REG_READ(GPIO_OUT_REG, BIT15);
REG_WRITE(GPIO_OUT_REG, val | (0x04 << 12));
```
- Mais attention : dual-Core !
 

```
Task A reads the reg value
Task A or's the value with (0x04 << 12)
***CONTEXT SWITCH***
Task B reads the reg value
Task B ors the reg value with (0x05 << 12)
Task B writes the value back
***CONTEXT SWITCH***
Task A writes the reg value back
```

41



## Lecture / Ecriture directe de registres pour l'ESP32

- Pour que l'opération reste **atomique**, utiliser des registres spéciaux :
- GPIO\_OUT\_W1TS\_REG  
mise à 1 « cablé » dans le microprocesseur  
(*Write One To Set*)
- GPIO\_OUT\_W1TC\_REG  
mise à 0 « cablé » dans le microprocesseur  
(*Write One To Clear*)
- Donc, mettre à 1 un seul bit d'un Port (bit 3) :  
(faisable aussi avec DigitalWrite)
 

```
REG_WRITE(GPIO_OUT_W1TS_REG, (0x04 << 12));
```
- Interet : mise à 1 de plusieurs bits  
simultanément :
 

```
REG_WRITE(GPIO_OUT_W1TS_REG, (0x1C << 12));
```

42