REDS
Reconfigurable & embedded Digital Systems

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Architecture and Drivers for Smartphones
Power Management

Cours APS

Salvatore Valenza

Version 1.0 (2012-2013)

Plan

- Petite présentation de la batterie
- Power states dans les téléphones portables.
- Gestion de l'alimentation extérieure au SoC.
- Gestion de l'alimentation intérieure au SoC.
- Ecriture de logiciel pour optimiser la vie de la batterie

La batterie

Energies dans les systèmes portable

Les téléphones portables sont alimentés par un système à énergie limitée.

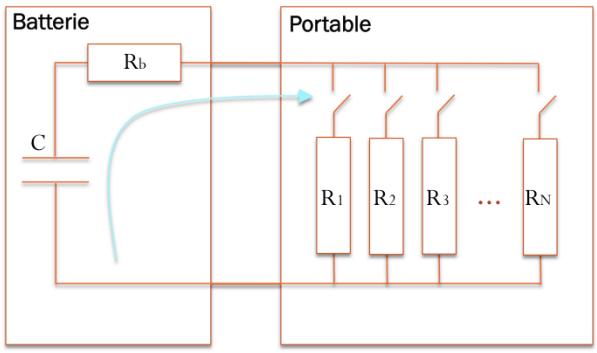
Pour la plupart du temps les systèmes portables attendent que les utilisateurs touchent le clavier ou qu'un message de paging du réseau soit reçu pour instaurer un appel.

La batterie donne courant au système et le temps de « vie » de la batterie dépend du courant fourni au système.

Il est évident qu'on doit optimiser la consommation du courant pour augmenter la durée de « vie » de la batterie le plus longtemps possible.

Durée de vie de la batterie

(overview extrêmement simplifiée)



R_b: résistance interne de la batterie

R_i: résistance liée a une fonctionnalité matériel ou logiciel du portable

$$R(t) = \frac{\prod_{i=1}^{N}[\bar{E}_i + E_i R_i(t)]}{\sum_{i=1}^{N} \bar{E}_i R_i(t)} \quad avec \quad E_i = \begin{cases} 1 & si\ i\ est\ active \\ 0 & si\ i\ est\ inactive \end{cases}$$

En supposant, pour simplifier, que le résistances R_1, R_2, … R_N soit constantes en fonction du temps, on a que la résistance du portable est

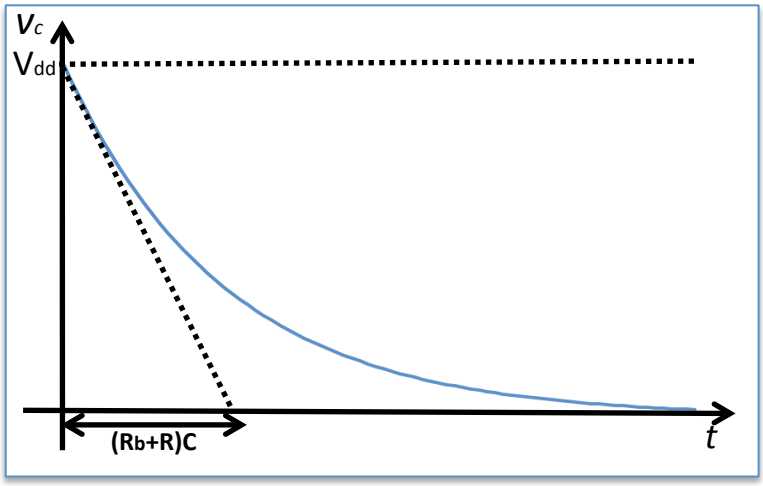
$$R = \frac{\prod_{i=1}^{N}(\bar{E}_i + E_i R_i)}{\sum_{i=1}^{N} \bar{E}_i R_i} \quad avec \quad E_i = \begin{cases} 1 & si\ i\ est\ active \\ 0 & si\ i\ est\ inactive \end{cases}$$

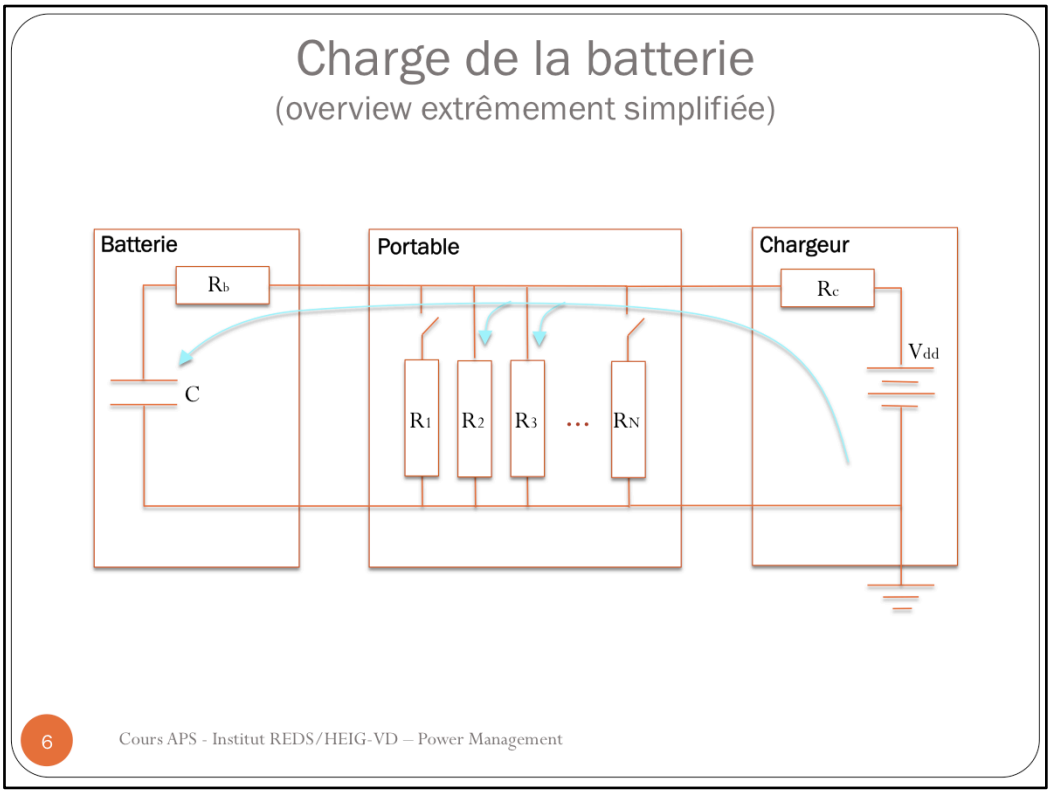
nous avons alors

$$v_c(t) + R_b i_c(t) + R i_c(t) = 0 \overset{et\ donc}{\Longrightarrow} v_c(t) + (R_b + R) C \frac{dv_c}{dt}(t) = 0$$

c'est à dire

$$\begin{cases} v_c(t) = v_c(0)\, e^{-\frac{t}{(R_b+R)C}} \\ v_c(0) = V_{dd} \end{cases} \Rightarrow v_c(t) = V_{dd}\, e^{-\frac{t}{(R_b+R)C}}$$





En supposant d'attacher le chargeur à t_0 (quand la tension sur C est V_x), que $R_b << R$ et que $R_c << R$, on peut voir que

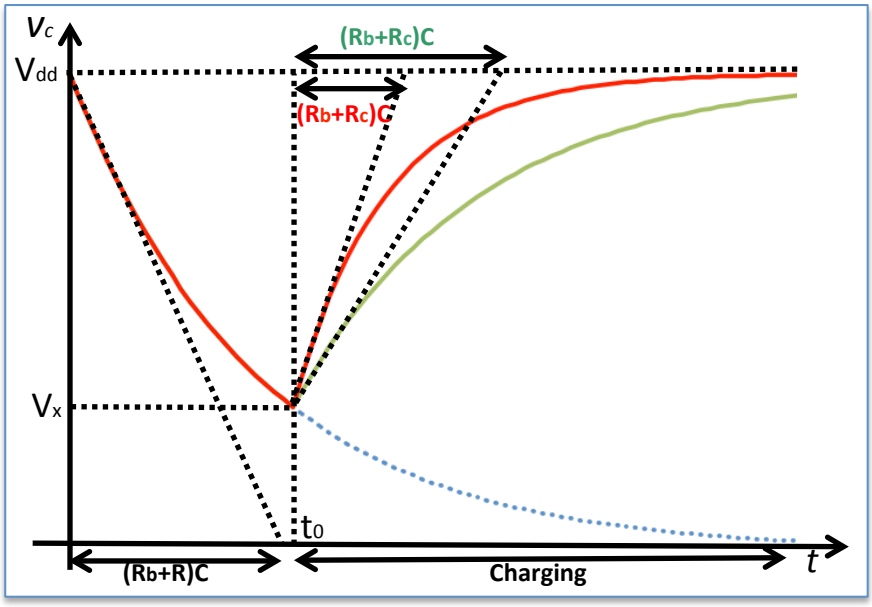
$$v_c(t) + R_b i_c(t) \cong V_{dd} - R_c i_c(t)$$

alors

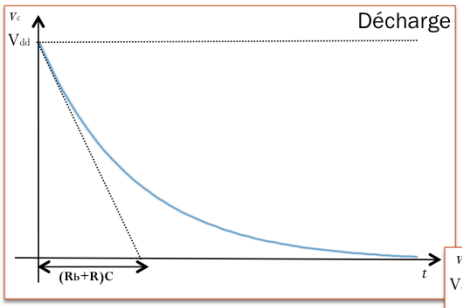
$$v_c(t) + (R_b + R_c)C\frac{dv_c}{dt}(t) = V_{dd}$$

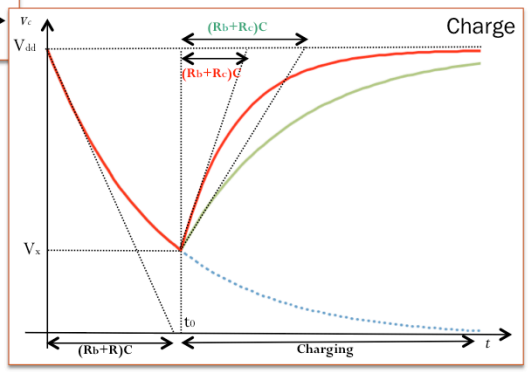
et donc

$$\begin{cases} v_c(t) = Ae^{-\frac{t-t_0}{(R_b+R_c)C}} + B \\ v_c(t_0) = V_x \\ \lim_{t\to+\infty} v_c(t) = V_{dd} \end{cases} \Longrightarrow \begin{cases} A = V_x - V_{dd} \\ B = V_{dd} \end{cases} \Rightarrow v_c(t) = (V_x - V_{dd})e^{-\frac{t-t_0}{(R_b+R_c)C}} + V_{dd}$$



Charge / décharge de la batterie





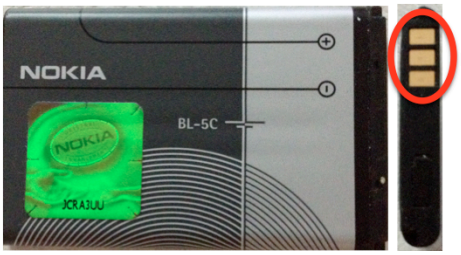
$$\lim_{N\to+\infty} R = \lim_{N\to+\infty} \frac{\prod_{i=1}^{N}(\bar{E}_i + E_i R_i)}{\sum_{i=1}^{N} \bar{E}_i R_i} = 0$$

$$\lim_{R_{i0}\to 0} R = \lim_{R_{i0}\to 0} \frac{\prod_{i=1}^{N}(\bar{E}_i + E_i R_i)}{\sum_{i=1}^{N} \bar{E}_i R_i} = 0$$

Algorithme de chargement et mesure de la durée de vie restante

Dans le systèmes réels les fonctions de chargement et de mensuration de la durée de vie restante pour la batterie, sont beaucoup plus complexes et propriétaires du OAM.

Le batterie aussi dispose de données, auxquelles on peut accéder avec des PINs particuliers. Par exemple on peut lire la résistance interne, le « product id », des informations sur la température de la batterie, etc.

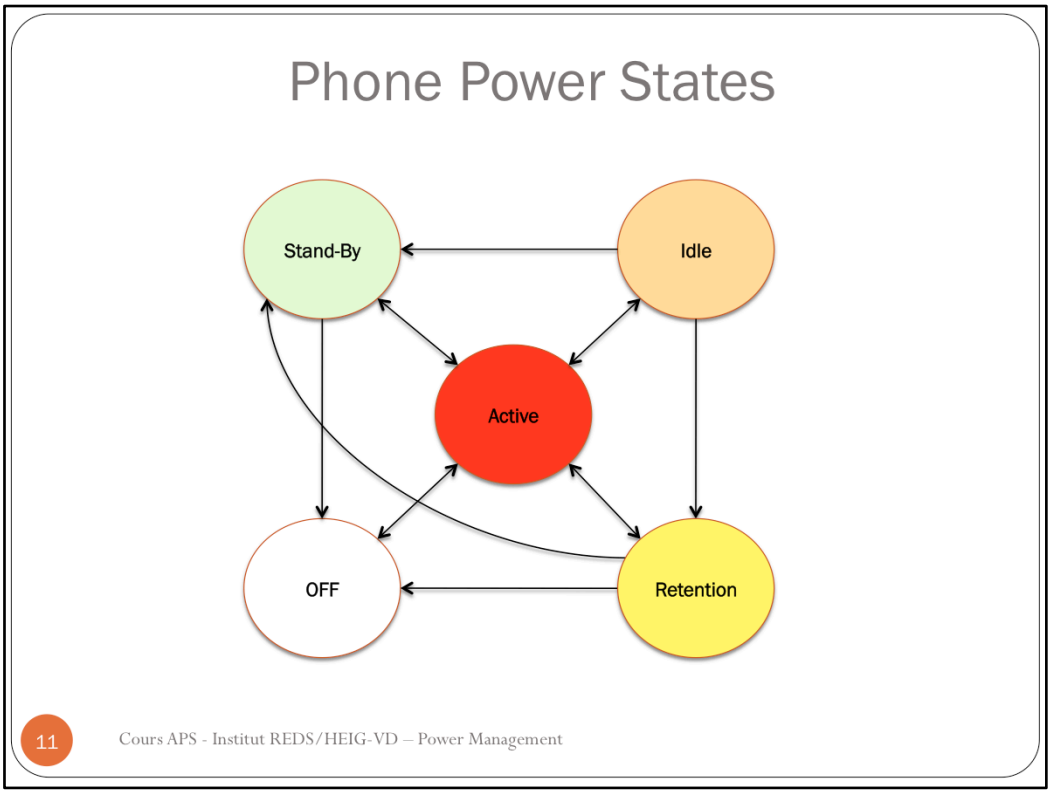


Info sur smart batteries ici: http://batteryuniversity.com/learn/article/the_smart_battery

Power States

Que fait le portable pour la majorité du temps?

En tenant compte de différentes états d'activité du portable et de ses composants matériel et logiciel, on peut optimiser la durée de vie de la batterie.



OFF – Le passage d'un composant matériel à cet état est le résultat de la suppression de l'alimentation du composant. Les données et l'état sont perdus. Le composant ne aura pas la capacité de répondre à des événements extérieur.

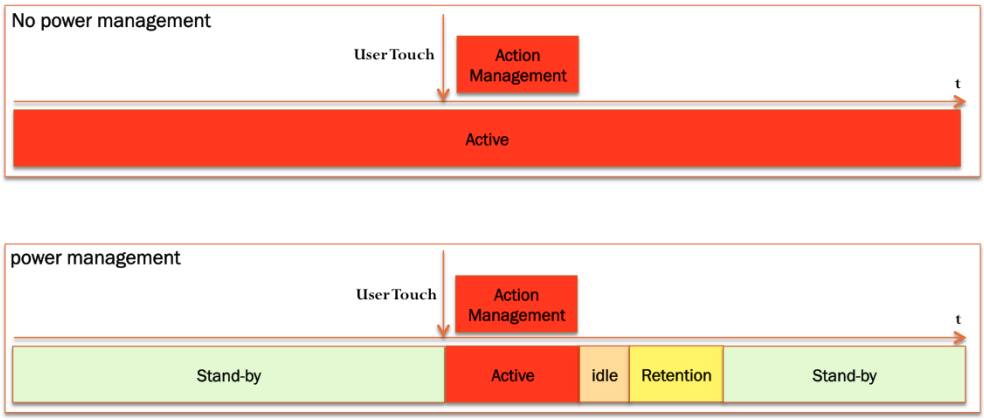
Active – Un composant matériel dans cet état est en train de traiter des tâches, des données et des événements sont générés et reçus à partir des portes d'input. L'exigence sur les ressources énergétiques est élevée pour garantir que ce traitement ait lieu. Il est peut-être en attente d'exécuter des tâches ou de répondre à des demandes de traitement.

Idle - Il s'agit généralement d'un état transitoire. Le composant matériel a terminé le traitement d'une tâche et aucune demande pour des services supplémentaires a été placé. Aucun de composants connectés agit sur les inputs, ni de nouveaux événements internes sont générés. Le composant a la capacité de répondre à de nouveaux événements et de les traiter. Son état et ses données sont conservés.

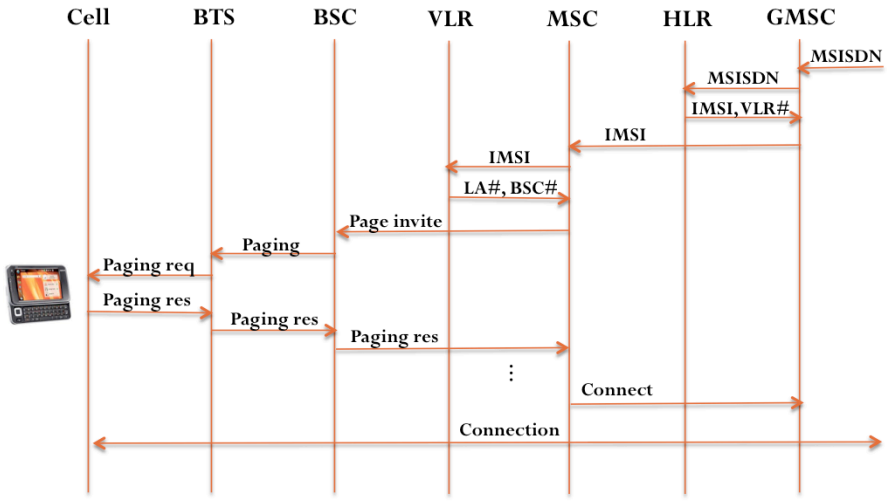
Rétention – Dans cet état, les exigences en matière de ressources énergétiques sont réduites à celles nécessaires pour préserver les données du composant, l'état interne et la capacité à détecter des événements externes ou de générer des événements internes. Le composant n'exécute pas de tâches actives, il n'est pas impliqué en transaction des données et il n'y a pas des traitements internes en cours.

Standby - Dans cet état de puissance, le composant matériel peut ne pas avoir la capacité de conserver les données et l'état, mais le logiciel de control de l'énergie peut les restaurer quand on transit vers le mode actif. Les exigences en matière de ressources énergétiques peuvent être considérablement réduit. Le composant peut conserver une certaine capacité à détecter et servir des événements extérieurs avec un temps de réponse lent.

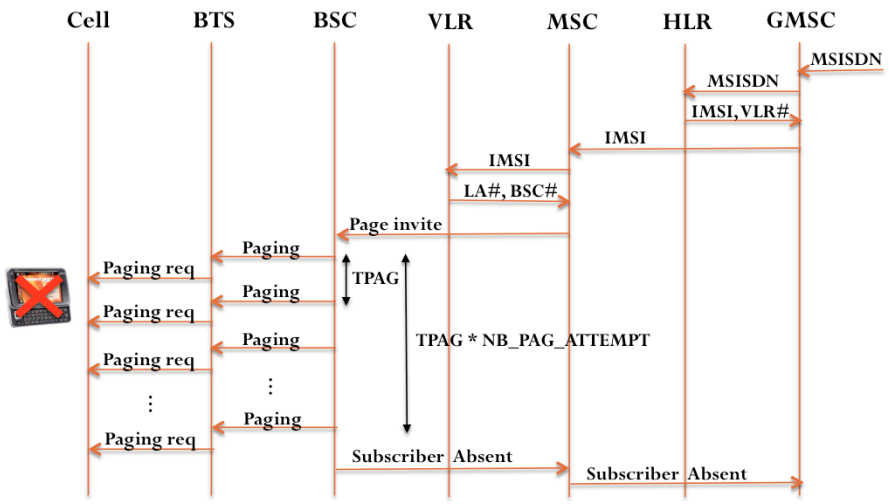
State Management – User touch



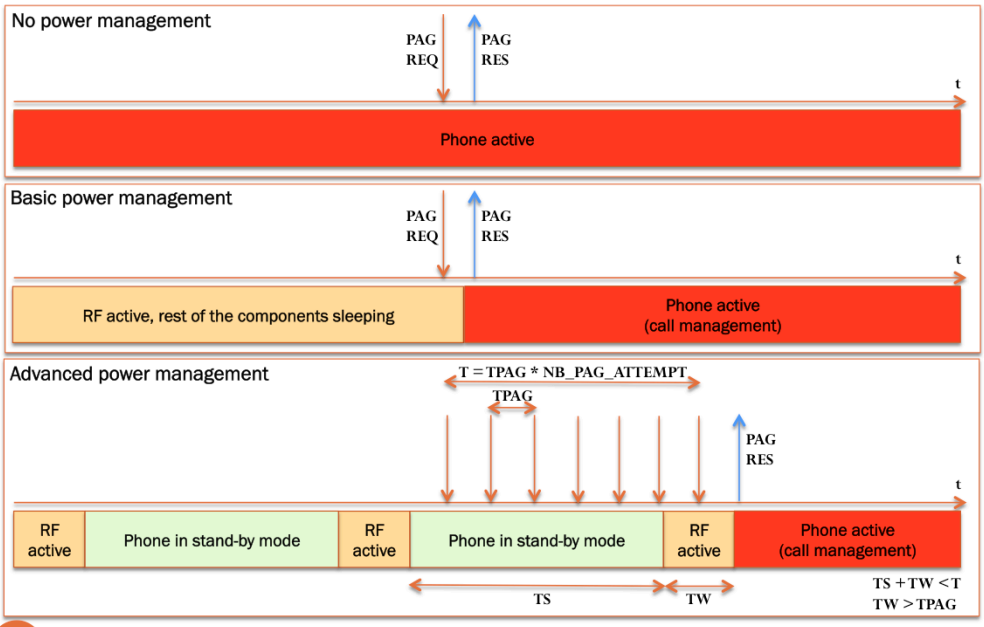
State Management - Paging (1/3)



State Management - Paging (2/3)



State Management – Paging (3/3)



Internal and External Power Management

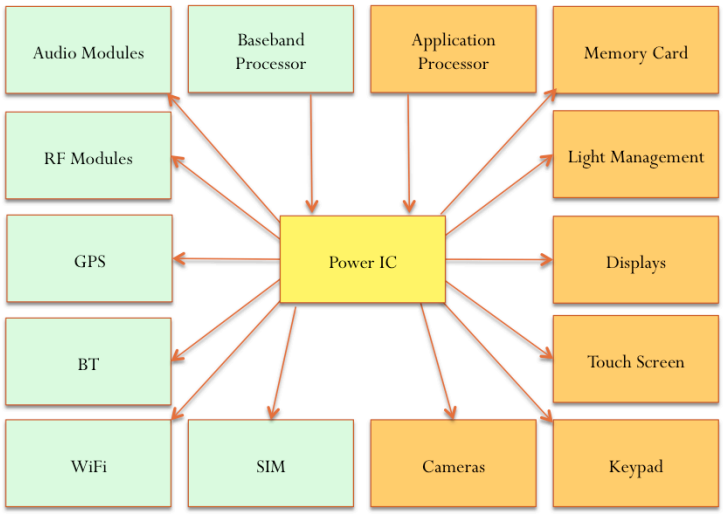
On peut séparer 2 aires de gestion de l'énergie dans un système portable:

1. Intérieur: les SoCs gèrent leur propre énergie (AP et BP):
   - Gestion des états de pouvoir du SoC
   - Optimisation de l'écriture du logiciel

2. Extérieur: les SoCs gèrent l'énergie des périphériques qui leurs sont connectées:
   - Gestion de l'alimentation des périphériques (à travers le Power IC)
   - Gestion des états du système portable

External Power Management

Power IC


Audio Modules
Baseband Processor
Application Processor
Memory Card
RF Modules
Light Management
GPS
Power IC
Displays
BT
Touch Screen
WiFi
SIM
Cameras
Keypad

Internal Power Management

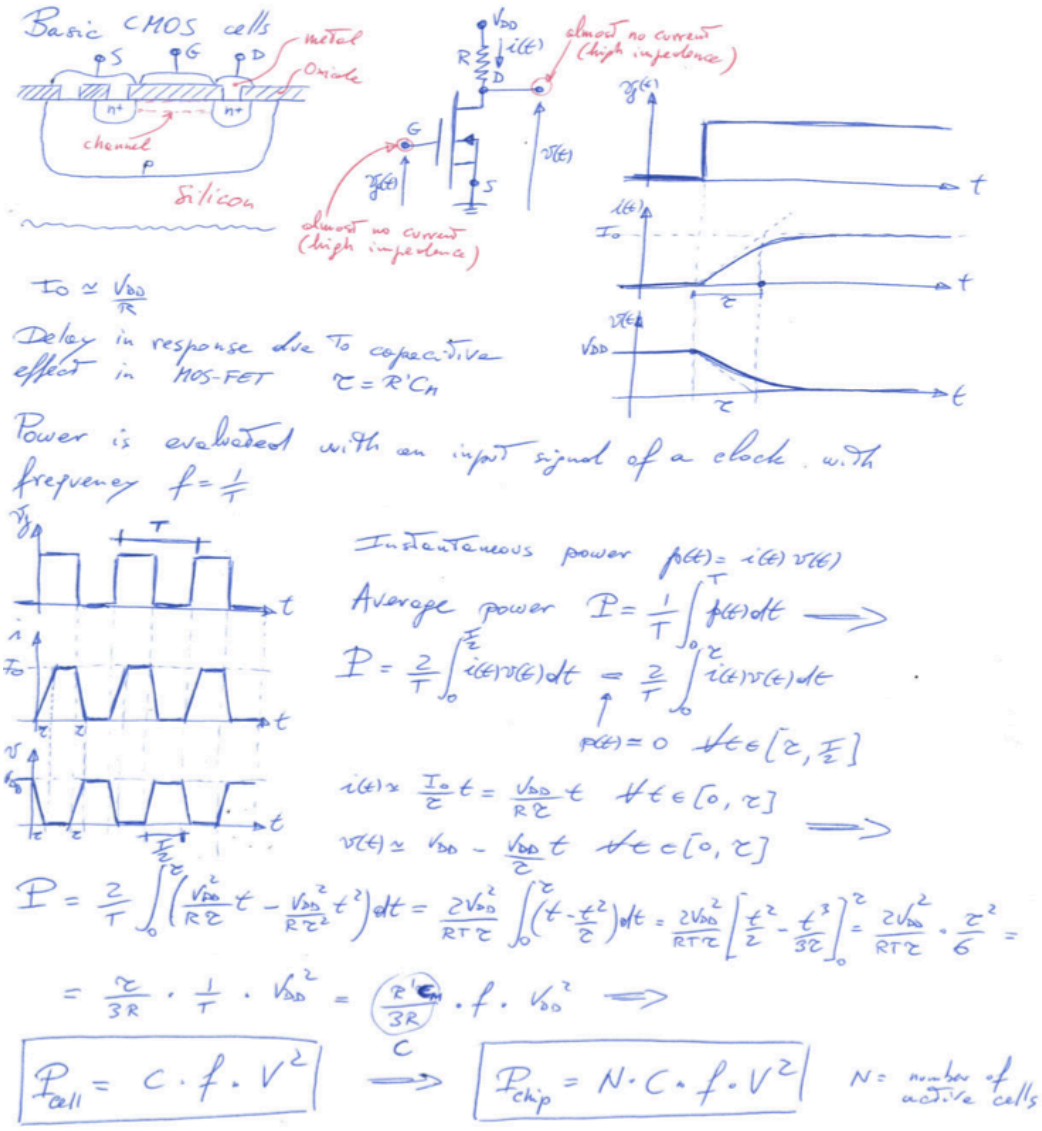
SoC Power Consumption

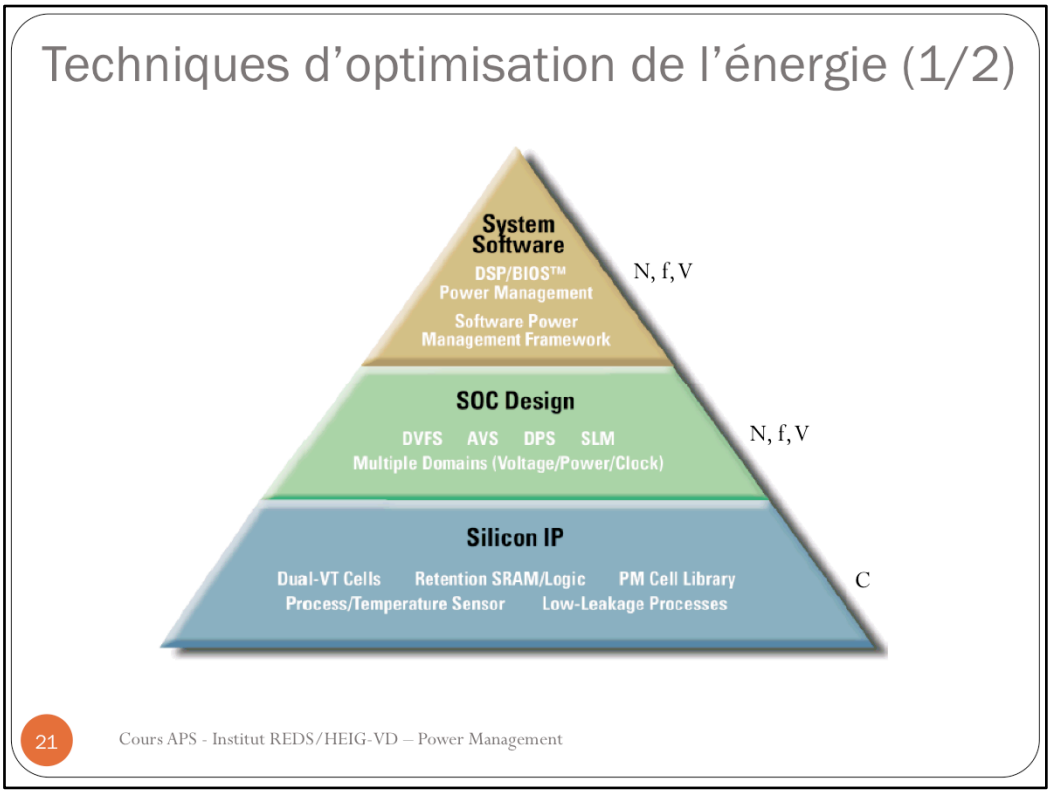
Depends on HW
(only producer can tune it)

$$P = C * N * f * V^2$$

These can be controlled at run time

P = Average Power
f = Clock frequency
V = Chip Voltage
C = Equivalent Capacitance of the system
N = Number of cells used on the chip





With the convergence of new computing, communication and entertainment applications on wireless handsets, power demands are increasing rapidly, yet the capacity of batteries cannot keep up.

Wireless mobile devices are approaching an impasse. With the convergence of new computing, communication and entertainment applications on wireless handsets, power demands are increasing rapidly, yet the capacity of batteries cannot keep up. At the same time, consumers want sleek, compact mobile devices they can slip into a pocket. Integration at the chip level--often combining multiple processing cores in the same device--and smaller, submicron fabrication processes help to reduce the size of wireless handsets while enabling added functionality. Unfortunately, smaller submicron processes exacerbate the problem of static leakage power. Manufacturers of wireless handsets and other mobile devices are challenged to reduce power consumption while enhancing system performance. In other words, do more for less.

Such rapid integration both in silicon and software space is posing a significant design problem for power management engineers. Power management no longer remains a hardware-only problem, rather it has become a system problem and being addressed by all engineers involved in system design process. Power management decisions are being taken at both hardware and software level. Techniques are invented and deployed in both hardware and software. Increasing focus towards system aspects of cellular phones forced designers to take a holistic and dynamic approach to power management to effectively decrease power consumption without degrading performance.

A good number of techniques are used by designers to reach the goal — efficient processors, variable-speed clocks, circuit shutdowns, low-voltage logic, software design aids, and advanced power-management software. System-level power-management architecture typically starts with conservation at the source. For example, battery-management ICs and system-power regulators let engineers design power efficient products. Consider a typical wireless product with an RF receiver. Designers can employ a linear RF power controller to act as an on-off switch to conserve energy when the wireless feature is not in use. Effective power management architectures, however, must address all levels of system design. And software architecture plays a key role in system-level power management.

Source: Power Management for Mobile Devices, Sabyasachi Dey, May 05, 2006
Link: http://www.drdobbs.com/mobile/184407880?nomobile=1

Techniques d'optimisation de l'énergie (2/2)

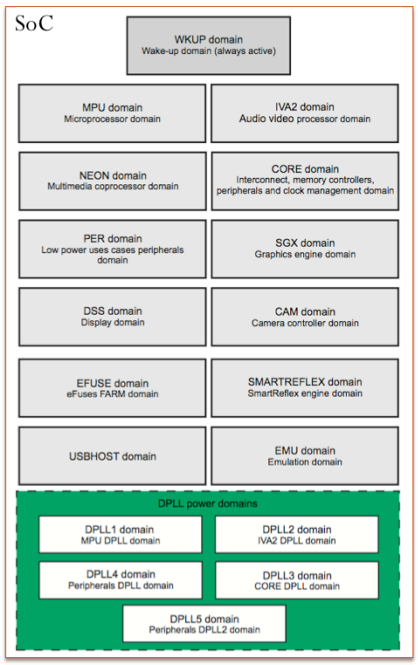
	Technology	Description
Software	Software Power Management Framework	Intelligence to control system power management
	Workload Monitoring and Prediction	Determines performance needs in order to adjust voltage and frequency accordingly
	Policy and Domain Managers	Performs dynamic control of system to meet performance needs with lowest power
SoC Design	Dynamic Voltage/Frequency Scaling (DVFS)	Dynamic adjustment of both voltage and frequency to adapt to performance required at the time
	Multiple Power/Voltage/Clock Domains	Physical domains that enable granular power management control by software
Silicon IP	Multi-Threshold CMOS (Dual-V_t) Cells	Higher Vt for lower-performance/low-leakage and lower Vt for higher-performance/higher-leakage
	Multi-Domain Support Cells (switch/buffer/isolation/level shifter)	Supports SoC multiple domains implementation

SoC Power Domains (1/2)

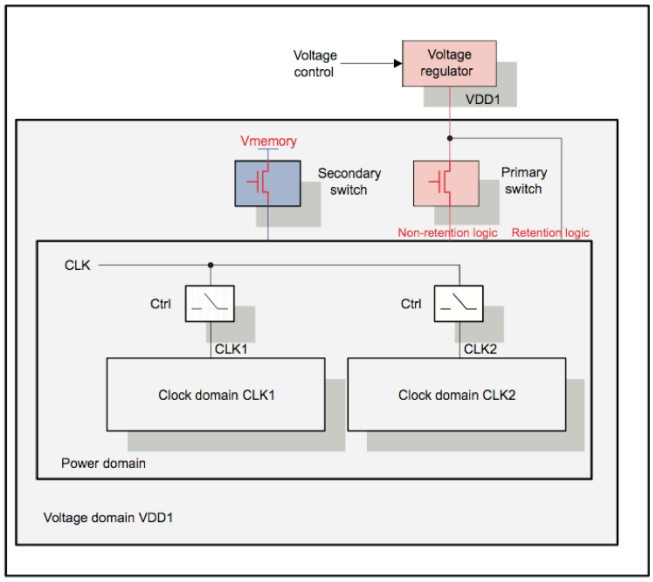
Normalement les SoC ont plusieurs « power domains » alimentés indépendamment.

Chaque « power domain » peut être activé ou désactivé par l'unité centrale.

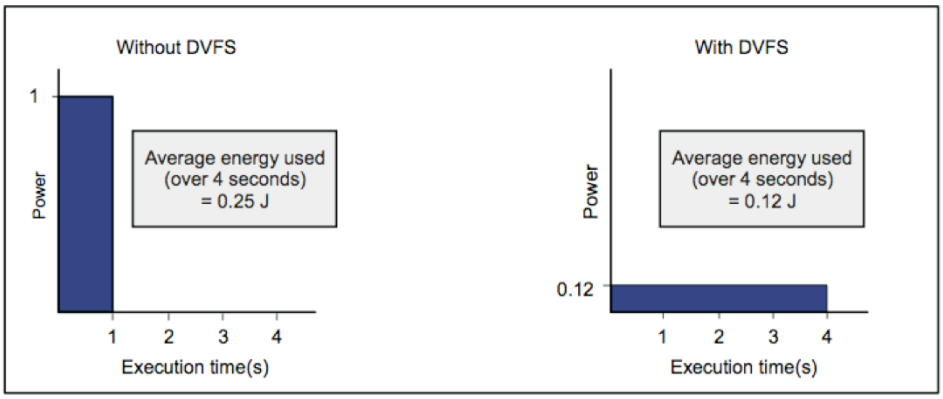
Le « Wake Up Domain » est normalement toujours actif et gère le système de réveille.



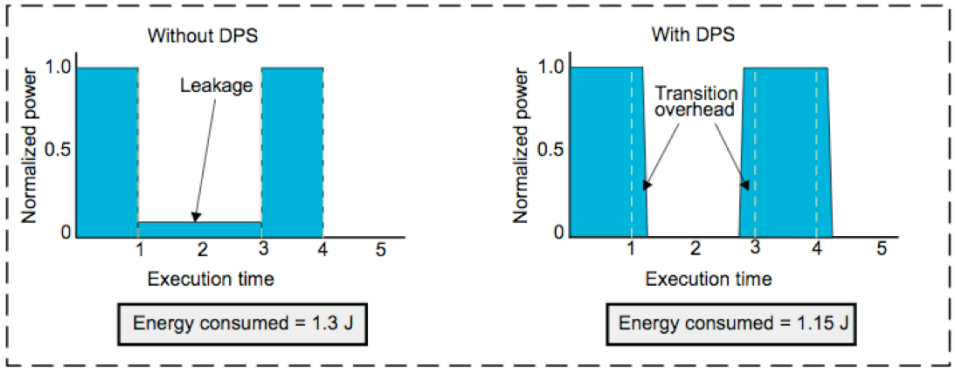
SoC Power Domains (2/2)



Dynamic Voltage and Frequency Scaling



Dynamic Power Switching



Ecriture de logiciel pour optimiser la consommation de batterie

Code Style Related Techniques

Les systèmes portables sont normalement utilisés par un seul utilisateur, contrairement aux systèmes de gestion de réseaux (Routers, Switches, Radio Access Nodes) où on gère des milers ou des millions d'utilisateurs par second.

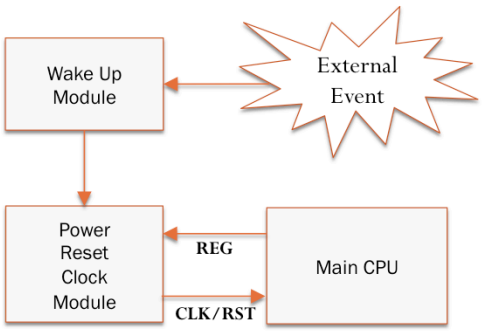
Donc il semblerait que dans le systèmes portable, on ait moins de nécessite d'optimisation de la vitesse d'exécution de routines.
C'est le cas dans les fonctionnalités d'un smartphone, mais il est utile de rappeler qu'on doit maximiser la durée de vie de la batterie.

Exécuter une routine plus rapidement signifie la terminer plus tôt pour pouvoir activer les « low power modes » du système.

Code Style Related Techniques

1. Assurer l'usage des LPM
2. Optimiser des boucles
3. Utiliser ISRs plutôt que « flag polling »
4. Minimiser les appels de fonction de la part des ISRs
5. Terminer les GPIOs non utilisés
6. Eviter les opérations de calcul intensif
7. Optimiser la gestion de variables
8. Optimiser la gestion de mémoire en utilisant le DMA
9. Utiliser les « bit-masks » plutôt que « bit-fields »

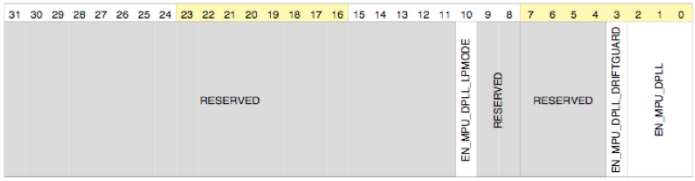
Ensure Low Power Mode Usage (1/3)



Ensure Low Power Mode Usage (2/3)

Table 3-118. CM_CLKEN_PLL_MPU

Address Offset	0x0000 0004		
Physical Address	0x4800 4904	**Instance**	MPU_CM
Description	This register controls the DPLL1 modes.		
Type	RW		

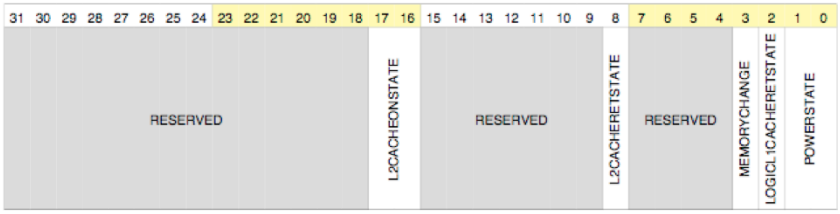


Bits	Field Name	Description	Type	Reset
31:11	RESERVED	Write 0s for future compatibility. Read returns 0.	R	0x000000
10	EN_MPU_DPLL_LPMODE	This bit allows to enable or disable the LP mode of the MPU DPLL. Writting this bit to switch the mode between LP or normal mode will take effect only when the DPLL will have transition into the bypass or stop state, followed by a lock or re-lock of the DPLL. 0x0: Disables the DPLL LP mode to re-enter the normal mode at the following lock or re-lock sequence. 0x1: Enables the DPLL LP mode to enter the LP mode at the following lock or re-lock sequence.	RW	0x0
9:8	RESERVED		RW	0x0
7:4	RESERVED	Reserved	RW	0x1
3	EN_MPU_DPLL_DRIFTGUARD	This bit allows to enable or disable the automatic recalibration feature of the MPU DPLL. The DPLL1 will automatically start a recalibration process upon assertion of the recal flag if this bit is set. 0x0: Disables the DPLL1 automatic recalibration mode 0x1: Enables the DPLL1 automatic recalibration mode	RW	0x0
2:0	EN_MPU_DPLL	DPLL1 control; Other enums: Reserved 0x5: Put the DPLL1 in low power bypass mode 0x7: Enables the DPLL1 in lock mode	RW	0x5

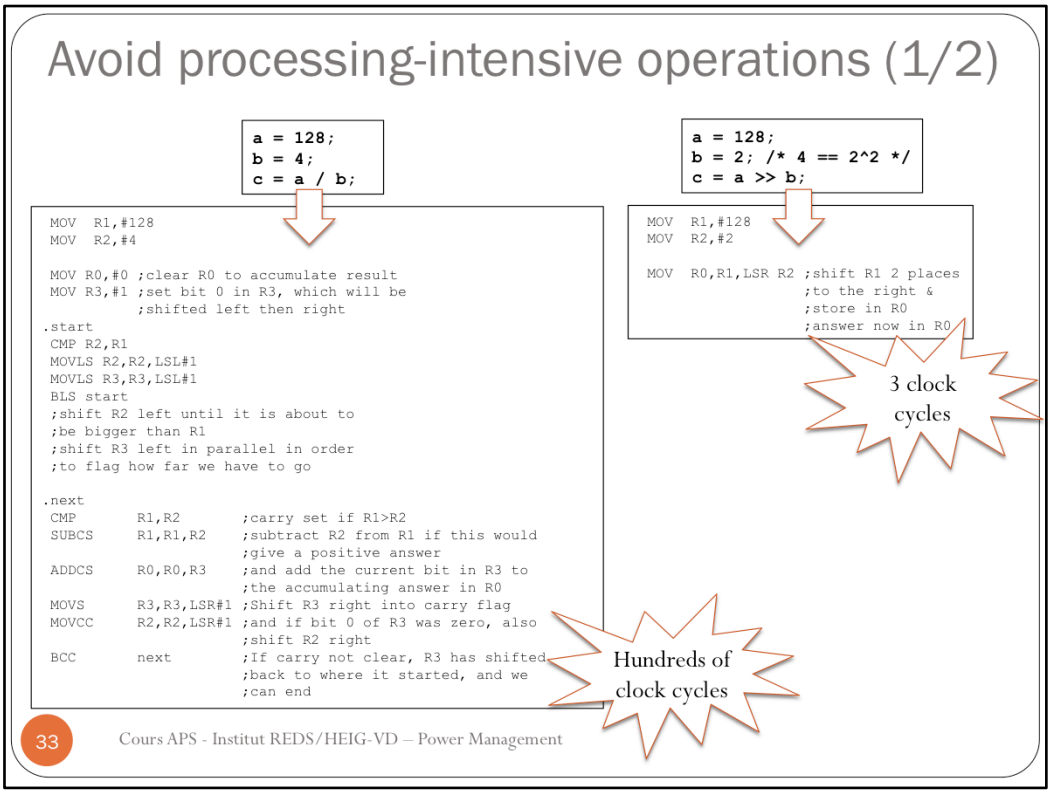
Ensure Low Power Mode Usage (3/3)

Table 3-331. PM_PWSTCTRL_MPU

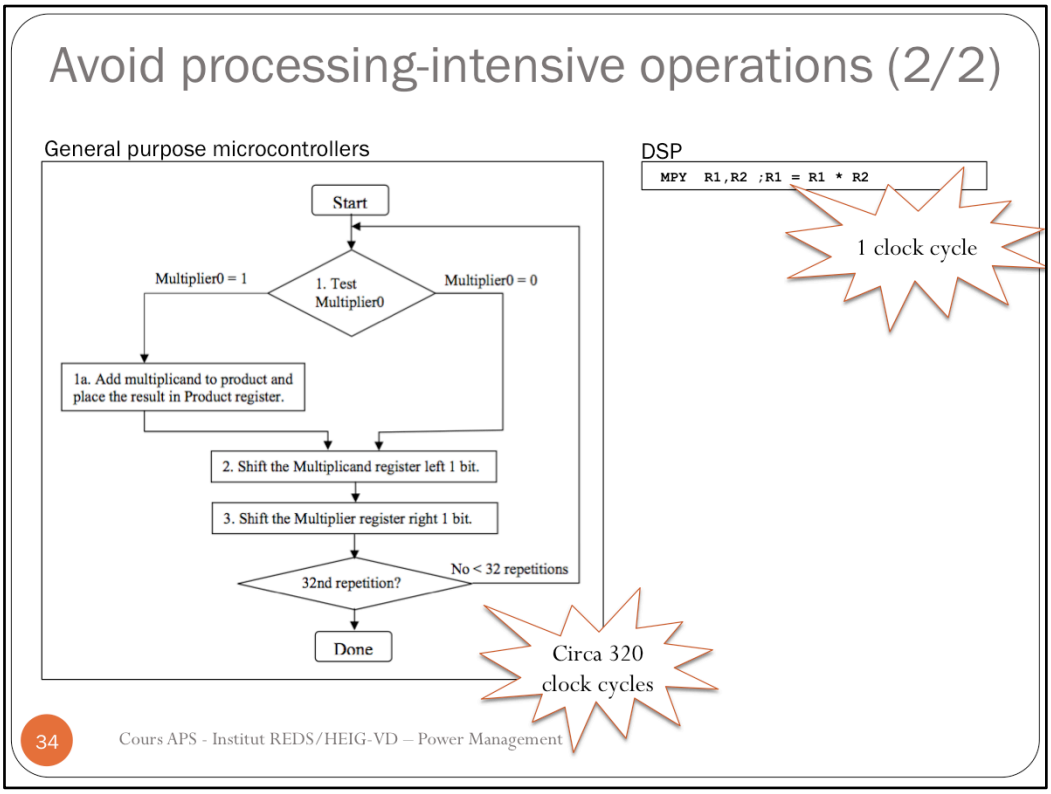
Address Offset	0x0000 00E0		
Physical Address	0x4830 69E0	**Instance**	MPU_PRM
Description	This register controls the MPU domain power state transition.		
Type	RW		



Bits	Field Name	Description	Type	Reset
31:18	RESERVED	Write 0s for future compatibility. Read returns 0.	R	0x0000
17:16	L2CACHEONSTATE	L2 Cache memory state when domain is ON; Other enums: Reserved 0x0: L2 Cache memory is OFF when domain is ON. 0x1: Reserved 0x2: Reserved 0x3: L2 Cache memory is ON when domain is ON.	RW	0x3
15:9	RESERVED	Write 0s for future compatibility. Read returns 0.	R	0x00
8	L2CACHERETSTATE	L2 Cache memory state when domain is RETENTION 0x0: L2 Cache memory is OFF when domain is in RETENTION state. 0x1: L2 Cache memory is retained when domain is in RETENTION state.	RW	0x1
7:4	RESERVED	Write 0s for future compatibility. Read returns 0.	R	0x0
3	MEMORYCHANGE	Memory change control in ON state 0x0: Disable memory change 0x1: Enable memory change state in ON state. This bit is automaticaly cleared when memory state is effectively changed.	RW	0x0
2	LOGICL1CACHERETSTATE	Logic and L1 Cache state when domain is RETENTION 0x0: Logic and L1 Cache are OFF when domain is in RETENTION state. 0x1: Logic and L1 Cache are retained when domain is in RETENTION state.	RW	0x1
1:0	POWERSTATE	Power state control 0x0: OFF state 0x1: RETENTION state 0x2: Reserved 0x3: ON state	RW	0x3

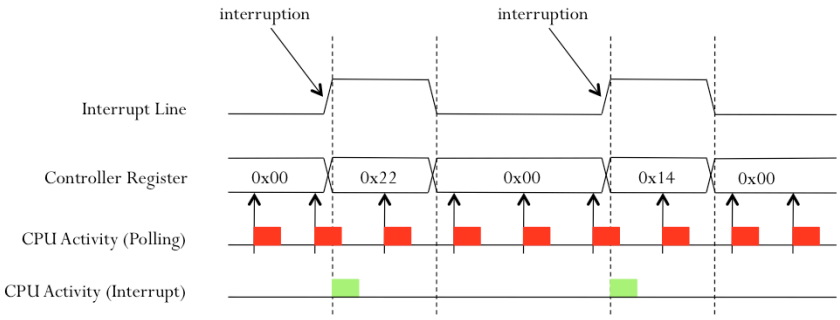


Les fonctions de division sont très intensives pour le processeur et consomment beaucoup de cycles d'instruction, qui gaspillent du temps et de l'énergie. Si la division est par une puissance de 2 on peut utiliser des « shifts » et des « bit masks ».

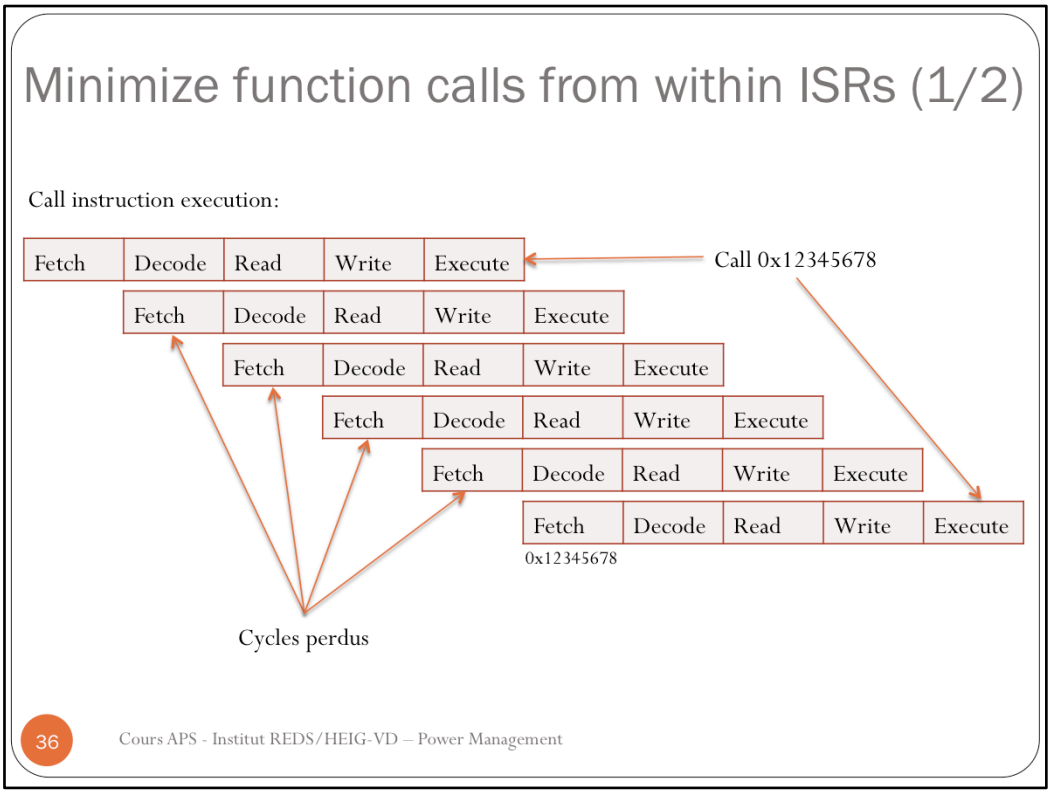


Sur le microcontrôleurs sans un module multiplicateur (MPY), les multiplications complexes (sans une puissance de 2) nécessitent que le compilateur génère du code supplémentaire pour simuler l'algorithme de multiplication. Cela pourrait potentiellement ajouter une quantité importante d'instructions pour le programme.

Use ISRs instead of flag polling



	Advantages	Inconvénients
Polling	Gestion simple	• Beaucoup de consommation d'énergie • Beaucoup de trafic sur le bus
Interrupts	• Optimisation de la consommation d'énergie • Optimisation du trafic sur le bus • Temps réel	• Gestion plus compliqué • Stack management



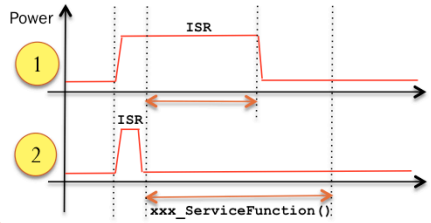
L'ISR doit être exécuté d'une manière courte et en temps opportun afin d'améliorer la réponse en temps réel du système.
Les ISRs doivent exécuter une quantité minimale d'instructions, ce qui permet au CPU de revenir rapidement au « sleep mode » ou à sa tâche précédente. La plupart du traitement devrait avoir lieu dans l'application principale en mode actif. Par conséquent, les appels de fonction au sein d'une routine ISR doit être évité afin de prévenir qu'une quantité considérable d'exécution de code supplémentaire soit introduite (ex. sur l'entrée: le stockage du PC actuel et l'état du processeur, le chargement de la fonction et la préparation de sa propre pile; sur la sortie: la restauration de la pile et l'état du processeur avant de revenir pour poursuivre l'exécution du code interrompu).

Au lieu d'appeler des fonctions dans l'ISR, essayer de fonctions « inline » ou « macros », c'est à dire déplacer les instructions de code directement dans l'ISR, ou de déplacer toute la fonction au dehors de l'ISR, si possible (en particulier pour les fonctions de traitement intensif). Pour cette dernière option, un indicateur global peut être configuré pour notifier au programme principal d'invoquer l'appel de fonction lorsque la CPU est hors de l'ISR.

Minimize function calls from within ISRs (2/2)

```
void xxx_ServiceFunction(xxx_param);

#pragma vector=xxx_VECTOR
__interrupt void xxx_ISR(void)
{
   reset_int_bit(XXX);
   xxx_ServiceFunction(xxx_param);
}

 void main(void)
 {
   while(1)
   {
     /* go to low power mode,
        wait for xxx interrupt */
   }
 }
```

1



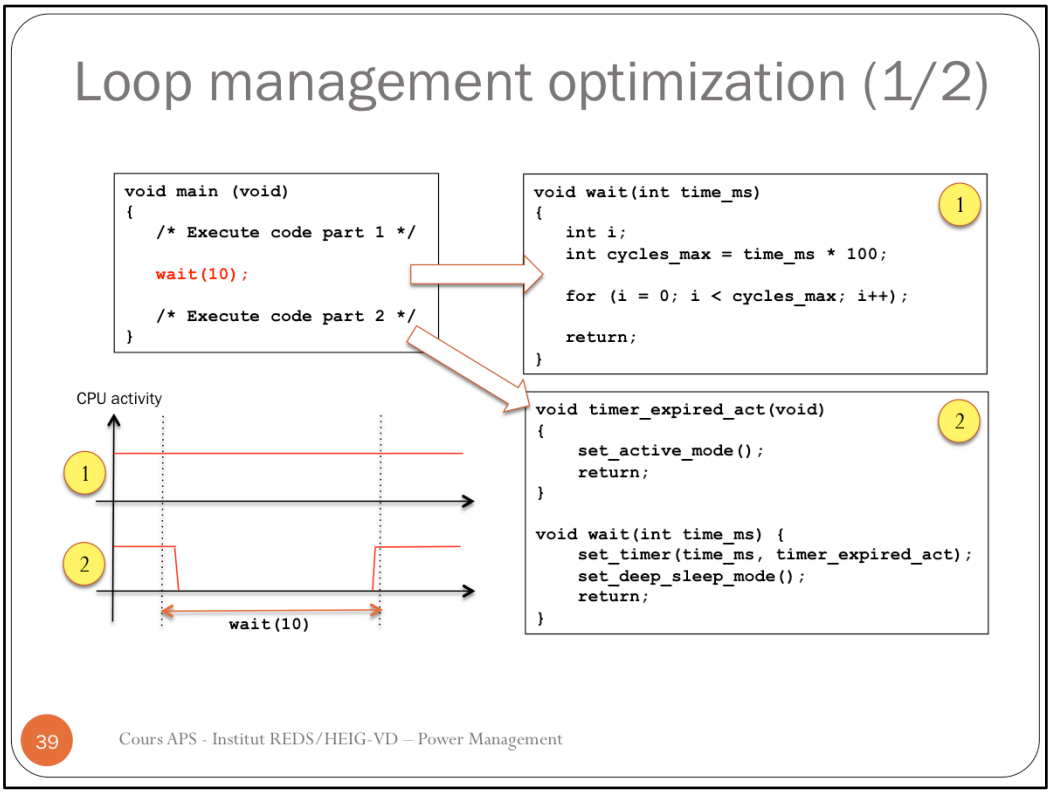

```
unsigned char interrupt_triggered = 0;
void xxx_ServiceFunction(xxx_param);

#pragma vector=xxx_VECTOR
__interrupt void xxx_ISR(void)
{
   interrupt_triggered = 1;
   reset_int_bit(XXX);
}

 void main(void)
 {
   while(1)
   {
     /* go to low power mode,
        wait for xxx interrupt */
     if (interrupt_triggered) {
       xxx_ServiceFunction(xxx_param);
       interrupt_triggered = 0;
     }
   }
 }
```

2

Terminate unused GPIOs

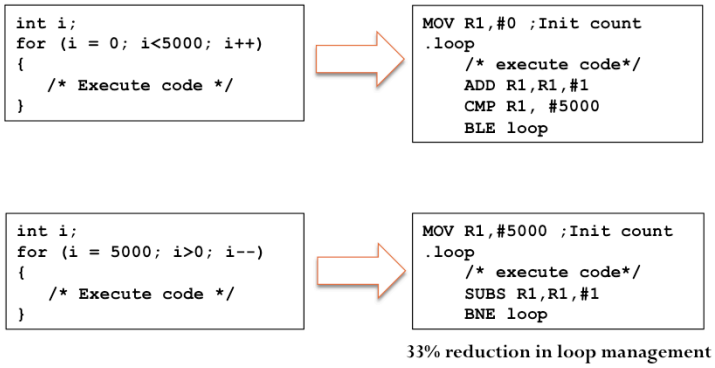
Si un PIN GPIO n'est pas utilisé et on ne l'initialise pas au BOOT, en dépendant de les valeurs random du registre de control, on peut avoir une consumation d' énergie random et naturellement pas optimale sur le portes GPIO.

Dans ce cas il est toujours mieux de les configurer comme output et en dépendant du circuit extérieur les configurer comme output 1 ou 0.



Les Microprocesseurs proposent différents types de temporisateurs et d'horloges qui peuvent être configurés pour fonctionner sans intervention du CPU. Quand un délai est nécessaire, un des périphériques « timer » peut être exploité pour générer un tel retard, sans que le CPU reste actif. Cette méthode réduit considérablement la consommation électrique de l'appareil. Ces « timers » peuvent permettre au microcontrôleur de rester dans un mode de faible consommation jusqu'au réveil du CPU grâce à l'expiration du « timer ».

Loop management optimization (2/2)



33% reduction in loop management

En assembly, un « branch » conditionnel basé sur la comparaison d'une variable envers une valeur non nulle exige deux instructions: « compare » et « branch ». Toutefois, lorsque le « branch » et « compare » est envers zéro, une instruction spécifique, BNE, peut être utilisée pour effectuer ces deux actions ensemble. Cela est également vrai pour une opération de branch en C. Ainsi un « counting down loop » permet de réduire d'une instruction pour chaque itération du loop par rapport à un « counting up loop ».

Optimize memory management using DMA

```
void *
memcpy (void *destaddr,
        void const *srcaddr,
        size_t len)
{
  char *dest = destaddr;
  char const *src = srcaddr;

  while (len-- > 0)
    *dest++ = *src++;

  return destaddr;
}
```

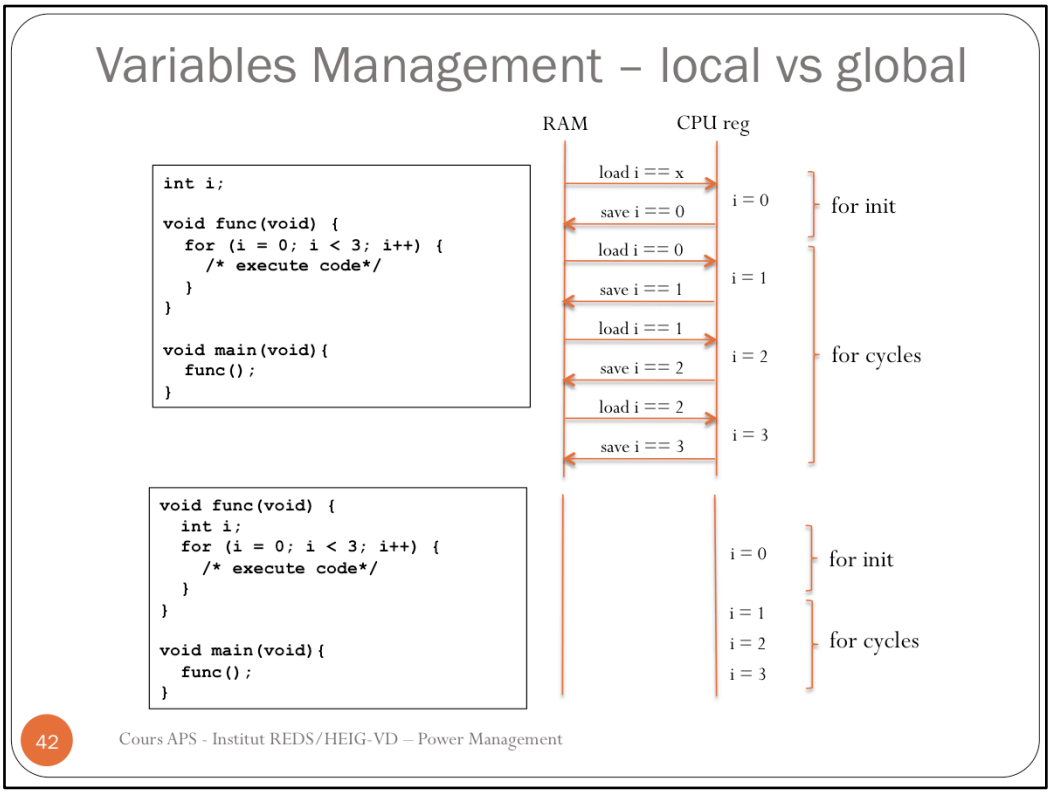
```
dest → r0
src  → r1
len  → r2
```

ASM

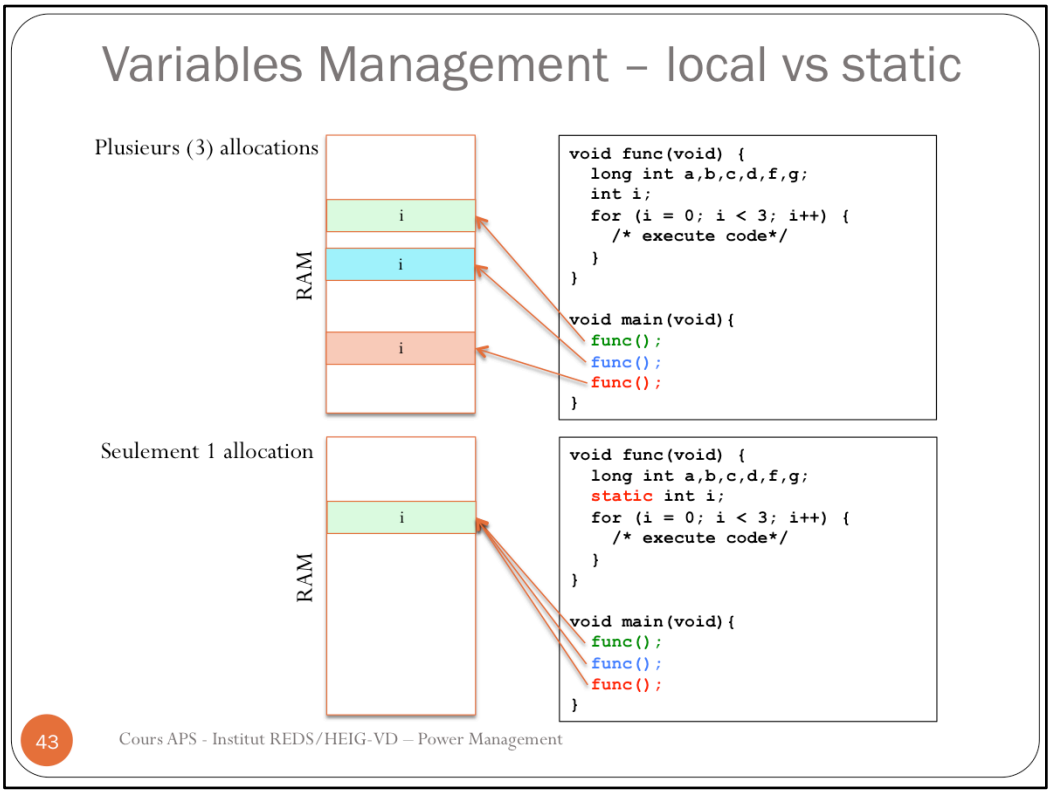
```
byte_loop:
    ldrb r3, [r1], #1
    strb r3, [r0], #1
    subs r2, r2, #1
    bne byte_loop
```

Chaque byte copy consume au moins 6 cycles de CPU clock

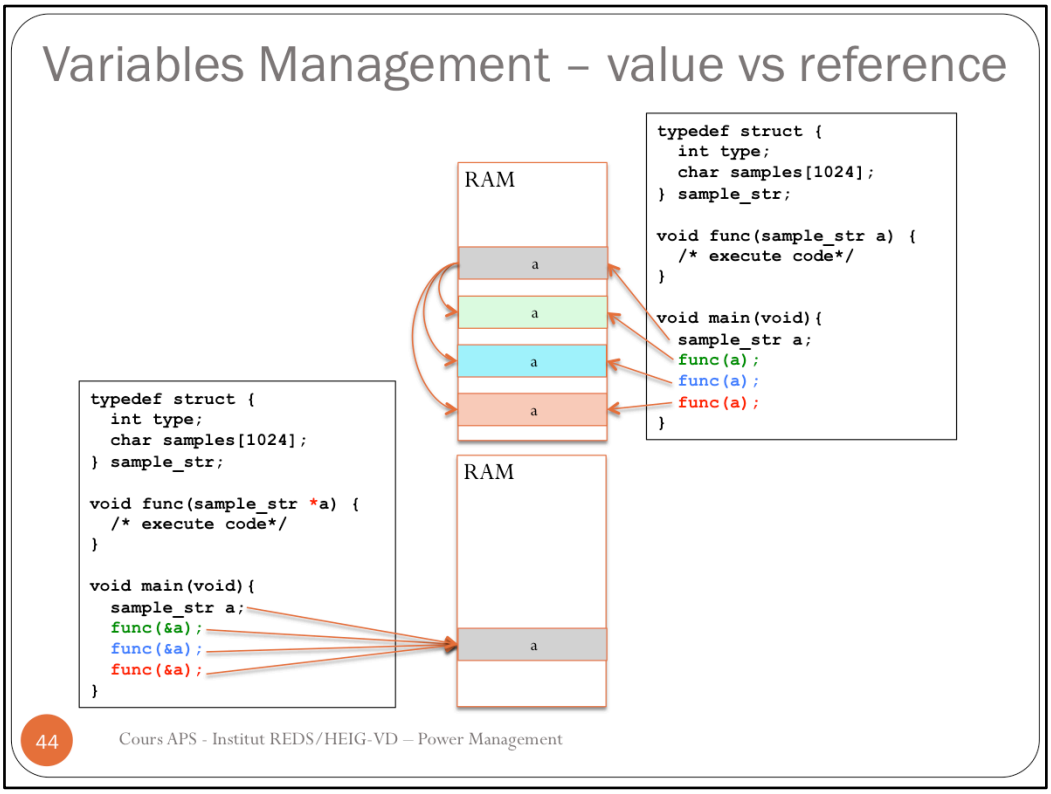
En utilisant un DMA on peut le réduire à un seul cycle en l'exécutant indépendamment de la CPU avec des matériels spécifiques pour le memory transfert



Les compilateurs C en général allouent les variables globales dans la mémoire RAM. D'autre part, les variables locales sont d'abord assignées dans les registres du processeur (si il y a des registres libres). Les accès aux registres CPU sont les plus efficaces et les plus rapides, par rapport à des accès à des locations de mémoire dans la RAM ou, pire encore, dans la mémoire FLASH. Par conséquence, toutes les variables globales qui sont accessibles seulement au sein d'une fonction, peuvent être en toute sécurité relogées dans le cadre de la fonction locale pour optimiser la vitesse d'accès de la mémoire et la consommation d'énergie.



Les variables locales sont allouées dynamiquement à chaque appel de fonction. Cela nécessite du code supplémentaire et de l'espace RAM et l'impact augmente en fonction de la fréquence d'appel de la fonction. D'autre part, lorsqu'elles sont déclarées comme «statiques», les variables sont générées qu'une fois et restent disponibles pendant toute la durée de l'application. Cela réduit la quantité de code nécessaire pour re-alluer / re-initialiser les variables chaque fois que la fonction est invoquée. En variante, lorsque le modificateur « const » est utilisé, la variable est stockée en tant que donnée de flash dans le cadre de la fonction, donc on ne nécessite pas de ré-allocation à chaque entrée de la fonction.



Les paramètres de les fonctions C peuvent être passés par valeur (par exemple int valeur) ou par référence (par exemple int *valeur). Lors du passage par valeur, le compilateur a besoin de générer du code pour copier les valeurs des variables dans les registres et dans la pile. Les registres ont un temps d'accès plus rapide, mais sont limités en nombre, par conséquence, ils sont utilisés pour les premières variables petites. Pour les grandes variables telles que les structures ou les "unions", la pile située dans la mémoire RAM est requise pour stocker les données supplémentaires. Copier des structures de données nécessite l'exécution de code supplémentaire. Lors du passage par référence, la fonction a seulement besoin de copier les adresses des variables, limitant les instructions pour copier les données globales seulement au nombre des paramètres de la fonction. Ceci réduit considérablement le nombre d'instructions en cours d'exécution à chaque fois que la fonction est invoquée.

Use bit-masks instead of bit-fields

```
void functionA(bool var1,
               bool var2,
               bool var3)
{
  if (var1)
  {
   /* Execute code */
  }

  if (var2)
  {
   /* Execute code */
  }

  if (var3)
  {
   /* Execute code */
  }
}

void main(void)
{
  bool variable1 = TRUE;
  bool variable2 = FALSE;
  bool variable3 = TRUE;

  /* Execute code */

  functionA(variable1, variable2, variable2);
}
```

```
#define FLAG_1     1
#define FLAG_2     2
#define FLAG_3     4
#define FLAG_4     8

 void functionA(unsigned char variable)
 {
   if ( variable & FLAG_1)
   {
    /* Execute code */
   }

   if ( variable & FLAG_2)
   {
    /* Execute code */
   }

   if ( variable & FLAG_4)
   {
    /* Execute code */
   }
 }

 void main(void)
 {
   unsigned int variable1 = FLAG_1 | FLAG_3;

   /* Execute code */

   functionA(variable1);
 }
```

45 Cours APS - Institut REDS/HEIG-VD – Power Management

Normalement dans le code embarqué C, on utilise, pour stocker plusieurs bits d'information en une seule variable, de champs binaires ou masque de bits. L'avantage des bit-mask sur les bit-fields, c'est que plusieurs bits masques peuvent être combinés en un masque de bits unique, permettant au compilateur d'émettre une instruction unique pour accéder à la variable. D'autre part, le compilateur ne peut pas gérer les bit-fields avec une seule instruction. Donc on doit exécuter le code supplémentaire et par conséquent utiliser plus d'énergie pour exécuter la même quantité de code efficace.

Références

- Jane Sales, Sysmbian OS Internals: "**Real-time Kernel Programming**", Wiley
- Sajal K. Das, "**Mobile Handset Design**", Wiley
- Sabyasachi Dey, **Power Management for Mobile Devices**, Dr.Dobb's article, May 05, 2006