41 pt. 5.7

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Nom : Grossrieder **Prénom :** Nadine **Classe : T-2/I-2** **Date : 20.04.2015**

Problème n° 1 (interfaçage C - assembleur)

a) Codez en assembleur la fonction « bar » ci-dessous. Note : les « **long** » ont 32 bits.

```
typedef long (*foo_t) (long a1, long a2);
long bar (long p1, foo_t foo, long p2, long p3, long p4, long p5)
{return p3 + p2 + foo (p5, p1) + p4;}
```

```
bar:  stmfd sp!, {lr}
      add  r2, r3
      add  r2, [sp, #4]   // p4  décaler de 4 puisque on sauve seulement lr
      mov  r3, r4         // foo
      mov  r4, r0         // p1
      mov  r0, [sp, #8]   // p5
      blx  r3
      add  r0, r2
      ldmfd sp!, {pc}
```

lr	← sp
p4	+4
p5	+8

5

bien.

r2 n'est pas définit

ok, mais l'ordre n'est pas respecté.

pourquoi?

↳ seulement quand appel à une interruption

b) Le graphique ci-dessous représente l'état du processeur (registres et pile sur 32 bits) à l'entrée de la fonction « f2 » (aucune instruction de « f2 » n'a encore été exécutée).

```
int f2 (int a1, int a2, struct S* a3) {return a2 + f3(a1, a3);}
```

Indiquez la valeur des paramètres a1 à a3.

a1 : 10
a2 : 1020
a3 : 0

low address	100
	414
	153
SP (à l'entrée de f2) →	0
	567
	898
	910
high address	1114

R0	10
R1	1020
R2	0
R3	45
R4	18
R5	90
R6	33

3

c) Implémentez en assembleur le passage par valeur de la variable « int var ; » dans le registre R0

```
ldr r0, =var
ldr r0, [r0]
```

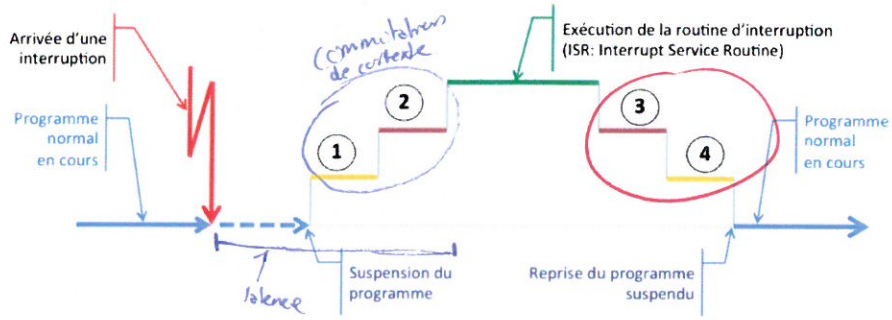
1

8

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 2 (Interruptions)

La figure ci-dessous représente les différentes transitions que le microprocesseur et son logiciel effectuent lorsque une interruption est soulevée avant de pouvoir exécuter la routine d'interruption (ISR).



a) Décrivez succinctement les opérations effectuées lors des transitions 1 à 4. Précisez si celles-ci sont effectuées par le microprocesseur (hw) ou par le logiciel (sw).

3

① effectuée par Hw
sauvegarde de l'état du processeur
recherche de l'adresse de la routine d'interruption ✓

② effectuée par Sw
Changement de contexte et sauvegarde des registres ✓

③ effectuée par Sw ✓
Changement de contexte et restauration des registre

④ effectuée par Hw ✓
restauration de l'état du processeur

b) Implémentez les opérations assembleur que le microprocesseur devra exécuter pour sauvegarder et restaurer l'état du microprocesseur lorsqu'une interruption matérielle (IRQ ou FIQ) est levée. Indiquez le numéro de la transition où s'effectuent ces opérations

2

Numéro de la transition : ① 2

```
sub   lr, #-4
stmfd sp!, {r0-r12, lr}

ldmfd sp!, {r0-r12, pc}^
```

Subs pc,lr,#4

2

1/2 c) Représentez sur le graphique ci-dessus la commutation de contexte d'interruption.

Partie ① et ②

1 d) Représentez sur le graphique ci-dessus la latence. ✓

de l'arrivée de l'interruption jusqu'à la fin de la partie ②

e) Donnez le terme technique de la variation du temps de latence et citez 2 exemples qui peuvent la faire varier fortement.

Gigue d'interruption ou jitter interrupt ✓

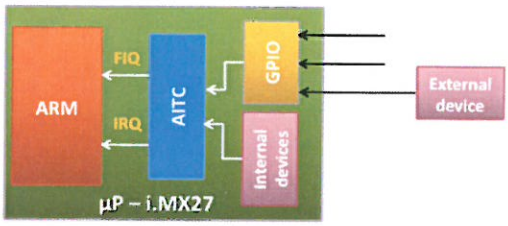
2 ① le temps d'exécution de l'instruction la plus longue ✓
② le temps où les interruptions sont désactivées ✓

(9)

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 3 (Interruptions)

La figure ci-contre représente le système d'interruption du microprocesseur i.MX27.



a) Décrivez succinctement le rôle des composants ci-dessous et indiquez la méthode utilisée par le composant pour identifier la source d'interruption.

- ARM : priorité d'interruption ✓
  Active l'interruption et cherche l'adresse de la routine d'interruption

3

- AITC : interruption vectorisé ✓
  Reçoit une interruption soit du GPIO soit des périphériques internes

- GPIO : scrutation logicielle
  reçoit l'interruption d'un périphérique externe
  et détermine de quel périphérique il provient

b) Implémentez en assembleur la fonction « `void init_sp (int mode, void* sp);` » permettant d'initialiser les pointers de piles pour les différents modes du µP (registre SP).

```
init_sp : stmfd sp!, {r0-r12, lr}
          mrs  r3, cpsr_c      mov r4, r3
          bic  r3, r3, #3F
          add  r3, r0
          msr  cpsr_c, r3
          mov  sp, r4
          msr  cpsr_c, r4
          ldmfd sp! {r0-r12, pc}^
```

3

IF M

M 01110

r3 00 00000

00 01100

c) Indiquez le type de pile utilisée et comment celle-ci fonctionne.

pile full descending ✓

On commence à stocker les données à l'adresse la plus grande et on remonte vers les adresses plus petites

1

d) Sachant que « `var` » est une variable de type « int », décrivez pour quelle raison l'opération « `var++;` » n'est pas atomique et indiquez les instructions à effectuer pour rendre cette opération atomique.

On ne sait pas quand l'incrément va être fait (car il peut y avoir une interruption entre deux)

2

il faut désactiver les interruptions avant, faire l'opération voulue puis réactiver les interruptions

entre deux qui ?

~~Car le~~

9

travail sur read modify write

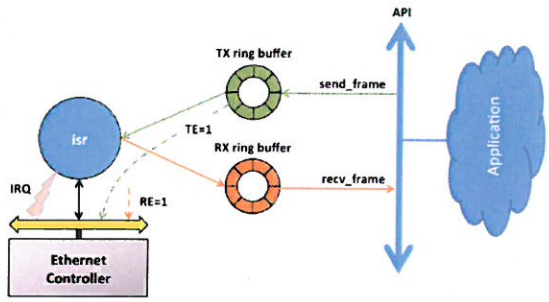
ldr
add
str

plusieurs instruction faite pour incrementer
et une interruption peu intervenir entre deux instructions

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 4 (Entrées/Sorties)

La figure ci-contre représente le schéma de principe pour le traitement par interruption d'un contrôleur pour une interface Ethernet à 100Mbit/s full-duplex.



a) Décrivez la fonction des tampons de réception et d'émission (RX ring buffer et TX ring buffer).

Les tampons permettent de stocker les informations pour que l'application ou le périphérique ne soit pas obligé de tout le temps scruter pour obtenir ou envoyer des données

1 à peu près

b) Décrivez le rôle de la routine d'interruption (isr).

La routine d'interruption va recevoir les interruptions du périphérique et va aller stocker ou chercher les informations dans les buffer

2

c) Implémentez la routine « `void send_frame (const msg_t* frame) ;` » permettant à l'application d'émettre des trames.

```
#define CTRL_TE (1<<1)  // transmitter interrupt enabled
static volatile struct ethernet_ctrl {
      uint16_t ctrl;       // control register
      /* ... other registers... */
} * ether = (struct ether_ctrl*)0x2001c000;
struct fifo {int in; int out, msg_t frames[512];} tx_fifo;

void send_frame (const msg_t* frame) {
```

~~ether->ctrl &= ~CTRL_TE~~ // désactive les interruptions

if (tx_fifo->in +1 != tx_fifo->out) { // on vérifie qu'il y a de la place dans le buffer

tx_fifo->frames[in] = frame

inc in

ether->ctrl |= CTRL_TE // active les interruptions

3-

```
while (tx.out == ((tx.in+1) % SIZE));
tx.buffer[tx.in] = data;
tx.in = (tx.in + 1) % SIZE;
ether->ether_ctrl |= CONTROL_TE;
```

d) Pour le contrôleur Ethernet ci-dessus et le mode de traitement par interruption, dimensionnez le tampon de réception afin que la latence maximale autorisée côté application soit au minimum de 45 ms.
Afin d'économiser la taille mémoire nécessaire, le tampon de réception sera formé de blocs de 200 bytes chacun. Si la taille du paquet dépasse la taille maximale d'un bloc, celui sera stocké sur plusieurs blocs.
Côté réseau des paquets de 125 bytes et/ou 1250 bytes (framing compris) sont émis en burst de 5 ms à plein débit et suivi ensuite d'une pause de 10 ms.

débit = 100Mbit/s

15ms

x · 200bytes = buffer

125 bytes = 1000bit
1250 bytes = 10000bit

$\frac{1000}{100 \cdot 10^6} = 10\mu s$

$\frac{10000}{100 \cdot 10^6} = 100\mu s$

$\frac{15ms}{10\mu s} = 1000$ paquets

$\frac{15ms}{100\mu s} = 100$ paquets

1500 buffer de 500 bytes

100MBit/s = 100 000 bit/ms
500 000 bit/15ms // 5ms + Pause de 10ms
1250 bytes = 10 000 bit par paquet
500 000 / 10 000 = 50 Paquet
200 bytes = 1600 bit
10 000 / 1600 = 6.25 -> 7 bloc pour 1 paquet
7 * 50 = 300 bloc pour 15 ms
1050 bloc pour 45 ms

en 15ms

$x \cdot 15ms = 6.5ms \Rightarrow y = 3$

$x \cdot 200 = 375'000 \Rightarrow x = 1875$ entrées

$3000 \cdot 125 = 375'000$ bytes

$3000 \cdot 1250 = 3750000$ bytes

taille buffer

1 1/2

7

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 5 (Systèmes temps-réel)

a) Citez les composants principaux du noyau d'un système d'exploitation.

- ~~Processeur~~
- Ordonnanceur /
- Periodic Timer /
- Mécanisme de synchronisation /
- Mécanisme de gestion des interruption /
- Mécanisme d'échange d'informations.
- Gestionnaire de mémoire /

3

b) Définissez la structure minimale du TCB (Thread Control Block)

```
struct tcb {
    uint32_t register[]
    uint32_t lr
    uint32_t sp
    uint32_t psr ✓
    tcb*     next ✓
    char*    state ✓
}
```

2 — stack

c) Implémentez la fonction de transfert implémentant le la commutation de contexte entre deux threads.

```
void transfer (struct tcb* former, uint32_t psr, struct tcb* new) ;
transfer :
```

```
    stmia  r0,{r0-r12,lr,sp}
    ~~msr~~ ldr  r1 ,[r0,#4*15]
    ~~mrs~~ ldr  r1 , [r2 , #4*15]    msr...
    ldmia  r2 , {r0-r12,pc,sp}
```

2 — Dans exam oral

d) Définissez la structure minimale d'un sémaphore.

```
struct semaphore {
    tcb*  running_thread
    bool  is_use
    tcb*  next_thread ✓
}
```

1/2 — counter — Dans exam oral

e) Implémentez la fonction « `void sema_signal(int id) ;` » permettant de libérer le sémaphore.

Quelques éléments de réalisation :

```
struct semaphore sema [200];
struct tcb* sema_extract_thread (int sema_id); // extrait 1er tcb contenu dans la liste du sémaphore, retourne 0 si aucun tcb n'est dans la liste
void sema_signal(int id) {
    if( id < 200 && sema[id] -> is_use ) {
        while (sema_extract_thread (id) != 0); ?
        sema[id] -> ~~is_use = false;~~
}
```

0

```
sema[id].counter++;
struct tcb* tcb = extract_thread(id)
if(tcb != 0) {
    tcb->state = WAITING
}
```

(7)