

- repérer les entités de base qui peuvent être réalisées d'un jet et en décrire les séquences,
- repérer les évènements déclencheurs
- repérer les évènements bloquants ... que l'on attend cela va nécessiter un mécanisme pour les notions d'attente
- repérer les actions répétitives
- isoler les paramètres de réglage
- repérer les actions critiques, nécessitant la plus faible latence :
 - envisager de les gérer par IT (si temps de réaction faible)**
 - envisager de les faire gérer par un périphérique**
- identifier les flux à gérer
 - envisager d'utiliser un DMA pour du transfert automatique**
- identifier le temps de boucle du While(1) du main :
 - quelle est la contrainte max de temps de réaction+ traitement ?**
- repérer les séquences trop longues par rapport ce temps de boucle et les fragmenter....
 - pour intercaler plusieurs fois la tâche critique (beurk....)

Partage du temps d'exécution ordonnancement sur petite cible

Chaque tâche doit rendre régulièrement la main volontairement :

- Fragmenter une tâche longue en de multiples états d'un "switch () case ... :
- Fragmenter une tâche longue en petites tâches qui se déclenchent en cascade
- La boucle du main est l'ordonnanceur des tâches de fond :

ATTENTION aux faux découpages, aux phénomènes de cascades directes

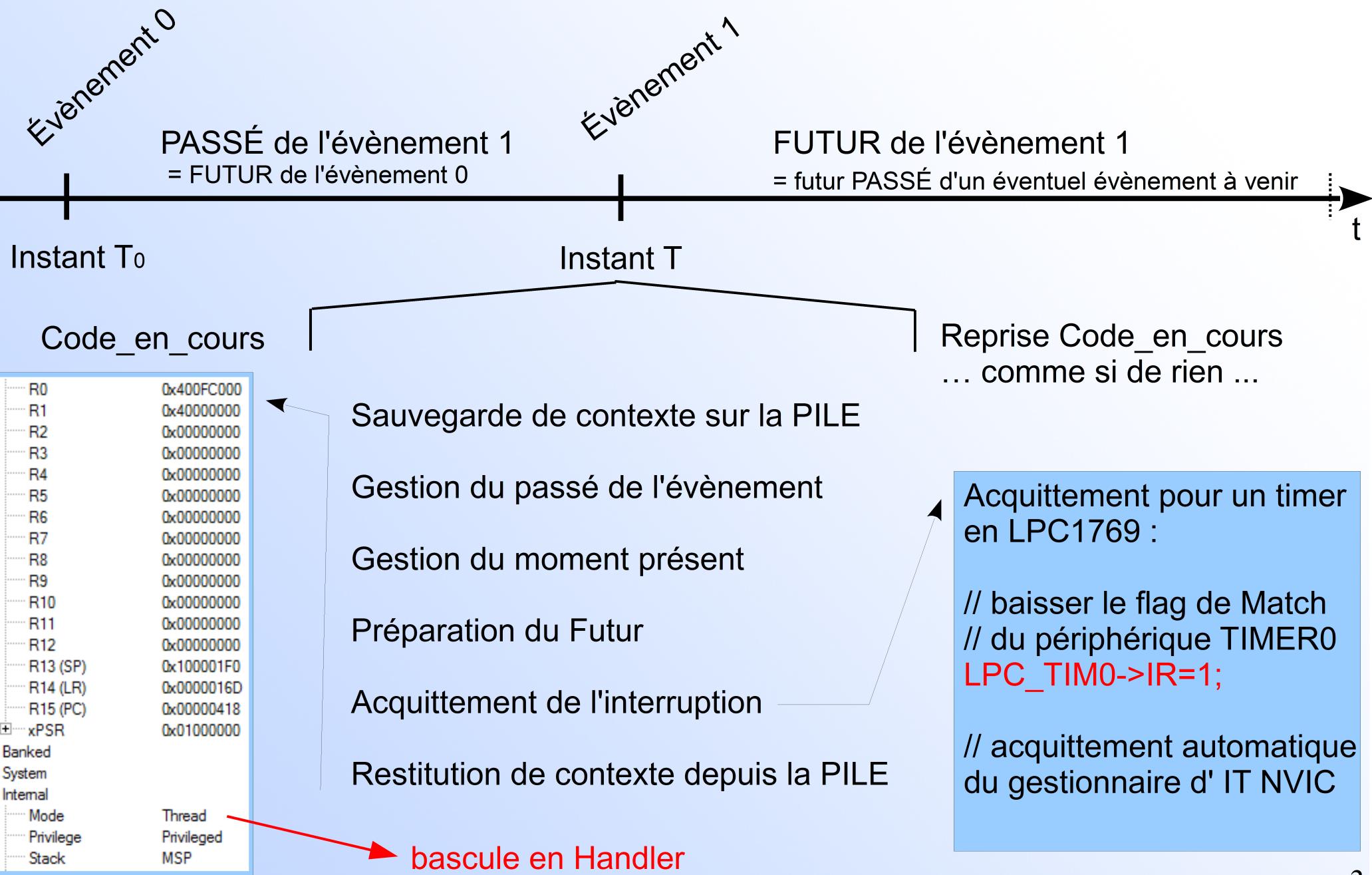
→ penser à ordonner à l'envers pour garantir une seule étape par boucle

AVOIR EN TÊTE le pire des cas pour savoir si toutes les tâches sont réalisables,
au besoin re-intercaler une même tâche dans la boucle.

**Le processeur a la capacité de suspendre une tâche
en cours pour passer la main à une Interruption :**

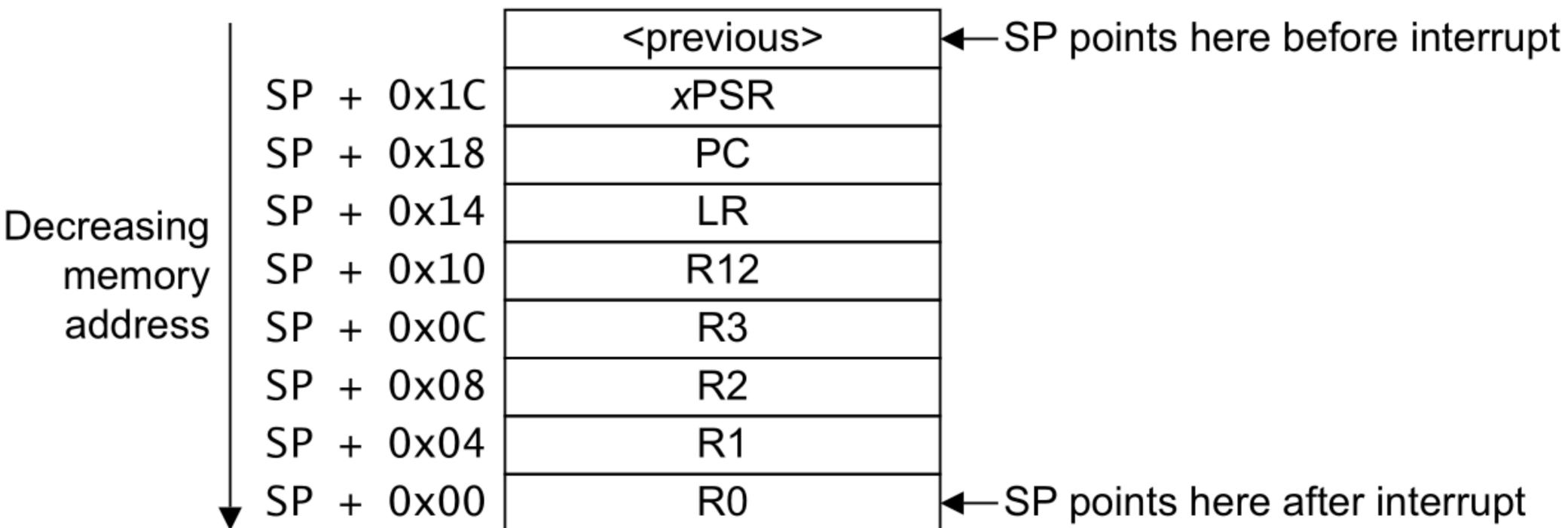
METTRE en Interruption les fonctionnalités les plus urgentes,
les plus contraignantes en terme d'échéances.

NE PAS METTRE en Interruption une tâche réalisable par scrutation,
dont l'échéance n'est pas une contrainte au niveau du temps de boucle

Préemption native sur micro-contrôleur
Interruption classique

Sauvegarde de contexte

Pile lors d'une interruption



Il y a une sauvegarde automatique partielle du contexte :

- minimiser le nombre de cycles nécessaires à la sauvegarde de contexte
- minimiser le nombre de cycles nécessaires à la restitution de contexte
- Pas de sauvegarde en cas de cascade d'IT (on restituera le contexte de la première)
- Sauvegarde/Restitution si une IT interrompt une IT moins prioritaire

Besoin de commutation de tâche

→ Limite à la fragmentation d'une tâche :

Une machine à état comportant de nombreux cas doit être optimisée :

Table de goto avec des Enum consécutifs

Distribution par dichotomie (4 tests pour 16 cas, 5 pour 32 cas)

Regrouper les cas d'attente en un seul au début pour rendre la main vite:

```
case ATTENTE : if(!cpt) {mae= mae_next;} break;  
case XXX : cpt = 10; mae_next= YYY; mae = ATTENTE; break;  
case YYY : //la suite
```

→ Limite de consommation de temps CPU à scruter trop de sémaphores

→ Répétition de code dans la boucle du main() pour gérer une tâche plus urgente

→ Un Os maison n'est pas compatible avec des librairies externes :

Les librairies pour le réseau, l'utilisation d'une mémoire avec FAT sont souvent dédiées à un OS particulier

→ Périphériques ou développements consommateurs de ressources et de temps CPU :

Ecran graphique

SDCard avec ou sans FAT

Réseau

→ Limitation des capacités du programmeur à évaluer le pire des cas

la commutation de tâche

→ Imiter le mécanisme d'interruption pour la commutation de tâche :

IT classique : interruption du code en cours,
exécution du code d'IT (IT handler)
reprise du code en cours comme si de rien n'était

Commutation de tâche :

interruption du code de la tâche en cours (sauver contexte)
choix d'une autre tâche
reprise du code de l'autre tâche là où elle en était (reprise contexte)

→ Le lancement d'une tâche pour la première fois doit se comporter comme une commutation de tâche tout ce qu'il y a de plus classique (reprise de contexte)

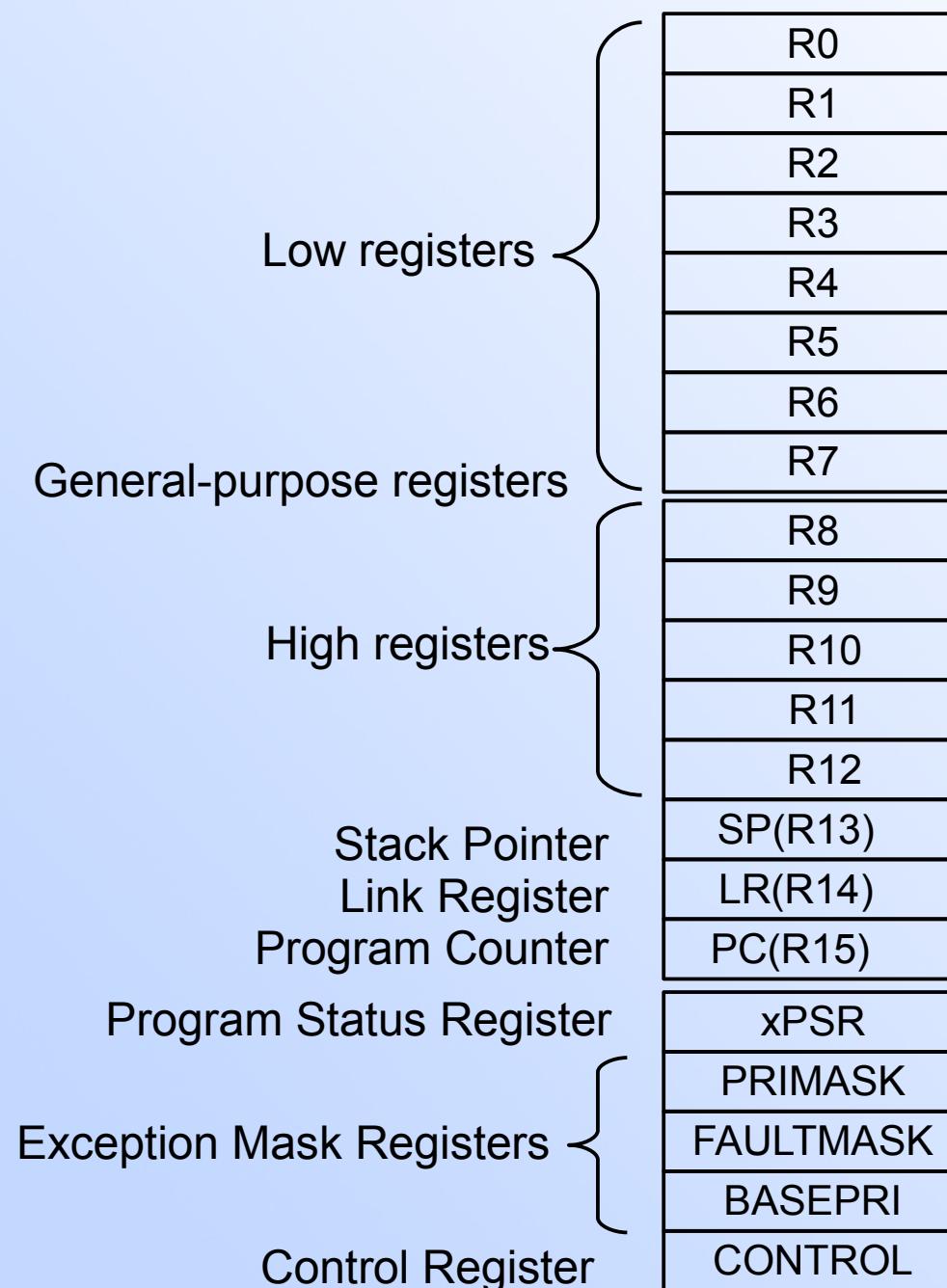
IL FAUT DONC FABRIQUER UN FAUX CONTEXTE à la création de tâche:
Imiter un appel de fonction avec des paramètres

→ Une tâche doit pouvoir fonctionner comme un main() classique, faire des appels de fonction, qui peuvent être appelées par diverses tâches sans que les variables locales se mélangent... IL FAUT AUSSI POUVOIR LANCER UNE TACHE avec des paramètres

IL FAUT DONC ETUDIER :

- LA STRUCTURE DE LA CIBLE CHOISIE au niveau registres, gestionnaire d'IT:
- LE COMPORTEMENT DU PROCESSEUR et le jeu d'instructions ASSEMBLEUR
- LE COMPORTEMENT DU COMPILEUR ET DU LINKER
POUR LES VARIABLES, LES APPELS DE FONCTIONS , LES INTERRUPTIONS

Registres du cœur cortex M3



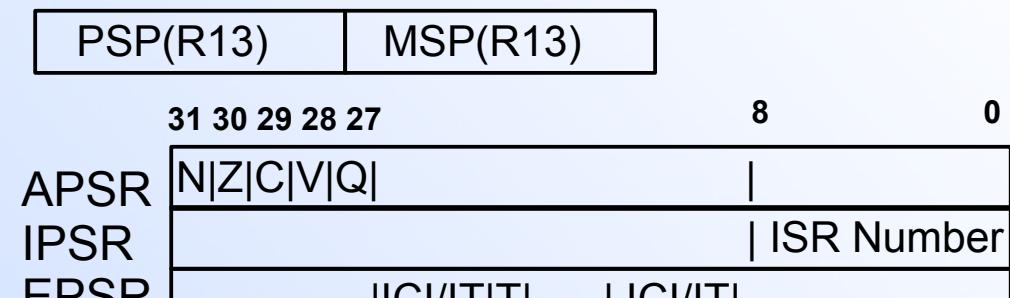
Deux modes de fonctionnement :

Thread mode pour les tâches
Handler mode pour les exceptions

Deux niveaux de Privilèges :

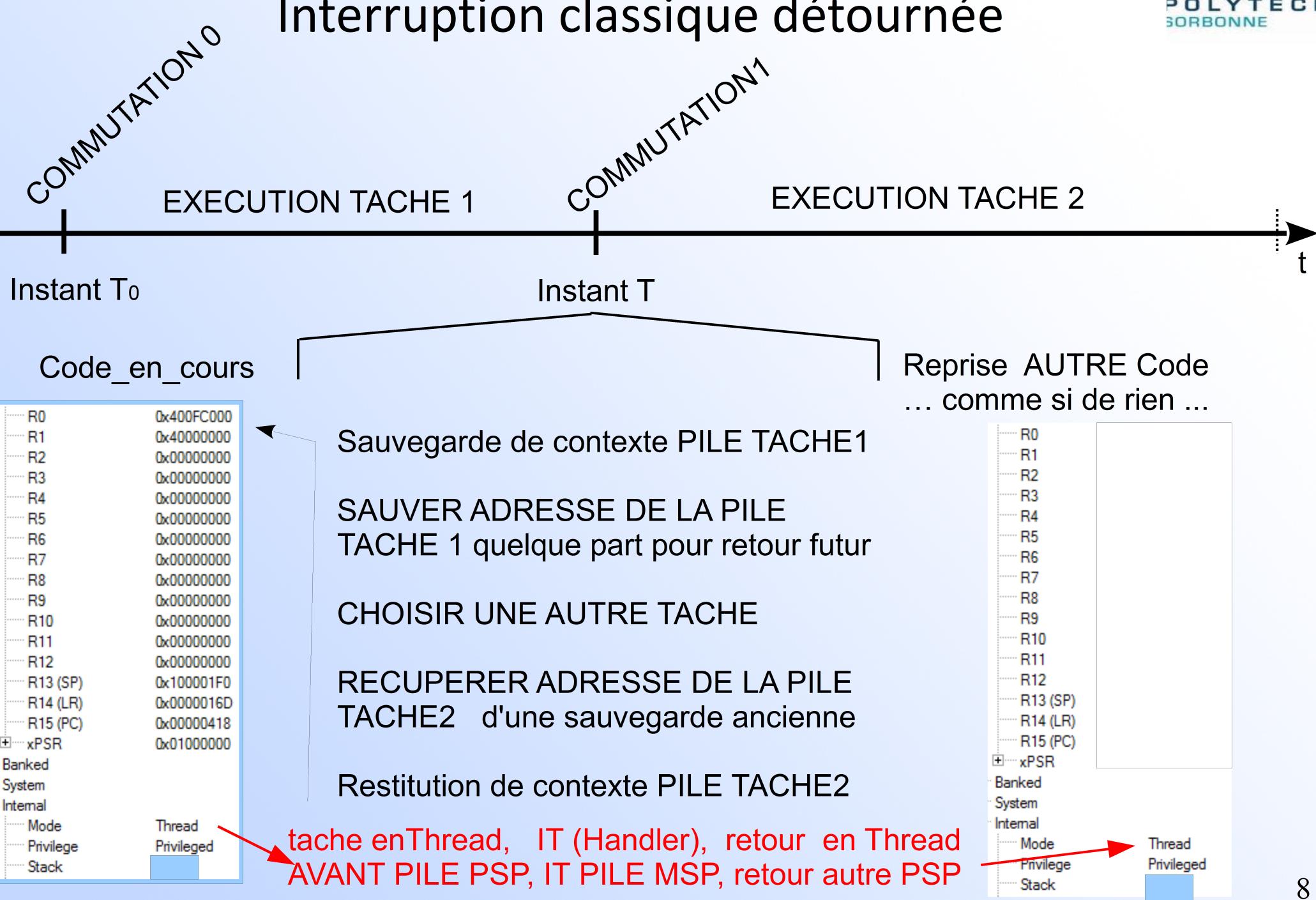
Privileged
Unprivileged

P : Process M : Main



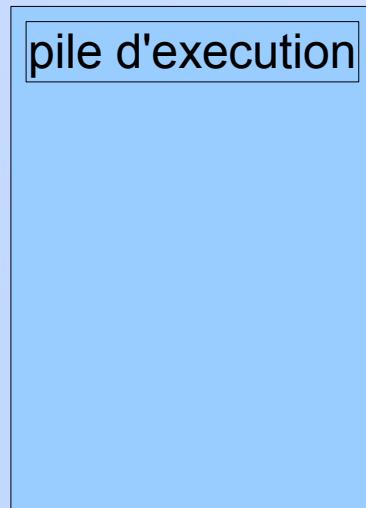
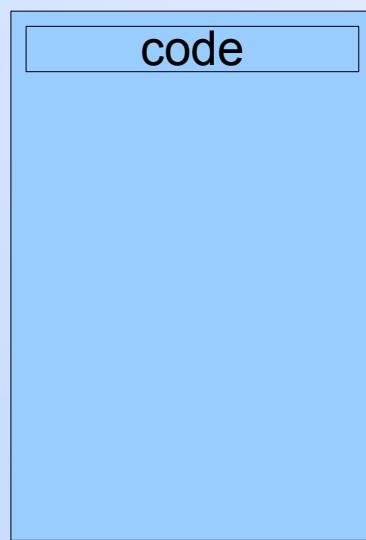
CHANGER DE TACHE

Interruption classique détournée

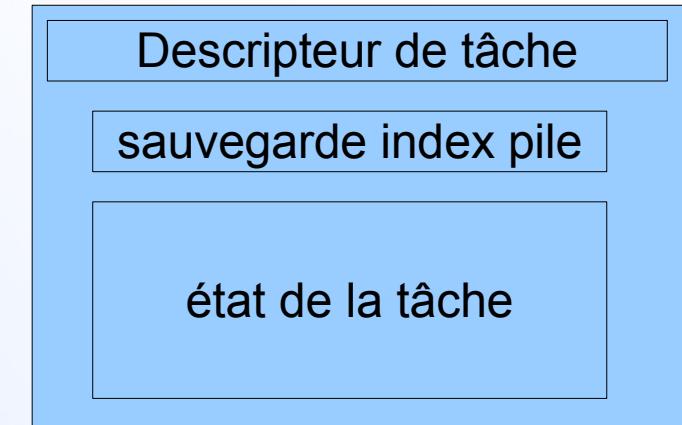


Que faut-il comme concepts à minima pour décrire informatiquement une Tâche dans l'idée de multi-tâches ?

Ressources



- en attente d'évènement :
 - attente délai
 - évènement externe
 - sémaphore ou signalisation venant d'une autre tâche

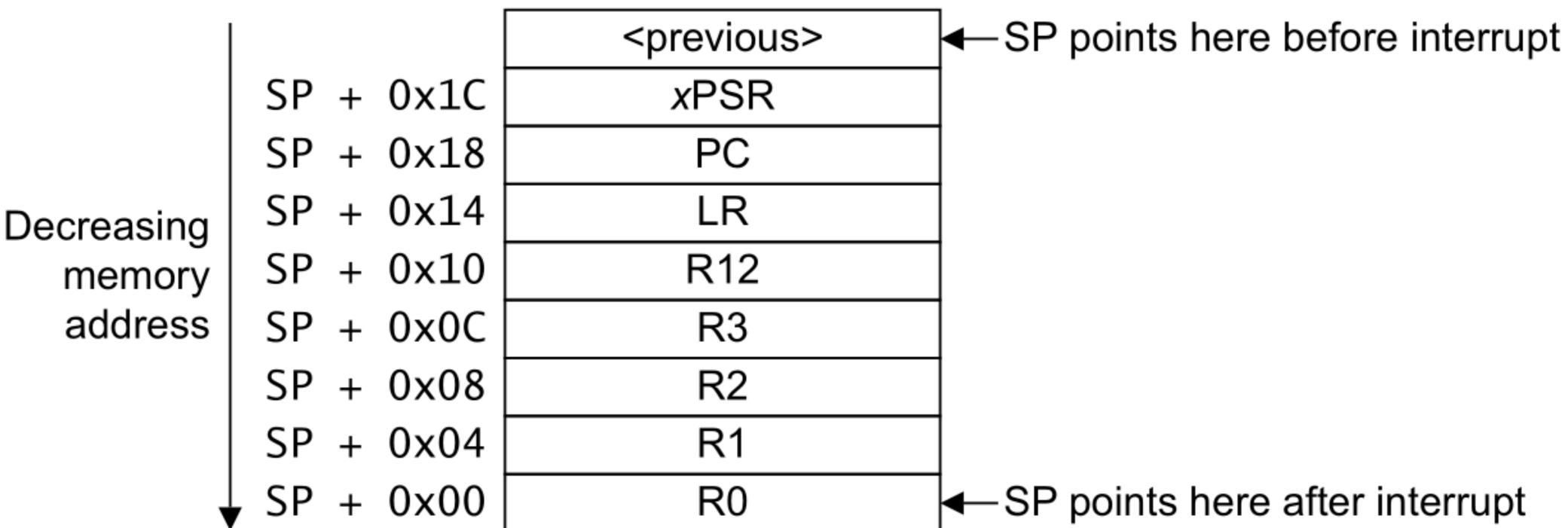


OS TEMPS REEL ORDONNANCEUR

- gérer la liste des tâches
- savoir quelle tâche est active
- sauvegarder le contexte
- choisir la prochaine tâche
- restituer un contexte
- gérer le temps
- fournir un contexte de gestion évènementielle

Sauvegarde de contexte

Pile lors d'une interruption



Il y a une sauvegarde automatique partielle du contexte :

A compléter d'une sauvegarde des registres R4 R5 R6 R7 R8 R9 R10 R11

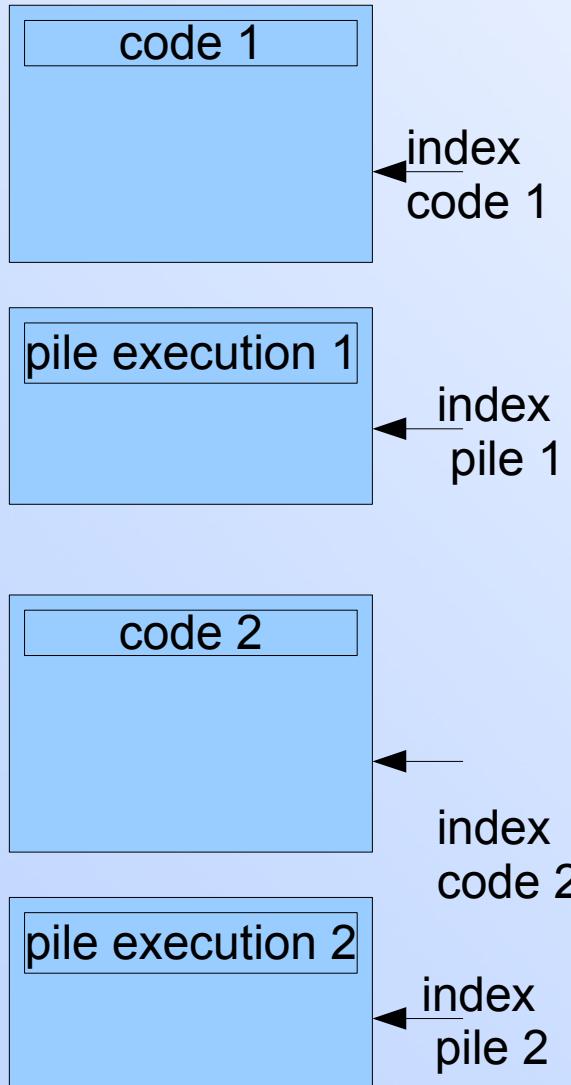
A compléter d'une sauvegarde de ce lieu de stockage pour la future restitution de contexte quand la tâche reprendra la main

comportement spécifique à la cible

- Sauver le contexte de la tâche en cours
 - Sauvegarder où l'on en est du code (R15) **automatique dans la pile**
 - Sauvegarder l'état de tous les registres :
 - de R0 à R14 sauf R13 (inutile de le sauver ici)
 - **R0 à R3, R12, R14 en sauvegarde automatique dans la pile**
 - **R4 à R11 en sauvegarde volontaire**
 - APSR : registre d'état du processeur (Applic. Processor Status Reg)
 - Le gestionnaire de tâches saura où ont été effectuées ces sauvegardes en stockant **R13 (SP)** dans le descripteur de la tâche
- Décider vers quelle tâche basculer : QUI EST PRIORITAIRE/URGENT ?

- Restituer le contexte de la tâche vers laquelle on veut basculer
 - Récupérer du descripteur de tâche l'endroit des sauvegardes
 - Restituer l'état des registres y compris le registre d'état du processeur
 - Faire reprendre le code là où il en était.

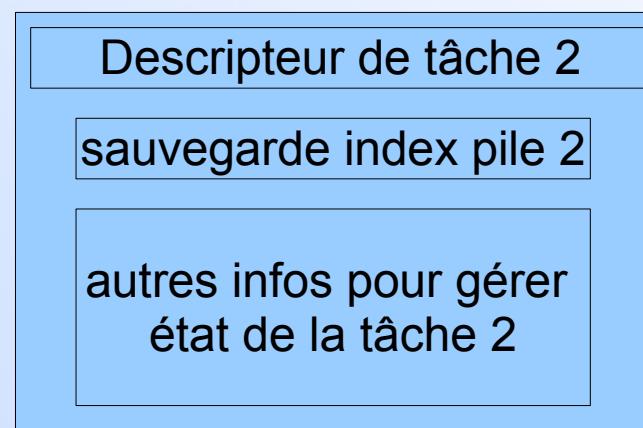
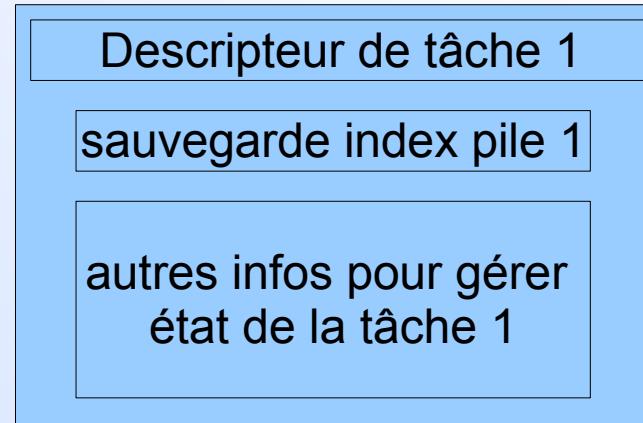
Ressources



P : Process M : Main

PSP(R13)

MSP(R13)



La pile de chaque tâche, à la reprise contient le contexte complet :

- tous les registres dont
- APSR (status register)
- index du code

IT changement de contexte

– sauvegarder le contexte de la tâche 1 sur la pile 1
Automatique, puis complété

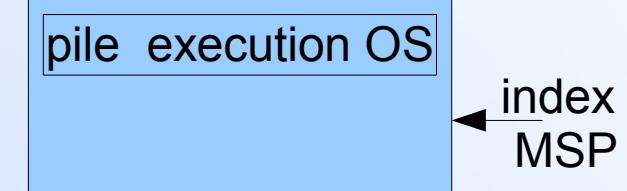
écrire l index de pile 1 dans le descripteur de tâche 1

– choisir la prochaine tâche :

= changer de descripteur

= CHANGER DE PILE PSP

– restituer le contexte de la tâche 2 depuis la pile 2 à la main, puis automatique au retour d'IT change contexte



EI2I4
RTOS 2023

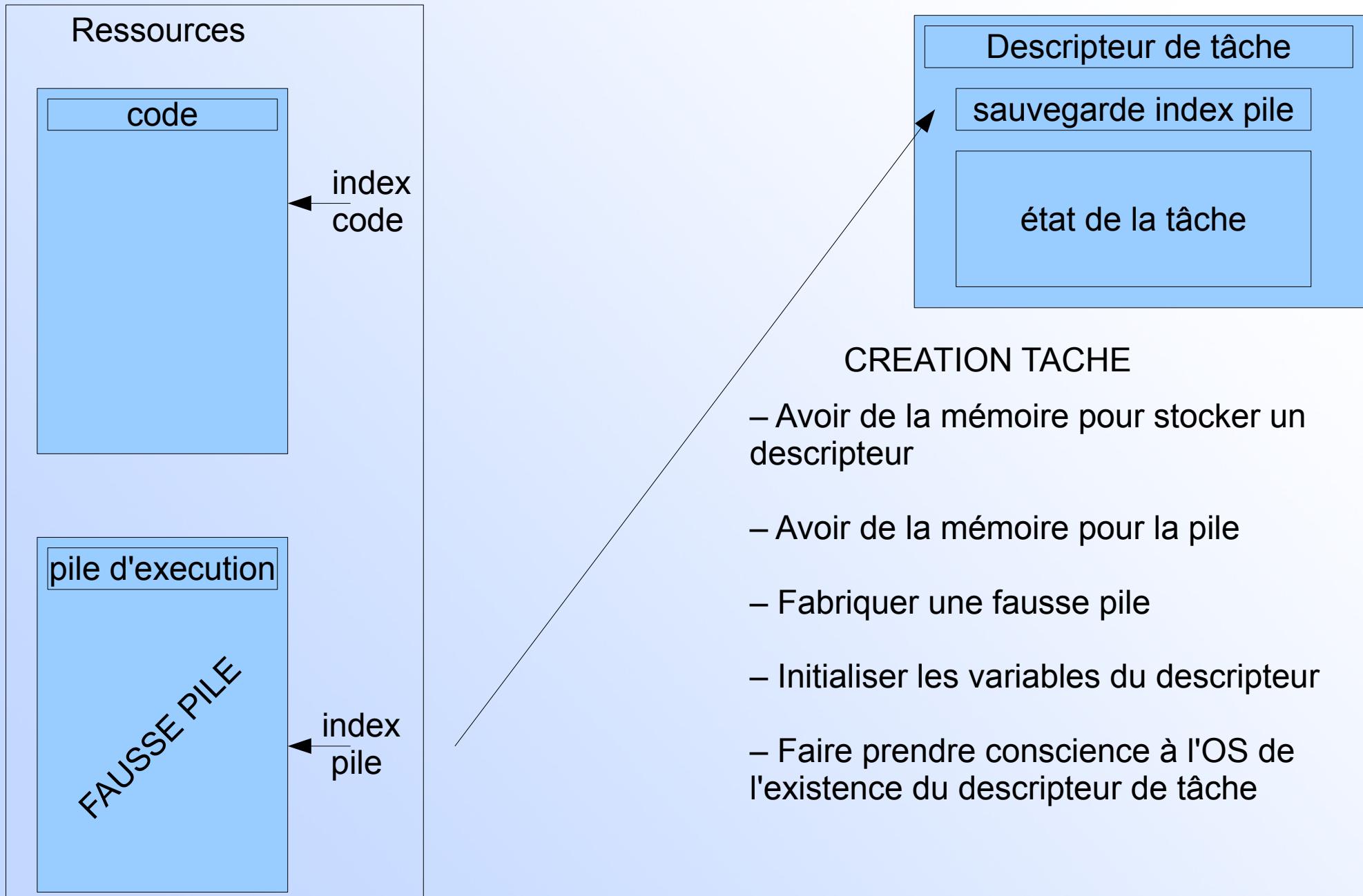
Comportement du compilateur conséquence sur le lancement de tâche

- Simuler un appel à la tâche comme si la première commutation de contexte avait lieu exactement pendant un faux appel de fonction
 - Lors d'un appel d'une fonction avec un paramètre , seul R0 est utilisé
Ce paramètre est un pointeur sur une structure sensée être connue
 - Si la tâche rend la main... ce n'est pas normal, toute tâche est comme une fonction main qui contient une boucle infinie de type while(1)
L'os devra alors reprendre la main :
On doit fabriquer un faux retour de fonction

VOIR EN ANNEXE COMMENT ANALYSER LE COMPORTEMENT du PROCESSEUR

Que faut-il pour créer une tâche ?

Avoir fabriqué une fausse pile dont l index
est sauvé dans le descripteur associé à la tâche



Création d'une Tâche

Ref manuel FREE RTOSpage 34

Prérequis :

- Disposer d'un code à exécuter : Une adresse = nom d'une fonction
- Avoir une liste de paramètres à passer par R0 :
 - Une adresse d'une variable ou d'une structure par ex concevoir l'architecture du PROJET
- Définir sa priorité :
- Anticiper son besoin mémoire :
 - taille de PILE en octets STACK_SIZE
 - Cascade d'appels de fonctions
 - Variables locales de ces fonctions
- Pouvoir agir de manière externe sur la tâche : passer un TaskHandle_t

```
void vTaskCode( void * pvParameters )
{
    Uint32 ul_Ma_var= *pvParameters ;

    while(1)
    {
        // code de la tâche.
    };
}
```

```
#define PRIORITY_1 1
#define STACK_SIZE 128
TaskHandle_t xHandle ;
Static uint32 param = 0x89ABCDEF;
```

```
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &param, PRIORITY_1, &xHandle );
```

- **Etude du comportement de la pile lors des IT :**

but = savoir dans quel ordre les registres sont sauvés
sont ils tous sauvés (nécessaire pour changer de tâche)

ON PEUT AINSI TRAVAILLER LE PROCEDE DE COMMUTATION DE TACHES,
LA FABRICATION DE LA FAUSSE PILE POUR LANCER LA TACHE DEPUIS L'OS,
LA MANIERE DE PASSER LA MAIN A LA PREMIERE TACHE (perte de droits)
attention aux modes superviseur (os) et utilisateur : Piles MSP et PSP

- **Passage des paramètres d'une fonction :**

Permet de savoir quelles valeurs mettre dans la fausse pile pour lancer
la tâche via une restitution de contexte comme s'il y avait eu IT à l'appel
d'une fonction (la tâche) avec des paramètres passés à cette fonction.

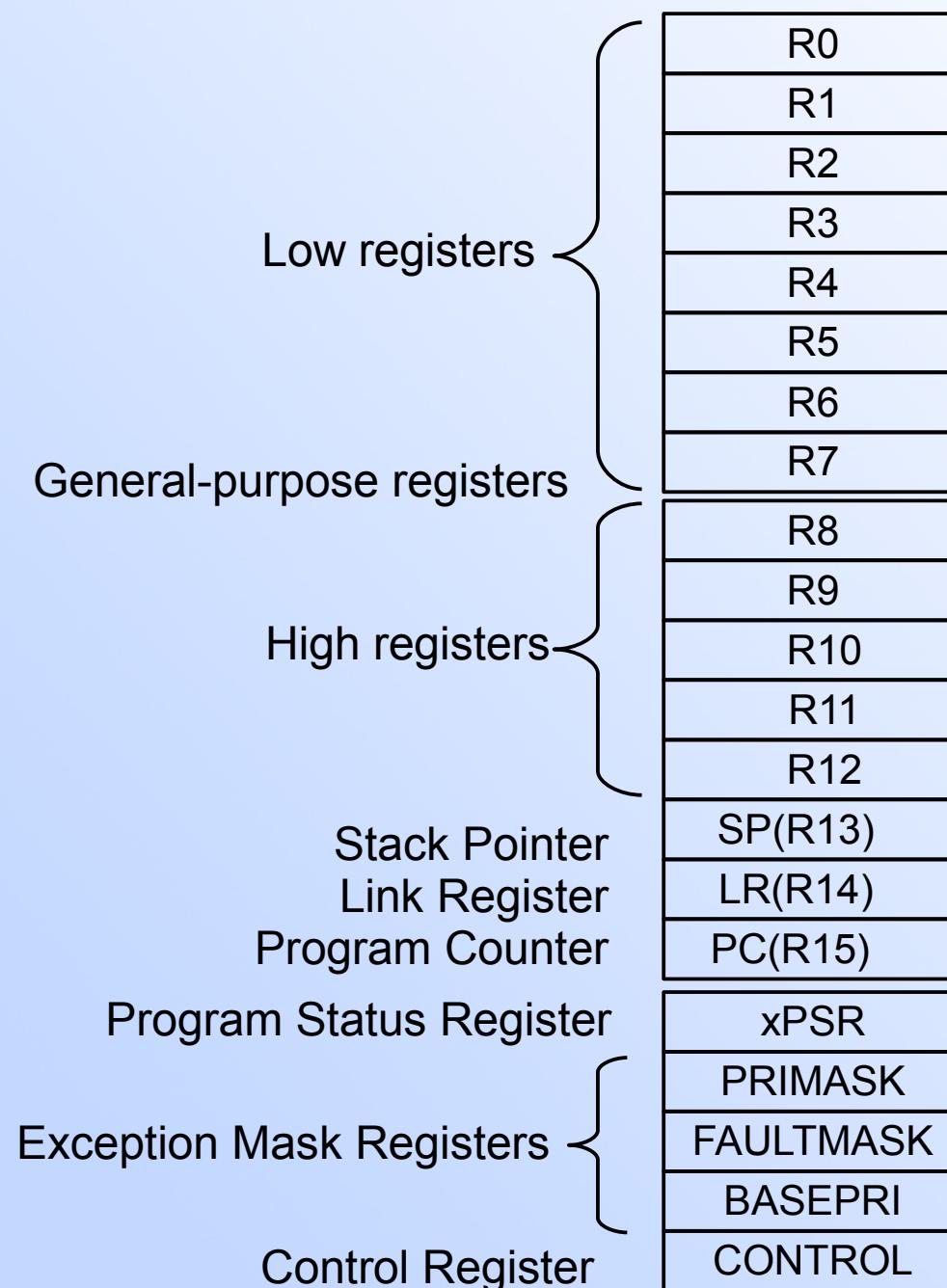
- **Etude des appels de fonction en cascade, et des retours de fonctions**

Si notre tâche rend la main... ce n'est pas normal, toute tâche est comme
une fonction main qui contient une boucle infinie de type while(1)

L'os devra alors reprendre la main :

Utiliser un faux retour de fonction pour redonner la main à l'OS
à une fonction de l'OS qui tuera la tâche et commutera sur une autre

Registres du cœur cortex M3



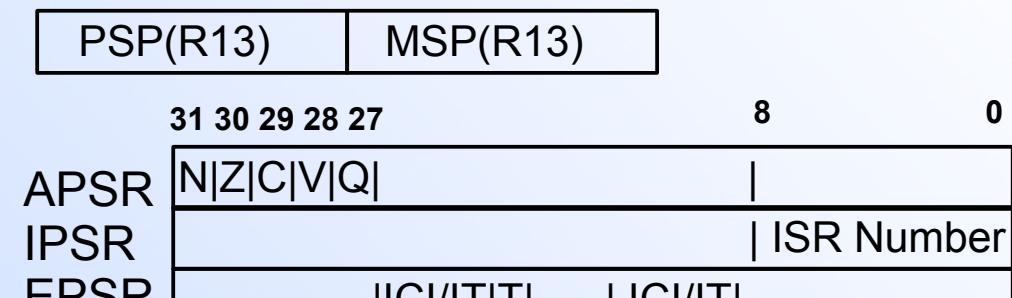
Deux modes de fonctionnement :

Thread mode pour les tâches
Handler mode pour les exceptions

Deux niveaux de Privilèges :

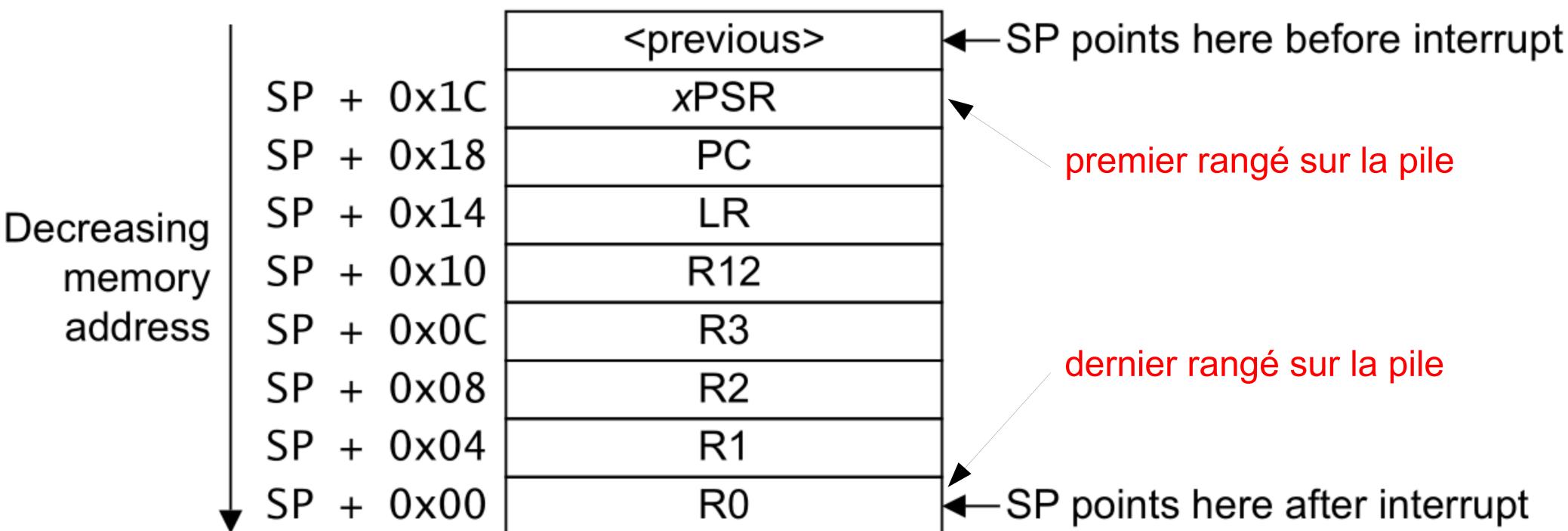
Privileged
Unprivileged

P : Process M : Main



Pile lors d'une interruption

PILE FULL DESCENDING : SP--; puis écriture



Il y a une sauvegarde automatique partielle du contexte :

- minimiser le nombre de cycles nécessaires à la sauvegarde de contexte
- minimiser le nombre de cycles nécessaires à la restitution de contexte
- Pas de sauvegarde en cas de cascade d'IT (on restituera le contexte de la première)
- Sauvegarde/Restitution si une IT interrompt une IT moins prioritaire
- LR est écrasé et contient le mode de retour de l'IT

A compléter d'une sauvegarde MANUELLE des registres R4 R5 R6 R7 R8 R9 R10 R11

Rôle de la valeur de LR en retour d'IT

le contenu de LR permet de savoir si on doit faire lors de l'instruction BX LR :

un retour de fonction : adresse accessible
un retour d'exception : adresse 0xFFFFFFFx

EXC_RETURN	Description
0xFFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

en cas de retour d'IT, le quartet poids faible (LETTRE HEXA) de LR indique :

- si on doit revenir en Thread ou Handler
- si on dépile sur la pile PSP ou MSP
- si on passe en mode Privileged ou Unprivileged

FIN de LANCEMENT
DE L'OS
commuter sur PSP

2023 Appel de fonction: passage des paramètres

```

66: void init_var(unsigned char a, unsigned char b,\n
    unsigned char c, unsigned char d, unsigned char e)\n
0x00000260 B530      PUSH    {r4-r5,lr}\n
67: {uc_a=a;\n
0x00000262 9D03      LDR     r5,[sp,#0x0C]\n
0x00000264 4C1B      LDR     r4,[pc,#108]\n
0x00000266 7020      STRB    r0,[r4,#0x00]\n
68:     uc_b=b;\n
0x00000268 7061      STRB    r1,[r4,#0x01]\n
69:     uc_c=c;\n
0x0000026A 70A2      STRB    r2,[r4,#0x02]\n
70:     uc_d=d;\n
0x0000026C 70E3      STRB    r3,[r4,#0x03]\n
71:     uc_e=e;\n
0x0000026E 7125      STRB    r5,[r4,#0x04]\n
72: }\n
0x00000270 BD30      POP     {r4-r5,pc}\n
73: //----- Main Program -----\n\n
74: int main (void) {\n
0x00000272 B508      PUSH    {r3,lr}\n
75:     init_var(1,2,3,4,5);\n
0x00000274 2005      MOVS    r0,#0x05\n
0x00000276 9000      STR     r0,[sp,#0x00]\n
0x00000278 2304      MOVS    r3,#0x04\n
0x0000027A 2203      MOVS    r2,#0x03\n
0x0000027C 2102      MOVS    r1,#0x02\n
0x0000027E 2001      MOVS    r0,#0x01\n
0x00000280 F7FFFFEE  BL.W   init_var (0x00000260)\n
76:     init_proc();\n

```

adresse	valeur
0x100001C4	
0x100001C8	
0x100001CC	
0x100001D0	
0x100001D4	
0x100001D8	
0x100001DC	
0x100001E0	
0x100001E4	
0x100001E8	
0x100001EC	
0x100001F0	
0x100001F4	
0x100001F8	
0x100001FC	
0x10000200	
<i>r₀</i>	
<i>r₁</i>	
<i>r₂</i>	
<i>r₃</i>	
<i>r₄</i>	
<i>r₅</i>	
<i>r₆</i>	
<i>r₇</i>	
<i>r₈</i>	
<i>r₉</i>	
<i>r₁₀</i>	
<i>r₁₁</i>	
<i>r₁₂</i>	
<i>r₁₃ (sp)</i>	
<i>r₁₄ (lr)</i>	
<i>r₁₅ (pc)</i>	

Quatre premiers paramètres
dans R0, R1, R2, R3
les suivants dans la pile

les Registres hors paramètres (R0 -R3) doivent être préservés pour le retour

Un rôle de la pile : Adresse de retour

```

54: void init_gpio(void)
55: { LPC_SC->PCOMP |= (1 << 15);
0x0000066E 4838 LDR r0,[pc,#224] ; @0x00000750
0x00000670 6800 LDR r0,[r0,#0x00]
0x00000672 F4404000 ORR r0,r0,#0x8000
0x00000676 4936 LDR r1,[pc,#216] ; @0x00000750
0x00000678 39C4 SUBS r1,r1,#0xC4
0x0000067A F8C100C4 STR r0,[r1,#0xC4]
56: LPC_GPIO3->FIODIR |= 0x03<<25;
0x0000067E 4833 LDR r0,[pc,#204] ; @0x0000074C
0x00000680 6E00 LDR r0,[r0,#0x60]
0x00000682 F04060C0 ORR r0,r0,#0x60000000
0x00000686 4931 LDR r1,[pc,#196]
0x00000688 6608 STR r0,[r1,#0x60]
57: LPC_GPIO0->FIODIR |= 0x01<<22;
0x0000068A 4608 MOV r0,r1
0x0000068C 6800 LDR r0,[r0,#0x00]
0x0000068E F4400080 ORR r0,r0,#0x40000000
0x00000692 6008 STR r0,[r1,#0x00]
58: }
0x00000694 4770 BX lr
59: //-----
60: void init_proc()
61: { init_gpio();
0x00000696 B500 PUSH {lr}
0x00000698 F7FFFFE9 BL.W init_gpio (0x0000066E)
62: init_timer0();;
0x0000069C F7FFFFB5 BL.W init_timer0 (0x0000060A)
63: }
0x000006A0 BD00 POP {pc} 64:
65: //----- Main Program -----
66: int main (void) {
67: init_proc();
0x000006A2 F7FFFF8 BL.W init_proc (0x00000696)

```

adresse	valeur
0x100001C4	
0x100001C8	
0x100001CC	
0x100001D0	
0x100001D4	
0x100001D8	
0x100001DC	
0x100001E0	
0x100001E4	
0x100001E8	
0x100001EC	
0x100001F0	
0x100001F4	
0x100001F8	
0x100001FC	
0x10000200	

Séquence d'appels de fonctions