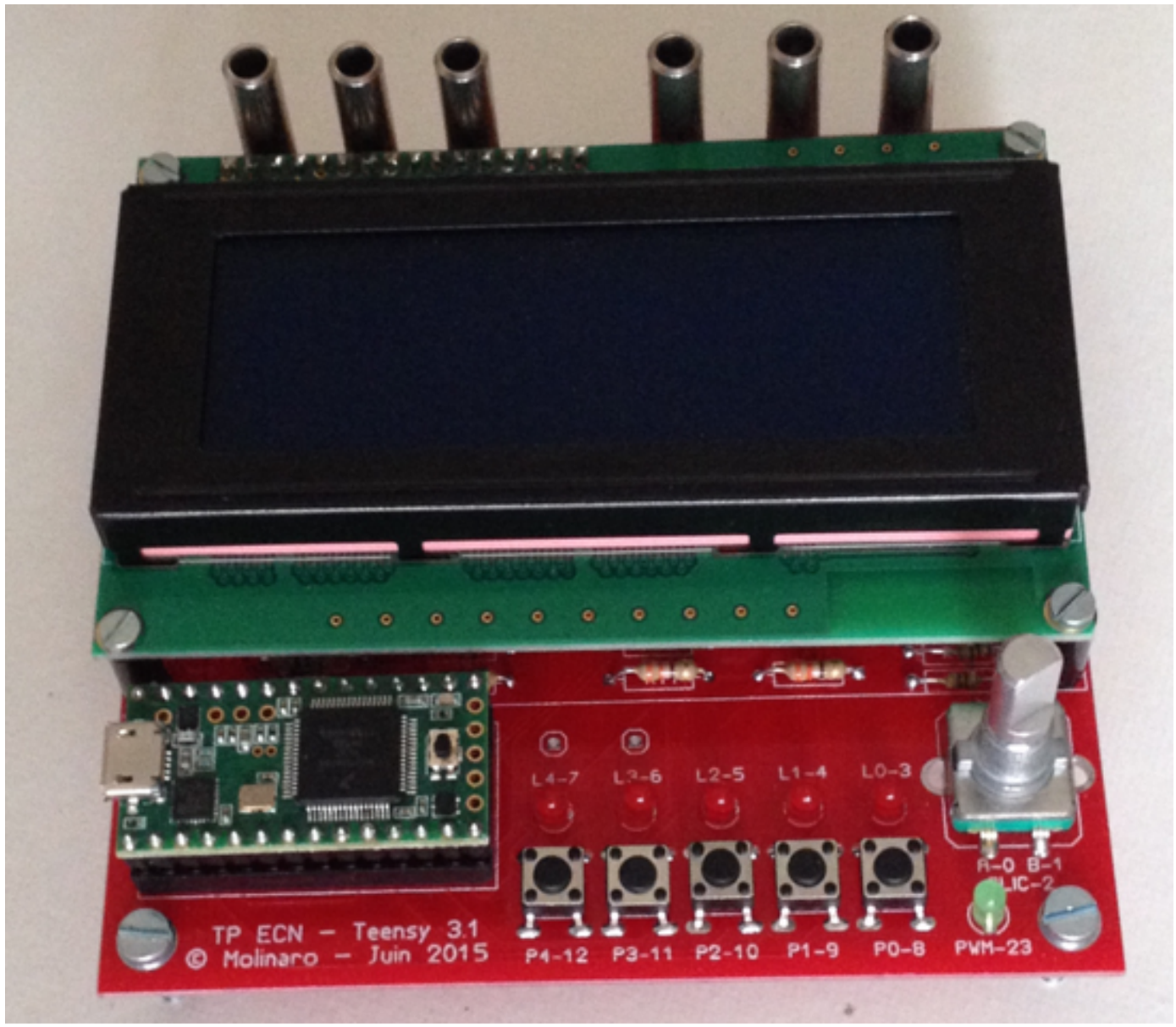
Temps Réel


L4-7
L3-6
L2-5
L1-4
L0-3
R-0 B-1
LIC-2
TP ECN - Teensy 3.1
© Molinaro - Juin 2015
P4-12
P3-11
P2-10
P1-9
P0-8
PWM-23

But de cette partie

Objectif :

- *utiliser un Service Call pour assurer l'atomicité de l'incrémentation d'une variable.*

Problèmes à résoudre :

- mise en place d'un gestionnaire d'appel système configurable.

Travail à faire :

Réaliser un programme qui incrémente quatre variables globales :

- dans une routine d'interruption périodique ;
- dans la routine `loop`, par l'intermédiaire de quatre appels de service ;

Au bout de 5 secondes, la routine d'interruption périodique est désinstallée et les quatre variables sont affichées.

L'instruction `svc #n`

Les processeurs Cortex-M4 possèdent une instruction `svc #n`, où `n` est un entier non signé sur 8 bits (0 à 255). `svc` signifie *Supervisor Call*.

L'exécution de l'instruction `svc` déclenche l'interruption n°11. La valeur de `n` est ignorée par le processeur.

Les registres du processeur Cortex-M4 (simplifié)

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)

PSR

Les registres R0 à R12 sont des registres destinés à recevoir des données.

Le registre R13 est le *Stack Pointer*, R14 est le *Link Register* (il reçoit l'adresse de retour de sous-programme), R15 est le *Program Counter* (désigne l'instruction à exécuter).

PSR est le *Program Status Register*.

Ceci est une description partielle ; pour une description complète, voir :

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CHDBIBGJ.html.

Le registre PSR

Numéro	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Nom	N	Z	C	V	Q	ICI	IT	T	—	—	—	—	GE				*Continuation status*						—	ISR_NUMBER								
Valeur initiale	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Les indicateurs `N` (ne*gative*), `Z` (*zero*), `C` (c*arry*) et `V` (*overflow*) retiennent les résultats des opérations arithmétiques.

Le bit `T` (*Thumb*) doit toujours être à 1 pour un Cortex-M4.

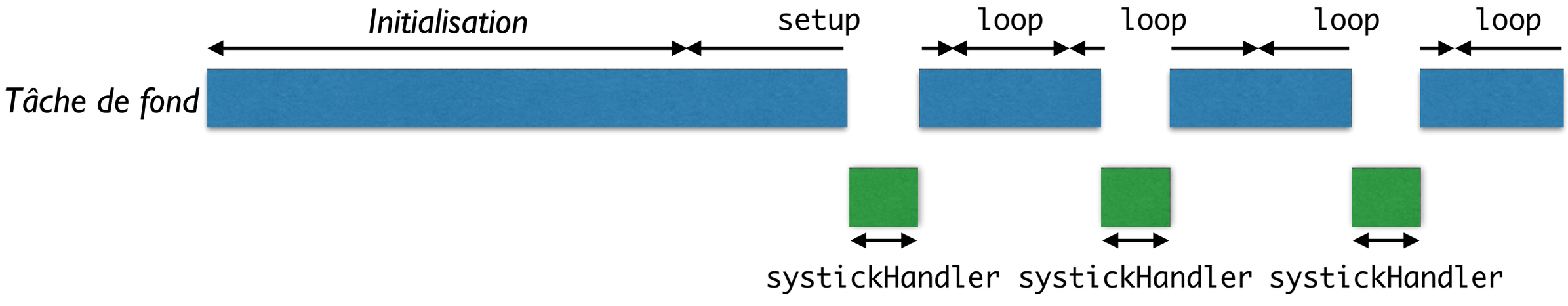
Le champ `ISR_NUMBER` indique le numéro de l'interruption en cours d'exécution ; `0` signifie aucune interruption : c'est le *Thread Mode*, le mode d'exécution des *threads* dans un exécutif, des routines `setup` et `loop` dans les programmes. **Quand `ISR_NUMBER` ≠ `0`, le processeur ignore les interruptions.**

Les autres bits ne sont pas décrits, on n'a pas besoin de connaître leur signification dans ce cours.

Lien :

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CHDBIBGJ.html.

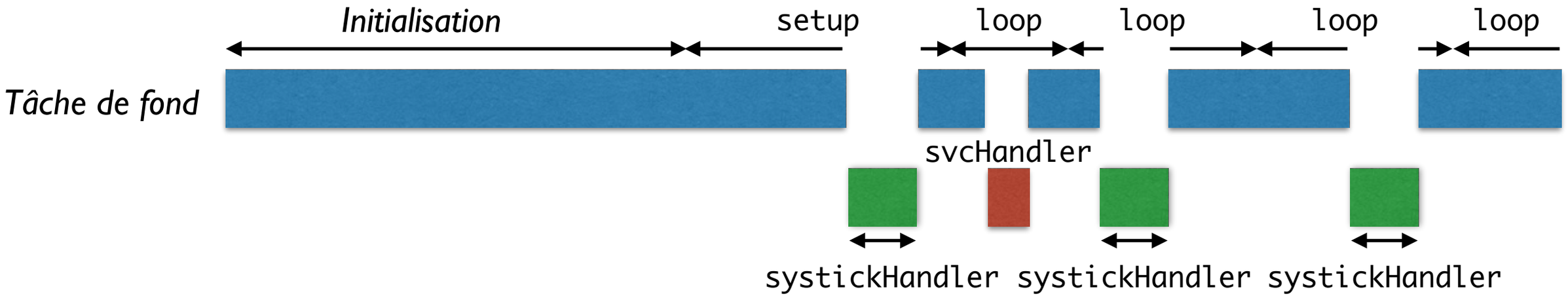
Exécution des programmes (valable pour les programmes 01 à 07)



ISR_NUMBER	0	15	0	15	0	15	0

Quand le `systickHandler` s'exécute, l'`ISR_NUMBER` vaut 15, c'est le numéro de l'interruption `sysTick` défini lors de la conception du processeur Cortex-M4.

Exécution des programmes (valable pour les programmes 08 à 10)

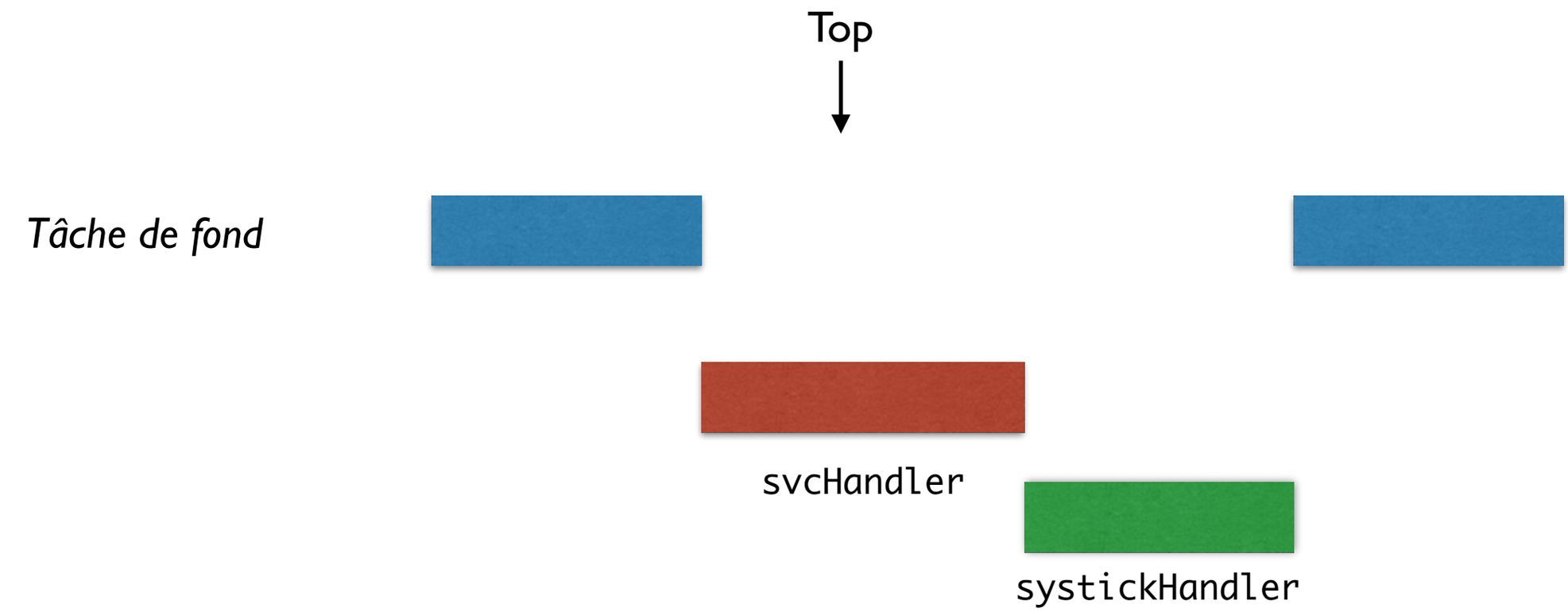


ISR_NUMBER	0	15	0	11	0	15	0	15	0

Quand le `svcHandler` s'exécute, l'`ISR_NUMBER` vaut `11`, c'est le numéro de l'interruption svc défini lors de la conception du processeur Cortex-M4.

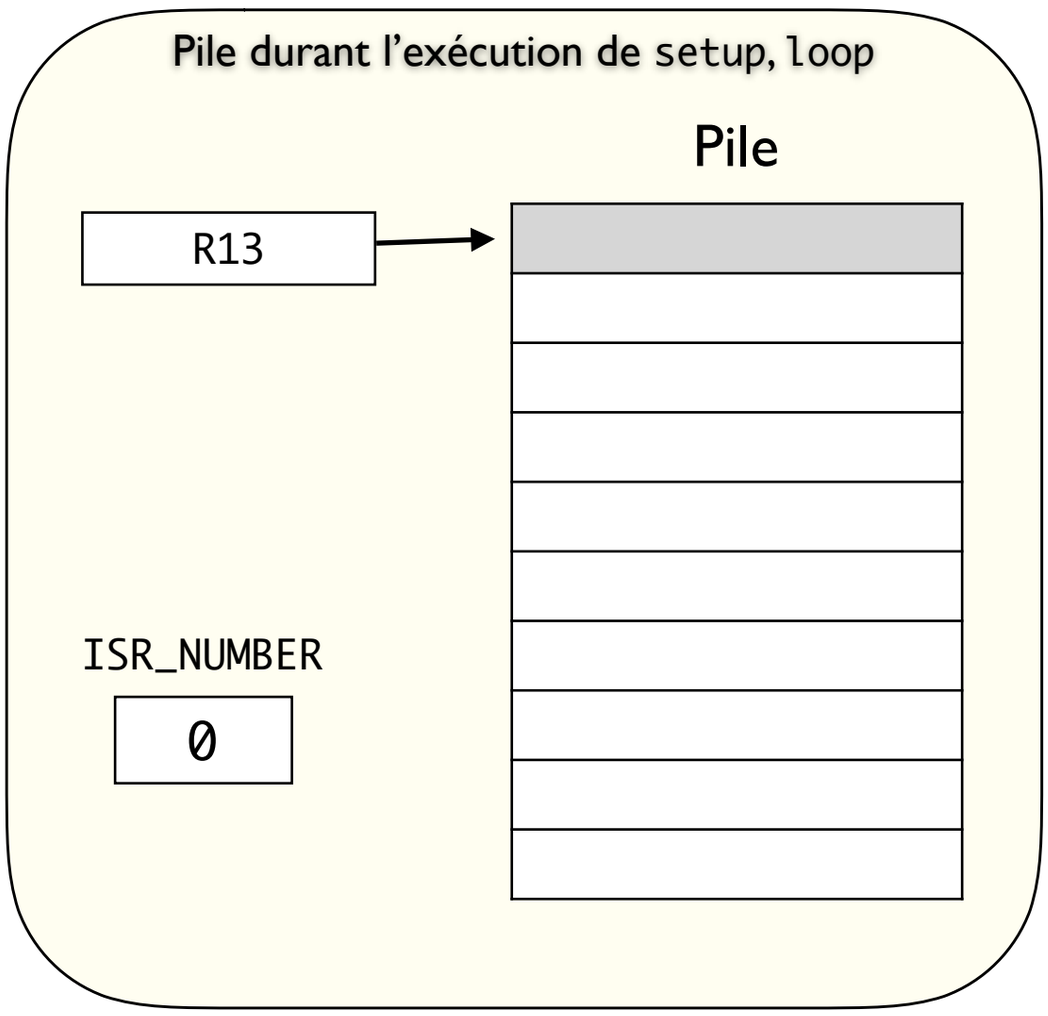
Une routine d'interruption n'est pas interruptible

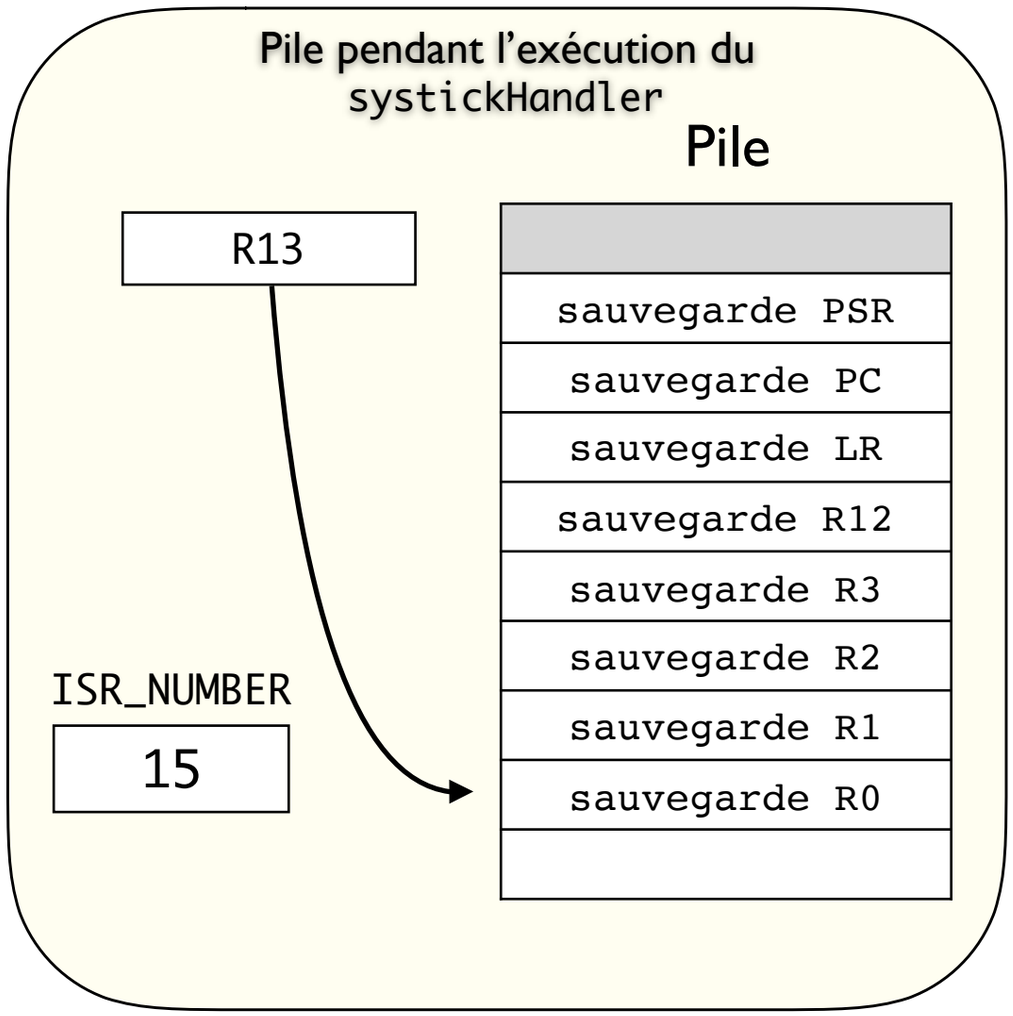
Si durant l'exécution du `svcHandler`, le signal de déclenchement du `systickHandler` survient, l'exécution du `systickHandler` est retardée jusqu'à la fin de l'exécution du `svcHandler`.



Interruption et pointeur de pile (valable pour les programmes 01 à 08)

Quand l'interruption sysTick survient, le contenu des registres PSR, PC, LR, R12, R3, R2, R1, R0 est empilé ; ensuite, le processeur va exécuter le systickHandler. Quand celui-ci est terminé, le contenu de ces registres est restitué.





Utiliser `svc` pour exécuter différents services

Dans le programme à écrire (n°08), il y a quatre services à écrire :

- l'incrémentation de la 1re variable globale `gCompteurA`,
- l'incrémentation de la 2e variable globale `gCompteurB`,
- l'incrémentation de la 3e variable globale `gCompteurC`,
- l'incrémentation de la 4e variable globale `gCompteurD`.

Or, il n'y a qu'une seule instruction `svc`.

Astuce : l'instruction `svc` présente un argument `#n` (`0 ≤ n ≤ 255`), qui n'est pas utilisé par le processeur. On va utiliser des valeurs particulières de `n` pour différencier les différents services :

- `svc #0` réalisera l'incrémentation de la 1re variable globale `gCompteurA`,
- `svc #1` réalisera l'incrémentation de la 2e variable globale `gCompteurB`,
- `svc #2` réalisera l'incrémentation de la 3e variable globale `gCompteurC`,
- `svc #3` réalisera l'incrémentation de la 4e variable globale `gCompteurD`.

Mais : l'instruction `svc` ne peut être appelée qu'en assembleur.

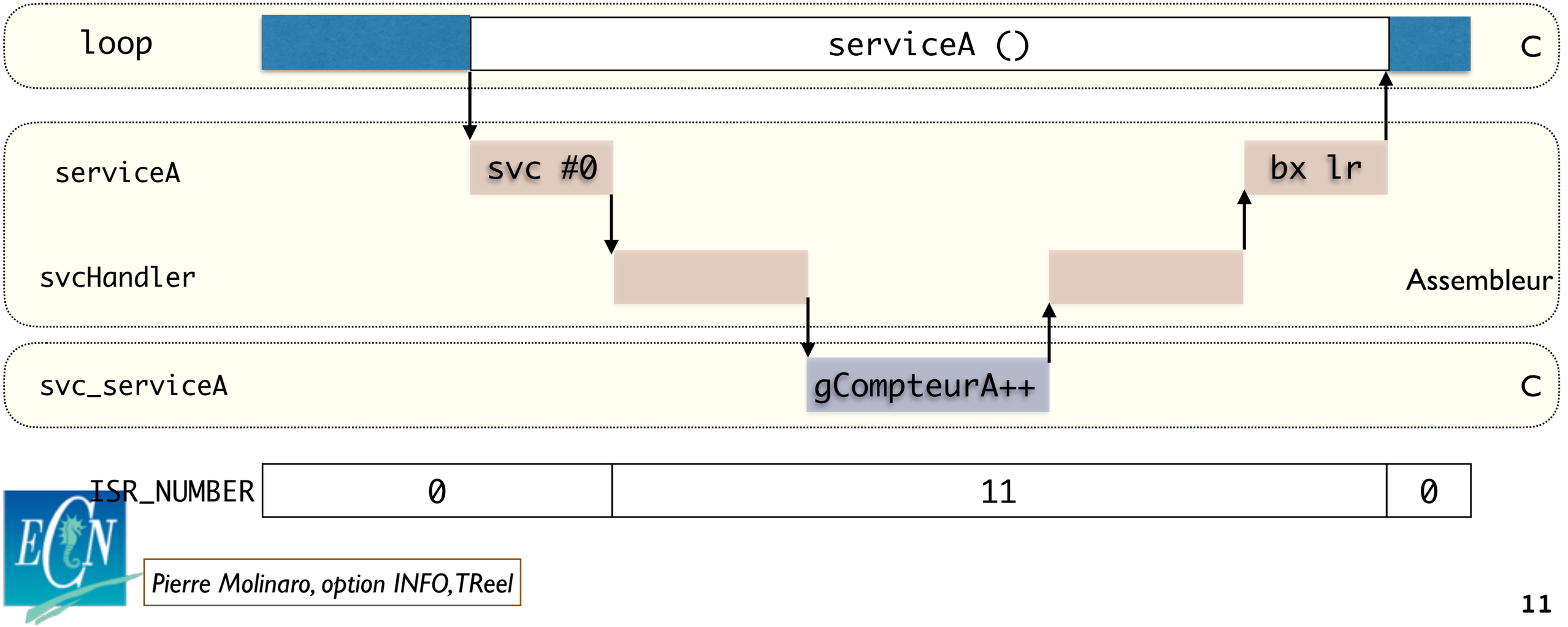
Utiliser svc pour exécuter différents services

Dans le programme 07, 2e étape, la routine loop s'écrivait :

```
void loop (void) {
  gCompteurA ++ ;
  gCompteurB ++ ;
  gCompteurC ++ ;
  gCompteurD ++ ;
  ……
}
```

Dans le programme 08, on appelle les routines de service au lieu d'incrémenter directement les variables.

```
void loop (void) {
  serviceA () ;
  serviceB () ;
  serviceC () ;
  serviceD () ;
  ……
}
```



Adopter un nommage systématique

La routine XYZ appelle, de manière indirecte via le svcHandler, la routine svc_XYZ.

Il faut donc écrire un fichier d'en-tête svc-handler.h qui déclare les prototypes des routines de service :

```
void serviceA (void) ;
void svc_serviceA (void) ;

void serviceB (void) ;
void svc_serviceB (void) ;

void serviceC (void) ;
void svc_serviceC (void) ;

void serviceD (void) ;
void svc_serviceD (void) ;
```

La routine XYZ est appelée en C et écrite en assembleur.

La routine svc_XYZ est appelée en assembleur et écrite en C.

Écriture des routines `svc_service`*N*

Vous écrivez ces routines dans `setup-loop.c` :

```
void svc_serviceA (void) {
  gCompteurA ++ ;
}

void svc_serviceB (void) {
  gCompteurB ++ ;
}

void svc_serviceC (void)  {
  gCompteurC ++ ;
}

void svc_serviceD (void)  {
  gCompteurD ++ ;
}
```

Écriture des routines en assembleur

La structure est choisie de façon à vous simplifier le travail, vous n'avez pas besoin de connaître l'assembleur ARM.

Le fichier à compléter est `svc-handler-single-stack.s` (sur le serveur pédagogique).
Pour chaque service, vous avez deux opérations à faire :

(1) écrire la routine XYZ en assembleur ;

(2) compléter le tableau `__svc_dispatcher_table` par l'appel de la routine `svc_XYZ`.

```
  .global serviceA
  .type serviceA, %function
serviceA:
  svc #0
  bx  lr

  .global serviceB
  .type serviceB, %function
serviceB:
  svc #1
  bx  lr
```

```
__svc_dispatcher_table:
  .word svc_serviceA @ 0
  .word svc_serviceB @ 1
```

La routine `service0` appelle `svc` avec l'argument `i`, la routine `svc_service0` doit être l'entrée n°`i` de la table `__svc_dispatcher_table`.

Commentaires en assembleur ARM

Le caractère « @ » délimite un commentaire jusqu'à la fin de la ligne courante.

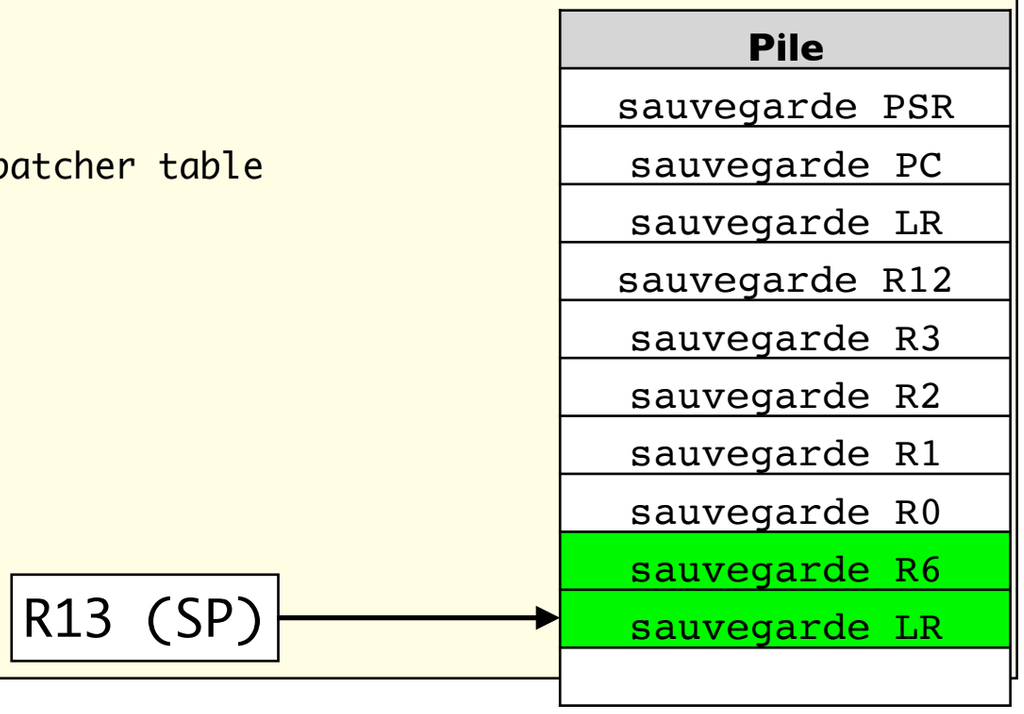
Respectez l'indentation : seule la définition d'un symbole doit commencer en colonne 1.

```
  .global serviceA  @ Le symbole serviceA est visible lors de l'édition de lien
  .type serviceA,   @ function @ Obligatoire pour signifier que serviceA est une fonction
serviceA:           @ Point d'entrée de la fonction serviceA (doit commencer en colonne 1)
  svc #0            @ Instruction System Call
  bx  lr            @ Retour à l'appelant
```

Comment est écrit `svcHandler` (programme `08`)

Vous n'avez pas à l'écrire, il est fourni à la fin du fichier `svc-handler-single-stack.s` (sur le serveur pédagogique).

```
  .global svcHandler
  .type svcHandler, %function

svcHandler:
@--- Save preserved registers
  push  {r6, lr}
@--- r12 <- address of dispatcher table
  ldr   r6, =__svc_dispatcher_table
  mov   r12, r6
@--- r6 <- Address of SVC instruction
  ldr   r6, [sp, #32] @ R6 <- Address of instruction after SVC (32 : offset of stacked PC)
  sub   r6, #2        @ R6 <- Address of SVC instruction
@--- r6 <- bits 0-7 of SVC instruction (LDRB loads a byte from memory and zero-extends the byte to a 32-bit word)
  ldrb  r6, [r6]
@--- r6 <- address of routine to call
  lsl   r6, #2    @ Multiply by 4, entries are 4 bytes words
  add   r6, r12   @ R6 <- Address of selected service entry in dispatcher table
@--- r6 <- address of routine to call
  ldr   r6, [r6]  @ R6 <- Address of selected service
@--- Call routine
  blx   r6
@--- Write return value in stack
  mov   r6, sp        @ R6 <- SP
  add   r6, #8        @ r6 points to saved R0
  stm   r6!, {r0, r1} @ Write r0, r1 to saved registers
@--- Return from interrupt
  pop   {r6, pc}
```



Comment est appelé svcHandler

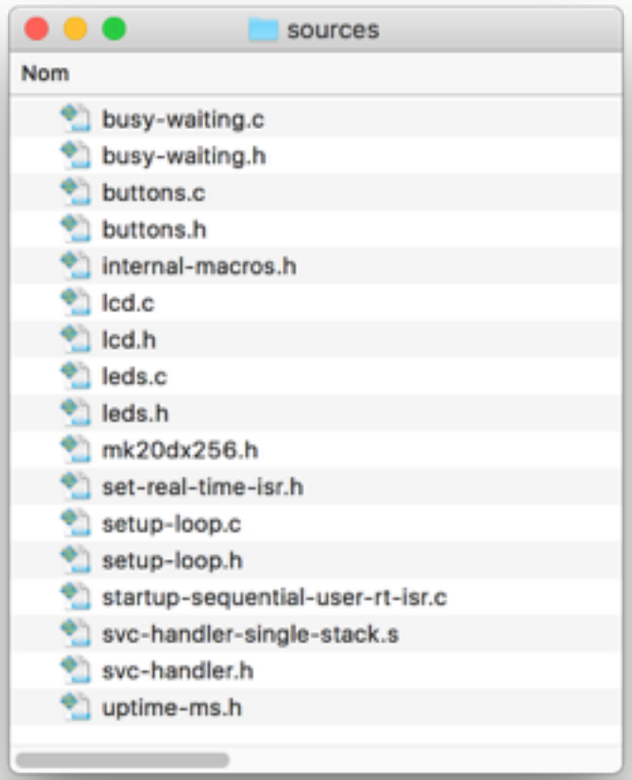
Vous avez à inscrire l'appel de svcHandler dans le vecteur n°11 de la table des vecteurs d'interruption (fichier startup-sequential-user-rt-isr.c).

```
void svcHandler (void) ; // Indispensable pour satisfaire la compilation C

//-------------------------------------------------------------------------------------------

const vectorStructSeq vector __attribute__ ((section (".isr_vector"))) = {
  & __system_stack_end, // 0
//--- ARM Core System Handler Vectors
  { ResetISR, // 1
    NULL, // 2
    NULL, // 3
    NULL, // 4
    NULL, // 5
    NULL, // 6 (Usage fault)
    NULL, // 7
    NULL, // 8
    NULL, // 9
    NULL, // 10
    svcHandler, // 11 (System call)
    NULL, // 12
    NULL, // 13
    NULL, // 14
    systickHandler // 15 (SysTick)
  },
//--- Non-Core Vectors
  { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
    ……
```

Travail à faire

Première étape :

- dupliquer le programme précédent (étape 2) et le renommer ;
- récupérer sur le serveur pédagogique l'archive `08-sources.tbz` qui contient `svc-handler-single-stack.s` ;
- compléter ce fichier ;
- écrire le fichier `svc-handler.h` ;
- dans le fichier `startup-sequential-user-rt-isr.c`, inscrire la routine `svcHandler` dans le vecteur n°11.



Résultat attendu

Les quatre variables globales sont incrémentées de manière atomique, on obtient à chaque fois quatre valeurs identiques. Ces valeurs peuvent changer d'une exécution à l'autre.