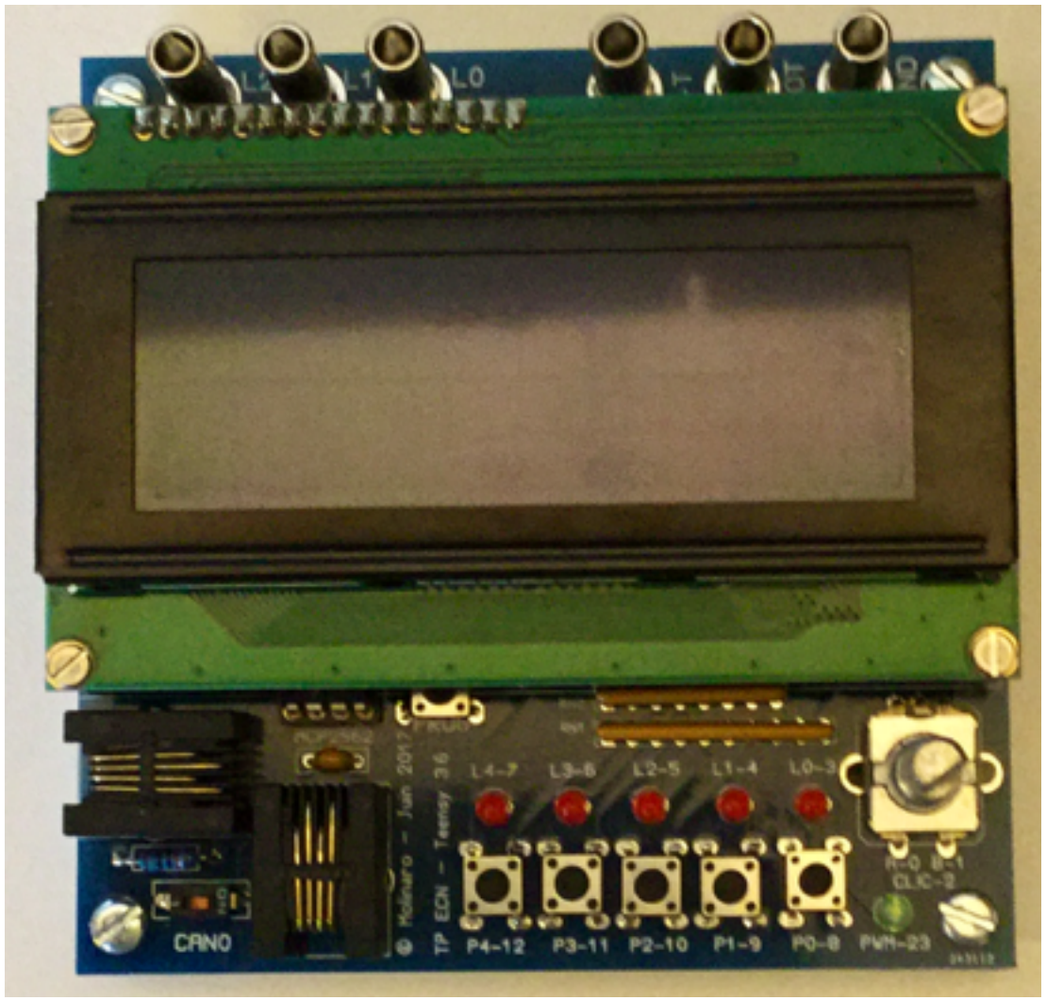
Temps Réel



Programme *11-system-call*

CENTRALE NANTES

Pierre Molinaro

Juin 2018

But de cette étape

Utiliser une instruction svc (Supervisor Call) pour effectuer la seconde phase d'initialisation.

Cette étape a pour principal objectif de montrer comment l'instruction `svc` fonctionne.

Dans un exécutif, cette instruction est importante, elle permet à une tâche d'invoquer les services de l'exécutif.

CENTRALE NANTES

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, mais peut être exploitée par la routine d'interruption. Dans cette étape, la valeur de *n* sera ignorée.

Lien :

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

Code contenu dans `reset-handler-sequential.s`

Étapes précédentes

Dans toutes les étapes précédentes, le *reset handler*, c'est-à-dire le code exécuté au démarrage est le suivant :

```
reset.handler: @ Cortex M4 boots with interrupts enabled, in Thread mode
@---------------------------------- Run boot, zero bss section, copy data section
  bl    start.phase1
@---------------------------------- Set PSP: this is stack for background task
  ldr   r0,  =background.task.stack + BACKGROUND.STACK.SIZE
  msr   psp, r0
@---------------------------------- Set CONTROL register (see §B1.4.4)
@ bit 0 : 0 -> Thread mode has privileged access, 1 -> Thread mode has unprivileged access
@ bit 1 : 0 -> Use SP_main as the current stack, 1 -> In Thread mode, use SP_process as the current stack
@ bit 2 : 0 -> FP extension not active, 1 -> FP extension is active
  movs  r2, #2
  msr   CONTROL, r2
@--- Software must use an ISB barrier instruction to ensure a write to the CONTROL register
@ takes effect before the next instruction is executed.
  isb
@---------------------------------- Run init routines, interrupt disabled
  cpsid i               @ Disable interrupts
  bl    start.phase2
  cpsie i               @ Enable interrupts
@---------------------------------- Run setup, loop
  bl    setup.function
background.task:
  bl    loop.function
  b     background.task
```

Code contenu dans `reset-handler-sequential-step11.s`

Uniquement cette étape

Le seconde phase d'initialisation s'exécute via une instruction `svc` :

```
reset.handler: @ Cortex M4 boots with interrupts disabled, in Thread mode
@---------------------------------- Run boot, zero bss section, copy data section
  bl    start.phase1
@---------------------------------- Set PSP: this is stack for background task
  ldr   r0,  =background.task.stack + BACKGROUND.STACK.SIZE
  msr   psp, r0
@---------------------------------- Set CONTROL register (see §B1.4.4)
@ bit 0 : 0 -> Thread mode has privileged access, 1 -> Thread mode has unprivileged access
@ bit 1 : 0 -> Use SP_main as the current stack, 1 -> In Thread mode, use SP_process as the current stack
@ bit 2 : 0 -> FP extension not active, 1 -> FP extension is active
  movs  r2, #2
  msr   CONTROL, r2
@--- Software must use an ISB barrier instruction to ensure a write to the CONTROL register
@ takes effect before the next instruction is executed.
  isb
@---------------------------------- Run init routines, from SVC handler
  svc   #0
@---------------------------------- Run setup, loop
  bl    setup.function
background.task:
  bl    loop.function
  b     background.task
```

Il faudra donc écrire du code pour le handler associé exécute la seconde phase d'initialisation.

Travail à faire

Dupliquer le répertoire de l'étape précédente, et renommez-le par exemple **11-system-call**.

Renommer le fichier **reset-handler-sequential.s** issu de l'étape précédente en **reset-handler-sequential-step11.s**, et effectuer la modification pour appeler `svc #0`.

Note : si vous exécutez le code tel quel, un message d'erreur s'affichera aussitôt, puisque le handler associé à `svc` n'est pas défini.

Écrivez dans un fichier **svc-handler-step11.cpp** le handler de l'interruption associée à `svc`, et déclarer le dans un fichier **svc-handler-step11.h**.

Les registres du processeur Cortex-M4 (simplifié)

Pour ceux qui veulent en savoir plus

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.

CENTRALE NANTES

Le registre PSR

Pour ceux qui veulent en savoir plus

Indice	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	IC	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 (negative), Z (zero), C (carry) 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`, et avec les réglages qui sont adoptés dans ce cours, le processeur ignore les interruptions.

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

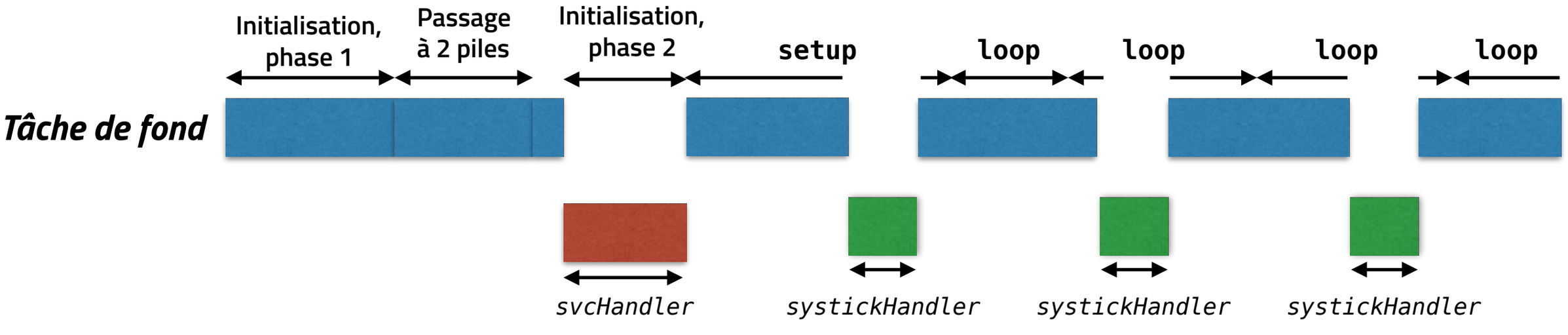
Lien :

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

CENTRALE NANTES

Exécution des programmes

Pour ceux qui veulent en savoir plus, pile utilisée et `ISR_NUMBER`



Le micro-contrôleur démarre en mode *pile unique*, et effectue dans ce mode la première phase d'initialisation. Ensuite, il est programmé en mode *double pile*, de façon que la tâche de fond et les handlers d'interruption s'exécutent sur des piles distinctes.