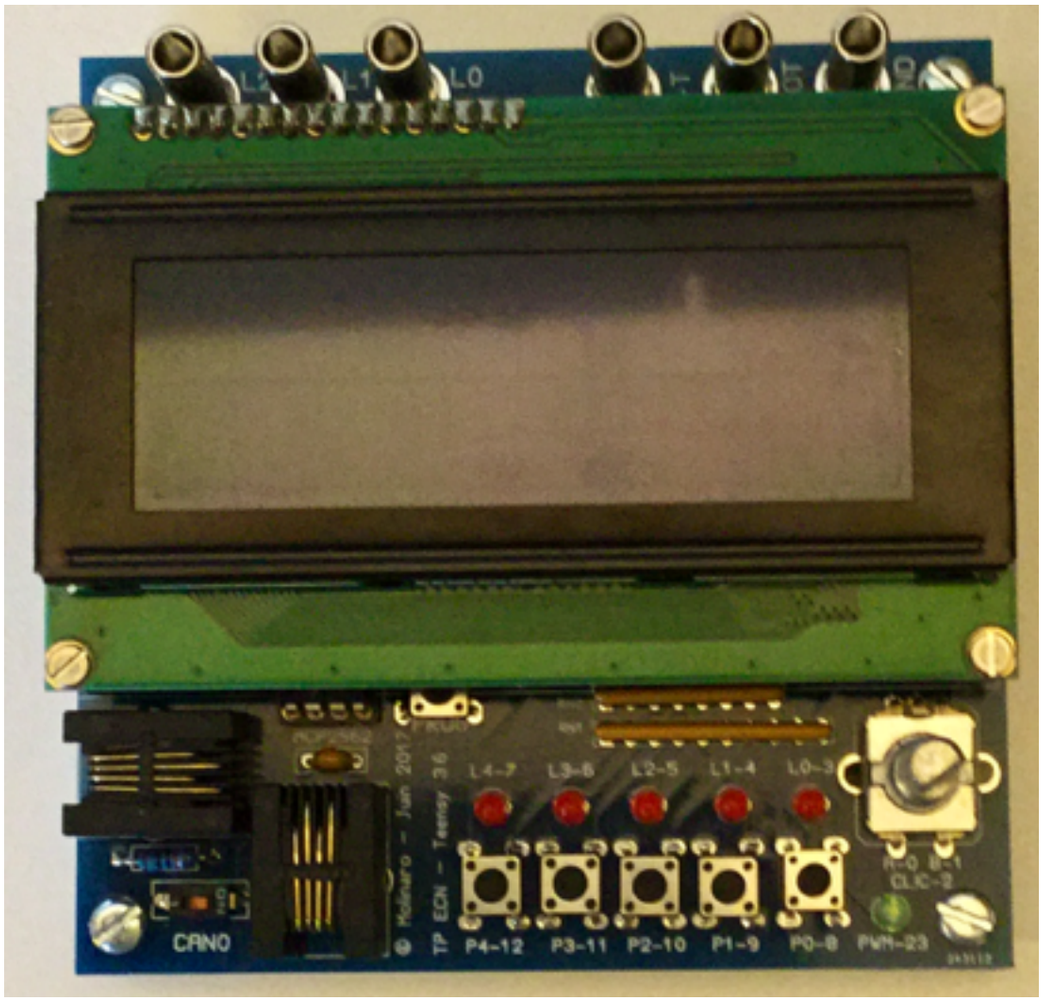
Temps Réel



Programme *12-first-real-time-kernel*

CENTRALE NANTES

Pierre Molinaro
Juin 2018

Description de cette étape

Premier exécutif ! Évidemment, ses possibilités sont très réduites :

- une seule tâche ;
- qui ne doit pas se terminer.

Dans les étapes suivantes, on ajoutera progressivement des propriétés.

Dans cette étape, nous allons d'abord décrire les opérations pratiques à réaliser, et ensuite nous décrirons la structure de l'exécutif utilisé dans ce cours.

Travail à faire (1/3)

Dupliquer le programme de l'étape précédente, et renommez-le par exemple **12-first-real-time-kernel**.

Ouvrez aussi l'archive **12-files.tar.bz2**. Recopiez dans le répertoire **sources** de votre programme :

- **reset-handler-xtr.s** ;
- **task-list--32-tasks.h** et **task-list--32-tasks.cpp** ;
- **xtr.h** et **xtr.cpp** ;
- **user-tasks.cpp**.

Supprimez de votre projet les fichiers suivants :

- **reset-handler-sequential-step11.s** (il est remplacé par **reset-handler-xtr.s**) ;
- **svc-handler-step11.h** et **svc-handler-step11.cpp** (le *svc handler* est automatiquement engendré et placé dans **zSOURCES/interrupt-handlers.s**) ;
- **setup-loop.h** et **setup-loop.cpp** (ils sont remplacés par **user-tasks.cpp**) ; à partir de cette étape, plus de fonction **setup** ni **loop**, mais des tâches.

Travail à faire (2/3)

Il faut aussi modifier le fichier **makefile.json** ; il doit maintenant avoir l'allure suivante :

```
{ "SOURCE-DIR" : ["sources"],
  "TEENSY" : "3.6",
  "CPU-MHZ" : 180,
  "TASK-COUNT" : 1,
  "SERVICE-SCHEME" : "svc",
  "SECTION-SCHEME" : "disableInterrupt"
}
```

Deux nouvelles clefs apparaissent : `TASK-COUNT` et `SERVICE-SCHEME`.

`TASK-COUNT`	Le nombre maximum de tâches. Ce nombre doit être strictement positif.
`SERVICE-SCHEME`	Le schéma utilisé pour réaliser les appels système. Actuellement, seul `"svc"` est implémenté.

Travail à faire (3/3)

Vous pouvez maintenant compiler et lancer l'exécution. Le code de l'unique tâche est défini dans **user-tasks.cpp** :

```
static void task1 (USER_MODE) {
  while (1) {
    digitalWrite (L4_LED, !digitalRead (P4_PUSH_BUTTON)) ;
    if (gDisplayTime <= millis ()) {
      const uint32_t s = systick () ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ s) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ millis ()) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned64 (MODE_ micros (MODE)) ;
      gDisplayTime += 1000 ;
    }
  }
}
```

Noter que la tâche comprend la construction `while (1) { ··· }` qui est une boucle infinie.

2 — Présentation du code

Le *reset handler*

Le code exécuté au démarrage est la fonction **reset.handler** contenue dans le fichier **reset-handler-xtr.s** :

```
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
@-------------------------------- Init
  svc   #0
@-------------------------------- This is the background task: turn off activity led
@  Activity led is connected to PORTC:5 (#13)
background.task: @ Only use R0, R1, R2, R3 and R12. Other registers are not preserved
  ldr   r0, =0x400FF088  @ Address of GPIOC_PCOR control register
  movs  r1, # (1 << 5)   @ Port D13 is PORTC:5
  str   r1, [r0]         @ Turn off
  b     background.task
```

Comme pour l'étape précédente, la seconde phase d'initialisation est réalisée via `svc #0`. Par contre, les appels aux fonctions **setup** et **loop** ont disparu, et sont remplacés par la tâche de fond de l'exécutif, qui boucle indéfiniment sur l'extinction de la led Teensy. Le paramètre `BACKGROUND.STACK.SIZE` est fixé à 32, c'est le nombre d'octets sauvés lors d'une interruption.

Le *svc handler* (1/2)

Le *svc handler* est maintenant automatiquement engendré et apparaît au début du fichier **zSOURCES/interrupt-handler.s** (fonction **interrupt.SVC**). Son algorithme est plus complexe car il inclut le *dispatching* des services système et le changement de contexte des tâches. Il est définitif, et sera inchangé dans les étapes ultérieures.

```
interrupt.SVC:
@----------- Save preserved registers
  push  {r4, lr}
@----------- R4 <- thread SP
  mrs   r4, psp
@----------- Restore R0, R1, R2 and R3 from saved stack
  ldmia r4!, {r0, r1, r2, r3}  @ R4 incremented by 16
@----------- R4 <- Address of SVC instruction
  ldr r4, [r4, #8] @ 8 : 2 stacked registers before saved PC
@----------- R12 <- bits 0-7 of SVC instruction
  ldrb  r12, [r4, #-2]    @ R12 is service call index
@----------- R4 <- address of dispatcher table
  ldr   r4, =svc.dispatcher.table
@----------- R12 <- address of routine to call
  ldr   r12, [r4, r12, lsl #2]  @ R12 = R4 + (R12 << 2)
@----------- R4 <- calling task context
  ldr   r4, =var.running.task.control.block.ptr
  ldr   r4, [r4]
@----------- Call service routine
  blx   r12      @ R4:calling task context address
```

```
handle.context.switch:
@----------- Select task to run
  bl    kernel.select.task.to.run
@------- R0 <- calling task context, R1 <- new task context
  ldr   r1, =var.running.task.control.block.ptr
  mov   r0, r4
  ldr   r1, [r1]
@----------- Restore preserved registers
  pop   {r4, lr}
@----------- Running task did change ?
  cmp  r0, r1 @ R0:calling task context, R1:new task context
  bne   running.state.did.change
  bx    lr  @ No change
@----------- Save context of preempted task
running.state.did.change:
  mrs   r12, psp
  cbz   r0, save.background.task.context
@--- Save registers r4 to r11, PSP (stored in R12), LR
  stmia r0, {r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}
  b     perform.restore.context
save.background.task.context:
  ldr   r2, =var.background.task.context
  str   r12, [r2]
@----------- Restore context of activated task
perform.restore.context:
  cbz   r1, restore.background.task.context
  ldmia r1, {r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}
  msr   psp, r12
  bx    lr
@----------- Restore background task context
restore.background.task.context:
  ldr   r2, =var.background.task.context
  ldr   r2, [r2]
  msr   psp, r2
  bx    lr
```

Le *svc handler* (2/2)

Le *svc handler* exécute principalement trois opérations :

- exécution d'un service ;
- appel de la fonction **kernel.select.task.to.run** ;
- changement de contexte si besoin est.

On décrit ici comment le service est exécuté : l'argument de l'instruction `svc` est utilisé comme indice du tableau commençant à l'adresse **svc.dispatcher.table**. Dans cette étape, ce tableau est le suivant (fichier **zSOURCES/interrupt-handler.s**) :

```
svc.dispatcher.table:
  .word start.phase2 @ 0
```

Ainsi, l'instruction `svc #0` du reset handler provoque l'exécution de la fonction **start.phase2**. On verra dans les étapes suivantes comment seront ajoutés d'autres services.

Le fichier `user-tasks.cpp` (1/3)

Le fichier **user-tasks.cpp** contient la tâche qui est exécutée. Dans les étapes suivantes, on écrira dans ce fichier le code des tâches.

On décrit ci-dessous et dans les pages suivantes les différents parties de ce fichier.

```
static uint64_t gStack1 [64] ;
```

Ceci déclare la pile de la tâche. `uint64_t` étant un entier sur 8 octets, la taille de la pile est de 64 * 8 octets. Utiliser le type `uint64_t` garantit que le tableau soit aligné sur une frontière de 8 octets, ce qui est recommandé pour un Cortex-M4. Quand plusieurs tâches sont déclarées, chacune doit avoir sa propre pile.

Le fichier `user-tasks.cpp` (2/3)

```
static uint32_t gDisplayTime = 0 ;

static void task1 (USER_MODE) {
  while (1) {
   if (gDisplayTime <= millis ()) {
      const uint32_t s = systick () ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ s) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ millis ()) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned64 (MODE_ micros (MODE)) ;
      gDisplayTime += 1000 ;
    }
  }
}
```

Voici la fonction qui contient le code de la tâche. Noter qu'elle ne se termine jamais : dans cette étape, la terminaison d'une tâche n'est pas implémentée, c'est-à-dire que l'exécution plante.

Le fichier `user-tasks.cpp` (3/3)

```
static void initTasks (INIT_MODE) {
  kernel_createTask (MODE_ gStack1, sizeof (gStack1), task1) ;
}

MACRO_INIT_ROUTINE (initTasks) ;
```

Dans ce cours, la création d'une tâche ne peut être effectuée qu'en mode `INIT`. La fonction **kernel_createTask** a trois arguments :

- l'adresse de la pile, **gStack1** ;
- la taille (en octets) de la pile, c'est-à-dire **sizeof (gStack1)** ;
- le code exécuté, désigné par la fonction **task1**.

Pour l'exécutif du cours, une tâche créée est aussitôt rendue prête.

Remarquez qu'il n'y a pas d'argument fixant la priorité : celle-ci est implicite, la première tâche déclarée est la plus prioritaire, les tâches sont déclarées par ordre de priorité décroissante.

3 — L’exécutif en détail

La variable `var.running.task.control.block.ptr`

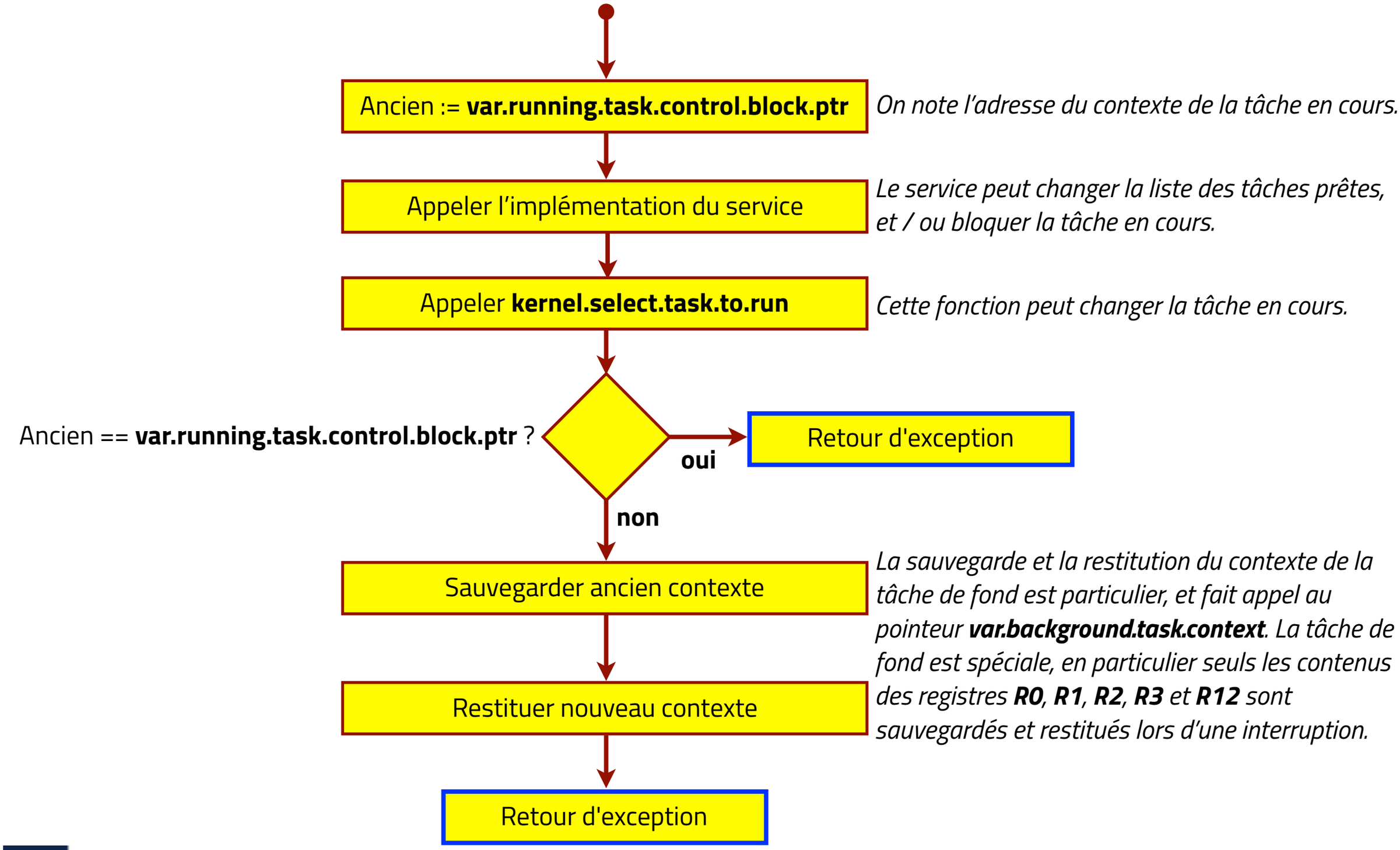
Cette variable est partagée entre l'assembleur (*svc handler*) et le code C++.

C'est un pointeur dont le rôle est de désigner le descripteur de la tâche en cours, plus précisément le champ qui contient la sauvegarde des registres qui ne sont pas sauvegardés dans la pile lors de l'interruption.

Si cette variable est nulle, c'est la tâche de fond qui est en cours.

Elle est la zone **bss**, c'est-à-dire qu'elle est initialisée à zéro quand cette zone est mise à zéro : ainsi, initialement, l'exécutif considère que c'est la tâche de fond qui est en cours quand le service d'initialisation est appelé par `svc #0`.

Organigramme du *svc handler*



On note l'adresse du contexte de la tâche en cours.

Le service peut changer la liste des tâches prêtes, et / ou bloquer la tâche en cours.

Cette fonction peut changer la tâche en cours.

*La sauvegarde et la restitution du contexte de la tâche de fond est particulier, et fait appel au pointeur **var.background.task.context**. La tâche de fond est spéciale, en particulier seuls les contenus des registres **R0**, **R1**, **R2**, **R3** et **R12** sont sauvegardés et restitués lors d'une interruption.*

Descripteur de tâche

Voici le descripteur d'une tâche (fichier **xtr.cpp**). Le champ **mTaskContext** contient le contexte d'une tâche qui n'est pas encours (voir page suivante). Le champ **mTaskIndex** est l'indice de la tâche (0 pour la plus prioritaire). Les champs placés en commentaire ne sont pas utiles pour ce premier exécutif, ils seront décommentés au cours des étapes suivantes.

```
typedef struct TaskControlBlock {
//--- Context buffer
  TaskContext mTaskContext ; // SHOULD BE THE FIRST FIELD
//--- This field is used for deadline (not used in this step)
//  uint32_t mDeadline ;
//--- Guards (not used in this step)
//   GuardDescriptor mGuardDescriptor ;
//   GuardState mGuardState ;
//--- Task index
  uint8_t mTaskIndex ;
//--- User result (not used in this step)
//   bool mUserResult ;
//---
} TaskControlBlock ;
```

Une remarque particulière sur le champ **mTaskContext**. Quand une tâche est en cours, la variable **var.running.task.control.block.ptr** pointe sur le champ **mTaskContext** de son descripteur de tâche. Or, partager entre assembleur et C l'adresse d'un champ peut être fragile, surtout si on reconstruit l'offset du champ en assembleur. Ici, **mTaskContext** est toujours le premier champ, aussi son adresse est l'adresse du descripteur de tâche.

Tableau des descripteurs de tâches

Le tableau des descripteurs de tâches est un simple tableau C (fichier **xtr.cpp**).

```
static TaskControlBlock gTaskDescriptorArray [TASK_COUNT] ;
```

L'initialisation du tableau des descripteurs des tâches est automatique (variable dans la zone **bss**) : tous les champs sont initialisés à des valeurs correspondant à des zéros binaires.

Deux fonctions sont disponibles, l'une pour accéder au descripteur d'une tâche à partir de son indice, et l'autre pour obtenir l'indice d'une tâche à partir de son descripteur :

```
TaskControlBlock * descriptorPointerForTaskIndex (const uint8_t inTaskIndex) {
  XTR_ASSERT (inTaskIndex < TASK_COUNT, inTaskIndex) ;
  return & gTaskDescriptorArray [inTaskIndex] ;
}

uint8_t indexForDescriptorTask (const TaskControlBlock * inTaskPtr) { // should be not nullptr
  XTR_ASSERT_NON_NULL_POINTER (inTaskPtr) ;
  return inTaskPtr->mTaskIndex ;
}
```

Contexte d'une tâche

Le ***contexte d'une tâche*** est l'ensemble des informations relatives à un avancement particulier de l'exécution d'un programme, c'est-à-dire pour un Cortex-M4 la valeur de tous les registres internes : **R0** à **R12**, **R13** (SP), **R14** (LR), **R15** (PC), **PSR** (*Program Status Register*).

Quand une tâche ***passe*** dans l'état ***TASK_RUNNING***, le contenu du champ contexte de son descripteur est copié dans les registres du processeur (« *restitution du contexte* »).

Quand une tâche ***est*** dans l'état ***TASK_RUNNING***, le contenu du champ contexte de son descripteur n'est pas significatif, car le contexte est contenu dans les registres du processeur.

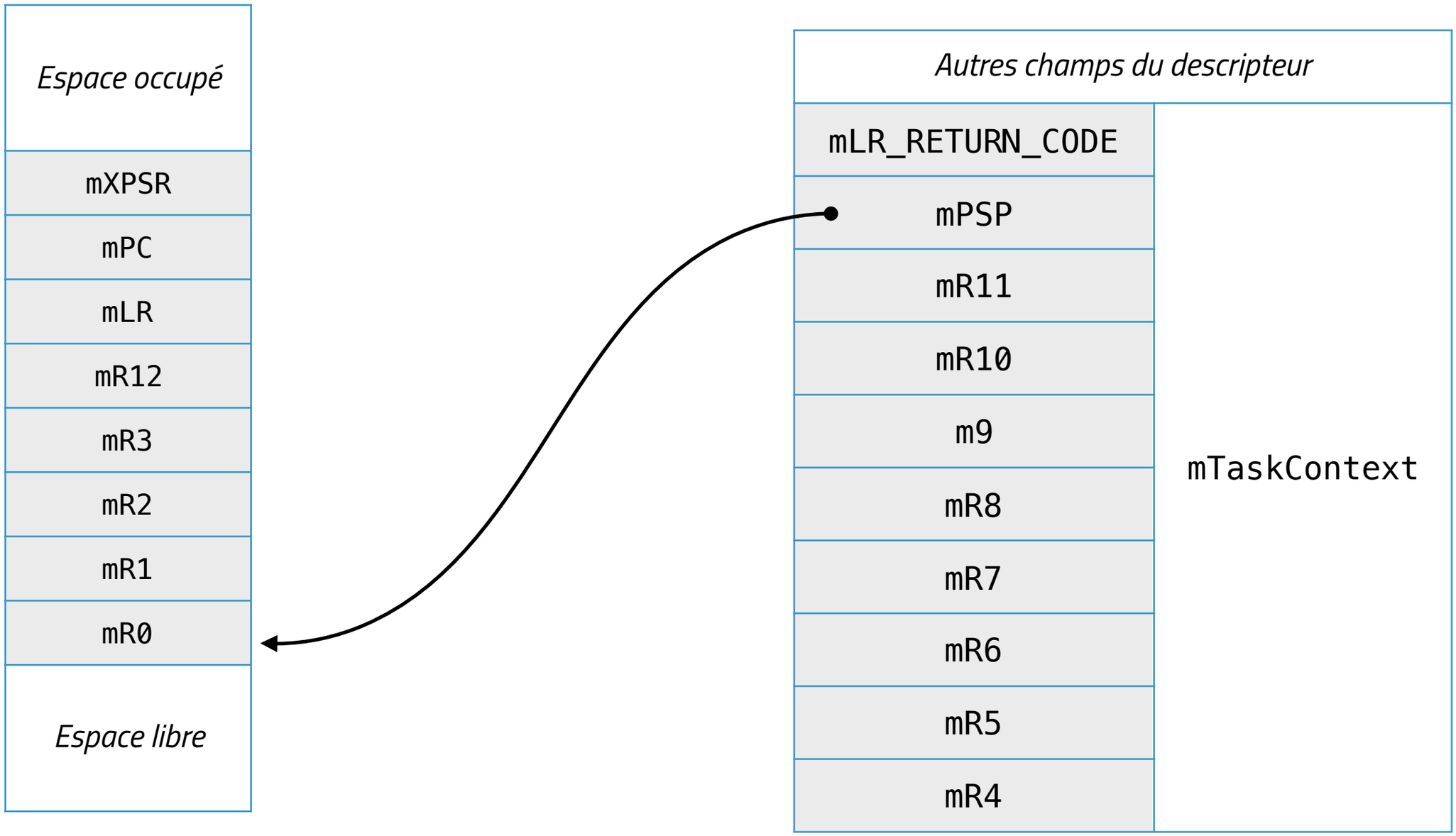
Quand une tâche ***quitte*** l'état ***TASK_RUNNING***, le contenu des registres du processeur est copié dans le champ contexte de son descripteur (« *sauvegarde du contexte* »).

```
typedef struct {
  uint32_t mR0 ;
  uint32_t mR1 ;
  uint32_t mR2 ;
  uint32_t mR3 ;
  uint32_t mR12 ;
  uint32_t mLR ;
  uint32_t mPC ;
  uint32_t mXPSR ;
} ExceptionFrame_without_floatingPointStorage ;
```

```
typedef struct {
  uint32_t mR4 ;
  uint32_t mR5 ;
  uint32_t mR6 ;
  uint32_t mR7 ;
  uint32_t mR8 ;
  uint32_t mR9 ;
  uint32_t mR10 ;
  uint32_t mR11 ;
  ExceptionFrame_without_floatingPointStorage * mPSP ;//R13
  uint32_t mLR_RETURN_CODE ;
} TaskContext ;
```

Contexte sauvegardé

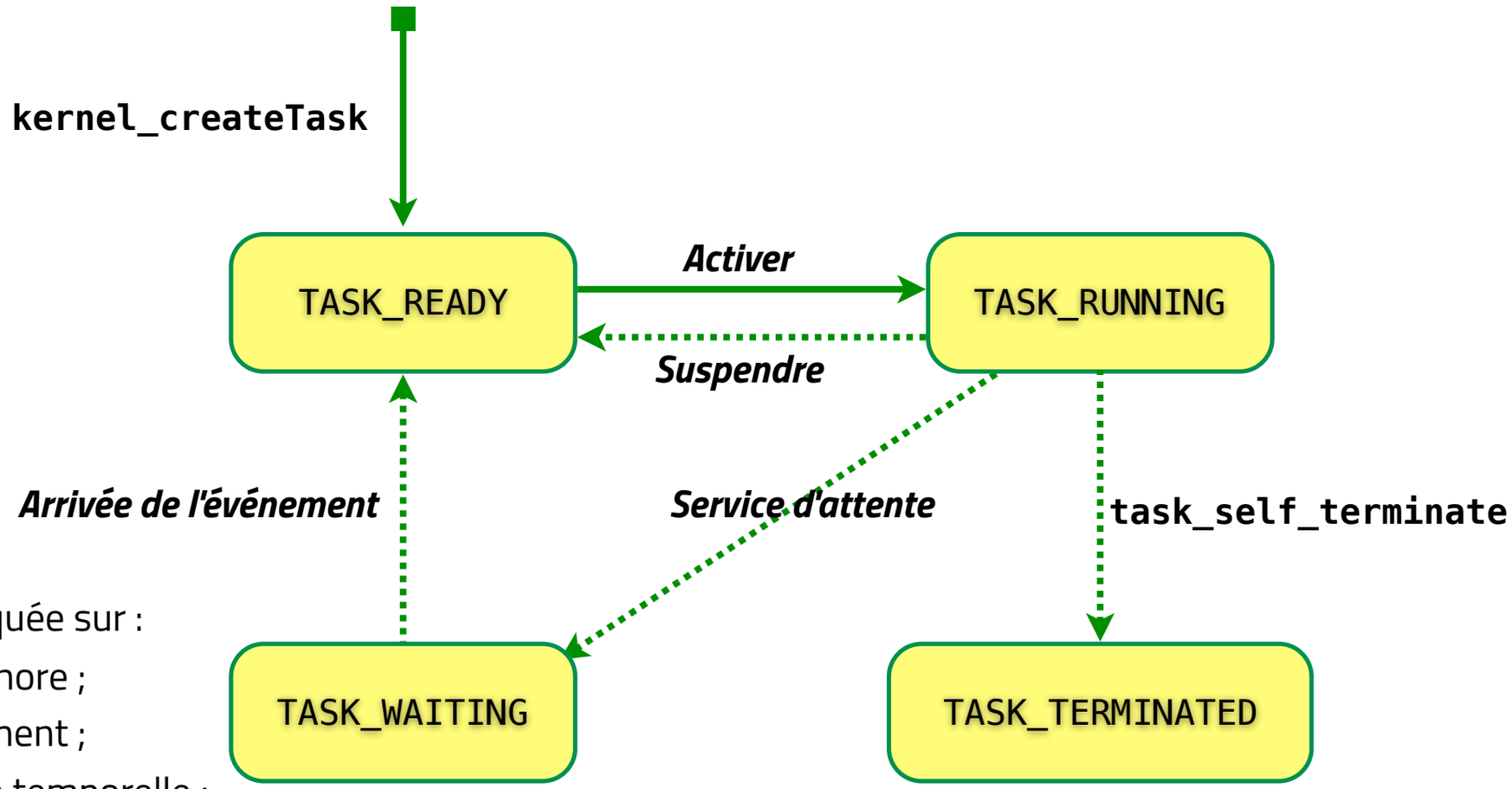
Quand une tâche n'est en cours, son contexte est sauvegardé dans le champ **mTaskContext** de son descripteur, et dans sa pile.



Pile de la tâche **Descripteur de la tâche**

États d'une tâche

Voici le graphe des états d'une tâche, pour l'exécutif présenté dans ce cours. L'état n'est pas mémorisé par un champ du descripteur, ce n'est pas nécessaire pour cet exécutif.



Tâche bloquée sur :

- sémaphore ;
- événement ;
- attente temporelle ;
- …

Les flèches en pointillés correspondent à des opérations qui sont pas encore implémentées.

Création d'une tâche : `kernel_createTask`

La fonction **kernel_createTask** est implémentée dans le fichier **xtr.cpp**.

```
static uint8_t gTaskIndex ;   Initialisée implicitement à zéro.

void kernel_createTask (INIT_MODE_
                        uint64_t * inStackBufferAddress,
                        uint32_t inStackBufferSize,
                        RoutineTaskType inTaskRoutine) {
  XTR_ASSERT (gTaskIndex < TASK_COUNT, gTaskIndex) ;   Cette assertion détecte si trop de tâches sont déclarées.
  TaskControlBlock * taskControlBlockPtr = & gTaskDescriptorArray [gTaskIndex] ;
  taskControlBlockPtr->mTaskIndex = gTaskIndex ;
//--- Initialize properties   Inutile d'initialiser explicitement les propriétés, la zone bss est mise à zéro.
//    As gTaskDescriptorArray is in bss, all properties are by default initialized to binary 0
//   taskControlBlockPtr->mDeadline = 0 ; // statically initialized to 0
//   taskControlBlockPtr->mUserResult = false ; // statically initialized to false
//   taskControlBlockPtr->mGuardState = GUARD_EVALUATING_OR_OUTSIDE ; // statically initialized
//   taskControlBlockPtr->mGuardDescriptor.mCount = 0 ; // statically initialized to 0
//--- Initialize Context
  kernel_set_task_context (MODE_
                           taskControlBlockPtr->mTaskContext,
                           (uint32_t) inStackBufferAddress,
                           inStackBufferSize,
                           inTaskRoutine) ;
//--- Make task ready
  kernel_makeTaskReady (MODE_ taskControlBlockPtr) ;   La tâche est rendue prête.
//---
  gTaskIndex += 1 ;
}
```

Établissement du contexte initial de la tâche.

Création du contexte initial d'une tâche

La fonction **kernel_set_task_context** est interne à l'exécutif (fichier **xtr.cpp**).

```
static void kernel_set_task_context (INIT_MODE_
                                     TaskContext & ioTaskContext,
                                     const uint32_t inStackBufferAddress,
                                     const uint32_t inStackBufferSize,
                                     RoutineTaskType inTaskRoutine) {
//--- Initialize LR
  ioTaskContext.mLR_RETURN_CODE = 0xFFFFFFFD ;
//--- Stack Pointer initial value
  uint32_t initialTopOfStack = inStackBufferAddress + inStackBufferSize ;
  initialTopOfStack -= sizeof (ExceptionFrame_without_floatingPointStorage) ;
//--- Initialize SP
  auto ptr = (ExceptionFrame_without_floatingPointStorage *) initialTopOfStack ;
  ioTaskContext.mPSP = ptr ;
//--- Initialize PC
  ptr->mPC = (uint32_t) inTaskRoutine ;
//--- Initialize CPSR
  ptr->mXPSR = 1 << 24 ; // Thumb bit
}
```

Rendre une tâche prête : `kernel_makeTaskReady`

La fonction **kernel_makeTaskReady** est interne à l'exécutif (fichier **xtr.cpp**) : elle insère le descripteur de la tâche passé en argument dans la liste des tâches prêtes **gReadyTaskList**.

```
static void kernel_makeTaskReady (IRQ_MODE_ TaskControlBlock * inTaskPtr) {
  XTR_ASSERT_NON_NULL_POINTER (inTaskPtr) ;
  gReadyTaskList.enterTask (MODE_ inTaskPtr) ;
//   inTaskPtr->mUserResult = 1 ;
}
```

La ligne en commentaire n'est utile dans cette étape (le champ **mUserResult** est en commentaire dans le descripteur de tâche).

La fonction `kernel.select.task.to.run`

La fonction **kernel.select.task.to.run** est interne à l'exécutif (fichier **xtr.cpp**), et uniquement appelée à partir du *svc handler* (et dans les étapes suivantes, dans les routines d'interruption), qui se trouvent dans le fichier **zSOURCES/interrupt-handlers.s**.

```
TaskControlBlock * gRunningTaskControlBlockPtr asm ("var.running.task.control.block.ptr") ;

void kernelSelectTaskToRun (IRQ_MODE) asm ("kernel.select.task.to.run") ;

void kernelSelectTaskToRun (IRQ_MODE) {
  if (gRunningTaskControlBlockPtr != nullptr) {
    gReadyTaskList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;
  }
  gRunningTaskControlBlockPtr = gReadyTaskList.removeFirstTask (MODE) ;
}
```

La variable assembleur **var.running.task.control.block.ptr** est connue dans le code C++ sous le nom **gRunningTaskControlBlockPtr**. La fonction **kernel.select.task.to.run** (**kernelSelectTaskToRun** en C++) est la seule à écrire cette variable, et réalise les opérations suivantes :

- si il y a une tâche en cours, elle est insérée dans la liste des tâches prêtes ;
- ensuite, la première tâche de cette liste est retirée, et devient la tâche en cours.

Remarquer que si la liste des tâches prêtes est vide, **removeFirstTask** renvoie simplement **nullptr** : c'est la tâche de fond qui devient la tâche en cours.

La liste des tâches prêtes (1/4)

Son type (ou plutôt sa classe) est **TaskList**, déclaré dans t**ask-list--32-tasks.h** et ses méthodes sont implémentées dans **task-list--32-tasks.cpp**.

C'est une structure de données qui est limitée à 32 tâches (sans compter la tâche de fond).

```
class TaskList {
//--- Default constructor
  public: inline TaskList (void) : mList (0) {}

//--- Block a task in list
  public: void enterTask (SECTION_MODE_ TaskControlBlock * inTaskPtr) ;

//--- Remove first task (returns nullptr if list is empty)
  public: TaskControlBlock * removeFirstTask (IRQ_MODE) ;

//--- Private property
  private: uint32_t mList ;

//--- No copy
  private: TaskList (const TaskList &) ;
  private: TaskList & operator = (const TaskList &) ;
} ;
```

La liste des tâches prêtes (2/4)

La propriété **mList** est un entier de 32 bits. Si le bit d'indice *n* est à 1, la tâche d'indice *n* est dans la liste. Si il est à 0, elle n'est pas dans la liste.

Aussi :

- si **mList** est à 0, la liste est vide (c'est sa valeur initiale) ;
- ajouter une tâche revient à faire un **ou bit-à-bit** entre deux entiers 32 bits ;
- retirer une tâche revient à faire un **et bit-à-bit** entre deux entiers 32 bits ;
- obtenir l'indice de la tâche la plus prioritaire consiste à obtenir l'indice du premier bit non nul, en partant du bit le mois significatif.

La liste des tâches prêtes (3/4)

Toutes ces opérations se font en temps constant, sauf *a priori* la dernière. En effet, si la liste n'est pas vide, il faut itérer afin de trouver l'indice du premier bit non nul :

```
uint32_t indice = 0 ;
uint32_t v = mList ;
while ((v & 1) == 0) {
  indice += 1 ;
  v >>= 1 ;
} ;
```

À titre d'information, ce n'est pas ce qui est implémenté.
*Attention, cet algorithme boucle indéfiniment si **mList** est nul.*

Mais le processeur Cortex-M4 possède deux instructions qui permettent de faire cette recherche en temps constant :

- l'instruction **CLZ** (*Count Leading Zeros*) retourne le nombre de bits significatifs à zéro ;
- l'instruction **RBITS** (*Reverse Bits*) qui renverse l'ordre des bits.

Ainsi, on obtient l'indice du premier à zéro par une séquence **RBITS** suivi de **CLZ**. Pour éviter d'écrire directement de l'assembleur dans du code C, ou d'appeler une fonction assembleur, on utilise la fonction intrinsèque de GCC **__builtin_ctz**, qui retourne directement l'indice du premier bit non nul.

Liens :

- **CLZ** : http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BABBBJGA.html
- **RBITS** : http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BABBBJGA.html
- **__builtin_ctz** : https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

La liste des tâches prêtes (4/4)

On peut donc maintenant montrer l'implémentation des deux méthodes de la classe **TaskList** (fichier **task-list--32-tasks.cpp**). D'abord, la méthode d'insertion :

```
void TaskList::enterTask (SECTION_MODE_ TaskControlBlock * inTaskPtr) {
  TASK_LIST_ASSERT_NON_NULL_POINTER (inTaskPtr) ;
  const uint32_t taskIndex = indexForDescriptorTask (inTaskPtr) ;
  TASK_LIST_ASSERT (taskIndex < TASK_COUNT, taskIndex) ;
  const uint32_t mask = 1U << taskIndex ;
  mList |= mask ;
}
```

Et la méthode **removeFirstTask** qui retire la tâche la plus prioritaire :

```
TaskControlBlock * TaskList::removeFirstTask (IRQ_MODE) {
  TaskControlBlock * taskPtr = nullptr ;
  if (mList != 0) {
    const uint32_t taskIndex = (uint32_t) __builtin_ctz (mList) ;
    TASK_LIST_ASSERT (taskIndex < TASK_COUNT, taskIndex) ;
    const uint32_t mask = 1U << taskIndex ;
    mList &= ~ mask ;
    taskPtr = descriptorPointerForTaskIndex (taskIndex) ;
  }
  return taskPtr ;
}
```