

Construire un exécutif Temps-Réel dual-core pas-à-pas sur Raspberry Pi Pico



Pierre Molinaro
État d'avancement (étape 13) au 30 avril 2021

Présentation

Objectifs

Objectifs :

- *comprendre le fonctionnement d'un exécutif Temps Réel dual-core, en réalisant son écriture pas à pas ;*
- *comprendre le développement sur un micro-contrôleur.*

Étapes de réalisation

01	Présentation / introduction : programme « blink led »	
02	SysTick	
03	Modes logiciels	
04	Boot et init routines	
05	Leds / boutons poussoir	1 ^{re} partie : exécution séquentielle sous interruption
06	Afficheur LCD	
07	Interruption Systick	
08	Qualificatif volatile	
09	Section critique	
10	Assertions et <i>Fault Handler</i>	
11	Appel SVC	
12	Premier exécutif : une seule tâche	
13	Terminaison des tâches	
14	Attente passive	
15	Outils de synchronisation (sémaphore de Dijkstra, ...)	
16	Attentes temporisées	
17	Allocation dynamique	2 ^e partie : exécutif <i>single-core</i>
18	Commandes de synchronisation gardées (à la CSP)	
19	Réseau CAN (attente actives)	
20	Réseau CAN (émission bloquante)	
21	Réseau CAN (réception bloquante)	
22	Début prise en charge du second processeur...	3 ^e partie : exécutif <i>dual-core</i>

Documentation

[datasheet] (tout sur le micro-contrôleur RP2040)

RP2040 Datasheet, A microcontroller by Raspberry Pi, version 1.4.1, 12 April 2021,
<https://datasheets.raspberrypi.org/rp2040/rp2040-datasheet.pdf>

[sdk] (le SDK officiel)

Raspberry Pi Pico C/C++ SDK, Libraries and tools for C/C++ development on RP2040 microcontrollers,
version 1.4.1, 13 April 2021,
<https://datasheets.raspberrypi.org/pico/raspberry-pi-pico-c-sdk.pdf>

[picoboard] (tout sur la carte Pi Pico)

Raspberry Pi Pico Datasheet, an RP2040-based microcontroller board, version 1.4.1, 13 April 2021,
<https://datasheets.raspberrypi.org/pico/pico-datasheet.pdf>

[gettingstarted] (guide de démarrage)

Getting started with Raspberry Pi Pico, C/C++ development with Raspberry Pi Pico and other RP2040-based microcontroller boards,
<https://datasheets.raspberrypi.org/pico/getting-started-with-pico.pdf>

Liens

[git-bootrom] (la bootrom est exécutée au démarrage du micro-contrôleur)

<https://github.com/raspberrypi/pico-bootrom>

[git-sdk] (le SDK sera chargé dans votre répertoire TREEL/archives, voir plus loin)

<https://github.com/raspberrypi/pico-sdk>

[git-arduino] (adaptation du SDK pour utilisation sous Arduino)

<https://github.com/earlephilhower/arduino-pico>

[arm-cortex-m0+] (Cortex-M0+ Technical Reference Manual)

<https://developer.arm.com/documentation/ddi0484/latest>

Installation de la chaîne de développement

Installation

Plateforme de développement :

- Linux 32 bits ;
- Linux 64 bits ;
- Mac OS X, processeur Intel ;
- Windows : installer une machine virtuelle Linux.

La procédure d'installation propre à chaque plate-forme est décrite dans les pages suivantes.

La chaîne de développement est GCC pour ARM, maintenue par ARM :

<https://developer.arm.com/open-source/gnu-toolchain/gnu-rm>

Installation sur Mac OS

Pré-requis : installer Xcode (gratuit, via l'App Store).

Installation :

1. importer le repository <https://github.com/pierremolinaro/real-time-kernel-pi-pico> ;
2. placer le répertoire où vous voulez, du moment que son chemin ne comporte ni espace ni caractère accentué ; **dans tout le poly, ce répertoire est nommé TREEL** ;
3. lancer la compilation de l'étape n°1 : TREEL/steps/01-blink-led/1-build.py ; ce script va télécharger le système de développement propre à votre plateforme, l'installer, puis va effectuer la compilation de l'étape n°1.

Les outils de développement sont installés dans le répertoire ~ raspberry-pi-pico-tools. Pour désinstaller, il suffit de supprimer ce répertoire, et le répertoire TREEL.

Les archives sont téléchargées dans TREEL/archives ; après installation, on peut supprimer ce répertoire pour gagner de la place. Toutefois, on trouve dans le répertoire pico-sdk-master des infos complémentaires.

Installation sur Linux

Pré-requis : installer gcc, g++, make, curl.

Installation :

1. importer le repository <https://github.com/pierremolinaro/real-time-kernel-pi-pico> ;
2. placer le répertoire où vous voulez, du moment que son chemin ne comporte ni espace ni caractère accentué ; **dans tout le poly, ce répertoire est nommé TREEL** ;
3. lancer la compilation de l'étape n°1 : TREEL/steps/01-blink-led/1-build.py ; ce script va télécharger le système de développement propre à votre plateforme, l'installer, puis va effectuer la compilation de l'étape n°1.

Les outils de développement sont installés dans le répertoire ~/raspberry-pi-pico-tools. Pour désinstaller, il suffit de supprimer ce répertoire, et le répertoire TREEL.

Les archives sont téléchargées dans TREEL/archives ; après installation, on peut supprimer ce répertoire pour gagner de la place.

Exemple de log d'installation (1/2)

Les logs d'installation sur Mac et Linux sont identiques.

Install GCC tools...

```
+ rm -f /Volumes/dev-svn/real-time-kernel-pi-pico/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2.downloading
URL: https://developer.arm.com/-/media/Files/downloads.gnu-rm/7-2017q4/gcc-arm-none-eabi-7-2017-q4-major-mac.tar.bz2
Downloading...
+ curl --fail -L https://developer.arm.com/-/media/Files/downloads.gnu-rm/7-2017q4/gcc-arm-none-eabi-7-2017-q4-major-mac.tar.bz2
-o /Volumes/dev-svn/real-time-kernel-pi-pico/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2.downloading
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload  Upload   Total Spent   Left  Speed
100  242  100  242    0      0  245      0  --::-- --::-- --::--  245
100 99.7M  100 99.7M    0      0 2813k      0  0:00:36  0:00:36  --::-- 2554k
+ mv /Volumes/dev-svn/real-time-kernel-pi-pico/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2.downloading /Volumes/dev-svn/
real-time-kernel-pi-pico/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2
+ cp /Volumes/dev-svn/real-time-kernel-pi-pico/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2 /Users/pierremolinaro/
raspberry-pi-pico-tools/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2
+ cd /Users/pierremolinaro/raspberry-pi-pico-tools
+ bunzip2 gcc-arm-none-eabi-7-2017-q4-major.tar.bz2
+ tar xf gcc-arm-none-eabi-7-2017-q4-major.tar
+ rm gcc-arm-none-eabi-7-2017-q4-major.tar
```

Installing elf2uf2...

```
+ rm -f /Volumes/dev-svn/real-time-kernel-pi-pico/archives/pico-sdk-master.zip.downloading
URL: https://github.com/raspberrypi/pico-sdk/archive/master.zip
Downloading...
+ curl --fail -L https://github.com/raspberrypi/pico-sdk/archive/master.zip -o /Volumes/dev-svn/real-time-kernel-pi-
pico/archives/pico-sdk-master.zip.downloading
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload  Upload   Total Spent   Left  Speed
100  125  100  125    0      0  361      0  --::-- --::-- --::--  360
100 974k    0 974k    0      0 1064k      0  --::-- --::-- --::-- 1064k
+ mv /Volumes/dev-svn/real-time-kernel-pi-pico/archives/pico-sdk-master.zip.downloading /Volumes/dev-svn/real-time-kernel-pi-
pico/archives/pico-sdk-master.zip
+ unzip -q /Volumes/dev-svn/real-time-kernel-pi-pico/archives/pico-sdk-master.zip
....
```

Exemple de log d'installation (2/2)

```
+ g++ -O2 -std=c++11 -I /Volumes/dev-svn/real-time-kernel-pi-pico/archives/pico-sdk-master/src/common/boot_uf2/include /Volumes/dev-svn/real-time-kernel-pi-pico/archives/pico-sdk-master/tools/elf2uf2/main.cpp -o /Users/pierremolinaro/raspberry-pi-pico-tools/bin/elf2uf2
--- Making /Volumes/dev-svn/real-time-kernel-pi-pico/solutions/01-blink-led
Making "zBUILDS" directory
[ 5%] Assembling unused-interrupt.s
[ 10%] Assembling reset-handler-sequential-cpu-0.s
[ 15%] Assembling rp2040-interrupt-vectors.s
[ 21%] Assembling boot2_w25q080_2_padded_checksum.s
Making "zSOURCES" directory
[ 26%] Build base header file
[ 31%] Build all headers file
[ 36%] Build interrupt files
[ 42%] Checking setup-loop.cpp
[ 47%] Compiling setup-loop.cpp
[ 52%] Checking start-raspberry-pico.cpp
[ 57%] Compiling start-raspberry-pico.cpp
[ 63%] Checking interrupt-handler-helper.cpp
[ 68%] Compiling interrupt-handler-helper.cpp
[ 73%] Assembling interrupt-handlers.s
[ 78%] Building interrupt-handler-helper.cpp.objdump.py
[ 84%] Building setup-loop.cpp.objdump.py
Making "zPRODUCTS" directory
[ 89%] Linking zPRODUCTS/product.elf
[ 94%] Building start-raspberry-pico.cpp.objdump.py
[100%] Converting elf to UF2 zPRODUCTS/product.elf
ROM code:    2080 bytes
ROM data:     0 bytes
RAM + STACK: 1536 bytes

[Opération terminée]
```

Étape 01 . 1

Compilation et exécution

Présentation

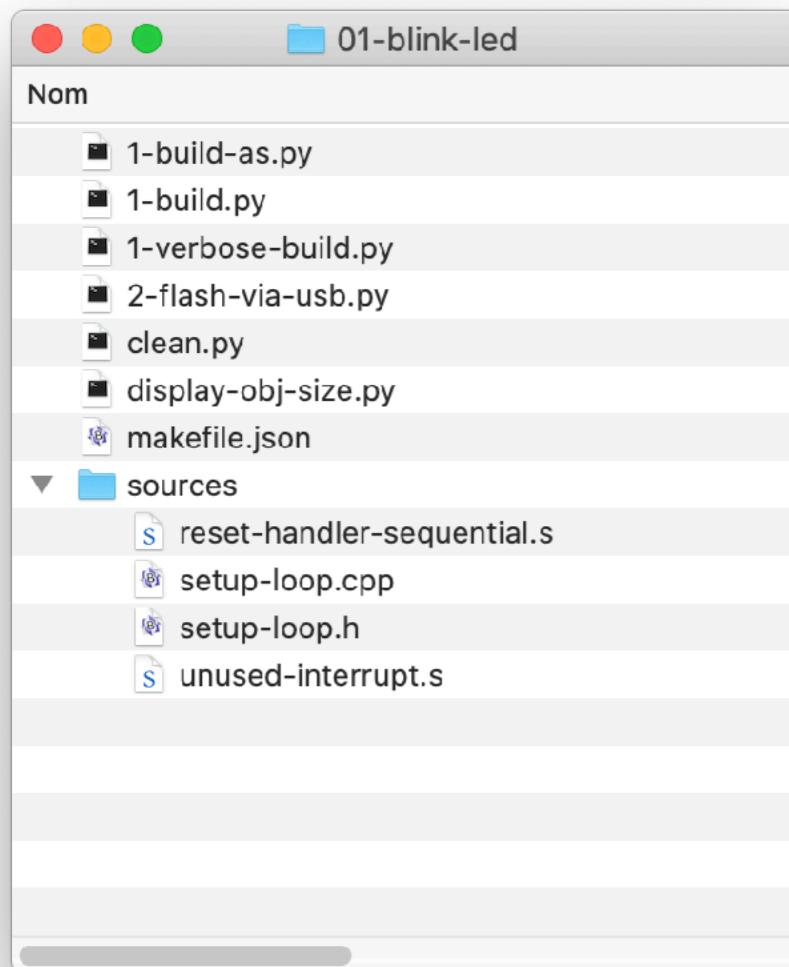
Le premier programme `01-blinkled` est un programme minimum pour débuter en faisant clignoter une led.

Il constitue la base de développement des programmes sous interruption.

Dans cette section, on présente comment compiler et exécuter le premier programme `01-blinkled` : ces explications seront valides pour tous les programmes suivants que vous écrirez.

L'explication du fonctionnement du premier programme fait l'objet des sections suivantes de ce document.

Ce que contient le répertoire d'un programme



1-build-as.py	Lance la compilation des sources en fichiers assembleur, qui sont rangés dans zASBUILDS. À n'utiliser que si on veut voir les fichiers assembleur.
1-build.py	C'est le fichier qui permet de lancer la compilation du projet (son utilisation est décrite dans les pages qui suivent).
1-verbose-build.py	Lance aussi la compilation du projet, mais à raison d'une opération à la fois (pas de parallélisme), et affiche le détail des commandes. Utile en cas d'erreur, pour localiser plus facilement l'erreur.
2-flash-via-usb.py	Enchaîne compilation et lancement de la commande de flashage de la cible.
clean.py	Efface tous les fichiers et répertoires produits par la construction.
display-obj-size.py	Affiche les tailles relatives à chaque fichier objet construits. À n'utiliser que si on est intéressé par cette information.
makefile.json	Organisation de la compilation. Voir page suivante.
sources	Répertoire qui, comme son nom l'indique contient les sources du projet.

Fichier makefile.json

La compilation d'un programme est décrite dans un fichier `makefile.json` au format JSON.

Référence :

https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

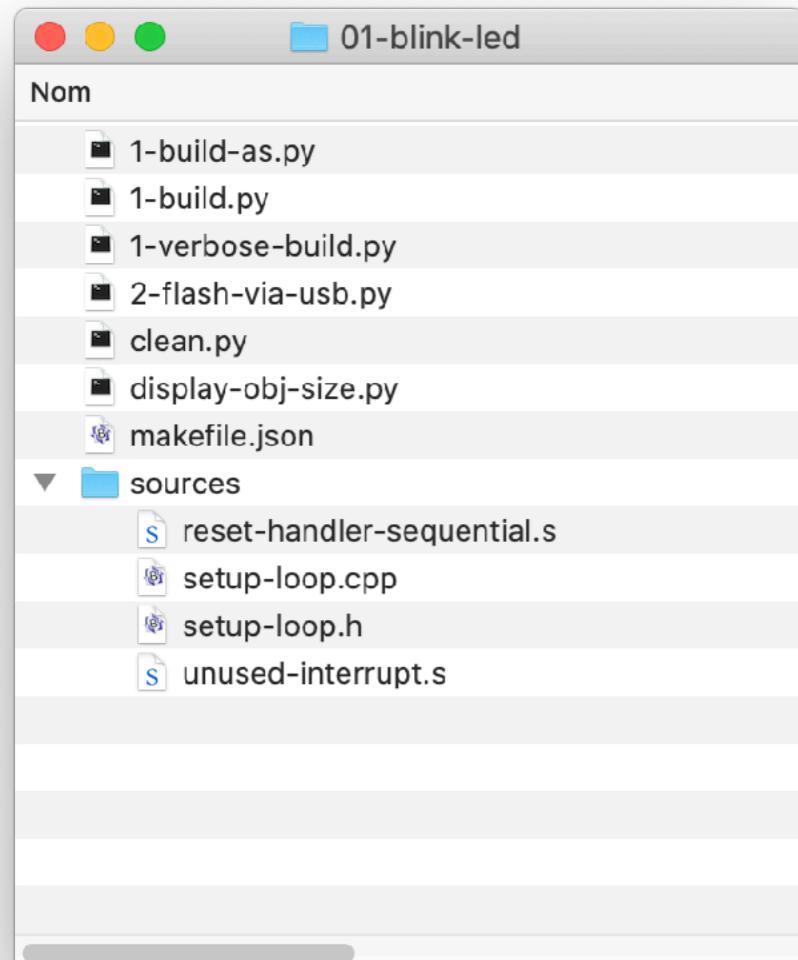
Pour le programme **01-blinkled**, le contenu de ce fichier est :

```
{ "SOURCE-DIR" : ["sources", "../..../dev-files/sources-common"],  
  "DEV-DIR" : "../..../dev-files",  
  "CPU-MHZ" : 125  
}
```

SOURCE_DIR	Liste des répertoires sources. Ne pas changer, sauf si vous répartissez vos sources dans d'autres répertoires.
DEV-DIR	Chemin vers le répertoire contenant les scripts de développement. Soit un chemin relatif à partir du répertoire courant (c'est le cas ci-dessus), soit un chemin absolu.
CPU-MHZ	Fréquence du processeur, en MHz. Vous pouvez choisir : 25, 125.

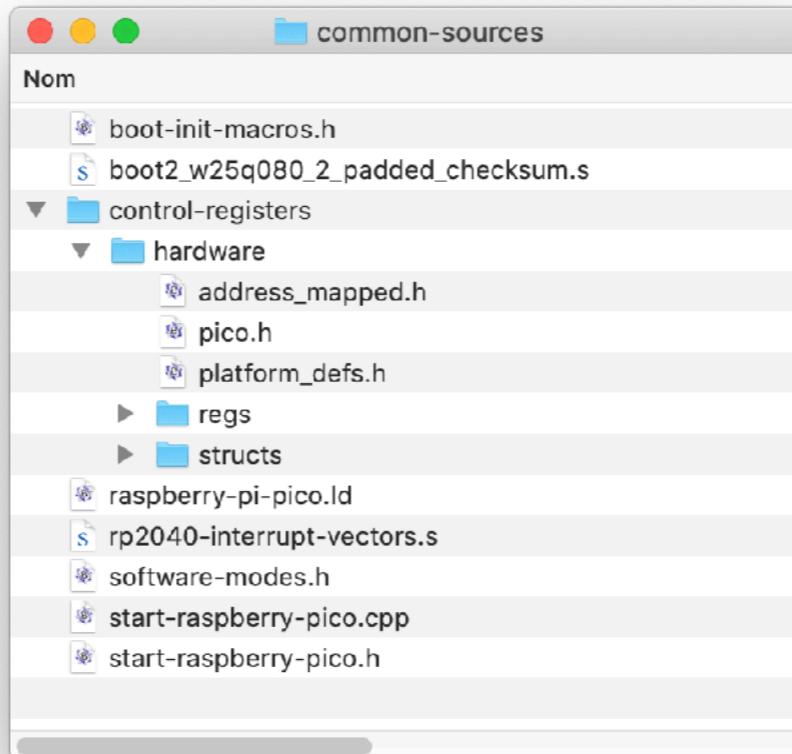
Le répertoire `sources` est celui du répertoire courant. Il est donc propre à chaque étape. Le répertoire `../..../dev-files/sources-common` désigne en fait le répertoire `TREEL/dev-files/sources-common`, il est commun à toutes les étapes. On y rassemble donc les sources communs à toutes les étapes.

Fichiers du répertoire sources



reset-handler-sequential-cpu-0.s	Routine de démarrage du micro-contrôleur (uniquement pour les programmes séquentiels, sera remplacé pour l'exécutif).
setup-loop.cpp	Contient le code utilisateur.
setup-loop.h	Déclaration des fonctions <code>setup0</code> et <code>loop0</code> .
unused-interrupt.s	Prise en charge des interruptions inutilisées (jusqu'à l'étape <i>fault-handler--assertion</i>).

Fichiers du répertoire TREEL/dev-files/sources-common



boot-init-macros.h	Fichier d'en-tête contenant la déclaration des macros d'inscription d'une fonction aux opérations de <i>boot</i> et d' <i>init</i> . Utilisé à partir de l'étape 04, ignoré auparavant.
boot2_w25q080_2_padded_checksum.s	Fichier assembleur spécial, prend en charge la configuration de l'accès en Flash.
control-registers	Ce répertoire contient la déclaration des registres de contrôle du processeur Cortex M0+ et du micro-contrôleur RP2040. Ils ont été obtenus à partir du SDK [git-sdk].
raspberry-pi-pico-flash.ld	Script de l'édition de liens.
rp2040-interrupt-vectors.s	Vecteur des interruptions du micro-micro-contrôleur RP2040.
software-modes.h	Fichier d'en-tête contenant la déclaration des <i>modes logiciels</i> . Utilisé à partir de l'étape 04, ignoré auparavant.
start-raspberry-pico.cpp	Routines de démarrage et d'initialisation.
start-raspberry-pico.h	Déclaration des prototypes des routines de démarrage et d'initialisation.

Compiler le programme d'exemple

Pour compiler le programme, rien de plus simple : il suffit de double-cliquer sur **1-build.py**.

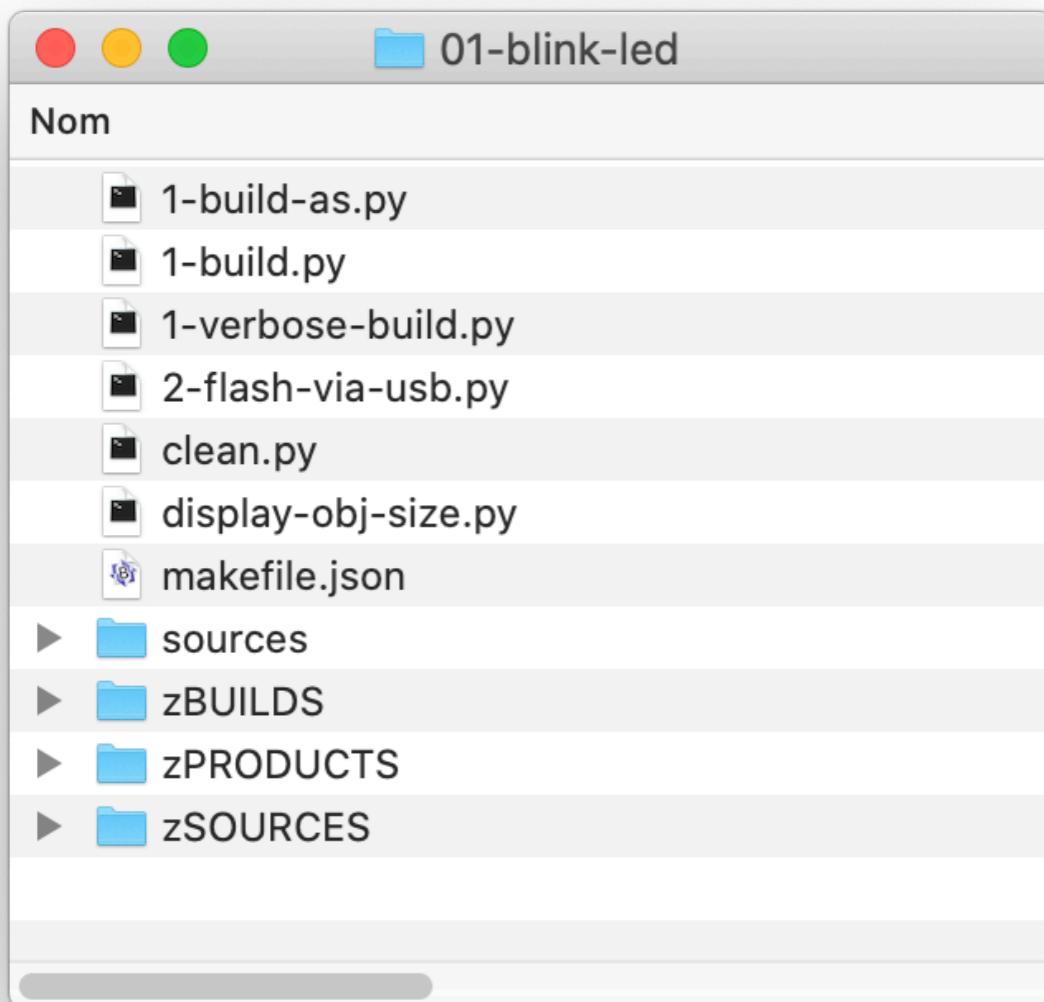
Il produit un ensemble de fichiers et de répertoires qui sont décrits dans les pages suivantes.

```
--- Making /Volumes/dev-svn/real-time-kernel-pi-pico/solutions/01-blink-led
Making "zBUILDS" directory
[ 5%] Assembling unused-interrupt.s
[ 10%] Assembling reset-handler-sequential-cpu-0.s
[ 15%] Assembling rp2040-interrupt-vectors.s
[ 21%] Assembling boot2_w25q080_2_padded_checksum.s
Making "zSOURCES" directory
[ 26%] Build base header file
[ 31%] Build all headers file
[ 36%] Build interrupt files
[ 42%] Checking setup-loop.cpp
[ 47%] Compiling setup-loop.cpp
[ 52%] Checking start-raspberry-pico.cpp
[ 57%] Compiling start-raspberry-pico.cpp
[ 63%] Checking interrupt-handler-helper.cpp
[ 68%] Compiling interrupt-handler-helper.cpp
[ 73%] Assembling interrupt-handlers.s
[ 78%] Building interrupt-handler-helper.cpp.objdump.py
[ 84%] Building setup-loop.cpp.objdump.py
Making "zPRODUCTS" directory
[ 89%] Linking zPRODUCTS/product.elf
[ 94%] Building start-raspberry-pico.cpp.objdump.py
[100%] Converting elf to UF2 zPRODUCTS/product.elf
    ROM code:    2080 bytes
    ROM data:     0 bytes
    RAM + STACK: 1536 bytes
```

Remarquez que chaque fichier C++ est d'abord vérifié, puis compilé.

La raison de la vérification sera expliquée à l'étape 03, d'ici là vous pouvez simplement ignorer ce que fait la vérification.

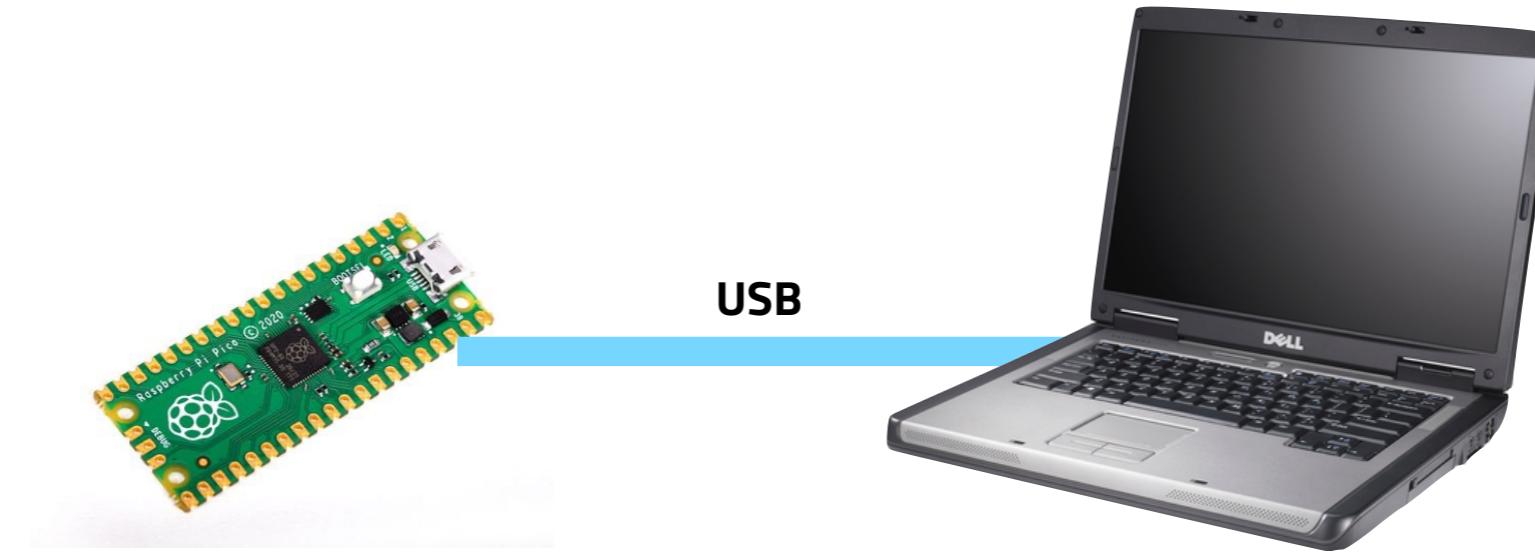
Le répertoire d'un programme après compilation



zASBUILDS	Contient les fichiers listing assembleur produits par 1-build-as.py
zBUILDS	Contient les fichiers objets et de dépendance engendrés
zPRODUCTS	Contient les fichiers exécutables.
zSOURCES	Contient les sources produits durant la compilation.

Flashage et exécution du programme

A) Connecter en maintenant BOOTSEL appuyé la carte avec le cordon USB.



B) Une fois la connexion réalisée, relâcher BOOTSEL ;

C) quand le volume RPi-RP2 apparaît, lancer le script **2-flash-via-usb.py** pour effectuer le flashage ;

```
--- Making /Volumes/dev-svn/real-time-kernel-pi-pico/solutions/01-blink-led
```

```
Nothing to make.
```

```
ROM code: 2080 bytes
```

```
ROM data: 0 bytes
```

```
RAM + STACK: 1536 bytes
```

```
Flashing Raspberry Pi Pico...
```

```
Converting to uf2, output size: 4608, start address: 0x2000
```

```
Flashing /Volumes/RPI-RP2 (RPI-RP2)
```

```
Wrote 4608 bytes to /Volumes/RPI-RP2/NEW.UF2
```

```
Success
```

D) le volume RPi-RP2 est éjecté, et le programme écrit en flash démarre.

Comment est effectuée la génération du fichier produit

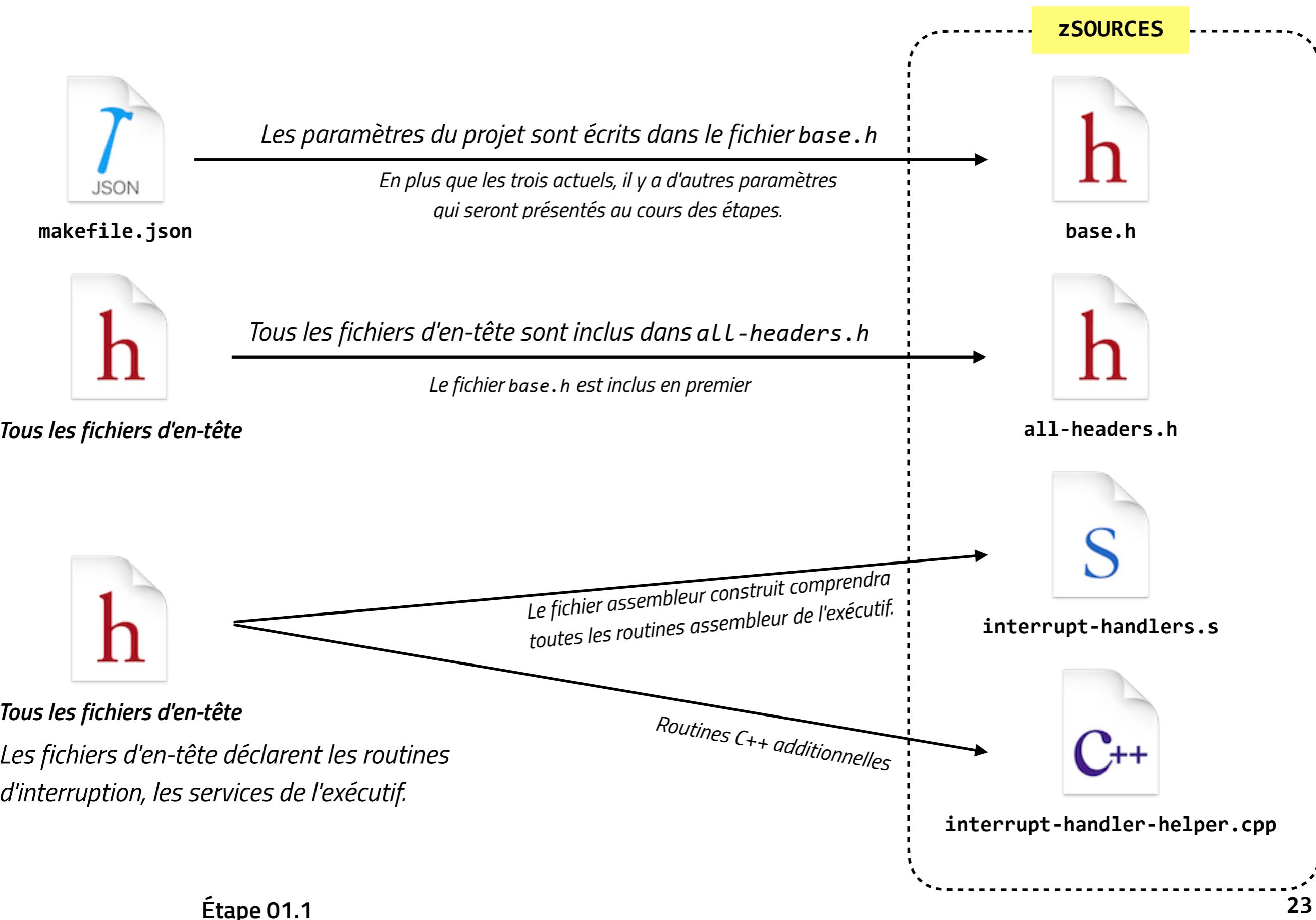
Le fichier produit qui contient le code exécutable est **zPRODUCTS/product.uf2**.

Des scripts Python organisent la génération à partir des fichiers source et du fichier **makefile.json** :

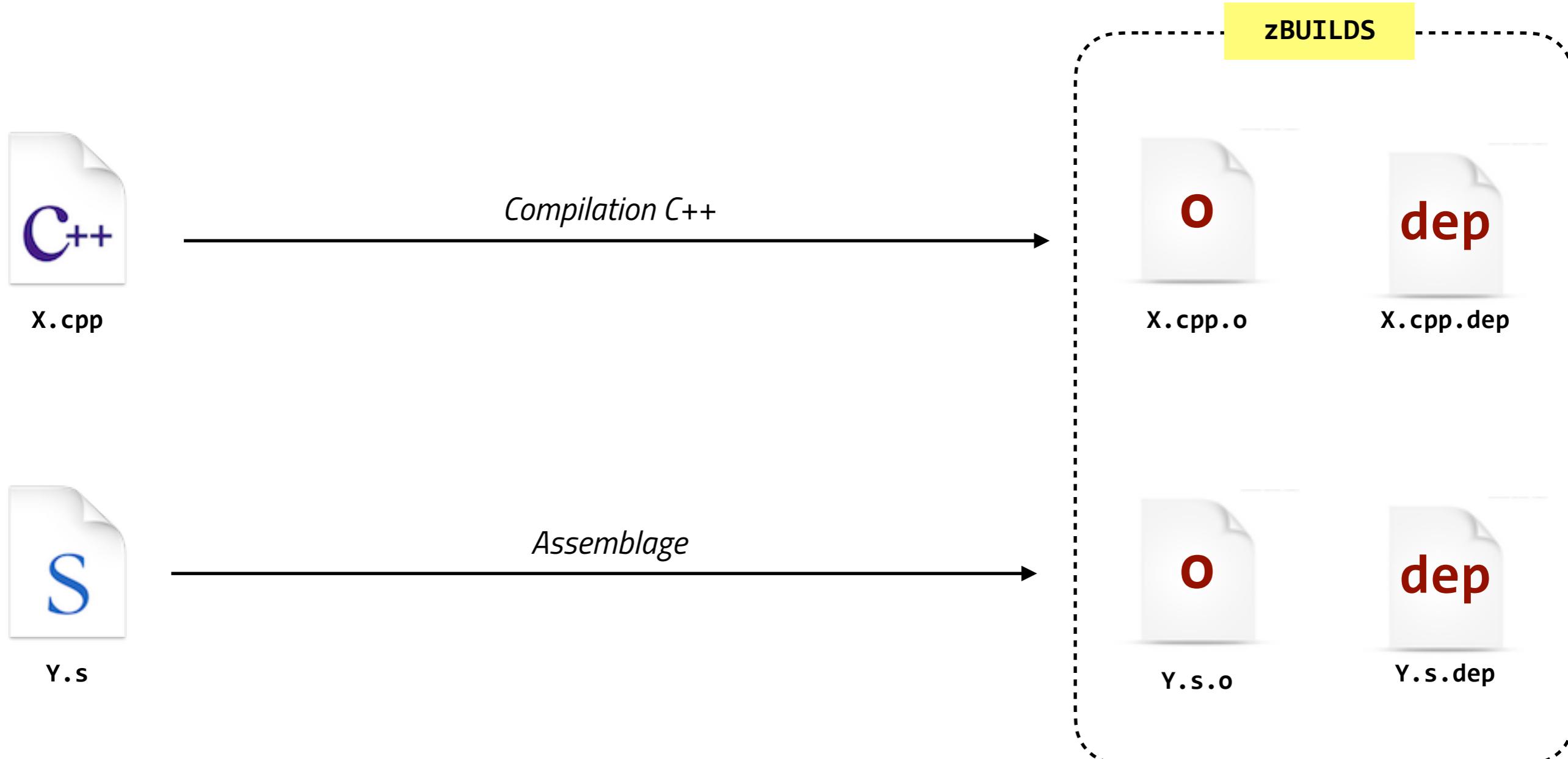
- des fichiers complémentaires C++ et assembleurs sont construits (voir page suivante) ;
- chaque source C++ est vérifié (dans les étapes 01 et 02, cette opération est sans intérêt, elle sera importante à partir de l'étape 03 qui introduit les *modes logiciels*) ;
- les fichiers source (C++ et assembleur) sont compilés ;
- puis l'édition des liens des fichiers objet est effectuée, ce qui produit **product.elf** et **product.map** ;
- enfin, le fichier **uf2** est produit.

Pour connaître le détail de chaque commande, utiliser le script **1-verbose-build.py**.

Construction des fichiers source complémentaires



Construction des fichiers objets et de dépendance



Un fichier de dépendance contient la liste des fichiers d'en-tête qui provoquent la recompilation si ils sont modifiés.

Édition des liens

o **ld**

Édition de liens

Tous les fichiers objets (.o) raspberry-pi-pico-flash.ld

L'édition des liens est paramétrée par le script **raspberry-pi-pico-flash.ld**. Elle produit deux fichiers, **product.elf** qui contient le code (*Executable and Linkable Format*), et **product.map**, un fichier texte qui contient la description du résultat de l'édition des liens.

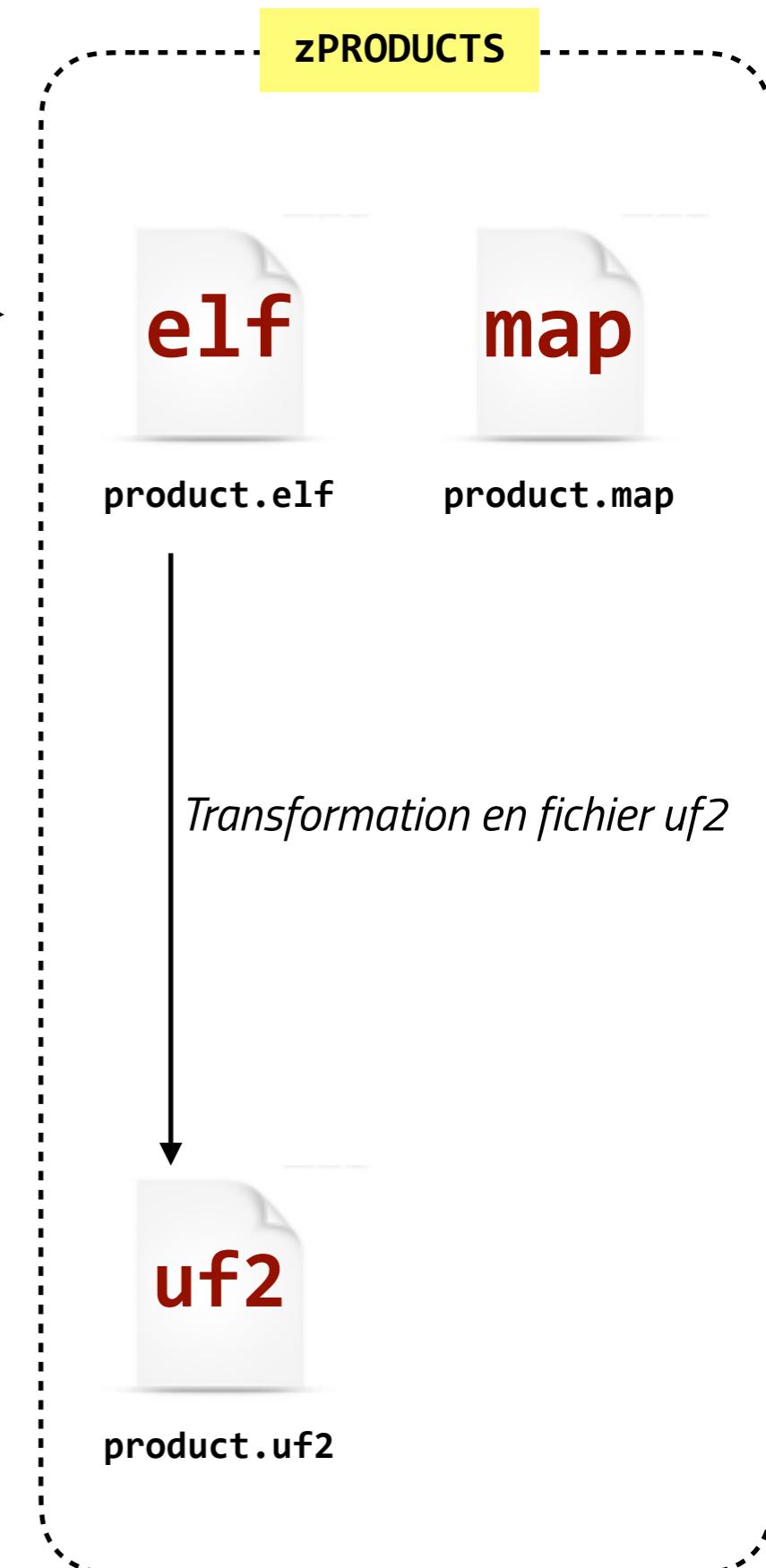
L'outil qui effectue la transformation est **uf2** est `~/raspberry-pi-pico-tools/bin/elf2uf2`. Il est construit lors de l'installation de la chaîne de développement.

L'outil qui effectue le transfert du code vers la cible est **TREEL/dev-files/uf2conv.py**.

Liens :

https://fr.wikipedia.org/wiki/Executable_and_Linkable_Format

<https://github.com/microsoft/uf2>



Étape 01 . 2

Description détaillée du programme

Pas de source C, uniquement C++ (et assembleur)

Utiliser le langage C++ plutôt que le C présente des avantages :

- passage par référence des arguments (`&`) ;
- vrai pointeur nul : `nullptr` ;
- structure de classe ;
- définir des objets non copiables ;
- énumérations plus strictes.

les *exceptions C++* ne sont pas utilisées (l'exécutif ne les prend pas en compte) ;

Les *templates C++* peuvent être utilisées en prenant soin de vérifier que le code engendré n'est pas trop volumineux.

Enfin, dans ce cours, on n'utilise pas de calcul en flottant, on n'en a pas besoin. Le processeur est un Cortex-M0+, c'est-à-dire qu'il n'intègre pas d'unité de calculs en flottant.

Rendre les objets C++ non copiables

Il suffit de déclarer l'opérateur d'affectation et le constructeur de recopie d'une classe avec le qualificatif **delete**.

```
class T {  
  
    //--- Interdire la recopie  
    private: T (const T &) = delete ; // Constructeur de recopie  
    private: T operator = (const T &) = delete ; // Opérateur d'affectation  
};
```

Lors d'une tentative de recopie, une erreur de compilation se déclenche parce que l'opérateur d'affectation et le constructeur de recopie sont marqués comme inexistant.

Fichiers d'en-tête : #pragma once

Dans un fichier d'en-tête, les *gardes d'inclusion* permettent de s'assurer qu'un fichier d'en-tête n'est analysé qu'une seule fois :

```
#ifndef SYMBOLE  
#define SYMBOLE  
    ...declarations...  
#endif
```

La directive **#pragma once** remplace avantageusement les gardes d'inclusion.

```
#pragma once  
    ...declarations...
```

La directive **#pragma once** n'est pas dans le standard C++, mais est implémentée par la plupart des compilateurs (dont GCC).

Liens :

https://en.wikipedia.org/wiki/Pragma_once

https://en.wikipedia.org/wiki/Include_guard

Quels types d'entiers ?

Le langage C permet d'utiliser une multitude de types d'entiers :

- **int**
- **unsigned int**
- **short**
- **short int**
- **signed short**
- **unsigned short**
- **signed short int**
- **unsigned short int**
- ...

La norme C ne fixe pas la taille de tous ces types, mais impose des contraintes de taille entre eux.

Dans ce cours, nous utilisons les types normalisés suivants, dont la taille est garantie :

	Non signé	Signé
8 bits	uint8_t	int8_t
16 bits	uint16_t	int16_t
32 bits	uint32_t	int32_t
64 bits	uint64_t	int64_t

Symboles communs au C++ et à l'assembleur (1/2)

Les symboles C++ sont traduits en assembleur suivant des règles précises (*name mangling*). Par exemple, une fonction dont le prototype serait :

```
void kernel_create_task (uint64_t * inStackBufferAddress,  
                        uint32_t inStackBufferSize,  
                        RoutineTaskType inTaskRoutine) ;
```

apparaîtrait en assembleur sous le nom : **_Z18kernel_create_taskPymPFvvE**.

Écrire les noms issus du *name mangling* est laborieux, dans ce cours on utilise une particularité de GCC, la construction **asm** dans la déclaration d'un prototype de fonction ou d'une variable.

Lien :

https://en.wikipedia.org/wiki/Name_mangling

Symboles communs au C++ et à l'assembleur (2/2)

Dans la déclaration d'un prototype de fonction, la construction **asm** permet de définir explicitement le nom de la fonction en assembleur. Par exemple, dans **setup-loop.h** :

```
void setup0 (void) asm ("cpu.0.setup") ;
```

Évidemment, on travaille sans filet ! Il faut s'assurer que le nom choisi n'entrera pas en collision avec un autre nom. En assembleur ARM, le point « . » est un caractère qui peut apparaître dans un identificateur : le nom **cpu.0.setup** est donc valide. Comme les noms issus de la compilation C++ ne comportent pas de point, on limite ainsi le risque de collision. Cette fonction est appelée en dans le code assembleur **sources/reset-handler-sequential-cpu-0.s** :

```
bl    cpu.0.setup
```

On procèdera de la même façon avec des variables communes au C++ et à l'assembleur. Dans des prochaines étapes, on aura :

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

Le nom choisi doit être différent d'un nom de registre du processeur. Là aussi, utiliser des points est permis et limite le risque de conflits.

Définition des registres de contrôle

400F_F08C	Port Toggle Output Register (GPIOC_PTOR)	32	W (always reads 0)	0000_0000h	63.3.4/2191
400F_F090	Port Data Input Register (GPIOC_PDIR)	32	R	0000_0000h	63.3.5/2191
400F_F094	Port Data Direction Register (GPIOC_PDDR)	32	R/W	0000_0000h	63.3.6/2192
400F_F0C0	Port Data Output Register (GPIOD_PDOR)	32	R/W	0000_0000h	63.3.1/2189
400F_F0C4	Port Set Output Register (GPIOD_PSOR)	32	W (always reads 0)	0000_0000h	63.3.2/2190
400F_F0C8	Port Clear Output Register (GPIOD_PCOR)	32	W (always reads 0)	0000_0000h	63.3.3/2190

Par exemple, le registre **GPIOD_PDOR**: c'est un registre de 32 bits à l'adresse **0x400F_F0C0**.

Son accès est défini par la macro :

```
#define GPIOD_PDOR  (*((volatile uint32_t *) 0x400FF0C0))
```

Ceci permet d'accéder à un registre de contrôle de la même façon que l'on accède à une variable :

```
GPIOD_PDOR = expression ; // Écriture  
variable = GPIOD_PDOR ; // Lecture
```

Description de la table des vecteurs d'interruption

Le RP2040 est particulier, la table des vecteurs d'interruption doit être située à l'adresse 0x10000100, c'est-à-dire 256 octets après le début de la mémoire Flash.

Le fichier **rp2040-interrupt-vectors.s** contient la table des vecteurs d'interruption. Il est de la responsabilité de l'édition de liens de placer cette table à l'adresse 0x10000100, ce qu'il fait à l'aide du nom **isr.vectors.cpu.0** de la section.

Techniquement, la table des interruptions est constituée des vecteurs du processeur Cortex-M0+ (entrées 0 à 15), suivies des interruptions du micro-contrôleur RP2040 (entrées 16 à 47).

```
.section isr.vectors.cpu.0, "a", %progbits

.word __system_stack_end_cpu_0
//--- ARM Core System Handler Vectors
.word reset.handler // 1
.word interrupt.NMI // 2
.word interrupt.HardFault // 3
.word -1 // 4, reserved
.....
.word interrupt.SysTick // 15
//--- Non-Core Vectors
.word interrupt.TIMER_IRQ_0 // 16, IRQ 0
.word interrupt.TIMER_IRQ_1 // 17, IRQ 1
.....
.word interrupt.I2C1_IRQ // 40, IRQ 24
.word interrupt.RTC_IRQ // 41, IRQ 25
.word -1 // 42, reserved, IRQ 26
.....
.word -1 // 47, reserved, IRQ 31
```

Démarrage du micro-contrôleur : résumé

Au démarrage, le micro-contrôleur lit l'entrée correspondant au poussoir BOOTSEL :

- si il est appuyé, il paramètre la liaison USB de façon à ce qu'elle apparaisse comme un disque externe, dont le nom est RPi-RP2 ; ensuite, en y déposant un fichier uf2, la flash est gravée avec le contenu du fichier déposé, et le micro-contrôleur est redémarré ;
- si il est relâché, le programme contenu dans la flash est exécuté.

Démarrage du micro-contrôleur : détail

Voir [datasheet], §2.7 page 156. Le démarrage s'effectue en trois phases.

Les deux processeurs du micro-contrôleur démarrent en :

- chargeant le pointeur de pile par le contenu du mot de 32 bits à l'adresse 0x0 ;
- et en exécutant le code situé à l'adresse 0x4.

Ces adresses correspondent à une ROM de 16 kio (appelée *bootrom*), gravée à la fabrication du composant, donc non modifiable par quelque moyen que ce soit. Son code source est publique et disponible en [bootrom].

L'exécution de la bootrom est la *première phase du boot* et consiste en :

- (1) le processeur 1 est immédiatement arrêté. Ainsi, seul le processeur 0 est actif, le micro-contrôleur se comporte comme par défaut comme un *single core* ;
- (2) si BOOTSEL est appuyé, l'exécution ne sort pas de la bootrom ; si BOOTSEL est relâché :
 - (a) les premiers 256 octets de la flash sont chargés au début de la SRAM5, c'est-à-dire à l'adresse 0x20041000 ;
 - (b) la somme de contrôle des premiers 256 octets de la flash est vérifiée ;
 - (c) en cas de succès, l'exécution se poursuit à partir de l'adresse 0x20041000 : c'est la *deuxième phase* du boot.

Les premiers 256 octets de la Flash (1/2)

Le rôle du code qu'il contient est rendre possible l'exécution du code contenu dans la Flash.

En effet, le RP2040 ne contient pas de Flash interne, et doit être connecté à une Flash externe. Il existe quantité de Flash candidates, chacune ayant des particularités de paramétrage, de taille, de vitesse (sur la carte Raspberry Pi Pico, la Flash est une W25Q16JVUXIQ de *Winbond*). Deux points à noter :

- toutes les Flash candidates doivent accepter l'accès en SPI pour la lecture et l'écriture ;
- l'horloge utilisée est l'horloge interne (6,5 MHz environ), car la bootrom ne connaît pas la fréquence du quartz externe, et il se peut même qu'il n'y ait de quartz externe ;
- mais l'accès en SPI ne permet pas l'exécution ; il faut que soit programmé l'accès via QSPI (SPI avec 4 signaux de données), et que le périphérique XIP (*eXecute In Place*) du RP2040 soit correctement configuré en conformité avec la Flash connectée.

Aussi, la bootrom utilise l'interface générique SPI pour lire (ou écrire, lors d'une opération de Flashage) des données. Ainsi, elle peut charger les 256 premiers octets dans la SRAM5. La somme de contrôle est vérifiée sur les données écrites dans la SRAM5 (ceci permet de rejeter, entre autres, une flash effacée qui ne contient que des « 1 »). Le périphérique XIP n'est toujours pas configuré, donc on peut pas exécuter le code contenu dans la flash : c'est pourquoi on lance l'exécution à la première adresse de SRAM5. Le code va :

- (1) configurer le périphérique XIP (l'exécution en flash est maintenant possible) ;
- (2) et, ensuite, charger le pointeur de pile avec le contenu de la flash à l'adresse 0x10000100, (`__system_stack_end_cpu_0` du vecteur d'interruption) et lancer le code à l'adresse contenue en 0x10000104 : ceci correspond à l'adresse `reset.handler.cpu.0` dans le vecteur d'interruption.

Les premiers 256 octets de la Flash

Comment obtenir le code correspondant ?

Ce code doit être compatible avec la Flash W25Q16JVUXIQ de *Winbond* connectée au RP2040, et enfermé dans une zone de 252 octets suivi des 4 octets de la somme de contrôle.

Le code est contenu dans le fichier `TREEEL/dev-files/sources-common/boot2_w25q080_2_padded_checksum.s`. C'est un fichier assembleur qui contient un tableau de 256 octets. L'éditeur de liens le reconnaîtra au nom de la section, qui est `.boot2`.

Le source est sur le repository du port du Raspberry Pi Pico pour Arduino <https://github.com/earlephilhower/arduino-pico>), fichier `assembly/boot2_w25q080_2_padded_checksum.S`.

Le source est sur le repository du Pico SDK (répertoire `TEEL/archives/pico-sdk-master`, ou bien <https://github.com/raspberrypi/pico-sdk>), fichier `src/rp2_common/boot_stage2/boot2_w25q080.S`.

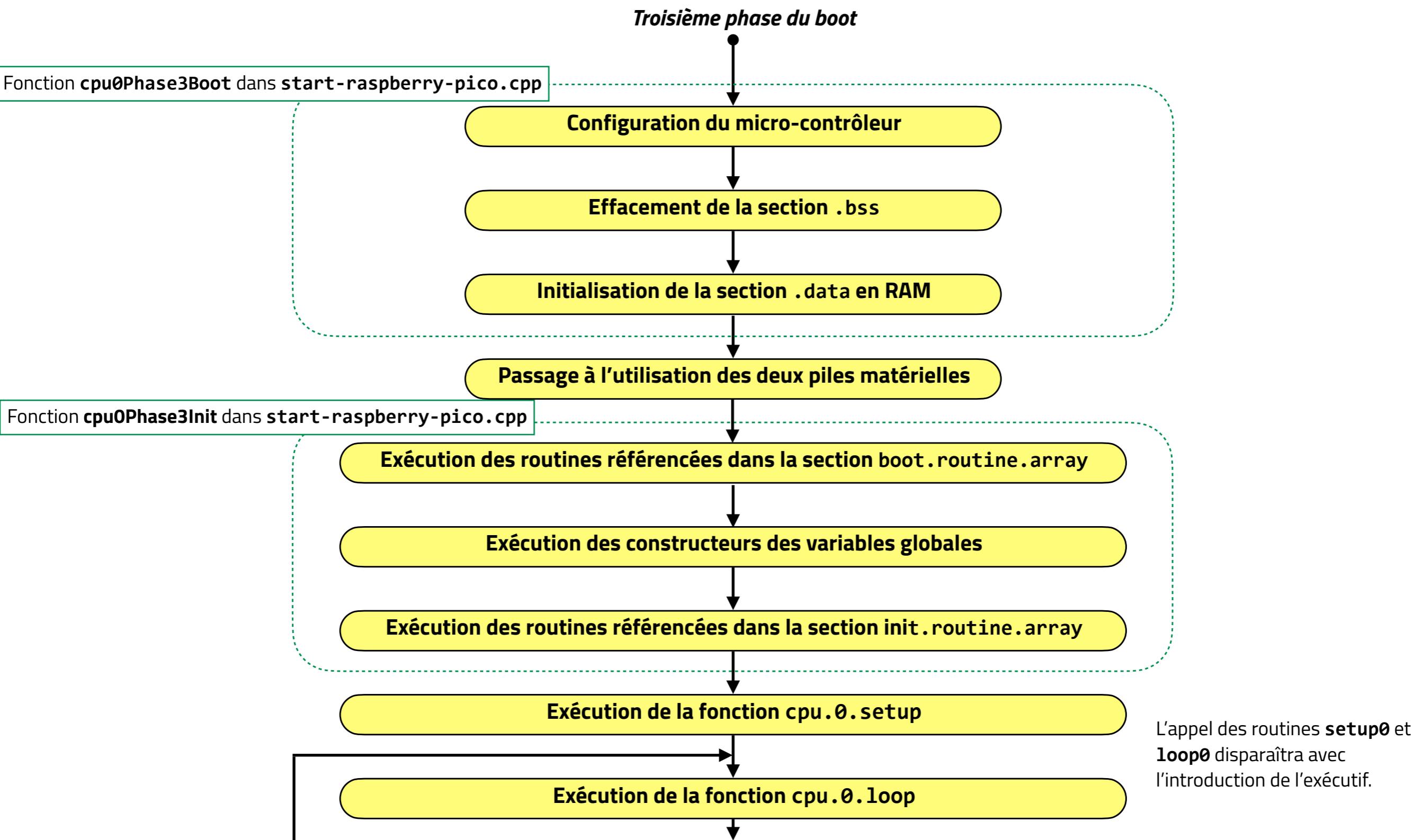
La troisième phase du boot

La *troisième phase du boot* est l'exécution de la fonction **reset.handler.cpu.0** défini dans le fichier **reset-handler-sequential.s**. On est maintenant en terrain connu, beaucoup de micro-contrôleurs ayant un processeur Cortex ont leur table des vecteurs d'interruption à l'adresses **0x0** et donc démarrent directement par l'exécution du **reset.handler.cpu.0**.

Note : en assembleur ARM, **b1** est l'instruction d'appel de sous-programme.

```
reset.handler.cpu.0: // Cortex M0+ boots with interrupts enabled, in Thread mode
//----- Run boot. zero bss section. copy data section
    b1    cpu.0.phase3.boot          Exécution de la fonction C++ cpu0Phase3Boot (définie dans start-raspberry-pico.cpp)
//----- Set PSP: this is stack for background task
    ldr    r0, =background.task.stack + BACKGROUND.STACK.SIZE
    msr    psp, r0
//----- Set CONTROL register
// 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
    movs   r2, #2
    msr    CONTROL, r2            Passage à l'utilisation des deux piles matérielles
//--- 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
    b1    cpu.0.phase3.init        Exécution de la fonction C++ cpu0Phase3Init (définie dans start-raspberry-pico.cpp)
    cpsie i           // Enable interrupts
//----- Run setup, loop
    b1    cpu.0.setup
background.task:
    b1    cpu.0.loop
    b     background.task          Exécution des fonctions C++ setup0 et loop0 (définies dans setup-loop.cpp)
```

Organigramme d'exécution de la 3^e phase du boot (1/2)



Organigramme d'exécution de la 3e phase du boot (2/2)

Configuration du micro-contrôleur. Le micro-contrôleur démarre sur une horloge interne à environ 6,5 MHz ; cette routine programme l'horloge du processeur, du bus, des périphériques à partir de la valeur de l'horloge processeur indiquée par le champ **CPU-MHZ** du fichier **makefile.json**. Attention, durant cette exécution, la mémoire n'est pas initialisée, et elle sera effacée par les étapes suivantes.

Effacement de la section .bss. L'éditeur de lien place toutes les variables globales non explicitement initialisées dans cette section. Le code initialise ces variables à des valeurs correspondant à des zéros binaires.

Initialisation de la section .data en RAM. L'éditeur de lien place toutes les variables globales explicitement initialisées dans cette section, et définit une section en Flash qui contient toutes les valeurs initiales de ces variables. Le code recopie ces valeurs initiales dans les variables en RAM.

Passage à l'utilisation des deux piles matérielles. Le micro-contrôleur démarre avec une seule pile.

Exécution des routines référencées dans la section boot.routine.array. Actuellement, cette section est vide, aucune routine n'est référencée. L'intérêt de cette section sera présenté dans l'étape 03 (modes logiciels).

Exécution des constructeurs des variables globales.

Exécution des routines référencées dans la section init.routine.array. Actuellement, cette section est vide, aucune routine n'est référencée. L'intérêt de cette section sera présenté dans l'étape 03 (modes logiciels).

Fonction cpu.0.setup et cpu.0.loop. Ces symboles assembleur correspondent aux fonctions C++ **setup0** et **loop0**, déclarées dans **setup-loop.h** et implémentées dans **setup-loop.cpp**.

Fonctions setup0 et loop0

Ces deux fonctions contiennent le code « utilisateur ». Elles sont déclarées dans **setup-loop.h** :

```
#pragma once
void setup0 (void) asm ("cpu.0.setup") ;
void loop0 (void) asm ("cpu.0.loop") ;
```

Elles sont implémentées à **setup-loop.cpp** :

```
#include "all-headers.h"

void setup0 (void) {
//--- Configure GP25 as output digital port
    padsbank0_hw->io [25] =
        PADS_BANK0_GPIO0_IE_BITS // Input enable
    | PADS_BANK0_GPIO0_DRIVE_VALUE_4MA // Drive strength
    | PADS_BANK0_GPIO0_SCHMITT_BITS // Enable schmitt trigger
;
    sio_hw->gpio_oe_set = 1 << 25 ;
    iobank0_hw->io [25].ctrl = 5 ;
}

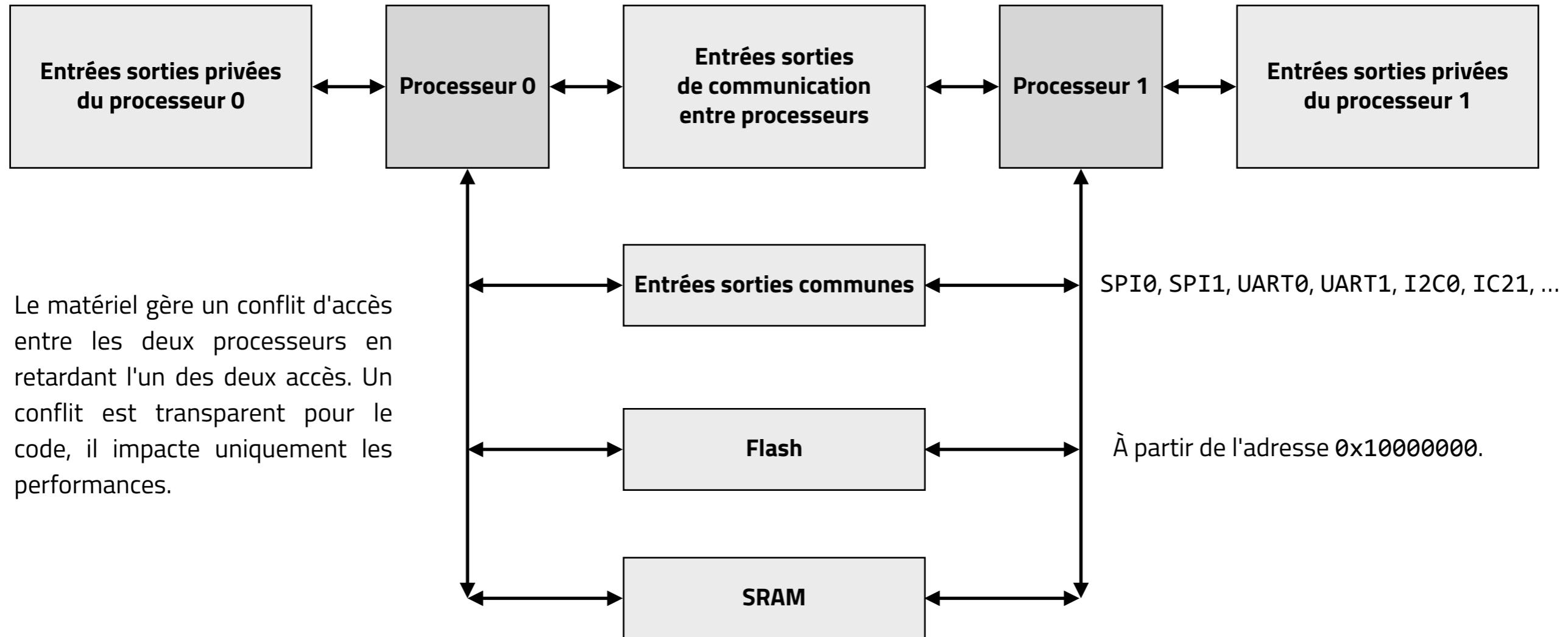
void loop0 (void) {
//--- Drive GP25 high --> led is on
    sio_hw->gpio_set = 1 << 25 ;
//--- Wait...
    for (volatile uint32_t i=0 ; i< 500 * 1000 ; i++) {}
//--- Drive GP25 low --> led is off
    sio_hw->gpio_clr = 1 << 25 ;
//--- Wait...
    for (volatile uint32_t i=0 ; i< 1000 * 1000 ; i++) {}
}
```

Dans cette première étape, les ports d'entrées / sorties sont configurés en programmant directement leurs registres de contrôle. L'étape 05 proposera des fonctions qui permettront d'exprimer plus simplement la configuration.

Dans cette première étape, les attentes temporelles sont effectuées par des boucles. Évidemment, changer la vitesse du processeur (paramètre **CPU-MHZ** du fichier **makefile.json**) change la durée. À partir de l'étape 02, on utilisera le compteur Systick intégré au Cortex-M0+.

Carte mémoire du RP2040 (simplifié)

Voir le détail sur le *RP2040 datasheet*, §2.2.2 page 25.

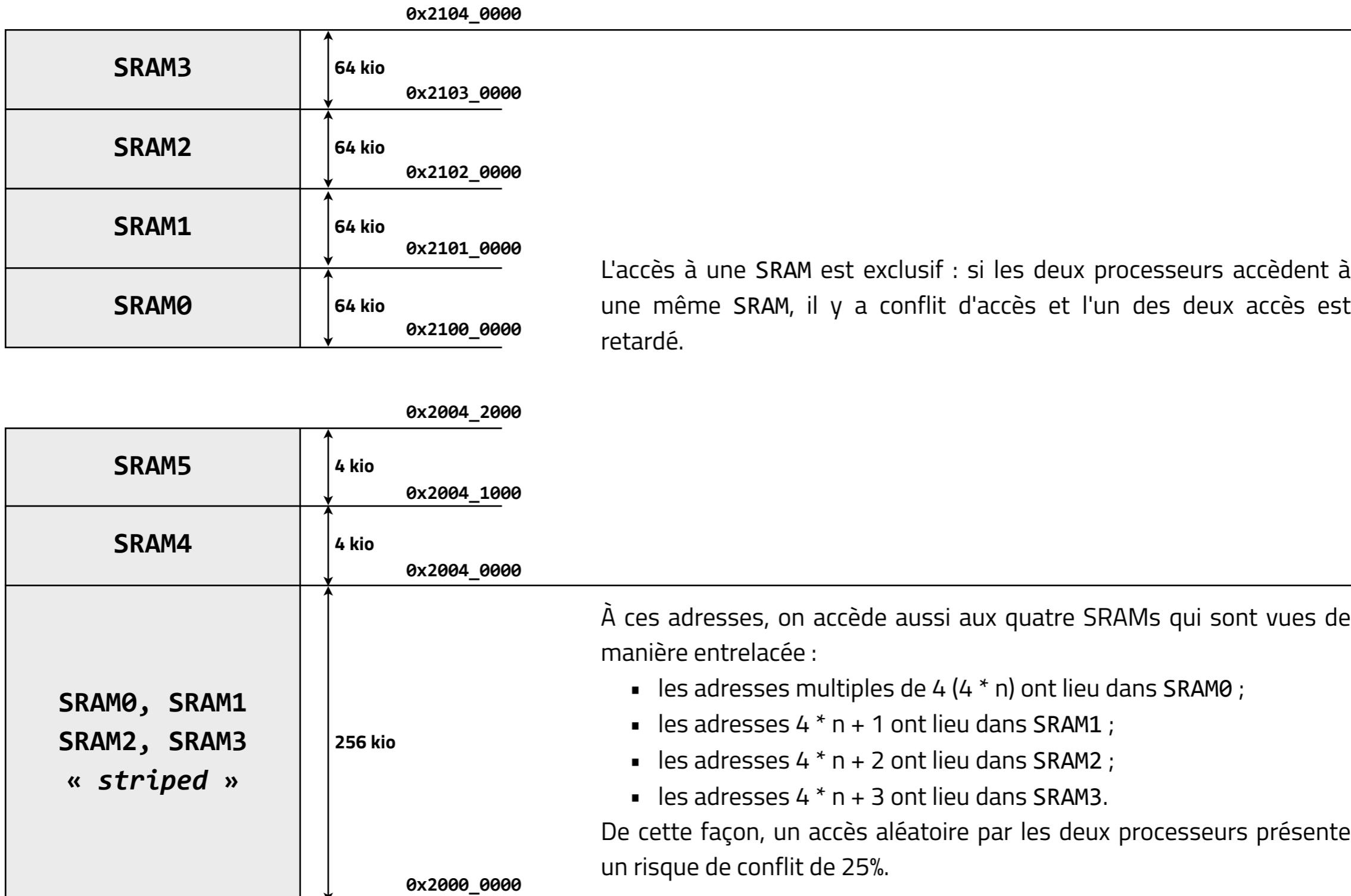


Il y a 6 bancs SRAM :

- SRAM0, SRAM1, SRAM2, SRAM3 : 64 kio chacune ;
- SRAM4, SRAM5 : 4 kio chacune.

La structure est particulière, voir page suivante.

Les SRAM RP2040



À la conception, il faut choisir : soit on accède aux SRAM0, ..., SRAM3 à l'adresse 0x20000000, soit à l'adresse 0x21000000. Ceci est fait grâce au script de l'éditeur de liens.

Répartition des SRAM

À la conception, il faut choisir comment on va utiliser les SRAM. Il n'a pas de configuration idéale, cela dépend de l'application. L'idée est de limiter le risque de conflits qui ralentissent l'exécution, cela ne peut se faire qu'en fonction de l'application.

Ici, on a choisi la répartition des rôles suivante :

- les SRAM0, ..., SRAM3 sont utilisées de manière entrelacée, c'est-à-dire à partir de l'adresse 0x20000000 ;
- les 256 kio de ces mémoires sont utilisées :
 - la première moitié pour l'allocation des variables globales, les piles des tâches ;
 - la seconde moitié pour l'allocation dynamique (à partir de l'étape 17) ;
- la SRAM4 pour la pile système du processeur 0 ;
- la SRAM5 pour la pile système du processeur 1 (donc inutilisée pour l'instant).

C'est script de l'éditeur de liens qui va formaliser cette répartition.

L'édition de liens (1/7)

Le fichier d'édition de liens **dev-files/sources-common/raspberry-pi-pico-flash.ld** décrit comment l'édition des liens est réalisée. C'est un fichier texte constitué d'une séquence de déclarations que nous allons décrire. L'ordre des déclarations est significatif.

Ce bloc décrit la mémoire du micro-contrôleur (Flash, RAM).

```
MEMORY {
    flash      (rx)  : ORIGIN = 0x10000000, LENGTH = 2048k
    global_vars (rwx) : ORIGIN = 0x20000000, LENGTH = 128k
    heap       (rwx) : ORIGIN = 0x20000000 + 128k, LENGTH = 128k
    SRAM4     (rwx) : ORIGIN = 0x20040000, LENGTH = 4k
    SRAM5     (rwx) : ORIGIN = 0x20041000, LENGTH = 4k
}
```

global_vars contient variables globales, les piles des tâches, et son organisation est décrite dans les pages qui suivent. **heap** sera utilisée pour l'allocation dynamique (à partir de l'étape 17).

L'édition de liens (2/7)

Liste des sections des fichiers objets qui ne sont pas placées dans le fichier produit.

```
SECTIONS {  
    /DISCARD/ : {  
        *(.gnu.*);  
        *(.glue_7t);  
        *(.glue_7);  
        *(.ARM.*);  
        *(.comment);  
        *(.debug_frame);  
        *(.vfp11_veneer);  
        *(.v4_bx);  
        *(.iplt);  
        *(.rel.*);  
        *(.igot.plt);  
        *(rel.ARM.*);  
    }  
}
```

L'édition de liens (3/7)

Cette section est placée dans la Flash (« > **flash** » à la dernière ligne) ; comme c'est la première section à placer dans la Flash, elle est placée à partir de l'adresse de début de la flash, 0x10000000.

L'instruction « **__boot2_start__ = .** » affecte au symbole **__boot2_start__** l'adresse courante de placement (symbolisée par un point) : pour cette instruction, l'adresse courante est 0x10000000.

KEEP (*.boot2) ordonne de placer les sections **.boot2** de tous les fichiers (« * »). **KEEP** signifie que cette section est une section racine, c'est-à-dire qu'il ne faut pas l'enlever si elle n'est pas référencée.

```
SECTIONS {
    .text : {
        /*----- BOOT2 */
        __boot2_start__ = . ;
        KEEP (*.boot2) ;
        __boot2_end__ = . ;
        /*----- Vectors */
        __vectors_start_cpu_0 = . ;
        KEEP (*(isr.vectors.cpu.0)) ;
        __vectors_end = . ;
        . = ALIGN (256) ;
        __vectors_start_cpu_1 = . ;
        KEEP (*(isr.vectors.cpu.1)) ;
        __vectors_end_cpu_1 = . ;
        /*----- Code */
        __code_start = . ;
        . = ALIGN(4) ;
        *(.text*) ;
        *(.text) ;
        *(text) ;
        /*----- Boot routine array */
        . = ALIGN (4) ;
        __boot_routine_array_start = . ;
        KEEP (*(boot.routine.array)) ;
        . = ALIGN (4) ;
        __boot_routine_array_end = . ;
```

```
        /*----- Global C++ object constructor call */
        . = ALIGN (4) ;
        __constructor_array_start = . ;
        KEEP (*.init_array) ;
        . = ALIGN (4) ;
        __constructor_array_end = . ;
        /*----- Init routine array */
        . = ALIGN (4) ;
        __init_routine_array_start = . ;
        KEEP (*(init.routine.array)) ;
        . = ALIGN (4) ;
        __init_routine_array_end = . ;
        /*----- Real time interrupt routine array */
        . = ALIGN (4) ;
        __real_time_interrupt_routine_array_start = . ;
        KEEP (*(real.time.interrupt.routine.array)) ;
        . = ALIGN (4) ;
        __real_time_interrupt_routine_array_end = . ;
        /*----- ROM data */
        . = ALIGN(4);
        *(.rodata*);
        . = ALIGN(4);
        /*----- End */
        __code_end = . ;
    } > flash
}
```

L'édition de liens (4/7)

```
SECTIONS {
    .bss : {
        . = ALIGN(4);
        __bss_start = . ;
        * (.bss*) ;
        . = ALIGN(4);
        * (COMMON) ;
        . = ALIGN(4);
        __bss_end = . ;
    } > global_vars
}
```

L'éditeur de liens range dans la section **.bss** du fichier produit le contenu des sections **.bss*** et **COMMON** des fichiers objet. Le caractère « * » dans **.bss*** signifie toute section dont le nom commence par **.bss**.

Les sections **.bss*** et **COMMON** contiennent les variables globales non initialisées explicitement dans les sources.

Deux symboles sont définis, **__bss_start** et **__bss_end** et encadrent ces variables. Les lignes « **. = ALIGN(4)** » assurent que ces symboles ont une valeur multiple de 4. Au démarrage du micro-contrôleur, cette zone est mise à zéro (*Effacement de la section .bss*, voir dans les pages précédentes). L'alignement des symboles permet au code d'effacement d'utiliser des instructions d'écriture de mots de 32 bits. **Voir aussi la description de l'étape 04-boot-and-init-routines.**

L'édition de liens (5/7)

```
SECTIONS {
    .data : AT (__code_end) {
        . = ALIGN (4) ;
        __data_start = . ;
        * (.data*) ;
        . = ALIGN (4) ;
        __data_end = . ;
    } > global_vars
}

__data_load_start = LOADADDR (.data) ;
__data_load_end   = LOADADDR (.data) + SIZEOF (.data) ;
```

Ce bloc concerne les variables explicitement initialisées. L'écriture est plus compliquée, il y a en fait deux opérations à réaliser :

- réservé l'emplacement des variables dans la RAM ;
- placer les valeurs initiales de ces variables en Flash.

Au démarrage du micro-contrôleur, la recopie des valeurs initiales est effectuée (*Initialisation de la section .data en RAM*, voir dans les pages précédentes). L'alignement des symboles permet au code d'effacement d'utiliser des instructions d'écriture de mots de 32 bits. **Voir aussi la description de l'étape 04-boot-and-init-routines.**

L'édition de liens (6/7)

```
SECTIONS {
    .system_stack_cpu_0 : {
        . = ALIGN (8) ;
        __system_stack_start_cpu_0 = . ;
        . += 4096 ;
        . = ALIGN (4) ;
        __system_stack_end_cpu_1 = . ;
    } > SRAM4
}
```

Ce bloc réserve 4096 octets pour la pile système du processeur 0, la SRAM4 est complètement utilisée. Le symbole **__system_stack_end** qui marque sa fin apparaît au début de la table des vecteurs d'interruption, c'est donc la valeur initiale du pointeur de pile.

```
SECTIONS {
    .system_stack_cpu_1 : {
        . = ALIGN (8) ;
        __system_stack_start_cpu_1 = . ;
        . += 4096 ;
        . = ALIGN (4) ;
        __system_stack_end_cpu_1 = . ;
    } > SRAM5
}
```

Idem pour le processeur 1.

L'édition de liens (7/7)

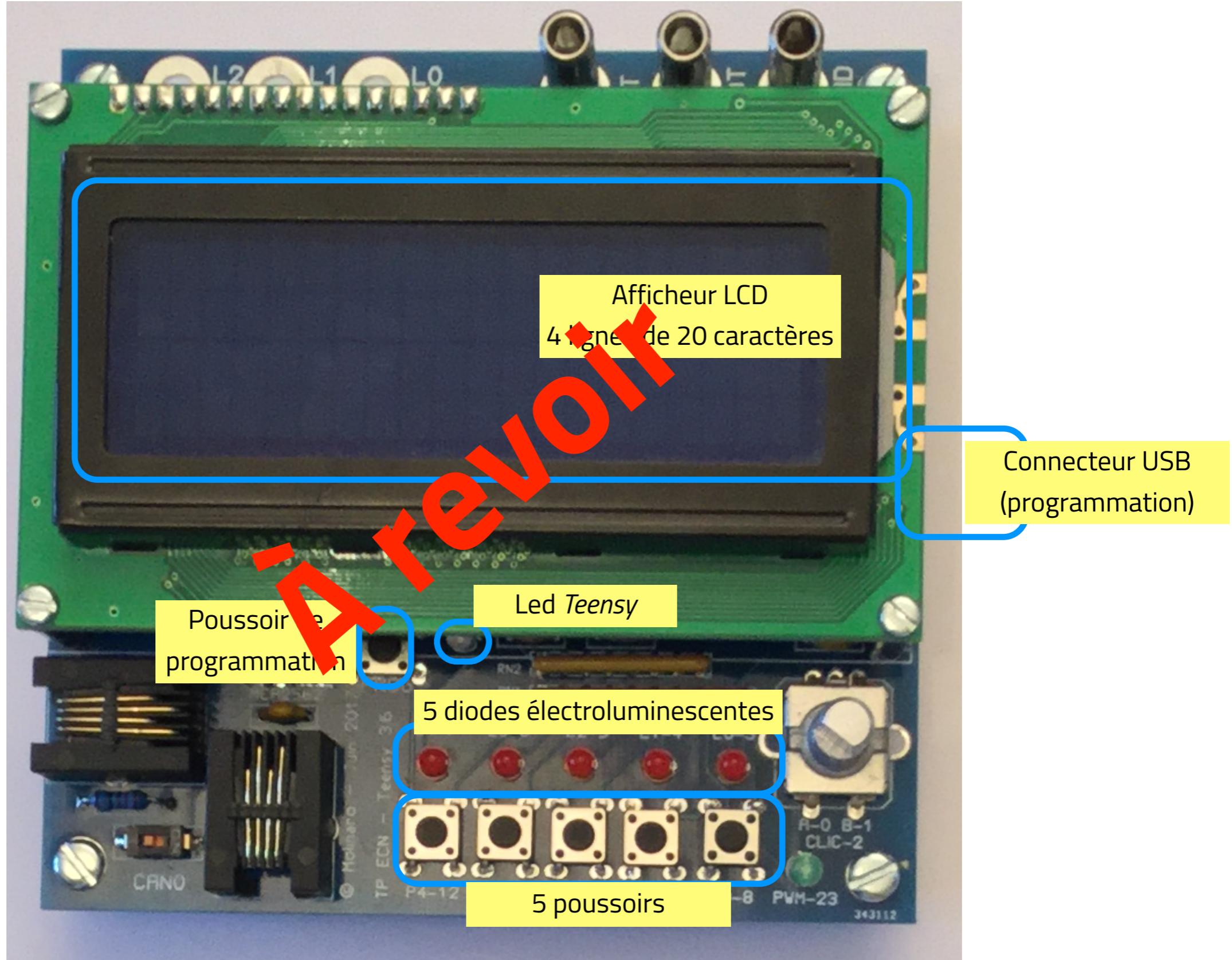
```
__heap_start = ORIGIN (heap) ;
__heap_end = ORIGIN (heap) + LENGTH (heap) ;
```

Le reste des mémoires (SRAM0, ..., SRAM3) est allouée au *tas (heap)*, c'est-à-dire pour l'allocation dynamique. L'allocation dynamique sera présentée à l'étape 17. Les symboles **`__heap_start`** et **`__heap_end`** encadrent cette zone.

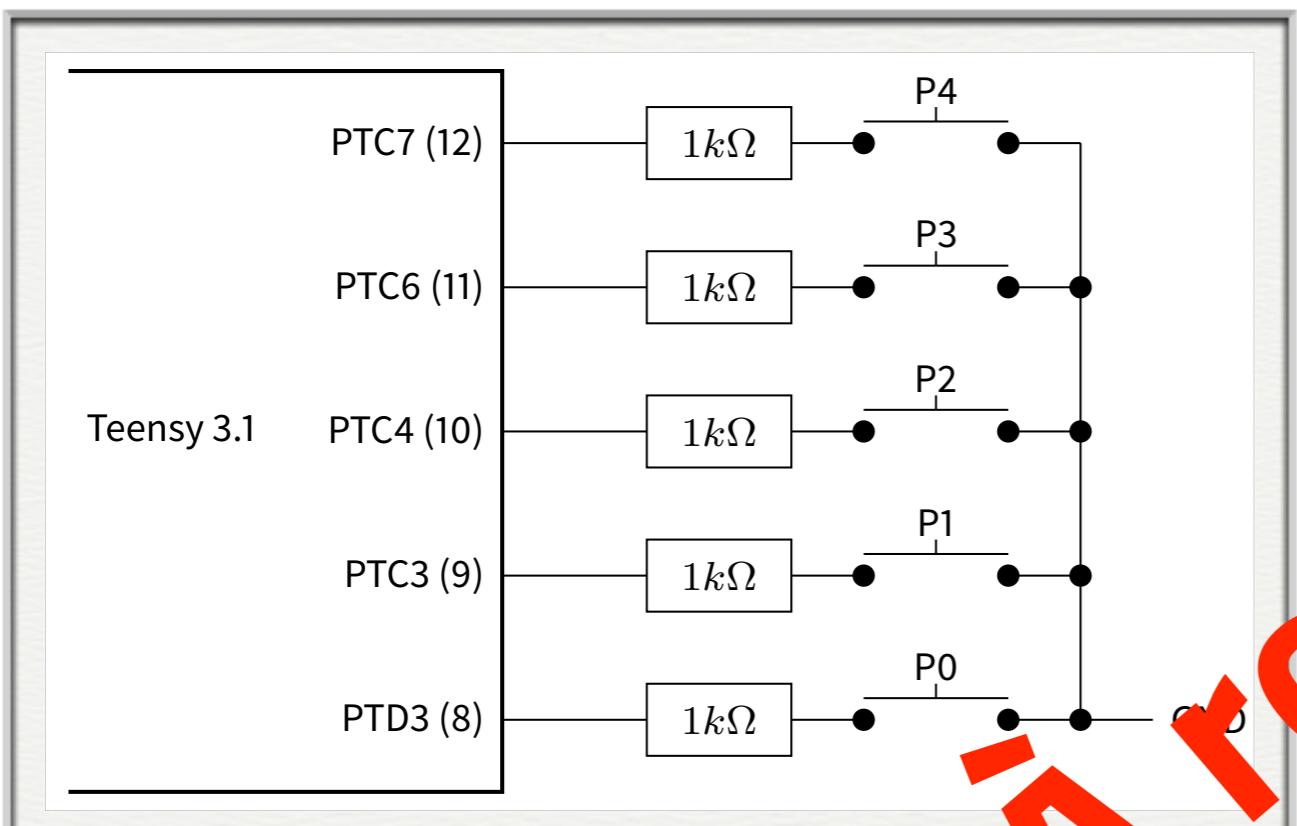
Étape 01 . 3

Carte de développement

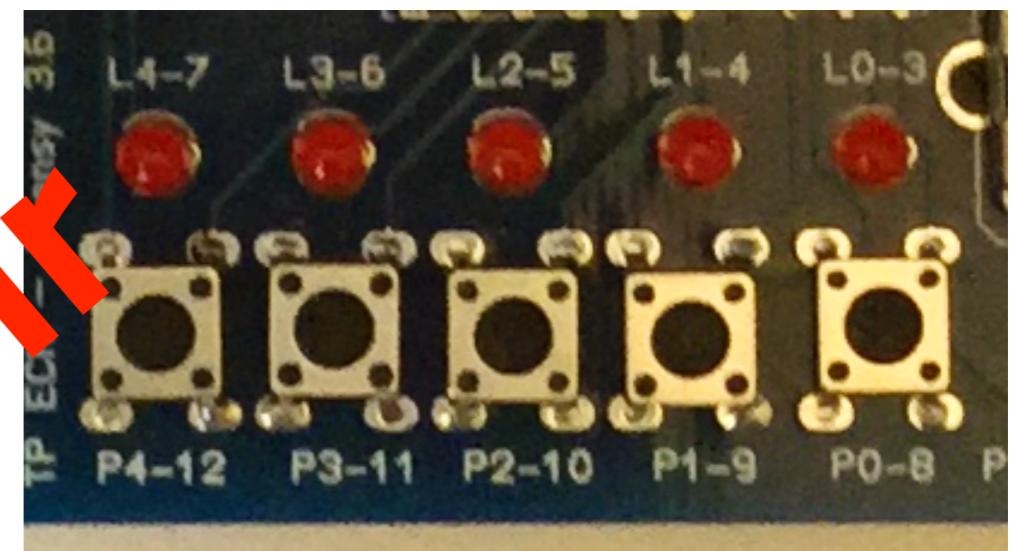
Entrées / sorties de la carte de TP



Connexion des entrées logiques



À revoir

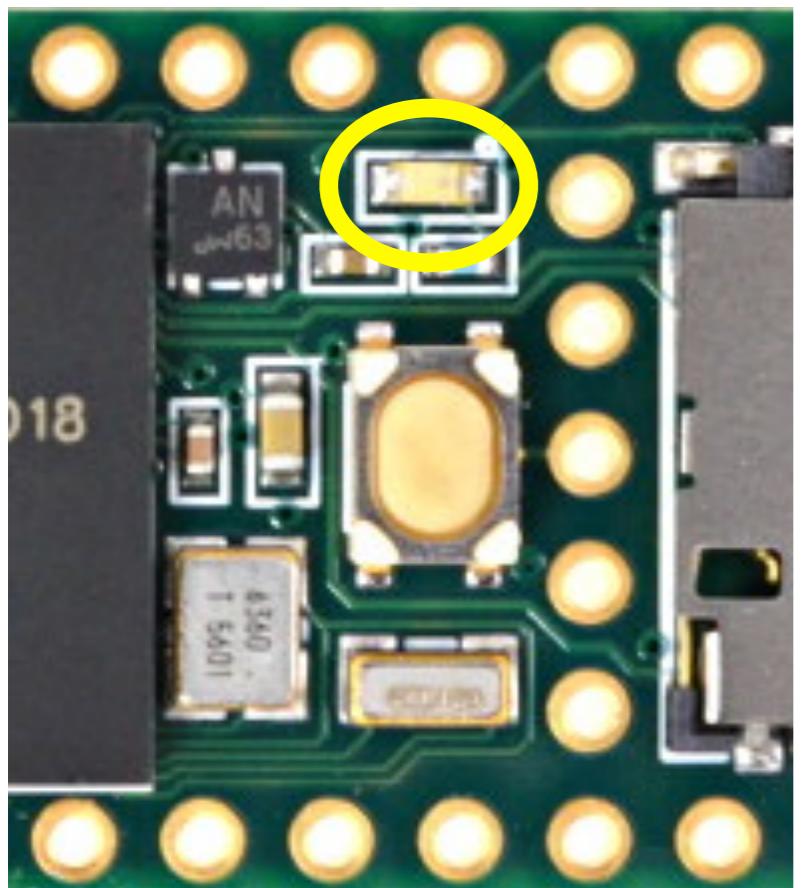


Poussoir P0 appuyé : le port #8 est au niveau bas.

Poussoir P0 relâché : le port #8 est au niveau haut, sous réserve qu'il soit programmé en mode INPUT_PULLUP.

Diode électroluminescente sur la carte Teensy

Led Teensy



Comme cette led est masquée par l'afficheur LCD, elle est répétée sur la carte de développement.



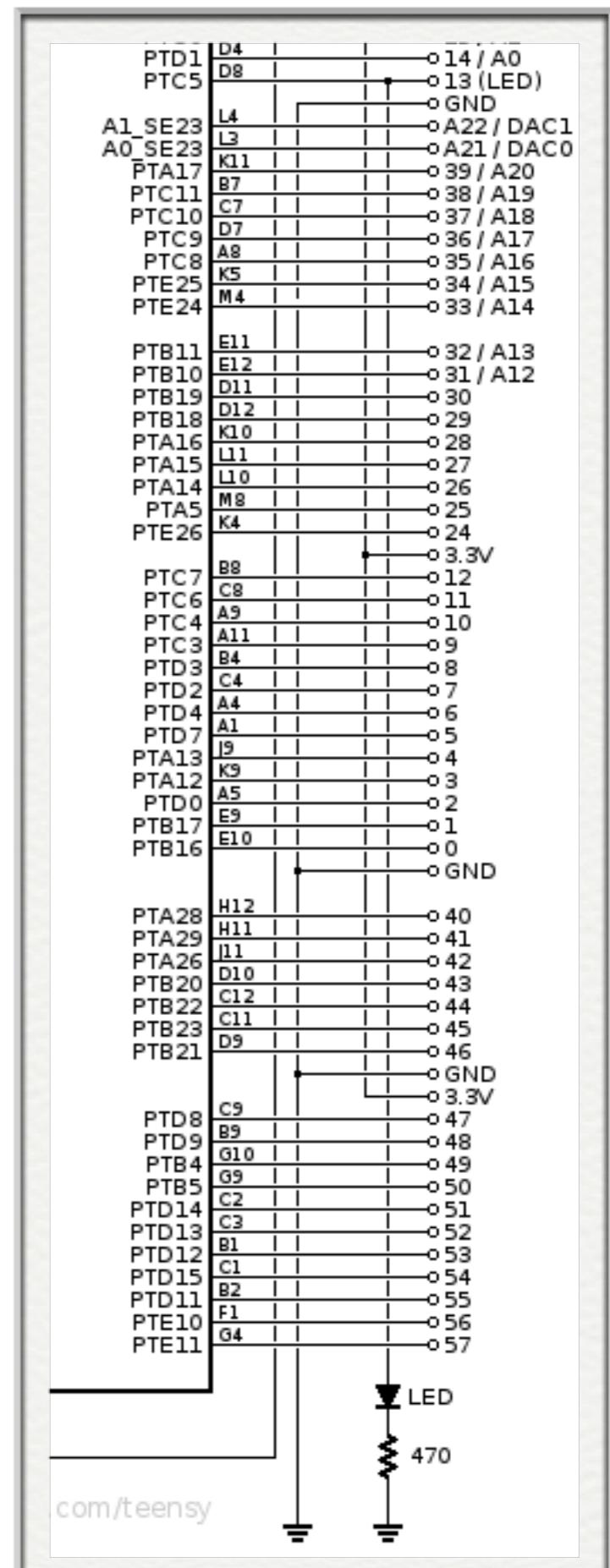
Note : cette led sera utilisée dans ce cours pour refléter l'activité processeur.

À revoir

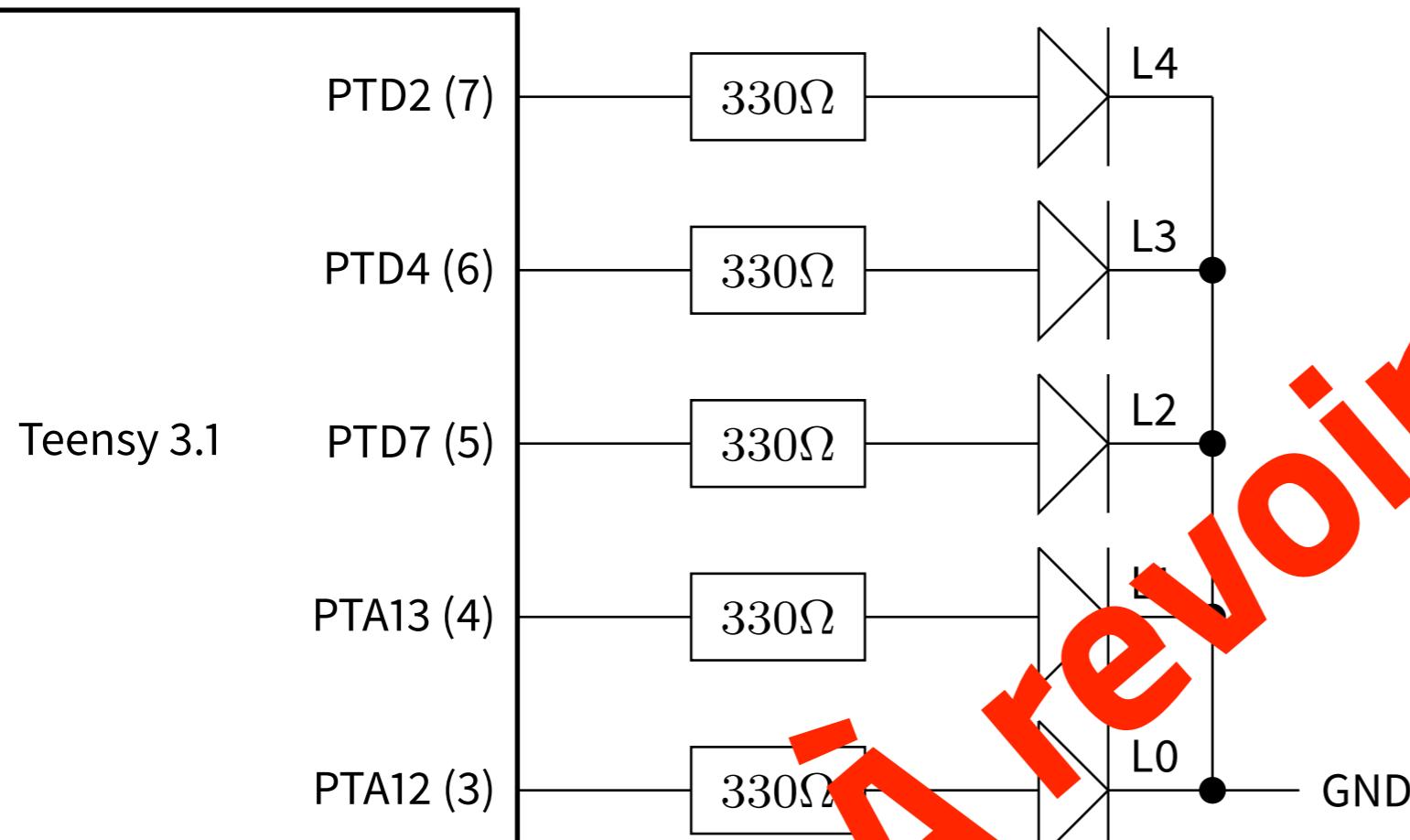
Port n°13 en sortie logique au niveau bas : le micro-contrôleur impose une tension proche de zéro Volt, la led est éteinte.

Port n°13 en sortie logique au niveau haut : le micro-contrôleur impose une tension proche de 3,3V, la led est allumée.

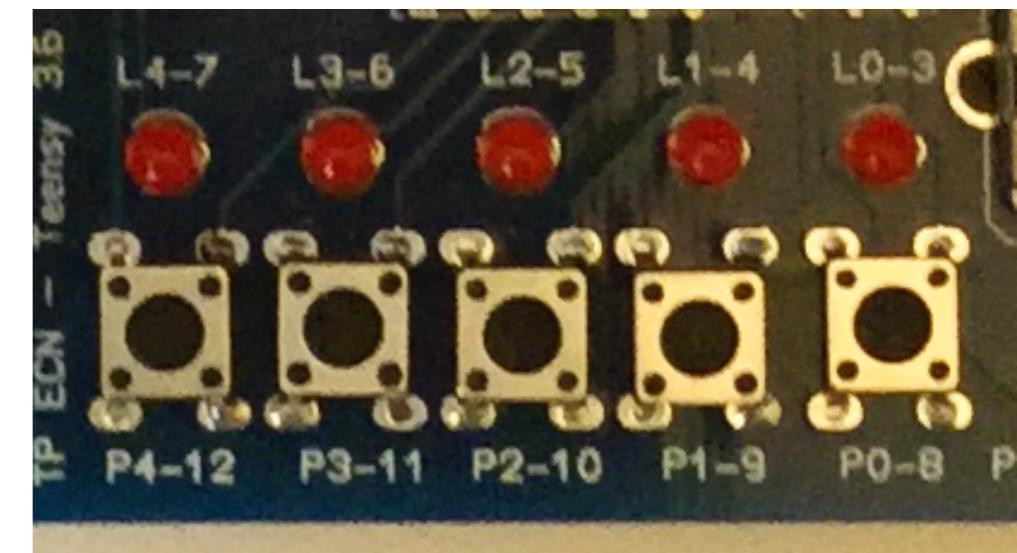
Étape 01.3



Diodes électroluminescentes sur la carte de TP



A révoir



Port n°7 en sortie logique au niveau bas : le micro-contrôleur impose une tension proche de zéro Volt, la led L4 est éteinte.

Port n°7 en sortie logique au niveau haut : le micro-contrôleur impose une tension proche de 3,3V, la led L4 est allumée.

Étape 01 . 4

**Compléments : opérateurs du
langage C**

L'opérateur « et bit-à-bit » : « & »

Attention, ne pas confondre « & » et « && ».

L'opérateur « & » effectue une opération *et* bit-à-bit sur des nombres entiers.

Voici un exemple sur des nombres de 8 bits :

	7	6	5	4	3	2	1	0
A	1	0	0	0	1	0	0	0
B	1	0	0	0	0	0	1	0
A & B	1	0	0	0	0	0	0	0

L'opérateur « et logique » : « && »

L'opérateur && : en C, cet opérateur n'évalue pas l'opérande de droite si celui de gauche est faux.

Ne pas confondre avec l'opérateur **&** (page précédente).

Par exemple :

```
if (a && b) {  
    X  
}else{  
    Y  
}
```

est équivalent à :

```
if (a) {  
    if (b) {  
        X  
    }else{  
        Y  
    }  
}else{  
    Y  
}
```

L'opérateur « ou bit-à-bit » : « | »

Attention, ne pas confondre « | » et « || ».

L'opérateur « | » effectue une opération *ou* bit-à-bit sur des nombres entiers.

Voici un exemple sur des nombres de 8 bits :

	7	6	5	4	3	2	1	0
A	1	0	0	0	1	0	0	0
B	1	0	0	0	0	0	1	0
A B	1	0	0	0	1	0	1	0

L'opérateur « ou logique » : « || »

L'opérateur || : en C, cet opérateur n'évalue pas l'opérande de droite si celui de gauche est vrai.

Ne pas confondre avec l'opérateur | (page précédente).

Par exemple :

```
if (a || b) {  
    X  
}else{  
    Y  
}
```

est équivalent à :

```
if (a) {  
    X  
}else if (b) {  
    X  
}else{  
    Y  
}
```

L'opérateur de décalage à gauche : « << »

Cet opérateur décale à gauche en insérant des zéros.

Un exemple sur des nombres de 8 bits :

	7	6	5	4	3	2	1	0
A	a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀
A << 1	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀	0
A << 2	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀	0	0

Conséquence : **1 << n** est le nombre dont le seul bit à **1** est le bit n°n.

	7	6	5	4	3	2	1	0
1 << 0	0	0	0	0	0	0	0	1
1 << 1	0	0	0	0	0	0	1	0
1 << 2	0	0	0	0	0	1	0	0

L'opérateur de décalage à droite : << >> >>

Sur des nombres non signés, cet opérateur insère des zéros.

A est un nombre non signé	7	6	5	4	3	2	1	0
A	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
A >> 1	0	a_7	a_6	a_5	a_4	a_3	a_2	a_1
A >> 2	0	0	a_7	a_6	a_5	a_4	a_3	a_2

Sur des nombres signés, cet opérateur conserve le bit de poids fort.

A est un nombre signé	7	6	5	4	3	2	1	0
A	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
A >> 1	a_7	a_7	a_6	a_5	a_4	a_3	a_2	a_1
A >> 2	a_7	a_7	a_7	a_6	a_5	a_4	a_3	a_2

L'opérateur « ou exclusif bit-à-bit » : « ^ »

L'opérateur « ^ » effectue une opération *ou exclusif* bit-à-bit sur des nombres entiers.

Voici un exemple sur des nombres de 8 bits :

	7	6	5	4	3	2	1	0
A	a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀
B	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
A ^ B	a ₇ ⊕ b ₇	a ₆ ⊕ b ₆	a ₅ ⊕ b ₅	a ₄ ⊕ b ₄	a ₃ ⊕ b ₃	a ₂ ⊕ b ₂	a ₁ ⊕ b ₁	a ₀ ⊕ b ₀

	7	6	5	4	3	2	1	0
A	1	0	0	0	1	0	0	0
B	1	0	0	0	0	0	1	0
A ^ B	0	0	0	0	1	0	1	0

L'opérateur « complément bit-à-bit » : « ~ »

L'opérateur « ~ » effectue une opération *complément* sur tous les bits d'un nombre entier.

Ne pas confondre avec l'opérateur « ! » (voir page suivante).

Voici un exemple sur un nombre de 8 bits :

	7	6	5	4	3	2	1	0
A	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$\sim A$	$\neg a_7$	$\neg a_6$	$\neg a_5$	$\neg a_4$	$\neg a_3$	$\neg a_2$	$\neg a_1$	$\neg a_0$

	7	6	5	4	3	2	1	0
A	1	0	0	0	1	0	0	0
$\sim A$	0	1	1	1	0	1	1	1

On peut utiliser cet opérateur pour « simplifier » certaines écritures. Par exemple :

```
uint32_t v = 0xFFFFFFFF ;
```

peut être écrit

```
uint32_t v = ~0 ;
```

L'opérateur « complément logique » : « ! »

Ne pas confondre avec l'opérateur « ~ » (voir page précédente).

L'opérateur « ! » effectue une opération *complément logique* sur un nombre entier n , c'est-à-dire :

- si n est égal à zéro, $!n$ vaut 1 ;
- si n est différent de zéro, $!n$ vaut 0.

En particulier, « ! » n'est pas idempotent, c'est-à-dire que dans le cas général $!!n \neq n$.

Remarque : $!!n \Leftrightarrow n != 0$

Étape 02 — Systick

Description de cette étape

Objectif :

- compter précisément le temps, grâce à un compteur intégré.

Problèmes à résoudre :

- configuration du compteur intégré ;
- écrire une routine d'attente active **busyWaitDuring**.

Objectif :

- écrire un programme *blinked* qui appelle **busyWaitDuring**. Pour cela, commencer par dupliquer le programme précédent.

SysTick

SysTick est un compteur implémenté dans le processeur Cortex-M0+.

Ses possibilités sont limitées mais il est très simple à mettre en œuvre.

Par défaut, il est inactivé, c'est-à-dire arrêté.

C'est en fait un décompteur cyclique : $0, N, N-1\dots, 1, 0, N, \dots$ N est une valeur programmable. La période est $N+1$.

Lorsqu'il est rechargé à N , il peut être configuré pour déclencher l'interruption n°15 (ce qui n'est pas fait dans cette étape).

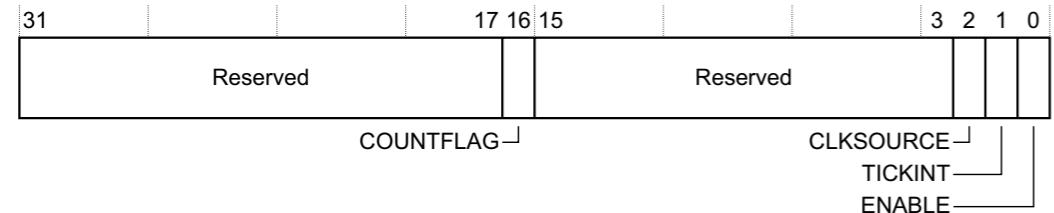
Registres du timer SysTick

Source : ARM®v7-M Architecture Reference Manual, référence ARM DDI 0403E.b.

Address	Name	Type	Reset	Description
0xE000E010	SYST_CSR	RW	0x0000000xa	<i>SysTick Control and Status Register, SYST_CSR</i>
0xE000E014	SYST_RVR	RW	UNKNOWN	<i>SysTick Reload Value Register, SYST_RVR</i> on page B3-678
0xE000E018	SYST_CVR	RW	UNKNOWN	<i>SysTick Current Value Register, SYST_CVR</i> on page B3-678
0xE000E01C	SYST_CALIB	RO	IMP DEF	<i>SysTick Calibration value Register, SYST_CALIB</i> on page B3-679
0xE000E020-	-	-	-	Reserved
0xE000E0FC				

a. See register description for information about the reset value of SYST_CSR bit[2]. All other bits reset to 0.

The SYST_CSR bit assignments are:



Bits[31:17] Reserved.

COUNTFLAG, bit[16]

Indicates whether the counter has counted to 0 since the last read of this register:

- 0 Timer has not counted to 0.
- 1 Timer has counted to 0.

COUNTFLAG is set to 1 by a count transition from 1 to 0.

COUNTFLAG is cleared to 0 by a software read of this register, and by any write to the Current Value register. Debugger reads do not clear the COUNTFLAG.

This bit is read only.

Bits[15:3]

Reserved.

CLKSOURCE, bit[2] Indicates the SysTick clock source:

- 0 SysTick uses the IMPLEMENTATION DEFINED external reference clock.
- 1 SysTick uses the processor clock.

If no external clock is provided, this bit reads as 1 and ignores writes.

TICKINT, bit[1]

Indicates whether counting to 0 causes the status of the SysTick exception to change to pending:

- 0 Count to 0 does not affect the SysTick exception status.
- 1 Count to 0 changes the SysTick exception status to pending.

Changing the value of the counter to 0 by writing zero to the SysTick Current Value register to 0 never changes the status of the SysTick exception.

ENABLE, bit[0]

Indicates the enabled status of the SysTick counter:

- 0 Counter is disabled.
- 1 Counter is operating.

Programmation du timer SysTick

```
void startSystick (void) {  
    systick_hw->rvr = CPU_MHZ * 1000 - 1 ; // Underflow every ms  
    systick_hw->cvr = 0 ;  
    systick_hw->csr = M0PLUS_SYST_CSR_CLKSOURCE_BITS | M0PLUS_SYST_CSR_ENABLE_BITS ;  
}
```

La séquence de configuration du SysTick comporte trois instructions.

```
systick_hw->rvr = CPU_MHZ * 1000 - 1 ; // Underflow every ms
```

Cette instruction fixe la période du compteur SysTick à `CPU_MHZ * 1000`. L'horloge de SysTick est l'horloge processeur. Dans le fichier `makefile.json`, le paramètre `CPU-MHZ` permet de choisir cette fréquence, qui apparaît dans le fichier engendré `zSOURCES/base.h` sous le nom `CPU_MHZ`. Ainsi, le compteur SysTick est rechargé toutes les millisecondes, indépendamment de la fréquence du processeur.

```
systick_hw->cvr = 0 ;
```

Met à zéro la valeur courant du timer SysTick.

```
systick_hw->csr = M0PLUS_SYST_CSR_CLKSOURCE_BITS | M0PLUS_SYST_CSR_ENABLE_BITS ;
```

Configure et démarre le timer :

- `M0PLUS_SYST_CSR_CLKSOURCE_BITS`, l'horloge du timer est l'horloge processeur (certains micro-contrôleurs peuvent sélectionner une autre horloge) ;
- `M0PLUS_SYST_CSR_ENABLE_BITS`, le compteur est activé.

Fonction busyWaitDuring

La fonction **busyWaitDuring** attend que le nombre de milli-secondes indiquées en argument soient écoulées.

L'attente est qualifiée **active** car le processeur est immobilisé durant l'attente du délai.

L'indicateur booléen **COUNTFLAG** (bit n°16 du registre **SYST_CSR**) est mis à 1 par le matériel à chaque fois que le compteur est passé de 1 à 0 (c'est-à-dire toutes les milli-secondes). Lire le registre **SYST_CSR** efface automatiquement ce bit.

```
void busyWaitDuring (const uint32_t inDelayMS) {
    for (uint32_t i=0 ; i<inDelayMS ; i++) {
        // Busy wait, polling COUNTFLAG
        while ((systick_hw->csr & M0PLUS_SYST_CSR_COUNTFLAG_BITS) == 0) {}
    }
}
```

Travail à faire

Écrire un programme *blinked* utilisant la fonction **busyWaitDuring** pour exprimer les délais. Pour cela :

- dupliquer le programme précédent et le renommer ;
- écrire les fonctions **startSystick** et **busyWaitDuring** dans un nouveau fichier **time.cpp** ;
- écrire le fichier d'en-tête **time.h**, qui déclare les prototypes de ces deux fonctions ;
- modifier la fonction **setup0** dans le fichier **setup-loop.cpp** de façon à appeler **startSystick** ;
- modifier la fonction **loop0** dans le fichier **setup-loop.cpp** de façon à appeler **busyWaitDuring**.

Indications :

- dans l'écriture du fichier d'en-tête **time.h**, il ne faut faire aucune supposition sur l'ordre d'inclusion des fichiers ; aussi, il est recommandé de faire précéder la déclaration des prototypes par une ligne **#include <stdint.h>** (elle-même précédée de **#pragma once**) ;
- la première ligne du fichier **time.cpp** doit être **#include "all-headers.h"** : ainsi, on est sûr que toutes les déclarations sont disponibles.

Étape 03 — Modes logiciels

Description de cette étape

Objectif : ajouter aux routines les annotations de mode logiciel.

Travail à réaliser : il est décrit à la dernière page.

Les modes du processeur

Un processeur Cortex-M0+ est dans l'un des deux modes suivants :

- le mode ***thread*** ;
- le mode ***handler***.

Les propriétés de ces deux modes sont configurables, aussi une description complète est trop vaste : nous nous limitons dans ce cours à la description des choix qui ont été faits.

Le mode ***thread*** sera le mode d'exécution des tâches. Par défaut, les interruptions sont activées dans ce mode. Le mode ***thread*** a sa propre pile. Actuellement, tout le code s'exécute dans ce mode : cela ne sera plus le cas à partir de l'étape 06 où apparaissent les interruptions.

Le mode ***handler*** sera le mode des routines d'interruption et des services de l'exécutif. Ceux-ci ne sont pas interruptibles. Le mode ***handler*** a sa propre pile.

Les modes logiciels

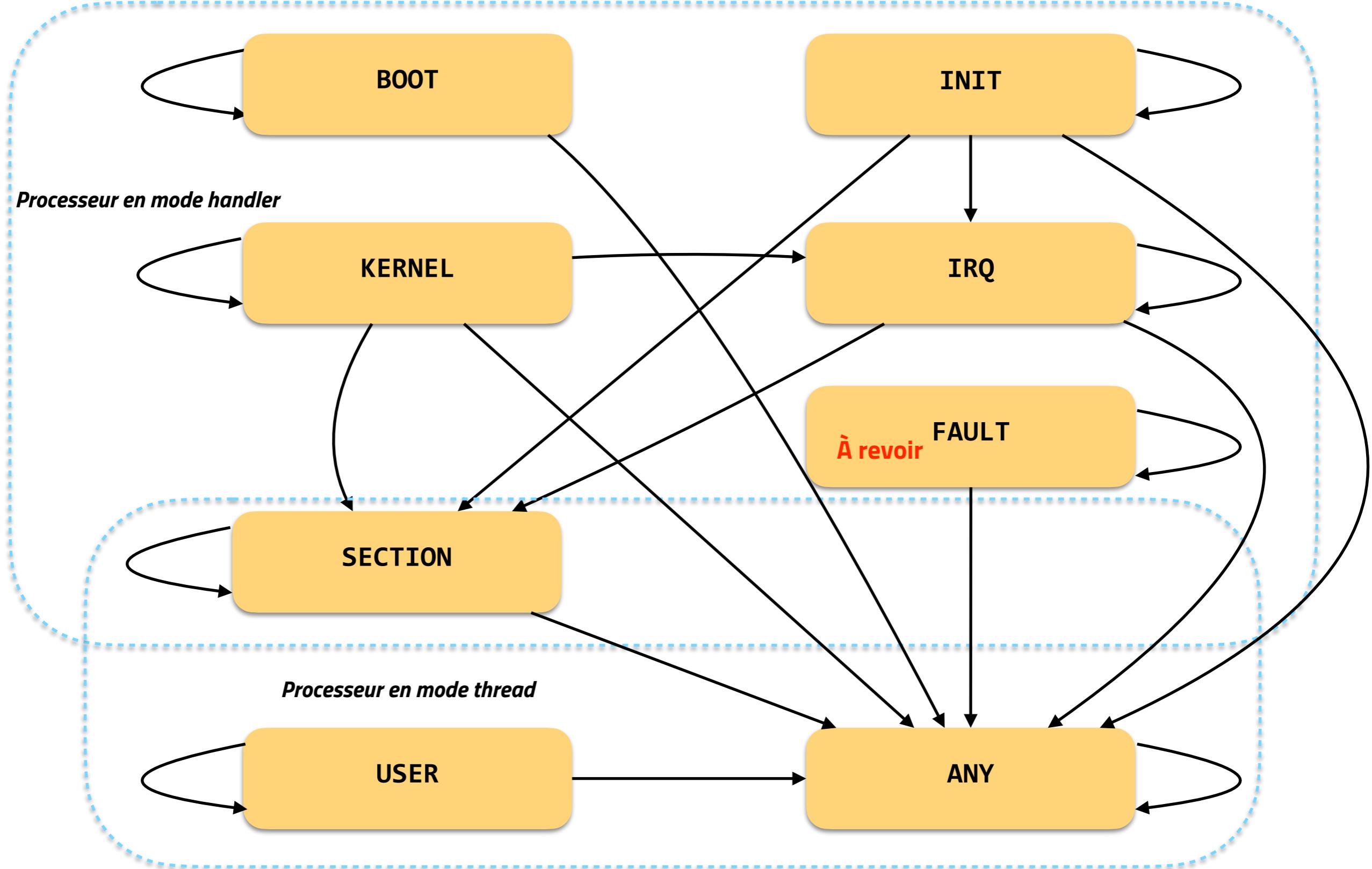
Le C et le C++ ne permettent pas de distinguer les fonctions qui doivent s'exécuter en mode *handler* et celles qui doivent s'exécuter en mode *thread*. Un appel incorrect n'est pas décelé à la compilation, et provoque le plus souvent un plantage à l'exécution.

De toute façon, ces deux seuls modes ne suffisent pas à caractériser toutes les classes de routines qui peuvent exister dans un exécutif.

Ce cours définit donc les huit modes logiciels suivants :

Mode logiciel	Commentaire
BOOT	C'est le mode des routines référencées dans la section boot.routine.array . Dans ce mode, les constructeurs des variables globales ne sont pas encore appelés, il ne faut pas référencer d'instances de classe C++. Les interruptions sont masquées.
INIT	C'est le mode des routines référencées dans la section init.routine.array . Dans ce mode, les constructeurs des variables globales ont été appelés. Les interruptions sont masquées.
KERNEL	Ce sera le mode des services de l'exécutif qui peuvent être appelés par le code d'une tâche, à travers une construction qui utilise l'instruction svc (Service Call). Exemple : la primitive P d'un sémaphore.
IRQ	Ce sera le mode des services de l'exécutif qui peuvent être appelés par le code d'une tâche (comme pour le mode KERNEL), ou par une interruption matérielle. Exemple : la primitive V d'un sémaphore.
SECTION	Mode d'une routine ininterruptible. Selon le contexte d'appel, le processeur peut être en mode <i>thread</i> ou en mode <i>handler</i> .
FAULT	Un processeur Cortex-M4 se surveille, et une exception Fault se déclenche en cas d'erreur. Actuellement, dans cette situation, l'exécution plante. Dans l'étape 09, on prendra en compte cette exception pour afficher un message d'erreur (à revoir).
USER	C'est actuellement le mode des routines setup0 et loop0 . Quand l'exécutif sera ajouté, ce sera le mode d'exécution des tâches.
ANY	Mode d'une routine pouvant être appelée à partir de n'importe quel mode.

Graphe des modes logiciels



Une flèche signifie que l'appel direct est autorisé : par exemple, une routine en mode **KERNEL** peut appeler directement une routine déclarée avec le mode **IRQ**.

Comment exprimer les modes logiciels (1/4)

Chaque fonction pour laquelle le mode logiciel est important va comporter en premier argument une variable dont la classe représente un des modes.

Lors de l'appel d'une fonction, cette variable devra être mentionnée.

Par la déclaration des constructeurs dans chacune de ces classes, on définit facilement les changements de modes qui sont autorisés.

Par exemple, la classe correspondante au mode **BOOT** est :

```
class BOOT_mode_class { // PROVISOIRE (voir dans les pages suivantes)
    private : BOOT_mode_class (void) = delete ;
    private : BOOT_mode_class & operator = (const BOOT_mode_class &) = delete ;
    public : BOOT_mode_class (const BOOT_mode_class &) ;
}
```

Par exemple, on déclare une fonction qui doit s'exécuter dans ce mode par :

```
void maFonctionBoot (const BOOT_mode_class MODE) ; // PROVISOIRE (voir pages suivantes)
```

Et cette routine doit être appelée par :

```
maFonctionBoot (MODE) ;
```

C'est-à-dire que l'on ne peut le faire qu'à partir d'une fonction elle-même déclarée dans le mode **BOOT**.

Comment exprimer les modes logiciels (2/4)

Autre exemple : le mode **SECTION** peut être appelé à partir des modes **KERNEL**, **IRQ** et **INIT**.

```
class SECTION_mode_class { // PROVISOIRE (voir dans les pages suivantes)
    private: SECTION_mode_class (void) = delete ;
    private: SECTION_mode_class & operator = (const SECTION_mode_class &) = delete ;

    public: SECTION_mode_class (const SECTION_mode_class &) ;
    public: SECTION_mode_class (const IRQ_mode_class &) ;
    public: SECTION_mode_class (const KERNEL_mode_class &) ;
    public: SECTION_mode_class (const INIT_mode_class &) ;
}
```

Les quatre constructeurs définissent les changements de mode autorisés.

Comment exprimer les modes logiciels (3/4)

Mais : en procédant comme indiqué dans les deux pages précédentes, l'argument de mode apparaît dans le code engendré, ce qui n'est pas souhaitable (augmentation de la taille du code, ralentissement) et même inutile.

Voici la solution qui a été retenue : les annotations relatives au mode **BOOT** sont définies comme suit (voir le fichier **software-modes.h** pour les autres déclarations) :

```
#ifdef CHECK_SOFTWARE_MODES // DÉFINITIF
    #define MODE inSoftwareMode
    #define MODE_ inSoftwareMode,
#else
    #define MODE
    #define MODE_
#endif

#ifndef CHECK_SOFTWARE_MODES
    class BOOT_mode_class {
        private: BOOT_mode_class (void) = delete ;
        private: BOOT_mode_class & operator = (const BOOT_mode_class &) = delete ;
        public: BOOT_mode_class (const BOOT_mode_class &) ;
    } ;
#endif

#ifndef CHECK_SOFTWARE_MODES
    #define BOOT_MODE const BOOT_mode_class MODE
    #define BOOT_MODE_ const BOOT_mode_class MODE,
#else
    #define BOOT_MODE void
    #define BOOT_MODE_
#endif
```

Comment exprimer les modes logiciels (4/3)

La construction d'un projet provoque deux compilations du même source C++ :

- chaque fichier source C++ est d'abord compilé avec l'option **-DCHECK_SOFTWARE_MODES** ; cette option a pour effet de définir la variable de compilation **CHECK_SOFTWARE_MODES**, et par suite d'activer la vérification des modes logiciels ; le code engendré par cette compilation n'est pas utilisé ;
- chaque fichier source C++ est ensuite compilé sans l'option ci-dessus ; la vérification des modes n'est pas activée, et le code engendré est exactement le même que celui que l'on obtiendrait sans ces annotations : c'est ce code qui est utilisé par l'édition de liens.

En construisant un projet avec **1-verbose-build.py**, on peut voir le détail des commandes :

```
[ 41%] Checking start-raspberry-pico.cpp
[ 41%] /Users/pierremolinaro/treel-tools/gcc-arm-none-eabi-7-2017-q4-major/bin/arm-none-eabi-gcc -mthumb -mcpu=cortex-m4
-x c++ -DCHECK_SOFTWARE_MODES -Os -Wall -Wextra -Werror -Wreturn-type -Wformat -Wshadow -Wsign-compare -Wpointer-arith
-Wparentheses -Wcast-align -Wcast-qual -Wwrite-strings -Wswitch -Wswitch-enum -Wuninitialized -Wsign-conversion
-ffunction-sections -fdata-sections -Wno-unused-parameter -Wshadow -std=c++14 -fno-rtti -fno-exceptions -Woverloaded-
virtual -Weffc++ -fno-threadsafe-statics -Wmissing-declarations -Wsuggest-override -c sources/start-raspberry-pico.cpp -o
zBUILDS/start-raspberry-pico.cpp.check.o -DSTATIC= -I zSOURCES -I /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/
solutions/03-software-modes -I sources -MD -MP -MF zBUILDS/start-raspberry-pico.cpp.check.o.dep
.....
[ 58%] Compiling start-raspberry-pico.cpp
[ 58%] /Users/pierremolinaro/treel-tools/gcc-arm-none-eabi-7-2017-q4-major/bin/arm-none-eabi-gcc -mthumb -mcpu=cortex-m4
-Os -Wall -Wextra -Werror -Wreturn-type -Wformat -Wshadow -Wsign-compare -Wpointer-arith -Wparentheses -Wcast-align
-Wcast-qual -Wwrite-strings -Wswitch -Wswitch-enum -Wuninitialized -Wsign-conversion -ffunction-sections -fdata-sections
-Wno-unused-parameter -Wshadow -std=c++14 -fno-rtti -fno-exceptions -Woverloaded-virtual -Weffc++ -fno-threadsafe-statics
-Wmissing-declarations -Wsuggest-override -c sources/start-raspberry-pico.cpp -o zBUILDS/start-raspberry-pico.cpp.o
-DSTATIC= -I zSOURCES -I /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/solutions/03-software-modes -I sources -MD -MP
-MF zBUILDS/start-raspberry-pico.cpp.o.dep
```

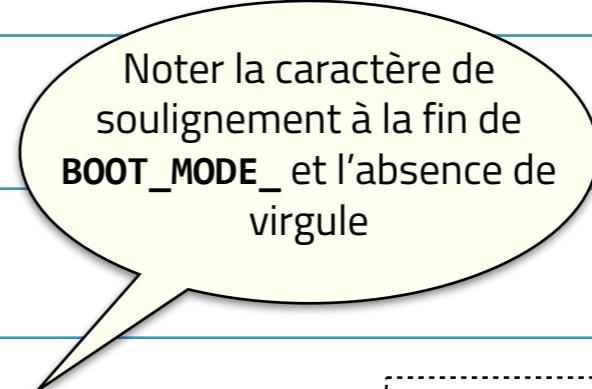
Utilisation pratique des modes logiciels (1/2)

La mise en œuvre pratique des modes logiciels est illustrée avec le mode **BOOT** ; on procèdera de manière analogue avec les autres modes.

En-tête d'une fonction sans argument :

En-tête de la fonction sans annotation de mode	<code>void f (void)</code>	
En-tête de la fonction avec annotation de mode	<code>void f (BOOT_MODE)</code>	Ce qu'il faut écrire dorénavant
En-tête de la fonction, vue par le compilateur lors de la vérification de mode	<code>void f (const BOOT_mode_class inSoftwareMode)</code>	Vous n'avez pas à écrire ceci, c'est une simple illustration du travail du précompilateur
En-tête de la fonction, vue par le compilateur lors de la génération du code	<code>void f (void)</code>	

Et une fonction avec au moins un argument :

En-tête de la fonction sans annotation de mode	<code>void f (argument1)</code>	 Noter la caractère de soulignement à la fin de BOOT_MODE_ et l'absence de virgule
En-tête de la fonction avec annotation de mode	<code>void f (BOOT_MODE_ argument1)</code>	Ce qu'il faut écrire dorénavant
En-tête de la fonction, vue par le compilateur lors de la vérification de mode	<code>void f (const BOOT_mode_class inSoftwareMode, argument1)</code>	Vous n'avez pas à écrire ceci, c'est une simple illustration du travail du précompilateur
En-tête de la fonction, vue par le compilateur lors de la génération du code	<code>void f (argument1)</code>	

Utilisation pratique des modes logiciels (2/2)

Pour l'instruction d'appel de fonction, on utilise l'annotation **MODE** (ou **MODE_** si il y des arguments), et ce quelque soit le mode de la fonction.

Appel d'une fonction sans argument :

Sans annotation de mode	<code>f ()</code>	
Avec annotation de mode	<code>f (MODE)</code>	Ce qu'il faut écrire dorénavant
Appel de la fonction, vu par le compilateur lors de la vérification de mode	<code>void f (inSoftwareMode)</code>	Vous n'avez pas à écrire ceci, c'est une simple illustration du travail du précompilateur
Appel de la fonction, vu par le compilateur lors de la génération du code	<code>f ()</code>	

Et L'appel d'une fonction avec au moins un argument :

Sans annotation de mode	<code>void f (argument1)</code>	Noter la caractère de soulignement à la fin de MODE_ et l'absence de virgule
Avec annotation de mode	<code>void f (MODE_ argument1)</code>	Ce qu'il faut écrire dorénavant
Appel de la fonction, vu par le compilateur lors de la vérification de mode	<code>void f (inSoftwareMode, argument1)</code>	Vous n'avez pas à écrire ceci, c'est une simple illustration du travail du précompilateur
Appel de la fonction, vu par le compilateur lors de la génération du code	<code>void f (argument1)</code>	

Travail à faire

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **03-software-modes**.

Ajoutez aux sources le fichier **software-modes.h**.

Ajouter les annotations de mode dans les fichiers d'en-tête suivants (et les fichiers C++ correspondants) :

- **setup-loop.h**, fonctions **setup0** et **loop0** ;
- **time.h**, fonctions **startSystick** et **busyWaitDuring**.

Par contre, ne pas modifier les prototypes des fonctions **cpu0Phase3Boot** et **cpu0Phase3Init** dans **start-raspberry-pico.h**.

Étape 04 — boot et init routines

Description de cette étape

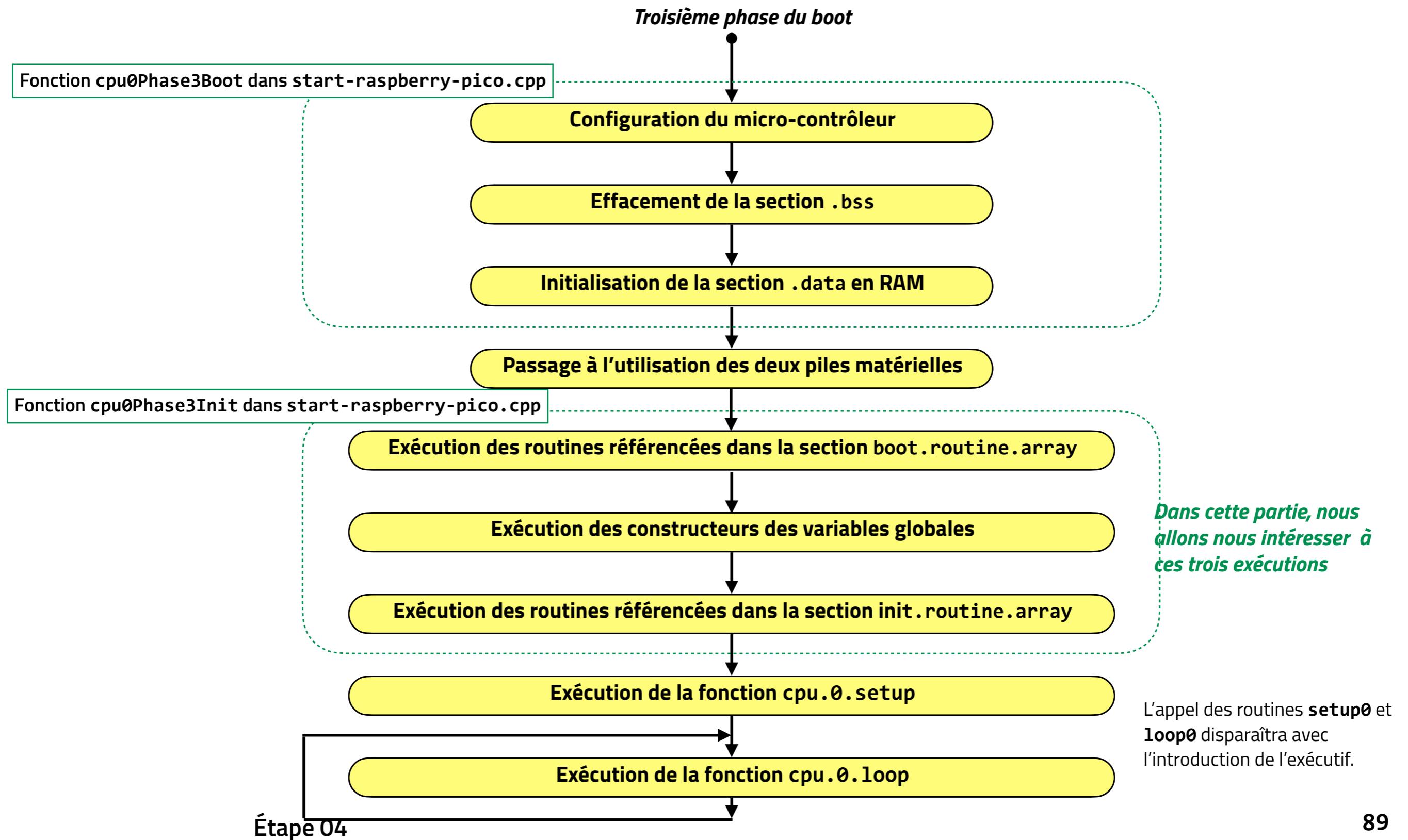
Objectif :

- décrire les différentes routines qui sont exécutées au démarrage ;
- montrer comment inscrire une routine pour exécution implicite au démarrage.

Travail à réaliser : il est décrit à la dernière page.

Organigramme d'exécution du programme

Rappel de l'étape 01



Classification des variables globales

En C++, il existe deux sortes de types :

- les POD (*Plain Old Data*) ou aussi nommés PDS (*Passive Data Structure*), c'est-à-dire issus des types C ;
- les classes C++, pour lesquels un constructeur est (le plus souvent) implémenté.

La différence est qu'une classe C++ définit (le plus souvent) le code d'initialisation de ses instances.

Une variable globale est une variable déclarée en dehors des fonctions.

On obtient trois situations différentes pour une variable globale :

- (1) variable globale PDS non initialisée explicitement ;
- (2) variable globale PDS initialisée explicitement ;
- (3) variable globale instance d'une classe ayant défini un constructeur.

Liens :

https://en.wikipedia.org/wiki/Passive_data_structure

Variable globale PDS non initialisée explicitement

Par exemple :

```
uint32_t gVariable ;
```

Dans le fichier objet, le compilateur place cette variable dans une section **.bss.xyz** (**xyz** étant un nom obtenu à partir du nom C++), ce qui permet à l'éditeur de liens d'inscrire la variable dans la section **.bss** de l'exécutable.

Extrait du script d'édition des liens **dev-files/raspberry-pi-pico-flash.1d** :

```
SECTIONS {  
    .bss : {  
        . = ALIGN(4);  
        __bss_start = . ;  
        * (.bss*) ;  
        . = ALIGN(4);  
        * (COMMON) ;  
        . = ALIGN(4);  
        __bss_end = . ;  
    } > global_vars  
}
```

La section **.bss** de l'exécutable est encadrée par les valeurs des symboles **__bss_start** et **__bss_end**. Ces valeurs sont des multiples de 4. Si la section est vide, alors ces deux symboles ont la même valeur.

L'initialisation de l'image mémoire de la section **.bss** est « *Effacement de la section .bss* », et effectuée à la fin de la routine **cpu0Phase3Boot** (fichier **start-raspberry-pico.cpp**) :

```
extern uint32_t __bss_start ;  
extern const uint32_t __bss_end ;  
uint32_t * p = & __bss_start ;  
while (p != & __bss_end) {  
    * p = 0 ;  
    p ++ ;  
}
```

Dans le script d'édition des liens, les symboles **__bss_start** et **__bss_end** sont des adresses. En C (et C++), les symboles représentent des valeurs. Or ici, nous avons besoin des adresses, d'où l'opérateur **&**.

Variable globale PDS initialisée explicitement

Par exemple :

```
uint32_t gVariable = 14 ;
```

Dans le fichier objet, le compilateur place cette variable et sa valeur initiale dans une section **.data.xyz** (**xyz** étant un nom obtenu à partir du nom C++). Le travail de l'éditeur des liens est un peu plus compliqué, il doit construire deux sections :

- la section **.data** qui contiendra les variables globales (cette section est en RAM) ;
- une section qui contient les valeurs initiales de ces variables (cette section est en Flash).

Extrait du script d'édition des liens **dev-files/raspberry-pi-pico-flash.ld** :

```
SECTIONS {
    .data : AT (__code_end) {
        . = ALIGN (4) ;
        __data_start = . ;
        * (.data*) ;
        . = ALIGN (4) ;
        __data_end = . ;
    } > global_vars
}
__data_load_start = LOADADDR (.data)
__data_load_end   = LOADADDR (.data)
```

Quatre symboles sont ainsi définis :

- **__data_start** : adresse de début de la section **.data** ;
- **__data_end** : adresse de fin de la section **.data** ;
- **__data_load_start** : adresse de début de la section contenant les valeurs initiales (en Flash) ; cette adresse est égale à **__code_end**, définie auparavant dans le script ;
- **__data_load_end** : adresse de fin de la section contenant les valeurs initiales.

La recopie des valeurs initiales est « *Initialisation de la section .data en RAM* », et effectuée à la fin de la routine **cpu0Phase3Boot** (fichier **start-raspberry-pico.cpp**) :

```
extern uint32_t __data_start ;
extern const uint32_t __data_end ;
extern uint32_t __data_load_start ;
uint32_t * pSrc = & __data_load_start ;
uint32_t * pDest = & __data_start ;
while (pDest != & __data_end) {
    * pDest = * pSrc ;
    pDest ++ ;
    pSrc ++ ;
}
```

Le symbole **__data_load_end** n'est pas utilisé.

Variables globales instances de classes

Par exemple :

```
maClasse objet (1, 2, 3) ; // Le constructeur peut présenter des arguments
```

Pour chaque fichier C++, le compilateur engendre une fonction sans argument qui appelle le constructeur de chaque variable globale et place l'adresse de cette fonction dans la section **.init_array**.

Extrait du script d'édition des liens **dev-files/raspberry-pi-pico-flash.1d** :

```
SECTIONS {
    .text : {
.....
    . = ALIGN (4) ;
    __constructor_array_start = . ;
    KEEP (*(.init_array)) ;
    . = ALIGN (4) ;
    __constructor_array_end = . ;
.....
    } > flash
}
```

Ainsi, le symbole **__constructor_array_start** est l'adresse de début d'un tableau dont les éléments sont les adresses des routines d'initialisation. Le symbole **__constructor_array_end** marque la fin de ce tableau.
Un point important est que l'ordre des éléments est directement lié à l'ordre d'apparition des fichiers objets dans la ligne de commande. S'appuyer sur cet ordre est fragile. Autrement dit, un constructeur d'une variable globale ne doit pas supposer qu'il s'exécute avant ou après le constructeur d'une autre variable globale. Par contre, il peut s'appuyer sur des variables globales PDS, car déjà initialisées à ce moment.

```
extern void (* __constructor_array_start) (void) ;
extern void (* __constructor_array_end) (void) ;
ptr = & __constructor_array_start ;
while (ptr != & __constructor_array_end) {
    (* ptr) () ;
    ptr ++ ;
}
```

Routines BOOT et INIT (1/4)

On utilise la même technique pour définir deux types de routines d'initialisation :

- les routines **boot** (dont le mode logiciel est **BOOT_MODE**, qui sont exécutées juste avant les constructeurs des variables globales (« *Exécution des routines référencées dans la section boot.routine.array* ») ;
- les routines **init** (dont le mode logiciel est **INIT_MODE**), qui sont exécutées juste après les constructeurs des variables globales (« *Exécution des routines référencées dans la section init.routine.array* »).

Pour cela, on définit dans le script d'édition des liens **dev-files/raspberry-pi-pico-flash.ld** des tableaux qui regroupent les sections **boot.routine.array** (adresses des fonctions **boot**) et **init.routine.array** (adresses des fonctions **init**) :

```
SECTIONS {
    .text : {
.....
    . = ALIGN (4) ;
    __boot_routine_array_start = . ;
    KEEP (*(boot.routine.array)) ;
    . = ALIGN (4) ;
    __boot_routine_array_end = . ;
.....
    . = ALIGN (4) ;
    __init_routine_array_start = . ;
    KEEP (*(init.routine.array)) ;
    . = ALIGN (4) ;
    __init_routine_array_end = . ;
.....
} > flash
}
```

Ces tableaux sont ensuite exploités pour exécuter les fonctions : « *Exécution des routines référencées dans la section boot.routine.array* » et « *Exécution des routines référencées dans la section init.routine.array* » dans la routine **cpu0Phase3Init** de **start-raspberry-pico.cpp**.

Routines BOOT et INIT (2/4)

Mais comment faire pour inscrire l'adresse d'une fonction dans une section particulière ?

Nous allons illustrer la démarche avec la fonction **startSystick** implémentée dans **time.cpp** :

```
void startSystick (USER_MODE) {  
    //----- Configure Systick  
    systick_hw->rvr = CPU_MHZ * 1000 - 1 ; // Underflow every ms  
    systick_hw->cvr = 0 ;  
    systick_hw->csr = M0PLUS_SYST_CSR_CLKSOURCE_BITS | M0PLUS_SYST_CSR_ENABLE_BITS ;  
}
```

Voici la fonction telle qu'elle a été écrite dans l'étape 03. Elle va être modifiée dans cette étape.

Plutôt que d'appeler cette fonction dans la fonction **setup0** (ce qui était fait dans les étapes précédentes), on va l'inscrire dans la section **boot.routine.array** (plutôt que **init.routine.array**, car ce serait incompatible avec les étapes suivantes).

La fonction doit donc s'exécuter en mode **boot** :

```
void startSystick (BOOT_MODE) {  
    .....  
}
```

Provisoire, il y a encore un détail à modifier (voir page suivante).

Pour inscrire son adresse dans une section particulière, il faut créer une variable initialisée à l'adresse de la fonction et préciser explicitement la section :

```
void (* unNom) (BOOT_MODE) __attribute__ ((section ("boot.routine.array"))) = startSystick ;
```

Routines BOOT et INIT (3/4)

Il reste quelques détails à régler.

D'abord, la fonction **startSystick** n'a pas été déclarée dans **time.h** (il faudra y supprimer sa déclaration), elle n'est référencée que dans le fichier **time.cpp**. On modifie son prototype en :

```
static void startSystick (BOOT_MODE) {  
    .....  
}
```

Définitif.

Ici, le mot réservé **static** signifie que la portée de la fonction est limité au fichier source courant.

Maintenant regardons le référencement de la fonction :

```
void (* unNom) (BOOT_MODE) __attribute__ ((section ("boot.routine.array"))) = startSystick ;
```

La variable **unNom** est globale à tout le projet, il faut choisir à chaque fois un nom unique. En ajoutant **static**, la contrainte d'un nom unique est limitée au fichier courant :

```
static void (* unNom) (BOOT_MODE)  
__attribute__ ((section ("boot.routine.array")))  
= startSystick ;
```

Routines BOOT et INIT, en pratique

Écrire ceci est très laborieux ! Et on n'a pas réglé le problème de l'unicité du nom.

```
static void (* unNom) (BOOT_MODE)
__attribute__ ((section ("boot.routine.array")))
__attribute__ ((unused))
__attribute__ ((used))= startSystick ;
```

On va utiliser une macro qui va simplifier cette écriture : **MACRO_BOOT_ROUTINE**, qui est définie dans **boot-init-macros.h**. Au passage, ce fichier définit aussi des macros construisant implicitement des identificateurs uniques.

Finalement, l'écriture qui sera adoptée est :

```
MACRO_BOOT_ROUTINE (startSystick) ;
```

De même, le fichier d'en-tête **boot-init-macros.h** définit la macro **MACRO_INIT_ROUTINE**.

Travail à faire

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **04-boot-and-init-routines**.

Ajoutez aux sources le fichier **boot-init-macros.h**.

Supprimer de **time.h** la déclaration de la fonction **startSystick**.

Supprimer son appel dans **setup**.

Modifier **time.cpp**, de façon à inscrire la fonction **startSystick** parmi les routines **boot**.

Étape 05

Leds et boutons poussoir

Description de cette étape

Objectif. Accéder aux leds et aux boutons poussoirs de la carte.

Description. L'archive **05-files.tar.bz2** contient deux fichiers : **rp2040-digital-io.h** et **rp2040-digital-io.cpp**. Ces fichiers définissent des fonctions qui permettent de programmer les ports du microcontrôleur en entrée ou en sortie logique, d'écrire et de lire ces ports. Pour ceux qui connaissent, ces fonctions sont analogues à celles de l'Arduino, avec de légères différences.

Les leds d'activité. Ce sont les deux leds placées devant le module TRaspberry Pi Pico. Pour ce cours, on a donné à ces leds un rôle particulier : elle rend compte de l'activité des processeurs. La led ACTIVITÉ 0 sera donc constamment allumée, jusqu'à ce que les attentes passives de l'exécutif soient implémentées. La led ACTIVITÉ 1 sera constamment éteinte, jusqu'à la troisième partie, l'élaboration de l'exécutif dual-core.

La gestion des leds ACTIVITÉ est complètement logicielle, le processeur Cortex-M0+ ne fournissant pas de signal reflétant son activité.

Objectif. Configurer les ports du micro-contrôleur qui sont associés aux leds et aux boutons poussoir.

Description du fichier rp2040-digital-io.h (1/5)

Le type énuméré `DigitalPort` définit les 30 ports du Raspberry Pi Pico :

```
enum class DigitalPort {
    GP0,    GP1,    GP2,    GP3,    GP4,    GP5,    GP6,    GP7,    GP8,    GP9,
    GP10,   GP11,   GP12,   GP13,   GP14,   GP15,   GP16,   GP17,   GP18,   GP19,
    GP20,   GP21,   GP22,   GP23,   GP24,   GP25,   GP26,   GP27,   GP28,   GP29,
    --- No port
    True, // Fictive port: no effect on write, always read true
    False // Fictive port: no effect on write, always read false
};
```

La déclaration `enum class` impose de qualifier les constantes : par exemple, pour se référer au port `GP0`, il faudra écrire `DigitalPort::GP0`.

Description du fichier rp2040-digital-io.h (2/5)

Le type énuméré `DigitalMode` définit les modes possibles d'un port :

```
enum class DigitalMode {  
    OUTPUT,  
    INPUT,  
    INPUT_PULLDOWN,  
    INPUT_PULLUP  
};
```

Mode	Description
<code>DigitalMode::OUTPUT</code>	Le port est en sortie : le micro-contrôleur impose une tension soit proche de 0V (sortie à zéro), soit proche de 3,3 V (sortie à 1).
<code>DigitalMode::INPUT</code>	Le port est en entrée : le port présente un haute impédance, et c'est un circuit extérieur qui impose la tension. Si il est non connecté, sa tension est imprévisible, et donc la valeur retournée par <code>digitalRead</code> l'est aussi.
<code>DigitalMode::INPUT_PULLDOWN</code>	Internement, le port est relié à 0 V à travers une résistance de ?? kΩ environ. Si il est non connecté, sa tension est proche de 3,3 V, et donc la valeur retournée par <code>digitalRead</code> est <code>true</code> .
<code>DigitalMode::INPUT_PULLUP</code>	Internement, le port est relié à 3,3V à travers une résistance de ?? kΩ environ. Si il est non connecté, sa tension est proche de 0 V, et donc la valeur retournée par <code>digitalRead</code> est <code>false</code> .

Description du fichier rp2040-digital-io.h (3/5)

Fonction pinMode. Elle permet de configurer un port :

```
void pinMode (const DigitalPort inPort, const DigitalMode inMode) ;
```

Remarquer qu'il y a pas d'annotation de mode, ce qui signifie que cette fonction peut être appelée dans n'importe quel mode.

Quand on programme un port en sortie (`DigitalMode::OUTPUT`), par défaut le port est placé au niveau bas.

Par exemple, si l'on veut configurer le port GP0 en entrée, on écrira :

```
pinMode (DigitalPort::GP0, DigitalMode::INPUT) ;
```

Description du fichier rp2040-digital-io.h (4/5)

Fonction digitalRead. Elle effectue la lecture d'un port :

```
bool digitalRead (const DigitalPort inPort) ;
```

Cette fonction renvoie :

- **true** si la tension du port est proche de 3,3V ;
- **false** si la tension du port est proche de 0V ;
- une valeur **false** ou **true** imprévisible dans les autres cas.

Noter qu'il est valide de lire un port configuré en sortie.

Description du fichier rp2040-digital-io.h (5/5)

Fonction digitalWrite. Écriture sur un port :

```
void digitalWrite (const DigitalPort inPort, const bool inValue) ;
```

Si le port est configuré en entrée, cette fonction n'a aucun effet.

Si le port est configuré en *sortie* (DigitalMode::OUTPUT) :

- si **inValue** est **false**, le micro-contrôleur impose une tension proche de 0V ;
- si **inValue** est **true**, le micro-contrôleur impose une tension proche de 3,3V.

~~Si le port est configuré en *sortie collecteur ouvert* (DigitalMode::OUTPUT_OPEN_COLLECTOR) :~~

- ~~si **inValue** est **false**, le micro-contrôleur impose une tension proche de 0V ;~~
- ~~si **inValue** est **true**, le micro-contrôleur n'impose aucune tension.~~

Fonction digitalToggle. Complémentation d'un port configuré en sortie :

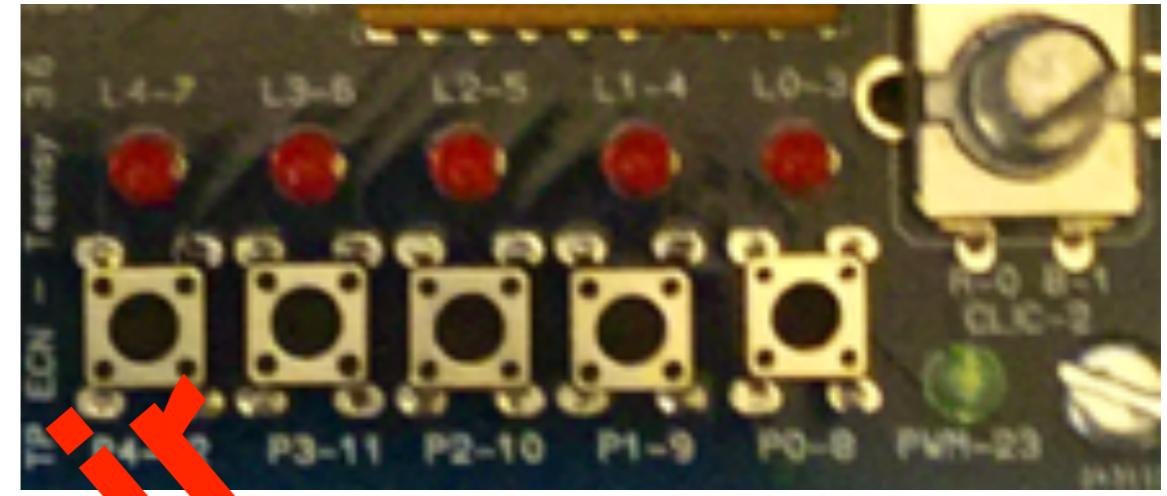
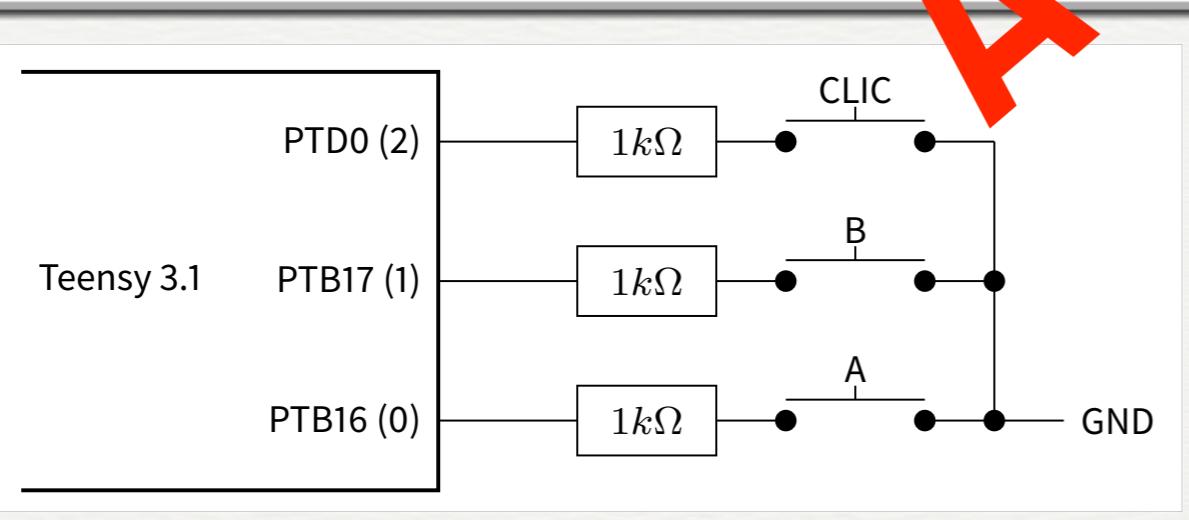
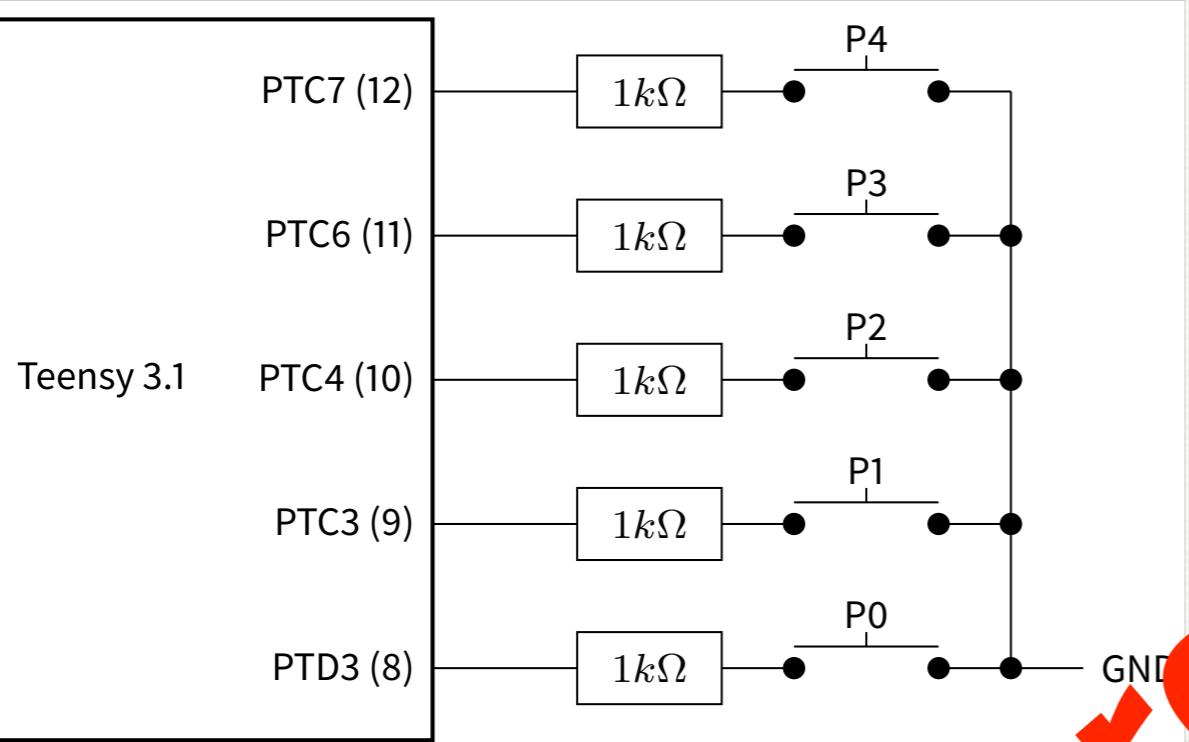
```
void digitalToggle (const DigitalPort inPort) ;
```

Fonctionnellement :

```
digitalToggle (port) ⇔ digitalWrite (port, ! digitalRead (port))
```

L'intérêt est que **digitalToggle** effectue la complémentation de manière atomique.

Poussoirs et encodeur



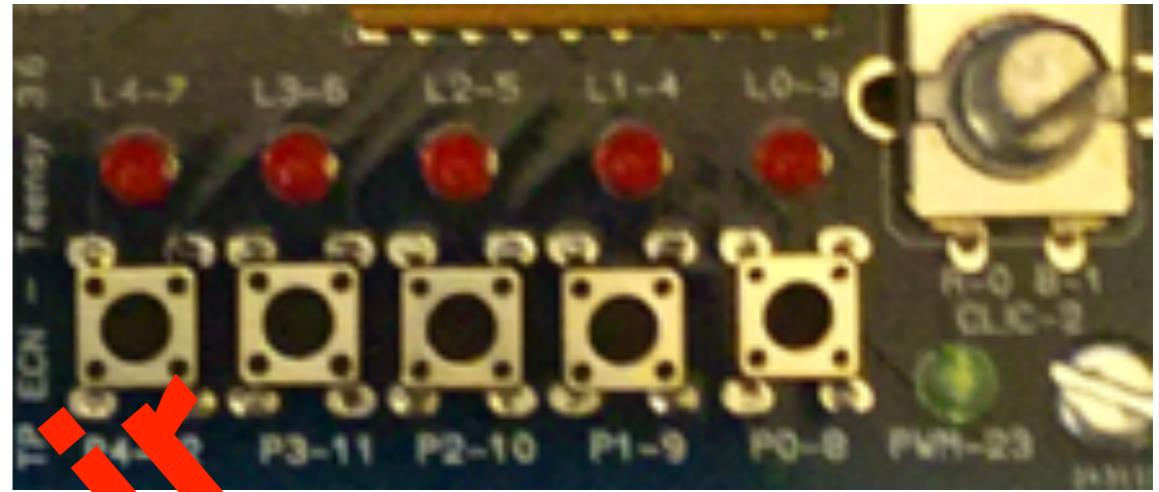
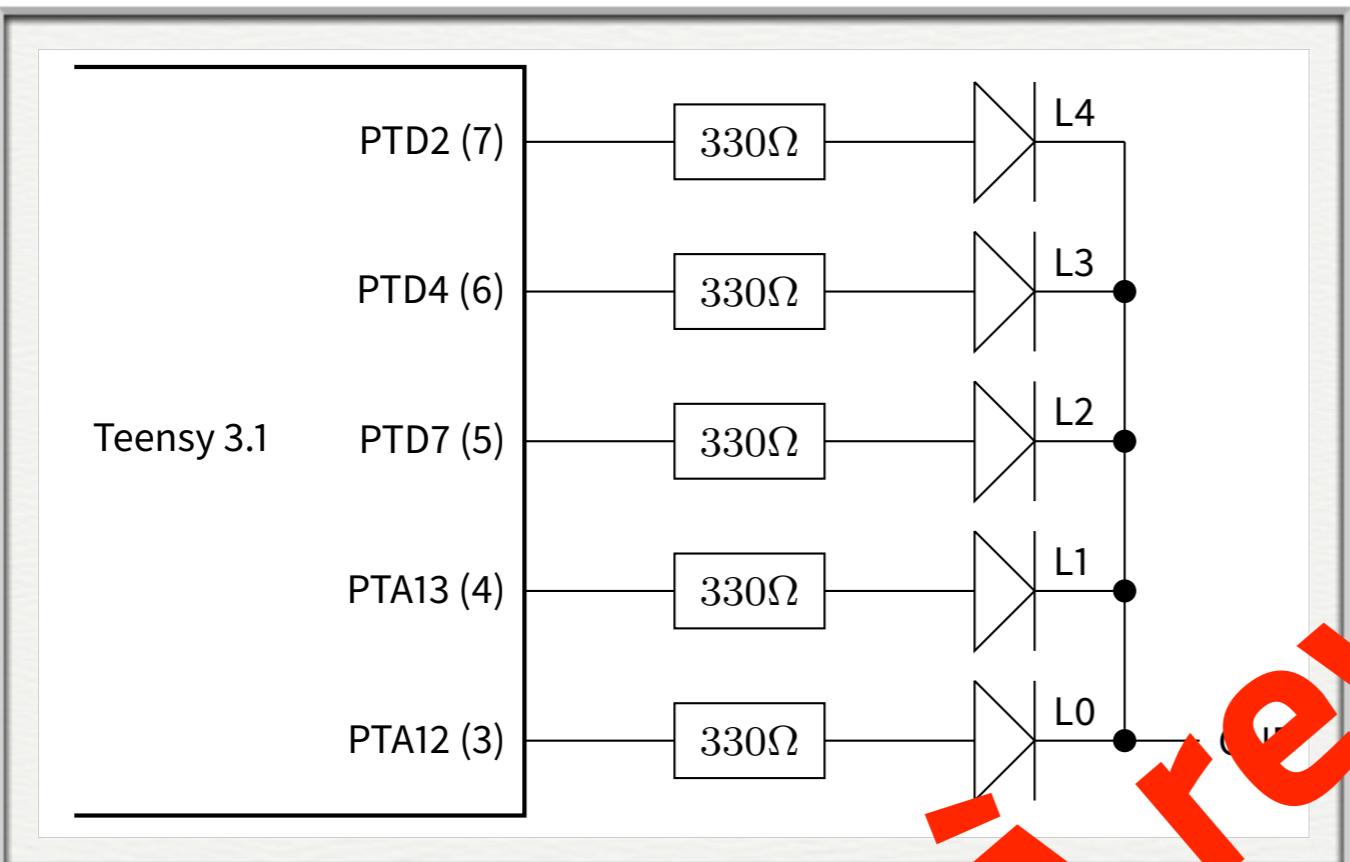
À revoir

Il faut évidemment configurer les ports correspondants en entrée.

Poussoir appuyé : le port correspondant est à une tension de 0V, l'appel de `digitalRead` retourne **false**.

Poussoir relâché : si le port est configuré en `INPUT`, la valeur renvoyée est imprévisible ; si le port est configuré en `INPUT_PULLUP`, l'appel de `digitalRead` retourne **true**.

Leds



À revoir

Il faut configurer les ports correspondants en sortie (OUTPUT).

Appel de digitalWrite avec un argument valant false : le micro-contrôleur impose au port correspondant une tension proche de 0V, la led est éteinte.

Appel de digitalWrite avec un argument valant true : le micro-contrôleur impose au port correspondant une tension proche de 3,3V, la led est allumée.

Travail à faire (1/2)

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **05-leds-pushbuttons**.

Ajoutez aux sources les deux fichiers **rp2040-digital-io.h** et **rp2040-digital-io.cpp** contenus dans l'archive **05-files.tar.bz2**.

Écrire un nouveau fichier **dev-board-io.h** qui définit des constantes désignant les ports associés aux leds et boutons poussoirs. En effet, désigner la led L0 par `DigitalPort::GP4` n'est pas très lisible. On déclarera donc dans ce fichier :

```
static const DigitalPort L0_LED = DigitalPort::GP4 ;
```

Pour la led sur le module Raspberry Pi Pico :

```
static const DigitalPort BUILTIN_LED = DigitalPort::GP25 ;
```

Et de même pour les poussoirs :

```
static const DigitalPort P0_PUSH_BUTTON = DigitalPort::GP13 ;
```

Écrire un nouveau fichier **dev-board-io.cpp** qui contient une fonction en mode **INIT** (voir étape précédente, utiliser **MACRO_INIT_ROUTINE**), qui configure :

- les ports correspondants aux leds et à la led Raspberry Pi Pico en sortie ;
- les ports correspondants aux poussoirs en entrée ;
- le port de la led ACTIVITÉ 0 (GP26) en sortie et au niveau haut (de façon à allumer la led, pour signifier que le processeur 0 est actif) ;
- le port de la led ACTIVITÉ 1 (GP27) en sortie et laisser au niveau bas (de façon à éteindre la led, pour signifier que le processeur 1 est inactif).

Travail à faire (2/2)

Modifier les fonctions **setup0** et **loop0**.

Retirer tous les accès directs aux registres de contrôle (la fonction **setup0** devient vide).

Dans la fonction **loop0**, toutes les 250 ms :

- changer l'état de la led Raspberry Pi Pico avec la fonction **digitalToggle** ;
- utiliser **digitalWrite** ou **digitalRead** pour de façon à ce que :
 - si on appuie sur le poussoir P4, la led L4 soit allumée ;
 - si on relâche le poussoir P4, la led L4 soit éteinte.

À cette étape et dans les suivantes, on n'agira plus jamais dans **setup0** et **loop0** sur les deux leds d'activité : elles sont maintenant considérées comme des leds système, et le code utilisateur (c'est-à-dire celui dans **setup0** et **loop0**) n'a pas à y accéder.

Étape 06 — LCD

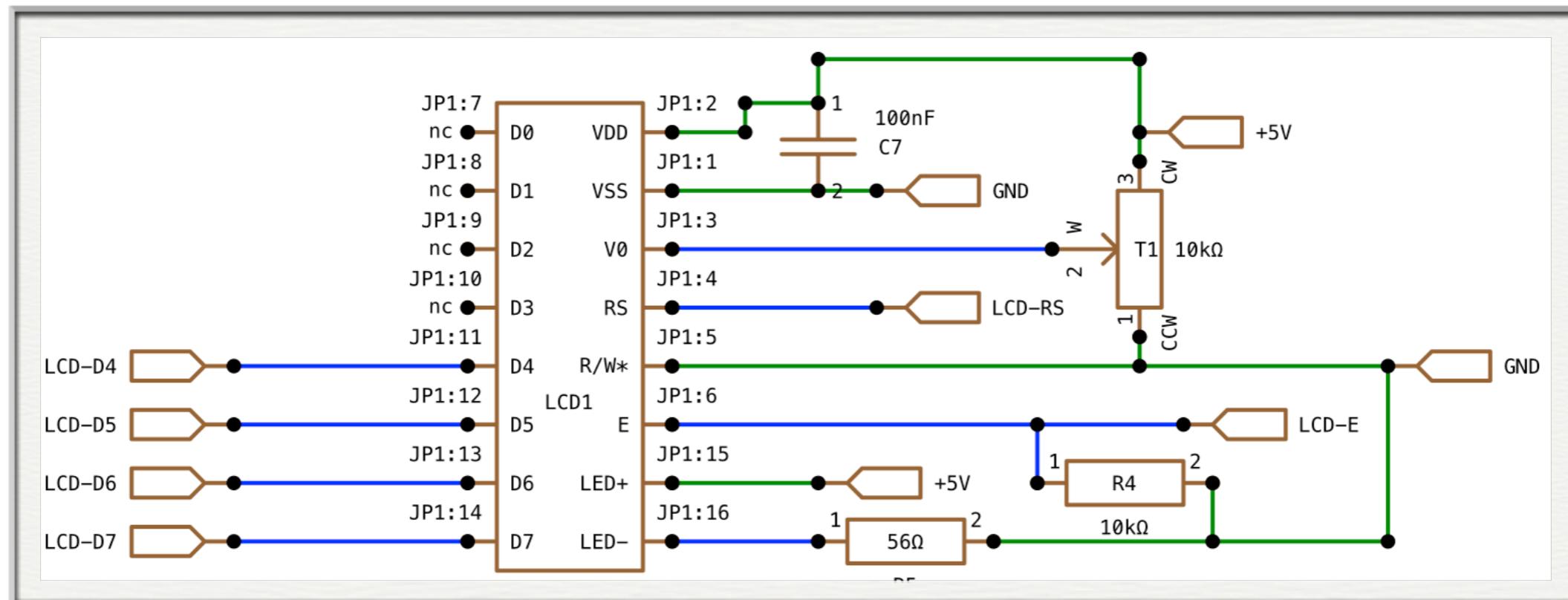
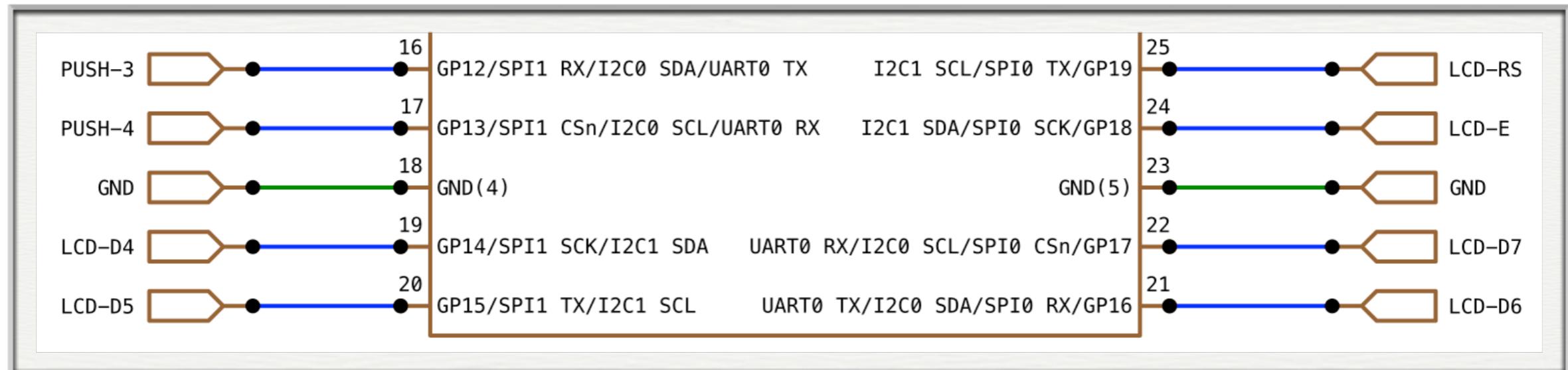
Description de cette partie

Objectif : disposer de fonctions permettant d'afficher sur l'afficheur LCD des chaînes de caractères et des nombres.

Pour cela, l'archive **06-files.tar.bz2** contient les fichiers **lcd-wo-fault-mode.h** et **lcd-wo-fault-mode.cpp**. Ils sont nommés ainsi car ils ne prennent pas en charge le mode logiciel **FAULT** (ce qui sera fait à partir de l'étape 09).

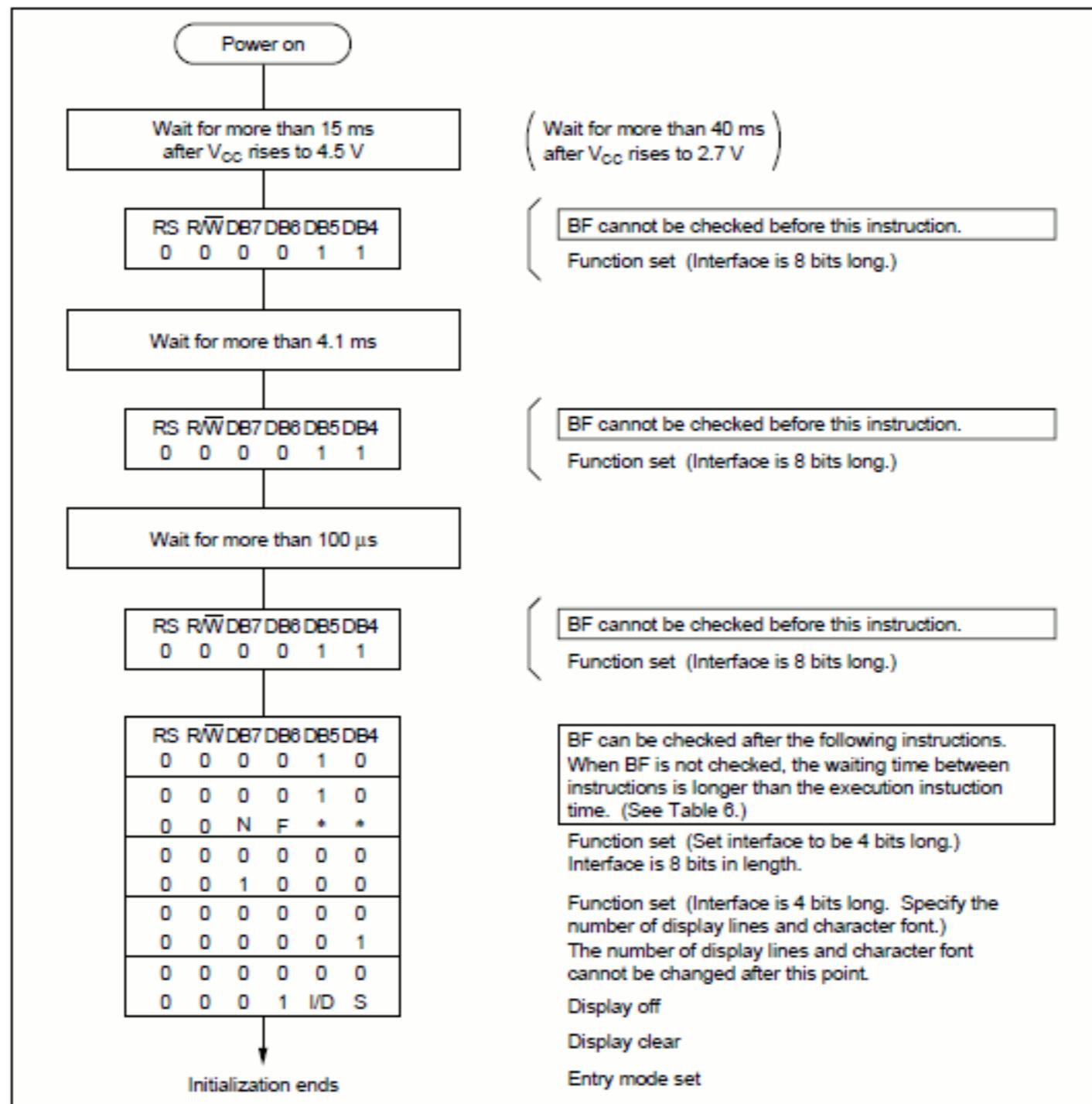
L'objectif est d'utiliser les fonctions d'affichage pour afficher du texte et des nombres.

Connexion de l'afficheur LCD



La connexion est de type « 4 bits ».

Initialisation d'un afficheur LCD en mode 4 bits



http://web.alfredstate.edu/faculty/weimandn/lcd/lcd_initialization/lcd_initialization_index.html

Initialisation : fonction setupLCD

La fonction **setupLCD** programme les ports correspondants aux signaux d'interface, et initialise l'afficheur LCD suivant la procédure illustrée à la page précédente. Elle est automatiquement exécutée lors du démarrage du micro-contrôleur.

```
static void setupLCD (INIT_MODE) {  
    //--- Step 1: configure ports  
    pinMode (LCD_D4, DigitalMode::OUTPUT) ;  
    pinMode (LCD_D5, DigitalMode::OUTPUT) ;  
    pinMode (LCD_D6, DigitalMode::OUTPUT) ;  
    pinMode (LCD_D7, DigitalMode::OUTPUT) ;  
    pinMode (LCD_RS, DigitalMode::OUTPUT) ;  
    pinMode (LCD_E, DigitalMode::OUTPUT) ;  
    //--- Step 2: wait for 15 ms  
    .....  
}  
  
MACRO_INIT_ROUTINE (setupLCD) ;
```

Fonctions d'affichage (1/3)

```
void clearScreen (USER_MODE) ;
```

Efface l'afficheur, en plaçant le curseur au début de la ligne du haut.

```
void gotoXY (USER_MODE_ const uint32_t inColumn, const uint32_t inLine) ;
```

Déplace le curseur sans rien écrire. `inLine` vaut 0 (ligne du haut) à 3 (ligne du bas), `inColumn` vaut 0 (colonne de gauche) à 19 (colonne de droite).

```
void printString (USER_MODE_ const char * inString) ;
```

Imprime la chaîne de caractères à la position du curseur.

```
void printChar (USER_MODE_ const char inChar) ;
```

Imprime le caractère à position du curseur.

```
void printSpaces (USER_MODE_ const uint32_t inCount) ;
```

Imprime `inCount` espaces à position du curseur.

Fonctions d'affichage (2/3)

```
void printUnsigned (USER_MODE_ const uint32_t inValue) ;
```

Imprime le nombre non signé à l'endroit du curseur.

```
void printUnsigned64 (USER_MODE_ const uint64_t inValue) ;
```

Imprime le nombre non signé à l'endroit du curseur.

```
void printSigned (USER_MODE_ const int32_t inValue) ;
```

Imprime le nombre signé à l'endroit du curseur.

Fonctions d'affichage (3/3)

```
void printHex1 (USER_MODE_ const uint32_t inValue) ;
```

Imprime à l'endroit du curseur les 4 bits de poids faible du nombre non signé de 32 bits sous la forme d'un chiffre hexadécimal.

```
void printHex2 (USER_MODE_ const uint32_t inValue) ;
```

Imprime à l'endroit du curseur l'octet de poids faible du nombre non signé de 32 bits sous la forme de deux chiffres hexadécimaux.

```
void printHex4 (USER_MODE_ const uint32_t inValue) ;
```

Imprime à l'endroit du curseur les deux octets de poids faible du nombre non signé de 32 bits sous la forme de quatre chiffres hexadécimaux.

```
void printHex8 (USER_MODE_ const uint32_t inValue) ;
```

Imprime à l'endroit du curseur le nombre non signé de 32 bits sous la forme de 8 chiffres hexadécimaux.

```
void printHex16 (USER_MODE_ const uint64_t inValue) ;
```

Imprime à l'endroit du curseur le nombre non signé de 64 bits sous la forme de 16 chiffres hexadécimaux.

Modifier time.h et time.cpp

Si vous jetez un coup d'œil sur le code contenu dans le fichier **lcd-wo-fault-mode.cpp**, vous verrez qu'il y a deux fonctions d'attente différentes qui sont appelées :

- **busyWaitDuring**, appelée à partir des fonctions en mode **USER** ;
- **busyWaitDuring_initMode**, appelée à partir des fonctions en mode **INIT**.

La première est déjà implémentée dans **time.cpp**, la seconde ne l'est pas.

Donc, dans le fichier **time.h**, ajouter :

```
void busyWaitDuring_initMode (INIT_MODE_ const uint32_t inDelayMS) ;
```

Dans le fichier **time.cpp**, implémenter la fonction :

```
void busyWaitDuring_initMode (INIT_MODE_ const uint32_t inDelayMS) {
    for (uint32_t i=0 ; i<inDelayMS ; i++) {
        // Busy wait, polling COUNTFLAG
        while ((systick_hw->csr & M0PLUS_SYST_CSR_COUNTFLAG_BITS) == 0) {}
    }
}
```

Vous constatez que l'implémentation de cette fonction est identique à celle de **busyWaitDuring** : les modes logiciels peuvent provoquer une duplication du code. De toute façon, la fonction **busyWaitDuring** sera modifiée quand l'attente passive sous exécutif sera implémentée. La fonction **busyWaitDuring_initMode** ne sera pas modifiée, l'exécutif n'étant pas actif durant l'initialisation.

Travail à faire

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **06-1cd**.

Ajoutez aux sources les deux fichiers **lcd-wo-fault-mode.h** et **lcd-wo-fault-mode.cpp** contenus dans l'archive **06-files.tar.bz2**.

Modifier la fonction **setup0** de façon à écrire "Hello!" au début de la première ligne.

Modifier la fonction **loop0** de façon à ce qu'elle exécute un délai de 500 ms, puis affiche au début de la deuxième ligne le nombre de ses exécutions. On peut ajouter la complémentation d'une led.

Dans un deuxième temps, modifier la fonction **loop0** que vous avez écrite de façon à afficher le nombre *modulo* 20 de ses exécutions (il y a un (petit) piège). Note : en C / C++, l'opérateur modulo est « % ».

Question : la fonction **setupLCD** s'exécute en mode INIT. Peut-on l'exécuter en mode BOOT ?

Étape 07 — Interruption Systick

Description de cette étape

Objectif : compter le temps sous interruption.

Ceci permettra de disposer des fonctions :

- **busyWaitUntil**, qui exprime l'attente jusqu'à une date absolue ;
- **busyWaitDuring**, attente durant une délai (cette fonction existe déjà, elle est re-écrite) ;
- **millis**, qui retourne la date courante, c'est-à-dire le nombre de milli-secondes depuis le démarrage du compteur SysTick ;
- **systick_current_cpu**, qui retourne la valeur courante du compteur SysTick.

Fonctions d'attente actuelles

Deux fonctions d'attente ont été implémentées à l'étape 06 (fichier `time.cpp`) :

```
void busyWaitDuring_initMode (INIT_MODE_ const uint32_t inDelayMS) {
    for (uint32_t i=0 ; i<inDelayMS ; i++) {
        // Busy wait, polling COUNTFLAG
        while ((systick_hw->csr & M0PLUS_SYST_CSR_COUNTFLAG_BITS) == 0) {}
    }
}

void busyWaitDuring (USER_MODE_ const uint32_t inDelayMS) {
    for (uint32_t i=0 ; i<inDelayMS ; i++) {
        // Busy wait, polling COUNTFLAG
        while ((systick_hw->csr & M0PLUS_SYST_CSR_COUNTFLAG_BITS) == 0) {}
    }
}
```

Dans cette étape, nous allons modifier le fichier `time.cpp` et la fonction `busyWaitDuring` de façon que le comptage du temps s'effectue sous interruption.

Activer l'interruption du compteur SysTick

Pour activer l'interruption SysTick, il suffit mettre le bit **TICKINT** du registre **SYST_CSR** à 1. Dans le fichier time.cpp, ajouter une fonction exécutée en mode **INIT** :

```
static void activateSystickInterrupt (INIT_MODE) {  
    systick_hw->csr |= M0PLUS_SYST_CSR_TICKINT_BITS ;  
}  
  
MACRO_INIT_ROUTINE (activateSystickInterrupt) ;
```

Maintenant, l'interruption se déclenche toutes les milli-secondes. Mais comment le processeur obtient-il l'adresse de la routine associée à cette interruption ?

La table des vecteurs d'interruption (1/2)

La table des vecteurs d'interruption est dans le fichier **TREEL/dev-files/rp2040-interrupt-vectors.s**:

```
.word __system_stack_end
//--- ARM Core System Handler Vectors
.word reset.handler.cpu.0 // 1
.....
.word interrupt.SysTick // 15
//--- Non-Core Vectors
.word interrupt.TIMER_IRQ_0 // 16, IRQ 0
.....
```

L'interruption SysTick est la n°15, et la routine d'interruption qui lui correspond a le nom assembleur **interrupt.SysTick**.

Le processeur connaît l'adresse de sa table des vecteurs d'interruption grâce au registre **VTOR** (RP2040 datasheet, page 86), dont la valeur par défaut est **0x0**. Cette valeur ne convient pas, cette adresse est dans la bootrom. On va voir page suivante comment on initialise **VTOR**.

La table des vecteurs d'interruption (2/2)

Le script d'éditeur de liens (fichier **TREEL/dev-files/raspberry-pi-pico-flash.ld**) spécifie où est placée cette table :

```
SECTIONS {
    .text : {
        /*----- BOOT2 */
        __boot2_start__ = . ;
        KEEP (*(.boot2)) ;
        __boot2_end__ = . ;
        /*----- Vectors */
        __vectors_start_cpu_0 = . ;
        KEEP (*(isr.vectors.cpu.0)) ;
        __vectors_end_cpu_0 = . ;
        .....
    }
}
```

La table est placée au début de la flash, après les 256 octets de la seconde phase du boot. Un rapide calcul donne son adresse : $0x10000000 + 256 = 0x10000100$. Il suffit donc d'exécuter :

```
scb_hw->vtor = 0x10000100 ;
```

Mais on peut faire mieux : on remarque que le symbole **__vectors_start_cpu_0** défini par le script d'édition de liens pointe le début de la table. L'affectation devient :

```
extern uint32_t __vectors_start_cpu_0 ;
scb_hw->vtor = uint32_t (& __vectors_start_cpu_0) ;
```

Où placer ces instructions ? Évidemment, avant que les interruptions deviennent actives. En fait, vous n'avez pas à les ajouter : la fonction **cpu0Phase3Boot** dans le fichier **start-raspberry-pico.cpp** les contient déjà.

Où est définie la fonction interrupt.SysTick ?

En effet, comme le vecteur d'interruption nomme `interrupt.SysTick`, cette fonction est définie par défaut — sinon on aurait une erreur d'édition de liens.

Il faut regarder dans le fichier engendré par la compilation `zSOURCES/interrupt-handlers.s` :

```
interrupt.SysTick:  
    movs r0, #15  
    b     unused.interrupt
```

Et la fonction `unused.interrupt` est défini comme une boucle sans fin dans `unused-interrupt.s` :

```
unused.interrupt:  
    b     unused.interrupt
```

C'est-à-dire que par défaut, le déclenchement d'une interruption bloque l'exécution.

Note : l'étape 09 modifiera le comportement par défaut au déclenchement d'une interruption : un message d'erreur sera imprimé sur l'afficheur LCD.

La fonction interrupt.SysTick (1/2)

Il faut signifier au système de compilation de ne pas engendrer le code par défaut pour l'interruption SysTick.

Ceci est fait en rajoutant une annotation particulière dans les fichiers d'en-tête. Dans le cas présent, on ajoute la ligne suivante dans **time.h** :

```
//$interrupt-section SysTick
```

Le compilateur ignore cette ligne : c'est un commentaire.

Le système de compilation examine ligne par ligne tous les fichiers d'en-tête à la recherche de type d'annotations (il y aura d'autres qui seront présentées dans les étapes suivantes). Il repère la chaîne `//$interrupt-section` qui doit être suivie par un nom valide d'interruption.

Si on recompile le programme avec l'annotation, on obtient une erreur d'édition de liens : le symbole **interrupt.section.SysTick** est indéfini. En effet, la fonction **interrupt.SysTick** dans zSOURCES/interrupt-handlers.s est devenue :

```
interrupt.SysTick:  
    ldr    r0, =0xD0000000 + 0x014 // Address of SIO_GPIO_OUT_SET control register  
    ldr    r1, = (1 << 26)      // Port GP26 is ACTIVITY 0  
    str    r1, [r0]              // turn on  
    b     interrupt.section.SysTick
```

Nous commentons ce code page suivante.

La fonction interrupt.SysTick (2/2)

Commentaires sur ce code :

```
interrupt.SysTick:  
    ldr    r0, =0xD0000000 + 0x014 // Address of SIO_GPIO_OUT_SET control register  
    ldr    r1, = (1 << 26) // Port GP26 is ACTIVITY 0  
    str    r1, [r0]           // turn on  
    b     interrupt.section.SysTick
```

Les trois premières instructions allument la led ACTIVITÉ 0, pour montrer l'activité processeur. Notons qu'actuellement ce code est inutile (la led est en permanence allumée), mais sera utile en présence de l'exécutif.

La dernière ligne effectue un branchement vers la fonction nommée `interrupt.section.SysTick` : il faut donc définir cette fonction. Nous allons le faire page suivante.

Définir la fonction interrupt.section.SysTick

Déclarer la fonction dans **time.h** :

```
void systickInterruptServiceRoutine (SECTION_MODE) asm ("interrupt.section.SysTick") ;
```

Le nom C++ est libre, mais la fonction doit être déclarée dans le mode SECTION. Par contre, le nom assembleur est imposé, d'après la page précédente.

Implémenter la fonction dans **time.cpp** :

```
static volatile uint32_t gUptime ;  
  
void systickInterruptServiceRoutine (SECTION_MODE) {  
    gUptime += 1 ;  
}
```

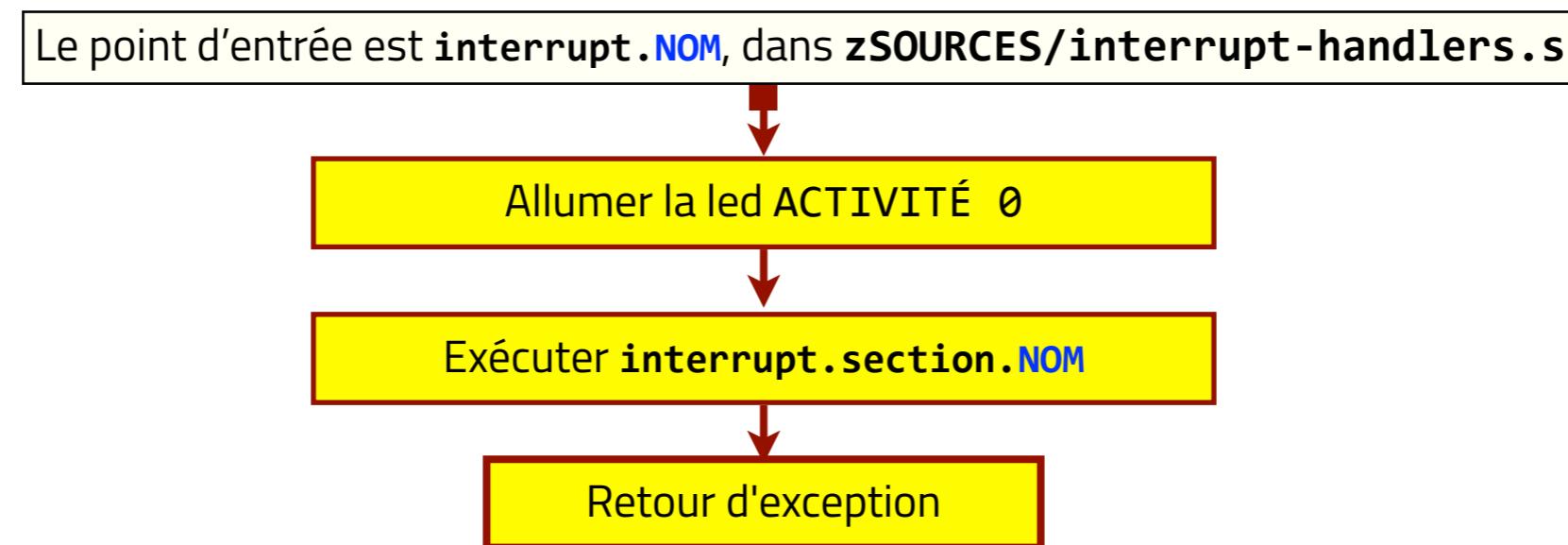
Ainsi, la variable `gUptime` est incrémentée toutes les milli-secondes : elle contient la date courante. Remarques sur sa déclaration :

- **static** limite la visibilité de la variable au fichier courant (ici **time.cpp**) ;
- **volatile** est **OBLIGATOIRE** car la variable va être partagée entre une routine d'interruption et des routines en mode USER (dont `busyWaitUntil`, voir page suivante).

Organigramme d'une routine d'interruption, mode SECTION

En résumé, une routine d'interruption en mode **SECTION** est déclarée dans un fichier d'en-tête par :

```
//$interrupt-section NOM  
void routine (SECTION_MODE) asm ("interrupt.section.NOM") ;
```



NOM est le nom de l'interruption. Le fichier Python `TREEL/dev-files/interrupt_names_teensy_3_6.py` contient le nom de toutes les interruptions du micro-contrôleur. On retrouve ces noms dans `sources/rp2040-interrupt-vectors.s`.

Fonctions busyWaitUntil et busyWaitDuring

Deux fonctions de gestion du temps doivent être écrites :

- **busyWaitUntil**, qui exprime l'attente jusqu'à une date absolue ;
- **busyWaitDuring**, attente durant une délai (cette fonction existe déjà, elle est re-écrite).

La fonction **busyWaitUntil** est :

```
void busyWaitUntil (USER_MODE_ const uint32_t inDeadlineMS) {  
    while (inDeadlineMS > gUptime) {}  
}
```

Modifier l'implémentation de la fonction **busyWaitDuring** : il suffit d'appeler **busyWaitUntil** (avec les bons arguments...)

Fonctions `millis` et `systick_current_cpu`

Écrire une fonction `millis` qui retourne la date courante, c'est-à-dire le nombre de milli-secondes depuis le démarrage du compteur SysTick. Cette fonction peut être appelée dans n'importe quel mode (annotation de mode : `ANY_MODE`).

Écrire une fonction `systick_current_cpu` qui retourne la valeur du registre `SYST_CVR`, qui contient la valeur courante du compteur SysTick. Cette fonction peut être appelée dans n'importe quel mode (annotation de mode : `ANY_MODE`).

Travail à faire

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **07-systick-interrupt**.

Implémenter les fonctions citées dans les pages précédentes (fichiers **time.h** et **time.cpp**).

Modifier la fonction **setup0** de façon à écrire la valeur retournée par **millis** au début de la première ligne.

Modifier la fonction **loop0** de façon à ce qu'elle attende durant 1000 ms, puis affiche au début de la deuxième ligne la valeur retournée par **millis**. Constater que la période d'exécution de **loop0** n'est pas 1000 ms.

Ensuite, modifier la fonction **loop0** pour obtenir une période d'exécution d'exactement 1000 ms.

Étape 08

volatile et interruption Systick

Description de cette partie

Objectif : cette étape a un double objectif :

- comment inscrire des fonctions pour qu'elles soient exécutées à chaque occurrence de l'interruption **SysTick** ;
- montrer pourquoi les variables partagées doivent être déclarées avec le qualificatif **volatile**.

Exécution d'une fonction par l'interruption SysTick (1/4)

La routine exécutée lors du déclenchement de l'interruption SysTick a été écrite lors de l'étape précédente (fichier **time.cpp**) :

```
void systickInterruptServiceRoutine (SECTION_MODE) {  
    gUptime += 1 ;  
}
```

Dans cette étape, nous voulons pouvoir disposer d'un mécanisme similaire à **MACRO_BOOT_ROUTINE** et **MACRO_INIT_ROUTINE**, de façon que l'on puisse facilement inscrire une fonction pour qu'elle soit exécutée à chaque occurrence de l'interruption SysTick.

Exécution d'une fonction par l'interruption SysTick (2/4)

Le script de l'édition des liens **dev-files/raspberry-pi-pico-flash.ld** rassemble toutes les sections **real.time.interrupt.routine.array**, à l'image de ce qui a été fait pour les sections **boot.routine.array** et **init.routine.array** :

```
SECTIONS {
    .text : {
        .....
        . = ALIGN (4) ;
        _real_time_interrupt_routine_array_start = . ;
        KEEP (*(real.time.interrupt.routine.array)) ;
        . = ALIGN (4) ;
        _real_time_interrupt_routine_array_end = . ;
        .....
    } > flash
}
```

Dans cette étape, nous voulons pouvoir disposer d'un mécanisme similaire à **MACRO_BOOT_ROUTINE** et **MACRO_INIT_ROUTINE**, de façon que l'on puisse facilement inscrire une fonction pour qu'elle soit exécutée à chaque occurrence de l'interruption SysTick.

Exécution d'une fonction par l'interruption SysTick (3/4)

Pour exploiter les sections `real.time.interrupt.routine.array`, on ajoute dans le fichier `time.h` la définition de la macro `MACRO_REAL_TIME_ISR`, similaire à `MACRO_BOOT_ROUTINE` et `MACRO_INIT_ROUTINE` :

```
#define MACRO_REAL_TIME_ISR(ROUTINE) \
    static void (* UNIQUE_IDENTIFIER) (SECTION_MODE_ const uint32_t inUptime) \
    __attribute__ ((section ("real.time.interrupt.routine.array"))) \
    __attribute__ ((unused)) \
    __attribute__ ((used)) = ROUTINE ;
```

Et il faut ajouter à la fonction `systickInterruptServiceRoutine` le parcours du tableau débutant à `_real_time_interrupt_routine_array_start` et l'exécution des fonctions qu'il contient :

```
void systickInterruptServiceRoutine (SECTION_MODE) {
    const uint32_t newUptime = gUptime + 1 ;
    gUptime = newUptime ;
//--- Run real.time.interrupt.routine.array section routines
    extern void (* _real_time_interrupt_routine_array_start) (SECTION_MODE_ const uint32_t inUptime) ;
    extern void (* _real_time_interrupt_routine_array_end) (SECTION_MODE_ const uint32_t inUptime) ;
    void (* * ptr) (SECTION_MODE_ const uint32_t) = & _real_time_interrupt_routine_array_start ;
    while (ptr != & _real_time_interrupt_routine_array_end) {
        (* ptr) (MODE_ newUptime) ;
        ptr ++ ;
    }
}
```

Exécution d'une fonction par l'interruption SysTick (4/4)

Voici un exemple d'utilisation de la macro MACRO_REAL_TIME_ISR :

```
static void rtISR (SECTION_MODE_ const uint32_t inUptime) {  
    .....  
}  
  
MACRO_REAL_TIME_ISR (rtISR) ;
```

Remarquer que la fonction présente l'argument `inUptime`, qui contient la valeur de l'instant courant.

Règle d'utilisation du qualificatif volatile

Toute variable globale partagée entre une routine d'interruption et une routine de la tâche de fond (fonctions dans le mode USER) doit être déclarée volatile.

Si cette règle n'est pas respectée, le programme pourra peut-être fonctionner correctement (ou pas).

Dans la suite de cette étape, on propose un exemple de programme qui illustre cette règle.

Le programme d'exemple (1/3)

Remplacer le fichier **setup-loop.cpp** par celui-ci (dans l'archive **08-files.tar.bz2**) :

```
#include "all-headers.h"

static uint32_t gCount ; // Volontairement, volatile est absent, c'est un bug

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
    printString (MODE_ "Hello!") ;
}

void loop0 (USER_MODE) {
    gCount += 500 ;
    gotoXY (MODE_ 0, 1) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Volontairement, le programme ci-dessus est bogué : la variable globale **gCount** est partagée entre la routine d'interruption **rtISR** et la routine de fond **loop0**, et n'a pas été déclarée **volatile**.

Exécutez le programme, et constatez qu'il fonctionne correctement, malgré l'oubli de **volatile**.

Le programme d'exemple (2/3)

Modifier maintenant la fonction `setup0` en ajoutant la ligne écrite en bleu :

```
#include "all-headers.h"

static uint32_t gCount ; // Volontairement, volatile est absent

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}

void loop0 (USER_MODE) {
    gCount += 500 ;
    gotoXY (MODE_ 0, 1) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Exécutez le programme, et constatez qu'il ne fonctionne plus correctement : en fait, l'exécution est bloquée sur la ligne qui a été ajoutée.

Le programme d'exemple (3/3)

Ajouter le qualificatif **volatile** à la déclaration de **gCount** :

```
#include "all-headers.h"

static volatile uint32_t gCount ; // Correct !

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}

void loop0 (USER_MODE) {
    gCount += 500 ;
    gotoXY (MODE_ 0, 1) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Exécutez le programme, et constatez qu'il fonctionne correctement.

Discussion : code engendré en présence de volatile

Le cœur du problème est le code engendré par la compilation de la fonction **setup**.

```
static volatile uint32_t gCount ; // Correct !
.....
void setup0 (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}
```

Le code engendré est (appeler **1-build-as.py** pour l'obtenir), les commentaires en vert ont été ajoutés :

```
cpu.0.setup:
    push {r4, lr} @ Sauvegarde de R4 et de LR dans la pile
    ldr r0, .L7          @ R4 <- adresse de la chaîne "Hello!"
    bl _Z11printStringPKc @ Appel de la fonction printString
    ldr r1, .L7+4 @ R1 <- Adresse de gCount
    ldr r3, .L7+8 @ R3 <- 2999

.L5:
    ldr r2, [r1] @ R2 <- valeur de gCount (lecture mémoire)
    cmp r2, r3 @ Comparaison entre R2 et R3
    bls .L5      @ Branch unsigned Lower or Same : if (r2 <= r3) goto .L5
    pop {r4, pc} @ Restauration de R4 et retour
```

Discussion : code engendré sans volatile

Maintenant, en enlevant le qualificatif **volatile** :

```
static uint32_t gCount ; // Bug
.....
void setup0 (USER_MODE) {
    printString (MODE_ "Hello!" ) ;
    while (gCount < 3000) {}
}
```

Le code engendré est (appeler **1-build-as.py** pour l'obtenir), les commentaires en vert ont été ajoutés :

```
cpu.0.setup:
    push {r4, lr} @ Sauvegarde de R4 et de LR dans la pile
    ldr r0, .L7          @ R0 <- adresse de la chaîne "Hello!"
    bl _Z11printStringPKc @ Appel de la fonction printString
    ldr r3, .L7+4 @ R3 <- Adresse de gCount
    ldr r2, [r3] @ R2 <- valeur de gCount (lecture mémoire)
    ldr r3, .L7+8 @ R3 <- 2999

.L5:
    cmp r2, r3 @ Comparaison entre R2 et R3
    bls .L5      @ Branch unsigned Lower or Same : if (r2 <= r3) goto .L5
    pop {r4, pc} @ Restauration de R3 et retour
```

Dans la boucle **.L5**, la valeur de **gCount** n'est pas rechargée par une lecture mémoire : la variable est lue une fois avant la boucle, et si le test est vrai le bouclage est infini.

La nécessité de volatile

Toute variable globale partagée entre une routine d'interruption et une routine de la tâche de fond (fonctions dans le mode USER) doit être déclarée volatile.

Voir le document suivant (page 42) :

https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

Et la video correspondante :

<https://www.youtube.com/watch?v=NCTf7wT5WR0>

Étape 09 — Section critique

Description de cette partie

Malheureusement, le qualificatif **volatile**, s'il est nécessaire quand on déclare une variable partagée, n'est pas suffisant dans bien des situations.

Objectif :

- cette étape va illustrer un comportement pour lequel le qualificatif **volatile** n'est pas suffisant ;
- et va introduire la notion de *section critique*, et comment les spécifier dans les sources.

Le programme d'exemple

Remplacer le fichier **setup-loop.cpp** par celui-ci (dans l'archive **09-files.tar.bz2**) :

```
#include "all-headers.h" Les variables partagées sont bien déclarées avec le qualificatif volatile.
static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                    const uint32_t inUptime) {
    if (gPerformCount) {
        gCount1 += 1 ;
        gCount2 += 1 ;
        gCount3 += 1 ;
        gCount4 += 1 ;
    }
}
MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
}

void loop0 (USER_MODE) {
    if (gPerformCount) {
        gCount0 += 1 ;
        gCount1 += 1 ;
        gCount2 += 1 ;
        gCount3 += 1 ;
        if (millis (MODE) >= 5000) {
            gPerformCount = false ;
            gotoXY (MODE_ 0, 0) ;
            printUnsigned (MODE_ gCount0) ;
            gotoXY (MODE_ 0, 1) ;
            printUnsigned (MODE_ gCount1) ;
            gotoXY (MODE_ 0, 2) ;
            printUnsigned (MODE_ gCount2) ;
            gotoXY (MODE_ 0, 3) ;
            printUnsigned (MODE_ gCount3) ;
        }
    }
}
```

Exécution du programme d'exemple

Exécutez le programme d'exemple, et examinez les valeurs affichées. Essayer plusieurs fois, en changeant la fréquence du processeur (paramètre **CPU-MHZ** dans le fichier **makefile.json**).

Voici un affichage obtenu (fréquence processeur : 125 MHz) :



Pourquoi les valeurs sont-elles différentes ?

Le programme d'exemple

L'incrémentation sous interruption est commentée

```
#include "all-headers.h"

static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                    const uint32_t inUptime) {
    if (gPerformCount) {
        // gCount1 += 1 ;
        // gCount2 += 1 ;
        // gCount3 += 1 ;
        // gCount4 += 1 ;
    }
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
```

```
void loop0 (USER_MODE) {
    if (gPerformCount) {
        gCount0 += 1 ;
        gCount1 += 1 ;
        gCount2 += 1 ;
        gCount3 += 1 ;
        if (millis (MODE) >= 5000) {
            gPerformCount = false ;
            gotoXY (MODE_ 0, 0) ;
            printUnsigned (MODE_ gCount0) ;
            gotoXY (MODE_ 0, 1) ;
            printUnsigned (MODE_ gCount1) ;
            gotoXY (MODE_ 0, 2) ;
            printUnsigned (MODE_ gCount2) ;
            gotoXY (MODE_ 0, 3) ;
            printUnsigned (MODE_ gCount3) ;
        }
    }
}
```



Le programme d'exemple

L'incrémentation dans la tâche de fond est commentée

```
#include "all-headers.h"

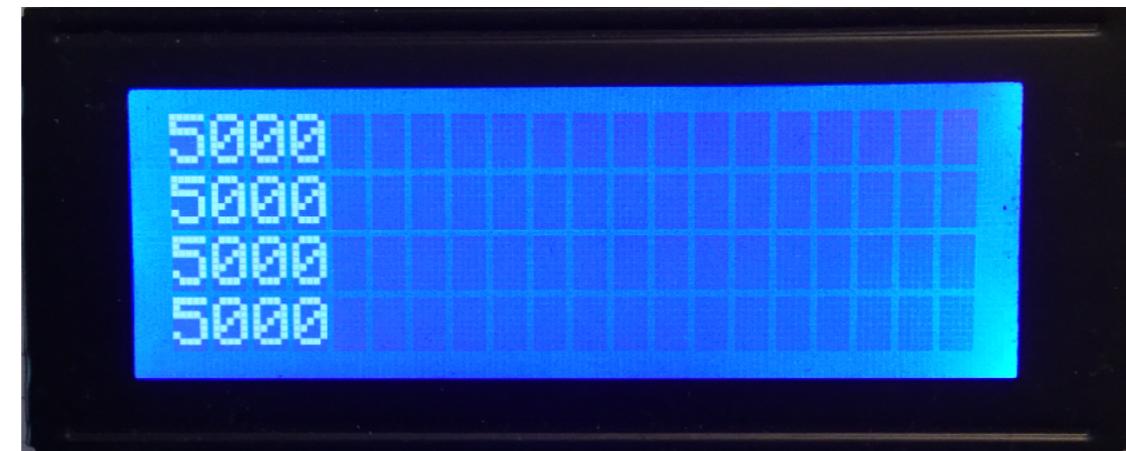
static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                    const uint32_t inUptime) {
    if (gPerformCount) {
        gCount1 += 1 ;
        gCount2 += 1 ;
        gCount3 += 1 ;
        gCount4 += 1 ;
    }
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup0 (USER_MODE) {
```

```
void loop0 (USER_MODE) {
    if (gPerformCount) {
        // gCount0 += 1 ;
        // gCount1 += 1 ;
        // gCount2 += 1 ;
        // gCount3 += 1 ;
        if (millis (MODE) >= 5000) {
            gPerformCount = false ;
            gotoXY (MODE_ 0, 0) ;
            printUnsigned (MODE_ gCount0) ;
            gotoXY (MODE_ 0, 1) ;
            printUnsigned (MODE_ gCount1) ;
            gotoXY (MODE_ 0, 2) ;
            printUnsigned (MODE_ gCount2) ;
            gotoXY (MODE_ 0, 3) ;
            printUnsigned (MODE_ gCount3) ;
        }
    }
}
```



Pourquoi les valeurs sont-elles différentes ? (1/2)

Le processeur embarqué dans le micro-contrôleur est un processeur Cortex-M4. Or ce processeur ne peut pas effectuer l'incrémentation d'une variable en mémoire en une seule instruction :

Code C

```
gCount1 += 1 ;
```

Code Assembleur ARM engendré

```
ldr r3, ..... @ R3 <- adresse de gCount1  
ldr r2, [r3]   @ R2 <- valeur de Count1 (lecture mémoire)  
add r2, r2, #1 @ R2 <- R2 + 1  
str r2, [r3]   @ Écriture de R2 en mémoire
```

Le code assembleur ARM montre que l'incrémentation d'une variable est réalisée par plusieurs instructions successives. Une interruption ne peut pas interrompre l'exécution d'une instruction assembleur, mais elle peut interrompre la séquence d'exécution.

```
ldr r2, [r3]  
add r2, r2, #1  
str r2, [r3]
```

←
←
←
←
Positions possibles de la prise en compte de l'interruption.

Pourquoi les valeurs sont-elles différentes ? (2/2)

Pour comprendre l'origine du piège, on va abstraire le code assembleur relatif à l'incrémentation d'une variable :

Dans la routine `loop0`

<code>y := gCount1</code>
<code>y ++</code>
<code>gCount1 := y</code>

Dans la routine `rtISR`

<code>x := gCount1</code>
<code>x ++</code>
<code>gCount1 := x</code>

Comme l'interruption peut être prise en compte à tout instant dans la routine `loop0`, on peut avoir par exemple les scénarios suivants :

Un scénario

<code>y := gCount1</code>
<code>y ++</code>
<code>gCount1 := y</code>
<code>x := gCount1</code>
<code>x ++</code>
<code>gCount1 := x</code>

La variable est correctement incrémentée de 2 unités.

Un autre scénario

<code>y := gCount1</code>
<code>y ++</code>
<code>x := gCount1</code>
<code>x ++</code>
<code>gCount1 := x</code>
<code>gCount1 := y</code>

La variable est incrémentée d'une seule unité.

Une possibilité de correction : `__atomic_fetch_add`

GCC pour Cortex-M possède des *fonctions intrinsèques* (intrinsics) qui engendre un code assurant que l'incrémentation se réalise de façon atomique.

Mais cette solution n'est pas applicable pour le RP2040 : ses processeurs Cortex M0+ ne contiennent pas les instructions nécessaires à la réalisation de la fonction intrinsèque `__atomic_fetch_add` : essayer de compiler le code suivant conduit à des erreurs d'édition de liens.

```
void loop0 (USER_MODE) {
    if (gPerformCount) {
        __atomic_fetch_add (& gCount0, 1, __ATOMIC_ACQ_REL) ;
        __atomic_fetch_add (& gCount1, 1, __ATOMIC_ACQ_REL) ;
        __atomic_fetch_add (& gCount2, 1, __ATOMIC_ACQ_REL) ;
        __atomic_fetch_add (& gCount3, 1, __ATOMIC_ACQ_REL) ;
    if (5000 <= millis (MODE)) {
        gPerformCount = false ;
        gotoXY (MODE_ 0, 0) ;
        printUnsigned (MODE_ gCount0) ;
        gotoXY (MODE_ 0, 1) ;
        printUnsigned (MODE_ gCount1) ;
        gotoXY (MODE_ 0, 2) ;
        printUnsigned (MODE_ gCount2) ;
        gotoXY (MODE_ 0, 3) ;
        printUnsigned (MODE_ gCount3) ;
    }
}
}
```

Sur un processeur Cortex-M3 ou mieux, on vérifie que les quatre compteurs ont alors toujours la même valeur.

Liens :

https://en.wikipedia.org/wiki/Intrinsic_function

https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

Seconde possibilité de correction : section critique

Une autre possibilité est rendre ininterruptible la séquence des incrémentations (ou chacune des incrémentations). Pour cela, on va définir des *sections critiques*.

Le système de compilation permet de choisir un schéma de section critique parmi plusieurs possibles. Le choix s'effectue dans le fichier `makefile.json`, avec la clé **SECTION-SCHEME** :

- "SECTION-SCHEME" : "disableInterrupt" : les interruptions sont masquées dans la section critique [**C'EST LE SCHÉMA À UTILISER**] ;
- "SECTION-SCHEME" : "swint" : (*software interrupt*) ce schéma utilise l'interruption n°80 pour exécuter la section critique dans le mode *handler* du processeur ;
- "SECTION-SCHEME" : "bkpt" : (*breakpoint*) ce schéma utilise l'instruction bkpt pour exécuter la section critique dans le mode *handler* du processeur [**NE PAS UTILISER, NE FONCTIONNE PAS**].

Comment spécifier une section critique

Pour spécifier une section critique, il faut :

- déclarer le prototype de la fonction appelée (en mode **USER**) ;
- implémenter la fonction qui exécute la section critique (en mode **SECTION**) ;
- déclarer la section critique au système de compilation par une annotation particulière ;
- quand on a une ou plusieurs sections critiques, préciser un schéma dans le fichier **makefile.json**, avec la clé **SECTION-SCHEME**.

Comment spécifier une section critique : en pratique

La fonction qui exécute la section critique en mode SECTION (dans **setup-loop.cpp**) :

```
void section_increments (SECTION_MODE) {  
    gCount0 += 1 ;  
    gCount1 += 1 ;  
    gCount2 += 1 ;  
    gCount3 += 1 ;  
}
```

Dans le fichier **setup-loop.h**, on déclare :

- le prototype de la fonction appelée (en mode **USER**) ;
- le prototype de la fonction qui exécute la section critique (en mode **SECTION**) ;
- la section critique au système de compilation par une annotation particulière.

```
//$section fonction.increments  
  
void increments (USER_MODE) asm ("fonction.increments") ;  
  
void section_increments (SECTION_MODE) asm ("section.fonction.increments") ;
```

Dans la fonction **loop0** (fichier **setup-loop.cpp**) on appelle la fonction **increments** :

```
void loop0 (USER_MODE) {  
    if (gPerformCount) {  
        increments (MODE) ;  
        if (5000 <= millis (MODE)) {  
            .....  
        }  
    }  
}
```

Code engendré par l'annotation //\$/section

L'annotation `//$/section` engendre le code dans `zSOURCES/interrupt-handlers.s`. Voici le code engendré quand le schéma choisi est `disableInterrupt`:

`fonction.incrementations:`

```
.fnstart
@--- Save preserved registers
    push {r6, lr}
@--- Save interrupt enabled state
    mrs r6, PRIMASK
@--- Disable interrupt
    cpsid i
@--- Call section, interrupts disabled
    bl  section.fonction.incrementations
@--- Restore interrupt state
    msr PRIMASK, r6
@--- Restore preserved registers and return
    pop {r6, pc}
```

En examinant le code ci-dessus, il apparaît que la fonction `fonction.incrementations` peut être appelée interruptions masquées : l'état de masquage est sauvé dans R6, et restitué après l'appel de la section critique.

Premier travail à faire

Mettre en place la section critique pour incrémenter les variables, et vérifier que les quatre valeurs obtenues sont les mêmes.

Étape 10

Fault Handler et assertion

Erreur, faute, défaillance

Avant d'entrer dans le détail des explications de cette étape, il convient de définir les termes **erreur, faute, défaillance**.

Une erreur (error) est la fourniture d'un résultat invalide. Par exemple, un résultat négatif (alors qu'il devrait positif ou nul), un pointeur nul (alors qu'il devrait être non nul). Une erreur n'a pas de conséquence visible, tant que le résultat n'est pas exploité.

Une faute (fault) est l'impossibilité de réaliser une opération : par exemple, extraire une racine carrée d'un nombre négatif, atteindre l'objet référencé par un pointeur nul. Dans du code engendré par un compilateur C++, une faute peut provoquer une exception (dans le sens de ce langage).

Une défaillance (failure) est la non réalisation du service demandé. On observe que le fonctionnement n'est pas celui attendu (tiens, ça a planté !).

Lien :

https://www.nasa.gov/pdf/636745main_day_3-algirdas_avizienis.pdf

Description de cette étape

Les fichiers qui vont être ajoutés dans cette étape (**fault-handlers--assertion.cpp** et **fault-handlers--assertion.h**) vont permettre d'afficher trois sortes de fautes :

- celles détectées par le processeur lui-même ;
- le déclenchement d'une interruption sans que son *handler* soit spécifié ;
- le non respect d'une assertion.

Dans les trois cas, l'occurrence d'une provoque :

- le clignotement synchrone des cinq leds ;
- l'affichage d'un message d'erreur.

L'exécution sera bloquée sur cet affichage.

Le fichier **apnt209.pdf** est un document qui décrit les registres du Cortex-M4 concernés par les interruptions **Fault** (voir plus loin, division entière par zéro).

Travaux à faire :

- à titre d'exemple, on écrira trois programmes mettant en évidence chaque type de faute.

Erreurs détectées par le processeur

Un processeur Cortex-M4 est capable d'engendrer une interruption quand certaines fautes surviennent :

- tentative d'accès à une adresse mémoire non autorisée ;
- exécution d'une instruction invalide ;
- ...

Certains comportements sont paramétrables. Par exemple, la division entière par zéro :

- par défaut, la division par zéro retourne silencieusement un quotient nul : c'est le comportement dans toutes les étapes précédentes ;
- à partir de cette étape et dans toutes les suivantes, le fichier **fault-handlers--assertion.cpp** change ce comportement par défaut, une interruption est déclenchée lors d'une division entière par zéro.

Interruption sans *handler* associé

Jusqu'à présent, cette exception n'était pas exploitée, si bien qu'une faute provoquait l'entrée dans une boucle sans fin (comme toutes les interruptions inutilisées dans **unused-interrupt.s**).

Les fichiers qui vont être ajoutés dans cette étape (**fault-handlers--assertion.cpp** et **fault-handlers--assertion.h**) vont intercepter cette exception.

Le déclenchement d'une interruption non utilisée (c'est-à-dire sans que son *handler* soit spécifié) provoquera l'affichage d'un numéro caractéristique. On retrouvera l'interruption correspondante en examinant le fichier engendré par **zSOULRC/interrupt-handlers.s**.

Assertion

Une assertion est une expression booléenne qui doit être évaluée vraie à l'exécution. Si elle est fausse, un message d'erreur est affiché.

Dans le fichier **fault-handlers--assertion.h**, la fonction **assertion** a pour prototype :

```
void assertion (const bool inAssertion,  
                const uint32_t inMessageValue,  
                const char * inFileName,  
                const int inLine) ;
```

Le deuxième argument **inMessageValue** est simplement une valeur qui sera affichée, et qui *peut aider* à retrouver la cause de l'erreur.

Un exemple d'appel :

```
assertion (expression_booléenne, valeur, __FILE__, __LINE__);
```

Les macros **__FILE__** et **__LINE__** contiennent respectivement le nom du fichier source et le numéro de la ligne courante. Ainsi, le message d'erreur indique précisément où l'erreur a eu lieu.

Travail à faire

Dupliquez le projet de l'étape précédente et renommez-le en par exemple **10-fault-handler--assertion**.

Ajoutez aux sources les fichiers **fault-handlers--assertion.h** et **fault-handlers--assertion.cpp** de l'archive **10-files.tar.bz2**. Remplacer les fichiers **Lcd.h**, **Lcd.cpp**, **dev-board-io.h** et **dev-board-io.cpp** par ceux de cette archive.

Supprimez des sources les fichiers **Lcd-wo-fault-mode.h**, **Lcd-wo-fault-mode.cpp** et **unused-interrupt.s**. En effet, jusqu'à l'étape précédente, ce fichier prenait en charge les interruptions inutilisées, ce qui est maintenant fait par le fichier C++ que l'on vient d'ajouter.

Trois programmes différents sont à écrire :

- un programme qui effectue une violation d'une assertion ;
- un programme qui effectue une division entière par zéro ;
- un programme qui déclenche une interruption sans que son handler soit défini.

Chacun de ses programme est décrit dans les pages suivantes.

Violation d'une assertion

Écrire un programme réalisant la violation d'une assertion. Vérifier que l'affichage nomme le fichier source et la ligne source de cette assertion.

À revoir

La division entière par zéro

Écrire un code effectuant une division entière par zéro.

Dans la fonction **configureFaultRegisters** du fichier **fault-handlers--assertion.cpp** que l'on vient d'ajouter, le bit 4 du registre **SCB_CCR** est mis à 1 (il est à zéro par défaut), ce qui a pour effet de déclencher l'exception **HardFault** lors d'une division par zéro.

Il n'est pas simple de retrouver la ligne source où la division par zéro a eu lieu.

L'affichage associé à l'exception **HardFault** comprend quatre pages (numéro en haut à droite), que l'on peut parcourir en tournant l'encodeur numérique. Retrouver l'erreur n'est pas toujours simple, pour une vision générale voir le document **apnt209.pdf**. L'éthique pour la division entière par zéro est présentée pages suivantes.

La division entière par zéro : retrouver l'erreur (1/2)

Pour retrouver où la division entière par zéro a été exécutée, procéder comme suit.

- ① D'abord regarder le registre UFSR (page 2 de l'affichage *HardFault*) : il a pour valeur 0x0200, qui caractérise la division entière par zéro (voir le document **apnt209.pdf**, page 11) ;
- ② Ensuite, noter la valeur de PC (page 3 de l'affichage *HardFault*) 0x7B2 dans le programme solution, cela est sans doute différent pour vous ; cette valeur est l'adresse de l'instruction fautive ;
- ③ Il faut maintenant trouver la fonction qui contient cette instruction ; pour cela, ouvrez le fichier texte **zPRODUCTS/product.map** et rechercher la fonction par son adresse :

```
.text.cpu.0.loop
    0x0000000000000077c      0x0000000000000077c      0x0000000000000077c
                                zBUILDS/setup-loop.cpp.o
                                cpu.0.loop
.text.start.function
    0x00000000000000800      0x00000000000000800      0x1ec zBUILDS/start-raspberry-pico.cpp.o
```

Ci-dessus, la fonction **cpu.0.loop** commence à l'adresse 0x77C et la suivante commence en 0x800. L'adresse 0x7B2 est dans cette fonction. L'adresse relative de l'instruction fautive par rapport au début de l'instruction est $0x7B2 - 0x77C = 0x36$.

La division entière par zéro : retrouver l'erreur (2/2)

- ④ Exécuter le fichier python **zBUILDS/setup-loop.cpp.objdump.py**. Celui-ci exécute *objdump*, qui affiche le code binaire désassemblé et le code source :

```
void loop0 (USER_MODE) {
    0: b570      push   {r4, r5, r6, lr}
    digitalWrite (L4_LED, !digitalRead (P4_PUSH_BUTTON))
    2: 200c      movs   r0, #12
    .....
    printUnsigned (MODE_ millis (MODE) / gDownCounter) ;
2e: f7ff fffe bl 0 <_Z6millisv>
32: 4d13      ldr r5, [pc, +76] ; (0 <cpu.0.loop+0x80>)
34: 682b      ldr r3, [r5, #1]
36: fbb0 f0f3 udiv   r0, r0, r3
3a: f7ff fffe bl 0 <_Z13printUnsignedm>
    gDownCounter -- ;
3e: 682b      ldr r3, [r5, #0]
```

L'instruction à l'adresse 0x36 est une division.

Interruption sans *handler* associé

Nous allons écrire un programme qui engendre une interruption quand on appuie sur le CLIC de l'encodeur. Cette entrée est la n°2 (notation Arduino), et pour le micro-contrôleur le port PTDO (voir étape 05).

Rappel (étape 05) : la routine **configurePorts** du fichier **dev-board-io.cpp** programme le port n°2 en entrée. Dans la fonction **setup**, il y a deux opérations à faire pour activer l'interruption :

- activer l'interruption sur front descendant du PTDO .

```
PORTD_PCR (0) |= PORT_PCR_IRQC (10) ;
```

- activer l'interruption correspondante (ISRSlot...PORTD) sur le processeur :

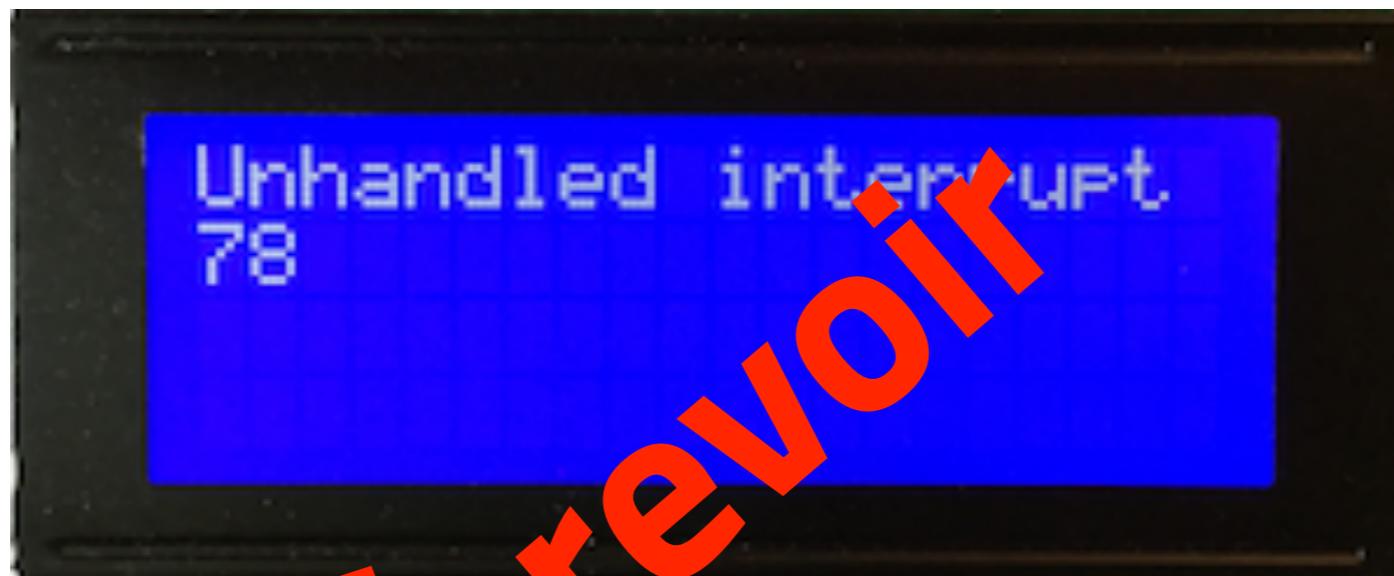
```
NVIC_ENABLE_IRQ (ISRSlot, PORTD)
```

Dans la suite nous allons voir comment

- retrouver le nom de l'interruption qui s'est déclenchée ;
- installer la routine d'interruption associée, qui à titre d'exemple, comptera le nombre d'appuis du bouton.

Interruption sans *handler* associé : l'affichage

Quand on appuie sur CLIC, l'interruption PORTD est déclenchée, et l'affichage devient :



78 est le numéro du vecteur d'interruption associé à l'interruption PORTD : on peut retrouver cette information dans le fichier **rp2040-interrupt-vectors.s** :

```
.word interrupt.PORTC @ 77  
.word interrupt.PORTD @ 78  
.word interrupt.PORTE @ 79  
.word interrupt.SWINT @ 80
```

Implémenter la routine d'interruption

Écrire la routine d'interruption :

- ne pas oublier de déclarer cette routine avec l'annotation `//$interrupt-section PORTD` dans le fichier d'en-tête **setup-loop.h** (comme cela a été présenté dans l'étape 07) ;

Note : il faut acquitter l'interruption dans la routine d'interruption :

```
PORTD_PCR (0) |= PORT_PCR_ISF ;
const uint32_t x __attribute__((unused)) = PORTD_PCR (0) ;
```

La seconde instruction est indispensable, elle force la lecture du registre (la valeur lue n'est pas utilisée), ce qui elle impose d'attendre que l'écriture de la ligne précédente soit réalisée, le retard possible étant dû au *write buffer*.

Étape 11 — Instruction *System Call*

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.

L'instruction svc #n

Les processeurs Cortex-M0+ 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-cpu-0.s

Étapes précédentes

Dans toutes les étapes précédentes, le fichier **reset-handler-sequential-cpu-0.s** contient le *reset handler*, c'est-à-dire le code exécuté au démarrage, est le suivant :

```
reset.handler.cpu.0: // Cortex M0 boots with interrupts enabled, in Thread mode
//----- Run boot, zero bss section, copy data section
    bl    cpu.0.phase3.boot
//----- Set PSP: this is stack for background task
    ldr    r0, =background.task.stack.cpu.0 + BACKGROUND.STACK.SIZE.CPU.0
    msr    psp, r0
//----- Set CONTROL register
// 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
    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    cpu.0.phase3.init
    cpsie i          // Enable interrupts
//----- Run setup, loop
    bl    cpu.0.setup
background.task:
    bl    cpu.0.loop
    b     background.task
```

Code dans reset-handler-sequential-cpu_0-step11.s

Uniquement cette étape

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

```
reset.handler.cpu.0: // Cortex M0 boots with interrupts enabled, in Thread mode
//----- Run boot, zero bss section, copy data section
    bl    cpu.0.phase3.boot
//----- Set PSP: this is stack for background task
    ldr    r0, =background.task.stack.cpu.0 + BACKGROUND.STACK.SIZE.CPU.0
    msr    psp, r0
//----- Set CONTROL register
// 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
    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    cpu.0.setup
background.task:
    bl    cpu.0.loop
    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-cpu-0.s** issu de l'étape précédente en **reset-handler-sequential-cpu-0-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-M0+ (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>.

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	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	

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 **setup0** et **loop0** dans les programmes. Quand ISR_NUMBER ≠ 0, et avec les réglages qui sont adoptés dans ce cours, le processeur ignore les interruptions.

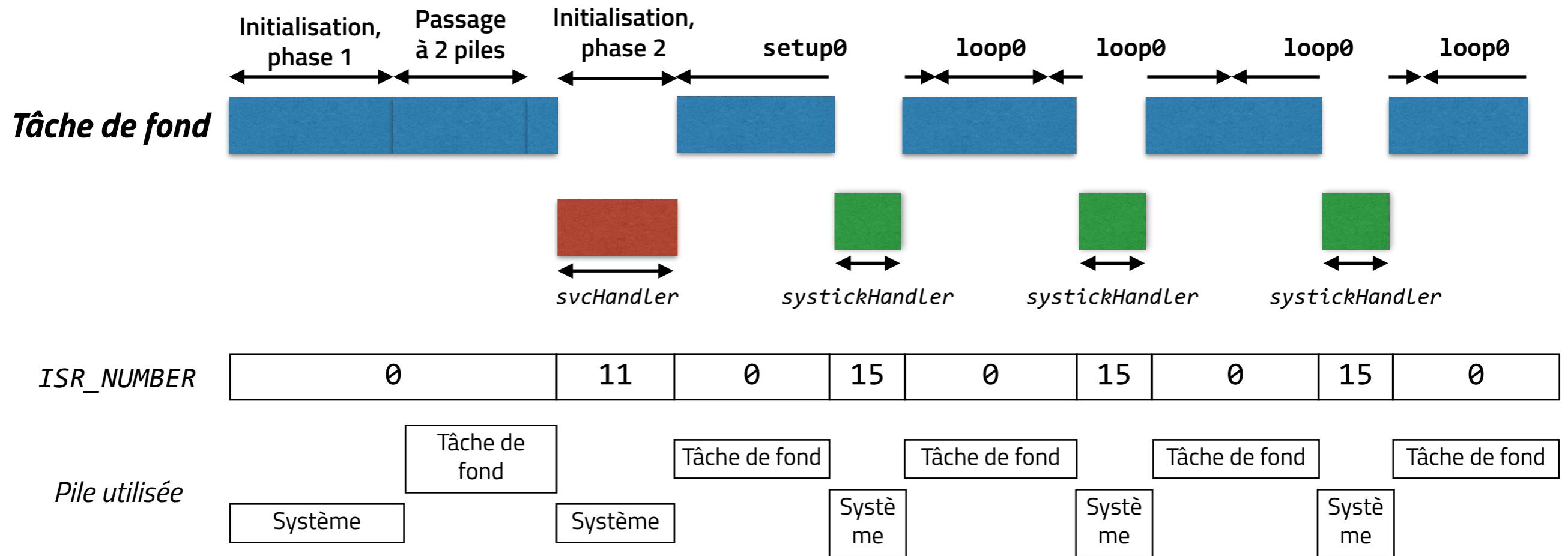
Les autres bits ne sont pas décrits, on n'a pas besoin de connaître leur signification dans ce cours.

Lien :

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

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.

Étape 12

Premier exécutif Temps-Réel

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 les fonctionnalités à l'exécutif.

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)

Duplicer 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-cpu-0.s** ;
- **task-list--32-tasks.h** et **task-list--32-tasks.cpp** ;
- **xtr-step12.h** et **xtr-step12.cpp** (ces fichiers seront modifiés dans l'étape suivante) ;
- **user-tasks.cpp**.

Supprimez de votre projet les fichiers suivants :

- **reset-handler-sequential-step11-cpu-0.s** (il est remplacé par **reset-handler-xtr-cpu-0.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 **setup0** ni **loop0**, 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",
    "../..../dev-files/sources-common",
    "../..../dev-files/sources-digital-io"
],
"DEV-DIR" : "../..../dev-files",
"SERVICE-SCHEME" : "svc",
"SECTION-SCHEME" : "disableInterrupt",
"TASK-COUNT" : 1,
"CPU-MHZ" : 125
}
```

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) {
        if (gDisplayTime <= millis (MODE)) {
            const uint32_t s = systick_current_cpu (MODE) ;
            gotoXY (MODE_ 0, 1) ;
            printUnsigned (MODE_ s) ;
            gotoXY (MODE_ 0, 2) ;
            printUnsigned (MODE_ millis (MODE)) ;
            gDisplayTime += 1000 ;
        }
    }
}
```

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

Étape 12.1 — 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.cpu.0: // Cortex M0 boots with interrupts enabled, in Thread mode
//----- Run boot, zero bss section, copy data section
    b1    cpu.0.phase3.boot
//----- Set PSP: this is stack for background task
    ldr    r0, =background.task.stack.cpu.0 + BACKGROUND.STACK.SIZE.CPU.0
    msr    psp, r0
//----- Set CONTROL register
// 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
    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
//----- This is the background task: turn off activity led
// Activity led 0 is connected to GP26
background.task.cpu.0: // Only use R0, R1, R2, R3 and R12. Other registers are not preserved
    ldr    r0, = 0xD0000000 + 0x018 // Address of GPIO_OUT_CLR control register
    ldr    r1, = (1 << 26) // Port GP26
    str    r1, [r0]           // Turn off
    b     background.task.cpu.0
```

Comme pour l'étape précédente, la seconde phase d'initialisation est réalisée via `svc #0`. Par contre, les appels aux fonctions **setup0** et **loop0** 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 + (R0 << 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 r0, =var.running.task.control.block.ptr  
    mrs r1, psp  
    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 r0 to r11, PSP (stored in R12), LR  
    stmia r0, {r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}  
    b perform.restore.context  
save.background.task.context:  
    ldr r1, =var.background.task.context  
    str r1, [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  cpu.0.phase3.init // 0
```

Ainsi, l'instruction **svc #0** du reset handler provoque l'exécution de la fonction **cpu.0.phase3.init**. 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-M0+. 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 (MODE)) {  
            const uint32_t s = systick_current_cpu (MODE) ;  
            gotoXY (MODE_ 0, 1) ;  
            printUnsigned (MODE_ s) ;  
            gotoXY (MODE_ 0, 2) ;  
            printUnsigned (MODE_ millis (MODE)) ;  
        }  
    }  
}
```

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.

Étape 12.2 — 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++.

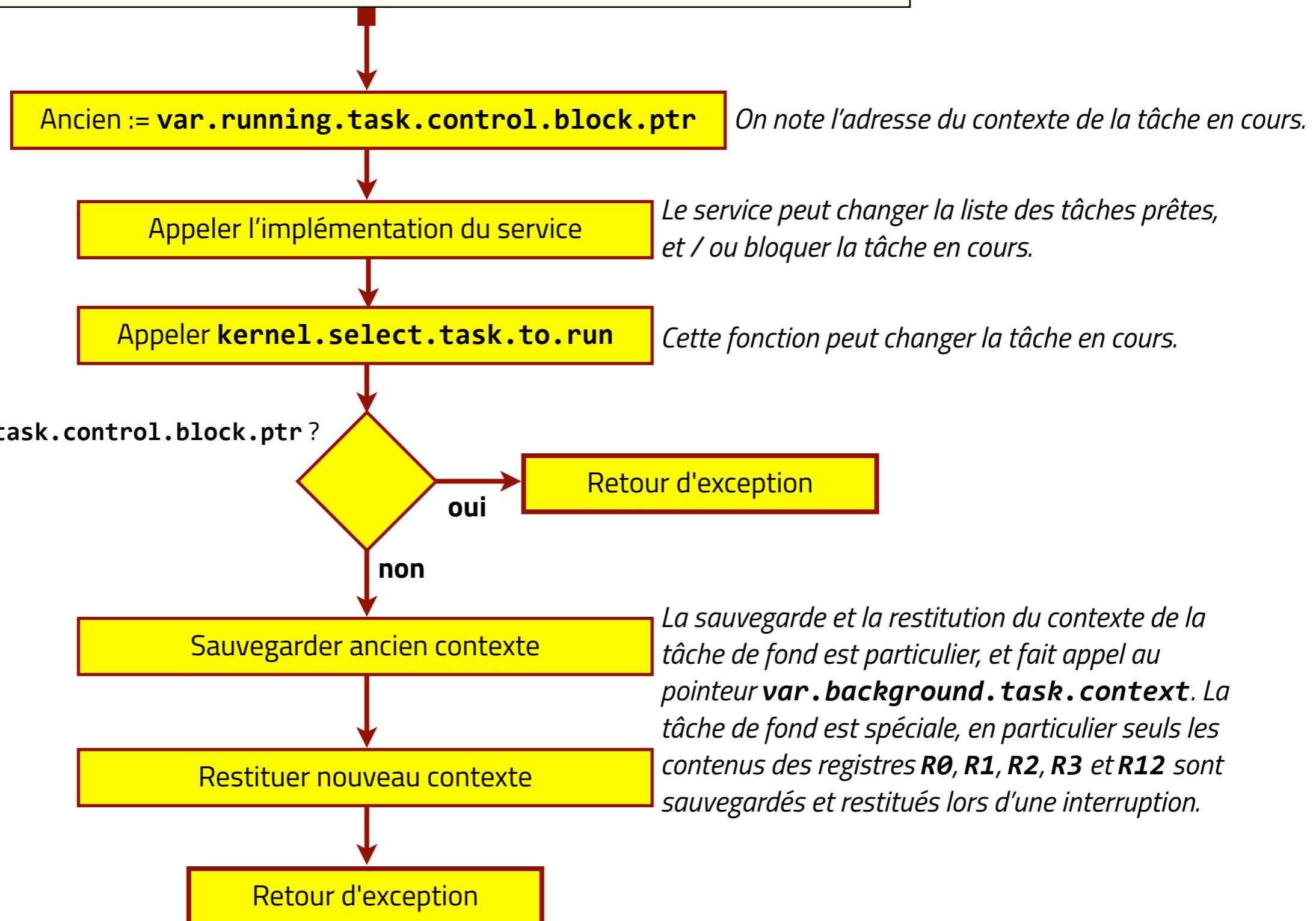
C'est un pointeur dont le rôle est de désigner le descripteur de la tâche en cours, plus précisément le champ qui contient la sauvegarde des registres qui ne sont pas sauvegardés dans la pile lors de l'interruption.

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*

Le point d'entrée est `interrupt.SVC`, dans `zSOURCES/interrupt-handlers.s`



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 en cours (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-M0+ 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.

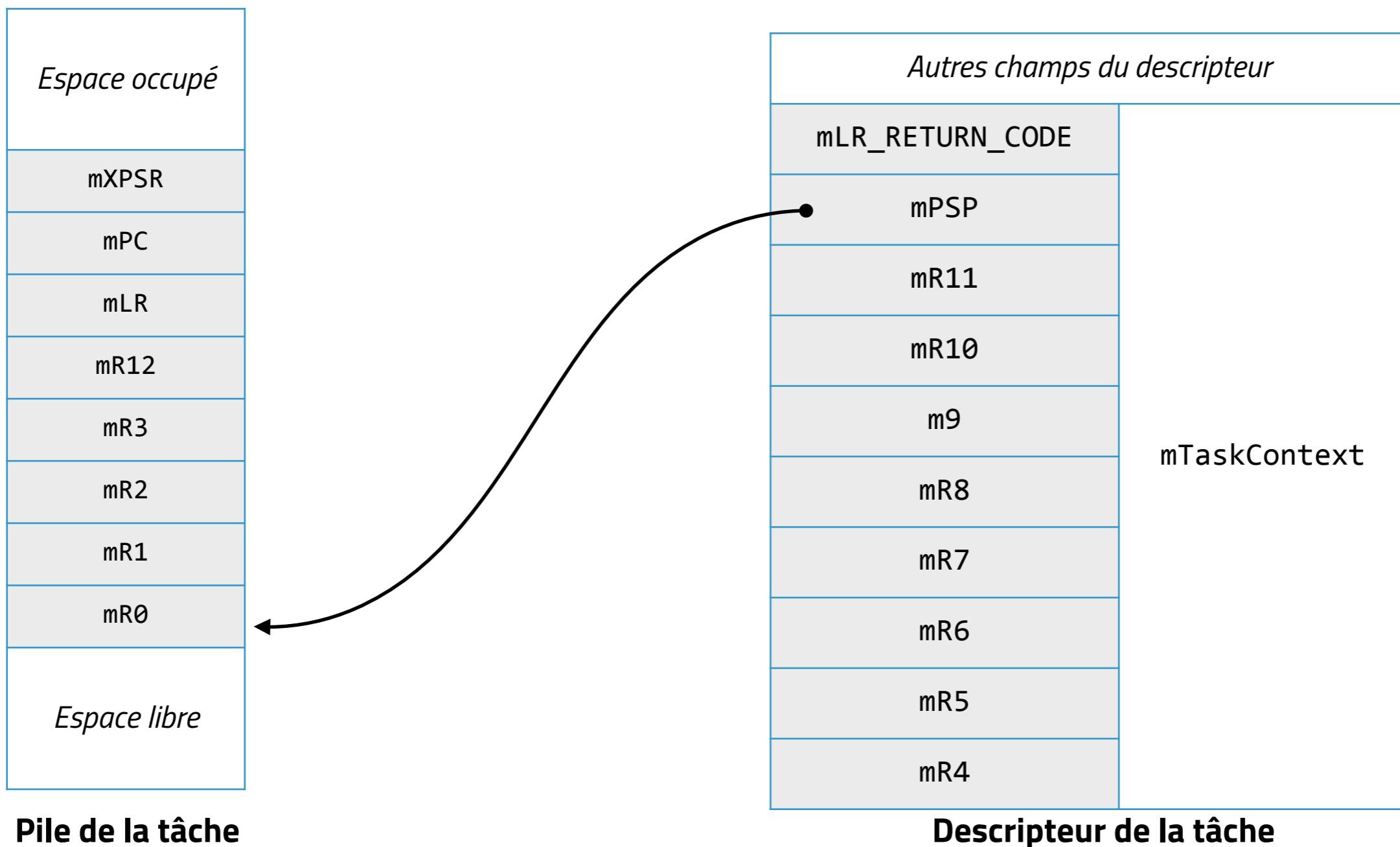
Quand une tâche **quitte** l'état **TASK_RUNNING**, le contenu des registres du processeur est copié dans le champ contexte de son descripteur (« *sauvegarde du contexte* »).

```
typedef struct {
    uint32_t mR0 ;
    uint32_t mR1 ;
    uint32_t mR2 ;
    uint32_t mR3 ;
    uint32_t mR12 ;
    uint32_t mL�� ;
    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 mL��_RETURN_CODE ;
} TaskContext ;
```

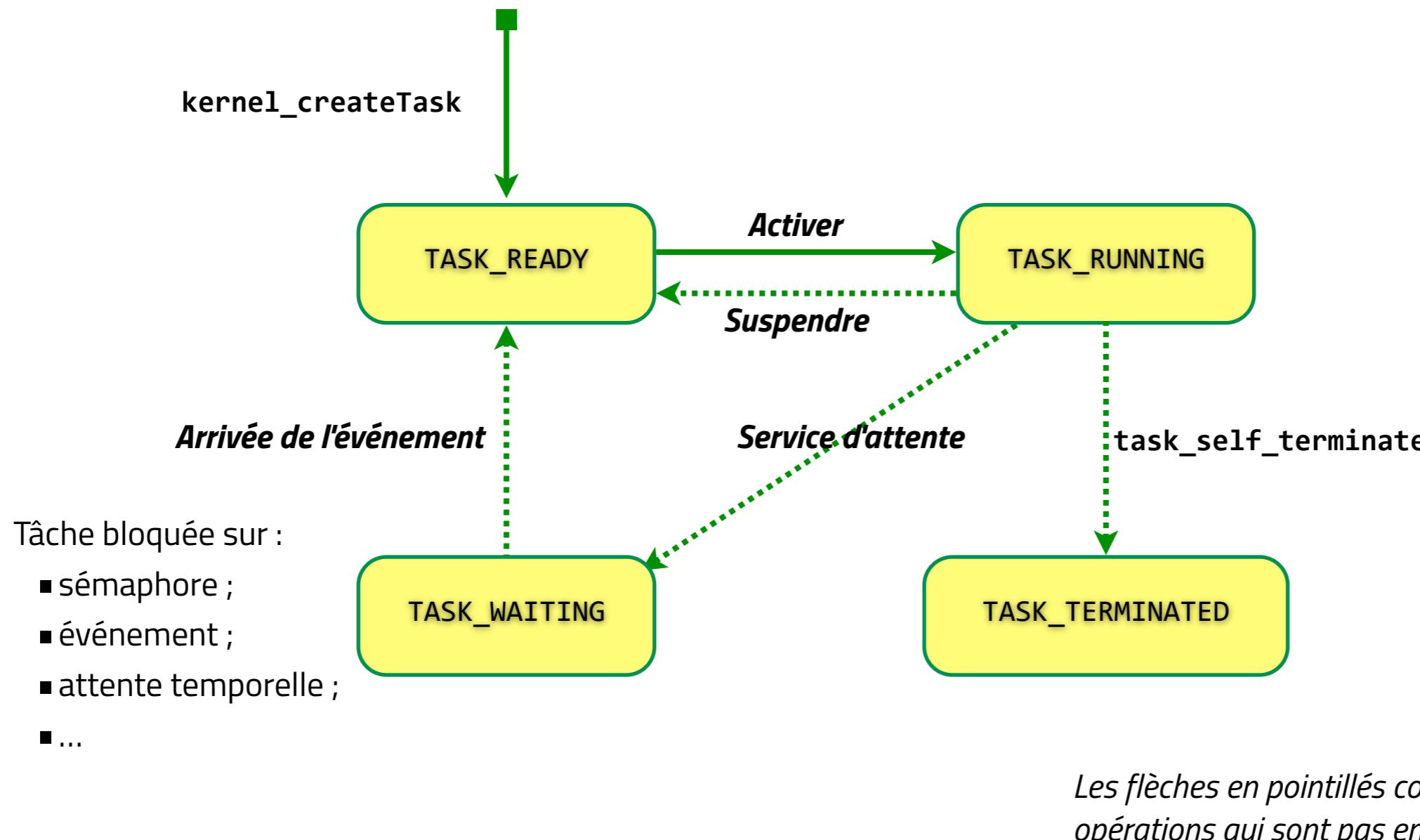
Contexte sauvegardé

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



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.



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) ;      Établissement du contexte initial de la tâche.

    //--- Make task ready
    kernel_makeTaskReady (MODE_ taskControlBlockPtr) ;      La tâche est rendue prête.
    //---
    gTaskIndex += 1 ;
}
```

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 = 0xFFFFFFF0 ;
    //--- 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 **task-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 &) = delete ;  
    private: TaskList & operator = (const TaskList &) = delete ;  
} ;
```

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 moins 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 ;
}
```

Étape 13

Terminaison des tâches

Description de cette étape

Dans cette étape, on ajoute la terminaison des tâches. Dans l'étape précédente, la terminaison d'une tâche provoquait un plantage.

La terminaison d'une tâche est effectuée via un *service de l'exécutif*, c'est donc l'occasion d'exposer comment un service est écrit.

Le code que vous avez à écrire est le suivant :

- déclarer et implémenter plusieurs tâches ;
- chaque tâche fait clignoter une dizaine fois une led, puis se termine.

Le comportement attendu est : la tâche la plus prioritaire fait clignoter sa led et se termine, permettant l'exécution de la tâche suivante, ... jusqu'à ce que la dernière tâche se termine. Il n'y a alors que la tâche de fond qui s'exécute, la led Teensy est quasiment éteinte.

Dupliquer le répertoire de **12-first-real-time-kernel** en **13-task-termination**. Comme l'exécutif va être modifié, renommer **xtr-step12.cpp** en **xtr-step13.cpp** et **xtr-step12.h** en **xtr-step13.h**.

Comment est réalisée la terminaison d'une tâche

On va écrire un *service de l'exécutif*, qui met en œuvre la terminaison de la tâche qui l'appelle. Ce sera l'occasion de montrer comment un service de l'exécutif doit être écrit.

Ensuite, on modifiera la construction du contexte initial d'une tâche, de façon que ce service soit automatiquement appelé.

Écrire un service de l'exécutif

Écrire un *service de l'exécutif*s'effectue en quatre opérations.

- ① Dans un fichier d'en-tête (ici, **xtr-step13.h**), déclarer le prototype du service appelé en mode **USER** :

```
void taskSelfTerminates (USER_MODE) asm ("task.self.terminates") ;
```

- ② Dans le même fichier d'en-tête, déclarer le prototype de la fonction implémentant le service :

```
void service_taskSelfTerminates (KERNEL_MODE) asm ("service.task.self.terminates") ;
```

Cette fonction s'exécute en mode **KERNEL**. Son nom C++ (**service_taskSelfTerminates**) est libre, son nom assembleur doit être le nom assembleur de la fonction correspondante appelée en mode **USER**, précédé par **service..**

- ③ Déclarer le service. Rappelons (étape 01) que tous les fichiers d'en-tête sont examinés par un script Python afin d'analyser différentes déclarations ; l'annotation `//$service` permet de déclarer un service de l'exécutif.

```
//$service task.self.terminates
```

- ④ Implémenter le service. Dans un fichier C++ (ici **xtr-step13.cpp**), écrire la fonction :

```
void service_taskSelfTerminates (KERNEL_MODE) {  
    .... // Voir le contenu pages suivantes  
}
```

Où est écrit `taskSelfTerminates` ?

Dans les quatre opérations de la page précédente, la fonction `taskSelfTerminates` déclarée en ① n'est pas implémentée en C++. La déclaration du service en ④ permet aux scripts Python d'ajouter sa prise en charge dans le fichier `zSOURCES/interrupt-handlers.s` (c'est le nom associé à l'annotation `//$service` qui apparaît) :
`task.self.terminates:`

```
svc #1  
bx lr
```

Le *svc handler* est donc appelé avec un argument égal à 1. Dans le même fichier assembleur, regardez la table des services :

```
svc.dispatcher.table:  
.word cpu.0.phase3.init // 0  
.word service.task.self.terminates // 1
```

Lorsque l'instruction `svc #1` est exécutée, le *svc handler* appelle le service qui est à l'indice 1 dans la table des services.

Note : dans les étapes suivantes, des services seront ajoutés. L'ordre de numérotation des services en fonction de l'ordre d'analyse des fichiers d'en-tête par les scripts Python : le service de terminaison de tâche pourra se voir attribuer un autre indice.

Écrire le service de terminaison de tâche

Plus précisément, ce service va terminer la tâche qui l'appelle.

L'écriture est simple :

```
void service_taskSelfTerminates (KERNEL_MODE) {  
    kernel_makeNoTaskRunning (MODE) ;  
}
```

La fonction **kernel_makeNoTaskRunning** n'est pas présente dans le fichier **xtr-step13.cpp**, il faut l'ajouter :

```
static void kernel_makeNoTaskRunning (KERNEL_MODE) {  
    gRunningTaskControlBlockPtr = nullptr ; // No running task  
}
```

Pourquoi mettre **gRunningTaskControlBlockPtr** à **nullptr** suffit ? En fait, appeler **service_taskSelfTerminates** invoque le *svc handler* (voir page suivante). Le *svc handler* (voir son algorithme dans l'étape 12) appelle d'abord la fonction d'implémentation du service (ici **kernel_makeNoTaskRunning**), puis la fonction **kernel.select.task.to.run**. Celle-ci — aussi décrite dans l'étape 12 — voyant **gRunningTaskControlBlockPtr** à **nullptr** retire une tâche de la liste des tâches prêtes et la rend en cours.

Appeler le service de terminaison de tâche (1/2)

Maintenant, le service de terminaison de tâche existe, mais n'est pas appelé. Il y a deux façons de le faire.

La première est d'écrire l'appel du service de terminaison comme dernière instruction de chaque fonction implémentant le code d'une tâche. Par exemple :

```
static void task1 (USER_MODE) {  
    .....  
    taskSelfTerminates (MODE) ;  
}
```

C'est une écriture que nous ne retiendrons pas, car elle est soumise au bon vouloir du programmeur : un oubli, et c'est le plantage.

Appeler le service de terminaison de tâche (2/2)

La seconde façon est d'inclure dans l'exécutif l'exécution automatique du service de terminaison de tâche. Pour cela, on s'appuie sur une caractéristique de l'*ABI* des processeurs ARM : l'adresse de retour d'un sous-programme est le contenu du registre **LR** du processeur lors de l'entrée dans ce sous-programme.

Autrement dit : lors de l'établissement du contexte initial d'une tâche, il faut mettre dans la sauvegarde du registre **LR** l'adresse de la fonction **taskSelfTerminates**. Pour cela, éditez la fonction **kernel_set_task_context** du fichier **xtr-step13.cpp** et ajouter l'affectation au champ **mLR** :

```
static void kernel_set_task_context (INIT_MODE_
                                      TaskContext & ioTaskContext,
                                      const uint32_t inStackBufferAddress,
                                      const uint32_t inStackBufferSize,
                                      RoutineTaskType inTaskRoutine) {
    .....
    //--- Self termination
    ptr->mLR = uint32_t (taskSelfTerminates) ;
}
```

Note : une *ABI* (*Application Binary Interface*) définit (entre autres) les conventions d'appel des fonctions.

Liens :

https://fr.wikipedia.org/wiki/Application_binary_interface

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

Pour les curieux : le retour de sous-programme (1/3)

Quand un sous-programme est appelé, l'adresse de retour est placée dans le registre **LR** (*Link Register*, ou **R14**). Le jeu d'instructions du Cortex-M4 ne définit pas d'instruction particulière de retour de sous-programme, le retour est simplement un branchement à l'adresse contenue dans **LR**.

On peut trouver un exemple illustratif en regardant comment la fonction **configureFaultRegisters** (fichier **fault-handlers--assertion.cpp**) est compilée :

```
static void configureFaultRegisters (BOOT_MODE) {
    SCB_CCR |= (1 << 4) | (1 << 3) ;
}
```

Le code assembleur engendré est dans **zASBUILDS/fault-handlers--assertion.cpp.s.list** :

```
19          _ZL23configureFaultRegistersv:
23 0000 024A          ldr r2, .L2
24 0002 1368          ldr r3, [r2]
25 0004 43F01803      orr r3, r3, #24
26 0008 1360          str r3, [r2]
27 000a 7047          bx lr
```

Pour les curieux : le retour de sous-programme (2/3)

À titre d'information, il y a des variantes qui sont adoptées pour optimiser (la vitesse, la taille) du code.

Par exemple, si le sous-programme appelle lui-même un autre sous-programme, il faut évidemment sauvegarder **LR**. Par exemple, la fonction **printSpaces** (fichier **lcd.cpp**) :

```
void printSpaces (USER_MODE_ const uint32_t inCount) {
    uint32_t count = inCount ;
    while (count > 0) {
        printChar (MODE_ ' ') ;
        count -- ;
    }
}
```

Le code assembleur engendré est dans **zAS-BUILDS/lcd.cpp.s.list** :

454	_Z11printSpaces:	
457 0000 10B5	push {r4, lr}	
458 0002 0446	mov r4, r0	
459	.L33:	
460 0004 24B1	cbz r4, .L31	
461 0006 2020	movs r0, #32	
462 0008 FFF7FEFF	bl _ZL9writeDatah	
463 000c 013C	subs r4, r4, #1	
464 000e F9E7	b .L33	
465	.L31:	
466 0010 10BD	pop {r4, pc}	pop {r4, pc} est fonctionnellement équivalent à la séquence pop {r4, lr} ; bx lr.

Pour les curieux : le retour de sous-programme (3/3)

Une autre optimisation peut être faite par le compilateur quand la dernière instruction de la fonction C ou C++ est l'appel d'une autre fonction : au lieu d'effectuer un appel de sous-programme, un simple branchement est effectué. Par exemple, la fonction **printHex2** (fichier **Lcd.cpp**) :

```
void printHex2 (USER_MODE_ const uint32_t inValue) {  
    printHex1 (MODE_ inValue >> 4) ;  
    printHex1 (MODE_ inValue) ;  
}
```

Le code assembleur engendré est dans **ZASBUILDS/Lcd.cpp.s.list** :

```
619          _Z9printHex2m  
622 0000 10B5      push {r4, lr}  
623 0002 0446      mov r4, r0  
624 0004 0009      lsrs r0, r0, #4  
625 0006 FFF7FEFF bl _Z9printHex1m  
626 000a 2046      mov r0, r4  
627 000c BDE81040 pop {r4, lr}  
628 0010 FFF7FEBF b _Z9printHex1m
```

Étape 14

Attentes passives
(waitDuring, waitUntil)

Description de cette étape

Dans cette étape, on ajoute l'attente temporelle passive. Jusqu'à l'étape précédente, les fonctions **busyWaitDuring** et **busyWaitUntil** effectuent des attentes actives, c'est-à-dire qu'elles accaparent le processeur durant l'attente — d'ailleurs la led Teensy qui visualise l'occupation processeur est active.

Refaites tourner le programme de l'étape précédente. Les tâches s'exécutent les unes après les autres, dans l'ordre décroissant de leur priorité.

Le code que vous avez à écrire est d'implémenter l'attente passive, les fonctions **busyWaitDuring** et **busyWaitUntil** étant renommées par conséquent **waitDuring** et **waitUntil**.

Attention, l'exclusion mutuelle de l'accès à l'afficheur LCD n'est pas (encore) assurée : votre code ne doit pas appeler les fonctions de l'afficheur LCD en concurrence.

Les fonctions **busyWaitDuring_initMode** et **busyWaitDuring_faultMode** sont inchangées.

Duplicer le projet de l'étape précédente et renommez-le par exemple **14-wait**.

Écriture de la fonction waitDuring

Dans les fichiers **time.h** et **time.cpp**, supprimez la fonction **busyWaitDuring** et remplacez-la par la fonction **waitDuring** qui appelle simplement la fonction **waitUntil** :

```
void waitDuring (USER_MODE_ const uint32_t inDelayMS) {  
    waitUntil (MODE_ gUptime + inDelayMS),  
}
```

Il vous faut aussi remplacer toutes les occurrences de l'appel à la fonction **busyWaitDuring** et par l'appel de **waitDuring**, notamment dans **lcd.cpp**.

Écriture de la fonction waitUntil

La fonction **waitUntil** est appelée en mode **USER**, or le blocage d'une tâche ne peut être effectué que par des routines en mode **KERNEL**. Il faut donc créer un service, comme cela a été fait à l'étape précédente avec **taskSelfTerminates**.

Déclarez dans le fichier **time.h** le service ; pour uniformiser, adoptez les prototypes suivants :

```
void waitUntil (USER_MODE_ const uint32_t inDeadlineMS) asm ... ;  
void service_waitUntil (KERNEL_MODE_ const uint32_t inDeadlineMS) asm ... ;
```

Écrire l'implémentation du service dans le fichier **time.c** (supprimez la fonction **busyWaitUntil**) :

```
void service_waitUntil (KERNEL_MODE_ const uint32_t inDeadlineMS) {  
    if (inDeadlineMS > gUptime) {  
        kernel_blockOnDeadline (MODE_KERNEL, inDeadlineMS) ;  
    }  
}
```

Le fonctionnement est simple : si l'échéance **inDeadlineMS** n'est pas atteinte, la tâche appelante est bloquée.

Il reste à écrire la fonction **kernel_blockOnDeadline** qui va bloquer une tâche en attente de l'échéance, et aussi à modifier la routine d'interruption temps-réel de façon qu'elle libère les tâches dont l'échéance est atteinte.

Écriture de la fonction kernel_blockOnDeadline

Il faut ajouter dans le descripteur des tâches (fichier **xtr.cpp**) la déclaration du champ **mDeadline** :

```
typedef struct TaskControlBlock {
    --- Context buffer
    TaskContext mTaskContext ; // SHOULD BE THE FIRST FIELD
    --- This field is used for deadline
    uint32_t mDeadline ;
    --- Task index
    uint8_t mTaskIndex ;
} TaskControlBlock ;
```

La fonction **kernel_blockOnDeadline** est simple à écrire : elle insère la tâche en cours dans la liste des tâches bloquées sur échéance, écrit l'échéance dans le descripteur de tâche, puis bloque la tâche en cours.

Déclarez dans le fichier **xtr.cpp** la liste des tâches bloquées sur échéance :

```
static TaskList gDeadlineWaitingTaskList ;
```

Écrire ensuite l'implémentation de **kernel_blockOnDeadline** dans le fichier **xtr.cpp** :

```
void kernel_blockOnDeadline (KERNEL_MODE_ const uint32_t inDeadline) {
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;
    --- Insert in deadline list
    gRunningTaskControlBlockPtr->mDeadline = inDeadline ;
    gDeadlineWaitingTaskList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;
    --- Block task
    kernel_makeNoTaskRunning (MODE) ;
}
```

XTR_ASSERT_NON_NULL_POINTER est une macro qui fait appel à une assertion (voir le début du fichier **xtr.cpp**).

Il reste à modifier la routine d'interruption temps-réel.

Modifier la routine d'interruption temps-réel (1/2)

Celle-ci est déclarée dans **time.h**. Depuis l'étape 07 où elle a été ajoutée, elle est déclarée comme suit :

```
//$interrupt-section SysTick  
void systickInterruptServiceRoutine (SECTION_MODE) asm ("interrupt.section.SysTick") ;
```

Elle s'exécute en mode **SECTION**. Or dans ce mode, les routines de l'exécutif ne peuvent pas être appelées : le mode **SECTION** est réservé aux routines exécutées à l'exécutif.

Ici, nous voulons appeler des routines de l'exécutif qui permettent de débloquer des tâches. Ce n'est pas exactement le mode **KERNEL**, car celui-ci permet aussi de bloquer la tâche en cours (ce qui n'aurait aucun sens dans une routine d'interruption). C'est le mode **IRQ** qui est réservé aux routines d'interruption qui peuvent débloquer des tâches. Il faut aussi changer l'annotation en **//\$interrupt-service** , car il faut prendre en charge un éventuel changement de contexte. Les changements à effectuer sont en bleu :

```
//$interrupt-service SysTick  
void systickInterruptServiceRoutine (IRQ_MODE) asm ("interrupt.service.SysTick") ;
```

Modifier la routine d'interruption temps-réel (2/2)

Mais ce n'est pas tout pour cette fonction. En effet, elle appelle les fonctions dont l'adresse est placée dans la section **real.time.interrupt.routine.array** (voir étape 08). Pour faciliter l'inscription d'une routine à cette section, la macro suivante a été déclarée dans le fichier **time.h** :

```
#define MACRO_REAL_TIME_ISR(ROUTINE) \
    static void (* UNIQUE_IDENTIFIER) (SECTION_MODE_ const uint32_t inUptime) \
    __attribute__ ((section ("real.time.interrupt.routine.array"))) \
    __attribute__ ((unused)) \
    __attribute__ ((used)) = ROUTINE ;
```

C'est-à-dire que les routines doivent être déclarées pour être exécutées en mode **SECTION**. Or, maintenant, elle doivent s'exécuter en mode **IRQ** :

```
#define MACRO_REAL_TIME_ISR(ROUTINE) \
    static void (* UNIQUE_IDENTIFIER) (IRQ_MODE_ const uint32_t inUptime) \
    __attribute__ ((section ("real.time.interrupt.routine.array"))) \
    __attribute__ ((unused)) \
    __attribute__ ((used)) = ROUTINE ;
```

Libérer les tâches dont l'échéance est atteinte (1/2)

On va donc inscrire une routine dans la section **real.time.interrupt.routine.array** ; ainsi, elle est exécutée à chaque occurrence de l'interruption temps-réel :

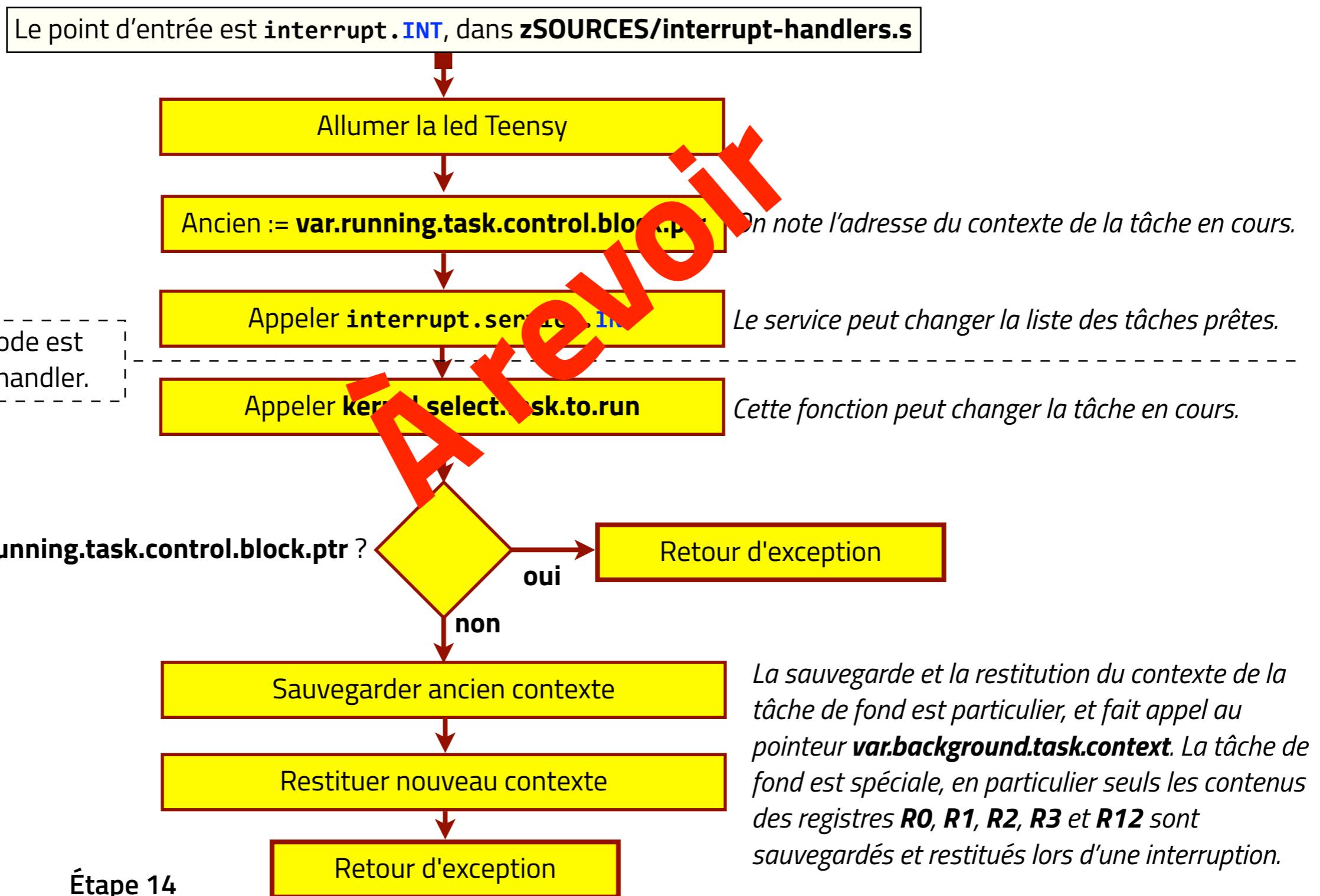
```
static void irq_makeTasksReadyFromDate (IRO_IODL const uint32_t inCurrentDate) {  
    .....  
}  
  
MACRO_REAL_TIME_ISR (irq_makeTasksReadyFromDate) ;
```

Que faut-il faire dans cette fonction ? Parcourir la liste des tâches bloquées, pour libérer celles dont l'échéance est atteinte.

Organigramme d'une routine d'interruption, mode IRQ

Cette routine est déclarée dans un fichier d'en-tête par :

```
//$interrupt-service INT  
void routine (IRQ_MODE) asm ("interrupt.service.INT") ;
```



Libérer les tâches dont l'échéance est atteinte (2/2)

On utilise la classe **TaskList::Iterator** qui implémente un itérateur de liste de tâches :

- le constructeur initialise l'itérateur ;
- la méthode **nextTask** renvoie :
 - ▶ **nullptr** si on est arrivé à la fin de la liste ;
 - ▶ le pointeur du descripteur courant, et avance au descripteur suivant.

La fonction **irq_makeTasksReadyFromCurrentDate** est à ajouter dans **xtr.cpp** :

```
static void irq_makeTasksReadyFromCurrentDate (MODE_ const uint32_t inCurrentDate) {  
    TaskList::Iterator iterator (MODE_ gDeadlineWaitingTaskList) ;  
    TaskControlBlock * task ;  
    while ((task = iterator.nextTask (0))) {  
        if (inCurrentDate >= task->mDeadline) {  
            //--- Remove task from deadline list  
            gDeadlineWaitingTaskList.removeTask (MODE_ task) ;  
            //--- Make task ready  
            kernel_makeTaskReady (MODE_ task) ;  
        }  
    }  
}
```

Recompilation et exécution

Les modifications et ajouts sont terminées, recompilez et exécutez le programme. Notez que maintenant les tâches s'exécutent en parallèle (en fait, pseudo-parallélisme), et que la led Teensy est très faiblement éclairée (activité processeur réduite).

À revoir

Étape 15

Outils de synchronisation

Description de cette étape

Cette étape est consacrée aux outils de synchronisation. On peut classer les primitives bloquantes des outils de synchronisation en trois niveaux :

- le blocage simple d'une tâche : par exemple `s.P()` ; c'est ce qui est présenté dans cette étape ;
- le blocage d'une tâche jusqu'à une échéance : par exemple `s.P_until(échéance)`, présenté dans l'étape suivante ;
- le blocage en garde : par exemple `[s.P() -> ... | ... P() ->]`, présenté dans la dernière étape.

Deux fonctions de base vont être présentées dans cette étape : `kernel_blockRunningTaskInList` et `irq_makeTaskReadyFromList` ; elles permettent d'exprimer la plupart des outils de synchronisation.

Dupliquer le projet de l'étape précédente et renommez-le par exemple **15-synchronization**.

La fonction kernel_blockRunningTaskInList

Cette fonction bloque la tâche en cours et insère son descripteur dans la liste passée en argument ; elle est appellable en mode **KERNEL** :

```
void kernel_blockRunningTaskInList (KERNEL_MODE TaskList & ioWaitingList) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    //--- Insert in task list  
    ioWaitingList.enterTask (MODE_ gRun...gTaskControlBlockPtr) ;  
    //--- Block task  
    kernel_makeNoTaskRunning (MODE) ;  
}
```

Insérer cette fonction dans **xtr.cpp**, et son prototype dans **xtr.h**.

La fonction irq_makeTaskReadyFromList

Cette fonction, appellable en mode **IRQ**, agit sur la liste de tâches passée en argument :

- si la liste est vide, la valeur **false** est renvoyée par la fonction ;
- si la liste n'est pas vide, la tâche la plus prioritaire en est retirée, cette tâche est rendue prête, et la valeur **true** est renvoyée par la fonction.

```
bool irq_makeTaskReadyFromList (IRQ_MODE_TaskList & ioWaitingList) {  
    TaskControlBlock * taskPtr = ioWaitingList.RemoveFirstTask (MODE) ;  
    const bool found = taskPtr != nullptr ;  
    if (found) {  
        kernel_makeTaskReady (MODE_taskPtr) ;  
    }  
    return found ;  
}
```

Insérer cette fonction dans **xtr.cpp**, et son prototype dans **xtr.h**.

Le sémaPhore de Dijkstra (1/5)

On trouve dans la littérature de nombreuses implémentations du sémaPhore de Dijkstra, par exemple :

Un sémaPhore de Dijkstra est constitué :

- d'une variable **e** positive, négative ou nulle ;
- une liste **l** de tâches.

Initialisation

```
e prend une valeur ≥ 0  
l est la liste vide
```

Primitive P

```
e := e - 1 ;  
si e < 0 alors  
    Bloquer la tâche en cours dans l  
    Appeler l'ordonnanceur  
finsi
```

Primitive V

```
e := e + 1  
si e ≥ 0 alors  
    une tâche est retirée de l  
    cette tâche est rendue prête  
    Appeler l'ordonnanceur  
finsi
```

Un sémaPhore de Dijkstra est constitué :

- d'une variable **e** positive ou nulle ;
- une liste **l** de tâches.

Initialisation

```
e prend une valeur ≥ 0  
l est la liste vide
```

Primitive P

```
si e == 0 alors  
    Bloquer la tâche en cours dans l  
    Appeler l'ordonnanceur  
sinon  
    e := e - 1 ;  
finsi
```

Primitive V

```
si l est vide alors  
    e := e + 1  
sinon  
    une tâche est retirée de l  
    cette tâche est rendue prête  
    Appeler l'ordonnanceur  
finsi
```

Le sémafor de Dijkstra (2/5) : déclaration de la classe

On choisit la seconde implémentation, car elle permet facilement l'extension à l'attente temporelle et aux commandes gardées. En effet, le blocage d'une tâche lors d'un appel de **P** n'a pas d'autre effet de bord que l'insertion dans la liste des tâches bloquées.

Le sémafor est déclaré comme une classe C++. Les primitives doivent être définies comme des services.

```
#include "task-list--32-tasks.h"

class Semaphore {
    //--- Properties
protected: TaskList mWaitingTaskList ;
protected: uint32_t mValue ;

    //--- Constructor
public: Semaphore (const uint32_t _initialValue) ;

    //--- V
//$service semaphore.V
public: void V (USER_MODE) asm ("semaphore.V") ;
public: void sys_V (IRQ_MODE) asm ("service.semaphore.V") ;

    //--- P
//$service semaphore.P
public: void P (USER_MODE) asm ("semaphore.P") ;
public: void sys_P (KERNEL_MODE) asm ("service.semaphore.P") ;

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

Un sémafor de Dijkstra est constitué :

- d'une variable **e** positive ou nulle ;
- une liste **l** de tâches.

À revoir

Le sémafor de Dijkstra (3/5) : initialisation

Initialisation

e prend une valeur ≥ 0
l est la liste vide

Le constructeur effectue l'initialisation :

```
Semaphore::Semaphore (const uint32_t initialValue) :  
mWaitingTaskList (),  
mValue (initialValue) {  
}
```

À revoir

Le sémaforo de Dijkstra (4/5) : primitive P

Primitive P

```
si e == 0 alors
    Bloquer la tâche en cours dans l
    Appeler l'ordonnanceur
sinon
    e := e - 1 ;
finsi
```

La méthode **sys_P** implémente la primitive P du sémaforo :

```
void Semaphore::sys_P (KERNEL_MODE) {
    ..... à vous d'écrire l'implémentaton .....
}
```

Notez que l'ordonnanceur n'est pas explicitement appelé. En effet, **sys_P** est appelé par la méthode **P** via le *svc handler* qui se charge lui-même d'appeler la fonction **kernel.select.task.to.run**.

Les annotations de mode garantissent qu'une routine d'interruption n'appellera ni **P** ni **sys_P**.

À revoir

Le sémaforo de Dijkstra (5/5) : primitive v

Primitive V

```
si l est vide alors
    e := e + 1
sinon
    une tâche est retirée de l
    cette tâche est rendue prête
    Appeler l'ordonnanceur
finsi
```

La méthode **sys_V** implémente la primitive **V** du sémaforo.

```
void Semaphore::sys_V (IRQ_MODE) {
    ..... à vous d'écrire l'implémentation ....
}
```

Notez que l'ordonnanceur n'est pas explicitement appelé. En effet :

- soit **sys_V** est appelé par la méthode **V** via le *svc handler* qui se charge lui-même d'appeler la fonction **kernel.select.task.to.run**;
- soit **sys_V** est appelé directement par une routine d'interruption, qui doit être déclarée en mode **IRQ**, ce qui provoquera l'appel de la fonction **kernel.select.task.to.run**.

Les annotations de mode garantissent qu'une routine d'interruption n'appellera pas **V** mais **sys_V**.

Travail à faire

Écrire le code du sémaphore dans des fichiers **Semaphore.h** et **Semaphore.cpp**.

Ajouter un sémaphore d'exclusion mutuelle dans **Lcd.cpp** de façon qu'un caractère soit écrit de façon indivisible.

Attention, dans le code de **Lcd.cpp**, on ne demande pas à ce qu'une séquence de caractères soit écrite de façon indivisible. Par exemple, si deux tâches écrivent deux nombres en parallèle, les deux séquences seront entrelacées.

On peut à titre d'information voir comment est géré le même problème sur les ordinateurs de bureau (voir pages suivantes).

Un exemple d'utilisation des sémaphores (1/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//-----
static const int NOMBRE_THREADS = 10 ;

//-----

static void codeThread (int tid) {
    std::cout << "thread " << tid << std::endl ;
}

//-----

int main (void) {
    //--- Déclaration des threads
    std::thread t [NOMBRE_THREADS] ;
    //--- Démarrage des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i] = std::thread (codeThread, i) ;
    }
    //--- Message
    std::cout << "main\n";
    //--- Attente de la fin de l'exécution des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i].join () ;
    }
    //---
    return 0;
}
```

Compilation : g++ main.cpp -o main

À revoir

Lire attentivement le programme ci-contre, et le faire tourner sur votre ordinateur de bureau.

11 threads se déroulent en parallèle, chacun d'eux affiche un message.

Voici le résultat de deux exécutions :

```
main
thread 9
thread 10 ethhhhhhahrrrrrrrdreeeeee eaaaaaaaaaddaaaa
d 23074568
```

```
mttattttttthihhrhhhrhnrrrrrer
eeaaaaaaaaaaaaadadd dddddd d 9 8 03
25164
7
```

Les caractères sont corrects, mais les affichages sont entremêlés.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- -std=c++11 (les threads sont définis à partir du C++ 11) ;
- -lpthread (édition des liens avec la librairie libpthread).

Un exemple d'utilisation des sémaphores (2/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//-----
static const int NOMBRE_THREADS = 10 ;
static std::mutex semaphore ; // Sémaphore initialisé à 1

//-----

static void codeThread (int tid) {
    semaphore.lock () ; // P(semaphore)
    std::cout << "thread " << tid << std::endl ;
    semaphore.unlock () ; // V (semaphore)
}

//-----

int main (void) {
    //--- Déclaration des threads
    std::thread t [NOMBRE_THREADS] ;
    //--- Démarrage des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i] = std::thread (codeThread, i) ;
    }
    //--- Message
    semaphore.lock () ; // P(semaphore)
    std::cout << "main\n";
    semaphore.unlock () ; // V (semaphore)
    //--- Attente de la fin de l'exécution des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i].join () ;
    }
    //---
    return 0;
}
```

À revoir

Compilation : g++ main.cpp -o main

On ajoute maintenant un sémaphore d'exclusion mutuelle (en bleu).

Voici le résultat de deux exécutions :

```
main
thread 0
thread 7
thread 8
thread 6
thread 9
thread 1
thread 3
thread 4
thread 2
thread 5
```

```
thread 2
main
thread 4
thread 8
thread 9
thread 5
thread 0
thread 1
thread 1
thread 6
thread 3
thread 7
```

Les affichages sont corrects, l'ordre peut varier d'une exécution à une autre.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- -std=c++11 (les threads sont définis à partir du C++ 11) ;
- -lpthread (édition des liens avec la librairie libpthread).

Quelques outils de synchronisation

L'évènement fugace

Un événement fugace contient une liste de tâches bloquées.

Initialement, la liste des tâches bloquées est vide.

La primitive **wait** bloque inconditionnellement la tâche qui l'appelle.

La primitive **signal** libère toutes les tâches bloquées.

À revoir

L'évènement mémorisé

Un événement mémorisé contient une liste de tâches bloquées et un booléen.

Initialement, la liste des tâches bloquées est vide et le booléen est faux ou vrai.

La primitive **wait** :

- si le booléen est vrai, il est mis à faux ;
- si il est faux, la tâche appelante est bloquée.

La primitive **signal** :

- si la liste des tâches bloquées n'est pas vide, toutes les tâches bloquées sont rendues prêtes ;
- si la liste des tâches bloquées est vide, le booléen est mis à vrai.

Porte logicielle

Une porte logicielle contient une liste de tâches bloquées et un booléen, qui indique si la porte est ouverte ou fermée.

Initialement, la liste des tâches bloquées est vide et la porte est ouverte ou fermée.

La primitive **wait** :

- si la porte est ouverte, la tâche appelante passe dans l'état bloquée ;
- si la porte est fermée, la tâche appelante reste bloquée.

La primitive **open** :

- si la porte est ouverte, aucune action : la liste des tâches bloquées est vide ;
- si la porte est fermée, elle est ouverte, et toutes les tâches bloquées sont rendues prêtes.

La primitive **close** :

- ferme la porte.

Rendez-vous : port de Silberschatz

Le *port de Silberschatz* est un outil de synchronisation basé sur le rendez-vous, très semblable à la synchronisation de CSP. Les différences sont :

- les commandes d'entrée et de sortie nomment un *port*, au lieu de nommer la tâche correspondante ;
- dans un premier temps, il n'y a pas de donnée transmise, c'est une synchronisation pure.

La commande d'entrée est notée **A?**, la commande de sortie **A!**.

Commande de sortie **A!** :

- si une ou plusieurs tâches ayant invoqué une commande d'entrée **A?** sur le même port sont bloquées alors :
 - ▶ une tâche bloquée est libérée ;
 - ▶ la commande de sortie n'est pas bloquante ;
- si aucune tâche ayant invoqué une commande d'entrée sur le même port n'est bloquée alors la commande de sortie est bloquante.

Commande d'entrée **A?** :

- si une ou plusieurs tâches ayant invoqué une commande de sortie **A!** sur le même port sont bloquées alors :
 - ▶ une tâche bloquée est libérée ;
 - ▶ la commande d'entrée n'est pas bloquante ;
- si aucune tâche ayant invoqué une commande de sortie sur le même port n'est bloquée alors la commande d'entrée est bloquante.

Port de Silberschatz avec transmission de données

Nous allons maintenant ajouter la transmission de données au *port de Silberschatz*. À titre d'exemple, nous considérerons que la donnée est de type `uint32_t`. Pour un type quelconque, utiliser un *template C++ de classe*.

Le problème est plus complexe qu'il n'y paraît. Il faut s'assurer de la synchronisation et que les recopies de données s'effectuent en exclusion mutuelle.

Le plus simple (à mon avis...) est définir une nouvelle classe qui possède comme propriétés un port de Silberchatz (sans donnée), et des sémaphores.

Étape 16 — Attentes combinées

Description de cette étape

Nous allons étendre les outils de synchronisation avec la possibilité d'effectuer une attente jusqu'à une échéance. C'est une première extension, moins générale que les *commandes gardées*.

Ceci impose de faire des modifications importantes dans l'exécutif, que nous allons détailler dans les pages qui suivent.

Dupliquez le projet de l'étape précédente et renommez-le par exemple **16-synchronization+wait-until**.

Primitive P_until

C'est un exemple typique de ce que l'on veut écrire. La primitive **P_until** effectue une attente jusqu'à une échéance. C'est une méthode qui sera rajoutée à la classe **Semaphore**, et dont l'appel s'écrit :

```
bool r = s.P_until (MODE_ échéance) ;
```

Le fonctionnement est le suivant :

- si, au moment de l'appel le sémaphore est strictement positif, le sémaphore est décrémenté, et la primitive retourne immédiatement avec la valeur **true**;
- si, au moment de l'appel le sémaphore est nul et l'échéance est atteinte ou dépassée, et la primitive retourne immédiatement avec la valeur **false**;
- sinon, la primitive est bloquante, la tâche est en attente sur le sémaphore et l'échéance, jusqu'à ce que :
 - ▶ la tâche est débloquée suite à l'invocation de **V** par une autre tâche ; la valeur **true** est retournée ;
 - ▶ la tâche est débloquée parce que l'échéance est atteinte ; la valeur **false** est retournée.

Dans tous les cas, la valeur de retour indique l'évènement qui a permis de passer la primitive :

- **true** si c'est le sémaphore ;
- **false** si c'est l'échéance.

Modifications à apporter à l'exécutif

Modifications à apporter à l'exécutif

Maintenant, une tâche peut se retrouver bloquée dans deux listes :

- la liste des tâches bloquées de l'outil de synchronisation ;
- la liste des tâches bloquées sur échéance.

Lors du déblocage par l'outil de synchronisation, il faut :

- enlever la tâche la liste des tâches bloquées sur échéance ;
- retourner la valeur **true**.

Lors du déblocage par l'atteinte de l'échéance, il faut :

- enlever la tâche la liste des tâches bloquées sur l'outil de synchronisation ;
- retourner la valeur **false**.

La valeur booléenne retournée est stockée temporairement dans un nouveau champ du descripteur de tâche, **mUserResult**.

Descripteur de tâches

Ajouter les champs **mBlockingList** et **mUserResult** au descripteur de tâche (fichier **xtr.cpp**) :

```
typedef struct TaskControlBlock {  
    //--- Context buffer  
    TaskContext mTaskContext ; // SHOULD BE THE FIRST FIELD  
    //--- This field is used for deadline  
    uint32_t mDeadline ;  
    //--- Task blocking list (nullptr if task is not blocked)  
    TaskList * mBlockingList ;  
    //--- Task index  
    uint8_t mTaskIndex ;  
    //--- User result  
    bool mUserResult ;  
} TaskControlBlock ;
```

À revoir

Par défaut, ces champs sont initialisés par des zéros binaires, ce qui correspond aux valeurs **nullptr** et **false**.

Fonctions kernel_setUserResult et getUserResult

Ces deux fonctions sont à ajoutées dans le fichier **xtr.cpp** :

```
void kernel_setUserResult (KERNEL_MODE_ const bool inUserResult) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    gRunningTaskControlBlockPtr->mUserResult = inUserResult ;  
}  
  
bool getUserResult (USER_MODE) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    return gRunningTaskControlBlockPtr->mUserResult ;  
}
```

Ajouter aussi la déclaration du prototype de ces fonctions dans **xtr.h** :

```
void kernel_setUserResult (KERNEL_MODE_ const bool inUserResult) ;  
  
bool getUserResult (USER_MODE) asm ("get.user.result") ;
```

L'annotation **asm** suggère que la seconde est appelée par des routines assembleur.

Fonction kernel_makeTaskReady

Ajouter l'argument **inUserResult** à la fonction **kernel_makeTaskReady** (fichier **xtr.cpp**) :

```
static void kernel_makeTaskReady (IRQ_MODE_
                                TaskControlBlock * inTaskPtr,
                                const bool inUserResult) {
    XTR_ASSERT_NON_NULL_POINTER (inTaskPtr);
    gReadyTaskList.enterTask (MODE_ inTaskPtr);
    inTaskPtr->mUserResult = inUserResult;
}
```

Ensuite, pour tous les appels de cette fonction, ajouter comme nouvel argument la valeur **true**, sauf pour l'appel fait dans **irq_makeTasksReadyFromDate** où il faut ajouter **false**.

Fonction irq_makeTasksReadyFromDate

La fonction **irq_makeTasksReadyFromDate** (fichier **xtr.cpp**) est appelée à chaque interruption temps-réel ; si une tâche est débloquée, il faut la retirer de la liste des tâches bloquées d'un outil de synchronisation ; le champ **mBlockingList** du descripteur de tâche désigne cette liste, ou vaut **nullptr** si la tâche n'est pas bloquée sur un outil de synchronisation :

```
static void irq_makeTasksReadyFromDate (IRQ_MODE_
                                         const uint32_t inCurrentDate) {
    TaskList::Iterator iterator (MODE_ gDeadlineWaitingTaskList) ;
    TaskControlBlock * taskPtr ;
    while ((taskPtr = iterator.nextTask (MODE))) {
        if (inCurrentDate >= taskPtr->mDeadline) {
            //--- Remove task from blocking list
            if (nullptr != taskPtr->mBlockingList) {
                taskPtr->mBlockingList->removeTask (MODE_ taskPtr) ;
                taskPtr->mBlockingList = nullptr ;
            }
            //--- Remove task from deadline list
            gDeadlineWaitingTaskList.removeTask (MODE_ taskPtr) ;
            //--- Make task ready
            kernel_makeTaskReady (MODE_ taskPtr, false) ;
        }
    }
}
```

Fonction irq_makeTaskReadyFromList

La fonction **kernel_blockRunningTaskInList** (fichier **xtr.cpp**) rend prête la tâche bloquée la plus prioritaire dans la liste passée en argument. Il faut maintenant retirer la tâche de la liste des tâches bloquées sur échéance (la fonction **removeTask** n'a pas d'effet si la tâche n'est pas bloquée sur échéance) :

```
bool irq_makeTaskReadyFromList (IRQ_MODE_ TaskList * ioWaitingList) {
    TaskControlBlock * taskPtr = ioWaitingList.removeFirstTask (MODE) ;
    const bool found = taskPtr != nullptr ;
    if (found) {
        taskPtr->mBlockingList = nullptr ;
        //--- Remove from deadline list
        gDeadlineWaitingTaskList.removeTask (MODE_ taskPtr) ;
        //--- Make task ready
        kernel_makeTaskReady (MODE_ taskPtr, true) ;
    }
    return found ;
}
```

Fonction kernel_blockRunningTaskInListAndDeadline

C'est une nouvelle fonction à ajouter (fichier **xtr.cpp**) : elle effectue le blocage de la tâche appelante sur un outil de synchronisation et une échéance :

```
void kernel_blockRunningTaskInListAndDeadline (KERNEL_MODE_
                                              TaskList & ioWaitingList,
                                              const uint32_t inDeadline) {
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;
    //--- Insert in task list
    ioWaitingList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;
    gRunningTaskControlBlockPtr->mBlockingList = &ioWaitingList ;
    //--- Insert in deadline list
    gRunningTaskControlBlockPtr->mDeadline = inDeadline ;
    gDeadlineWaitingTaskList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;
    //--- Block task
    kernel_makeNoTaskRunning (MODE_)
}
```

À revoir

Ajouter aussi la déclaration du prototype de cette fonction dans **xtr.h** :

```
void kernel_blockRunningTaskInListAndDeadline (KERNEL_MODE_
                                              TaskList & ioWaitingList,
                                              const uint32_t inDeadline) ;
```

Modifications à apporter aux outils de synchronisation

Le sémafor de Dijkstra : déclaration de P_until

Pas de modification des primitives existantes.

On va ajouter la primitive **P_until**. D'abord, sa déclaration dans la classe **Semaphore** (fichier **Semaphore.h**) :

```
//$bool-service semaphore.P_until

public: bool P_until (USER_MODE_
                      const uint32_t inDeadline) asm ("semaphore.P_until") ;

public: void sys_P_until (KERNEL_MODE_
                          const uint32_t i Deadline) asm ("service.semaphore.P_until") ;
```

Deux points sont à souligner :

- une nouvelle annotation de service : `//$bool-service` ; celle-ci signifie que la primitive va renvoyée une valeur booléenne — **P_until** est déclarée renvoyant **bool** — qui est en fait la valeur du champ **mUserResult** ;
- la fonction d'implémentation du service **sys_P_until** ne renvoie pas de valeur booléenne, elle déclarée avec **void**.

Le sémafor de Dijkstra : implémentation de sys_P_until

Ajouter l'implémentation de la primitive **sys_P_until** dans le fichier **Semaphore.cpp** :

```
void Semaphore::sys_P_until (KERNEL_MODE_ const uint32_t inDeadline) {  
    const bool userResult = mValue > 0 ;  
    kernel_setUserResult (MODE_ userResult) ; // SOULD BE CALLED BEFORE TASK BLOCKING  
    if (userResult) {  
        mValue -= 1 ;  
    }else if (inDeadline > millis (MODE)) {  
        kernel_blockRunningTaskInListAndDeadline (MODE_ mWaitingTaskList, inDeadline) ;  
    }  
}
```

Rappelons le fonctionnement de la primitive **P_until** décrite au début de ce document :

- si, au moment de l'appel le sémafor est strictement positif, le sémafor est décrémenté, et la primitive retourne immédiatement avec la valeur **true** ; **[kernel_setUserResult est appelée avec la valeur true]**
- si, au moment de l'appel le sémafor est nul, et l'échéance est atteinte ou dépassée, et la primitive retourne immédiatement avec la valeur **false** ; **[kernel_setUserResult est appelée avec la valeur false]**
- sinon, la primitive est bloquante **[kernel_setUserResult est appelée avec une valeur sans importance]**, la tâche est en attente sur le sémafor et l'échéance, jusqu'à ce que :
 - ▶ la tâche est débloquée suite à l'invocation de **V** par une autre tâche **[irq_makeTaskReadyFromList est appelé, ce qui provoque l'appel de kernel_setUserResult avec la valeur true]** ; la valeur **true** est retournée ;
 - ▶ la tâche est débloquée parce que l'échéance est atteinte **[irq_makeTasksReadyFromDate est appelé, ce qui provoque l'appel de kernel_setUserResult avec la valeur false]**; la valeur **false** est retournée.

Le sémaforo de Dijkstra : appels svc engendrés

Après compilation, on examine le fichier engendré **zSOURCES/interrupt-handlers.s** (les numéros attribués aux appels **svc** peuvent être différents) :

```
semaphore.P:  
    .fnstart  
    svc #7  
    bx lr  
  
.....  
  
semaphore.P_until:  
    .fnstart  
    svc #8  
    b    get.user.result
```

Pour la primitive **P**, l'annotation `//service` engendre un appel qui se termine par un retour (instruction **bx lr**).

Pour la primitive **P_until**, l'annotation `//$bool-service` engendre un appel qui se termine par un branchement à la fonction **get.user.result**.

À revoir

Travail à faire

Imaginer un programme qui met en évidence le fonctionnement de la primitive **P_until**.

Pour ceux qui sont en avance : implémenter les primitives d'attente temporisées sur les autres outils de synchronisation présentés à l'étape précédente.

A revoir

Étape 17 — Allocation dynamique

Description de cette étape

Jusqu'à présent, l'allocation dynamique n'était pas nécessaire pour l'exécutif.

L'implémentation des commandes gardées de l'étape suivante utilise l'allocation dynamique (il est possible d'adopter une implémentation des commandes gardés sans allocation dynamique, il faut simplement allouer statiquement des tableaux de taille suffisante).

Dans la suite, on présente les principales fonctions d'allocation dynamique en C et C++, puis comment elles sont réalisées dans l'exécutif.

Allocation dynamique en C

Vu du programmeur C, l'allocation dynamique se résume à ces deux principales fonctions :

- **malloc**, pour allouer la mémoire;
- **free**, pour la libérer.

Ces fonctions sont implémentées par la librairie C. Sous Unix, les commandes **man alloc** et **man free** permettent d'afficher leur description :

```
void * malloc(size_t size);
```

The malloc() function allocates size bytes of memory and returns a pointer to the allocated memory.

```
void free(void *ptr);
```

The free() function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

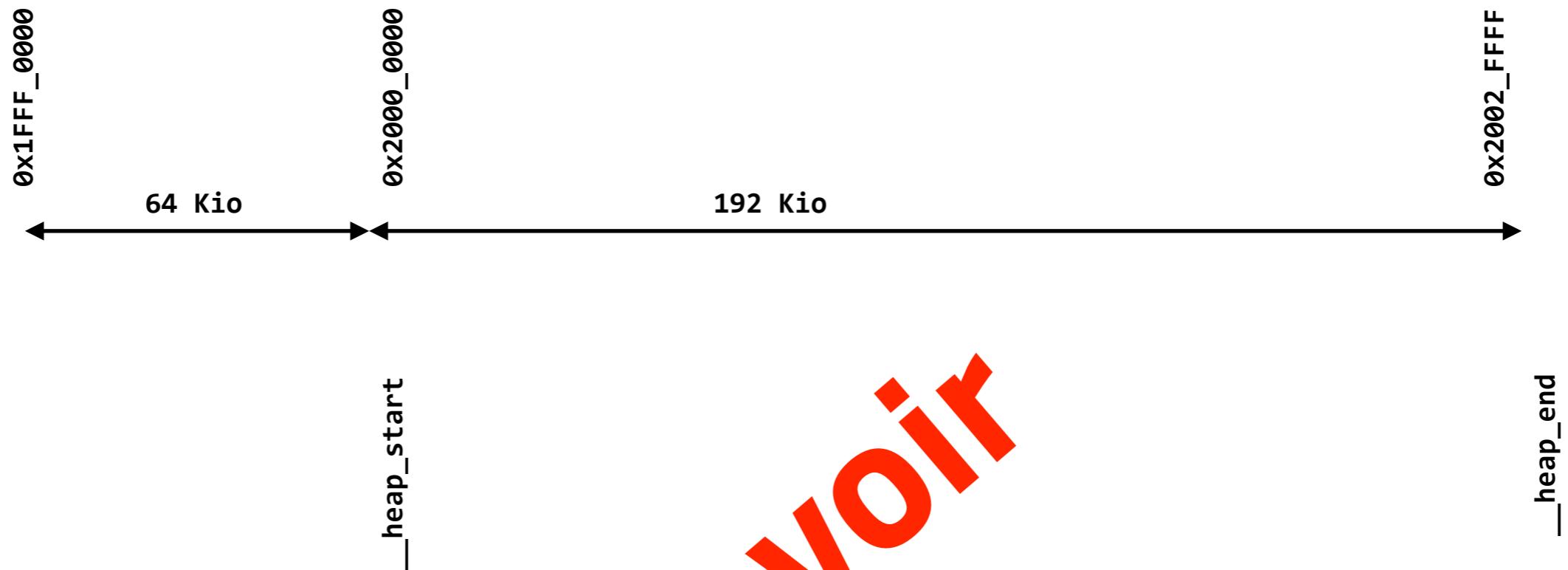
Allocation dynamique en C++

L'allocation dynamique en C++ s'exprime par les constructions **new** et **delete** :

- **new type**, pour allouer un objet ;
- **new type [taille]**, pour allouer un tableau d'objets ;
- **delete ptr**, pour libérer la mémoire associée à un objet ;
- **delete [] ptr**, pour libérer la mémoire associée à un tableau d'objets.

Ces constructions sont définies dans le langage C++ (à la différence du C), et on verra comment ce langage permet de rediriger les allocations et les libérations vers les fonctions C correspondantes.

Utilisation de la mémoire du micro-contrôleur



Le micro-contrôleur MK66FX1M0 qui équipe le module Teensy 3.6 intègre deux RAM :

- la première (64 Kio), aux adresses `0x1FFF_0000` à `0x2000_0000` ;
- la seconde (192 Kio, aux adresses `0x2000_0000` à `0x2002_FFFF`.

La première est utilisée par les variables globales et les piles des tâches.

La seconde est dédiée au **tas** (« heap »), c'est-à-dire l'espace disponible pour l'allocation dynamique. L'éditeur des liens maintient deux symboles qui délimitent le tas, `__heap_start` et `__heap_end`. Vous pouvez connaître la valeur de ces symboles en consultant pour chaque programme le fichier **zPRODUCTS/product.map**.

Gestion du tas

Il existe de nombreuses politiques de gestion de la mémoire, nous allons adopter une politique très simple, qui consiste à gérer des blocs de taille fixe :

- l'adresse retornnée par l'allocateur doit être un multiple de 8 : en effet, l'allocateur ne connaît pas la contrainte d'alignement de la donnée qui sera contenue dans le bloc, aussi il faut adopter la contrainte la plus sévère, qui est 8 pour un Cortex-M4 ;
- en fonction de l'application, on détermine une taille maximum allouable (ici 2048 octets) ;
- la taille minimum allouable doit être un multiple de 8 octets : ça simplifie grandement la vérification de la contrainte d'alignement de 8 octets.

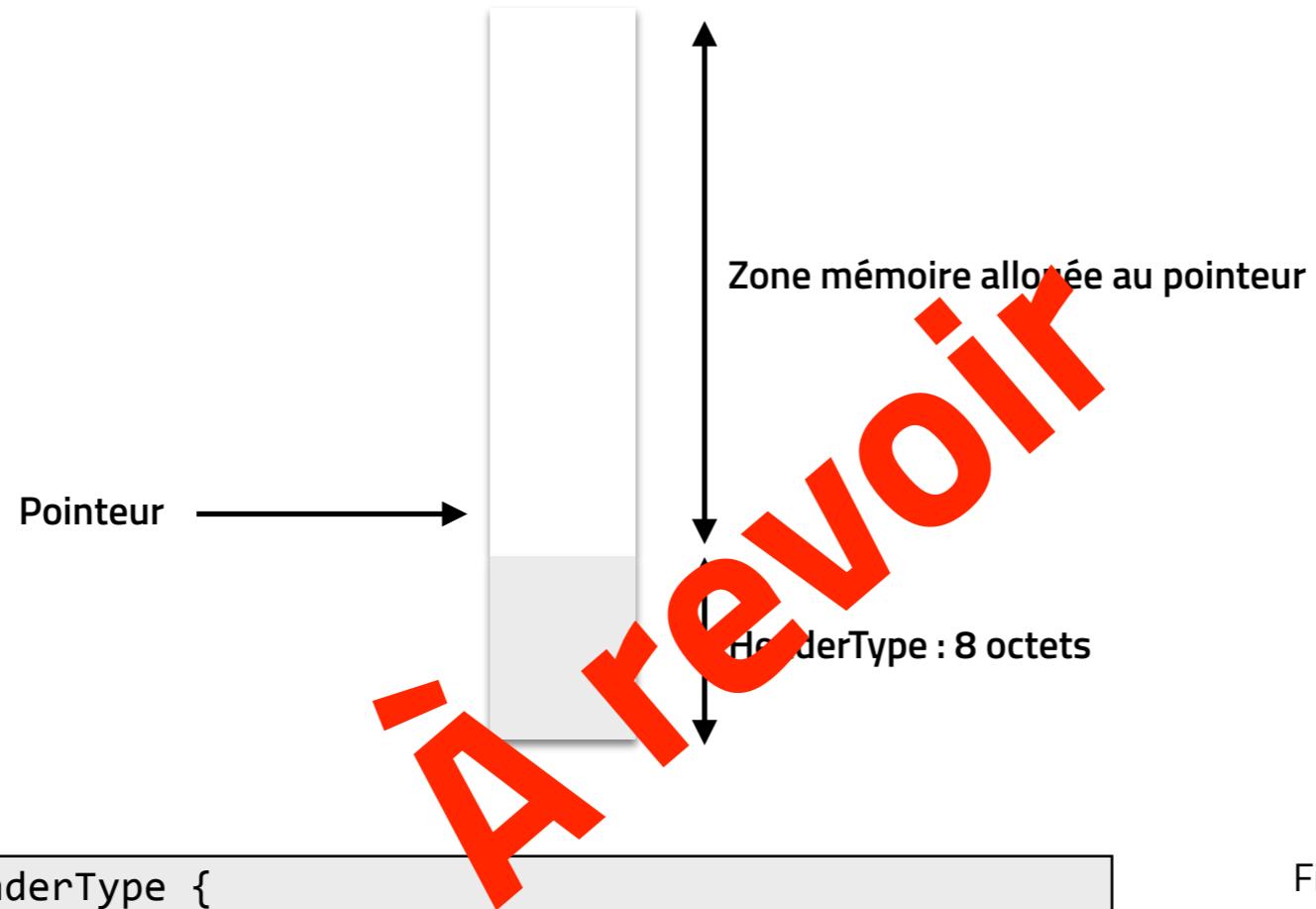
On choisit donc d'allouer des blocs de 8, 16, 32, ..., 24 octets :

- il existe une liste par taille de bloc ;
- quand un bloc est libéré, il est inseré dans la liste des blocs libres correspondant à sa taille ;
- quand on veut allouer un bloc :
 - * il est pris dans la liste des blocs libres correspondant à sa taille (en fait une *pile*, c'est le plus simple à gérer) ;
 - * si cette liste est vide, il est prélevé dans la zone libre du tas.

La technique utilisée est inspirée des « **fat pointers** » (voir page suivante) : l'idée est d'associer au pointeur des informations indiquant la taille du bloc alloué.

Structure d'un *fat pointer*

<http://libcello.org/learn/a-fat-pointer-library>: Instead we can use the ideas behind Fat Pointers, and store our runtime information in the memory space before pointers (le terme *fat pointer* a été semble-t-il introduit par les concepteurs du langage D).



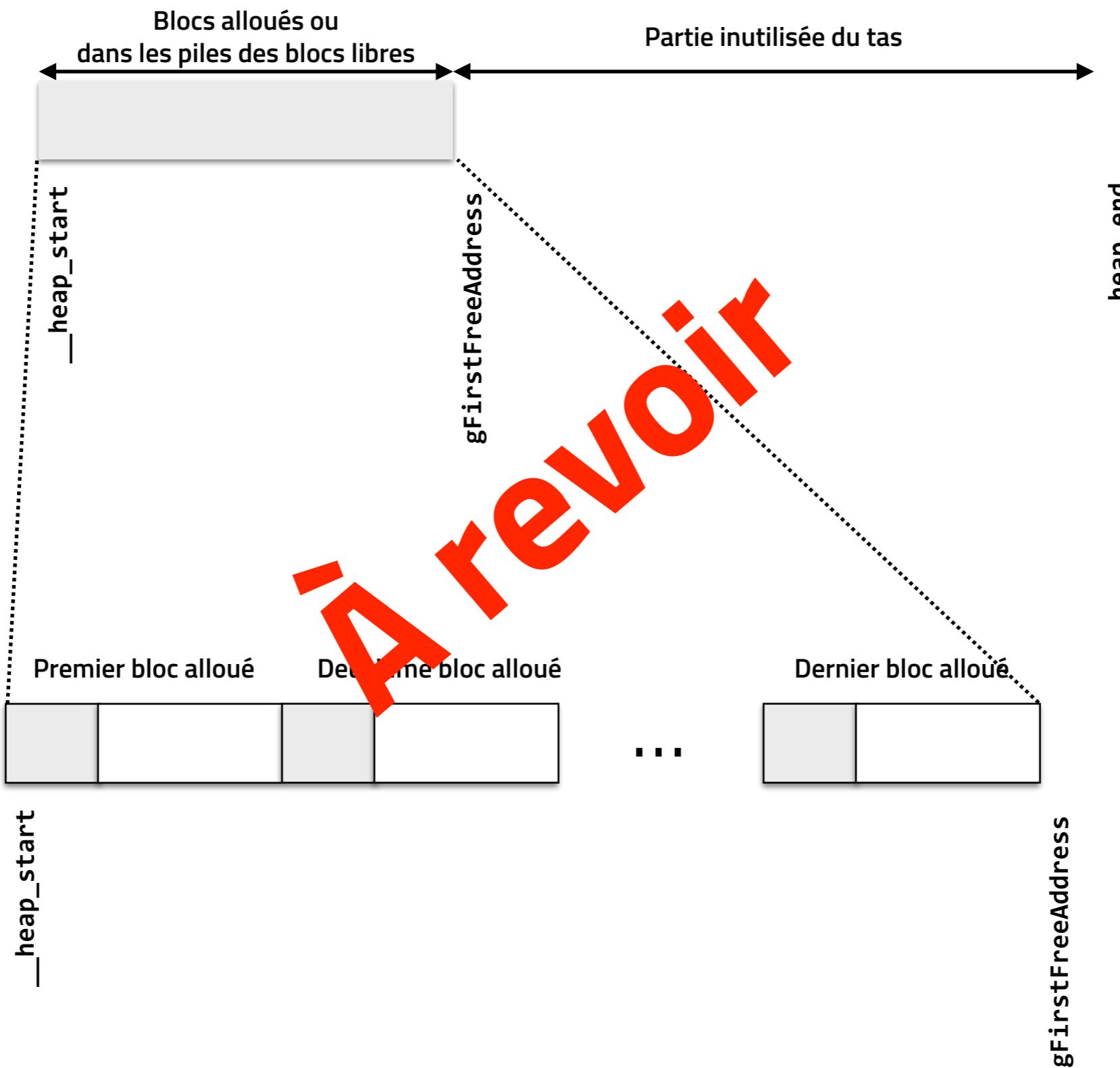
```
typedef struct HeaderType {  
    HeaderType * mNextFreeBlock ; // Used only when block is free  
    uint16_t mFreeListIndex ;  
    uint16_t mAllocatedByteSize ;  
} HeaderType ;
```

Fichier Heap.cpp

Pour accéder à l'en-tête d'un bloc :

```
HeaderType * ptr = (HeaderType *) pointeur ;  
ptr -- ;
```

Zone mémoire dédiée au tas



L'archive

L'archive **17-files_tar.bz2** contient les fichiers **heap.h** et **heap.cpp** qui implémentent les routines de gestion de la mémoire.

Dans la suite, nous allons décrire les définitions et fonctions disponibles.

A revoir

Les piles de blocs libres

À la fin du fichier **heap.h**, deux définitions fixent les tailles autorisées :

```
// Biggest block being allocated = 2 ** kMaxSizePowerOfTwo  
static const size_t kMaxSizePowerOfTwo = 10  
// Smallest block being allocated = 2 ** kMinSizePowerOfTwo (SHOULD BE >= 3)  
static const size_t kMinSizePowerOfTwo = 8;
```

C'est-à-dire que les valeurs ci-dessus permettent d'allouer des blocs de 8, 16, 32, 64, 128, 256, 512 ou 1024 octets. La demande d'allocation d'une taille supérieure renverra la valeur **nullptr**.

Allocation et concurrence

Les fonctions d'allocation et de libération peuvent être appelées en concurrence par différentes tâches et par des routines d'interruption. Il faut donc s'assurer que leur exécution soit exempt de concurrence.

Une solution simple est de les exécuter interruptions masquées, dans une *section*. Ainsi, le fichier **heap.h** déclare les prototypes suivants :

```
//$section fat.pointer.alloc
void * fatPointerAlloc (const size_t inBlockSize) asm ("fat.pointer.alloc") ;
void * section_fatPointerAlloc (SECTION_MODE_ const size_t inBlockSize)
    asm ("section.fat.pointer.alloc") ;

//$section fat.pointer.free
void fatPointerFree (void * inPointer) asm ("fat.pointer.free") ;
void section_fatPointerFree (SECTION_MODE_ void * inPointer) asm ("section.fat.pointer.free") ;
```

Remarquer que les prototypes des fonctions **fatPointerAlloc** et **fatPointerFree** n'ont pas d'annotation de mode, ce qui signifie qu'elles peuvent être appelées dans n'importe quel mode, tout en assurant que l'exécution des fonctions **section_...** s'effectuent interruptions masquées. On verra dans la suite que cela est indispensable pour prendre en charge l'allocation et la libération en C++.

Fonction section_fatPointerAlloc

```
void * section_fatPointerAlloc (SECTION_MODE_ const size_t inBlockSize) {  
    HeaderType * result = nullptr ;  
    if (inBlockSize > 0) {  
        //--- Compute smallest block with size equal to a power of two bigger or equal to required size  
        uint32_t smallestPowerOfTwo = 32 - uint32_t (__builtin_clz (inBlockSize)) ;  
        //--- Allocate if not too large  
        if (smallestPowerOfTwo <= kMaxSizePowerOfTwo) {  
            if (smallestPowerOfTwo < kMinSizePowerOfTwo) {  
                smallestPowerOfTwo = kMinSizePowerOfTwo ;  
            }  
            const uint32_t freeListIndex = smallestPowerOfTwo - kMinSizePowerOfTwo ;  
            tFreeBlockListDescriptor & descriptorPtr = gFreeBlockDescriptorArray [freeListIndex] ;  
            if (descriptorPtr.mFreeBlockCount > 0) { // Allocate from free list  
                descriptorPtr.mFreeBlockCount -- ;  
                result = descriptorPtr.mFreeBlockList ;  
                descriptorPtr.mFreeBlockList = result->mNextFreeBlock ;  
                result->mAllocatedByteSize = uint32_t (1) << smallestPowerOfTwo ;  
                result ++ ;  
                gCurrentlyAllocatedCount += 1 ;  
                gAllocationCount += 1 ;  
            }else{ // Allocate from heap  
                result = (HeaderType *) gFirstFreeAddress ;  
                gFirstFreeAddress += (uint32_t (1) < smallestPowerOfTwo) + sizeof (HeaderType) ;  
                if (gFirstFreeAddress >= size_t (& __heap_end)) {  
                    gFirstFreeAddress = size_t (& __heap_end) ;  
                    result = nullptr ;  
                }else{  
                    result->mFreeListIndex = freeListIndex ;  
                    result->mAllocatedByteSize = uint32_t (1) << smallestPowerOfTwo ;  
                    result ++ ;  
                    gAllocationCountArray [freeListIndex] += 1 ;  
                    gCurrentlyAllocatedCount += 1 ;  
                    gAllocationCount += 1 ;  
                }  
            }  
        }  
    }  
    return result ;  
}
```

Si un bloc de taille nulle est demandé, la valeur **nullptr** est retournée.

La fonction **__builtin_clz** retourne le nombre de bits consécutifs à zéros à partir du bit de poids fort.

Si un bloc de taille trop grande ($> 2^{**} \text{kMaxSizePowerOfTwo}$) est demandé, la valeur **nullptr** est retournée.

Si un bloc de taille $< 2^{**} \text{kMinSizePowerOfTwo}$ est demandé, le bloc alloué aura la taille $2^{**} \text{kMinSizePowerOfTwo}$.

La pile des blocs libres correspondant à la taille demandée n'est pas vide, on prélève un bloc dans cette pile.

La pile des blocs libres correspondant à la taille demandée est vide, on prélève un bloc dans la mémoire inutilisée. **gFirstFreeAddress** contient son adresse.

La mémoire inutilisée a une taille $<$ la taille demandée : retourner **nullptr**.

à revoir

Fonction section_fatPointerFree

```
void section_fatPointerFree (SECTION_MODE_ void * inPointer) {
    if (nullptr != inPointer) {
        HeaderType * p = (HeaderType *) inPointer ;
        p -- ;
        const uint32_t idx = p->mFreeListIndex ;
        //...
        p->mNextFreeBlock = gFreeBlockDescriptorArray [idx].mFreeBlockList ;
        gFreeBlockDescriptorArray [idx].mFreeBlockList = p ;
        gFreeBlockDescriptorArray [idx].mFreeBlockCount ++ ;
        gCurrentlyAllocatedCount -= 1 ;
    }
}
```

A revoir

Appeler avec un argument null est sans effet.

Le bloc libéré est simplement empilé dans la pile des blocs libres correspondant à son indice mémorisé dans le champ **mFreeListIndex**.

Allocation / libération en C++

Les constructions **new** et **delete** du C++ appellent des fonctions qui doivent être définies, et qui le sont le plus souvent dans la librairie standard. Ici, elles sont implémentées dans **heap.cpp**.

```
void * operator new (size_t inSize) {
    return fatPointerAlloc (inSize) ;
}

void * operator new [] (size_t inSize) {
    return fatPointerAlloc (inSize) ;
}

void operator delete (void * ptr) {
    fatPointerFree (ptr) ;
}

void operator delete [] (void * ptr) {
    fatPointerFree (ptr) ;
}
```

À revoir

Voilà pourquoi il faut pouvoir appeler **fatPointerAlloc** et **fatPointerFree** sans annotation de mode : en effet, il n'y a aucun moyen de les faire apparaître dans les prototypes des fonctions requises par le C++.

Fonctions complémentaires...

Mais ce n'est pas tout : le runtime C++ a besoin que la variable **`__dso_handle`** et que la fonction **`__cxa_pure_virtual`** soient définies.

```
void * __dso_handle ;  
  
void __cxa_pure_virtual (void) ;  
  
void __cxa_pure_virtual (void) {  
    assertionFailure (0, __FILE__, __LINE__);  
}
```

De plus, comme l'exécution ne termine jamais, les destructeurs des variables globales C++ ne sont jamais appelées. On inhibe leur génération avec l'option de compilation **`-fno-use-cxa-atexit`** (voir le fichier **`dev-files/common_definitions.py`**) : on élimine ainsi du code mort.

Cas particulier : C++17

Et en C++17, deux autres fonctions doivent être implementées. Elles sont simplement ignorées dans les versions précédentes de la norme du langage.

```
void operator delete (void * ptr, unsigned int) {
    fatPointerFree (ptr) ;
}

void operator delete [] (void * ptr, unsigned int) {
    fatPointerFree (ptr) ;
}
```

A revoir

Travail à faire

Imaginer un programme permettant de tester la gestion mémoire de façon intensive :

- la tâche de plus faible priorité boucle sur des allocations / libérations ;
- une autre tâche périodique effectue aussi des allocations / libérations ;
- afficher les statistiques d'allocations / libérations (voir le fichier **heap.h**) ;
- quand on appuie sur ~~un bouton~~ poussoir, les allocations cessent, les libérations s'effectuent : on doit aboutir à un nombre nul d'objets alloués.

Étape 18 — Commandes gardées

Description de cette étape

L'attente temporisée permet d'exprimer une attente sur le premier événement qui se déclenche, soit le passage par une primitive pouvant être bloquante, soit lorsqu'une date est atteinte.

Les commandes gardées sont une généralisation qui permet d'exprimer une attente sur des événements quelconques.

Imaginées par Dijkstra, les commandes gardées ont été étendues par Hoare pour le langage *Communicating Sequential Processes* avec trois commandes de communication basées sur le rendez-vous. Dans la suite, nous décrivons comment toute commande susceptible d'être bloquante peut apparaître en garde (exemple typique : la primitive **P** d'un sémaphore).

Liens :

https://en.wikipedia.org/wiki/Guarded_Command_Language

https://en.wikipedia.org/wiki/Communicating_sequential_processes

Les commandes gardées de Dijkstra

Commande gardée. Une commande gardée a pour syntaxe :

expression booléenne -> Liste d'instructions

L'expression booléenne est appelée *garde*. La liste d'instructions qui suit est dite *associée à cette garde*.

État d'une garde. À un moment donné de l'exécution, une garde est fausse ou vraie, selon la valeur de son expression booléenne.

Commande répétitive. Une *commande répétitive* est constituée d'une ou plusieurs commandes gardées.

Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis la commande répétitive est exécutée de nouveau ;
- si toutes les gardes sont fausses, la commande répétitive est terminée, l'exécution se poursuit par celle de l'instruction suivante.

Voici un exemple de commande répétitive, qui trie x_1 , x_2 et x_3 par ordre croissant :

```
*[ x1 > x2 -> x1, x2 := x2, x1 // Échange x1, x2  
| x2 > x3 -> x2, x3 := x3, x2 // change x2, x3  
]
```

Commande alternative. Une *commande alternative* est constituée d'une ou plusieurs commandes gardées.

Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis l'exécution se poursuit par celle de l'instruction qui suit la commande alternative ;
- le cas où toutes les gardes sont fausses est considéré comme une erreur d'exécution.

Garde avec une primitive de synchronisation

Garde avec une primitive de synchronisation. La syntaxe d'une garde est étendue, en acceptant la présence d'une primitive de synchronisation susceptible d'être bloquante :

expression booléenne ; synchro -> liste d'instructions

État d'une garde. À un moment donné de l'exécution :

- si l'expression booléenne est fausse, la garde est **fausse**;
- si l'expression booléenne est vraie :
 - ▶ si la primitive de synchronisation n'est pas bloquante, la garde **vraie** ;
 - ▶ si la primitive de synchronisation est bloquante, la garde est **neutre**.

Commande répétitive. Une *commande répétitive* est constituée d'une ou plusieurs commandes gardées.

Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis la commande répétitive est exécutée de nouveau ;
- si toutes les gardes sont fausses, la commande répétitive est terminée, l'exécution se poursuit par celle de l'instruction suivante ;
- si aucune garde n'est vraie, et il y a une ou plusieurs gardes neutres, l'exécution est suspendue en attendant qu'une ou plusieurs gardes deviennent vraies, ou qu'elles deviennent toutes fausses.

Commande alternative. Son fonctionnement est étendu de façon analogue.

Résumé : les trois formes d'une garde

Garde booléenne pure.

expression_booléenne -> liste d'instructions

Garde constituée d'une expression booléenne et une primitive de synchronisation.

expression_booléenne ; synchro -> liste d'instructions

Garde constituée d'une primitive de synchronisation.

synchro -> liste d'instructions

Cette forme est sémantiquement équivalente à la deuxième forme dans laquelle l'expression booléenne est toujours vraie.

L'expression booléenne ne peut être constituée que de variables privées de la tâche. Il y a plusieurs raisons à cela :

- cette règle ne restreint pas l'expressivité des commandes gardées, et en facilite même la compréhension, puisque les gardes ne peuvent évoluer que par l'action de la tâche courante ;
- implémenter les commandes gardées avec des variables susceptibles d'être modifiées par d'autres tâches est très compliqué (voir les implémentations de Ada...)

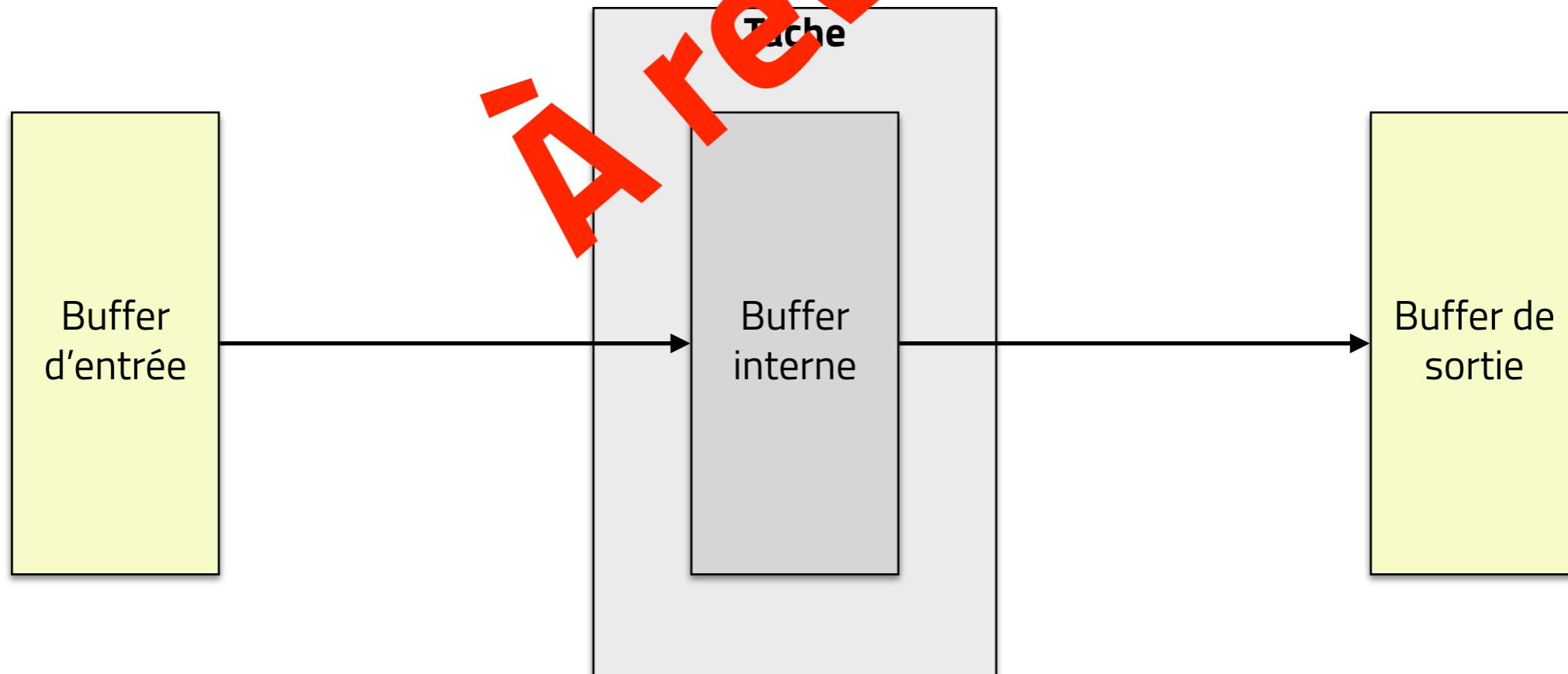
Un exemple

Une tâche reçoit des données en provenance d'un *buffer d'entrée*, les stocke dans son *buffer interne*, pour les retransmettre dans le *buffer de sortie*.

Les buffers d'entrée et de sortie sont des outils de synchronisation, le buffer interne est une simple structure de données.

Ainsi, la tâche réalise répétitivement :

- si le buffer interne n'est pas plein et que le buffer d'entrée contient une donnée, effectuer le transfert ;
- si le buffer interne n'est pas vide et que le buffer de sortie l'est pas plein, effectuer le transfert.



Écriture des commandes gardées en C++

Garde booléenne pure.

```
if (guard_booleanExpression (MODE_ expression_booléenne)) { Liste d'instructions }
```

Une garde booléenne pure a un effet de bord, aussi il faut appeler la fonction dédiée **guard_booleanExpression** qui fait les opérations nécessaires.

Garde constituée d'une expression booléenne et une primitive de synchronisation.

```
if (expression_booléenne && outil.guarded_op (MODE)) { Liste d'instructions }
```

Attention, il faut toujours utiliser l'opérateur **&&**, qui n'évalue pas l'expression de droite si celle de gauche est vraie.

Garde constituée d'une primitive de synchronisation.

```
if (outil.guarded_op (MODE)) { Liste d'instructions }
```

État d'une garde. L'expression d'un **if** ne pouvant prendre que deux valeurs (**true** ou **false**), on peut pas rendre compte des trois états possibles (*faux, neutre ou vrai*) d'une garde. Aussi :

- `outil.guarded_op(MODE)` renvoie le booléen **true** si la commande gardée est *vraie* ;
- `outil.guarded_op(MODE)` renvoie le booléen **false** si la commande gardée est *neutre* ;
- l'expression d'un **if** est **true** si la garde est vraie, et **false** si elle est *neutre ou fausse*.

Écriture d'une commande répétitive en C++

```
bool loop = true ;  
while (loop) {  
    if (bufferInterneNonPlein && bufferEntrée.guarded_out (MODE_ data)) {  
        Entrer data dans le buffer interne  
    }else if (bufferInterneNonVide && bufferSortie.guarded_in (MODE_ dataBufferInterne)) {  
        Retirer La donnée envoyée du buffer interne  
    }else{  
        loop = guard_waitForChange (MODE) ;  
    }  
}
```

Noter que l'ordre d'examen des commandes gardées est deterministic et figé.

Si une garde est *vraie*, la liste d'instructions associée est exécutée, et la commande répétitive est exécutée de nouveau.

L'exécution parvient à la branche **else** si toutes les expressions booléennes sont toutes **false**, ce qui correspond à des gardes *neutres* ou *fausses*. La primitive **guard_waitForChange** exploite les effets de bord de l'appel aux primitives **guarded_xxx** :

- si toutes les gardes sont *fausses*, la primitive n'est pas bloquante et renvoie **false** (ce qui fait quitter la commande répétitive) ;
- sinon la primitive est bloquante, et sera débloquée en renvoyant **true** dès que l'état d'une primitive **guarded_xxx** a changé.

À revoir

Écriture d'une commande alternative en C++

```
bool loop = true ;  
while (loop) {  
    if (une_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else if (autre_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else{  
        loop = guard_waitForChange (MODE) ;  
        if (!loop) {  
            Erreur, toutes les gardes sont fausses.  
        }  
    }  
}
```

À revoir

Le code ci-dessus fait apparaître deux modifications par rapport à une commande répétitive :

- après chaque *liste d'instructions associée*, le booléen **loop** est mis à **false** pour sortir de la boucle ;
- évaluer toutes les gardes à l'état *faux* constitue une erreur.

Attente temporelle en garde : guard_waitUntil

```
bool guard_waitUntil (USER_MODE_ const uint32_t inDeadlineMS) ;
```

Elle renvoie :

- **true** si l'échéance est atteinte (garde *vraie*) ;
- **false** si l'échéance n'est pas atteinte (garde *neutre*).

La primitive **guard_waitUntil** est une commande de synchronisation particulière, elle peut être précédée d'une expression booléenne.

Noter que l'argument est une date et non pas un délai. En effet, utiliser un délai provoquerait des comportements imprévisibles (voir l'exemple de la page suivante).

Nouvelle écriture de P_until

La primitive **P_until** peut-être maintenant vue comme un cas particulier : c'est une commande alternative constituée de deux gardes :

```
bool loop = true ;
bool response = false ;
while (loop) {
    if (sémaphore.guarded_P (MODE)) {
        loop = false ;
        response = true ;
    }else if (guard_waitUntil (MODE_ échéance)) {
        loop = false ;
    }else{
        guard_waitForChange (MODE) ; /* Il n'est pas utile de tester la valeur renvoyée
    }
}
```

A revoir

Pourquoi il n'existe pas d'attente de délai en garde. Les gardes peuvent être évaluées un nombre imprévisible de fois, à des dates imprévisibles : l'échéance effective serait recalculée à chaque fois et serait « fuyante ».

Écriture d'une primitive apparaissant en garde

```
bool loop = true ;  
while (loop) {  
    if (une_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else if (autre_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else{  
        loop = guard_waitForChange (MODE) ;  
        if (!loop) {  
            Erreur, toutes les gardes sont fausses  
        }  
    }  
}
```

À revoir

Ajouter la prise en compte des commandes gardées dans un outil de synchronisation nécessite l'utilisation d'une classe complémentaire et de deux fonctions.

La classe GuardList.

Exemple : le sémaforo et guarded_P

```
bool loop = true ;  
while (loop) {  
    if (une_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else if (autre_commande_gardée) {  
        liste d'instructions  
        loop = false ;  
    }else{  
        loop = guard_waitForChange (MODE) ;  
        if (!loop) {  
            Erreur, toutes les gardes sont fausses.  
        }  
    }  
}
```

À revoir

Le code ci-dessus fait apparaître deux modifications par rapport à une commande répétitive :

- après chaque *liste d'instructions associée*, le booléen *loop* est mis à **false** pour sortir de la boucle ;
- évaluer toutes les gardes à l'état *faux* constitue une erreur.

Travail à faire

Dupliquer le projet de l'étape précédente et renommez-le **18-guarded-commands**.

Il y a trop de modifications à faire pour les présenter simplement. Prendre l'archive **18-files.tar.bz2** qui contient les nouvelles versions des fichiers :

- **task-list--32-tasks.h** et **task-list--32-tasks.cpp** ;
- **time.h** et **time.cpp** ;
- **xtr.h** et **xtr.cpp**.

Expérimenter les commandes gardées en mettant en œuvre une chaîne de transmission de données à travers plusieurs tâches.

Les données sont produites par une routine d'interruption temps-réel. Elle insère des données dans un premier buffer si il n'est pas plein. Un compteur des données ainsi insérées est affiché. Si on appuie sur une touche, l'insertion est inhibée.

À l'autre bout de la chaîne, le nombre de données récupérées est affiché. Appuyer sur une autre touche inhibe la récupération des données.

À revoir

Étape 19

Réseau CAN (attentes actives)

Description de cette étape

Dans cette étape, nous allons utiliser les deux interfaces CAN intégrées dans le microcontrôleur.

L'émission et la réception seront implémentées sous la forme d'attentes actives. Dans l'étape suivante, nous remplacerons ces attentes actives par des attentes passives.

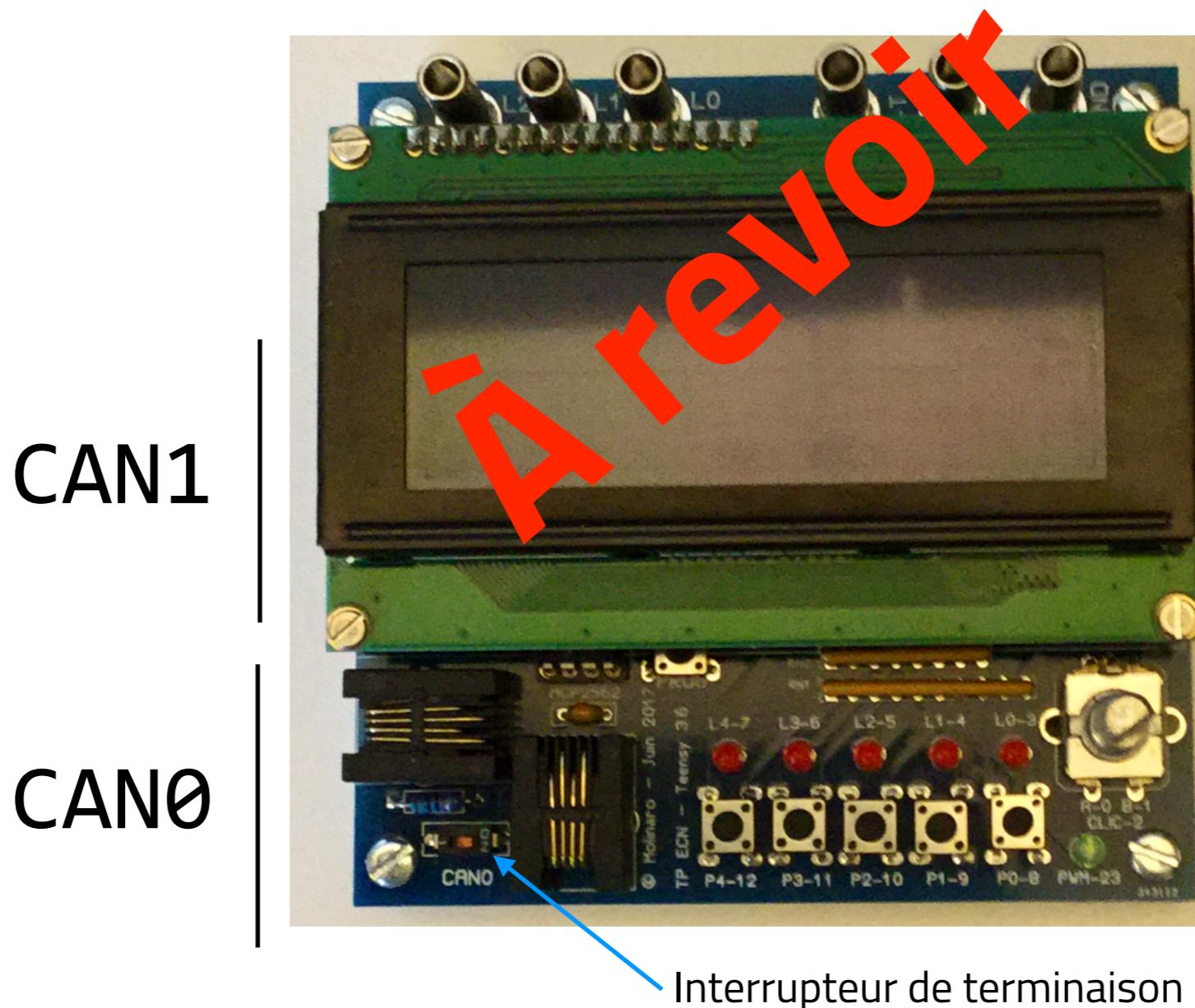
La description de cette étape commence par présenter la connectique CAN de la carte, puis les informations nécessaires pour comprendre l'émission et la réception de messages.

Les interfaces CAN de la carte de TP

La carte de TP possède deux interfaces CAN indépendantes, **CAN0** et **CAN1**.

Chacune présente :

- deux connecteurs identiques ;
- un interrupteur de terminaison (voir son usage page suivante).



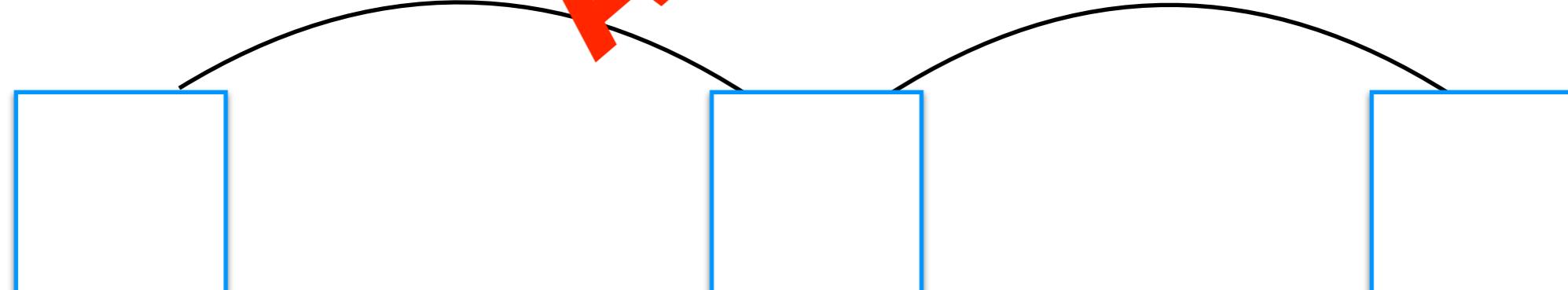
Comment réaliser un réseau CAN

Au moins deux nœuds. Un nœud CAN seul se bloque sur l'émission d'une trame CAN. Il faut au moins une autre nœud. Vous pouvez donc par exemple :

- faire un réseau en connectant les deux interfaces d'une même carte ;
- connecter plusieurs interfaces de plusieurs cartes.

Un réseau CAN est un bus, vous ne devez pas faire un réseau en étoile. Ceci signifie que vous devez réaliser un réseau linéaire, avec deux extrémités :

- à chaque extrémité, la terminaison doit être activée (interrupteur sur *ON*) ;
- sur les nœuds intermédiaires, la terminaison ne doit pas être activée (interrupteur sur *OFF*).



Interrupteur de terminaison activé

Interrupteur de terminaison non activé

Interrupteur de terminaison activé

Interface CAN en mode *Loopback*

Mode loopback. Pour faire des essais, vous pouvez programmer une interface CAN en mode *loopback*, ce qui signifie :

- aucune connexion sur les connecteurs de l'interface
- toute trame envoyée est également reçue.

Ce mode est utile pour appréhender la programmation CAN : vous pouvez faire les essais sans câble de connexion et avec une seule carte.

Ce qu'il faut savoir pour utiliser un réseau CAN

Tous les nœuds CAN d'un même réseau doivent être programmés à la même vitesse. Il y a un grand nombre de vitesses possibles, toutefois je vous recommande une de ces valeurs suivantes : 125 kbit/s, 250 kbit/s, 500 kbit/s, 800 kbit/s ou 1Mbit/s.

Les informations d'une trame CAN :

- une trame CAN est au format *standard* ou *étendu* ;
- l'identificateur est un nombre non signé :
 - ▶ sur 11 bits (trame *standard*) ;
 - ▶ sur 29 bits (trame *étendue*).
- une trame CAN peut contenir entre 0 et 8 octets de données.

Note. Il existe aussi des trames de requête, qui ne sont pas prises en compte par le pilote.

Fondamental : deux nœuds CAN différents ne doivent pas émettre des trames de même format et de même identificateur. Le même identificateur est accepté, sous réserve que l'un des nœuds émet une trame standard, et l'autre une trame étendue. Ne pas respecter cette règle bloque le réseau.

Fichiers sur le serveur pédagogique

L'archive **19-files.tar.gz** contient les fichiers qui implémentent le pilote CAN avec des attentes actives. Dupliquer l'étape 18, renommez le répertoire obtenu en **19-can-network--active-send-receive** et insérez dans le sous répertoire **sources** les fichiers obtenus.

L'archive contient 5 fichiers :

- le pilote **can-driver.h** et **can-driver.cpp** ;
- le paramétrage du pilote **can-settings.h** et **can-settings.cpp** ;
- la description d'un message CAN **CANMessage.h**.

La description d'un message CAN

La classe **CANMessage** et le type énuméré **tFrameFormat** sont déclarés dans le fichier **CANMessage.h** :

```
typedef enum {kStandard, kExtended} tFrameFormat ;  
  
class CANMessage {  
public: uint32_t mIdentifier = 0 ; // Frame identifier  
public: tFrameFormat mFormat = kStandard ;  
public: uint8_t mLength = 0 ; // Length of data (0 ..)  
public: union {  
    uint64_t mData64 ; // Caution: subject toendianness  
    uint32_t mData32 [2] ; // Caution: subject toendianness  
    uint16_t mData16 [4] ; // Caution: subject toendianness  
    uint8_t mData [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;  
} ;  
} ;
```

À revoir

Toutes les propriétés sont initialisées par défaut : l'objet obtenu représente une trame standard sans donnée, et d'identificateur égal à 0.

Configuration du pilote CAN

La configuration du pilote CAN doit être réalisé en mode **INIT**, vous pouvez le faire dans la fonction qui crée les tâches de votre fichier **user-tasks.cpp** :

```
static void initApplication (INIT_MODE) {
    ACANSettings settings (MODE_ 125 * 1000) ;
    settings.mLoopBack = true ; // Uniquement si vous voulez le mode Loopback
    uint32_t errorCode = ACAN::can0.begin (MODE_ settings) ;
    assertion (errorCode == 0, errorCode, __FILE__, __LINE__) ;
    errorCode = ACAN::can1.begin (MODE_ settings) ;
    assertion (errorCode == 0, errorCode, __FILE__, __LINE__) ;
//--- Déclaration des tâches
    kernel_createTask (MODE_ gStack0, sizeof (gStack0), code0) ;
    ...
}
```

Le constructeur de **ACANSettings** a pour second argument la vitesse du bus : ici, 125 000 signifie 125 kbit/s.

Le pilote de l'interface **CAN0** est **ACAN::can0**, et celui de **CAN1** est **ACAN::can1**. La fonction **begin** effectue la configuration de l'interface, et ne modifie pas son argument **settings**, si bien que vous pouvez l'utiliser pour configurer les deux interfaces. Elle renvoie un code d'erreur — qui vaut 0 si la configuration s'est faite sans erreur. Il est prudent de vérifier que la valeur 0 est effectivement renvoyée.

Si vous voulez utiliser que l'une des interfaces, il est inutile de configurer l'autre.

Envoi de messages CAN

Quand une interface CAN est configurée, la fonction `tryToSend` envoie un message CAN (appeler `ACAN::can0.tryToSend` pour **CAN0** et `ACAN::can1.tryToSend` pour **CAN1**). Cette fonction doit être appelée en mode **USER** et renvoie un booléen qui indique si le message a été inséré avec succès dans le buffer d'émission. Elle n'est jamais bloquante. Voici une utilisation typique :

```
static void taskCode (USER_MODE) {  
    ...  
    CANMessage message ; // Standard frame, identifier equal to no data  
    // ... set identifier, format, data  
    if (ACAN::can0.tryToSend (MODE_ message)) {  
        // Message has been successfully buffered  
    }else{  
        // Buffer was full: retry later...  
    }  
}
```

Attention, la réponse **true** ne signifie pas que le message a été envoyé, elle signifie l'insertion dans le buffer d'émission. En fonction de l'occupation du bus, les messages en attente sont envoyés dès que possible.

Réception de messages CAN

Quand une interface CAN est configurée, la fonction `receive` permet de récupérer un message reçu (appeler `ACAN::can0.receive` pour **CAN0** et `ACAN::can1.receive` pour **CAN1**). Cette fonction doit être appelée en mode **USER** et renvoie un booléen qui indique si un message a été effectivement reçu. Elle n'est jamais bloquante. Voici une utilisation typique :

```
static void taskCode (USER_MODE) {  
    ...  
    CANMessage message ; // Standard frame, identifier equal to 0, no data  
    if (ACAN::can0.receive (MODE_ message)) {  
        // Message has been received and assigned to message  
    }else{  
        // No available message  
    }  
}
```

Lors de l'appel de **receive**, le contenu de l'argument `message` est sans importance : en cas de réception (réponse **true**), son contenu est écrasé par les valeurs de la trame reçue, en cas de non réception (réponse **false**), son contenu est inchangé.

Premier programme — en mode *loopback*

Vous avez maintenant toutes les informations pour écrire des programmes utilisant les interfaces CAN. Dans toute cette étape, on ne se préoccupe pas de l'occupation processeur — les fonctions d'émission et de réception imposent des attentes actives.

Voici la description du premier programme :

- utiliser uniquement **CAN0** (pour commencer) ;
- configurer **CAN0** à la vitesse que vous voulez, et en mode loopback (aucune connexion à faire) ;
- toutes les secondes, envoyer un message sur **CAN0** (en cas de succès, incrémenter un compteur d'émission et l'afficher) ;
- interroger **CAN0** pour recevoir une réponse (en cas de succès, incrémenter un compteur de réception et l'afficher).

Normalement, compteurs d'émission et de réception doivent toujours être égaux et s'incrémenter toutes les secondes.

À revoir

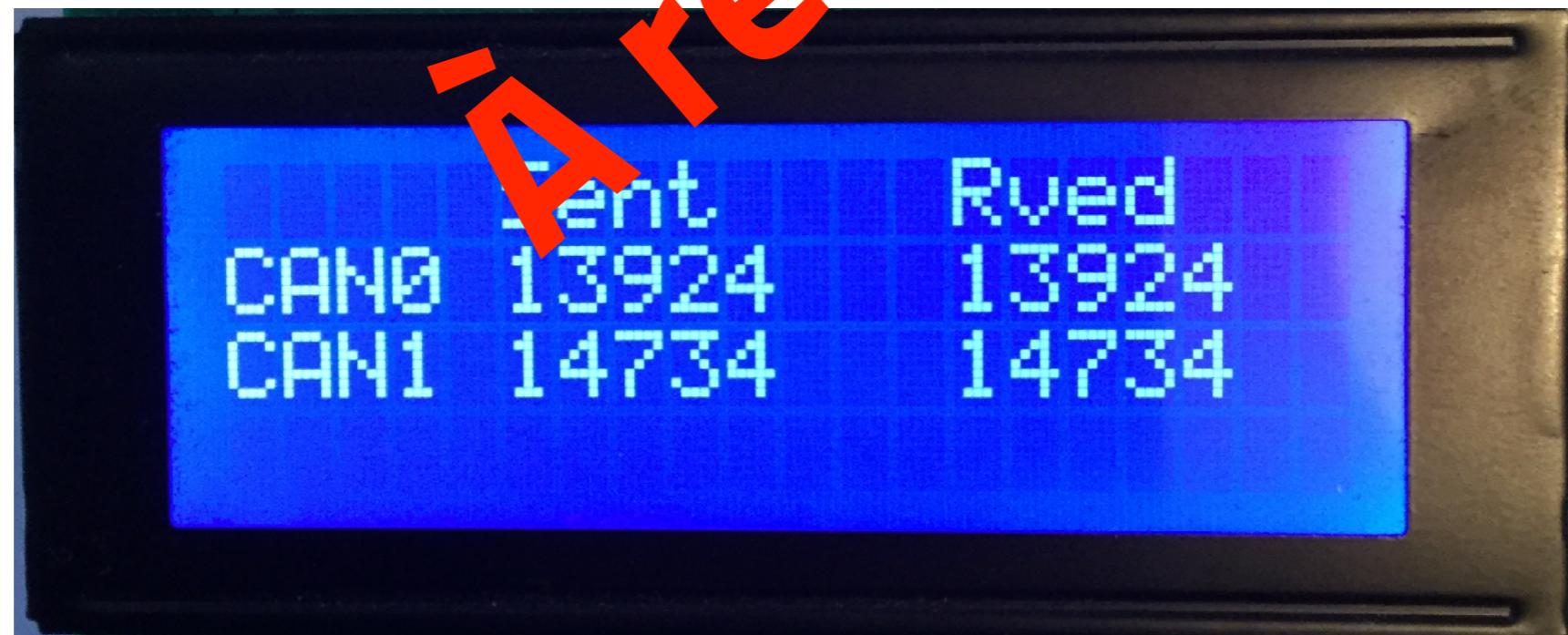
Deuxième programme — en mode *loopback*

Pour ce deuxième programme, configurez et utilisez **CAN1** pour émettre et recevoir en mode *loopback* — vous n'êtes pas obligé d'adopter la même vitesse pour les deux interfaces.

Modifiez le programme de façon à envoyer et recevoir aussi rapidement que possible, c'est-à-dire que la boucle sans fin de votre tâche appelle sans aucune attente les fonctions d'émission et de réception des deux interfaces. Toutes les secondes, affichez le nombre de trames émises et reçues pour chaque interface.

Suggestion : programmer la prise en compte d'un poussoir, dont l'appui inhibe l'émission des trames : pour chaque interface, les compteurs d'émission et de réception doivent être égaux.

Voici l'affichage que l'on peut obtenir :



Troisième programme — en réseau (1/2)

On va maintenant réaliser un réseau CAN entre les deux interfaces d'une même carte. Attention, rappelez-vous :

- tous les nœuds CAN d'un même réseau doivent être programmés à la même vitesse ;
- deux nœuds CAN différents ne doivent pas émettre des trames de même format et de même identificateur.

Donc dans votre programme veillez à ce que `CAN0` et `CAN1` n'envoient pas de trames de même format et de même identificateur.

Commentez ou ôtez la ligne `setting.mLoopback = true` ; qui programme les interfaces en mode *loopback*.

Sur votre carte, vérifiez que les deux interrupteurs de terminaison sont sur *ON*.

Aller à la page suivante pour découvrir les opérations à réaliser.

À revoir

Troisième programme — en réseau (2/2)

Premier essai. Ne reliez pas les interfaces CAN par un câble et regardez ce qui est affiché : les compteurs d'émission se stabilisent à 17 et les compteurs de réception restent à 0. Pourquoi ?

Une interface CAN qui n'est pas en mode *loopback* ne reçoit pas les trames qu'elle émet.

Pour l'émission, un contrôleur CAN exige qu'il y ait au moins un récepteur pour acquitter l'émission, ce qui n'est pas le cas ici. Le contrôleur CAN va re-émettre la première trame tant qu'il n'y a pas d'acquittement. Ceci bloque la file d'attente d'émission, qui accepte de nouvelles trames tant qu'elle n'est pas pleine. La taille de la file d'attente du pilote est de 16 (en fait, valeur de la constante `TRANSMIT_BUFFER_SIZE` déclarée dans `can-driver.h`) et il faut ajouter une unité pour le buffer d'émission matériel.

Second essai. Ne redémarrez pas la carte, et reliez les interfaces par un câble : maintenant, les compteurs d'émission et de réception s'incrémentent.

Étape 20

Réseau CAN (émission bloquante)

Description de cette étape

Dans l'étape précédente, l'émission et la réception sont implémentées sous la forme d'attentes actives.

Dans cette étape, nous mettons en œuvre une émission bloquante.

La description de cette étape commence par présenter comment la commande d'émission `tryToSend` était implémentée dans l'étape précédente.

Ensuite, on implémentera la commande bloquante à l'aide d'un sémaphore.

Duplicer l'étape précédente, et renommer le répertoire obtenu **20-can-network--blocking-send**.

Comment tryToSend est implémenté (1/4)

Ouvrez le fichier **can-driver.h** et repérez les lignes suivantes :

```
//----- Transmitting messages

//$section try.to.send
public: bool tryToSend (USER_MODE_ const CANMessage & inMessage) asm ("try.to.send") ;
private: bool section_tryToSend (SECTION_MODE_ const CANMessage & inMessage) asm ("section.try.to.send") ;

//--- Driver transmit buffer
private: CANMessage mTransmitBuffer [TRANSMIT_BUFFER_SIZE] ;
private: uint32_t mTransmitBufferReadIndex ; // 0 ... TRANSMIT_BUFFER_SIZE-1
private: uint32_t mTransmitBufferCount ; // 0 ... TRANSMIT_BUFFER_SIZE

//--- Internal send method
private: void writeTxRegisters (SECTION_MODE_ const CANMessage & inMessage, const uint32_t inMBIndex) ;
```

À revoir

On distingue :

- la fonction `tryToSend` est implémentée en mode **SECTION** par `section_tryToSend` ; ainsi, `section_tryToSend` ne peut pas être interrompue ;
- le buffer de transmission du pilote, avec les deux propriétés (initialisées à 0 dans le constructeur) qui contiennent l'indice de lecture et le nombre d'éléments dans le buffer ;
- enfin, la fonction `writeTxRegisters` sur laquelle vous n'aurez pas à intervenir.

Comment tryToSend est implémenté (2/4)

Ouvrez le fichier **can-driver.cpp** et l'implémentation de la fonction **section_tryToSend** :

```
bool ACAN::section_tryToSend (SECTION_MODE_ const CANMessage & inMessage) {
    bool sent = false ;
    const uint32_t TxMailBoxIndex = 15 ;
    const uint32_t status = FLEXCAN_get_code (FLEXCANb_MBn_CS (mFlexcanBaseAddress, TxMailBoxIndex)) ;
    if (status == FLEXCAN_MB_CODE_TX_INACTIVE) { //--- Write directly to send mailbox if inactive
        writeTxRegisters (MODE_ inMessage, TxMailBoxIndex);
        sent = true ;
    }else if (mTransmitBufferCount < TRANSMIT_BUFFER_SIZE) { // Not sent and buffer not full ?
        //--- Append message to buffer
        const uint32_t transmitBufferWriteIndex
            = (mTransmitBufferReadIndex + mTransmitBufferCount) % TRANSMIT_BUFFER_SIZE ;
        mTransmitBuffer [transmitBufferWriteIndex] = inMessage ;
        mTransmitBufferCount += 1 ;
        sent = true ;
    }
    //---
    return sent ;
}
```

A revoir

Comment tryToSend est implémenté (3/4)

Pour comprendre l'implémentation de la fonction `section_tryToSend`, il faut avoir quelques notions sur les modules FlexCAN implantés dans le silicium du micro-contrôleur.

Chaque module (*FlexCAN0*, *FlexCAN1*) implémente 16 *mailboxes*. Une *mailbox* peut être configurée pour envoyer ou recevoir des messages. Le pilote utilise la *mailbox* n° 15 pour envoyer des messages. La *mailbox* n° 15 peut être dans l'un des deux états suivants :

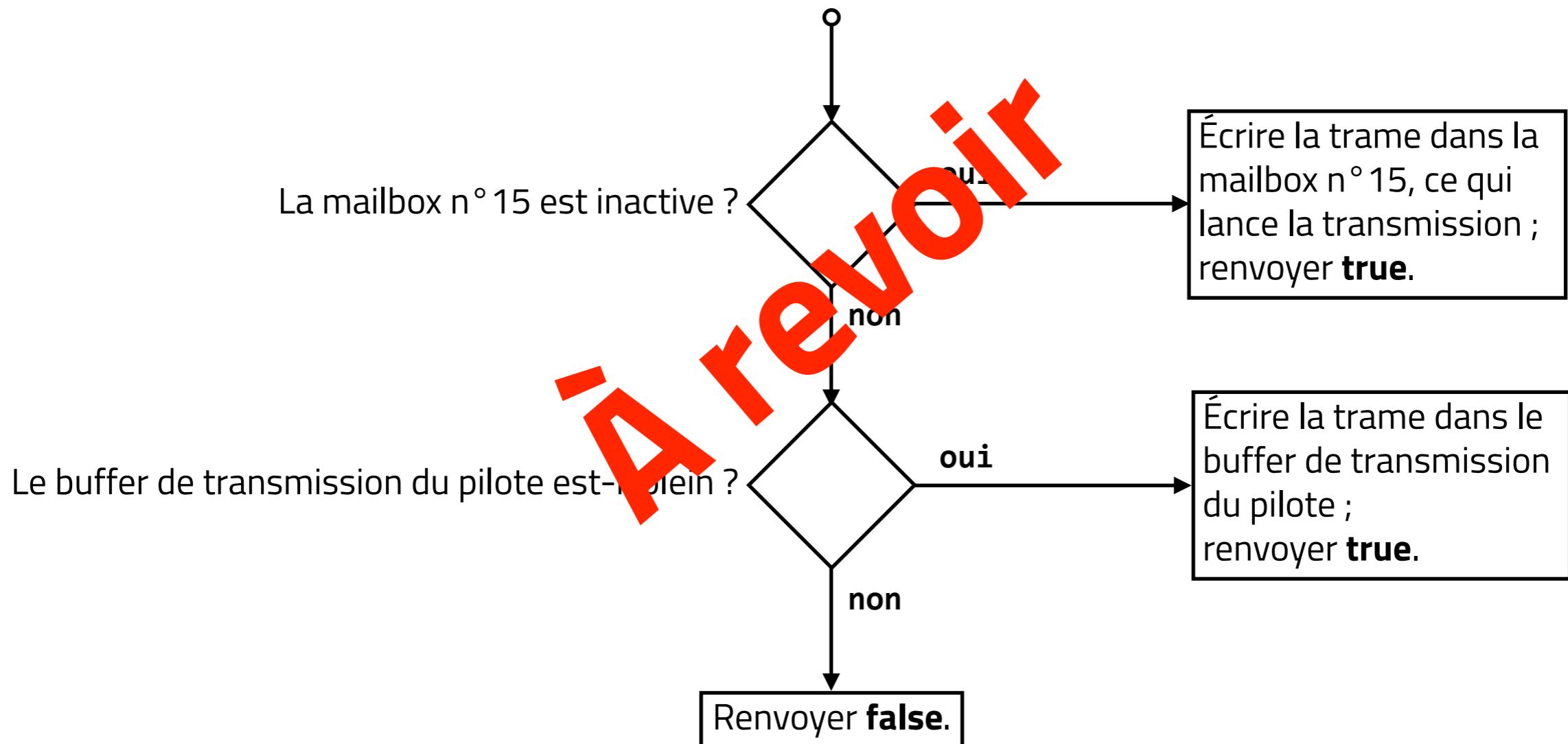
- *inactive*, elle n'effectue aucune action ;
- *active*, elle contient un message en cours d'émission.

Appeler la fonction `writeTxRegisters` écrit le message à transmettre dans la *mailbox* et la fait passer de l'état *inactive* à *active*.

Quand le message sera effectivement émis sur le réseau (c'est-à-dire acquitté par au moins un récepteur), la *mailbox* passera de l'état *active* en *inactive*. Ce changement d'état provoque le déclenchement de l'interruption liée au module CAN.

Comment tryToSend est implémenté (4/4)

On peut donc donner l'organigramme de la fonction `section_tryToSend` :



La routine d'interruption ACAN::message_isr (1/2)

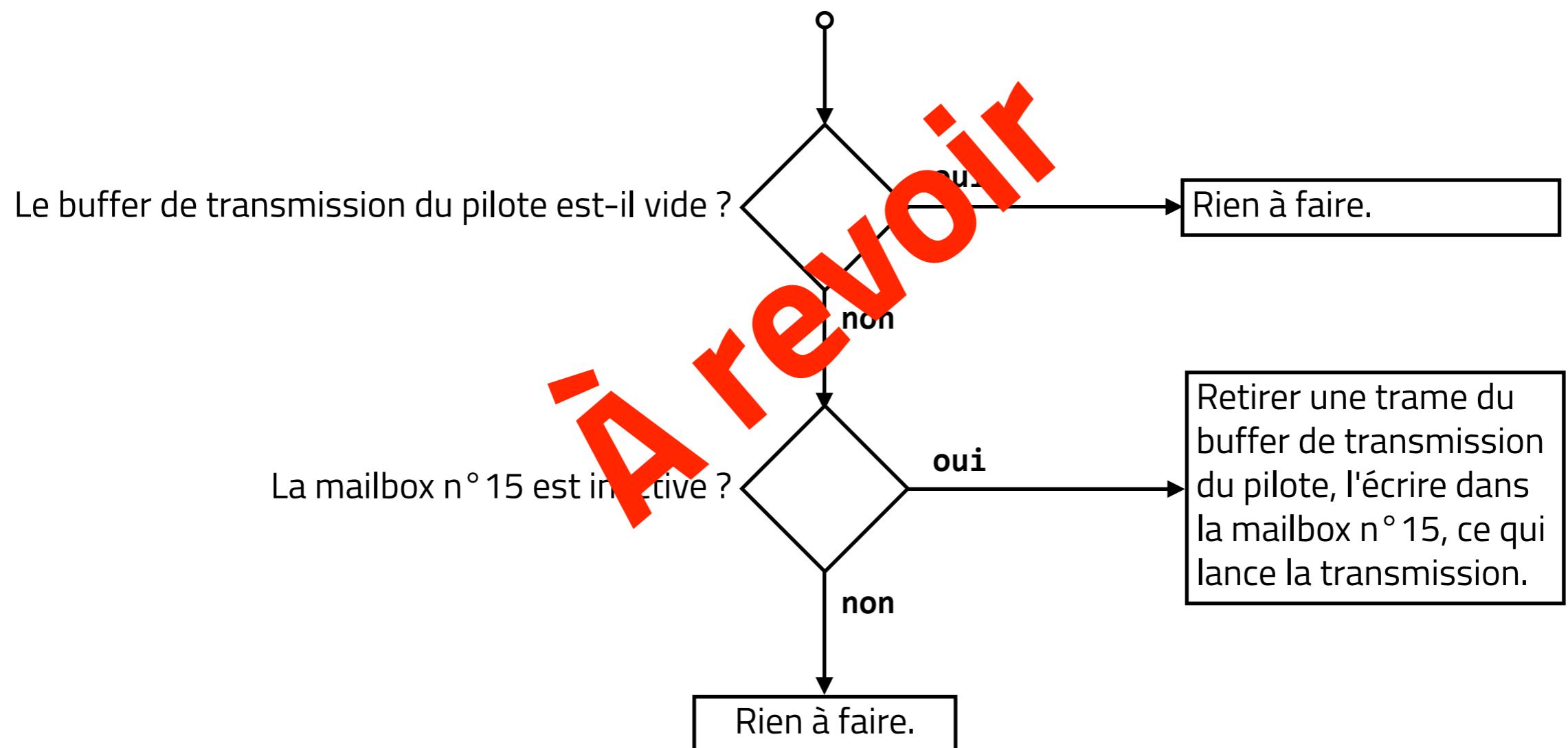
La fonction **ACAN::message_isr** (fichier **can-driver.cpp**) est exécutée tant que le module *FlexCAN* engendre une interruption. Voici cette fonction, après avoir commentée le code qui ne concerne pas l'émission de trame :

```
void ACAN::message_isr (IRQ_MODE) {
    ...
    //--- Handle Tx MBs
    if (mTransmitBufferCount > 0) { // There is a frame in the queue to send
        const uint32_t TxMailBoxIndex = 15 ;
        const uint32_t code
            = FLEXCAN_get_code (FLEXCANb_MBn_CS [mMailboxIndex].CanBaseAddress, TxMailBoxIndex) ;
        if (code == FLEXCAN_MB_CODE_TX_INACTIVE)
            writeTxRegisters (MODE_ mTransmitBuffer [mTransmitBufferReadIndex], TxMailBoxIndex);
        mTransmitBufferReadIndex = (mTransmitBufferReadIndex + 1) % TRANSMIT_BUFFER_SIZE ;
        mTransmitBufferCount -= 1 ;
    }
}
...
}
```

A revoir

La routine d'interruption ACAN::message_isr (2/2)

On peut donc donner l'organigramme de la fonction **ACAN::message_isr** :



Implémentation de l'émission bloquante (1/5)

Dans le fichier **can-driver.h**, effectuez les changements suivants (en bleu) :

```
----- Transmitting messages
public: void send (USER_MODE_ const CANMessage & inMessage) ;

//$/service internal.send
private: void internalSend (USER_MODE_ const CANMessage & inMessage) asm ("internal.send") ;
private: void kernel_internalSend (KERNEL_MODE_ const CANMessage & inMessage)
        asm ("service.internal.send") ;

//--- Driver transmit buffer
private: Semaphore mTransmitSemaphore ;
private: CANMessage mTransmitBuffer [TRANSMIT_BUFFER_SIZE] ;
private: uint32_t mTransmitBufferReadIndex ; // 0 ... TRANSMIT_BUFFER_SIZE-1
private: uint32_t mTransmitBufferCount ; // 0 ... TRANSMIT_BUFFER_SIZE

//--- Internal send method
private: void writeTxRegisters(SECTION_MODE_ const CANMessage & inMessage, const uint32_t inMBIndex);
```

À revoir

Commentaires :

- la fonction `tryToSend` est supprimée, au profit de `send` (la fonction à appeler en mode **USER** pour transmettre un message), et de la fonction `internalSend` ;
- le sémaphore `mTransmitSemaphore` est ajouté (ne pas oublier d'inclure **Semaphore.h**).

Implémentation de l'émission bloquante (2/5)

Dans le fichier **can-driver.cpp**, modifier le constructeur de la classe **ACAN** pour initialiser le sémafor à la valeur de la taille du buffer d'émission :

```
ACAN::ACAN (const uint32_t inFlexcanBaseAddress) :  
mFlexcanBaseAddress (inFlexcanBaseAddress),  
mTransmitSemaphore (TRANSMIT_BUFFER_SIZE),  
mTransmitBufferReadIndex (0),  
.... {  
}
```

À revoir

Implémentation de l'émission bloquante (3/5)

Dans le fichier **can-driver.cpp**, ajouter la fonction **send** (ne pas modifier **section_tryToSend** pour le moment) :

```
void ACAN::send (USER_MODE_ const CANMessage & inMessage)
{
    mTransmitSemaphore.P (MODE) ;
    internalSend (MODE_ inMessage) ;
}
```

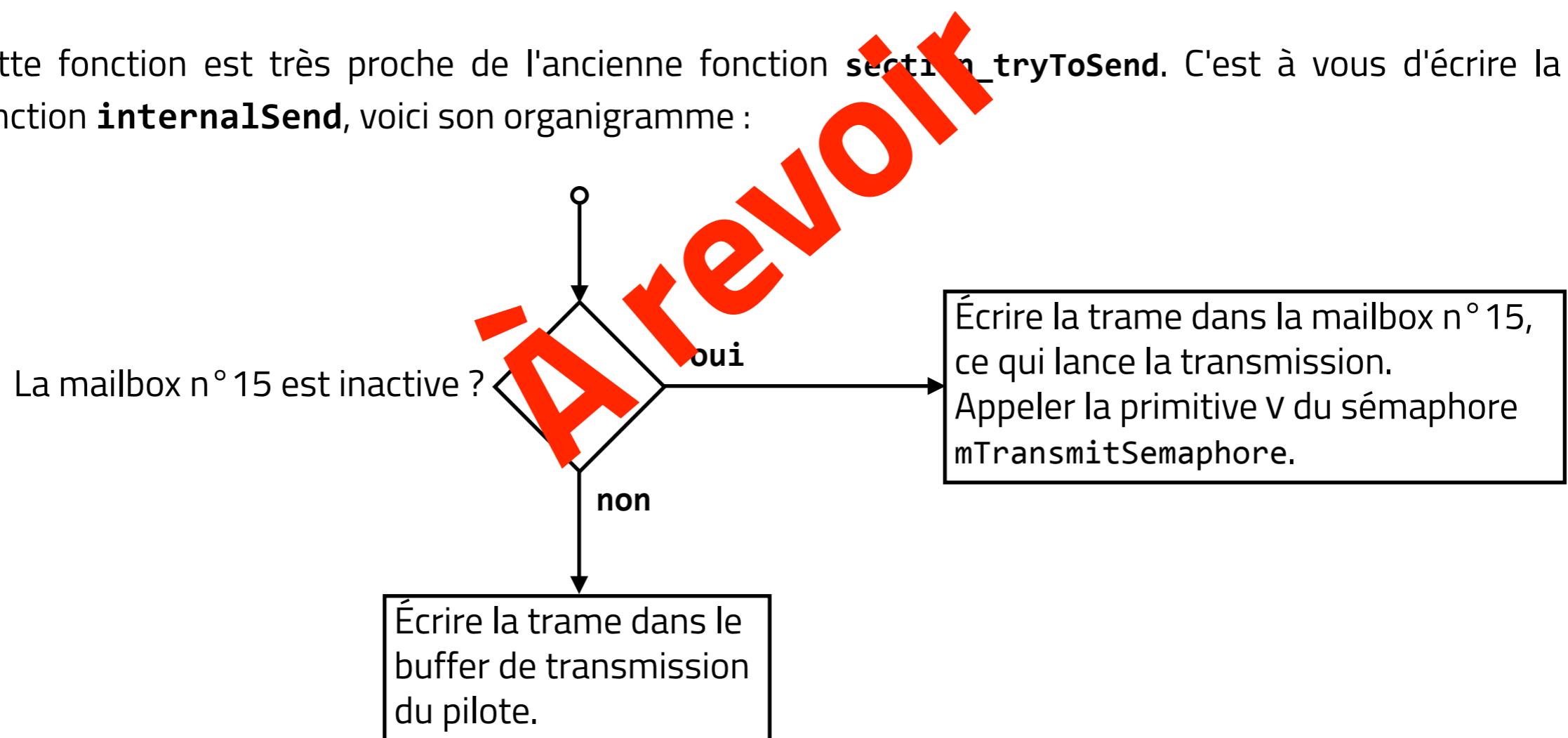
Ainsi la fonction **internalSend** ne sera appelée que si il y a au moins une place dans le buffer de transmission.

Implémentation de l'émission bloquante (4/5)

Écriture de la fonction **internalSend**.

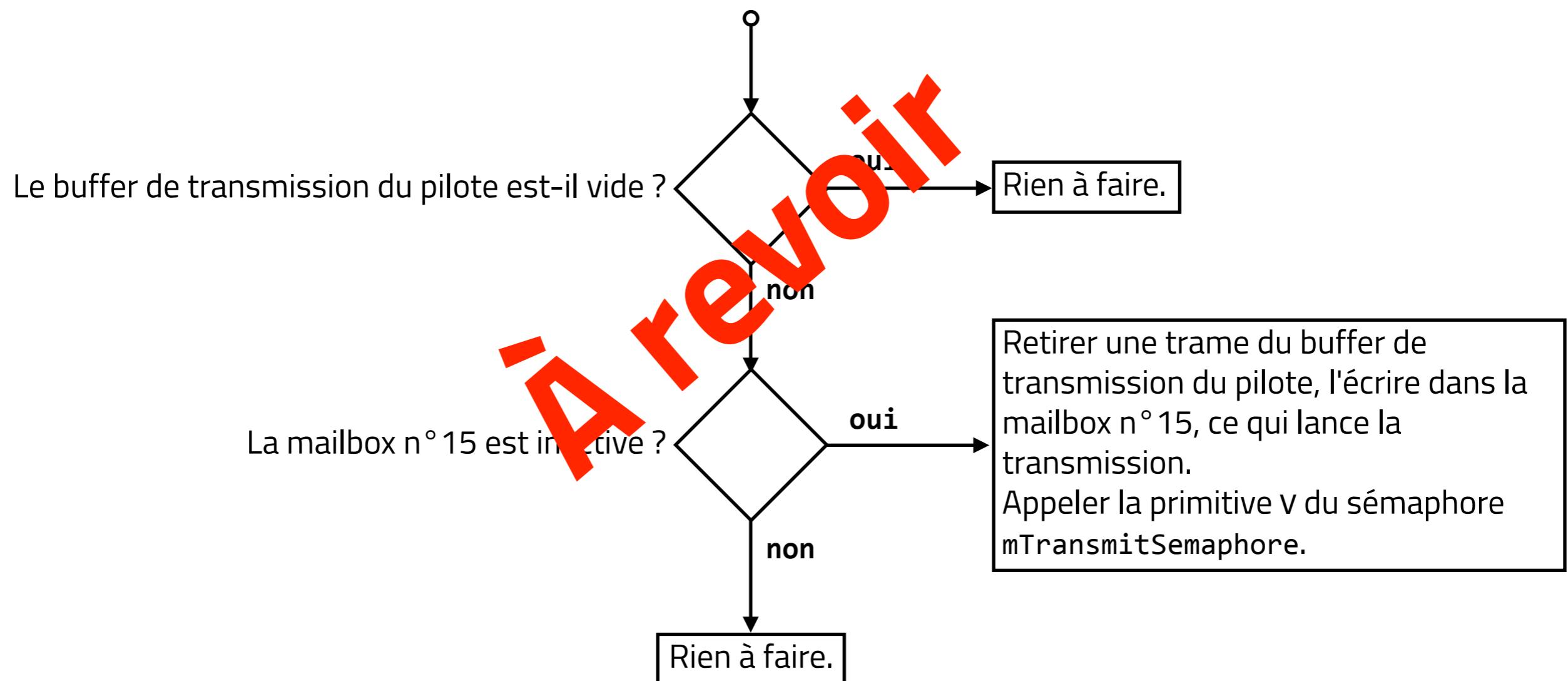
```
void ACAN::kernel_internalSend (KERNEL_MODE_ const CANMessage & inMessage) {  
    ...  
}
```

Cette fonction est très proche de l'ancienne fonction `section_tryToSend`. C'est à vous d'écrire la fonction **internalSend**, voici son organigramme :



Implémentation de l'émission bloquante (5/5)

Il faut aussi modifier la fonction **ACAN::message_isr** :



Programme à réaliser

Maintenant que vous avez l'émission bloquante, reprendre le programme précédent et diviser le code en quatre tâches :

- émission via **CAN0** ;
- émission via **CAN1** ;
- réception via **CAN0** et **CAN1** ;
- affichage toutes les secondes du nombres de trames émises et reçues via **CAN0** et **CAN1**.

À revoir

Étape 21

Réseau CAN (réception bloquante)

Description de cette étape

Le but de cette étape est d'implémenter la réception bloquante.

De manière analogue à l'étape précédente, la description de cette étape commence par présenter comment la commande d'émission reçue était implémentée.

Ensuite, on implémentera la commande bloquante à l'aide d'un sémaphore.

Duplicer l'étape précédente, et recommencer le répertoire obtenu **21-can-network--blocking-receive**.

Comment receive est implémenté (1/2)

Ouvrez le fichier **can-driver.h** et repérez les lignes suivantes :

```
----- Receiving messages
#$section can.receive
public: bool receive (USER_MODE_ CANMessage & outMessage) asm ("can.receive") ;
public: bool section_receive (SECTION_MODE_ CANMessage & outMessage) asm ("section.can.receive") ;

--- Driver receive buffer
private: CANMessage mReceiveBuffer [RECEIVE_BUFFER_SIZE] ;
private: uint32_t mReceiveBufferReadIndex ;
private: uint32_t mReceiveBufferCount ;
private: uint32_t mReceiveBufferPeakCount ; // == mReceiveBufferSize + 1 if overflow did occur
private: uint8_t mFlexcanRxFIFOFlags ;
private: void readRxRegisters (IRQ_MODE_ CANMessage & outMessage) ;
```

À revoir

On distingue :

- la fonction `receive` est implémentée en mode **SECTION** par `section_receive` ; ainsi, `section_receive` ne peut pas être interrompue ;
- le buffer de réception du pilote, suivi des propriétés (initialisées à 0 dans le constructeur) qui contiennent l'indice de lecture et le nombre d'éléments dans le buffer ;
- enfin, la fonction `readRxRegisters` que vous aurez à modifier.

Comment receive est implémenté (2/2)

Ouvrez le fichier **can-driver.cpp** et l'implémentation de la fonction **section_receive** :

```
bool ACAN::section_receive (SECTION_MODE_ CANMessage & outMessage)
{
    const bool hasMessage = mReceiveBufferCount > 0 ;
    if (hasMessage) {
        outMessage = mReceiveBuffer [mReceiveBufferReadIndex] ;
        mReceiveBufferReadIndex = (mReceiveBufferReadIndex + 1) % RECEIVE_BUFFER_SIZE ;
        mReceiveBufferCount -= 1 ;
    }
    return hasMessage ;
}
```

À revoir

Le code est simple : il examine si le buffer de réception contient des messages ou non. Si oui, un message est retiré.

La routine d'interruption ACAN::message_isr (1/2)

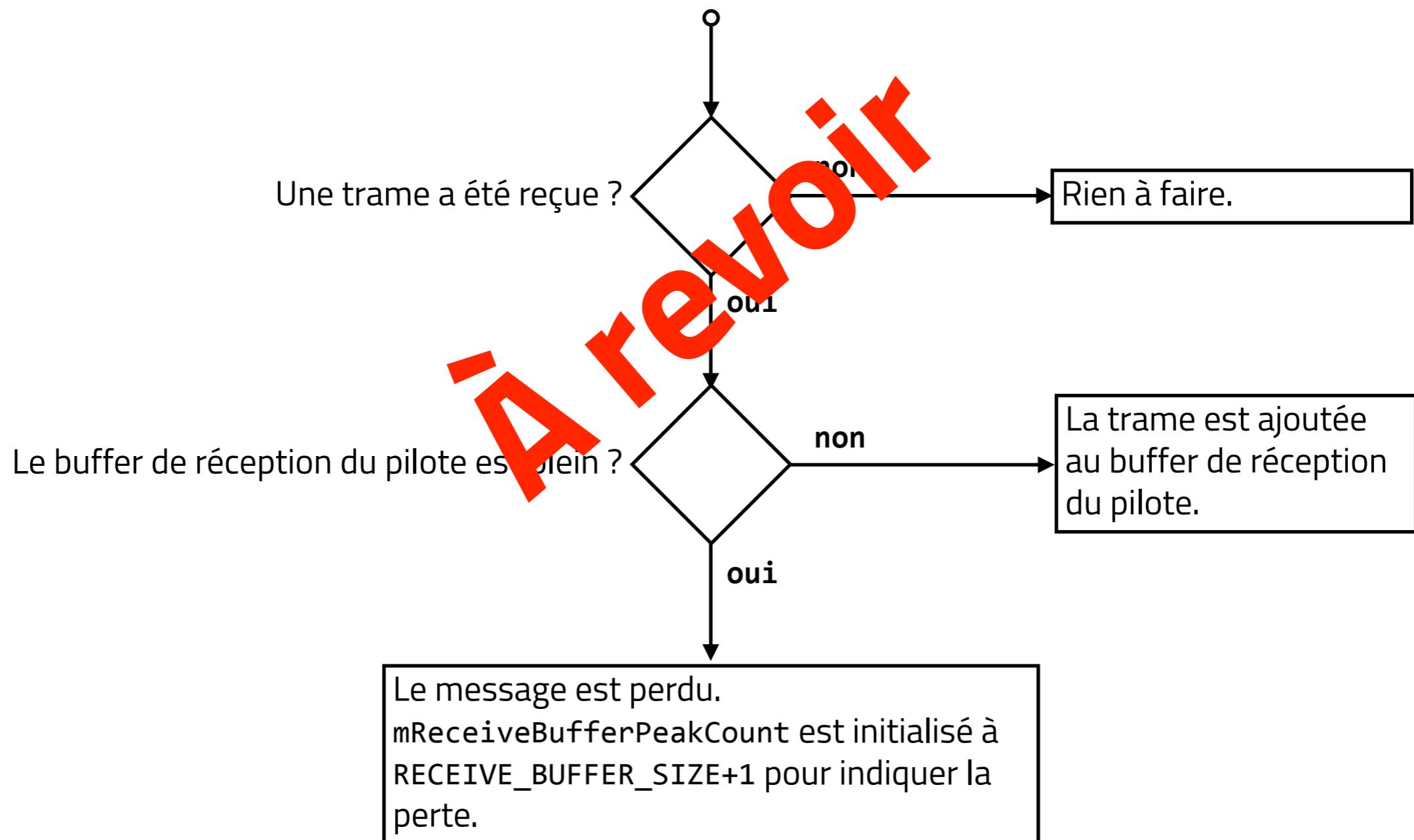
La fonction **ACAN::message_isr** (fichier **can-driver.cpp**) est exécutée tant que le module *FlexCAN* engendre une interruption. Voici cette fonction, après avoir commentée le code qui ne concerne pas la réception de trames :

```
void ACAN::message_isr (IRQ_MODE) {  
    ...  
    //--- A trame has been received in RxFIFO ?  
    if ((status & (1 << 5)) != 0) {  
        CANMessage message ;  
        readRxRegisters (MODE_ message) ;  
        if (mReceiveBufferCount == RECEIVE_BUFFER_SIZE) { // OverFlow! Receive buffer is full  
            mReceiveBufferPeakCount = RECEIVE_BUFFER_SIZE + 1 ; / Mark overflow  
        }else{  
            uint32_t receiveBufferWriteIndex = mReceiveBuffer.readIndex + mReceiveBufferCount ;  
            if (receiveBufferWriteIndex >= RECEIVE_BUFFER_SIZE) {  
                receiveBufferWriteIndex -= RECEIVE_BUFFER_SIZE ;  
            }  
            mReceiveBuffer [receiveBufferWriteIndex] = message ;  
            mReceiveBufferCount += 1 ;  
            if (mReceiveBufferCount > mReceiveBufferPeakCount) {  
                mReceiveBufferPeakCount = mReceiveBufferCount ;  
            }  
        }  
    }  
    ...  
}
```

A revoir

La routine d'interruption ACAN::message_isr (2/2)

On peut donc donner l'organigramme de la fonction **ACAN::message_isr** :



Implémentation de la réception bloquante (1/4)

Dans le fichier **can-driver.h**, effectuez les changements suivants (en bleu) :

```
//----- Receiving messages
public: void receive (USER_MODE_ CANMessage & outMessage) ;

//$/section internal.receive
public: void internalReceive (USER_MODE_ CANMessage & outMessage) asm ("internal.receive") ;
public: void section_internalReceive (SECTION_MODE_ CANMessage & outMessage)
    asm ("section.internal.receive") ;

//--- Driver receive buffer
private: Semaphore mReceiveSemaphore ;
private: CANMessage mReceiveBuffer [RECEIVE_BUFFER_SIZE] ;
private: uint32_t mReceiveBufferReadIndex ;
private: uint32_t mReceiveBufferCount ;
private: uint32_t mReceiveBufferPeakCount ; // == mReceiveBufferSize + 1 if overflow did occur
private: uint8_t mFlexcanRxFIFOFlags ;
private: void readRxRegisters (IRQ_MODE_ CANMessage & outMessage) ;
```

À revoir

Commentaires :

- la fonction `receive` ne renvoie plus de booléen, et la fonction `internalReceive` est ajoutée ;
- le sémaphore `mReceiveSemaphore` est ajouté.

Implémentation de la réception bloquante (2/4)

Dans le fichier **can-driver.cpp**, modifier le constructeur de la classe **ACAN** pour initialiser le sémaphore **mReceiveSemaphore** à 0 (initialement, le buffer de réception est vide).

Dans le fichier **can-driver.cpp**, ajouter la fonction **receive** (ne pas modifier **section_receive** pour le moment) :

```
void ACAN::receive (USER_MODE_ CANMessage & outMessage) {  
    mReceiveSemaphore.P (MODE) ;  
    internalReceive (MODE_ outMessage) ;  
}
```

Ainsi la fonction **internalReceive** ne sera appelée que si il y a au moins une message dans le buffer de réception.

Implémentation de la réception bloquante (3/4)

Écriture de la fonction **internalReceive**.

```
void ACAN::section_internalReceive (SECTION_MODE_ CANMessage & outMessage) {  
    ...  
}
```

Cette fonction est très proche de l'ancienne fonction **section_receive**. C'est à vous de l'écrire, il suffit simplement de récupérer le message le plus ancien du buffer de réception.

À revoir

Implémentation de la réception bloquante (4/4)

Il faut aussi modifier la fonction **ACAN::message_isr**; cette modification est simple, il suffit d'appeler la primitive V du sémaaphore **mReceiveSemaphore** quand un message est inséré dans le buffer de réception.

Attention, ne pas faire cette opération si le buffer de réception est plein et que le message reçu est perdu.

A revoir

Programme à réaliser

Maintenant que vous avez émissions et réceptions bloquantes reprendre le programme précédent et diviser le code en cinq tâches :

- émission via **CAN0** ;
- émission via **CAN1** ;
- réception via **CAN0** ;
- réception via **CAN1** ;
- affichage toutes les secondes du nombres de trames émises et reçues via **CAN0** et **CAN1**.

À revoir