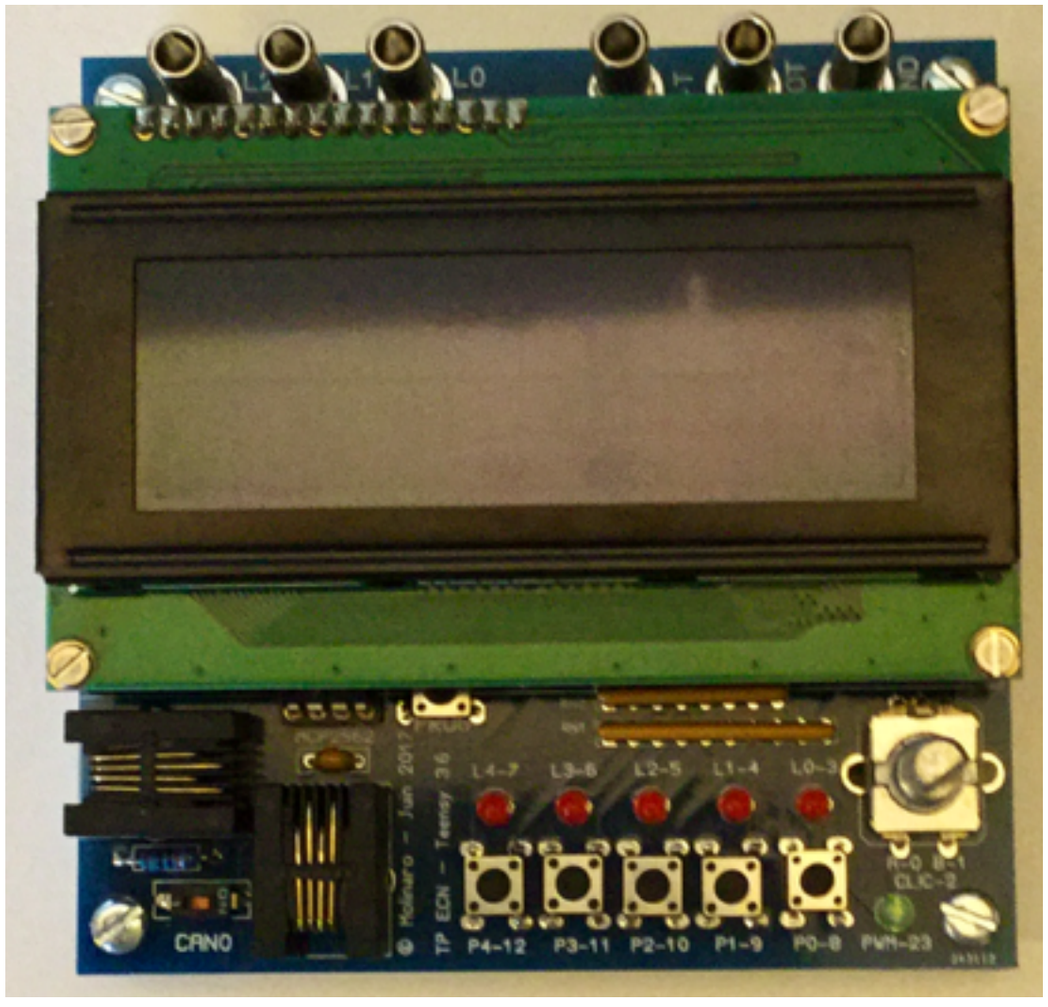
Temps Réel

Construire un exécutif Temps-Réel pas-à-pas



Étape 01-blink-led

Pierre Molinaro

Juin 2018

1 — Présentation

Objectifs de l'enseignement TReel

Objectifs :

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

Moyens :

- mise à disposition de cartes micro-contrôleurs ;

Organisation :

- cours / TD / TP ;

Étapes de réalisation

10 séances de 3 h, pour réaliser les étapes suivantes :

1. présentation / introduction : programme « blink led » ;
2. SysTick ;
3. Modes logiciels ;
4. Boot et init routines;
5. Leds / boutons poussoir ;
6. Afficheur LCD ;
7. Interruption Systick ;
8. Section critique ;
9. Assertions et *Fault Handler* ;
10. Appel SVC ;
11. Premier exécutif : une seule tâche ;
12. Terminaison des tâches ;
13. Attente passive ;
14. Sémaphore de Dijkstra ;
15. Exclusion mutuelle accès afficheur LCD ;
16. Outils de synchronisation ;
17. Allocation dynamique ;
18. Commandes de synchronisation gardées (à la CSP).

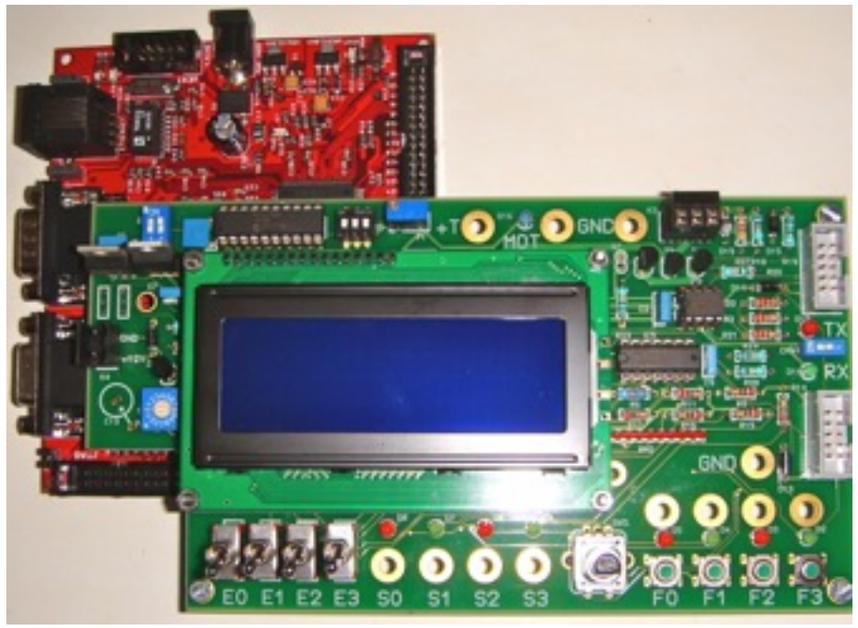
Évaluation :

- devoir surveillé.

Note

Le matériel de développement a été changé plusieurs fois.

2011/2012 → 2014/2015



Micro-contrôleur : LPC2294

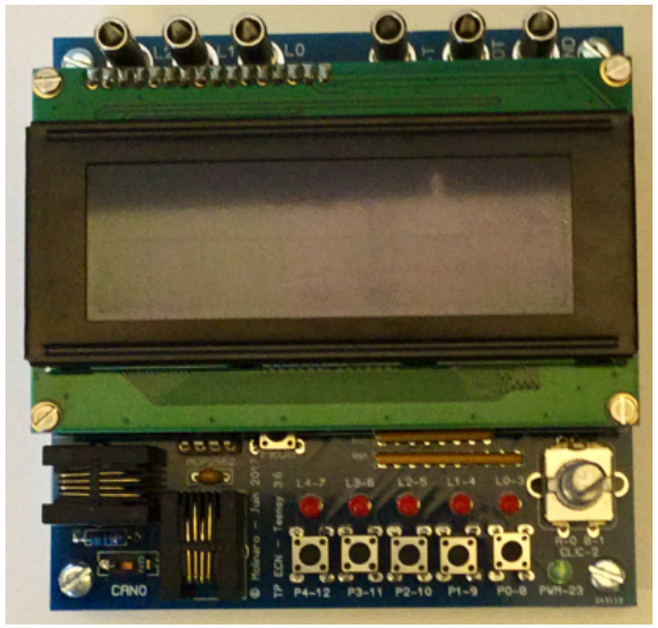
Processeur : ARM7TDMI

2015/2016 → 2017/2018

Micro-contrôleur : MK20DX256VLH7

Processeur : Cortex-M4

2018/2019 → ?



Micro-contrôleur : MK66FX1M0

Processeur : Cortex-M4F

2 — 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. récupérer l'archive `info-treel.tar.bz2` sur le serveur pédagogique ;
2. décompressez-la ;
3. 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** ;
4. 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 ;

Le système de développement est installé dans le répertoire `~/treel-tools`. Pour désinstaller, il suffit de supprimer ce répertoire.

Les archives sont téléchargées dans `TREEL/archives`.

Exemple de *log* d'installation sur Mac

```
Install GCC tools...
+ rm -f /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/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...
 100.0% of 99MiB
+ mv /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2.downloading /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/
archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2
+ cp /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2 /Users/pierremolinaro/treel-tools/gcc-arm-none-eabi-7-2017-
q4-major.tar.bz2
+ cd /Users/pierremolinaro/treel-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
Install Teensy CLI Loader...
+ curl -L -o /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/archives/master.zip https://github.com/PaulStoffregen/teensy_loader_cli/archive/master.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   137    0   137    0     0    119      0 --:--:--  0:00:01 --:--:--   119
100  130k  100  130k    0     0  53365      0  0:00:02  0:00:02 --:--:--  159k
+ unzip master.zip
Archive:  master.zip
d239f76f01f922e115c2661a6a3855dfc660a69c
   creating: teensy_loader_cli-master/
............
  inflating: teensy_loader_cli-master/teensy_loader_cli.c
+ cp teensy_loader_cli-master/teensy_loader_cli.c teensy_loader_cli.c
+ rm master.zip
+ rm -r teensy_loader_cli-master
+ gcc -O2 -fomit-frame-pointer -DUSE_APPLE_IOKIT /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/archives/teensy_loader_cli.c -framework Foundation -framework IOKit
-o /Users/pierremolinaro/treel-tools/gcc-arm-none-eabi-7-2017-q4-major/bin/teensy_loader_cli
--- Making /Volumes/dev-svn/GITHUB/real-time-kernel-teensy/steps/01-blink-led
Making "zBUILDS" directory
[  6%] Assembling reset-handler-sequential.s
[ 13%] Assembling teensy-3-6-interrupt-vectors.s
[ 20%] Assembling unused-interrupt.s
Making "zSOURCES" directory
[ 26%] Build base header file
[ 33%] Build all headers file
[ 40%] Build interrupt files
[ 46%] Checking setup-loop.cpp
[ 53%] Checking start-teensy-3-6.cpp
[ 60%] Compiling setup-loop.cpp
[ 66%] Compiling start-teensy-3-6.cpp
[ 73%] Checking interrupt-handler-helper.cpp
[ 80%] Compiling interrupt-handler-helper.cpp
[ 86%] Assembling interrupt-handlers.s
Making "zPRODUCTS" directory
[ 93%] Linking zPRODUCTS/product-internal-flash.elf
[100%] Hexing zPRODUCTS/product-internal-flash.ihex
*** Internal FLASH:
  ROM code:    2312 bytes
  ROM data:    0 bytes
  RAM + STACK: 1544 bytes
```

Installation sur Linux

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

Installation :

1. récupérer l'archive `info-treel.tar.bz2` sur le serveur pédagogique ;
2. décompressez-la ;
3. 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** ;
4. 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 ; **sur Linux, l'installation demande en plus le mot de passe administrateur pour installer l'accès à l'USB (udev)**.

Le système de développement est installé dans le répertoire `~/treel-tools`. Pour désinstaller, il suffit de supprimer ce répertoire, et de supprimer le fichier `/etc/udev/rules.d/49-teensy.rules`.

Les archives sont téléchargées dans `TREEL/archives`.

Exemple de *log* d'installation sur Linux

```
Install GCC tools...
+ rm -f /home/pierre64/Bureau/real-time-kernel-teensy/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-linux.tar.bz2
Downloading...
 100.0% of 95MiB
+ mv /home/pierre64/Bureau/real-time-kernel-teensy/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2.downloading /home/pierre64/Bureau/real-time-kernel-teensy/
archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2
+ cp /home/pierre64/Bureau/real-time-kernel-teensy/archives/gcc-arm-none-eabi-7-2017-q4-major.tar.bz2 /home/pierre64/treel-tools/gcc-arm-none-eabi-7-2017-q4-
major.tar.bz2
+ cd /home/pierre64/treel-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
Install Teensy CLI Loader...
+ curl -L -o /home/pierre64/Bureau/real-time-kernel-teensy/archives/master.zip https://github.com/PaulStoffregen/teensy_loader_cli/archive/master.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   137    0   137    0     0    150      0 --:--:-- --:--:-- --:--:--   150
100  130k  100  130k    0     0  63443      0  0:00:02  0:00:02 --:--:--  236k
+ unzip master.zip
Archive:  master.zip
d239f76f01f922e115c2661a6a3855dfc660a69c
   creating: teensy_loader_cli-master/
..........
 inflating: teensy_loader_cli-master/teensy_loader_cli.c
+ cp teensy_loader_cli-master/teensy_loader_cli.c teensy_loader_cli.c
+ rm master.zip
+ rm -r teensy_loader_cli-master
+ gcc -O2 -fomit-frame-pointer /home/pierre64/Bureau/real-time-kernel-teensy/archives/teensy_loader_cli.c -o /home/pierre64/treel-tools/gcc-arm-none-eabi-7-2017-
q4-major/bin/teensy_loader_cli -DUSE_LIBUSB -lusb
--- Making /home/pierre64/Bureau/real-time-kernel-teensy/steps/01-blink-led
Making "zBUILDS" directory
[  6%] Assembling reset-handler-sequential.s
[ 13%] Assembling teensy-3-6-interrupt-vectors.s
[ 20%] Assembling unused-interrupt.s
Making "zSOURCES" directory
[ 26%] Build base header file
[ 33%] Build all headers file
[ 40%] Build interrupt files
[ 46%] Checking start-teensy-3-6.cpp
[ 53%] Checking setup-loop.cpp
[ 60%] Compiling start-teensy-3-6.cpp
[ 66%] Checking interrupt-handler-helper.cpp
[ 73%] Compiling setup-loop.cpp
[ 80%] Compiling interrupt-handler-helper.cpp
[ 86%] Assembling interrupt-handlers.s
Making "zPRODUCTS" directory
[ 93%] Linking zPRODUCTS/product-internal-flash.elf
[100%] Hexing zPRODUCTS/product-internal-flash.ihex
*** Internal FLASH:
  ROM code:    2312 bytes
  ROM data:    0 bytes
  RAM + STACK: 1544 bytes
```

3 — Compilation et exécution

Présentation

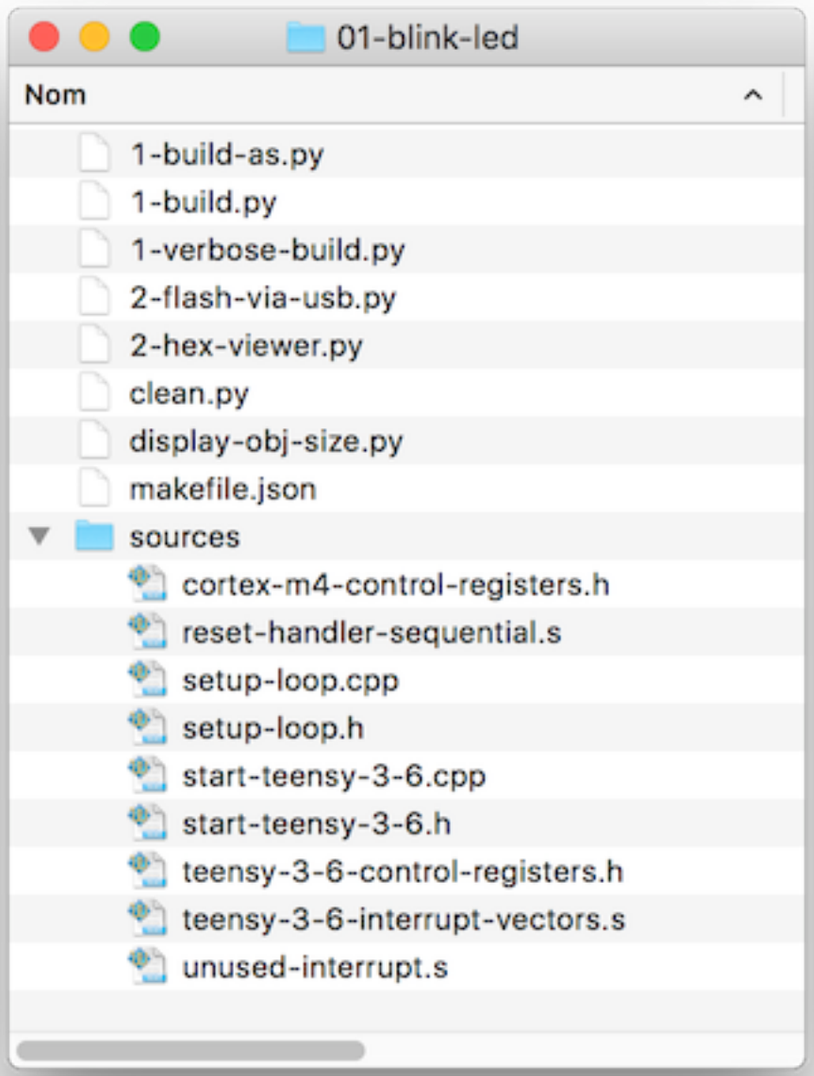
Le premier programme `01-blinkled` est un programme minimum pour débuter en faisant clignoter une led.

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.
`2-hex-viewer.py`	Enchaîne compilation et affichage du code binaire obtenu.
`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 :

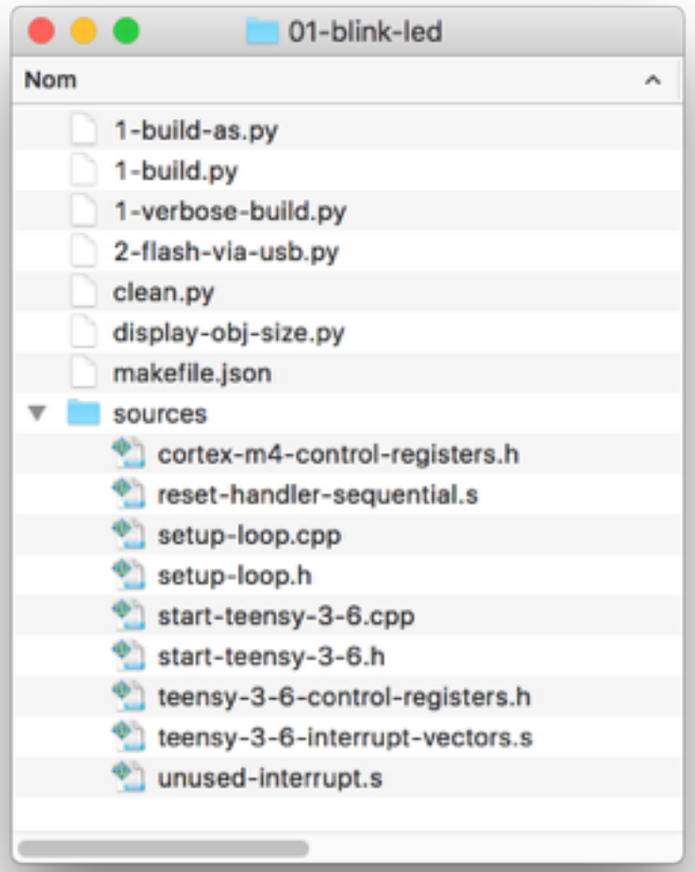
https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

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

```
{ "SOURCE-DIR" : ["sources"],

  "TEENSY" : "3.6",

  "CPU-MHZ" : 180
}
```

`TEENSY`	Caractérise la plateforme de développement. Ne pas changer.
`SOURCE_DIR`	Liste des répertoires sources. Ne pas changer, sauf si vous répartissez vos sources dans plusieurs répertoires.
`CPU-MHZ`	Fréquence du processeur, en MHz. Vous pouvez choisir : 24, 48, 72, 96, 120, 144, 168, 180, (overclock) 192, 216, 240.

Fichiers du répertoire `sources`



`cortex-m4-control-registers.h`	Déclaration des registres de contrôle du processeur Cortex-M4
`reset-handler-sequential.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 `setup` et `loop`.
`start-teensy-3-6.cpp`	Routines de démarrage et d'initialisation.
`start-teensy-3-6.h`	Déclaration des prototypes.
`teensy-3-6-control-registers.h`	Déclaration des registres de contrôle du micro-contrôleur.
`teensy-3-6-interrupt-vectors.s`	Vecteur des interruptions du micro-contrôleur.
`unused-interrupt.s`	Prise en charge des interruptions inutilisées (jusqu'à l'étape *`fault-handler--assertion`*).

Compiler le programme d'exemple

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

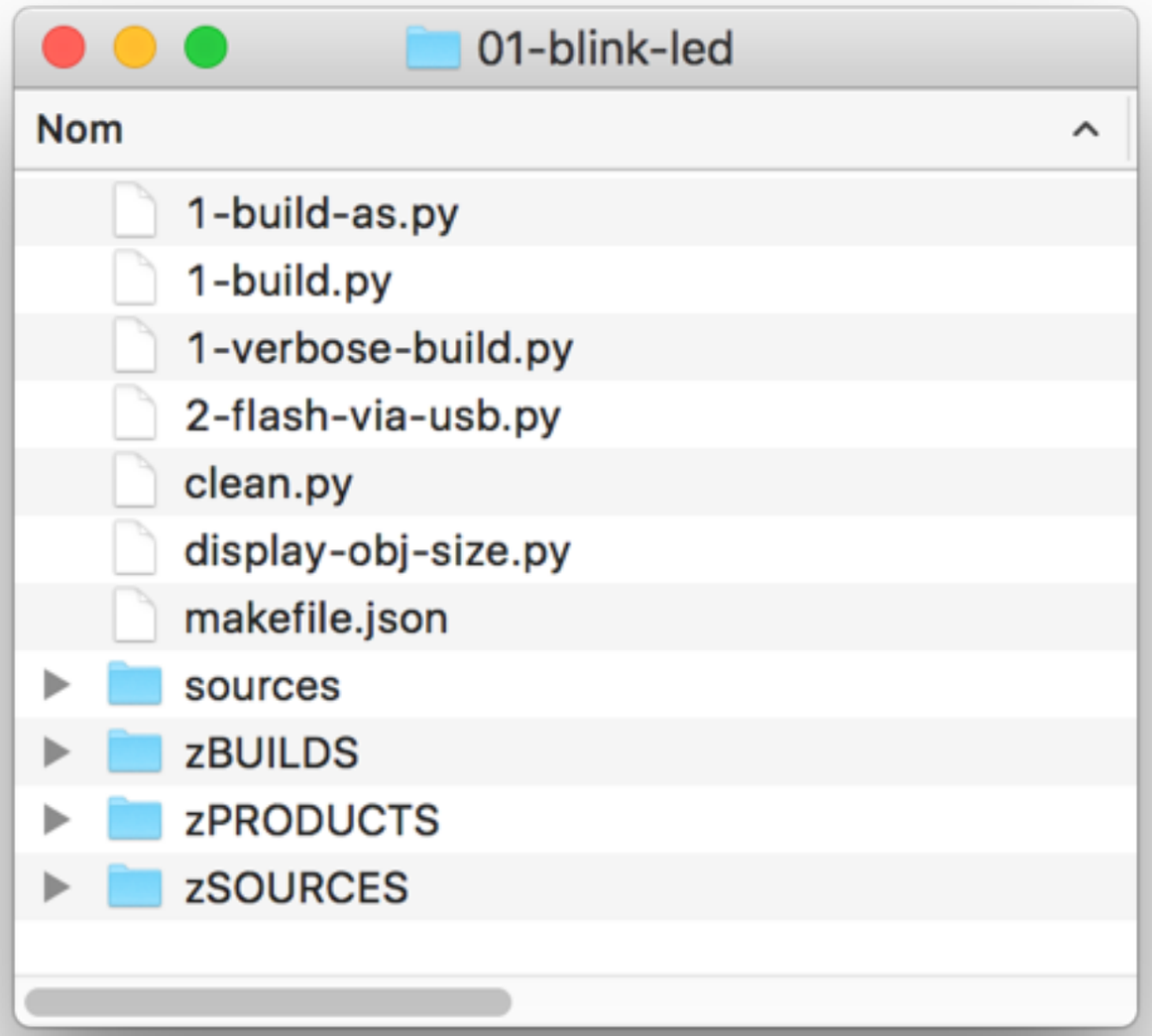
Il produit un ensemble de fichiers et de répertoires qui sont décrits dans les pages suivantes.

```
--- Making steps/01-blink-led
Making "zBUILDS" directory
[  6%] Assembling reset-handler-sequential.s
[ 13%] Assembling teensy-3-6-interrupt-vectors.s
[ 20%] Assembling unused-interrupt.s
Making "zSOURCES" directory
[ 26%] Build base header file
[ 33%] Build all headers file
[ 40%] Build interrupt files
[ 46%] Checking setup-loop.cpp
[ 53%] Checking start-teensy-3-6.cpp
[ 60%] Compiling setup-loop.cpp
[ 66%] Compiling start-teensy-3-6.cpp
[ 73%] Checking interrupt-handler-helper.cpp
[ 80%] Compiling interrupt-handler-helper.cpp
[ 86%] Assembling interrupt-handlers.s
Making "zPRODUCTS" directory
[ 93%] Linking zPRODUCTS/product-internal-flash.elf
[100%] Hexing zPRODUCTS/product-internal-flash.ihex
  ROM code:    2312 bytes
  ROM data:    0 bytes
  RAM + STACK: 1544 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.

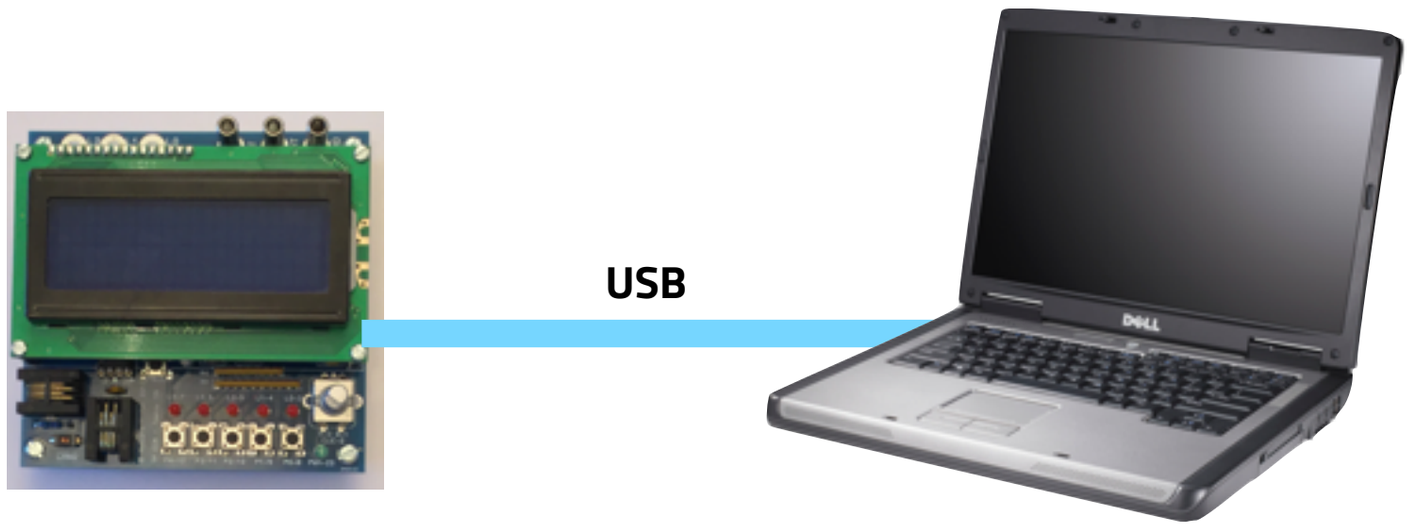
Le répertoire d'un programme après compilation



`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 la carte avec le cordon USB : le programme en Flash s'exécute.

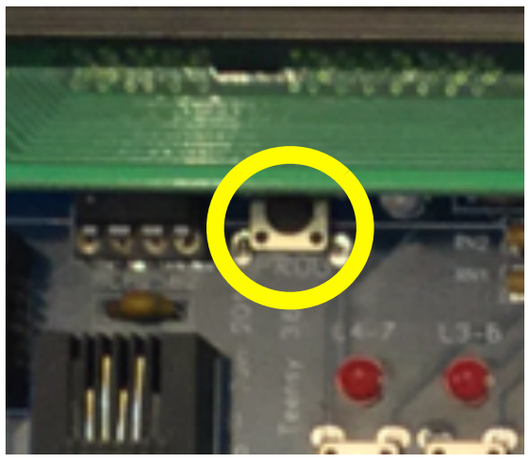


B) Double cliquer sur le script **`2-flash-via-usb.py`** :

(1) ceci lance le programme de flashage ;

```
Teensy Loader, Command Line, Version 2.1
Read "zPRODUCTS/product-internal-flash.ihex": 2312 bytes, 0.2% usage
Waiting for Teensy device...
 (hint: press the reset button)
```

(2) appuyer fugitivement sur le poussoir **PROG** pour lancer le flashage ;



(3) le flashage s'effectue ;

(4) quand il est achevé, le script se termine, et le programme flashé s'exécute.

```
Found HalfKay Bootloader
Read "zPRODUCTS/product-internal-flash.ihex": 2312 bytes, 0.2% usage
Programming..
Booting
logout
```

Comment est effectuée la génération du fichier produit

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

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 ;
- enfin, le fichier *hex* est produit.

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

Construction des fichiers source complémentaires

zSOURCES

makefile.json

Les paramètres du projet sont écrits dans le fichier `base.h`

En plus que les trois actuels, il y a d'autres paramètres qui seront présentés au cours des étapes.

base.h

Tous les fichiers d'en-tête

Tous les fichiers d'en-tête sont inclus dans `all-headers.h`

Le fichier `base.h` *est inclus en premier*

all-headers.h

Tous les fichiers d'en-tête

Les fichiers d'en-tête déclarent les routines d'interruption, les services de l'exécutif.

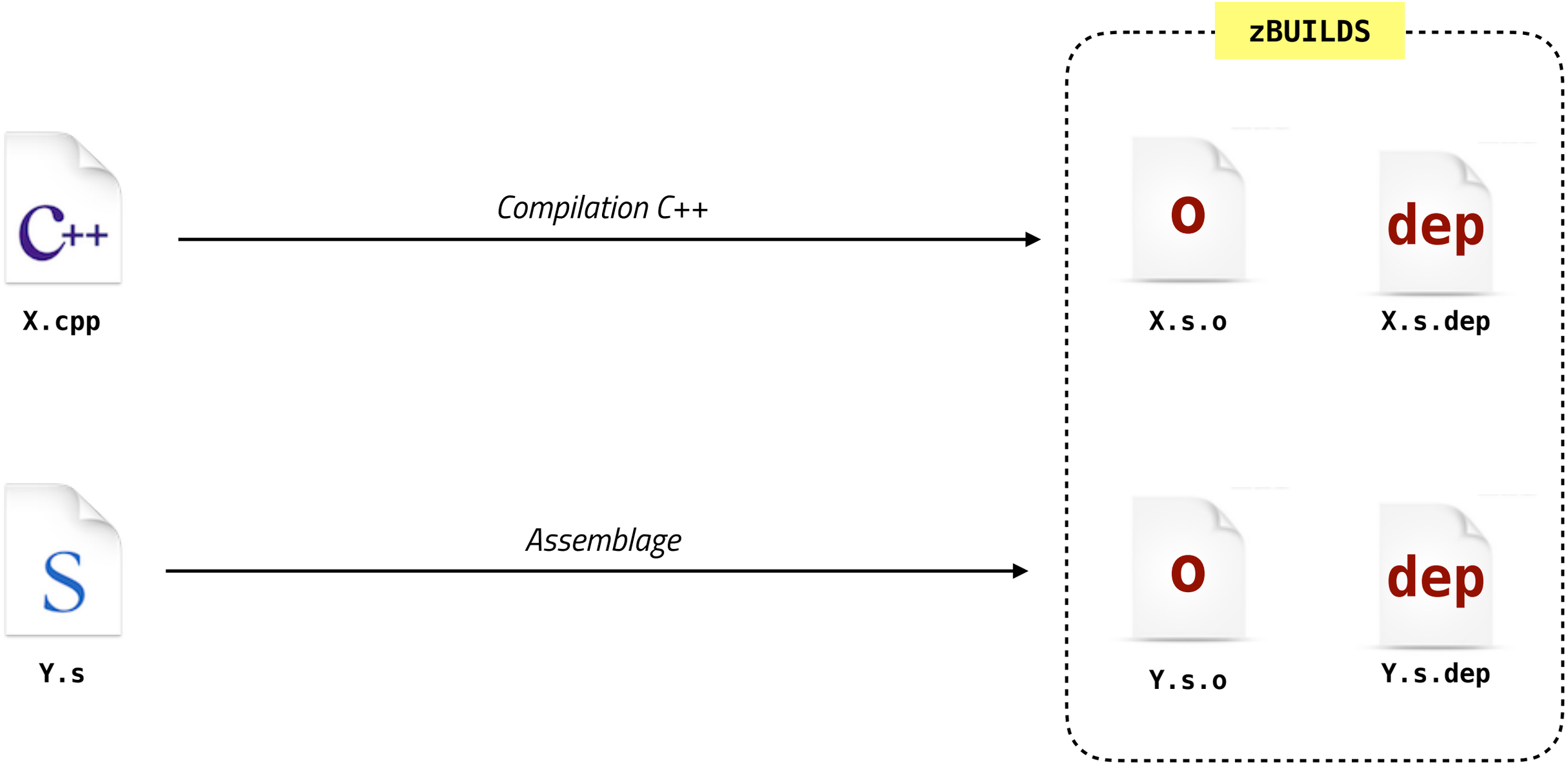
Le fichier assembleur construit comprendra toutes les routines assembleur de l'exécutif.

interrupt-handlers.s

Routines C++ additionnelles

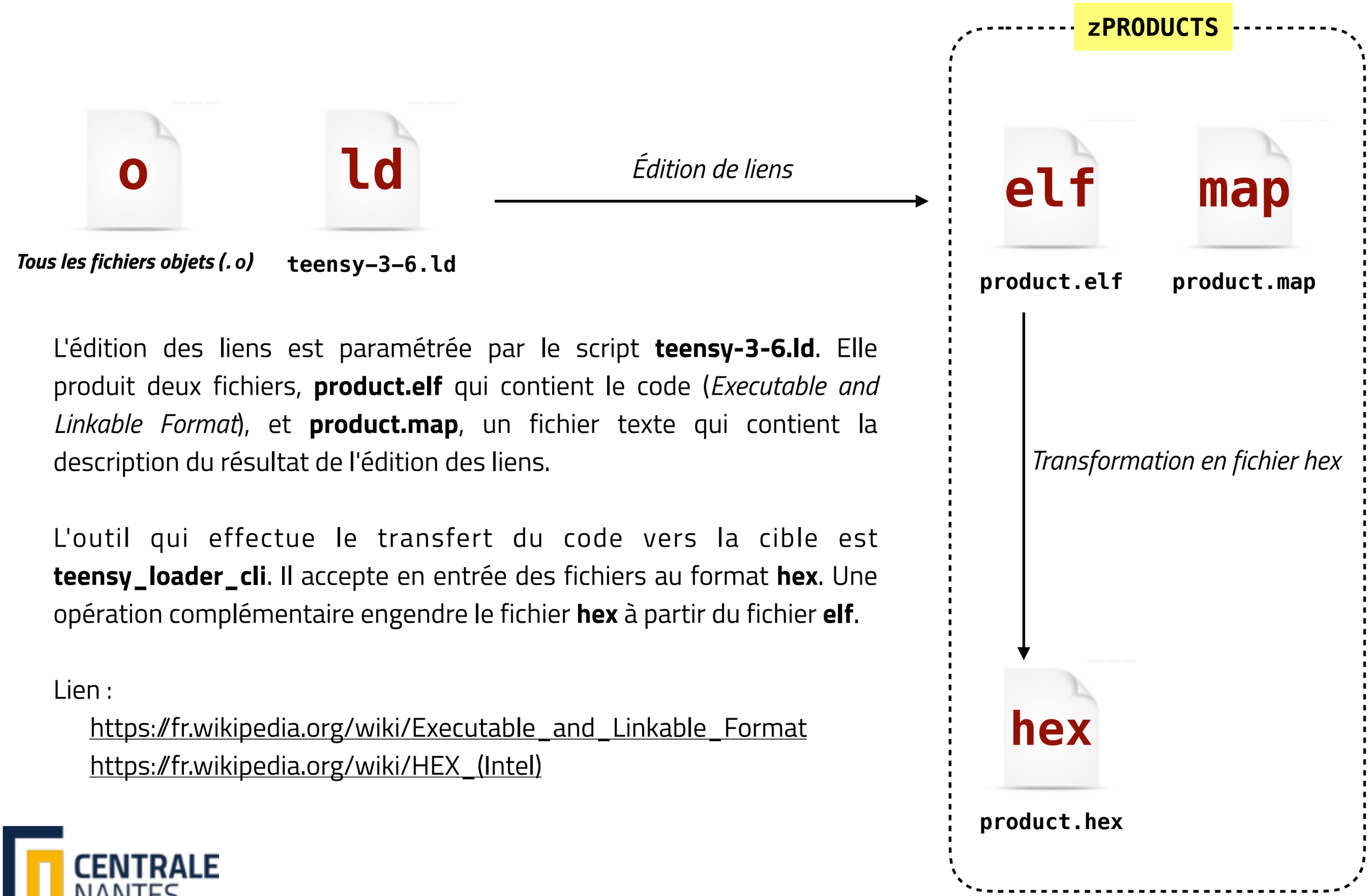
interrupt-handler-helper.cpp

Construction des fichiers objets et de dépendance



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

Édition des liens



L'édition des liens est paramétrée par le script **teensy-3-6.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 le transfert du code vers la cible est **teensy_loader_cli**. Il accepte en entrée des fichiers au format **hex**. Une opération complémentaire engendre le fichier **hex** à partir du fichier **elf**.

Lien :

https://fr.wikipedia.org/wiki/Executable_and_Linkable_Format
https://fr.wikipedia.org/wiki/HEX_(Intel)

4 — 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 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-M4F, c'est-à-dire qu'il intègre une unité de calculs en flottant, toutefois l'utiliser sous exécutif impliquerait de sauvegarder le contexte flottant, ce qui n'est pas fait.

Rendre les objets C++ non copiables

Il suffit de déclarer l'opérateur d'affectation et le constructeur de recopie d'une classe, sans les implémenter.

```
class T {

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

Lors d'une tentative de recopie, une erreur de compilation se déclenche, sauf si cette recopie apparaît dans les méthodes de la classe. Dans ce dernier cas, on obtient une erreur d'édition de liens.

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
  …déclarations…
#endif
```

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

```
#pragma once
  …déclarations…
```

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 language 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 :

```
void kernel_create_task (uint64_t * inStackBufferAddress,
                         uint32_t inStackBufferSize,
                         RoutineTaskType inTaskRoutine) asm ("kernel.create.task");
```

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 **kernel.create.task** est donc valide. Comme les noms issus de la compilation C++ ne comportent pas de point, on limite ainsi le risque de collision.

On procède de la même façon avec des variables communes au C++ et à l'assembleur.

```
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

Address	Vector	IRQ[1]	NVIC non-IPR register number [2]	NVIC IPR register number [3]	Source module	Source description
ARM Core System Handler Vectors						
0x0000_0000	0	–	–	–	ARM core	Initial Stack Pointer
0x0000_0004	1	–	–	–	ARM core	Initial Program Counter
0x0000_0008	2	–	–	–	ARM core	Non-maskable Interrupt (NMI)
0x0000_000C	3	–	–	–	ARM core	Hard Fault
0x0000_0010	4	–	–	–	ARM core	MemManage Fault
0x0000_0014	5	–	–	–	ARM core	Bus Fault
0x0000_0018	6	–	–	–	ARM core	Usage Fault
0x0000_001C	7	–	–	–	—	—
0x0000_0020	8	–	–	–	—	—
0x0000_0024	9	–	–	–	—	—
0x0000_0028	10	–	–	–	—	—
0x0000_002C	11	–	–	–	ARM core	Supervisor call (SVCall)
0x0000_0030	12	–	–	–	ARM core	Debug Monitor
0x0000_0034	13	–	–	–	—	—
0x0000_0038	14	–	–	–	ARM core	Pendable request for system service (PendableSrvReq)
0x0000_003C	15	–	–	–	ARM core	System tick timer (SysTick)
Non-Core Vectors						
0x0000_0040	16	0	0	0	DMA	DMA channel 0 transfer complete
0x0000_0044	17	1	0	0	DMA	DMA channel 1 transfer complete
0x0000_0048	18	2	0	0	DMA	DMA channel 2 transfer complete
0x0000_004C	19	3	0	0	DMA	DMA channel 3 transfer complete
0x0000_0050	20	4	0	1	DMA	DMA channel 4 transfer complete
0x0000_0054	21	5	0	1	DMA	DMA channel 5 transfer complete
0x0000_0058	22	6	0	1	DMA	DMA channel 6 transfer complete

•••

0x0000_01AC	107	91	2	22	Port control module	Pin detect (Port E)
0x0000_01B0	108	92	2	23	—	—
0x0000_01B4	109	93	2	23	—	—
0x0000_01B8	110	94	2	23	Software	Software interrupt[4]

La table des vecteurs d'interruption doit être située à l'adresse `0`, c'est-à-dire au début de la mémoire Flash.

Le fichier **teensy-3-6-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 `0`.

```
  .section isr.vectors, "a", %progbits
@———————————————————————————————————————

  .word __system_stack_end
@--- ARM Core System Handler Vectors
  .word reset.handler @ 1
  .word interrupt.NMI @ 2
  .word interrupt.HardFault @ 3
  .word interrupt.MemManage @ 4
  .word interrupt.BusFault @ 5
  .word interrupt.UsageFault @ 6
  .word -1 @ 7, reserved
  .word -1 @ 8, reserved
  .word -1 @ 9, reserved
  .word -1 @ 10, reserved
  .word interrupt.SVC @ 11
  .word interrupt.DebugMonitor @ 12
  .word -1 @ 13, reserved
.......
```

Démarrage du micro-contrôleur

Le micro-contrôleur démarre 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`.

```
 .section isr.vectors, "a", %progbits
.word __system_stack_end
@--- ARM Core System Handler Vectors
  .word reset.handler @ 1
```

Le mot à l'adresse `0x0` est **`__system_stack_end`** ; ce symbole est défini par l'éditeur de liens. Le mot à l'adresse `0x4` est **`reset.handler`**, ce symbole correspond à la fonction définie dans dans le fichier **reset-handler-sequential.s**.

Exécution du programme

Le code exécuté à la mise sous tension est la fonction **reset.handler** défini dans le fichier **reset-handler-sequential.s**.

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

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

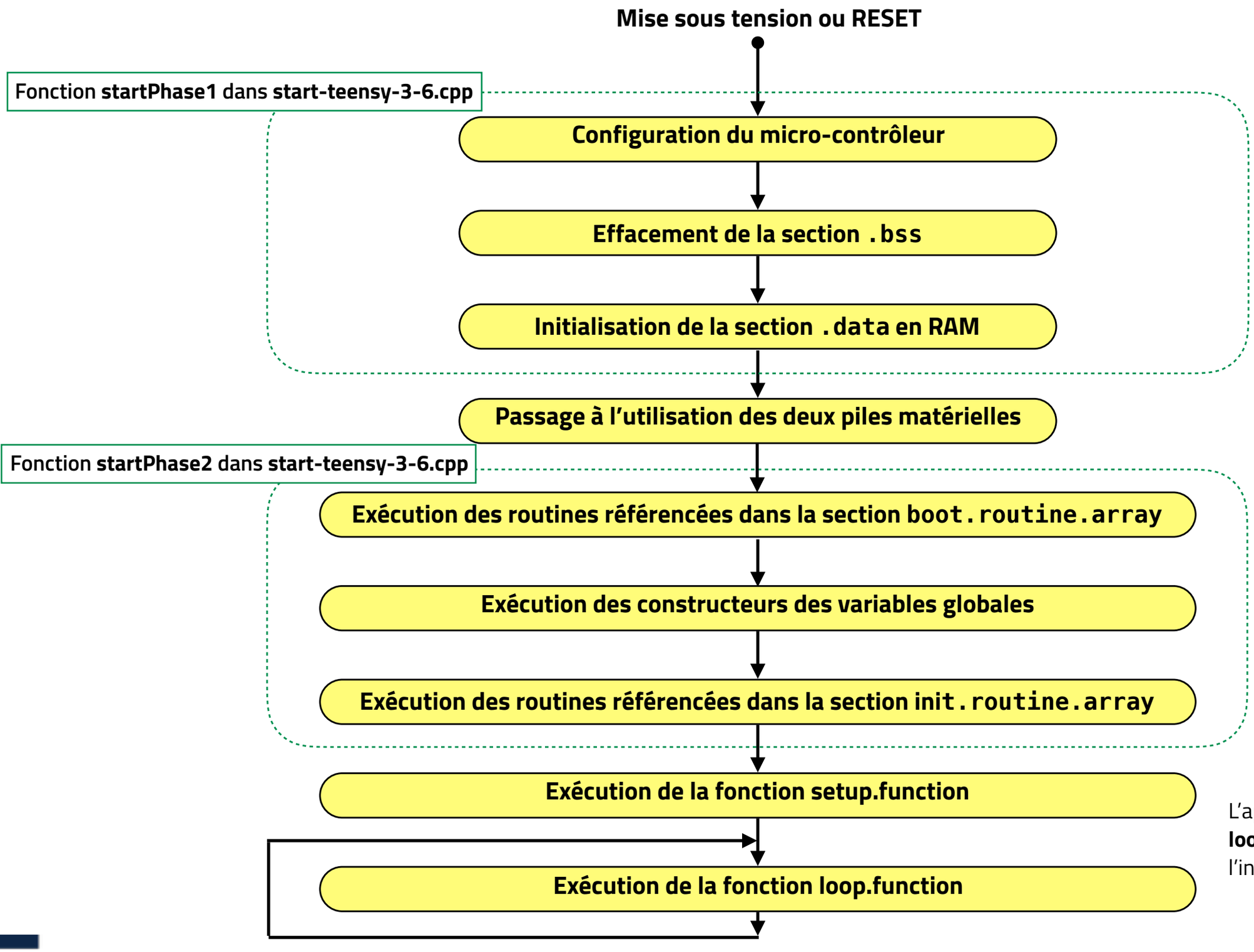
Exécution de la fonction C++ **startPhase1** (définie dans **start-teensy-3-6.cpp**)

Passage à l'utilisation des deux piles matérielles

Exécution de la fonction C++ **startPhase2** (définie dans **start-teensy-3-6.cpp**)

Exécution des fonctions C++ **setup** et **loop** (définies dans **setup-loop.cpp**)

Organigramme d'exécution du programme (1/2)



L'appel des routines **setup** et **loop** disparaîtra avec l'introduction de l'exécutif.

Organigramme d'exécution du programme (2/2)

Configuration du micro-contrôleur. Le micro-contrôleur démarre sur une horloge interne à 8 MHz ; cette routine programme l'horloge du processeur, du bus, du *Flexbus*, et de l'accès à la Flash à partir de la valeur de l'horloge processeur indiquée par le champ **CPU-MHZ** du fichier **makefile.json**.

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 `setup.function` et `loop.function`. Ces symboles assembleur correspondent aux fonctions C++ **setup** et **loop**, déclarées dans **setup-loop.h** et implémentées dans **setup-loop.cpp**.

Fonctions `setup` et `loop`

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

```
#pragma once
void setup (void) asm ("setup.function") ;
void loop (void) asm ("loop.function") ;
```

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.

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

```
#include "all-headers.h"
// Led L2 is connected to PORTD:7 (active high)

void setup (void) {
//--- Configure PTD7 as digital port (input or output)
  PORTD_PCR (7) = PORT_PCR_MUX (1) ;
//--- Configure PTD7 as digital output port (output level is low --> led is off)
  GPIOD_PDDR |= (1 << 7) ;
}

void loop (void) {
//--- Drive PTD7 high --> led is on
  GPIOD_PSOR = 1 << 7 ;
//--- Wait...
  for (volatile uint32_t i=0 ; i< 10 * 1000 * 1000 ; i++) {}
//--- Drive PTD7 low --> led is off
  GPIOD_PCOR = 1 << 7 ;
//--- Wait...
  for (volatile uint32_t i=0 ; i< 10 * 1000 * 1000 ; i++) {}
}
```

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-M4.

L’édition de liens (1/5)

Le fichier d'édition de liens **dev-files/teensy-3-6.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 = 0, LENGTH = 1024k
  sram (rwx) : ORIGIN = 0x1FFF0000, LENGTH = 256k
}
```

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 (2/5)

Ce bloc est placé dans la Flash (« **`> flash`** » à la dernière ligne) ; comme c'est le premier bloc à placer dans la Flash, il est placé à partir de l'adresse `0x0`.

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

`KEEP (*(isr.vectors))` ordonne de placer les sections **`isr.vectors`** 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 : {
  /*-------------------- Vectors */
    __vectors_start = . ;
    KEEP (*(isr.vectors)) ;
    __vectors_end = . ;
  /*-------------------- Flash magic values */
    LONG (-1) ;
    LONG (-1) ;
    LONG (-1) ;
    LONG (-2) ;
  /*-------------------- 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 (3/5)

Ce bloc est placé dans la RAM (« **`> sram`** » à la dernière ligne), et contient 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***.

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

L'édition de liens (4/5)

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

- réserver 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.***

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

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

L’édition de liens (4/5)

Ce bloc réserve 1024 octets pour la pile système. 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 :{
    . = ALIGN (8) ;
    __system_stack_start = . ;
    . += 1024 ;
    . = ALIGN (4) ;
    __system_stack_end = . ;
  } > sram
}
```

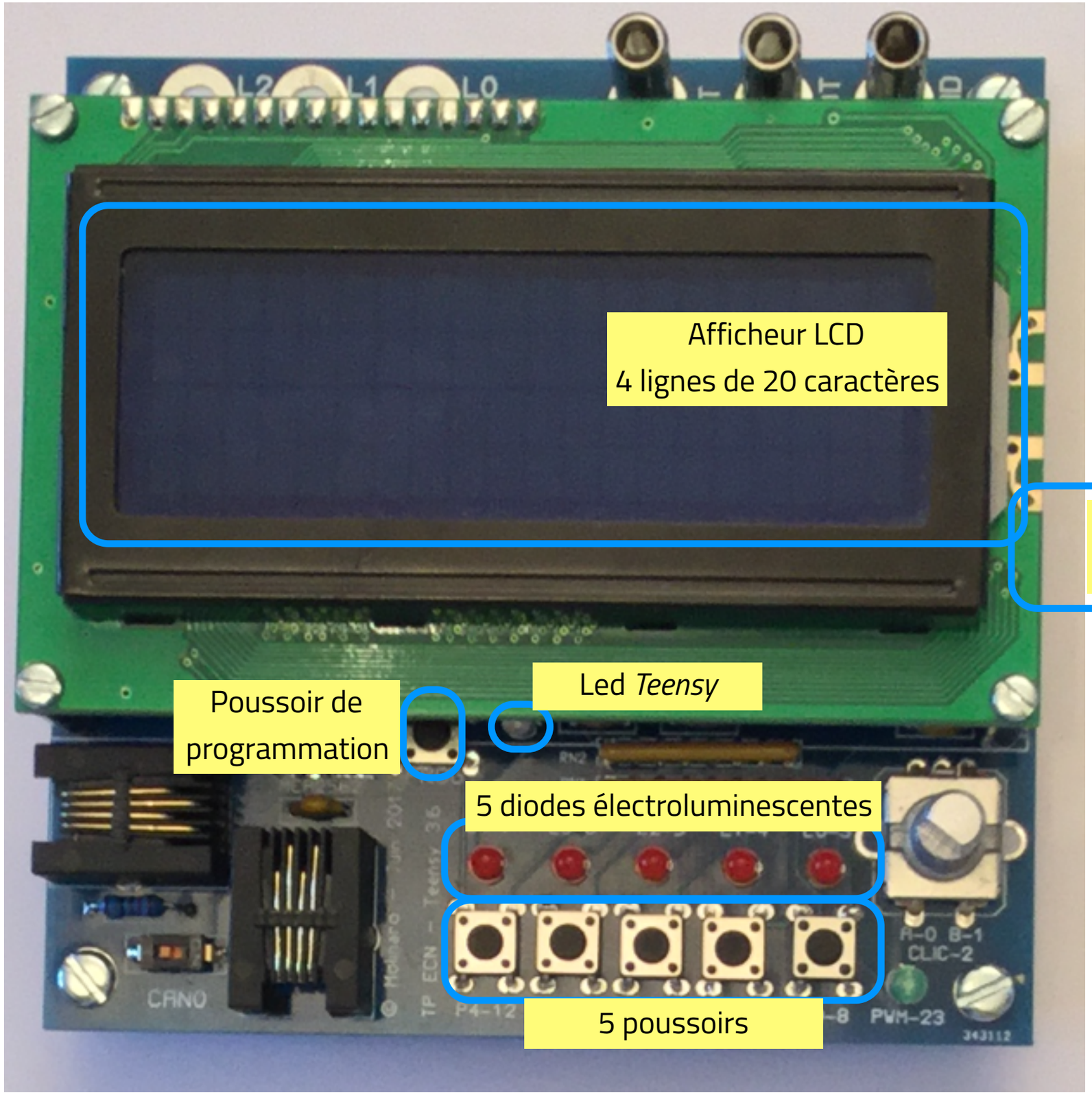
L’édition de liens (5/5)

Le reste de la mémoire est allouée au *tas* (*heap*), c’est-à-dire pour l’allocation dynamique. L’allocation dynamique sera présentée à l’étape 16. Les symboles **`__heap_start`** et **`__heap_end`** encadrent cette zone.

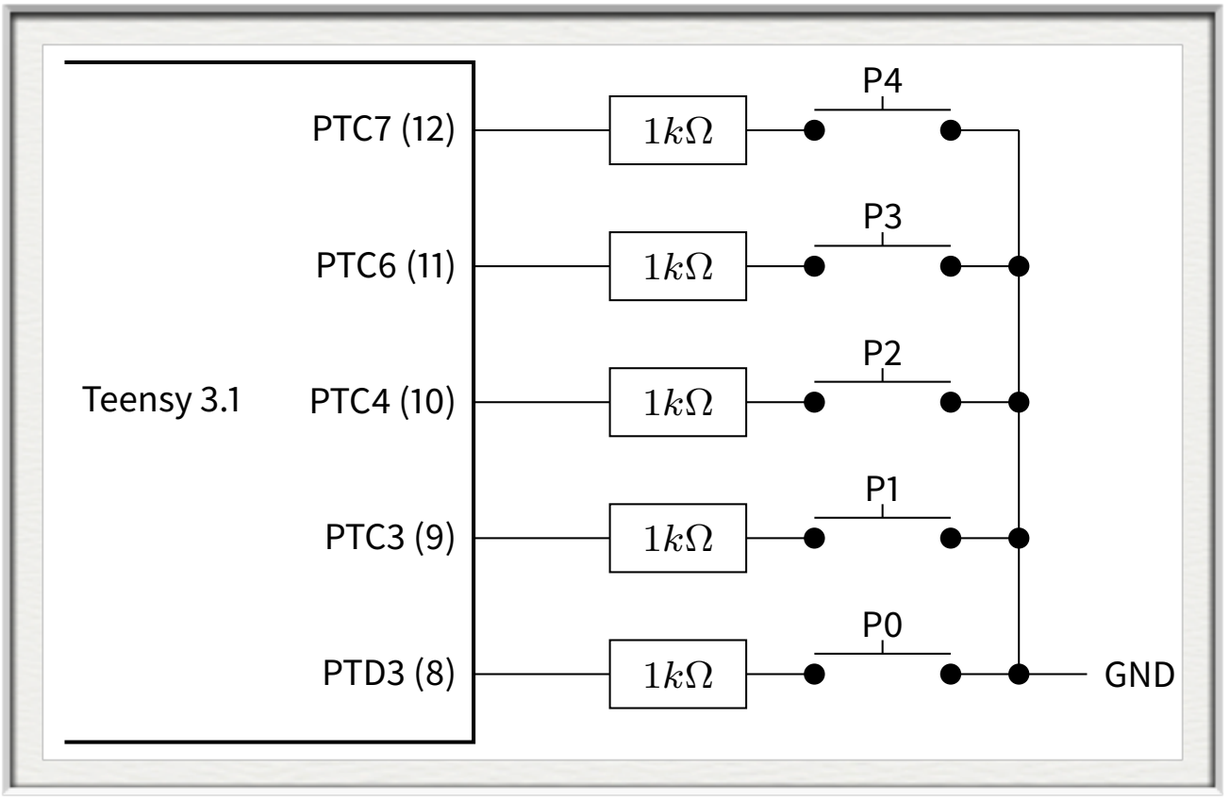
```
SECTIONS {
  .heap : {
    . = ALIGN (8) ;
    __heap_start = . ;
  } > sram
}

__heap_end = ORIGIN(sram) + LENGTH(sram) ;
```

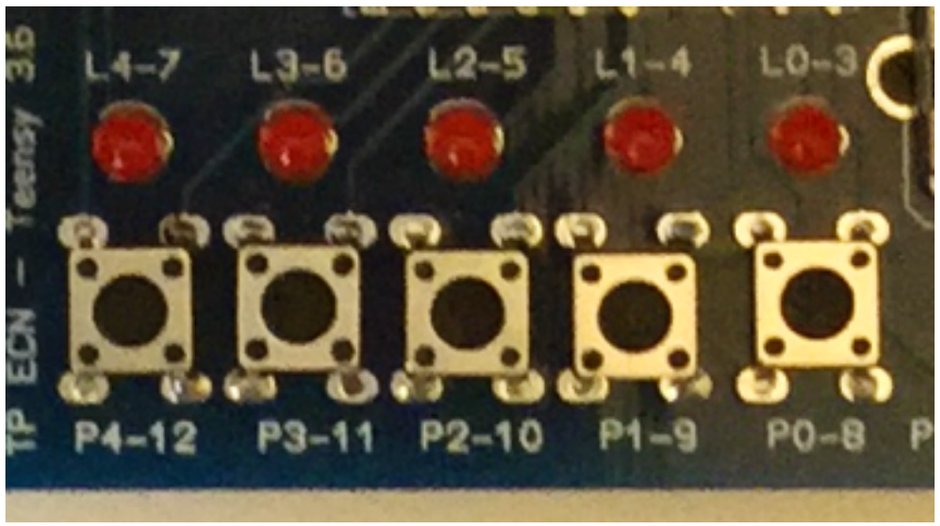
5 — Carte de développement

Entrées / sorties de la carte de TP



Connexion des entrées logiques



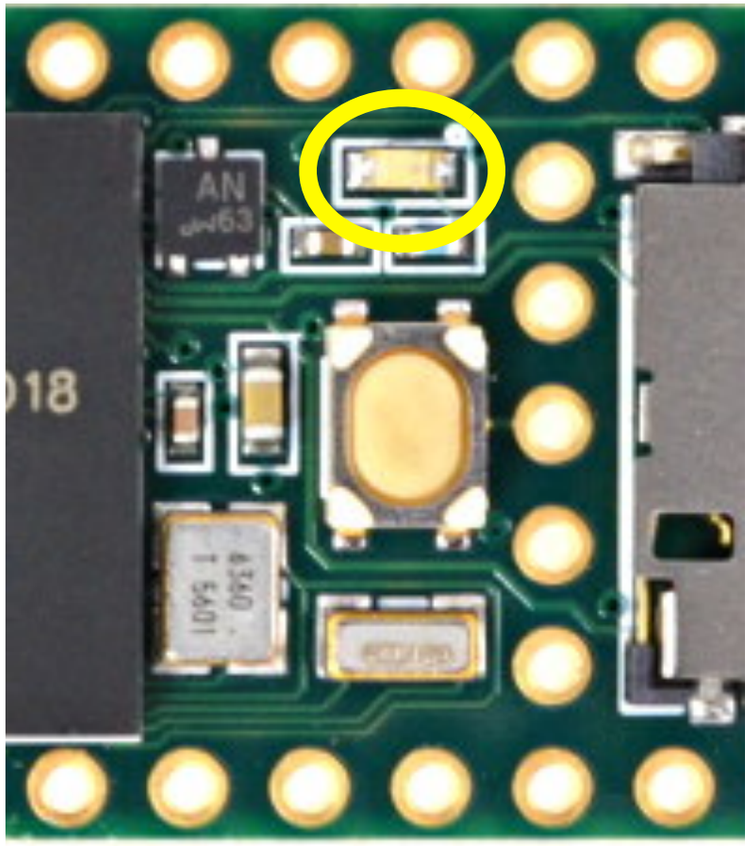


Poussoir `P0` appuyé : le port #8 est au niveau bas.

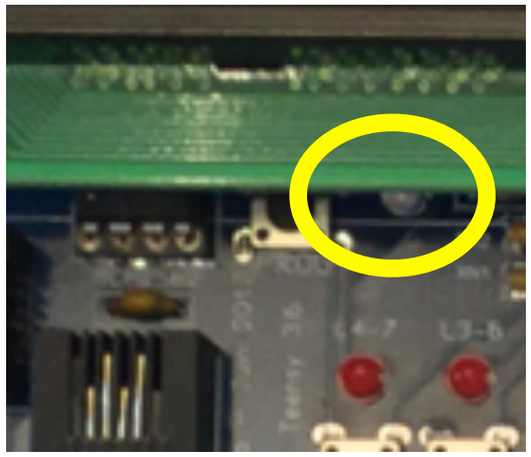
Poussoir `P0` relâché : le port #8 est au niveau haut, sous réserve qu'il soit programmé en mode `INTPUT_PULLUP`.

Diode électroluminescente sur la carte Teensy

Led Teensy



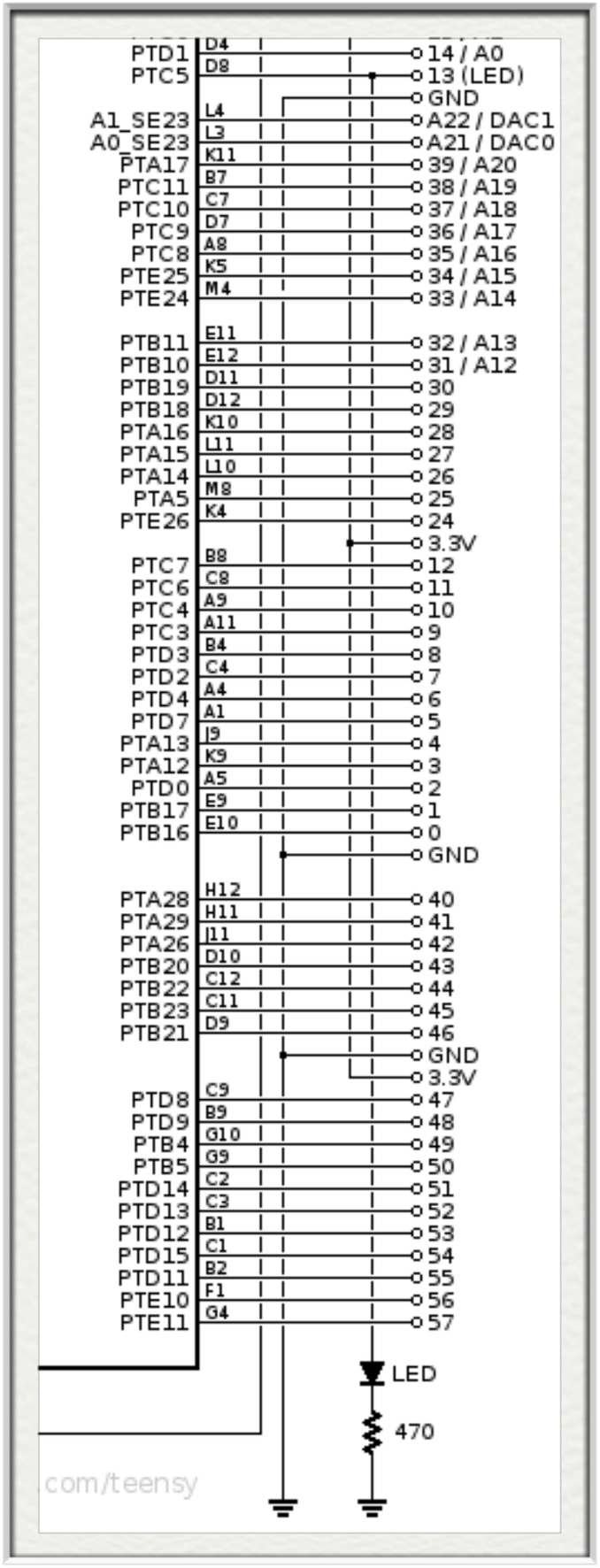
Comme cette led est masquée par l'afficheur LCD, elle est répétée sur la carte de développement.



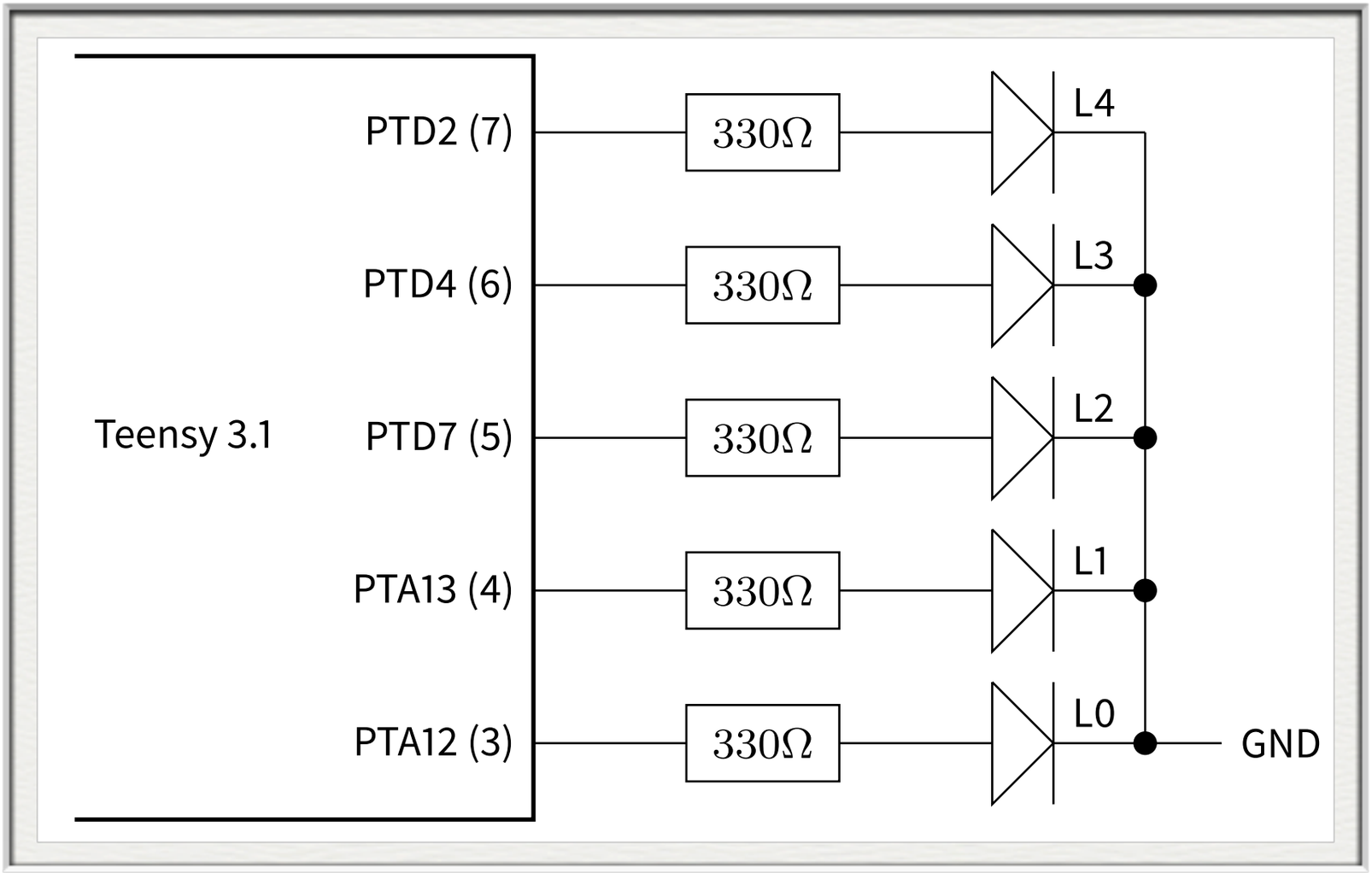
Note : cette led sera utilisée dans ce cours pour refléter l'activité processeur.

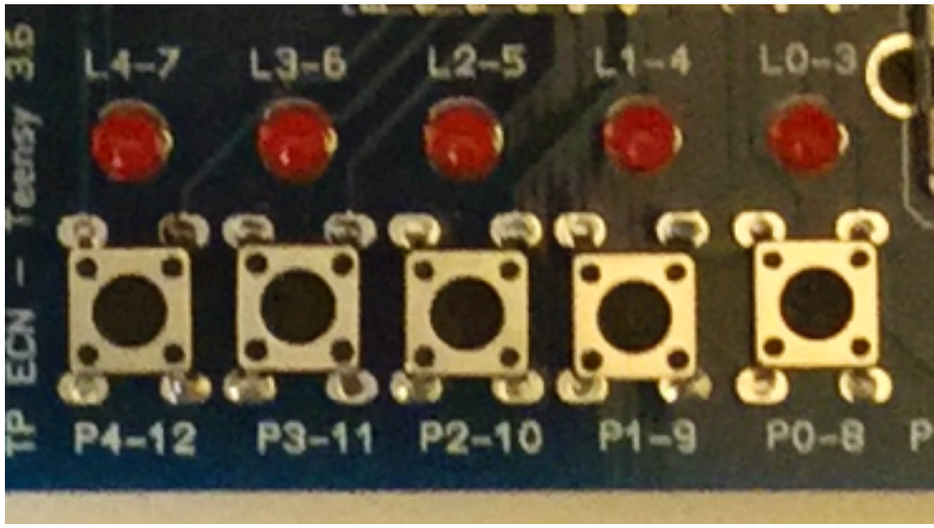
Port n°13 en sortie logique au niveau bas : le micro-contrôleur impose une tension proche de zéro Volt, la led est éteinte.

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



Diodes électroluminescentes sur la carte de TP





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.

6 — Complément : 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	$a_7 \oplus b_7$	$a_6 \oplus b_6$	$a_5 \oplus b_5$	$a_4 \oplus b_4$	$a_3 \oplus b_3$	$a_2 \oplus b_2$	$a_1 \oplus b_1$	$a_0 \oplus b_0$

	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
~ A	¬ a_7	¬ a_6	¬ a_5	¬ a_4	¬ a_3	¬ a_2	¬ a_1	¬ a_0

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

L’opérateur « complément logique » : « ! »

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 ≠ n**.

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