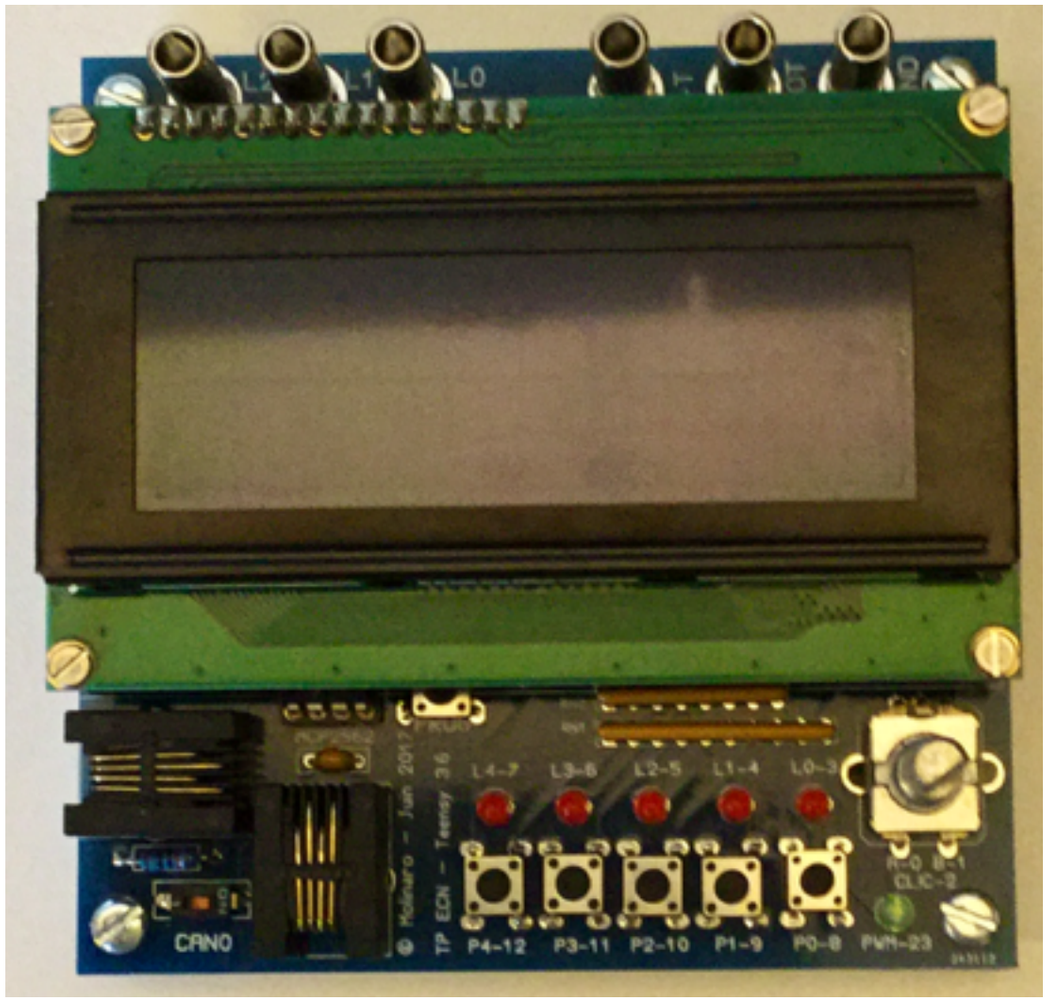
Temps Réel



Étape 04-boot-init-routines

Pierre Molinaro

Juin 2018

Description de cette étape

Objectif :

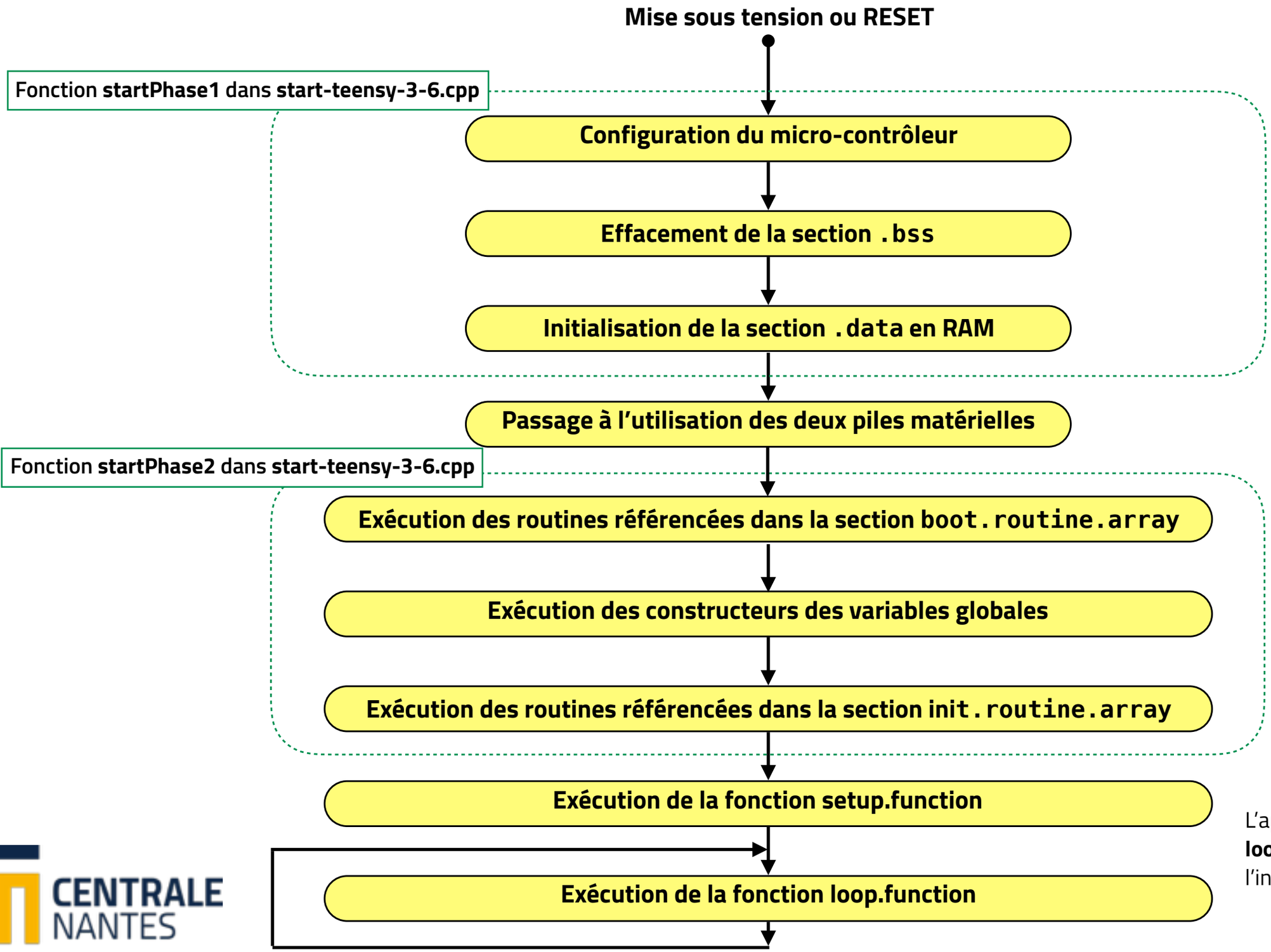
- décrire les différentes routines qui sont exécutées au démarrage ;
- montrer comment inscrire une routine pour exécution implicite au démarrage.

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

CENTRALE NANTES

Organigramme d'exécution du programme

Rappel de l'étape 01



Dans cette partie, nous allons nous intéresser à ces trois exécutions

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

CENTRALE NANTES

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

CENTRALE NANTES

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/teensy-3-6.ld`** :

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

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 **`startPhase1`** (fichier **`start-teensy-3-6.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/teensy-3-6.ld`** :

```
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) ;
```

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 **`startPhase1`** (fichier **`start-teensy-3-6.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/teensy-3-6.ld** :

```
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.

L'appel des ces fonctions est « *Exécution des constructeurs des variables globales* », et effectuée dans la routine **startPhase2** (fichier **start-teensy-3-6.cpp**) :

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

CENTRALE NANTES

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/teensy-3-6.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 **startPhase2** de **start-teensy-3-6.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
  SYST_RVR = CPU_MHZ * 1000 - 1 ; // Underflow every ms
  SYST_CVR = 0 ;
  SYST_CSR = SYST_CSR_CLKSOURCE | SYST_CSR_ENABLE ;
}
```

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 **setup** (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 ;
```

CENTRALE NANTES

Routines BOOT et INIT (3/4)

Il reste quelques détails à régler.

D'abord, la fonction **startSystick** n'a pas être 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 ;
```

CENTRALE NANTES

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_ROTINE (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 **start-teensy-3-6.cpp**, de façon à inscrire la fonction **startSystick** parmi les routines **boot**.

CENTRALE NANTES