Règles et bonnes pratiques de l'UE ELEC223

Alexis Polti

TELECOM ParisTech

Template : Felix Breuer

Article 1 : un processeur ne plante pas

Un processeur exécute ce qui se trouve en PC puis passe à l'instruction d'après. Sauf en cas de double faute, où il passe en mode erreur.

L'emploi d'un débugger au niveau assembleur pour comprendre ce qui se passe doit être un réflexe :
- exécution pas à pas en assembleur
- examen des registres à chaque pas
- examen de la mémoire à chaque pas

GDB quick ref card : `https://sourceware.org/gdb/download/onlinedocs/refcard.pdf.gz`

Article 2 : les types qu'on utilisera

`int` est le type "naturel" du processeur, celui dont les manipulations sont les plus rapides. On l'emploiera quand on n'a pas spécifiquement besoin d'un autre type.

`char` sera utilisé pour manipuler des caractères ou des chaînes de caractères.

Si on souhaite des type de taille précise, on utilisera les types de C99, définis dans `stdint.h` :

```
int8_t / uint8_t
int16_t / uint16_t
int32_t / uint32_t
int64_t / uint64_t
```

Article 3 : C99

La version du standard C utilisé sera celle de 1999. On utilisera si nécessaire les extensions GNU.

On compilera donc avec le flag `-std=gnu99`

Toujours **commenter** son code.

```
/* Ce fichier implémente telle et telle
   fonctionnalité super intéressante.
*/

// Recherche d'un chemin par l'algo A*
void A_star(void) {
  blabla();
  // Cette partie utilise telle optimisation : ...
  reblabla();
}
```

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

Toujours **indenter** son code.

Makefile

Les compilations, comme toutes les tâches répétitives, seront automatisées avec `make`.

Édition de code

L'édition de code se fera avec un éditeur de code comme emacs et dérivés, ou bien vi et dérivés.

On appendra à utiliser ces éditeurs au **clavier uniquement**, sans souris. Pour cela, vous aurez les ETOTD / VTOTD (Emacs / Vi Tip Of The Day).

Tous les fichiers textes, code y compris, seront encodés en **UTF8 sans BOM**.
`file` pour connaître l'encodage d'un fichier , et `iconv` ou `recode` pour le modifier.

Article 7 : où écrire les choses

Objets globaux privés

Ils seront définis `static` et uniquement dans un fichier source (.c). Jamais dans un header.

Seule exception : les fonctions `static inline` qui peuvent être définies dans un header. Chaque source C incluant ce header a alors sa propre copie privée de la fonction.

Objets globaux exportés

Ils seront définis (instanciés) dans un et un seul fichier source.

Ils seront aussi déclarés dans le header associé, qui sera importé par **tous** les sources C utilisant ces objets.

Unique exception : un symbole peut être instancié dans deux fichiers du moment que l'une des instanciation est déclarée `weak`.

Avant tout

Pas de magie noire : tout ce qui écrit doit être compris.

Prenez le temps d'apprendre.

Posez des questions.

Article 4 : inclusions cycliques

On insèrera en en-tête de chaque header des gardes, pour les protéger contre les inclusions multiples.

git

Toujours travailler directement dans son dépôt git.

Ne **jamais** commiter de fichiers résultats de compilation :
- fichiers objets
- binaires

Pour cela, ne jamais faire de `git add` d'un répertoire complet, toujours vérifier par `git status` ce qu'on s'apprête à commiter, et utiliser les `.gitignore`.

Les messages de commit doivent être explicites et clairs.

git

Toute production électronique doit être stockée dans votre dépôt **git**.
Pas d'échange de code par mail, même pour poser une question.
Au besoin, utiliser les **gist** ou équivalents.

Faire des `git commit` fréquemment (10 minutes max).
Faire des `git push` fréquents aussi.

Un travail n'a pas besoin d'être terminé pour faire un commit / push : utiliser les messages **"WIP"** (work in progress)

Pour avoir un historique "propre", utiliser les branches.

Article 5 : type modifiers

Tout ce qui est constant sera déclaré `const`.
Tout ce qui est volatile sera déclaré `volatile` (!)

Article 6 : registres mappés en mémoire

Pour accéder à un registre mappé en mémoire, on utilisera la construction suivante : `#define REG (*(volatile uint32_t *)0xff00ff00)`

Pour accéder à des bancs de registres, il existe d'autres constructions tout aussi élégantes, qui seront étudiées dans d'autres UE (ELECINF344 / ELECINF381).

Article 8 : optimisations

Il n'y a aucune bonne raison pour compiler en `-O0`.
On compilera donc toujours avec :
- `-O1` pour du code facile à lire,
- `-O2` pour les optimisations standards, mais avec un code plus difficile à suivre.

Article 9 : warning = intervention obligatoire

Lorsqu'il ne sait pas décider si une construction est correcte ou fausse, le compilateur émet un warning pour demander une intervention humaine.

On compilera donc toujours avec `-Wall -Wextra -Werror`