TP 3 Compilation et débogage

Nous allons aujourd'hui analyser et modifier le contenu des fichiers binaires produits par la chaîne de compilation, et comprendre le rôle des sections et des symboles dans les fichiers ELF.

1 Compilation

1. Revenez au projet `1_blinky` fourni au tout premier cours (copiez vos versions modifiées et faites "`git reset -hard`" ; attention, toute modification au dépôt sera perdue !). Compilez-le.
2. Listez les symboles de `main.o`. Maintenant désactivez la production de symboles de déboggages et recommencez. Qu'observez-vous ?
3. Affichez les sections de `main.o`. Comment expliquer la taille des sections `.data` et `.bss` ?
4. Rajoutez à `main.c` une variable globale initialisée `volatile int x[4] = {1,2,3,4};`. Quel est son effet sur la taille des sections ? Affichez le contenu de la section qui a grossi.
5. Affichez le contenu de la section `.comment`.
6. Rajoutez à `main.c` une variable globale *non* initialisée `volatile int y[4];` et changez la condition de boucle `while(1)` en `while(y[0]==0)` [1]. Quel est l'effet de cette déclaration sur la taille de la section `.bss` de `main.elf` ?

2 Edition des Liens

1. Affichez l'en-tête ELF de `main.o` ; s'agit-il d'un *relocatable* ou d'un *executable* ? Même question pour `main.elf`.
2. Listez les symboles du code de démarrage assemblé `startup-stm32f303.o`. Quels sont les symboles qui ne sont pas associé à une donnée (UNDEFINED) ? Cherchez où sont définis chacun d'eux au moment de l'édition des liens.
3. L'outil `arm-none-eabi-strip` supprime les sections et les symboles inutiles des fichiers ELF. Observez l'effet de `strip -g` sur les objets avec symboles de déboggage (vous devrez réactiver la production de ces symboles).
4. Que se passe-t-il si vous appliquez `strip -s` à `main.o`, puis essayez d'éditer les liens ? Pourquoi ?

3 Débogage

1. Lancez le serveur de débogage (`st-util -v`), puis en parallèle le client gdb (`make debug`). Chargez l'exécutable ELF sur la carte (commande `lo`) et placez des *breakpoints* (e.g., `br main.c:22`) ; lancez le programme (`c`) ; l'exécution s'arrête au premier *breakpoint* rencontré. Observez le contenu de variables locales/globales (si vous en avez) ou de registres (e.g., `p GPIOE->ODR`). Familiarisez-vous avec toutes les commandes vues en cours. [2]

1. Ce test sera toujours vrai et donc la sémantique du programme est inchangée ; cela empêche simplement le compilateur d'ignorer la variable `y`, car elle est maintenant utilisée dans le programme
2. C'est important, vous en aurez largement besoin dans le reste de cet enseignement !