Résumé Systèmes Embarqués et Robotique

Groupe Semaine 2
Aude Chigard, Daniel Da Silva, Georg Schwabedal & Dominic Stratila

Mars 2024

1 Structure du compilateur C

1.1 Compilateur C

Ce cours se base sur le compilateur GCC en mode croisé. GCC est le compilateur C du projet GNU, démarré en 1984 pour réaliser un OS similaire à UNIX mais libre. Le développement de GCC a débuté en 1987 et on est actuellement à la version 12.2 , le projet fait 15M lignes de code. GCC est soumis à la licence GPL (General Public License). GCC supporte la grande majorité des processeurs existants à ce jour, est modulaire et permet de la compilation croisée (cross-compilation) pour un grand nombre de cibles.
Le compilateur C prend le code écrit en C et le transforme en code assembleur, ensuite un encoding est réalisé pour pouvoir arriver à l'exécution.

Compiler
Un compilateur est un programme informatique (ou un ensemble de programmes) qui transforme un code source écrit dans un langage informatique (le langage source) en un autre langage informatique (le langage cible). Le nom "compilateur" est principalement utilisé pour les programmes qui traduisent le code source d'un langage de programmation de haut niveau vers un langage de niveau inférieur (par exemple en langage assembleur) pour créer un programme exécutable.

Encoding
Pour que les instructions assembleur puissent être exécutées par le microcontroleur de l'e-puck 2, elles doivent être traduites. En effet, le microcontroleur principal ne peut pas lire l'assembleur et donc notre code source directement. Cet encodage est fait par le programme AS du compilateur. Les différents bouts de code binaire ainsi générés sont ensuite passés à l'éditeur de lien pour le mettre ensemble.

Linker (éditeur de liens)
En informatique, un éditeur de liens est un programme qui prend un ou plusieurs objets générés par un compilateur et les combine en un seul programme exécutable.

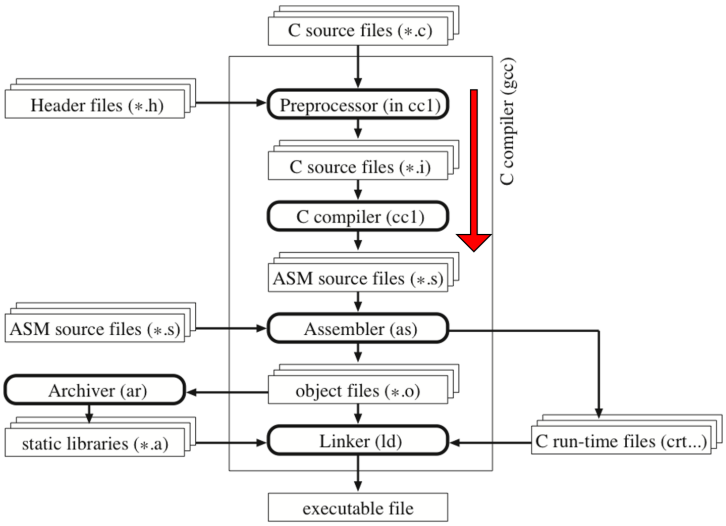


Figure 1: Structure du compilateur C

Du fichier ".c" à l'exécutable

Lorsqu'on compile notre code, les fichiers source ".c" et les fichiers header ".h" passent par une étape de pré-processeur où ces fichiers sont mis ensemble et transformés en seul fichier ".i" (un fichier ".i" par paire de fichier ".c" et ".h"). C'est aussi lors de cette étape que les directives "define" sont remplacées par leur valeurs respectives. Puis, les fichiers ".i" sont compilés et transformés en des fichiers assembleur ".s" . Enfin les fichiers assembleur sont transformés en fichiers objets ".o".

Les fichiers crt (C run-time files) sont mis ensemble avec les fichiers objets et les librairies statiques lors de la dernière étape de l'éditeur de liens et donnent le fichier executable. Ces fichiers crt permettent d'initialiser les éléments nécessaires (comme les pointeurs, la mémoire, etc) afin de pouvoir démarrer l'exécutable correctement.

Il est possible d'effectuer seulement certaines étapes de ce processus avec certaines commandes précises dans le compilateur. Par exemple, la commande "gcc -S" permet au compilateur de s'arrêter au fichier assembleur ".s"

Il est important de mentionner, qu'un code source en C peut donner des instructions assembleur différentes en fonction du compilateur et des paramètres d'optimisation. Par ailleurs, le code assembleur généré par le compilateur n'est jamais optimal (sauf pour des codes très simples).

1.2 Cross-compilateur C

On parle de cross-compilateur quand le code est généré pour un autre processeur que celui sur lequel tourne le compilateur, par exemple on compile un programme sur un ordinateur pour qu'il tourne sur un robot. Afin de distinguer gcc (propre à l'ordinateur) des cross-gcc on ajoute un préfix au nom de gcc (exemple : pic30-gcc, xc16-gcc, arm-none-eabi-gcc ...). Dans le cadre de ce cours,on compile du code pour des microprocesseurs ayant une architecture de la famille ARM Cortex-M.

La suite d'outils de gcc prennent la même forme (arm-none-eabi-ld, arm-none-eabi-as, etc.)

2 Organisation de code

Une solide compréhension de la structuration d'un code est indispensable pour assurer un fonctionnement optimisé et une méthodologie de travail qui favorise le développement rapide et sans bugs, ainsi qu' une reprise efficace du code.

2.1 Compilation grâce au makefile

La compilation d'un projet peut être réalisée soit par des commandes ciblées de compilation, soit par la commande "make" qui génère l'exécutable grâce à l'invocation du fichier "makefile". Il crée un exécutable au format Executable and Linkable Format (ELF) utilisable par la suite. Enfin, le "makefile", s'il est conçu de manière judicieuse, peut inclure d'autres "makefiles" et sélectionner uniquement les fichiers modifiés, évitant ainsi une recompilation complète.

2.2 Le rôle et l'optimisation des fichiers

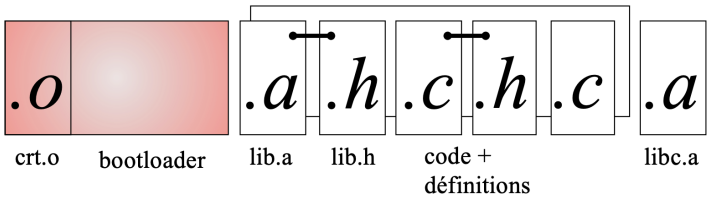


Figure 2: Structure de fichiers

La conception d'un code implique la distinction entre les fichiers source (.c) et les fichiers d'en-tête (.h). Cette organisation est cruciale pour garantir une encapsulation efficace des tâches. Lors de la compilation ces fichiers sont regroupés dans un seul fichier. Des fichiers finalisés et testés peuvent être convertis en une bibliothèque (.a) et inclus dans une archive. Cette bibliothèque est alors placée au même niveau que les bibliothèques standard du langage C et n'est pas recompilée.
Le fichier crt.o, fourni par les développeurs du système (ici le robot e-puck), assure la mise en place initiale du système, comme par exemple la configuration des entrées-sorties au niveau de la fonction main().
Enfin, la région de bootloader est une zone de code constante située au début de la mémoire flash de certains systèmes protégée à la réécriture. Elle permet entre autres la reprogrammation du système sans avoir de hardware spécifique (comme dans le robot le deuxième microcontrôleur).

2.3 BIOS/OS

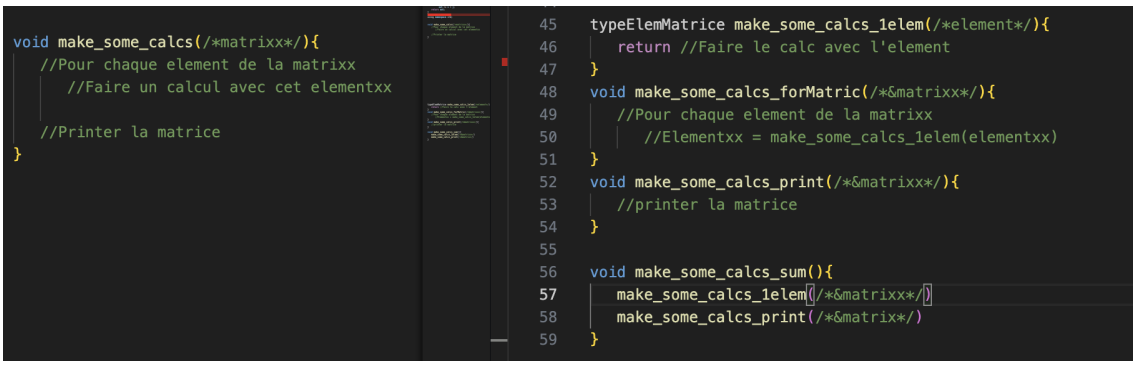
Le BIOS (Basic Input/Output System) indique le regroupement des fichiers crt.o, bootloader, et des autres fonctionnalités de base du système. Il est présent dès le démarrage. Pour le robot e-puck c'est le code qui gère la lecture des capteurs par exemple. Le OS (Operating System) regroupe la gestion de fichiers, mémoire, tâches et Input/Output. Ensuite vient le code de notre application pour réaliser des tâches spécifique.

3 Abstraction matérielle

Le but d'une abstraction dans l'écriture du code est d'écrire du code court et clair pour le codeur. Ceci est essentiel pour un code propre, ce qui facilite l'écriture, la lecture(surtout après un long intervalle de temps, mais aussi pour d'autres lecteurs) et le debug. On pourrait écrire tout en assembleur ou même qu'avec des 1 et des 0, mais cette manière est évidemment plus longue et lourde.
Cette abstraction peut être réalisée avec différentes méthodes complémentaires:

- Les mécanismes de redéfinition de noms (define)
- Les structures ou les classes(absentes en C, mais pas dans C++)
- La division des longues fonction en fonctions courtes et simples qui font une seule tâche mieux délimitée.

Il est aussi essentiel de nommer les variables et les fonctions avec des noms représentatifs. C'est mieux d'avoir un nom long, mais représentatif et clair, qu'un nom court mais qui n'explique rien en soi(ex: "int aireDuRectangle" vs "int a").
Dans le cas d'une fonction longue qui fait plusieurs tâches ou contient plusieurs étapes (ex: (1) faire un certain calcul (2) pour tous les éléments d'une matrice et puis (3) printer la matrice) c'est mieux de la subdiviser en plusieurs, où chacune fait une tache précise et courte (ex: une fonc . séparée pour (1), (2) et (3), et puis les assembler ensemble; une fonc-sommaire avec deux fonctions, où la première(1) appelle la (2) pour chaque élément, et puis la (3) fait le print). Voir Figure.3.



```
void make_some_calcs(/*matrixx*/){
    //Pour chaque element de la matrixx
        //Faire un calcul avec cet elementxx

    //Printer la matrice
}
```

```
45  typeElemMatrice make_some_calcs_1elem(/*element*/){
46      return //Faire le calc avec l'element
47  }
48  void make_some_calcs_forMatric(/*&matrixx*/){
49      //Pour chaque element de la matrixx
50          //Elementxx = make_some_calcs_1elem(elementxx)
51  }
52  void make_some_calcs_print(/*&matrixx*/){
53      //printer la matrice
54  }
55
56  void make_some_calcs_sum(){
57      make_some_calcs_1elem(/*&matrixx*/)
58      make_some_calcs_print(/*&matrix*/)
59  }
```

Figure 3: Exemple de deux réalisations d'une tâche