INF1900

Projet initial de système embarqué
Travail pratique no 7
Mise en commun du code et formation de librairies

Objectif: Faire le point sur le projet. Introduction au travail technique en équipe de quatre. Regrouper le code développé. Introduction aux librairies de code avec les Makefiles.
Durée: Une semaine
Travail préparatoire: Code des semaines précédentes
Documents à remettre: Code complet et rapport à remettre avant le ~~jeudi 28 février 23h00~~ **dimanche 3 mars 23h00 NOUVEAU**.
Présentation en classe: fichier LibreOffice Impress

> «La définition d'ouvert: "mkdir android ; cd android ; repo init -u git://android.git.kernel.org/platform/manifest.git ; repo sync ; make"»
>
> *-Andy Rubin, ancien responsable d'Android chez Google, répondant à Steve Jobs d'Apple qui critiquait l'aspect «fragmenté» du produit compétiteur du iPhone*

Introduction

Vous devez maintenant vous joindre à une autre équipe afin d'en former une seule comptant quatre membres. Initialement, il faut faire le point et réviser les programmes rédigés jusqu'à maintenant par chacun des membres afin d'identifier les principaux défauts et surtout les meilleures productions. Il est également important que vous discutiez de la manière dont vous entendez procéder d'ici la fin de la session.

Le but de l'exercice est de faire un peu de «ménage» dans votre code et de voir ce que vous voulez conserver et ce que vous devez rejeter. Avec le code conservé, il faudra former des librairies de manière à pouvoir utiliser ce code facilement sans avoir à recopier des fichiers ou faire des modifications à ces fichiers.

Entrepôts Git

La réunion de deux équipes de deux crée le besoin d'un entrepôt Git commun à quatre personnes. Il faut donc réunir le code développé précédemment et le placer dans un nouvel entrepôt. La procédure pour y arriver peut causer quelques problèmes. Il faut donc y aller délicatement. Un exemple nous servira à illustrer la démarche.

Supposons que les équipes 01 et 30 doivent être réunies. Supposons aussi que leurs anciens entrepôts soient inf1900-01 et inf1900-30 respectivement:

- Les équipes de deux feront leurs dernières soumissions de code (`git commit`) vers leurs anciens entrepôts s'il leur reste du code qui n'y est pas encore (si nécessaire).
- Les deux équipes de deux feront une dernière mise à jour de leur copie de code (`git pull`) à partir des anciens entrepôts (si nécessaire).
- Pour éliminer toute référence aux anciens entrepôts, il faut supprimer tous les répertoires cachés «`.git`» dans le répertoire racine de votre copie du code. Une commande «`rm -rf .git`» peut faire l'affaire après avoir fait un «`ls -a`» pour vérifier que le répertoire caché y est bien présent. Être bien réveiller avant de faire la commande de suppression car plus rien n'est réversible par la suite...
- Comme cette ancienne copie de code n'est maintenant plus qu'un ensemble de fichiers et de répertoires sans lien à aucun entrepôt, on peut l'importer (ou l'ajouter) dans un nouvel entrepôt. Il est préférable de la placer dans un sous-répertoire dont le nom indique tout de même son lieu d'origine (comme `branche-01` et `branche-30` par exemple). Il sera donc toujours possible d'accéder aux anciens fichiers. Par contre, l'historique des fichiers de l'ancien entrepôt est perdu. Il n'est plus possible de revenir à une ancienne version de ces fichiers, à moins de retourner à l'ancien entrepôt (qui resteront toujours présents d'ici la fin de la session de toute façon).
- Il est préférable de créer un nouveau sous-répertoire (appelé `codeCommun` par exemple), où le nouveau code sera placé. Le code réutilisé, réorganisé, et mis en commun peut aussi être placé quelque part sous ce répertoire, selon vos choix et vos décisions d'équipe.
- On aura donc une organisation qui ressemble à peu près à ceci:
```
https://githost.gi.polymtl.ca/git/inf1900-01 // ancien entrepôt de l'équipe 01
https://githost.gi.polymtl.ca/git/inf1900-30 // ancien entrepôt de l'équipe 30
https://githost.gi.polymtl.ca/git/inf1900-0130 // nouvel entrepôt commun
// copie de l'ancien entrepôt dans le nouveau (sans l'historique)
https://githost.gi.polymtl.ca/git/inf1900-0130/branche-01
// copie de l'ancien entrepôt dans le nouveau (sans l'historique)
https://githost.gi.polymtl.ca/git/inf1900-0130/branche-30
// code développé en équipe de 4 pour certains exercices
https://githost.gi.polymtl.ca/git/inf1900-0130/codeCommun
```
- Il est préférable de vérifier que tout s'est bien déroulé en prenant une copie fraîche (`git clone`) du nouvel entrepôt et ainsi s'assurer que tout est bien en place.

Formation de librairies avec les Makefiles

Une des meilleures façons de regrouper du code en vue de sa réutilisation dans plusieurs contextes demeure la formation de librairies ou bibliothèques logicielles. Par la suite, il suffit de lier ces librairies à un nouveau code développé pour éviter d'avoir à recopier le code des librairies. Depuis le début du projet, c'est par ce mécanisme que la programmation est facilitée avec l'utilisation de la librairie AVRLibC par exemple.

Il y a deux grandes catégories de librairies, les statiques et les dynamiques. Les librairies dynamiques sont plus complexes à gérer parce qu'elles peuvent être partagées par plus d'un exécutable. Sur Linux par exemple, presque tous les exécutables référencent la librairie libc. S'il fallait que chaque exécutable référence sa propre copie, l'espace mémoire total consommé serait très grand. C'est pourquoi le système d'exploitation n'en place qu'une seule en mémoire et tous les exécutables y font référence. Les librairies dynamiques ont une extension .so à leur nom sur Linux/Unix. Il y en a plusieurs dans le répertoire /usr/lib par exemple. Leurs équivalents sous Windows sont plus connus et se nomment DLL. Le dossier `C:\Windows\System32` en contient plusieurs (extension .dll) par exemple.

Les librairies statiques sont plus simples à produire. En fait, les librairies statiques ne sont qu'un vecteur de bouts de code compilés et disposés dans un fichier binaire. Lorsqu'on fait l'édition de liens pour former un exécutable, le compilateur va chercher dans le vecteur les morceaux de code dont il a besoin et les inclut à l'exécutable directement sans possibilités de les partager avec d'autres exécutables durant l'exécution. Avec un petit microcontrôleur tel que celui que nous utilisons au laboratoire, on a recourt uniquement aux librairies statiques puisqu'on ne charge qu'un seul exécutable dans la puce de toute façon. L'utilisation de librairies dynamiques n'a donc aucun sens dans ce contexte.

Un aspect important à souligner est qu'en INF1900, le compilateur utilisé à la base est GCC. En réalité, il s'agit d'un compilateur croisé (*cross-compiler*), ce qui signifie que le code produit n'est pas destiné à tourner sur le même système sur lequel s'exécute le compilateur (Linux dans le cas présent). Le code produit étant plutôt pour l'architecture Atmel AVR, c'est la raison pour laquelle certains exécutables dont il sera question dans la suite du texte sont précédés du suffixe «avr-». Il faut voir ici qu'il s'agit de faire cohabiter deux exécutables (gcc et avr-gcc dans notre cas) sur le même système Linux sans conflits. Par contre, les deux compilateurs se comportent à peu près de la même façon pour ce qui est des options qu'ils supportent en ligne de commande.

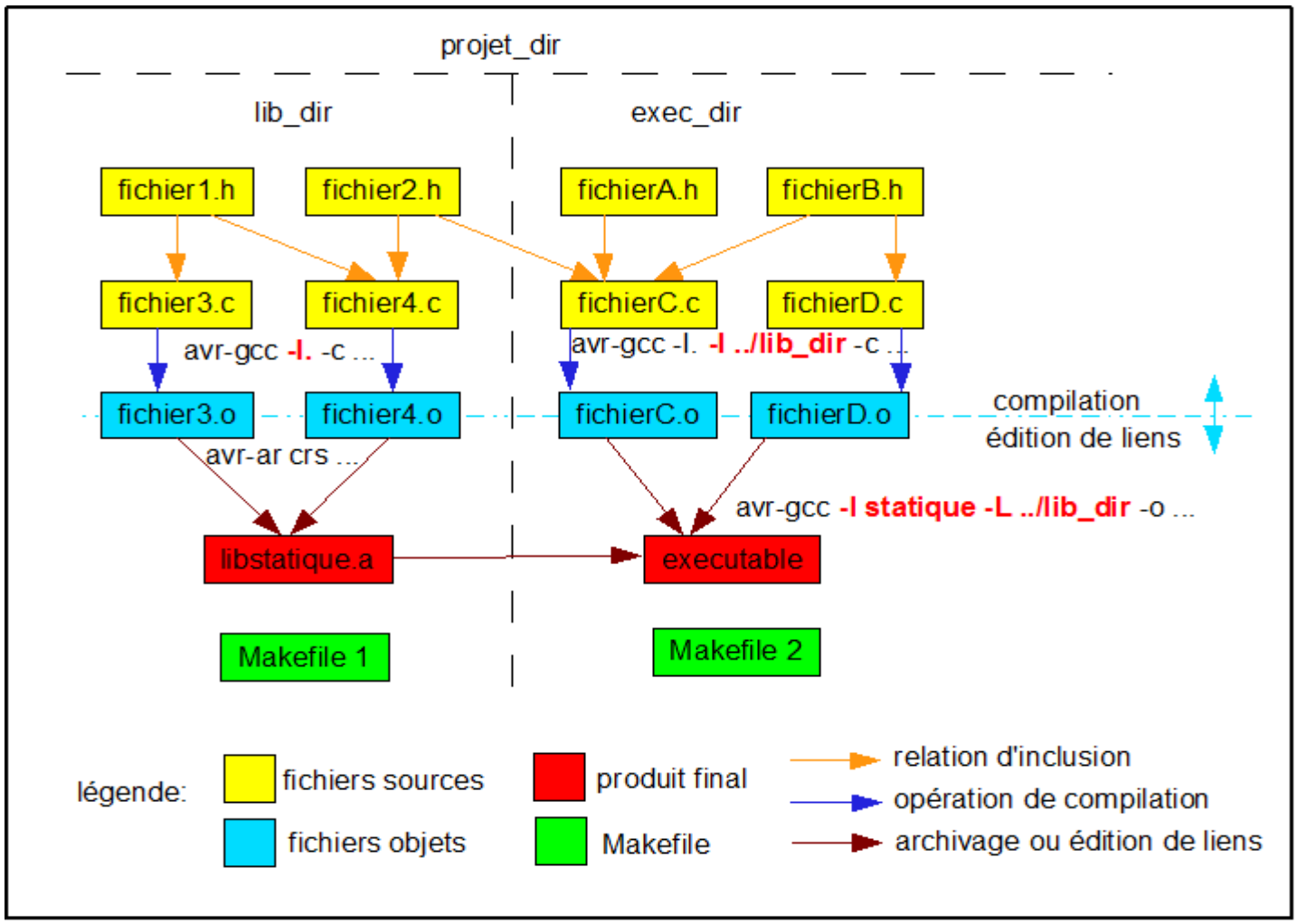
La question devient donc de savoir comment former une librairie et l'utiliser. Il peut y avoir différents moyens pour y arriver, mais la plupart du temps, le programme «make» et les fichiers Makefile (tel qu'introduit en LOG1000) sont à la base d'une solution élégante (le site http://gl.developpez.com/tutoriel/outil/makefile/ est très intéressant).

Les Makefiles sont basés sur l'exécution d'une série de règles organisées sous forme de dépendance. De cette façon, les actions sont exécutées uniquement lorsque nécessaires et dans le bon ordre. Il devient possible d'exécuter des tâches très variées comme par exemple télécharger un programme sur la carte mère utilisée au laboratoire ou exporter de la documentation vers un répertoire. Dans les faits, avec toute leur complexité et leur flexibilité, les Makefiles deviennent pratiquement un langage interprété permettant de «*scripter*» des commandes tels que bash ou python. Cependant, une telle utilisation dévie largement de l'esprit dans lequel l'outil a été créé.

De façon plus régulière, on se limite souvent à ces quelques principes simples:

1. Un seul Makefile par répertoire.
2. Les règles et variables communes à tout un système sont regroupées dans un seul fichier (appelé par exemple `Makefile.commun`) que tous les autres Makefiles dans les différents répertoires peuvent inclure (avec une instruction *include* au haut du fichier un peu comme on le fait en C/C++). Voir cette partie de la documentation de GNU Make pour ce cas particulier.
3. Chaque Makefile compile uniquement les fichiers sources présents dans le répertoire.
4. Chaque Makefile produit une seule librairie ou un seul exécutable dans le répertoire.
5. Des fichiers peuvent être copiés ou téléchargés ailleurs au besoin. On parle ici surtout des fichiers «include» (.h), les librairies (.a ou .so), les exécutables et la documentation. Presque jamais des fichiers sources C/C++.
6. Une règle (souvent appelée «clean») permet d'effacer les fichiers produits par la compilation (qui ne sont donc pas considérés comme étant les fichiers sources).
7. Il y a une règle par défaut qui correspond souvent à celle appelée «all».
8. Utiliser des exemples disponibles sur Internet!
9. Exécuter les Makefiles et regarder ce qui est produit à l'écran. Ajouter des instructions pour afficher plus d'informations au besoin ou appeler la commande `make` avec l'option `-d`.

Les principes 1 et 2 sont relativement simples à appliquer. Il en va de même pour les 5, 6, 7 puisqu'ils sont mis en application dans le Makefile utilisé depuis le début de la session. On peut donc consulter comment les règles sont écrites. Il est moins évident d'appliquer les principes 3 et 4 puisqu'il s'agit du cœur d'un Makefile! Les règles peuvent devenir complexes et il est bon d'utiliser les principes 8 et 9 pour les comprendre. Cependant, il est peut-être plus important encore de comprendre les fichiers et les outils impliqués dans le processus en se référant à la figure et aux explications suivantes:



a. Un fichier .c ou .cpp est souvent appelé «unité de compilation».
b. Chaque unité de compilation, une fois compilée, donne un fichier binaire avec extension .o.
c. Dans notre cas, le Makefile appelle avr-gcc pour chaque fichier source avec l'option -c pour produire un fichier .o, appelé aussi «fichier objet». C'est aussi à ce moment qu'on précise où se situent les fichiers «include» avec l'option -I (i majuscule!)
d. Les fichiers .o peuvent être regroupés pour former un seul exécutable. C'est ce qu'on appelle l'édition de liens (*linking*). L'éditeur de liens avr-ld est rarement appelé directement par le Makefile pour effectuer cette opération. **On préfère appeler avr-gcc avec l'option -o suivi du nom de l'exécutable et de la liste des fichiers .o.** On laisse le soin à avr-gcc d'appeler avr-ld (avec certaines options supplémentaires) pour réaliser l'opération.
e. Une fois l'exécutable produit, il peut être téléchargé sur la carte par l'utilitaire avrdude avec la règle «install» (ou en suivant les règles qui y mènent).
f. **Si on doit produire une librairie statique, on doit remplacer l'appel à avr-gcc qui produit l'exécutable par un appel à une commande qui se nomme «ar» (pour archive). Généralement, on utilise les options *c* (pour créer l'archive), *r* (pour insérer les fichiers membres) et *s* (pour produire en plus l'index des fichiers**

objets) avec cette commande. Une liste de fichiers objets et le nom de l'archive doivent aussi être passés en arguments. Le résultat produit par l'archive est un fichier avec l'extension . a. C'est la librairie elle-même.

g. **Pour utiliser la librairie en formant l'exécutable, il faut généralement donner son nom (avec l'option -l et, dans la plupart des cas, où elle se situe avec l'option -L). On passera donc ces options en plus des fichiers .o du répertoire courant à avr-gcc lors de l'édition de liens. L'option *-I* (i majuscule) précisant l'endroit des fichiers *includes* n'est plus nécessaire puisque tout est au niveau binaire dans l'édition de liens. Il n'y a plus de fichiers sources à cette étape.**
h. Évidemment, comme pour toutes les commandes déjà présentes dans le Makefile, il est recommandé de créer deux variables de Makefile: une pour le nom de la commande et une pour les options à cette commande. Pour être clair, il est préférable d'appeler une commande « $(MACOMMANDE) $(MES_OPTIONS) » par exemple dans le Makefile après avoir défini dans le début du Makefile quelque chose qui pourrait ressembler à ce qui suit par exemple:

```
MACOMMANDE= masupercommand
MES_OPTIONS= -xyz
```

Travail demandé

Après avoir discuté du code à conserver, considérer ce qui pourrait être réutilisé, et donc placer dans une librairie (peut-être pour le contrôle des moteurs, de la DEL, etc...) Reprenez un code vraiment très simple que vous avez écrit et essayez d'en former une librairie pour commencer, juste pour vérifier et visualiser les étapes de construction avec `make`. Par la suite, il serait sage de se limiter à augmenter cette librairie avec d'autre code à compiler. Éviter de former plus d'une librairies. Il vaut peut-être mieux en former une seule bonne que quelques mauvaises... En parallèle, créer un autre répertoire qui utilise cette librairie. Le code écrit a peu d'importance tant qu'il compile correctement et utilise les fonctions de la librairie. Ce code n'a pas à être chargé sur la carte mère ni à être testé. Il n'est présent que pour vérifier que l'édition de liens se produit bel et bien avec la librairie.

Le prochain point est un peu plus difficile à obtenir, mais il mérite tout de même d'être souligné. Il est possible de factoriser les déclarations et règles de plusieurs Makefiles utilisés pour créer différentes librairies et plusieurs exécutables. On peut ainsi créer un fichier, nommé par exemple *Makefile_common.txt* et l'inclure (avec une instruction #include “Makefile_common” comme en C) dans un Makefile. Les déclarations et règles se retrouvent donc à un seul endroit (ce qui simplifie les modifications), mais peuvent être utilisées partout dans les Makefiles. Les Makefiles deviennent donc petits et ne déclarent que les règles et variables locales à la formation d'un exécutable ou d'une librairie précis. Donc, essayez de factoriser les variables et déclarations de fonctions communes pour les regrouper dans un fichier unique qui peut être inclus par tous les Makefiles dans vos différents répertoires.

Il peut s'agir d'un travail immense, mais en réalité, il y a très peu de modifications à effectuer au Makefile utilisé depuis le début de la session. Tout au plus, 20 lignes sont à modifier! En particulier, aucune dépendance ou cible n'est à modifier dans les règles du Makefile; simplement quelques commandes. Il est recommandé de travailler de façon ordonnée en suivant quelques conseils:

- Utiliser make avec l'option -d (pour déboguer)
- Ajouter une commande «echo» à certains endroits pour confirmer l'exécution d'une commande.
- En fin de compte, il devrait y avoir utilisation de la commande *make* deux fois. Une fois pour former la librairie (avec le Makefile qui se trouve dans le répertoire où sera construite la librairie) et une fois pour recompiler le code de l'application (bidon dans ce cas-ci) qui utilise la librairie (avec le Makefile qui fait référence à la librairie précédemment formée).
- La librairie (le fichier .a) ne devrait pas être placée sous Git, car il peut être reconstruit à tout moment et n'est donc pas un fichier source.
- La librairie ne doit pas être copiée manuellement ailleurs. On peut toujours la référencer en modifiant le Makefile qui compile le code de l'application qui a besoin d'utiliser la librairie en question.
- Une modification à la fois...

Soumettre le code dans le répertoire tp/tp7 de l'entrepôt de l'équipe de quatre. De plus, écrire le rapport dont on trouve ici un gabarit et le placer dans le même répertoire.

Références

En terminant, voici quelques liens très intéressants sur les Makefiles. Par contre, ils vont probablement un peu trop loin pour ce qui est demandé comme travail ici, mais il n'est jamais mauvais de regarder profondément le problème.

- http://www.tenouk.com/ModuleW.html
- http://ubuntuforums.org/showthread.php?t=619324
- http://www.commentcamarche.net/faq/14440-la-compilation-et-les-modules-en-c-et-en-c

Un ancien chargé de laboratoire de ce cours, Philippe Carphin, aimait particulièrement ce sujet et a réalisé des vidéos sur youTube très pertinents. La démarche commence avec la ligne de commande jusqu'à l'utilisation de librairies. C'est vraiment très bien fait. On vous le recommande grandement!