Polytechnique Montréal

Département de génie informatique et génie logiciel

Cours INF1900:
Projet initial de système embarqué

Travail pratique 7

# Makefile et production de librairie statique

Par l'équipe

No 108110

Noms:

Xavier Brazeau
Simon Zhang
Hany Bassi
Yan Joliot

Date:
3 mars 2019

**Partie 1 : Description de la librairie**

Après avoir comparé les codes des deux équipes, on a choisi les fonctions les plus importantes que l'on voulait implémenter dans notre librairie.

COULEURDEL

Parmi les fonctions de base on retrouve la fonction **allumerDEL(uint8_t couleur)** du code source couleurDel.cpp qui prend en paramètre la couleur voulue, soit rouge, vert ou encore aucune couleur, c'est à dire que la DEL est éteinte. On pourrait utiliser cette fonction à n'importe quel moment pour vérifier le fonctionnement de notre code, par exemple allumer la DEL en vert si on satisfait une condition particulière.

ANTIREBOND

La fonction **antiRebond()** permet de savoir si le bouton d'interruption est appuyé ou non en appliquant un filtre antirebond permettant d'être sûre d'avoir la bonne valeur de l'input. Le filtre antirebond test une première fois que l'input change, et après un délai de 10 ms le programme revérifie si le bouton est encore bien actionné pour ignorer les « bounces ». La fonction ne prend rien en paramètre et retourne un « true » si le bouton d'interruption est appuyé et retourne un « false » s'il n'est pas appuyé.

MÉMOIRE

On a ajouté à notre libraires les différentes fonctions pour l'accès à la mémoire, qui se retrouvent dans memoire_24.cpp. Parmi les fonctions présentes dans le fichier on retrouve :

-**void Memoire24CXXX::init()** qui initialise le port série et l'horloge de l'interface I2C.

-**uint8_t Memoire24CXXX::choisir_banc(const uint8_t banc)** qui permet de choisir un banc de mémoire.

On a aussi des fonctions pour la lecture de l'EEPROM et l'écriture d'un bloc de données en mode page dans l'eeprom I2C.

MINUTERIE

On a rajouté dans notre librairie l'accès aux fonctions de la minuterie en y incluant Minutrie.cpp , ce qui facilite son utilisation future au lieu de réécrire le code et de rechercher à nouveau les différents registres à modifier. On retrouve alors deux fonctions principales :

-**void startMinuterie (uint16_t duree)** qui crée une interruption de la minuterie après la durée spécifiée.

-**void resetMinuterie ()** qui remet le compteur la minuterie a son état initial, soit 0.

USART

Pour le control de USART nous avons créé une fiche qui contient deux fonctions. Il faut s'assurer tout d'abord que rien n'est branché sur les broches D0 et D1 car elles sont reliées au USART du USART du ATMega8.

- **void initialisationUART() :** Cette fonction ne prend rien en paramètre et ne retourne également rien. Son utilité est d'initialiser certains registres pour permettre la réception et transmission par UART0. Par default, les registres sont initiés pour que la communication se fasse à 2400 bauds avec des trames de 8 bits.
- **void transmissionUART(uint8_t donnee)** : Cette fonction permet de transmettre la donnée passer en paramètre et de l'écrire dans la mémoire du PC. Cette fonction ne permet que l'entrée d'un octet à la fois donc il ne peut écrire qu'un caractère à la fois.

PWM

Dans la librairie, nous avons une fiche servant au control des roues en utilisant des rapports de PWM. Pour utiliser c'est fonction, il faut tout d'abord s'assurer d'alimenter le pont H pour que les roues tournent.

- **void initPWM()** : Initier les registres nécessaires pour permettre l'utilisation des fonction PWM, des comparateurs et compteurs.
- **void roueGauche(bool direction, uint8_t rapport) et void roueGauche(bool direction, uint8_t rapport)** : Ces fonctions permettent le contrôle individuel de la roue gauche et droite respectivement. Elles prennent en entrée une direction (« true » pour avancer et « false » pour reculer) et rapport, un octet qui définit le rapport PWM : 0x00 étant un arrêt complet et 0xff la vitesse maximale.
- **Void avancer(uint8_t rapport) et Void reculer(uint8_t rapport)** Ces deux fonctions appellent les deux fonction de contrôle de chaque roue en les faisant soit avancer ou recule avec le même rapport passe en paramètre.
- **Void arreter()** : cette fonction immobilise les deux roues pour un arrêt complet.
- **Void tournerADroite() et Void tournerAGauche()** : Ces deux fonctions utilisent les fonctions de contrôle des roues individuel. La fonction tournerADroite() arrête la roue droite et fait avancer la roue gauche pour permettre au robot de tourner, par exemple. La fonction tournerAGauche() arrête la roue gauche et fait avancer la roue droite.

**Partie 2 : Décrire les modifications apportées au Makefile de départ**

Nous avons observé qu'il était possible de soit utiliser deux makefiles ou un seul selon l'énoncé du TP7. Nous avons choisi d'utiliser la première configuration, soit celle des deux makefiles. Un des makefile sert à compiler un fichier au robot comme pour tous les autres TPs. L'autre makefile sert à créer l'archive de la librairie statique.

Les modifications apportées au Makefile peuvent être observé par les différences du Makefile original quelque peu modifiée et le Makefile de la librairie:

| **mo-dif. #** | **./exec_dir/makefile** | **./lib_dir/makefile** |
|---|---|---|
| 1 | PRJSRC=main.cpp | PRJSRC= ***$(wildcard *.cpp)*** |
| 2 | INC=***-I ../lib_dir*** | INC= |
| 3 | LIBS=***-L ../lib_dir/ -l108*** | LIBS= |
| 4 | TRG=$(PROJECTNAME).out | TRG=***lib108.a*** |
| 5 | all: ***lib*** $(TRG) | all: $(TRG) |
| 6 | $(TRG): $(OBJDEPS)<br>$(CC) $(LDFLAGS) -o $(TRG) $(OBJDEPS) ***$(LIBS) $(INC)*** | $(TRG): $(OBJDEPS)<br>***avr-ar -crs $@ $^*** |
| 7[1] | ***lib:***<br>***cd ../lib_dir && $(MAKE)*** | - |
| 8 | clean:<br>$(REMOVE) $(TRG) $(TRG).map $(OBJDEPS) $(HEXTRG) *.d \<br>***&& cd ../lib_dir && $(MAKE) clean*** | clean:<br>$(REMOVE) $(TRG) $(TRG).map $(OBJDEPS) $(HEXTRG) *.d |

Tableau 1. Différences entre le makefile de la librairie et le makefile de compilation du fichier source du robot. Les parties surlignées correspondent à ce qui est différent du makefile original donné par Jérôme. Les cellules vides correspondent à aucun changement

1. Cette ligne est une ligne qui n'était pas dans le makefile original.

**./lib_dir/makefile**

Afin de construire une librairie, il faut concevoir un fichier makefile qui compile tous les sous-programmes de la librairie individuellement (pour s'assurer qu'ils fonctionnent), puis qui crée un fichier archive avec tous les fichiers objets à insérer dans la libraire. Le fichier archive sortant doit absolument prendre un nom commençant par lib et se terminant par .a, alors nous avons choisis le nom lib108.a. Finalement la commande à utiliser pour créer une librairie AVR est : avr-ar rcs <library name> <list of object modules>. Les fichiers objet de tous les fichiers .cpp du répertoire doivent être crées puisque le fichier archive contient ces derniers. Le makefile de la librairie doit donc créer tous ces fichiers objets avant de lancer la commande pour créer le fichier .a. Le flag r sert à insérer les fichier objets listés dans le fichier archive en autorisant les remplacements, le flag c crée l'archive et le flag S de créer ou de mettre à jour un object file-index.

Pour désigner le makefile, nous avons tout simplement utilisés le makefile fourni en classe comme squelette pour concevoir le makefile de la librairie. Nous avons ensuite commencé par changer le nom de projetsrc (ligne 32, modif. 1) par $(wildcard *.cpp) puisque comme décrite plus tôt, il est important de précompiler tous les fichiers d'une librairie avant de créer cette dernière. Nous avons ensuite changé le target pour lib108.a (ligne 95, modif 4) puisque c'est le nom de du fichier archive que nous voulons créer. Ensuite, pour les lignes 131 à 143 (modif 6), nous avons optés pour le code suivant:

```
131  all: $(TRG)
132
133  # Implementation de la cible
134  $(TRG): $(OBJDEPS)
135      avr-ar -crs $@ $^
136
137  # Production des fichiers object
138  # De C a objet
139  %.o: %.c
140      $(CC) $(CFLAGS) -c $<
141  # De C++ a objet
142  %.o: %.cpp
143      $(CC) $(CFLAGS) $(CXXFLAGS) -c $<
144
```

Ce code commence par vouloir créer la target (le fichier archive) mais ne peut créer cette target avant d'avoir créé OBJDEPS qui se trouve à être tous les fichiers .cpp du répertoire en .o (défini entre aux lignes 111-112). La ligne 142 (modif 6) s'occupe de faire cette opération. Finalement, lorsque toutes les dépendances .o sont créées, le makefile actionne la ligne 135 qui se trouve à être la commande citée ci-dessus: avr-ar rcs <library name> <list of object modules>.

### ./exec_dir/makefile

Le makefile du dossier exec_dir où se trouve le fichier à installer au robot a aussi connu certains changements. Les modifications 2,3,5,6,7,8 du tableau concernent ce makefile.

2. INC=***-I ../lib_dir***
3. LIBS=***-L ../lib_dir/ -l108***
6. $(TRG): $(OBJDEPS)
   $(CC) $(LDFLAGS) -o $(TRG) $(OBJDEPS) ***$(LIBS) $(INC)***

Le but en gros de ces changements (2,3,6) est permettre au fichier main.cpp dans le dossier "*exec_dir*" d'avoir accès aux fonctions de la librairie.

La modification 2 est nécessaire afin de créer le flag qui laissera au compilateur savoir où se trouvera les fichiers headers contenant les signatures ou les variables globales dont aura besoin le main pour appeler aux fonctions liées de la librairie. Le flag "*-I* " (i majuscule) permet au compilateur avr-gcc de connaître des chemins (path), hormis les chemins de défaut*, où il pourra trouver les fichiers d'inclusion.

Dans notre cas, puisque nous voulions faire un minimum de changement pour comprendre mieux le fonctionnement des makefiles, le makefile faisant l'installation du binaire au robot est resté dans le même dossier que le fichier "*main.cpp*". Ainsi, le chemin passé au flag doit faire un chemin à rebours pour retrouver les fichiers d'entête dans le dossier "*lib_dir*". Finalement, comme argument à passer pour la variable INC, il en est qu'on obtienne: -I ../lib_dir

* Voici les chemins de défaut par exemple sur un ordinateur dans les laboratoires de linux de Polytechnique pour toute inclusion de type #include <...>.

- /usr/lib/gcc/avr/7.2.0/include
- /usr/lib/gcc/avr/7.2.0/include-fixed
- /usr/lib/gcc/avr/7.2.0/../../../../avr/include

Cependant pour les librairies de type #include"...", avr-gcc n'a aucun chemin d'inclusion par défaut. Ainsi, il faut les lui fournir.

La modification 3 est nécessaire à son tour de créer deux flags que le compilateur utilisera pour trouver la librairie statique pour qu'il puisse l'inclure au binaire produit avec le fichier "*main.cpp*" dans le dossier "*exec_lib*". Le premier flag "*-L* " permet au compilateur avr-gcc de connaître le chemin où aller chercher la librairie. Le deuxième flag "*-l* " (L minuscule) est complémentaire à l'autre flag, puisqu'il donne à avr-gcc le nom de la librairie à aller chercher. Plus spécifiquement, le nom de la librairie que avr-gcc cherchera devient: "*lib<argument donné au flag>.a*".

Dans notre cas, puisque notre librairie est formée dans le dossier "*lib_dir*", les flags sont donc:

-L ../lib_dir
-l 108

5. all: ***lib*** $(TRG)
7. ***lib: cd ../lib_dir && $(MAKE)***

Ces règles sont probablement les plus importantes. Elles permettent de créer la librairie avant de passer à la règle de production du fichier à installer sur le robot $(TRG).

La modification 5 permet de faire un appel dans le terminal à make comme ceci:

$ make

et cet appel fera successivement une vérification des dépendances selon l'ordre défini dans le makefile. Dans notre cas, un appel à make remplit la règle de production liée au target "*lib*" et ensuite au target "*$(TRG)*". Ainsi, nous prenons avantage de la manière que make lit l'ordre des dépendances d'une règle pour construire la librairie avant de passer à la compilation du fichier "*main.cpp*".

La modification 6 est justement la règle de production qui permet de créer la librairie dans le dossier "*lib_dir*". La commande "*$(MAKE)*" permet de faire appel à faire un appel au make all du makefile dans le dossier présent (qui sera "*lib_dir*" grâce à "*cd*").

```
8. clean:
        $(REMOVE) $(TRG) $(TRG).map $(OBJDEPS) $(HEXTRG) *.d \
        && cd ../lib_dir && $(MAKE) clean
```

Cette modification peut être vue comme une règle de qualité de vie. Elle permet de faire un nettoyage des artéfacts de la compilation et de la librairie.

Le raisonnement est que, puisque ce makefile construit la librairie, il serait important qu'il soit capable de nettoyer la librairie aussi. Il serait peut-être plus intéressant de créer une commande qui le ferait en séparer. Cependant, pour nos besoins cette règle ira.

## Considérations futures et améliorations possibles

Sur le niveau de la structure du projet, il serait bien possible de travailler avec un makefile à la racine du répertoire git faisant appel au makefile du TP actif.