

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 0106

Noms:

Aya Kamate 1991158

Calie Armande Paulin 1990403

Mélody Roy 1991902

Pier-Luc Tanguay 1953707

Date:

22 octobre 2019

**Partie 1 : Description de la librairie**

À la suite de la formation de notre équipe de quatre, nous avons établi une librairie qui comporte plusieurs fonctions que nous avons écrites nous-mêmes et qui nous sembles utiles pour la continuité de ce projet. Nous avons également intégré deux classes qui ont été fournies lors des deux derniers TP. Un fichier main.cpp (./exec\_dir/source) a été créé pour tester la fonctionnalité des fonctions et de la librairie. Il appelle couleurAmbre(); par une boucle et clignote six fois de rouge au vert, et ce à l’infinie. Les prochaines lignes décriront en détails les fonctions choisies pour la création de la libraire.

**FontionsCommunes**

Ce fichier .h et son implémentation .cpp contient toutes les fonctions des travaux pratiques 1 à 6 qui pourront possiblement être réutilisées pour la suite du projet. Elles sont globales. Certaines d’entre elles ont été légèrement modifiées afin d’être plus générales et de pouvoir s’adapter à plusieurs situations.

1. bool **boutonAppuye**();

Cette fonction permet de déterminer si un interrupteur a été volontairement actionné. Elle comprend un anti-rebond pour éliminer toutes erreurs liées à un contact involontaire des plaques de métal de l’interrupteur. Elle retourne vrai dans le cas où l‘interrupteur a été appuyé, sinon elle retourne faux. Cette fonction ne prend aucune valeur en paramètre.

1. void **couleurAmbre**();

Cette fonction permet de faire allumer une DEL de couleur ambrée. Elle alterne entre une lumière verte et une lumière rouge rapidement. Ainsi l’œil est trompé et une couleur orangée est visible. Cette fonction ne prend aucune valeur en paramètre et ne retourne aucune valeur.

1. void **attenuerDEL**(const uint8\_t couleur, double dureeSeconde);

Cette fonction éteint progressivement une DEL. Elle prend la couleur de la DEL et la durée de l’atténuation en secondes en paramètre. Cette fonction ne retourne aucune valeur.

1. void **pwmLogiciel**(double pourcentage, double frequence, double dureeSeconde);

Cette fonction modifie le PWM de manière logicielle et elle utilise la fonction delay pour générer le PWM. Celle-ci prend en paramètre trois valeur de type double : le pourcentage compris entre 0 et 1, la fréquence et la durée en seconde. Cette fonction ne retourne aucune valeur.

1. void **ajustementPWM**(double pourcentage);

Cette fonction modifie le PWM de manière matérielle en phase-correct, elle considère les deux moteurs de notre robot en modifiant les registres OCR1A et OCR1B. Cette fonction reçoit en paramètre un pourcentage entre 0 et 1, donc de type double. De plus, nous utilisons un diviseur d’horloge de 8 pour fixer la fréquence de notre PWM. Les moteurs recevront le produit de ce pourcentage avec le nombre de bit maximal. Cette fonction ne retourne aucune valeur.

1. void **clignoter2Couleur**(uint8\_t couleur1, uint8\_t couleur2, uint8\_t nFois);

Cette fonction permet de faire clignoter une lumière entre deux couleurs. La LED reste allumer d’une certaine couleur pendant 1 demi-seconde avant de changer de couleur (pour une durée totale de 1 seconde pour 1 cycle). La fonction prend les deux couleurs en paramètre (de type uint8\_t) et le nombre de fois que les deux lumières vont clignoter.

1. void **changementDeLumiere**(uint8\_t voltageNiveau1, uint8\_t voltageNiveau2, uint8\_t positionBrocheA, int8\_t couleurDelNiveau1, int8\_t couleurDelNiveau2, int8\_t voltageMaxTheorique, can convertisseurAnaNum);

Cette fonction permet d’associer une couleur de DEL avec la quantité de lumière que perçoit le potentiomètre. Elle permet de détecter trois niveaux de lumière : un niveau faible, un niveau moyen et un niveau fort. Elle prend en paramètre deux niveaux de voltages (voltageNiveau1 et voltageNiveau2) qui marquent la limite entre les niveaux en volt. La fonction prend en paramètre la couleur de la DEL pour un niveau élevé (couleurDelNiveau2) et la couleur de la DEL pour un niveau faible (couleurDelNiveau1). La couleur du niveau moyen est un mélange du niveau élevé et du niveau faible, c’est-à-dire une couleur ambrée. Le voltageMaxThéorique en paramètre représente le voltage maximum théorique du potentiomètre. Le constructeur Can en paramètre permet d’utiliser la fonction de conversion d’une donnée analogique à numérique. Tous les paramètres de la fonction sont en uint8\_t sauf le constructeur can. Cette fonction ne retourne aucune valeur.

---------------------------------------------------------------------------------------------------------------------

Les deux prochaines classes sont initialement fournies pour ce projet. Nous n’avons pas modifié celles-ci, cependant, nous jugeons qu’elles sont importantes à intégrer dans notre librairie statique.

**Can**

La classe Can permet de contrôler une conversion analogique/numérique. Le fichier can.h contient un constructeur permettant l’initialisation de ce convertisseur, un destructeur, ainsi que la définition d’une seule méthode qui est implémentée dans le fichier can.cpp, soit la méthode : uint16\_t lecture (uint8\_t pos). Cette méthode prend en paramètre l’entier sur 8 bits : pos. Ce-dernier correspond au port A qui est relié au potentiomètre. La valeur analogique sera donc représentée numériquement grâce à ce port.

**Memoire24**

La classe Memoire24 permet l’accès à la mémoire externe en modifiant les ports C0 et C1. Celle-ci comporte quatre méthodes qui nous sembles utiles, deux méthodes pour lire la mémoire et deux autres pour écrire dans la mémoire.

1. uint8\_t **lecture**(const uint16\_t adresse, uint8\_t \*donnee);

Cette méthode permet la lecture de la mémoire une donnée à la fois. Cette fonction prend l’adresse d’écriture en paramètre de type uint16\_t et un pointeur vers la donnée à écrire en uint8\_t. Elle retourne la donnée lue en uint8\_t.

1. uint8\_t **lecture**(const uint16\_t adresse, uint8\_t \*donnee, const uint8\_t longueur);

Cette méthode permet la lecture d’un bloc de donnée ayant une certaine longueur qui ne doit pas dépasser 127 bits. Cette fonction prend l’adresse d’écriture en paramètre de type uint16\_t et un pointeur vers une chaine de données de type uint8\_t et la longueur de cette chaine de données de type uin8\_t. Cette fonction retourne le nombre de données lues en uint8\_t;

1. uint8\_t **ecriture**(const uint16\_t adresse, const uint8\_t donnee);

Cette méthode permet l’écriture dans la mémoire une donnée à la fois. Cette fonction prend l’adresse d’écriture en paramètre de type uint16\_t et un pointeur vers la donnée à écrire en uint8\_t. Cette fonction retourne la donnée écrite de type uint8\_t.

1. uint8\_t **ecriture**(const uint16\_t adresse, uint8\_t \*donnee, const uint8\_t longueur);

Cette méthode permet l’écriture dans la mémoire par bloc de donnée qui ne doit pas dépasser 127 bits. Cette fonction prend l’adresse d’écriture en paramètre de type uint16\_t et un pointeur vers la chaine de données à écrire de type uint8\_t et la longueur de cette chaine de type uint8\_t. Cette fonction retourne le nombre de données écrites en uint8\_t.

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

Tout d’abord, nous avons modifié la majorité des lignes du *makefile* donné initialement au tp1 afin de simplifier le code et de créer une structure sous-dossiers permettant de séparer les types de fichiers de façon à s’y retrouver plus facilement. La même arborescence est utilisée pour les deux dossiers (lib\_dir et exec\_dir)*.*Cela dit, les *makefiles* ont été ajustés en fonction de cette arborescence grâce aux variables personnalisées et de différentes commandes de chemin d’accès. Ce processus nous permet de simplifier la gestion des fichiers dans le cas qu’ils y en auraient plusieurs (au lieu de les avoir tous dans un seul dossier: \*.o, \*.cpp, \*.h, etc.), mais également pour objectif de s’entraîner pédagogiquement à la conception d'un *makefile.*

Malgré l’arborescence, les *makefiles* de la librairie et de l’exécutable sont en mesure de trouver tous les fichiers nécessaires. En effet, selon les *includes* du main.cpp, le m*akefile* exécutable trouve les fichiers .h du *lib\_dir* et du *exec\_dir*. De plus, Lors de la commande *make clean*, les fichiers et dossiers créés s’éliminent pour revenir à l’état originale. Le *makefile* de *exec\_dir* est en mesure de créer une nouvelle librairie, au besoin, et de *clean* les deux dossiers par l’utilisation de ‘make -C …’.

Finalement, nous avons créé une fonction *debug* pour tester les variables personnalisées qui s’affichent à l’aide de *@echo.*Finalement, nous avons retiré les éléments correspondant à la compilation en C, puisque nous utilisons que le C++ dans notre cas.

**Quelques modifications makefile librairie:**

* Variables pour les sous-dossiers
* Cible TRG représente le fichier librairie qui dépend des objets
* Variable de fichiers source qui prend tous les .cpp (wildcard, \*.cpp)
* Variable de fichiers objets qui remplace les .cpp et .o (subst)
* Ajout du flag -DF\_CPU=8000000 pour remplacer les #define F\_CPU
* Ajout de la variable AR pour commande archive ‘avr-ar’ et création librairie
* Ajout de l’option ‘-I’ pour les chemin d’accès des fichiers .h
* Cible finale : .a ,dépend des fichiers .o
* Dépendance des .o avec ajout des variables sous-dossiers
* ‘make clean’ supprime les .o et .d, et le dossier ‘objet’

**Quelques modifications makefile exécutable:**

* Variables pour les sous-dossiers et chemin d’accès du dossier contenant la librairie: ../lib\_dir/
* Variable de fichiers source qui prend tous les .cpp (wildcard, \*.cpp)
* Variable de fichiers objets, qui remplace les .cpp et .o (subst)
* Ajout du flag -DF\_CPU=8000000 pour remplacer les #define F\_CPU
* -I pour les inclusions des fichiers .h du main.cpp et de ceux qui se trouvent dans le dossier ‘lib\_dir/include’
* Ajout d’une variable pour le nom de la librairie sans le ‘lib’ et avec -l (L minuscule). Nous avons remarqué que s’il était avant le -o, nous avions une erreur de *linkage* (undefined reference to ‘fonctionAppellée’)
* Dans la commande d’implémentation de la cible .out, nous avons ajouté -L pour la localisation de la libraire
* Pour l’implémentation des .o, nous avons ajouté les inclusions -I du sous-dossier de ‘exec\_dir’ et celui de ‘lib\_dir’

*Le rapport total ne doit pas dépasser 7 pages incluant la page couverture.*

*Barème: vous serez jugé sur:*

* *La qualités et le choix de vos portions de code choisies ( 5 points sur 20 )*
* *La qualités de vos modifications aux Makefiles ( 5 points sur 20 )*
* *Le rapport ( 7 points sur 20 )*
* *Explications cohérentes par rapport au code retenu pour former la librairie (2 points)*
* *Explications cohérentes par rapport aux Makefiles modifiés (2 points)*
* *Explications claires avec un bon niveau de détails (2 points)*
* *Bon français (1 point)*
* *Bonne soumission de l'ensemble du code (compilation sans erreurs, …) et du rapport selon le format demandé ( 3 points sur 20 )*