Polytechnique Montréal

Département de génie informatique et génie logiciel

Cours INF1995:
Projet initial en génie informatique et travail en équipe

Travail pratique 8

Makefile et production de librairie statique

Par l'équipe

6771

Noms:
Cadorette, Sébastien
Châteauvert, Mathieu
La Berge, William
Pham, Son-Thang
Stéphenne, Colin

Date:
14 mars 2017

Partie 1 : Description de la librairie

Pour introduire, la construction d'une librairie est énormément bénéfique pour la continuité de notre projet. Cette construction peut sauver énormément de temps puisqu'elle limite la réécriture des mêmes lignes de codes d'un nouvel exercice à l'autre. Il est alors optimal d'écrire des classes, méthodes et fonctions qui sont hautement réutilisables. En effet, les fonctions générales qui reviennent souvent d'un problème à un autre sont alors celles qui doivent être ciblées.

constantes.h

Premièrement, la définition du fichier .h contenant des DEFINE semblait évident afin de ne pas toujours réécrire les mêmes conventions établies d'un fichier à un autre. Le choix d'écrire celles-ci dans un fichier permet d'illustrer toutes les conventions que nous avons établies.

del.cpp & del.h

Ensuite, les fonctions reliées à l'affichage de la Del ont été hautement utilisées à travers les semaines. En effet, la Del libre avait un rôle clé dans la majorité de nos travaux pratiques. Alors, l'obligation de créer un fichier source contenant une classe Del est flagrant. Cette classe contient alors ses principales méthodes modifiant la couleur de sortie de la Del comme afficherAmbre() ou afficherVert().

minuterie.cpp & minuterie.h

Nous avons créé une fonction delai_us() permettant de faire un délai, mais dont le paramètre reçu par la fonction n'est pas une constante. En effet, le paramètre est une variable qui change avec le temps. Dans quelques-uns de nos travaux pratiques, cette fonction nous a été grandement utile. Le classique _delay_us() ne permet pas d'avoir en paramètre une variable non constante.

Aussi, dans ces fichiers, on retrouve certaines fonctions permettant de gérer une minuterie. Ces fonctions ont été très utiles lors de nos travaux pratiques, raison pour laquelle nous avons trouvé fort bien de les inclure dans notre librairie.

bouton.cpp & bouton.h

La fonction etatBouton() permet d'obtenir l'état dans lequel le bouton est à cette instant. Deux états sont possibles pour le bouton, soit l'état pressé et l'état relâché. C'est la fonction la plus réutilisée dans nos travaux pratiques et elle inclut les notions d'anti-rebond.

moteur.cpp & moteur.h

Par la suite, nous avons regroupé toutes les fonctionnalités liées au moteur. Ces moteurs permettent tous les déplacements du robot. Brefs, ces fonctions de déplacement sont nécessairement utiles pour le futur de notre robot. Nous avons créé plusieurs fonctions comme une fonction permettant de faire avancer le robot, une fonction permettant de faire reculer le robot, une fonction permettant de faire tourner le robot à droite, et une dernière fonction permettant de faire tourner le robot à gauche.

memoire_24.cpp & memoire_24.h, can.cpp & can.h

Lors des travaux pratiques, certains fichiers .cpp et .h nous ont été fournis. Nous les avons donc intégrés à notre librairie. Deux de ces fichiers permettent de gérer toutes les opérations mémoires. Il y a le fichier memoire_24.h et memoire_24.cpp. Deux autres fichiers permettent de contrôler un convertisseur analogique/numérique qui permettent d'utiliser une photorésistance. On parle ici des fichiers can.h et can.cpp.

uart.cpp & uart.h

Finalement, la classe uart permet d'initialiser et de permettre les transferts des données de l'ordinateur jusqu'au robot et vice-versa. Cela est utile, puisque la réception de données sera une méthode très appropriée pour aider le débogage du robot. De plus, on pourrait permettre au robot de transmettre des messages. Bref, le tout permettra au robot d'avoir une certaine méthode de transmission de données.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Nous avons modifié et créé au total deux makefiles, un pour la librairie et un pour implémenter sur la carte, comme on le faisait précédemment. Cependant, nous avons aussi créé un fichier « makefile-common.txt » qui fait un ramassis de ce que les deux fichiers ont en commun. Ce fichier est un surplus, mais il a été suggéré de le faire dans le TP8.

Makefile commun

Nous avons décidé de créer un fichier regroupant une multitude de règles et de variables communes aux Makefiles : le fichier « makefile_common.txt ». L'objectif de celui-ci est d'alléger le code, d'éviter la répétition de code et de saisir l'essentiel d'un Makefile. De plus, elle nous permet créer différentes librairies et plusieurs exécutables. C'est pour cette raison que nous les incluons dans chacun des Makefiles en y insérant « include [chemin]/makefile_common » au début du fichier.

Pour créer ce fichier commun, nous avons sélectionné une multitude d'éléments déjà présent dans les fichiers Makefiles utilisés depuis le début du projet. Nous avons conservé la section « Définition de tous les fichiers objets » et la section « variables » en lui ajoutant la variable « AR=avr-ar » afin de pouvoir créer des librairies. Le nom du microcontrôleur, le niveau d'optimisation et le « programmer ID » restent inchangés. Pour les options de compilation nous conservons la majorité, par contre nous avons rejeté les « Flags » pour le compilateur en C puisqu'elles changent dépendamment de l'option que nous désirons. Enfin, les éléments et sections que nous n'avons pas mentionnés sont rejetés car c'est à l'aide de ces informations que nous pouvons définir ce qui est essentiel pour un Makefile.

Makefile pour les librairies

Le but de ce Makefile est de créer une librairie complète qui rassemble tous les fichiers .h et .cpp. Le Makefile viendra aider à compiler tous les fichiers ensemble afin de créer une bibliothèque .a où le fichier .exe viendra chercher ce qu'il lui faut.

Le Makefile de la librairie inclus premièrement le fichier commun qui prend ce qui est redondant dans les Makefiles. Par la suite, nous devons donner un nom à notre librairie, comme on donne un nom à un projectname. Le target de la librairie est « ../ » qui fait seulement un retour en arrière. Étant donné que nous ne voulons pas la « librairie6771.a » dans notre dossier librairie, l'instruction (../) vient diriger la cible dans le bon emplacement. Le nom du projet est aussi à modifier. On lui a donné le nom « LIBNAME » qui retourne le nom qu'on lui a donné précédemment.
Plus loin, après avoir déplacé plusieurs parties de code dans le fichier commun, nous venons changer les options de compilation. Pour le « CFLAGS », nous venons rajouter un « -c » comme il est indiqué dans l'exemple sur le site web afin de créer l'archive. On doit aussi ajouter un « ARFLAGS », qui est en fait un flag pour le compilateur de la librairie. Ceci permet de mettre les fichiers .o dans la « librairie6771.a ».
Pour ce qui est des targets de la librairie, il faut encore modifier le Makefile initial. La ligne « LIBTRG=$(TRGDIR)$(LIBNAME).a » vient faire en sorte de donner une direction et un nom à « librairie6771.a » qu'on veut créer. Il faut bien sûr après implémenter cette cible-là. Nous changeons alors la ligne dans la création des données de Makefile et par la suite, il s'agit seulement de remplacer commande par commande les éléments. À la place d'avoir un fichier exécutable, nous voulons avoir un fichier archive (CC par AR), ainsi que les flags qui y sont reliés. De plus, lorsqu'il est temps de clean la librairie, il faut s'assurer de ne pas tout remove. Par exemple, il faut supprimer les fichiers .o, mais pas la librairie .a qui se situe dans l'autre dossier. Elle doit être conservé pour futur utilisation. Ceci complète donc ce qu'il faut modifier dans le Makefile.

Makefile pour les exécutables (.exe)

L'objectif de ce Makefile est de recompiler le code de l'application qui utilise la librairie avec le Makefile qui fait référence à la librairie formée. Nous prendrons comme base un des fichiers Makefiles que nous utilisons depuis le début du projet. D'ailleurs, nous pouvons déjà le simplifier grâce au fichier « makefile_common.txt ». Il restera alors à modifier les quelques éléments et sections qui restent. Nous répétons l'appel au ATMega324PA, puisqu'il est plus facile de différencier les deux makefiles. Nous utilisons alors la librairie formée lors du premier « make », c'est-à-dire « librairie6771.a ». Elle sera liée et nous pourrons alors inclure des fonctions de celle-ci dans le fichier « test.cpp ». En effet, c'est pour cette raison que nous rajoutons « -I ./lib » suivi de « -c » (-c pour créer l'archive) du « CFLAGS » pour les options de compilation. De plus, dans l'implémentation de la cible nous rajoutons « $(LIBS) » dans sa liste de dépendance (LIBS étant une variable pour « librairie6771.a ») parce que nous voulons ajouter la librairie que nous avons inclus et l'utiliser.