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 0222

Noms:

Samuel Ouvrard
Marc-Olivier Riopel
Marc-Alain Tétreault
Émilie Vaudrin

Date:
3 mars 2019

Partie 1 : Description de la librairie

La suite de ce document explique le contenu de la librairie “libanane” et les notions importantes pour être en mesure de s’en servir efficacement par la suite. Le code retenu et modifié par l’équipe 0222 pour former la librairie provient des TP complétés préalablement dans le cours de INF1900. Le code a été retenu et modifié selon trois critères soit son bon fonctionnement, sa compréhensibilité et sa pertinence quant à la suite du projet. En terme de bon fonctionnement, tout code qui ne donnait pas le résultat désiré a été rejeté. Quant à la compréhensibilité, les sections de code retenus ont été modifiés pour qu’ils s’auto-documentent pour facilement comprendre le fonctionnement du code source. Par rapport à la pertinence du code, puisque les objectifs du projet n’ont pas encore été données les membre de l’équipe ont décidé de créer une librairie très générale et versatile qui pourra s'avérer utile dans une multitudes de tâches différentes. C'est pourquoi les fonctionnalités spécifiques implémentés dans les TP ont été laissé de côté et les fonctions plus générales ont été retenues. Même les fonctions retenues ont été modifiés pour qu’elles puissent être d’usage dans différentes tâches.

La prochaine section est séparé selon les différents fichiers de la librairie. Chaque section contient les fonctions d’utilisation d’une partie spécifique du robot et les informations nécessaires à la manipulation de ces fonctions.

bouton.h et bouton.cpp

debounce():
S’assure que le signal du bouton poussoir de la carte mère envoyé à la PIND2 est stable pendant 10 ms.

initInterruptionBouton(type type, vecteur vecteur):
L’énumération type {RISING_EDGE, ANY_EDGE, FALLING_EDGE} défini ce qui génère l’interruption et l’énumération vecteur {INT0_vect, INT1_vect, INT2_vect} défini le vecteur qu’il faut définir dans la fonction d’interruption ISR.

DEL.h et DEL.cpp

allumerDEL(couleur couleur):
La DEL doit être connectée aux PINB0 et PINB1.

detecterLum():
Appelle la fonction *allumerDEL(couleur couleur)* avec la couleur selon l'intensité de lumière reçu par la cellule photovoltaïque: ROUGE si l’intensité de lumière est faible, AMBRE si l’intensité est moyenne et VERT si l’intensité est forte.

moteur.h et moteur.cpp

ajustementPWM(uint8_t pourcentageRoueA, uint8_t pourcentageRoueB): ajuster le PWM des deux roues individuellement pour permettre au robot de pouvoir tourner en utilisant une seule fonction. Un pourcentage est passé en paramètre pour chaque roue et ce pourcentage détermine la fréquence du PWM.

allumerMoteur(uint8_t pourcentage, int frequence, double duree): permet d’allumer les moteurs pour les deux roues. Il est possible d’utiliser le PWM établi avec ajustementPWM en entrant 100 comme pourcentage en paramètre. Il est aussi possible de contrôler la vitesse avec la fréquence si éventuellement le besoin se présente. Également, il est possible de choisir la durée pour laquelle les moteurs seront allumés ce qui permettra éventuellement de contrôler une distance précise sur laquelle le robot doit se déplacer.

void ajustementDirectionRoues (bool roueAAvance, bool roueBAvance): permet contrôler la direction des deux roues indépendamment l'une de l'autre (avancer ou reculer).

minuterie.h et minuterie.cpp

partirMinuterie_1(uint8_t secondes) et partirMinuterie_2(uint8_t secondes): ces deux fonctions permettent de générer une interruption après une durée en seconde déterminée par l'utilisateur. Elles utilisent chacune un registre différent, ce qui permet l'utilisation des deux minuteries simultanément.

memoire_24.h et memoire_24.cpp

Ces fichiers contiennent la classe "Memoire24CXXX" qui permet de manipuler la mémoire externe de deux façons différentes.

Lire et écrire dans la mémoire un bit à la fois des données spécifiées par l'utilisateur à l'aide des deux méthodes suivantes:

-uint8_t lecture(const uint16_t adresse, uint8_t *donnee)
-uint8_t ecriture(const uint16_t adresse, const uint8_t donnee)

Lire et écrire dans la mémoire par bloc de données dont la taille et le contenu sont spécifiés par l'utilisateur à l'aide des deux méthodes suivantes:

-uint8_t lecture(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur)
-uint8_t ecriture(const uint16_t adresse, uint8_t *donnee,const uint8_t longueur)

memoire.h et memoire.cpp

effacerMemoire():
Remplace tous les caractères contenus dans la mémoire externe par le caractère de fin. Cette fonction se verra utile s'il y a besoin éventuel de remettre la mémoire à zéro.

can.h et can.cpp

Ces fichiers contiennent la classe "Can" qui permet de lire des données analogiques et de les convertir en données numériques à l'aide de la méthode uint16_t lecture(uint8_t pos). Cette méthode retourne la valeur numérique correspondant à la valeur analogique sur le port A où seul les 10 bits de poids faibles sont significatifs.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Dans le cadre de cette librairie, nous avons créé deux makefiles de toutes part pour s'assurer de bien comprendre le processus de compilation et de "linking" lors de la construction de la librairie.

Makefile librairie

```
SRC = $(wildcard $(SRC_dir)/*.cpp)
```

Collecte de tous les fichiers terminant par “.cpp” dans SRC_dir, qui correspond au dossier sources.

```
OBJ = $(subst $(SRC_dir),$(BUILD_dir),$(SRC:.cpp=.o))
```

Conversion de tous les fichiers “.cpp” contenus dans la variable SRC en fichiers “.o” avec le préfixe BUILD_dir qui leur permettront d’être entreposés dans le fichier build.

```
$(TRG): $(OBJ)
    $(AR) $(LIBFLAGS) -o $@ $(OBJ)
```

Création de libanane.a avec tous les fichiers “.o” entreposés dans la variables OBJ. Les variables AR et LIBFLAGS contiennent respectivement le nom de l’archiveur “avr-ar” et les règles d’archivage “-csr”.

```
$(BUILD_dir)/%.o: $(SRC_dir)/%.cpp
    $(CC) $(CFLAGS) $(CXXFLAGS) -c $< -o $@
```

Création des des fichiers “.o” dans BUILD_dir selon leurs dépendances dans SRC_dir. La variable CFLAGS contient la définition du microcontrôleur “-mncu=$(MCU)” ou MCU correspond au nom du microcontrôleur “atmega324pa” et les règles de compilation en c que nous avons repris du makefile donné en exemple dans le cours. La variable CXXFLAGS contient les règles de compilation en c++ que nous avons repris du makefile donné en exemple dans le cours.

```
vars:
    @echo "SRC = $(SRC)"
    @echo "OBJ = $(OBJ)"
    @echo "TRG = $(TRG)"
```

Permet d’afficher sur le terminal le contenu des différentes variables du makefile, ce qui a facilité le débogage de celui-ci.

```
clean:
    rm -f $(BUILD_dir)/*.o $(TRG)
```

Supprime les fichiers “.o” et la librairie pour permettre de recompiler la librairie à neuf.

Makefile projet

Seulement les différences notables entre ce makefile et le makefile de la librairie seront explicitées dans cette section pour éviter de se répéter. Le principe est le même, mais nous avons ajouté quelques commandes. L'intérêt principal de créer deux makefile est d'augmenter la clarté générale du projet.

```
compileLib: makefile
    cd $(LIB_dir); make
```

Appelle la commande "`make all`" du makefile de la librairie. Cette commande est une dépendance de la fonction `all` du makefile projet ce qui permet entre autre de n'avoir qu'a appelé le makefile projet sans se soucier de penser à recompiler la librairie si nécessaire.

```
$(TRG): $(OBJ)
    $(CC) $(LDFLAGS) -o $(TRG) $(OBJ) \
    -lm $(LIBS)
```

Création de l'exécutable qui pourra être installé sur la carte mère du robot le liant avec la librairie libanane.a grâce à la variable `LIBS` qui contient "`-L $(LIB_dir) -l $(LIB_name)`" ou `LIB_dir` représente la direction vers la librairie et `LIB_name` le nom de celle-ci.

```
$(BUILD_dir)/%.o: $(SRC_dir)/%.cpp
    $(CC) -o $@ $(CFLAGS) $(CXXFLAGS) -c $<
```

Cette commande est la même que dans le makefile de la librairie sauf que cette fois-ci la variable `CFLAGS` contient "`-I. -MMD $(INC)`" dans lequel `INC` représente "`-I $(LIB_inc)`" ou `LIB_inc` est la direction vers les fichiers sources de la librairie.

```
clean: cleanLib
    rm -f $(BUILD_dir)/*.o *.out $(HEXTRG) *.d $(TRG).map
```

Supprime les fichiers "`.o`", "`.out`", "`.d`", "`.map`" et "`.hex`" créés lors de la compilation du projet tout en appelant la fonction `cleanLib` si nécessaire.

```
cleanLib: makefile
    cd $(LIB_dir); make clean
```

Appelle la commande `make clean` du makefile de la librairie un peu comme le faisait `compileLib` pour la commande `make all`.

```
install: compileLib $(HEXROMTRG)
    avrdude -c usbasp    \
    -p $(MCU) -P -e -U flash:w:$(HEXROMTRG)
%.hex: %.out
    avr-objcopy -j .text -j .data \
        -O ihex $< $@
```

Les deux dernières commandes de ce makefile ont été reprises du makefile fourni dans le cours pour permettre de convertir l'exécutable en `.hex` et ensuite l'installer sur la carte mère du robot à l'exception de l'ajout d'une dépendance pour recompiler la librairie au besoin lors de la commande `make install`.