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 5367

Noms:

Hee-Ming Yang
Adam Halim
Chun Yang Li
Jean Janssen

Date:
3 novembre 2020

Partie 1 : Description de la librairie

Décrire la librairie construite et formée (définitions, fonctions ou classes, utilité, etc...) pour que cette partie du travail soit bien documentée pour la suite du projet pour le bénéfice de tous les membres de l'équipe.

Notre librairie est composée de quatres classes de base: can, Led, Motor et Uart. Chacune de ces classes réutilisent des fonctions prélevées des tps précédents. Parfois, les méthodes sont modifiées pour permettre un usage plus flexible.

can

Cette classe contient un constructeur, un destructeur et une fonction "lecture". La fonction "lecture" sert à convertir les signaux analogiques (en volt) en valeurs numériques dans une variable à 16 bits où seuls les 10 premiers bits sont significatifs. Cette classe nous a été fournie lors du tp6, donc rien n'est à modifier pour l'implémenter en librairie.

Led

Cette classe contient une fonction "blink" et une fonction "buttonIsPressed". Cette classe regroupe le code qui sont en lien avec les manipulations de la DEL ainsi que le bouton-poussoir.

“blink”

La fonction "blink" est un code simple qui sert à allumer une DEL pour un certain temps. Elle prend en paramètres:

- uint8_t ledPinSource: pin qui génère le courant.
- uint8_t ledPinGround: pin qui reçoit le courant.
- uint8_t duration: durée pour laquelle la DEL sera allumée
- volatile uint8_t& port: port connecté à la DEL sur l'Atmega.

Comparé au code original, nous avons ajouté un paramètre "port". Ce paramètre est dû au fait que le "blink" original ne fonctionnait que sur le port A de l'Atmega. Pour l'implémenter en librairie, il faut le rendre moins spécifique (ou plus flexible). En ajoutant ce paramètre, on permet alors à l'utilisateur de choisir le port voulu. Les paramètres ledPinSource et ledPinGround sont aussi ajoutés pour permettre à l’utilisateur de choisir les pins à utiliser du port ainsi que de changer la couleur de la DEL (selon le sens du courant). De plus les opérateurs d'affectation (=) pour assigner les ports (ex. PORTA = 1 << PORTA1) sont désormais des opérateurs de logique OR et AND avec complément (pour éteindre) sur les pins spécifiques à la méthode pour préserver la valeur des autres pins non reliés.

“buttonIsPressed”

La fonction "buttonIsPressed" vérifie si le bouton est appuyé avec un anti-rebond. Elle retourne un true ou false selon si le bouton est appuyé ou non. Elle prend en paramètres:

- uint8_t button: pin lié au bouton à appuyer.
- volatile uint8_t pinx: port configuré en entrée qui est relié au bouton.

Similairement à la fonction “blink”, cette fonction contient un paramètre volatile pour permettre à l'utilisateur de choisir son port en entrée en mode lecture. De plus, elle a une variable locale "DEBOUNCE_DELAY" qui contrôle le délai entre les 2 vérifications de l'anti-rebond. La fonction originale utilise une variable globale à la place.

Motor

Cette classe contient une fonction "turnMotorPWM", une fonction "convertPercentInPWM8BitTimer" et une fonction "adjustPWM". Cette classe regroupe les codes qui sont en lien avec l'ajustement de la vitesse des roues par le biais de PWM sur 8 bits.

“AdjustPWM”

La fonction "adjustPWM" sert à ajuster le PWM. Elle prend en paramètres:

- uint8_t percentage: pourcentage du PWM.
- volatile uint8_t ocrnx: pin responsable de générer l’onde PWM.

La fonction originale ajuste le PWM pour 2 pins fixes et comporte aussi l'initialisation du timer très spécifique au tp. L'initialisation est enlevée de la fonction en librairie pour le rendre plus flexible et permettre à l'utilisateur d'initialiser son propre timer. La fonction ne peut désormais qu'ajuster un PWM à la fois, puisque le nombre de PWM à ajuster dépend de la situation et avec l'ajout du paramètre "ocrnx", l'utilisateur peut choisir son pin sur l'Atmega.

"convertPercentInPWM8BitTimer"

La fonction "convertPercentInPWM8BitTimer" permet de prendre un pourcentage et de retourner une valeur équivalente sur 8 bits soit entre 0 et 254. Cette fonction prend en paramètre:

- uint8_t percentage: permet de choisir le pourcentage à convertir.

Cette fonction est déjà très peu spécifique, donc aucune modification n'est nécessaire.

“turnMotorPWM”

La fonction "turnMotorPWM" permet de faire tourner une roue pour un certain intervalle de temps. Elle prend en paramètres:

- double PWM: pourcentage de PWM voulu.
- double frequency: fréquence à laquelle la roue va tourner.
- uint8_t directionPin: pin direction relié à la roue sur l'Atmega.
- uint8_t enablePin: pin enable relié à la roue sur l'Atmega.
- double duration: durée pour laquelle la roue va tourner.
- volatile uint8_t& port: port relié à la roue sur l'Atmega.

Le code original a une roue fixe sur un port spécifique et cette roue tourne pendant un intervalle de temps qui est aussi spécifique. Dans la nouvelle fonction, les paramètres "duration", "enablePin", "port" et “directionPin” sont ajoutés pour avoir plus de flexibilité en permettant à l'utilisateur de choisir le temps pendant lequel la roue va tourner. Le pin enable ainsi que le port déterminent où l’utilisateur veut relier la roue. Finalement, le “directionPin’’ sert à déterminer la direction que tourne la roue. De plus, les opérateurs d'affectation sont changés en opérateur de logique OR sur les pins spécifiques à la méthode pour préserver la valeur des autres pins non reliés.

Uart

Cette classe contient un constructeur par défaut, une fonction "initialisation" ainsi qu'une fonction "transmission". Cette classe regroupe les codes qui sont en lien avec la transmission des données avec le protocole RS232 vers le PC.

“Uart”

Le constructeur sans paramètre utilise la fonction “initialisation” puisque nous voulons configurer l’Atmega correctement lorsque nous allons utiliser les méthodes de cette classe.

“initialisation”

Active les registres qui permettent à l'Uart de transmettre et de recevoir les données. Cette fonction n'a pas été modifiée.

“transmission”

La fonction "transmission" permet de transmettre des données vers le PC. Elle prend en paramètre:

- uint8_t data: donnée ASCII à transmettre.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Décrire les quelques modifications apportées au Makefile de la librairie pour démontrer votre compréhension de la formation des fichiers. Faire de même pour les modifications apportées au Makefile du code (bidon) de test qui utilise cette librairie.

Makefiles

“Makefile_common.txt”

Le fichier “Makefile_commons.txt” est un fichier texte qui contient la déclaration des variables qu’on utilisera dans les “Makefiles”. En effet, les valeurs de ces variables sont communes aux “Makefiles” que nous allons utiliser. De plus, cela nous permet d’alléger les “Makefiles” et d’éviter de réécrire la valeur d’une variable à plusieurs reprises si on change celle-ci.

Les variables ajoutées ont les définitions suivantes:

- MCU représente le nom du microcontrôleur soit l’Atmega324pa.
- CC représente le compilateur utilisé, soit “avr-gcc”.
- LIBNAME représente le nom de notre librairie “lib”.
- LIBFOLDER est le chemin vers le répertoire contenant la librairie liblib.a.
- SRC représente le nom du fichier source de la librairie. Elle est représentée par l’expression “$(wildcard *.cpp)”. Le mot clé wildcard *.cpp prend la liste de tous les fichiers .cpp peu importe leur nom.
- OBJ représente “$(SRC:.cpp=.o)”. Cette expression signifie que pour chaque fichier .cpp provenant de SRC (tous les fichiers .cpp), il faut créer un fichier .o correspondant.
- HEADERS représente l’expression $(SRC:.cpp=.h). Pour chaque fichier .cpp de SRC, il associe les fichiers .h aux fichiers .cpp correspondants.
- LIB représente l’expression lib$(LIBNAME).a. Il s’agit du nom du fichier de la librairie.
- CFLAGS représente les options de compilation. Ceux-ci nous ont été fournis et n’ont pas été changés en conséquence. L’inclusion de la librairie simavr avec le flag “-I/usr/include/simavr” nous permet d’avoir accès aux diverses fonctionnalités de l’Atmega324pa, notamment l’accès aux registres et aux ports du microcontrôleur.
- CXXFLAGS représente les flags d’avertissements pour le compilateur C++. Ceux-ci ont été fournis et n’ont pas été modifiés en conséquence.
- REMOVE nous permet d’effectuer la commande “rm -f” qui sert à supprimer certains fichiers lors de l’appel d’un make clean.

Makefile “lib_dir”

“include ../Makefile_common.txt”

Le makefile du dossier “lib_dir” inclut le fichier “Makefile_commons.txt” décrit précédemment. On utilise les variables LIB, OBJ, HEADERS, REMOVE, CC, CFLAGS et CXXFLAGS provenant du fichier “Makefile_common.txt”.

“$(LIB):$(OBJ) $(HEADERS)”

La première règle du “Makefile” définit les dépendances pour la génération de la librairie. Effectivement, comme nous l’avons vu dans le fichier “Makefile_commons.txt”, $ (OBJ) représente tous les fichiers .o pour chaque fichier .cpp. Cela veut dire qu’on prend les fichiers “output” de toutes les classes de notre librairie. De plus, nous faisons référence aux fichiers “.h” lorsque nous voulons faire des appels aux méthodes des classes de la librairie avec la variables “$(HEADERS)”. Alors, il faut aussi mettre tous les fichiers headers de toutes les classes comme dépendance de la librairie.

Pour créer notre objet librairie, nous faisons le “linking” de tous les fichiers spécifiés dans la variable “$(OBJ)” avec la commande “ar -rcs $(LIB) $(OBJ)”. Par contre, la création de la librairie génère des fichiers résiduels indésirables. Nous les enlevons alors avec la commande “make clean”.

“%.o: %.cpp %.h”

La deuxième règle est “%.o: %.cpp %.h”. Le symbole “%” représente le nom du fichier objet qu’on souhaite créer. On l’utilise pour éviter de répéter le même nom de fichier dans la règle. De plus, l’utilisation de cette règle permet de vérifier les dépendances de tous les fichiers “.o” indépendamment. Dans ce cas-ci, on crée un fichier objet ayant un nom spécifique qui dépend des fichiers .cpp et .h ayant le même nom. Le makefile répètera ce processus pour tous les objets “.o” à créer.

Ensuite, on a “$(CC) $(CFLAGS) $(CXXFLAGS) -c $<” pour faire la compilation de nos classes. En effet, on fait la compilation du fichier “.cpp” avec “-c $<” où le flag “-c” spécifie les fichiers à compiler et “$<” la première dépendance, soit “%.cpp”, le fichier cpp spécifié. On fait alors cette compilation avec des options spécifiques grâce aux flags CFLAGS et CXXFLAGS qui ont été décrits dans la partie “Makefile_common.txt” puisqu’on compile des fichiers cpp et dû à l’utilisation des macros se trouvant dans la librairie “-I/usr/include/simavr”.

“clean:”

Il s’agit de la règle qui supprime les fichiers de manière automatique. On utilise la variable REMOVE décrite précédemment. Cette règle supprime tous les fichiers .d et .o. En mettant une “ * ” devant chaque type de fichier, on n’a pas à spécifier le nom des fichiers .d ou .o à supprimer.

Makefile “exec_dir”

“include $(PATHLIB)/../Makefile_common.txt”

Dans ce “Makefile”, plusieurs déclarations de variables ont été enlevées. À la place, le “Makefile” inclut désormais le fichier “Makefile_commons.txt” décrit précédemment qui déclare les variables LIB, OBJ, HEADERS, REMOVE, CC, CFLAGS et CXX. Cela permet d’avoir les déclarations dans un seul “Makefile” pour faciliter les changements de leur valeur.

“-L $(PATHLIB) -l$(LIBS)”

Sur la ligne 118 du “Makefile” on a ajouté le terme “-L” et transformé “-lm$(LIBS)” en “-l $(LIBS)”. Le terme “-L” permet de spécifier le répertoire où se trouve la librairie tandis que le terme “-l $(LIBS)” permet de spécifier le nom de la librairie.