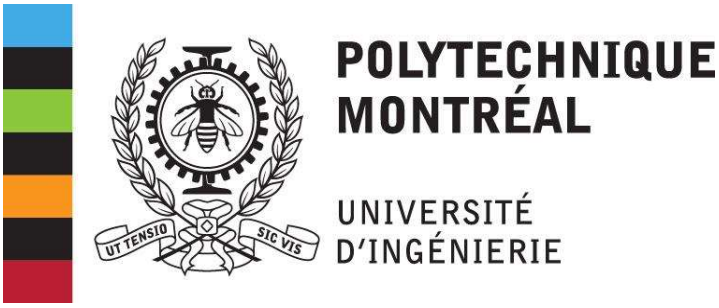

Département de génie informatique et génie logiciel

Cours INF1900:
Projet initial de système embarqué

Travaux pratiques 7 et 8

Production de librairie statique et stratégie de débogage

Par l'équipe

No 1012

Noms:

Jad Ben Rabhi
Képhren Delannay Sampany
Amine Ghabia
Amine Zerouali

Date:
13 Mars 2023

Partie 1 : Description de la librairie

1. Enums

1.1. Direction

Cet enum définit les valeurs à mettre sur les pin du portB définissant la direction de rotation des moteurs du pont en H. On peut ainsi déplacer le robot vers l'avant et l'arrière ainsi qu'effectuer des rotations sur lui-même vers la droite et la gauche.

1.2. EtatLed

Cet énum définit comment allumer la led sur la carte en rouge, vert ou l'éteindre en assignant les valeurs adéquates au portA.

2. Classe Bouton

Classe de type utilitaire définissant un bouton fonctionnant avec une interruption ISR pouvant à la fois implémenter le bouton poussoir de la carte du robot ou un bouton externe.

2.1. Constructeur par défaut et constructeur avec masque

Par défaut, un bouton est construit avec le masque *0x04* (correspondant à la configuration de base du bouton poussoir sur le deuxième in du port D). On peut aussi passer un masque alternatif au bouton lors de sa construction pour accommoder toutes les dispositions matérielles.

2.2. peutChangerEtat_

Cette variable privée de type booléen permet d'empêcher le bouton de changer d'état lorsque celui-ci est maintenu enfoncée. Elle est initialisée à vrai et devient fausse lorsque le debounce est effectué puis redevient vrai au relâchement du bouton.

2.3. Init()

Cette fonction configure le bouton pour l'utiliser avec une interruption ISR. Elle utilise la configuration classique et initialise aussi la variable de type booléen *peutChangerEtat_* à vrai.

2.4. estActif()

Cette variable booléenne est la variable publique utilisée par le système pour déterminer si le bouton est effectivement enfoncé. Cette booléenne est vraie uniquement si le debounce a été effectué et que le bouton peut changer d'état (voir section 2.2).

2.5. debounce()

Méthode de debounce implémentée à l'aide de deux mesures de l'état du PIND séparées par un intervalle défini dans la constante **DEBOUNCE_TIME.**

3. Classe Timer1

Classe créant un objet qui peut être utilisé comme une minuterie utilisant les interruptions ISR. Son implémentation requiert la définition d'une routine d'interruption avec le vecteur TIMER1_COMPA_vect.

3.1. Constructeur

Le constructeur d'un objet de type Timer1 prend en paramètre un pointeur vers une variable de type **int volatile** pour fonctionner avec les interruptions et donne cette valeur au pointeur privée de l'objet.

3.2. partirMinuterie(int duree)

Cette fonction prend la durée de la minuterie en paramètre et configure les registres adéquats du timer1 afin de produire une interruption après la durée spécifiée. Elle fonctionne avec le mode CTC du timer1 avec une horloge divisée par 1024.

3.3. attendre(int duree)

Cette fonction est une implémentation d'une fonction wait permettant d'attendre une durée variable (impossible à faire avec un delay) et produit une interruption après la durée spécifiée en paramètre.

4. Classe Moteur

Classe permettant de construire un objet de type moteur pour contrôler le pont en H du robot à l'aide de signaux PWM et du timer0. La direction des moteurs est contrôlée en modifiant la valeur du signal renvoyée sur le PortB (configuration du robot faite ainsi).

4.1. Constructeur

Le constructeur par défaut du Moteur ne prend pas de paramètres et initialise simplement les attributs privés de celui-ci. Ceux-ci sont la direction du moteur qui par défaut est Avant, la vitesse du moteur de la roue droite et la vitesse de celui de la roue gauche initialement à 0.

4.2. ajustementPWM(int a, int b)

Cette méthode prend deux paramètres (a et b) qui sont la vitesse des moteurs droits et gauches respectivement. La méthode configure ensuite les registres du Timer0 et des compteurs *OCR0A* et *0CR0B*

4.3. changerDirection()

Cette méthode prend une direction en paramètre (voir section 1.1) et modifie la variable privée du moteur gardant en mémoire la direction du robot puis ajuste le portB avec les valeurs de l'enum Direction correspondante.

4.4. getDirection()

Méthode permettant de renvoyer la direction actuelle du robot.

5. Classe RS232

Classe qui implémente la communication série RS232 via un port UART, en utilisant l'ATMega328P.

5.1 InitialisationUART()

La méthode initialisationUART initialise les registres nécessaires pour configurer l'UART du microcontrôleur en mode **asynchrone** avec une vitesse de transmission de **2400 bauds**. Les registres de contrôle des **interruptions**, de contrôle des erreurs et de contrôle de la vitesse de transmission sont configurés pour permettre la **réception** et la **transmission** de données.

5.2 transmissionUART(uint8_t donnee)

Cette méthode prend en paramètre une variable donnée de type uint8_t, et la transmet via l'UART. Elle attend que le registre de données soit prêt à être écrit, puis stocke la donnée dans une variable membre et l'écrit dans le registre de données afin de pouvoir la transmettre via l'UART.

6. Classe CAN

Classe qui implémente un convertisseur analogique-numérique (CAN) sur l'ATMmega324P. Ce dernier fonctionne en graduant une plage délimitée par deux voltages en 210 valeurs.

6.1 Constructeur

Le constructeur can() initialise les registres nécessaires pour configurer le CAN du microcontrôleur en mode de **référence de tension analogique externe**. Le registre de contrôle du convertisseur est configuré pour activer le convertisseur, sans démarrer de conversion pour l'instant. De plus, aucun **déclenchement automatique** ni **interruption** ne suivra la fin d'une conversion. La fréquence de l'horloge du convertisseur est divisée par **64**.

6.2 Destructeur

Le destructeur ~can() arrête le convertisseur pour **économiser** la consommation d'énergie.

6.3 Lecture(uint8_t pos)

La méthode lecture(uint8_t pos) effectue une conversion sur l'entrée spécifiée par la variable pos, et retourne un résultat sur **16 bits**. Tout d'abord, la méthode configure le registre de multiplexage du CAN pour sélectionner l'entrée désirée, puis elle démarre la conversion et attend la fin de la conversion. Une fois la conversion terminée, la méthode lit le résultat sur 16 bits et retourne la valeur lue. Enfin, pour une question de précision, seuls les **10 bits** de poids faibles seront significatifs.

7. Classe LED

La classe Led permet de contrôler une LED bicolore connectée à une broche du microcontrôleur.

7.1 Constructeur()

Led(EtatLed l) est le constructeur de la classe. Ce dernier initialise l'état de la LED à l'état l qu'il prend en paramètre, et ne retourne aucune valeur.

7.2 changerCouleur(EtatLed l)

Cette méthode permet de changer la couleur de la LED en passant en paramètre la nouvelle couleur sous forme d'une valeur de l'enum EtatLed défini plus haut.

7.3 changerCouleurAmbre(Timer1 t, int t_vert, int t_rouge, int n)

Cette méthode permet de faire clignoter la LED en alternant entre les couleurs vert et rouge et ainsi obtenir une couleur ambre. La durée de chaque couleur est définie par les paramètres t_vert et t_rouge qui représentent des temps en millisecondes. Le paramètre n permet de spécifier le nombre de fois que la boucle doit tourner. Cette méthode utilise une instance de la classe Timer1 pour synchroniser les changements de couleur de la LED.

7.4 recupererCouleur()

Cette méthode permet de récupérer l'état actuel de la LED sous forme d'une valeur de l'enum EtatLed. Elle retourne un objet de type EtatLed.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Makefile pour la librairie

Le makefile crée la librairie et place le fichier .a dans un répertoire *build.* Nous avons décidé d'écrire notre propre makefile plutôt que de partir de celui existant pour des raisons de simplicité, de facilité de compréhension et de contrôle du résultat.

La règle pour créer la librairie utilise le compilateur avr-ar pour compiler le fichier **librairie.a** à partir des fichiers **librairie.o mémoire_24.o** et **debug.o**.

```
$(OUTPUTFILE): librairie.o memoire_24.o debug.o
	avr-ar rcs $@ $^
```

La règle **make install** crée un répertoire *build,* compile la librairie grâce à la règle précédente, déplace la librairie dans le répertoire et clean les fichiers temporaires (.o, .a et .d)

```
install: $(OUTPUTFILE)
	mkdir -p $(INSTALLDIR)
	cp -p $(OUTPUTFILE) $(INSTALLDIR)
	for file in $(CLEANEXTS); do rm -f *.$$file; done
```

Les règles restantes permettent de créer les fichiers temporaires et dépendances de la librairie.

```
librairie.o: librairie.h
debug.o: debug.h
memoire_24.o: memoire_24.h
```

Makefile de l'executable

Ce fichier makefile est le même que celui utilisé depuis le début de la session avec quelques changements apportés afin de supporter la librairie.

```
# Inclusions additionnels (ex: -I/path/to/mydir)
INC= -I ../lib

# Libraires a lier (ex: -lmylib)
LIBS= -lrairie -L ../build
```

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

Barème: vous serez jugé sur:

- *La qualité et le choix de vos portions de code choisies (5 points sur 20)*
- *La qualité 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)*