POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

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 1617

Noms:

Evan Blanc

Ali Mouchahid

Ryan Lahbabi

Ahmed Zghal

Date:

31 Octobre 2022

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 au bénéfice de tous les membres de l'équipe.

Notre librairie est composée de plusieurs classes ayant chacune des fonctions bien spécifiques. Nous allons décrire ces classes en montrant leur utilité, leurs objectifs, leurs constructeurs et les méthodes implémentées.

Fichier Bouton.cpp, *en-tête Bouton.h* :

Fonction bool appuiBouton():

Cette fonction est de type booléen et permet de vérifier l'appui du bouton avec un délai anti rebond de 10 ms afin de compter le signal qu'une fois par appui. Nous utilisons ici un masque pour vérifier si notre PIN a bien été activée par l'appui du bouton. Dans le cas où le bouton est appuyé, elle retourne True, sinon elle retourne False.

Fichier Del.cpp, *en-tête Del.h* :

Del::Del()

Cette fonction permet d'établir le PORT de sortie par défaut au PORTA que l'on pourra changer par la suite.

void Del::SetCouleurLumiere (Couleur couleur)

Cette fonction permet de configurer la couleur que l'on veut donner à notre DEL. Nous avons trois possibilités : DEL verte, DEL rouge ou DEL éteinte. Pour différencier ces trois couleurs, nous avons déterminé nos 3 états dans notre en-tête à l'aide d'un enum class. Une fois l'état déterminer en paramètre, le PORTA indiquera la couleur à donner sur la DEL prédéfinie dans notre en tête.

void Del::clignoter(uint8_t nbFois, Couleur couelur)

Cette fonction permet de faire clignoter notre DEL le nombre de fois que l'on veut en l'indiquant dans la variable nbFois. La couleur de la DEL est aussi changeable par le biais de la variable couleur en appelant la fonction setCouleurLumiere. Les fonctions delay_ms de #include <util/delay.h> permettent d'établir un délai plus ou moins court que nous avons établie à 250 ms afin de faire clignoter la DEL à une vitesse modérée.

A noter que nous avons établi par défaut la DEL sur le PORTA mais si jamais nous sommes amenés à changer le branchement de notre robot nous pouvons changer le PORT voulu en changeant la variable PORT le DDR dans notre fonction Del().

Fichier Moteur.cpp, *en-tête Moteur.h* :

Ce fichier permet de configurer le comportement des deux moteurs de notre robot notamment pour donner des instructions aux roues de notre robot par le biais de l'ajustement de notre PWM.

void initialisationMinuterie()

Cette fonction configure la minuterie afin d'ajuster le PWM, nous utilisons ici un prescaler qui divise l'horloge par 8 et le PWM est en mode Phase Correct sur 8 bit. Le mode de comparaison choisi fait que nous réinitialisons OCnA et OcnB sur *Compare Match* lors de l'incrémentation du compteur de la minuterie et fixons OCnA et OCnB sur *Compare Match* lors de la décrémentation du compteur. A noter que nous utilisons la minuterie 0 qui est sur 8 bit et qui suffit à combler le besoin de notre classe.

Moteur::Moteur(uint8_t pinDirectionDroite, uint8_t pinDirectionGauche)

Cette fonction prend deux variables pinDirectionDroite et pinDirectionGauche afin de différencier le comportement du moteur de droite et celui de gauche. PB3 et PB4 sont les pins enable qui transmettent le courant au PWM.

void Moteur::avancer(uint8_t pourcentagePWM)

Afin de configurer notre PWM nous utilisons les registres de comparaison OCR0A pour le moteur de gauche et OCR0B pour le moteur de droite. Ces deux derniers sont configurés à la valeur maximale du PWM soit 255 étant donné que les registres sont sur 8 bits afin de faire avancer les roues du robot. Nous convertissons sur 255 la valeur entrée en paramètre afin d'être capable d'entrer des pourcentages pour faciliter la manipulation de la méthode avancée. Nous avons ajusté à l'aide du complément à 1 les pins de direction sur le PORTB afin de faire avancer le robot.

void Moteur::reculer()

Même chose que pour la fonction void Moteur :: avancer(cependant les pins de directions du PORTB ont été forcer à 1 afin d'indiquer au robot de reculer.)

void Moteur::arret()

Cette fonction sert à arrêter les moteurs de notre robot et donc faire arrêter ses roues, pour cela nous avons tout simplement assigner les valeurs de nos registres de comparaison à 0 pour indiquer une vitesse nulle.

void Moteur::tourner (uint8_t pourcentageRoueDroite, uint8_t pourcentageRoueGauche)

Cette fonction permet d'ajuster le pourcentage de PWM que l'on veut donner à chaque moteur à l'aide des deux variables *pourcentageRoueDroite* et *pourcentageRoueGauche.* Par conséquent, nous pourrons choisir la vitesse des roues du robot et faire tourner notre robot à gauche ou à droite de façon brusque ou progressive. Les pourcentages que nous indiquerons seront converties par nos deux registres deux comparaisons sur 255 bits.

Fichier Afficher.cpp, *en-tête* Afficher*.h* :

Ces fonctions permettent l'affichage du contenu mémoire flash de notre Atmega324. Grâce au port RS32 et au cable serieViaUSB.

void Afficher::USART_Init().

Cette méthode a pour but d'initialiser l'USART avant la transmission d'informations
Elle met en marche le receveur RX et le transmetteur TX du RS232 et fixe la transmission des données par paquets 8 bits avec 2 stop bits.

void Afficher::afficherCaractere(const char caractere),

Cette méthode fait en sorte d'attendre que le buffer de transmission soit vide avant d'y intégrer les données prises en paramètres.

void Afficher::afficherChaineCaractere(const char *caractere)

Cette méthode permet d'afficher une chaîne de caractères prend en paramètre le pointeur du paramètre de la fonction précédente. Celle-ci suite à une condition spécifique appelle afficher Caractere.

void Afficher::lireCaractere()

Cette méthode attend que les données soient reçues dans le receveur RX et retourne le registre UDR0 afin de permettre au micro contrôleur de lire les données inclus dans le registre.

Fichier Interrupt.cpp, *en-tête* Interrupt*.h* :

Ce fichier et son en-tête nous permettent de contrôler par interruption les boutons poussoir, que ce soit de la carte mère ou du breadbord, nous utiliserons dans ce cas là une fonction ISR qui implémente les variables volatiles à changer et qui sera accompagner de l'appel de la méthode suivante pour l'initialiser.

BouttonInterruption::BouttonInterruption(**uint8_t** id, ModeInterruption mode)
Cette méthode représente l'initialisation de la boucle d'interruption, on y retrouve les fonctions cli() et sei() permettant respectivement de bloquer toutes les interruptions et de recevoir à nouveau les interruptions. Nous avons aussi fait en sorte d'avoir un paramètre id permettant d'insérer le vecteur que nous voulons utiliser qui permettra le bitshift nécessaire dans le registre EIMSK. En outre, nous avons un switch case qui permet de configurer le registre EICRA en fonction du mode que nous souhaitons, soit *falling edge*, *any edge, rising edge ou low level.*

Fichier Mémoire_24.cpp, *en-tête* Mémoire_24.*h* :

Ce fichier et son en-tête nous permettent de manipuler la mémoire externe de notre robot à l'aide du protocole sérieViaUSB. Grâce à celle-ci on peut lire et écrire des données

Plusieurs méthodes sont implémentées ici tel que :

uint8_t lecture(const uint16_t adresse, uint8_t *donnee):

Cette méthode permet d'effectuer la lecture du fichier implémenter d'une donnée à la fois.

uint8_t lecture(const uint16_t adresse, uint8_t *donnee,const uint8_t longueur):

Cette méthode permet d'effectuer la lecture du fichier implémenter d'un bloc de données à la fois allant de 0 à 127 caractères.

uint8_t ecriture(const uint16_t adresse, const uint8_t donnee):

Cette méthode copie une donnée à la fois à l'adresse de la mémoire externe.

uint8_t ecriture(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur): Cette méthode copie un bloc de données allant de 0 à 127 caractères à l'adresse de la mémoire externe.

Fichier can.cpp, *en-tête* can.*h* :

Le but de ce fichier et de son en-tête est de créer et construire la class can qui permet de configurer et de permettre l'accès au convertisseur analogique numérique (CAN) du microcontrôleur grâce à ses méthodes

can::can():

le constructeur can() permet d'initialiser le convertisseur et de l'activer sans pour autant initier une conversion

can::~can():

Le destructeur ~can() permet d'arrêter le convertisseur et de le rendre inactif.

can::lecture(uint8_t pos) :

Cette méthode prend en paramètre la position du PORT A et retourne la valeur numérique sur 16 bits correspondant a la valeur analogique du PORT choisis en paramètre (la fonction n'est valable que pour le PORT A) seul les 10 bits de poids faibles sont significatifs.

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.

Nous avons deux Makefile, celui de la librairie et celui de l'exécutable :

Makefile Librairie:

PROJECTNAME= libtest

Cette commande permet de nommer la librairie que l'on utilise.

PRJSRC= $(wildcard * .cpp)

Cette commande permet de définir l'ensemble des fichiers .cpp dans le répertoire que notre Makefile va compiler à chaque commande «make ». La commande *wildcard* permet d'indiquer au makefile de prendre en compte tous les fichiers .cpp du dossier lib

TRG=$(PROJECTNAME).a

Cette commande permet d'indiquer l'extension du fichier libstatique qui nous permettra d'avoir comme cible par défaut libstatique.a lors de la compilation du makefile. Ce fichier sera très important pour importer les libraires lors de la compilation du makefile de l'exécution.

$(TRG): $(OBJDEPS)

avr-ar crs $(TRG) $(OBJDEPS)

Cette commande permet de faire l'implémentation de la cible avr-ar par le biais du flag crs.

Makefile de l'éxecutable:

PROJECTNAME= tp7.cpp

Cette commande permet de nommer le projet que l'on utilise.

PRJSRC= main.cpp

Cette commande permet de définir l'ensemble des fichiers .cpp dans le répertoire que notre Makefile va compiler à chaque commande «make ». Pour ce Makefile nous allons compiler seulement main.cpp.

INC=-I../lib

Cette commande permet de spécifier le chemin des inclusions que l'on voudrait ajouter à notre Makefile. Dans notre cas on veut inclure le dossier lib qui se trouve dans le dossier tp7

LIBS= -L../lib -ltest

Cette commande permet de spécifier le chemin vers la librairie que l'on veut lier à notre Makefile.

debug: CFLAGS += -DDEBUG

debug: CXXFLAGS += -DDEBUG

debug: install

Cette commande permet de définir une règle make debug qui sera capable de faire un make install grâce à la cible *"debug: install"* et de permettre le débogage des DEBUG_PRINT que l'on a inséré dans notre code à l'aide de serieViaUSB -l.

Le rapport total ne doit pas 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)*