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 2728

Noms:

Souhayl Sejjari
Mohamed Omar Zedek
Sebastian Cristescu
Mohamed Yassir Merzouki

Date:
11 mars 2024

Partie 1 : Description de la librairie

Cette librairie regroupe différentes classes qui nous permettent de facilement programmer différentes actions pour notre robot. (Broche : 1-8, Pin : 0-7)

Classes:

DEL:

Cette classe contient 5 méthodes afin d'allumer une LED ou de l'éteindre. La classe utilise par défaut les broches A0-A1 respectivement en positif-négatif, mais ceux-ci peuvent être modifiés lors de la construction de l'objet.

LED(port, brochePos, brocheNeg): Ce constructeur est une méthode qui prend en paramètre le port sur lequel la LED sera branchée (A, B, C ou D), ainsi que le numéro des broches reliées aux bornes positives et négatives. Cette méthode est un constructeur, elle retourne alors un objet LED.

allumerRouge(): Cette méthode ne prend rien en paramètre et allume la LED en rouge. Les broches utilisées sont déclarées lors de la construction de l'objet. Cette méthode ne retourne rien.

allumerVert(): Cette méthode ne prend rien en paramètre et allume la LED en vert. Les broches utilisées sont déclarées lors de la construction de l'objet. Cette méthode ne retourne rien.

allumerAmbre() : Cette méthode ne prend rien en paramètre et allume la LED en ambre pendant 2 fois la durée DELAI_COULEUR_AMBRE soit 20 ms. Les broches utilisées sont déclarées lors de la construction de l'objet. Cette méthode ne retourne rien.

allumerAmbre(dureeEnMS) : Cette méthode prend en paramètre une durée en millisecondes pour un maximum de 65535 ms. Celui-ci est converti en cycles, puis la méthode rentre dans une boucle pour le nombre de cycles calculés. Cette méthode est donc bloquante pour la durée spécifiée. Cette méthode ne retourne rien.

eteindre(): Cette méthode ne prend rien en paramètre et éteint la LED. Les broches utilisées sont déclarées lors de la construction de l'objet. Cette méthode ne retourne rien.

CAN:

Cette classe nous a été fournie dans le cadre du cours. Elle permet de recevoir des données du convertisseur analogique/numérique du microcontrôleur. Elle ne contient qu'une seule méthode de lecture des données. Les données analogiques peuvent seulement être lues du port A. Cette classe utilise deux registres: ADMUX et ADCSRA.

lecture(pos): Cette méthode prend en paramètre la position de la broche d'où viennent les informations analogiques. *pos* doit être entre 0 et 7 inclusivement (uint8_t) et doit se retrouver sur le port A. Cette méthode retourne la valeur numérique en uint16_t, correspondant aux données analogiques. Ce n'est pas une fonction bloquante, mais elle utilise les registres ADMUX et ADCSRA.

MEMOIRE:

Cette classe nous a été fournie dans le cadre du cours. Elle permet de lire et écrire des données dans la mémoire EEPROM. Elle est composée de 7 méthodes, mais 4 d'entre elles sont importantes pour nous. Elle utilise les registres: TWSR, TWBR, TWCR et TWDR.

*lecture(adresse, *donnee)*: Cette méthode prend en paramètre l'adresse à laquelle il faut lire un char et un pointeur où enregistrer cette donnée. L'adresse est un uint16_t et la donnée un uint8_t. Ce n'est pas une méthode bloquante. Elle retourne la valeur lue de la mémoire en uint8_t.

*lecture(adresse, *donnee, longueur)*: Même méthode que ^, mais peut lire un bloc de données entre 0 et 127 bites qui est spécifié dans le paramètre longueur, un uint8_t. Cette méthode retourne 0.

ecriture(adresse, donnee): Cette méthode prend en paramètre un char et l'adresse à laquelle il faut l'écrire. L'adresse est un uint16_t et la donnée un uint8_t. Cette méthode retourne 0 et n'est pas bloquante.

*ecriture(adresse, *donnee, longueur)*: Même méthode que ^, mais peut écrire un bloc de données entre 0 et 127 bites qui est spécifié dans le paramètre longueur, un uint8_t. Cette méthode retourne 0.

UART:

Un regroupement de fonctions (utiliser une classe ne semble pas naturel) qui permet d'utiliser la communication RS232 afin d'écrire et recevoir de l'information avec le microcontrôleur. Elle est composée de 3 fonctions qui utilisent les registres: UBRR0H, UBRR0L, UCSR0A, UCSR0B, UCSR0C, UDR0. Ce ne sont pas des fonctions bloquantes, par contre, il faut s'assurer d'appeller initialisationUART() si l'on veut utiliser les deux autres fonctions.

initialisationUART(): Cette méthode ne prend rien en paramètre. Elle sert à initialiser les registres pour permettre une communication UART. Elle ne retourne rien.

transmissionUART(donnee): Cette méthode sert à transmettre des données, qui sont en paramètre, au microcontrôleur, plus précisément dans UDR0. Elle ne retourne rien.

receptionUART(): Cette méthode ne prend rien en paramètre et permet de recevoir des informations du microcontrôleur. Plus précisément, les données dans UDR0. Retourne un uint8_t.

PWM:

Cette classe permet de générer un signal PWM sur certains pins: B04, B05, D15, D16, D27 ou D28 (Lettre Port-Numéro Timer-Numéro Broche) qui doivent être spécifié lorsque l'objet PWM est construit grâce à l'enum class "PinPWM", avec un pourcentage de type uint8_t. Elle possède 3 méthodes qui permettent d'interagir avec le pourcentage du PWM. Ce ne sont pas des méthodes bloquantes et elles utilisent les registres: TCNT0, OCR0A, TCCR0A, TCCR0B, DDRB, OCR0B, TCNT1, OCR1A, TCCR1A, TCCR1B, DDRD, OCR1B, TCNT2, OCR02A, TCCR2A, TCCR2B, DDRD, OCR2B dépendamment de quelles broches ont été choisies. Aucune des méthodes de cette classe n'est bloquante.

PWM(pinPWM, pourcentage): Cette méthode est le constructeur de l'objet et prend en paramètre la pin sur laquelle on souhaite générer un signal PWM en sortie, ainsi qu'un pourcentage de PWM

voulu qui est par défaut défini à 0. Ce constructeur permet la configuration correcte des registres selon les paramètres à l'aide d'un switch-case adéquat. Ce constructeur retourne un objet PWM.

setPourcentage(pourcentage): Cette méthode prend en paramètre un uint8_t qui sert de pourcentage pour fixer un pourcentage pour le signal PWM. Cette méthode ne retourne rien.

augmenterPourcentage(pourcentage): Cette méthode prend en paramètre un uint8_t pour augmenter le pourcentage du signal PWM. Cette méthode ne retourne rien.

diminuerPourcentage(pourcentage): Cette méthode prend en paramètre un uint8_t pour augmenter le pourcentage du signal PWM. Cette méthode ne retourne rien.

MOTEUR:

Cette classe permet de facilement activer les moteurs du robot pour les faire tourner vers l'avant ou l'arrière, et à une certaine vitesse. Celle-ci se base sur une autre classe PWM, qui permet d'initialiser les signaux de PWM nécessaires à la rotation des roues. Aucune des méthodes de cette classe n'est bloquante.

Moteur(port, brocheDirection, pinPWM): Ce constructeur permet de créer un objet Moteur en prenant les paramètres port, brocheDirection et pinPWM. Les paramètres port et brocheDirection permettent de configurer les broches de direction pour choisir le sens de rotation du moteur, et le paramètre pinPWM permet d'instancier un objet PWM pour délivrer du courant au moteur afin de gérer sa vitesse de rotation.

avancer(pourcentage): Cette méthode prend en paramètre le pourcentage désiré pour que le moteur roule vers l'avant avec un pourcentage de vitesse maximale choisi, qui est de 100% par défaut. La méthode ne retourne rien.

reculer(pourcentage): Cette méthode prend en paramètre le pourcentage désiré pour que le moteur roule vers l'arrière avec un pourcentage de vitesse maximale choisi, qui est de 100% par défaut. La méthode ne retourne rien.

arret(): Cette méthode ne prend aucun paramètre. Elle permet de mettre la vitesse du moteur à 0 afin d'immobiliser la roue. La méthode ne retourne rien.

MOTRICITÉ:

Cette classe permet de facilement activer les deux moteurs du robot pour le faire avancer, reculer, tourner à droite et à gauche. Celle-ci se base sur une autre classe Moteur, qui permet d'initialiser les broches pour que les moteurs tournent vers l'avant, l'arrière ou qu'ils s'arrêtent. Cela permet de contrôler les deux moteurs simultanément pour faire des actions précises.

Motricite(moteurGauche, moteurDroit): Ce constructeur prend en paramètre deux moteurs qui serviront de moteur droit et gauche pour le robot. Il ne construit donc pas lui-même les moteurs. Ce constructeur se contente d'attribuer aux attributs de la classe des moteurs qui seront ensuite contrôlés. Il retourne un objet Motricité.

arret(): Cette méthode ne prend rien en paramètre et sert à arrêter les deux moteurs. La méthode

ne retourne rien.

marcheAvant(): Cette méthode prend en paramètre un uint8_t pourcentage, qui permet de contrôler la vitesse à laquelle les deux moteurs feront avancer tout le robot. La méthode ne retourne rien.

marcheArriere(): Cette méthode prend en paramètre un uint8_t pourcentage, qui permet de contrôler la vitesse à laquelle les deux moteurs feront reculer tout le robot. La méthode ne retourne rien.

tournerDroite(): Cette méthode prend en paramètre deux uint8_t, le degré de rotation et la vitesse à laquelle on veut que le robot tourne vers la droite. Étant donné que nous n'avons pas suffisamment d'informations sur le robot (vitesse maximale exacte des moteurs, distance entre les roues), la rotation ne sera pas exacte même si à une vitesse de 50%, elle se rapproche grandement de l'exactitude. La méthode ne retourne rien.

tournerGauche(): Cette méthode prend en paramètre deux uint8_t, le degré de rotation et la vitesse à laquelle on veut que le robot tourne vers la gauche. Étant donné que nous n'avons pas suffisamment d'informations sur le robot (vitesse maximale exacte des moteurs, distance entre les roues), la rotation ne sera pas exacte même si à une vitesse de 50%, elle se rapproche grandement de l'exactitude. La méthode ne retourne rien.

twerk(): Cette méthode prend en paramètre deux uint8_t, le nombre et l'angle de rotation. Cette méthode fait une sorte de danse avec le robot mais n'a pas tant d'utilité que ça, elle permet seulement de s'assurer que les branchements sont corrects étant donné que toutes les méthodes de la classe sont mises en œuvre grâce à cette méthode. La méthode ne retourne rien.

TIMERCTC / TIMERCTC8BITS / TIMERCTC16BITS: **n =** numero du timer

Ces 3 classes permettent d'implémenter le mode **CTC** avec prescaler de 1024 de l'ensemble des timers disponibles. La classe **TimerCTC** correspond à la classe de base dont héritent **TimerCTC8bits** et **TimerCTC16bits**. La classe **TimerCTC8bits** permet l'utilisation du mode CTC sur le **timer0** et le **timer2** tandis que la classe **TimerCTC16bits** le permet pour **timer1**. Pour créer un objet timer, il faut fournir l'unité de temps qui sera utilisée à avec **enum** "Unite" et le timer utilise.

initialiserTimer(): Cette méthode est spécifique a **TIMERCTC8BITS / TIMERCTC16BITS,** elle permet de configurer les registres des timers aux bonnes valeurs. C'est une méthode privée seulement à des fins d'implémentation. Les 3 classes utilisent les registres **TCNTn**, **TCCRnA**, **TCCRnB**, **OCRnA** et **TIMSKn**.

getUniteTemps(): Cette méthode retourne l'unité de temps utilisée par l'objet timer actuel.

setUniteTemps(): Cette méthode permet de changer l'unité de temps actuelle d'un objet timer.

demarrerChronometre(): Cette méthode ne prend aucun paramètre.Il s'agit d'une méthode permettant de lancer un chronomètre qui s'incrémente de 1 à chaque unité de temps (seconde ou milliseconde). Son utilisation nécessite un ISR qui incrémente une variable volatile.

arreterChronometre(): Méthode qui permet d'arrêter le chronomètre en sauvegardant le dernier cycle atteint par le registre TCNT**n** correspondant. Il est uniquement pertinent de l'utiliser lorsque la méthode **demarrerChronometre()** a été utilisée précédemment.

reprendreChronometre(): Il s'agit d'une méthode permettant de reprendre le chronomètre exactement là où il s'était arrêté. Il est uniquement pertinent de l'utiliser lorsque la méthode **arreterChronometre()** a été utilisée précédemment.

lancerEvenementIntervalle(): Il s'agit d'une méthode qui prend en paramètre une durée dans l'unité actuelle (ms ou secondes) et génère une interruption à chaque fois que cette durée de temps est passée. Il est nécessaire d'utiliser un ISR(TIMER**n**_COMPA_vect) pour effectuer les opérations désirées à chaque intervalle de temps dans le main(). Il est essentiel de mentionner que cette durée est limitée à 8 secondes environ pour le timer 16 bits et environ 36 ms pour les timers 8 bits.

convertirTempsEnCycles(): Il s'agit d'une méthode privée permettant de convertir une durée en ms ou en secondes en nombre de cycles de clk .Elle est seulement utilisée à des fins d'implémentation de la classe.

INTERRUPTIONS EXTERNES:

Cette classe permet simplement d'activer une interruption externe sur une des broches disponibles à cet effet **(B2,D2,D3)** lorsqu'un évènement d'horloge spécifique a lieu (front montant, front descendant, les 2 fronts ou encore niveau bas). Ainsi, pour créer un objet de cette classe, il faut spécifier en paramètre la broche utilisée ainsi que l'évènement d'horloge spécifique qui provoque l'interruption. La classe utilise le registre **EIMSK** et **EICRA**.

L'ensemble de ces méthodes de cette classe ne prennent aucun paramètre, elles utilisent uniquement les attributs privés de la classe : broche_ et conditionClock_ (front montant, descendant, etc…).

Pour le fonctionnement correct des méthodes, il est nécessaire d'inclure la routine ISR avec l'identifiant correspondant à INT0 / INT1 / INT2 dans le main.

activerInterruptionExterne: il s'agit d'une méthode permettant de configurer les différents registres nécessaires pour l'activation des interruptions externes et mettre la broche correspondante en entrée.

desactiverInterruptionExterne: il s'agit d'une méthode qui rend la broche spécifiée insensible aux interruptions externes.

configurerPort: Il s'agit d'une méthode privée à des fins d'implémentation. Elle est destinée à mettre la broche correspondante en entrée et à configurer le registre d'interruption EIMSK pour rendre la broche correspondante sensible aux interruptions.

configurerConditionInterruption: Il s'agit d'une méthode privée à des fins d'implémentation. Elle est destinée à correctement configurer le registre de contrôle EICRA.

DEBUG:

Permet d'utiliser des macros DEBUG qui peuvent être insérées dans des programmes qu'on

utilise pour voir la valeur de variables. Ceux-ci peuvent être activés en utilisant make debug.

DEBUG_INIT(): Appelle la fonction initialisationUART(). Voir informations de la fonction dans la section UART. La fonction ne retourne rien et ne prend rien en paramètre.

DEBUG_PRINT(str): Passe à travers chaque char d'un *string* pour l'envoyé en utilisant transmissionUART(). Voir informations de la fonction dans la section UART. La fonction ne retourne rien.

DEBUG_VALUE(valeur): Appelle la fonction transmissionUART(). Voir informations de la fonction dans la section UART. Transmet directement la valeur en paramètre dans transmissionUART(). La fonction ne retourne rien.

DEBUG_LARGE_VALUE(valeur): Appelle la fonction transmissionUART(). Voir informations de la fonction dans la section UART. Envoie les bits de droite en premier, puis les bits de gauche de la valeur passée en paramètre. La fonction ne retourne rien.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Makefile pour la librairie

1. *PRJSRC= $(wildcard *.cpp)* : Avant, le Makefile prenait le nom complet de chaque fichier. Maintenant on le remplace par un *wilcard* qui nous permet d'automatiquement sélectionner tous les fichiers .cpp dans le même répertoire.
2. *TRG=$(PROJECTNAME).a* : Avant le Makefile produisait un fichier .elf. Pour une librairie, nous avons besoin d'un fichier .a.
3. *all: $(TRG)*
   *$(REMOVE) $(filter-out %.a, $(TRG) $(TRG).map $(OBJDEPS) $(HEXTRG) *.d)* : Ces commandes assurent que la cible $(TRG) doit d'abord être construite et suppriment tous les fichiers sauf ceux avec l'extension .a dans la liste des fichiers $(TRG), $(TRG).map, $(OBJDEPS), $(HEXTRG) et tous les fichiers avec l'extension .d.
4. *$(TRG): $(OBJDEPS)*
   avr-ar crs $(TRG) $(OBJDEPS) : Ces commandes utilisent avr-ar pour compiler les fichiers en une librairie selon des paramètres qui insèrent les fichiers d'objet et ajouter un index à l'archive pour un accès plus rapide

Makefile pour exécutable

1. *INC= -I ../lib* et *LIBS= -L../lib -lLibrairie* : Ces deux commandes s'assurent que notre librairie des classes et fonctions sont incluses dans la compilation de l'exécutable.