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 8094

Noms:

Samy Benchaar
Riyad Bettayeb
Soeuchelle Michel
Kaydon Mohamed

Date:
17 mars 2025

Partie 1 : Description de la librairie

Cette librairie est constituée de plusieurs classes permettant de faciliter la programmation de différentes actions pour le robot. Elle permet notamment de gérer les boutons, les lumières, la mémoire EEPROM, la conversion analogique-numérique, les timers, le contrôle des roues et la communication série.

CLASSE MINUTERIE

La classe Minuterie permet de configurer et de contrôler une minuterie en utilisant le TIMER1, sur 16 bits, avec un prescaler de 1024 pour générer des interruptions à des intervalles précis. Elle utilise le mode CTC (Clear Timer on Compare Match), ce qui permet de définir des durées spécifiques en secondes pour les interruptions.

Le constructeur de la classe Minuterie initialise le timer en mode CTC avec un prescaler de 1024, ce qui permet une plage de valeurs maximales pour des durées allant jusqu'à 9 secondes. La fonction partirMinuterie(uint16_t duree, uint16_t a) configure la minuterie pour générer une interruption après une durée spécifiée en secondes. La durée est convertie en cycles du timer, en tenant compte de la fréquence de 7813 cycles équivalent à une seconde. La minuterie peut être configurée pour utiliser l'une des deux sorties de comparaison (OCR1A ou OCR1B), selon le paramètre a. Lors de l'appel de cette fonction, l'interruption correspondante, soit le TIMSK, est activée.

La fonction arreterMinuterie() arrête la minuterie en désactivant le mode CTC et en supprimant les interruptions associées. Cette fonction arrête la minuterie et force à 0 les bits actifs des registres pour le timer. Cela arrête le comptage et annule toute minuterie en cours.

Si OCR1A est utilisé pour compter, alors PD4 sera réservé et ne pourra être exploité. De même, si OCR1B est utilisé pour compter, alors PD5 sera réservé et ne pourra être exploité.

CLASSE ROUE

La classe Roue permet de contrôler des moteurs en utilisant le PWM (Pulse Width Modulation) avec le TIMER2. La classe Roue configure les broches PD4 à PD7 pour contrôler les moteurs et utilise un timer pour générer des signaux PWM. Le constructeur initialise les broches en mode sortie et configure le TIMER2 pour travailler en mode PWM Phase Correcte 8 bits, avec un prescaler de 8.

Afin d'exploiter le PWM, les broches PD4 et PD5 doivent être connectés aux broches enables du pont H.

La méthode ajusterPWM(uint8_t intensiteA, uint8_t intensiteB) permet de modifier l'intensité des moteurs A et B en ajustant les registres OCR2A et OCR2B, contrôlant ainsi la vitesse des moteurs avec des valeurs entre 0 et 255. Les fonctions avancer(uint8_t vitesse) et reculer(uint8_t vitesse) contrôlent la direction des moteurs (avant ou arrière) tout en ajustant la vitesse, avec des versions permettant de spécifier une intensité différente pour chaque moteur (avancer(uint8_t intensiteA, uint8_t intensiteB) et reculer(uint8_t intensiteA, uint8_t intensiteB)).

La méthode arreter() arrête les moteurs en mettant les vitesses des moteurs à 0, ce qui désactive le PWM.

CLASSE BOUTON:

La classe Bouton permet de gérer un bouton-poussoir connecté sur la broche PD2, en utilisant des interruptions pour détecter les appuis. Elle est conçue pour faciliter l'utilisation de boutons poussoirs sur la carte-mère ou sur le breadboard externe.

Le constructeur par défaut initialise la broche PD2 en entrée, configure l'interruption sur INT0 pour détecter un changement d'état du bouton et active l'interruption au niveau du registre EIMSK. Il configure également le registre EICRA pour permettre la détection de l'appuie d'un bouton (donc sur front montant seulement). Cela permet au programme de réagir instantanément lorsqu'un appui est détecté, sans besoin de surveiller en permanence l'état du bouton.

Un second constructeur permet de spécifier le type de bouton à utiliser, entre celui de la carte-mère (TypeBouton::CARTE) ou un bouton connecté sur le breadboard (TypeBouton::BREADBOARD).

La fonction estAppuye() vérifie si le bouton est appuyé. Selon le type de bouton, elle vérifie l'état de la broche PD2. Si le bouton est connecté à la carte-mère, elle vérifie si la broche PD2 est à un niveau haut (appui), sinon elle vérifie si la broche est à un niveau bas pour un bouton externe sur le breadboard.

Afin d'utiliser cette classe, le IntEN doit être mis en place.

CLASSE LUMIERE:

La classe Lumiere permet de contrôler une DEL en utilisant deux broches d'un port spécifique. Elle offre des méthodes pour allumer et éteindre les lumières avec différentes couleurs (rouge, vert, ambre) en manipulant les états des broches du port spécifié.

Le constructeur de la classe Lumiere prend en paramètre un port (PORTx) ainsi que deux broches (broche1 et broche0), permettant de définir quel port et quelles broches seront utilisées pour contrôler les lumières. Le constructeur configure les broches spécifiées en mode sortie, en écrivant dans les registres de direction (DDRx) du port.

La méthode eteindreLumiere() éteint la DEL (représentées par les broches spécifiées) en les mettant à un état haut (en écrivant un 1 dans les broches correspondantes du port). La méthode allumerVert() allume la lumière verte en mettant la broche correspondante à bas et l'autre à haut. Inversement, la méthode allumerRouge() allume la lumière rouge en mettant la broche correspondant à la couleur rouge à bas et l'autre à haut. La méthode allumerAmbre() allume alternativement la lumière rouge pendant un certain délai, puis la lumière verte pendant un autre délai, créant ainsi un effet de lumière ambre.

CLASSE CAN

La classe Can permet de recevoir les données provenant du convertisseur analogique/numérique du microcontrôleur. Elle comprend une seule méthode pour la lecture des données.

Le constructeur de la classe Can initialise le convertisseur. Il configure les registres ADMUX et ADCSRA pour activer le convertisseur et définir le mode de référence de tension . Le convertisseur est configuré pour ne pas démarrer immédiatement la conversion, avec une division de l'horloge par 64.

Le destructeur arrête le convertisseur afin de réduire la consommation d'énergie lorsqu'il n'est plus nécessaire.

La méthode lecture(uint8_t pos) permet de lancer une conversion sur l'entrée analogique spécifiée par le paramètre pos (compris entre 0 et 7), en ajustant le registre ADMUX pour sélectionner la source d'entrée. Après avoir lancé la conversion, la méthode attend la fin de celle-ci, puis retourne la valeur numérique obtenue.

CLASSE RS232

La classe Uart permet de configurer et d'utiliser le module UART0 (Universal Asynchronous Receiver/Transmitter) pour la communication série sur un microcontrôleur AVR. Elle offre des méthodes pour transmettre des données et afficher des chaînes de caractères.

Le constructeur Uart() initialise la communication série en configurant le registre UBRR0 pour une vitesse de transmission de 2400 bauds. Il active la réception et la transmission en ajustant les registres UCSR0B et UCSR0C.

La méthode transmission(uint8_t donnee) permet d'envoyer un octet de donnée via UART en attendant que le buffer de transmission soit vide. La méthode afficher(char* donnee) permet d'envoyer une chaîne de caractères via UART, en envoyant chaque caractère individuellement suivi d'un caractère de nouvelle ligne (\n).

Afin d'utiliser cette classe, il faut que le jumper DbgEN soit en place. Par ailleurs les ports D0 et D1 seront utilisés pour la transimission

CLASSE MÉMOIRE:

Cette classe permet de lire et écrire des données dans la mémoire EEPROM. Elle utilise les registres: TWSR, TWBR, TWCR et TWDR.
La méthode choisir_banc() permet de sélectionner un banc de mémoire et de modifier l'adresse du périphérique en fonction du banc choisi.
Les méthodes de lecture incluent lecture(uint16_t adresse, uint8_t *donnee) pour une donnée et lecture(uint16_t adresse, uint8_t *donnee, uint8_t longueur) pour un bloc de données (max 127 octets). Ces méthodes transmettent l'adresse et la commande de lecture et récupèrent les données après avoir attendu la fin de la transmission.
La méthode ecrire_page() permet d'écrire un bloc dans une seule page de 128 octets.

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.

MAKEFILE DE LA LIBRAIRIE

Dans le Makefile de la librairie, quelques modifications y étaient apportées pour permettre une compilation de ses fichiers, et empêcher l'installation de cette compilation sur le robot directement. Ces fichiers compilés ne représentent pas des fichiers de programme à implémenter, mais des fichiers comportant des implémentations de fonctions de chacune des classes définies auparavant.

Premièrement, dans les premières variables qui furent définies dans le Makefile, la variable « ARCHIVEUR » fut ajoutée, avec la commande ARCHIVEUR = avr-ar. Elle permettait de créer une archive de tous les fichiers .o qui seraient compilés avec le compilateur avr-gcc, pour les mettre sous une archive avec une extension .a. Cette dernière serait utilisée lors du linkage des fichiers de programme avec ceux de la librairie lors de leur utilisation future. De plus, cet archiveur fut ajouté dans l'implémentation de la cible avec la commande, $(TRG) : $(OBJDEPS) $(ARCHIVEUR) -crs $(TRG) $(OBJDEPS), pour permettre la compilation des fichiers .o produits par la compilation avec l'archiveur ($(ARCHIVEUR)).

Puisque cette libraire sera compilée pour permettre une utilisation des méthodes définies dans chacune des classes d'objets implémentées, elle ne sera donc pas utilisée directement sur le robot, puisqu'elle n'est pas un programme en soi, mais des références pour le comportement de chacun des objets. Pour cela, toutes les sections reliées à la production de fichiers .hex et .elf furent retirées pour empêcher une installation de ces fichiers sur le robot lors de la compilation des futurs programmes. Les sections aussi relatant à la commande « make install » furent elles aussi enlevées, puisqu'elles utilisent les fichiers .hex pour les téléverser au robot avec l'avrdude communiquant avec l'ATmega8, pour que cette dernière la mette sur l'ATMega324PA de la carte-mère du robot.

Finalement, pour permettre une compilation efficace de tous les fichiers de la librairie sans avoir à importer individuellement chacun des fichiers qui s'y trouvent, la commande PRJSRC = $(wildcard *.cpp) fut utilisée pour la définition de tous les fichiers sources du projet, qui permettraient de compiler tous les fichiers se trouvant dans la librairie directement lors de l'appel de sa compilation avec la commande « make » dans le futur, afin de compiler tous les fichiers dans une archive. Le reste des commandes qui sont par rapport au « make clean » et la production de fichiers .o demeurent les mêmes entre les Makefile de la librairie et du fichier exécutable puisque dans les deux situations, ces fichiers seront utilisés. Dans le cas de la librairie, ils seront mis sous la forme d'une archive avec une extension .a.

MAKEFILE DU CODE (BIDON) DE TEST

Contrairement au Makefile de la librairie, le Makefile du code (bidon) de test garde la majorité de ses sections, telles que la production de fichiers .hex et .elf qui sont utiles pour l'installation du programme sur la carte-mère du robot.

Premièrement, telle que pour le Makefile de la librairie, la commande PRJSRC = $(wildcard *.cpp) fut ajouté pour permettre la compilation de tous les fichiers .cpp se trouvant dans le directoire, sans avoir à les lister individuellement. De plus, le chemin vers la librairie fut ajouté dans les inclusions additionnelles, avec la commande ../lib/, avec les « .. » permettant de remonter au directoire supérieur, et d'aller au dossier de la librairie. Dans les librairies à lier, avec la commande LIBS = -lstatique, permis de lier la librairie en l'appelant par son nom, avec -l, représentant le diminutif « lib ».

Les variables de ce fichier restent les mêmes qu'originalement, avec l'utilisation des mêmes compilateurs qu'à l'origine et des objets permettant le transfert des fichiers vers le microcontrôleur (AVRDUDE = avrdude) et la production de fichiers en format .hex (HEXFORMAT = ihex) pour leur installation future sur le robot. Dans les options de compilations, pour s'assurer de la bonne édition de liens ou du linking des dépendances, à la variable LDFLAGS, -L ../lib, fut utilisée pour indiquer avec quelle librairie faire la liaison.

Finalement, la section des commandes du Makefile demeurent les mêmes, avec les définitions des commandes « make install » et « make clean » pour permettre respectivement l'installation des fichiers de compilation sur le robot et d'effacer les fichiers de compilation précédents quand on veut faire une nouvelle compilation en appelant la commande « make ». Les mesures de productions de fichiers .hex et .elf précédemment enlevées du Makefile de la librairie demeurent pour permettre d'installer les fichiers sur la carte-mère. Seulement, l'implémentation de la cible est différente, avec la commande $(TRG) : $(OBJDEPS) $(CC) $(LDFLAGS) -O $(TRG) $(OBJDEPS) \ -lm $(LIBS), permettant de cibler tous les fichiers .o avec $(OBJDEPS) et de faire le linkage avec les fichiers de la librairie avec $(LIBS). Tel que mentionné auparavant, la commande « make clean » ainsi que la production de fichiers .o demeurent la même entre les Makefile des deux sections.