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 0203

Noms:

Maxime Grégoire
Sasha Turgeon
Damien Jean
William Komeiha

Date:
17 mars 2025

Partie 1 : Description de la librairie

Nous avons séparé notre librairie en plusieurs sections distinctes associées à des parties différentes du robot. Chacune des sections contient un fichier header ainsi qu'un fichier cpp. Le premier fichier sert à définir la classe de l'objet concerné, donc ses méthodes et ses attributs, alors que le deuxième fichier va plutôt définir les méthodes employées par l'objet. L'utilité de la librairie est d'éviter la duplication de code et de faciliter l'écriture de futurs programmes, car toutes les fonctionnalités dont nous avons besoin seront déjà codées, testées et réunies au même endroit.

Convertisseur analogique-numérique

La première partie concerne le convertisseur analogique-numérique. Celui-ci nous permet de convertir des données analogiques, signaux numériques, notamment recueillies grâce à des capteurs.

Le code pour cette classe et ses fonctions avait déjà été fourni et il se résume par une classe *can*, son constructeur et son destructeur. Elle contient également deux méthodes : *lecture()* qui retourne la valeur numérique correspondant à la valeur analogique sur le port A et *decalage()* (rajoutée nous-mêmes) qui permet de décaler les bits lus par le capteur afin de les mettre sur 8 bits au lieu de 16.

La seule dépendance matérielle est que le signal qu'on veut envoyer au can pour se faire convertir doit passer par une des broches du port A. Il est donc requis d'appliquer un masque sur *PINA* lorsqu'on veut lire une valeur d'une broche spécifique.

Gestion des interruptions

La deuxième partie concerne les interruptions. Elle est primordiale dans la gestion matérielle de la minuterie ou afin de détecter les changements au niveau d'un bouton-poussoir.

La classe utilisée se nomme *Interruption*. Elle comprend 4 fonctions différentes agissant sur le *timer1* ou *Int0.* La première, *setINT0*, permet de déclencher les interruptions sur le bouton-poussoir branché en entrée sur PD2 grâce au cavalier *IntEn*. Les trois autres fonctions permettent la gestion de la minuterie interne du robot qui n'interfère pas avec le reste de la partie logicielle(À l'inverse d'une *_delay_ms*) et qui nous permet par exemple de créer un compteur qui s'incrémente selon une certaine période.

Comme dit plus haut, un cavalier doit être installé sur *IntEn* pour le bon fonctionnement de la fonction reliée au bouton-poussoir. La gestion de la minuterie permet une grande flexibilité puisqu'on peut appeler celle-ci avec ou sans *prescaler* ainsi qu'avec des modes différents. De plus, une fonction *delay()*, qui prend un temps en millisecondes, fait pour nous la conversion en nombre de cycle d'horloge afin de passer la bonne valeur de comparaison (le calcul est basé avec un *prescaler* /1024).

Contrôle de la DEL

La troisième partie concerne le contrôle de la DEL. La DEL a été utilisée lors de presque tous les travaux pratiques, il va donc de soi qu'une implémentation permettant le contrôle de la DEL soit faite dans la librairie. Elle peut être utile pour démontrer une égalité ou une inégalité visuelle. (Ex : Vert si le robot avance)

La classe utilisée pour le contrôle de la DEL s'appelle *Led*. Comme la DEL peut être branchée sur n'importe quel broche, le constructeur de *Led* nous permet de passer en paramètre le port (A, B, C ou D) ainsi que les deux numéros de pin que nous voulons utiliser. De cette façon, l'utilisation de la DEL est très flexible, car on peut la brancher n'importe où sur les ports de la carte mère. Cette classe contient aussi les méthodes *setRed*, *setGreen* et *setOff*, qui ont comme fonctionnalité de changer la couleur de la DEL ainsi que de l'éteindre.

Comme il a été dit, la DEL peut être branchée sur n'importe quel pin de la carte mère. La seule exigence matérielle est donc que le fil soit branché sur les 2 pins adjacents à la DEL, sans quoi elle ne peut recevoir de courant. Il est aussi à noter que les couleurs peuvent être inversées en inversant le fil. Une mauvaise couleur affichée peut donc être réglée en tournant le fil, sans devoir apporter de modifications dans le code.

Débogage
La quatrième partie concerne le débogage. Cette fonctionnalité est très importante, car plus le code devient gros, plus le débogage devient difficile. Pour faciliter le débogage, il peut être très intéressant de pouvoir observer la valeur d'une variable lors de l'exécution du programme. Pour ce faire, il faut créer des fonctions qui font appel à la communication RS232, nous permettant d'envoyer des données du robot vers le PC.

Cette fonctionnalité est implémentée par une classe *Debug*, contenant des méthodes statiques *print()*. L'utilisation des méthodes statiques nous permet d'appeler les méthodes de classe sans devoir créer un objet de classe *Debug*. Celle-ci ne contient pas d'attributs, donc créer un objet serait inutile. La fonction *print(),* qui est utilisée lors du débogage (méthode définit par *DEBUG_PRINT*) prend en paramètre un texte et une valeur sur 8 bits. Cela nous permet un affichage pratique pour savoir quelle variable est affichée à l'écran.

Comme cette fonctionnalité utilise le RS232, il est nécessaire qu'un cavalier soit présent sur *dbgEN*. Le reste de l'implémentation matériel est déjà fait sur la carte et il faut seulement que le robot soit branché à l'ordinateur par le câble USB.

Mémoire externe
La cinquième partie concerne la mémoire externe du ATmega324PA. Cette section permet d'écrire et de lire dans la mémoire externe. Cela peut être utile si nous avons besoin de plus d'espace mémoire pour stocker des données.

Le code pour cette classe et ses fonctions avait déjà été fourni et il se résumait à une classe *Memoire24CXXX* ainsi que deux fonctions : *lecture* et *ecriture*. Ces deux fonctions prennent l'adresse à laquelle il faut lire/écrire, la variable dans laquelle il faut stocker la lecture ou qu'il faut stocker dans la mémoire ainsi que la longueur du message. La mémoire externe peut servir afin de stocker des instructions de déplacement du robot par exemple.

Tout comme le RS232, toute l'implémentation matérielle est déjà faite sur la carte, il ne faut qu'ajouter un cavalier sur *memEN*.

Motricité du robot

La sixième partie concerne la motricité du robot. Comme son nom l'indique, cette section gère les mouvements du robot en contrôlant la vitesse des roues ainsi que leur direction.

La classe qui gère ceci se nomme *Motors* et elle contient les attributs *leftSpeed*, *rightSpeed* et un *enum Directions*. Elle possède également un constructeur par défaut et de nombreuses fonctions : *move, setLeftSpeed*, *setRightSpeed, setSameSpeed, setNoSpeed,* et *setPwm.* La fonction *move* permet d'ajuster la direction du robot, selon le paramètre Direction, dans quatre sens différents : *FRONT, BACK, LEFT* et *RIGHT.* Cette fonction vient changer les ports de direction des moteurs et appelle *setPwm* qui lance une minuterie grâce au *timer0* afin d'avancer à la vitesse voulue. Les fonctions *speed* permettent de changer les vitesses associées aux roues de manière indépendante ou pas et de faire arrêter le robot.

Puisqu'on utilise le *timer0,* on doit monopoliser les pins 3 et 4 du port B pour gérer le signal PWM des deux moteurs. Pour leur direction, on a choisi les pins 5 et 6. Pour les deux roues, si leur pin de direction est de 0, la roue avance et s'il est de 1, la roue recule. Les fonctions utilisées ne bloquent pas l'exécution du programme, puisqu'elles utilisent les minuteries matérielles et non logicielles.

Communication RS232
Finalement, la septième et dernière partie concerne la communication RS232. Cette fonctionnalité est très importante pour la communication entre le robot et le PC. Les méthodes de la classe *debug* entre autres, utilisent ce processus pour envoyer les données passées aux méthodes *print()* vers le PC.

La classe utilisée pour implémenter les méthodes s'appelle *Communication*

Partie 2 : Décrire les modifications apportées au Makefile de départ

Modifications apportées au Makefile pour la librairie
Plusieurs changements ont dû être apportés au Makefile pour que celui-ci crée une librairie au lieu d'un exécutable. Premièrement, nous avons pu enlever tout ce qui était en lien avec la compilation d'un fichier exécutable, car ce n'est pas ce que nous voulons générer avec ce Makefile. Nous avons aussi retiré la commande *install*, car la librairie sert seulement à la compilation de l'exécutable. Elle n'a pas besoin d'être installée sur la carte.

Par la suite, nous avons ajouté une nouvelle cible, *LIBTRG*, qui est simplement le nom de la librairie (fichier .a) à compiler. Pour s'assurer de la compilation de cette librairie, nous avons ajusté la « chaîne » de dépendances pour s'assurer que la librairie est compilée dès qu'il y a un changement dans les fichiers cpp. Les dépendances vont comme ceci (ce qui est à gauche de la flèche dépend de ce qui est à droite) :
Appel *make all* → LIBTRG → OBJDEPS → fichier.cpp

Grâce à cela, lorsque *make all* (ou *make* tout court) est appelé, le makefile va vérifier si la librairie est compilée ou a besoin d'être recompilée en raison d'une modification dans les fichiers .cpp ou .h.

Les fichiers objets(.o) sont compilés de la même manière que pour la génération de l'exécutable. Cependant, pour la compilation de la librairie, il faut plutôt utiliser la

commande avr-ar avec les options -crs plutôt que d'utiliser avr-gcc. Ces commandes ont été placées sous les variables ARCHIVE et OPTIONS pour améliorer la lisibilité.

Modifications apportées au Makefile pour l'exécutable
Une fois la librairie compilée, nous voulons évidemment être en mesure d'utiliser les fonctionnalités de la librairie lorsqu'on veut rédiger le code pour l'exécutable. Pour ce faire, nous devons inclure la librairie lorsque vient le temps de compiler notre exécutable.

Pour faire cette inclusion, il faut utiliser quelques options supplémentaires à l'appel *make all*. Il faut tout d'abord ajouter l'option -L suivie du path vers le répertoire où se trouve la librairie ainsi que l'option -l suivie du nom de notre librairie pour inclure la librairie lors de la compilation.

L'autre changement majeur est l'ajout d'une commande *debug*, qui, lors de la compilation, définit une macro *DEBUG*. Lorsque définie, cette macro permet la définition de *DEBUG_PRINT* qui est un appel de la fonction *print()*, implémentée dans la classe *Debug* de la librairie. Le mode debug permet donc d'afficher les valeurs souhaitées dans le code pour faciliter le débogage de celui-ci.

Outre ces deux modifications, le Makefile reste le même.