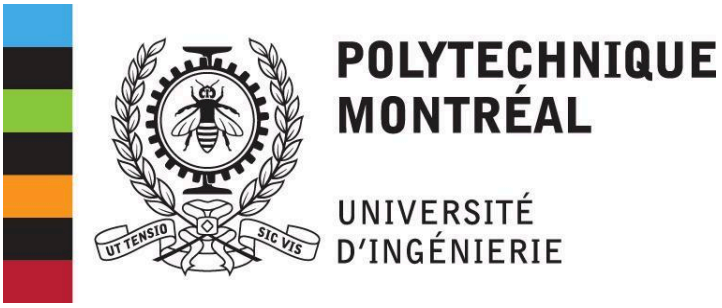

Département de génie informatique et génie logiciel

Cours INF1900:
Projet initial de système embarqué

Travaux pratiques 7 et 8

Production de librairie statique et stratégie de débogage

Par l'équipe

No 8290

Noms:

Kyle Bouchard
Edelina Alieva
Elyes Bradai
Ilyes Belhaddad

Date:
17 mars 2025

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 essaie de simplifier les interactions avec les composantes du robot le plus simple possible tout en restant flexible afin de pouvoir compléter le projet. Nous avons aussi mis de l'emphase sur réduire la possibilité d'erreur d'inattention en utilisant beaucoup de `enum class` et de l'utilisation du patron singleton.

Dans notre librairie, nous avons une classe `Board` qui est un singleton qui contient des instances vers toutes les composantes statiques du robot. La classe en tant que telle ne fait rien sauf que pouvoir garantir qu'une instance des autres classes sera instanciés. Elle agit en tant que façade à notre librairie.

Un autre classe importante est la classe `Pin`, qui se charge de gérer comment nous interagissons avec le port d'entrée et de sortie. Elle est configurée pour prendre en paramètre une `Region`, un `Direction` et un `Id`. La région spécifie dans quelle section la pin se retrouve, soit A, B, C ou D. La direction, quant à elle, sert à configurer le registre `DDR` correspondant à la région et à l'identifiant automatiquement. Ensuite, l'identifiant lui est l'identificateur entre 0 et 7 du chiffre du pin. Des fonctions sont définies pour régler ou lire la pin en format de booléen, ce qui simplifie les opérations de comparaisons. Aussi, un constructeur sans la direction est présent qui ne configuré pas automatiquement les pins. La décision d'utiliser des `enum class` dans cette classe a été prise afin de limiter des erreurs causées par spécifier des chiffres de pin littéraux. Finalement, à l'interne, les identifiants et la région sont indexés dans un tableau fixe dans le but de déterminer les bons registres à utiliser pour la `Pin`.

La classe `Adc` et `Memory24` nous a été fournie dans le cadre du projet, mais nous avons apporté des modifications à ceux-ci afin de mieux respecter le guide de codage, et utiliser des noms de fonctions correspondant au style établi dans notre librairie.

La classe `Button` est assez simple et se base autour du `Pin`. Elle a aussi été conçue pour prendre en considération l'état actif du bouton, ainsi qu'un mode événement du bouton. De plus, elle a été conçue afin de prendre en considération qui envoie des signaux haut lorsque le bouton n'est pas appuyé, via le paramètre de construction `pressedIsHigh`. Nous avons décidé d'utiliser un modèle d'événement afin que notre application puisse regarder si le bouton vient juste d'être appuyé, soit examiner s'il y a eu un front montant.

La classe `BidirectionalLed` est similaire au bouton, car elle est aussi un genre d'enveloppe autour des `Pin`s. En effet, pour sa construction, nous donnons les deux pins d'entrée, soit la négative et la positive, et nous pouvons facilement manipuler sa couleur via `Color`, un `enum class` représentant les différentes couleurs de base que la DEL peut prendre. Aussi, afin de pouvoir avoir la couleur d'ambre, nous utilisons une fonction nommée `executeAmberCycle`, qui se charge d'exécuter un changement de couleur assez rapide entre le rouge et le vert. Par contre, pour simplifier des situations ou nous

voulons allumer en ambre et bloquer l'exécution, une fonction `executeAmber` se charge. de faire cela.

La classe `Timer` est une classe générique qui peut représenter les `Timer0`, `Timer1` et `Timer2` en même temps. Ceci est possible grâce au paramètre `T` du `template` qui représente la grandeur du `Timer`, soit `uint8_t` et `uint16_t`, et `U` qui représente le "prescaler" à utiliser, qui est soit `TimerPrescalerSynchrounous` ou `TimerPrescalerAsynchronous`. Ces deux classes implémentent une interface `TimerPrescaler`, qui permet à la classe générique de pouvoir demander au "prescaler" son facteur de division et ses options spécifiques à régler dans les registres. Le `Timer` supporte aussi des différentes configurations pour représenter les différentes configurations offertes par ceux-ci. En effet, il y a une configuration en mode compteur, vu dans la structure `ConfigCounter`, et une configuration de Pwm, vu dans la structure `ConfigPwm`. Nous avons décidé d'utiliser des structures plutôt que des classes, car c'est du POD (seulement des données sans de méthodes).

La classe `Uart` est une classe autour des systèmes USART du Atmega324PA. Elle permet de configurer tous les modes de parité et bit d'arrêt, ce qui peut aider avec des différents programmes de réception USART. Aussi, le taux de transmissions (Baud rate) est configurable dynamiquement, ce qui aide aussi dans un scénario dans lequel nous n'avons pas `seriaViaUSB`. De plus, cette classe supporte le USART0 et le USART1. Aussi, nous avons créé un `FILE` virtuel pouvant représenter les entrées et sorties du USART. Ceci a comme effet de pouvoir être configurable sur `stdin` et `stdout`, ce qui rend disponible les fonctions comme `printf` et `scanf`. C'est aussi possible de manuellement recevoir ou de transmettre via les fonctions `receive` et `transmit`. Comme les autres modules, il est aussi possible d'arrêter et de commencer le module via `stop` et `start` ce qui aide dans des situations d'économie de batterie.

La classe `WatchdogTimer` permet de gérer et de configurer la minuterie de surveillance du système, qui peut aussi être utilisée comme une minuterie pour dormir. Nous utilisons ce module afin de faire des délais plus optimisés, en mettant le processeur en veille pendant quelques secondes, ce qui aide à faire une meilleure consommation de batterie.

Nous avons aussi implémenté la classe `MovementManager`, qui s'occupe de faire le mouvement en haut niveau afin de simplifier les opérations avec les moteurs. Ceci fonctionne en coordonnant les moteurs différents. Par exemple, pour tourner, on active une roue au maximum, et une roue partiellement selon un paramètre variable.

La classe `Motor`, quant à elle, s'occupe de gérer un moteur. Elle décide la vitesse à régler ainsi que s'il devrait avancer ou reculer. Un aspect intéressant de cette classe est le `offset`, qui sert de contrôle de calibration pour le moteur. En effet, nous avons déduit que les deux moteurs n'auront pas exactement les mêmes performances et demanderons donc d'être calibré à différents output pour que le robot avance en ligne droite. Nous avons donc implémenté une variable `offset` qui alignera les valeurs de sortie aux moteurs afin qu'ils aient un comportement uniforme.

La classe `Photoresistance`, s'occupe du fonctionnement d'une photorésistance. Cette classe permet de convertir un courant analogique en un courant numerique qui est ensuite traité pour en tirer une intensité. Il y a trois palier d'intensité

Enfin, nous avons deux fichiers d'en tête `common.h` et `debug.h` qui ne représentent pas de classes, mais plutôt des utilitaires de libraire. En effet, `debug.h` définit des manières d'imprimer à la console via USART0 avec des différentes sévérités et des indicateurs de code, ce qui peut nous aider lors du développement du robot afin de déboguer. Aussi, `common.h` nous permet de définir certains utilitaires de la librairie standard C++ qui ne sont pas disponibles par défaut. Ceci aide dans les méthodes génériques dans le but de pouvoir dynamiquement déterminer le type des objets à la compilation.

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.*

Notre librairie utilise un Makefile commun, appelé `common.mk` qui définit des règles par défaut qui est inclus par les Makefile des projets en tant que tel. Vu l'utilisation de la minuterie de surveillance et le manque de support de celle-ci dans SimulIDE, nous avons trois configurations de construction. Ces trois configurations sont `release`, qui est adapté pour le déploiement sur le robot sans imprimer les messages de débogage, `debug` qui est aussi adapté pour le déploiement sur le robot, mais en affichant les messages de débogage, et `simulation` qui est plutôt adaptée pour une simulation cohérente dans SimulIDE. Des règles portant sur la compilation de fichiers C et C++ vers des objets sont définis dans le fichier commun. Par contre, vu que chaque projet utilisant ce fichier aura des dépendances objets différents, il était difficile de faire une règle par défaut qui pouvait les accommoder. C'est pour ceci que nous avons définis des fonctions pouvant faire ceci.

Ensuite, dans le Makefile de la libraire, nous définissons en premier les sources, et ensuite les différentes cibles de compilations qui ne font qu'appeler les fonctions précédemment définies. Ceci aide à enlever la duplication de code. Aussi, la libraire n'a pas de configuration `debug`, parce qu'elle n'était qu'exposer une fonctionnalité de débogage, sans l'utiliser elle-même.

Finalement, le Makefile de la suite de test définit non seulement des cibles de compilation, mais aussi des cibles d'installation et de test. Les cibles de compilations ont un fonctionnement très similaire à celle de la librairie, sauf qu'au lieu d'assembler tous les fichiers objets dans une archive, l'éditeur de lien est appelé afin de créer un exécutable valide. Aussi, vu sa dépendance sur la libraire, les cibles de nettoyage ou de compilation font l'effort d'aller compiler la librairie avant de faire la compilation des tests. Pour ce qui est de l'installation, nous suivons presque la même chose que le Makefile qui nous avait

été fourni, tout en rappelant l'usager de débrancher certains ports avant de poursuivre. Pour ce qui est des cibles de test, un script est lancé qui s'occupe automatiquement de redémarrer le robot et d'écouter le USART0, afin d'accélérer le processus de débogage.

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)*