Polytechnique Montréal

Département de génie informatique et génie logiciel

Cours INF1900:
Projet initial de système embarqué

Travail pratique 7

Makefile et production de librairie statique

Par l'équipe :
No. 03-18

Noms:
Alexander Ciaciek
Antoine Déry
Stéphanie Ly
Wael Tarifi

Date:
3 novembre 2020

Dans le cadre du travail pratique 7 du cours INF1900, nous avons pour mandat de créer une librairie statique. Une librairie permet de regrouper du code dans le but de le réutiliser dans différents contextes. Le présent rapport présente les différentes classes de notre librairie ainsi que les modifications apportées au *Makefile* de départ.

Partie 1 : Description de la librairie

Classe Port(port.cpp/port.h)

Utilité

La classe `Port` permet une utilisation simplifiée des ports sans plonger dans l'ambiguïté des macros `PORTx`, `DDRx` et `PINx`. Le changement du mode de chaque broche, l'écriture en sortie et la lecture d'une broche sont appelés par des méthodes plus intuitives.

Description

Les énumérations `Tag` et `Mode`, respectivement `{A, B, C, D}` et `{IN, OUT}`, permettent de choisir le port et le mode que l'on veut instancier avec notre constructeur. Par exemple, si l'on veut instancier le port A en sortie, on appelle le constructeur de la façon suivante :

```
Port PORT_A(Port::A, Port::OUT);
```

Le deuxième constructeur permet d'instancier un port sans modifier son mode, ce qui pourrait être utile si l'on veut que certaines broches soient en sorties et d'autres en entrées :

```
Port PORT_A(Port::A);
```

Chaque constructeur appelle la méthode `initPort(const Tag& tag)`. Cette méthode se sert d'un *switch case* pour initier les variables privées `port_`, `ddr_` et `pin_` en fonction du port choisi. De plus, cette méthode est gardée en privé pour empêcher aux développeurs de modifier les variables privées par accident. En effet, il est préférable d'avoir une seule instance pour chaque port `A`, `B`, `C` et `D` dans tout notre code.

Le développeur peut choisir de mettre un port en entrée ou en sortie avec les méthodes `In()` et `Out()`. En fournissant un entier non signé de 8 bits (`uint8_t`) aux méthodes `setIn(const uint8_t& pin)` et `setOut(const uint8_t& pin)`, il est aussi aussi possible de choisir de mettre certaines broches du port en entrée ou en sortie. Il y a trois méthodes permettent la lecture, l'écriture et la remise à zéro d'un port en sortie, soit `read(const uint8_t& pin)`, `write(const uint8_t& pin)` et `clear()`, respectivement.

Classe Led(led.cpp/led.h)

Utilité

La classe `Led` permet de faciliter le contrôle d'une DEL reliée au microcontrôleur. Elle permet aussi d'allumer une certaine couleur pour un certain intervalle de temps.

Description

L'utilité du constructeur est d'initialiser la position de la DEL. Le constructeur prend trois paramètres : le port, la première broche et la deuxième broche :

```
Led maDEL(PORT_A, PORTA0, PORTA1);
```

Trois méthodes permettent le contrôle de l'affichage de la couleur. Aussi, un type énuméré a été déclaré dans la classe pour le choix des couleurs.

D'abord, la méthode `turnOn(const Color& color)` allume les couleurs rouge, verte ou ambre, en fonction de la valeur passée en paramètre. Cette méthode permet l'affichage de la couleur ambre seulement si celle-ci est appelée dans une boucle du code principal.

La méthode `timeLapse(const Color& color, const uint16_t& cycles)` permet l'affichage d'une certaine couleur en fonction d'un certain temps. La méthode se sert de la classe `Delay`, décrite ci-bas, pour effectuer un délai global en fonction du nombre de cycles passés en paramètre. Cette classe est particulièrement utile pour afficher la couleur ambre sans avoir à créer des boucles supplémentaires dans le code principal. Il est à noter que la classe dépend de la méthode en millisecondes de la classe `Delay` seulement. Il faut aussi noter que pour la couleur ambre, le nombre de cycles est divisé par deux, puisqu'autrement, le délai est deux fois plus grand que les couleurs rouges et vertes pour le même nombre de cycles. Par exemple, pour allumer une DEL rouge pendant une seconde, l'instruction sera :

```
maDEL.timeLapse(LED::RED, 100);
```

Finalement, la méthode `turnOff()` remet les sorties à zéro et éteint la DEL.

Classe Delay(delay.cpp/delay.h)

Utilité

Les fonctions de délais de la librairie `util/delay.h` requiert le passage d'une constante en paramètre. Or, cela peut être embêtant dans les situations où les délais doivent être variables, d'où l'utilité de cette classe.

Description

Des boucles ont été créées dans les méthodes `delayMS(const uint16_t &cycles)` et `delayUS(const uint16_t &cycles)` pour itérer sur des délais de 10 ms et 10 µs, respectivement. Le paramètre fourni est le nombre d'itérations désirées. Pour afficher une couleur sur un intervalle de 1000 ms (une seconde), il faut itérer 100 fois. Il est à noter que les intervalles de temps possibles sont respectivement de 10 ms à 655 350 ms et 10µs à 655 350 µs.

Classe Button(button.cpp/button.h)

Utilité

La classe `Button` permet la lecture d'un bouton par scrutation lorsqu'il est seulement appuyé ou appuyé et relâché.

Description

Le constructeur initie le bouton au port désiré, tel que :

```
Button BUTTON(PORT_A);
```

Deux méthodes permettent la lecture de la broche par scrutation. Les méthodes `isPressed(const uint8_t& pin)` et `isClicked(const uint8_t& pin)` vont respectivement vérifier si le bouton est maintenu ou relâché. On fournit la broche à lire en paramètre.

Classe motorPWM (motorPWM.cpp/motorPWM.h)

Utilité

La classe `motorPWM` permet de configurer facilement les deux moteurs à l'aide de signaux PWM phase correcte, générés par la minuterie 0.

Description

Le constructeur prend quatre paramètres : le pourcentage PWM du moteur gauche, celui du moteur droit ainsi que la direction de chacun des deux moteurs. Dans cette classe, l'énumération `Direction` permet de configurer le sens de rotation des moteurs de manière intuitive grâce aux mots clés *Backward* et *Forward*. À titre d'exemple, pour générer un signal PWM où le moteur gauche est à 75%, le moteur droit est à 100% et où les deux moteurs vont vers l'avant, la déclaration sera :

```
motorPWM pwm(75, 100, FORWARD, FORWARD);
```

Les méthodes `invertDirectionLeft()` et `invertDirectionRight()`la direction de chacun des deux moteurs de manière séparée. La méthode `setPercent(const uint8_t &percentLeft, const uint8_t &percentRight)` permet de mettre à jour le pourcentage PWM des deux moteurs. Finalement, la méthode `stopMotor()` permet d'arrêter les deux moteurs.

Pour cette classe, il est important de noter que les moteurs sont connectés aux ports reliés à la minuterie 0, c'est-à-dire que le moteur gauche est connecté sur le port `B3` et le moteur droit est connecté sur le port `B4`. Pour ce qui est de la direction, la direction du moteur gauche est connectée au port `B2` et la direction du moteur droit est connectée au port `B5`.

Classe USART (usart.cpp/usart.h)

Utilité

La classe `USART` permet de configurer le USART0 du microcontrôleur, qui permet de transmettre des données du microcontrôleur vers le terminal de SimulIDE. Cette classe sera particulièrement utile pour l'utilisation d'un débogueur.

Description

Le constructeur permet d'initialiser les registres de USART0. Il configure donc le nombre de bauds à 2400 bps, active la réception ainsi que la transmission par le USART0 et indique le format des données (8 bits, 1 stop bit, sans parité). La seule méthode de cette classe est `transmit(uint8_t data)`. Elle permet de transmettre un octet de donnée du microcontrôleur vers le terminal SimulIDE via le USART0. Il s'agit de la même fonction utilisée dans le travail pratique 5.

Classe Timer (timer.cpp/timer.h)

Utilité

La classe `Timer` permet de configurer facilement une minuterie sur 16 bits grâce à la minuterie 1.

Description

Outre le constructeur par défaut, qui permet de créer un objet `Timer`, cette classe a trois méthodes. D'abord, la méthode `startTimer(uint16_t duration)` permet de partir la minuterie en configurant les registres nécessaires. Le temps voulu, en millisecondes, doit être passé en paramètre. Lorsque le temps désiré est atteint, une interruption est lancée et doit être prise en charge avec la macro `ISR` appropriée. Par la suite, la méthode `setDuration(const uint16_t& duration)` permet de modifier la durée de la minuterie sans avoir à la reconfigurer. Finalement, la méthode `stopTimer()` permet d'arrêter la minuterie. Donc pour démarrer la minuterie `monTimer` de 2 secondes, il suffit de déclarer :

```
monTimer.startTimer(2000);
```

Classe Can (can.cpp/can.h)

Utilité

La classe `Can` permet d'accéder au convertisseur analogique/numérique du microcontrôleur. Cette classe a été écrite par Jérôme Collin et Matthew Khouzam et nous a été fournie pour le travail pratique 6.

Description

Le constructeur par défaut configure les différents registres nécessaires au bon fonctionnement du convertisseur. Outre le destructeur, qui arrête le convertisseur, une seule méthode est présente dans cette classe, soit `lecture (uint8_t pos)`. Cette méthode

prend en paramètre un octet, qui représente la position sur le `Port A`. La valeur retourne un entier non signé sur 16 bits où les 10 bits les moins significatifs sont les données importantes. Les 8 bits les moins significatifs sont normalement conservés donc il faut faire un décalage de 2 vers la droite. Par exemple, pour utiliser le convertisseur et stocker la valeur de retour dans `resultat`, il faut déclarer :

```
Uint8_t resultat = (convertisseur.lecture(0x00) >> 2);
```

Partie 2 : Description des modifications apportées au Makefile de départ

Avant tout, notre bibliothèque statique est une archive regroupant des classes C++. Lors de la compilation du code d'un projet, les classes utilisées de la librairie seront copiées et transférées vers le programme avant de les exécuter. En prenant en considération que la librairie statique n'est pas exécutable, nous avons créé deux répertoires. D'abord le répertoire `lib_dir` contient tous les fichiers sources des classes C++ alors que `exec_dir` contient le programme principal et fait la liaison des fichiers sources. Chaque répertoire comporte un fichier *Makefile* dont certaines modifications ont été effectuées par rapport à celui écrit par Simon Barrette et Jérôme Collin. Les changements faits dans chacun des *Makefiles* sont expliqués ci-dessous.

2.1 Répertoire lib_dir

Le *Makefile* du répertoire permet de créer les fichiers objets `.o` à partir des fichiers `.cpp` et `.h` du même nom. Par la suite, les fichiers `.o` produits sont liés ensemble pour créer la librairie `libstat.a`. Donc, les commandes permettant de créer des fichiers .elf et .hex peuvent être supprimés puisqu'elles ne sont pas utiles dans ce *Makefile*. Donc, les modifications suivantes ont été apportées :

- Ligne 26 : Ajout d'un nom de librairie : `26 PROJECTNAME=libstat`
- Ligne 32 : Ajout d'un *wildcard* pour récupérer et compiler tous les fichiers `.cpp` du répertoire `lib_dir` sans avoir à les expliciter un à un : `32 PRJSRC= $(wildcard *.cpp)`
- Ligne 65 : Ajout de la variable `AR`, qui contient l'archiveur AVR-AR. L'archiveur permet de créer une librairie statique : `65 AR=avr-ar`
- Ligne 73 : Suppression de la variable `HEXFORMAT = ihex` puisqu'elle n'est plus utilisée en raison des suppressions des lignes 152 à 161 (voir page suivante).
- Ligne 94 : Remplacement de l'extension `.elf` de la variable `TRG` pour l'extension `.a`, pour que la librairie ait la bonne extension : `96 TRG=$(PROJECTNAME).a`
- Lignes 95 et 96 : Suppression des variables `HEXROMTRG = $(PROJECTNAME).hex` et `HEXTRG=$(HEXROMTRG) $(PROJECTNAME).ee.hex` puisqu'elles ne sont plus utilisées en raison des suppressions des lignes 152 à 161.

- Lignes 134 et 135 : Remplacement du compilateur par l'archiveur AVR-AR et les options `crs`, tel qu'indiqué dans la documention d'AVR-libc. Ces trois commandes permettent de créer l'archive si elle n'existe pas ou de la même à jour. De plus, une commande `echo` a été ajoutée dans le but d'afficher un message dans le terminal lorsque la librairie est créée avec succès :

```
133 $(TRG): $(OBJDEPS)
134     $(AR) -crs $(TRG) $(OBJDEPS)
135     @echo Creation de la librairie reussie
```

- Lignes 151 à 160 : Suppression des commandes `%.hex` : `%.elf` puisqu'aucun fichier ayant ces extensions n'est utilisée lors de la création de l'archive. De plus, la commande `install` a également été enlevée puisqu'elle n'est pas utile pour la création d'une librairie.

2.2 Répertoire exe_dir

Le *Makefile* du répertoire permet de compiler le fichier `main.cpp` et de créer un fichier exécutable à partir de la libraire `libstat.a`. Pour y parvenir, les modifications suivantes ont été apportées :

- Ligne 26 : Changement du nom du fichier exécutable : `26 PROJECTNAME= myExecutable`
- Ligne 35 : Ajout du chemin vers le répertoire `lib_dir` pour permettre au compilateur de trouver les fichiers d'en-tête inclus au début du programme `main.cpp`. Le « `I` » signifie simplement « include » : `35 INC= -I ../lib_dir`
- Ligne 37 : Ajout de la librairie à lier. La commande « `l` », suivie du mot `stat` permet de retrouver la librairie `libstat` dans le répertoire spécifié après le « `-L` » et de l'utiliser pour la compilation et la création de l'exécutable : `37 LIBS= -l stat -L ../lib_dir`