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
n°4546

Noms:
Catalina Andrea Araya Figueroa
Laurent Bourgon
Mehdi Benouhoud
Ihsane Majdoubi

Date:
13 mars 2023

Partie 1 : Description de la librairie

`lib1900` a été créée en suivant quelques principes. Tout d'abord, la visibilité de l'implémentation est minimisée le plus possible. Ensuite, chaque classe de la librairie a une seule responsabilité. Subséquemment, aucune classe de la librairie ne dépend d'une autre. Ainsi, la cohésion élevée et l'absence de couplage présent dans la librairie offrent une puissante flexibilité à l'utilisateur.

Une attention particulière a été mise sur la création d'une interface simple et intuitive : l'utilisateur de la librairie ne donne que le « quoi », puis la librairie s'occupe du « comment ».

La librairie est divisée en deux parties : les espaces de noms, comportant les fonctions utilitaires, et les classes, comportant les composants du robot.

Espaces de noms

La programmation de systèmes embarqués AVR pose un certain problème au niveau de la sémantique. AVR Libc utilise plusieurs abréviations et manipulations au niveau des bits qui peuvent nuire à la lisibilité du code. Les espaces de noms regroupent des fonctions utilitaires de la librairie.

Les espaces de noms sont *header only* puisqu'ils utilisent des fonctions `static inline`. Cela permet d'avoir des performances similaires aux macros tout en forçant la vérification des types. Lorsque le code est optimisé, des outils comme Compiler Explorer confirment le remplacement par le compilateur de l'appel de la fonction `static inline` par son contenu.

lib/interruptButton.cpp

```
...
void InterruptButton::start()
{
    // (p.68) External Interrupt Mask: Interrupt 0
    io::setActive(&EIMSK, INT0);
}
...
```

→

Compiler Explorer (AVR gcc 11.1.0)

```
_ZN15InterruptButton5startEv:
  sbi 0x1d,0
  ret
```

`io`

Manipule les entrées et sorties.

Tout programme avec AVR Libc, incluant notre librairie, utilise les macros correspondant à la documentation d'Atmel comme `DDRB`, `TCCR1B` ou `WGM00`. Pour faciliter la compréhension des paramètres d'entrée/sortie dans notre librairie, des alias ont été créés :

Alias	Exemple
Register	`TCCR1A, OCR1B, UCSR0C...`
DataDirectionRegister	`DDRA, DDRB, DDRC...`
Port	`PORTA, PORTB, PORTC...`
Pin	`PINA, PINB, PINC...`
Position	`PA0, ISC00, WGM11...`

L'espace de nom `io` possède aussi des fonctions qui nomment l'intention derrière les différentes manipulations de bits. Chaque fonction ne prend qu'un seul registre et une seule position : cela met l'accent sur chacun des bits modifiés, en plus de facilité la gestion avec `git`.

`interrupts`

Active et désactive la capture des interruptions pour contrôler l'exécution des routines d'interruption.

Ces méthodes ne sont jamais appelées par les classes de la librairie : elles doivent toujours être appelées par l'utilisateur.

debug

Communique par RS232 lorsque `DEBUG` est défini.

Lorsque `DEBUG` n'est pas défini, le corps des fonctions est vide. Compiler Explorer permet de vérifier que le compilateur n'ajoute aucune instruction supplémentaire.

(+) **`void initialize()`** : ajuste les registres pour activer la transmission des données. Envoie « (`DEBUG enabled`) » une fois l'initialisation finie. Doit être appelée au début du programme.

En plus d'offrir la même interface d'envoi que `Communication`, `debug` permet aussi d'envoyer une valeur avec une étiquette :

(+) **`void send(const char* label, const uint16_t data)`** : envoie sous le format « label: data ». Bloque le programme pour 4,17 ms/caractère.

Repose sur la classe `Communication`, voir sa documentation pour plus d'information.

Classes

Les classes représentent des composantes concrètes du robot. Certaines classes sont entièrement composées de méthodes statiques. Il a été jugé raisonnable de les garder en classe plutôt que de les convertir en espace de noms puisqu'elles représentent une entité du robot.

Led

Contrôle une DEL bicolore.

(+) **`Led(io::DataDirectionRegister dataDirectionRegister, io::Port port, const io::Position plus, const io::Position minus)`** : sauvegarde les attributs et configure les registres de direction.

(+) **`void setColor(const Color &color)`** : ajuste la polarité de la DEL selon le `enum Led::Color`.

(+) **`void setAmberForMs(const uint16_t durationMs)`** : alterne la DEL entre vert et rouge pour créer la couleur ambrée. Bloque l'exécution normale pour le nombre de millisecondes passé.

Wheels

Contrôle les roues du robot à l'aide d'un signal PWM généré avec le Timer2 de 8 bits.

Timer2 a été choisi parce qu'il libère le Timer1 pour la minuterie (`InterruptTimer`). Puis, Timer0 est plus compliqué à utiliser, car il utilise des broches dans B4:7, prises par le ATmega8 lors du *flash*.

Gauche		**Droite**	
Enable	*Direction*	*Enable*	*Direction*
PD6	PD4	PD7	PD5

Cette classe est entièrement statique parce qu'elle utilise toujours le Timer2.

Toutes les méthodes publiques de la classe peuvent prendre en paramètre optionnel spécifiant un côté de roue. S'il n'est pas spécifié, l'opération sera appliquée aux deux roues.

(+) **`void initialize(const Side &side = Side::BOTH)`** : doit être appelée au début du programme. Configure les registres de direction pour les côtés initialisés.

(+) **`void setDirection(const Direction &direction, const Side &side = Side::BOTH`**) : ajuste la broche *Direction* spécifiée.

(+) **`void setSpeed(const uint8_t speed, const Side &side = Side::BOTH)`** : ajuste la sortie PWM de la broche *Enable* spécifée. Le paramètre **`speed`** est un pourcentage (0 à 100). Passer une valeur en dehors de ces bornes a des effets indésirables.

(+) **`void turnOff(const Side &side = Side::BOTH)`** : arrête les roues des côtés spécifiés.

InterruptButton

Gère un bouton-poussoir sur D2 en déclenchant l'interruption INT0 selon le mode spécifié.

Cette classe est entièrement statique.

Fonctionne autant avec le bouton de la carte mère (avec le cavalier IntEn) qu'avec le bouton-poussoir externe sur le *breadboard*.

Ces méthodes configurent les registres :

(+) void initialize(const Mode &mode = Mode::ANY) : doit être appelée au début du programme. Mode est un enum class qui correspond aux conditions générant l'interruption : front montant, front descendant, ou les deux.

(+) void setMode(const Mode &mode) : choisi un mode après l'initialisation.

Ces méthodes gèrent les interruptions :

(+) void start() : active la génération d'interruption par INT0. Est appelée lors de l'initialisation.

(+) void stop() : désactive la génération d'interruption par INT0.

(+) void clear() : enlève les interruptions par INT0 en attente.

(+) static void waitForDebounce() : Utilisé comme antirebond dans la routine d'interruption. Attends la stabilisation du signal en bloquant l'exécution de la routine (temps déterminé par la constante DEBOUNCE_DELAY_MS).

Une routine d'interruption avec la macro InterruptButton_vect est à utiliser comme ceci :

main.cpp

```
#include <lib/interruptButton.hpp>
#include <lib/interrupts.hpp>

ISR(InterruptButton_vect)
{
    InterruptButton::waitForDebounce();
    ...
    InterruptButton::clear();
}

int main()
{
    ...
    InterruptButton::initialize();
    interrupts::startCatching();
    ...
}
```

InterruptTimer

Gère une minuterie par interruption sur le Timer1 de 16 bits.

Cette classe est entièrement statique.

Ces méthodes configurent les registres :

(+) `void initialize(const Mode &mode, const double delayS)` : doit être appelée au début du programme. `Mode` est un `enum class` qui modifie le comportement de la minuterie. La minuterie générera une interruption après `delayS` secondes.

(+) `void setMode(const Mode &mode)` : pour changer le mode après l'initialisation.

(+) `void setSeconds(const double delayS)` : pour changer la durée après l'initialisation.

Ces méthodes gèrent les interruptions :

(+) `void reset()` : réinitialise le compteur à 0.

(+) `void start()` : active la génération d'interruption et réinitialise le compteur à 0. Doit être appelée après l'initialisation.

(+) `void stop()` : désactive la génération d'interruption.

Une routine d'interruption avec la macro `InterruptTimer_vect` est à utiliser comme ceci :

main.cpp

```
#include <lib/interruptTimer.hpp>
#include <lib/interrupts.hpp>

ISR(InterruptTimer_vect)
{
    ...
}

int main()
{
    ...
    InterruptTimer::initialize(InterruptTimer::Mode::CLEAR_ON_COMPARE, 0.5);
    InterruptTimer::start();
    interrupts::startCatching();
    ...
}
```

Il est important de noter que `InterruptTimer` choisit le plus petit *prescaler* du Timer1 qui supporte la durée configurée. Ainsi, la résolution de la minuterie n'est pas la même selon sa durée. La table suivante associe la durée maximale et la résolution de chacun des *prescaler* du Timer1 :

Prescaler	Durée maximale (s)	Résolution (µs)
CLK	0,01	0,2
CLK8	0,07	1,1
CLK64	0,52	7,9
CLK256	2,09	31,9
CLK1024	8,39	128,0

Communication

Gère la communication par RS232 avec l'USART0.

Cette classe est entièrement statique.

Configurée à 2400 bauds avec des trames de 8 bits, 1 stop bit et aucun bit de parité. Ainsi, si un caractère prend 10 bits (1 start bit + 8 bits + 1 stop bit), cela prend environ 4,17 ms par caractère.

Attention, lorsque `Communication` est initialisée, les broches `D0` et `D1` ne doivent pas être utilisées.

(+) `void initialize()` : ajuste les registres pour activer la transmission des données. Doit être appelée au début du programme.

(+) `void send(const char* data)` : envoie par RS232 une chaine de caractères. Bloque le programme pour 4,17 ms/caractère.

(+) `void send(const uint16_t data)` : envoie par RS232 un entier non signé de 16 bits. Puisque la valeur est convertie en chaîne de caractère, entre 1 et 5 caractères sont envoyés. De cette manière, le programme est bloqué entre 4,17 ms et 20,85 ms.

(-) `void sendCharacter(const uint8_t data)` : les méthodes publiques appellent cette méthode pour envoyer un seul caractère par RS232. Bloque le programme pour 4,17 ms.

AnalogReader

Lit une valeur analogue sur `PORTA` à l'aide du convertisseur analogue numérique.

(+) `uint8_t read(io::Position position)` : Lit la valeur analogique sur la broche spécifiée. Avant son appel, le registre de direction de la broche lue doit être en entrée. De ce fait, l'utilisation de **`io::setInput(DataDirectionRegister port, const Position bit)`** au début du programme est recommandée. Bloque le programme pour 104 ms.

Cette classe est une modification de la classe `can` (2006, Matthew Khouzam & Jérôme Collin). En plus d'uniformiser ses noms avec le reste de la librairie, la fonction `read` a été modifiée pour effectuer un décalage de 2 bits vers la droite avant de retourner une valeur sur 8 bits.

Memory

Accède par I^2C à la mémoire externe en gérant les registres nécessaires.

Cette classe enveloppe les fonctions de `Memoire24CXXX` (2005, Matthew Khouzam & Jérôme Collin) pour uniformiser ses noms avec le reste de la librairie et pour faciliter son utilisation.

Attention, lorsqu'une instance `Memory` est en vie, les broches `C0` et `C1` ne doivent pas être utilisées.

Ces méthodes concernent une seule valeur :

(+) `void write(const uint16_t address, const uint8_t data),`
(+) `void writeMessage(const uint16_t startAddress, const char* message)` : écrit la valeur passée en paramètre. La fonction est bloquante 5 ms pour que le cycle d'écriture soit complété.

(+) `const uint8_t read(const uint16_t address) const` : lit un caractère unique à l'adresse passée.

Ces méthodes concernent une chaîne de caractères :

(+) `const char* readMessage(const uint16_t startAddress,const uint8_t messageSize) const` : lit une chaîne de caractères et retourne un pointeur vers celle-ci. Le pointeur est vers le tampon privé `readMessageBuffer_` de taille `N_MAX_CHARACTERS`. Lorsque la méthode est appelée une seconde fois, le nouveau message est écrit par-dessus. Ainsi, il faut utiliser la chaîne de caractères tout de suite ou la copier (p. ex., avec `strcpy`) si l'on souhaite lire plusieurs chaînes.

(+) `void clearBuffer()` : remets à zéro le tampon `readMessageBuffer_`.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Règles communes

Les fichiers de configuration de GNU Make ont été modifiés pour réduire la duplication. Les règles communes ont été mises à la racine du projet.

`Makefile.common`

Règles globales pour tous les projets C++ avec l'ATmega324PA.

- `make clean` : supprime les fichiers créés lors de la compilation. La variable `ARTIFACTS` doit être définie avant d'inclure `Makefile.common`.
- `make debug` : ajoute la définition `DEBUG` aux options de compilation. Il ne reste plus qu'à spécifier la règle à exécuter en plus. Par exemple, pour exécuter la règle `all` :

lib/Makefile

```
...
ARTIFACTS=$(LIB) $(OBJECTS) $(OBJECTS:.o=.d)
...
all: ...
...
include ../../../Makefile.common
debug: all
...
```

`Makefile.exec.common`

Règles pour les exécutables sous l'ATmega324PA.

En plus d'utiliser les règles définies dans `Makefile.common`, définit les règles suivantes :

- `make all` : compile le projet.
- `make install` : compile et installe le projet.
- `make debug` : compile et installe le projet avec `DEBUG` défini.

Cependant, `PROJECTNAME`, `PRJSRC` (et, optionnellement, `INC`, `LIBS`) doivent être défini avant d'inclure `Makefile.exec.common`.

Librairie statique

Le `Makefile` de la librairie statique `lib1900` a été modifié pour archiver les objets avec `avr-ar`.

Tous les fichiers situés dans le dossier `lib/` sont compilés et ajoutés dans la librairie.

Un fichier `lib1900.a` sera généré dans le dossier `lib/` grâce aux flags `c` (créer), `r` (remplace les fichiers), `s` (crée un index).

Exécutable

Le programme `exec` permet de tester tous les composants de lib1900. `Makefile.exec.common` (et, par extension, `Makefile.common`) est utilisé pour compiler, installer et *debug* le programme.

Pour forcer la recompilation du programme lorsque la librairie est modifiée, une dépendance est ajoutée à l'archive de la librairie.

exec/Makefile

```
...
$(OBJDEPS): $(LIBDIR)lib$(LIBNAME).a
...
```