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 31-35

Noms:

Mohamed Bassiouni, Motassembellah
Tadrous, Astir
Doumon, Elena
Ramanitranja, Raphael

Date:
31 octobre 2022

Partie 1 : Description de la librairie

Classe **Communication** :

Pour la classe communication, nous avons créé un fichier .h (communication.h) qui permet de déclarer le constructeur (Communication()), mais aussi de déclarer deux fonctions ayant le même nom (transfertUart), mais possédant des paramètres différents. La première reçoit en paramètre uniquement une donnée (char de taille 1) en entier sur 8 bits en paramètre, la deuxième reçoit un tableau composé de char dont la taille est inférieure à 255, à condition de spécifier cette taille. Cette technique se nomme en anglais "function overload". C'est vraiment dans le fichier .cpp (communication.cpp) que l'on a implémenté les méthodes et le constructeur déclarés dans communication.h.

Ainsi, pour le constructeur, nous avons commencé par initialiser l'UART. Pour ce faire on met la valeur du premier registre UBRR0H à 0. Pour ce qui est du deuxième registre, soit UBRR0L, sa valeur est fixée à 0xCF. Ensuite, on décide d'activer le récepteur et le transmetteur de USART sur UCSR0B. On met alors le bit TXENn à 1 pour activer le transmetteur et le bit RXENn à 1 pour activer le récepteur. Pour le UCSR0C, on précise uniquement le nombre de bits qu'auront les données, soit 8 bits en mettant à 1 les bits UCSZ01 et UCSZ00.

Pour transférer un char, il est possible d'utiliser la fonction fournie dans la documentation du Atmega324 et qu'on retrouve dans l'implémentation de la fonction transfertUart (uint8_t data). On y trouve une fonction bloquante, c'est le cas du while qui empêche l'exécution du code UDR0 = data jusqu'à ce que la condition ne soit pas satisfaite, c'est à ce moment qu'on sort de la boucle. Finalement, la deuxième fonction de transfertUart, qui prend deux paramètres, va itérer sur le tableau de char et transférer chaque caractère du tableau grâce à la première fonction.

Dans cette optique, on fixe l'adresse de départ à 0 vu qu'on cherche le premier caractère du tableau et on incrémente l'adresse pour nous pouvoir transférer tous les caractères du tableau. Cette boucle s'arrête uniquement lorsqu'on a transféré tous les caractères du tableau, ce qui évite de continuer à faire cette instruction pour rien et donc utiliser beaucoup de temps. D'ailleurs, on ne peut transférer un tableau de char de taille supérieur à 255 parce qu'on a juste 8 bits, voilà pourquoi on rajoute cette condition au départ.

Classe **Led :**

Dans le fichier led.h, on a un constructeur qui prend en paramètre pinCathode sur 8 bits, ce qui correspond à la broche du port B qui est lié à la cathode de la Del. Il prend également pinAnode comme argument sur 8 bits pour spécifier la broche liée à l'anode de la Del. Nous avons aussi déclaré quatre fonctions n'ayant pas de valeur de retour, ni de paramètre, mais qui ont comme objectif de donner des commandes à la del bicolore. La del est donc configurable sur le port B uniquement et dans notre cas nous utilisons le port B0 et le port B1.

Par exemple, setLedRed(), setLedGreen(), setLedAmber(), setLedOff(), permettent respectivement d'allumer la couleur rouge, allumer la couleur verte, allumer la couleur ambrée et fermer la Del. Ces constructeurs et méthodes sont considérés comme public. Pour les attributs

privés, on a initialisé le nombre d'itérations à 2000 fois pour créer la couleur ambrée. On a également rajouté les attributs delRed et delGreen.

Dans le fichier led.cpp, on a un constructeur qui va permettre de mettre le bit de la broche du port B qui est lié à la cathode de la Del à HIGH pour initialiser l'attribut delRed, représentant la commande pour envoyer la couleur rouge. Il va également faire la même chose avec celle qui est lié à l'anode en initialisant l'attribut delGreen pour représenter la commande pour envoyer la couleur verte. Nous avons décidé d'utiliser uniquement les broches PB0 et PB1 en sortie et donc on a initialisé le DDRB en fonction de ceci.

Par la suite les fonctions setLedRed() et setLedGreen() vont initialiser le PORTB en fonction des valeurs fixé par le constructeur pour les attributs delRed et delGreen. Pour la fonction setLedOff(), on inverse delRed et delGreen pour mettre les bits des broches PB0 et PB1 à 0 et on initialise le PORTB en conséquence.

Finalement, on utilise les fonction setLedGreen() et setLedRed() dans la fonction setLedAmber() pour créer la couleur ambré en alternant entre la couleur verte et la couleur couleur rouge. Pour ce faire on met un délai pour la couleur verte inférieur au délai de la couleur rouge et on répète ceci 2000 fois dans une boucle for pour que le changement soit visible à l'œil nu.

Classe **Bouton :**

La classe Button comprend un fichier d'entête button.h où on instancie les différentes méthodes et d'un fichier source button.cpp où on les définit. Button possède parmi ses attributs publics un enum class de type typeEdge qui a pour but de nous permettre de choisir quelle mode d'interruption nous souhaitons utiliser. Celle-ci est composée de trois éléments soit ANY correspondant au lancement d'une interruption, peu importe si le bouton est appuyé ou relâché, FALLING qui déclenche l'interruption lorsque le bouton est relâché et RISING qui déclenche l'interruption lorsque le bouton est appuyé.

Button possède aussi un constructeur Button() qui prend en paramètre un entier volatile de 8 bits appelés typeButton pour être utilisée globalement dans le code et cette variable permet de définir quel bouton sera pris en compte par l'interruption soit INT0 ou INT1. Les instructions d'AVR. n'ont pas de variables permettant de généraliser les ports or, nous pouvons les appeler en utilisant des uint8_t ou en pointant à un registre I/O pour ensuite lire et écrire via ce pointeur. Dans cette optique, il est possible d'accéder à n'importe quel registre I/O en la référençant à l'emplacement de la mémoire. Il est aussi important que l'on mette ces variables en type volatile, car on ne veut pas que le programme l'optimise. En effet, il arrive que l'optimisation du compilateur puisse être indésirable, car il modifie des valeurs afin d'exécuter le code plus rapidement. Dans notre cas, nous voulons nous assurer que la variable externe, soit la position de la broche quelconque, affecte le code correctement d'où l'usage du volatile pour nos pointeurs. Le deuxième élément passé en paramètre dans le constructeur est un objet de type typeEdge de l'enum class. Le constructeur établi d'abord que le bouton n'a pas été appuyé avec la variable booléenne volatile gButtonPressed.

Lorsqu'on construit un bouton, on active respectivement la broche liée au INTn tel que INT0 active le port D2. Ensuite, on met le bit INT0 à 1 dans le registre EIMSK ce qui signifie que c'est celui-

ci qui est sélectionné pour l'interruption. Une variable qui dicte l'état initial de typeEdge est créé et on s'en sert dans un switch-case. Le premier cas de la machine est FALLING qui ajuste le bit ISC01 à 1 dans le registre EICRA. La fonction sei() est appelée par la suite permettant de débloquer les interruptions. Dans le deuxième cas soit RISING, ce seront les bits ISC01 et ISC00 qui seront ajustés à 1 et sei() est encore appelée. Pour le troisième cas soit ANY, seulement le bit ISC00 sera mis à 1 dans EICRA suivi par sei().

Par contre, si le bouton choisi est INT1, on saute la condition précédente pour entrer dans le else. Ici, ce sera le bit INT1 qui sera ajusté à 1 dans EIMSK. Ensuite une seconde machine à états est créée identique à la précédente sauf que le bit ISC01 et ISCOO deviennent ISC11 et ISC10 respectivement. La dernière méthode dans la classe set la fonction debounce() qui permet de vérifier si un bouton est bel et bien appuyé. Une première variable booléenne FIRST_LECTURE prend la valeur du résultat entre PIND et le masque imposé sur PIND2. Après un délai de 30 millisecondes, on refait la même opération dans la variable booléenne SECOND_LECTURE. Si les deux variables booléennes sont par la suite équivalentes, on retourne true (1) ce qui signifie que le bouton est correctement appuyé.

Classe **Motor :**

La classe Motor contient deux attributs de type enum classe (soit timer_ et direction_). La première classe enum est la classe TIMER qui contient deux choix de minuterie soit TIMER0 et TIMER1. La deuxième classe enum est la classe Direction qui contient le choix de direction du robot soit la direction REVERSE ou la direction ADVANCE. Bref, les 2 attributs de la classe correspondent à la direction du robot ainsi que la minuterie utilisée pour générer le PWM.

Le constructeur de la classe prend en paramètre une minuterie de type enum classe Timer. La minuterie rentrée sera affectée à l'attribut minuterie (timer_) de la classe moteur. Si la minuterie choisie est la TIMER0, le constructeur ajustera les registres WGM00, COM0A1 et COM0B1 à 1. Ajuster le registre WGM00 permet de générer du PWM en mode phase correcte sur les broches B3 et B4. Les 2 autres registres permettent de remettre à 0 les pins de comparaison OCR0A et OCR0B lors de leur égalité avec le registre TCNT0. Toutefois, si la minuterie 2 est passée en paramètre, le constructeur va ajuster les registres WGM20, COM2A1 et COM2B1 à 1. WGM20 permet de générer du PWM sur les broches D6 et D7 en mode phase correcte. Les 2 autres registres permettent de remettre à 0 les pins de comparaison OCR2A et OCR2B lors de leur égalité avec le registre TCNT2. De plus, dans le constructeur, on ajuste nos ports (DDRD ou DDRB) de sorte qu'ils produisent un signal de sortie (ils sont mis à 1). De plus, on met le registre CS01 ou CS21 à 1 afin de diviser la fréquence initiale de 8MHZ du microcontrôleur par 8. Choisir quels registres ajuster se fait à l'aide d'un switch case.

La fonction *adjustDirection* prend en paramètre une direction de type enum classe Direction. Si la direction passée en paramètre est ADVANCE, l'attribut direction_ sera définie à ADVANCE et selon la minuterie définie lors de la construction de la classe, les broches D4 et D5 (pour la minuterie 2) ou B5 et B6 (pour la minuterie 0) seront forcées à 0. Si la direction rentrée en paramètre est plutôt REVERSE, les broches ci-haut seront forcés à 1. La fonction *ajustPWMPourcentage* prend en paramètre 2 entiers ne dépassant pas 255. Selon la minuterie choisie lors de la construction de la classe, ces 2 entiers seront assignés à OCR0A et OCR0B

(minuterie 0) ou à OCR2A et OCR2B (minuterie 2). Ces valeurs indiquent la période lors de laquelle le PWM est à high.

Classe **Navigation :**

La classe contient un seul attribut soit motor_ qui est de type classe Motor. Le constructeur prend en paramètre une minuterie de type Motor ::Timer. Il appelle le constructeur de la classe Motor avec la minuterie passée en paramètre et l'assigne à son propre attribut motor_.

La fonction *setDirectionAdvance* apelle la fonction *adjustDirection* de motor_ avec un paramètre de type Motor::Direction::ADVANCE. Ceci permet au robot d'avancer. La fonction ne retourne rien. La fonction *setDirectionReverse* appelle la fonction *adjustDirection* de motor_ avec un paramètre de type Motor::Direction::REVERSE. Ceci permet au robot de reculer. La fonction ne retourne rien.

Les fonctions *turnRightSlight* et *turnLeftSlight, turnRightHard, turnLeftHard, turnRight* et *turnLeft,goStraightSlow*, *goStraight* et *goStraightFast* ne prennent rien en paramètre, mais appellent tous la fonction *ajustPWMPourcentage* de motor_ avec deux paramètres de type uint8_t déjà établis (ces paramètres ne seront pas rentrés par l'utilisateur). Les valeurs que ces deux paramètres peuvent avoir sont de 0 à 255.

Classe **Mémoire:**

La classe memoire_24 fournie par les chargés permet l'accès à la mémoire du robot. Cette classe a deux attributs qui sont les dimensions de la page ainsi que l'adresse à laquelle on souhaite écrire les données.

Le constructeur de Memoire24CXXX() appelle la méthode void init() qui elle initialise l'horloge de l'interface I2C et appelle la fonction uint8_t choisir_banc(const uint8_t banc) qui est initialisée à zéro. Ainsi le constructeur de Memoire24CXXX() initialise l'horloge de l'interface I2C ainsi que le port série grâce à la fonction choisir_banc initialiser à 0.

Les méthodes implémentées commencent avec uint8_t lecture(const uint16_t adresse, uint8_t *donnée) : la méthode permet la lecture d'une seule donnée à la fois et pour ce faire, elle prend l'adresse à laquelle la valeur est supposée se situer et un pointeur vers la donnée unique. Ensuite, il y a uint8_t lecture(const uint16_t adresse, uint8_t *donnée, const uint8_t longueur) : La méthode permet la lecture d'un bloc de données de moins de 127 et pour ce faire, elle prend l'adresse à laquelle la première valeur du bloc se situe, un pointeur vers la première donnée du tableau à lire ainsi que la longueur du bloc de donnée. Il y a la méthode uint8_t ecriture(const uint16_t adresse, const uint8_t donnée) : la méthode permet l'écriture d'une donnée en mémoire en donnant l'adresse à laquelle on souhaite insérer la valeur ainsi que la donnée à insérer. La dernière méthode publique est uint8_t ecriture(const uint16_t adresse, uint8_t *donnée, const uint8_t longueur) : la méthode permet l'écriture d'un bloc de données en mémoire en donnant l'adresse à laquelle on insère le bloc de donnée, un pointeur vers la première donnée du tableau à écrire ainsi que la longueur du bloc de donnée. L'unique private est uint8_t ecrire_page(const uint16_t adresse, uint8_t *donnée, const uint8_t longueur) : Cette méthode privée permet d'écrire en tenant compte des limites physiques d'une page et pour ce faire, elle prend l'adresse à laquelle on souhaite écrire

la page, un pointeur vers la première valeur des données à écrire sur la page ainsi que la longueur des données à écrire. Ainsi, l'écriture se fera en respectant les limites physiques de la page pour qu'elle ne soit pas en dehors. Le destructeur de Memoire24CXX() détruit l'objet instancié à partir de la classe.

Classe **Can:**

Cette classe fournie par les chargés de laboratoire est une classe permettant le contrôle du convertisseur analogique/numérique. Ainsi grâce à cette classe, nous pouvons transformer les données analogiques du monde extérieur en numérique pour que le robot puisse réagir convenablement ou même envoyer des données du robot au monde extérieur.

Le constructeur de can ne prend en compte aucun paramètre d'entrée. Puisque le port A est le seul port permettant la conversion des données. Le port A est ainsi initialisé grâce à des références analogiques externes pour être en mesure de recevoir des signaux externes et les convertir.

La méthode implémentée uint16_t lecture (uint8_t pos) prend en paramètre la position du pin qui lira les données analogiques et retournera la valeur numérique associée à la valeur analogique lue à la position sur le port A.Cette fonction utilise aussi le principe de masque pour être en mesure de garder les bits intacts lors de la sélection du pin responsable de l'entrée des données analogiques. Ensuite lorsque la conversion est effectuée, le résultat est retourné sur 16 bits au terminal. Le destructeur permet de rendre le port inactif ainsi, sa fonction principale qui est de convertir les données analogiques/numériques n'est plus accessible.

Classe **Counter:**

La classe counter sert à avoir une minuterie fonctionnelle avec le timer 1 de sorte que l'on puisque manipuler les actions du robot plus précisément.

D'abord son constructeur ne prend rien en paramètre, mais il configure la variable publique volatile bool gExpiredTimer à faux. Elle détermine que le temps à compter n'est pas encore expiré.

Par la suite, nous avons trois méthodes publiques void startTimerCTC(uint16_t cycles), void startTimerPWM(uint16_t cycles) et void startTimerFastPWM(uint16_t cycles). Le principe de ces méthodes est la suivante : la méthode prend en paramètre un nombre de cycles selon le mode CTC, PWM et FastPWM. Pour faire cela, le code affecte plusieurs registres : TCNT1, la variable initiale du compteur allant jusqu'à 16 bits; OCR1A est la valeur de comparaison avec notre timer, lorsque celle-ci est égale à TCNT1 le ISR de TIMER1_COMPA_vect est lancée; il y a TCCR1A qui nous permet de choisir le mode du timer et qui dicte le comportement des OCnA et OCnB, mis en mode normal donc qui ne les affecte pas à un *comparaison match*; TCCR1B qui choisit le prescaler qui sera à 1024 par défaut dans notre situation. TCCR1C qui n'exerce aucun rôle pour cette minuterie; dernièrement TIMSK1 qui permet la comparaison avec OCnA dans notre cas. Selon le comportement que nous voulons que notre minuterie fonctionne, nous aurons des méthodes prêtes à bien initialiser le timer. De plus, la méthode uint16_t convertTimeToCycles(uint32_t timeUs) prend en paramètre un temps en microsecondes de taille maximale de 32 bits et retourne le nombre de cycles qui correspond au temps passé en paramètre.

Debug :

Debug.h contient la fonction void *print (x) et la fonction void printSentence* qui prennent en paramètre une donnée de type uint8_t ou un char [] respectivement et nous transfère la donnée ou chaque lettre de notre char [] en appelant la fonction *transferUart* de la classe communication. Ces fonctions ne seront exécutées que si on définit debug (en utilisant la commande make debug). Si on ne définit pas debug, une boucle while où rien ne se passe (code mort) sera exécutée au lieu du transfert de donnée(s).

Partie 2 : Modifications apportées au Makefile

Afin de pouvoir compiler tous les fichiers cpp à la fois, sans devoir inclure tous les noms.cpp des fichiers, on a modifié la ligne 34 en déclarant PRJSRC = $(wildcard *.cpp). Une nouvelle variable (avr-ar) a été rajoutée à la ligne 64. Cette variable remplace la variable ar puisqu'elle est plus adaptée pour la création de libraires AVR. La variable TRG (ligne 92) a été modifiée afin de produire une librairie statique (. a) au lieu d'un fichier .hex. Nous avons enlevé les lignes suivants : AVRDUDE=avrdude et HEXFORMAT = ihex puisque le makefile de la librairie ne doit pas envoyer celle-ci vers le microcontrôleur et puisque la librairie ne produit pas de fichier .hex. La ligne 124 a finalement été modifiée pour retirer HEXROMTRG de la commande all, car aucun fichier .hex ne doit être produit.