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 5758

Noms:

Loic Nguemegne-Temena

Arman Lidder

Jonathan Arrobas Gilbert

Hamza Boukaftane

Date:

31 octobre 2022

Cette documentation porte sur la librairie statique que nous avons conçue lors des travaux pratiques 7 et 8. Une librairie est dite statique quand cette dernière est compilée avec le programme qui l'utilise. Dans ce rapport, il s'agit d'établir une documentation exhaustive sur le fonctionnement de notre librairie et de son utilité dans la programmation de notre robot. Notre librairie est composée de fichiers « .cpp » et « .h » associés à différentes classes que nous avons déterminées nécessaires à la programmation des travaux pratiques 1 à 6. L'utilisation de cette librairie est compatible avec les microcontrôleurs ATMega324PA. De plus, notre librairie est programmée en C++ avec l'importation de divers modules de la librairie avr-libc. Il sera question d'une description des différentes classes de la librairie et d'une description des modifications apportées au Makefile de départ.

Description de la librairie statique

- Description de la classe Led:

L'objectif de cette classe est de permettre d'allumer la LED sur la carte-mère. Pour ce faire, quatre attributs sont nécessaires. Voici la liste, leur utilité ,et un exemple de valeur pour chacun:

*volatile uint8_t *ddrx_ :* Permet d'ajuster la direction du port voulu (Ex:&DDRB)

*volatile uint8_t * port_ :* Permet d'écrire sur le port voulu (Ex:&PORTB)

int pinOne_ , int pinTwo_ : Permettent de déterminer les deux bits sur lesquels il faudra ajuster la direction du port et écrire dessus pour allumer la LED. (Ex: PB0 ET PB1)

Afin d'allumer la LED, il est tout d'abord nécessaire de préciser ces valeurs durant l'instanciation d'un objet, donc le constructeur les prend en paramètre afin de bien initialiser correctement ses attributs. Le constructeur va directement ajuster le port voulu en mode sortie, et ce, uniquement sur les pins passés en paramètres. Après l'instanciation, la LED est éteinte et il faudra utiliser les méthodes pour la changer. Ensuite, la classe comprend 5 méthodes qui permettent de recouvrir ce que l'utilisateur va vouloir faire raisonnablement avec cette classe. Voici la liste et les descriptions:

void setLedOff(): Éteint la LED en mettant les deux bits du port à 0 selon *pinOne_* et *pinTwo_*.

void setLedRed(): Allume la LED en rouge en mettant le premier des deux bits à 1 et l'autre à 0.

void setLedGreen(): Allume la LED en vert en mettant le deuxième des deux bits à 1 et l'autre à 0.

void setLedAmbreOnce(): Allume la LED à la couleur verte suivi d'un délai, suivi de la couleur rouge, suivi d'un autre délai. Ce délai est une constante uint8_t valant 10 ms. Cette fonction prend donc 20ms à s'exécuter. C'est donc une méthode bloquante. La combinaison des couleurs donne une couleur ambre.

void setLedAmbreLoop(uint16_t loopTime): Allume la LED à la couleur ambré pendant un temps de [*loopTime*] ms. C'est donc une méthode bloquante.

- Description de la classe ButtonInterruption:

L'objectif de cette classe est de permettre l'activation d'interruption externe par l'intermédiaire de bouton-poussoir sur le robot. Pour ce faire, la classe doit être en mesure de configurer les registres suivants du microcontrôleur : *EIMSK* qui permet les interruptions externes (max = 3) et *EICRA* qui établit la sensibilité des interruptions externes aux changements de niveaux engendrés par le bouton-poussoir (max = 4). Le fichier en-tête comporte aussi deux types enum : *IntConfig* représentant les configurations de *EIMSK* (int0, int1 et int2) et *Edge* représentant les configurations possibles de EICRA (Low, Any, Falling, Rising).

La classe ne comporte qu'un seul constructeur qui prend en paramètre *IntConfig* et *Edge* afin d'ajuster les bits des registres concernés. Ainsi, lors de l'instanciation d'un objet de la classe, les arguments passés en paramètres au constructeur passeront dans deux switch-case, un pour la configuration de *EISMK* et l'autre pour la configuration de *EICRA*. De plus, le constructeur initialise une routine de blocage des autres interruptions [cli()] et la permission de recevoir d'autres interruptions une fois celle en action terminée [sei()].

Pour être fonctionnelle, la classe doit être complémentée d'une routine d'interruption ISR avec le vecteur approprié (voir documentation avr-libc) dans un fichier main.cpp. Cette dernière permet de configurer le comportement d'une interruption par bouton-poussoir et le registre *EIFR* (souvenir des autres interruptions qui ont été bloquées). De plus, au niveau du matériel, un cavalier doit être placé sur *IntEN* afin de bien fonctionner (port D2 est utilisé par le microcontrôleur) .

- **Description de la classe Motor:**

L'objectif de cette classe est de contrôler les 2 moteurs présents sur notre carte mère alimentés par le pont-H afin de faire avancer ou reculer le robot. Une seule instance de Motor sera suffisante pour contrôler les 2 roues motrices. Le constructeur par défaut configure le moteur afin que le robot avance avec une puissance à 100% sur les 2 roues. L'autre constructeur de cette classe prend 4 paramètres représentant les 4 attributs privés de la classe (dans le bon ordre) permettant une plus grande flexibilité de programmation:

uint8_t leftMotorDirection_ : Cet attribut privé permet de définir la direction du moteur gauche, il peut recevoir les valeurs constante int8_t *FORWARD* = 0 pour avancer et BACKWARD = 1 pour reculer.

uint8_t right Motor Direction_ : Cet attribut privé permet de définir la direction du moteur droit, il reçoit les mêmes valeurs que le paramètre précédent.

uint8_t leftMotorPower_ : Cet attribut privé permet de déterminer la puissance de rotation du moteur gauche. Il peut prendre une valeur entière positive entre 0 et 255. Cependant, 5 constantes de type uint8_t représentant le ratio du moteur ont été définies pour faciliter l'utilisation: *ZERO = 255* **(0%)**, *TWENTY_FIVE = 192* **(25%)**, *FIFTY = 128* **(50%)**, *SEVENTY_FIVE = 64* **(75%)**, *HUNDRED = 0* **(100 %)**. À 25%, les roues ne tournent pas dû à l'inertie dans le moteur.

uint8_t rightMotorPower_ : Cet attribut privé permet de déterminer la puissance de rotation du moteur droit, il reçoit les mêmes valeurs que le paramètre précédent.

Voici les méthodes qui compose la classe:

void setLeftMotorDirection(uint8_t directionMotor): Cette méthode publique permet de changer le sens de rotation du moteur gauche grâce au paramètre *directionMotor* qui peut avoir les valeurs: **FORWARD**(avancer) et **BACKWARD**(reculer).

void setRightMotorDirection(uint8_t directionMotor): Cette méthode publique permet de changer le sens de rotation du moteur droit en utilisant le principe de la méthode précédente

void setLeftMotorPower(uint8_t puissanceMotor): Cette méthode publique permet de changer la puissance du moteur gauche grâce au paramètre *puissanceMotor* qui peut prendre une valeur de **0** à **255.**

void setRightMotorPower(uint8_t puissanceMotor): Cette méthode publique permet de changer la puissance du moteur droit en utilisant le principe de la méthode précédente.

void PWM(): Cette méthode privée est utilisée par le constructeur pour configurer les registres nécessaires à la génération d'une onde PWM pour le fonctionnement des moteurs. Dans notre cas, nous avons utilisé la minuterie 1 et l'avons configurée dans le mode PWM phase correct voir la documentation d'Atmel(16)

void configurePWMPort(): Cette méthode privée est utilisée par la méthode PWM(), pour configurer les ports PD4 et PD5 pour la sortie du PWM.

void changeLeftMotorDirection(uint8_t direction): Cette méthode privée est utilisée par *setLeftMotorDirection(uint8_t directionMotor)* qui l'appelle après avoir mis à jour l'attribut de la classe à une nouvelle direction pour que le changement soit effectif.

void changeRightMotorDirection(uint8_t direction): Cette méthode privée est utilisée par: Elle est utilisée par *setRighttMotorDirection(uint8_t directionMotor)* de la même manière que la précédente.

Au niveau matériel, OC1A (contrôle de la puissance du moteur gauche) génère un signal de sortie au port D5 et le port D7 influence la direction de la roue gauche. OC1B (contrôle de la puissance du moteur droit) génère un signal de sortie du port D4 et le port D6 influence la direction de la roue droite.

- **Description de la classe Timer:**

Le but de la classe Timer est d'avoir recours à un autre timer différent de celui utilisé pour les moteurs. Pour cela, nous avons opté pour l'utilisation de *timer/counter0*, un des deux timers 8-bit de l'ATMega. Lors du réglage du timer0, nous devons d'abord définir notre valeur de fond en utilisant le registre TCNT0. Ce registre contient la valeur du timer pendant qu'il compte. Notre initialisation par défaut du timer0 commence à compter à partir de 0. De plus, les bits des registres TCCR0x (x est A,B ou C) déterminent notre mode de comptage ainsi que la valeur du prescaler. Nous avons opté pour le mode CTC et une valeur de prescaler de 1024. La valeur supérieure du timer est stockée dans le registre OCR0 et sera déterminée par le paramètre *duration*. Notez que si nous souhaitons modifier les valeurs dans les registres, nous devons désactiver les interruptions en utilisant cli() et les réactiver en utilisant sei() par la suite. Bien que cette classe ne fonctionne pas directement avec un ISR, cette classe peut et sera utilisée en conjonction avec un ISR implémenté en dehors de la classe timer elle-même. En ce qui concerne la classe et ses méthodes, 3 méthodes ont été implémentées pour l'utilisation de cette minuterie:

initDefault: Initialise le *timer0* à notre état par défaut. Prend *duration* comme paramètre, qui définit la limite supérieure de notre timer.

setTimer: Permet de réinitialiser le *timer0* et de commencer à compter jusqu'à une nouvelle valeur supérieure. Il prend *duration* comme paramètre.

setClkScale: Contient un switch-case, nous permettant de choisir de nouvelles valeurs de prescaler interne en fonction de nos besoins.

Pour utiliser efficacement cette classe, nous devons d'abord appeler *initDefault* avec notre paramètre *duration* spécifié. La durée peut être calculée au moyen d'une formule simple, ce qui nous permet de déterminer le nombre exact de "comptes" (c'est-à-dire la valeur du paramètre *duration*). Cette formule est la suivante : $Timer\ count = \frac{Real-time\ delay}{Clock\ frequency} - 1$. Une fois que le *timer0* est initialisé avec *duration* spécifiée, nous avons la possibilité de re-spécifier notre valeur supérieure ainsi que notre valeur de prescaler. Ceci peut être réalisé en utilisant respectivement setTimer et set Clk Scale. Comme cette minuterie est utilisée de manière indépendante, nous n'avons pas à nous soucier de la réinitialiser une fois que la classe a été instanciée et la minuterie initialisée, que ce soit pour une utilisation alternative ou une allocation de ressources.

- **Description de la classe Usart:**

L'objectif de cette classe est de permettre la transmission de données à partir du *USART* du microcontrôleur ATMega324PA par le protocol RS232 vers le ATmega8. Pour ce faire, cette classe les registres suivants: *UCSR0A* qui agit sur les flags Réception complète et Transmission complète, *UCSR0B* qui permet l'activation de la transmission ou la réception de donnée et UCR0C qui configure le mode, le bit de parité, le stop bit et la taille des caractères reçus. Comme nous utilisons le *USART0* du microcontrôleur, sa configuration de base est de 2400 baud, 8 bits, aucune parité, 1 start/stop. La configuration de de la vitesse de transmission d'information de 2400 baud est faite par les registres *UBRR0H* (4 bits les plus significatifs) et *UBRR0L (autres bits moins significatifs)*. La classe contient le type enum DataType qui permet de représenter les différentes types de données qui seront transmises (UNIQUE_CHAR, CHAR_LIST, UNIQUE_INT). Voici les différentes méthodes implémentées par la classe:

USART(): Constructeur de la classe qui configure les registres UBRR0H à 0 et UBRR0L à 0xCF pour obtenir une vitesse de transmission des données de 2400 bauds. De plus, le registre *UBSR0B* est configuré pour activer la transmission et la réception de données (*RXEN0* et *TXEN0*) mis à 1. Enfin, le registre *UBSR0C* est configuré pour ajuster la taille des caractères reçus à 8 bits (*UCSZ01* et *UCSZ00* mis à 1). Ainsi, le constructeur permet la configuration de communication de 2400 bauds, sans bit de parité et des trames de 8 bits séparées par un bit d'arrêt.

void transmitDataInt (DataType dataType, uint8_t data, char text[ ]) : Méthode publique permettant la transmission de données en fonction du type de donnéee. Cette méthode dépend des méthodes suivantes soit transmitDataUnique(uint8_t data), transmitText (char text[]) et transmitInt (int integer). En effet, cette méthode est constituée d'un switch-case qui selon le dataType passé en argument à la méthode va exécuter une des trois autres méthodes de transmission. Pour ce qui est des autres arguments passés à la méthode, ils seront réutilisés dans les autres méthodes incluses dans transmitDataInt. Aussi, les paramètres data et text[ ] possèdent des valeurs par défaut. Pour data, c'est 0 et pour text[], c'est un tableau vide. Cela permet au programmeur de passer en argument à cette méthode seulement ce dont il a besoin.

void transmitDataUnique(uint8_t data): Méthode publique permettant la transmission d'une donnée unique. En fait, cette méthode prend en argument un uint8_t data qui représente la donnée à transmettre. Ensuite, la méthode attend que le tampon de transmission soit vide (*while(!(UCSR0A & (1<<UDRE0)*, on ne fait rien!). Une fois le tampon vide, on place la donnée dans le tampon et on transmet la donnée (*UDR0 = data*).

void transmitText(char text[]): Méthode publique permettant la transmission de données sous forme de chaîne de caractères. Pour ce faire, on passe en argument un tableau de caractères à la méthode. En effet, dans la méthode, on définit le bit d'arrêt comme étant 0 (char stop = "/0"). Puis, on itère sur le tableau de caractères passé en argument tant que l'élément du tableau est différent du bit d'arrêt et on passe chaque élément en argument à la méthode de transmission de données unique transmitDataUnique(uint8_t data).

void transmitInt(uint8_t integer): Méthode publique permettant la transmission de données de type entier. Pour ce faire, nous définissons un tableau de caractère possédant une dimension de 5 (char buffer[5]). Ensuite, la méthode formate l'argument passé qui lui est passé dans le buffer sous forme de C string grâce à la fonction snprintf. Ainsi, on obtient un tableau de caractère représentant l'integer passé en argument. Puis, à l'aide de la méthode transmitText (incluse dans la méthode transmitInt) à qui nous passons la variable buffer en argument, on transmet l'entier formaté. Finalement, nous utilisons la méthode transmitDataUnique (incluse dans la méthode transmitInt) pour transmettre la chaîne de caractère "/t" (commande tabulation) par souci d'affichage.

Enfin, cette classe permet de transmettre des données du ATMega324PA vers le ATMega8 par protocole RS232. Pour être fonctionnelle, le cavalier doit être placé sur *DbgEN*. Ainsi, les ports D0 et D1 du ATMega324PA seront utilisés pour la transmission. Cette classe est utile notamment pour la classe Debug.

- **Description de Debug:**

Le Debug est un outil permettant de faire l'affichage de certaines variables si besoin. Son utilisation est couplée à un makefile qui aura la responsabilité de définir ce dernier. Ainsi, lorsque *DEBUG* est définie en exécutant le programme grâce à la commande *make debug*, une instance transmit de la classe Usart sera créée et toutes les expressions *DEBUG_PRINT(datatype, data, data2)* se trouvant dans le code seront remplacées par *transmit.transmitDataInt(datatype, data, data2)* permettant ainsi la transmission de donnée vers le ATMega8. On fera appel à *serieViaUSB -l* pour visualiser ces données sur un ordinateur.

- **Description de la classe can :**

L'objectif de cette classe est de permettre l'utilisation du convertisseur analogique/numérique du microcontrôleur. Cette classe manipule les registres *ADMUX* et *ADCSRA*. Elle est composée de trois éléments : un constructeur, un destructeur et une méthode *lecture*.

Constructeur *can()*: Ce constructeur ajuste tous les bits du registre *ADMUX* (*ADC Multiplexer Selection Register*) à 0. Il va également ajuster les bits *ADEN*, *ADPS2*, et *ADPS1* du registre *ADCSRA* (*ADC Status and Control Register A*) à 1.

Destructeur *~can()*: Ceci permet de rendre le convertisseur inactif en ajustant le bit *ADEN* du registre *ADCSRA à 0.*

Méthode *uint16_t lecture(uint8_t pos):* Cette méthode prend en paramètre l'entrée du signal analogique sur le port A (Ex: PA0). Elle utilise les registres *ADMUX*, *ADCSRA* et les registres de données *ADCL* et *ADCH* dans son processus afin de retourner un résultat *adcVal* sur 16 bits (puisque la résolution est de 10 bits).

Ainsi, pour convertir, par exemple, une tension analogique à numérique, il suffit de créer un objet de type *can* et appeler la méthode *lecture* sur celui-ci en passant en paramètre PORTAX avec x entre 0 et 7. Finalement, nous devons manuellement retirer les 2 derniers bits (LSB) de la valeur retournée par la lecture, car ceux-ci ne sont pas significatifs. Cette opération se fait à l'extérieur de la classe. Nous ne sommes pas les concepteurs de cette classe pour d'informations veuillez consulter la documentation Atmel section 23.

- **Description de la classe Memoire24CXXX :**

L'objectif de cette classe est de permettre l'accès à la mémoire du robot et permet la lecture ou l'écriture de la mémoire externe du microcontrôleur ATMega324PA. Cette classe est composée des éléments suivants : un constructeur, un destructeur, des méthodes publiques, des méthodes privées et des attributs privés.

Memoire24CXXX() : Constructeur ne prenant aucun argument qui appelle la méthode init().

~Memoire24CXXX() : Destructeur abolissant l'accès à la mémoire.

void init() : Procédure d'initialisation à zéro du *memory* bank qui est appelé par le constructeur.

static uint8_t choisir_banc(const uint8_t banc) : Permet de changer l'adresse si nécessaire.

uint8_t lecture(const uint16_t adresse, uint8_t *donnee) : Méthode publique de lecture des données une valeur à la fois.
uint8_t lecture(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur) : Méthode publique de lecture de bloc de données dont la longueur doit inférieure ou égale à 127 bits.

uint8_t ecriture(const uint16_t adresse, const uint8_t donnee) : Méthode publique d'écriture des données une valeur à la fois.
uint8_t ecriture(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur) : Méthode publique d'écriture de bloc de données dont la longueur doit inférieure ou égale à 127 bits.

uint8_t ecrire_page(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur) : Méthode privée d'écriture appartenant à la classe.

static uint8_t m_adresse_peripherique : attribut privée qui représente l'adresse du périphérique.
const uint8_t PAGE_SIZE : attribut privée qui représente la taille d'une page.

Dans notre cas, nous utiliserons seulement les méthodes publiques de lecture ou d'écriture, ainsi que le constructeur pour permettre l'accès en mémoire. Au niveau matériel, un cavalier doit être placé sur *MemEn* afin qu'un programme utilisant cette classe soit fonctionnel. Dans un tel cas, les ports C0 et C1 du microcontrôleur seront utilisés. Nous n'avons pas conçu cette classe, donc pour plus de détails concernant cette classe, veuillez consulter la documentation sur Mémoire24CXXX.

Modifications apportées au Makefile de départ

Afin de pouvoir installer un programme utilisant notre librairie statique dans notre robot, nous avons généré deux Makefiles à partir du Makefile initial : un Makefile pour la librairie et un Makefile pour l'exécutable chargé dans le robot. Dans un premier temps, abordons les modifications apportées au Makefile de la librairie. Comme notre librairie contient plusieurs fichiers sources, nous avons affecté la fonction $(wildcard *.cpp) à la variable *PRJSRC*. Cela permet d'aller chercher tous les fichiers .cpp contenu dans le répertoire de la librairie sans avoir à les lister un par un. Ensuite, sachant que ce dernier doit générer un fichier de bibliothèque statique (extension .a dans les système Unix), nous avons défini une variable *LIBCC* affectée à la commande unix avr-ar qui appelle le compilateur générant un fichier .a. Puis, nous avons modifié l'extension de la variable *TRG* par .a. Enfin, nous avons modifié la commande associée à l'implémentation de la cible par $(*LIBCC*) -crs $(TRG) $(OBJDEP) qui permet de compacter tous les fichiers .o du répertoire actuel dans le fichier archive (TRG). Nous avons également supprimé du makefile les règles qui permettent la création de fichiers .HEX inutiles pour notre librairie.

Dans un deuxième temps, abordons les modifications apportées au Makefile de l'exécutable. Nous avons affecté la commande -I ./../lib (i majuscule) à la variable *INC*. La dernière commande Unix permet d'inclure le chemin du répertoire lib qui va être utilisé dans le flag du compilateur en C. Ensuite, nous avons affecté la variable *LIBS* à la commande unix -L ./../lib -lrary qui permet d'ordonner au compilateur de lier le fichier library.a généré par le Makefile de la librairie à l'étape du *Linkage*. Ultimement, cette commande sera utilisée lors de l'implémentation de la cible et permettra l'édition d'un lien de la librairie statique vers l'exécutable généré. Enfin, la dernière modification effectuée sur ce Makefile est l'ajout d'une règle *debug*. Elle permet de définir le DEBUG implémenté dans debug.h. Pour créer cette règle, nous avons suivi les étapes suivantes: créer un flag pour les compilateurs C(debug: CCFLAGS += -DDEBUG -g) et C++(debug: CXXFLAGS += -DDEBUG -g), ensuite définir l'action qui sera posée par la règle: (debug: install, le programme va être chargée sur la carte) et enfin, nous avons lancer le programme permettant de visualiser le contenue de la mémoire interne pour que tout se fasse automatiquement (debug: serieViaUSB -l).