Département de génie informatique et génie logiciel

Cours INF 1900:
Projet initial de système embarqué

Travail pratique 7

Makefile et production de librairie statique

Numéro d'équipe : 0320

Noms:

Zakarya Khnissi
Einstein Franck Tiomo Epongo
Zabiullah Shair Zaie
Moussa Daoud

Date:
Mercredi 17 mars 2021

Partie 1 : Description de la librairie

1. PWM.cpp / PWM.h

➢ **`adjustPWM(uint8_t duration)`**

La fonction adjustPWM() génère deux signaux de sortie qui sont assignés aux broches 4 et 5 du PORTD.

Le registre TCCR1A a été configuré de façon qu'il change le signal de sortie a chaque fois que le compteur TCNT1 atteint la valeur de l'un des registres OCR1A ou OCR1B (TCCR1A = (1 << COM1A1) | (1 << COM1B1)). Cette valeur varie de 0 à 255 qui est la valeur du paramètre "duration" passé en paramètre.

Le timer1 a été initialisé en mode PWM phase correcte, ou le compteur TCNT1 ne se remet pas à zéro quand la valeur de OCR1A/OCR1B est atteinte, il continue jusqu'à atteindre la valeur du TOP qui est de 0xFF et revient après à sa valeur initiale (TCCR1A = (1 << WGM10)).

Le registre TCCR1B a été configuré de façon qu'il divise la fréquence d'horloge du CPU par 8 (TCCR1B = (1 << CS11)).

2. Button.h / Button.cpp

➢ **`Button(volatile uint8_t* DDRx, uint8_t pin)`**

Construction de la classe Button qui permet de mettre la PIN sur laquelle un est connecté en entrée.

➢ **`Button(volatile uint8_t* DDRx[], uint8_t pins[], uint8_t nbButtons)`**

Dans le cas où on voudrait mettre en entrée les pins de plusieurs boutons simultanément, cette fonction peut être appelée, plutôt que de faire des appels successifs au premier constructeur.

➢ **`setSenseControl(PseudoPin pseudoPin,InterruptSenseControl ics)`**

Décide de quel front/niveau le bouton passé en paramètre activera une interruption et configure également le registre qui permettra de recevoir des interruptions externes sur le bouton passé en paramètre. PseudoPin est une énumération qui ne contient que les PINS PD2, PD3 et PB2 étant donné que ce sont les seules sur lesquelles un bouton peut déclencher une interruption.

➢ **`bool debounce(uint8_t &pin, uint8_t pressedButton)`**

Pour un bouton donné, cette fonction vérifie son état et retourne vrai si le bouton est pressé après un certain temps d'attente qui a pour but de laisser le signal se stabiliser. La fonction requiert en paramètre le masque correspondant un bouton sur le port et la pin sur laquelle est connecté ce dernier.

➢ **`uint8_t mask(uint8_t &port, uint8_t pin)`**

Retourne 1 si le signal lue sur une pin est 1 et retourne 0 autrement. De cette façon, on sait si un bouton a envoyé un signal sur une PIN donnée.

3. Utils.h / Utils.cpp

➢ **`void setPinState(volatile uint8_t &port, uint8_t pins[], PinState pinState[], uint8_t nbPins)`**

Définit l'état (LOW, HIGH) de chacune des PINS reçues en paramètre, ce qui permettra respectivement de faire entrer une valeur dans le circuit via cette PIN ou de pouvoir l'utiliser comme sortie pour envoyer des valeurs vers l'extérieur du circuit.

➢ **`void setDELColor(uint8_t &port, uint8_t color)`**

Change la couleur d'une DEL qui serait connectée à deux PINS d'un certain port, en changeant la valeur sur le port en question.

➢ **`void dynamic_delay_ms(uint16_t delay)`**

Étant donné la contrainte que l'utilisation des fonctions **`_delay_ms()`** et **`_delay_us()`** à propos du type constant du paramètre qu'elles doivent prendre, le besoin d'avoir des délais variables surgit. Cette fonction permet, pour n'importe quelle valeur de délai, de retarder les opérations du microcontrôleur de cette valeur donnée. C'est de cette façon qu'un PWM de façon logicielle peut être généré.

4. timer.h / timer.cpp

La librairie inclut aussi des fonctions qui seront utiles pour le timer du ATmega324PA. Pour cette partie de la librairie, on se limite au timer 1 et aux quelques modes de fonctionnement vus dans le cadre de cours. Les fonctions retenues sont :

initializeTimer() qui permet la configuration des registres, “startTimer” qui va partir la minuterie et ajuster les registres pour effectuer une comparaison et la dernière est le setPrescaler() qui nous permet de choisir les prescaler qu’on désire pour la minuterie.

- **`startTimer(uint16_t duration, uint16_t stop, unsigned int prescaler)`**

  La fonction « startTimer » a été retenue dû à la grande importance de la notion timer et de son usage fréquent dans les travaux pratiques. Il est nécessaire d’avoir une fonction pour partir la minuterie pour réaliser une minuterie.

  L’utilité de cette fonction est de pouvoir passer une valeur de début et une valeur d’arrêt en paramètre et déclencher une interruption au moment où les deux valeurs de comparaison sont égales. La minuterie est ajustée au mode CTC du timer 1 avec horloge divisée par 1024 pour déclencher une interruption après un certain moment. Pour réaliser ceci, la 6iem bit de registre de TIMSK qui correspond à OCIE1A est activé. Le bit qui contrôle le « WGM12 » de registre TCCRnA est activé et le registre « TCCR1C » reste à 0. Ensuite, la fonction prend en paramètre la valeur des registres « TCNTn » et « OCRn1 » pour donner plus de flexibilité sur le choix de moment de déclenchement de l’interruption.

- **`initializeTimer(uint8_t valueDDRx, uint8_t valueDDRy, volatile uint8_t* DDRx, volatile uint8_t* DDRy)`**

  La fonction « initializeTimer » du timer 1 de ATmega324PA. La raison pour laquelle elle a été retenue est parce qu’il est nécessaire de faire des ajustements sur les registres de « External interrupts » pour pouvoir utiliser la minuterie. Aussi, il permet de configurer les « Data Direction Register » en sortie ou entrée selon nos besoins.

  En ce qui concerne le fonctionnement de celle-ci, elle prend en paramètre des « DDRx », « DDRy », « valueDDRx » et « valueDDRy » pour donner plus de liberté sur le choix des ports pour des registres d' entrées et de sorties. La fonction est enveloppée par « cli() » et « sei() » pour bloquer toutes les interruptions. Ensuite, les registres « External interrupts » sont configurés où les deux premiers bits de registre EIMSK sont activés et seulement le premier bit de EIMSK est activé.

- **`setPrescaler(unsigned int prescaler)`**

  La fonction « setPrescaler » a été implémentée dans le but de rendre plus générale et améliorer la réutilisabilité des fonctions qui nécessitent une

configuration de preScaler. Puisque la notion minuterie est très importante dans le cadre de ce cours, c'est une bonne idée de généraliser cette fonction et pouvoir choisir les prescaler de nos choix.

La fonction, comme son nom l'indique, permet d'ajuster le « Prescaler » de la minuterie. La valeur de prescaler qu'on désire ajuster sur la minuterie doit être passée en paramètre et la fonction va effectuer la configuration avec l'instruction switch/case.

5. UART.h / UART.cpp

➢ **`initializeUART(void)`**

Le RS232 est le protocole de communication utilisé, il est défini sur 2400 bauds (UBRR0H = 0 / UBRR0L = 0xCF).

Le registre UCSR0B est initialisé de façon qu'il permet la transmission est la réception, cela est fait en activant les bits TXEN0 et RXEN0 ((1 << RXEN0) | (1 << TXEN0)).

Le format des trames : 8 bits, 1 stop bits, none parity, cela est réalisé en configurant les bits du registre UCR0C ((1 << UCSZ01) | (1 << UCSZ00) | (0 << USBS0) | (0 << UPM01) | (0 << UPM00)).

➢ **`transmissionUART (uint8_t data)`**

La fonction transmissionUART() permet la transition des données octet par octet. Cette fonction vérifie si UDRE0 du registre UCSR0A n'est plus à 0 est que le buffer est vide après elle commence la transmission des données passé en paramètre.

6. can.h / can.cpp

La classe can est utilisée par le convertisseur analogique numérique. On a jugé nécessaire d'inclure cette classe dans la librairie pour éventuellement s'en servir dans le projet final pour capter ou relever la distance entre un obstacle et notre robot lors du parcours de ce dernier.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Makefile de la librairie:

1. Nous avons d'abord utilisé le wildcard $(wildcard *.cpp) pour la variable $(PRJSRC) afin de créer la librairie à partir des fichiers sources .cpp présents dans le répertoire.

2. Par la suite on a mis le nom "libstatique" dans la variable $(PROJECTNAME) et avons modifié la cible $(TRG) en ajoutant l'extension .a pour référence à la librairie statique.

3. On a créé une variable $(AR) pour "avr-ar"

4. La variable $(MESOPTIONS) englobe les options "-crs" utilisées lors l'étape de linkage.

5. Lors de l'édition de lien, on a remplacé le compilateur "arv-gcc" pour faire appel à "avr-ar" pour l'archivage de la librairie tel que demandé dans l'énoncé en utilisant sa variable $(AR).

6. Finalement on a enlevé toutes les lignes et les options non nécessaires pour la création de la librairie comme par exemple les options de linkage représenter par la variable $(LDFLAGS).

Ce makefile produit tous les fichiers objets .o à partir des fichiers sources .cpp présents le même répertoire afin de réaliser l'édition de lien pour créer la librairie demandée.

Makefile de l'application test:

1. Nous avons complété les lignes commençant par "INC=" et "LIBS=" pour indiquer d'abord où se situent les fichiers «include» avec l'option -I (i majuscule) et puis on forme l'exécutable en donnant son nom avec l'option -L en inscrivant dans la variable $(LIBS) "-l statique -L../../lib_dir" et dans la variable $(INC) "-I../../lib_dir".

De cette façon, la librairie est appelée automatiquement.

2. Et puis nous avons modifié les variables $(PROJECTNAME) et $(PRJSRC) pour indiquer le nom et les fichiers source propres pour chaque projet.