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 151-165

Noms:

Safa Ikhlef - 2058906
Ivan Ivanov - 2087256
Mymanat Mohammed - 2068301
Eliott Schuhmacher-Wantz - 2081542

Date:
17 mars 2021

Partie 1 : Description de la librairie

Del

Pour construire notre librairie, nous avons créé plusieurs fonctions basées sur le code créé lors de la réalisation des travaux pratiques. De manière générale, ces fonctions correspondent à des méthodes qui pourront être réutilisées.

Tout d'abord, nous avons retenu plusieurs fonctions en lien avec les Del que nous avons regroupé ensemble dans un fichier del.h . Voici la première fonction :

void allumerDel(uint8_t couleur, uint8_t &port)

Cette fonction est assez simple. Elle allume une Del située sur un port qu'il faut préciser de la couleur désirée. Elle prend deux paramètres, *couleur* et *port*. Le paramètre *couleur* est de type uint8_t, il correspond au port qu'on met à HIGH (en sortie) et qui va donner la couleur verte ou rouge à la Del. Le paramètre *port* est de type uint8_t aussi qu'on passe par référence pour pouvoir modifier et il correspond au port où est branché la Del que l'on souhaite allumer. La Del peut être placée sur n'importe quel des ports A, B, C et D. Et donc n'importe quel de ces ports peut être passé en paramètre. Le port qui est utilisé pour la Del n'a pas besoin d'être spécifié en sortie. Cette fonction permet de rajouter très facilement une Del allumée de couleur verte ou rouge et sur n'importe quel port en permanence sur notre circuit.

La deuxième fonction est :

void allumerDelAmbre(volatile uint8_t &port)

Cette fonction allume de la couleur ambre (changer très rapidement entre vert et rouge) une Del. Elle prend en paramètre le port par référence sur lequel se situe la Del. La Del peut être placée sur n'importe quel des ports A, B, C et D. Et donc n'importe quel de ces ports peuvent être passés en paramètre. Le port qui est utilisé pour la Del n'a pas besoin d'être spécifié en sortie dans le main().

La troisième fonction est :

bool antiRebond(const uint8_t bouton)

Cette fonction est simplement la fonction d'anti-rebond, qui s'assure que l'on est bien en train d'appuyer sur un interrupteur. Elle prend une paramètre *bouton* qui est de type uint8_t et est constant. Ce paramètre correspond au signal du port (en assumant que les interrupteurs sont toujours sur le port D) dont on souhaite vérifier s'il est à 1. Le port qui est utilisé pour le bouton-poussoir n'a pas besoin d'être spécifié en entrée. Préalable : il faut que le bouton-poussoir du circuit soit sur le port D.

La quatrième fonction est :

void allumerDelBouton(const uint8_t couleur, const uint8_t bouton, volatile uint8_t &port)

Cette fonction permet d'allumer une Del située sur un port qu'il faut préciser de la couleur désirée à l'aide d'un bouton-poussoir. Elle prend trois paramètres, *couleur, bouton* et *port*. Le paramètre *couleur* est de type uint8_t, il correspond au signal du port qu'on met à 1 (en sortie) et qui va donner la couleur verte ou rouge à la Del. Le paramètre *port* est de type uint8_t et on le passe par référence. Il correspond au port où est branché la Del que l'on souhaite allumer. Le paramètre *bouton* est de type uint8_t, il correspond au signal du port D (en assumant que les interrupteurs sont toujours sur le port D) dont on va vérifier si elle est à 1. La Del peut être placée sur n'importe quel des ports A, B et C. Et donc n'importe quel de ces ports peuvent être passés en paramètre. Pas besoin de spécifier dans le main() les ports en sorties et entrée. Préalable : il faut que le bouton-poussoir soit sur le port D.

Mémoire

Ensuite, nous avons retenu plusieurs fonctions en lien avec la mémoire EEPROM que nous avons regroupé ensemble dans un fichier memoire.h . Voici la première fonction :

void initialisationUART8Bits1StopBitNoneParity(void)

Cette fonction permet d'initialiser le UART à 2400 bauds et permet la réception et la transmission par le UART0 selon le format des trames: 8 bits, 1 stop bits, sans parity bit. Nous avons décidé de ne rajouter que cette initialisation du UART à notre bibliothèque parce que c'est la seul que nous utilisons.

La deuxième fonction est :

void transmissionUART(uint8_t donnee)

Cette fonction permet l'envoie d'un octet passé en paramètre du microcontrôleur vers la portion de fenêtre du simulateur qui contrôle les opérations série en RS232.

La troisième fonction est :

void transmettreChainePc(uint8_t mots[], uint8_t nCaractere, uint8_t nTransmissions)

Cette fonction sert simplement à transmettre une chaîne de caractère vers le PC. *nCaractere* correspond au nombre de caratères qu'il y a dans la chaîne et *nTransmissions* est le nombre de fois que l'on veut transmettre la chaîne de caractère au PC. Cette fonction utilise la transmission via UART. Si on veut utiliser le mode spécifié à la fonction *initialisationUART8Bits1StopBitNoneParity(void)* il faut avoir fait un appel au préalable de cette fonction dans le main().

La quatrième fonction est :

void lireEepromAPc(uint8_t adresse, uint8_t nCaractere, uint8_t nTransmissions)

Cette fonction sert à transmettre les valeurs inscrites en mémoire eeprom vers le PC. On devrait normalement pouvoir voir ces valeurs s'afficher dans la console du simulateur. Le paramètre *adresse* doit contenir l'adresse à laquelle on souhaite commencer la lecture en mémoire. *nCaractere* correspond au nombre de caractères qu'il y a à lire (où s'arrêter) et *nTransmissions* est le nombre de fois que l'on veut transmettre la chaîne de caractère au PC. Nous avons décidé d'implémenter cette fonction en faisant un appel transmettreChainePC pour éviter la répétition du code. Si on veut utiliser le mode spécifié à la fonction *initialisationUART8Bits1StopBitNoneParity(void)* il faut avoir fait un appel au préalable de cette fonction dans le main().

La cinquième fonction est :

void ecrireLireEepromAPc(uint8_t tableauAEcrire[], uint8_t adresse, uint8_t nCaractere, uint8_t nTransmissions)

Cette fonction est assez semblable à lireEepromToPc, sauf qu'elle permet d'écrire la chaine de caractère contenue dans le paramètre *tableauAEcrire* dans la mémoire eeprom, puis de la lire et de la transmettre vers le PC. Le paramètre *adresse* doit contenir l'adresse à laquelle on souhaite écrire dans la mémoire. Les paramètres *nCaractere* et *nTransmissions* sont les mêmes que pour lireEepromToPc. Cette fonction fait un appel à la fonction lireEepromAPc(uint8_t adresse, uint_8 nCaractere, uint8_t nTransmissions) pour pouvoir faire la lecture et la transmission de la chaine de caractères vers le PC. Si on veut utiliser le mode spécifié à la fonction *initialisationUART8Bits1StopBitNoneParity(void)* il faut avoir fait un appel au préalable de cette fonction dans le main().

Minuterie

Nous avons aussi retenu plusieurs fonctions en lien avec les minuteries que nous avons regroupé ensemble dans un fichier minuterie.h . Voici la première fonction :

void initialisationMinuterie1(void)

Cette fonction permet d'initialiser le Timer 1 pour être sensible aux interruptions. Nous avons décidé de mettre cette initialisation dans la bibliothèque parce qu'on utilise très souvent le Timer 1 en combinaison avec les interruptions.

La deuxième fonction est :

void partirMinuterieTimer1ModeCTCPrescaler1024(uint16_t duree)

Cette fonction permet de partir le Timer 1 pendant une durée passée en paramètre. Le Timer 1 est réglé selon le mode CTC avec un prescaler de 1024. Nous avons choisi d'implémenter notre minuterie avec ce réglage spécifique parce que c'est le réglage de la minuterie que nous utilisons le plus souvent pour faire rouler la minuterie pendant une période donnée.

Moteur

Nous avons aussi retenu plusieurs fonctions en lien avec les moteurs que nous avons regroupé ensemble dans un fichier moteur.h . Voici la première fonction :

void ajustementPWMmaterielle(uint16_t pourcentage1, uint16_t pourcentage2)

Cette fonction ajuste les bons registres du timer 1 pour que deux signaux de PWM soient générés dans les broches réservées OC1B et OC1A (D4 et D5 respectivement). Elle ajuste le timer 1 en mode PWM 8 bits, phase correcte et valeur de TOP fixe à 0xFF (mode 1) et met à 1 les sorties de OC1A et OC1B sur comparaison réussie. Ainsi, si un E du circuit du pont en H est relié au signaux D4 ou D5 du microcontrôleur, la roue avancera (deux roues peuvent avancer en même temps). Il faut par contre régler la direction voulue pour chaque roue directement dans le main(), cette fonction ne l'ajuste pas (donc à 0 par défaut si les signaux du port du microcontrôleur où sont reliés les deux D du pont en H ne sont pas mis à 1 dans le main()). Il y deux paramètre, *pourcentage1* et *pourcentage2* qui correspondent respectivement au pourcentage du PWM qu'on envoie à la roue reliée à D5 et à D4. Ce pourcentage doit être donnée sur 8 bits, donc entre 0 et 255.

Nous avons décidé de ne pas inclure la PWM logicielle dans notre bibliothèque parce que nous voulons essayer de privilégier l'utilisation de la programmation par interruptions plutôt que la scrutation. De plus la PWM logicielle demande une redéfinition de la fonction à chaque fois que nous désirons choisir de nouvelles valeurs pour les pourcentages.

La deuxième fonction est :

void changerVitesseAvecLuminosite(uint8_t lumBasse, uint8_t lumAmbiante);

Cette fonction change la vitesse des roues en fonction de la luminosité. Elle prend en paramètre un seuil où la lumière est basse et un où elle ambiante. Si la luminosité est plus faible que *lumBasse*, le pwm des roues et de 25%, si elle est entre *lumBasse* et *lumAmbiante*, le pwm est de 50% et si elle est plus grande que *lumAmbiante* le pwm est de 75%. Un exemple de calcul de *lumBasse* serait : lumBasse = 2.4/4*255 .

Ajuster del

Finalement, nous avons aussi retenu deux fonctions en lien avec l'ajustement d'une Del que nous avons regroupé ensemble dans un fichier ajusterDel.h . Voici la première fonction :

void ajusterDelCan(can objet, const uint8_t palier1, const uint8_t palier2, const uint8_t palier3);

Cette fonction ajuste la couleur d'une Del en fonction de l'intensité lumineuse

captée par une photorésistance. Elle prend en paramètre trois paliers qui correspondent respectivement (de palier1 à palier3) au seuil où l'intensité luminosité captée est considérée basse, à un bon niveau et trop forte. Elle va changer la couleur de la Del pour vert pour une luminosité basse, pour ambre si elle est à un bon niveau et pour rouge, si elle est trop forte.
Cette fonction prend en paramètre aussi un objet de type can. En effet elle dépend de la classe can dont le constructeur initialise le CAN, et a une méthode qui permet de lire et retourner le résultat de la conversion sur 16 bits.
Préalables : il faut créer un objet can dans le main(), et définir les 3 paliers selon l'intensité en kOhms

La deuxième fonction est :

void ajusterDelPWM(uint16_t pourcentage, uint16_t signal)

Cette fonction ajuste la luminosité de la Del. Cette fonction fonctionne comme ajustementPWMmaterielle sauf qu'elle prend un seul pourcentage (une seule Del à la fois) et un signal du port D (0x10 pour D4 et 0x20 pour D5). La couleur de la Del dépendra du sens dans le lequel les fils sont branchés puisque le signal doit obligatoirement sortir de D4 ou D5.

Nous avons décidé de ne pas inclure de fonction qui implémenterai une machine à états parce que cette fonction n'est pas réutilisable dans le cas où il faut implémenter une machine à états avec un diagramme d'états différent.

Partie 2 : Décrire les modifications apportées au Makefile de départ

Makefile du dossier lib/ :

Makefile utilisé pour créer la librairie. Nous avons ajouté à la variable PRJSRC tous nos fichiers .cpp du dossier lib/, étant nos fichiers sources contentant les fonctions à ajouter à la librairie.

Pour la variable INC, nous avons mis : INC = -I ./include pour accéder au fichier .h utilisé pour le fichier ajusterDelCan.cpp qui fait un #include 'can.h'.

Sûrement une des modifications les plus importantes pour créer la librairie fut le fait qu'on a ajouté une variable AR = avr-ar crs qui nous permet de spécifier au compilateur que l'on veut créer une librairie plutôt qu'un exécutable.

De plus, le TRG (target) a été changer pour le PROJECTNAME.a, car nous créons une librairie et non pas un .elf qui sera transformé en .hex pour simuleIDE. Donc nous avons aussi supprimer les variable HEXPROMTRG et HEXTRG.

Pour les commandes, nous avons tout d'abord changé le all pour avoir all : $(TRG) qui correspond au nom de notre librairie.a. Donc pour l'implémentation de la cible, nous avons appelé AR suivi de notre cible étant librairie.a et de nos objets .o. La compilation de ces objets .o n'a pas changé.

Finalement, nous avons retiré la règle %.hex : %.elf car celle-ci est utilise seulement si l'on veut créer un executable pour simuleIDE. Et nous avons donc aussi supprimer la règle install : $(HEXPROMTRG).

Makefile du dossier exec/ :

Tout d'abord on a renommé ces variables-ci :
PROJECTNAME = main
PRJSRC = main.cpp

Puis nous avons ajouté notre instruction d'include à la vairable INC :

INC = -I ../lib/include

Cela va chercher dans le dossier d'avant le dossier lib et va trouver notre dossier rempli de .h pour les #include.

Et comme nous voulons lier notre librairie, nous avons ajouté le chemin à notre librairie à la variable LIBS. Or notre librairie ne se trouve pas dans le dossier exec, donc nous devons revenir en arrière et entre dans le dossier lib. Nous avons donc ceci :

LIBS = -L -I ../lib/librairie.a

Finalement, au niveau des commandes, nous avons légèrement modifié la règle pour l'implémentation de la cible. Nous avons ajouté $(LIBS) à la recette pour que le compilateur puisse trouver la librairie.