Polytechnique Montréal

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 2027

Noms:

Jordan Lecourtois

Hugo Paquin

Benjamin Thériault

William Trépanier

Date:

11 octobre 2019

**Partie 1 : Description de la librairie**

La librairie *libutils.a* contient les fonctions les plus utilisées dans le cadre de la programmation du robot. Elle contient les fonctions suivantes :

**Fonction qui permet d’attendre un nombre de cycles sans limite la fonction delay\_loop\_2 (delay.cpp)**

Fonctions :

delay\_cycles() : permet d’attendre un nombre de cycles. Prend en paramètre un nombre de cycles.

Utilisation : il faut simplement appeler la fonction avec un certain nombre de cycles.

**Fonctions utilisant le timer1 pour avoir une minuterie d’une durée théorique sans limites (timer.cpp):**

Fonctions:

init\_timer1() : initialise le timer1 et calcul le nombre de cycles à effectuer. Prend une durée en ms

delay\_timer1() : effectue le plus grand nombre de cycles que permet timer1 et ajuste delayTimer1\_cycles

reset\_timer1() : arrête le timer1

Il faut également une routine d’interruption qui permet de rappeler la fonction delay\_timer si la minuterie n’est pas terminée et qui appelle reset\_timer si la minuterie est terminée.

ISR (TIMER1\_COMPA\_vect )

{

if (delayTimer1\_cycles == 0)

{

reset\_timer1();

timer1done = true;

}

else {

delay\_timer1();

}

}

Utilisation : il suffit d’appeler la fonction init\_timer1 et de lui passer une durée, en ms, en paramètre. Il faut également initialiser la routine d’interruption et déclarer les variables globales suivantes :

delayTimer1\_cycles: nombre de cycles a effectué pour avoir un délai de duree\_ms

timer1done: sait si le délai total est termine

**Fonctions permettant la communication en série (UART) vers un PC (uart.cpp)**

Fonctions :

initialisationUART() : initialise la communication UART

transmissionUART() : envoie la donnée qui lui est passée en paramètre

Utilisations : il faut appeler initialisation UART puis transmission avec les données qu’on souhaite transmettre.

Pour voir les données transmises par le robot, il faut entrer la commande suivant dans le terminal :

SerieViaUSB -l -d -s 1

**Fonction permettant de générer un PWM avec le timer1 (navigator.cpp)**

Fonctions :

ajustementPWM() : permet d’initialiser le PWM. La sortie du OCRA est sur PD5 et la sortie OCRB est sur PD4. Cette fonction prend en paramètre puissanceDroite et puissanceGauche qui représente la vitesse [0,255].

Utilisation : il faut simplement appeler la fonction ajustementPWM avec, en paramètre, les deux puissances.

**Classe permettant l’accès au convertisseur analogique/numérique (can.cpp)**

Méthodes :

can() : Constructeur d’un objet permettant l’accès aux méthodes internes et qui initialise les registres pertinents.

lecture() : Retourne une lecture numérique correspondante à une valeur analogique.

Utilisation : Il suffit de créer un objet de la classe can, ainsi qu’à effectuer une lecture à la position passée en paramètre à la méthode lecture. La position correspond à un numéro de l’entrée sur le PORTC.

**Classe permettant le contrôle d’une diode électroluminescente (led.cpp)**

Méthodes :

Led() : Constructeur qui prend en paramètres le port ainsi que les deux pins qui sont connectés à la DEL.

turnGreen() : Met la DEL au vert.

turnRed() : Met la DEL au rouge.

turnOff() : Éteint la DEL.

turnAmber() : Oscille entre la couleur rouge et la couleur verte, permettant d’obtenir une couleur ambrée.

Utilisation : Il suffit de créer un objet représentant une LED en passant le port ainsi que les pins en paramètres. Les commandes permettent ainsi de choisir la couleur désirée ou de fermer la LED. Pour la méthode turnAmber(), il faut déterminer le temps requis et le passer en paramètre.

**Classe permettant l’accès à la mémoire (memoire\_24.cpp)**

Méthodes :

Memoire24CXXX() : Constructeur d’un objet permettant d’accéder à la mémoire.

init() : Méthode qui configure les registres appropriés et qui est automatiquement appelée par le constructeur.

lecture(adresse, \*donnee) : Lit un octet à l’adresse et écrit celle-ci à l’adresse spécifiée.

lecture(adresse, \*donnee, longueur) : Lit une quantité d’octets et écrit ceux-ci à l’adresse spécifiée.

ecriture(adresse, donnee) : Écrit la donnée passée en paramètre à l’adresse spécifiée.

ecriture(adresse, \*donnee, longueur) : Écrit les octets passés en paramètre à l’adresse spécifiée.

Utilisation : Il suffit de construire un objet et d’appeler les méthodes pertinentes. Lorsqu’une méthode nécessite un pointeur, il est possible de passer l’adresse du premier octet ainsi que le nombre d’octets.

**Partie 2 : Décrire les modifications apportées au Makefile de départ**

**Makefile de la librairie**

Nous avons modifié PRJSRC de sorte à utiliser tous les fichiers .cpp et .c plutôt que de spécifier chacun des fichiers à utiliser. Ainsi, dans l’éventualité de l’ajout d’un fichier, nous n’aurons pas à modifier le Makefile. En effet, nous avons modifié :

PRJSRC=fichier.cpp

Par :

PRJSRC=$(wildcard \*.cpp) $(wildcard \*.c)

Qui permet d’obtenir tous les fichiers .cpp et .c en utilisant un wildcard.

Également, nous avons ajouté une variable AR qui permet de spécifier l’archiveur, qui permet de créer et de maintenir les archives des librairies. Dans notre cas, nous utilisons avr-ar. Nous avons également ajouté une variable ARFLAGS permettant d’ajouter des flags lors de la compilation de notre librairie.

Nous avons ensuite modifié l’extension de notre fichier de sortie de sorte à produire un fichier .a, plutôt qu’un fichier .out, comme c’était le cas jusqu’à présent.

Finalement, nous avons changé la règle permettant la compilation de la cible. En effet, nous n’utilisons plus un compilateur, mais l’archiveur. La règle permettant l’implémentation de la cible est passée de :

# Implementation de la cible

$(TRG): $(OBJDEPS)

    $(CC) $(LDFLAGS) -o $(TRG) $(OBJDEPS) \

    -lm $(LIBS)

À :

# Implementation de la cible

$(TRG): $(OBJDEPS)

    $(AR) $(ARFLAG) -o $(TRG) $(OBJDEPS)

On peut voir que nous utilisons les variables AR et ARFLAG, qui sont respectivement l’archiveur utilisé et les flags utilisés pour la création de notre librairie. En l’espèce, les options (ARFLAGS) correspondent à -crs, où -c permet de créer une archive, -r insère les fichiers membres alors que -s produit l’index des fichiers objets.

**Makefile commun (à la source)**

Nous avons ajouté plusieurs variables dont INC, LIBINC, LIBDIR et LIBS.

INC=$(LIBINC)

LIBINC=-I ../libutils

La variable INC est passé à avr-gcc lors de chaque compilation (on le retrouve dans CFLAGS). Nous avons séparé INC de LIBINC, car il est possible d’inclure d’autres fichiers qui ne font pas partis d’une librairie. Pour limiter la prolifération des variables dans les commandes de compilation, LIBINC est inclue dans INC.

Pour que l’édition de liens fonctionne avec une librairie statique, il faut passer en paramètre le nom de la librairie avec le paramètre -l ainsi que le répertoire de la librairie avec le paramètre -L. On retrouve le nom de la librairie dans LIBS, nom qui est passé au linker, tout comme le répertoire de la librairie :

LIBDIR=-L ../libutils

LIBS=-l $(LIBNAME)

Le Makefile commun est conçu pour être inclus dans un autre Makefile, car il ne peut par lui-seul compiler un exécutable. Seules trois variables ont besoin d’être spécifiées dans le Makefile dans le dossier des fichiers sources :

PROJECTNAME=exec

SRCDIR=.

PRJSRC=$(wildcard $(SRCDIR)/\*.cpp) $(wildcard $(SRCDIR)/\*.c)

En plaçant le Makefile dans le même dossier que les fichiers .c et .cpp, l’ensemble de ceux-ci seront inclus dans le Makefile commun avec l’instruction suivante :

*include* ../Makefile