Polytechnique de Montréal

Cours INF1900:
Projet initial de système embarqué

TRAVAIL PRATIQUE #7
Makefile et production de librairie statique

Par l'équipe : 0817

Noms: Jérémy Perreault (1903274)
Éloïse Brosseau (2011037)
David Anastacio (2056948)
Douâa Bergheul (2006583)

Remis à
M. Tristan Rioux
M. Sami Sadfa
M. Jérôme Collin

Date:
Le mardi 3 novembre 2020

Partie 1: Description de la librairie

La librairie construite dans le cadre de ce travail pratique est composée de plusieurs éléments. Ceux-ci sont décrits dans les sous-sections suivantes.

isPressedDn() de la classe Bouton

La fonction `isPressedDn()` permet de vérifier que le bouton-poussoir, branché à la broche « n » du port « D », a bel et bien été pesé. En effet, en raison du phénomène de rebond, il faut mesurer deux fois la valeur d'un signal. La seconde fois confirme le résultat de la première, signifiant que le contact s'est stabilisé. Dans ce cas-ci, deux délais de 10 ms sont imposés afin d'effectuer cette vérification.

L'implémentation de la fonction est présentée à la figure 1 suivante:

```
bool isPressedD2(){
    if (PIND & (1 << PIND2)){
        _delay_ms(10);
        return (PIND & (1 << PIND2));
    }
    return false;
}

bool isPressedD3(){
    if (PIND & (1 << PIND3))
    {
        _delay_ms(10);
        return (PIND & (1 << PIND3));
    }
    return false;
}
```

Figure 1: Implémentation de isPressedDn()

partirMinuterie() de la classe Minuterie

La fonction `partirMinuterie(uint16_t)` permet de configurer la minuterie selon les besoins du projet. Plusieurs ajustements ont été faits sur les registres afin de répondre aux spécifications demandées.

Pour commencer, la fonction prend en paramètre une variable de type `uint16_t`. Cette dernière correspond à la durée de temps précédant la première interruption.

Ensuite, le mode CTC ou « Clear Timer on Compare Mode» a été configuré afin de pouvoir remettre à zéro le compteur lorsque la valeur du registre de la minuterie `TCNT1` est égal à celle du registre de comparaison `OCR1A`. Cette configuration est représentée par l'opération `(1 << WGM12)` effectuée au registre `TCCR1B`. Quant à elles, les opérations `(1 << CS12)` et `(1 << CS10)` permettent d'agir sur les registres pour configurer le *prescaler*. Celui-ci permet de diviser l'horloge par 1024 et d'atteindre le délai voulu.

Finalement, la fonction `partirMinuterie(uint16_t)` configure le registre `TIMSK1` dans le but de permettre à la minuterie de déclencher une interruption. Celle-ci est déclenchée seulement si la minuterie atteint la valeur de comparaison ou si elle se trouve en cas d'*overflow*. Ces conditions sont respectivement définies par les opérations `(1 << OCIE1A)` et `(1 << TOIE1)`.

L'implémentation de la fonction est présentée à la figure 2 suivante:

```
void partirMinuterie(uint16_t duree){
    // mode CTC du timer 1 avec horloge divisée par 1024
    // interruption après la durée spécifiée
    //initialisation du timer a zero
    TCNT1 = 0;

    OCR1A = duree;
    //on le met egal a zero etant donner que nous ne prenons pas
    //valeur provenant des PORT pour l'instant
    TCCR1A = 0;
    //set mode to 1024(divise le clock par 1024)
    // on veut modifier CS12 et CS10
    //Registre a 8 bit
    TCCR1B = (1 << WGM12) | (1 << CS12) | (1 << CS10);

    TCCR1C = 0;
    //set flags interrupt timer
    //on veut que le timer arrête quand il arrive a sa fin
    //ou lorsque nous somme arrivée a OCR1A donc on veut les deux
    //OCIE1A Fait une intéruption lorsque OCR1A = TCNT1
    //TOIE1 en cas d'overflow fait une intéruption
    TIMSK1 = (1 << OCIE1A) | (1 << TOIE1);
}
```

Figure 2: Implémentation de partirMinuterie()

ajustementPWM() de la classe PWM

La fonction `ajustementPWM()` utilise la minuterie `Timer1` du microprocesseur ATmega324PA pour générer deux signaux de sortie. Ceux-ci ont été assignés aux broches 4 et 5 du `PORTD` de l'ATmega324PA.

Pour commencer, le mode `Timer1` a été initialisé en PWM. Ce mode fait en sorte que le compteur `TCNT1` ne se remet pas zéro lorsqu'il atteint la valeur du registre `OCR1A` ou du registre `OCR1B`. Au contraire, le compteur continue jusqu'à ce qu'il atteigne son maximum, puis retourne à sa valeur initiale en prenant le chemin inverse. Pour obtenir cette configuration, l'opération (`1 << WGM10`) a été définie dans le registre `TCCR1A`.

Par la suite, le registre `TCCR1A` a été ajusté dans le but de changer le signal de sortie à chaque fois que le compteur `TCNT1` correspond aux valeurs des registres `OCR1A` et `OCR1B`. Cette configuration est démontrée par la ligne suivante : `TCCR1A = (1 << COM1A1) | (1 << COM1B1)`. Les valeurs enregistrées dans les registres `OCR1A` et `OCR1B` sont des nombres situés entre 0 et 255 et elle sont fournies par les paramètres de la fonction.

Finalement, le registre `TCCR1B` est configuré de la sorte à ce que la fréquence d'horloge du CPU soit divisée par 8 : `1 << CS11`.

L'implémentation de la fonction est présentée à la figure 3 suivante:

```
void ajustementPWM(uint8_t duree){
    // mise à un des sorties OC1A et OC1B sur comparaison
    // réussie en mode PWM 8 bits, phase correcte
    // et valeur de TOP fixe à 0xFF (mode #1 de la table 17-6
    // page 177 de la description technique du ATmega324PA)
    OCR1A = duree;
    OCR1B = duree;
    // division d'horloge par 8 - implique une frequence de PWM fixe
    TCCR1A = (1 << COM1A1) | (1 << COM1B1) | (1 << WGM10) ;
    //La division CSn2,1,0 Permet de change a frequence
    TCCR1B = (1 << CS11);
    TCCR1C = 0;
}
```

Figure 3: Implémentation de ajustementPWM()

initialisationUART() de la classe UART

Cette méthode, ainsi que la prochaine, permettent toutes deux d'utiliser le protocole de communication `RS232`. Évidemment, avant de faire une telle chose, les `U(S)ART` ont été initialisés. Pour ce faire, la communication `RS232` a été définie pour se faire à 2400 bauds, sans parité, sur 8bits et séparés par des bits d'arrêt. Les valeurs des deux premiers registres `UBRR0H` et `UBRR0L` sont fournies afin de permettre la transmission des données. Les *flags* `RXEN0` et `TXEN0` ont également été activés afin de permettre la réception et la transmission des données.

L'implémentation de la fonction est présentée à la figure 4 suivante:

```
void initialisationUART(void){
    // 2400 bauds. Nous vous donnons la valeur des deux
    // premier registres pour vous éviter des complications
    UBRR0H = 0;
    UBRR0L = 0xCF;
    // permettre la réception et la transmission par le UART0
    UCSR0A = 0;
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    // Format des trames: 8 bits, 1 stop bits, none parity
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
}
```

Figure 4: Implémentation de initialisationUART()

transmissionUART() de la classe UART

Cette fonction très simple permet la transmission de données. C'est seulement lorsque le `UDR0` passe à 1 et le *buffer* est vide que l'information est envoyée; d'où la boucle *while*. La donnée est ensuite envoyée octet par octet au port `RS232`.

L'implémentation de la fonction est présentée à la figure 5 suivante:

```
void transmissionUART(uint8_t donnee){
    //attendre tant que la donnee est entrée
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0 = donnee;
}
```

Figure 5: Implémentation de transmissionUART()

Partie 2: Décrire les modifications apportées au Makefile de départ

L'utilisation des librairies implique des modifications dans les fichiers `Makefile`.

Makefile pour notre librairie

Afin de pouvoir faire appel à du code provenant d'une librairie, et ce, à partir d'autres fichiers, il est important de pouvoir compiler cette dite librairie. Cela nécessite donc la création d'un `Makefile` qui lui est propre. De ce fait, certaines modifications doivent être apportées au `Makefile` de départ. Tout d'abord, la variable `PRJSRC` prend maintenant la fonction `wildcard` qui permet de regrouper tous les fichiers `.cpp` lors de la formation de l'exécutable.

Pour ce qui est de la deuxième modification, un second compilateur a été ajouté à la variable `CC` qui prenait `avr-gcc`. En effet, une seconde variable appelé `AR` reçoit le compilateur `avr-ar`. L'appel de ce dernier permet de compiler un fichier d'archive, et par le fait même, le code provenant d'une librairie statique.

La troisième modification concerne le retrait des commandes `AVRDUDE=avrdude` et `HEXFORMAT=ihex`. En fait, ces dernières ne sont plus pertinentes pour la compilation d'une librairie statique étant donné que celles-ci ne s'appliquent qu'au transfert allant au microcontroleur ainsi qu'aux fichiers `.hex` utilisés par le microprocesseur dans *SimulIDE*.

La quatrième modification touche à la variable `LDFLAGS` qui prend maintenant `-crs`. Les options c, r et s concernent respectivement la création de l'archive, l'insertion des fichiers membres et la production de l'index des fichiers objets.

Pour ce qui concerne la cinquième modification, le nom des cibles par défaut ont été remplacés par le nom du projet suivie d'un `.a` . En effet, le seul fichier à produire est la librairie elle-même.

La sixième modification s'applique à l'implémentation de la cible. En effet, l'implémentation fait plutôt appel au compilateur `avr-ar` représenté par la variable `AR` ainsi qu'aux nouvelles options définies par le `LDFLAGS` . Les variables `TRG` et `OBJDEPS` ont aussi été modifiées de façon indirecte, puisqu'elles font appel aux variables `PROJECTNAME` et `PRJSRC` qui ont été renommées. L'utilisation de l'option `-lm \$(LIBS)` est négligée, puisqu'il n'y a pas de librairie à lier lors de la création de la librairie elle-même.

La septième modification affecte la production des fichiers `.hex` à partir des fichiers `.elf`. Comme mentionné précédemment, la création de la librairie n'implique pas de fichiers `.hex` et `.elf`. Ceux-ci sont donc inutiles dans ce `Makefile`.

La dernière modification implique le retrait de la commande `install`. En fait, tel que mentionné ultérieurement, il n'est pas nécessaire de conserver des commandes permettant l'écriture du programme en mémoire flash du microcontroleur lors de la création d'une librairie.

Makefile pour l'executable

L'objectif de ce `Makefile` est de produire les fichiers qui sont exécutés par le microprocesseur de l'ATmega324PA. L'utilisation d'une librairie personnelle force la modification de ce `Makefile`; un chemin doit lui être ajouté afin de permettre l'accès à la librairie en question. La première modification est au niveau de la variable `PRJSRC` qui, au lieu de prendre le nom des fichiers sources, a été définie pour prendre la variable `\$(wildcard *.cpp)`. Cette variable prend en charge tous les fichiers qui finissent par `.cpp` du projet pour former l'exécutable.

La seconde modification apportée au `Makefile` est l'ajout d'un chemin lui permettant d'accéder à la librairie désirée. Pour ce faire, la variable `INC` a été ajoutée au chemin de la librairie `(ex: -I ../librairie)`. Cette variable est utile pour les *flags* qui génèrent les fichiers `.o` utilisés pour produire l'exécutable.

La dernière modification apportée est l'ajout du nom et du chemin de la librairie à la variable `LIBS.`L'option `-l` a été ajoutée avec le nom de la librairie sans le `lib`. Par exemple, si la librairie s'appelle `libCommune,`l'affection correspondante est `LIBS=-l Commune`. Par la suite, le chemin vers cette librairie a été ajouté avec l'option `-L`. En prenant le même chemin que celui fourni à l'exemple précédent `(../librairie)`, la variable `LIBS` correspondante est la suivante: `LIBS=-l Commune -L ../librairie`. Cette variable est utile pour l'implémentation de la cible.