École Polytechnique de Montréal

Département de génie informatique et génie logiciel

Cours INF1995
Projet initial en génie informatique et travail en équipe

Travail pratique 8

Makefile et production de librairies statiques

Par l'équipe

No 0948

Marc-André Gagnon
Willy Phannarat
Antoine Hoang
Djamel Aïouadj
Bartholomew Chwalek

Date:
2012-03-06

Partie 1 : Description de la librairie

Notre librairie va contenir originalement 4 fonctions globales ou classe. Nous avons listés dans cette partie leur fonctionnalité et particularités. Chaque entrée dans ce document indique le nom, la description technique de la fonction ou de la classe, la raison de sa présence dans la librairie en un court paragraphe et les particularités dans un dernier paragraphe. Il serait possible de transformer les fonctions en classe pour rendre leur utilisation plus facile plus tard. Il est aussi probable que d'autres fonctions et classes s'ajoutent au cours de la session.

1. Fonction globale : allumer DEL

Déclaration	Void libAllumerDEL (uint8_t couleur, uint8_t numero)
Paramètres	uint8_t couleur unit8_t numero
Valeur de retour	Aucune
Fichier	del.cpp

Nous avons décidé d'ajouter cette fonction à la librairie pour contrôler efficacement l'envoi d'un code de couleur sur un numéro de pin. Cela peut nous servir à des fins de débogage en plus de pouvoir simplement allumer la DEL.

Le paramètre « couleur » est prévu d'être soit à 0 pour éteint, 1 et 2 pour les deux couleurs. Le paramètre « numero » est pour indiquer sur quelle paire de pin mettre la couleur. Une particularité dans le code est prévue pour changer facilement entre le port A, B, C et D. En effet, il suffit de changer la ligne particulière « define PORTX PORTA» et de recompiler.

2. Fonction globale : ajuster signal PWM

Déclaration	Void ajusterPWM (uint8_t A, uint8_t B)
Paramètres	uint8_t A unit8_t B
Valeur de retour	Aucune
Fichier	ajusterPWM.cpp

Nous avons décidé d'ajouter cette fonction à la librairie puisque la fonction devrait être constante d'un travail à un autre. De plus, nous sommes certains de la réutiliser pour le travail final.

Les deux paramètres A et B sont envoyés à « OCR1A » et « OCR1B » respectivement. L'appel de cette fonction devra se faire après l'initialisation correcte des « timers » du microcontrôleur. Les pins D5 pour B et D6 pour A sur la carte mère sont utilisés pour les sorties.

3. Classe : Convertisseur analogique-numérique

Déclaration	class can
Methodes construction/destruction	can() ~can()
Methodes publiques	uint16_t lecture(uint8_t pos)
Methodes privees	Aucunes
Attributs	Aucuns
Fichiers	can.h can.cpp

Comme le convertisseur analogique-numérique est contrôlé entièrement par cette classe et que l'utilisation du convertisseur en est grandement simplifié, il est naturel d'ajouter cette classe à notre librairie.

Le constructeur s'occupe d'initialiser les bons registres du microcontrôleur et rend le convertisseur actif. Le destructeur, à l'opposé, rend le convertisseur inactif. La méthode lecture prend en paramètre la position de l'entrée voulue et retourne sur 10 bits le résultat de la lecture.

4. Classe : Controle de la mémoire

Déclaration	class Memoire24CXXX;
Methodes construction/destruction	Memoire24CXXX() ~Memoire24CXXX()

Methodes publiques	void init();
	
	static uint8_t choisir_banc(const uint8_t banc);
	
	uint8_t lecture(const uint16_t adresse, uint8_t *donnee);
	
	uint8_t lecture(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur);
	
	uint8_t ecriture(const uint16_t adresse, const uint8_t donnee);
	
	uint8_t ecriture(const uint16_t adresse, const uint8_t donnee, const uint8_t longueur);

Methodes privees	uint8_t ecrire_page(const uint16_t adresse, uint8_t *donnee, const uint8_t longueur);
Attributs	static uint8_t m_adresse_peripherique; const uint8_t PAGE_SIZE;
Fichiers	memoire_24.h memoire_24.cpp

La classe Memoire24CXXX nous a été fourni pour le TP6 pour contrôler la mémoire externe du ATMega16. C'est une classe indispensable à notre librairie.

Les méthodes importantes pour contrôler la mémoire dans cette classe sont lecture et ecriture. Les paramètres «adresse» indique où lire ou écrire dans la mémoire et le paramètre «donnee» indique la variable du programme main qui est impliquée. Le paramètre longueur peut être utilisé si on veut lire une plage de données.

Partie 2 : Décrire les modifications apportées au Makefile de départ

1. Introduction au makefile.

Le makefile est un fichier qui contient des spécifications de dépendances et des commandes en format texte. Make est l'utilité logicielle qui permet de construire un programme exécutable et des librairies à partir d'un code source en utilisant les logiciels nécessaires. Make utilise le makefile qui spécifie comment dériver le programme cible à partir de ses dépendances. Une utilité intéressante du système makefile réside dans la vérification faite des cibles et dépendances au niveau des temps de modifications. Make détecte donc les mises à jour de code source pour reconstruire le code nécessaire à la cible sans reconstruire inutilement les parties inchangées.
Make cherche automatique le fichier Makefile dans le répertoire courant et interprète le fichier. Dans ce fichier sont spécifiés, en format texte, les règles et les macros.

Une règle est une ligne qui débute par une ligne de dépendance. Sur cette ligne on trouve tout d'abord la cible, un deux points et une liste des dépendances (autres cibles et fichiers). Ces dépendances sont des pré-réquis à la construction de la cible. À la suite apparaissent une série de lignes de commandes espacées d'une tabulation de la marge de gauche. Ces commandes sont exécutées si l'une des dépendances est modifiée. Les lignes de commandes sont facultatives. Les lignes de commande peuvent avoir des préfixes dont : - pour ignorer les erreurs, @ pour ne pas imprimer à l'écran la commande avant de l'exécuter, + pour exécuter la commande même si make est appelé en mode sans exécution.

Un macro est considéré comme une variable quand il ne contient qu'une valeur simple, une définition textuelle, tel CC=avr-gcc. Nous adoptons la convention de nommer tous les macros en majuscules. Un macro peut être référencé pour obtenir sa valeur après sa déclaration en utilisant $(). C'est-à-dire $(CC) va retourner *avr-gcc* (tel un define en c++). Un macro peut également être composé de valeurs retournées par des commandes shell en utilisant l'opérateur ‘ (entre guillemets simples). DATE = ‘ date ‘ affectera la date courante au macro DATE. Nous pouvons explicitement spécifier des valeurs de macros lors de l'exécution de make en ajoutant NOMMACRO=valeur après la commande make. Il y a également des macros prédéfinis tels $@ qui retourne le nom de la cible courante, $< qui retourne le premier fichier de dépendance, $ ^ retourne la liste des dépendances courantes, $? Qui retourne la liste des dépendances mises à jour.

Les commentaires débutent par # et nous pouvons changer de ligne avec \ lors de l'écriture du fichier (pour éviter les lignes trop longues) sans que cela affecte la commande ou la liste de dépendances lors de l'exécution de make (aspect visuel seulement).

La brève description ci-haut devrait être assez complète pour assurer la compréhension chez un individu ayant peu de connaissances sur le système makefile sous linux.

Pour le présent projet, nous nous sommes inspirés de la lecture du makefile donné dans le cours pour la compilation des travaux pratiques et la programmation du microcontrôleur. Nous avons divisé le Makefile tout d'abord en quatre entités discrètes. Nous avons un Makefile spécifique par travail pratique dans le répertoire codeCommun/tp/tpX/ qui précise les variables spécifiques du

projet. Nous avons un Makefile_commun.txt dans le répertoire principal codeCommun/src qui précise les macros, les cibles et commandes généralisables à tous les projets. Nous avons finalement un Makefile dans le répertoire codeCommun/lib/ qui est spécifique à la construction de librairies. Et finalement, un fichier ATMega16Defs dans /codeCommun/src/ pour les variables globales du microcontrôleur. Ci-dessous, nous décrivons le contenu de ces quatres entités, leur interaction et leur raison d'être.

2. Le Makefile commun.

Voici un aperçu de notre structure de dossiers :

/codeCommun/	Le répertoire de base.
/codeCommun/lib/	La librairie.
/codeCommun/src/	Fichiers makefile commun et ATMega16Defs.
/codeCommun/tp/tpX/	Code source des différents TP et makefile spécifique. C'est le répertoire de travail.

Le makefile commun est l'entité la plus générale, elle est incluse par les makefile spécifiques (par la commande include ../../src/Makefile_commun.txt). Ce makefile contient la liste des exécutables nécessaires pour la construction de notre projet (avr-gcc), la programmation sur avr (avr-dude) et le débogage (objdump, avr-objdump, avr-size) (niveau assembleur, symboles). Nous avons également généralisé l'exécutable pour la suppression de fichiers lors du nettoyage (REMOVE = rm –f). Nous avons également inclut l'exécutable Kate comme notre environnement de développement principal et l'exécutons lors d'un make edit. Ceci nous permet d'ouvrir tous les fichiers sources dépendants d'une seule commande dans Kate.

Le makefile commun contient des *flags* nécessaires au compilateur qui sont généraux pour tous les projets dans notre cadre. Nous avons enlevé les références au code en C et en ASM puisque nous ne programmerons qu'en C++ ici. Nous rajoutons un répertoire d'inclusion pour assurer que le compilateur trouve les dépendances (principalement les en-têtes de classes et de fonctions de la librairie) en ajoutant –I$(ROOTDIR)lib comme modificateur à avr-gcc. Nous rajoutons également un répertoire de librairie avec –L$(ROOTDIR)/lib et le nom des librairies (automatiquement préfixées de lib lors de l'exécution de avr-gcc) avec –lATMega16 par exemple pour référencer à la librairie libATMega16.a contenue dans codeCommun/lib/.

Pour le compilateur avr-gcc, nous utilisons les flags décrits dans la macro CFLAGS (*compiler flags)* et CPPFLAGS (*c++ flags*) .

Avr-gcc s'occupe du pré-processing, de la compilation, de l'assemblage et de l'édition des liens. En temps normal toutes ces fonctions sont réalisées. Des options fournies lors de l'exécution peuvent modifier des évènements et variables lors de ces processus et même cesser l'exécution à un endroit spécifique. L'option –c par exemple permet d'interrompre avant l'édition des liens et produire que des .o en sortie sans produire l'exécutable. Les options de avr-gcc se comptent à plus de 100, nous n'allons donc pas toutes les énumérer mais plutôt décrire celles nécessaires à notre projet. Le niveau d'optimisation du code est aussi ajusté dans le makefile. Voici une liste des flags utilisés dans notre makefile principal.

-mmcu=atmega16: spécifier notre plateforme (afin d'utiliser le bon set d'instructions).
-Wall : affiche tous les avertissements possibles lors de la construction.
-Werror : parfois utilisé, pour s'arrêter sur tout avertissement comme une erreur.
-Os : optimisation pour la grandeur du hex, fichier de sortie (plus que -O2).

-std=c++98 : utiliser le standard 1998 ISO C++.
-Wa,*commande*,*commande*... : les commandes pour l'assembleur délimités par une virgule.

L'assembleur utilisé par avr-gcc est avr-as. Avr-gcc se charge de lui passer les options qu'on spécifie, ici avec -Wa,-ahlms=liste des fichiers lst, nous demandons à l'assembleur :

-a(sub-options...) = fichier lst. : pour allumer les options de listing.
h : inclure code source de haut niveau.
l : inclure le code assembleur.
m : inclure les expansions de macros.
s : inclure les symboles.

Notez bien que $(firstword, arguments) retourne le premier mot de *arguments* dans un makefile. Il existe d'autres commandes spécifiques pour le makefile (en fait, une panoplie, voir *http://www.makelinux.net/make3/make3-CHP-4-SECT-2*). Nous avons inclut aussi une macro ASMFLAGS pour mettre d'autres options à l'assembleur (--statistics par exemple). Des commandes pour le déboggage avec gdb on été omises mais possiblement remises pour les besoins futurs.

Nous avons aussi inclut une commande makefile lib, makefile libclean et makefile libedit pour effectuer directement du répertoire courant les différentes commandes sur la librairie (Par exemple, pour compiler la librairie directement. Le .a n'est pas sur SVN alors il faut recompiler la librairie localement). Dans le makefile est appelé alors un autre make de manière récursive en spécifiant l'option -c <repertoire> pour indiquer le répertoire dans lequel exécuter le second make.

3. Makefile spécifique.

Dans ce makefile nous inscrivons les macros spécifiques au projet qu'on désire compiler. Ce fichier réside dans le répertoire du projet. Donc, nous inscrivons le nom du projet qui sera la cible finale (l'exécutable ou fichier en sortie (hex pour avr)). Nous devons également inscrire une liste des fichiers sources nécessaires. En fin du fichier, nous inscrivons la commande d'inclusion du Makefile_commun.txt du répertoire principal $(ROOTDIR)src/Makefile_commun.txt. Nous incluons également le fichier des définitions pour le microcontrolleur $ (ROOTDIR)src/ATMega16Defs. La variable macro ROOTDIR est réside aussi dans ce makefile spécifique. Elle indique le répertoire du codeCommun relativement au répertoire courant pour pouvoir accéder convenablement au Makefile_commun.txt, ATMega16Defs et à la librairie.

4. Makefile spécifique pour la librairie.

Ce fichier makefile réside dans le répertoire de la librairie et est général pour toute la librairie. Ce fichier est moins volumineux que le makefile commun. Ce fichier permet de produire en fin de ligne une librairie d'extension .a avec avr-ar (avr-archiever) après la compilation en .o avec avr-gcc des dépendances nécessaires. Nous devons donc créer une macro avec les fichiers sources nécessaires. Un fichier en-tête (.h) doit être fourni pour chaque classe et un pour les fonctions globales. Ce fichier spécifie les signatures des fonctions dans la librairie et sera inclut par les fichiers sources qui utiliseront ces classes ou fonctions. (Il sert en pratique au compilateur à retrouver les fonctions à copier statiquement de la librairie dans l'objet produit en s'assurant que les bon paramètres, noms et types sont utilisés). Deux exécutables principaux sont utilisés : avr-gcc pour créer les objets .o à partir des sources sans créer l'exécutable final (donc avec l'option –c). Le second exécutable est avr-ar (avr-archiver) qui permet d'assembler les objets sous une librairie .a qui contient un index et les fonctions compilées (code machine spécifique à la plateforme

d’exécution). Nous devons préciser certains flags pour avr-ar, les voici :

-r : Pour remplacer dans l’archive les fonctions ayant le même nom que les nouvelles (mises-à-jour) et non de simplement les ajouter.

-s : Écrire un index ou mettre à jour l’existant.

-c : Créer l’archive.

Ces options, dans le cas de avr-ar et non avr-gcc, peuvent être concaténées sous la forme –crs puisque toutes les options de avr-ar sont à caractère unique.

Tous les fichiers cibles construits avec avr-ar doivent absolument débuter avec le préfixe lib. LEDS.a devrait plutôt se nommer libLEDS.a afin d’assurer la cohérence avec avr-gcc et une bonne interprétation des multiples fichiers dans notre projet.

5. Makefile ATMega16Defs

Ce makefile fut la dernière création. Ce qui nous manquait était de faire des définitions globales qui transcenderaient tous les fichiers sources et d'inclusion. Ces variables seraient spécifiques globalement au projet, donc des définitions globales qui restent fixes une fois établies. Nous nous disions que cela serait trop répétitif de respécifier des valeurs telles : fréquence du processeur central, port sur lequel le moteur serait branché, port avec les diodes électroluminescentes, etc. Donc, après recherche, la solution trouvée était de spécifier au compilateur une option au préprocesseur de manière globale, automatique pour tous les fichier. Ainsi donc, le flag -D pour avr-gcc répond à ce besoin. L'utilisation se fait en spécifiant -D<nom du define>=<valeur du define><type de donnee>. Comme par exemple, pour la fréquence CPU, nous juxtaposons au CPPFLAGS, -DF_CPU=8000000UL. UL signifiant *unsigned long*. Cette option nous épargne beaucoup de temps de réécriture inutile pour divers fichier où cette donnée est nécessaire.

6. Informations complémentaires

Tous les fichiers make dans la présent projet auront des cibles telles clean et edit. Clean sera une série de commandes permettant le nettoyage du répertoire (suppression des fichiers intermédiaires, exécutables, tout produit de avr-gcc ou avr-ar). Edit sera une commande qui permet de lancer l’édition avec notre environnement de développement favori mais non intégré comme Eclipse ou Visual Studio. Nous avons préféré utiliser Kate qui intègre l’ouverture simultanée de multiple fichiers avec triage, une ligne de commande interne, un soulignage spécifique au langage de programmation détecté, un affichage du numéro de ligne et d’autres possibilités utiles tout en demeurant léger.

7. En conclusion

La complexité du système linux, du compilateur gcc, du make et de svn nécessite une courbe d’apprentissage assez inclinée pour l’utilisateur novice peu familier. Cette courbe, une fois triomphalement surmontée, permet la standardisation, la compréhension mutuelle, l’ajustement des normes, l’approche itérative, l’*extreme programming*, l’intégration d’équipes de travail et surtout prévient la nuisance et la complexité inutile de tout réinventer et de refaire un travail déjà fait ou faire le même que son coéquipier. Cet agencement d’outil s’avère être une puissante source d’efficience en milieu de travail de groupe et nous est donc particulièrement utile.