Activité bare-metal — Télécom Robotics

Adressez toute faute, demande de précision ou d'ajout à Pierre Gimalac [pierre.gimalac@gmail.com]
ou directement sur le dépôt github [https://github.com/TelecomParistoc/bare-metal-activity]

Sommaire

1 Introduction

L'objectif de cette activité est d'apprendre la programmation *bare-metal*, c'est à dire sur microcontroleur, sans système d'exploitation pour nous faciliter la tâche.

En particulier il n'y a presque aucune abstraction avec le matériel à part celles fournies par le processeur, c'est à nous de charger le programme en mémoire et de gérer les interruptions.

De plus, il n'y a aucune fonction fournie et pas de gestion des processus donc tout doit se faire dans le même processus (dans un premier temps).

Une grosse partie de la programmation sur microcontroleur est de lire la documentation pour connaître le fonctionnement précis de celui-ci. Ce TP vous indiquera quels documents et quelles pages doivent être regardées pour vous éviter de lire des centaines de pages comme le pauvre auteur.

De plus, une architecture initiale est fournie pour éviter de passer trop de temps sans même arriver à exécuter un programme (voir sous-section 3.1 Dépendances et section 4 Les fichiers fournis).

Le premier objectif du TP est d'arriver à allumer une LED située sur le microcontroleur en appuyant sur l'un des boutons.

1.1 Que lire ?

Ce TP est assez long, et il n'est pas *obligatoire* de tout lire (mais je vous invite à le lire en entier tant qu'à faire), ci-dessous une liste d'un minimum à lire (parmi les sections 2, 3 et 4) :

General Purpose Input Output pour comprendre le TP

Règle générale de configuration d'un microcontroleur pour connaître les bonnes pratiques de programmation d'un microcontroleur

Dépendances pour pouvoir faire le TP

Configuration pour pouvoir faire le TP

Langage C pour arriver à écrire quoi que ce soit

Algèbre de Boole et opérations bit-à-bit en C pour configurer les registres

How to debug : GDB pour flasher le programme sur le microcontroleur et le déboguer

Documents qui décrit les fichiers de documentation fournis

Include qui décrit les headers fournis, et comment les utiliser

Makefile pour compiler le projet et lancer les différents programmes

Si vous faites le choix de ne lire que ces sections il se peut qu'il y ait des choses que vous ne compreniez pas, mais ça ne devrait pas vous empêcher de faire le TP.

2 Quelques points théoriques

2.1 La mémoire des processus

Chaque appareil dispose d'une quantité donnée de différents types de mémoire, en particulier des registres, de la mémoire vive (RAM) et de la mémoire morte (SSD, HDD).

Les exécutables sont composés de différents espaces de mémoire : text, rodata, data, bss et stack.

text : le code qui sera exécuté, certaines constantes

rodata : les variables constantes (rodata = "read-only data")

data : les variables modifiables

bss : les variables initialisées à zéro (elles sont à part car cela fait gagner de la place)

stack : la pile, l'espace de mémoire dynamique qui sert à stocker les variables locales et les informations lors des appels de fonction

Ci-dessous un tableau indiquant où est stocké chaque type de variable en C (voir sous-section 3.3 Langage C si besoin).

			.text	.rodata	.data	.bss	stack
globale	initialisée				X		
	non initialisée					X	
	const		X	X			
locale	static	initialisée			X		
		non initialisée				X	
	non static	initialisée					X
		non initialisée					X
	const		X	X			
valeur immédiate			X	X			

Le tas est un espace de mémoire dans lequel on peut allouer de la mémoire de taille variable et persistante (ne dépendant pas de l'exécution d'une fonction), il s'agit en fait d'une zone mémoire fournie par l'OS (sans trop rentrer dans les détails) donc que nous n'aborderons pas ici.

2.2 Les périphériques

Les périphériques sont "mappés en mémoire", c'est à dire qu'à chaque périphérique est associé un ensemble d'adresses virtuelles qui permettent de communiquer avec celui-ci.

Les documentations du périphérique et du microcontroleur indiquent quelles adresses sont à regarder et comment la communication et la configuration s'effectuent.

2.3 General Purpose Input Output

Les General Purpose Input Output ou GPIO sont des moyens de communication génériques situés sur les microcontroleur. Leur avantage est d'être configurable de manière logicielle (et donc dynamique) en tant qu'entrée (un registre permet alors de lire la valeur), sortie (un registre permet alors d'écrire la valeur) ou même des fonctions alternatives spéciales.

Dans le cas qui nous intéresse, chaque GPIO dispose de 16 broches pouvant individuellement être programmées. Chaque GPIO possède des adresses virtuelles qui permettent de configurer ses broches de manière très précise. Il est même possible de générer des interruptions depuis des périphériques branchés sur une broche d'un GPIO.

La documentation indique les adresses de chacun des registres ainsi que la manière de les configurer. Les pins des GPIO apparaissent dans la documentation sous la forme PXY où X est la lettre du GPIO et Y le numéro du pin (par exemple PB6 pour le pin 6 du GPIO B).

2.4 Les registres ARM

Cette partie apporte des détails sur le fonctionnement des processeurs ARM mais n'est pas du tout nécessaire pour la réalisation du TP.

Les processeurs ARM disposent de 16 registres [1], ils sont appelés r0, r1, ..., r15.

Les registres r0 à r10 sont des registres d'utilité générale, c'est à dire que l'on peut les utiliser comme on veut.

Le registre r15 est aussi appelé Program Counter (PC), il pointe sur l'instruction que l'on est en train d'exécuter.

Le registre r14 est le link register (LR), il contient l'adresse de retour lors de l'appel à une fonction.

Le registre r13 est le stack pointer (SP), il s'agit d'un pointeur vers l'extrémité de la pile.

Le registre r11 est le frame pointer (FP), il s'agit d'un pointeur vers le début de la zone de la pile attribuée à la fonction courante.

Ci-contre une comparaison avec les registres sous x86.

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Lors d'un appel de fonction il faut savoir quels registres doivent être sauvegardés et par qui (fonction appelante ou fonction appelée), il s'agit d'une convention qui a des conséquences.
Dans le cas qui nous intéresse les registres r0 à r3 sont sauvegardées par l'appelant, ces registres vont contenir les paramètres des fonctions (s'il y a plus que 4 paramètres [2] on mettra les autres dans la pile). De plus le registre r0 [3] est utilisé pour stocker la valeur de retour.

Les registres r4 à r11 sont sauvegardés par la fonction appelée, cela évite de les sauvegarder s'ils ne sont pas utilisés, ils vont typiquement contenir les variables locales.

Le registre r12 n'est sauvegardé ni par la fonction appelée ni par l'appelante, il est donc utilisé de manière temporaire et il faut penser que chaque appel de fonction peut le modifier.

2.5 Règle générale de configuration d'un microcontroleur

La règle qui suit est presque toujours vraie (il y a donc des cas où on ne l'appliquera pas mais c'est rare) : il ne faut modifier que les bits qui nous intéressent dans un registre, et laisser leur valeur aux autres pour éviter de créer des problèmes.

Pour voir comment modifier un bit particulier d'un registre, voir sous-section 3.4 Algèbre de Boole et opérations bit-à-bit en C

1. en vrai il y en a plus mais pas d'utilité courante
2. ou que certains paramètres font 64 bits
3. r0 + r1 si on renvoie 64 bits

3 Prérequis

3.1 Dépendances

- make
- toolchain arm-none-eabi
  - sur ArchLinux :
    1. arm-none-eabi-binutils
    2. arm-none-eabi-gcc
    3. arm-none-eabi-gdb
    4. arm-none-eabi-newlib
  - sur Debian :
    1. binutils-arm-none-eabi
    2. gcc-arm-none-eabi
    3. gdb-arm-none-eabi
    4. libnewlib-arm-none-eabi
- JLinkGDBServer :
  - Sur ArchLinux : jlink (AUR)
  - Autre : débrouillez vous https://www.segger.com/downloads/jlink/
- latex (texlive) et pdflatex (pour compiler le sujet)

3.2 Configuration

J'ai fait la configuration il y a plusieurs mois donc c'est compliqué de savoir quoi mettre ici.

Si vous rencontrez des erreurs ou des comportements étranges, signalez-le.

Pour pouvoir se connecter sur le microcontroleur sans être root il faut installer des règles udev, je crois qu'elles sont installées automatiquement sur ArchLinux mais je ne suis pas sûr de ce qui se passe avec d'autres OS.

Pour que GDB lise le .gdbinit fourni, il faut faire quelque chose dont je ne me rappelle pas, mais qui sera affiché la première fois que vous essaierez de lancer GDB (signalez-le à ce moment là).

Si vous êtes sur un autre Linux que Arch il faudra peut être modifier le Makefile pour donner le bon nom aux exécutables.

3.3 Langage C

Si vous connaissez le langage C vous pouvez sauter cette section, il s'agit vraiment de points basiques pour que tout le monde puissent faire le TP.

Le langage C est ce que l'on appelle un langage bas niveau (comparativement à la plupart des langages modernes [4]), ce qui signifie qu'il offre peu d'abstraction par rapport à l'assembleur et en particulier par rapport à l'utilisation de la mémoire.

3.3.1 La syntaxe

En première approximation sa syntaxe est proche de celle de Java :

4. demandez à Pierre Gimalac de vous parler du langage Rust

```c
// déclaration de fonction
int sum(int a, int b) {
    return a + b;
}
```

```c
// boucle while
while (condition) {
    // do something
}
```

```c
// boucle for
for (int i = 0; i < 10; i++) {
    // do something
}
```

```c
// fonction main et appel de fonction
int main(int argc, char *argv[]) {
    return sum(-1, 1);
}
```

```c
// condition ternaire
long x = (y == 0 ? 0 : 1);
```

```c
// exécution conditionnelle
if (condition) {
    // do something
} else {
    // do something else
}
```

```c
// déclaration de tableau non
    initialise
int tab[10];
// déclaration de tableau avec
    contenu explicite
int tab2[] = { 1, 2, 3 };
```

```c
// opérations bit-a-bit
// et bit-à-bit
int x = a & b;
// ou bit-à-bit
int x = a | b;
// xor bit-à-bit
int x = a ^ b;
// non bit-à-bit
int x = ~a;
```

3.3.2 Les types

Les types primitifs en C sont

Pour les entiers :
- char, short, int, long, long long
- on peut rajouter unsigned devant (unsigned int) pour la version non signée

Pour les flottants :
- float, double, long double

Tous ces types ont des tailles différentes et variables selon les architectures.

Pour éviter ce piège potentiel, des types supplémentaires sont définis dans *stdint.h*, qui sont int8_t, int16_t, int32_t et int64_t, la version non signée s'obtient en rajoutant un *u* devant le type (uint32_t). Le nombre correspond à la taille du type en bit. On utilisera ces types aussi souvent que possible, les seules exceptions sont le type char pour les chaînes de caractères et int qui correspond à la taille usuelle de calcul du processeur.

La partie délicate du C est en fait la partie qui diffère de Java. Il y a trois éléments sources de problèmes qui diffèrent de la syntaxe de Java.

3.3.3 Préprocesseur

Tout d'abord le préprocesseur, il s'agit d'un ensemble d'instructions qui seront exécutées à la compilation, et qui permettent de définir des valeurs et des fonctions, d'inclure des fichiers, ainsi que de faire des instructions conditionnelles.

Le préprocesseur remplace littéralement les occurrences des macros par leur définition.

```c
// on peut définir DEBUG depuis le fichier
// ou à la compilation en donnant le paramètre -DDEBUG à gcc
#define DEBUG
#include "file.h"
```

```c
#ifdef DEBUG
    // en mode debug, LOG est une fonction qui affiche des messages sur la
        sortie erreur
    #define LOG(txt) printf(stderr, "%s\n", txt)
#else
    // hors mode debug LOG n'est pas défini et les appels a LOG disparaissent
    #define LOG(txt)
#endif
```

La directive *#define* remplace directement les occurrences du texte par la valeur. Ainsi un *#define VALUE 42* remplacera toutes les occurrences de *VALUE* par la valeur 42.

3.3.4 Gestion des fichiers multiples

La gestion des fichiers se fait avec le préprocesseur et la commande "include".

Sans trop rentrer dans les détails sur le "pourquoi", pour chaque fichier *.c* on définit un fichier *.h* qui contient l'en-tête des fonctions que l'on veut exporter. On y met aussi les *#define* que l'on veut exporter, les structures et les directives préprocesseur *#include* qui sont nécessaires. On inclut ensuite le *.h* dans le *.c*, et tout se passe bien.

Pour éviter les problèmes d'inclusions mutuelles de fichiers (un fichier *a.h* qui inclue un fichier *b.h* et vice-versa), on rajoute une sécurité dans les fichiers *.h* :

```c
// on choisit un nom qui sera unique à l'échelle d'un projet
// souvent on prend le nom du fichier auquel on rajoute un _H
#ifndef FILE_NAME_H
#define FILE_NAME_H

// ...
// contenu du fichier .h
// ...

#endif
```

3.3.5 Pointeur

On arrive enfin au cœur du sujet qui donne son nom de *bas niveau* au langage C : les pointeurs et la gestion de la mémoire.

Les pointeurs sont un type de variable particulier, il s'agit d'une adresse mémoire qui pointe sur une valeur.

C'est plus compréhensible avec un exemple :

```c
int x = 42;
int *y = &x; // crée un pointeur vers x
int z = *y; // on déréférence y, ce qui récupère la valeur de x
```

y est ici un pointeur vers un entier (ce qui est indiqué par l'étoile dans le type *int **), le déréférencement de y **y* récupère la valeur pointée, donc celle de x.

Il faut faire attention : déréférencer un pointeur invalide (en particulier un pointeur nul) déclenche une erreur qui termine le programme.

Les avantages des pointeurs sont

— la taille fixe (32 ou 64 bits sur les plate-formes récentes), cela permet de passer un pointeur à la place d'une grosse structure, ce qui évite de grosses copies de mémoire

— ils permettent de modifier des variables passées en argument, contournant ainsi la limite d'une seule valeur de retour

— c'est grâce aux pointeurs que l'on utilise le tas et des buffers de taille variable [5] : la fonction malloc de la bibliothèque standard renvoie un pointeur vers une zone de mémoire de la taille demandée

3.3.6 Volatile

Le mot-clé volatile indique qu'une variable peut changer de valeur "spontanément", c'est-à-dire sans que le programme ne touche à la variable.

Le cas typique d'utilisation de ce mot-clé est pour les pointeurs sur des registres de périphériques.

```
volatile uint32_t *reg = (volatile uint32_t *)0x40000800;
// reg est un pointeur vers un registre de 32 bits
// son adresse est 0x40000800
```

L'utilisation de volatile désactive certaines optimisations du compilateur, par exemple sans ce mot-clé deux lectures successives seraient transformées en une seule lecture. De même, pour deux écritures, ou pour des enchaînements de lectures et d'écritures.

3.3.7 Petite précision sur les chaînes de caractères

Les chaînes de caractères en C sont un peu particulières : une chaîne de caractères est un tableau de caractères qui termine par le caractère nul. Il faut faire attention de ne pas écrire après la limite du tableau (ce qui est techniquement tout à fait possible). Il faut également ne pas oublier le caractère nul à la fin sinon il est impossible de savoir quand arrêter de lire...

3.4 Algèbre de Boole et opérations bit-à-bit en C

Lorsque l'on paramètre un registre, on veut en général changer un bit (ou plusieurs) particulier et laisser les autres à leur valeur. Pour cela on doit utiliser des opérations bit-à-bit qui ne vont changer que ce que l'on veut changer.

Les opérations bit-à-bit disponibles sont :

nom	opérateur	nombre d'opérandes	description
et	&	2	renvoie 1 si et seulement si les deux opérandes valent 1
ou	|	2	renvoie 0 si et seulement si les deux opérandes valent 0
xor	^	2	renvoie 1 si et seulement si une des deux opérandes vaut 1
non	~	1	renvoie 0 si l'opérande vaut 1 et inversement
left shift	<<	2	décale les bits vers la gauche (indiqué par le 2e opérande)
right shift	>>	2	décale les bits vers la droite (indiqué par le 2e opérande)

3.4.1 Mettre un bit à 1

Supposons que l'on ait une variable *var* dont on veut mettre à 1 le 10e bit, sans toucher aux autres :

```
// peu importe la valeur initiale de var
// on doit conserver tous ses bits,
// et mettre le 10e à 1
uint32_t var = 123456789;

var |= 1 << 10;
// 1 << 10 == 0b00000000000000000000010000000000
// on sait que 1 | 0 == 1 et 0 | 0 == 0 (le ou avec 0 conserve la valeur)
// donc le seul bit qui peut être modifié est le 10e
// il passe à 1, peu importe sa valeur initiale
```

5. c'est également possible dans la pile mais de manière plus limitée

3.4.2 Mettre un bit à 0

Supposons que l'on ait une variable *var* dont on veut mettre à 0 le 10e bit, sans toucher aux autres. Celui là est un peu plus complexe que le précédent.

```
// peu importe la valeur initiale de var
// on doit conserver tous ses bits,
// et mettre le 10e à 0
uint32_t var = 123456789;

var &= ~(1 << 10);
// 1 << 10 == 0b00000000000000000000010000000000
// ~(1 << 10) == 0b11111111111111111111101111111111
// on sait que 1 & 1 == 1 et 0 & 1 == 0 (le et avec 1 conserve la valeur)
// donc le seul bit qui peut être modifié est le 10e
// il passe à 0, peu importe sa valeur initiale
```

3.4.3 Lire un bit

Supposons que l'on ait une variable *var* dont on veut lire le 10e bit.

```
// on veut lire le 10e bit
uint32_t var = 123456789;

if (var & (1 << 10)) {
    // do something
}
// 1 << 10 == 0b00000000000000000000010000000000
// on sait que 1 & 1 == 1 et 0 & 1 == 0 (le et avec 1 conserve la valeur)
// donc le seul bit qui ne sera pas forcément à 0 est le 10e
// il garde sa valeur initiale

// donc var & (1 << 10) != 0 si et seulement si le 10e bit est à 1.
```

3.4.4 Aide

Pour éviter les problèmes sur les opérations bit-à-bit, des macros sont fournies :

```
#define SET_BIT(REG, BIT) ((REG) |= (BIT))

#define CLEAR_BIT(REG, BIT) ((REG) &= ~(BIT))

#define READ_BIT(REG, BIT) ((REG) & (BIT))
```

Les noms et les définitions sont assez explicites, il est conseillé de les utiliser pour éviter les erreurs stupides. On fera attention au fait que le deuxième opérande BIT correspond à 1 << 10 dans nos exemples et pas juste à 10.

3.5 La compilation du C

La compilation du C est a priori moins facile que pour la plupart des autres langages : il n'y a pas de d'outil officiel pour compiler automatiquement et on a plusieurs types de fichiers sources et plusieurs types de fichier produits.

Cependant la compilation du C n'est pas si compliquée et la comprendre permet d'être beaucoup plus efficace quand on fait de gros projets.

La première étape de compilation est le préprocesseur, chaque directive include insert le code du fichier indiqué dans le fichier source, chaque define remplace le code en fonction de la définition. On peut demander à gcc [6] de ne faire que cette étape avec l'option -E.

6. GNU C Compiler, le compilateur le plus utilisé pour le langage C

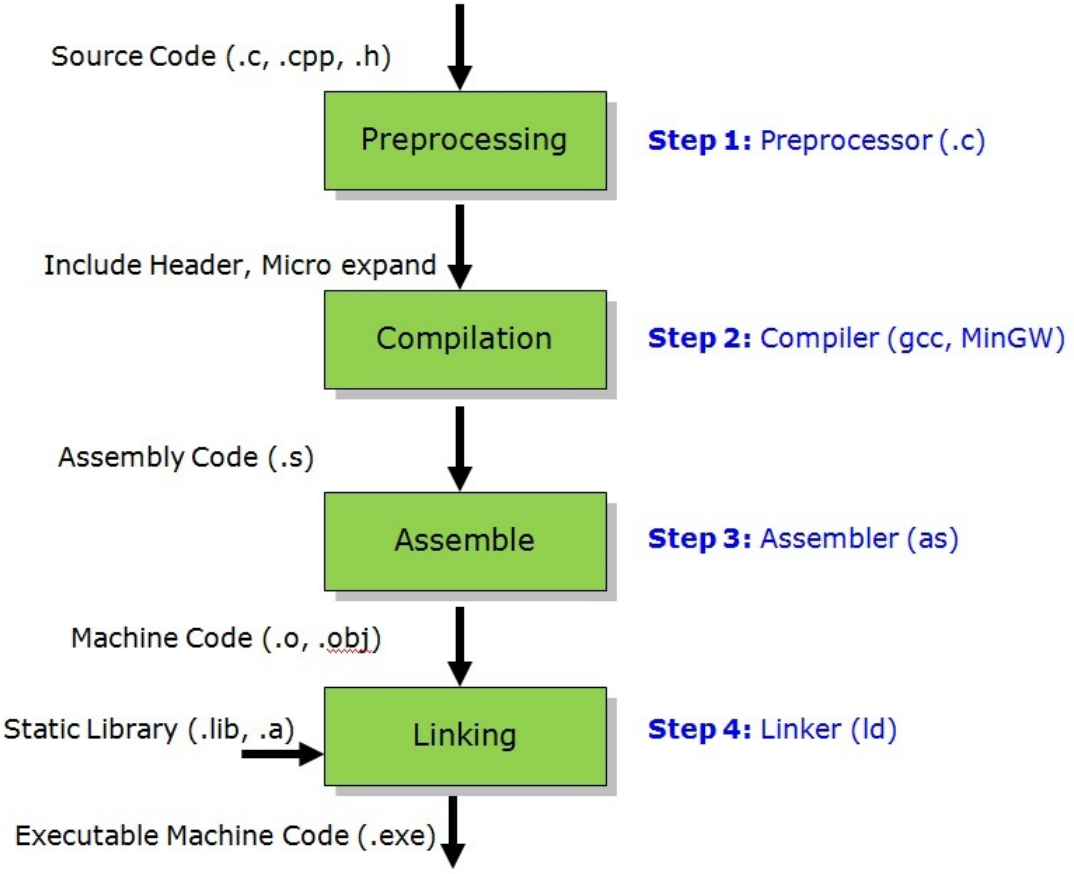
L'étape suivante est la compilation du C vers l'assembleur par le compilateur. Les symboles inconnus (situés dans d'autres fichiers) ne sont pas résolus et on suppose qu'ils seront fournis plus tard. On demande à gcc de faire cette étape avec l'option S.

Ensuite, on génère un fichier binaire objet .o depuis le fichier assembleur. On a un fichier objet par fichier source (que ce soit c ou assembleur) On demande à gcc de faire cette étape avec le paramètre -c.

Enfin, on "lie" tous les fichiers objets entre eux (ainsi que les éventuelles bibliothèques utilisées) pour produire un unique binaire.

En général les fichiers intermédiaires (autres que les fichiers sources et les fichiers objets) sont supprimés. Il s'agit d'ailleurs du comportement par défaut de gcc (sans donner de paramètre particulier, gcc produit directement l'exécutable).

Cette méthode de compilation permet de ne recompiler un objet que si ses sources sont plus récentes, on évite ainsi pas mal d'opérations, ce principe est au cœur du fonctionnement des Makefile.



3.6 How to debug : GDB

Le GNU Debugger (GDB) est un outil très puissant qui permet de déboguer un programme en mettant des pauses dans l'exécution, en observant l'état de la mémoire (y compris des registres) et en exécutant instruction par instruction un programme.

Pour un meilleur débogage il faut compiler avec l'option -g, ce qui rajoute des infos de débug dans le binaire produit.

Les commandes principales sont

b main.c :23 ajoute un breakpoint (une pause) à la ligne 23 du fichier main.c

c reprend l'exécution d'un programme, attention dans le cas d'un microcontroleur on utilisera c pour lancer l'exécution du programme

run lance l'exécution d'un programme (il est possible de donner des arguments à la commande run qui seront transmis au programme), ne pas utiliser sur un microcontroleur

mon reset permet de reset le microcontroleur, "mon" indique que la commande est transmise au microcontroleur

load charge le programme en mémoire

p var affiche une variable

x var affiche une variable en hexadécimal et le contenu de la case pointée (pratique pour un pointeur, sinon permet simplement d'avoir la valeur en hexadécimal)

Le fichier .gdbinit fourni contient quelques commandes pour vous faciliter la tâche et activer un meilleur affichage.

flash effectue un "mon reset" puis un "load"

split rajoute l'état des registres et le code assembleur dans la vue

ss rajoute l'état des registres et le code C dans la vue

armex commande pour déboguer en cas d'erreur, ça ne vous servira pas ici

Pour déboguer sur microcontroleur il faut lancer un serveur GDB qui fera le lien avec le microcontroleur et sur lequel GDB va se connecter ensuite (voir sous-section 4.3 Makefile pour savoir comment le lancer).

4 Les fichiers fournis

4.1 Documents

Le dossier *docs* contient la documentation nécessaire pour faire le TP, il y a trois documents : la datasheet des stm32f030, le manuel de référence des stm32f030 et le manuel utilisateur des stm32.

On vous indiquera dans quel document chercher chaque information à chaque étape.

De manière générale le manuel utilisateur contient des informations liées au modèle précis de microcontroleur comme la disposition matérielle.

La datasheet fournit également des informations spécifiques au modèle de microcontroleur mais plus techniques : par exemple la disposition des pins ou la liste des fonctions alternatives des GPIO.

Enfin le manuel de référence est le document plus complet sur la disposition de la mémoire, la configuration de chaque registre, etc. Il s'agit du principal document que l'on va utiliser.

4.2 Include

Le dossier *include* fournit des header C contenant beaucoup de macros très pratiques pour la programmation sur microcontroleur.

En particulier, il y a deux fichiers intéressants :

stm32f030x8.h contient les adresses de tous les registres de la doc, ainsi que des macros pour chaque bit de chaque registre et des bitmask. Il fournit également des types et des structures pour une utilisation plus lisible.

```c
// exemple d'utilisation
#include "stm32f030x8.h"

// GPIOA est une structure fournissant tous les registres du GPIOA
// GPIOA->BSRR est donc le registre BSRR du GPIOA

// Il y a une macro pour chaque bit et chaque masque
// Le format est comme ci-dessous, pour un registre lié aux GPIO
// ce sera GPIO_NOM_BIT (les noms sont les mêmes que dans la doc)
GPIOA->BSRR |= GPIO_BSRR_BS6;
```

Vous pouvez ouvrir le fichier et chercher dedans pour vérifier le nom des macros.

stm32f0xx.h contient des macros utiles pour interagir avec les registres (mentionnées dans la sous-section 3.4 Algèbre de Boole et opérations bit-à-bit en C)

```c
#define SET_BIT(REG, BIT) ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT) ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT) ((REG) & (BIT))
#define CLEAR_REG(REG) ((REG) = (0x0))
#define WRITE_REG(REG, VAL) ((REG) = (VAL))
#define READ_REG(REG) ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK) \
    WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
```

Il est conseillé d'utiliser intensivement les 3 premières et d'oublier les autres qui ne sont pas d'un grand intérêt.

Ce header importe également les sous headers en fonction du microcontroleur utilisé. Ici, il inclue donc le fichier "stm32f030x8.h" (pas besoin d'inclure les deux). Il suffit de définir STM32F030x8 (*#define STM32F030x8*) pour que cela fonctionne, et la définition est faite à l'échelle du projet dans le Makefile.

Ainsi un simple

```c
#include "stm32f0xx.h"
```

dans un fichier permet d'utiliser toutes les macros des deux fichiers.

4.3 Makefile

Le Makefile est le fichier qui permet de compiler le projet de manière automatique et efficace : il contient des règles de construction de chaque fichier ainsi que des dépendances entre fichier, il ne reconstruit pas un fichier s'il est plus récent que sa source.

4.3.1 Contenu

Les premiers symboles définis sont des variables, les plus intéressantes / celles que vous pourriez vouloir modifier (ce n'est pas nécessaire mais vous pouvez expérimenter) sont :

ASFLAGS contient les paramètres de compilation des fichiers assembleur

CPPFLAGS contient les paramètres du préprocesseur (on peut fournir des définition de macro à la compilation), on indique par exemple dans quels dossiers chercher les headers

CFLAGS contient les paramètres de compilation des fichiers c, par exemple le flag -Ox indique le niveau d'optimisation (-O0 est stupide, -O3 fait des trucs incroyables)

LDFLAGS contient les paramètres du linker

DIRS est la liste des dossiers contenant des fichiers sources vous pouvez rajouter des dossiers

EXE est le nom de l'exécutable (avec des extensions .elf et .bin selon ce que vous voulez)

Tout fonctionne de manière automatique : vous pouvez rajouter des fichiers .c ou .s (assembleur) dans n'importe quel dossier de DIRS et ils seront compilés et liés automatiquement.

4.3.2 Utilisation

La commande associée au Makefile est “make”. Il faut faire “make + *nom de la règle*” pour construire la règle voulue, et un simple “make” construit la première règle (qui est ici le fichier binaire).

make va construire le binaire

make gdb_server va lancer le serveur pour la connexion de gdb

make gdb va lancer gdb et le connecter au serveur (donc faire make gdb_server avant)

make tp va produire le pdf de l'énoncé

make clean va nettoyer les fichiers produits, sauf l'exécutable et le pdf de l'énoncé

make clean-all va nettoyer tous les fichiers produits, y compris l'exécutable et le pdf

make re fait “make clean-all” puis “make”

Attention il faut se situer dans le dossier parent du TP (celui contenant le Makefile) pour utiliser toutes ces commandes.

4.4 ld_ram.lds

Ce fichier indique au linker de quelle manière est organisée la mémoire du microcontroleur et de l'exécutable que l'on veut produire.

On peut voir les deux zones de mémoire (RAM et flash) avec leurs positions et tailles, ainsi que les différentes parties de l'exécutable et leurs positions en mémoire.

4.5 lib/crt0.s

Il s'agit du programme qui est exécuté initialement, on y définit une fonction _start qui sera le point d'entrée dans le programme, on écrit dans le registre sp la valeur initiale de la pile, on appelle init_bss qui initialise le bss (les variables initialisées à 0) et enfin on appelle la fonction main. On peut observer que l'on boucle indéfiniment après la fonction main.

4.6 lib/init.c

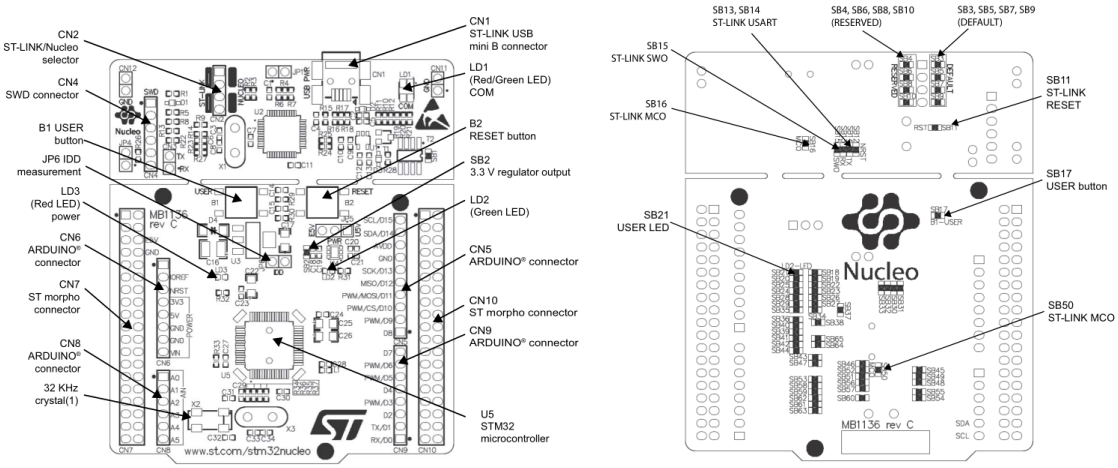
Ce fichier contient la fonction init_bss qui initialise la bss (là où sont stockées les variables globales / static initialisées à 0) à 0.

4.7 main.c

Il s'agit du fichier principal, il contient la fonction main qui est (de votre point de vue) le point d'entrée du programme.

5 Le vif du sujet

L'objectif de ce TP est d'allumer une LED lorsque l'on appuie sur un bouton du microcontroleur.



En particulier, on veut allumer LD2 qui est une led verte située au milieu légèrement à droite sur le dessus du microcontroleur (voir schémas) en appuyant sur le bouton B1, qui est le bouton bleu du microcontroleur (le noir est le bouton RESET).

5.1 Organisation du TP

On essaierai de répartir nos fonctions dans des fichiers différents pour ne pas tout mélanger, et d'importer les bons fichier là où il le faut.

On créera ainsi des fichiers led.c et led.h, ainsi que button.c et button.h.

5.2 Allumage de la led

On va commencer par essayer d'allumer la LED.

La première étape pour cela est d'aller chercher dans le manuel utilisateur à quel broche de quel GPIO est connectée la led.

Une fois que vous avez trouvé, on va voir comment configurer ce pin pour l'utiliser comme une sortie.

5.2.1 Configuration de la broche du GPIO

Ecrire une fonction led_init qui fait la configuration ci-dessous.

1. Pour commencer on doit allumer l'horloge du GPIO, voir la section RCC (reset and clock control) dans le manuel de référence, en particulier RCC_AHBENR.
2. Ensuite on doit indiquer que l'on compte utiliser notre pin comme une sortie, voir la section GPIO dans le manuel de référence et en particulier GPIO_MODER.

5.2.2 Allumer / Éteindre

Enfin on veut allumer la led et donc mettre la sortie à l'état haut : il y a pour cela plusieurs registres, on vous laisse lire le reste de la section GPIO du manuel de référence.

Vous pouvez la survoler, on voit assez vite à quoi sert chaque registre, pas besoin d'y passer trop de temps.

Écrivez dans led.c une fonction led_on et une fonction led_off qui allument et éteignent respectivement la led.

Depuis le main appelez led_init et led_on et vérifiez que la led s'allume. Appelez ensuite led_off et vérifiez que la led est éteinte [7].

5.3 Le bouton

On va refaire la même chose que dans la section précédente, mais cette fois avec le bouton.

Commencez par chercher dans le manuel utilisateur à quelle broche de quel GPIO est connecté le bouton, puis effectuez la configuration de la même manière que précédemment (mais en mettant la broche en entrée cette fois) dans une fonction button_init.

Enfin, cherchez dans le manuel de référence quel registre permet de lire la valeur du pin, et écrire une fonction button_pressed qui renvoie la valeur du pin.

On fera attention au fait que le bouton est inversé par rapport à ce à quoi on peut s'attendre : la valeur est 0 lorsque l'on appuie et 1 sinon.

5.4 Conclusion

Dans le main, débrouillez-vous pour que la led soit allumée lorsque l'on appuie sur le bouton et éteinte sinon.

7. en théorie tout va trop vite, on ne devrait même pas voir la led s'allumer