

TD : Mise en œuvre d'une application embarquée

Objectifs

Le but de ce TD est :

- de vous familiariser avec la cross-compilation
- de vous donner une première expérience de développement sur carte nue (bare-metal).

Ce TD n'est pas simple : contrairement à ce dont vous avez l'habitude, ici vous allez devoir développer dans un environnement où rien n'est fait. Vous n'aurez à votre disposition ni OS, ni bibliothèque C, ni code de boot, ni rien... Ce sera à vous de tout faire !

C'est pourquoi le TD sera progressif : vous allez progresser étape par étape, et construire petit à petit un programme complet. N'hésitez à pas solliciter les encadrants (de visu, ou par mail en dehors des heures de TD).

Bon courage !

Introduction

La carte, le processeur

Vous allez développer sur des cartes FRDM-KL46Z. Ces cartes sont basées sur un microcontrôleur de Freescale, le MKL46Z256VLL4. Un microcontrôleur est un circuit intégré rassemblant un coeur de processeur ainsi que plusieurs périphériques usuels. En l'occurrence le coeur de processeur est un ARM Cortex M0+ associé à plusieurs périphériques dont :

- des entrées-sorties basiques, appelées GPIO (Global Purpose Input Ouput)
- des bus I2C
- des bus SPI
- des ports série (UART)
- des générateurs PWM
- des convertisseurs analogiques-numériques
- etc.

Cette cartes inclut plusieurs capteurs, dont :

- un accéléromètre
- un magnétomètre
- deux LED
- un afficheur LCD
- des boutons
- un slider capacitif
- des connecteurs d'expansion pour brancher des cartes additionnelles

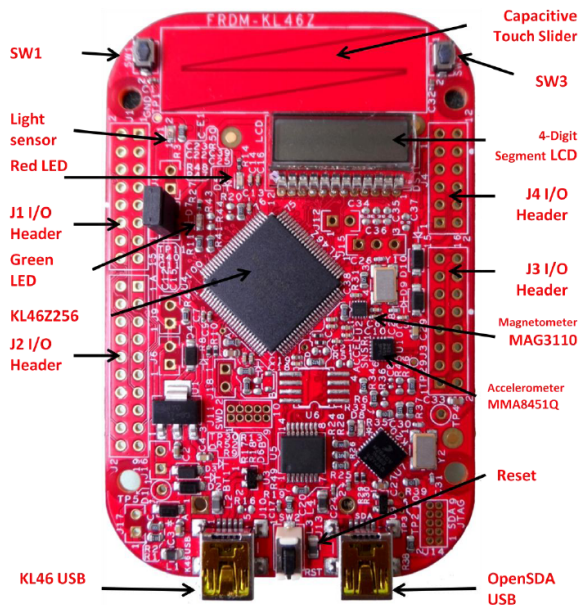


Figure 2. FRDM-KL46Z main components placement.

Vous trouverez plusieurs documents importants en bas de cette page :

- le [manuel de référence du microcontrôleur](#)
- un [quick start guide](#) des périphériques, qui vous explique en quelques phrases comment programmer la carte en bare-metal et mettre en marche les périphériques. Il vous sera utile dès le chapitre sur les GPIO !
- le [manuel d'utilisation de la carte](#), et [son schéma électronique](#). Un conseil : téléchargez au moins le manuel de référence et stockez-le dans votre dépôt git, ce sera votre bible pour les 4 semaines prochaines...

Branchement de la carte

Pour alimenter et déboguer la carte, il suffit de brancher un câble mini-USB sur la prise OpenSDA de la carte et sur le PC. C'est tout !

Mapping mémoire

Le mapping mémoire du processeur est disponible en pages 113 et suivantes du [reference manuel du processeur](#). Pas la peine de tout lire, pour l'instant retenez juste les emplacements et tailles des éléments suivants :

- flash : adresse de début = 0x00000000, taille = 256k
- RAM : elle est séparée en deux blocs contigus :
 - RAML : début = 0x1ffe000, taille = 8k
 - RAMH : début = 0x20000000, taille = 24k

Dans un premier temps, le programme qu'on écrira sera logé en RAM. Pour des raisons qu'on verra plus tard, nous allons le loger dans RAMH. RAML sera réservé pour autre chose (patience petit Padawan).

Outils de debug

La carte de développement intègre une sonde JTAG, ce qui est bien pratique. Elle est disponible sur le connecteur de la figure ci-dessus appelé "OpenSDA USB". Plus exactement, ce connecteur USB vous donne accès à deux choses :

- la sonde JTAG intégrée
- un port série sur USB, vu sous Linux comme `/dev/ttyACM0` (115200 baud, pas de parité, 8 bits, pas de contrôle de flux)

Comme pour toutes les sondes JTAG, nous devrons utiliser un "driver" pour faire le pont entre gdb et la

sonde. Ce driver s'appelle JLinkGDBServer. Il est disponible [ici](#) (pour ceux qui souhaitent travailler sur leur portable). Ce "driver" est installé sur toutes les stations de la A406.

Pour lancer / débbugger un programme, la procédure est la suivante :

1. lancer le driver : JLinkGDBServer -if swd -device "MKL46Z256xxx4" -LocalhostOnly
2. dans un autre terminal, lancer le cross-débugger : arm-none-eabi-gdb xxx.elf (remplacer xxx.elf par le nom du programme que vous voulez débbugger)
3. dire à gdb qu'on fait du débbug distant et comment communiquer avec le driver de sondes : target ext :2331
4. transmettre au driver de sonde la commande disant que le processeur est en mode little-endian : mon endian little
5. transférer le programme sur la carte : load
6. lancer l'exécution : cont
7. débbugger de façon normale (si pour avancer d'une instruction assembleur, etc.)

Plutôt que de taper à chaque fois toutes ces commandes, il est préférable de se créer un fichier .gdbinit, qui sera exécuté à chaque lancement de arm-none-eabi-gdb dans le répertoire où ce fichier se trouve. Nous vous conseillons le .gdbinit suivant :

```
target ext :2331
mon endian little
mon halt

# User interface with asm, regs and cmd windows
define split
    layout split
    layout asm
    layout regs
    focus cmd
end
```

Attention : pour des raison de sécurité, par défaut les .gdbinit ne sont pas lus. Lisez attentivement le message de arm-none-eabi-gdb après avoir créé le .gdbinit pour voir ce que vous avez à faire pour qu'il soit exécuté.

Une fois que vous avez ce .gdbinit en place (pensez à le committer), tout ce que vous avez à faire est :

1. load, pour charger le programme sur le processeur
2. si vous avez défini un ENTRY point dans votre script de link, gdb positionne automatiquement le PC à la bonne valeur
3. cont pour démarrer le programme.





Conseils :

- dans votre Makefile, définissez une cible virtuelle qui lance JLinkGDBServer avec les bonnes options
- à partir de maintenant vous devriez avoir, dans un répertoire nommé TD, un Makefile qui permet de lancer JLinkGDBServer, ainsi qu'un .gdbinit.

L'environnement de débbug est en place, on peut maintenant entrer dans le vif du sujet !

Au fait, vous avez pensé à committer / pusher combien de fois ? Une fois que vous pensez avoir terminé cette partie, mettez le tag git INTRO sur le commit de fin, et pushez le par git push --tags.

Attention : lors de la correction du TP, seuls les commit avec les tags précisés ici seront examinés.

Fichier attaché	Taille
 Manuel de référence du processeur MKL46Z256VLL4	5.16 Mo
 Manuel d'utilisation de la carte	1.77 Mo
 Schéma électronique de la carte	173.16 Ko
 klqrug.pdf	2.03 Mo

Création d'un exécutable

Avant de commencer à débbugger, on va déjà créer un exécutable minimal qui nous permettra de vérifier qu'on peut :

- télécharger un programme sur la carte dans une zone adaptée (en RAM dans un premier temps)
- lancer ce programme pas à pas (instruction assembleur par instruction assembleur)
- bref, qu'on sait générer un exécutable correct et le débbugger.

Pour cela nous allons procéder en plusieurs temps :

1. d'abord la création d'un script de link minimal qui assurera que l'exécutable est logé aux bonnes adresses
2. écriture d'un programme minimal (un main qui fait une boucle infinie) et on le testera
3. écriture d'un programme un peu plus complexe, faisant appel à la pile, qu'on testera
4. une fois arrivés là, on aura de quoi commencer à programmer les périphériques !

Mapping mémoire

On rappelle que le mapping mémoire du processeur est disponible en pages 113 et suivantes du [reference manual du processeur](#). L'emplacement des zones qui nous intéressent est rappelé ci-dessous :

- flash : adresse de début = 0x00000000, taille = 256k
- RAM : elle est séparée en deux blocs contigus :
 - RAML : adresse de début = 0x1ffe000, taille = 8k
 - RAMH : adresses de début = 0x20000000, taille = 24k

Dans un premier temps, le programme qu'on écrira sera logé en RAM. Plus précisément, le programme (code + données) sera dans RAMH et la pile dans RAML.

L'avantage de cette façon de faire, est qu'un débordement de pile se traduira par un accès dans une zone réservée qui passera le processeur en mode erreur. On pourra donc tout de suite identifier le problème. À contrario, si on avait mis la pile tout en haut de la RAM (à la fin de RAMH), un débordement de pile se serait traduit par un écrasement de la zone de données (bss ou data), ce qui ne produit un comportement erratique (crash) que longtemps après. Et à débbugger, c'est très difficile !

Linker script

Layout mémoire

Créez un fichier appelé `ld_ram.lds`, dans lequel à l'aide de la directive `MEMORY` vous définirez les différentes régions de mémoire disponibles dans le processeur.

Création des sections

Dans votre script de link, à l'aide de la directive `SECTION`, créez les différentes sections dont vous aurez besoin. Pour l'instant on partira du principe que l'exécutable réside entièrement en RAM, il n'y a donc pas de recopies à faire. Autrement dit, pas besoin de spécifier les LMA des sections, la flash n'étant pas utilisée.

On mettra en premier la section `.text`, puis la section `.rodata`, puis `.data`, puis le BSS / COMMON.

Point d'entrée

Le programme est destiné à être exécuté directement par le processeur, sans passer par un loader ELF. Il n'y aurait donc pas besoin de spécifier un point d'entrée.

Mais : le processeur est câblé pour booter à l'adresse 0, qui se situe en flash. Or on voudrait qu'il boote directement sur notre programme en RAM. Pour cela, on pourrait flasher à l'adresse 0 un petit bout de code faisant juste un saut au début de RAMH. Mais on va plutôt exploiter une caractéristique bien pratique de `gdb` : lorsqu'on lui demande de transférer un exécutable ELF sur une carte, si celui-ci comporte un point d'entrée, alors `gdb` positionne automatiquement le PC sur ce point d'entrée. On n'a plus après qu'à faire "continue", et tout se passe comme si on avait booté directement depuis la RAM.

À l'aide de la directive `ENTRY`, définissez donc un point d'entrée, par exemple sur la fonction `main`.

Programme de base

Écriture du programme de test

Écrivez en C un programme test le plus simple possible : une fonction `main` qui fait une boucle infinie.

Compilation du programme

Pour compiler votre programme, il faut indiquer au compilateur que le processeur cible est un cortex-M0+ en mode thumb. Ce qui donne une commande du style : `arm-none-eabi-gcc -c -g -O1 -mcpu=cortex-m0plus -mthumb main.c -o main.o`

Pour linker votre programme, la commande est du style : `arm-none-eabi-gcc -T ld_ram.lds main.o -o main`

Bien entendu, il est **hors de question** de taper ces lignes à la main. Créez donc un Makefile, en utilisant les variables et règles implicites adéquates !

Vérification du link

Avant de charger votre programme sur la carte, **il est impératif de vérifier qu'il a bien été généré correctement.**

Pour cela utilisez objdump pour vérifier que

- le point d'entrée est bien en 0x20000000
- tout l'exécutable est bien logé dans RAMH (à partir de 0x20000000).

Tant que ce n'est pas le cas, ne passez surtout pas à la suite, vous risqueriez de programmer la flash et passer la carte dans un état irrécupérable !

Test du programme de base

1. Lancez le driver de sonde : `make startgdbserver` (cf. page précédente)
2. Dans un autre terminal, lancez le débogueur en lui passant en argument le fichier ELF généré :
`arm-none-eabi-gdb main`
3. Chargez le programme : `load`
4. Mettez-vous en affichage "registres + code assembleur + fenêtre de commande" : `split`
5. Vérifiez que le PC est positionné à la bonne valeur
6. Exécutez votre programme pas à pas (instruction assembleur par instruction assembleur) : `si`

Si tout se passe bien, tant mieux ! Sinon, recherchez la cause de l'erreur en examinant à chaque fois les registres du processeur et en vérifiant que ce qui est exécuté l'est correctement.

Pour sortir de gdb, tapez `quit` OU `control-d`.

Attention : si vous terminal se retrouve dans un mode bizarre après la sortie de gdb, ne paniquez pas ! Tapez `reset`, et tout devrait rentrer dans l'ordre :)

Programme un plus évolué

Fibonacci

Écrivez une fonction récursive `int fibo(int n)`, qui calcule le n-ième nombre de la suite de Fibonacci.

Modifier `main` pour qu'il ne fasse que renvoyer `fibo(8)`.

Compilez, et vérifiez à coup d'objdump que le programme est bien logé aux bonnes adresses.

Testez-le en vrai. Que se passe-t-il, et pourquoi ?

Indice crucial : exécutez le programme instruction assembleur par instruction assembleur et vérifiez à chaque étape que tout s'est bien passé :

- pour toute opération arithmétique / logique, vérifiez le résultat en examinant les registres
- pour tout accès à la mémoire, regardez le contenu de la mémoire **avant** l'instruction puis **après** l'instruction et vérifiez que c'est cohérent

Correction des choses

Vous venez de constater qu'il manque quelque chose de crucial avant `main` pour que les choses s'exécutent correctement.

Créez donc le fichier qui va bien, qui se chargera de mettre en place un environnement d'exécution correct pour le code C.

Compilez, et vérifiez à coup d'objdump que le programme est bien logé aux bonnes adresses.

Testez votre programme, qui doit à présent s'exécuter correctement.

Attention :

- pour compiler de l'assembleur, on utilisera les flags suivants : `ASFLAGS = -g -mcpu=cortex-m0plus`
- de plus, il est nécessaire de mettre en première ligne de votre fichier assembleur la directive suivante : `.thumb`
- contrairement au C, en assembleur les symboles sont privés par défaut. Pour les exporter (ce façon à ce qu'ils soient visibles depuis le C ou le linker script), il faut les déclarer `.global`.

Exemple :

```
.thumb
.global _start
_start:
    blablabla
```

Initialisation du BSS

Dans le fichier que vous venez de créer, appelez avant `main` une procédure `void init_bss()` (écrite en C dans un fichier appelé `init.c`), qui se chargera d'initialiser le BSS à zéro, en s'aidant de symboles exportés depuis le script de link.

Testez cette procédure en déclarant des variables qui seront stockées dans le BSS et en vérifiant qu'une fois arrivé à `main`, elles valent bien 0.

Conclusion

Vous avez maintenant un environnement d'exécution correct pour exécuter du C. Nous pouvons passer à la mise en marche du premier périphérique, le plus basique : l'allumage d'une LED !

Au fait, vous avez pensé à committer / pusher combien de fois ? :)
Mettez le tag `LDSOCKET` sur le commit prêt à être corrigé et poussez le.

Les GPIO**Introduction**

Le premier périphérique qu'on met en marche traditionnellement quand on démarre une carte en bare-metal est le contrôleur de GPIO, de façon à pouvoir allumer / éteindre une LED. Les GPIO sont des broches du processeur qu'on peut configurer à volonté en entrée ou en sortie. Lorsqu'elles sont en entrée, on peut lire leur état dans un registre spécial. Lorsqu'elles sont en sortie, on peut les mettre à l'état haut ou bas, en écrivant dans un registre spécial.

La plupart des GPIO peuvent avoir des configurations supplémentaires :

- on peut leur demander de générer une interruption si le signal qui arrive dessus change d'état, ou fait un front montant ou un front descendant
- on peut configurer l'intensité qu'elles peuvent débiter (ce qu'on appelle le `slew-rate`)
- activer des résistances de pull-up / pull-down,
- etc.

Multiplexage des fonctionnalités des broches

Un microcontrôleur n'a qu'un nombre limité de broches. Pas assez pour pouvoir utiliser tous les périphériques internes. Chaque broche peut donc être configurée pour être utilisée comme GPIO, port série, port I2C, port SPI, pour USB, etc. Les broches sont aussi groupées par "PORTS" (groupe de 32 broches) pour faciliter leur repère. Par exemple, la broche `PTD5` est la broche 5 du port D. Sachant que notre processeur dispose de cinq ports : `PORTA` à `PORTE`.

Certains contrôleurs permettent à chaque broche d'être configurée pour servir à n'importe quel périphérique (exemple : `nRF51822`), mais ce n'est pas le cas du nôtre. La section 10.3.1 du [reference manual](#) (page 171 et suivantes) liste, pour chaque broches, les fonctions disponibles dessus.

La fonction qui sera utilisée sur une broche particulière est réglée dans le registre `PORTx_PCRn` (cf. page 193), dans les bits 8, 9 et 10 appelés `MUX`. Par exemple, pour utiliser la broche `PTD5` comme une GPIO, on écrira `001` dans les bits [10:8] de `PORTD_PCR5`. Pour utiliser la même broche comme le fil de transmission TX de l'UART2, on écrira `011` dans les mêmes bits du même registre.

Pour l'instant, nous allons piloter les deux LED verte et rouge de la carte, qui sont respectivement branchées sur les broches `PTD5` (led verte) et `PTE29` (led rouge). Pour allumer les LED, il faut mettre la broche correspondante à l'état bas, et pour les éteindre la mettre à l'état haut.

Économie d'énergie : clock gating

Les Cortex-M0+ sont des processeurs faits spécialement pour les applications à très faible consommation. Pour réduire la consommation au minimum, l'utilisateur a la possibilité d'arrêter l'horloge de chaque périphérique : sa consommation devient donc nulle. Par défaut, tous les périphériques ont leur horloge arrêtée. Si on veut en utiliser un, il faut donc d'abord activer son horloge.

Pour les GPIO, l'horloge correspondante est celle du PORT à laquelle la broche appartient. Les horloges sont activées dans le module SIM, dans l'un des registres `SIM_SCGCx` (voir page 216 du reference manual).

Au travail !

On va déjà commencer par organiser les fichiers de façon propre : chaque périphérique aura son propre fichiers source, avec le header associé dans lequel il spécifie les fonctions / variables à exporter. Pour les LED, on créera donc les fichiers `led.c` et `led.h`.

Pour chaque périphérique, on créera une fonction d'initialisation (`void led_init(void)` pour les LED, etc.) qui se chargera d'initialiser le périphérique : activation de l'horloge associée, configuration diverse, etc.

Initialisation des GPIO LED

Écrivez une fonction `void led_init()`, qui se charge de :

- activer les horloges des ports D et E dans le registre `SIM_SCGCx`.
- configure les broches PTD5 et PTE29 en GPIO : registres `PORTD_PCR5` et `PORTE_PCR29`.
- configure les broches PTD5 et PTE29 en sorties : registres `GPIOD_PDDR` et `GPIOE_PDDR`.
- allume les deux LEDs : registres `GPIOD_PCOR` par exemple, et `GPIOE_PCOR`.

Écrivez pour chacune des LED, les fonctions suivantes :

- `led_g_on()` / `led_r_on()`
- `led_r_off()` / `led_g_off()`
- `led_r_toggle()` / `led_g_toggle()`

Indice : la lecture de la documentation des registres `GPIOx_PSOR` / `PCOR` / `PTOR` pourra s'avérer utile !

Depuis, le `main`, appelez la fonction `led_init()`, puis faites clignoter chaque LED en faisant des boucles d'attente active ainsi (attention, c'est sale) :

```
for (int i=0; i<....; i++)
    asm volatile("nop");
```

On verra plus tard comment implémenter des délais de façon plus précise et (beaucoup) plus propre.

Conclusion

Félicitation, vous avez mis en marche votre premier périphérique. Passons maintenant à quelque chose de plus complexe !

Mais auparavant, mettez le tag `GPIO` sur le commit prêt à être corrigé et poussez-le !

Horloges, PLL & Co. - Part I

Introduction

Lorsque l'on démarre un processeur, une des premières choses à faire est de configurer les différents éléments intervenant dans la génération des horloges afin d'obtenir les différentes fréquences d'horloges désirées.

Dans le processeur utilisé sur vos cartes de TP, le coeur et les périphériques utilisent des horloges différentes. Le schéma de la documentation du processeur page 124 détaille les différentes sources d'horloge et les différents composants intervenant dans la génération des horloges utilisées au sein de la puce.

Comme vous pouvez le constater, c'est un sujet complexe. Mais pour la suite du TP, nous allons avoir besoin d'avoir des horloges précises (notamment pour l'UART et les timers).

Pour l'instant nous allons vous donner un fichier objet contenant une fonction `void clocks_init(void)`, qui initialise les horloges ainsi :

- `MCGOUTCLK` : 48MHz

- MGCPLLCLK/2 : 24MHz
- Core clock, Platform clock et System clock : 48 MHz
- Bus clock, Flash clock : 24 MHz

Récupérez-donc le fichier [clocks.o.gz](#), décompressez-le, puis committez-le dans votre dépôt (oui, pour une fois vous avez le droit de committer un objet ! :)).

Appelez la fonction `clocks_init()` au début de votre `main`. Puis passez à la suite. Ce n'est qu'à la fin du TP que vous ferez vous-même [l'initialisation des horloges](#).

Fichier attaché Taille

 [clocks.o.gz](#) 1.19 Ko

Le port série (UART)

Après l'allumage d'une LED, l'un des premiers périphériques qu'on utilise dans un système embarqué est le port série, qui permet de disposer d'une console et donc de fonction "à la" `printf`.

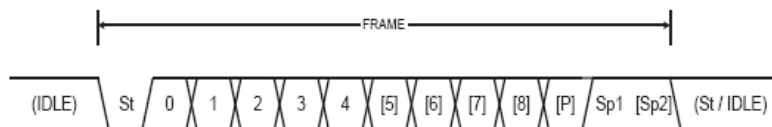
UART : le protocole

Le port série (UART) est un port de communication qui était très répandu sur les PC jusqu'il y a peu. Il est par contre présent sur la totalité des systèmes embarqués. C'est un protocole série, qui permet de transmettre des trames d'octets les uns à la suite des autres. Pour que la communication fonctionne, les deux dispositifs désirant communiquer ensemble doivent se mettre d'accord sur un certain nombre de points :

- le nombre de bits par trame : généralement 8, mais ça peut aller de 5 à 10.
- la durée de chaque bit : pour une communication lente mais sûre, on utilise traditionnellement une vitesse de 9600bps, pour une transmission rapide on transmet à 115200bps. D'autres vitesses sont possibles (de 1200bps à 3Mbps).
- la présence ou non d'un bit de parité (paire ou impaire), permettant de détecter certaines erreurs de transmission.
- la durée du bit de stop (le bit de fin de trame) : généralement 1.

L'état de repos est l'état haut (VCC).

Le format standard d'une trame au format 8N1 115200 (8 bits de données, pas de parité, 1 bit de stop, 115200bps, soit 115200 8N1 en abrégé) est celle-ci :



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.

En pratique

Les PC portable ne sont plus équipés de port série, et les PC nécessitent un adaptateur de tension (pour des raisons historiques, l'état haut correspond à une tension de -9 à -15V, et l'état bas à une tension de +9 à +15V), ce qui n'est pas pratique.

Heureusement, la sonde JTAG intégré à la carte de développement intègre un pont UART0 / USB. Lors du branchement du câble USB sur le PC pour débayer la carte, deux périphériques sont automatiquement créés par Linux :

- celui qui permet de communiquer avec le driver de sonde et gdb
- un port série "virtuel", avec lequel tout se passe comme si on était directement branché sur l'UART0 du microcontrôleur. Ce port série, sous Linux, a pour nom `/dev/ttyACM0`.

Attention : pour ceux qui ont un PC personnel sous Ubuntu, un programme est installé par défaut qui empêche la communication avec ce port série : `modem-manager`. Pensez à le supprimer en tapant `sudo apt-get purge modemmanager`.

Pour communiquer avec le port série depuis le PC, plusieurs programmes existent dont :

- minicom : purement textuel, un peu buggé, installé sur toutes les stations des salles de TP
- putty : interface graphique, pratique, installé sur toutes les stations des salles de TP
- cutecom : interface graphique plus épurée que putty, pratique lui aussi.
- tio : joli, pratique, mais pas installé sur les stations des salles de TP
- kermi : purement textuel, fiable, mais peu ergonomique
- cu, tip, screen, etc.

Nous vous conseillons tio, putty ou cutecom. Lorsque vous les lancez, réglez les sur :

- serial : /dev/ttyACM0
- vitesse : 115200
- 8 bits de données
- pas de contrôle de flux (ni logiciel ni matériel)
- 1 bit de stop
- pas de parité

Une fois lancé, chaque caractère tapé est envoyé à l'UART du microcontrôleur, et vice-versa.

Programmation de l'UART du processeur

Configuration de l'UART0

Vous allez devoir envoyer des caractères sur le port série UART0 du processeur. Pour cela, il va falloir le piloter manuellement (pas de libC, pas de printf !). Les registres clefs sont indiqués ci-dessous (cf. pages 745 et suivantes du [reference manual](#) et pages 77 et suivantes du [quick start guide](#)).

Dans un fichier uart.c, écrivez une fonction void uart_init() qui se charge de :

- activer l'horloge du port série UART0, à partir de MCGPLLCLK/2 : registres SIM_SOPT2 et SIM_SCGC4.
- configurer l'oversampling à la valeur la plus grande possible, sans dépasser 3% d'erreur sur la vitesse résultante de l'UART (registre UART0_C4).
- configurer la vitesse du port série à 115200 bauds : registres UART0_BDH et UART0_BDL. Pour savoir quoi mettre dedans, cf le quick start guide.
- configurer l'UART0 en 8 N 1 (à vous de trouver les bons registres)
- passer les broches RX et TX du port A (à vous de trouver lesquelles) en mode UART (ne pas oublier d'activer l'horloge du PORTA par la même occasion).
- activer le transmetteur et le récepteur.

Attention : cette configuration doit être faite alors que l'UART0 est désactivé !

Envoi et réception

Écrivez dans l'ordre les fonctions suivantes :

- void uart_putchar(char c), qui attend que l'UART soit prêt à transmettre quelque chose, puis lui demande de l'envoyer (registres UART0_S1 et UART0_D)
- unsigned char uart_getchar(), qui attend que l'UART ait reçu un caractère puis le retourne (registres UART0_S1 et UART0_D)
- void uart_puts(const char *s), qui fait la même chose que puts sous Linux.
- void uart_gets(char *s, int size), qui fait la même chose que fgets sous Linux (sauf pour le EOF, qui n'a pas de sens pour un port série)

Attention :

1. testez vos fonction à chaque étape (= n'attendez pas d'avoir tout écrit pour tester).
2. si quelque chose ne marche pas, débinez instruction [assembleur](#) par instruction [assembleur](#).

Test de votre UART

Pour tester le fonctionnement correct de votre UART, téléchargez le programme [checksum.py](#) en bas de cette page, renommez-le en checksum.py et rendez-le exécutable : `chmod +x checksum.py`. Ce programme génère des octets aléatoires et les envoie sur le port série. Pour voir les options que vous pouvez lui passer, lancez-le avec l'option -h : `./checksum.py -h`.

Puis :

1. Écrivez une fonction pour votre carte qui reçoit des octets sur le port série et en calcule la somme
2. Lancez votre programme, et envoyez des octets à l'aide de checksum.py
3. Vérifiez que la somme que vous recevez est bien correcte.

Conclusion

Nous avons maintenant de quoi simuler des `printf`, la suite sera beaucoup plus simple !
Avez-vous pensé à committer / pusher, et mettre le tag UART sur le commit que nous devons corriger ? :)

Passons maintenant au pilotage d'une carte fille : un module de LED tel que [celui-ci](#).

Fichier attaché **Taille**

 [checksumpy.txt](#) 1.88 Ko

Afficheur à LED

Introduction

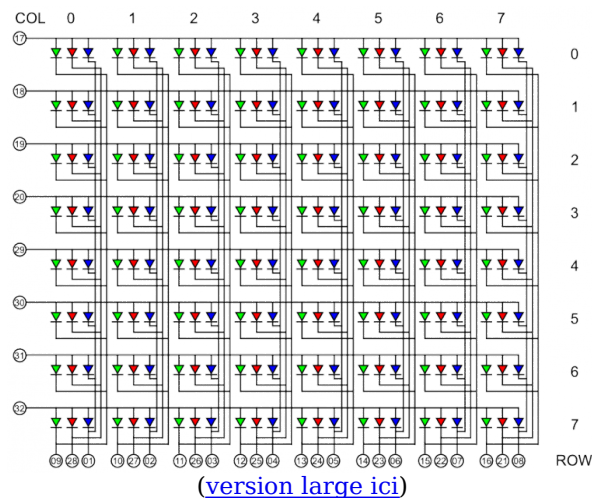
Nous allons maintenant ajouter une carte fille à la carte de TP. Cette carte fille est un afficheur sur une matrice de 8X8 LEDs RGB. Il est basé sur le circuit intégré DM163, qui permet de faire varier l'intensité de chaque couleur des LEDs de façon "très simple".

Principe du fonctionnement de la matrice de LED

La matrice de LED comportent 8 rangées de 8 colonnes de LED RVB (Rouge Vert Bleu - on dit aussi RGB en anglais). Chaque LED RGB est composée en fait de 3 LED mises dans un même chip : une rouge, une verte et une bleue.

Comme indiqué dans le schéma ci-dessous, les LED sont reliées entre elles ("*multiplexées*") de façon à réduire le nombre de broches de la matrice à 32 :

- Les anodes ("le côté +") des LED d'une même rangée sont reliées ensemble : 8 broches, reliées à VCC
- Les cathodes ("le côté -") des LED d'une même couleur et d'une même colonne sont reliées ensemble : $3 \times 8 = 24$ broches, reliées au contrôleur de LED DM163.



Pour allumer une certaine LED, il suffit de mettre la broche de sa ligne à VCC, et la broche de sa colonne à 0.

Mais le multiplexage pose un problème : il est impossible d'allumer en même temps deux LED situées sur des lignes et colonnes différentes. Par exemple, si on veut allumer la led verte la plus en haut à gauche et la bleue en bas à droite, on obtiendra en fait ceci :

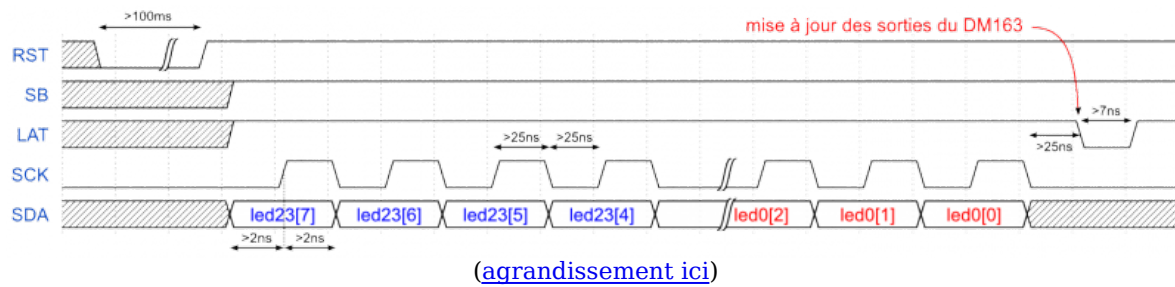


Le contrôleur de LED

- on mettra la GPIO correspondant à la ligne 0 à VCC, et celles aux autres lignes à GND.
- on programmera le DM163 pour qu'il fasse passer du courant sur la LED rouge de la colonne 0 (broche 28 de la matrice), et pas de courant sur les autres broches de colonne.

- le premier, appelé BANK0 qui stocke 6 bits par LED. Il contient donc $24 \times 6 = 144$ bits.
- le deuxième, appelé BANK1 qui stocke 8 bits par LED. Il contient donc $24 \times 8 = 192$ bits.

1. on commence par sélectionner le registre à décalage qu'on veut mettre à jour à l'aide du signal `sb` : 0 pour `BANK0`, 1 pour `BANK1`
2. on envoie sur `SDA` le bit de poids fort de la dernière LED (`led23[7]` si on met à jour le `BANK1`, `led23[5]` si on met à jour le `BANK0`),
3. puis on fait un pulse positif sur `sck` (front montant puis front descendant)
4. et on recommence en 2 jusqu'à ce que tous les bits aient été envoyés
5. enfin, on fait un pulse négatif sur `IAT`, ce qui transfère le contenu du registre à décalage dans le `BANK` choisi. Les sorties du `DM163` sont alors mises à jour instantanément.



Branchement sur la carte KL46Z

Le driver se branche naturellement sur la carte KL46Z. Attention de ne pas vous tromper de sens sinon vous cramez tout !

La matrice de LED se branche sur la carte driver, mais le sens est un peu difficile à repérer : la broche marquée 1 sur la matrice doit être branchée sur le connecteur **blue 1** du driver.

Les broches du drivers, telles que [documentées dans le user guide](#), sont branchées ainsi sur le processeur :

Broche du driver	Broche du processeur
SB	PB0
LAT	PB1
RST	PB2
SCK	PC8
SDA	PC9
C0	PA13
C1	PD2
C2	PD4
C3	PD6
C4	PD7
C5	PD5
C6	PA12
C7	PA4

Attention : la broche PD5 est aussi branchée sur la LED verte. Si vous utilisez la matrice de LED, n'utilisez plus la LED verte !

Contrôle basique

Initialisation des broches

Écrivez une fonction `void matrix_init()` qui :

1. met en marche les horloges des ports A, B, C et D (registre `SIM_SCGC5`).
2. configure toutes les broches du driver en mode GPIO (registres `PORTx_PCRn`)
3. configure toutes les broches du driver en sorties (registres `GPIOx_PDDR`)
4. positionne les sorties à une valeur initiale acceptable :
 - RST : 0 (reset le DM163)
 - LAT : 1
 - SB : 1
 - SCK et SDA : 0
 - C0 à C7 : 0 (éteint toutes les lignes)
5. attend au moins 100ms que le DM163 soit initialisé, puis passe RST à l'état haut.

Contrôle des broches

Écrivez les fonctions ou macros suivantes permettant de piloter indépendamment chaque broche :

- `RST(x)`
- `SB(x)`

- LAT(x)
- SCK(x)
- SDA(x)
- ROW0(x) à ROW7(x)

Par exemple RST(0) mettra la broche RST à 0, LAT(1) mettra la broche LAT à 1, ROW6(1) mettra C6 à 1, etc.

Génération de pulse

1. Écrivez, à l'aide de la macro SCK, une macro (ou fonction) pulse_SCK qui effectue un pulse positif (état bas, attente, état haut, attente, état bas, attente) sur SCK respectant les timings attendus par le DM163.
2. Écrivez, à l'aide de la macro LAT, une macro (ou fonction) pulse_LAT qui effectue un pulse négatif (état haut, attente, état bas, attente, état haut, attente) sur LAT respectant les timings attendus par le DM163.

Si vous devez faire des pauses, faites pour l'instant des boucles d'attente active à l'aide de asm volatile ("nop").

Contrôle des lignes

1. Écrivez, à l'aide des macros ROW0 à ROW7, une fonction void deactivate_rows() qui éteint toutes les lignes.
2. Écrivez, à l'aide des macros ROW0 à ROW7, une fonction void activate_row(int row) qui active la ligne dont le numéro est passé en argument.

Contrôle du DM163

- Écrivez une fonction void send_byte(uint8_t val, int bank) qui, à l'aide des macros SB, pulse_SCK et SDA, envoie 8 bits consécutifs au bank spécifié par le paramètre bank (0 ou 1) du DM163 (dans l'ordre attendu par celui-ci).
- Définissez le type rgb_color comme une structure représentant la couleur d'une case de la matrice :

```
typedef struct {
    uint8_t r;
    uint8_t g;
    uint8_t b;
} rgb_color;
```

- Écrivez, à l'aide de send_byte et activate_row, une fonction void mat_set_row(int row, const rgb_color *val) qui :
 - prend en argument un tableau val de 8 pixels
 - à l'aide de send_byte envoie ces 8 pixels au BANK1 du DM163 dans le bon ordre (B7, G7, R7, B6, G6, R6, ..., B0, G0, R0)
 - puis à l'aide de activate_row et pulse_LAT active la rangée passée en paramètre et les sorties du DM163.

Initialisation du BANK0

Écrivez une fonction void init_bank0() qui à l'aide de send_byte et pulse_LAT met tous les bits du BANK0 à 1. Faites en sorte que cette fonction soit appelée par matrix_init.

Test basique

Écrivez une fonction void test_pixels() qui teste votre affichage, par exemple en allumant successivement chaque ligne avec un dégradé de bleu, puis de vert puis de rouge. Cette fonction devra être appelée depuis le main, de façon à ce que nous n'ayons qu'à compiler (en tapant make) puis à loader votre programme pour voir votre programme de test fonctionner.

Committez tout ça avec le tag TEST_MATRIX.

Test d'affichage d'une image statique

Récupérez le fichier [image.raw](#). Ce fichier est un binaire contenant les valeur de chaque pixels stockés au format suivant :

- octet 0 : ligne 0, LED 0, rouge
- octet 1 : ligne 0, LED 0, vert
- octet 2 : ligne 0, LED 0, bleu
- octet 3 : ligne 0, LED 1, rouge
- octet 4 : ligne 0, LED 1, vert

- octet 5 : ligne 0, LED 1, bleu
- ...
- octet 189 : ligne 7, LED 7, rouge
- octet 190 : ligne 7, LED 7, vert
- octet 191 : ligne 7, LED 7, bleu

Faites en sorte que votre `main` affiche automatiquement cette image, en cyclant sur les lignes suffisamment vite pour que l'œil ait l'impression d'une image statique.

Vous devriez obtenir une image ressemblant à peu près à ceci (désolé pour la qualité de la photo) :



Committez le code avec le tag `TEST_STATIC_IMAGE`.

Conclusion

Nous savons maintenant piloter l'afficheur. On va voir comment faire pour afficher des images animées, mais pour cela il va falloir apprendre à :

- contrôler le temps de façon plus précise qu'avec des boucles d'attente active,
- pouvoir faire plusieurs tâches en parallèle : gérer les interruptions.

C'est le but de la [prochaine étape](#).

Fichier attaché	Taille
 Manuel d'utilisation du ColorShield	400.99 Ko
 Schéma électronique du ColorShield	19.88 Ko
 Datasheet du driver de LED utilisé sur le ColorShield (DM163)	1.22 Mo
 Datasheet de la matrice de LED	526.61 Ko
 image.raw	192 octets

IRQ

Vous savez maintenant récupérer un flux sur le port série, contrôler des GPIO, etc. Mais que se passe-t-il si on envie de faire tout ça en même temps ? Si des caractères arrivent sur le port série alors qu'on est en train de faire une autre tâche (longue), on risque de les rater. De même, comment faire pour exécuter une tâche périodique en fond ? Pour cela nous allons avoir besoin de deux choses :

- gérer les interruptions / exceptions
- générer automatiquement des interruptions à intervalles réguliers. C'est le rôle des timers.

Commençons par les interruptions.

Les exceptions sur processeurs à base de Cortex M

Introduction

Les exceptions sont des événements qui interrompent le cours normal d'exécution d'un programme. Lorsqu'une exception arrive, le programme en cours d'exécution est stoppé, et un bout de code spécifique à cette exception (appelé *handler*) est exécuté. Puis si l'exception n'était pas fatale, le programme reprend son cours comme si de rien n'était.

Les exceptions sont de deux types :

- causées par un événement interne au processeur (Cortex M0) : reset, bus fault (quand on accède à une adresse non mappée en mémoire), division par 0, ... Elles sont généralement graves et empêchent souvent la reprise du cours normal du programme interrompu.
- causées par un événement externe au processeur : un périphérique qui a besoin de signaler quelque chose, un changement d'état d'une GPIO, ... On les appelle alors communément *interruptions* (*interrupt request* ou *IRQ*).

En pratique, on utilise souvent indifféremment le terme *exception* ou *interruption*...

Le microcontrôleur KL46 est composé d'un coeur d'ARM (Cortex M0+) et de périphériques. Nous aurons donc deux types d'exceptions : celles propres au Cortex, et les IRQ dues aux périphériques du KL46.

Lorsqu'une exception est traitée, le processeur doit sauvegarder son état actuel. Cette sauvegarde peut être automatique (c'est le cas sur les Cortex M) ou manuelle (SPARC, ARM7TDMI, etc.).

Les exceptions peuvent avoir différentes priorités, permettant à une exception prioritaire d'interrompre le traitement d'une autre moins prioritaire. Ces priorités peuvent être réglées manuellement ou fixes. Ces priorités sont représentées par un nombre. Sur Cortex M, plus ce nombre est petit, plus l'exception est prioritaire.

En plus de ces priorités, la plupart des exceptions peuvent être désactivées ou activées logiciellement.

Les exceptions du Cortex M0+

Le Cortex M0+ intègre un contrôleur d'interruption très souple appelé *NVIC* (Nested Vectored Interrupt Controller) qui permet de gérer :

- 32 sources d'IRQ externes au processeur
- 1 NMI (Non Maskable Interrupt) : une interruption externe non désactivable
- les exceptions internes au processeur.

Chaque exception a un numéro qui lui est propre :

- les exceptions internes au processeur et la NMI sont numérotées de 1 à 15
- les 32 IRQ externes sont numérotées de 16 à 47. Elles ont en plus leur propre numéro d'IRQ externe qui vaut numéro d'exception - 16.

Exceptions internes

Les 15 exceptions internes au Cortex M0+ sont les suivantes :

- Exception 1 : le reset. C'est l'exception de plus haute priorité, et cette priorité est fixe : -3.
- Exception 2 : la NMI. Priorité fixe : -2. Elle n'est pas désactivable. Elle apparaît quand on présente un niveau bas sur la broche NMI du KL46.
- Exception 3 : Hard Fault. Priorité fixe : -1. Elle apparaît lorsqu'on essaye d'exécuter une instruction inconnue, quand on veut accéder à une zone non mappée en mémoire, quand on cherche à basculer du mode Thumb au mode ARM, ...
- Exception 11 : SVC (Service Call). Priorité réglable. Elle est utilisée pour faire un appel système à un OS
- Exception 14 : PendSVC (Pendable Service Call). Priorité réglable. Similaire au SVC, mais peut être retardée (si une tâche importante est en cours d'exécution par exemple).
- Exception 15 : SysTick. Priorité réglable. C'est un timer, permettant typiquement à un OS de déclencher un changement de contexte à intervalles réguliers.
- Les exceptions 4 à 10, 12 et 13 ne sont pas attribuées sur le Cortex M0+.

IRQ externes

Les 32 IRQ externes au Cortex M0+ sont numérotées de 16 à 47, sont de priorités réglables et désactivables / activables à volonté. Il y a une par périphérique (UART0, UART1, SPI0, etc.)

Table des vecteurs d'interruption

Le Cortex s'attend à avoir en mémoire une table donnant pour chaque exception l'adresse du handler associé. C'est ce qu'on appelle la *table des vecteurs d'interruption*. L'emplacement de cette table en mémoire est stocké dans le registre [VTOR](#) (situé à l'adresse 0xE000ED08). Au reset, le VTOR est chargé avec la valeur 0x00000000, ce qui veut dire que la table des vecteurs d'interruption est en flash. Mais vous avez la liberté de construire une autre table en RAM contenant les adresses de vos propres handlers, et de faire pointer VTOR sur cette table.

Traitement d'une exception

Lors de l'arrivée d'une exception :

1. le processeur sauvegarde automatiquement sur la pile les registres R0 à R3, R12, R14, PC, et xPSR
2. il stocke dans LR une valeur spéciale EXC_RETURN signifiant "retour d'exception" : 0xffffffff1, 0xffffffff9 ou 0xffffffffd. ([voir ici pour les détails](#))
3. il va chercher l'adresse du handler à exécuter à l'adresse suivante en mémoire : VTOR + exception_number * 4.
4. il saute à cette adresse et exécute le code qui s'y trouve.
5. à la fin de ce code, on trouve typiquement l'instruction BX LR, qui signifie "branchement à EXC_RETURN".
Le processeur recharge depuis la pile les registres sauvegardés, et reprend le cours normal de l'exécution du programme.

Vous avez dû remarquer que le processeur sauvegarde sur la pile les registres *caller saved* (en plus du PC et de xPSR). Cela permet de coder les handlers d'interruption comme des fonctions C tout à fait normales ! Il suffit juste que le SP soit positionné à une adresse correcte avant le déclenchement d'une exception.

Cela pose un problème pour la NMI qui n'est pas désactivable et peut se déclencher dès le boot avant que le SP ne soit bien positionné. Pour cela, les Cortex disposent d'une fonctionnalité rusée. La table des vecteurs d'interruption est organisée ainsi :

Table des vecteurs d'interruption

Adresse	Vecteur (numéro d'exception)	Numéro d'IRQ externe	Description
Exceptions internes au Cortex M0+			
0x00000000	0	-	SP initial
0x00000004	1	-	PC initial (Reset Handler)
0x00000008	2	-	NMI Handler
0x0000000C	3	-	Hard Fault Handler
0x00000010	4	-	-
...
0x0000003C	15	-	SysTick Handler
IRQ externes au Cortex M0+			
0x00000040	16	0	DMA0
0x00000044	17	1	DMA1
0x00000048	18	2	DMA2
0x0000004C	19	3	DMA3
...

Autrement dit :

- la première entrée est la valeur à laquelle positionner le SP au reset
- la deuxième entrée est la valeur à laquelle positionner le PC au reset (= le point d'entrée du programme)

Au reset, le processeur va positionner automatiquement le SP et le PC avec les deux premières entrées de la table. Une interruption peut donc survenir tout de suite, elle sera traitée correctement : le pointeur de pile sera bien positionné. Et sinon, c'est le point d'entrée du programme (généralement `_start`) qui sera exécuté.

Toutes les exceptions du KL46 sont disponibles en page 56 du [reference manual du processeur](#).

Le NVIC

Le contrôleur d'interruption (NVIC) n'est pas spécifique au contrôleur mais au Cortex M0+. Sa documentation est donc dans la [documentation officielle d'ARM sur les Cortex M0+](#). Cette documentation est aussi [disponible au format PDF en bas de cette page](#).

Il dispose de plusieurs registres qui permettent de contrôler les **IRQ externes** (et seulement les externes). Les deux principaux sont :

- NVIC_ISER : qui permet, en écrivant un 1 sur le bit n , d'activer l'IRQ externe numéro n (= l'exception

numéro 16+n)

- NVIC_ICER : qui permet, en écrivant un 1 sur le bit n, de désactiver l'IRQ externe numéro n (= l'exception numéro 16+n)

Par exemple, pour activer / désactiver les interruptions spécifique à l'UART0, dont le numéro d'IRQ externe est 12 (numéro d'exception 28) on écrira ceci :

```
// Pour activer les IRQ de l'UART0
NVIC_ISER = 1 << 12;

// Pour désactiver les IRQ de l'UART0
NVIC_ICER = 1 << 12;
```

Seule l'écriture d'un 1 dans ces registres a un effet. Cela évite d'avoir à faire un read-modify-write pour activer ou désactiver une interruption dans le NVIC.

Les exceptions internes ne sont pas modifiables par le NVIC.

Activation / désactivation de toutes les interruptions

Il est possible d'autoriser ou d'interdire toutes les interruptions (sauf le reset et la NMI), en positionnant le [registre PRIMASK](#) du processeur. On modifie ce registre par les instructions assembleur suivantes :

- cpsie i : active les interruptions
- cpsid i : désactive les interruptions

Au boulot !

Création de fonctions d'activation / interdiction des exceptions

Dans un fichier irq.h, définissez les macros suivantes :

- enable_irq() : qui autorise les interruptions (en positionnant PRIMASK)
- disable_irq() : qui interdit les interruptions (en positionnant PRIMASK)

Création d'une table de vecteur d'interruption par défaut

Les tables de vecteurs d'exception sont généralement écrites en assembleur ([exemple](#)) mais dans le cas des Cortex il est possible de les écrire en C. C'est ce que nous allons faire.

Dans un fichier irq.c, créez une table de vecteurs d'interruptions, sur ce modèle :

```
void *vector_table[] = {
    // Stack and Reset Handler
    &_stack,           /* Top of stack */
    _start,           /* Reset handler */

    // ARM internal exceptions
    NMI_Handler,      /* NMI handler */
    HardFault_Handler, /* Hard Fault handler */
    0,                /* Reserved */
    0,                /* Reserved */
    0,                /* Reserved */
    0,                /* Reserved */
    0,                /* Reserved */
    0,                /* Reserved */
    0,                /* Reserved */
    SVC_Handler,      /* SVC handler */
    0,                /* Reserved */
    0,                /* Reserved */
    PendSV_Handler,   /* Pending SVC handler */
    SysTick_Handler,  /* SysTick handler */

    // KL46 External interrupts
    DMA0_IRQHandler,   /* DMA0 interrupt */
    DMA1_IRQHandler,   /* DMA1 interrupt */
    DMA2_IRQHandler,   /* DMA2 interrupt */
    ...
}
```

Cette table est un tableau de "pointeurs sur n'importe quoi" (void *), qu'on peuple avec les adresses de handlers par défaut.

Pour respecter certaines conventions ([CMSIS](#)), appelez les handlers d'exceptions internes XXX_Handler, et les handlers d'IRQ externes XXX_IRQHandler.

Création des handlers d'interruption par défaut

Dans le fichier `irq.c`, définissez (avant la table des vecteurs) des handlers par défaut qui feront la chose suivante :

- désactiver toutes les interruptions
- faire une boucle sans fin

On pourra ainsi vite voir si on sait générer une interruption et si le bon handler est appelé.

Bien entendu, il est hors de question d'écrire 40 fois le même code ! Écrivez donc une macro `MAKE_DEFAULT_HANDLER`, qui prend en argument un nom de handler (par exemple `truc_IRQHandler`) et qui déclare et instancie la fonction `truc_IRQHandler`.

Les handlers par défaut seront amenés à être surchargés par d'autres fichiers C, qui voudront mettre en place leur propre handler. Pour cela, faites en sorte que la définition des handlers inclue bien l'attribut `weak` : `void __attribute__((weak)) truc_IRQHandler(void) {...}`

Initialisation des interruptions

Toujours dans `irq.c`, écrivez les fonctions suivantes (qui devront être exportées) :

- `void irq_init(void)` : qui stocke dans `VTOR` l'adresse de la table des vecteurs d'interruption
- `void irq_enable(int irq_number)` : qui autorise dans le NVIC l'IRQ externe numéro `irq_number` (allant de 0 à 31)
- `void irq_disable(int irq_number)` : qui autorise dans le NVIC l'IRQ externe numéro `irq_number` (allant de 0 à 31).

Pensez à appeler `irq_init` dans votre `main...`

Problème de la NMI

Malheureusement, la broche permettant de générer une NMI est reliée au color shield, et dès la mise sous tension de la carte, une NMI est générée en permanence. Le processeur passe alors en mode handler de la NMI (visible par la valeur 2 sur l'octet de poids faible du registre processeur `xPSR`), et comme c'est une des IRQ de plus haute priorité, les autres IRQ ne seront jamais honorées. Il va donc falloir bloquer la NMI.

Par définition, une NMI est non masquable. Heureusement, le KL46 dispose d'un mécanisme permettant de désactiver non pas la NMI mais la broche reliée à la NMI (PORTA 4). Ce mécanisme est un peu complexe et passe par un système de mots de configuration stockés à une adresse spéciale en flash.

1. Récupérez le fichier assembleur [flash.s](#), et incluez le dans la liste des objets à compiler.
2. Modifiez votre `ldscript` :
 1. Dans le layout mémoire, faites commencer la flash en 2k (après la zone de configuration) et donnez-lui une taille de 254k
 2. Dans le layout mémoire, créez une zone `flash_config`, qui commence en 0x400 et a une taille de 32 octets
 3. Dans les sections, faites en sorte que la section `.flash_config` de `flash.o` soit placée dans la zone mémoire `flash_config`

Compilez, et **vérifiez** par `objdump -s` que les mots de configuration de la flash sont bien mis en 0x400 avec les bonnes valeurs.

Flashez votre carte de façon habituelle. Puis débranchez-la, rebranchez-la, et relancez GDB et JLinkGDBServer.

Vérifiez que votre programme habituel continue de fonctionner normalement.

Génération d'une interruption par l'appui du bouton SW1

Nous allons faire en sorte que l'appui sur le bouton poussoir SW1 (relié à la broche `PORTC3`) génère une interruption qui fera toggler la LED rouge.

Les GPIO sont capables de générer des interruptions sur état (haut ou bas), sur front (montant ou descendant) etc. Une fois que l'interruption est déclenchée, elle reste active jusqu'à que vous l'acquittiez en écrivant ce qu'il faut dans le registre `PORTx_PCRn`.

Créez dans un fichier `buttons.c` la fonction `void button_init(void)` qui

- active l'horloge du port C,
- configure la broche `PORTC3` en GPIO, en entrée, avec un pull-up et déclenchant une interruption sur front descendant.
- active l'interruption du `PORTC` (elle est commune avec le `PORTD`)

Compilez, et testez en appuyant sur SW1 que le handler par défaut des GPIO du `PORTC` est bien appelé.

Rajoutez maintenant dans `buttons.c` un handler spécifique, qui primera sur le handler par défaut. Ce handler fera les choses suivantes :

- acquittement de l'interruption (registre `PORTC_PCR3`)
- toggle de la LED rouge

Compilez et testez : à chaque appui sur SW1 la LED rouge doit toggler, et ce sans que cela perturbe votre programme habituel (affichage d'une l'image sur le color shield).

Conclusion

Vous savez maintenant comment générer et traiter des exceptions sur Cortex M. Nous allons voir dans la suite comment générer des interruptions périodiques pour effectuer des tâches de fond par exemple. En attendant, tag `IRQ` ! :)

Fichier attaché	Taille
 Documentation du coeur Cortex M0+	939.96 Ko
 flash.s	232 octets

UART + IRQ + LED !

Vous savez récupérer des octets sur le port série et piloter l'afficheur de LED. L'objectif de ce contrôle est de pouvoir depuis un PC envoyer des trames par le port série à la carte de TP et de les afficher en temps réel. Si on arrête d'envoyer des trames, la dernière en date doit rester affichée. Et on doit pouvoir recommencer à envoyer des trames sans avoir à redémarrer le programme sur la carte.

Le port série

Les échanges sur le port série seront faits à la vitesse de **38400 bauds**, 8 bits, pas de parité, 1 bit de stop (38400 8N1).

À chaque fois que la carte est débranchée et rebranchée, le port série du PC doit être reconfiguré en 38400 8N1 en exécutant `sh ./stty.sh` ([programme fourni en bas de cette page](#)).

Pour envoyer le contenu d'un fichier `bidule.bin` sur le port série, on utilisera la commande suivante : `cat bidule.bin > /dev/ttyACM0`

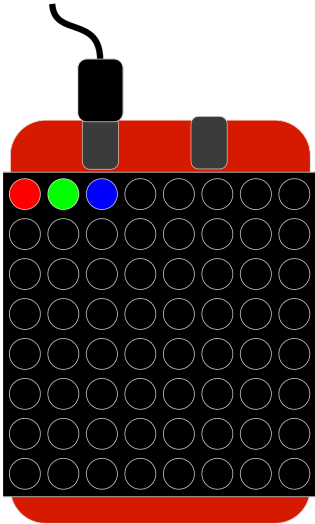
Format des trames envoyées par le PC

Les fichiers qu'on affichera contiennent une ou plusieurs trames au format suivant :

- Chaque début de trame est indiqué par l'octet `0xff`.
- Chaque octet différent de `0xff` fait partie d'un pixel.
- Les pixels sont stockés dans l'ordre naturel : d'abord ceux de la première ligne en commençant par la gauche, puis ceux de la deuxième ligne, etc.
- Chaque pixel est composé de 3 octets, d'abord R, puis G puis B (chacun compris entre `0x00` et `0xfe`).

Ainsi une trame qui allume le pixel en haut à gauche en bleu sera composée des octets suivants : `0xff 0x00 0x00 0xfe 0x00 ... 0x00`.

Le fichier de test [one_frame.bin](#) contient une seule trame et permet d'afficher l'image suivante :



Vous pouvez voir le contenu du fichier en question par la commande `od -tx1 -v one_frame.bin`

Architecture du code

Vous devez pouvoir afficher une image en même temps que vous recevez des pixels. Il va donc falloir faire en sorte que deux tâches s'exécutent en parallèle :

1. une qui reçoit les trames et les stocke à un endroit approprié en mémoire,
2. l'autre qui rafraichit en permanence l'écran.

Vous allez donc faire en sorte que le port série fonctionne sous interruption, de façon à ce qu'une IRQ soit générée dès qu'un octet est reçu. Votre handler d'IRQ traitera cet octet et mettra à jour au fur et à mesure la trame qui est affichée. On n'attendra pas qu'une trame soit complètement reçue pour mettre à jour l'affichage, ni que les trois composantes d'un pixel soient reçues entièrement.

Remarque : la trame affichée est un objet partagé entre la tâche de réception et celle d'affichage. Des problèmes de concurrence peuvent donc se poser, si on envoie une ligne à l'afficheur alors qu'on n'a reçu qu'une ou deux des trois composantes d'un pixel. On ne s'en préoccupe pas ici, l'oeil n'étant pas assez rapide pour le voir. Ceux qui feront ELECINF344 verront en détail comment implémenter tout ça proprement.





Première partie

1. Définissez un objet global qui contiendra la trame affichée. Il sera modifié par le handler d'IRQ du port série, et lu par la tâche d'affichage.
2. Écrivez la tâche de réception du port série (handler d'IRQ) qui traite les octets reçus. Rappel, on fonctionne à 38400 bauds.
3. Écrivez la tâche d'affichage qui affiche permanence la trame courante (normalement vous l'avez déjà écrite dans le TP...)

Pour tester votre code, vous avez trois fichiers à votre disposition. Par ordre de difficulté croissante :

- [one_frame.bin](#) qui contient une seule trame
- [many_frames.bin](#) qui contient plusieurs trames pour avoir un affichage animé
- [final.bin](#) qui contient lui aussi plusieurs trames et vous permettra de tester la robustesse de votre code face à des trames potentiellement mal formées. Ce fichier vous affichera aussi un message vous donnant l'URL de la suite de cette partie.

N'oubliez pas le tag `UART_IRQ_1` !

Fichier attaché	Taille
 one_frame.bin	193 octets
 many_frames.bin	1.56 Mo
 final.bin	227.63 Ko
 stty.sh	77 octets

Timers

Tous les microcontrôleurs sont équipés de dispositifs appelés "timers". Ce sont des compteurs / décompteurs qui peuvent avoir beaucoup de possibilité :

- générer une interruption lorsqu'ils ont atteint une certaine valeur, ce qui permet d'effectuer des tâches à intervalles réguliers
- faire bouger automatiquement une broche du microprocesseur, ce qui permet de générer facilement du [PWM](#)
- compter non pas les cycles d'horloges mais des événements externes
- déterminer automatiquement la largeur d'un créneau appliqué à une broche externe
- etc.

Les Cortex incluent tous un timer spécial appelé `sysTick`. Ce timer est fait spécifiquement pour donner une base de temps aux OS multitâche, de façon à ce qu'ils puissent effectuer des changements de contexte à intervalles réguliers.

Les microcontrôleurs incluent d'autres timers en tant que périphériques externes, disponibles pour l'utilisateur (le `sysTick` étant généralement réservé à l'OS). Le KL46 dispose de plusieurs timers :

- le `TPM`, qui contient deux timers sur 16 bits, permettant de faire à peu près tout ce qui est cité ci-dessus
- le `PIT` (Periodic Interrupt Timer) dont le but est de générer des interruptions à intervalles réguliers, comme le `sysTick` mais disponible pour l'utilisateur. C'est lui que nous allons utiliser.

Sa documentation est disponible en page 587 du [manuel de référence du KL46](#).

Mise en marche du PIT

Le PIT est un double compteur sur 32 bits, qui compte les cycles de la bus clock fonctionnant à 24MHz. Plus exactement, c'est un décompteur. On le charge avec une valeur initiale, et dès qu'il arrive à zéro il génère une IRQ, qu'il faudra comme d'habitude acquitter.

On peut utiliser séparément chaque décompteur, ou chaîner les décompteurs de façon à faire un décompteur sur 64 bits, permettant d'atteindre des intervalles très grands (heure, jour, mois, année...). Ici, nous n'utiliseront qu'un seul de ces compteurs, un intervalle d'une seconde tenant sans problème sur 32 bits.

Dans un fichier `pit.c`, écrivez les fonctions suivantes :

- `void pit_init(void)` qui configure le PIT pour générer une IRQ toutes les secondes
- un handler d'IRQ spécifique, qui fera toggler la LED rouge à chaque interruption du PIT.

Testez ! La LED rouge doit toggler maintenant toutes les secondes, ou à chaque appui sur SW1.

N'oubliez pas le tag `TIMER_1` !

Fonctionnement de la matrice sous IRQ

Faites maintenant en sorte que l'affichage de la matrice soit géré par des interruptions : affichage de l'image courante au moins 70 fois par seconde.

N'oubliez pas le tag `TIMER_2` !

Horloges, PLL & Co. - Part II

Introduction

Warning : accrochez-vous, cette partie est un peu complexe ! Lisez bien le texte du TP, et prenez le temps de lire les docs mentionnées.

Lorsque l'on démarre un processeur, une des premières choses à faire est de configurer les différents éléments intervenant dans la génération des horloges afin d'obtenir les différentes fréquences d'horloges désirées.

Dans le processeur utilisé sur vos cartes de TP, le cœur et les périphériques utilisent des horloges différentes. Le schéma suivant, issu de la [documentation du processeur](#) (page 124), détaille les différentes sources d'horloge et les différents composants intervenant dans la génération des différentes horloges utilisées au sein de la puce. Référez-vous aussi au "[Quick start guide](#)" pour une bonne introduction à ce sujet un peu complexe.

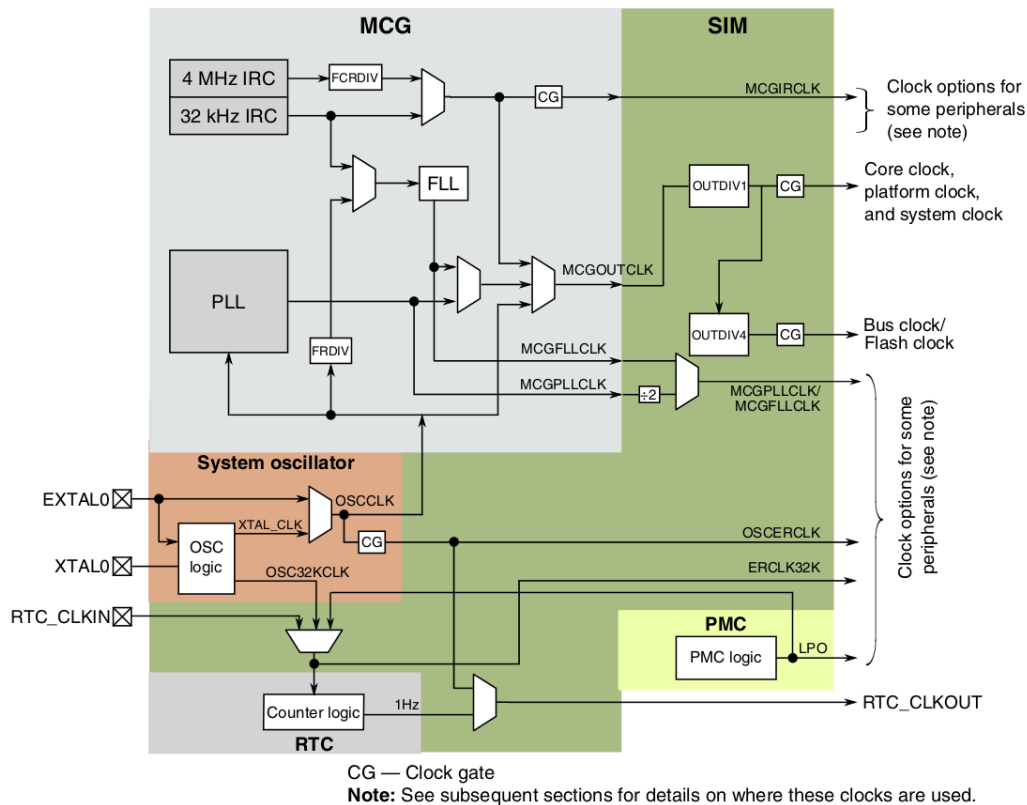


Figure 5-1. Clocking diagram

La description des différentes horloges et des plages de fréquences autorisées est également donnée dans la [documentation](#) (tables pp. 124-127).

Ce schéma, assez complexe, est typique des processeurs et des System-on-Chip actuels. Ne stressiez pas, on va vous l'expliquer pas à pas.

Sources d'horloges

Le processeur dispose de 4 sources d'horloges principales :

- une horloge interne au chip, peu précise et de fréquence ~4MHz, appelée Internal Reference Clock (4MHz-IRC)
- une horloge interne au chip, lente, peu précise et de fréquence ~32kHz, appelée Internal Reference Clock (32kHz-IRC)
- une horloge générée par un quartz extérieur au chip, donc précise, de fréquence 3 à 32MHz. Le quartz est branché entre les broches EXTAL0 et XTAL0.
- une horloge générée par un quartz extérieur au chip, précise à 32.768kHz, et ne servant qu'à garder l'heure courante (Real Time Clock - RTC).

Seules les 3 premières permettent de faire fonctionner le microcontrôleur, à des vitesses plus ou moins rapides (ce qui influe sur la consommation) et plus ou moins précises.

Malheureusement, certains périphériques comme les UART (ports série) ou l'USB demande une précision que les horloges internes ne peuvent pas fournir. Il faut donc dans ce cas utiliser le quartz externe.

Architecture du générateur d'horloge

Les différents composants du processeur n'ont pas les mêmes besoins d'horloge. Certains ont besoin d'une horloge bien spécifique (l'USB, qui doit avoir une horloge de 96MHz divisée par deux, l'UART, les timers, ...), d'autres non. Plus une horloge va vite, plus le circuit consomme. C'est donc au programmeur du système de choisir la source des horloges (interne ou externe) et d'ajuster leur fréquence de façon à obtenir une consommation minimale tout en assurant un fonctionnement correct du circuit.

Pour cela, la génération des horloges internes est séparée en 3 blocs.

Le MGC - Multipurpose Clock Générateur

Pour faire fonctionner le processeur au maximum de ses possibilités, il est possible d'utiliser des circuits spéciaux qui multiplient la fréquence des horloges. Il en existe deux dans ce microcontrôleur :

- une FLL (Frequency Locked Loop) : elle est peu précise, mais ne consomme pas beaucoup. Elle permet de passer d'une fréquence d'entrée entre 31.25kHz et 39.0625kHz à une fréquence de 20MHz à 48MHz environ.
- une PLL (Phase Locked Loop) : elle est très précise mais consomme plus que la FLL. Elle permet de passer d'une fréquence de 2 à 4Mhz à une fréquence de 48 à 100MHz environ.

Le MGC contient une PLL et une FLL, ainsi que des diviseurs d'horloge, et produit les signaux suivants :

- MGCOUTCLK : une horloge qui servira à produire les horloges du coeur de processeur, du bus, des mémoires et de certains périphériques. Elle est produite à partir de la PLL, de la FLL, ou directement depuis un des oscillateurs internes ou externes.
- MGCFLCLK : une horloge produite par la FLL, qui sert éventuellement à certains périphériques non critiques.
- MGCPLLCLK : une horloge produite par la PLL, qui sert aux périphériques nécessitant une horloge précise.

Il produit aussi d'autres horloges (OSCERCLK, ERCLK32K et LP0) dont nous ne nous préoccupons pas ici.

Conclusion : pour tout ce qui est mise en marche de la PLL et choix de la fréquence, ça se passe dans le MGC !

Le SIM - System Intégration Module

Le SIM permet entre autres :

- d'activer ou non les horloges des périphériques
- de choisir laquelle on lui donne
- et éventuellement sous forme divisée (pour limiter la consommation).

Conclusion : pour activer l'horloge d'un périphérique, ça se passera dans le SIM !

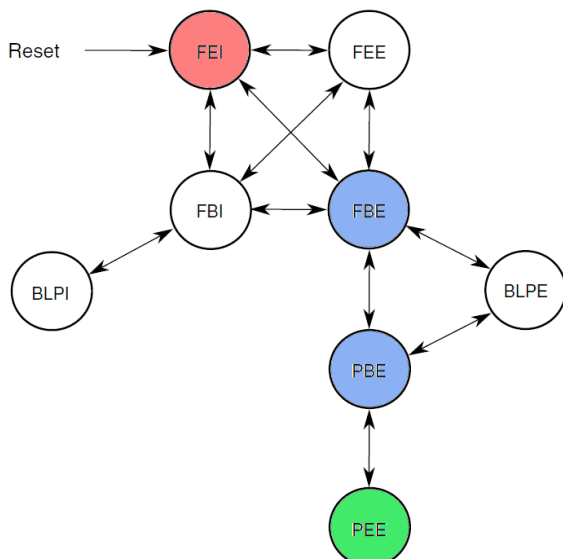
Passage de la référence interne au quartz externe

Au démarrage, le circuit est en mode "FLL Engaged Internal (FEI)" (voir description p. 398). Dans ce mode, la référence interne à 32 kHz est utilisée. Cette fréquence est multipliée par un facteur 640 par la FLL (Frequency-Locked Loop) pour donner une fréquence de base MGCOUTCLK d'environ 21MHz (peu précise).

L'avantage de ce mode est que le processeur peut fonctionner sans horloge externe mais l'inconvénient majeur est que la référence interne n'est pas précise et donc qu'il est difficile de connaître la valeur exacte de la fréquence des horloges utilisées, ce qui peut être problématique pour des périphériques ayant besoin d'une fréquence précise (ex. USB, UART, timers, etc.).

Nous allons donc basculer le circuit du mode FEI au mode PEE ("PLL Engaged External"), où un quartz externe de 8MHz très précis va alimenter la PLL et générer un MGCOUTCLK à 48MHz.

Cette transition ne peut pas se faire directement. Il faut passer par des modes intermédiaires, résumés dans l'image ci-dessous (pour la signification des sigles, cf. pages 398 et suivantes du [reference manual](#)).



Un exemple de procédure permettant de basculer du mode FEI au mode PEE est décrite dans le [reference manual](#) page 409. Attention, dans l'exemple, un quartz à 4 MHz est utilisé, or, sur la carte de TP, c'est un quartz à 8 MHz, il faudra donc adapter les applications numériques. On cherche à obtenir une fréquence de 48 MHz pour MCGOUTCLK.

Vous pouvez aussi vous inspirer du pseudo code du [quick start guide](#) pages 35 et suivantes (ils ne suivent pas exactement le même ordre que le reference manual, mais la manipulation revient au même).

La procédure est simple :

1. passage du mode FEI au mode FBE
2. passage du mode FBE au mode PBE
3. passage du mode PBE au mode PEE

Pour chaque étape,

1. Écrivez le code nécessaire pour effectuer un basculement. Allez-y doucement, et vérifiez bien ce que vous écrivez.
2. Testez chaque changement de mode : faites clignoter une LED à une fréquence facilement vérifiable (1Hz par exemple), en regardant le code assembleur pour voir combien de cycles prend votre boucle d'attente active. N'oubliez pas que les registres SIM_CLKDIV1 et SIM_CLKDIV2 divisent MCGOUTCLK par un certain facteur (spécifié par défaut par FTFA_FOPT, qui, lorsque la flash est effacée comme sur vos cartes, vaut 11 : fast-clock boot. Cf. page 128 du [reference manual](#)).

Configuration et activation des horloges des périphériques

Dans cette partie nous allons maintenant configurer les horloges du coeur et des différents périphériques.

Maintenant que l'horloge MCGOUTCLK est configuré à 48 MHz avec pour source l'oscillateur à quartz externe, configurez les horloges suivantes :

- Core clock, Platform clock et System clock à leurs maximum (48 MHz) (indice : registre SIM_CLKDIV1)
- Bus clock, Flash clock à leur maximum (24 MHz) (indice : registre SIM_CLKDIV1)

Pour cela, regardez la documentation du "System Integration Module (SIM)" page 201.

Conclusion

Vous avez pensé à committer / pusher avec le tag HORLOGES ? :)

Mise en flash

Introduction

Le but de cette partie est d'obtenir une carte qui puisse fonctionner toute seule, dès qu'on la met sous tension, sans avoir besoin d'uploader un programme ni de lancer gbd. Pour cela vous allez placer votre exécutable en flash.

Attention : une mauvaise manipulation de votre part, et la carte devient inutilisable ! Suivez donc bien les instructions à la lettre.

Mapping mémoire

On rappelle que le mapping mémoire du processeur est disponible en pages 113 et suivantes du reference manuel du processeur. Maintenant que vous maîtrisez les choses, le mapping mémoire est un peu plus complexe que celui présenté en début du TD.

- flash : adresse de début = 0x00000000, taille = 256k
 - 0x00000000 - 0x000000BF : table des vecteurs d'interruption (48 mots de 32 bits, cf. page 57 du manuel de référence)
 - 0x000000C0 - 0x000000FB : réservé
 - 0x000000FC - 0x000000FF : **IRC user trim values** (cf. page 115 du manuel de référence)
 - 0x00000400 - 0x0000040F : **configuration de la protection de la flash** (cf. page 437 du manuel de référence)
 - 0x0000040C - 0x0003FFFF : libre
- RAM : elle est séparée en deux blocs contigus :
 - RAML : début = 0x1fffe000, taille = 8k
 - RAMH : début = 0x20000000, taille = 24k

Les zones en rouge ci-dessus sont des zones à ne surtout pas modifier. Une mauvaise écriture dedans peut mettre la carte dans un mode "sécurisé" où ne peut plus ni relire, ni modifier, ni effacer le contenu de la flash : la carte devient donc inutilisable pour les prochains TP...

Par précaution, vous n'utiliserez donc que les zones suivantes de la flash :

- 0x00000000 - 0x000000BF : table des vecteurs d'interruption (48 mots de 32 bits)
- 0x00000400 - 0x0000040F : flash_config (16 octets de configuration, provenant du [flash.s qu'on vous a fourni](#))
- 0x00001000 - 0x0003FFFF : votre programme

Au travail !

Définissez un nouveau layout mémoire dans votre linker script selon le modèle ci-dessus.

Passage du code en XIP

En vous rappelant des étapes de boot d'un processeur et de tout ce que doit faire le crt0.s, modifiez votre linker script et votre code d'initialisation de façon à ce que la carte puisse booter et que le programme s'exécute depuis la flash.

Attention : avant de reflasher votre carte avec gdb, vérifiez bien avec objdump que vous ne touchez qu'aux zones autorisées en flash, et que le contenu de la section flash_config est bien situé à l'adresse 0x400.

Fermez gdb et le driver de sonde JLink, débranchez votre carte puis rebranchez-la. Vérifiez que votre s'exécute correctement.

Recopie du code en RAM

Sur beaucoup de processeurs l'accès à la flash est plus lent que l'accès à la RAM. Ce n'est pas le cas de ce processeur, mais pour l'exercice nous allons faire semblant que si !

Modifiez votre code et votre linker script de façon à ce que le code

1. commence par s'exécuter en flash
2. s'auto-recopie en RAM
3. transfère son exécution à la partie située en RAM

On laissera la section .rodata en flash. On relogera la table des vecteurs d'interruption en RAM, de façon à donner à l'utilisateur la possibilité de mettre en place ses propres handlers d'IRQ.

Conclusion

Si vous êtes arrivé jusqu'ici, BRAVO ! Vous savez maintenant comment un système embarqué démarre, est configuré, comment accéder à ses périphériques et comment lancer un exécutable.

Nous espérons que cette UE et ce TD vous ont plu et ont pu démystifier la génération des exécutables et le fonctionnement d'un système à processeur basique.

N'oubliez pas de mettre le tag FINAL sur votre dernier commit et de pusher le tout sur master :)