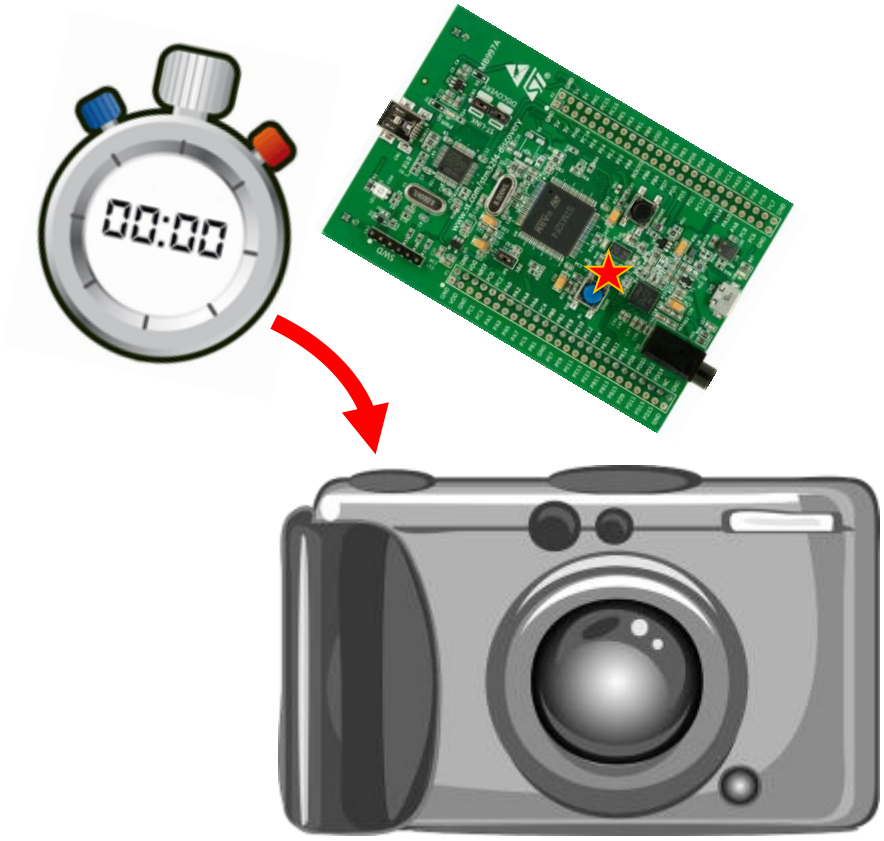
Mission 3 – Le déclencheur automatique d'appareil photo



2015

Objectifs de la mission :

- Prise en main des outils, de la cible, de l'environnement de développement
- Lecture et modification d'un programme existant
- Utilisation du débogueur
- Utilisation du périphérique TIMER
- Développement d'une machine à états

1 Avant de commencer, prenons un peu de recul

(Prenez le temps de lire attentivement cette partie).

Vos précédentes missions consistaient à prendre en main la cible STM32 et ses outils.

Vous avez pu entrevoir une arborescence de nombreux dossiers, fichiers sources et header… et vous avez utilisé un microcontrôleur.

Essayons maintenant d'y voir plus clair…

Tout d'abord, qu'elle transformation permet de passer de ces nombreux fichiers sources au résultat observable sur le microcontrôleur ?

Lorsque vous cliquez sur le bouton 'build', un ensemble de fichier est généré (allez faire un tour dans le répertoire 'Debug' pour vous en convaincre).

- Le compilateur est appelé pour fabriquer un fichier objet (.o) à partir de chaque fichier source (.c en langage C ou .s en assembleur).
- L'éditeur de liens (linker) est ensuite appelé. Il rassemble tous ces objets et établi les liens entre eux. C'est grâce à cela qu'on peut appeler une fonction décrite dans un autre fichier.
- Les fichiers binaires obtenus contiennent des instructions élémentaires en « langage machine » que le processeur peut interpréter. Ces fichiers sont évidemment inexploitables directement par un humain (normalement constitué…)

Un microcontrôleur contient un processeur, de la mémoire, et un ensemble de périphériques (nous y reviendront).

Lorsque le microcontrôleur est mis sous tension, il « démarre », et va exécuter la première instruction, celle qui se trouve à l'adresse 0. (voir Reset_Handler dans le fichier startup.s)

Ensuite, tant que l'alimentation électrique est présente, le microcontrôleur exécute les instructions les unes après les autres. Certaines lui demandent de faire des calculs (additionner, multiplier, charger telle ou telle donnée…) tandis que d'autres l'invitent à partir exécuter des instructions qui se trouvent ailleurs en mémoire… ou à revenir de là d'où il était parti.

La partie du processeur qui effectue ces traitements est l'unité arithmétique et logique. Un séquenceur assure le déroulement du programme et la sélection des portes logiques utilisées.

Finalement, un jeu d'instructions simples permet des applications très variées !

Un processeur seul ne sert pas à grand-chose… A moins de vouloir un radiateur excessivement complexe, il peut être malin d'y ajouter de la mémoire. Dans le STM32, on trouve de la mémoire RAM (dont le contenu s'efface si l'alimentation est coupée), et de la mémoire FLASH (comme sur une clé USB, dont le contenu est conservé). C'est cette mémoire qui contient le programme.

Comme avec les maisons dans un pays où chaque maison est référencée par une adresse, chaque octet de la mémoire être rangé à une adresse. Ainsi, un plan d'adressage est constitué par le fabricant du microcontrôleur. A l'adresse 0x20001234, vous trouvez une cellule d'un octet de RAM. A l'adresse 0x80004567, il y a un octet de FLASH…

Faire des calculs et ranger le résultat en mémoire est une chose… mais sans périphériques, à quoi bon ? Même une simple calculatrice nécessite des périphériques, pour y relier les boutons et l'écran !

Les concepteurs du microcontrôleur y ont donc ajouté des composants que l'on appelle des périphériques. Ils peuvent être décrits en VHDL, ou dessinés avec des schémas électriques. (Ils contiennent d'ailleurs de nombreuses bascules D, des amplis, des résistances, de la mémoire, des portes logiques…)

Ces périphériques disposent d'une part d'autonomie. On leur sous-traite des tâches qu'ils ont été conçus pour réaliser.

Par exemple, un timer est un périphérique qui sert de chronomètre, on lui demande de compter. Un ADC sait mesurer une tension analogique et la convertir en une grandeur numérique. L'UART sait comment envoyer des octets sur une liaison série à la vitesse demandée. L'I2C connait les règles de fonctionnement du bus I2C…

Pour dialoguer avec ces périphériques, il suffit au processeur d'écrire et de lire à certaines adresses du plan d'adressage. On n'y trouve pas forcément des cases mémoires, mais chaque périphérique dispose d'un ensemble de registres de périphériques dont chacun a une signification et un rôle précis.

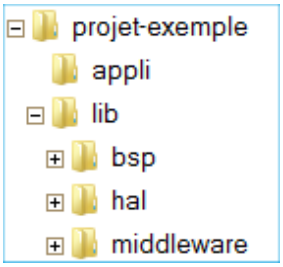
Pour mieux comprendre ce concept, imaginez de minuscules ouvriers, qui disposent de formulaires pour dialoguer avec le processeur. Pour l'ouvrier « PWM », chargé de fabriquer des signaux modulés en largeur d'impulsion, le processeur rempli les champs du formulaire : période, horloge utilisée, rapport cyclique. Le périphérique PWM produit alors le signal demandé.

Le périphérique ADC a pour mission de numériser des grandeurs analogiques. Le processeur lui indique par ses registres que l'ADC doit être alimenté, qu'il doit recevoir une horloge pour cadencer son travail précisément, qu'il doit numériser le signal venant de la broche AN6, qu'il doit ranger le résultat sous telle forme, dans tel registre, et qu'il doit prévenir le processeur de telle façon.

Avec la datasheet qui détaille le fonctionnement interne du microcontrôleur et de ses périphériques, on peut rédiger les instructions qui permettent de faire fonctionner chaque périphérique. Vu le nombre de périphériques disponibles sur le STM32F407 utilisé, nous sommes très heureux que ST ait déjà fait le travail pour nous !

De nombreuses librairies sont à notre disposition. Elles sont organisées en différents niveaux, selon leur proximité avec les registres de périphériques. Ces couches d'abstractions permettent d'ailleurs de faciliter l'organisation et le portage d'une application. Un programme dont les couches d'abstraction sont bien conçues peut être porté facilement sur plusieurs microcontrôleurs complètements différents, simplement en changeant la couche basse (souvent nommé HAL, Hardware Abstraction Layer, ou couche d'abstraction matérielle).

Le projet qui vous est fourni est structuré comme ceci :



appli : votre application. Vos sources. Ce code est souvent spécifique à une application.

lib : les librairies. Ce code qui s'y trouve est soit construit par vous soit fourni par un tiers. Il est modulaire et réutilisable dans d'autres projets.

hal : *hardware abstraction layer*. Au plus bas niveau, au plus proches des périphériques.

bsp : *board support package*. Les fonctions sont construites au niveau supérieur à l'HAL. Bien conçues, elles sont utilisables sur plusieurs cibles. On y trouve des fonctions qui facilitent l'accès aux périphériques externes du microcontrôleur (accéléromètre, écran LCD, carte micro-SD, …) et des fonctions qui s'appuient sur l'HAL pour fournir des fonctionnalités logicielles plus avancées.

middleware : ces ensembles logiciels permettent des fonctionnalités logicielles plus complètes (gestion d'un système de fichier FAT, d'une pile TCP/IP, de l'USB)

2 Le cahier des charges

.

Un client très important, photographe à ses heures perdues, vous demande de réaliser un déclencheur automatique pour son appareil photo.

Il a déjà démandé à une équipe la réalisation hardware de la carte électronique. Cette réalisation n'étant pas terminée, et les délais étant serrés, nous devrons, comme bien souvent dans l'industrie, anticiper la rédaction du logiciel.

L'équipe 'matériel' nous a toutefois donné quelques informations sur le design :

- ➔ Le déclenchement de l'appareil photo est relié à la broche RD15.
- ➔ La led rouge indiquant le compte à rebours est reliée à la broche RD14.
- ➔ Le bouton poussoir déclenchant le compte à rebours est relié à la broche RA0

L'un de vos collègues à planché sur le sujet hier, et à réalisé la machine à états de l'application.

Nous ne disposons que d'une carte d'évaluation STM32F4-DISCOVERY… et on remarque que les choix effectués par l'équipe hardware sont assez judicieux, et pratiques pour notre développement logiciel.

3 Les spécifications

Après discussions avec le client, vous avez pu établir un cahier de spécifications, qu'il a signé.

Un cahier de spécifications décrit le comportement du système que vous allez développer. (Il ne décrit pas « comment on va faire », mais « quel est le comportement du système du point de vue de l'utilisateur»).

Voici quelques éléments extraits de ces spécifications.

Lorsque l'on appuie sur le bouton, un compte à rebours de 5 secondes est lancé.

- Pendant les **3 premières secondes** : la **led rouge** clignote toutes les **500 ms**.
- Pendant les **2 dernières secondes** : la **led rouge** clignote toutes les **100 ms**.
- Pendant **10 ms** à l'instant du déclenchement : pilotage du **déclenchement**.

L'objectif pédagogique de cette mission est double :

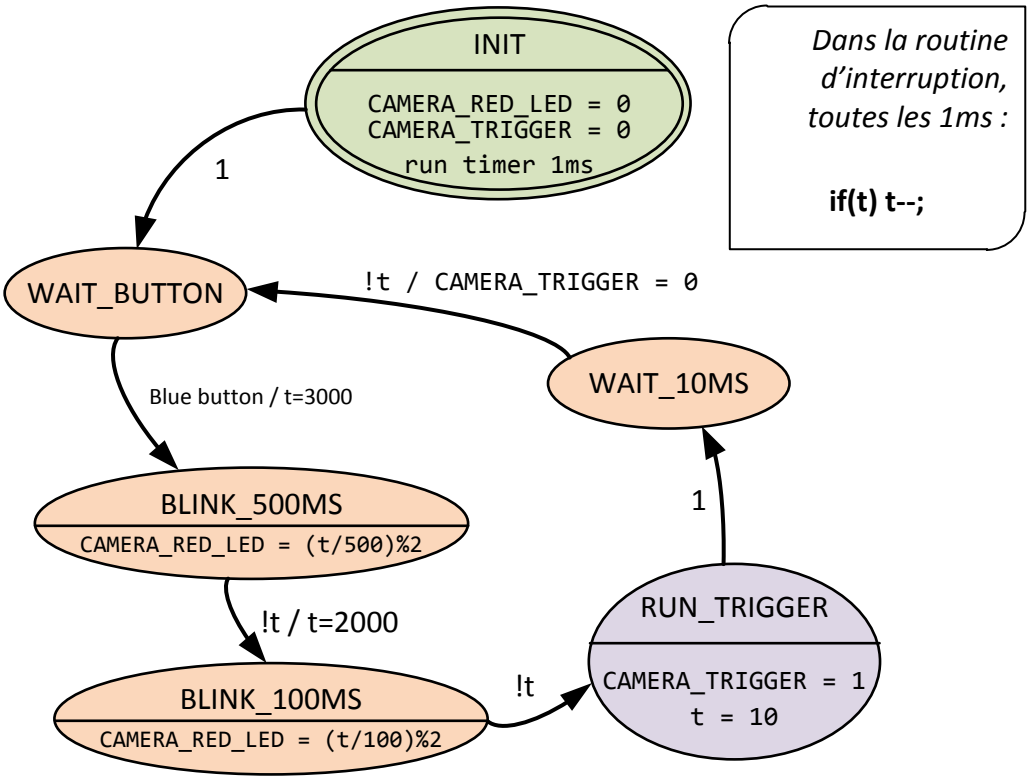
- Utiliser un périphérique timer, via les fonctions déjà écrites dans la mission n°3.
- Coder une machine à états dont la conception est fournie.

Sur le campus, vous avez à votre disposition le guide de la machine à états qui présente un exemple représentatif de codage d'une machine à états en C.

Les étapes à suivre :

- Pour avoir une vue d'ensemble, dessinez le chronogramme de l'application.
- Dans appli.c, rédigez l'ossature de la machine à états (typedef enum, switch, transitions)
- Dans chaque état, appelez les fonctions concernant le GPIO et le timer
- Ajoutez votre routine d'interruption `TIMER2_user_handler_it_1ms()` dans appli.c
- Les indications « CAMERA_RED_LED = … » sont à remplacer par des appels de fonctions...
- Testez, corrigez, débuggez l'application

Voici **une** des nombreuses machines à états répondant au cahier des charges :



Comment déclarer une variable partagée entre tâche de fond et interruption ?

A la façon d'un minuteur de cuisine, on utilise ici une variable t que l'on charge à la durée souhaitée, et qui est décrémentée chaque ms par la routine d'interruption.
Cette variable est partagée entre la tâche de fond et la routine d'interruption.
Pour cette raison, on la définit ainsi au début du fichier appli.c :

```
static volatile uint32_t t = 0;
```

static signifie lorsqu'il est ajouté devant la déclaration d'une variable globale que cette variable n'appartient qu'à ce fichier. Elle ne peut pas être connue des autres fichiers.
volatile indique qu'à chaque accès à cette variable, le compilateur ne doit pas prendre de raccourci en utilisant les registres du processeur: il doit accéder à la variable en mémoire, là où elle se trouve. Ainsi, on évite le risque de mauvaises interprétations du compilateur qui pourrait optimiser le code produit.