TP - Projet - ARM - EPITA Promo 2020

Le TP suivant est à réaliser durant (et entre) les séances. Il est dimensionné pour être réalisé en 5 séances. Les deux dernières séances auront pour but de préparer le projet, dont la soutenance sera sanctionnée par une note. Le travail s'effectuera par groupe de 3 maximum.

Les différents niveaux du TP sont organisé par ordre croissant de difficulté et complexité, mais restent relativement indépendants. Les réaliser proprement est fortement conseillé, car ils serviront à complèter la note du projet.

L'ensemble du code des différents niveaux sera à rendre à l'issue de la dernière séance. Le code du projet sera à rendre la veille de la soutenance (donc le 8 Juillet), par email: olivier1.dufour@epita.fr.

Pré-requis

Les outils suivants devront avoir été installés:

- la toolchain arm-none-eabi (`apt-get install gcc-arm-none-eabi`)
- gdb (`apt-get install gdb-arm-none-eabi`)
- binutils (`apt-get install binutils-arm-none-eabi`)
- st-link utilities
- STM32CubeMX

Vous devrez venir en TP avec une carte Nucléo F401 par groupe.

Niveau 1: prise en main

Génération d'un binaire

Le but de ce premier niveau est de mettre en place le code minimal pour faire fonctionner le microcontroleur. Pour fonctionner, votre microcontroleur n'a besoin que d'une stack et d'un point d'entrée. Pour ce faire, lorsqu'il exécute la routine de reset, il va charger le premier mot de la flash dans son registre `sp`, et le deuxième mot de la flash dans son registre `pc` (ce qui équivaut à appeler la fonction pointée par le 2e mot de la flash).

Vous noterez qu'il doit y avoir un espace entre le code et le début de votre binaire. Cette place est nécessaire, et on abordera plus tard comment correctement l'utiliser. Vous pouvez pour l'instant vous contenter d'aligner le début de votre code sur 512 octets.

Une fois que vous avez pu réaliser un binaire de cette forme, il est important qu'il soit situé à la bonne adresse physique. En effet, votre code va être exécuté

directement depuis la flash, aussi le code doit être relogé en flash. Pour rappel, l'adresse de la flash est `0x8000000`.

Pour réaliser cela, vous allez devoir écrire un script LD. Ce script va à la fois décrire l'organisation mémoire de votre microcontroleur, et ordonner le binaire. Il va permettre en particulier de déterminer dans quel ordre les sections sont insérées dans le binaire final.

Votre linker script devrait ressembler à ceci (code non contractuel):

```
MEMORY
{
    FLASH (rx): ORIGIN = 0x8000000, LENGTH = 512K
    RAM (rwx): ORIGIN = 0x20000000, LENGTH = 96K
}

ENTRY(reset_handler)

SECTIONS
{
    .text :
    {
        *(.text)
        . = ALIGN(4);
        _etext = .;
    } > FLASH

    .bss :
    {
        __bss_start__ = .;
        *(.bss)
        __bss_end__ = .;
    } > RAM

    /DISCARD/ :
    {
        *(.ARM.exidx*)
    }
    PROVIDE(_heap = __bss_end__);
    PROVIDE(_heap_end = _stack - _stack_size);
    PROVIDE(_stack_size = 1024);
    PROVIDE(_stack = ORIGIN(RAM) + LENGTH(RAM));
}
```

Pour plus d'informations, référez vous à la documentation de LD.

Une fois votre script correctement reseigné, il ne vous reste plus qu'à compiler votre code. Le binaire que vous allez générer n'est pas, comme sur votre

distribution linux favorite, un ELF, mais bien un binaire. Il va falloir rajouter une étape à votre chaine de compilation, car `gcc` ne vous fournira qu'un ELF en sortie. Pour ce faire, vous allez utiliser `objcopy`.

Voici un exemple de compilation d'un programme très simple:

```
    [CC] main.c
arm-none-eabi-gcc -Os -gdwarf-2 -mthumb -fno-builtin -mcpu=cortex-m4 -Wall -std=c99 -ffuncti
    [LD] test.elf
arm-none-eabi-gcc main.o -nostartfiles -Wl,-gc-sections -Tscript.ld -L. --specs=nano.specs -
    [OBJCOPY] test.bin
arm-none-eabi-objcopy -O binary -S  test.elf test.bin
```

Pour vérifier que votre binaire est bien formaté, il suffit de regarder son contenu:

```
hexdump -C test.bin | head
00000000  00 80 01 20 75 26 00 08  73 26 00 08 73 26 00 08  |... u&..s&..s&..|
00000010  73 26 00 08 73 26 00 08  73 26 00 08 00 00 00 00  |s&..s&..s&......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 73 26 00 08  |............s&..|
00000030  73 26 00 08 00 00 00 00  73 26 00 08 bd 26 00 08  |s&......s&...&..|
00000040  73 26 00 08 73 26 00 08  73 26 00 08 73 26 00 08  |s&..s&..s&..s&..|
*
00000080  89 28 00 08 99 28 00 08  73 26 00 08 00 00 00 00  |.(...(..s&......|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 73 26 00 08  |............s&..|
000000a0  73 26 00 08 73 26 00 08  73 26 00 08 73 26 00 08  |s&..s&..s&..s&..|
*
```

Programmation et debug

Une fois votre `test.bin` généré, il vous faut le charger dans la flash du micro-controleur. Pour ce faire, on va utiliser une sonde JTAG. Si vous êtes sur une carte Nucléo, vérifiez que les deux cavaliers de CN2 sont fermés. Sur la discovery, c'est le connecteur CN4.

Pour programmer la carte, il suffit d'exécuter la commande suivante:

```
st-flash write test.bin 0x8000000
```

Une fois le code chargé, il se lance automatiquement.

Cependant, vous n'êtes à ce stade pas vraiment sûr que votre code tourne réellement (sauf si vous avez déjà implémenté un effet visuel sur la LED !), et la meilleure solution pour s'en assurer reste d'utiliser `gdb`.

Là encore, il faut avoir fait le branchement de la sonde JTAG, et lancer le serveur gdb comme suit:

```
st-util
```

Puis vous pouvez vous y connecter avec gdb, en prenant bien soin de fournir l'ELF pour avoir les symboles de debug:

```
arm-none-eabi-gdb test.elf
(gdb) tar ext :4242
(gdb) b main
(gdb) c
```

Si vous êtes bien arrêté dans votre main, vous avez atteint l'objectif du premier niveau.

Niveau 2: système avancé

Pour le deuxième niveau, on va utiliser un générateur de code pour configurer le système, et faire l'initialisation des périphériques.

L'objectif est de réaliser un interrupteur avec le bouton de votre carte, et de l'utiliser pour allumer la LED de votre carte.

A l'aide du Cube, créez un projet dans lequel vous allez configurer:

- le pin du bouton comme un GPIO_Input
- le pin de la LED comme un GPIO_Output
- l'horloge de référence de manière à avoir HCLK à sa valeur maximale
- la toolchain Makefile

Chaque pin du micro-contrôleur a deux numéros: le numéro de pin sur le package, et son numéro de GPIO couplé à son port. Par exemple, la pin 57 sur le package LQFP64 est la pin B5 (GPIO port B, numéro 5). Il est inutile de s'occuper du numéro du pin sur le package, cette information n'intéresse que l'électronicien qui va fabriquer la carte. En termes de programmation, vous devez utiliser la nomenclature port + numéro.

Vous pouvez générer du code depuis le cube, qui contient l'initialisation des GPIO et des horloges, ainsi que les bibliothèques nécessaires pour les manipuler.

Il ne vous reste plus qu'à gérer le comportement du bouton.

Niveau 3: timers et interruptions

Une des raisons pour lesquelles on se tourne vers les microcontrôleurs, c'est pour tirer partie d'un système de temps précis. En effet, la plupart des microcontrôleurs sont équipés de plusieurs timers indépendants de l'horloge système, et on est donc capable d'avoir des programmes qui vont répondre plus efficacement en temps, quelle que soit la charge du CPU.

Dans ce niveau, on va voir différentes utilisations des timers, aussi bien pour des applications directes que pour du debug.

Pour commencer, on va s'intéresser à un timer particulier: le watchdog. Ce timer a un comportement unique dans le système: il effectue un reset du microcontrôleur quand il expire. Aussi il sert à s'assurer que le système fonctionne toujours correctement.

Vous allez activer le watchdog dans votre routine d'initialisation du système (avant ou après la configuration des horloges), et le rafraîchir à un endroit clé de votre programme. Ce faisant, votre carte ne rebootera que si vous ne passez pas assez régulièrement dans la fonction de rafraichissement. Attention cependant, il ne faut pas l'appeler partout et tout le temps, sinon son utilité s'en retrouve amoindrie.

Notez que via les registres de `DBGMCU`, vous pouvez configurer le Watchdog pour s'arrêter quand un breakpoint est actif. Cela vous évitera d'avoir un reset lors de vos séances de debug.

Une fois le watchdog activé et testé, vous pouvez passer à l'étape suivante: activer un timer classique. Vous pouvez encore une fois utiliser le cube pour générer le code d'initialisation.

Ce timer va vous servir à piloter la LED de la carte. A chaque expiration du timer, vous allez inverser l'état de la LED. Si vous règlez la fréquence à 2 Hz, vous devriez la voire clignoter. Pour ce faire vous devez:

- activer l'interruption de votre timer
- renseigner votre routine d'ISR dans la table des vecteurs
- configurer le timer pour qu'il émette une interruption lors qu'il expire
- configurer le timer pour qu'il se relance sans intervention

La gestion des interruptions est dans ce cas précis très simple, vous n'avez qu'à gérer le signal qui est émis par le timer. Votre routine d'ISR n'est donc qu'une fonction qui va aller piloter une GPIO, et mettre à jour le status d'un timer.

Une fois cette étape validée, vous allez paramètrer un deuxième timer avec la même fréquence, mais en mode PWM. Vous devrez donc:

- choisir un timer qui expose une sortie sur un pin accessible du MCU
- configurer le timer pour qu'il se relance sans intervention
- configurer le duty cycle à 50%

Attention, la fréquence du PWM devra être deux fois plus petite que celle du timer.

En branchant une LED avec une résistance en série sur la sortie du PWM, on la verra clignoter à la même vitesse que la LED qui se trouve sur la carte. IRQ, WDG et timer

Niveau 4: communication UART

Maintenant que vous êtes capables de traiter une interruption, vous allez pouvoir implémenter une interface de communicatio. La plus simple qui existe est l'UART. Vous allez implémenter une interface UART avec les caractéristiques suivantes:

- full duplex
- 115200 baud
- connectée au port COM de la ST-Link
- utilisation des interruptions pour détecter un caractère entrant

Pour cela, vous pouvez utiliser le cube pour la configuration. Référez vous à la documentation de votre carte pour savoir quel UART utiliser, et quels pins sélectionner.

Le premier programme à réaliser sera un simple écho sur la console. Une fois cela fait, vous pourrez implémenter un printf fonctionnel, et ainsi utiliser l'UART comme moyen de debug.

Pour finaliser le niveau (et améliorer les performances de votre implémentation), vous allez réaliser un programme qui va lire une chaine de caractères, et retourner son hash (SHA-256). La taille de la chaine ne sera pas connue à l'avance. Votre programme devra afficher le résultat sur la console.

Niveau 5: bootloader et mise à jour

Jusqu'à présent, vous avez toujours utilisé une sonde JTAG pour charger votre code. Dans ce niveau, vous allez voir comment vous en passer, et utiliser l'UART pour charger votre code.

La première étape consiste à faire un programme qui est capable de recevoir un binaire par l'UART, d'en vérifier le hash et de l'écrire en flash. Pour cela vous allez devoir implémenter un driver pour la flash du MCU. Notez bien qu'il est impossible de programmer un secteur sur lequel du code est en cours d'exécution, aussi l'adresse d'écriture du binaire devra se trouver dans un autre secteur que votre bootloader. Il faudra ajouter un mécanisme pour exécuter le 2e programme qui se trouve plus loin en flash.

Une fois cette étape validée, vous allez devoir faire un nouveau linker script qui vous permettra de former un binaire qui ne se trouve pas au début de la flash, mais à une adresse arbitraire.

Vous n'avez plus qu'à charger ce binaire via l'UART, et valider son exécution. Notez bien que votre bootloader devra exécuter ce programme à chaque démarrage, pas uniquement lors de la mise à jour par UART.

La dernière étape consistera à protéger en écriture votre bootloader. En effet pour éviter de l'effacer par erreur, il est fortement conseillé de vérrouiller en écriture les secteurs de flash dans lesquels il se situe. C'est en configurant les

options de la flash que vous pourrez faire ceci. Référez vous au manuel de votre MCU sur les options possibles.

Attention, quand vous allez modifier les options de la flash, ne touchez pas au niveau de protection (RDP), au risque de rendre votre flash impossible à reprogrammer.

Projet final

Le projet à réaliser reprend en grande partie ce qui a déjà été fait dans les différents niveaux, et donne surtout un cadre applicatif aux différentes notions abordées.

L'objectif est de réaliser un appareil de signature RSA.

Le MCU sera chargé avec un bootloader, qui aura les fonctionnalités suivantes:

- vérifie que le logiciel à démarrer est authentifié (signé)
- met à jour le logiciel à démarrer de manière sécurisée (i.e. on ne dois pas pouvoir mettre un binaire arbitraire en flash)
- démarre le logiciel en flash s'il est valide

Le logiciel à démarrer aura les fonctionnalités suivantes:

- génère une clé privée RSA (une et une seule fois)
- exporte sur l'UART la clé publique associée
- signe un hash (sha-256) fourni via l'UART avec la clé privée
- protège l'utilisation de la clé privée par un mot de passe
- aucune opération n'est possible tant qu'un mot de passe n'a pas été configuré

Vous devrez proposer un protocol UART pour utiliser les 4 fonctionnalités ci-dessus.

Bonus possibles (si vous avez du temps et/ou l'envie):

- ajouter une fonctionnalité de gestionnaire de mot de passe
- utilisation de l'interface USB à la place de l'interface UART (en implémentant la classe CDC)
- proposition de sécurisation des données secrètes