Temps réel STM32

Travaux pratiques IMERIR 2019

- **Rappels de vos cours**
- **Contraintes d'un système embarqué :**
  - Faible encombrement en taille, en poids.
  - Autonome en énergie et en interaction utilisateur.
  - Sûr et sécurisé, le cycle de fonctionnement peut être long (24h, 1 an)
  - Compromis de coût : budget hardware (8 bits ? 32 bits ? Linux ? OS ?) et budget de développement (combien de personnes, sur combien de temps)
- **Contraintes système temps réel :**
  - Le respect du temps dans l'exécution d'un traitement est aussi important que le résultat du traitement en lui-même.

- **Prendre en main un micro-contrôleur depuis 0**.
- Votre but : **Acquérir des réflexes** :
  - Comment démarrer sur une plateforme
  - Lire et écrire du code pour un micro-contrôleur embarqué
  - Intégrer un OS temps réel et une carte capteur à ce micro-contrôleur
  - Les travaux sont découpés en étapes à réaliser
- Mon but : Aider à prendre en main cette plateforme.

- Le matériel à votre disposition :

- **STM32L476 Nucleo-64**

  - Un micro-contrôleur low-power STM32L476

  - Une carte intégré ST-Link : celle-ci gère le flashage, le debuggage et l'alimentation du micro-contrôleur

- **Carte capteur accéléromètre / gyroscope**

  - Attention : 2 références de cartes, notez bien votre référence !

- **Les ressources pour débuter :**
  - Datasheets STM32L476, Manuel STM32L476 et User manual STM32L476 Nucleo64
  - Installer GCC pour votre ordinateur : https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads
  - Récupérer STM32 Cube L4 (sur le Moodle de l'école). Ce package comprend les drivers, les fichiers CPU, le code de startup.
  - Datasheets / manuels de la carte capteur.

- **Votre travail** :
  - Le plus simple, le mettre sur GitHub, d'autres idées ?
  - Envoyez-moi votre compte GitHub : baptiste@tikam.tech
  - Mettez dans une section commentaire de votre code ce que fait votre code et les réponses aux éventuelles questions.
  - Découpez bien votre code avec le nom des étapes dans le nom des fonctions (ou en commentaires) pour que je puisse vous aider et vérifier votre code !
  - Plus vous me clarifiez votre travail (par un readme, par des commentaires), plus je peux vous aider !
  - Avancez à votre rythme.
  - Hésitez pas à créer un projet par étape si vous jugez que cela améliore la clarté.

- **Etape 1** : Réussir un compiler le code le plus simple au monde…

- Pour créer un fichier firmware, plusieurs étapes sont nécessaires : compilation du .c vers un fichier objet (format .o), puis link de ces fichiers objet vers un fichier exécutable (en général format ELF) et transformation de ce fichier ELF vers un format compréhensible par le micro-contrôleur.

- Lorsque vous compilez pour un OS Linux / Windows / UNIX, l'étape de transformation est inexistante : l'OS se charge de décoder et exécuter ce fichier. Mais lorsque vous programmez directement au niveau CPU, cela n'est pas possible : il n'y a aucun système pour charger cet exécutable.

- Pour créer un fichier objet (.o), le compilateur a besoin de plusieurs choses : votre code source (réparti ou non dans plusieurs fichiers), les options de compilation correct pour la plateforme (calcul flottant émulé ou non, type de CPU, où trouver les includes externes, où trouver les répertoires des sources…)

- Une fois compilé en fichier objet (.o), il faut à nouveau indiqué au linker les options (où trouver les libs externes, type de CPU, comment placer les différentes sections de code / data dans le micro-contrôleur). A la fin de cette étape vous avez un fichier au format exécutable ELF.

- Enfin dernière étape, lancer la phase de transformation du fichier ELF produit par le linker au format du micro-contrôleur.

- Dernière étape, écrire ce fichier firmware dans la mémoire flash du micro-contrôleur.

- **Etape 1** : Tester votre chaîne de compilation avec le Makefile fourni avec le code le plus simple (3 lignes de code maximum !).
  - Utiliser le projet Makefile de base (sur le Moodle de l'école)
  - Vérifier l'installation de la chaîne de compilation et du STM32 Cube L4

- Notes sur les fichiers disponibles :
  - Dans le projet par défaut (cf Moodle de l'école), plusieurs fichiers sont disponible : Le Makefile, le fichier .ld et un fichier .h de définition.
  - Le fichier .ld a pour but d'indiquer au linker où placer les différentes parties de votre code : les données constantes, les données à initialiser, le code en lui-même, le code de démarrage, où commence la RAM, où placer la pile, etc…
  - Sur les OS Linux / UNIX / Windows, ce fichier .ld est masqué et beaucoup de partie sont dynamique : L'OS s'occupe d'attribuer à la volée les zones de RAM et de code.

- **Etape 2 et étape 3 :** Lire une entrée (bouton) ou écrire une sortie (LED) de la carte (voir slide suivant).

- Chaque Entrée / Sortie d'un micro-contrôleur doit être configuré en 2 étapes.

  - Une étape pour indiquer ce que fait physiquement cette broche du micro-contrôleur : est-ce une broche configuré en output ? est-ce une broche UART RX ? est-ce une broche SDA du bus I2C ?

  - Une seconde étape de configuration de ce bloc fonctionnel. Si on a définie précédemment GPIO output, alors il faut activer et configurer le bloc GPIO.

- Il faut jongler avec les différentes documentation du micro-contrôleur. Le **manuel** vous donnent la description fonctionnelle des blocs. Le **datasheet** vous donnent le mapping des GPIO. Enfin le **manuel de la carte** de dev vous donnent la position physique des GPIO.

- **Etape 2 :** Allumer la LED puis la faire blinker à une fréquence, à 1 Hz par exemple.
    - Faites bien attention d'écrire sur la bonne sortie ! Ne copier pas du code « qui fait blink » pour une autre carte !
- **Etape 3 :** Lire l'état du bouton pressé / relâché de la carte. La lecture n'est pas dangereuse par rapport à l'écriture sur une sortie de l'étape 2.
    - Lorsque vous appuyez sur le bouton, allumez la LED.

- **Etape 3** : **IRQ**
  - Interrupt Request : événement généré par le CPU pour interrompre l'exécution en cours et lancé un « interrupt handler » (qui est du code à vous ou d'un OS)
  - IRQ peut être logique, lié au CPU directement : Division par 0, memory fault, instructions fault…
  - IRQ peut être matériel : sur réception données UART, sur event timer, sur fin d'envoie UART…, sur front montant GPIO…
  - Le but : permettre de répondre à des événements externes sans avoir à les check manuellement en permanence (éviter le polling et de consommer du CPU / énergie pour rien)
  - Mais : comme on interrompt la séquence normal du programme, on considère que l'interupt handler doit être concis pour revenir le plus rapidement possible dans le déroulement normal du programme :
    - Dans un système temps réel, on casse le respect du temps !
    - Le temps passé dans le code IRQ et le nombre d'IRQ / seconde est à considérer dans un système temps réel !
    - Le code de l'interrupt handler ne doit pas être bloquant ou attendre une autre IRQ…

- **Etape 3** Lire l'état du bouton pressé / relâché de la carte
  - Modifier votre code pour activer une IRQ sur le bouton :
    - Lorsque on presse le bouton, une IRQ est lancée
    - Lorsque on relâche le bouton, une IRQ est lancée
    - Lorsque on presse et / ou on relâche le bouton, une IRQ est lancée pour chaque événement.

- **Etape 4 :** Envoyer la chaine de caractère « Hello World ! » sur le port série de debug de la carte
  - La carte dispose d'un port série virtuel : la carte ST-Link relie le port série virtuel (en USB vers votre ordinateur) au port série physique du micro-contrôleur.
  - Utiliser l'API UART pour envoyer cette chaîne de caractère.
  - Utiliser un programme série coté ordinateur (screen, TeraTerm, …) pour vérifier le succès de l'envoie.
  - Seconde étape, utiliser la fonction printf pour simplifier le debug de votre programme.
  - Est-ce que printf affiche quelque chose sur le port série ?
    - Implémenter une solution pour le printf
  - Faites un printf à 2Hz avec la LED pour vérifier le fonctionnement.

- **Etape 5** : Activer le Watchdog du micro-contrôleur
  - Le watchdog est un des éléments central pour la robustesse d'un firmware.
  - Fonctionnement : si le watchdog n'est pas reset par votre code, alors le watchdog va reset le micro-contrôleur.
    - Il faut placer le reset watchdog dans la boucle critique de votre programme.
    - Cela permet de vérifier si votre code tourne toujours.
  - Si le micro-contrôleur ne redémarre jamais par le watchdog, alors on peux considérer votre firmware comme « fiable ».
  - Le reset du watchdog ne doit jamais être placé dans une IRQ ou dans un timer.
    - Pourquoi ?
  - Activer le watchdog mais ne faites pas de reset watchdog dans votre code
    - Comment vérifier au démarrage de votre code, que le reset viens du watchdog ?
      - Implémenter une solution pour indiquer que le reset du micro-contrôleur viens du watchdog (et ne viens pas du reset normal).

  - Créer un projet comprenant les étapes précédentes : le blink de LED, l'IRQ bouton, le printf et le watchdog pour vérifier que tout fonctionne ensemble.

- **Etape 6 :** Debugger depuis votre ordinateur votre code source
  - La méthode la plus simple : printf
  - Seconde méthode, avec GDB :
    - Installer OpenOCD pour votre plateforme (brew, apt-get)
    - Compiler firmware en mode gdb (cf Makefile) et le copier sur la carte
    - Lancer OpenOCD : *openocd -f /usr/local/share/openocd/scripts/board/st_nucleo_l476rg.cfg*
    - Lancer le debugger: *arm-none-eabi-gdb build/<firmware-name>.elf*
    - Se connecter sur la carte : *target remote localhost:3333*
    - Placer le firmware arrêter, en reset: *monitor reset halt*
    - **A vous les joies de GDB en ligne de commandes :** placer un breakpoint, y aller, faire du pas à pas, faire du pas à pas sans aller dans une sous fonction, executer une fonction et quitter une fonction en cours, lister les breakpoints
    - *Note : J'ai validé cette seconde méthode sur mon ordinateur, si cette étape est trop longue à mettre en place (> 1 heure), indiquez moi bien les soucis et passez à l'étape suivante.*

- **Etape 7 :** L'un des avantages des micro-contrôleur est de pouvoir contrôler finement leur consommation électrique. Différent mode de veille existe sur la gamme cortex-m (du plus simple à programmer au plus complexe)

    - Mettre le micro-contrôleur en SLEEP (à l'aide de l'instruction __WFI) et le réveiller à intervalle régulier (1 Hz ?). Est-ce que le réveil fait perdre l'état de votre programme ? Comment vérifier que votre programme ne repart pas de 0 ?

    - Mettre le micro-contrôleur en très basse consommation et le réveiller par le bon moyen (par exemple à l'aide du bouton présent sur la carte de dev). Est-ce que le réveil fait perdre l'état de votre programme ?

    - Si l'état de votre programme est perdu, comment peux-on conserver l'état précédent ? Regarder si il existe une solution dans le hardware du micro-contrôleur et tester là.

    - Vous pouvez mettre dans une section commentaire de votre code ce que fait votre code et les réponses à ces questions.

- **Etape 8 :** Intégrer un OS temps réel : **FreeRTOS**
  - Programmation « bare metal » : aucun OS utilisé, c'est votre code qui gère la boucle principale d'exécution.
  - Programmation avec un RTOS : gestion de la concurrence par un OS (thread, mutex, semaphore, event, signal…)
    - OS peux fournir plus de services que uniquement la concurrence : driver, framework réseaux, mise à jour, gestion mémoire…
  - FreeRTOS fournit uniquement la gestion de la concurrence
    - Vous pouvez le voir dans le nombre de fichiers de FreeRTOS : moins de 10 fichiers sources !

- **Etape 8 :** Intégrer un OS temps réel : **FreeRTOS**
  - Télécharger le code source officiel FreeRTOS sur freertos.org
    - Version actuelle : FreeRTOSv10.2.1
  - Modifier votre Makefile pour intégrer FreeRTOS
    - 9 fichiers sources à ajouter au Makefile
    - 1 fichier de config à créer dans votre projet
    - Inspirez vous d'un projet démo dans le répertoire FreeRTOS (par exemple CORTEX_M4F_STM…)
  - Tester la création d'un thread avec un blink de LED à 1 Hz
  - Ajouter une gestion de la concurrence pour le printf de l'étape 4 :
    - Que se passe-t-il si 2 threads font un printf en même temps ?
    - Essayer de déclencher le bug avec 2 threads (voir avec plus de threads !).
    - Corriger l'implémentation pour supporter le multi-thread avec le printf
  - Réveiller un (ou plusieurs) thread à partir de l'IRQ bouton de l'étape 3.
  - N'hésitez pas à utiliser les fonctionnalités de l'OS pour la suite des travaux.

- **Etape 9** : Intégrer un driver : Accéléromètre / Gyroscope.

  - Le matériel n'est pas plug-and-play : Débrancher l'alimentation puis brancher la carte fille

  - Le but : Lire les données de l'accéléromètre et les données du gyroscope depuis le micro-contrôleur.

  - Obtenir la documentation de la carte et des composants, puis obtenir les données accéléromètre / gyroscope par n'importe quel moyen.

  - Vérifier que les données récupérées sont correctes lors de mouvement « à la main ».

  - Il y a d'autres capteurs présent sur la carte : Récupérer les données de ces autres capteurs.

- **Etape 10 :** Intégration accéléromètre / gyroscope

  - Configurer la fréquence d'acquisition à 1000Hz et obtenir les données à cette fréquence.

  - Vérifier les autres paramètres des composants (fréquence ? filtre ? valeur min/max possible ?)

  - Quels paramètres peuvent être intéressant pour filtrer le signal ?

- **Etape 11 :** Intégration accéléromètre / gyroscope
    - Indiquer si la carte est posé horizontalement ou verticalement sur la table
    - Indiquer si la carte est en chute libre
    - Calculer la vitesse de rotation de la carte lorsque la carte est posé sur la table, à plat puis tourner à la main (toujours à plat).
    - Calculer les angles pitch / roll / yaw de la carte et vérifier les angles lorsque vous faites des tours complets

- **Etape 12 :** Intégration accéléromètre / gyroscope

  - Calculer la vitesse puis la distance à partir de l'accélération (on fait un mouvement en translation, à plat sur la table)

  - Tester expérimentalement cette intégration : Avez-vous des problèmes sur cette intégration ? Les données sont-elles juste ?

- **Etape 13 :** Lire / écrire dans la flash du micro-contrôleur

  - Émuler le fonctionnement d'une EEPROM, souvent utiliser pour stocker des paramètres de configuration d'un firmware.

  - EEPROM : accès octet par octet via une adresse, par exemple avec l'API suivante : *data = read_eeprom(address)* ou *write_eeprom(address, data)*

  - Le micro-contrôleur n'a pas d'EEPROM : Il faut donc utiliser un morceau de la flash pour émuler cette EEPROM.

  - Implémenter une API d'accès à votre EEPROM :

    - *uint8_t read_eeprom(const uint32_t address);*

    - *void write_eeprom(const uint32_t address, const uint8_t data);*

- **Etape 14 :** Protocole UART

  - Envoyer les données accéléromètre / gyroscope vers votre ordinateur sur le port série.

  - Protocole à respecter : 1 octet d'en tête (0x5A), 1 octet du type de paquet, 1 octet indiquant la taille des données (sans compter l'en-tête, le type et la taille), <data dépendante du type de paquet>, 1 octet de checksum

  - checksum calculé sur l'ensemble du paquet (depuis l'en-tête jusqu'à la partie data) (simple : crc += data[i])

  - Valider ce protocole à l'aide d'un outil à développer sur votre ordinateur.

  - Type de paquet :
    - micro-contrôleur vers PC : 0x01 -> données accel / gyro
    - PC vers micro-contrôleur : 0x02 -> Enable en continue l'envoie de données, 0x03 -> Disable l'envoie en continue de données, 0x04 -> Demander l'envoie d'un seul paquet de données accel / gyro

- **Etape 15 :** Outil de reporting PC

  - Développer un outil pour récupérer les données du protocole précédent et les afficher dans un premier temps en texte

  - Et second temps, afficher les courbes de l'accéléromètre / gyroscope / angle / autres capteurs

  - Vous êtes libre d'utiliser le langage / librairie graphique de votre choix : C++ avec Qt, Python…

  - Configurer l'accéléromètre / gyroscope à haute fréquence (1000Hz) et vérifier si vous parvenez à tenir le débit entre votre ordinateur et le firmware.

- **Etape 16 : Bootloader**
    - Un firmware spécialisé dans la mise à jour du firmware principale d'une carte électronique. Composant logiciel critique !
    - Par défaut, un micro-contrôleur est livré avec un bootloader constructeur écrit en ROM : On ne peux ni le changer, ni le modifier. Ce bootloader par défaut se lance en général sur l'UART ou l'USB (DFU mode).
    - Problème du bootloader constructeur :
        - Une tierce partie, en faisant un reset physique peux lire ou écrire notre firmware.
        - On est bloqué sur les périphériques par défaut ou par le protocole du bootloader constructeur.
    - Solution : Implémenter notre propre bootloader. La séquence de démarrage va donc changer : Le bootloader constructeur va lancer notre bootloader. Celui-ci va lancer notre firmware.
    - Implémentation bootloader :
        - Votre bootloader doit minimiser sa taille en flash : pas de printf, malloc / free, d'OS (sauf si vous arrivez à minimiser les fonctions de l'OS lui même)
        - Définir et implémenter un protocole UART de mise à jour entre votre ordinateur et ce bootloader
        - Implémenter les fonctions d'écriture en flash et le code pour exécuter un firmware
    - Implémentation programme de test :
        - Ecrire un firmware de test à flasher par votre bootloader (par exemple, Hello World par un printf)