Un environnement exécutif visant la compatibilité POSIX : NuttX pour contrôler un analyseur de réseau à base de STM32

G. Goavec-Mérou, J.-M. Friedt, 31 juillet 2017

1 Introduction

Nous avons récemment discuté de FreeRTOS [1] comme méthode de programmation ajoutant un niveau d'abstraction au-dessus d'une programmation en C brut (*baremetal*) sans induire de pénalités significatives quant aux performances et ressources utilisées, l'ordonnanceur étant relativement léger (les applications nécessitent moins de quelques KB pour fonctionner) pour un gain de productivité lié à la portabilité du code, la capacité à mettre en commun la contribution de plusieurs développeurs au travers des tâches et des mécanismes d'échange de données entre ressources partagées (queues, mutex), mais sans fournir aucun niveau d'abstraction sur l'accès aux périphériques matériels.

Entre FreeRTOS et un système d'exploitation complet tel que Linux (256 KB RAM pour elinux.org/images/c/ca/Spreading.pdf) ou *BSD, voir Inferno, exécuté sur plateforme embarquée, avec la capacité à charger dynamiquement un exécutable et les bibliothèques associées, un nouvel arrivant vient se positionner dans la course des systèmes visant la compatibilité POSIX, mais à ressources réduites : NuttX, nommé d'après son auteur Greg Nutt. Ce système, initialement développé entre 2004 et 2007, pour être distribué en licence BSD depuis cette date, supporte désormais une multitude de plateformes, notamment autour de l'architecture ARM qui équipe la gamme des STM32. Sa maturité et sa stabilité ont amené son utilisation par des grands groupes aussi prestigieux que Samsung (TinyAra à git.tizen.org/cgit/rtos/tinyara/ puis Tizen RT, wiki.tizen.org/Tizen_RT) et Sony (www.youtube.com/watch?v=TjuzH6JthxQ et www.youtube.com/watch?v=T8fLjWyI5nI) : nous pouvons donc penser que NuttX est amené à survivre quelques années. Bien que l'arborescence soit quelque peu confuse avec certaines redondances, l'investissement intellectuel pour un développeur sous GNU/Linux est réduit, puisque nous retrouverons nos appels systèmes familiers de la compatibilité POSIX, et une architecture des pilotes fortement ressemblante à celle des modules noyau Linux.

Pourquoi présenter NuttX dans Linux Magazine ? Gregory Nutt décrit son logiciel comme "*Think of NuttX as a tiny Linux work-alike with a much reduced feature set*". Notre première rencontre avec cet environnement de travail a été arrangée par PX4, la suite logicielle (*firmware*) équipant de nombreuses centrales inertielles pilotant drones et autres aéronefs autonomes, en particulier au travers de la plateforme Pixhawk (pixhawk.org). Au-delà de l'effet de mode, nous ne pouvons que nous féliciter de voir proliférer des systèmes embarqués communiquant (IoT) basés sur des environnements de développement libres qui permettront de réduire le nombre de trous de sécurité que la communication sur liaison radiofréquence ne manquera certainement pas de mettre en évidence, et de faire auditer les codes exécutés sur IoT par la communauté.

Nous nous proposons de continuer à travailler [1] sur une plateforme aux ressources réduites architecturée autour du plus petit STM32F4 disponible chez Farnell pour un coût modique (ref 2725150 par exemple, 3,52 euros/p pour 10 composants), mais le lecteur moins électronicien pourra expérimenter sur la plateforme prête à l'emploi Nucleo-F410 (13,38 euros chez le même fournisseur). Notre contribution au projet NuttX tient au support du STM32F410 et en particulier renseigner la liste des interruptions, le support de la configuration de la carte Nucleo-F410 qui nous a permis de maîtriser les divers fichiers nécessaires à supporter une nouvelle plateforme, quelques corrections de bugs et la mise en œuvre d'exemples applicatifs faisant appel aux diverses couches d'abstractions, du pilote avec appel des ressources mises à disposition pour gérer un périphérique sur bus SPI, jusqu'à l'utilisation des fonctions mises à disposition dans le norme POSIX, et en particulier les *threads*. Ces divers points seront abordés en détail dans la prose qui va suivre.

Notre objectif reste, comme dans le projet sous FreeRTOS [1], de caractériser les propriétés spectrales, en phase et en amplitude, d'un dispositif radiofréquence inconnu (nommé DUT – *Device Under Test*, dans la suite du document). Pour ce faire (Fig. 1), nous concevons un analyseur de réseau numérique, comprenant une source radiofréquence Analog Devices AD9834 programmée en SPI et cadencée par une horloge issue d'un *timer* (signal MCLK – *Master Clock*) fourni par le microcontrôleur. Les mesures se font par deux convertisseurs analogique-numériques, l'un connecté à un détecteur de puissance LT5537, l'autre à un détecteur de phase assemblé autour d'un comparateur suivi d'une porte logique XOR et un filtre passe bas. Le schéma et routage d'un circuit dédié est proposé à github.com/jmfriedt/tp_freertos/tree/master/FreeRTOS-Network-Analyzer-on-STM32/examples (format Eagle). L'objectif de cette présentation est donc d'accéder à ces périphériques au travers d'une couche d'abstraction qu'est la notion de pilote, qui doit éviter la lecture de la documentation technique (*datasheet*) si les développeurs ont convenablement fait leur travail, et proposer une application qui se charge de configurer chaque périphérique selon les conditions requises par la mesure. Nous ne nous contenterons pas de rester au niveau utilisateur – qui s'apparente au programmeur sous GNU/Linux – mais proposerons deux pilotes afin de nous familiariser avec la programmation au niveau du noyau. En effet, nous verrons que NuttX ne fournit pas d'interface vers le bus SPI mais impose d'écrire un pilote représentant le composant connecté au bus, pilote qui fera lui même appel à la couche d'abstraction du bus SPI fournie par le système d'exploitation (nuttx.org/doku.php?id=wiki:nxinternal:devices-vs-buses).

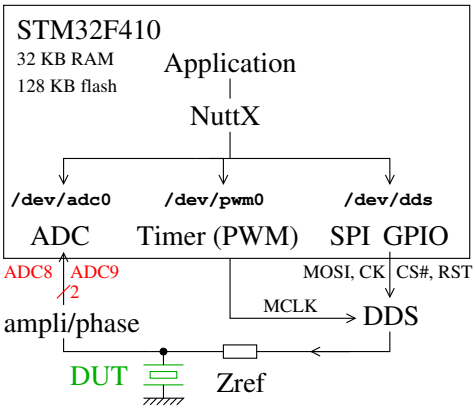

Figure 1: Schéma de principe de l'analyseur de réseau numérique, avec les divers périphériques mis en œuvre sur le microcontrôleur pour accéder aux fonctionnalités nécessaires à caractériser le dispositif radiofréquence inconnu DUT (*Device Under Test*).

2 Matériel

NuttX supporte une multitude de processeurs et de plateformes. Notre intérêt se porte sur le STM32F4, donc nous commençons par aborder le problème de l'installation et l'exécution de NuttX sur une plateforme facilement accessible, la STM32F4-Discovery (référence 2506840 chez Farnell pour une peu moins de 20 euros, décrite à www.st.com/en/evaluation-tools/stm32f4discovery.html) pour se familiariser avec le processus de configuration et de compilation. Les sources de NuttX s'obtiennent en clonant par `git` le dépôt à bitbucket.org/nuttx/nuttx.git. Au même niveau de l'arborescence que les sources de NuttX, nous clonons le dépôt des applications qui sera lui aussi nécessaire à la compilation : bitbucket.org/nuttx/apps.git. Enfin, nous aurons besoin des outils KConfig disponibles à ymorin.is-a-geek.org/projects/kconfig-frontends pour configurer NuttX. Notre choix se porte sur la chaîne de compilation croisée que nous maîtrisons bien et compilons à partir des sources au moyen du script disponible à github.com/jmfriedt/summon-arm-toolchain qui fournit `arm-none-eabi-gcc` comme compilateur pour les cibles ARM Cortex M3 et M4.

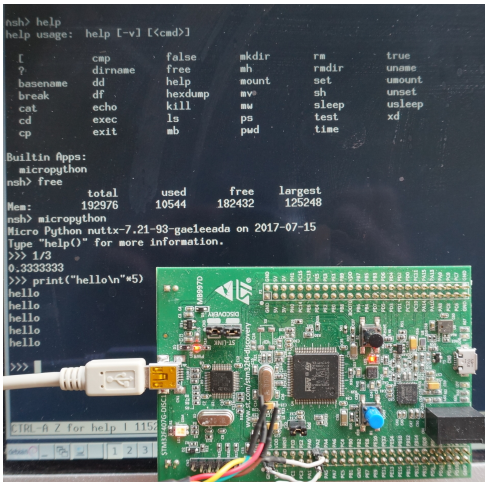

Figure 2: Exécution de NuttX sur le STM32F407 de la carte STM32F4Discovery pour rapidement se familiariser avec l'environnement de développement. La liaison avec le PC se fait avec un convertisseur USB-série sur le port2 du microcontrôleur (PA2, PA3). Cette configuration est obtenue par `stm32f4discovery/nsh`. Nous avons ajouté `Application Configuration` → `Interpreters` → `Micro Python support` qui nécessite pour compiler `Library Routines`→ `Standard Math library`. Seuls 10 KB de RAM sont nécessaires pour exécuter le NuttX Shell `nsh`, un peu plus pour Python.

Nous adaptons l'environnement de compilation à notre configuration. Dans un premier temps, initialiser une certaine *plateforme* avec une certaine *configuration*. En effet, nous constatons dans le répertoire de la *plateforme* `configs/stm32f4discovery` qu'un certain nombre de configurations existent : `nsh` (le NuttX SHell), `netnsh`, `usbnsh`, `rgbled` ... Chacun de ces répertoires contient un script de configuration `defconfig` sélectionnant

diverses fonctionnalités de la même plateforme STM32F4-Discovery. La configuration s'obtient par `./tools/configure.sh stm32f4discovery/nsh` depuis le répertoire racine de NuttX. Fournir un nom de configuration erronée se traduira par une liste de toutes les configurations connues.

À partir d'ici, la configuration se poursuit comme classiquement avec `buildroot` ou le noyau Linux : nous remontons au sommet de l'arborescence et configurons par `make menuconfig`. En particulier, nous prendrons soin que de nombreux projets sont configurés pour compiler sous Cygwin, incompatible avec un travail sous GNU/Linux. Pour remédier à ce problème, `Build Setup` → `Build Host Platform` → `Linux`, et `System Type` → `Toolchain Selection` → `Generic GNU EABI toolchain under Linux` (le second point est la configuration par défaut une fois la plateforme hôte Linux sélectionnée).

`make` se charge de compiler l'image `nuttx.bin` que nous flashons sur la carte au moyen de l'outil libre de communication avec le protocole ST-Link disponible à github.com/texane/stlink.git :
`sudo st-flash write nuttx.bin 0x8000000` (alternativement, `openocd` est aussi capable d'une telle opération, en plus de faire office de serveur `gdb`). Le résultat est un shell fonctionnel, accessible en 115200 bauds sur l'UART2 du STM32F407 (Fig. 2).

Pour exécuter notre projet (github.com/jmfriedt/tp_nuttx/tree/master/stm32_dds/) sur la carte Nucleo-F410, quelques modifications aux fichiers de configuration de la plateforme sont nécessaires :

- soit souder un quartz 20 MHz sur l'emplacement prévu à cet effet, soit sélectionner comme source d'horloge l'oscillateur interne RC, cadencé à 16 MHz, par `#define STM32_BOARD_USEHSI 1` dans le fichier de configuration de l'horloge de la plateforme dans `include/stm_dds.h`. On prendra soin de modifier les coefficients de la PLL en accord,
- adapter l'UART pour la communication, de 1 à 2 dans `f410-nsh/defconfig`,
- on pourra tester avec un oscilloscope les transactions SPI sur broches MOSI et CK par activation du pilote accessible à `/dev/dds`, mais une mise en œvre pratique du DDS nécessitera de connecter un tel circuit aux broches associées à SPI1, ainsi que la source d'horloge (PWM du timer 1) et du CS# (PA4),
- finalement, les deux voies d'ADC que nous lisons sont `PB8` et `PB9`, toutes deux disponibles pour une connexion de signaux analogiques sur la Nucleo-F410.

Toute modification de la configuration par défaut se pérennise par `make savedefconfig` qui crée un fichier `defconfig` dans le répertoire racine de NuttX, qui pourra ensuite être copié dans le répertoire de configuration de la plateforme sous `configs`.

3 Ajout du support d'une variante de STM32

Nos efforts se sont portés sur le STM32F410, un des microcontrôleurs de la gamme ST avec les ressources les plus réduites (et donc parmi les moins chers). De nombreux STM32 sont supportés, y compris ceux munis du cœur Cortex M4 de la gamme STM32F4. La ressemblance paronymique entre STM32F411, pleinement supporté par NuttX, et STM32F410, est trompeuse. En effet, ces deux modèles diffèrent par la taille de la RAM, de la flash, du boîtier et par le type et le nombre de périphériques disponibles. Autant dire que leur seul point commun est le cœur Cortex M4. Cet ensemble de différences nécessite de modifier la table des interruptions, de fournir les informations correspondantes à la RAM et les capacités offertes par ce composant.

Ce travail d'ajout d'une variante repose sur la modification de plusieurs fichiers :

- `arch/arm/include/stm32/stm32f40xxx_irq.h` et `arch/arm/src/stm32/chip/stm32f40xxx_vectors.h` pour la description de la table d'interruption du composant. Comme nous le verrons, ces fichiers couvrent l'ensemble des STM32F4 en utilisant des règles de pré-processeur, ce qui rend ces fichiers quelque peu difficiles à lire, et sans doute à maintenir sur le long terme ;
- `arch/arm/src/stm32/Kconfig`, dans la philosophie des outils avec une configuration type *curses* : ce fichier permet à l'utilisateur d'activer des options, ou bien d'en sélectionner automatiquement. Il est donc nécessaire d'ajouter le support pour le F410 et pour notre modèle exact (le F410RB) ;
- `arch/arm/include/stm32/chip.h` qui décrit les caractéristiques, en terme de périphérique et de hiérarchie, d'une version particulière de microcontrôleur. Ce fichier semble partiellement redondant avec le `KConfig` ;
- `arch/arm/src/stm32/stm32_allocateheap.c` fournit des informations relatives à la mémoire disponible, et plus précisément à l'adresse de fin.

Dans cette section, nous allons limiter au maximum le code pour éviter une surcharge inutile : le code du patch accepté par NuttX pour le support sur STM32F410, représentatif du travail réalisé, est disponible à bitbucket.org/nuttx/nuttx/commits/02535be36a9abfa8c526e0a3d84f1306d42bf5c6). Nous n'allons pas, par ailleurs, détailler plus la modification du fichier `stm32_allocateheap.c`, le contenu du patch et l'explication ci-dessus suffisent à cet aspect.

La partie la plus importante (sur un aspect de nécessité mais également de complexité) est la modification des fichiers relatifs à la table d'interruption. Le fichier `arch/arm/src/stm32/chip/stm32f40xxx_vectors.h` définit la table des vecteurs d'interruptions pour l'ensemble des STM32F4. Il est intégré au moment de la compilation dans le fichier assembleur `arch/arm/src/stm32/gnu/stm32_vectors.S` et placé dans la zone ad-hoc de la mémoire du microcontrôleur.

Le fait que ce travail soit complexe n'est pas spécialement lié au composant en lui-même mais plus à l'approche de NuttX (historique ?). Le fichier en question est une suite relativement longue (la partie du fichier définissant le vecteur d'interruptions fait $\approx$ 250 lignes pour, au plus, produire 98 lignes une fois passé dans le pré-processeur) de définitions conditionnelles au niveau pré-processeur, pour lier, ou non, une interruption à une ISR. Ceci s'explique par le fait que selon le modèle du STM32, sa taille de boîtier, et les fonctionnalités, certains numéros d'interruption peuvent être utilisés ou pas.

Il faut, donc, pour chaque interruption, vérifier que la valeur fournie en l'état est compatible avec le STM32F410 (la liste est présentée pages 197 à 201 de [2]). Dans la situation contraire deux cas sont possibles : un test est déjà présent pour un autre modèle, le code doit être adapté, si besoin, afin d'inclure notre modèle. Ceci se fait par l'ajout d'un `defined(CONFIG_STM32_STM32F410)` à la suite de ceux déjà présents.

L'autre cas apparaît si le F410 présente une particularité vis-à-vis des autres modèles et qu'aucun test n'est présent pour cette interruption. Dans cette situation, il est nécessaire d'ajouter la variante (et donc des `#if`, `#else` et `#endif`). Un exemple présenté dans le code 1 correspond à l'interruption 16+39 (les 16 premières interruptions sont liées au cœur M4, les autres sont laissées libres pour le fondeur). Cette interruption est liée à l'USART3 présent sur l'ensemble des STM32F4 sauf le 410. À l'origine seule la ligne 3 était présente. La macro `VECTOR`, définie dans le fichier `stm32_vectors.S`, lie une numéro d'interruption (la 55) avec une fonction (ici `stm32_uart3`), définie dans le pilote du périphérique, qui sera appelée lors d'un événement sur ledit périphérique. Le code proposé lors de l'intégration présente maintenant un test pour différencier le F410 des autres et utilise la macro `UNUSED` qui ne prend que la constante `STM32_IRQ_RESERVED39`.

```
1#if defined(CONFIG_STM32_STM32F410)
 UNUSED(STM32_IRQ_RESERVED39)              /* Vector 16+39: Reserved */
3#else
 VECTOR(stm32_usart3, STM32_IRQ_USART3)    /* Vector 16+39: USART3 global interrupt */
5#endif
```

Listing 1 – gestion de l'interruption 16+39, liée à l'USART3 sur tous les modèles supportés, sauf le F410.

La constante utilisée pour le F410 n'étant pas, à l'origine, présente dans NuttX, nous avons du la créer et faire un travail équivalent de test au niveau du fichier `stm32f40xxx_irq.h`, tel que présenté dans le code 2.

```
1#if defined(CONFIG_STM32_STM32F410)
 #define STM32_IRQ_RESERVED39    (STM32_IRQ_FIRST+39) /* 39: Reserved */
3#else
 #define STM32_IRQ_USART3        (STM32_IRQ_FIRST+39) /* 39: USART3 global interrupt */
5#endif
```

Listing 2 – Déclaration de constantes relatives aux addresses d'interruptions.

La dernière partie de nos contributions concerne l'intégration réelle du processeur dans NuttX. Jusqu'à présent, nous avons ajouté des informations relatives à la table d'interruption et à la taille de la RAM mais NuttX et son mécanisme de construction basé sur des `kconfig` et des `Makefile` n'a, en l'état, aucune notion de l'existence de ce processeur, ni des périphériques disponibles.

Cette partie se passe dans le fichier `arch/arm/src/stm32/kconfig`. Il est d'une part nécessaire d'ajouter le support du processeur (le STM32F410) et d'autre part la variante qui nous intéresse (le F410RB). Il y a finalement peu de choses à dire, le contenu du patch étant suffisant, car ce n'est que de la sélection (mot clé `select`) de variables cachées pour décrire les périphériques présents. Cette série de variables est utilisée pour autoriser l'utilisateur à activer telle ou telle instance des périphériques disponibles.

Concernant le fichier `arch/arm/include/stm32/chip.h`, le but de l'ajout est la désactivation des autres modèles de processeurs, et la description du composant en terme de nombre d'entités de chaque

type de périphériques. L'intérêt de ce fichier nous semble particulièrement discutable pour deux raisons : le type du STM32 passe par un menu dont les options sont mutuellement exclusives. Ainsi, si un modèle est choisi, les autres versions sont logiquement désactivées. La seconde raison est relative au nombre d'instances de chaque type de périphérique : dire qu'il y a trois USART n'est pas pertinent puisque dans le cas du F410 ce sont les USART1, 2 et 6 qui sont présents. Pire, toujours pour ce périphérique, une correction du pilote est nécessaire sous peine de crash car un tableau de structures est initialisé selon cette taille mais les instances sont indicées en dur, donc l'USART6 sera pointé par la sixième case d'un tableau de trois éléments... Les corrections sont triviales à proposer mais nécessitent une analyse globale pour s'assurer de ne pas introduire d'effets de bords ou de régressions.

4 Ajout d'une nouvelle plateforme

Mis à part ces modifications sur les vecteurs d'interruption (maintenant intégrées dans NuttX), les derniers paramètres dépendant de chaque déclinaison de processeur sont la vitesse de cadencement de l'horloge et la carte mémoire. Le premier point est simple : chaque plateforme est configurée dans le répertoire `configs` avec un nom de circuit sous lequel on trouvera un répertoire comportant un nom de configuration. Plusieurs configurations peuvent être proposées pour un même circuit, par exemple avec plusieurs listes d'applications plus ou moins gourmandes en ressources ou en périphériques.

Nous avons choisi de cadencer notre microcontrôleur avec un quartz à 20 MHz (afin de l'overclocker au-delà de 100 MHz, une décision discutable pour une application industrielle mais parfaitement valable pour un unique montage) : où trouver cette information et quelle est la conséquence sur les registres de configuration des horloges ? Chaque plateforme contient `include/plateforme.h` avec `plateforme` adapté au nom de la plateforme. Ce fichier comporte les macros qui définissent la fréquence du quartz (`STM32_BOARD_XTAL`), dont on déduit la fréquence de cadencement du cœur comme $SysClk = VCO/P = (Quartz/M \times N)/P$ en tenant compte que l'oscillateur contrôlé en tension du multiplieur de fréquence par vérouillage de phase (PLL) doit être compris entre 192 et 432 MHz. La procédure habituellement proposée par ST est de définir M tel que $Quartz/M = 1$ MHz, puis N pour amener $Quartz/M \times N \in [192; 432]$ MHz. L'étape d'oveclocking tient au dernier facteur de division P qui devrait alimenter le processeur avec une cadence en dessous de 100 MHz, mais que nous fixons à 2 de façon à ce que avec $M = 20$ et $N = 280$, nous obtenions 140 MHz. Tous nos programmes ont bien fonctionné dans ces conditions. Nous finissons par définir la constante `STM32_SYSCLK_FREQUENCY` à `140000000ul` : toutes les autres horloges – en particulier APB1 et APB2 qui cadencent les périphériques matériels – vont se déduire de cette horloge maîtresse : ce sera un point fondamental lors de l'utilisation des timers et du bus SPI.

Trois fichiers suffisent pour porter NuttX à une nouvelle plateforme : la fichier de configuration `include/plateforme.h` pour l'horlge, la configuration de NuttX dans `configuration/defconfig` qui initialisera le contenu de `make menuconfig`, et `scripts/*.ld` pour la carte mémoire telle que nous avons l'habitude de la définir par l'option `-T` de `ld` lors de son appel par `gcc`. Cependant, nous devons prendre soin de noter qu'en plus de la configuration de l'éditeur de lien, NuttX veut lui aussi connaître la configuration mémoire de la plateforme : nous devons renseigner une seconde fois ces mêmes informations dans le fichier de configuration sous forme de `CONFIG_RAM_START=0x20000000` et `CONFIG_RAM_SIZE=32768`.

Nous ne modifions pour le moment aucun des codes sources inspirés de plateformes existantes dans `src` ni `include/board.h` qui contient les macros de périphériques auxquels nous ferons appel plus tard. Nous reprenons la procédure de compilation vue auparavant, à savoir revenir au sommet de l'arborescence de NuttX, initialiser l'environnement par `./tools/configure.sh plateforme/configuration` en adaptant `plateforme` au nom de notre plateforme et `configuration` à la configuration que nous désirons appliquer. Nous pouvons nous convaincre de l'application de notre nouvelle configuration par `make menuconfig` – en particulier on vérifiera que `Build Setup` → `Build Host Platform` → `Linux` puisqu'un certain nombre d'exemples sont configurés pour Cygwin, et que `Board Selection` → `Select target board` contient bien le nom de notre plateforme. Enfin, tout comme dans le cas de la carte STM32F4Discovery, l'utilisation de `arm-none-eabi-gcc` nous induit à définir `System Type` → `Toolchain Selection` → `Generic GNU EABI toolchain under Linux`. Alternativement, l'option `-l` de `./tools/configure.sh` force l'hote à GNU/Linux.

Finalement, `make -j8` ne prend que quelques dizaines de secondes pour compiler NuttX et, si tout se passe bien, se conclut par

```
LD: nuttx
make[1]: Leaving directory '[...]/nuttx/arch/arm/src'
CP: nuttx.hex
CP: nuttx.bin
```

avec les deux images de la mémoire non-volatile prêtes à être transférées au microcontrôleur par son outil favori. Une fois l'image flashée et le microcontrôleur réinitialisé, lancer `minicom` en 115200 bauds nous accueille avec le prompt du NuttX shell `nsh>`. Un certain nombre de commandes unix sont supportées, dont celles de gestion des fichiers on d'analyse de la mémoire disponible :

```
nsh> free
             total       used       free    largest
Mem:         20416       7200      13216      13216
```

NuttX tient donc largement dans la mémoire mise à disposition par le STM32F410. La liste des commandes supportées et applications liées au noyau s'obtient par `help`. Même `ps` est disponible, sous réserve d'avoir activé le pseudo-système de fichiers (`CONFIG_FS_PROCFS=y` dans le `.config`) lors de la configuration, et l'avoir monté (`mount -t procfs /proc`) :

```
nsh> mount -t procfs /proc
nsh> ps
  PID PRI POLICY   TYPE    NPX STATE    EVENT     SIGMASK   STACK COMMAND
    0   0 FIFO     Kthread N-- Ready              00000000 000000 Idle Task
    1 100 FIFO     Task    --- Running            00000000 002028 init
```

Nous voilà prêts à passer au cœur du problème : développer.

5 Mon premier pilote

NuttX s'exécute sur STM32 sans le périphérique matériel qui garantit la séparation de l'espace mémoire du système de l'espace utilisateur, le gestionnaire de mémoire (MMU). Ainsi, l'utilisateur a le droit d'accéder à toute zone mémoire, y compris les registres contrôlant le matériel. Tout comme en programmation C bas niveau ou sous uClinux, nous serions donc en droit d'accéder au matériel en écrivant directement dans un pointeur dont l'adresse vise les registres matériels : ce serait une approche qui romprait avec l'intérêt d'un système d'exploitation d'élever le niveau d'abstraction, fournir des interfaces standardisées pour éviter aux autres programmeurs de relire la datasheet pour chaque nouveau processeur, et garantir l'intégrité de l'accès aux ressources sous le contrôle du noyau. Nous allons donc dès maintenant nous familiariser avec la structure d'un pilote noyau et son intégration dans l'arborescence de la plateforme. Contrairement à Linux, NuttX ne supporte pas le chargement dynamique de pilote (ni d'applications d'ailleurs, différence entre l'environnement exécutif et le système d'exploitation) : l'appel au pilote se fera dans le fichier d'initialisation de la plateforme `void board_initialize(void)` de `src/stm32_boot.c`. Cette première exploration du répertoire `src` nous permet de constater qu'il contient tous les codes d'initialisation spécifiques à chaque plateforme : les noms de fichiers n'ont pas d'importance tant qu'ils sont référencés dans la variable `CSRCS` du `Makefile`, et qu'un des fichiers contient la fonction d'initialisation `board_initialize()` pour initialiser le shell.

Une fonction de démarrage, de nom arbitraire, est appelée dans le fonction d'initialisation de la plateforme pour charger le pilote. Nous y décrivons un point d'entrée dans le pseudo-système de fichier accessible dans `/dev`, et comme pour les pilotes de type caractère sous Linux, nous avons une structure de données `file_operations` qui associe une fonction à chaque appel système POSIX associé aux fichiers. Dans notre exemple. le pilote commence par

```
1#include <nuttx/config.h>
 #include <nuttx/arch.h>
3
 #include <stdio.h>
5
 typedef FAR struct file    file_t;
7
 static int      dds_open(file_t *filep);
9static int      dds_close(file_t *filep);
```

```
static ssize_t dds_read(file_t *filep, FAR char *buffer, size_t buflen);
static ssize_t dds_write(file_t *filep, FAR const char *buf, size_t buflen);

static const struct file_operations dds_ops = {
  dds_open,   /* open */
  dds_close,    /* close */
  dds_read,   /* read */
  dds_write,    /* write */
  0,      /* seek */
  0,      /* ioctl */
};
```

La fonction `board_initialize()` sera appelée si la configuration de la plateforme – le fichier `defconfig` dans notre configuration `f410-nsh` – active l'option `CONFIG_BOARD_INITIALIZE=y`. Le périphérique dans `/dev` est créé par la fonction d'initialisation du pilote

```
void stm32_dds_setup(void)
{(void)register_driver("/dev/dds", &dds_ops, 0444, NULL);}
```

qui est appelée par `board_initialize(void)`.

Enfin, les fonctions implémentant les divers appels systèmes et décrites dans la `file_operations` sont implémentées

```
static int dds_open(file_t *filep)  {printf("Open\n");return OK;}
static int dds_close(file_t *filep) {printf("Close\n");return OK;}
static ssize_t dds_read(file_t *filep, FAR char *buf, size_t buflen)
{int k;
 if (buflen>10) buflen=10;
 for (k=0;k<buflen;k++) {buf[k]='0'+k;}
 printf("Read %d\n",buflen);
 return buflen;
}
static ssize_t dds_write(file_t *filep, FAR const char *buf, size_t buflen)
{printf("Write %d\n",buflen);return 1;}
```

pour donner un exemple complet et fonctionnel. Après ajout des fichiers sources de ce pilote dans le `Makefile` de `src` par `CSRCS += stm32_dummy.c` et compilation, nous trouvons bien dans `/dev/` notre nouveau point d'entrée avec lequel nous pouvons jouer. Seule subtilité : autant écrire par `cat` dans le périphérique se traduit bien par la séquence open-write-close comme sous Linux, autant la lecture ne peut pas se faire par `cat` mais doit se faire par `dd` auquel nous précisons la taille du bloc de données à lire (*block size* `bs`) et le nombre de blocs de données (`count`).

```
nsh> ls -l /dev
/dev:
 cr--r--r--       0 adc0
 crw-rw-rw-       0 console
 crw-rw-rw-       0 dummy
 crw-rw-rw-       0 null
 crw-rw-rw-       0 ttyS0
nsh> echo "1" > /dev/dummy
Open
Write 3 - 1
Close
nsh> cat < /dev/dummy
nsh: cat: open failed: 2
nsh> dd if=/dev/dds of=/dev/console bs=5 count=1
Open
Read 5
01234Close
```

6 Utilisation des pilotes fournis par NuttX

Nous sommes donc capables d'écrire notre propre pilote en vue d'implémenter une interface de type fichier pour contrôler un périphérique spécifique, que nous verrons être le synthétiseur de fréquences (*Direct Digital Synthesizer* – DDS). Ce composant se situe sur le bus SPI, qui est supporté par NuttX, et se voit activé par une broche Chip Select (`CS#` pour indiquer son activation par passage au niveau bas). NuttX interdit d'écrire un pilote décrivant un bus, mais nous devons nous limiter à des pilotes décrivant un composant, c'est à dire un bus et le signal d'activation qui lui est associé. Avant d'attaquer

le problème du bus SPI, il nous faut cependant cadencer le DDS : pour ce faire, une sortie périodique cadencée par un timer fera l'affaire.

6.1 Utilisation du timer du STM32

NuttX supporte une large variété de modes de fonctionnement de timers et permet de router les signaux résultant sur les diverses broches du microcontrôleur. Pour notre part, le cœur du processeur cadenceé à 140 MHz doit pouvoir alimenter un timer cadencé à la même vitesse, que nous réinitialisons un coup sur deux, soit une sortie à 70 MHz. Selon les consignes décrites à nuttx.yahoogroups.narkive.com/SRAmkJMt/stm32-pwm, nous activons ce périphérique dans `src/stm32_boot.c` par

```
pwm=stm32_pwminitialize(1); // timer1/output1
if (!pwm) lederr("ERROR: Failed to get the STM32 PWM lower half\n");
else
  {pwm_register("/dev/pwm0", pwm);
   info.frequency = 100000;
   info.duty = (50<<16); // en % decale' de 16 bits
   pwm->ops->setup(pwm);
   pwm->ops->start(pwm, &info);
  }
```

qui fournit un nouveau point d'entrée `pwm0` dans `/dev` afin de configurer la fréquence et le rapport cyclique de la PWM. En effet, NuttX demande à ce que tout périphérique soit inactif au démarrage du système, et soit activé explicitement au lancement de l'application. La liste des `ioctl` supportés par la PWM s'obtient par exemple en consultant le code source de l'application `apps/examples/pwm`. Depuis l'espace utilisateur, ces `ioctl` sont appelés par l'application NSH `pwm -f 1000000 -t 20 -d 50` pour une fréquence de 10 MHz et un rapport cyclique de 50%, en ayant pris soin d'activer l'exemple `pwm` dans `make menuconfig` puis `Application Configuration` → `Examples` → `Pulse width modulation (PWM) example`.

6.2 Utilisation d'un composant sur bus SPI du STM32

Nous avons mentionné qu'un bus ne doit pas être accessible au travers d'un pilote mais que ce sont les composants sur un bus qui sont représentés (http://nuttx.org/doku.php?id=wiki:nxinternal:devices-vs-buses explique que "*There will never be an application accessible interface to SPI. If your application were to use SPI directly, then you would have have embedded a device driver in your application and would have violated the RTOS functional partition*"). Ainsi, NuttX fournit les outils pour initialiser et utiliser le bus SPI, ainsi qu'une macro pour activer et désactiver le signal d'activation du périphérique CS#, que nous exploitons dans un pilote susceptible de communiquer avec le DDS AD9834.

NuttX propose d'initialiser le bus SPI dans un des 4 modes possibles définissant l'état au repos de l'horloge et le front d'horloge sur lequel échantillonner les données (paramètres habituellement nommés CPOL et CPHA). Se tromper de mode est garantie de dysfonctionnement de la communication. Ainsi, l'initialisation du bus ressemble à

```
stm32_spidev_initialize();
spi=stm32_spibus_initialize(1);

(void)SPI_LOCK(spi, true);
SPI_SETMODE(spi, SPIDEV_MODE2); // CK idle high & mesure falling edge
SPI_SETBITS(spi, 8);
(void)SPI_HWFEATURES(spi, 0);
printf("SPI wanted freq: 2000000, actual freq:%d\n",SPI_SETFREQUENCY(spi, 2000000));
```

la fréquence d'horloge ne peut pas satisfaire toutes les valeurs possibles puisque seules des divisions par puissance de 2 de l'horloge du processeur sont accessibles (Fig. 3). Nous renvoyons donc la fréquence réellement configurée lorsque la demande de l'utilisateur est déraisonnable, comme c'est le cas ici.

L'utilisation du bus SPI s'obtient par

```
SPI_SELECT(spi, SPIDEV_USER(0), true); // which component on the SPI bus to select
SPI_SEND(spi, (entree>>8)&0xff);
SPI_SEND(spi, (entree)&0xff);
SPI_SELECT(spi, SPIDEV_USER(0), false);
```

qui suppose qu'une fonction dont le nom est fixée par convention (cf `arch/arm/src/stm32/stm32_spi.c` qui fait le lien entre la méthode `.select` et la fonction `stm32_spi1select`) est définie dans la pilote :

```
#define GPIO_SPI_CS_DDS \
    (GPIO_OUTPUT | GPIO_PUSHPULL | GPIO_SPEED_2MHz | \
```

```
      GPIO_OUTPUT_SET | GPIO_PORTA | GPIO_PIN4)
4
void stm32_spi1select(FAR struct spi_dev_s *dev, uint32_t devid, bool selected)
6{spiinfo("devid: %d CS: %s\n", (int)devid, selected ? "assert" : "de-assert");
 stm32_gpiowrite(GPIO_SPI_CS_DDS, !selected);
8}
```

après avoir pris soin de placer `GPIO_SPI_CS_DDS` en sortie par `stm32_configgpio(GPIO_SPI_CS_DDS);` dans l'initialisation du pilote.

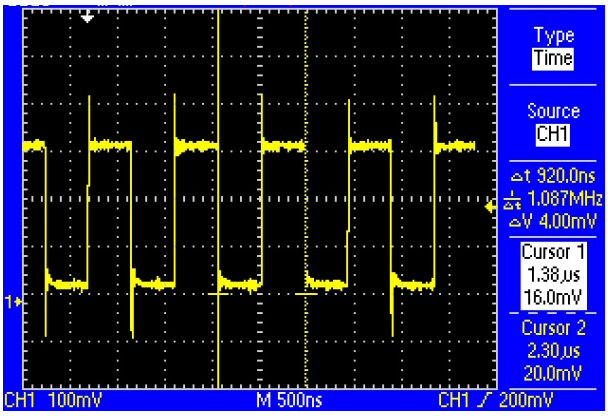

FIGURE 3 – Horloge du bus SPI pour une configuration requérant une fréquence de 2 MHz, inaccessible en divisant l'horloge à 140 MHz par une puissance de deux. NuttX fait le choix de diviser l'horloge à 140 MHz par 128 pour donner 1.09375 MHz (tel que annoncé comme valeur retournée par `SPI_SETFREQUENCY`), et non par 64 qui aurait donné une fréquence plus proche mais supérieure à celle voulue [2, p.717].

6.3 Utilisation d'un ADC du STM32

Les convertisseurs analogique-numériques (ADC) ne sont pas pris en charge par défaut. Il est donc nécessaire d'ajouter ce support au niveau de la plateforme en créant un fichier nommé par exemple `src/stm32_adc.c`.

Ce fichier définit un nouveau pilote qui sera accessible au travers de `/dev/adc0` : Le fichier contient :

```
#define ADC1_NCHANNELS 1                                    // nbre canaux lus
2static const uint8_t  g_adc1_chanlist[ADC1_NCHANNELS]={9}; // PB1 = ADC9
 static const uint32_t g_adc1_pinlist[ADC1_NCHANNELS] ={GPIO_ADC1_IN9};
4
int stm32_adc_setup(void)
6{struct adc_dev_s *adc;
 int ret,i;
8 for (i = 0; i < ADC1_NCHANNELS; i++)
     stm32_configgpio(g_adc1_pinlist[i]); // pins as analog inputs
10 adc = stm32_adcinitialize(1, g_adc1_chanlist, ADC1_NCHANNELS);
 if (adc == NULL)
12    {aerr("ERROR: Failed to get ADC interface\n");return -ENODEV;}
 ret = adc_register("/dev/adc0", adc);
14 if (ret < 0)
    {aerr("ERROR: adc_register failed: %d\n", ret);return ret;}
16 return OK;
}
```

Nous constatons qu'une seule voie est lisible (`ADC1_NCHANNELS`) à chaque appel de l'ADC, voie qui est définie dans le tableau `g_adc1_chanlist` et associée à la broche définie dans `g_adc1_pinlist` qu'on aura pris soin de configurer en entrée analogique en exploitant les constantes définies dans `arch/arm/src/stm32/chip/stm32f40xxx_pinmap.h`. Le premier paramètre de `stm32_adcinitialize` est le numéro de l'ADC initialisé : contrairement à ses grands frères qui possèdent 2 ADCs matériels et permettent deux conversions simultanées, le STM32F410 ne possède qu'un ADC matériel et nous n'avons d'autre choix que d'initialiser l'ADC numéro 1.

La fonction `stm32_adcinitialize` retourne une structure fournissant les méthodes de reset, l'ioctl, de configuration, ainsi qu'une structure spécifique aux ADCs des STM32s.

La dernière étape enregistre le périphérique auprès de la couche ADC de NuttX, en fournissant le nœud dans le système de fichier.

Cette fonction `stm32_adc_setup` est explicitement appelée au démarrage de la carte dans `board_initialize` de `stm32_boot.c` en ayant pris soin d'activer, dans `make menuconfig`, le support d'ADC1 : `System Type` → `STM32 Peripheral Support` → `[*] ADC1`. La couche Analog doit également être activée : `Device Drivers` → `[*] Analog Device(ADC/DAC) Support` → `[*] Analog-to-Digital Conversion`, `(8) ADC buffer size` et `[*] Do not start conversion when opening ADC device`.

Le bon comportement de l'ensemble se teste par l'application `adc` fournie par NuttX dont nous nous inspirerons par la suite pour notre propre utilisation de l'ADC : `Examples` → `[*] ADC example` → `(1) Number of Samples per Group` → `[*] Use software trigger`

L'exécution du programme une fois NuttX compilé et flashé s'obtient par `nsh> adc -n 10`. Sans le paramètre `-n`, l'application ne fournira un résultat qu'à son premier appel. Du point de vue applicatif, nous nous contentons de reprendre la séquence de mesure fournie dans `apps/examples/adc/adc_main.c` qui ouvre le descripteur de fichier en lecture, lance une conversion par l'ioctl `ANIOC_TRIGGER`, et lit le résultat de la conversion par un appel à `read()` sur le descripteur de fichier. Le pilote ADC renvoie une structure avec de multiples éléments de type `struct adc_msg_s` : la taille de la lecture est égale à la taille de cette structure multipliée par le nombre de canaux de conversion, 1 dans notre cas. Avec une seule conversion effectuée à chaque mesure, nous nous contentons de récupérer la première instance de cette structure renvoyée dans le pointeur fourni en argument à `read(fd,sample,num_read)` : `sample[0].am_data`. Cette solution est fonctionnelle mais ne satisfait pas notre exigence de mesurer l'amplitude *et la phase* du DUT lors de la mesure de l'analyseur de réseau. Comment mesurer deux grandeurs analogiques à travers l'unique interface proposée par NuttX ?

6.4 Utilisation de deux ADC du STM32

Lire une valeur d'un ADC est satisfaisant, mais ne convient pas à notre application qui doit lire deux valeurs de deux ADCs, l'un connecté au détecteur de puissance et l'autre au détecteur de phase, pour une caractérisation complète de la fonction de transfert du dispositif testé. Le STM32F410 ne possédant qu'un ADC matériel, nous ne pouvons pas configurer deux périphériques dans `dev`, puisque la deuxième configuration de ADC1 écrasera la première lors de l'initialisation du NuttX. La lecture de plusieurs canaux du même ADC passe nécessairement, sous NuttX, par l'utilisation de l'accès direct en mémoire (DMA) dans laquelle une série de conversions est programmée dans les registres de configuration de l'ADC, et un déclenchement unique de la séquence de conversion se traduit par le remplissage d'un tableau, dont nous avons fourni le pointeur, avec les valeurs issues de la conversion. Très pratique à utiliser, ce mode de conversion est quelque peu complexe à déverminer tant qu'il ne fonctionne pas. En particulier, nous devons identifier quel numéro et canal de DMA sont associés à quel périphérique. L'information est distribuée en divers emplacements de la documentation technique du STM32F410 [2] mais est résumée dans la section 8.3.3 (p.166) où nous apprenons que le canal 0 et flux (*stream*) 4 ou 0 de la DMA2 sont associés à ADC1. NuttX fournit toutes les configurations nécessaires, sous réserve de connaître les bonnes incantations à renseigner dans le fichier `defconfig` de configuration de la plateforme : l'activation des options `CONFIG_STM32_ADC1` (`System Type` → `STM32 Peripheral Support`) et `CONFIG_STM32_DMA2` (même sous-menu) validera automatiquement `CONFIG_STM32_HAVE_ADC1_DMA` (variable cachée activée dans le `Kconfig` si `select STM32_HAVE_ADC1_DMA if !STM32_STM32F10XX && STM32_DMA2`) qui autorisera l'accès à `CONFIG_STM32_ADC1_DMA` (`System Type` → `ADC Configuration` → `ADC1 DMA`) que nous sélectionnerons manuellement.

Pour mieux comprendre le cheminement qui nous a amené à cette réflexion, les divers codes sources que nous avons analysé sont

1. le pilote ADC de NuttX proposé dans `arch/arm/src/stm32/stm32_adc.c` définit la structure de données

```
static struct stm32_dev_s g_adcpriv1 =
#ifdef ADC1_HAVE_DMA
 .dmachan     = ADC1_DMA_CHAN,
 .hasdma      = true,
```

`#endif`

présentant une constante `ADC1_HAVE_DMA`. À force de recherche dans le code et dans l'interface *curse* nous avons pu déterminer l'ensemble des options à activer. La deuxième information fournie par cette portion est la constante `ADC1_DMA_CHAN`

2. celle-ci doit être définie dans le fichier de configuration `board.h` de la plateforme sous forme de `#define ADC1_DMA_CHAN DMAMAP_ADC1_1`. Donc pointant sur une autre constante spécifique à l'architecture
3. et finalement c'est dans `/arch/arm/src/stm32/chip/stm32f40xxx_dma.h` qu'est définie `#define DMAMAP_ADC1_1 STM32_DMA_MAP(DMA2,DMA_STREAM0,DMA_CHAN0)`

L'absence de documentation est donc bien palliée par l'aide de la recherche dans l'interface de configuration et un parcours exhaustif des codes sources : une souplesse avec les options de `grep` et `find` est indispensable pour s'en sortir.

7 Une nouvelle application faisant appel aux pilotes

Les pilotes ayant individuellement été validés comme fonctionnels depuis le shell, il ne reste qu'à écrire une application faisant appel à toutes ces fonctionnalités pour programmer le DDS, lire la valeur analogique, et répéter l'opération en balayant la fréquence de sortie du synthétiseur sur la plage analysée (Fig. 4). Nous nous inspirons pour ce faire de `apps/examples/adc` et `apps/examples/pwm`, ainsi que `apps/examples/gpio` après avoir activé le support de ce dernier périphérique `Device Drivers` → `IO Expander/GPIO Support` → `GPIO driver`. Ces exemples fournissent les divers `ioctl` associés à chaque pilote pour configurer chaque périphérique. Étant les auteurs de `/dev/dds`, la question de connaître l'action associée à chaque appel système ne se pose pas, puisque nous les avons définis. Nous avons choisi d'initialiser le DDS dans `open` (commutation du signal RESET et configuration du registre de contrôle à l'adresse 00), de définir la fréquence du signal généré par le DDS lors d'une écriture, et de renvoyer la fréquence courante lors d'une lecture. Aucun `ioctl` n'a été implémenté, bien que RESET pourrait faire partie de ces fonctionnalités. Les codes sources des diverses applications pour démontrer le bon fonctionnement de NuttX pour nos besoins sont accessibles à github.com/jmfriedt/tp_nuttx, dont on reliera le contenu au répertoire `apps` par des liens symboliques.

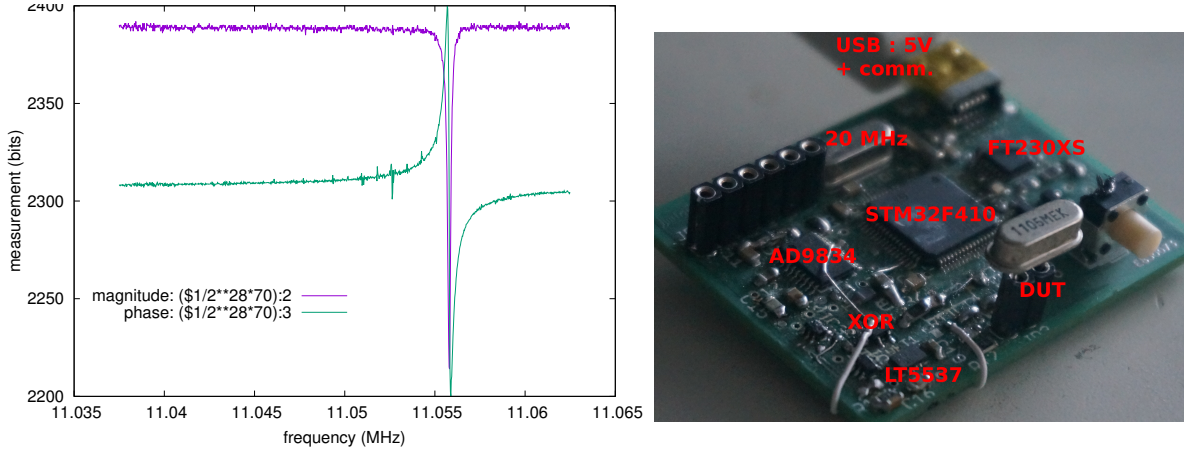

FIGURE 4 – Gauche : caractérisation d'un résonateur à quartz de fréquence nominale 11,05 MHz au moyen de l'analyseur de réseau décrit dans ce document. La phase et l'amplitude du signal sont toutes deux acquises sur deux voies de l'ADC. Droite : montage expérimental, avec une mesure de phase par porte XOR (le comparateur entre le DUT et le XOR est sous le circuit imprimé), et mesure d'amplitude par LT5537.

La création d'une nouvelle application est très simple : dans le répertoire `apps`, au même niveau que les sources `nuttx`, nous créons un répertoire qui contient les sources de notre programme applicatif. Dans ce répertoire, en plus du programme principal qui contient la fonction `main.c`, nous trouvons un fichier

de configuration `Kconfig`, un `Makefile` et un `Make.defs`. Le `Makefile` ne se présente pas, nous avons juste à renseigner le nom du programme principal `MAINSRC`, tandis que `Make.defs` reprend le nom du répertoire qui contient notre application et une variable de la forme `CONFIG_APP_XXX`, avec `XXX` le nom de notre application, qui indique si l'application est liée au noyau. Enfin, `KConfig` contient quelques descriptions du programme et constantes qui seront validées pour indiquer au système de compilation de lier notre application au noyau. On pensera finalement dans le `.config` à définir à `y` la variable qui déclenche la compilation de l'application, de la forme `CONFIG_APP_XXX`.

La lecture des paramètres et le découpage des arguments fournis sur la ligne de commande (fréquence de début de balayage, fréquence de fin et nombre de points de mesure) s'inspire elle aussi des exemples fournis dans `apps`, avec une boucle sur `argv` testant si le premier caractère est un symbole "-" puis considérant le caractère qui suit pour sélectionner le paramètre qui sera défini. La disponibilité des fonctions fournies par `stdio` et `stdlib`, habituellement proscrites lors du développement sur microcontrôleur compte tenu de la taille prohibitive de `newlib` et autres implémentations de `libc`, telles que `strtol()` pour convertir la chaîne de caractères en valeur numérique, rend le traitement des paramètres très simple.

8 Un OS sur microcontrôleur ... aspect utilisateur : `pthreads`

Nous avons vu que NuttX s'annonce comme un mini-Linux. Quelle fonctionnalité est plus emblématique de la compatibilité POSIX que les `pthreads`, l'implémentation POSIX de la capacité d'un programme à créer des fils qui partagent un même espace mémoire mais s'exécutent en parallèle (du point de vue de l'utilisateur) ? Le programme classique sous GNU/Linux démontrant le concept de threads et la gestion de la cohérence de l'accès à la mémoire partagée par les *mutex* est

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NTHREADS 10

void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;  // cette init a 0 ne suffit pas, il FAUT faire dans main

#ifdef CONFIG_BUILD_KERNEL
int main(int argc, char *argv[])
#else
int threadjmf_main(int argc, char *argv[])
#endif
{int d[10];
 pthread_t thread_id[NTHREADS];
 int i,n;
 counter=0;
 if (argc>1) n=strtol(argv[1],NULL,10); else n=NTHREADS;
 for (i=0; i < n; i++)
     {d[i]=i;
      printf("pthread_create%d: %d\n",i,pthread_create( &thread_id[i], NULL, →
          ↪thread_function, &d[i] ));
     } // CREATION DES n THREADS
 for(i=0; i < n; i++) {pthread_join(thread_id[i], NULL);} // ATTENTE MORT DES FILS
 printf("Final counter value: %d\n", counter);
 return(0);
}

void *thread_function(void *d)
{int lcount;
 pthread_mutex_lock( &mutex1 );
 lcount=counter;
 lcount++;
 printf("Thread number %d: %lx\n", *(int *)d, pthread_self());
 usleep(100000); // 100 ms
 counter=lcount;
 pthread_mutex_unlock( &mutex1 );
 return(NULL);
}
```

qui se compile sous GNU/Linux par `gcc thread_main.c -DCONFIG_BUILD_KERNEL -lpthread`. Pouvons nous compiler tel quel ce programme conçu pour GNU/Linux afin de l'intégrer à la liste des exécutables de NuttX ?

La réponse courte est évidemment positive, mais donne l'opportunité de se frotter à quelques originalités du développement sur environnement exécutif à faible empreinte mémoire. Premier point : la quantité de mémoire allouée à la pile de chaque thread. La granularité très fine de configuration de NuttX permet de définir dans le fichier `.config` la constante `CONFIG_PTHREAD_STACK_DEFAULT`. La valeur par défaut de 2048 ne permet de ne lancer que 4 threads sur le petit STM32F410, avant de se faire insulter par un message d'erreur 12 indiquant que la mémoire est épuisée (`ENOMEM` dans `include/errno.h`). Descendre sous 512 induit un comportement aléatoire, voire le crash du système. En restant conservateur à 1024 nous pouvons lancer jusqu'à 7 threads avant d'épuiser la mémoire. Second point : le programme n'est pas chargé en mémoire comme dans un système d'exploitation classique, mais est ici lié au noyau au moment de la compilation. Nous constatons qu'initialiser la variable globale `counter` lors de sa déclaration induit son incrément à chaque exécution du programme et non une réinitialisation comme nous nous y attendrions en travaillant sous GNU/Linux par exemple. Initialiser explicitement la variable globale `counter` à 0 en début de `main` s'avère nécessaire pour repartir dans des conditions vierges à chaque nouvelle exécution.

La démonstration de l'efficacité des `mutex` dans la gestion des variables partagées se fait comme classiquement : une variable globale est chargée en début de `thread` dans une variable locale, une attente simule un algorithme complexe qui nécessite cette copie de la variable locale, et finit en sauvant le résultat du calcul dans la variable globale. Ce comportement simule toute opération non-atomique qui doit dupliquer sur la pile locale le contenu de la variable globale.

avec mutex : réponse correcte

```
nsh> threadjmf 4
Thread number 0: b
pthread_create0: 0
pthread_create1: 0
pthread_create2: 0
pthread_create3: 0
Thread number 1: c
Thread number 2: d
Thread number 3: e
Final counter value: 4
```

sans mutex : réponse incorrecte

```
nsh> threadjmf 4
Thread number 0: 1a
pthread_create0: 0
Thread number 1: 1b
pthread_create1: 0
Thread number 2: 1c
pthread_create2: 0
Thread number 3: 1d
pthread_create3: 0
Final counter value: 1
```

trop de threads : 12=ENOMEM

```
nsh> threadjmf 9
Thread number 0: 1c
pthread_create0: 0
pthread_create1: 0
pthread_create2: 0
pthread_create3: 0
pthread_create4: 0
pthread_create5: 0
pthread_create6: 0
pthread_create7: 12
pthread_create8: 12
Thread number 1: 1d
Thread number 2: 1e
Thread number 3: 1f
Thread number 4: 22
Thread number 5: 23
Thread number 6: 24
Final counter value: 7
```

Sans protection par mutex, tous les threads s'exécutent en 100 ms (durée de l'attente) et incrémentent simultanément le compteur, résultant dans une valeur erronée de 1 en fin d'exécution de 4 threads qui devaient chacun incrémenter de une unité (cas du milieu). Au contraire en protégeant la fonction exécutée par les fils par mutex, l'exécution prend bien 400 ms et le résultat du compteur incrémenté séquentiellement par chaque fils atteint la valeur attendue de 4 (cas de gauche). Ce résultat est conforme à celui obtenu sous GNU/Linux. Lancer trop de threads se solde par l'incapacité à exécuter les deux derniers (erreur 12 de dépassement de mémoire) et là encore une valeur de compteur erronée en fin d'exécution (cas de droite).

9 Conclusion

Nous avons pris en main, en portant sur une nouvelle plateforme, l'environnement exécutif NuttX qui vise à fournir un environnement de développement compatible POSIX à très faible empreinte mémoire, puisque fonctionnel dans moins de 20 KB de RAM. Les nombreuses fonctionnalités fournies par NuttX, en particulier l'abstraction du matériel au travers de pilotes accessibles par les appels classiques à `read()`, `write()` et `ioctl()` dans `/dev`, facilitent considérablement la prise en main d'une nouvelle architecture matérielle, sous réserve d'être supportée.

Ce constat optimiste cache un problème lié au support d'une grande gamme de processeurs, et en particulier la multitude de déclinaisons du STM32 : le code devient un ignoble plat de spaghettis avec des options de préprocesseur `#ifdef` qui rendent sa lecture excessivement fastidieuse. Une refonte du code visant à séparer plus clairement les diverses déclinaisons du processeur, du même niveau que la refonte du support des SoCs imposée par Linus Torvalds lorsque le support de la branche ARM était devenu ingérable sans les *devicetree*s dans Linux, serait nécessaire pour garantir une évolution sereine de NuttX.

Références

[1] Q. Macé, J.-M Friedt, *FreeRTOS : application à la réalisation d'un analyseur de réseau numérique sur STM32*, GNU/Linux Magazine France **XXX** (2017)

[2] *RM0401 Reference Manual – STM32F410 advanced ARM-based 32-bit MCUs, Rev. 2*, ST Microelectronics (Oct. 2015)