TP 3 : Construction d'un système linux embarqué complet

1 Introduction

Les ordinateurs embarqués fonctionnant sous le système d'exploitation Linux sont massivement présents dans les technologies modernes (transports, multimédia, téléphonie mobile, appareils photos …).

Contrairement aux versions de Linux destinées aux ordinateurs personnels et aux serveurs, les différents systèmes Linux embarqués sont conçus pour des systèmes aux ressources limitées.

Les systèmes embarqués sous Linux disposent généralement de peu de RAM et utilisent fréquemment de la mémoire flash plutôt qu'un disque dur. Comme ils sont souvent dédiés à un nombre de tâches réduites sur une cible matérielle bien définie, ils utilisent plutôt des versions du noyau Linux optimisées pour des contextes précis.

Les nécessités de déterminisme et de réactivité entraine parfois les concepteurs à se tourner vers des solutions temps réelles.

L'objectif de ce TP est de réaliser un système complet qui sera déployé sur un ordinateur Raspberry PI et d'en évaluer ces performances temps réelle.

2 Durée

Le TP dure **4h30** mais il est fortement recommandé de prendre connaissance des informations concernant Buildroot et d'effectué les recherches concernant le matériel de la carte Raspberry Pi en amont (parties 4 et 5).

3 Evaluation

L'évaluation portera sur la rédaction d'un compte rendu détaillé :

- explicitant les différentes opérations menées durant le TP.
- présentant une analyse des différents tests et des mesures mis en oeuvre.
  Pensez à prendre des copies d'écran pour illustrer ces opérations.

4 Buildroot

4.1 Introduction

Dans le domaine de l'embarqué, on se retrouve souvent en situation de devoir reconstruire un système complet à partir des sources, pour une architecture cible souvent différente de celle de l'hôte. La cross-compilation et l'organisation d'un système embarqué sont des étapes longues et fastidieuses, surtout lorsque les éléments du système à compiler nécessitent des adaptations. Il existe heureusement des outils libres qui simplifient et accélèrent cette tâche, en proposant généralement des fonctionnalités complémentaires intéressantes.

Buildroot est un de ces outils qui simplifie et automatise le processus de construction d'un système Linux complet pour un système embarqué. Afin d'atteindre cet objectif, Buildroot est capable de générer une chaîne de compilation croisée, un système de fichiers (rootfs), une image du noyau Linux (kernel) et un chargeur de démarrage (firmware/bootloader) pour la cible.

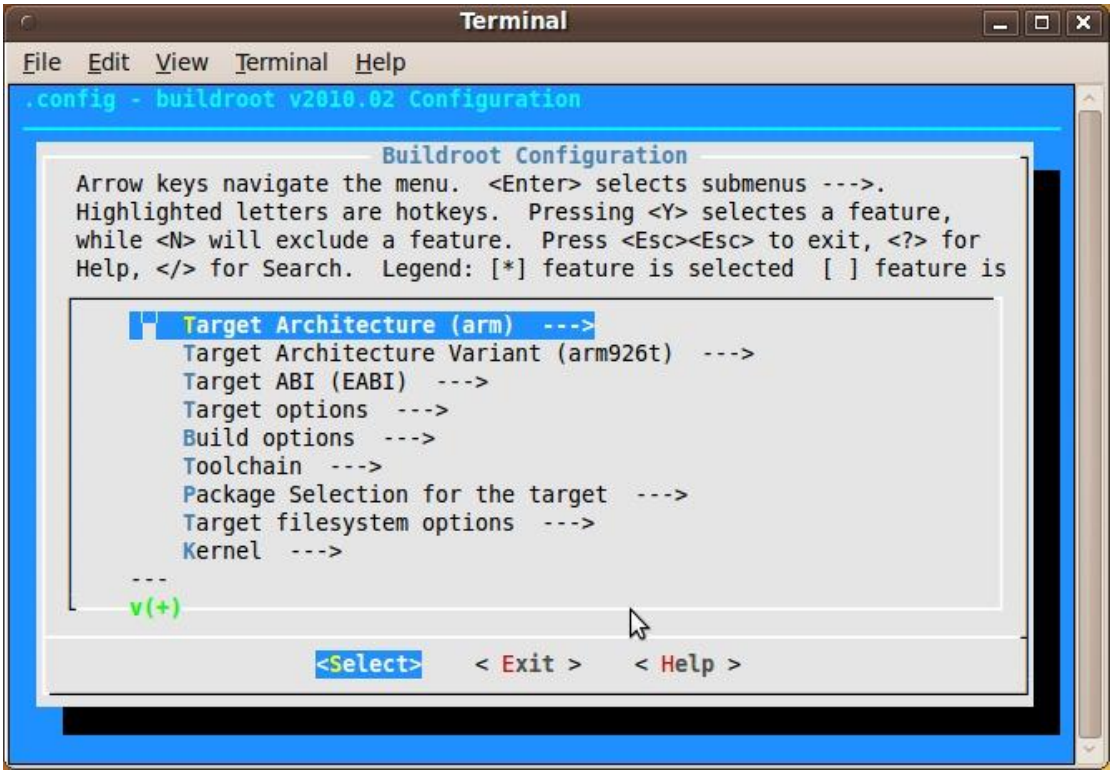
Il prend en charge de nombreux processeurs et leurs variantes (x86, PowerPC, MIPS, ARM, NIOS, etc). Il est également livré avec des configurations par défaut pour un grand nombre de cartes disponibles sur le marché.

4.2 Principe de fonctionnement

Buildroot est techniquement un ensemble de Makefiles définissant, en fonction des options paramétrées par l'utilisateur, la manière de compiler chaque paquet sélectionné avec des options particulières. Il construit finalement une distribution complète et cohérente dont chaque composant a été compilé.

Il possède un outil confortable de configuration, basé et très similaire à celui du noyau Linux : **menuconfig**, que l'on retrouve également avec **Busybox**, **uClibc** et qui peut être utilisé dans tout projet.

```
make menuconfig
```



Une fois que tout est configuré, l'outil de configuration génère un fichier .config qui contient l'ensemble de la configuration. Ce fichier sera lu par le fichier Makefile lors du processus de construction.

Pour démarrer le processus de construction, il suffit de lancer :

```
make
```

La commande **make** effectue généralement les étapes suivantes :

- téléchargement des fichiers sources (si nécessaire);
- configuration, compilation et installation de la chaîne de compilation croisée
- configuration, compilation, corrections (application des patchs) et installation des paquets cibles sélectionnées;
- construction d'une image du noyau
- construction d'une image de bootloader;
- création d'un système de fichiers racine (rootfs) dans les formats sélectionnés.

Le résultat de la construction est stocké dans un répertoire unique, **output/**. Ce répertoire contient plusieurs sous-répertoires :

- **`images/`** : où toutes les images (image du noyau, bootloader et système de fichiers racine) sont stockés. Ce sont les fichiers qui seront copiés sur le système cible.
- **`build/`** : où tous les composants sont construits (ce qui inclut les outils nécessaires par Buildroot sur l'hôte et des paquets compilés pour la cible). Ce répertoire contient un sous-répertoire pour chacun de ces composants.
- **`staging/`** **:** contient une hiérarchie similaire à une hiérarchie de système de fichiers racine. Ce répertoire contient les en-têtes et les bibliothèques de la chaîne d'outils de compilation croisée et tous les paquets de l'espace utilisateur sélectionnés pour la cible. Cependant, ce répertoire ne vise pas à être le système de fichiers racine pour la cible: il contient un grand nombre de fichiers de développement, les binaires et les bibliothèques qui font qu'il est beaucoup trop grand pour un système embarqué non dénudés. Ces fichiers de développement sont utilisés pour compiler les bibliothèques et les applications pour la cible qui dépendent d'autres bibliothèques.
- **`target/`** : contient presque le système de fichiers **rootfs** complet pour la cible : tout le nécessaire est présent, sauf les fichiers de périphérique dans **/dev** (Buildroot ne peut pas les créer car Buildroot ne fonctionne pas en tant que root). En outre, il ne possède pas les autorisations appropriées (par exemple de **setuid** pour le binaire **busybox**). Par conséquent, ce répertoire ne doit pas être utilisé sur la cible. Par rapport à la mise **`staging/`**, **`target/`** contient uniquement les fichiers et les bibliothèques nécessaires pour exécuter les applications cibles sélectionnés : les fichiers de développement (en-têtes, etc.) ne sont pas présents, les binaires sont dépouillés.
- **`host/`** : contient l'installation des outils compilés pour l'hôte qui sont nécessaires pour la bonne exécution de Buildroot, y compris **la chaîne d'outils de compilation croisée**.

4.3 Cross-compilation toolchain

La toolchain désigne l'ensemble des outils à compiler qui permettra ensuite d'avoir un environnement capable de cross compiler depuis l'architecture host (x86_64) vers l'architecture cible (ARM).

La toolchain regroupe un certain nombre de composants obligatoires comme :

- un compilateur ;
- un linker ;
- un assembleur.

Buildroot propose plusieurs mécanismes pour utiliser une toolchain. Le plus direct est d'utiliser la toolchain interne, et dans ce cas c'est Buildroot qui gérera la création et l'utilisation de la toolchain.

4.4 Init system / bootloader

Les systèmes Linux embarqués sont généralement chargés par le bootloader U-boot, mais ce n'est pas le cas du Raspberry Pi, qui dispose de son propre outil de démarrage qui comprend les fichiers suivants :

- **bootcode.bin** : sert à l'initialisation du GPU
- **bcm2709-rpi-2-b.dtb** : (Device Tree Blob) Fichier de description du matériel. Permet de gérer certaines allocations des ressources et les chargements de modules.
- **start.elf** et **fixup.dat** : bootloader qui termine le processus d'initialisation du CPU et du GPU en attribuant à chacun une partie de la RAM disponible.
- **config.txt** : fichier texte pour personnaliser le comportement du GPU. Entre autres, l'option gpu_mem est très utile puisqu'elle permet de répartir la mémoire disponible entre le GPU et le CPU. Par exemple la ligne « gpu_mem=32 » affecte 32 Mo de Ram au GPU et le reste au CPU.
- **cmdline.txt** : fichier texte qui contient des paramètres supplémentaires à ceux déjà embarqués dans l'image du noyau compilé. Ce fichier est indispensable et ne doit pas être vide. Il contient par défaut les options :
  - **rootwait** : attendre (éventuellement indéfiniment) sans échouer, que la partition contenant l'arborescence des fichiers soit prête ; ceci est nécessaire lorsque l'initialisation du périphérique bloc correspondant peut être longue (notamment pour les disques USB) ;
  - **root=/dev/mmcblk0p2** : la racine de l'arborescence des fichiers se trouve sur la seconde partition de la première (et seule) carte SD ;
- **zImage** : l'image du noyau à démarrer.

4.5 Le noyau

L'élément probablement le plus spécifique d'une plateforme embarquée est le noyau Linux. Contrairement aux kernels fournis avec les distributions pour postes de travail ou serveurs, nous ne voulons pas d'un noyau générique capable de fonctionner sur une multitude de machines différentes, mais d'une configuration bien ajustée, contenant tous les drivers, protocoles, systèmes de fichiers indispensables, sans en ajouter plus que nécessaire.

Le noyau linux standard peut être ajusté en utilisant la commande suivante dans le dossier de Buildroot.

```
make linux-menuconfig
```

Cette opération télécharge, extrait (dans le répertoire output/build/linux-<version>), et pre-configure le kernel avec les options paramétrées dans Buildroot pour la configuration du noyau. La règle **make** ci-dessus rentre dans le répertoire et lance un « **make menuconfig** » avec les options d'environnement pour la cible.

4.6 Le système de fichiers racine rootfs

C'est le système complet installé sur la carte qui sera utilisé après le démarrage du bootloader puis de Linux. Sa configuration s'effectue principalement dans le choix des paquets à installer sur la cible, à partir du menu **Package Selection for the target** de Buildroot.

Il contient également l'ensemble des commandes disponibles sur le futur système. La solution la plus efficace pour un système embarqué est d'utiliser Busybox.

BusyBox est un logiciel qui implémente un grand nombre des commandes standard sous Unix. Il est conçu comme un unique fichier exécutable, ce qui le rend très adapté aux distributions Linux utilisées sur les systèmes embarqués.

Il est notamment très répandu de nos jours sur les périphériques réseaux. On le trouve par exemple sur des points d'accès, des routeurs, des téléphones IP, certains serveurs de stockage en réseau (Network Attached Storage ou NAS) ou encore dans les dernières générations de robots (AR-Drone). En France, le code de BusyBox est également intégré dans les box de certains fournisseurs d'accès Internet : Livebox, Freebox et Neufbox. Il est aussi possible d'installer son propre BusyBox sur Android, très utilisé sur les smartphone et tablettes tactiles, pour obtenir un meilleur contrôle de son matériel.

Pour personnaliser l'ensemble des commandes de Busybox à installer sur la cible :

```
make busybox-menuconfig
```

4.7 La construction

Le processus de construction (**make**) peut être très long. Aussi vaut-il mieux être sur d'avoir bien tout configuré avant de le lancer.

Toutefois, il est possible de relancer une compilation partielle dans certains cas.

Voici quelques règles de base qui peuvent vous aider à comprendre comment travailler avec Buildroot :

- **Lorsque la configuration de l'architecture cible est modifié, une reconstruction complète est nécessaire**. La modification de l'architecture du CPU, le format binaire ou la stratégie de gestion des nombres à virgule flottante par exemple, ont un impact sur l'ensemble du système.
- **Lorsque la configuration de la toolchain est modifiée, une reconstruction complète est généralement nécessaire**. La modification de la configuration de la toolchain implique souvent de modifier la version du compilateur, du type de la bibliothèque C ou sa configuration, ou un autre élément de configuration fondamentale. Ces changements ont un impact sur l'ensemble du système.
- **Quand un paquet supplémentaire est ajouté à la configuration, une reconstruction complète n'est pas forcément nécessaire**. Buildroot détectera que ce paquet n'a jamais été construit, et le construira.
- **Quand un paquet est supprimé de la configuration, Buildroot ne fait rien de spécial**. Il ne supprime pas les fichiers installés par ce paquet du système de fichiers racine de la cible**. Une reconstruction complète est nécessaire pour se débarrasser de ce paquet.**
- **Lorsque les sous-options d'un paquet sont modifiées**, le paquet n'est pas automatiquement reconstruit. Après avoir fait ces changements, **la reconstruction de ce paquet est souvent suffisante**.
- **Quand un changement au squelette de système de fichiers racine est fait, une reconstruction complète est nécessaire**. Cependant, lorsque **des modifications du système de fichiers racine sont faites dans le dossier de sortie (output/target) ou dans le dossier overlay, dans un script de post-construction ou dans un script de post-image, il n'y a pas besoin d'une reconstruction complète**.

Pour en savoir plus sur Buildroot et son utilisation, reportez vous au manuel :

http://buildroot.uclibc.org/downloads/manual/manual.html

5 Raspberry pi - le matériel

- Quel est la référence du processeur qui équipe votre carte Raspberry Pi 2 ?
- Quel est son type d'architecture ? Combien de cœurs possède-t-il ?
- Quel est la quantité de RAM disponible ?
- La mémoire est-elle gérée par une MMU ?
- Quel est le ratio entre la mémoire CPU/GPU par défaut ?
- Quels sont les bus de communication disponibles ?
- Quelle version de VFP (Vector Foating Point) support-il ?

6 Buildroot pour Raspberry Pi 2

6.1 Téléchargement

Nous allons créer ici un système linux complet pour Raspberry Pi 2 à partir d'une version de Buildroot native qu'il faudra configurer :

https://buildroot.org/download.html (NE PAS TELECHARGER ! C'est déjà fait. Lire la suite.)

6.2 Construction d'un système de base

La compilation d'un système avec les options par défaut peut être particulièrement longue, aussi, une pré-compilation a déjà été effectuée dans le dossier **~/Documents/TP-Buildroot/buildroot-rpi**.

- Prenez connaissance du fichier **README**, quelles sont les actions à mener pour construire un système ?
- Supprimez le fichier **.config** dans le dossier **buildroot-rpi**.
- Consultez le menu de configuration de buildroot, notamment **Target options**.
- Prenez connaissance du fichier de définition de la configuration pour une cible Raspberry Pi 2. Les fichiers de configuration standard des différentes cibles supportées par Buildroot se trouvent dans le dossier **configs**. Analysez le contenu de ce fichier et identifiez les principaux changements qui seront effectués par l'application de configuration, notamment par rapport aux spécificités du matériel.
- Appliquez la commande **make raspberrypi2_defconfig**. Observez la création du fichier **.config**.
- Consultez à nouveau le menu de configuration de buildroot, notamment **Target options**. Modifiez la taille de la partition **Filesystem à 262144 blocks** (1 block = 1ko).
- Consultez le fichier **readme.txt** du dossier **board/raspberrypi2**. Déployez le système obtenu sur votre carte micro-SD. Quelle est la taille du noyau et du système de fichier (en Mo). Décrivez les différentes étapes à mener.
- Testez votre système et commentez les éléments suivant:
    - Estimez la durée de boot
    - Bannière d'accueil, mot de passe root
    - Allure du prompt, disposition du clavier
    - Connexion réseau, connexion à distance par ssh
    - Utilisation des GPIO
    - Identifiez la version du noyau linux installé.
- Consultez le script **/etc/init.d/rcS** et expliquez la procédure de démarrage du système.
    - Consultez et expliquez le fonctionnement des scripts **/etc/network/interfaces** et **/etc/init.d/Sxxnetwork** pour que l'interface **eth0** soit activée et recherche ses paramètres IP automatiquement.
      Remarque : votre système minimum ne dispose que de **vi** comme éditeur !

6.3 Amélioration du système

6.3.1 Clavier AZERTY

En connexion directe avec un clavier USB et un écran HDMI, le clavier est reconnu suivant une organisation Qwerty et non Azerty.

Pour configurer le clavier correctement, il faut appeler, au sein d'un script de démarrage, la commande **loadkmap** de **Busybox** en lui envoyant sur son entrée standard le contenu d'un fichier de configuration. On obtient ce fichier simplement en invoquant l'applet **dumpkmap** de **Busybox** après l'avoir recompilé sur un système où le clavier est configuré correctement comme, par exemple, votre PC de développement :

```
$ sudo busybox dumpkmap > azerty.kmap
```

Le fichier **azerty.kmap** obtenu doit ensuite être transféré dans le dossier **/etc** de la cible et invoqué dans un script de démarrage.

- Créez le fichier **azerty.kmap** sur votre hôte et transférez-le sur la cible.
- Créez dans le dossier **/etc/init.d** de la carte micro-SD un script de démarrage **S10keyboard** pour charger le gestionnaire de clavier à l'aide de la commande :

  ```
  loadkmap < /etc/azerty.kmap
  ```

- Redémarrer la cible et testez la configuration du clavier.
- Expliquez le contenu de votre script et son fonctionnement lors du processus de démarrage.

6.3.2 Prompt du shell global

Le système minimal présente un shell minimaliste dans lequel il n'est pas aisé de savoir où l'on se trouve dans le système de fichier et où l'utilisateur connecté n'est pas identifié clairement. On sait juste s'il s'agit ou non de root (**$** ou **#**).

Pour configurer le prompt du shell afin qu'il indique clairement l'utilisateur connecté et le dossier courant, il faut modifier le fichier **/etc/profile**. Le prompt du shell d'un terminal y est représenté par la variable **PSn** ou n est le numéro du terminal. Ici, nous n'avons qu'un seul terminal configuré : **PS1**

Pour obtenir un prompt de type **user@hostname : directory**

```
...
if [ "$PS1" ]; then
    export PS1="\u@\h:\w "
...
      if [ "`id -u`" -eq 0 ]; then
            export PS1=${PS1}"# "
      else
            export PS1=${PS1}"$ "
      fi
```

- Modifiez le fichier **/etc/profile** comme ci-dessus.
- Redémarrer la cible et observez l'allure du prompt quand vous changez de dossier.
- Expliquez le test `if [ "`id -u`" -eq 0 ]; then ...` du script.

6.3.3 Accès à distance

La prise en main à distance d'un système est en général assuré au moyen d'un serveur ssh. Vous pourrez utiliser au choix le package dropbear ou openSSH.

make menuconfig -> Target packages -> Networking applications -> openssh (ou dropbear)

Dans les deux cas, il faudra au préalable sécuriser l'accès local au système en créant un mot de passe pour root et un au moins un utilisateur local.

- Sécurisez l'accès au système en attribuant un mot de passe à root :
  make menuconfig -> System configuration -> Root password
- Relancez le processus de construction (il sera partiel et ne prendra en compte que les changements à effectuer sur le système de fichiers rootfs.ext)
- Déployez le système obtenu sur votre carte micro-SD.
- Redémarrez la cible et observez les modifications réalisées (le prompt et le clavier ?).
- Assurez vous de la nécessité de fournir le bon mot de passe root pour accéder au système.
- Créez un nouvel utilisateur "admin" et attribuez lui un mot de passe par défaut "password".
- Accédez à la cible depuis votre PC hôte via une session ssh. Assurez-vous que root ne puisse pas y accéder directement. (**Bug** : Le nombre de jours (depuis le 1er Janvier 1970) depuis le dernier changement du mot de passe est 0 dans le fichier shadow pour l'utilisateur admin).
- Expliquez les différentes étapes que vous avez mené. Expliquez pourquoi root ne devrait jamais pouvoir ouvrir une connexion distante directement.

6.3.4 Persistance des modifications sur rootfs

Les modifications réalisées directement sur cible ou sur la carte micro-SD à partir de l'hôte fonctionnent très bien pour un système unique et final. Cependant, lors la prochaine reconstruction du système, ces changements auront disparu car ils ne sont pas présents dans la configuration de buildroot.

Il est important que le processus de construction reste entièrement reproductible, si l'on veut s'assurer que la prochaine version inclura les configurations personnalisées.

Pour ce faire, le plus simple est d'utiliser le mécanisme de recouvrement de rootfs de Buildroot (overlay mechanism). Cette substitution de fichiers est spécifique au projet en cours. Il faut donc créer un répertoire personnalisé pour ce projet dans le source de buildroot pour la cible visée :

board/<manufacturer>/<boardname>/fs-overlay/

Dans la config de buildroot (**buildroot menuconfig**), il faut spécifier ce chemin dans l'option **Root filesystem overlay directories** du menu **System configuration**.

Il faut ensuite recréer l'architecture des dossiers contenant les scripts à personnaliser dans le dossier **fs-overlay** :

```
marco@marco-CERI:~/Documents/buildroot/buildroot-2016.08/board/raspberrypi2/fs-overlay$ tree
.
├── etc
│   ├── azerty.kmap
│   ├── group
│   ├── init.d
│   │   └── S10keyboard
│   ├── passwd
│   ├── profile
│   └── shadow
```

- Créez le dossier de recouvrement **fs-overlay** et copiez-y les scripts précédents assurant la bonne configuration du clavier, du prompt et des utilisateurs.
- Indiquez la présence de se dossier à buildroot.
- Ajoutez au système un nom d'hôte et une bannière d'accueil.
  make menuconfig -> System configuration -> System hostname
  -> System banner

- Relancez le processus de construction.
- Remplacez le système de fichier racine sur la carte micro-SD.
- Redémarrez la cible et observez les modifications réalisées.

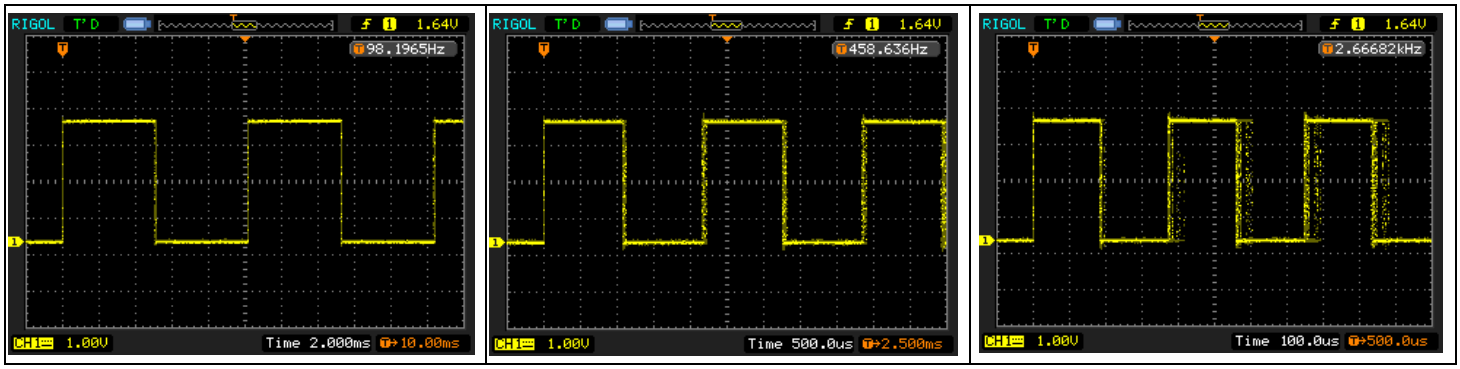
7 production d'un signal périodique

7.1 Noyau linux standard

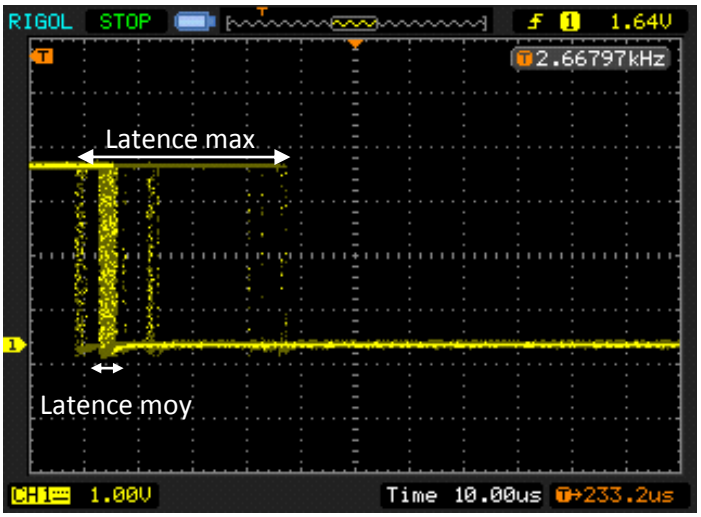
Le but de cette partie est de générer un signal périodique sur un port GPIO et d'évaluer ses caractéristiques.

La première idée consiste à écrire un simple programme qui sera exécuter en arrière plan et met successivement à 1, puis à 0 la sortie GPIO concernée. La durée de maintient de l'état logique est réalisée par un appel à la fonction usleep().

- Ouvrez le programme squareSignal.cpp :
- Repérez physiquement la broche qui sera utilisée pour la génération du signal.
- Compilez le programme et transférez l'exécutable sur la cible.
  - A l'aide d'un oscilloscope, observez la génération d'un signal pour des durées d'impulsions de 5ms, 1ms et 100µs.



- Sur l'oscilloscope, activez la fonction de persistance de l'affichage et observez la gigue du signal. Quantifiez la gigue moyenne et maximale observée sur une trentaine de seconde (il faudrait en fait pouvoir faire ce teste sur plusieurs heures, voir jours de fonctionnement pour avoir des données fiables). Cette gigue est la conséquence de la latence lors de la production d'une impulsion.



Gigue observée en fin de production d'une impulsion de 100µs pendant 30s

- Le système n'est que très peu chargé et la gigue semble relativement faible. Pour qualifier le système de "temps réel", il faut pouvoir certifier quelle sera sa latence maximale et donc, son temps de

réponse, quelque soit les conditions d'utilisation. Nous allons tester notre système en condition de stress. Ajoutez à votre projet buildroot le package Stress :

make menuconfig -> Target packages -> Debugging, profiling and benchmark -> Stress

- Relancez le processus de construction.
- Remplacez le système de fichier racine sur la carte micro-SD.
- Redémarrez la cible, vérifiez l'état de stress du système, puis stresser le système :

```
$ uptime
$ stress -c 4 -i 2 -m 2 --vm-bytes 128M -t 30s
$ uptime
```

- Commentez la sortie de la commande uptime avant et après le stress du système.
- Indiquez la signification des options utilisées dans la commande stress.
- Exécutez en tâche de fond le programme squareSignal pour produire des impulsions de 1ms de durée, puis stressez le système. Observez à l'oscilloscope la fin de l'impulsion et quantifiez la durée moyenne et maximale de la gigue.
- En conclusion, peut-on qualifier un tel système de "temps réel dur", de "temps réel mou" ou absolument pas temps réel ? Argumentez votre choix.

7.2 Noyau linux et Xenomai

Xenomai est une extension libre du noyau Linux lui apportant des fonctionnalités temps réel dures. Il apporte une approche symétrique entre programmation noyau et programmation système au niveau utilisateur sous Linux. Il introduit le concept de machine virtuelle en programmation temps réel et permet ainsi de disposer de plusieurs interfaces de programmation, au choix du programmeur.

7.2.1 Installation

Buildroot propose une interface d'installation de Xenomai mais elle ne semble pas encore mure pour le déploiement sur une Raspberry Pi. Nous allons donc procéder à une installation manuelle.

Xenomai 3 propose deux modèles de fonctionnement :

Dans le mode Mercury, les bibliothèques de Xenomai permettent de faire fonctionner du code applicatif (utilisant éventuellement l'API d'un autre système comme VxWorks ou pSOS) sur un noyau Linux standard, de préférence une version PREEMPT_RT.

Dans le mode Cobalt, le système Adeos (ipipe) capture les interruptions avant le noyau Linux et est donc capable d'activer des tâches Xenomai dans un temps plus prévisible. L'intégration est un peu plus complexe puisqu'elle nécessite de modifier le code du noyau Linux.

C'est ce dernier mode (Cobalt) que nous allons installer :

7.2.2 Préparation

- Sur votre PC hôte, vous disposez d'un dossier **~/Documents/TP-Buildroot/xenomai** dans lequel vous trouverez les archives des sources de linux (**linux.tar.gz**) et de xenomai (**xenomai-3.0.2.tar.bz2**) ainsi que le dossier **build** qui contient les résultats des compilations. Vous pourrez utiliser le contenu de se dossier si le temps vous presse, la compilation du noyau linux pouvant prendre un certain temps en fonction des caractéristiques de votre machine hôte. Dans ce cas, passez directement à la copie du noyau (fin 7.2.3) et à la copie des fichiers (fin 7.2.4).

- Décompactez les deux archives.
- Dans le dossier linux décompacté, vérifiez la version du noyau :

```
$ head -3 Makefile
```

Remarque : Le noyau de la branche 4.1 pour Raspberry Pi (https://github.com/raspberrypi/linux/tree/rpi-4.1.y) est un 4.1.21. Le patch ipipe de xenomai est prévu pour un noyau 4.1.18, mais il s'applique très bien au 4.1.21.

- Testez puis appliquez le patch **ipipe-core-4.1.18-arm-4.patch** sur les sources de linux.

```
$ patch --dry-run -p1 < ../xenomai-3.0.2/kernel/cobalt/arch/arm/patches/ipipe-core-4.1.18-arm-4.patch
checking file arch/arm/Kconfig
Hunk #3 succeeded at 533 (offset 36 lines).
...
checking file mm/vmalloc.c
$ cd .. #sortir du dossier linux
$ xenomai-3.0.2/scripts/prepare-kernel.sh --linux=linux/ --arch=arm --ipipe=xenomai-3.0.2/kernel/cobalt/arch/arm/patches/ipipe-core-4.1.18-arm-4.patch
```

- Ce noyau linux fait référence à la famille de Soc bcm2708, or la raspberrypi2 utilise un bcm2709. L'application de l'extension xenomai sur ce noyau ne fonctionne pas. Le problème a été résolu par un patch de Mathieu Rondonneau qui permet d'utiliser version 3 de Xenomai : http://www.xenomai.org/pipermail/xenomai/2016-April/036110.html

  Ce patch est disponible dans le dossier xenomai : **patch-xenomai-3-on-bcm-2709.patch**

  Testez puis appliquez ce patch sur les sources de linux.

```
$ cd linux/
$ patch -p1 < ../patch-xenomai-3-on-bcm-2709.patch --dry-run
checking file arch/arm/Kconfig
...
checking file arch/arm/mach-bcm2709/include/mach/entry-macro.S
$ patch -p1 < ../patch-xenomai-3-on-bcm-2709.patch
patching file arch/arm/Kconfig
...
patching file arch/arm/mach-bcm2709/include/mach/entry-macro.S
```

7.2.3 Compilation du noyau

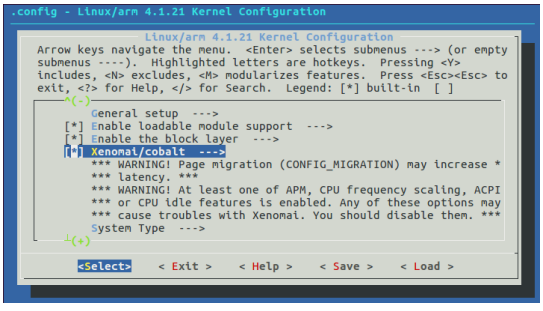
- Appliquez le fichier de définition de la configuration du Soc :

```
$ make ARCH=arm bcm2709_defconfig
```

- Configurez le noyau avant compilation :

```
$ make ARCH=arm menuconfig
```

Notez l'apparition d'une nouvelle entrée dans le menu de configuration du noyau :



- Deux messages « WARNING » nous indiquent que certains éléments de la configuration actuelle entrent en conflit avec Xenomai. Désactivez les entrées suivantes :
  - **CPU Power Management -> CPU Frequency scaling -> CPU Frequency scaling**
  - **Kernel Features -> Allow for memory compaction**
  - **Kernel Features -> Contiguous Memory Allocator**
  - **Kernel Hacking -> KGDB: kernel debugger**
  - **Boot options -> Kernel command line type -> (X) Extend bootloader kernel arguments**
- Lancez la compilation. **ATTENTION : DUREE DE COMPILATION !!! Vous disposez d'un noyau compilé dans le dossier /home/xenomai/build**.

```
$ PATH=$PATH:/<TP-Buildroot dir>/buildroot-rpi/output/host/usr/bin/
$ make ARCH=arm CROSS_COMPILE=arm-linux-
```

- Copiez le noyau obtenu sur la partition BOOT de la carte microSD.

```
$ cp  arch/arm/boot/zImage  /media/$USER/BOOT/
$ cp  arch/arm/boot/dts/bcm2709-rpi-2-b.dtb  /media/$USER/BOOT/
```

7.2.4 Compilation de xenomai

Cette compilation a pour but de fournir les bibliothèques et les exécutables nécessaires au fonctionnement du code depuis l'espace utilisateur.

- Lancez la compilation.

```
$ cd ../xenomai-3.0.2
$ ./scripts/bootstrap
...
$ ./configure  --host=arm-linux  --enable-smp
...
$ make
...
```

Notez la présence de l'option --enable-smp : SMP est l'abréviation de Symetric Multi-Processors qui désigne la capacité des noyaux Linux 2.0 et suivantes de fonctionner sur des machines à plusieurs processeurs.

- Préparez et copiez les fichiers obtenus sur la partition ROOTFS de la carte microSD : (Attention, vous pouvez avoir une erreur "no space left on device". En fait, il n'y a plus d'inode disponible pour référencer les nouveaux fichiers. Reprendre buildroot et ajouter 2048 inodes en extra au système de fichiers).

```
$ make  DESTDIR=$(pwd)/target  install
...
$ sudo  cp  -a  target/*  /media/$USER/ROOTFS/
```

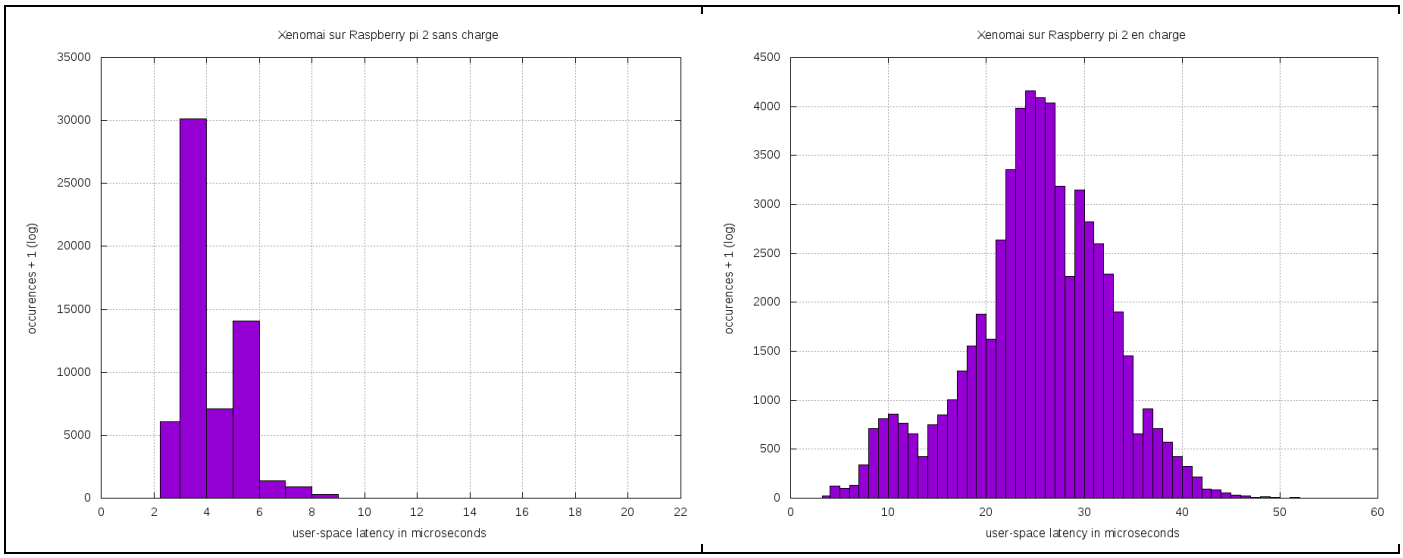
7.2.5 Tests

Pour tester le bon fonctionnement de xenomai, une série d'outils est disponible dans le dossier **/usr/xenomai/bin**. Nous allons utiliser l'outil latency qui montre les fluctuations (en micro-secondes) d'un timer de fréquence 1kHz par défaut. Cette expérience est également assez révélatrice des variations du temps de latence des interruptions. Les résultats sont présentés en colonnes dont la plus intéressante est la dernière à droite, puisqu'elle montre le retard maximum atteint pendant l'exécution du test.

- Insérez la carte microSD sur la cible et redémarrez-la.
- identifiez la version du noyau, de xenomai (/proc/xenomai/version) et de ipipe (/proc/ipipe/version).
- Testez la latence du système avec l'outil **`latency`** :

```
# /usr/xenomai/bin/latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      2.551|      2.795|      7.290|       0|     0|      2.551|      7.290
RTD|      2.551|      2.809|      7.082|       0|     0|      2.551|      7.290
RTD|      2.498|      2.825|      8.748|       0|     0|      2.498|      8.748
RTD|      2.446|      2.787|      7.393|       0|     0|      2.446|      8.748
RTD|      2.549|      2.784|      6.768|       0|     0|      2.446|      8.748
RTD|      2.497|      2.784|      6.716|       0|     0|      2.446|      8.748
RTD|      2.549|      2.781|      6.611|       0|     0|      2.446|      8.748
```

- Consultez la documentation de l'outil **`latency`**. Tracez avec **gnuplot** l'histogramme de la répartition des latences observées sur une période de 1mn lorsque le système est soumis à une faible charge.
- Stressez le système et tracez la nouvelle répartition des latences sur une période de 1mn. **Remarque** : xenomai fourni l'outil **`dohell`** qui permet de charger le système.



- Concluez sur les caractéristiques "temps réel dure" de la carte Raspberry pi 2 faisant tourner notre OS linux embarqué maison. Critiquez les tests effectués et proposez une procédure de test appropriée qui tiendrait compte notamment des régimes transitoires entre périodes de charges légères et intenses du système.

7.2.6 Création d'une tâche périodique

Xenomai dispose de plusieurs API qui dépendent de son mode de fonctionnement. Nous utilisons ici le mode Cobalt qui s'appui sur l'utilisation de l'API native depuis la version 2.x de xenomai.

Remarque : à partir de la versions 3.x, l'API native a évolué vers l'API alchemy. Pour des raisons de rétrocompatibilité, nous utiliserons ici l'API native :
http://xenomai.org/documentation/trunk/html/api/group__task.html

Les tâches peuvent être créées depuis le noyau ou l'espace utilisateur :

- **Kernel-based task** : tâche Xenomai lancée par **`rtdm_task_init()`** ;
- **User-space task** : tâche Xenomai ou Linux lancée explicitement par l'utilisateur.

Nous travaillerons depuis l'espace utilisateur.

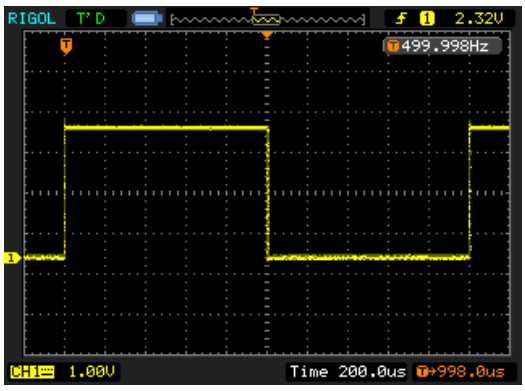
- Ouvrez le programme xeno_squareSignal.c
- Analysez le programme et indiquez notamment le nom de la tâche temps réel, la taille de sa pile et son niveau de priorité. Expliquez également les lignes :
  - **`mlockall(MCL_CURRENT|MCL_FUTURE);`**
  - **`rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP);`**
  - **`rt_task_wait_period(NULL);`**
- Ecrivez un fichier makefile afin de cross-compiler le programme. Pour cela, il nous faut connaitre les chemins des librairies utiles. Xenomai fourni un outils permettant d'obtenir les flags cflags et ldflags en fonction de l'API utilisée :

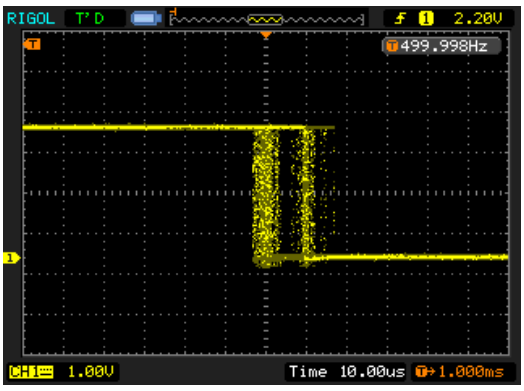
```
$ <path_to_xenomai>/usr/xenomai/bin/xeno-config --native --cflags
$ <path_to_xenomai>/usr/xenomai/bin/xeno-config --native --ldflags
```

- Cross-compilez puis transférez l'exécutable obtenu sur la cible.
- Exécutez le programme et observez la génération d'impulsions à l'oscilloscope.

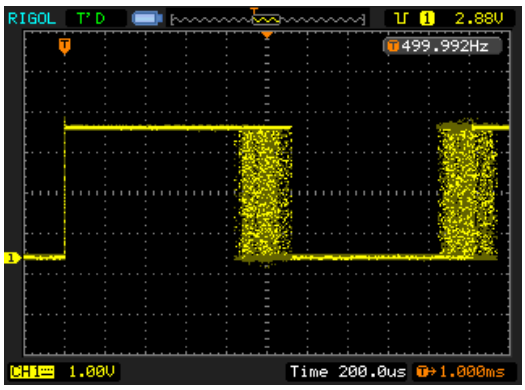
```
# ./xeno_squareSignal
# xeno_squareSignal : can't load library 'libtrank.so.0'
# export LD_LIBRARY_PATH=/usr/xenomai/lib
# ./xeno_squareSignal
```

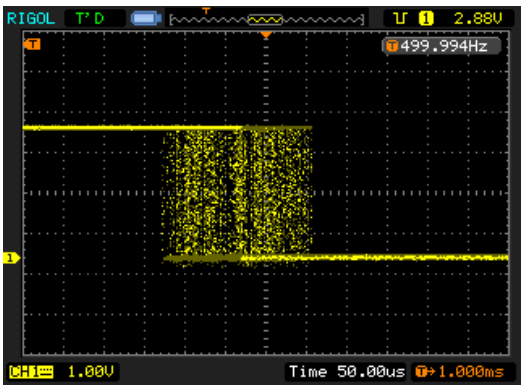
- Mesurez la durée des impulsions et la latence maximale lorsque le système n'est pas chargé puis lorsqu'il est en charge.
- Analysez vos résultats et comparez les performances des solutions "noyau linux seul" et "noyau linux + extension Xenomai".





Système faiblement chargé (Connexion réseau déconnectée)





Système fortmement chargé (dohell + connexion réseau)

8 Références web

8.1 Buildroot

- http://buildroot.uclibc.org/
- http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-155/Linux-from-scratch-Construire-une-chaine-de-compilation
- http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-155/Linux-from-scratch-Construire-un-systeme-complet
- http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-158/Raspberry-Pi-from-scratch-3
- http://linux.developpez.com/tutoriels/embarque-buildroot/
- http://free-electrons.com/fr/formation/buildroot/
- https://github.com/gamaral/rpi-buildroot

8.2 Raspberry Pi

- https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
- http://www.adafruit.com/pdfs/raspberrypi2modelb.pdf
- https://learn.adafruit.com/downloads/pdf/introducing-the-raspberry-pi-2-model-b.pdf
- https://en.wikipedia.org/wiki/ARM_Cortex-A7

8.3 Linux

- https://www.kernel.org/doc/man-pages/
- https://github.com/raspberrypi/linux.git
- https://github.com/torvalds/linux

8.4 Buzybox

- http://www.busybox.net/

8.5 vi

- http://www.linux-france.org/prj/support/outils/vi.html
- http://wiki.linux-france.org/wiki/Utilisation_de_vi

8.6 Xenomai

- https://xenomai.org/
- http://www.blaess.fr/christophe/2016/05/22/xenomai-3-sur-raspberry-pi-2/

- http://dchabal.developpez.com/tutoriels/linux/xenomai/