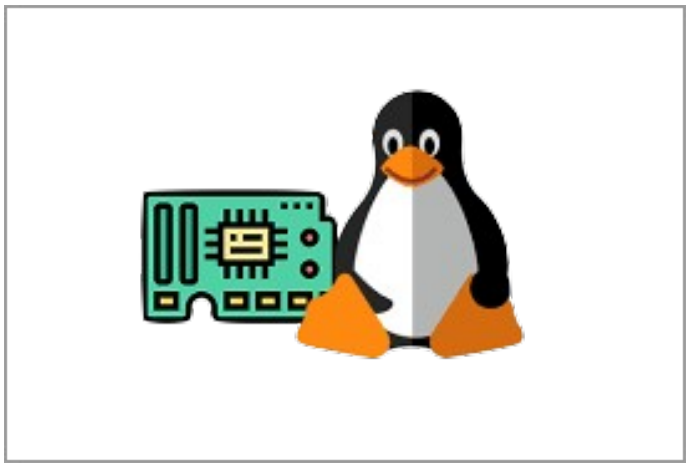
Systèmes Embarqués Communicants

Liste des TD et des TP :



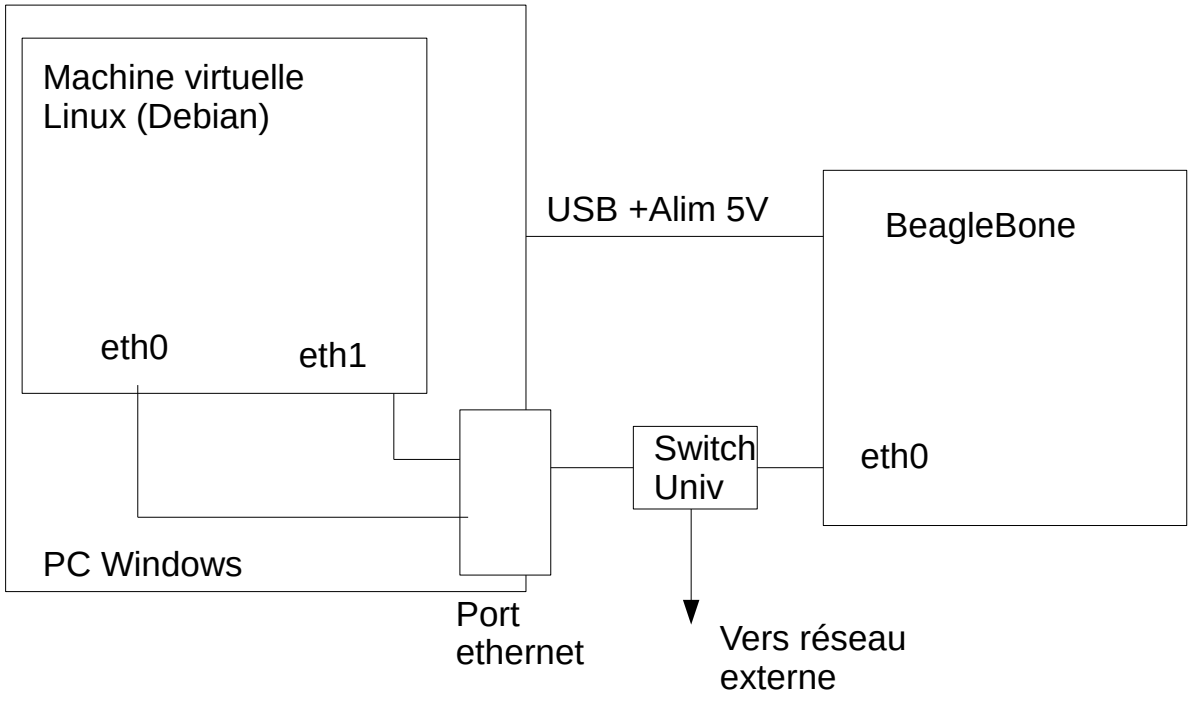
Consignes pour l'utilisation de la carte BeagleBone.

Dans cette série de TD et TP, vous allez utiliser une carte BeagleBone qui embarque :

- un processeur TI Sitara (coeur ARM Cortex-A8), cadencé à 500MHz/720MHz ;
- 256Mo de mémoire vive (RAM DDR2 400MHz) ;
- de nombreux contrôleurs de périphériques : USB, UART, I2C, ethernet…

Cette carte a suffisamment de ressources (processeur/mémoire) pour permettre l'installation d'un Linux embarqué (*embedded Linux*). Ce système d'exploitation permet de faciliter le développement d'applications, en ayant le rôle de « chef d'orchestre », pour la gestion des différentes ressources : temps processeur, mémoire, accès aux périphériques… Il serait possible de s'en passer et donc de faire du *bare-metal programming* (cf. cours Systèmes Embarqués), mais cela nécessiterait une connaissance très fine du processeur TI Sitara et impliquerait un processus de développement plus long pour les différents projets.

Le développement d'applications pour la carte Beaglebone se fera à partir d'un PC sous **Linux Debian** avec une chaîne de développement croisée basée sur l'outil **Buildroot**. Le système d'exploitation Linux Debian sera mis en œuvre sur une machine virtuelle « VirtualBox », à partir de Windows.



L'interface eth0 sera de type NAT[1] sous VirtualBox, c'est elle qui permet de communiquer avec internet. Pour accéder à internet, il faut :

1. dans Périphériques/Réseau : cocher « connecter la carte réseau 1 », si ce n'est pas fait ;
2. ouvrir un terminal et taper : *dhclient eth0*, afin de renouveler l'adresse IP de l'interface (en **mode super-utilisateur**)
3. ouvrir le navigateur *Iceweasel* pour tester
4. si cela ne fonctionne pas, il peut être utile de configurer le proxy utilisé par le navigateur (*Preferences/Advanced/Network.Settings*) :
   Manual proxy configuration :
   HTTP Proxy: cache.univ-st-etienne.fr
   Port : 3128
   cocher la case, utiliser pour tous les protocoles.

[1]NAT signifie Network Address Translation. Ce procédé permet à la machine virtuelle d'accéder à internet, via l'adresse IP de l'ordinateur.

L'interface eth1 est de type « Pont » (*bridge*), elle permet de communiquer avec la carte BeagleBone, son adresse IP doit donc être dans le **même sous-réseau que la carte (192.168.101.x)**.

Partage d'un répertoire entre pc et machine virtuelle :

- dans **périphériques → dossiers partagés** : ajouter (si ce n'est pas déjà fait) le dossier Windows que vous souhaitez partager. Par exemple *D:\SEC\mv_share*.
  Indiquer le nom du dossier. Par exemple : mv_share.
  Cocher : Configuration permanente

- Dans un terminal Linux, créer un dossier de partage : *mkdir /home/tpuser/share*

- Dans un terminal Linux, **en mode super-utilisateur**, taper la commande suivante :
  mount -t vboxsf mv_share /home/tpuser/share

Le partage est normalement actif.

Consignes générales

La carte Beaglebone s'alimente **en 5V** soit directement via le port USB, soit à l'aide d'une alimentation externe configurée en 5V.

Afin que ces séances de TD/TP soient profitables et en vue des examens finaux, pensez à prendre des notes au fur et à mesure : commandes Linux utiles, réponses aux différentes questions posées, utilisation des différents protocoles présentés...

Séance n°1 : Prise en main de la carte Beaglebone

Compétences visées :

- découverte de l'architecture Linux embarqué ;
- communication avec une cible (carte BeagleBone) par interface série ;
- mise en place et utilisation d'une liaison pc/cible par protocole Telnet ;
- mise en place et utilisation d'un partage de fichiers pc/cible par protocole NFS ;
- développement d'applications en langage C : chaine de *cross-compilation*, fichier *Makefile*...

I.1 Pilotage de la carte Beaglebone

I.1.1 Accès au système de développement

Pour le développement d'application sur la carte Beaglebone, nous allons utiliser une machine virtuelle de type Linux Debian. Deux utilisateurs sont définis :

- **root** (super-utilisateur avec tous les droits sur la machine), mot de passe **admintest**
- **tpuser** (utilisateur classique, avec des droits réduits) mot de passe **tpuser**

Ce système invité est exécuté sous Windows via le logiciel de virtualisation VirtualBox.
Si votre machine n'apparaît pas dans les machines installées, aller sous le répertoire D:\SEC, et choisir le répertoire de la machine adaptée au TP. Ajouter cette machine à Virtualbox, puis la lancer.

I.1.2 Pilotage de la carte par le port USB

La carte Beaglebone étant un système embarqué, elle ne dispose pas, par défaut, d'interface homme-machine. Vous devez donc piloter la carte à l'aide d'une interface externe. La solution initiale retenue ici est d'utiliser une interface USB associée à un convertisseur série/USB côté système embarqué, de la relier au PC à l'aide d'un câble USB et d'utiliser un logiciel de communication sous Linux.

Lors de l'utilisation d'une machine virtuelle, il est nécessaire de **raccorder « manuellement »** vos périphériques USB à cette machine virtuelle. Pour la liaison avec la carte Beaglebone, il faut utiliser le menu de la machine virtuelle :

périphériques → USB → FTDI Beaglebone :

un premier clic connecte la carte BeagleBone à Linux,
un second clic, la rend à Windows.

Si cela a fonctionné, FTDI Beaglebone doit être coché dans le menu périphériques/USB.

Le logiciel choisi sous Linux pour communiquer avec la carte, via l'interface série, est ***gtkterm***. C'est l'équivalent sous Windows de PuTTY ou TeraTerm. Plusieurs solutions pour lancer ce logiciel :

1/ ouvrir une fenêtre terminal puis *gtkterm &*[2]
2/ Menu des applications / Accessoires / Terminal pour port série

Ce logiciel est préconfiguré pour se connecter sur le port ttyUSB0, avec un débit de 115200 bits/sec.
Il arrive que *gtkterm* « *freeze* » et que vos commandes ne soient plus prises en compte. Il suffit alors de presser les touches *Ctrl+O,* de cliquer sur *default* et la situation revient à la normale.

[2]Pour rappel, le symbole & après une commande dans un terminal sous Linux permet de lancer cette commande en arrière-plan et donc de ne pas bloquer l'accès au terminal.

Faire un **reset physique** de la carte (bouton poussoir à côté du port ethernet au-dessus de la connexion USB). Vous êtes alors au niveau du **bootloader U-Boot**, attendez quelques secondes et le démarrage automatique du système d'exploitation Linux embarqué a lieu. Pour éviter l'attente de la connexion réseau, reliez le connecteur ethernet du système embarqué à une prise sur le bandeau à l'aide d'un câble réseau.
Si cela a fonctionné, vous devez voir apparaître le message :

Welcome to Buildroot
beaglbone login :

Vous pouvez vous connecter sur la carte via l'utilisateur **root, sans mot de passe.** Cela n'a pas d'incidence sur la sécurité car votre carte n'est accessible qu'en local. Si elle était accessible, via une interface réseau, il faudrait bien entendu mieux la protéger !

A cet instant de la séance vous êtes connectés sous le système Linux de la carte et vous transmettez les commandes à la carte Beaglebone via la liaison USB. Toutes les commandes que vous tapez sous *gtkterm* de votre PC sont envoyées caractère par caractère à la carte Beaglebone via la liaison USB. Toutes les informations de retour sont aussi envoyées caractère par caractère par la carte Beaglebone vers le PC et affichées sur la fenêtre *gtkterm*. Les commandes que vous allez passer à la carte Beaglebone seront interprétées dans un *shell* UNIX en mode texte.

I.1.3 Découverte de la structure des répertoires du Linux embarqué.

La commande *uname -r* permet de connaître la version du noyau Linux utilisé par la carte. La version actuelle est la 6.3.5 (mai 2023). Utilise-t-on la version la plus récente ?

Explorer l'arborescence du système de fichiers à partir de la racine, comparer à celle du système hôte :
Commandes :

cd /
ls -la

Explorer les répertoires suivants et en déduire leur contenu et leur rôle dans le système Linux :

/dev
/etc
/lib
/proc
/usr

Aller dans les répertoires */bin* et */usr/bin* de la carte Beaglebone (répertoires où sont stockés les fichiers exécutables des commandes). Visualiser leur contenu. Comment sont mises en œuvre les commandes ? Où sont « réellement » localisés les fichiers exécutables ?

La dernière question a dû vous amener à voir apparaître le nom *BusyBox*, dans le terminal Linux. *BusyBox* est un logiciel libre, écrit en langage C, conçu comme un **unique fichier exécutable** (environ 1Mo, dernière version avril 2023) donnant accès à de nombreuses commandes standard de Linux (*cd, ls, more, cat…*). L'idée est de regrouper en un seul fichier environ 200 programmes Linux, avec l'avantage de la simplicité et de la souplesse, car il est possible de configurer BusyBox en fonction des besoins. C'est donc une solution très utilisée dans le domaine des systèmes embarqués.

Utiliser les commandes suivantes pour connaître la taille des différents types de mémoire du système :

- *fdisk -l* : mémoire Flash
- *grep MemTotal /proc/meminfo* : mémoire RAM

En utilisant la commande *df -h -T* visualisez les emplacements physiques de montage des systèmes de fichiers. A partir de la taille des systèmes de fichiers, déterminez ce qui est en mémoire RAM et en mémoire Flash.

Aller sous le répertoire */var* : se situe-t-il en RAM ou en Flash ? Comment la possibilité d'écriture fréquente de certains sous-répertoires (*/var/log* par exemple) est-elle gérée ?

Tout le système de fichiers monté à la racine / est bien dans la mémoire Flash. Dans le répertoire *var*, on trouve des liens symboliques vers */tmp* qui lui est en mémoire RAM (système de fichiers *tmpfs*). Cela permet un accès plus rapide en écriture/lecture, mais les données disparaissent à chaque démarrage du système. */dev/shm* (mémoire partagée), */dev* (fichiers spéciaux des périphériques) et */run* (données volatiles d'exécution) sont aussi stockés en RAM.

Afin de bien maîtriser les divers accès au système embarqué, nous allons maintenant configurer les accès réseaux entre le PC et la carte Beaglebone. Ceci permettra d'utiliser des protocoles différents afin de communiquer avec la carte (transfert de fichiers, partage de répertoires…).

I.2 Communications réseaux

I.2.1 Test de la liaison Ethernet entre le PC et la carte Beaglebone

Depuis la carte Beaglebone, vérifier la configuration de sa carte réseau : commande *ifconfig -a* ou *ip addr* . Une adresse IP lui a été attribuée au démarrage, mais il faut sans doute la changer. Pour permettre la communication entre tous les postes choisir l'adresse privée *192.168.101.« n°carte »*. Attribuer cette nouvelle adresse :

ifconfig eth0 192.168.101.xxx netmask 255.255.255.0

où xxx est le numéro de votre carte Beaglebone.

La machine virtuelle sous Debian possède 2 cartes réseaux : une (*eth0*) en NAT et DHCP qui lui permet de communiquer sur internet (vous n'y toucherez pas), et une deuxième (*eth1*) en pont qui va vous permettre de communiquer avec votre système embarqué. Il faut donc configurer *eth1* dans le même sous-réseau que l'interface de la carte Beaglebone avec l'adresse privée 192.168.101.« n°PC -100». Attribuer la nouvelle adresse avec la commande (dans une fenêtre terminal) :

ifconfig eth1 192.168.101.xxx
ou ip addr add 192.168.101.xxx dev eth1

Remarque 1 : il faut être en mode *root* pour effectuer cette commande : *su*.
Remarque 2 : dans les nouvelles versions de Linux la commande *ifconfig* est considérée comme obsolète.

Tester les communications entre les 2 interfaces par des commandes *ping*. Quand un ping réussit vous devez voir apparaître le message suivant :

```
# ping 192.168.101.183
ping 192.168.101.183 ( 192.168.101.183) : 56 data bytes
64 bytes from 192.168.101.183 : seq=0 ttl=128 time=2.456ms
64 bytes from 192.168.10&.183 : seq=1 ttl=128 time=0.736ms
```

A cet instant de la séance, le PC sous Debian et la carte Beaglebone ont établi un lien réseau via Ethernet.

I.2.2 Pilotage de la carte par liaison Ethernet via le protocole Telnet

L'objectif de cette partie est de pouvoir piloter la carte via le lien ethernet en utilisant le protocole Telnet. Dans notre cas, la carte Beaglebone est la machine distante (donc le **serveur Telnet**) et le PC sous Debian propose l'interface homme-machine (c'est le **client**). Ainsi, l'utilisateur a l'impression de travailler directement sur la machine distante.
Un serveur Telnet est installé par défaut dans le système de fichiers *rootfs* de la carte Beaglebone. De même, un client telnet existe par défaut sur le PC Debian.

a) Lancement du démon Telnetd sur la carte Beaglebone

La carte Beaglebone possède un serveur Telnet. Vérifier par la commande *ps aux | grep telnet* qu'il a été lancé au démarrage. Si ce n'est pas le cas, il faut le faire manuellement en lançant la commande *telnetd* dans la fenêtre *gtkterm*.

b) Prise en main à distance de la carte via la commande cliente telnet

Ouvrir une nouvelle fenêtre Terminal sous le *PC Host* puis taper la commande : *telnet IP_Beaglebone,* IP_Beaglebone étant bien sûr à remplacer par l'adresse IP de votre carte. Pour des raisons de sécurité, utiliser comme **login : *user1*, mot de passe : *user1*** (un accès direct en *root* est déconseillé en telnet, pour des raisons de sécurité. Le mot de passe serait en effet trop facilement accessible.).

Vous pouvez taper des commandes Linux sous cette fenêtre Terminal pour vérifier que vous accédez à distance à la carte Beaglebone.

A cet instant de la séance, 3 fenêtres sont ouvertes sur le PC Host :

- Une fenêtre Terminal qui permet de lancer des commandes Linux sur le PC Host ;
- Une fenêtre Terminal où est lancée la commande *telnet* permettant de piloter à distance la carte Beaglebone via la liaison Ethernet ;
- Une fenêtre *gtkterm* qui permet de piloter la carte Beaglebone via la liaison USB ;

I.2.3 Transfert de fichiers entre le PC Host et la carte Beaglebone via TFTP

Pour transférer des fichiers du PC Host vers la carte Beaglebone, on peut utiliser le lien Ethernet et le protocole **TFTP**.
TFTP (pour *Trivial File Transfert Protocol*) est un protocole de transfert de fichiers entre un serveur et client (couche 6 du modèle *OSI*). Ce protocole est simple et rapide, mais il fonctionne sans authentification et l'utilisateur n'est pas averti en cas de pertes de données. Il faut par exemple connaître à l'avance le nom du fichier que l'on veut récupérer. TFTP utilise le port 69 et s'appuie sur le protocole UDP (*User Datagram Protocol,* couche 4 du modèle OSI) pour le transport des données.

Dans cette configuration, le PC sous Debian sera le **serveur TFTP** et la carte Beaglebone **le client**.

a) Installation du serveur TFTP sur le PC Debian **(mode super-utilisateur)**

1. Installer le paquet *tftpd-hpa* qui se trouve dans */opt/packages/server* par la commande suivante : *dpkg -i tftpd-hpa_5.2[...].deb*

2. Editer le fichier *tftpd-hpa : gedit /etc/default/tftpd-hpa &*
   Changer la ligne *TFTP_DIRECTORY* en *TFTP_DIRECTORY= "/tftp_dir"*

3. Créer le répertoire */tftp_dir*, répertoire accessible par un client tftp externe, taper dans une fenêtre Terminal du PC Host :
   mkdir /tftp_dir
   chmod 777 /tftp_dir

Quelle est l'utilisé de cette dernière commande ?

4. Relancer le service par la commande : *systemctl restart tftpd-hpa*

5. Ne pas oublier de revenir en mode tpuser : *exit*

b) Création du fichier à transférer dans /tftp_dir sur le PC Host (en utilisateur tpuser)

Sous le répertoire */tftp_dir*, taper la commande *nano hello*, puis rentrer quelques caractères, sauvegarder (*Ctrl + S*) puis fermer l'éditeur de texte (*Ctrl+X*).

nano est un éditeur de texte simple et léger présent sur de nombreuses distributions linux.

Vérifier le contenu du fichier : *more hello*

Quelle est l'utilité des commandes *more, less* et *most* sous Linux ?

c) Transfert de fichiers vers la carte Beaglebone

Le but est de transférer le fichier précédent vers le répertoire *tmp* de la carte Beaglebone.

1. Placer vous dans le répertoire */tmp* de la carte Beaglebone

2. Lancer la requête tftp cliente :
   tftp IP_du_serveur –r hello –g -*r* signifiant : *remote* et -*g* : get

3. Vérifier qu'une copie du fichier se trouve bien dans le répertoire /tmp

I.2.4 Utilisation de NFS pour le partage de fichiers

Plutôt que de copier le fichier physiquement sur la carte, nous allons voir une méthode pour l'utiliser à distance via un partage de fichiers. Le protocole NFS (*Network File System*) permet à un ordinateur d'accéder à un système de fichiers distant (présent physiquement sur un autre ordinateur) via le réseau. Ceci permet de partager des données principalement entre systèmes UNIX.

a) Installation d'un serveur NFS sur le PC Host **(mode super-utilisateur)**

1. Installer le paquet *nfs-kernel-server* qui se trouve dans */opt/packages/server* par la commande suivante : *dpkg -i nfs-kernel-server_[...].deb*

2. Créer le répertoire */share*, répertoire qui contiendra tous les fichiers que le serveur exportera, taper dans une fenêtre Terminal du PC Host :
   mkdir /share
   chmod 777 /share

3. Configurer les répertoires exportés : modifier le fichier */etc/exports* en y ajoutant une ligne exportant le répertoire */share* :
   */share *(rw,no_root_squash,sync,no_subtree_check)*

Le détail des différentes options choisies ci-dessus dépasse le cadre de la 2e année. Les seules à connaître sont :

* : répertoire accessible quelque soit l'adresse IP du client ;
rw : répertoire accessible en lecture et écriture (par défaut, le partage est en lecture seule).

4. Relancer le service par la commande :
   systemctl restart nfs-kernel-server

5. Visualiser quels répertoires sont exportés par la commande *exportfs*.

6. Ne pas oublier de revenir en mode tpuser : *exit*

b) Utilisation du partage de fichiers côté carte Beaglebone

1. Vérifier que le point de montage */usr/local/bin* existe sur la carte Beaglebone

2. Rendre le montage effectif par la commande suivante (le mode super-utilisateur peut être nécessaire) :
   mount -t nfs -o nolock IP_serveur:/share /usr/local/bin

3. Créer un fichier dans */share* sur le PC Host et visualiser le contenu de */usr/local/bin* à partir de la carte Beaglebone : *nano test_server*

4. Créer un fichier dans le répertoire */usr/local/bin* à partir de la carte Beaglebone, puis visualiser le contenu de */share* sur le PC Host :
   echo « ceci est un test NFS » > test_client

I.3 Développement d'applications en C

Afin de développer des programmes pour un système embarqué, il est nécessaire de mettre en place les outils de développement (compilateur, debugger...) sur une machine tierce. En effet, le système embarqué, du fait de ses faibles ressources, ne peut pas héberger un environnement complet de développement. On va donc utiliser un BSP (*Board Support Package*) installé sur la machine virtuelle. Ce BSP va être utilisé pour créer une application pour la carte Beaglebone sous forme d'un fichier exécutable adapté à son processeur ARM. Ce BSP représente 5 Go de données sur le système hôte Debian.
Il faut normalement assurer une compatibilité entre la version du système d'exploitation et celle de l'application. Nous verrons dans une séance ultérieure comment reconstruire ce BSP pour assurer un développement homogène.

Le BSP est basé sur l'outil Buildroot, installé à partir de l'arborescence */sec/buildroot*, il comprend :

- les outils de compilation/chaînage croisés dans un sous-répertoire *output/host/usr/bin ;*
- les fichiers sources du système embarqué : le noyau (kernel), le système de fichiers (*rootfs*), le chargeur (*bootloader*) dans le sous-répertoire *output/build.* Chaque composant est regroupé dans un sous répertoire contenant les sources et tous les fichiers nécessaires à la compilation (*Makefile…*) ;
- une arborescence des fichiers cibles rassemblés dans une structure équivalente au système cible sous un sous-répertoire *output/target ;*
- les fichiers binaires nécessaires à la création de la Flash, ou à un système de fichiers complet pour un partage nfs sous le sous-répertoire output/images ;
- et beaucoup d'autres choses que nous verrons au fur et à mesure !!!

Sur votre machine virtuelle tout le BSP sera donc stocké à partir de l'arborescence */sec/buildroot* dans un disque virtuel spécifique. Le répertoire de développement des applications sera sous votre compte utilisateur (*tpuser*) lui aussi sur un disque virtuel différent.

Dans la suite de l'énoncé, la carte sera pilotée via telnet.

I.3.1 Compilation « manuelle »

a) Création des répertoires et du fichier source

Pour bien identifier les programmes que vous allez développer, vous allez créer un répertoire sous votre compte *tpuser* : */home/tpuser/se1* qui vous servira de répertoire de travail.
Sous ce répertoire copier le fichier source nommé *affiche.c situé sous Mootse* :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void affiche_fct (int v)
{
	printf ("valeur= %d", v) ;
}

int main (int ac, char **av)
{
	for (uint8_t i = 0 ; i < 5 ; i++)
	{
		affiche_fct (i);
	}
	return 0 ;
}
```

b) Construction de l'exécutable

1. Rechercher dans l'arborescence le répertoire de stockage du cross-compilateur *arm-linux-gcc* :
   find / -name « arm-linux-gcc »

2. Regarder le contenu de ce répertoire. Quelle est la fonction des exécutables suivants :
   arm-linux-as
   arm-linux-gcc
   arm-linux-ld
   arm-linux-objcopy

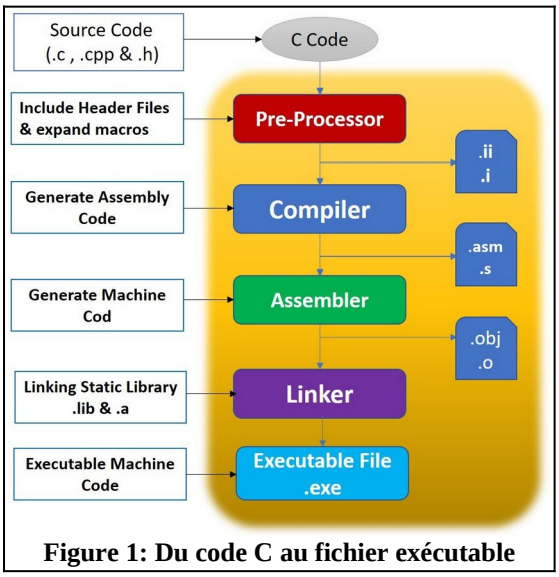


Figure 1: Du code C au fichier exécutable

3. Ajouter ce répertoire au *PATH Linux*, afin que ses fichiers exécutables puissent être appelés depuis n'importe quel répertoire du système de fichiers.

Attention folder_arm_linux_gcc est à remplacer par le chemin complet du répertoire adéquat.

export PATH=$PATH:/folder_arm_linux_gcc
echo $PATH vérification de la bonne modification du *PATH*

4. Compiler *affiche.c* à l'aide de la commande de cross-compilation :
   arm-linux-gcc affiche.c -g -ansi -std=c99 -o affiche

Quelle est l'utilité des options *-g, -ansi, -std=c99* et *-o* ?

5. Vérifier que le fichier affiche pourra bien s'exécuter sur un processeur de type *ARM 32 bits* :
   file affiche

6. Essayer d'exécuter ce fichier sur le système hôte : que se passe-t-il ? Etait-ce prévisible ?

c) Chargement et exécution sous le système cible

Pour lancer l'exécutable sur le système cible, il faut qu'il soit présent dans son système de fichiers. Lors d'un développement de nombreuses tentatives vont être faites. Pour éviter qu'elles entraînent des écritures multiples dans la mémoire flash, il est préférable d'utiliser une autre solution basée sur le partage de fichiers sur un réseau. Nous pouvons donc utiliser le partage NFS précédent (*/share* vers */usr/local/bin*).

Remarque : l'utilisation du répertoire */usr/local/bin* (qui fait normalement partie du *PATH* de la carte Beaglebone) permet d'avoir une exécution sans avoir à taper le chemin complet de l'exécutable.

1. Copier le fichier affiche du répertoire de travail vers le répertoire exporté :
   cp affiche /share

2. Tester en lançant l'exécutable sur le système cible.
   Vérifier que vous pouvez lancer votre exécutable de n'importe quel répertoire du système cible. Il se comporte alors comme une commande sur le système cible.

3. Editer le fichier source et modifier le (boucle 6 fois plus longue) puis recompiler-le. Recopier le nouveau fichier exécutable dans le répertoire adéquat et tester.

Le programme affiche paraît très simple, mais il existe pourtant un point de difficulté jusqu'à présent non abordé. La fonction *printf* en langage C transmet un flux de données formatées, vers la sortie standard définie, nommée *stdout*. Pour un ordinateur, la sortie standard est « naturellement » l'écran, mais qu'en est-il pour un système embarqué, sans écran ? Dans la plupart des systèmes embarqués, s'est à l'utilisateur de définir la sortie standard, par exemple un écran LCD ou un contrôleur UART. C'est ici la seconde solution qui a été utilisée. Les données formatées transmises par *printf* sont donc envoyées, via le contrôleur UART au *pc host* par le câble USB et peuvent ainsi s'afficher dans la console *gtkterm*. Vous voulez une preuve ? Côté carte BeagleBone, taper les commandes suivantes :

```
cd /proc/self/fd
ls -la
```

0 (*stdin*, entrée standard), 1 (*stdout*, sortie standard) et 2 (*stderr*, gestion des erreurs) sont bien des liens symboliques vers */dev/ttyO0*, qui représente le contrôleur UART. La commande suivante, permet par exemple de faire un affichage dans la console *gtkterm* :

echo « test » > /dev/ttyO0

En résumé, attention à l'utilisation des fonctions entrées/sorties (*printf* ou *scanf*) sur un système embarqué !

I.3.2 Automatisation de la compilation

Comme il est assez fastidieux de retaper à chaque fois la ligne de commande de compilation nous allons voir quelques méthodes pour simplifier le développement.

a) Utilisation d'un script de compilation

Placer la commande de compilation dans un fichier texte nommé *compilauto* sous */home/tpuser/se1*. Modifier le fichier source (boucle plus longue x10), recompiler-le avec le script *compilauto*. Recopier et tester.

Commande utiles (à comprendre) :

```
chmod +x compilauto
./compilauto
```

b) Finalisation du script

Placer aussi la commande de copie de l'exécutable dans le script. Modifier, compiler, tester.
Que se passe-t-il si vous relancez le script sans avoir modifié le fichier source ?

Pour éviter ces actions inutiles nous allons utiliser un utilitaire très utile permettant de gérer des scripts conditionnels : l'utilitaire *make*.

c) Ecriture d'un *Makefile*

A partir d'un éditeur texte créer, dans le répertoire de travail */home/tpuser/se1*, un fichier nommé *Makefile* (**avec une majuscule !**), son contenu sera le suivant :

affiche : *affiche.c*
« chemin »/arm-linux-gcc affiche.c -g -ansi -std=c99 -o affiche

La syntaxe générale d'un fichier *Makefile* est la suivante :
cible : dépendances séparées par des espaces
commandes à exécuter

où les cibles et les dépendances sont en général des fichiers. Pour que tout soit logique, la commande exécutée doit correspondre à la création de la cible à l'aide des dépendances.
Attention l'intervalle après le « : » de la cible est une **tabulation** (les commandes sont alignées sur cette tabulation).

En lançant l'utilitaire *make*, les commandes seront exécutées si la date de la cible est antérieure à celles des dépendances. Cette vérification est faite de manière arborescente en commençant par la première cible puis make recherche si les dépendances ont elles-mêmes des dépendances, et ainsi de suite.

d) Compilation conditionnelle

Modifier le fichier *affiche.c*.
Sous le répertoire contenant vos fichiers sources et donc le *Makefile*, lancer la commande :
make affiche
Que-se passe-t-il ?
Relancer une deuxième fois la commande *make affiche*.
Que se passe-t-il ? Commentaires ?

e) Intégration de la copie de l'exécutable vers le répertoire partagé

Ajouter de même la copie du fichier affiche dans le répertoire partagé, modifier *affiche.c* et faire un *make* puis tester l'exécution.
Qu'est ce qui n'est pas logique dans ce *Makefile* ? Modifier le en conséquence.

Séance n°2 : Mise en œuvre d'un projet complexe (serveur WEB)

Cette séance illustre le développement d'un projet complexe (*Makefile*, fichiers source multiples...) et la mise en œuvre du résultat au sein du système d'exploitation du système embarqué.

Compétences visées :

- gestion d'un projet C de plusieurs fichiers ;
- utilisation d'un *Makefile* avancé ;
- compilation croisée et configuration d'un server WEB de type Boa ;
- utilisation du server WEB Boa : page html et script CGI (page dynamique) ;

Dans cette séance la carte sera pilotée via telnet.

II.1 Gestion d'un projet « complexe »

Afin de se rapprocher d'un vrai projet industriel, nous allons découper le fichier *affiche.c* précédent en plusieurs fichiers. Dans cette phase, la compilation (traduction du langage C vers le code machine) et le chaînage (construction de l'exécutable à partir de tous les fichiers objets .o) seront séparés en 2 étapes. Les fichiers objets (**.o*) issus de la compilation auront le même nom que les fichiers sources, mais pas la même extension.

II.1.1 Fichiers sources multiples

a) Découpage des fichiers sources

Placer la fonction *affiche_fct* dans le fichier *affichef.c* et la fonction *main* dans le fichier *affichemain.c*., en distribuant les directives *#include* de manière pertinente entre les 2 fichiers.

b) Test et ajout du fichier inclus.

Compiler manuellement le fichier *affichemain.c :*

arm-linux-gcc affichemain.c -c -ansi -std=c99 -o affichemain.o

-c : stoppe le processus de compilation (cf. figure 1) avant l'édition de liens. On obtient donc un fichier objet, c'est-à-dire un fichier contenant du code machine, mais dans lequel il manque des liens entre les différents symboles. Par exemple, dans notre cas, où est le code correspondant à la fonction *affiche_fct* ?
-std=c99 : compilation en respectant la norme C99

Quel est le résultat de cette compilation ? Si *arm-linux-gcc* n'est pas connu, penser à mettre à jour le *PATH* Linux.

Déclarer le prototype de la fonction *affiche_fct*, dans un fichier d'en-tête (*header file*), nommé *affichef.h.* Ajoutez alors dans le fichier *affichemain.c :*

#include « affichef.h »

Recompiler (avec l'option -c) les deux fichiers *affichemain.c* et *affichef.c*.
Chaîner ses 2 fichiers :

arm-linux-gcc affichef.o affichemain.o -o affiche

c) Automatisation

Modifier le fichier *Makefile* (fichier disponible sous Mootse) en conséquence :

```
affiche:        affichemain.o  affichef.o
                /sec/buildroot/output/host/usr/bin/arm-linux-gcc affichemain.o affichef.o -o affiche
                cp affiche /share

affichemain.o: affichemain.c
                /sec/[...]/arm-linux-gcc affichemain.c -c -g -ansi -std=c99 -o affichemain.o

affichef.o:     affichef.c
                /sec/[...]/arm-linux-gcc affichef.c -c -g -ansi -std=c99 -o affichef.o
```

Tester après avoir modifié le nombre de boucles.

Modifier un seul fichier, comment réagit le *make* ?

d) Enrichissement du *Makefile*

Pour éviter de répéter à chaque fois le chemin complet du cross-compilateur, on peut le mettre dans une variable d'environnement. Ajouter, en première ligne du fichier *Makefile* :

```
CC = /sec/buildroot/output/host/usr/bin/arm-linux-gcc
```

Le chemin du compilateur sera ainsi remplacé par *$(CC)*.

De même, pour éviter de répéter les options de compilation à chaque ligne, il est possible de créer une variable dédiée :

```
CFLAGS = -c -g -ansi -std=c99
```

Les options de compilation seront ainsi remplacées par *$(CFLAGS)*.

Enfin, il est possible d'ajouter une cible *clean* permettant de « nettoyer le projet » :

```
clean:
        rm -rf *.o
        rm affiche
```

Modifier et tester votre *Makefile*, en suivant les indications données ci-dessus.

II.2 Compilation d'un serveur WEB (Boa) à partir des sources

II.2.1 Présentation

Boa est un serveur WEB (http) libre utilisé dans les systèmes embarqués. L'installation d'un serveur WEB sur un système embarqué va permettre de créer une interface simple de gestion des applications via l'utilisation d'un navigateur (client http : Firefox, Brave, Opera …). Il suffira que l'application crée des pages WEB sur le serveur avec une syntaxe html pour que l'on puisse visualiser à distance des informations présentes sur la carte. Le lancement d'applications résidentes pouvant se faire via une interface CGI (Common Gateway Interface), celle-ci est aussi gérée par le serveur Boa.

III.2.2 Conditions de développement

Par défaut, le système de fichiers (*rootfs*) enregistré en mémoire Flash du système Beaglebone ne contient pas les fichiers nécessaires à la mise en œuvre de ce serveur. Il faut donc insérer dans ce système

de fichiers, les fichiers et répertoires nécessaires à Boa. Il faudra donc créer un système de fichiers complet que nous exporterons.
Afin de bien comprendre le mécanisme d'installation d'une nouvelle application dans un système de fichiers existants, nous allons faire cela en pas à pas et non en passant par la configuration du BSP. Ce mécanisme comprend plusieurs étapes : chargement de l'application sous forme de fichier archive compacté ou non (xxx.tar.gz ou xxx.tar), décompactage, application des correctifs (patches), construction des exécutables (compilation, chaînage), mise en oeuvre d'un script d'installation

Vous allez donc créer un sous-répertoire **se2** sous /home/tpuser et y travailler afin que tous les fichiers restent dans ce répertoire (ce qui évitera de polluer l'environnement de développement). L'exécutable sera construit à l'aide d'un *Makefile* comme à la séance précédente et recopié dans un répertoire partagé (par exemple /share).

Il faudra alors réaliser un montage du système de fichiers via NFS pour tester notre serveur.

II.2.3) Récupération des sources

Copier l'archive *Boa-tse.0.94.tar.gz* (récupérée par exemple dans */sec/downloads* ou sous Mootse) dans le répertoire */home/tpuser/se2*, puis décompacter l'archive en vous plaçant dans ce répertoire :

tar xvf Boa-tse.0.94.tar.gz

Aller dans le répertoire décompacté et repérer les répertoires et les fichiers intéressants (répertoire des sources, fichiers permettant la compilation, documentation...). Lancer la commande *make*.
Que se passe-t-il ? Est-ce le résultat attendu ?

II.2.4) Adaptation des fichiers sources à la cible

Les sources Linux étant livrées sans choix de processeur et d'environnement de développement nous allons personnaliser les fichiers décompactés afin de les adapter à la carte Beaglebone. L'application Boa étant très simple, il sera relativement « aisé » de l'adapter à notre système. Nous en profiterons pour essayer de comprendre un peu mieux le cycle de développement système sous Linux.

La première étape est de construire des fichiers *Makefile* adaptés à notre système. Normalement cette construction est totalement automatique via le script *configure* présent dans le répertoire principal.
Ce script *configure* utilise certaines variables d'environnement pour personnaliser le fichier *Makefile.in* et le transformer en un fichier *Makefile* utilisable. Comme nous voulons faire de la compilation croisée (vers un processeur ARM) il faudra le préciser (car le script aura détecté un processeur type i686) via des affectations de variables d'environnement.

Pour cela lancer le script *configure* avec les options :

./configure CC="/sec/[...] /arm-linux-gcc" CFLAGS="-Os -pipe -Wall" --host=arm

-Os permet d'optimiser la taille (on développe pour un système embarqué)
-pipe permet de ne pas stocker de fichiers temporaires (travail en RAM donc plus rapide)
-Wall permet de visualiser tous les warning

Refaire un listing des répertoires et comparer avec le listing avant le script.
Visualiser le *Makefile* du répertoire principal. Que fait-il principalement ?
Visualiser de même le *Makefile* du répertoire *./src*. Que fait-il ?

Nous allons maintenant créer l'exécutable.

II.2.5) Compilation et chaînage

Lancer une commande *make* dans le répertoire de Boa.
Cela se passe-t-il correctement ? Où sont générés les fichiers exécutables ?

II.3) Finalisation de l'installation et test du serveur Boa

II.3.1 Copie de l'exécutable dans un répertoire partagé

Nous allons utiliser le partage /share du PC host pour mettre les fichiers nécessaires.

Modifier le *Makefile* pour qu'il place une copie de l'exécutable Boa directement dans le répertoire */share* (c'est à dire */usr/local/bin* pour la carte Beaglebone). Relancer le.

II.3.2 Utilisation de ce partage par la carte

Démarrer la carte BeagleBone puis monter le système de fichiers */share* sur */usr/local/bin* en nfs.
mount -t nfs -o nolock IP_server:/share /usr/local/bin

L'exécutable peut alors être utilisé. Sur la carte *Beaglebone* lancer le serveur par la commande *Boa*.
Que se passe-t-il alors ?

Il manque plusieurs étapes que nous allons voir maintenant.

II.3.3 Récupération des fichiers de configuration

Copier le fichier *Boa.conf* situé dans le répertoire *examples* (du répertoire Boa) dans le répertoire */share* sur le système de fichiers partagé. Relancer. Que se passe-t-il ?

Rechercher le fichier manquant et le placer dans le système de fichiers partagé. Relancer le server Boa.
Que se passe-t-il ?

II.3.4 Fichier *mime.types*

Le fichier mime.types contient l'ensemble des types de fichier reconnu sous Linux et est donc nécessaire au bon fonctionnement du server Boa.
Ce fichier se trouve dans le répertoire *examples* du répertoire Boa. Il suffit donc de le copier dans le répertoire */share* du *pc host* et de modifier le fichier Boa.conf en conséquence, ligne 202 :

MimeTypes /usr/local/bin/mime.types

Relancer le server Boa. Que se passe-t-il ?

II.3.5 Gestion des fichiers de log

Les fichiers de log permettent de répertorier les problèmes de connexion au serveur Boa. Par défaut, ces fichiers de log sont stockés dans le répertoire */var/log/boa/*. Pour ne pas modifier le système de fichiers en mémoire Flash nous allons modifier le fichier de configuration Boa.conf pour que le répertoire de log soit sous le partage */share* (fichier *Boa.conf*, variable ErrorLog /usr/local/bin/log/error_log1). Il faudra bien entendu créer dans le partage /share le répertoire idoine.
Le serveur se lance-t-il ?

Vérifier qu'il s'est bien lancé à l'aide de la commande :

ps -elf | grep Boa

Si ce n'est pas le cas, commenter (symbole #) dans le fichier boa.conf, la ligne indiquant l'emplacement des *access logs*, ces fichiers ne vont pas nous servir dans la suite.

II.3.6 Création d'une page WEB de démarrage

Par défaut, dans le fichier *Boa.conf*, la première page téléchargée (*index.html*) se trouve sous */var/www*.
Pour la même raison que précédemment nous allons donc créer un répertoire www sous */share* et modifier la directive adéquate dans le fichier Boa.conf. Enfin nous allons créer le fichier *index.html* au bon endroit.

Remplir le fichier *index.html* par :

```
<html>
   <head>
      <title> Welcome on the Beaglebone Board </title>
   </head>
   <body>
      <nowiki><h1> This is a simple test </h1></nowiki>
   </body>
</html>
```

Pour tester le serveur Boa, lancer sur le PC Hôte le navigateur Iceweasel et taper l'URL suivante :
http://IP_BeagleBone/ *IP_BeagleBone* étant l'adresse IP de la carte BeagleBone

Pensez bien à désactiver le proxy si ce dernier a été configuré pour accéder à internet !

Vous devez voir apparaître la page correspondant à votre fichier *index.html.*
Sinon,
1 / vérifier le fichier *log/error_log* de votre carte pour connaître le problème…
2/ cliquer avec le bouton gauche de la souris dans le navigateur sur la page blanche et « afficher le code source ».

II.4) Mise en œuvre de script CGI sur le serveur Boa

Le but des scripts CGI est de créer des pages dynamiques, c'est-à-dire dont le contenu varie dans le temps. Par exemple, le but de notre script sera de faire afficher sur une nouvelle page WEB l'adresse IP de la carte à partir d'un lien (*Get the Board ip config*), puis la date et l'heure (*Get the date*). De cette manière il est aisé de renvoyer des informations de la carte vers le système hôte, la mise en forme se faisant au format html côté serveur sur la carte Beaglebone.

II.4.1 Modification du fichier Boa.conf

Ouvrir le fichier *Boa.conf* :
Assurez-vous que la ligne suivante qui se trouve à la fin du fichier soit **non commentée** et modifier le chemin :

```
ScriptAlias /cgi-bin/ /usr/local/bin/lib/cgi-bin/
```

II.4.2) Ecriture du script CGI (affichage de l'adresse IP de la carte)

Créer le répertoire *lib/cgi-bin/* s'il n'existe pas, puis sous ce répertoire créer un fichier script CGI nommé *get_ipconfig.cgi*.

Ajouter les lignes suivantes :

```
#!/bin/sh
echo -e "Content-type: text/html\r\r";
echo -e `/sbin/ifconfig  eth0 | grep "inet addr"`;
```

Attention aux caractères : ` est différent de ‘

Rendre ce fichier exécutable : *chmod +x get_ipconfig.cgi*

Puis modifier la page de démarrage *index.html*

```
<html>
   <head>
      <title> Welcome on the Beaglebone Board </title>
   </head>
   <body>
      <h1> Welcome on the Beaglebone Board </h1>
      <br> <br>
      <a href="http://IP_BeagleBone/cgi-bin/get_ipconfig.cgi"> Get the Board ipconfig </a>
      <br>
   </body>
</html>
```

Fermer le server Boa sur la carte : commande *kill -9 boa*
Relancer le server et tester le résultat à partir du navigateur Linux.

II.4.3) Ecriture du script CGI (affichage de la date et de l'heure)

Refaire la même démarche en faisant afficher la date et l’heure sachant que le programme sous Linux est */bin/date*.

II.5) Modification des sources du serveur (pour aller plus loin)

II.5.1) Changement des messages de log.

A partir des sources du serveur Boa, modifier les messages de log affichés lors de l'accès par un navigateur. Pour cela rechercher dans les fichiers sources celui qui gère les messages de log. Recompiler puis tester.

II.5.2) Emission d’un message sur la console en cas de lecture d’une page manquante.

Afficher un message sur la console de la carte Beaglebone s’il y a une tentative d’accès à l’adresse *http://Board_IP_address/verboten.*

Séance n°3 : Utilisation des GPIOs, mise en œuvre de LED et de boutons poussoirs

Compétences visées :

- configuration et utilisation des ports GPIO via les commandes *shell* ;
- configuration et utilisation des ports GPIO dans un programme C ;
- utilisation des leds et boutons poussoir de la carte BeagleBone ;
- approfondissement des connaissances Linux ;

III.1 Principes d'accès aux périphériques

III.1.1 Présentation

Dans un système d'exploitation multi-tâches comme Linux, l'accès aux périphériques (en général non partageables) se fait de manière indirecte. Le programme utilisateur réalise des **appels systèmes**[3] (*Input Ouput Control* ou *IOCTL*) pour mettre en œuvre les entrées-sorties. Pour cela, le système d'exploitation fournit une couche d'interface constituée de modules logiciels appelés *drivers*. Ils sont accessibles via des **points d'entrée** particuliers situés dans des répertoires spécifiques. Initialement le répertoire */dev* avait cette fonction exclusive. Au cours du temps d'autres points d'entrée ont été définis soit via le répertoire */proc* (plutôt utilisé pour visualiser des informations bas niveau) et via le répertoire */sys* de plus en plus utilisé.

Concernant le répertoire */dev*, chaque entrée correspond à un **couple *driver*/périphérique**.
Exemples : */dev/ttyS0* point d'entrée pour le *driver* pour le port série n°0
/dev/ttyS1 point d'entrée pour le *driver* pour le port série n°1

L'accès aux périphériques se faisant principalement par des lecture/écriture sur ces entrées correspondant à des **pseudo-fichiers**. Les fonctions particulières sont quant à elles accessibles via une fonction nommée *ioctl (input output control)* qui permet un accès bas-niveau aux périphériques (registres et mémoire notamment).

Pour rappel, dans le système de fichiers Linux, il existe 4 types de fichier :

1. les fichiers normaux ou ordinaires (-) ;
2. les répertoires (*d*) ;
3. les liens symboliques (*l*) ;
4. les pseudo-fichiers (*c, p ou b*). Ces pseudo-fichiers servent d'interface entre Linux et différents périphériques (mémoire, GPIO…) et permettent d'afficher des informations ou de configurer ces derniers.

Pour le répertoire */sys*, l'accès est basé sur un principe différent. Chaque périphérique se retrouve dans une arborescence plus complexe, mais les accès sont simplifiés par la création de pseudo-fichiers associés aux opérations à réaliser : par exemple une modification de direction se fera par écriture de la chaîne de caractères *in* ou *out* dans un pseudo-fichier nommé *direction*. L'architecture et le nom de chaque fichier de gestion d'un périphérique sera fonction du *driver* installé. Cela permet un accès à plus haut niveau.

[3]Un appel système ou *system call* est une communication entre un programme (espace utilisateur) et le noyau Linux, afin que ce dernier effectue une action : modification du système de fichiers (*open, write, close*…), contrôle des processus (*kill, fork…)* … Ce fonctionnement permet de fiabiliser et sécuriser le système en séparant clairement les droits entre l'espace utilisateur (*user space*) et l'espace noyau (*kernel space*)

III.1.2) GPIO sous Beaglebone

Le processeur Ti-Sitara-am335x dispose de 4 registres GPIO (*General Purpose Input Output*) de 32 bits qui peuvent suivant leur mode de configuration :

- servir de ports génériques : entrées / sorties numériques
- générer des interruptions[4] sur front montant ou descendant ;
- être multiplexés entre diverses fonctions prédéfinies ;

Nous verrons plus tard comment choisir leur mode de fonctionnement via le ***device tree*** de Linux. A ce stade, le système Linux fournit des points d'entrée pour que les utilisateurs accèdent simplement aux GPIOs via des pseudo-fichiers.

Les ports *gpio* utilisables directement sur la carte BeagleBone sont accessibles via des répertoires spécifiques à leur usage, par exemple, la led *usr3* se contrôle via le répertoire :

/sys/class/leds/Beaglebone:green:usr3

La carte Beaglebone comprend 4 leds dites « leds utilisateur » : usr0, usr1, usr2, usr3.
Dans les répertoires associés à chaque led, plusieurs pseudo-fichiers permettent de réaliser les différentes opérations possibles : *brightness* permet par exemple de contrôler la luminosité (0 ou 1 en tout ou rien car c'est un port d'entrée/sortie binaire).

Concernant les ports génériques, ils doivent **être activés** par réservation dans le répertoire */sys/class/gpio*.
Il faut tout d'abord réserver le port *gpio* pour le programme utilisateur, car ce dernier est une ressource non partageable. Elle doit donc être verrouillée lorsqu'une application l'utilise. Cette réservation nécessite « un peu de calcul » : une multiplication et une addition !
Calcul du numéro du port (ports de 32 bits) :

GPIOx_y = 32*x + y

Par exemple, pour réserver le port GPIO1_23, il faut réserver le port 32 x 1 + 23 = 55.

La réservation se fait par écriture de la valeur dans le **pseudo-fichier *export*** du répertoire */sys/class/gpio*.
Un nouveau répertoire est alors créé avec le nom *gpioN*. Sous ce répertoire, il y a les différents pseudo-fichiers nécessaires à l'usage du port (*direction* et *value* principalement).

III.2) Gestion directe des GPIO via le répertoire /sys

Créer un répertoire /home/tpuser/se3 pour placer les différents fichiers lors du développement.
Mettre en place le partage NFS du répertoire /share vers /usr/local/bin.
Dans les fichiers *Makefile*, une copie des fichiers exécutables sera effectuée vers le répertoire /share, afin de faciliter l'accès côté carte BeagleBone.

III.2.1) Accès à une led de la carte BeagleBone le *shell*.

Utilisation de la « user led n°3 » : on peut y accéder via le répertoire */sys/class/leds/Beaglebone:green:usr3*

Aller dans ce répertoire puis taper : `echo 1 > brightness`

Faire de même pour éteindre la led (pour de longues durées, il est préférable de laisser la led éteinte) :
`echo 0 > brightness`

[4]Lors d'une interruption (*interrupt*), le processeur interrompt le fil normal d'exécution des instructions pour aller exécuter une routine d'interruption. Par exemple, une interruption *timer* peut permettre d'interrompre le processeur toutes les 100ms, pour aller regarder la valeur fournie par un capteur.

III.2.2) Accès à un port gpio générique le *shell*.

On utilisera pour cela la carte d'extension LCD4 (LCD4 cape) qui outre un écran tactile de taille 4 pouces comprend 1 led et 5 boutons.

La led D4 se situe sur le port GPIO1_28. Calculer le numéro N du GPIO suivant la méthode précédente.

Mettre en œuvre les commandes à réaliser pour éclairer ou éteindre la led D4 :

1. aller dans le répertoire de gestion des ports : *cd /sys/class/gpio*
2. réserver ce port en écrivant dans le pseudo-fichier *export* : *echo N >export*
3. aller dans le répertoire de gestion du port qui vient d'être créé : *cd /sys/class/gpio/gpioN*
4. mettre le port en sortie en écrivant dans le pseudo-fichier *direction* : *echo out >direction*
5. allumer et éteindre la led en écrivant dans le pseudo fichier *value* : *echo 0(1) > value*

PS : si vous utilisez la carte LCD7 au lieu de la LCD4, voici les ports GPIO utilisés :

LED D4 : GPIO1_28
ENTER : GPIO0_3
DOWN : GPIO3_16
UP : GPIO1_19
RIGHT :GPIO1_17
LEFT : GPIO1_16

La LED D3 sert à vérifier la bonne alimentation du et n'est donc pas accessible.

III.2.3) Accès en lecture à un port gpio depuis les commandes du *shell*.

De la même manière, on va pouvoir lire l'état des boutons poussoirs.
Les 5 boutons poussoirs sont directement reliés à des ports gpio

- *ENTER* : GPIO0_15
- *DOWN* : GPIO3_16
- *UP* : GPIO1_19
- *RIGHT* : GPIO1_17
- *LEFT* : GPIO1_16

A l'aide des commandes *shell,* lire l'état des boutons :

1. réservation du bouton : écriture dans le pseudo-fichier *export*
2. choix de la direction : écriture dans le pseudo-fichier *direction* (**attention bien le mettre en *in***), car sinon risque de court-circuit entre le port en *out* et le bouton poussoir
3. lecture de value par la commande *more* : *more value*

Quel résultat obtenez-vous lors d'un appui sur le bouton ? Même question sans appui. Que peut-on en déduire concernant les résistances de tirage de la carte : *pull-up* or *pull-down* ?

III.3) Accès avec les fonctions standard, via un programme C

Nous allons utiliser les fonctions C standard de gestion des fichiers non formatés, pour gérer les ports du GPIO : *open, write, read, close.* Ces fonctions sont définies dans le fichier en-tête *fcntl.h* et utilisent un *handle* de type entier pour communiquer.

III.3.1) Clignotement de la LED D4 sur la LCD4 Cape .

Ecrivez le code suivant et compilez, puis testez. Que se passe-t-il sur la carte ?

```
#include <fcntl.h>
#include <stdint.h>
#include <unistd.h>

int main()
{
      int i ;
      int gpio_export, gpio_dir, gpio_led5 ;

  /* port reservation */
      gpio_export = open ("/sys/class/gpio/export", O_WRONLY);
      write (gpio_export,"60",3) ;
      close (gpio_export) ;

  /* port in output direction */
      gpio_dir = open ("/sys/class/gpio/gpio60/direction", O_WRONLY);
      write (gpio_dir,"out",4) ;
      close (gpio_dir) ;

  /* blinking loop */
      for(uint8_t i=0;i<10;i++)
      {
            gpio_led5 = open ("/sys/class/gpio/gpio60/value", O_WRONLY);
            write (gpio_led5,"1",2) ;
            close (gpio_led5) ;
            sleep(1) ;

            gpio_led5 = open ("/sys/class/gpio/gpio60/value", O_WRONLY);
            write (gpio_led5,"0",2) ;
            close (gpio_led5) ;
            sleep(1) ;
       }

      return 0 ;
}
```

III.3.2) Ajout du contrôle par bouton poussoir.

Utiliser le bouton poussoir *ENTER* pour démarrer la séquence de clignotement. Voici les étapes à suivre :

1. configurer le bouton : réservation du port et configuration de la direction ;

2. ajouter une boucle d'attente (type *while),* ayant comme condition de sortie que la valeur du port associé au bouton soit à 0.

Attention : pour enchaîner des lectures de l'état du bouton poussoir, il faut que les fonctions *open* et *close* soient dans la boucle de lecture sinon on relit toujours la même valeur.
Fonctions nécessaires : *open*, *write*, *close* et *read ;*

3. Compiler et tester votre programme.

Le programme réalisé ci-dessus utilise la méthode du *polling*, c'est-à-dire qu'il vérifie en permanence l'état du port lié au bouton poussoir, en monopolisant le temps du processeur. Une solution plus élégante (et professionnelle) aurait été d'utiliser une interruption déclenchée par appui sur le bouton poussoir, libérant ainsi le processeur jusqu'à ce que l'interruption se produise. Cette solution dépasse pour l'instant l'état de vos connaissances.

III.3.3) Simulation d'un télérupteur.

A l'aide des 4 boutons *DOWN*, *UP*, *RIGHT* et *LEFT*, mettre en place un fonctionnement de type télérupteur : un appui sur un bouton allume la led, un nouvel appui sur n'importe quel bouton éteint la led et ainsi de suite.

Pour écrire ce programme, vous pourrez notamment utiliser une fonction indépendante de test d'appui sur un bouton, de prototype : *uint8_t appui(int n) ;*

paramètre d'entrée : *n*, le numéro du port GPIO du bouton à tester ;
valeur retournée : 1, en cas d'appui, 0 sinon.

Fonctions C utiles : *sprintf, strcat, open, read, write, close, sleep*...

III.3.4) Utilisation des informations systèmes (pour aller plus loin)

Le fichier /proc/stat renferme les statistiques d'utilisation du processeur : *more /proc/stat*

Intéressons-nous à la première ligne, de gauche à droite (unité = 10ms) :

- **user** : processus normaux s'exécutant en mode *user* ;
- **nice** : processus *niced* s'exécutant en mode user. Les processus *niced* sont des processus dont la priorité a été changée via la commande *nice* (cf. 3e année) ;
- **system** : processus s'exécutant en mode noyau
- **idle** : temps pendant lequel le processeur est « inoccupé ». Il exécute en fait un processus de fond, dit *idle process*, car un processeur doit toujours avoir une instruction valide à exécuter ;
- les colonnes 5 à 7 concernent les interruptions.

La somme de toutes les colonnes indique donc le temps total de processeur, depuis l'allumage de la carte. La 4e colonne indique le temps de processeur « inoccupé ». La différence entre les 2 indique donc le temps de processeur « occupé ». La charge de travail du processeur (en %) peut donc se calculer facilement.

Ajouter au programme précédent une fonction qui fait clignoter la led proportionnellement à la charge de travail du CPU. (rafraichissement toutes les 5 secondes, par exemple).

Séance n°4 : Développement croisé pour processeur ARM (carte Beaglebone)

Compétences visées :

- configuration et construction d'un BSP : U-boot, noyau Linux, *device-tree* et système de fichiers ;
- démarrage de la carte BeagleBone avec système de fichiers en NFS ;
- modification du système de fichiers Linux (server WEB Boa) ;

IV.1 Environnement de développement croisé

IV.1.1 Introduction

Nous avons vu aux séances précédentes comment un BSP (*Board Support Package*) pouvait permettre le développement d'applications embarquées.
Une fois opérationnel, un système embarqué est autonome et constitue un bloc indivisible comprenant le matériel support (processeur et périphériques), le logiciel de démarrage, un système d'exploitation réduit, les configurations liées au matériel et le logiciel applicatif. La compatibilité de tous ces éléments assure un bon fonctionnement. Les différents blocs logiciel doivent donc avoir été construits avec une même version d'outils et être parfaitement adaptés au matériel. La première étape de réalisation d'un système embarqué est donc la création des outils logiciel, à partir de fichiers sources génériques adaptés à la cible matériel. Le BSP est donc totalement reconstruit, c'est cette étape que nous allons étudier lors de cette séance. Nous compléterons cette séance par une mise en œuvre pratique d'un environnement de développement permettant des tests simples sans réécriture de la flash (via le protocole NFS).

Vu les ressources réduites d'un système embarqué, il est nécessaire de mettre en place les outils de développement sur une machine tierce. Le BSP adapté à la carte Beaglebone et à son processeur ARM représente 5 Go de fichiers sur le système Debian, sa reconstruction complète peut donc prendre plusieurs heures.

Le BSP utilisé est basé sur **l'outil Buildroot** modifié pour être adapté à nos besoins. Il est installé à partir de l'arborescence */sec/buildroot* et comprend :

- Les outils de compilation/chaînage croisés dans un sous-répertoire *output/host/usr/bin*.
- Les fichiers sources des logiciels nécessaires au fonctionnement du système embarqué :
  - le chargeur (*bootloader*) U-Boot
  - le noyau (*kernel*) de type Linux
  - le système de fichiers (*rootfs*)
  - le *device tree* (description de la configuration matérielle de la carte)

  Tous les fichiers sources sont dans le répertoire *output/build* . Chaque composant est regroupé dans un sous répertoire contenant les sources et tous les fichiers nécessaires à la compilation (*Makefile*...).
- Une arborescence des fichiers cibles rassemblés dans une structure équivalente au système cible sous un répertoire *output/target*, cette arborescence permet de modifier de manière statique des éléments du système de fichiers cible
- Les fichiers binaires nécessaires à la création de la Flash, ou à un système de fichiers complet pour un partage nfs sous le répertoire output/images.

Dans cette séance, il n'est pas nécessaire de brancher la carte avant la partie V.2.

Un schéma complet de la structure de développement est donné sur le schéma suivant.

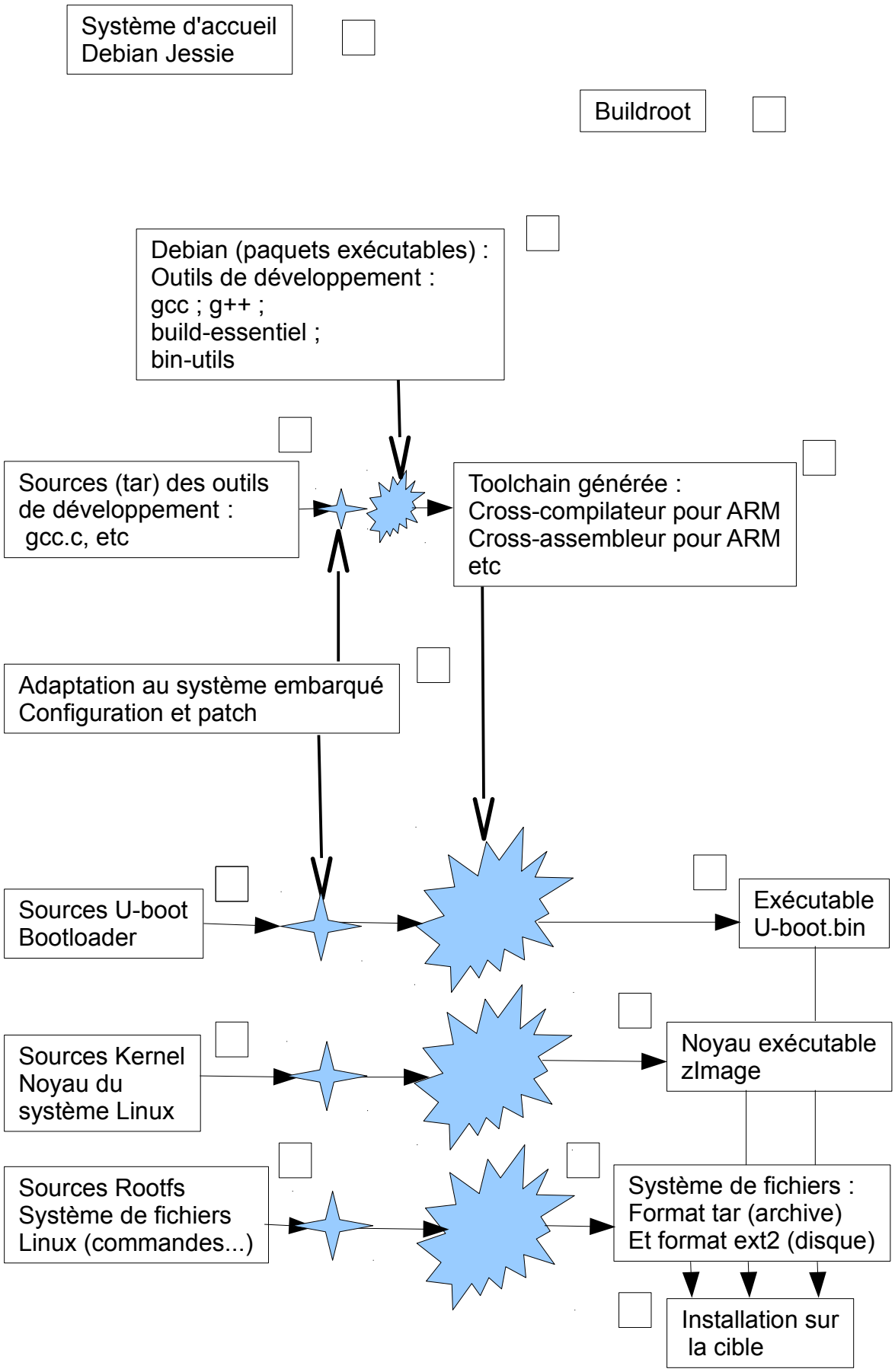


Schéma simplifié de fonctionnement d'un BSP

IV.1.2 Construction du BSP par étapes

1) Installation des paquets Linux nécessaires à la machine support (host)

Pour un gain de temps les paquets (*.deb) debian sont placés dans un répertoire sur la machine (cela évite d'avoir à les télécharger).

Aller dans le répertoire */opt* (répertoire des applications optionnelles), lister les sous-répertoires.
Attention : pour la suite des opérations de cette première étape **vous devrez être *superuser***. Pour cela, une bonne habitude est d'utiliser une **fenêtre terminal spécifique**.

Aller dans le sous-répertoire */opt/packages/gcc*, lancer le script gcc_inst : *bash gcc_inst*
Aller dans le répertoire /opt/packages/build, lancer le script build_inst : *bash build_inst*

Quelles opérations avez-vous faites à cette étape 1 ? Numérotez sur le schéma général le bloc correspondant.

Repasser bien dans la fenêtre relative à l'utilisateur tpuser pour la suite du TP, sinon les modifications seront attribuées à root et donc inaccessibles pour un utilisateur normal.

2) Installation de la structure de base de Buildroot (**mode tpuser**)

Aller sous le répertoire */sec/download*, puis lancer l'extraction du système de base du BSP par la commande tar : *tar xvf buildroot-tse.tar.gz -C /sec*

Attention, pour ne pas polluer le système de fichier, il est impératif d'extraire le fichier archive **vers le répertoire /sec (option -C)**.

Vous avez créer toute une structure de fichiers sous le répertoire */sec/buildroot*
Visualiser cette arborescence, nous verrons plus tard l'utilité de chacun (ou presque !) des fichiers et des dossiers.

Quelles opérations avez-vous faites à cette étape 2 ? Numérotez sur le schéma général le bloc correspondant.

3) Configuration de Buildroot pour la carte Beaglebone (**mode tpuser**)

Pour l'instant l'outil *Buildroot* est un outil générique, adapté à aucune cible spécifique. Nous allons réaliser une première étape simple de personnalisation en lui précisant pour quelle cible (Beaglebone) il va être utilisé.
Se placer dans le répertoire */sec/buildroot* : *cd /sec/buildroot*
Lancer la commande : make beaglebone_defconfig

Cette commande va simplement copier le fichier de configuration par défaut de la carte Beaglebone dans le fichier *.config* dans le répertoire principal. **Cette opération ne doit être effectuée qu'une seule fois car si vous refaites cette opération après d'autres phases de personnalisation vous risquez d'effacer ces dernières.**

Visualiser le fichier *.config* créé par la commande : *more .config*

Quelles opérations avez-vous faites à cette étape 3 ? Numérotez sur le schéma général le bloc correspondant.

4) Construction de la chaîne de cross-compilation pour le processeur (ARM) de la carte Beaglebone (**mode tpuser**).

Cette opération se fait en 2 étapes :

1. récupération des fichiers source ;
2. compilation adaptée à la cible embarquée.

Afin de ne pas encombrer internet, nous allons récupérer les fichiers source directement sur la machine en les copiant du répertoire */sec/download/dltool* dans le répertoire */sec/buildroot/dl* :

cp -a /sec/download/dltool/ /sec/buildroot/dl*

pour cela vous devez vous assurer que le répertoire */sec/buildroot/dl* existe.
L'option *-a* permet de s'assurer que les fichiers copiés conservent leurs droits et métadonnées.

Se placer dans le répertoire */sec/buildroot* : *cd /sec/buildroot*
Lancer la construction de la chaîne de cross-compilation : *make toolchain*

Cela doit prendre moins que quelques minutes mais plus que quelques secondes (normalement la reconstruction prend plusieurs minutes **mais une partie a déjà été pré-compilée** pour cette séance).

La *toolchain* est constituée de programmes exécutables au format i686 mais générant du code adapté au processeur ARM, d'où le nom de cross-compilateur.

Quelles opérations avez-vous faites à cette étape 4 ? Numérotez sur le schéma général le bloc correspondant.

A ce stade là, il peut être intéressant de comprendre le fonctionnement général du BSP qui est basé sur des invocations de l'utilitaire *make*.

- En tapant *make help*, visualisez les différentes possibilités offertes pour l'utilisateur (ne les essayez pas car elles peuvent avoir des conséquences importantes et désastreuses).
- La commande make est contrôlée par un fichier nommé *Makefile*, dont vous devez déjà connaître le principe. Visualisez son contenu sans le modifier : *more Makefile*

5) Construction des différents fichiers binaires nécessaires pour la carte Beaglebone (**mode tpuser**).

Nous allons maintenant générer les différents fichiers binaires nécessaires au fonctionnement de la carte Beaglebone (*bootloader*, noyau, système de fichiers...). Ces opérations peuvent se faire en une seule étape mais nous allons les décliner les unes après les autres.

Dans un premier temps, il faut ajouter dans le répertoire de téléchargement ./dl les fichiers sources normalement téléchargés par internet: ils sont dans le répertoire /sec/download/dlLinux.

cp -a /sec/download/dllinux/ /sec/buildroot/dl*

La (re)construction des fichiers pour la cible se fait par des commandes *make* . A chaque étape vous pourrez consulter le répertoire */sec/buildroot/output/images* pour déterminer les fichiers qui auront été construits.

Etape 5.1 : *make uboot*
Etape 5.2 : *make kernel* (un appel à make linux-menuconfig peut parfois être nécessaire pour éditer le fichier .config et permettre la compilation des fichiers du noyau)
Etape 5.3 : *make device-tree*
Etape 5.4 : *make filesystem*

Quelle est la taille du *bootloader*, du noyau et du système de fichiers ?
Quelles opérations avez-vous faites à cette étape 5 ? Numérotez sur le schéma général les blocs correspondants.

Nous avons à cet instant construit un BSP ainsi que les fichiers nécessaires au fonctionnement de base du système embarqué dans une configuration standard. Suivant les périphériques utilisés et les bibliothèques nécessaires, nous devrons aussi modifier la configuration du BSP. Cette phase de configuration va permettre de choisir les périphériques, fonctions et beaucoup d'autres éléments qui vont se rajouter pour créer un système embarqué complet.
Cette configuration peut-être faite par plusieurs commandes :

make menuconfig	configuration générale, *toolchain*, bibliothèques...
make Linux-menuconfig	configuration du noyau Linux
make busybox-menuconfig	configuration de *Busybox* : boîte à outils des commandes UNIX

En phase de développement standard, il n'est nécessaire de lancer, après chaque modification de configuration, qu'une commande *make* sans argument. Cette commande reprend alors toutes les dépendances. Toutefois, lors de changement important, il est conseillé de reconstruire tout l'ensemble de la chaîne car l'outil Buildroot ne détecte pas forcément toutes les relations.

6) Exportation du résultat de compilation vers la cible

Afin de faire fonctionner le système créé (*Uboot,* noyau Linux...) sur la cible, plusieurs possibilités existent :

1. reprogrammer la mémoire flash contenant les fichiers (ici une SDCard)
2. utiliser un partage par NFS pour conserver les fichiers sur le PC de développement. Cette seconde option est conseillée en phase de développement et est utilisée dans le monde industriel. C'est donc cette deuxième solution que nous allons détailler maintenant.

IV.2. Montage du système de fichiers et/ou du noyau par NFS

Pour ne pas reprogrammer la SDCard trop souvent, vous allez utiliser le **partage NFS** lors de vos développement futurs. Pour cela, il faut construire un système de fichiers pour la carte Beaglebone en local sur la machine hôte, ensuite on doit modifier la procédure de démarrage au niveau du *bootloader*, afin d'indiquer que le système de fichiers est accessible via le protocole NFS.

IV.2.1 Préparation du système de fichier au niveau du système hôte

1) Construction du système de fichiers au format compacté (tar)

Jusqu'à présent le système de fichiers pour la Beaglebone était stocké sous un format ext2 dans la SDCard (équivalent à un disque Linux). Afin de pouvoir le reconstruire à l'endroit choisi, il est nécessaire de le stocker sous un format tar pour conserver les liens logiques ainsi que les droits spécifiques pour les répertoires spéciaux.

Ceci est fait en configurant Buildroot :

commande : *make menuconfig*
menu : *Filesystem*
sous-menu : *Filesystem images*
cocher l'item *tar the root filesystem* à l'aide de la barre d'espace.
Save, *OK* puis *Exit*

Relancer la construction du système par la commande : *make filesystem*
Vérifier dans le répertoire */sec/buildroot/output/images*, la présence de *rootfs.tar*.

2) Décompactage de ce système de fichiers au niveau du système hôte (**mode super-utilisateur)**

Créer le répertoire */targetfs*, s'il n'existe pas : *mkdir /targetfs*
Se placer dans le répertoire */sec/buildroot/output/images : cd ...*
Décompacter le fichier tar pour que celui-ci se place sous */targetfs* par la commande :
tar xvf rootfs.tar -C /targetfs

Vérifier sous */targetfs* la présence du système de fichiers et contrôler sa structure.

3) Compléter le système de fichiers pour faciliter la mise au point (**mode super-utilisateur**).

Créer le répertoire */targetfs/boot* s'il n'existe pas : mkdir */targetfs/boot*
Se placer dans le répertoire */sec/buildroot/output/images : cd …*
Copier dans */targetfs/boot* les fichiers suivants (commande *cp*) :

am335x-bone.dtb	(*device tree* pour la carte BeagleBone)
zImage	(noyau de Linux, sous forme d'un unique fichier binaire)
uEnv.txt	(fichier de configuration utilisateur de la carte)

Si uEnv.txt n'est pas présent, recherche le et copier le : *find /sec/buildroot -name « uEnv.txt »*

4) Partager le système de fichier qui se trouve sous */targetfs (**mode super-utilisateur**)*

Configurer l'exportation du répertoire : pour cela modifier le fichier */etc/exports* en y ajoutant une ligne exportant le répertoire */targetfs* :

*/targetfs *(rw, no_root_squash, sync, no_subtree_check)*

Pour prendre en compte les modifications relancer le service par la commande :

systemctl restart nfs-kernel-server

Visualiser quels répertoires sont exportés par la commande *exportfs*.

V.2.2 Configuration de u-boot pour démarrer en NFS

U-Boot (Universal Boot Loader) est le BIOS[5] présent sur les cartes Beaglebone. Si aucun système d'exploitation n'est requis pour votre application, U-Boot peut être utilisé comme base de développement **mono-tâche**, c'est-à-dire qu'il sait gérer qu'un unique processus.

1) Accès au bootloader u-boot

Si l'on intervient au niveau du noyau Linux, le partage NFS doit se faire dès le démarrage. Il faut alors interrompre le processus de *boot* et accéder au *bootloader U-boot*. Pour cela il faut redémarrer la carte Beaglebone en appuyant sur le bouton de Reset, puis taper un caractère dans les premières secondes de démarrage pour interrompre le démarrage automatique, à partir de la carte SD. La carte BeagleBone est alors sous le contrôle du *Bootloader U-Boot*.

Accéder à U-Boot
Visualiser la liste des commandes disponibles via la commande : help
Visualiser les variables d'environnement par la commande : *printenv*

2) Chargement de la configuration modifiée stockée dans uEnv.txt

Afin de personnaliser le fonctionnement de la carte on peut modifier les variables d'environnement par le chargement du fichier *uEnv.txt (*2046 octets*)*. Ce fichier est normalement dans la première partition FAT de la SDCard.

Lancer la commande : *run loadbootenv*
loadbootenv est une variable d'environnement qui permet à *U-boot* d'aller lire le fichier *uEnv.txt* contenu dans la carte SD. Le fichier est alors transféré vers la mémoire RAM.

Lancer commande : *run importbootenv*
importbootenv est une variable d'environnement qui permet d'afficher un message à l'écran et d'importer les variables d'environnement présentes en mémoire RAM. Cette commande n'a donc **aucun sens** sans la commande précédente !

[5]BIOS signifie Basic Input Output System. Il s'agit d'un firmware issu de la norme IBM PC qui est le premier logiciel à ce lancer au démarrage de l'ordinateur. Le BIOS permet par exemple d'initialisation la carte mère et différents périphériques et de lancer le système d'exploitation. Pour en savoir plus : https://www.techtarget.com/whatis/definition/BIOS-basic-input-output-system

Visualiser les nouvelles commandes ajoutées : *printenv*
Regarder notamment le contenu des variables : *nfsboot, loadaddr, serverip, rootpath*...

Si vous avez des problèmes de visualisation sous *gtkterm,* vous pouvez sauvegarder l'historique via le menu *fichiers* puis *sauvegarder en fichier brut* de *gtkterm*. Cela crée un fichier sur l'hôte que vous pouvez lire plus facilement..

3) Configuration de l'adresse IP de la carte réseau avec la commande :
commande : *setenv ipaddr 192.168.101.xxx* où xxx est le numéro de la carte

4) Configuration de l'adresse IP du serveur NFS avec la commande :
commande : *setenv serverip 192.168.101.yyy* où yyy est le numéro du PC - 100

Testez la communication via un ping depuis la carte vers le PC Host (car sinon le boot NFS ne fonctionnera pas et la carte sera bloquée).
Pourquoi le test dans l'autre sens ne fonctionne pas ?

5) Lancement du système Linux
Taper la commande *run nfsboot,* regarder l'affichage pour identifier les différentes étapes du démarrage :

1. chargement du noyau *zImage* en mémoire
2. chargement du *device tree* en mémoire
3. démarrage du noyau
4. montage du système de fichiers par NFS

IV.3. Ajout d'une application via le BSP (serveur Boa)

Nous allons voir maintenant comment il est possible de compléter les fonctionnalités de la carte BeagleBone via le *BSP* : cela automatise le développement vu à la séance 3.

IV.3.1 Ajout d'un serveur WEB (Boa)

1) Modification de la configuration du système de fichiers
Pour ajouter une commande ou un service à Linux, il suffit de modifier la définition du système de fichiers et de le recréer.

Nous allons donc utiliser la commande : *make menuconfig* (comme nous l'avons fait précédemment)
menu : *Target Packages*
sous-menu : *Networking Application*
Boa (cocher la case à l'aide de la barre espace), *Save*, *OK* puis *Exit*

2) Recompilation du système de fichiers
Il faut ensuite récupérer les sources officielles de Boa (*boa-tse-0.94.14rc21.tar.gz*) sous */sec/download* en copiant ce fichier dans le sous répertoire dl :
cp boa-0.94.14rc21.tar.gz /sec/buildroot/dl

Il reste à reconstruire le système de fichiers, par la commande : *make filesystem*
Si on avait ajouter des *drivers* de périphérique (carte son, écran…), il aurait fallu recompiler le noyau Linux et le système de fichier.

3) Reconstruction du système de fichiers dans le répertoire partagé
Décompacter le fichier *rootfs.tar* dans le répertoire */targetfs : tar xvf rootfs.tar -C /targetfs*

4) Redémarrage de la carte en NFS et test

Cette phase peut-être nécessaire car nous avons modifié le système de fichiers sur lequel était montée la carte BeagleBone (les résultats risqueraient d'être **imprévisibles** sans redémarrage).

Comme au TD sur le serveur Boa, utiliser le navigateur de la machine virtuelle pour tester votre serveur WEB, **en résolvant d'éventuelles erreurs.**

Programmation de la SDCard (pour mémoire)

Opérations à ne pas faire sur les cartes de TP

Pour finaliser le développement d'un système embarqué, il est nécessaire de le rendre autonome. Il faut donc stocker tous les programmes et les fichiers de configuration dans une mémoire non-volatile de la carte BeagleBone (mémoire Flash).

IV.4.1) Procédure à partir d'un système hôte

Lors d'une mise en production de masse, la programmation doit être faite à partir d'un système hôte. Par exemple la reprogrammation d'une carte SD sur notre système peut se faire en insérant un adaptateur sur notre PC permettant de voir le disque flash comme un périphérique USB.

Ces opérations doivent être faite en mode *superuser*, elles peuvent être dangereuses car l'accès aux périphériques est direct, et vous pouvez donc reformater votre disque dur si vous vous trompez de *device*.
La programmation se fait en plusieurs étapes :

1. mise en place de la carte SD dans un slot USB avec l'adaptateur
2. raccordement de la carte SD sous la machine virtuelle (onglet périphérique)
3. effacement de la table de partition : *dd if=/dev/zero of=/dev/sdc bs=1M count=1*
4. création de la table de partition :
```
sfdisk -H 255 -S 63 /dev/sdc <<EOF
0,9,c,*
,124
EOF
```
5. formatage de la partition FAT32 : *mkfs.vfat /dev/sdc1 -n boot*
6. montage de cette partition sur un point du système de fichiers hôte (*/media/boot*), le créer s'il n'existe pas (*mkdir /media/boot*) : *mount /dev/sdc1 /media/boot*
7. copie des fichiers nécessaires au boot : *cp MLO u-boot.img uImage /media/boot*
8. copie directe du système de fichiers compacté (rootfs.ext2 : ce fichier à le format d'un disque, vous pouvez d'ailleurs le monter pour en voir le contenu) dans la deuxième partition :
```
dd if=rootfs.ext2 of=/dev/sdc2 bs=128k
```

Une fois la carte SD reprogrammée, elle peut être réinsérée dans son emplacement sur la carte qui est alors opérationnelle avec la dernière version du système. Ces opérations sont relativement longues en phase de test, c'est pour cela que l'on préfère le partage NFS

IV.4.2) Procédure sans interruption du système embarqué

Lors de la phase finale d'essai, on peut aussi accéder directement à la mémoire flash à fin de modification sans arrêter la carte. En effet toutes les partitions des mémoires flash sont accessibles par montage sous Linux (à condition bien entendu que la version antérieure soit opérationnelle).
Pour une Beaglebone, 2 mémoires flash sont présentes, on les retrouve dans le répertoire des *devices* (/dev) sous les noms de /dev/mmcblk*x*p*y* où *x* est le numéro de la mémoire et *y* le numéro de la partition. On peut alors par montage accéder directement à ces partitions.

Par exemple la partition de boot de la carte SD se retrouve sous */dev/mmcblk0p1* (formatée en FAT32), la partition Linux de la carte SD sous */dev/mmcblk0p2* (formatée en ext3).

De même pour la flash internet sous */dev/mmcblk1p1* et */dev/mmcblk1p2*.

Quoi qu'il en soit la modification directe de ces partitions n'est pas sans conséquence sur le fonctionnement du système et doit être réservée à des personnes averties (comme vous le serez, mais seulement à la fin du module !)

Séance n°5 : Utilisation des Convertisseurs Analogique-Numérique

Compétences visées :

- modification du *device-tree* Linux pour ajouter un périphérique (CAN) ;
- utilisation de ce périphérique pour la mesure de tension via les commandes du *Shell ;*
- utilisation de ce périphérique pour la mesure de tension via l'API C ;
- développement d'une application embarqué complète avec mesure, traitement et affichage de l'information ;

V.1 Présentation

V.1.1 Principe d'accès aux convertisseurs

Le processeur ARM de la carte Beaglebone possède en interne 8 Convertisseurs Analogique-Numérique (CAN) ou *Analog-to-Digital Converter* (ADC). Ils sont accessibles comme des périphériques « classiques » via un *driver*. Dans le cas de la carte Beaglebone, certains convertisseurs (*ain0-ain3*) sont associés avec l'écran tactile (*touchscreen*) . Le *driver* associé aux convertisseurs (*ti_am335x_adc*) est donc en concurrence avec celui de l'écran tactile (*ti_am335x_tsc*). Pour gérer l'accès partagé, un *driver* de plus bas niveau est ajouté (*ti_am335x_tscadc*).

Lors de l'installation réussie des *drivers*, il y a création de toute une arborescence de répertoires au sein du répertoire spécial */sys*. Des pseudo-fichiers (*in_voltagex_raw*) correspondant aux convertisseurs AIN0 à AIN7 sont aussi créés :

répertoire */sys/bus/platform/devices/44e0d000.tscadc/TI-am335x-adc/iio:device0*.

Une simple lecture de ce pseudo-fichier lance une conversion et renvoie la valeur convertie en ASCII. La conversion en Volt se fait ensuite par calcul (**la plage totale du convertisseur 12 bits est de 1,8V**)

V.1.2 Opérations nécessaires pour l'accès à un nouveau périphérique

L'ajout d'un périphérique sous Linux, ici un convertisseur, va nécessiter plusieurs étapes :

a) Ajout de la description physique du périphérique

Il faut procéder dans un premier temps à l'ajout de la description physique du périphérique (adresse des registres, gestion des interruptions, de l'horloge...) dans une structure arborescente appelée *flattened device tree*. Cette structure va décrire l'ensemble du matériel associé au système embarqué : elle est donc **essentielle et sensible**. La syntaxe est basée sur des nœuds (*nodes*) et sous-noeuds (*child nodes*). Chacun des nœuds ayant des propriétés sous la forme d'un couple clef-valeur.

b) Compilation des *drivers*

Les *drivers* des périphériques non essentiels au fonctionnement du noyau ne sont pas intégrés à celui-ci. C'est le cas des *drivers* liés aux convertisseurs analogiques numériques. Il faut donc les compiler et soit les intégrer au noyau soit les utiliser comme des modules autonomes. L'outil Buildroot via les menus de configuration va nous permettre de faire cette opération sans intervenir manuellement sur les fichiers *Makefile.*

c) Chargement des *drivers*

Lorsque les *drivers* sont sous forme autonome, il faut les lancer manuellement via la commande *modprobe* qui va charger en mémoire le code lié au *driver* et gérer les dépendances (chargement de *drivers* complémentaires).

V.2 Test des convertisseurs

V.2.1 Ajout de la description physique des convertisseurs

La carte Beaglebone est décrite physiquement via le fichier *am335x-bone.dts*. Celui-ci se situe dans le répertoire des sources de Linux */sec/buildroot/output/build/linux-tse/* dans le sous-répertoire spécifique à l'architecture ARM : *./arch/arm/boot/dts*. Vous pouvez visualiser ce fichier sans le modifier.

Il contient principalement un fichier inclus nommé *am33xx.dtsi* qui décrit la plupart des périphériques.
Vous pouvez visualiser ce fichier et y retrouver les différentes descriptions.
Vous allez ajouter au fichier *am335x-bone.dts* un nouveau fichier *am335-sec.dtsi*. Ce fichier doit être en dernière position des fichiers inclus, pour modifier les définitions de base déjà faites dans les autres fichiers inclus.

Créer le fichier *am335x-sec.dtsi* dans le répertoire adéquat : *gedit am335x-sec.dtsi &.*
Voici le code à copier dans le fichier :

```
&tscadc {
	status = "okay";
	am335x_adc: adc {
		ti,adc-channels = <0>,<1>,<2>,<3>,<4>,<5>,<6>,<7>;
	};
};
```

Dans le fichier am33xx.dtsi, on peut voir (ligne 814), que le nœud *tscadc* (adresse 44e0d000) comprend 2 sous-noeuds : *tsc*, pour l'écran tactile et *am335x_adc,* pour le CAN.

Dans notre fichier *am335x-sec.dtsi*, on modifie le nœud *tscadc,* sans préciser son arborescence complète, grâce au symbole &. On active le CAN en :

- modifiant la valeur de la clef *status* de *disabled* à *okay ;*
- choisissant les voies actives : *<0>,<1>,<2>,<3>,<4>,<5>,<6>,<7>*

Sauvegarder le fichier.
Ensuite, il est normalement nécessaire de traduire les fichiers dts sous forme de fichier dtb (*am335x-bone.dtb* pour notre cas), ce fichier étant chargé après le noyau lors du boot. La traduction des fichiers se fait manuellement avec l'utilitaire « *dtc* » mais comme nous allons modifier le noyau, nous n'aurons pas à le faire car le *make* global inclus aussi cette traduction. Lors du lancement de *make*, il faudra bien vérifier que la traduction du *device tree* se passe correctement en regardant les messages à l'écran.

V.2.2 Compilation des *drivers*

La deuxième étape va consister à construire le code exécutable du *driver*. Nous choisirons de le faire sous la forme de modules indépendants (modules dynamiques), ce qui nous permettra de mieux maîtriser le démarrage des *drivers*. Cette phase va être mise en œuvre grâce aux menus de configuration du noyau Linux.

Make linux-menuconfig
menu : *Device Drivers*
sous-menu : *Multifunction device drivers*
activer en mode dynamique (<M>) l'item *TI ADC / Touch Screen chip support* , en tapant le nombre de fois requis avec la barre espace.
Cet item concerne le *driver* générique qui va assurer le multiplexage des convertisseurs entre l'écran tactile et l'usage direct en tant que convertisseur.
remonter au menu *Device Drivers* : un appui sur *Exit*
activer l'item *Industrial I/O support* en mode dynamique (<M>)

(nouvellement créé) : *Industrial I/O support*
sous-sous-menu : *Analog to digital convertersdcd*
activer l'item *TI's AM335X ADC driver,* en mode dynamique <M>.

Ce dernier item concerne le *driver* gérant les convertisseurs en mode acquisition de données. Enfin ressortir des menus en sauvant votre nouvelle configuration.

Save, Ok, Exit

Reconstruire le *device-tree*, le noyau et le système de fichiers :

make device-tree
make kernel
make filesystem

Pour vérifier que les modules ont bien été compilé vous pouvez aller dans le répertoire où se situe le code source, le code compilé (*.o) et le module (*.ko) :

cd /sec/buildroot/output/build/linux-tse/drivers/iio/adc

Pour la carte Beaglebone, le fichier source est *ti_am335x_adc.c*. Vous pouvez visualiser son contenu et vérifier que des fichiers de même nom et d'extensions différentes (*.o, *.ko...) ont été créés dans le même répertoire.
Faire de même, pour le *driver* générique de multiplexage *ti_am335x_tscadc* sous le répertoire :

/sec/buildroot/output/build/linux-tse/drivers/mfd

L'exécution de *make* a donc créé les modules dynamiques du noyau, nommés *.ko dans les mêmes répertoires que les fichiers sources. De plus, *make* va aussi installer ces modules dans une copie du répertoire de l'arborescence du système de fichiers */lib/modules/3.x.x/kernel/drivers/* relative aux modules dynamique du noyau dans les sous-répertoires correspondant à l'arborescence des sources, c'est à dire *./iio/adc/* pour le *driver* du convertisseur et *./mfd* pour le *driver* générique.
Cette copie sera dans l'arborescence contenue dans les fichiers *rootfs.ext2* (pour écriture dans la 2ème partition de la flash) et *rootfs.tar* (pour décompactage vers un répertoire partagé en NFS). On peut aussi la retrouver dans l'arborescence de la cible */sec/buildroot/output/target/lib/modules/3.12.10/kernels/...*

Vérifier qu'ils sont bien dans cette dernière arborescence.

Vous pouvez visualiser sa présence dans le fichier .tar grâce à la commande :

tar tvf rootfs.tar | grep -e ".ko"

Un des intérêts des modules dynamiques est de ne pas augmenter la taille du noyau *zImage*, l'exécutable se situant dans le système de fichiers.

V.2.3 Démarrage de la carte Beaglebone avec le nouveau noyau

Se positionner sous le répertoire */sec/buildroot/output/images* puis extraire le fichier *rootfs.tar* vers */targetfs* par la commande :

tar xvf rootfs.tar -C /targetfs

Copier aussi les fichiers modifiés :

cp zImage /targetfs/boot
cp am335x-bone.dtb /targetfs/boot

Puis redémarrer la carte Beaglebone, via NFS, après avoir chargé les variables d'environnement.

V.2.4 Chargement du *driver*

Nous allons maintenant voir comment charger les *drivers* ainsi que les modifications dans le système de fichiers */sys* qu'ils vont effectués.

Aller sous le répertoire */sys/bus/platform/devices/*. Visualiser son contenu.

Existe-t-il un répertoire correspondant aux convertisseurs (nom contenant tscadc) ? Comment est construit son nom ? Aller dans ce sous-répertoire et visualiser son contenu.

Lancer le chargement dynamique du *driver* des convertisseurs par la commande :

modprobe ti_am335x_tscadc

modprobe cherche automatiquement dans tous les sous-répertoires de */lib/modules/3.x.x* pour trouver un fichier de ce nom avec l'extension .ko et exécute l'installation du *driver*.

Vérifier le bon chargement du *driver* par la commande : *lsmod*.

Visualiser à nouveau le contenu du répertoire . Quels éléments ont été rajoutés ? Aller dans le sous-répertoire concernant le convertisseur (nom contenant *adc*). Visualiser le.

Faire la même opération (*modprobe*) pour le *driver* du convertisseur (*ti_am335x_adc*).

Vérifier le bon chargement du *driver* et les dépendances par la commande *lsmod*.

Regarder le contenu du répertoire. Puis aller dans le sous répertoire *iio:device0*, ce qui correspond à l'arborescence :

/sys/bus/platform/devices/44e0d000.tscadc/TI-am335x-adc/iio:device0

Lire le contenu des convertisseurs *in_voltage*_raw* par la commande *more*. Normalement, c'est le **numéro 1** qui correspond au convertisseur, mais cela peut dépendre du câblage de la carte d'extension.

Vérifier le bon fonctionnement du potentiomètre en prenant des mesures pour plusieurs positions de rotation.
Quelle opération faut-il effectuer pour retrouver la tension analogique (V) correspondant au nombre obtenu ? Vérifier votre résultats avec un voltmètre.

V.3 Utilisation dans des applications C

Créer un répertoire /home/tpuser/se5 pour placer les fichiers sources, objets et exécutables ainsi que le *Makefile* lors du développement.
La compilation sera contrôlée par un *Makefile*. Une copie de l'exécutable vers le répertoire /targetfs/usr/local/bin sera aussi effectuée dans le *Makefile* pour un accès simple côté carte Beaglebone. Si le répertoire n'existe pas, créer le et ajouter le au *PATH* de la carte Beaglebone :

export PATH=$PATH:/usr/local/bin

V.3.1 Programme d'affichage

Faire un programme qui affiche la valeur renvoyée par le convertisseur toutes les secondes. Il affichera sur une ligne la valeur numérique lue et la valeur convertie en Volt.

L'interface avec le convertisseur étant de type texte, on utilisera les fonctions d'entrée/sortie formatées standard du langage C : *printf, scanf, fopen, fprintf, fscanf, fclose...*

Voici le canevas du code C, à compléter :

```
#include <stdio.h>
#include <unistd.h>
#define AIN1 « chemin du pseudo-fichier du convertisseur AIN1 »

int main(void)
{
      FILE *ain1_desc ;
      char str[8] ;
      float tension = 0 ;

      while(1)
      {
//on accède au pseudo-fichier en lecture
            ain1_desc = fopen (AIN1, "r");
            fscanf (ain1_desc,"%s",str) ;
            tension = atoi(str)* ???
            printf("Valeur = ??? \n",???) ;
            printf("Tension = ??? \n",???) ;
            sleep(1)
      }
      fclose (ain1_desc) ;

      return 0 ;
 }
```

Compiler (options de compilation : -g -ansi -std=c99 -o) et tester votre programme

Compléter l'affichage par un pseudo « *baregraph* » sur la ligne suivante. La longueur du pseudo barre-graphe sera proportionnelle à la valeur lue.
Exemple : Tension : 1,2 V ********

V.3.2 Programme d'affichage via un serveur WEB

Ecrire un programme en C (*convert1.c*) qui va lire la valeur de la tension sur le convertisseur et qui la renvoie vers la sortie standard.

Créer une page WEB avec un script CGI (*get_value.cgi*) qui exécute le programme précédent.

Mettre en route le serveur Boa sur la carte Beaglebone et suivre l'évolution de la tension sur le convertisseur via le navigateur de la machine virtuelle.

Si un autre binôme en est au même point, vous pourrez aussi visualiser la valeur de la tension sur son convertisseur et modifiant votre page WEB.

V.3.3) Mesure de bruit

Lancer une boucle de mesure infinie dans laquelle vous affichez la valeur lue ainsi que la moyenne et l'écart type de toutes les valeurs lues depuis le lancement
Vous pouvez de même qu'au 3.2 envoyer ces résultats sur une page WEB via un fichier tampon.
Quels problèmes doit-on résoudre pour que cela fonctionne correctement.

Séance n°6 : Mise en œuvre d'un télémètre ultrason sur le bus I2C

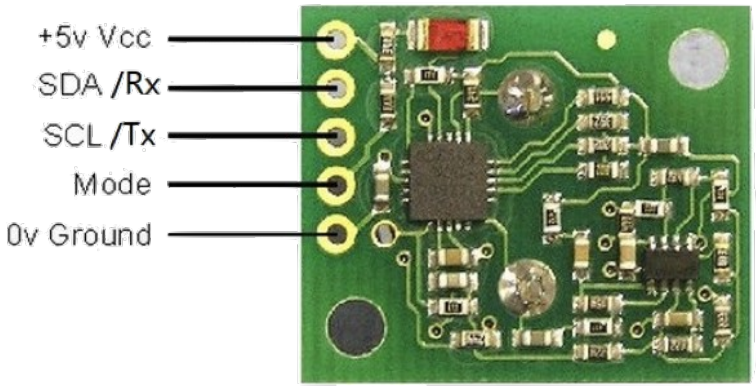
Compétences visées :

- modification du *device-tree* Linux pour ajouter un périphérique (contrôleur I2C)
- utilisation d'une liaison I2C pour communiquer avec un capteur, type SRF02
- configuration et utilisation d'un capteur ultrasonore, type SRF02
- utilisation d'un server WEB pour afficher les résultats de mesure

VI.1 Présentation

VI.1.1 Télémètre à ultrason SRF02

Le télémètre à ultrason SRF02 est capable de déterminer la distance qui le sépare d'un obstacle se présentant devant lui (entre 15 cm et théoriquement 6 m). Doté d'une seule cellule ultrason, son principe repose sur celui des « sonars ». Il détermine la valeur de la distance en « mm », en « inch » ou sous forme d'une durée (en µs) en mesurant le temps d'aller-retour de l'écho d'un émetteur ultrason.
Il peut s'interfacer à l'aide d'un bus I2C (cas du TP) ou bien alors d'une liaison série classique (RS232).



VII.1.2 Bus I2C

I²C pour *Inter-Integrated Circuit* est un protocole simple de communication, mis an point dans les années 80 et dont la norme évolue toujours (dernière version en 2021). Le bus I2C est constitué de seulement trois fils : un signal de données (*SDA*), un signal d'horloge (*SCL*), et un fil de masse. Il s'agit d'une liaison de type :

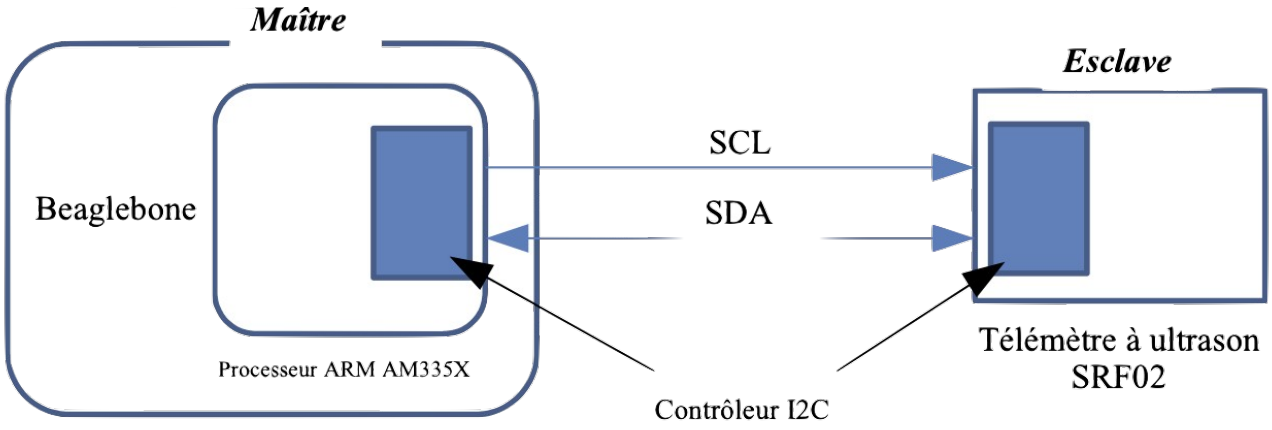
- maître/esclave, avec identification par adresse ;
- série ;
- synchrone : l'horloge est partagée entre le maître et l'esclave ;
- bidirectionnel : le maître peut envoyer des données à l'esclave et inversement ;
- half-duplex : il n'y a qu'une ligne de données (SDA) donc elle ne peut être utilisée en même temps par le maître et l'esclave.

Quels sont les débits binaires possibles sur un bus I2C ?
L'I2C est un bus dit « informatique » et non de « terrain » (ethernet, CAN, Lin…), car il ne peut être utilisé que sur de faible distance (par exemple 1 à 10m). Pourquoi ?

L'ensemble des caractéristiques détaillées ci-dessus font que le bus I2C est très utilisé dans les systèmes embarqués, pour différents types de communication : processeur/mémoire, micro-contrôleur/capteur ou actionneur...

VII.1.3 Mise en oeuvre

Dans le cadre de ce TP, la carte BeagleBone est le **maître** et va piloter un seul **esclave** le télémètre à ultrason SFR02. Sur la carte Beaglebone, c'est le microcontrôleur ARM am335x qui intègre **un contrôleur I2C** permettant ainsi de gérer le protocole. De l'autre côté, le télémètre SRF02 intègre aussi un contrôleur I2C.



Le contrôle I2C du SRF02 a l'**adresse 0x70** = 0b111000. Cette adresse sur 7 bits sera complétée par un 0 (0b11100000=0xE0) en cas d'écriture (transfert de données du maître vers l'esclave) ou par un 1 (0b11100001=0xE1), en cas de lecture. Ce complément sera ici géré l'API C, en fonction de l'utilisation des fonctions *read* ou *write*, mais ce n'est pas le cas pour tous les systèmes embarqués !

Ce contrôleur comporte **6 registres** dont les adresses vont de **0x00 à 0x05 :**

- registre 0, adresse 0x00 :
  - en lecture, on obtient le numéro de révision du logiciel du capteur ;
  - en écriture, on configure le capteur : lancement de la mesure, envoi d'une onde ultrasonore, changement de l'adresse I2C…

Quelle valeur (hexadécimale) faut-il écrire pour lancer la mesure d'une distance, exprimée en cm ?
Même question pour obtenir une durée d'aller-retour à l'obstacle, exprimée en microsecondes.

- registre 1, adresse 0x01 : accessible seulement en lecture et doit renvoyé la valeur 0x80 ;

- registres 2 et 3, adresses 0x02 et 0x03 : *MSB* et *LSB* de la donnée.

Sachant que la donnée est non signée, codée sur 16 bits, comment obtient-on cette donnée à partir du *MSB* et du *LSB* .

- registres 4 et 5, adresses 0x04 et 0x05 : associés au mécanisme d'*auto-tune* et à la distance minimale mesurable.

Une fois l'onde ultrasonore envoyée, l'émetteur continue à vibrer durant un temps Δt dépendant des conditions expérimentales (température, pression…). Le capteur SRF02 est donc muni d'un algorithme dit *AutoTune* permettant une mesure précise de ce temps Δt et enregistrant donc dans les registres 4 et 5, la distance minimale détectable : $d_{min} > v*\Delta t/2$.

VII.1.4) Exemple de communication

En langage C, l'accès au périphérique I2C se fait principalement par des lectures/écritures dans un pseudo-fichier (ici */dev/i2c-2*). On utilisera les fonctions :

- *open* : *descripteur de fichier = open (« nom du pseud-fichier », mode d'ouverture)*
- *ioctl* : ioctl *(descripteur de fichier, commande, valeur)*
- *write* : *write (descripteur de fichier, caractères, nombre de caractères)*
- *read* : *read (descripteur de fichier, caractères, nombre de caractères)*
- *close* : *close (descripteur de fichier)*

Une communication sur le bus I2C correspond à 3 phases (cf. figure ci-dessous) :

1. sélection de l'esclave et sens d'échange (lecture ou écriture) ;
2. choix de l'adresse du registre du contrôleur esclave
3. échange des données.
   Lors de l'échange de plusieurs octets de données, l'adresse du registre est automatiquement incrémentée.

Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

Figure 2: Trames I2C

On souhaite dans un premier temps lire la donnée de la distance (en cm). Voici les étapes à suivre :

1. prendre la main sur le contrôleur I2C de la carte, en mode lecture ou écriture (*O_RDWR*). On communiquera par la suite avec ce contrôleur à l'aide du pseudo fichier créé (*i2cFile*) :
   ```
   i2cFile = open("/dev/i2c-c" ,O_RDWR);
   ```

2. ouvrir la communication avec le capteur (adresse 0x70), en configurant ce dernier en tant qu'esclave :
   ```
   ioctl(i2cFile, I2C_SLAVE_FORCE,0x70);
   ```

3. configurer le capteur pour qu'il renvoie une donnée en cm.
   Quelle valeur DATA faut-il écrire dans quel registre RA du capteur ?

   ```
   I2C_WR_Buf[0] = RA;
   I2C_WR_Buf[1] = DATA;
   write(i2cFile, I2C_WR_Buf, 2);
   ```

4. indiquer au capteur que l'on souhaite recevoir 2 octets : *MSB* et *LSB*.
   Quelle est l'adresse RA du registre contenant le MSB ?

   ```
   I2C_WR_Buf[0] = RA;
   write(i2cFile, I2C_WR_Buf,1);
   read(i2cFile, I2C_RD_Buf,2) ;   //lecture de 2 octets
   ```

VII.2) Utilisation dans un programme en C

VII.2.1) Activation du bus I2C sur la carte Beaglebone.

D'une manière comparable au Convertisseur Analogique-Numérique, le bus I2C que nous allons utiliser n'est pas accessible directement sur la carte Beaglebone. Nous allons donc l'activer. Une opération supplémentaire est nécessaire, en effet le **bus i2C-2** que nous utilisons est situé sur une broche multiplexée et nous devons donc choisir le bon mode. Ces opérations vont être réalisées via le *device tree*.

Pour ne pas interférer avec la définition par défaut livrée avec le BSP, nous allons utiliser le fichier inclus *am335x-sec.dtsi* dans le fichier *am335x-bone.dts* comme à la séance précédente. Nous allons ajouter dans ce fichier les directives suivantes (attention la syntaxe est différente de celle de la séance précédente : les 2 sont valables) :

```
/ {
     am33xx_pinmux: pinmux@44e10800 {
          i2c2_pins: pinmux_i2c2_pins {
               pinctrl-single,pins = <0x178 0x73 0x17c 0x73 > ;
          } ;
     } ;
     ocp {
          i2c2: i2c@4819c000 {
          pinctrl-names = "default" ;
          pinctrl-0 = <&i2c2_pins> ;
          status = "okay" ;
          } ;
     } ;
};
```

`0x178 0x73` **:** utilisation pour SDA la broche 20 du GPIO, avec résistance de PULL-UP
`0x17c 0x73` : utilisation pour SCL la broche 19 du GPIO, avec résistance de PULL-UP

Il faut ensuite refaire un make du *device tree* (*make device-tree*) et recopier le fichier généré dans le répertoire */targetfs/boot* pour qu'il puisse être utilisé lors du prochain boot en NFS.

VII.2.2) Lecture de la distance en cm.

Vous devez dans un premier temps écrire le programme qui permet de récupérer la distance en cm :

```
#include <Linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define I2C_PERIPH "/dev/i2c-2"          //Peripheric name in /dev
#define ADDR_SRF02 ???                   //Peripheric address
#define MAX_BUFF_SIZE 64
```

```
int main()
{
  int i2cFile;
  int flag, value;
  unsigned char I2C_WR_Buf[MAX_BUFF_SIZE];   // Contains sent data
  unsigned char I2C_RD_Buf[MAX_BUFF_SIZE];   // Contains read data

  i2cFile = open(I2C_PERIPH, O_RDWR);
  if (i2cFile < 0)
  {
     perror("i2cOpen error"); //prints out error messages
     exit(1);  //unsuccessful terminaison
  }

  flag=ioctl(i2cFile, I2C_SLAVE_FORCE, ADDR_SRF02) ;
  if ( flag < 0)
  {
     perror("i2cSetAddress error:");
     exit(1);
  }

  I2C_WR_Buf[0] = ???;
  I2C_WR_Buf[1] = ???;
  flag =  write(i2cFile, I2C_WR_Buf, 2);
  if( flag!= 2)
  {
     perror("Error in First Send_I2C");
     exit(1);
  }

  sleep(1);

  I2C_WR_Buf[0] = ???;
  flag =  write(i2cFile, I2C_WR_Buf, 1);
  if( flag!= 1)
  {
     perror("Error in Second Send_I2C");
     exit(1);
  }

  flag = read(i2cFile, I2C_RD_Buf, 2);
  if( flag!=2)
  {
     perror("Read Error I2C_Byte");
}

printf(" MSByte= %x LSByte=%x
",???, ???);

close(i2cFile) ;

return 0 ;
}
```

Remplacer l'ensemble des points d'interrogation (???) par les valeurs adéquates.
Compléter ce programme en calculant et affichant la distance en cm.

Compiler et tester votre programme sur la carte BeagleBone, après avoir branché correctement le capteur.

VII.2.3) Fonctions supplémentaires

1. Lecture de la durée d'aller-retour de l 'écho.

Ajouter à votre programme, la fonctionnalité de récupération de la durée (en µs) d'un d'aller-retour. Afficher la valeur. Calculer ensuite la vitesse du son dans l'air et afficher cette dernière.
La valeur vous paraît-elle cohérente ?

Pour plus de précision, modifier votre programme pour effectuer une moyenne sur 10 valeurs de vitesse.

2. Estimation de la température de la pièce

En assimilant l'air à un gaz parfait diatomique, il est possible d'approcher la célérité du son par la formule suivante : c_{air} (m/s) = 331,5+0,6*Θ, où Θ est la température en degrés Celsius.

En utilisant cette relation, modifier votre programme pour afficher la température de la pièce. Commentaire ?

3. Vérification de la distance minimale mesurable

En utilisant les registres appropriés, lire et afficher la distance minimale mesurable par le capteur SRF02. Modifier votre programme pour afficher un avertissement dans la console, si c'est distance est atteinte.

4. Observation de trames I2C

Observer quelques trames I2C à l'oscilloscope numérique et retrouver les éléments attendu : signal START, signal STOP, adresse de l'esclave, adresse du registre, donnée…

5. Lecture de la valeur via une interface WEB

Utiliser le serveur Boa pour afficher les différentes valeurs obtenues sur une interface WEB de l'hôte.

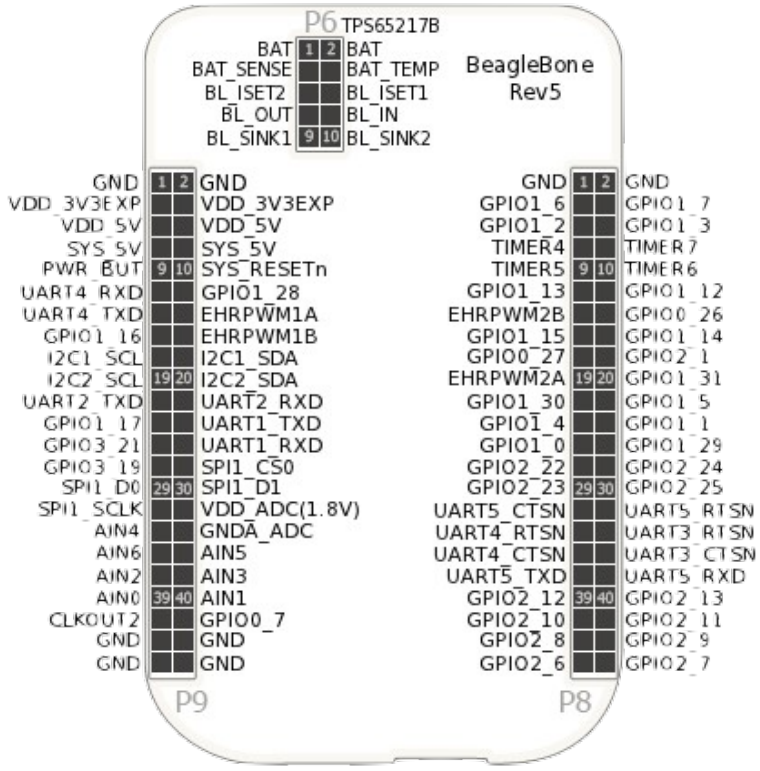


Figure 3: Pin map BeagleBone

Séance n°7 : Utilisation d'une liaison série RS232

Compétences visées :

- modification du *device-tree* Linux pour ajouter un périphérique (interface série UART) ;
- utilisation de l'interface série UART via les commandes du *shell ;*
- utilisation de l'interface série UART dans un programme en C

VII.1 Présentation

VII.1.1 Principe d'accès

L e processeur ARM de la carte Beaglebone possède en internet **6 interfaces séries de type RS232** (UART0 à UART5). Elles sont accessibles comme des périphériques sous le répertoire */dev : /dev/ttyO0* à */dev/ttyO5*. Dans le cas de la carte Beaglebone, l'UART0 est utilisé par le système comme **port console**. C'est celui auquel vous accéder via *gtkterm* et une interface de type série sur USB. La conversion série-USB est faite par un circuit spécifique (FTDI) sur la carte Beaglebone, la conversion inverse étant réalisée par un *driver* sous le système hôte Debian.

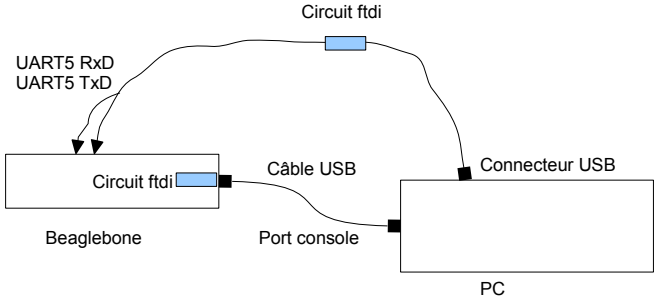
Contrairement à d'autres bus de communication (I2C, SPI...), une liaison série est une liaison **asynchrone** équilibrée, **sans maître et sans esclave**, elle nécessite donc 3 fils :

- TxD (Transmission) ;
- RxD (Réception) ;
- masse.

La communication étant asynchrone, elle est limitée en nombre de bits car il faut pouvoir resynchroniser fréquemment les horloges d'émission et de réception. La taille des données échangées est habituellement de 8 bits. De plus la fréquence des horloges est réglable, il faut donc avoir configuré **la même fréquence des 2 côtés de la transmission**.

La norme RS232 prévoit des tensions de fonctionnement de **+12V à +5V pour un niveau bas (0)**, et de **-12V à -5V pour un niveau haut (1)**. Ces tensions sont incompatibles avec une liaison directe sur des ports du processeur ARM qui ont des tensions de fonctions de **0 à 3,3V**. Il faut normalement prévoir des circuits d'adaptation de niveaux si l'on veut utiliser les ports séries du processeur ARM avec le port série d'un PC par exemple. Pour ce TP, nous utiliserons des **câbles spéciaux** qui opèrent une conversion série-USB comme pour le port console.
Pour le câble spécial, côté RS232 : le fil **vert** est le TxD, le **blanc** le RxD et le **noir** la masse (ou TxD **orange** ; RxD **jaune**). Le fil **rouge** (qui est le +5V) **ne doit pas être utilisé !**
Côté Beaglebone, si l'on utilise l'**UART 5** le TxD est sur la broche **P8-37** et le RxD sur la broche **P8-38**, la masse est en **P8-1 (cf. figure 3).**

Pour un bon fonctionnement il faut relier le TxD du câble au RxD de la Beaglebone et réciproquement… il faut donc croiser les fils.



VII.1.2 Opérations nécessaires pour l'accès à un nouveau périphérique de type UART

Comme pour les autres périphériques, l'utilisation d'une liaison série supplémentaire va nécessiter plusieurs étapes :

1. ajout de la description physique du périphérique supplémentaire dans le *device tree* ;
2. compilation et chargement des *drivers* : cette opération étant déjà faite, car du fait du port console, la gestion des liaisons série est intégrée au noyau de manière statique ;
3. branchement des ports séries via un câblage adapté.

VII.2 Test de la liaison série

VII.2.1 Ajout de la description physique des uarts

Nous allons rajouter dans le fichier de la séance précédente (am335-sec.dtsi) la définition de l'interface série UART5.

```
/ {
	am33xx_pinmux: pinmux@44e10800 {
		pinctrl-names = "default";

		i2c2_pins: pinmux_i2c2_pins {
			pinctrl-single,pins = <0x178 0x73 0x17c 0x73>;
		};

		uart5_pins: pinmux_uart5_pins {
			pinctrl-single,pins = <
				0x0C4 (PIN_INPUT_PULLUP | MUX_MODE4)	/*uart
Rx*/
				0x0C0 (PIN_OUTPUT_PULLDOWN | MUX_MODE4) /*uart
Tx*/
			>;
		};
	};

	ocp {
		i2c2: i2c@4819c000 {
			pinctrl-names = "default";
			pinctrl-0 = <&i2c2_pins>;
			status = "okay";
		};

		uart5: serial@481aa000 {
			pinctrl-names = "default";
			pinctrl-0 = <&uart5_pins>;
			status = "okay";
		};
	};
};
```

Sauvegarder le fichier.
Ensuite, il est nécessaire de traduire les fichiers dts et dtsi complets sous forme de fichier dtb (*am335x-bone.dtb* dans notre cas). Comme nous n'avons fait aucune modification sur le noyau, il est juste nécessaire de reconstruire le *device tree* en lançant la commande *make* device-tree sous le répertoire */sec/buildroot.*

Vous devez voir apparaître le lancement de la commande *dtc* et vérifier que la traduction s'est effectuée sans erreur.

VII.2.2 Démarrage de la carte Beaglebone avec le nouveau noyau

Se positionner sous le répertoire */sec/buildroot/output/images* puis copier le fichier modifié en étant super-utilisateur :

cp am335x-bone.dtb /targetfs/boot

Redémarrer la carte Beaglebone, via nfs après avoir changé les variables d'environnement.

VII.2.3 Test de la liaison en mode console

a) Mise en œuvre

Sous Linux, les ports série étant associés à des **périphériques de type caractères**, il est possible d'utiliser les commandes standard pour envoyer ou recevoir des caractères. Le port série UART5 est connecté via le *driver* au pseudo-fichier */dev/ttyO5* (c'est un O devant le 5 et non un 0 !).

1. Brancher le câble spécial sur un des ports USB du PC ;
2. Ouvrir un nouveau terminal sur le PC, avec PuTTY sous Windows (un port COM a été ajouté au moment de la connexion) ;
3. Régler le débit binaire (*baudrate*) à 9600 bits/sec ;
4. Régler les autres paramètres à 8N1 : paquet de 8 bits, pas de bit de parité (None) et 1 bit de stop ;
5. Côté carte BeagleBone, configurer le port par la commande : *stty -F /dev/ttyO5 9600*

b) Test d'envoi vers l'UART5

En utilisant la commande *cat*, envoyer des caractères de la console Beaglebone vers le pc :

cat >/dev/ttyO5

Taper des caractères sur la console. Apparaissent-ils sur le deuxième terminal ? Si non quel caractère faut-il envoyer pour vider le buffer d'entrée ?

Cette commande permet de rediriger la sortie standard non pas vers votre fenêtre console, mais vers la liaison série ttyO5. C'est pour cela que vous ne voyez pas les caractères rentrés sous la première console. Vous pouvez sortir du mode d'entrée par l'appui des touches : *CTRL + C*.

Vous pouvez faire la même opération en utilisant la commande : *echo texte >/dev/ttyO5*

c) Test de réception

Avec les mêmes commandes que précédemment (mais d'une manière inverse), nous pouvons recevoir des caractères de la liaison série sur la console par la commande : *cat < /dev/ttyO5*

d) Echange de console

En redirigeant l'entrée et la sortie standard vous changer de fenêtre console :

sh </dev/ttyO5 >/dev/ttyO5

Il aurait été possible de dédoubler les fenêtres console (puisque Linux est multitâche) en lançant la commande précédente en tâche de fond (avec un & à la fin)…

VII.3 Utilisation dans des applications C

On créera un répertoire */home/tpuser/se7* pour placer les fichiers sources, objets et exécutables lors du développement. Une copie vers le répertoire */targetfs/usr/bin* sera aussi effectuée dans le *Makefile* pour un accès simple côté carte Beaglebone.

VII.3.1 Programme d'échange

Faire un programme qui envoie les caractères tapés dans la console vers le deuxième terminal sous Windows.

```
#include <stdio.h>

int main ()
{
      FILE *uart;
      char str[11] ;

      printf("Entrer un texte de 10 caracteres :\n");
      scanf("%s",str);

      uart= fopen("/dev/ttyO5","w");
      fprintf (uart,"texte envoyé :%s\n",str) ;
      fclose(uart);
}
```

VII.3.2 Programme d'échange inverse

Faire un programme qui récupère les caractères tapés sur le deuxième terminal et les affiche sur le premier.

Est-il possible facilement de faire cohabiter les 2 programmes ? Pourquoi ?

VII.3.3) Utilisation de la liaison série pour commander une action à distance

En reprenant le programme de commande des leds, faire allumer la led par l'entrée de la commande *ON* sur le deuxième terminal. De même, l'éteindre par l'envoi de la commande *OFF*.

Fonctions utiles : *fscanf, strcmp...*

VII.3.4) Utilisation de la liaison série pour récupérer des données

Reprendre le programme de mesure de la température, par le capteur SRF02 et afficher le résultat de la mesure sur l'ordinateur. Une application embarquée simple mais complète aura ainsi été développée : mesure, calcul et communication.

Si vous lancer votre programme en tâche de fond, pouvez-vous utiliser la première console simultanément ?

Séance n°8 : Utilisation de la bibliothèque SDL pour un écran tactile

Compétences visées :

- modification du noyau Linux et du *device tree* pour gérer l'écran tactile ;
- utilisation de la bibliothèque SDL pour l'affichage d'image ou de texte ;
- utilisation de la bibliothèque TSLIB pour l'utilisation de l'écran tactile ;
- développement d'une application complète sous Linux embarqué ;

VIII.1 Présentation

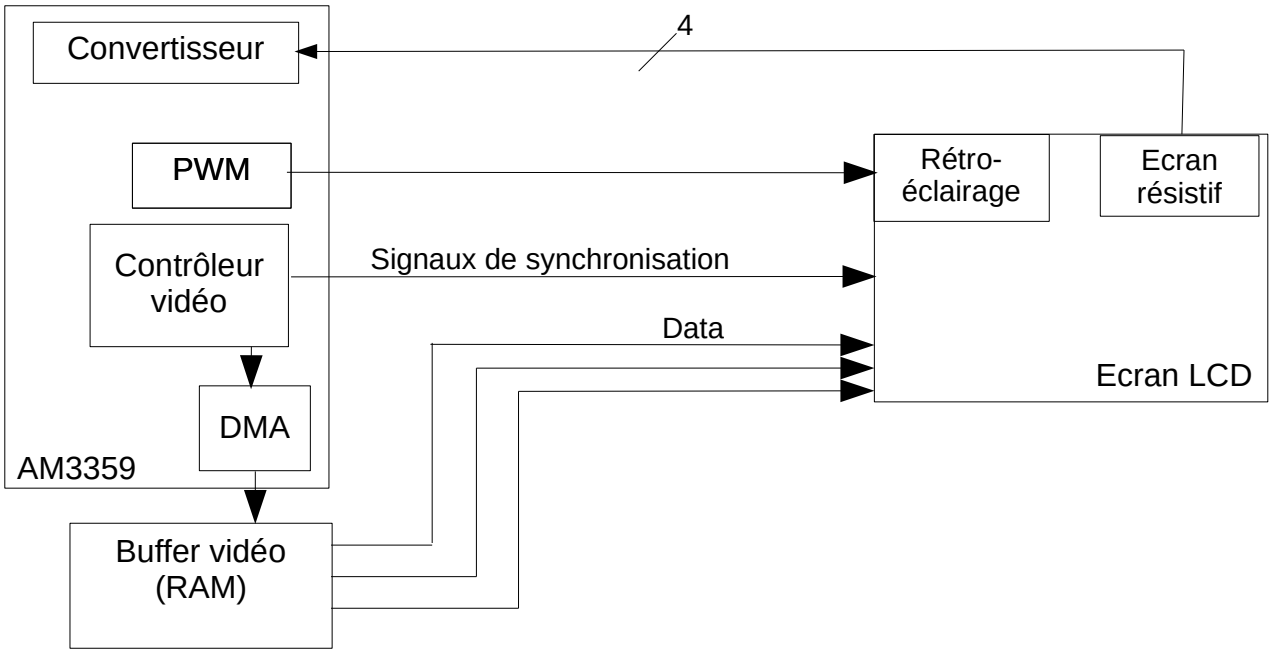
VIII.1.1 Ecran tactile

Dans un certain nombre d'applications des systèmes embarqués, un écran tactile peut être utilisé pour permettre des entrées-sorties utilisateur. Plusieurs écrans sont associés à la carte BeagleBone (3 pouces, 4 pouces et 7 pouces). Ils utilisent tous le même principe de connexion :

- 16 bits de données (3 couleurs sur 8 bits)
- 5 fils de contrôle et de synchronisation (LCD_VSYNC, LCD_HSYNC, LCD_PCLK..)
- 1 fil pour le réglage de la luminosité (EHRPWM1A)
- 4 entrées analogiques (AIN0 à AIN3) pour l'écran tactile résistif (*touchscreen*)

Tous ces signaux se retrouvent sur les connecteurs extérieurs de la carte BeagleBone et sont à l'usage exclusif de l'écran. L'écran utilise aussi le port i2c2 à des fins de détection automatique (comme toutes les cartes d'extension *capes*, l'écran possède un petite mémoire flash avec un code permettant de reconnaître le périphérique associé).

Côté carte BeagleBone, un contrôleur vidéo est intégré dans le microcontrôleur am335x, il va permettre de générer les différents signaux, il est associé à une zone mémoire réservée (*buffer* vidéo) pour stocker l'image courante envoyée vers l'écran (un seul changement de pointeur permet de modifier l'image sur l'écran : *flip*). Les données de l'image sont envoyées en permanence sur l'écran via un contrôleur *Direct Memory Access* (DMA), car l'écran n'a pas de mémoire locale.
Le rétro-éclairage est géré en modulant par impulsions un signal logique. Quatre entrées multiplexées (AIN0 à AIN3) du convertisseur internet du processeur sont réservées pour lire l'écran résistif et déterminer la position du *touch* par triangulation (avec 4 mesures !).



VIII.1.2 *Device Tree*

L'écran utilisant de nombreux signaux, les modifications du *device tree* sont donc importantes. Elles concernent :

- le contrôleur vidéo
- le frame *buffer* (zone de mémoire contiguë permettant un accès DMA)
- la réservation des signaux vidéo
- l'écran
- le contrôle du rétro-éclairage
- la lecture du *touchscreen*

Le fichier am335x_lcd4.dtsi décrivant le *device tree* adapté à l'écran 4 pouces se trouve dans le répertoire /sec/examples. Lors de l'insertion de ce fichier dans le fichier am335-bone.dts, il faut vérifier que les nouvelles déclarations ne viennent pas en contradiction avec vos précédents fichiers inclus.

Retrouvez chaque élément précédent dans ce fichier : à chaque élément est associé un *driver* (programme exécuté au niveau du noyau). Ce *driver* est identifié par le champ « compatible ».

VIII.1.3 Bibliothèque SDL.

La bibliothèque SDL (*Simple DirectMedia Layer*) fournit des fonctions graphiques (et multimédia) adaptées aux jeux, car elle permet un accès efficace au matériel et donc une fluidité dans la restitution. Elle est aussi très utilisée dans les systèmes embarqués dans lesquels il est nécessaire d'avoir une interface graphique sans trop utiliser de ressources.
Elle est constituée de plusieurs modules :

- gestion matériel de base
- lecture et affichage de fichiers image
- affichage de textes avec une police paramétrable
- fonctions liées à la restitution de sons et au transfert par le réseau (fonctions que nous n'utiliserons pas ici)

VIII.2 Modification du noyau et du système de fichiers

VIII.2.1 Ajout du support SDL dans le système de fichiers

La bibliothèque SDL va fournir au développeur un ensemble de fonctions aptes à gérer l'affichage sur l'écran : ces fonctions sont appelées directement par le programme utilisateur : elles ne sont donc pas liées au noyau mais juste présentes dans le système de fichiers via des **bibliothèques statiques** (ajoutées au moment du chaînage au programme exécutable) ou **dynamiques** (ajoutées en cours d'exécution).
Leur utilisation dans un programme C nécessite aussi **d'inclure les fichiers de définition (*.h).**

Pour modifier le système de fichiers nous allons donc lancer l'utilitaire *make menuconfig*

menu *Target Packages*
sous-menu *Graphics library*
SDL : activer, par appui sur la barre d'espace
SDL_image : activer
sous-sous-menu *SDL_image file format support*
enable JPEG file format support : activer
enable PNG file format support : activer
SDL TTF : activer

Nous ne relancerons pas directement un *make* car nous allons aussi modifier le noyau pour y ajouter les *drivers* nécessaires.

VIII.2.2 Ajout des *drivers* de l'écran tactile

La modification du noyau se fait via la commande : *make Linux-menuconfig*
Nous avons 3 éléments principaux à ajouter : l'écran et sa mémoire graphique associée, le rétro-éclairage et le *touchpad* (Tous ces éléments seront activés en modules statiques <*>).

Concernant le rétro-éclairage :
menu Device Drivers
- sous-menu *Pulse-Width Modulation Support*
  - *EHRPWM PWM Support* : activer
- sous-menu *Graphics Support*
  - *Backlight and LCD device support* : activer
    - *Low Level LCD Controls* : activer
    - *Low Level Backlight Controls* : activer
    - *Generic PWM Based Backlight* Driver : activer

Concernant le contrôleur graphique, le frame *buffer* et l'écran :
menu *Device Drivers*
- sous-menu *Graphics Support*
  - sous-sous-menu *Support for Frame Buffer Devices*
    - *DA8XX/OMAP-L1XX/AM335x Frame Buffer Support* : activer
    - *NXP TDA998 HDMI driver for Beaglebone black :* **désactiver**
  - sous-sous-menu *OMAP2+ Display Subsystem support*
    - *HDMI Support* : **désactiver**
    - *OMAP5 HDMI Support :* **désactiver**
    - sous-sous-sous menu *OMAP Display Device Drivers*
      - *HDMI Connector :* **désactiver**
  - *Direct Rendering Manager :* activer
  - *DRM Support for TI LCDC Display Controler :* activer

Concernant le *touchpad* :
menu *Device Drivers*
- sous-menu *Input device support*
  - *Event interface* : activer
  - *KeyBoards* : **désactiver**
  - *Touchscreens* : activer
    - *TI_Touchscreen interface* : activer
- sous-menu *Multifunction Drivers*
  - *TI_ADC/Touchscreen Chip Support* : activer (en statique *)
- sous-menu *Industrial I/O Support* : activer (en statique *)
  - sous-sous-menu *Analog to Digital Converters*
    - *TI's AM335x ADC Driver* : activer

Une fois le noyau et le système de fichiers modifiés, procéder à la compilation/création de l'ensemble, puis démarrer la carte BeagleBone en NFS.

VIII.3 Mise en œuvre dans un programme en langage C

VIII.3.1 Principe

Même si la bibliothèque SDL fournit des fonctions de haut niveau, nous allons la mettre en œuvre en définissant des fonctions directement interfaçables avec notre application, ce qui évitera de mélanger le code graphique avec l'algorithmique. Nous ferons une application avec 2 fichiers sources : le programme principal (display.c) et les fonctions graphiques (graph.c), le processus de compilation/chaînage sera contrôlé par un *makefile*.

VIII.3.2 Affichage d'une image

Nous avons tout d'abord besoin de créer les fonctions graphiques que nous utiliserons. Elles seront regroupées dans le fichier *graph.c* dont le code est donné ci-dessous :

```
#include <SDL.h>
#include <SDL_image.h>
#include <SDL_ttf.h>
#include "graph.h"

SDL_Surface *screen;        //variable globale cf commentaires

void Init_Graph()           //initialisation de SDL
{
  SDL_Init(SDL_INIT_VIDEO);
  TTF_Init();

//permet un acces direct au framebuffer
  screen = SDL_SetVideoMode(480, 272, 32, SDL_FULLSCREEN);
  if(screen==NULL)
  {
     printf("Error in video driver\n:%s",SDL_GetError());
  }
  SDL_ShowCursor(SDL_DISABLE); ///< plus de curseur pour l'ecran tactile

}

void Release_Graph()        // Arrêt de SDL
{
  SDL_FreeSurface(screen);
  SDL_Quit();
  TTF_Quit();
}

void Display()              // Affichage du buffer
{
    SDL_Flip(screen);
}

void Init_BG()              // Initialisation du fond
{
   Uint32 fond;

   fond = SDL_MapRGB(screen->format, 220,50,50); // fond bleu
   SDL_FillRect(screen, NULL, fond);
}
```

```
void Display_Image(char *Filename, int x, int y) // affichage image
{
    SDL_Surface *Image;
    SDL_Rect position;

    Image = IMG_Load(Filename);
    mask(Image);
    position.x = x;
    position.y = y;
    SDL_BlitSurface(Image, NULL, screen, &position);
    SDL_FreeSurface(Image);
}

void mask(SDL_Surface *surface)       // gestion problème RVB → BVR
{
     surface->format->Rmask = 0xFF0000;
     surface->format->Gmask = 0x00FF00;
     surface->format->Bmask = 0x0000FF;
}
```

Une remarque s'impose relativement à ce fichier qui ne contient que des fonctions. La variable « *screen* » qui est un pointeur sur une structure SDL est déclarée en globale. C'est un des rares cas où cet usage est toléré. Cela va nous permettre de ne pas déclarer de type graphique dans le programme principal et d'alléger l'appel des différentes fonctions graphiques. Cette usage peut être rapproché de la programmation objet où les méthodes peuvent accéder librement aux champs de l'objet. Si la variable globale n'est pas déclarée dans le fichier inclus alors son espace sémantique de validité est limité au fichier C et l'ensemble agit comme un objet sans en avoir la lourdeur.

Les différentes fonctions doivent être appelées par le programme principal (*display.c*) :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdbool.h>

#include "graph.h"

int main(int argc, char *argv[])
{
  Init_Graph() ;
  Init_BG() ;

  Display_Image("./OnOff.png", 10,10);
  Display();

  while(getchar()!='q');

  Release_Graph() ;

  return 0 ;
}
```

Créer par ailleurs le fichier en-tête, nommé *graph.h*, contenant les prototypes des fonctions écrites dans le fichier *graph.c*.

Enfin un fichier *Makefile* doit être créé afin de compiler et de chaîner ces fichiers avec les bonnes bibliothèques :

```
# Chemin Compiler :
CC = /sec/buildroot/output/host/usr/bin/arm-Linux-gcc
# Chemin Lib SDL :
CLSDL = /sec/buildroot/output/target/usr/lib

# Chemin Includes SDL :
CISDL = /sec/buildroot/output/build/sdl-1.2.15/include
CISDLI = /sec/buildroot/output/build/sdl_image-1.2.12
CISDLT = /sec/buildroot/output/build/sdl_ttf-2.0.11

#Option de compilations
CFLAGS = -g -ansi -std=c99

/targetfs/usr/local/bin/tp9 : tp9
      cp tp9 /targetfs/usr/local/bin/tp9

tp9 : tp9.o graph.o
      $(CC) -L$(CLSDL) -o tp9 -lSDL -lSDL_image -lSDL_ttf tp9.o graph.o

tp9.o : tp9.c
      $(CC) -c -I$(CISDL) -I$(CISDLI) -I$(CISDLT) $(CFLAGS)-o tp9.o tp9.c

graph.o : graph.c
      $(CC) -c -I$(CISDL) -I$(CISDLI) -I$(CISDLT) $(CFLAGS) -o graph.o
graph.c
```

On remarque dans ce fichier les chemins supplémentaires pour les fichiers inclus relatifs à SDL ainsi que le lien vers les bibliothèques statiques.

Une fois votre programme compilé et chaîné, vous devez le copier dans le répertoire */usr/bin* de votre cible (par NFS dans */targetfs*). Ce programme utilisant des fichiers annexes, ils doivent être eux aussi dans le même répertoire (*/usr/bin*) : vous devez donc copier les images *.png et les polices de caractères *.ttf (pour la suite) dans ce répertoire.

Enfin, préalablement à l'exécution vous devez désactiver la gestion de la souris par SDL (c'est un écran tactile) par la commande sous la cible : **export SDL_NOMOUSE=1** qui va positionner une variable d'environnement. Sans cela, le message « *souci driver* » va s'afficher sur la console.

Tester votre programme.

VIII.3.2 Affichage du texte

Une fois l'image affichée nous allons ajouter l'affichage d'un texte. Cette fonction est ajoutée dans le fichier graph.c :

```
void Display_Text(char *text, int x, int y, int size)
{
  SDL_Surface *text_surf ;
  SDL_Rect position;
  SDL_Color couleurPolice = {0, 255, 255};
  TTF_Font *police_ttf;

  police_ttf = TTF_OpenFont("/usr/local/bin/tp9.ttf", size); // police size

  position.x = x;    position.y = y;
  text_surf  = TTF_RenderText_Blended(police_ttf, text, couleurPolice);
  SDL_BlitSurface(text_surf, NULL, screen, &position);

  SDL_FreeSurface(text_surf);
  TTF_CloseFont(police_ttf);
}
```

Elle utilise 3 paramètres :

- le texte à afficher
- la position en x et en y du début du texte
- la taille du texte (les polices sont générées à partir de fichiers au format *ttf* qui est un format *true type*)

Ajouter au programme principal l'appel à cette fonction en affichant un texte reprenant vos noms et prénoms affichés en jaune.
Tester votre programme.

VIII.3.3 Tracé d'un trait (optionnel)

La bibliothèque SDL ne fournit pas directement des fonctions permettant de dessiner des éléments graphiques tels que les traits ou les polygones. Vous allez donc mettre au point un programme qui dessine des traits à partir d'images de taille faible en gérant le déplacement linéaire pour faire afficher un trait.

VIII.4 Ecran tactile (*touchpad*)

VIII.4.1 Présentation

Afin d'utiliser la fonctionnalité «écran tactile », il faut utiliser une bibliothèque supplémentaire appelée *TSLIB*.

Pour déterminer le point de toucher sur l'écran, on utilise les caractéristiques résistives de la zone sensible de l'écran. Lors du toucher, il y a contact entre 2 films résistifs. On mesure par triangulation la position du toucher à l'aide de 4 convertisseurs reliés à chaque bord d'un des films. Une position approximative est trouvée, souvent la précision est améliorée par un moyennage lors de la mesure.

VIII.4.2 Mise en œuvre

Comme à chaque fois qu'un nouveau périphérique est ajouté, il faut :

- modifier le *device tree* et le mettre au format utilisable par le noyau (*make device-tree*)
- ajouter des nouveaux éléments au noyau et le reconstruire (*make kernel*)
- ajouter des bibliothèques et autres logiciels dans le système de fichiers (*make filesystem*)

En fait nous avons fait l'essentiel du travail dans la partie précédente : le *device tree* fourni prend déjà en compte le *touchpad*, et les modules noyau ont aussi été ajoutés. Il ne reste donc qu'à ajouter la bibliothèque nécessaire pour faire fonctionner le *touchpad*.

Ajout de la bibliothèque :

Normalement, il suffirait de sélectionner cette bibliothèque dans la configuration de Buildroot. Comme nous utilisons une version modifiée de Buildroot (pour gagner en temps de compilation), à l'inverse l'ajout complet d'une bibliothèque nécessite une étape de plus : il faut la réintégrée à l'environnement Buildroot, cela est fait par la commande : *make packages/enable/tslib*

Une fois cette étape faite, il faut l'activer dans Buildroot : *make menuconfig*
et cocher dans les bibliothèques additionnelles : la bibliothèque *tslib*

Comme les machines virtuelles ne sont pas reliées directement à internet il faut récupérer les fichiers sources *tslib-1.1.tar.gz* sur un dépôt qui vous sera précisé en séance.

Reconstruire le système de fichiers : *make filesystem*.

Si tout c'est bien passé, le support pour le *touchpad* est opérationnel. Nous allons donc maintenant le tester.

VIII.4.3 Programme de test

Le programme de test vous est fourni. Il utilise un nouveau périphérique qui a été créé par le nouveau noyau (*/dev/input/event0*). Ce périphérique va créé un événement à chaque pression sur le touchpad. Nous allons donc intercepter cet événement et lire la position à ce moment là.

Ce programme est constitué de 2 fichiers source *.c* et d'un fichier d'en-tête *.h*.

Tout d'abord le fichier gérant le *touchpad* (*touch_screen.c*) :

```
#include "tslib.h"

int touch_position(int*px, int*py)
{
  struct tsdev *ts;
  struct ts_sample samp;
  int flag;

  ts = ts_open("/dev/input/event0", 0)           //opening touchscreen device
  if (!ts)
  {
     perror("ts_open");
     return(1);
  }
  if (ts_config(ts))
  {
     perror("ts_config");
     return(2);
  }

  flag = ts_read(ts, &samp, 1);                  // reading position
  if (flag < 0)
  {
     perror("ts_read");
     return(3);
```

```
  }

  *px= samp.x ; *py = samp.y ;

  ts_close(ts);
  return 0;
}
```

On peut remarquer que le *touchpad* est géré comme un pseudo-fichier, ce qui est très courant sous Linux pour la gestion des différents périphériques.

Le programme principal (main_ts.c) :

```
#include <stdlib.h>
#include <stdio.h>
#include "touch_screen.h"


int main(int argc, const char *argv[])
{
  int x, y ;
  while (1)
  {
     touch_position(&x,&y) ;
     printf ("position : %d, %d\n", x, y);
     usleep(1000);                    // just a small pause
  }
  return 0;
}
```

Ce programme effectue simplement un affichage de la position et une temporisation pour éviter d'avoir trop de valeurs envoyées.

Le fichier *Makefile* compile et chaîne les 2 fichiers ainsi que la bibliothèque en voici un extrait :

```
#chemin include TSLIB
CITSLIB=/sec/buildroot/output/build/tslib-1.1
touch_screen.o: touch_screen.c
	$(CC) -c -I$(CITSLIB) -ansi -std=c99 -o touch_screen.o touch_screen.c
```

Le chaînage doit se faire avec la bibliothèque *tslib* (c'est le -lts) :

```
main_ts: main_ts.o touch_screen.o
	$(CC) -lts -o main_ts main_ts.o touch_screen.o
```

Il ne vous reste plus qu'à tester.

Séance n°9 : Gestion de tâches sous Linux : processus et *thread*

Compétences visées :

- découverte des notions de processus/*thread* ;
- gestion des processus via les commande du Shell Linux ;
- gestion des processus via une API C ;
- gestion des *threads* via une API C ;
- développement d'une application complète multi-tâches sous Linux ;

Linux est un système d'exploitation dont le rôle principal est de gérer les ressources du système sur lequel il est installé : temps processeur, mémoire, accès aux périphériques… Linux est un système d'exploitation multi-tâches, il sait donc gérer, grâce à son ordonnanceur (*scheduler*), l'exécution « en parallèle » des différentes tâches lancées par le noyau ou l'utilisateur. La carte BeagleBone n'étant équipé que d'un seul processeur mono-coeur, Linux doit donc partager le temps processeur au mieux entre les différentes tâches, en donnant à l'utilisateur une impression de rapidité et de « simultanéité » des tâches alors qu'un processeur ne peut exécuter des instructions que de **manière séquentielle.**

Cette séance a pour objectif global une première approche des tâches sous Linux, en explorant la notion de processus et de *thread* (processus léger). Ces notions seront approfondies et complétées en 3e année : notion de « temps-réel », fonctionnement détaillée de l'ordonnanceur Linux…

IX.1 Les processus sous Linux

IX.1.1 Tâches et système d'exploitation

La plupart des systèmes d'exploitation actuels sont multi-tâches (contre-exemple : MsDOS et U-boot), il est donc important d'introduire cette notion générique, avant d'étudier la notion de processus sous Linux.

Afin d'utiliser au mieux les possibilités d'un système d'exploitation multi-tâches, l'utilisateur doit « découper » son programme en « petites actions » simples à exécuter pour le processeur, appelées **tâches.** Une tâche est donc définie comme une unité d'exécution du code (*thread of execution*), car elle est associée à un code à exécuter. Une tâche **n'est par contre pas une fonction en langage C**, car 2 tâches (ou plus) peuvent exécuter les mêmes lignes de code. Le rôle de l'**ordonnanceur** d'un système d'exploitation est de gérer la concurrence entre les différentes tâches pour l'accès aux ressources, notamment le temps processeur et l'accès aux périphériques.

Une tâche est définie :

- par un identifiant / nom, surtout utile à des fins de débogage ;
- une priorité (cf. cours 3e année) ;
- un *TCB*, pour *Task Control Block.* C'est une structure de données qui va contenir l'ensemble des informations liées à une tâche (nom, priorité, lignes de code…). L'ordonnanceur gère les tâches au travers des différents TCB. On parle de « descripteur de tâche ».

Dans le système Linux, on parle de *PCB*, pour *Process Control Block*. La structure de données associée est définie dans le fichier : /include/linux/sched.h

- une pile (*stack*), afin de stocker les données (variables, appel de sous-fonction…)
- un code en C à exécuter

IX.1.2 Processus - introduction

La première forme de tâche sous Linux se nomme : **processus.**

Les processus :

- ils sont référencés par un unique identifiant, le PID.

Ce PID peut servir à par exemple changer la priorité d'un processus ou arrêter ce dernier ;

- les processus peuvent être liés à de nombreuses ressources (cf. cours 3e année) : espace mémoire, *threads*, mémoire partagée, sémaphore, *timers*…

- le contexte d'un processus est l'ensemble de l'état du système lors de l'exécution de ce processus (principalement registres internet du processeur et état de la mémoire). Lors d'un changement de processus, il y a donc un changement de contexte avec sauvegarde du contexte du processus 1 et chargement du contexte du processus 2 ;

- au démarrage du système, l'ordonnanceur est le premier processus lancé, il a donc comme PID, 0 et comme PPID, 0, car il n'a pas de parent. *init* est le second processus lancé, il a donc le PID 1 et un PPID de 0. D'autres processus sont ensuite lancés par *init*, ce sont donc les « fils » de *init* et *init* est le père de ces derniers. Par exemple le processus *rpc*, a pour PID 439 et pour PPID 1, car c'est le fils du processus *init*.

- plusieurs **primitives C** peuvent être utilisées pour gérer les processus :
  - *fork()* : création dynamique d'un nouveau processus
    paramètres de retour : 0, pour le fils et le PID du fils, pour le père
    si échec, retour = -1 (manque de mémoire, par exemple) ;

  - *getpid()* : connaitre le PID d'un processus ;
  - *getppid()* : connaitre le PID du père d'un processus ;

  - famille de primitives *exec* pour charger en mémoire un nouveau code exécutable ;
  - *wait / waitpid* : le père attend la mort de l'un de ses fils ;

  - *sleep()* : mise en sommeil pour une durée déterminée
  - *pause()* : mise en sommeil sur l'arrivée d'un signal

Les signaux sont historiquement le premier outil de coordination/communication des processus. Il en existe différents types, par exemple :

- SIGINT. Ce signal est envoyé par le shell lorsque l'utilisateur tape Ctrl-C pendant l'exécution d'un programme. Il provoque normalement la terminaison du processus.

- SIGTERM. Ce signal est le signal utilisé par défaut par la commande kill(1) pour demander la fin d'un processus. Par défaut, la réception de ce signal provoque la terminaison du processus.

- Liste complète : https://sites.uclouvain.be/SystInfo/notes/Theorie/Fichiers/fichiers-signaux.html

  - *exit()* : terminaison d'un processus, sans condition ;
  - *kill()*:envoi d'un signal à un processus

Ces différentes primitives seront utilisées lors de la gestion des processus par un programme C.

IX.1.3 Processus – commandes shell

Ouvrir un terminal côté pc *host* et tester les commandes suivantes :

- *ps*

Combien de processus s'affichent ? Pourquoi ?
TTY : terminal contrôlant les différents processus ;
TIME : temps CPU utilisé par les processus.

- *ps -f*

Quelles informations supplémentaires se sont affichées par rapport à la commande précédente ?

- *ps -ef*

Quelles informations supplémentaires se sont affichées par rapport à la commande précédente ?

- *ps axjf* ou *ps -efH*

Cette commande permet de faire apparaître l'arborescence des processus, avec notamment la notion de parenté (père/fils)

- *top*

Quelle est l'utilité de cette commande ?

Cette commande monopolise a priori le shell, puisque vous n'y avez plus accès. Essayer la solution suivante :

- touches *CTRL +Z*

Que se passe-t-il ? Le processus *top* est-il détruit ou suspendu ?
CTRL+Z a pour effet d'envoyer le signal *SIGTSTP* au processus courant.

- *jobs -l*

Cette commande affiche les processus suspendu

- *bg %n*, avec n numéro (et non PID) du processus suspendu

Relance le processus suspendu en tâche de fond

De même, *fg %n*, relance le processus suspendu, mais au premier plan

- *kill PID* ou *kill %n* : met fin à un processus (suspendu ou non)

kill -9 PID met fin à un processus récalcitrant.

- touches *CTRL +C*

Que se passe-t-il ? Le processus top est-il détruit ou suspendu ?
CTRL+C a pour effet d'envoyer le signal *SIGINT* au processus courant.

Essayer à présent l'ensemble des ces commandes sur la carte BeagleBone.
Lesquelles fonctionnent ? Pourquoi certaines commandes ou options ne sont pas connues, alors qu'on a un système Linux comme sur le pc *host*.

IX.1.4 Processus – langage C

Les différents programmes ci-dessous ont pour objectif de vous montrer les possibilités de gestion de processus Linux, en langage C.

Un air de famille

Le programme ci-dessous crée grâce à la commande *fork* un processus fils qui exécute les mêmes lignes de code (fonction *main*) que le processus père.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main(void)
{
        int pid=0;
        int i=0;

        pid=fork();
        printf("PID du processus en cours :%d\n",getpid());
        printf("PID du père du processus en cours :%d\n",getppid());
        while(1);

        return 0;
}
```

Prévoir les lignes qui doivent s'afficher dans la console.
Copier les lignes de code dans un fichier, nommé *process.c.*
Compiler à l'aide d'un fichier *Makefile* votre programme (options de compilation : -g -ansi -std=c99)
Exécuter votre programme sur la carte BeagleBone
Les lignes affichées sont-elles conformes à votre prédiction ? Si non, prenez le temps de comprendre les différentes informations affichées.

En l'état, vous n'avez plus la main sur la console, car le programme ci-dessus comporte une boucle infinie *while(1)*. On peut sortir de cette boucle par l'appui sur les touches CTRL+C. Cela revient à envoyer le signal SIGINT (*interrupt*) aux processus en cours donc au père et à son fils, qui ont le même nom.

Le père attend son fils

Le programme ci-dessous est une version modifiée du programme précédent dans laquelle on fait appelle à la primitive *wait*. Cette dernière permet à un processus père d'attendre la fin de son processus fils, afin de continuer son exécution. Cela éviter de créer des processus Zombie que nous verrons par la suite.
Plus d'information ici sur la primitive *wait* et ses différentes options.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main(void)
{
        int pid=0;
        int i=0;

        pid=fork();
```

```
        switch(pid)
        {
                case -1:
                        printf("Erreur création processus !\n");
                        exit(1);
                        break ;

                case 0 :
                        printf("Je suis le fils PID=%d et mon père est PPID=%d\n",getpid(),getppid());
                        exit(0);
                        break;

                default:
                        printf("Ici le père, mon fils a un PID : %d\n",pid);
                        wait(0);
                        printf("Fin du père\n");
        }
        return 0;
}
```

Prévoir les lignes qui doivent s'afficher dans la console.
Modifier votre fichier *process.c*.
Compiler à l'aide d'un fichier *Makefile* votre programme (options de compilation : -g -ansi -std=c99)
Exécuter votre programme sur la carte BeagleBone.
Les lignes affichées sont-elles conformes à votre prédiction ? Si non, prenez le temps de comprendre les différentes informations affichées.

Le fils choisit une autre voie

Dans cette partie, l'objectif est de faire exécuter au processus fils un programme différent du processus père, grâce à la famille des primitives *exec*.

Commencer par créer un nouveau fichier, nommé *son.c* dont la fonction *main* effectuera les actions suivantes :

1. afficher le message « nous nous trouvons actuellement dans le programme fils ! » ;
2. afficher le PID du processus exécutant le programme.
3. afficher le message « nous quittons le programme fils ! » ;

Modifier votre fichier process.c, en rajoutant pour le cas 0 du switch : `execv (" . / son"" , 0) ;`

Prévoir les lignes qui doivent s'afficher dans la console.
Compiler à l'aide d'un fichier *Makefile* les 2 programmes (options de compilation : -g -ansi -std=c99)
Exécuter le programme *process* sur la carte BeagleBone.
Les lignes affichées sont-elles conformes à votre prédiction ? Si non, prenez le temps de comprendre les différentes informations affichées.

La famille exec comporte au total 6 primitives permettant de lancer l'exécution d'un programme. Les différences entre ses 6 primitives portent :

1. sur la forme des arguments utilisés lors de l'appel à la primitive. Par exemple :

```
        execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

ou

```
        char* arr[] = {"ls", "-l", "-R", "-a", NULL};
        execv("/bin/ls", arr);
```

2. sur l'utilisation des variables du *PATH* :

execl("/bin/ls", "ls", "-la", NULL);

ou

execlp("ls", "ls", "-la", NULL); *ls* est connue via le *PATH* Linux donc pas besoin d'indiquer son chemin absolu.

Un processus Zombie ?

On parle de processus Zombie si un processus est terminé, mais a toujours une existence dans la table des processus, un PID et est donc toujours présent dans le système. Cela arrive si un processus père n'a pas été conçu pour détecter la fin d'un processus fils, à l'aide des primitives *wait* ou *waitpid()*. Même s'ils ne monopolisent pas de ressources, l'accumulation de processus Zombie peut créer un bug dans le système d'exploitation.

Le programme ci-dessous est un exemple de création de processus Zombie.

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
 {
        int child_pid = fork ();
        if (child_pid > 0)
        {
                sleep (300);
        }
        else
         {
                exit (0);
        }
        return 0;
}
```

Copier les lignes de code dans un fichier zombie.c
Compiler à l'aide d'un fichier *Makefile* le programme (options de compilation : -g -ansi -std=c99)
Exécuter le programme sur la carte BeagleBone, en tâche de fond: *./zombie &*
Utiliser la commande : *top*

La colonne **STAT** fait apparaître l'état des différents processus :

- *SW = uninterruptible sleep (wait for an event)*
- *I = idle*
- ***R = running***
- ***S = sleeping***
- ***T/t = stopped***
- ***Z = zombie***

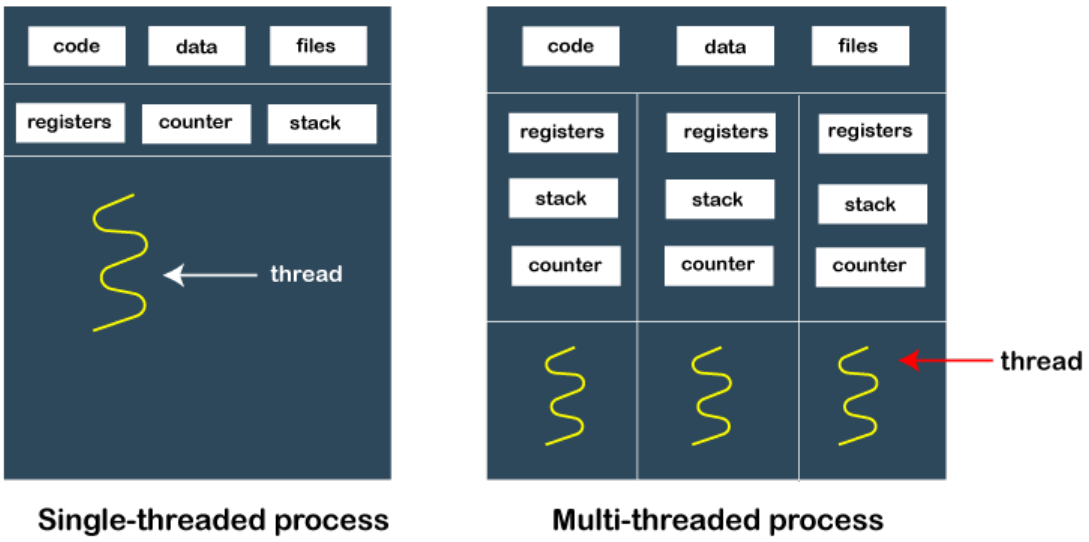
A-t-on réussi à créer un processus Zombie ? Pourquoi ?
Modifier votre programme, afin que le père attende la fin du processus de son fils, faisant ainsi disparaître l'état zombie.

IX.2 Les *threads* sous Linux

IX.2.1 Introduction

Les *threads* sont des processus « légers », c'est-à-dire qu'ils permettent de « paralléliser » l'exécution de plusieurs programmes sous Linux, mais ils existent par contre à l'intérieur d'**un processus** (cf. figure ci-dessous). Dans la partie précédente, nous avons vu que l'appel à la primitive *fork* permettait de créer un processus fils qui exécutait le même programme que son père, mais avec son propre contexte (espace mémoire alloué, descripteur de fichiers…). Tous les *threads* d'un même processus partagent par contre un même espace mémoire et un même contexte. Cela permet plus facilement de partager des ressources entre *thread*, mais peut avoir des inconvénients. Une variable modifiée dans un *thread* le sera par exemple, dans tous les *threads* du processus.

GNU/Linux implémente l'API de *threading* standard POSIX[6] (appelée aussi *pthreads*). Toutes les fonctions et types de données relatifs aux *threads* sont déclarés dans le fichier d'entête : <*pthread*.h>. Les fonctions de *pthread* ne font pas partie de la bibliothèque standard du C. Elles se trouvent dans la bibliothèque *libpthread*, vous devez donc ajouter *-lpthread* sur la ligne de commande lors de l'édition de liens de votre programme. Les différentes primitives utiles de cette bibliothèque seront vues au fur et à mesure des exemples.



IX.2.2 Exemples

Création d'un *thread*

Le programme ci-dessous permet la création d'un *thread* à l'aide de la primitive *pthread_create* qui attend comme arguments *:*

1. un pointeur vers une structure de type *pthread_t*, structure qui sera remplie après la création du *thread* et dont l'ordonnanceur Linux va se servir ;
2. un pointeur vers une structure de type *pthread_attr_t* qui peut contenir les paramètres du *thread*. *NULL* ici, car aucun paramètre ;
3. une fonction à exécuter dans le *thread* ;
4. un éventuel argument à passer à la fonction ;

Cette primitive est l'équivalent de *fork* pour les processus.

[6]POSIX pour Portable Operating Système Interface est un ensemble de normes développées et maintenues depuis 1988 et ayant pour objectif la portabilité du code entre différents systèmes d'exploitation : UNIX, Linux, macOs, Android… Les primitives de l'*API threading* sont ainsi utilisables sans modifications majeurs sous Linux ou Android, par exemple.

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

void* fonct_thread(void* p_i)
{
        int* n = p_i ;
        while(1)
        {
                printf("je suis le thread numero : %d \n",*n);
        }
        return NULL ;
}

int main(int argc, char * argv[])
{
        pthread_t pthread_id;
        thread_create(&pthread_id,NULL,fonct_thread,i);

        sleep(3) ;

        return 0 ;
}
```

Prévoir les lignes qui doivent s'afficher dans la console.
Créer le fichier *thread*_ex1.c et copier ces lignes de code.
Compiler à l'aide d'un fichier *Makefile* votre programme (options : -ansi -std=c99 -lp*thread*)
Exécuter votre programme sur la carte BeagleBone.
Les lignes affichées sont-elles conformes à votre prédiction ? Si non, prenez le temps de comprendre les différentes informations affichées.

« Suicide » d'un *thread*

Une fois créé, un *thread* peut se « suicider » à l'aide de la primitive :
*void pthread_exit(void *return_value);*
C'est l'équivalent de la primitive *exit* pour les processus.

Modifier votre programme pour mettre fin à la boucle infinie du *thread*, après la première itération.
Compiler et tester.

Synchronisation des *threads*

Comme dans le cas des processus, il faut que le *thread* principal, c'est-à-dire celui qui exécute la fonction *main*, attende la fin des différents *threads* qu'il a créés, avant de se terminer. Autrement dit, il faut qu'un *thread* père attende la fin de ses *threads* fils, sous peine de créer des *threads* Zombie. Cette attente est possible grâce à la primitive : *int pthread_join(pthread_t thread, void *return_value);*

Le programme (d'après Université de Aix-Marseille) ci-dessous comporte 3 *threads* :

- le *thread* principal (le père)
- le *thread* lire (premier fils), qui lit les caractères tapés au clavier et les transmet à un second *thread*, par l'intermédiaire de la variable theChar
- le *thread* affichage (second fils) qui affiche les caractères transmis. Tout s'arrête quand le caractère 'F' est tapé au clavier.

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

volatile char theChar = '\0';
volatile char afficher = 0;
void* lire (void* name)
{
	do {
	while (afficher == 1) ;		/* attendre mon tour */
	theChar = getchar();
	afficher = 1;			/* donner le tour */
	}
	while (theChar != 'F');
	return NULL;
}

void* affichage (void* name)
{
	int cpt = 0;
	do {
		while (afficher == 0)
		{
			cpt ++; /* attendre */
		}
		printf("cpt = %d, car = %c\n", cpt, theChar);
		afficher = 0; /* donner le tour */
	}
	while (theChar != 'F');
	return NULL;
}

int main (void)
{
	pthread_t filsA, filsB;
	if (pthread_create(&filsA, NULL, affichage, NULL))
	{
		perror("pthread_create");
		exit(EXIT_FAILURE);
	}
	if (pthread_create(&filsB, NULL, lire, NULL))
	{
		perror("pthread_create");
		exit(EXIT_FAILURE);
	}

	printf("Fin du pere\n") ;

	return (0);
}
```

Créer le fichier *thread*_ex2.c et copier ces lignes de code.
Compiler à l'aide d'un fichier *Makefile* votre programme (options : -g -ansi -std=c99 -lp*thread*)
Exécuter votre programme sur la carte BeagleBone.
Le programme a-t-il le comportement voulu ? Pourquoi ? Aide : regarder la liste des processus en fin de programme à l'aide de la commande *ps*
Modifier votre programme pour qu'il ait le comportement souhaité.

Mise en application

A l'aide des différentes primitives concernant les threads vues précédemment, créer un programme affichant sur l'écran une image et faisant clignoter une led toutes les 2s. On pourra mettre fin au programme par appui sur la touche 'q' pour la partie écran et sur la touche 's ' pour la partie LED.

Compiler et tester votre programme sur la carte BeagleBone.

Complément de cours : **https://mtodorovic.developpez.com/linux/programmation-avancee/?page=page_4**

Séance bonus : *Communications réseau et notion de sockets*

IX.2) Présentation

Les *sockets* sont l'interface de programmation proposée sur tous les ordinateurs utilisant TCP/IP (c'est à dire tous !). Elles est basée sur la pile de protocole TCP/IP et propose des fonctions bas niveaux d'accès au réseau. Elle se situe juste au-dessus de la couche transport (couche 4 du modèle OSI) et n'a donc pas à s'occuper de la transmission. Cette interface ne « voit » qu'un échange entre 2 postes. Elle est basée sur une approche client-serveur : c'est à dire qu'un côté est en attente de l'échange. La mise en œuvre complète et fiable nécessite d'utiliser des *threads* séparés pour l'émission et la réception car la réception est bloquante : pour ce TP on définira des échanges où il ne peut y avoir (sauf incident) de blocage et donc nous n'utiliserons pas les *threads.* Il est à noter que l'interface *sockets* peut aussi être utilisée pour communiquer entre 2 processus, sous Linux, sans notion de réseau.

La pile TCP/IP initiale a été créée sous Linux, elle utilise donc le concept de pseudo-fichiers pour faire les communications mais les descripteurs sont associés à des structures de type *socket*. Sous Windows, la même interface a été réalisée, les programmes sont donc quasiment compatibles (sauf l'initialisation). De nombreuses implémentations de plus haut niveau existent (C++, Java, Javascript…) mais elles sont toutes basées sur la pile TCP/IP.

Plusieurs types de *sockets* existent : les 2 plus courantes sont les *sockets* TCP[7] (avec connexion) et les *sockets* UDP[8] (sans connexion). Nous commencerons par ces dernières même si la complexité est à peu prêt équivalente.

Comme le réseau est utilisé par défaut sur les cartes Beaglebone, il n'y aura aucun ajout de périphériques ou *drivers* à prévoir.

IX.2.2) Server/client UDP

Code Server

Le programme server UDP sera exécuté sur le pc *host*. Il faudra donc compiler ce dernier, à l'aide d'un fichier *Makefile* pour processeur i386 et non pour architecture ARM.

Les bibliothèques utiles sont :

```
#include <sys/socket./h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
```

Voici les différentes étapes à suivre :

1. pour utiliser une *socket*, il faut la déclarer avec le type *int : int sockfd* ;

2. il faut créer la *socket* à l'aide de la fonction *socket* dont le prototype est le suivant :

```
int socket(int domain, int type, int protocole) ;
```

   - *domain* : famille de protocoles utilisée, AF_INET pour le protocole IPv4, AF_INET6 pour le protocole IPv6…
   - *type* : type de service choisi, SOCK_STREAM pour TCP et SOCK_DGRAM pour UDP ;
   - *protocole* : inutile dans le cadre de la pile TCP/IP. Toujours mis à 0, dans nos applications.

[7]TCP : Transmission Control Protocol
[8]UDP : User Datagram Protocol

- Le paramètre de retour est un nombre entier correspondant à un descripteur de *socket*, descripteur qui sera utilisé dans les autres fonctions du code. En cas d'erreur, la fonction renvoie -1.

```
int sockfd = socket(AF_INET, SOCK_DGRAM,0) ;
if(sock==-1)
{
        .... traitement de l'erreur...
}
```

3. une fois la *socket* créée, il est possible (mais pas obligatoire) de la configurer avec la fonction :

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)
```

Pour plus de détails, voir ici : https://linux.die.net/man/3/setsockopt

4. il faut ensuite attacher la *socket* à un point de communication local, c'est-à-dire un port internet et une adresse IP, grâce à la fonction *bind* :

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

- *sockfd* : fichier descripteur de la *socket* ;
- *addr* : pointeur (adresse) de la structure *sockaddr* du server (cf. ci-dessous)
- *addrlen* : taille mémoire occupée par le contexte d'adressage du server. Pour l'obtenir, il suffit d'utiliser la fonction sizeof : sizeof(addr) ;
- la fonction retourne 0 en cas de succès ou *SOCKET*_ERROR (-1), en cas d'erreur.

La structure *sockaddr* utilisée avec le protocole TCP/IP est une adresse AF_INET, définie dans le fichier d'en-tête <netinet/in.h> :

```
struct sockaddr_in
{
        short sin_family;       //type de famille - e.g. AF_INET
        u_short sin_port;       //port à contacter – e.g. htons(1500)
        struct in_addr sin_addr;        //adresse de l'hôte
        char sin_zero[8];               /* initialise à zéro */
}
```

La structure *sin_addr* contient un unique champ, nommé *s_addr*, dont la valeur est l'adresse IP du server. 2 solutions :

- `inet_addr("192.168.101.16") ;` //adresse spécifique
- `htonl(INADDR_ANY) ;` //accès à toutes les adresses possibles côté server

```
struct sockaddr_in server;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_family = AF_INET;
server.sin_port = htons(1500);
bind(sockfd, (struct sockaddr*)&server, sizeof(server));
```

5. les étapes de configuration étant faites, il reste à gérer la réception des messages client et éventuellement à envoyer des informations à ces derniers. Les fonctions à utiliser sont :

```
ssize_t recvfrom(int fd, void *buf, size_t count, int flags, const struct sockaddr *addr, socklen_t addrlen);
```

- *fd* : descripteur de *socket* ;
- *buf* : tableau de caractères dans lequel mettre le message reçu ;
- *count* : nombre de caractères à recevoir ;
- *flags* : liste des drapeaux pour modifier le comportement des *sockets* ;
- *addr* : structure symbolisant l'adresse du client ;
- *addrlen* : taille de l'objet addr ;
- le code de retour correspond au nombre de caractères reçus ou à -1, en cas d'erreur

Pour écrire dans une *socket*, on utilise la fonction :

```
ssize_t sendto(int fd, const void *buf, size_t count, const struct sockaddr *addr, socklen_t addrlen);
```

- *fd* : descripteur de *socket* ;
- *buf* : tableau de caractères contenant le message à envoyer ;
- *count* : nombre de caractères à envoyer
- *flags* : liste des drapeaux pour modifier le comportement des *sockets*.
- *addr* : structure symbolisant l'adresse du client ;
- *addrlen* : taille de l'objet addr ;
- le code de retour correspond au nombre de caractères envoyés ou à -1, en cas d'erreur

```
const char mess[20]="message reçu par le server\n"
struct sockaddr_in client ;
unsigned int l = sizeof(client) ;
while(1)
{
        int n ;
        n=recvfrom(sockfd,buffer,BUFFERSIZE,0,(struct sockaddr*)&client,&l) ;
        if(n==-1)
        {
                ...traitement de l'erreur…
        }
        else
        {
                ...traitement du buffer…
        }

        n=sendto(sockfd,mess,strlen(mess),0,(struct sockaddr*)&client,l) ;
        if(n==-1)
        {
                ...traitement de l'erreur…
        }
}
```

A partir des éléments ci-dessus, créer un fichier server.c permettant la mise en place d'un server UDP.
Compiler votre programme.

Code Client

Le programme client UDP sera exécuté sur la carte BeagleBone. Il faudra donc compiler ce dernier, à l'aide d'un fichier *Makefile* pour architecture ARM.

Les étapes à respecter sont :

1. créer une socket UDP ;
2. envoyer un message au server ;
3. attendre une éventuelle réponse du server ;
4. retourner éventuellement à l'étape 2 ;
5. fermer le descripteur (fonction *close(socketfd)*) de socket et sortir du programme (*exit(0)*)

En reprenant les éléments du server UDP, créer un fichier client.c permettant la mise en place d'un client UDP. Le canevas du programme est donné ci-dessous.
Compiler votre programme.

```
#include <sys/socket./h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#define BUFFERSIZE          20

int main(void)
{
        int sockfd ;
        struct sockaddr_in server ;
        server.sin._addr.s_addr=inet_addr("192…. ")        // à compléter
        server.sin_family=     //à compléter
        server.sin_port=htons(1500) ;
        struct sockaddr_in client ;
        unsigned int l=sizeof(server) ;

        char mess[BUFFERSIZE] ;
        const char mess2[BUFFERSIZE]= "message test client" ;

        sockfd=         //création socket
        if(sockfd==-1)
        {
                … gestion d'erreur…
        }
        while(1)
        {
                //envoi message
                //réception message
        }

}
```

En exécutant le programme server sur le pc *host* et le programme client côté carte BeagleBone, tester votre server UDP.
Modifier votre fichier *Makefile* pour permettre le fonctionnement inverse : server côté carte et client côté pc. Tester votre nouvelle solution.

IX.2.3) Server/client TCP

Transformer votre server UDP en server TCP, avec mécanisme de connexion :
https://www.tala-informatique.fr/wiki/index.php/C_socket#C.C3.B4t.C3.A9_client
document sous Mootse

IX.2.4) Exemples d'application

- Envoyer sur le PC la position des appuis sur l'écran tactile, via le réseau et afficher cette dernière
- Afficher 2 icônes (no_sound et on_off) différentes sur l'écran et déclencher 2 actions différentes (des affichages) à distance sur le PC lors des appuis.