ELE674 Labo

Labo 1

Ressources used

`Lab/Lab1-System/CustomSystem`
⇒ Go to `CustomSystem`
⇒Decompress files with command:

```shell
for file in *.tar*; do
    tar -xf "$file"
done
```

Contenu Lab

Step 1: Manually compiling the image

In the first labo there is compilation and creation of the elements required by the boot loader and the OS. In the end we get all the files that are used to load the OS on the beagle bone.

- **Compilation U-Boot**
- **Compilation Kernel**
- **Compilation BusyBox**
- **Creating Root Filesystem**
- **Creating the final image from the previous elements**

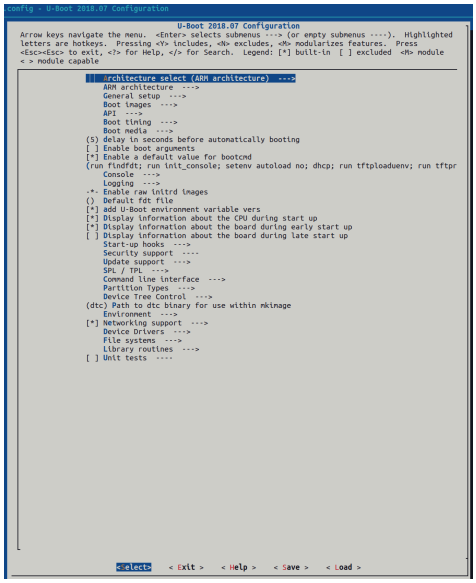
The following commands are used for that:

```shell
cd u-boot-2018.07/
patch -p1 < ../ele674_uboot_patches/ele674_20200901.patch
cp ../.config_uboot_ele674 .config
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- all

cd ../KERNEL/
cp ../.config_ti-linux-rt-4.14.y .config
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- zImage
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- modules
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- dtbs

cd ../busybox-1.31.1/
cp ../.config_busybox_ele674 .config
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi-

cd ..
mkdir RootFS
cd RootFS
mkdir dev proc sys etc lib bin sbin usr root mnt tmp var
mkdir etc/init.d usr/bin usr/sbin usr/lib var/log

cd ../KERNEL/
make -j12 ARCH=arm INSTALL_MOD_PATH=../RootFS modules_install

cd ../busybox-1.31.1/
make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- CONFIG_PREFIX=../RootFS install

cd ../RootFS/
cp -r ../temp/lib/* lib/.
cp -r ../temp/usr/lib/* usr/lib/.
cp -r ../temp/etc/* etc/.
rm lib/modules/4.14.108/build
rm lib/modules/4.14.108/source

cd ..
sudo makeimage.sh RootFS
```

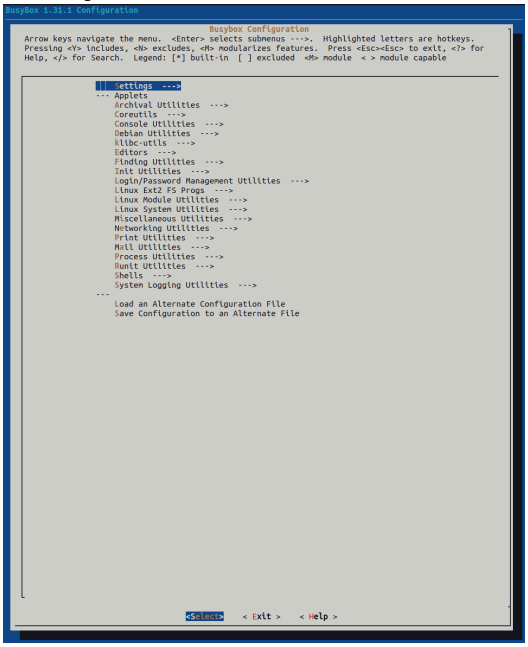
Commands explanation

1. **U-Boot Build:**
   - `cd u-boot-2018.07/`: Change to the U-Boot source directory.
   - `patch -p1 < ../ele674_uboot_patches/ele674_20200901.patch`: Apply a patch to U-Boot source code.
   - `cp ../.config_uboot_ele674 .config`: Copy a pre-existing configuration file for U-Boot.

- `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- all`: Build U-Boot with 12 parallel jobs for the ARM architecture using a specified cross-compiler.

> **Info**
>
> By default the `all` option will compile and load configuration from the `.config` file.
> ⇒ It is also possible to open a ***GUI*** to add/remove some of these options with the command `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- menuconfig`
> We then get the GUI:
>
> 
>

2. **Kernel Build:**
   - `cd ../KERNEL/`: Change to the Linux kernel source directory.
   - `cp ../.config_ti-linux-rt-4.14.y .config`: Copy a configuration file for the specific version of the Linux kernel.
   - `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- zImage`: Build the kernel image (zImage). This is a format that can be loaded by bootloaders.
   - `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- modules`: Build kernel modules. Modules are pieces of code that can be loaded into the kernel on demand, allowing for a modular approach to kernel functionality.
   - `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- dtbs`: Build device tree blobs (DTBs) needed for booting. Device Tree is a data structure used to describe the hardware components of a system to the Linux kernel. DTBs are essential for ARM devices to inform the kernel about the hardware configuration.
3. **BusyBox Build:**
   - `cd ../busybox-1.31.1/`: Change to the BusyBox source directory.
   - `cp ../.config_busybox_ele674 .config`: Copy a BusyBox configuration file.
   - `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi-`: Build BusyBox.

> **Info**
>
> By default the `all` option compile and load configuration from the `.config` file.
> ⇒ It is also possible to open a ***GUI*** to add/remove some of these options with the command `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- menuconfi`
> We then get the GUI:
>
> 
>

4. **Root Filesystem Setup:**

- `cd ..`: Go back to the parent directory.
- `mkdir RootFS`: Create a directory for the root filesystem.
- `cd RootFS`: Change to the new RootFS directory.
- Create various directories (`dev`, `proc`, `sys`, etc.) necessary for a Linux filesystem.
- `cd ../KERNEL/`: Return to the kernel directory.
- `make -j12 ARCH=arm INSTALL_MOD_PATH=../RootFS modules_install`: Install kernel modules into the RootFS directory.

5. **BusyBox Installation:**
   - `cd ../busybox-1.31.1/`: Return to the BusyBox directory.
   - `make -j12 ARCH=arm CROSS_COMPILE=/usr/src/gcc-linaro-7.5/bin/arm-linux-gnueabi- CONFIG_PREFIX=../RootFS install`: Install BusyBox into the RootFS.
6. **Copy Additional Files:**
   - `cd ../RootFS/`: Change to the RootFS directory.
   - Copy libraries and configuration files from a temporary directory into the appropriate locations in RootFS.
   - Remove build and source links from the kernel modules directory to clean up.
7. **Create Final Image:**
   - `cd ..`: Go back to the parent directory.
   - `sudo makeimage.sh RootFS`: Run a script to create a filesystem image from the RootFS directory.

Step 2: Compiling the image using Buildroot

The last step was quite laborious, and takes a lot of time.
⇒ That's why tools such as ***BuildRoot*** were done.

Using these tool, it is now possible to manually select the packages/ configurations you want to use for the différents blocks used for booting (U-boot, kernel, busybox, other modules...).

Ressources used

⇒ Go to `buildroot-202xxx` directory.
There will be a default .config file containing the default configuration.

Defining the configuration using a GUI

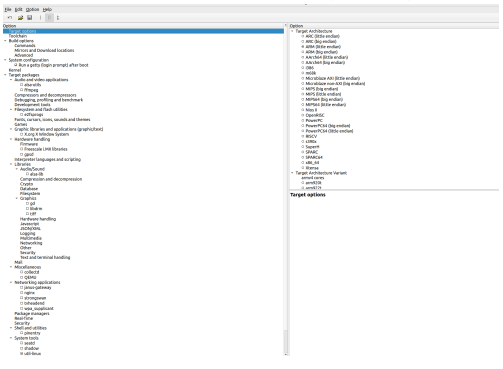
It is also possible to edit with a gui. For that there is two possibility:

```shell
make xconfig
```

The other one is:

```shell
make menuconfig
```

Both are very similar and will open a gui configuration windows in which you can select which modules/options you want to use.



Step 3: Chargement de l'image compilée sur le drone

Dans le résultat des opérations/compilations, récupérer les fichiers suivants:

Récupération des fichiers à déposer sur le drone

1. **am335x-boneblue.dtb** :
   - **Description** : C'est un fichier de blob d'arbre de périphériques (Device Tree Blob).
   - **Fonction** : Il décrit le matériel de la carte AM335x (par exemple, BeagleBone Blue). Le noyau Linux utilise ce fichier pour comprendre la configuration matérielle et les périphériques disponibles.
2. **rootfs.ext2.gz** :
   - **Description** : C'est une image compressée du système de fichiers racine (Root File System) au format ext2.
   - **Fonction** : Elle contient tous les fichiers et répertoires nécessaires pour le système Linux, y compris les utilitaires, bibliothèques et configurations. Lors du démarrage, cette image est décompressée et montée comme système de fichiers racine.
3. **u-boot.img** :

- **Description** : C'est l'image principale de U-Boot.
- **Fonction** : U-Boot est un chargeur de démarrage utilisé pour initialiser le matériel et charger le noyau Linux (et éventuellement le système de fichiers) en mémoire au démarrage.

4. **u-boot-spl.bin** :
   - **Description** : C'est l'image du Secondary Program Loader (SPL) de U-Boot.
   - **Fonction** : Le SPL est une version simplifiée de U-Boot, qui est utilisée pour initialiser des systèmes où la mémoire et les ressources sont limitées. Il est généralement chargé en premier, puis il charge l'image principale de U-Boot.
5. **zImage** :
   - **Description** : C'est l'image compressée du noyau Linux.
   - **Fonction** : Le noyau est le cœur du système d'exploitation. L'image zImage est utilisée pour démarrer le système Linux et gérer les ressources matérielles et les processus.

Sure! Here's the translation:

> **Info**
>
> These files are relatively complex to find; they are also located in the `/Lab/Lab1-System/SystemTest` folder.

Loading Files onto the Drone

⇒ Place the files mentioned earlier in the `/tftpboot` directory.
⇒ Then, when the drone starts up, it will automatically look for its bootloader and image at this location.

Serial Connection with the Drone

⇒ Open the *Serial Port Terminal* application and select the following settings:

- Port: /dev/ttyUSB0
- Baud rate: 115200
  ⇒ Then, power off and power on the drone: The drone will then fetch the bootloader, the image, and other configurations from the `/tftpboot` folder of the connected machine.
  ⇒ It will then boot up. Messages will normally be displayed in the terminal indicating the different phases of the boot process.

> **Info**
>
> It is possible to interrupt the boot process and modify options at this point.

⇒ When prompted for a login, enter `root`.

Labo 2

Consignes

⇒ Lire la documentation utilisation Eclipse

Importer le projet existant

import > General > Existing project into workspace >
select root directory

Configurer le projet

click droit sur le projet > properties >

Interconnexion avec le drone

cf la documentation Eclipse page 12

- MAIS l'adresse page 15 n'est pas la bonne, c'est `192.168.5.1`
- MAIS page 24: pas besoin de créer un fichier *.gdbinit*, celui-ci est fournit
- MAIS page 27: saisir le chemin de l'application lorsque installée sur le drone
  `/root/worksplace/ELE674-Lab1/ELE674-Lab1`
  ⇒ La dernière partie *ELE674-Lab1* est le nom de l'exécutable

  > ⚠ **Warning**
  >
  > Le chemin est supprimé à chaque fois sur le drone, il faut donc le recréer à chaque build, c'est long et inutile

⇒ just mettre `/root/DroneFirmwareBeaglebone` à la place. */root/* existe par défaut et *DroneFirmwareBeaglebone* est le nom de l'exécutable

Date de remise : sera définie par mail probablement vendredi minuit

- Fichier config pour le noyau linux dégraissé processus manuel
- Fichier config pour le noyaux linux dégraissé processus avec BuildRoot

Chargement de l'image crée

- Copier les dossiers de `labo1/system_test` dans `/tftpboot`

Connexion avec le drone

- ouvrir l'application *serial port terminal*
- sélectionner port > USB0

Connexion au drone par terminal

Configuration de l'interface réseau

Taper la commande

```shell
if config -a
```

Le résultat de la commande contient normalement

```shell
usb_beaglebone: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::dcad:beff:feef:0  prefixlen 64  scopeid 0x20<link>
        ether de:ad:be:ef:00:00  txqueuelen 1000  (Ethernet)
        RX packets 10  bytes 656 (656.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 4835 (4.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

⇒ Il va falloir configurer cette interface
Pour cela:

```shell
sudo ifconfig usb_beaglebone 192.168.5.1 up
```

On a alors:

```shell
usb_beaglebone: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.1  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::dcad:beff:feef:0  prefixlen 64  scopeid 0x20<link>
        ether de:ad:be:ef:00:00  txqueuelen 1000  (Ethernet)
        RX packets 11  bytes 712 (712.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 48  bytes 10807 (10.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

⇒ L'interface possède désormais une adresse côté ordinateur. Celle ci est `192.168.5.1`

Cependant l'adresse du beagle est `192.168.5.200`

Pour tester la connexion avec le beagle on peut utiliser la commande ping:

```shell
ping 192.168.5.200
```

Si le ping fonctionne cela signifie que la connexion fonctionne

Connexion en ssh au drone

On peut donc désormais se connecter au drone, notamment en *ssh* en utilisant lal commande:

```shell
ssh root@192.168.5.200
```

⇒ Accepter la clé d'encryption la première fois

Pour se déconnecter, utiliser la commande:

```shell
exit
```

Utilisation de l'interface graphique pour naviguer dans le système de fichier du drone

Ouvrir le navigateur de fichier > Other location > saisir l'adresse `sftp://root@192.168.5.200/`

Architecture du logiciel embarqué

Les capteurs envoient des messages aux à des fichiers virtuels: ex: /dev/i2c_gyro

⇒ Ces capteurs vont renvoyer des données sous forme de structure

```c
struct Sensor_struct {
const char
const char
int
uint16_t
pthread_spinlock_t
pthread_mutex_t
pthread_cond_t
pthread_t
pthread_t
uint16_t
uint16_t
SensorParam
SensorRawData
SensorData
};
```

"Pasted image 20241008110152.png" could not be found.

Contrôle moteur

```c
struct motor_struct {
uint16_t pwm[4]; //motor speed 0x00-0x1ff
uint16_t led[4];
int file;
pthread_t MotorThread;
pthread_spinlock_t MotorLock;
};
```

⇒ La trame à envoyer au moteur est présenté dans l'énoncé de laboratoire

Remarque:

Dans `DroneFirmare.c` possibilité de tester manuellement les moteurs:

```c
/******************** Utile pour vos tests avec les moteurs ********************

if (ch > 0) {

printf("%c", ch);

switch (ch) {

case '1' : Motor.pwm[0] += (Motor.pwm[0] < 510) ? 1 : 0;

break;

case '2' : Motor.pwm[1] += (Motor.pwm[1] < 510) ? 1 : 0;

break;

case '3' : Motor.pwm[2] += (Motor.pwm[2] < 510) ? 1 : 0;

break;
break;

case '4' : Motor.pwm[3] += (Motor.pwm[3] < 510) ? 1 : 0;

break;

case '!' : Motor.pwm[0] -= (Motor.pwm[0] > 100) ? 1 : 0;

break;

case '@' : Motor.pwm[1] -= (Motor.pwm[1] > 100) ? 1 : 0;

break;

case '#' : Motor.pwm[2] -= (Motor.pwm[2] > 100) ? 1 : 0;

break;

case '$' : Motor.pwm[3] -= (Motor.pwm[3] > 100) ? 1 : 0;

break;

default : break;

}

}

*********************************************************************************/
```

Dans `DroneFirmware.c` commenter les fonctions suivantes:

```c
// SensorsStart();
// AttitudeStart();
// SensorsLogsStart();
// MavlinkStart();
// ControlStart();



// MavlinkStop(&Mavlink);
// ControlStop(&Control);
MotorStop(&Motor);
// SensorsLogsStop(SensorTab);
// SensorsStop(SensorTab);
// AttitudeStop(AttitudeTab);
```

Altitude.C
⇒ les foncitons *MotorInit()*, *MotorStart()* et *MotorStop()* sont à coder.
⇒ La fonction pour initialiser les tâches moteurs sont déjà présentes

Cadence moteur

⇒ C'est un système temps réel, il faut donc disposer d'une horloge .
Pour cela la fonction *SigTimerHandler(int signo)* est fournie.
⇒ Celle ci va générer un signal (équivalent d'une interruption) à des intervalles donnés.

⇒ Du code devra être rajouté à l'intérieur pour gérer les différentes tâches

⇒ Les fonctions *StartTimer()* et *StopTimer()* sont également fournies afin d'initialiser et d'arrêter l'horloge

Questions

- Lors d'un dévellopement en équipe, faut il mettre les path/ fichiers src de lib et paramètres de workplace dans le repo git pour que n'importe qui soit capable de dévelloper après un clone?