ENSEIRB-MATMECA

MISE EN ŒUVRE DE LINUX EMBARQUE SUR CARTE RASPBERRY PI

Patrice KADIONIK
kadionik.enseirb-matmeca.fr

TABLE DES MATIERES

1. BUT DES TRAVAUX PRATIQUES

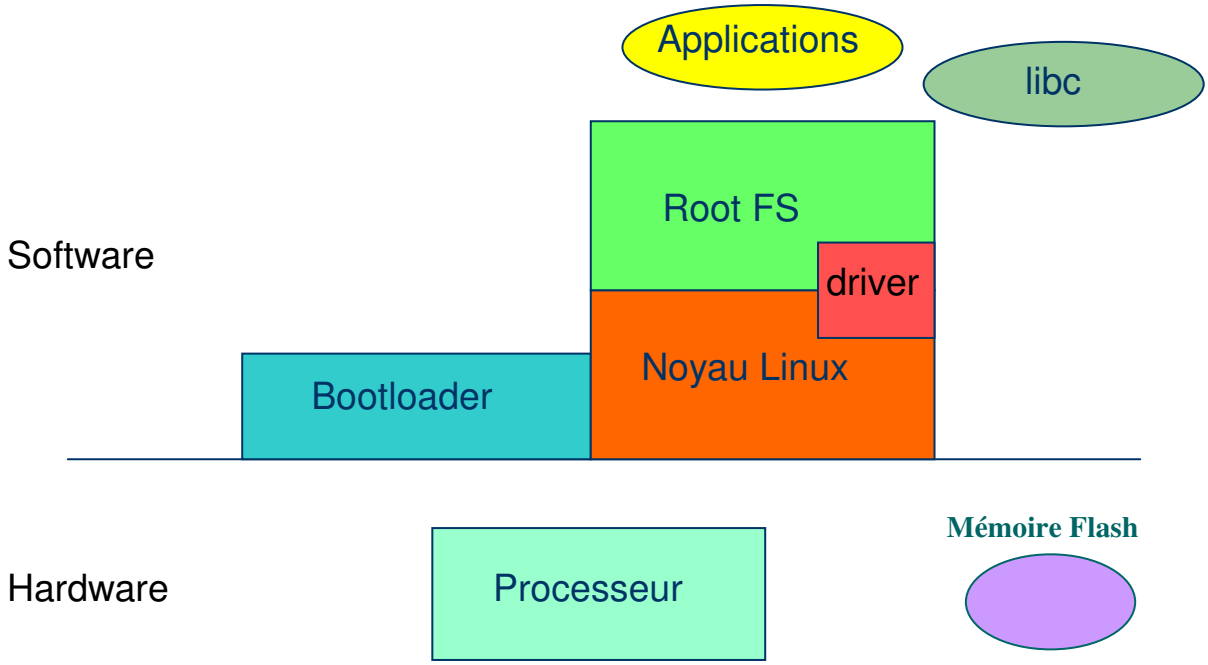
Le but de ces Travaux Pratiques est d'étudier la mise en œuvre de Linux embarqué sur une carte Raspberry Pi. Ces TP ont été créés en 2001 initialement sur une carte d'évaluation Motorola ColdFire EVB5407C3. Puis, ils ont été portés sur la carte Analog Devices Blackfin BF537 en 2011 afin de pouvoir entre autre travailler avec le *bootloader u-boot*. Enfin, en 2022, ils ont été portés sur la carte Raspberry Pi 3B.

Il s'agit de créer une distribution Linux embarqué adaptée et optimisée à la carte cible Raspberry Pi comprenant les pilotes de périphériques nécessaires pour contrôler les E/S.

Pour rappel, un système Linux embarqué contient les briques logicielles de base suivantes :

- Un *bootloader* Linux pour initialiser la carte au *reset* et pour charger un noyau Linux de façon manuelle ou automatique. On utilise généralement *u-boot*.
- Le noyau Linux adapté à la carte cible.
- Un système de fichiers *root*. On généralement *busybox*.
- Les pilotes de périphériques intégrés au noyau Linux ou bien sous forme de modules Linux.
- Les applications spécifiques intégrées au système de fichiers *root*.

Nous avons donc le schéma suivant :



Objet connecté sous Linux embarqué

On verra les points suivants :

- Mise en œuvre de Linux embarqué.
- Mise en œuvre du *bootloader u-boot*.
- Développement d'applications sous Linux embarqué : contrôle des E/S, Hello World, application client/serveur.
- Protocole http : mise en œuvre d'un serveur Web embarqué sous Linux embarqué. Application au pilotage par le Web de systèmes embarqués.
- Protocole SMTP : mise en œuvre d'un client SMTP embarqué sous Linux embarqué. Application au pilotage par SMTP de systèmes embarqués.

Pour cette première série de TP autour de Linux embarqué, on privilégiera les lignes de commandes Linux dans un premier temps avant d'utiliser plus tard des *shell scripts* (scripts `goxxx`) qui feront la même chose.

Mots clés : Raspberry Pi, ARM, Linux embarqué, langage C, API sockets, HTTP, Web, SMTP

2. INFORMATIONS ESSENTIELLES SUR LA CARTE CARTE RPI-XENOMAI

La carte Raspberry Pi ou RPi est une carte bon marché largement utilisée pour le DIY (*Do It Yourself*) afin de développer de petits systèmes embarqués ou des objets connectés. Le modèle mis en œuvre dans ces TP est la carte Raspberry Pi 3B.

La carte RPi 3B possède ainsi les éléments suivants :

- Un SoC (*System on Chip*) Broadcom BCM2837 avec un processeur quadricœur ARM Cortex-A53 à 1,2 GHz.
- 1 Go de RAM.
- 4 ports USB.
- Sortie vidéo HDMI.
- Sortie audio HDMI et Jack 3,5 mm.
- Support microSD.
- Ethernet 10/100 Mb/s.
- 17 E/S GPIO, 1 UART, 1 bus I2C et 1 bus SPI.

L'image suivante présente la carte cible RPi 3B :

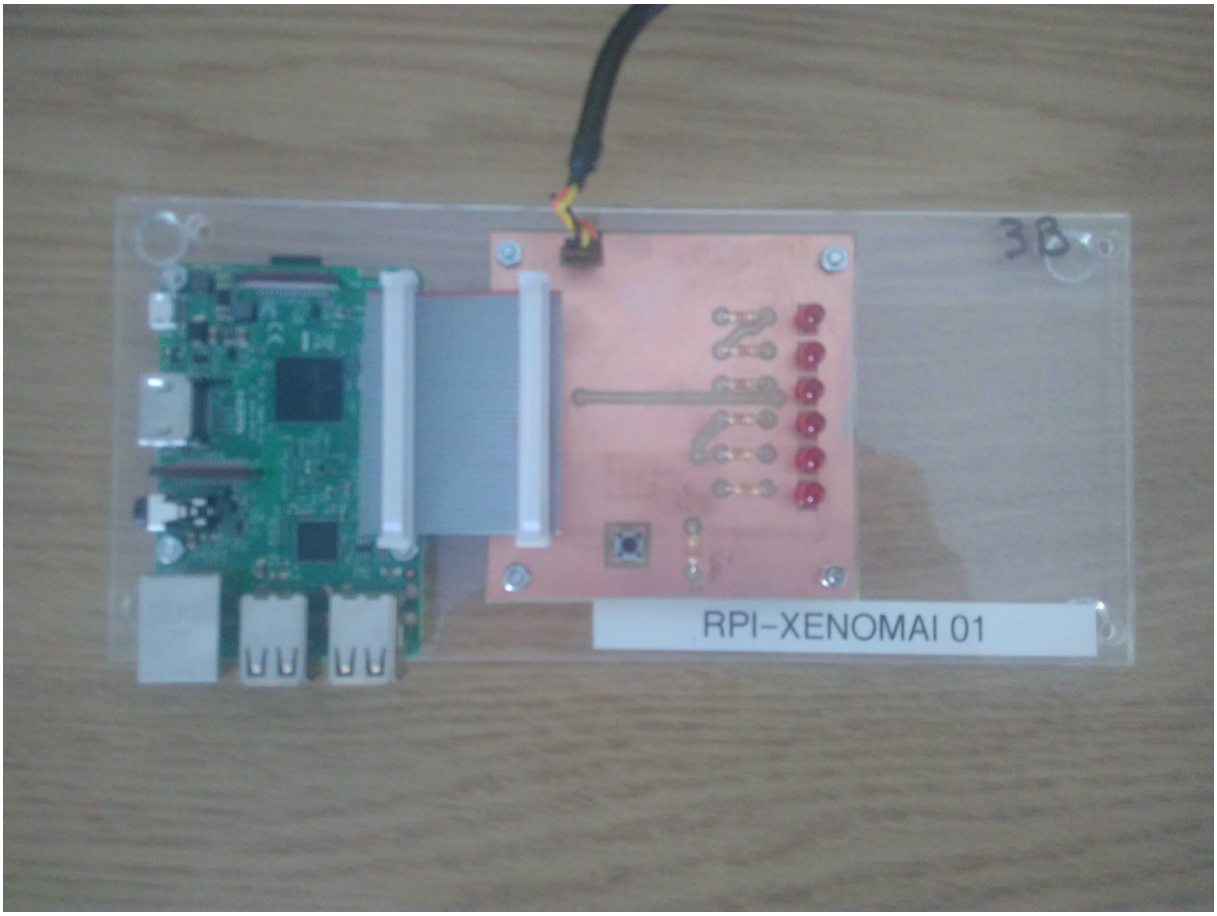
Carte cible RPi 3B

La carte cible rpi-xenomai est une carte « maison » construite autour d'une carte Raspberry Pi 3B. Elle est complétée d'une carte d'entrées/sorties (E/S) connectée à l'aide d'un connecteur 2x20 broches au connecteur d'E/S de la carte RPi.

La carte d'E/S de le carte cible rpi-xenomai possède :
- 6 leds : LED1 à LED6.
- 1 bouton poussoir : BP1.

Le schéma électronique de la carte d'E/S de la carte rpi-xenomai est donné en annexe 1.

La photo est de la carte rpi-xenomai est donnée ci-après.



Carte cible rpi-xenomai

On notera sur le tableau suivant la correspondance entre le numéro de la broche du connecteur 2x20 de la carte RPi et le nom du signal de la carte d'E/S.

Numéro de broche connecteur RPi	Nom du signal carte d'E/S	Fonction
1	V33	3,3 V
2	VCC	5 V
3	SDA	Bus I2C
5	SCL	Bus I2C
6	GND noir	RS232
7	PIN_7	BP1
8	TxD jaune	RS232
10	RxD orange	RS232
11	PIN_11	LED6
12	PIN_12	LED2
16	PIN_16	LED3
18	PIN_18	LED4
22	PIN_22	LED5
25	GND	Masse
26	PIN_26	LED1

Connecteur 2x20 broches et signaux de la carte rpi-xenomai

3. FONCTIONS UTILITAIRES DU BOARD SUPPORT PACKAGE POUR LA CARTE RPI-XENOMAI

Une bibliothèque de fonctions utilitaires a été écrite pour initialiser et utiliser les ressources matérielles de la carte rpi-xenomai. Il s'agit en fait du BSP (*Board Support Package*). Le BSP correspond aux deux fichiers `bsp.c` et `bsp.h`. On notera que le BSP est développé avec la bibliothèque BCM2835 (fichiers `bcm2835.c` et `bcm2835.h`).

Voici la description des fonctions utilitaires et leur prototype :

```
/*
** Fonction: BSP_init()
**              entree(s) : rien
**              sortie(e) : rien
** Description :
** Initialisation ressources et extinction des 6 leds
**/
void  BSP_init(void);


/*
** Fonction: BSP_setLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Allumage d'une led (1 à 6) de la carte rpi-xenomai
**/
void  BSP_setLED(unsigned char lite);


/*
** Fonction: BSP_clrLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Extinction d'une led (1 à 6) de la carte rpi-xenomai
**/
void  BSP_clrLED(usigned char lite);


/*
** Fonction: BSP_release()
**              entree(s) : rien
**              sortie(e) : rien
** Description :
** Libération ressources
**/
void  BSP_release(void);
```

4. TP 0 : PRISE EN MAIN

- Démarrer le PC sous Linux. Se connecter sous le nom **se01**, mot de passe : **se01** ☺ pour le groupe 1 et sous le nom **se02**, mot de passe : **se02** ☺ pour le groupe 2.

- Se placer dans son répertoire de travail :
  `host% cd`

- Etablir le schéma de l'environnement de développement :
  - Matériels.
  - Liaisons : série, réseau…
  - Logiciels et OS utilisés.
  - Adresses IP du PC de développement (hôte ou *host*) et de la carte cible (cible ou *target*).

- Se connecter à la carte d'évaluation (cible) en utilisant l'outil `minicom` :
  `host% minicom -b 115200 -D /dev/ttyUSB0`
  Pour sortir de `minicom`, il suffit de taper la combinaison de touches : CTRL A, Z pour accéder au menu et taper q pour quitter. On arrêtera le compte à rebours de 3 secondes d'*u-boot* en appuyant sur la touche espace du clavier…

- A quoi sert la macro `gok` ? Quel fichier est téléchargé et à quelle adresse ?
  `U-Boot> print gok`

- A quoi sert la macro `gor` ? Quel fichier est téléchargé et à quelle adresse ?
  `U-Boot> print gor`

- A quoi sert la macro `godtb` ? Quel fichier est téléchargé et à quelle adresse ?
  `U-Boot> print godtb`

- A quoi sert la macro `go` ?
  `U-Boot> print go`

- A quoi sert la macro `ramboot` ?
  `U-Boot> print ramboot`

Par la suite, on adoptera les conventions suivantes :
Commande Linux PC hôte :

```
host% commande Linux
```

Commande Linux carte cible :

```
RPi3# commande Linux
```

Commande *u-boot* :

```
U-Boot> commande u-boot
```

Astuce !

Pour éviter de regénérer à chaque fois le système de fichiers *root*, on réalisera une compilation croisée de son application puis on recopiera sous `/tftpboot/` l'exécutable ainsi produit :

```
host% make
host% cp mon_appli /tftpboot
```

On pourra alors télécharger l'exécutable en utilisant la commande `tftp` et lancer l'application :

```
RPi3# tftp -g -r mon_appli @IP_host
RPi3# chmod u+x mon_appli
RPi3# ./mon_appli
```

NB :

Il faudra au préalable configurer l'interface Ethernet de la carte cible RPi avec la commande Linux `ifconfig` suivant l'annexe 2.

5. TP 1 : GENERATION DU RAM *DISK*

Nous allons dans un premier temps créer un *RAM disk*. Un *RAM disk* est un système de fichiers *root* en mémoire RAM. Il est donc volatile et disparaît au *reboot* ou à l'arrêt de la carte cible.

- Se placer dans son répertoire de travail :
  `host% cd`

- Dans son répertoire à son nom, recopier le fichier `tp-linux.tgz` sous `~kadionik/` :
  `host% cp /home/kadionik/tp-linux.tgz .`

- Décompresser et installer le fichier `tp-linux.tgz` :
  `host% tar -xvzf tp-linux.tgz`

- Se placer ensuite dans le répertoire `tp-linux/`. **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**
  `host% cd tp-linux`

- Créer le système de fichiers *root* squelette `root_fs` pour la carte cible RPi. Que fait le *shell script* `goskel` ?
  `host% cd ramdisk`
  `host% ./goskel`

- Compiler `busybox`. A quoi correspondent les variables `ARCH` et `CROSS_COMPILE` ?
  `host% cd ramdisk/busybox`
  `host%` **`Question Q1`**

- Installer `busybox` dans le système de fichiers *root* `root_fs` pour la carte cible RPi :
  `host% cd ramdisk/busybox`
  `host%` **`Question Q2`**

- Intégrer les bibliothèques dynamiques dans le système de fichiers *root* `root_fs` pour la carte cible RPi. Que fait le *shell script* `golib` ?
  `host% cd ramdisk`
  `host% ./golib`

- Générer le système de fichiers *root* final `root_fs` pour la carte cible RPi. Il est demandé de rentrer son mot de passe. Que fait le *shell script* `goramdisk` ? Quel fichier est généré ? A quoi correspond-il ?
  `host% cd ramdisk`
  `host% sudo ./goramdisk`

- Copier le nouveau *RAM disk* dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`. Pourquoi utilise-t-on le caractère \ avant la commande *cp* ?
  `host% \cp ramdisk.img.gz /tftpboot`

6. TP 2 : COMPILATION DU NOYAU LINUX ET BOOT

Nous allons voir comment compiler le noyau Linux embarqué exécuté par le processeur ARM de la carte cible RPi.

- Compiler le noyau Linux standard pour la carte cible RPi. Quelle est la version du noyau ?
  `host% cd linux`
  `host% **Question Q3**`

- Installer le fichier du noyau Linux dans le répertoire de téléchargement d'*u-boot* `/tftpboot/` :
  `host% \cp arch/arm/boot/zImage /tftpboot`

- Depuis *u-boot* de la carte cible RPi, lancer la commande suivante. Quels sont les fichiers téléchargés depuis le PC hôte en RAM de la carte RPi et quel est leur rôle ?
  `U-Boot> run ramboot`

- Observer les traces de boot du noyau Linux dans la fenêtre `minicom` :

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 6.11.0-v7 (kadionik@ipcchipik) (arm-buildroot-linux-gnueabihf-gcc.br_real
(Buildroot 2021.11-4428-g6b6741b) 11.3.0, GNU ld (GNU Binutils) 2.38) #2 SMP Tue Sep 24
15:26:54 CEST 2024
CPU: ARMv7 Processor [410fd034] revision 4 (ARMv7), cr=10c5383d
CPU: div instructions available: patching division code
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: Raspberry Pi 3 Model B
Memory policy: Data cache writealloc
Reserved memory: created CMA memory pool at 0x37400000, size 64 MiB
OF: reserved mem: initialized node linux,cma, compatible id shared-dma-pool
percpu: Embedded 19 pages/cpu s47488 r8192 d22144 u77824
Built 1 zonelists, mobility grouping on. Total pages: 240555
Kernel command line: earlyprintk dwc_otg.lpm_enable=0 console=ttyAMA0,115200 console=tty1
root=/dev/ram ramdisk_size=131072 rootfstype=ext4 rootwait rw
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes, linear)
. . .
sched_clock: 32 bits at 1000kHz, resolution 1000ns, wraps every 2147483647500ns
clocksource: timer: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 191126
0446275 ns
bcm2835: system timer (irq = 27)
arch_timer: cp15 timer(s) running at 19.20MHz (phys).
clocksource: arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x46d987e47, m
ax_idle_ns: 440795202767 ns
sched_clock: 56 bits at 19MHz, resolution 52ns, wraps every 4398046511078ns
Switching to timer-based delay loop, resolution 52ns
Console: colour dummy device 80x30
printk: console [tty1] enabled
Calibrating delay loop (skipped), value calculated using timer frequency.. 38.40 BogoMIPS
(lpj=192000)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 2048 (order: 1, 8192 bytes, linear)
Mountpoint-cache hash table entries: 2048 (order: 1, 8192 bytes, linear)
Disabling memory control group subsystem
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x100000 - 0x10003c
rcu: Hierarchical SRCU implementation.
smp: Bringing up secondary CPUs ...
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
CPU2: thread -1, cpu 2, socket 0, mpidr 80000002
CPU3: thread -1, cpu 3, socket 0, mpidr 80000003
smp: Brought up 1 node, 4 CPUs
SMP: Total of 4 processors activated (153.60 BogoMIPS).
. . .
bcm2835-mbox 3f00b880.mailbox: mailbox enabled
raspberrypi-firmware soc:firmware: Request 0x00000002 returned status 0x80000001
```

```
. . .
bcm2835-rng 3f104000.rng: hwrng registered
brd: module loaded
loop: module loaded
bcm2835-power bcm2835-power: Broadcom BCM2835 power domains driver
Loading iSCSI transport class v2.0-870.
libphy: Fixed MDIO Bus: probed
usbcore: registered new interface driver lan78xx
usbcore: registered new interface driver smsc95xx
dwc_otg: version 3.00a 10-AUG-2012 (platform bus)
dwc2 3f980000.usb: supply vusb_d not found, using dummy regulator
dwc2 3f980000.usb: supply vusb_a not found, using dummy regulator
dwc2 3f980000.usb: DWC OTG Controller
dwc2 3f980000.usb: new USB bus registered, assigned bus number 1
dwc2 3f980000.usb: irq 33, io mem 0x3f980000
usb usb1: New USB device found, idVendor=1d6b, idProduct=0002, bcdDevice= 5.07
usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=1
usb usb1: Product: DWC OTG Controller
usb usb1: Manufacturer: Linux 5.7.19-v7+ dwc2_hsotg
usb usb1: SerialNumber: 3f980000.usb
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
usbcore: registered new interface driver usb-storage
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
i2c-bcm2835 3f805000.i2c: Could not read clock-frequency property
bcm2835-wdt bcm2835-wdt: Broadcom BCM2835 watchdog timer
bcm2835-cpufreq: min=600000 max=1200000
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhost-bcm2835 3f202000.mmc: /aliases ID not available
sdhost: log_buf @ (ptrval) (f7441000)
mmc0: sdhost-bcm2835 loaded - DMA enabled (>1)
sdhci-pltfm: SDHCI platform and OF driver helper
ledtrig-cpu: registered to indicate activity on CPUs
hid: raw HID events driver (C) Jiri Kosina
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
Initializing XFRM netlink socket
NET: Registered protocol family 17
Key type dns_resolver registered
Registering SWP/SWPB emulation handler
registered taskstats version 1
Loading compiled-in X.509 certificates
3f201000.serial: ttyAMA0 at MMIO 0x3f201000 (irq = 81, base_baud = 0) is a PL011
 rev2
mmc0: host does not support reading read-only switch, assuming write-enable
printk: console [ttyAMA0] enabled
mmc0: new high speed SDHC card at address 59b4
of_cfs_init
mmcblk0: mmc0:59b4 USD   7.47 GiB
of_cfs_init: OK
 mmcblk0: p1 p2
RAMDISK: gzip image found at block 0
usb 1-1: new high-speed USB device number 2 using dwc2
EXT4-fs (ram0): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
usb 1-1: New USB device found, idVendor=0424, idProduct=9514, bcdDevice= 2.00
usb 1-1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
Run /sbin/init as init process
hub 1-1:1.0: USB hub found
hub 1-1:1.0: 5 ports detected
EXT4-fs (ram0): re-mounted. Opts: (null)
uart-pl011 3f201000.serial: no DMA platform data
random: fast init done
usb 1-1.1: new high-speed USB device number 3 using dwc2
usb 1-1.1: New USB device found, idVendor=0424, idProduct=ec00, bcdDevice= 2.00
usb 1-1.1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
smsc95xx v1.0.6
smsc95xx 1-1.1:1.0 eth0: register 'smsc95xx' at usb-3f980000.usb-1.1, smsc95xx U
SB 2.0 Ethernet, b8:27:eb:f0:98:aa
random: crng init done

Please press Enter to activate this console.
RPi3:/# uname -r
5.7.19-v7+
RPi3:/#
```

7. EX 1 : APPLICATION HELLO WORLD

Nous allons faire une compilation croisée de la célèbre application « *Hello World!* ».

- Se placer dans le répertoire `tst/hello/` et modifier le fichier `hello.c` afin de créer le fameux « *Hello World!* » :
  `host% cd tst/hello`

- Le contenu du fichier `Makefile` est le suivant. Expliquer son fonctionnement. Pour quel type de compilation est-il configuré ?

```
#CC=gcc
CC= arm-buildroot-linux-gnueabihf-gcc
CFLAGS = -c
OBJS = hello.o

main : $(OBJS)
   $(CC) $(LDFLAGS) $(OBJS) -o hello

%.o: %.c
   $(CC) $(CFLAGS) $<

clean:
   rm -f *.o *~ hello
```

- Compiler l'application `hello` pour la carte cible RPi :
  `host% make`

- Recopier l'application `hello` dans le répertoire de téléchargement d'*u-boot* `/tftpboot/` :
  `host% \cp hello /tftpboot`

- Télécharger par *tftp* l'application `hello` dans le système de fichiers *root* de la carte cible RPi en utilisant l'astuce de la page 10 et tester.

Par la suite, on appliquera cette méthodologie pour toute autre application compilée de façon croisée…

8. EX 2 : CHENILLARD

Nous allons créer un chenillard sur les leds 1 à 6 en utilisant le BSP de la carte RPi décrit au chapitre 3.

- Se placer dans le répertoire `tst/chenillard/` et modifier le fichier `chenillard.c` afin de créer un chenillard à la « K2000 ».
  ```
  host% cd tst/chenillard
  ```

- Créer le fichier `Makefile` à partir de celui de l'exercice précédent `hello`.

- Compiler l'application `chenillard` pour la carte cible RPi :
  ```
  host% make
  ```

- Recopier l'application `chenillard` dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`.

- Télécharger par *tftp* l'application `chenillard` dans le système de fichiers *root* de la carte cible RPi et tester.

Indications :

On n'oubliera pas d'utiliser la fonction d'initialisation du BSP `BSP_init()` et de libération des ressources `BSP_release()`.

9. EX 3 : PROTOCOLE HTTP. SERVEUR WEB BOA

Nous allons voir la mise en œuvre d'un serveur Web embarqué appelé `boa` sur la carte RPi qui permettra de contrôler à distance la carte cible. Le protocole utilisé pour ce contrôle est HTTP couplé à l'interface CGI (*Common Gateway Interface*) qui permet de faire exécuter un script par le serveur Web. Le but de l'exercice est de piloter par le Web les 6 leds globalement de la carte cible.

- Se placer dans le répertoire `tst/boa/src/`. Compiler `boa`. Que fait le *shell script* `go` ?
```
host% cd tst/boa/src
host% ./go
```

- Installer `boa` dans le système de fichiers *root* qui servira de base au *RAM disk* :
```
host% cd tst/boa/src
host% ./goinstall
```

- Installer les fichiers de configuration de `boa` dans le système de fichiers *root*. Que fait le *shell script* `goinstall` ?
```
host% cd tst/boa
host% ./goinstall
```

- Regénérer le *RAM disk* et le recopier dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`.

- Relancer le noyau Linux embarqué :
```
RPi3# reboot
```

- Configurer l'interface réseau `eth0`. On se réfèrera à l'annexe 2 pour les valeurs d'adresse IP à utiliser.
  A titre d'exemple, la configuration réseau de la carte cible `rpi001` sera :
```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

- On lancera le serveur Web `boa` :
```
RPi3# boa -c /etc
```

- On vérifiera que le serveur Web `boa` est bien actif :
```
RPi3# ps
```

Dans un script CGI, les E/S sont redirigées (annexe 4). Il convient simplement d'utiliser des appels système `printf()` dans le source d'un script CGI écrit en langage C pour générer l'entête HTTP et les données de la réponse http renvoyée vers le navigateur Web.

S'il s'agit d'un *shell script*, on peut utiliser alors la commande `echo`. Une requête CGI est de la forme `http://@IP_cible:port/cgi-bin/mon_cgi?param1`.

Le script CGI a accès à des variables d'environnement et la variable d'environnement `QUERY_STRING` contient le paramètre de la requête CGI envoyée par le navigateur.

- La configuration réseau étant préalablement réalisée, à l'aide d'un navigateur Web (*Firefox*), exécuter le script CGI `printenv` en forgeant l'URL adéquate. On observera l'évolution de la variable d'environnement `QUERY_STRING` :
  ```
  http://@IP_cible/cgi-bin/printenv
  http://@IP_cible/cgi-bin/printenv?1
  http://@IP_cible/cgi-bin/printenv?0
  ```

- Le fichier source CGI pour le pilotage des leds de la cible est le fichier C `putleds.c` situé sous dans le répertoire `tst/boa/`. Modifier ce fichier pour traiter les commandes CGI envoyées par le navigateur Web. Une requête CGI est de la forme `http://@IP_cible:port/cgi-bin/mon_cgi?param`. Le script CGI a accès à des variables d'environnement CGI et la variable d'environnement `QUERY_STRING` contient le paramètre de la requête envoyée par le navigateur. Modifier le fichier `putleds.c` en utilisant les fonctions du BSP pour contrôler l'allumage (`param=1`) ou l'extinction (`param=0`) des 6 leds de la carte cible. On pourra récupérer la valeur courante de la variable d'environnement `QUERY_STRING` grâce à l'appel système C `getenv()` :
  ```
  char *param;
  param = getenv("QUERY_STRING");
  ```

- Créer le fichier `Makefile` pour la compilation de l'application `putleds`.

- Compiler l'application `putleds` pour la carte cible RPi.

- Recopier l'application `putleds` dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`.

- Télécharger par *tftp* l'application `putleds` dans le répertoire `/home/httpd/cgi-bin/` du système de fichiers *root* de la carte cible RPi et tester le script CGI `putleds` depuis le navigateur Internet pour contrôler alors l'allumage et l'extinction des 6 leds de la carte cible en forgeant l'URL adéquate depuis le navigateur Web.

10. EX 4 : DEVELOPPEMENT D'UNE APPLICATION CLIENT/SERVEUR

On désire mettre en œuvre une application client/serveur TCP/IP pour le contrôle des leds de la carte cible RPi. Ce TP met ici en valeur la connectivité IP à l'aide d'une interface de contrôle propriétaire (la sienne) d'un système embarqué.

On doit créer une application `myserver` que l'on intègrera dans le système de fichiers *root* de la carte cible RPi. On développera un serveur TCP qui acceptera les requêtes envoyées par un client TCP sous Linux du PC hôte dans la *socket* de connexion.

La requête envoyée par le client comportera 2 octets :

Octet 1 : numéro de la led à contrôler (entre 1 et 6)

Octet 2 : valeur de la led (0 : led éteinte, 1 : led allumée)

Le serveur enverra en retour un octet comme code de retour :

0 si OK et 1 si ERREUR

Le canevas en langage C d'un serveur TCP itératif est le suivant (les **???** sont à remplacer par les bonnes valeurs…) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

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

{
  int sd;
  struct sockaddr_in sa;              /* Structure Internet sockaddr_in */
  struct hostent *hptr ;      /* Infos sur le serveur */
  int port;                   /* Numero de port du serveur */

  int newsd;                  /* Id de la socket entrante */
  struct sockaddr_in newsa;   /* sockaddr_in de la connection entrante */
  int newsalength;
  struct hostent *newhptr;


  if (argc != 2) {
    printf("Syntaxe : %% myserver numero_port\n");
    exit(1);
  }

  /* Recuperation numero port passe en argument */
  port = atoi(argv[1]);

  /* Famille d'adresse : AF_INET = PF_INET */
  sa.sin_family = AF_INET;

  /* Initialisation du numero du port */
  sa.sin_port = htons(port);
  sa.sin_addr.s_addr = INADDR_ANY;

  if((sd = socket(AF_???, SOCK_???, 0)) < 0) {
    printf("Probleme lors de la creation de socket\n");
    exit(1);
  }
```

```
  if(bind(sd, (struct sockaddr *) &sa, sizeof(sa)) == -1) {
    printf("Probleme avec le bind\n");
    exit(1);
  }

  listen(sd, 1);

  while(1) {
    newsalength = sizeof(newsa);
    if((newsd = accept(sd, (struct sockaddr *) &newsa, &newsalength)) < 0 ) {
      printf("Erreur sur accept\n");
      exit(1);
    }

    . . .

    close(newsd);
  }

  close(sd);
  exit(0);
}
```

- Se placer dans le répertoire `tst/myserver/` et modifier le fichier `myserver.c` afin de créer le serveur TCP.
  `host% cd tst/myserver`

- Créer le fichier `Makefile` pour la compilation de l'application `myserver`.

- Compiler l'application `myserver` pour la carte cible RPi.

- Recopier l'application `myserver` dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`.

- Télécharger par *tftp* l'application `myserver` dans le système de fichiers *root* de la carte cible RPi et lancer le serveur sur le numéro de port 2000.

On doit créer une application cliente `myclient.c` qui réalisera un chenillard. Il suffira d'envoyer successivement les ordres pour allumer ou éteindre les leds correspondantes.

Le canevas en langage C d'un client TCP est le suivant (les **???** sont à remplacer par les bonnes valeurs…) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

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

{
  int sd;
  struct sockaddr_in sa;            /* Structure Internet sockaddr_in */
  struct hostent *hptr ;            /* Infos sur le serveur */

  char *serveur ;                   /* Nom du serveur distant */
  int port;
```

```
  /* verification du nombre d'arguments de la ligne de commande */
  if (argc != 3) {
    printf("myclientP. Erreur d'arguments\n");
    printf("Syntaxe : %% myclient nom_serveur numero_port\n");
    exit(1);
  }

  /* Recuperation nom du serveur */
  serveur = argv[1];

  /* Recuperation numero de port */
  port = atoi(argv[2]);

  /* Recuperation des infos sur le serveur dans /etc/hosts pour par DNS */
  if((hptr = gethostbyname(serveur)) == NULL) {
    printf("Probleme de recuperation d'infos sur le serveur\n");
    exit(1);
  }

  /* Initialisation la structure sockaddr sa avec les infos  formattees : */
  /* bcopy(void *source, void *destination, size_t taille);               */
  bcopy((char *)hptr->h_addr, (char*)&sa.sin_addr, hptr->h_length);

  /* Famille d'adresse : AF_INET ici */
  sa.sin_family = hptr->h_addrtype;


  /* Initialisation du numero du port */
  sa.sin_port = htons(port);

  sd = socket(AF_???, ???, 0);
  connect(sd, (struct sockaddr *) &sa, sizeof(sa));
  . . .
  close(sd);
  exit(0);
}
```

- Se placer dans le répertoire `tst/myclient/` et modifier le fichier `myclient.c` afin de créer le client TCP.
  `host% cd tst/myclient`

- Créer le fichier `Makefile` pour la compilation de l'application `myclient`.

- Compiler l'application `myclient` pour le PC hôte. Quel est le type de compilation ?

- Lancer le client sur le numéro de port 2000 et tester le bon fonctionnement du chenillard.

11. EX 5 : MISE EN ŒUVRE D'UN CLIENT SMTP SOUS LINUX

On va dans un premier temps tester le serveur de mails SMTP du PC hôte sous Linux et comprendre le dialogue entre un client et un serveur SMTP et constater que :

- Il est du type commande/réponse.
- Il est orienté flux d'octets structurés sous forme de chaînes de caractères ASCII.

- Se connecter par *telnet* au service mail du PC hôte. Les commandes envoyées au serveur de mail (protocole SMTP *Simple Mail Transfer Protocol*, RFC821) sont structurées sous forme de lignes de commandes ASCII. Un exemple d'échanges de commandes SMTP est proposé ci-après pour l'utilisateur `se01` (C: commande à taper, R: réponse du serveur SMTP) :

```
R: 220 linux01.localdomain ESMTP Postfix

C: MAIL FROM:<se01@localhost>
R: 250 2.1.0 Ok

C: RCPT TO:<se01@localhost>
R: 250 2.1.5 Ok

C: DATA
R: 354 End data with <CR><LF>.<CR><LF>
un test
.

R: 250 2.0.0 Ok: queued as E3382A50439

C: QUIT
R: 221 2.0.0 Bye
```

- En s'aidant de l'exemple, envoyer manuellement un mail à l'utilisateur `se01` (ou `se02`) :
  `host% telnet localhost 25`

- Vérifier sa bonne réception en utilisant la commande de lecture de mail `mail` :
  `host% mail`

Pour développer une application cliente SMTP sous Linux, il convient donc de développer une application réseau cliente comme dans le TP précédent en formatant le flux de données échangées sous forme de chaînes de caractères ASCII respectant le protocole SMTP.

Pour cela, on va se simplifier la vie et partir d'un client SMTP existant issu des sources du livre de T. Jones cité en référence.

- Se placer dans le répertoire `tst/tstsmtp/` :
  `host% cd tst/tstsmtp`

- Ce répertoire contient un fichier principal `main.c`, un fichier bibliothèque de fonctions `emstp.c` et son fichier .h associé ainsi qu'un fichier `Makefile`. Par analyse du code source, modifier les fichiers C pour envoyer un mail au format **texte** et au format **HTML** à l'utilisateur `guest` du PC hôte.

- Compiler et tester le client Linux SMTP `mysmtp`.

12. EX 6 : MISE EN ŒUVRE D'UN CLIENT SMTP SOUS LINUX EMBARQUE

Il convient maintenant de transformer l'exemple précédent en application Linux embarqué.

- Se placer dans le répertoire `tst/mysmtp/` :
  `host% cd tst/mysmtp`

- On ajustera au préalable le fichier `/etc/hosts` du système de fichiers *root* de la cible pour ajouter une entrée correspondant à l'adresse IP du PC hôte suivant l'annexe 2 :
  `target# echo "@IP_host linux01" >> /etc/hosts`

- Ce répertoire contient un fichier principal `main.c`, un fichier bibliothèque de fonctions `emstp.c` et son fichier .h associé ainsi qu'un fichier `Makefile`. Par analyse du code source, modifier les fichiers C pour envoyer un mail au format **texte** et au format **HTML** à l'utilisateur `se01` (ou `se02`) du PC hôte nommé `linux01` depuis la carte cible RPi.

- Quelle est l'adresse mail de l'expéditeur ? Quelle est l'adresse mail du destinataire ?

- Compiler l'application `mysmtp` pour la carte cible RPi.

- Recopier l'application `mysmtp` dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`.

- Télécharger par *tftp* l'application `mysmtp` dans le système de fichiers *root* de la carte cible RPi et tester.

- Tester l'application `mysmtp` de la cible.

- Vérifier la bonne réception de l'email en utilisant la commande `mail` :
  `host% mail`

- Que faut-il penser de cette méthode de connectivité IP ? Dans quel type d'échanges peut-on utiliser cette méthode ?

13. CONCLUSION

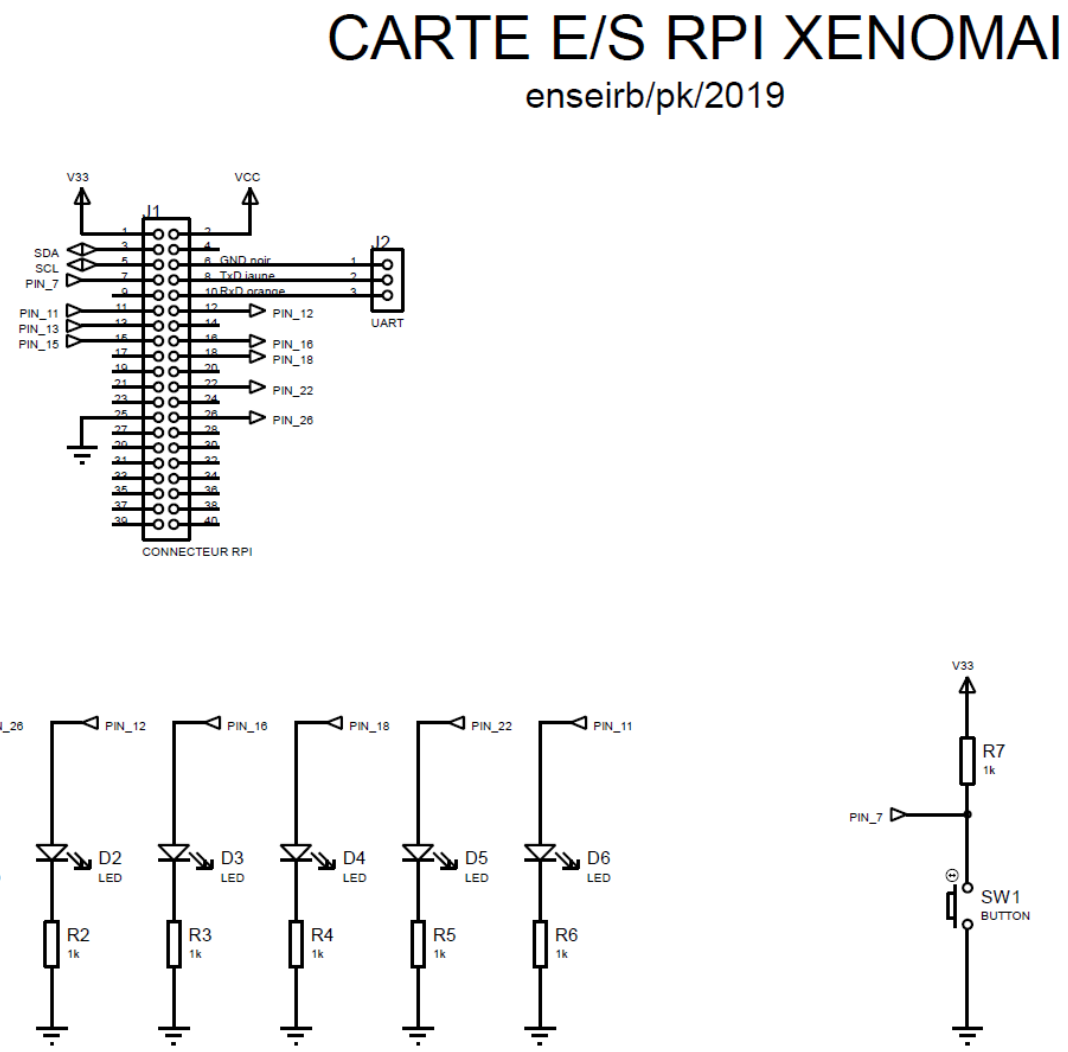
Nous avons pu voir la mise en œuvre de Linux embarqué sur la carte RPi.

Nous avons construit à la main une distribution Linux embarqué correspondant à notre besoin et développé différentes applications de contrôle local ou à distance des E/S de la carte RPi.

14. REFERENCES

- Carte Raspberry Pi : https://www.raspberrypi.org/
- TCP/IP Application Layer Protocols For Embedded Systems. T. Jones. Editions Charles River Media. 2002

15. ANNEXE 1 : SCHEMA ELECTRONIQUE DE LA CARTE D'E/S DE LA CARTE CIBLE RPI-XENOMAI



16. ANNEXE 2 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	rodirula	10.7.4.223
CIBLE	rpi01	10.7.2.118

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	sarracenia	10.7.4.225
CIBLE	rpi02	10.7.2.119

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	utricularia	10.7.4.227
CIBLE	rpi03	10.7.2.120

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	ibicella	10.7.2.112
CIBLE	rpi04	10.7.2.121

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	nepenthes	10.7.2.114
CIBLE	rpi05	10.7.2.122

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	pinguicula	10.7.2.116
CIBLE	rpi06	10.7.2.123

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `rpi001` :

```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

	POSTE PC07	
	NOM	**ADRESSE IP**
HOTE	genlisea	10.7.1.245
CIBLE	rpi07	10.7.2.124

	POSTE PC08	
	NOM	**ADRESSE IP**
HOTE	heliamphoria	10.7.2.110
CIBLE	rpi08	10.7.2.125

	POSTE PC09	
	NOM	**ADRESSE IP**
HOTE	darlingtonia	10.7.1.239
CIBLE	rpi09	10.7.7.57

	POSTE PC10	
	NOM	**ADRESSE IP**
HOTE	dionaca	10.7.1.241
CIBLE	rpi10	10.7.7.58

	POSTE PC11	
	NOM	**ADRESSE IP**
HOTE	drosera	10.7.1.243
CIBLE	rpi11	10.7.7.59

	POSTE PC12	
	NOM	**ADRESSE IP**
HOTE	catopsis	10.7.1.40
CIBLE	rpi12	10.7.7.60

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `rpi001` :

```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

ANNEXE 3 : REPONSES AUX QUESTIONS

R1 :

```
host% make ARCH=arm CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
```

R2 :

```
host% make ARCH=arm CROSS_COMPILE= arm-buildroot-linux-gnueabihf- CONFIG_PREFIX=../root_fs install
```

R3 :

```
host% make ARCH=arm CROSS_COMPILE= arm-buildroot-linux-gnueabihf- zImage
```