IV.3 – Personnaliser le support du matériel

Christophe BLAESS - janvier 2020

La démarche pour la création d'un système embarqué est généralement la suivante :

- **Prototypage et expérimentations sur un SBC** (*Single Board Computer*) du type Raspberry Pi, Beaglebone, Odroid… Durant cette étape on utilise généralement une distribution (Debian, Raspbian, Arch Linux, etc.) pour valider la faisabilité et l'intérêt du projet le plus rapidement possible.
- **Première implémentation en utilisant le kit de développement du SOM** (*System On Module*) que l'on a choisi pour continuer le projet. Il existe un nombre assez conséquent de SOM — aujourd'hui souvent architecturés autour d'un processeur i.MX6 — et le choix doit se faire en fonction de différents paramètres (adéquation aux besoins du projet, coût, pérennité, disponibilité en nombre, support de Linux, etc.). Pour cette étape, on commencera à utiliser Yocto ou Buildroot pour assurer le support du matériel. La connexion aux périphériques spécifiques du projet est souvent faite en «fils volants».
- Production de la **première série avec une carte porteuse spécifique** au projet comprenant tous les périphériques spécifiques (capteurs, actionneurs, etc.) et intégrant le module SOM. Il est parfois nécessaire de faire évoluer la carte porteuse pendant quelques itérations avant la production en nombre.
- Dans le cas où l'on envisage une production suffisamment importante (plus de dix mille unités), on peut envisager de se passer de module et d'intégrer directement le processeur — ou plus précisément le SOC (*System On Chip*) — sur la carte métier.

L'utilisation de Yocto, telle que nous l'avons vue jusqu'ici, permettra de mener à bien la seconde phase du projet, c'est-à-dire l'implémentation sur le kit de développement

du module SOM. En effet la plupart des fabricants de modules proposent un *layer* pour Yocto regroupant les recettes nécessaires au fonctionnement sur leur kit de développement.

Nous allons nous intéresser dans cette séquence à l'étape ultérieure : le support de périphériques non intégrés dans le kit de développement (par exemple des convertisseurs analogiques-numériques, des capteurs de température, de lumière, de présence…).

Pour supporter un périphérique non présent dans la configuration du kit de développement du SOM, deux étapes sont nécessaires :

- configurer le noyau Linux pour qu'il intègre le pilote (*driver*) du périphérique en question,
- configurer le *device tree* pour que le noyau sache comment communiquer avec ce périphérique.

Configuration du noyau

Le noyau (*kernel*) Linux assure la mise à disposition des ressources matérielles pour les applications. C'est lui qui contient les pilotes (*drivers*) de périphériques qui communiquent avec les équipements autour du processeur. Mais il contient également beaucoup d'autres éléments : systèmes de fichiers, protocoles réseau, gestion de la mémoire, mécanisme d'ordonnancement…

Le noyau comporte des milliers d'options de configuration, que nous pouvons ajuster par l'intermédiaire d'un menu. Les options que l'on modifie le plus souvent en préparant un système embarqué concernent des *drivers* qu'il faut intégrer dans le noyau. Malheureusement pour notre expérimentation, l'essentiel des *drivers* susceptibles de nous intéresser sont déjà inclus dans la configuration du noyau fournie par le *layer* pour Raspberry Pi.

Nous allons donc modifier des options qui ne sont pas directement intéressantes pour nous, mais qui ont des effets visibles depuis la ligne de commande :

- Une première option sera la «préemptibilité» du noyau. Derrière ce mot un peu étrange, se cache la possibilité de suspendre un traitement dans le *kernel* pour activer une tâche de l'espace utilisateur. Ainsi le système s'avère plus réactif aux événements extérieurs. C'est une option qui concerne généralement les systèmes soumis à des contraintes temps réel. Lorsqu'un noyau est préemptible, cela est mentionné par un *flag* dans le résultat de la commande «`uname -a`».
- La seconde option sera un déclencheur (*trigger*) pour provoquer l'allumage de la led verte du Raspberry Pi. Par défaut celle-ci s'allume lors des accès à la

carte micro-SD. Mais on peut sélectionner un déclencheur différent, par exemple un *timer*, un battement de cœur (*hearbeat*), etc. Un type de déclenchement est possible, qui n'est pas compilé par défaut avec le noyau du Raspberry Pi : «*activity*». La led dans ce cas clignote avec une fréquence dépendant de la charge instantanée du système.

Nous allons tout d'abord vérifier que ces options sont bien absentes d'un noyau compilé avec les options par défaut :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyS0

mybox login: root
Password: (linux)
root@mybox:~# uname  -a
Linux mybox 4.19.75 #1 SMP Thu Dec 26 11:13:52 UTC 2019 armv7l
```

Pas de mot-clé «PREEMPT» dans le résultat de `uname`. Vérifions les *triggers* disponibles pour la led :

```
root@mybox:~# cat  /sys/class/leds/led0/trigger
none rc-feedback kbd-scrolllock kbd-numlock kbd-capslock kbd-k
lock kbd-altlock kbd-shiftllock kbd-shiftrlock kbd-ctrllock k
ht gpio cpu cpu0 cpu1 cpu2 cpu3 default-on input panic mmc1 [m
root@mybox:~#
```

Le *trigger* sélectionné est encadré par des crochets ; c'est «`mmc0`». Dans la liste des *triggers* disponibles, aucune trace de «`activity`».

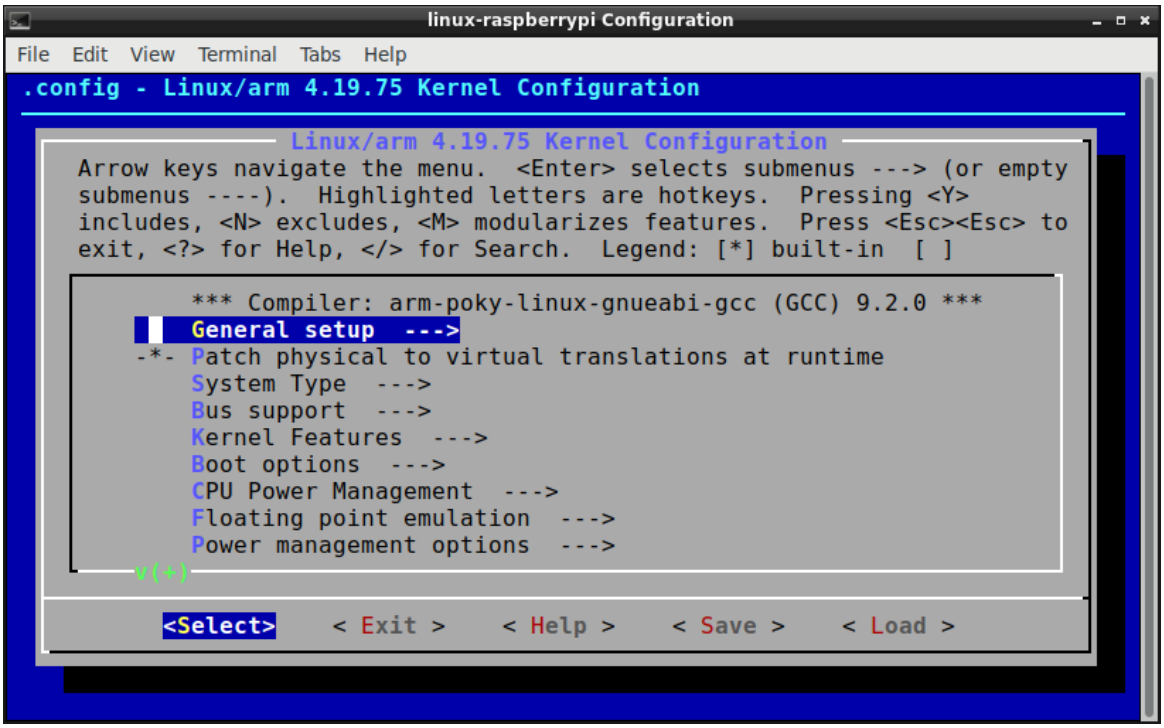
La configuration du *kernel* ressemble un peu à celle de Busybox que nous avons vue dans <u>la séquence précédente</u> (plus exactement Busybox utilise le mécanisme de configuration mis au point pour le noyau Linux). Nous appelons `bitbake` avec la même option «`-c menuconfig`» que précédemment. Toutefois le nom de la recette est particulier : comme nous ne connaissons pas le nom exact de la recette gérant le *kernel*, et que cela peut varier en fonction de la machine cible, de la version de Yocto, etc. on utilise un nom générique «`virtual/kernel`» qui sera traduit par `bitbake` :

```
[build-rpi]$ bitbake  -c menuconfig  virtual/kernel
```

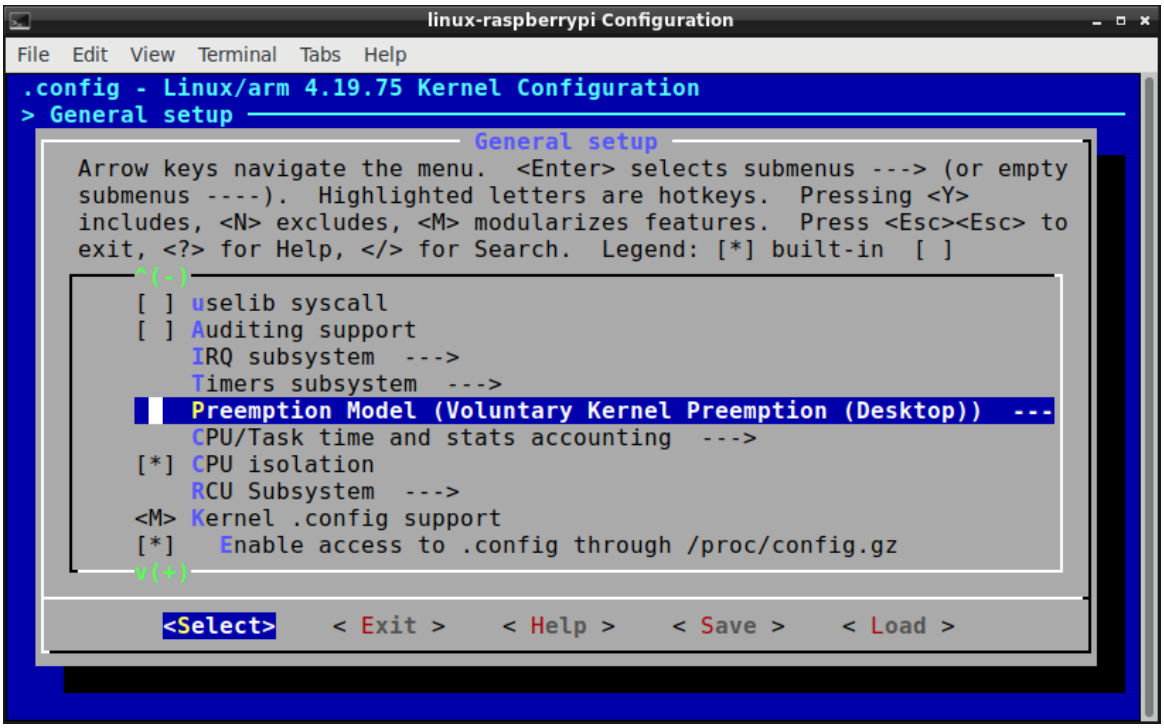
Une nouvelle fenêtre (ou un nouvel onglet de terminal) s'ouvre. Je vous encourage à parcourir quelques menus pour vous familiariser avec la configuration du *kernel* Linux, notamment le menu «*Device Drivers*». Pour avoir un peu d'explication sur une option, on peut appuyer sur «?».

Deux options ont retenu notre attention.

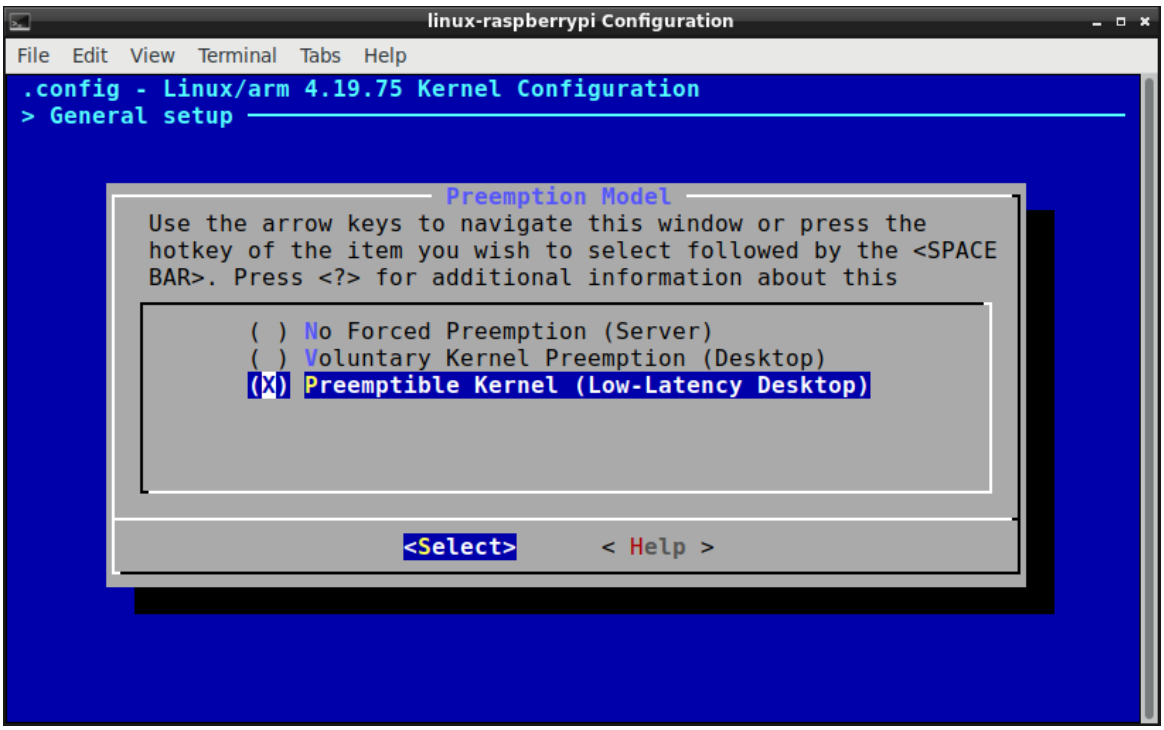
La première est dans le menu «*General Setup*» (figure IV.4-1)…



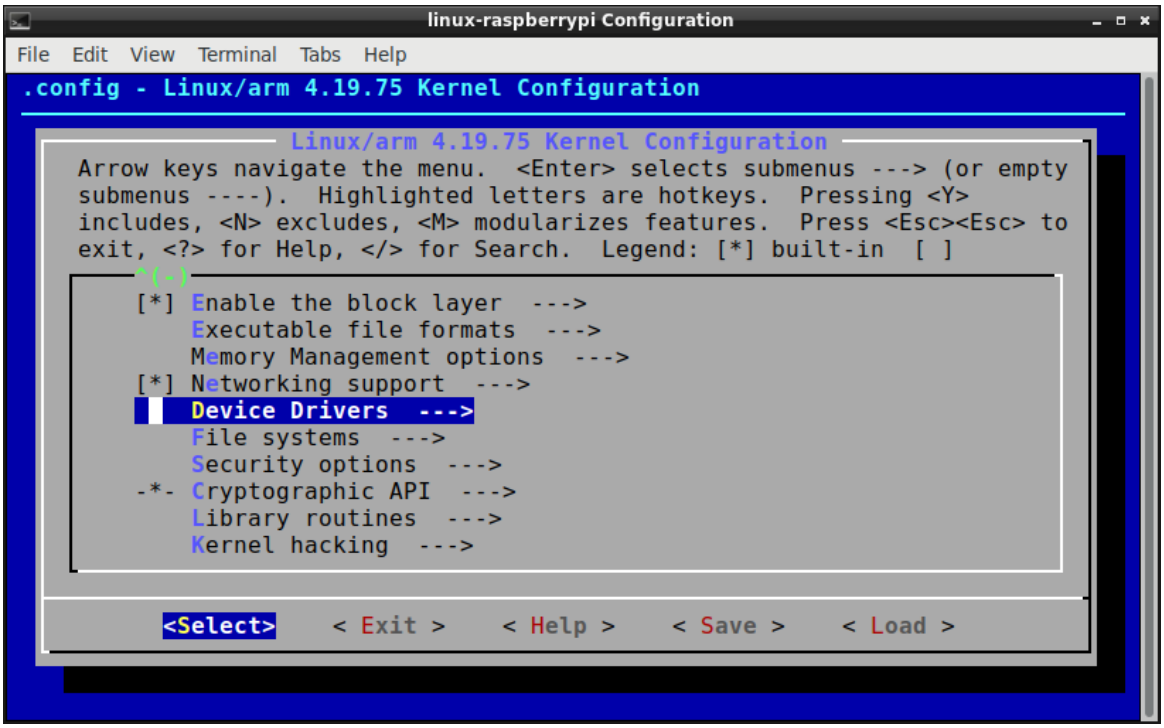
… sous-menu «*Preemption Model*» (figure IV.4-2)…



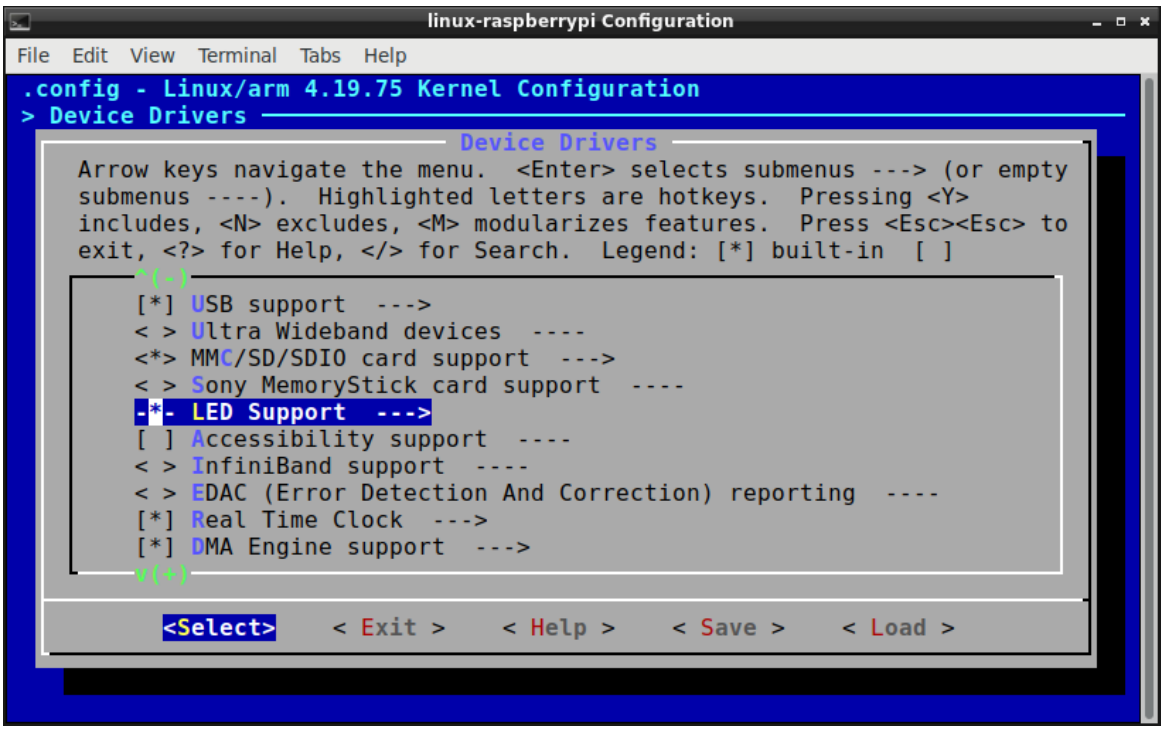
… sélectionner l'option «*Preemptible Kernel (Low-Latency Desktop)*» (figure IV.4-3).



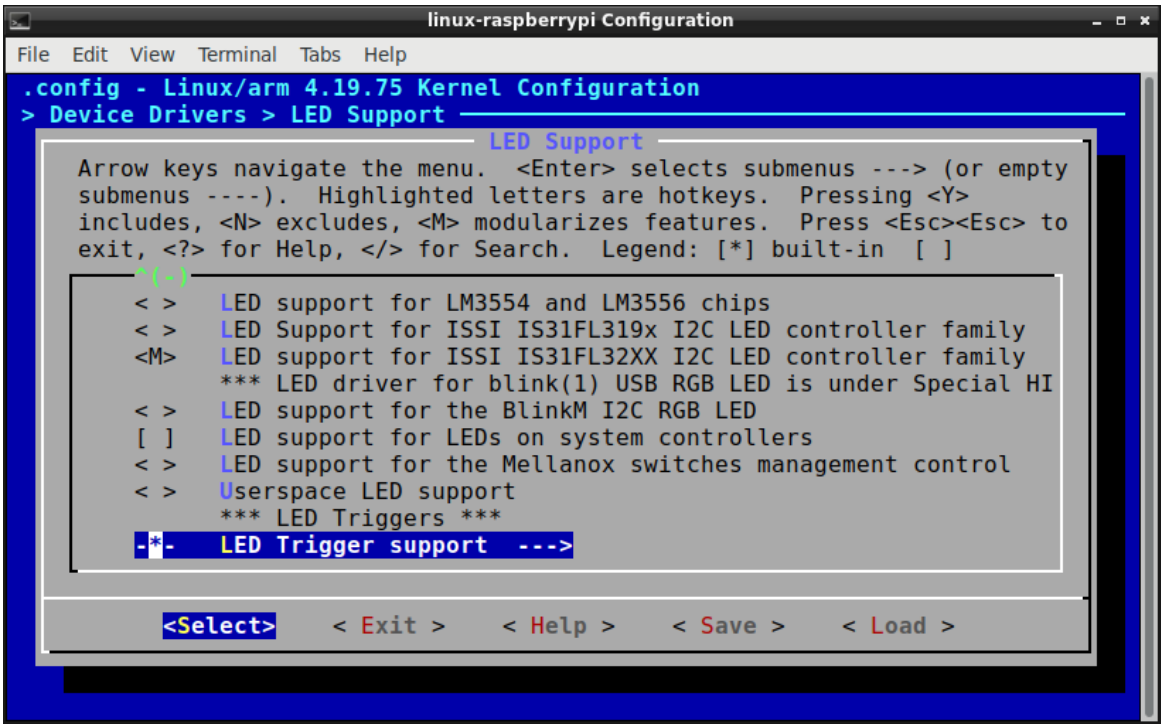
La seconde option se trouve en ouvrant le menu «*Device Drivers*» (figure IV.4-4)…



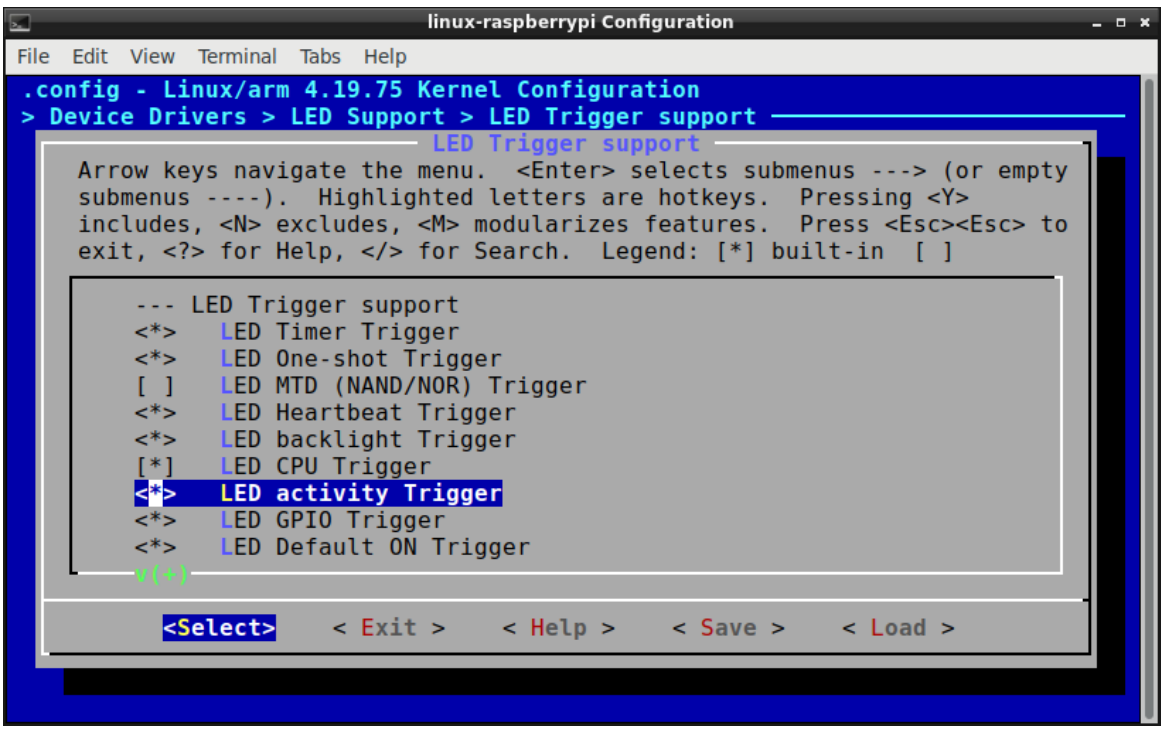
… en descendant jusqu'au sous-menu «*LED Support*» (figure IV.4-5)…



… puis le sous-menu «*LED Trigger Support*» (figure IV.4-6)…



… il faut activer (en pressant «Y») l'option «*LED activity Trigger*» (figure IV.4-7).



Comme lors de la configuration de Busybox, nous demandons à `bitbake` de nous préparer un *fragment* de configuration regroupant les options qui diffèrent des valeurs par défaut :

```
[build-rpi]$ bitbake  -c diffconfig  virtual/kernel
```

```
Config fragment has been dumped into:
 /home/cpb/Yocto-lab/build-rpi/tmp/work/raspberrypi4-poky-linu
NOTE: Tasks Summary: Attempted 235 tasks of which 234 didn't r
```

Et nous appelons `recipetool` pour qu'il crée l'extension de recette nécessaire en intégrant le *fragment* dont nous lui fournissons le chemin.

```
[build-rpi]$ recipetool appendsrcfile -w ../meta-my-layer/
```

À l'issue de cette opération, nous voyons qu'un nouveau sous-répertoire «`recipes-kernel`» est apparu dans notre *layer* et qu'il contient une extension de recette pour le noyau :

```
[build-rpi]$ ls ../meta-my-layer/
conf  COPYING.MIT  README  recipes-core  recipes-custom  recip
[build-rpi]$ ls ../meta-my-layer/recipes-kernel/
linux
[build-rpi]$ ls ../meta-my-layer/recipes-kernel/linux/
linux-raspberrypi  linux-raspberrypi_%.bbappend
[build-rpi]$ ls ../meta-my-layer/recipes-kernel/linux/linux-r
fragment.cfg
[build-rpi]$
```

Avant de relancer un *build* de l'image nous forçons l'effacement de la recette «`virtual/kernel`» pour éviter les *warnings* de `bitbake` :

```
[build-rpi]$ bitbake -c clean virtual/kernel
 [...]
[build-rpi]$ bitbake my-image
 [...]
```

Une fois la compilation terminée, et l'image installée sur le Raspberry Pi, nous allons vérifier le résultat de nos modifications :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyS0
```

```
mybox login: root
Password: (linux)
root@mybox:~# uname  -a
Linux mybox 4.19.75 #1 SMP PREEMPT Sun Jan 5 09:17:11 UTC 2020
```

Notre noyau est bien marqué comme préemptible. Voyons la configuration de la led verte, et les *triggers* disponibles :

```
root@mybox:~# cat /sys/class/leds/led0/trigger
none rc-feedback kbd-scrolllock kbd-numlock kbd-capslock kbd-k
lock kbd-altlock kbd-shiftllock kbd-shiftrlock kbd-ctrllock k
ht gpio cpu cpu0 cpu1 cpu2 cpu3 activity default-on input pani
```

Le *trigger* «`activity`» est apparu. C'est toujours «`mmc0`» qui est sélectionné par défaut, mais nous pouvons en changer :

```
root@mybox:~# echo  activity  >  /sys/class/leds/led0/trigger
root@mybox:~# cat  /sys/class/leds/led0/trigger
none rc-feedback kbd-scrolllock kbd-numlock kbd-capslock kbd-k
lock kbd-altlock kbd-shiftllock kbd-shiftrlock kbd-ctrllock k
ht gpio cpu cpu0 cpu1 cpu2 cpu3 [activity] default-on input pa
```

La led verte clignote avec un bref flash toutes les secondes, le processeur est très faiblement chargé. Lançons une petite boucle active :

```
root@mybox:~# while  true  ;  do  :  ;  done
```

Instantanément, la led clignote plus intensément. Elle revient à son motif initial dès que l'on presse *Contrôle-C*. Si on lance plusieurs instances de cette boucle en parallèle (en faisant suivre le «`done`» d'un «`&`»), la led est quasiment allumée en permanence.

Nous voyons que les modifications de configuration que nous avons apportées au noyau Linux sont bien visibles sur notre cible. Dans un projet réel, il s'agit le plus souvent d'activer des *drivers*, des protocoles réseau, etc. qui ne le sont pas dans la configuration par défaut.

Modification du *Device Tree*

Lorsque le noyau démarre, il n'a que très peu d'informations sur le système sur lequel il s'exécute. Il doit déterminer de nombreux paramètres concernant le processeur (nombre de cœurs et fréquence, type et adresse du contrôleur d'interruption, nombre et adresses des *timers*, etc.), la mémoire (quantité disponible, adresses, etc.), les bus de communication avec les périphériques, les périphériques effectivement présents…

Sur un PC par exemple, l'ensemble de ces informations est maintenu par le *BIOS* que l'on peut partiellement configurer avec le menu *Setup* à la mise sous tension.

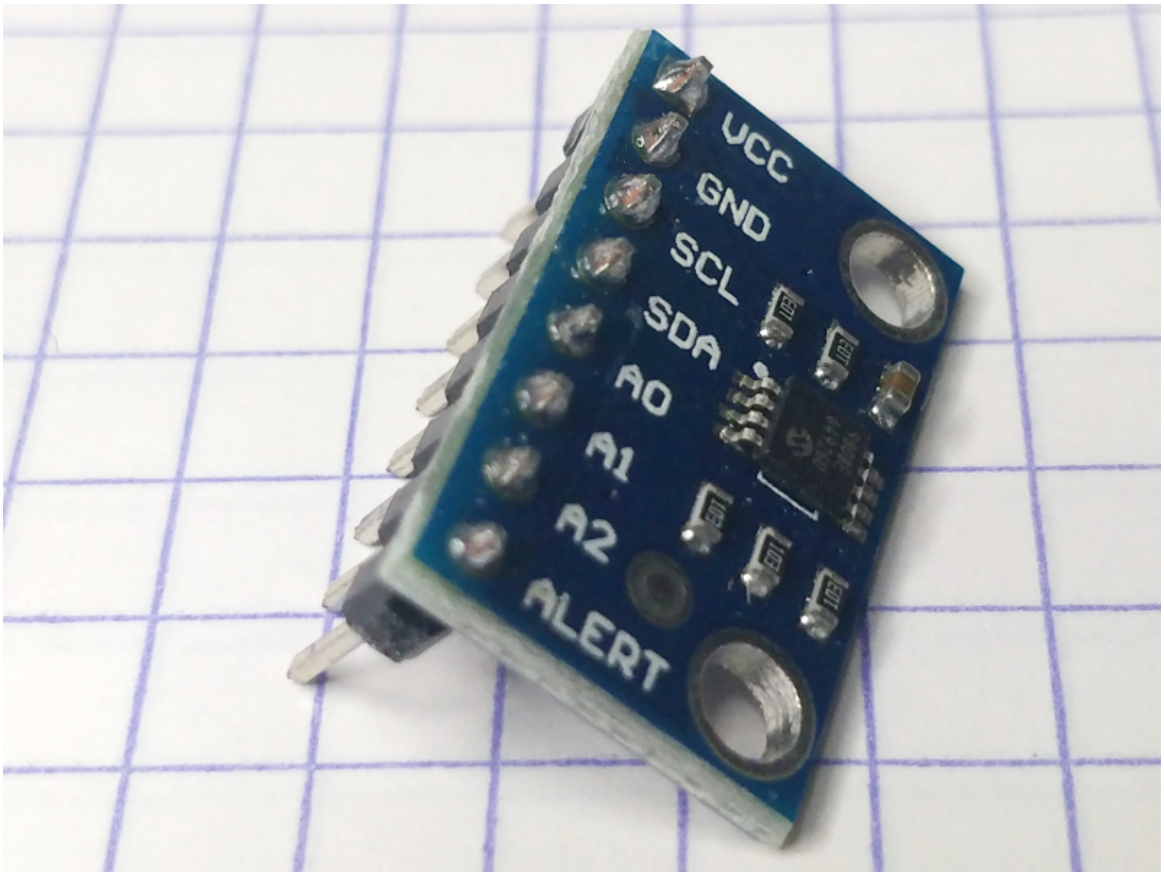
Sur l'architecture ARM — largement majoritaire pour les systèmes embarqués — il n'y a pas de *BIOS* et les paramétrage doit être explicitement fourni au noyau. C'est le rôle du *Device Tree*<.

Le *Device Tree* pour un système donné est renseigné manuellement dans **un fichier avec l'extension «`.dts`» (*Device Tree Source*)**. Celui-ci peut hériter de paramètres génériques (concernant par exemple la famille de *System On Chip*) se trouvant dans des fichiers d'extention «**`.dtsi`**» (*Device Tree Source Include*).

Les fichiers «`.dts`» et «`.dtsi`» sont livrés avec le noyau Linux pour un grand nombre de plateformes (dont les Raspberry Pi).

On utilise ensuite **un outil nommé «`dtc`» (*Device Tree Compiler*)** pour obtenir une représentation binaire du *Device Tree* prête à être interprétée par le noyau. Cette image est inscrite dans **un fichier d'extension «`.dtb`» (*Device Tree Blob*)** qui sera chargé en mémoire par le *bootloader* et transmis ainsi au *kernel*.

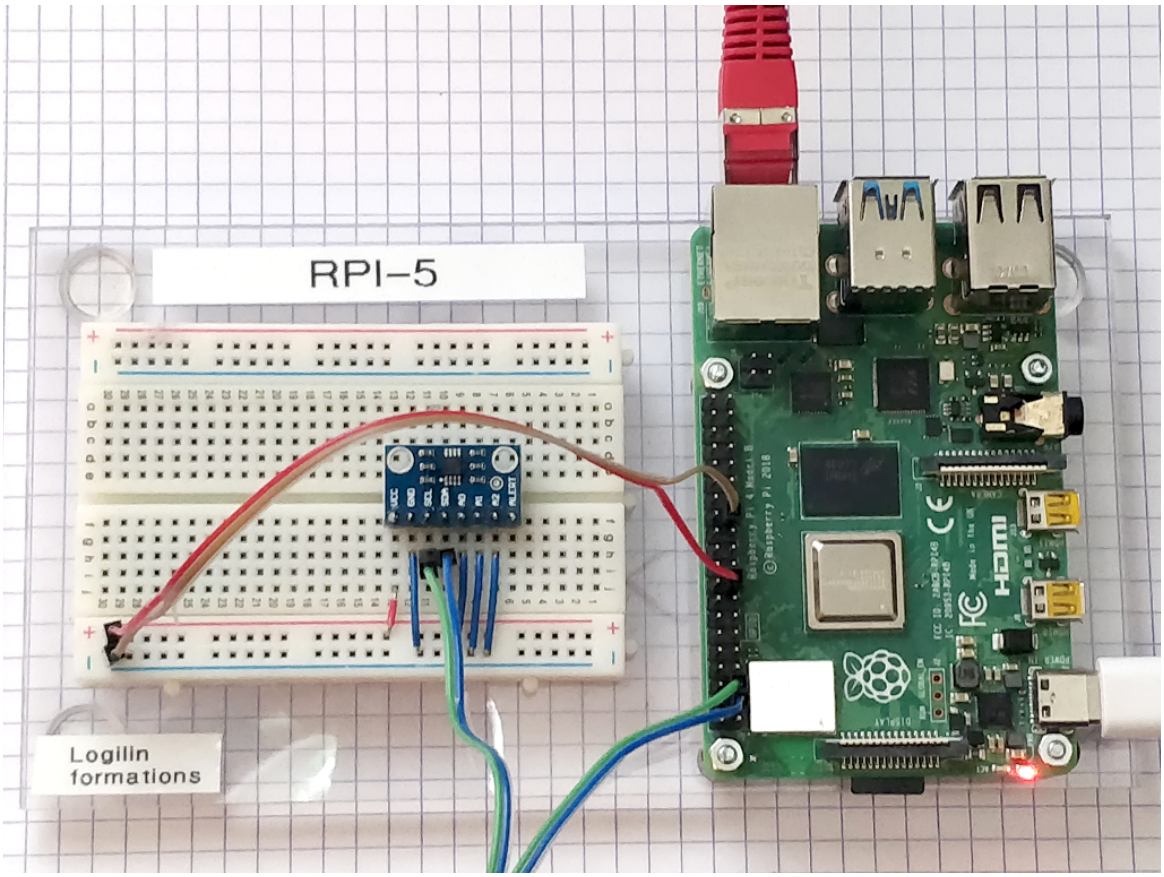
Pour nous donner un but réaliste, nous allons ajouter un petit capteur de température qui communique avec le processeur par l'intermédiaire du bus i^{2}c. J'ai choisi un capteur `MCP9808` monté sur un module simple à connecter comme on le voit sur la figure IV.3-8.



Ce petit module peut facilement être installé sur un plaquette d'essai. J'ai utilisé une plaquette que nous employons en session de formation (le «RPI 5» sur la la figure IV.3-9 est le numéro de la platine d'essai, pas celui d'un Raspberry Pi du futur !)

Sur certains bus (PCI, USB…) il est possible d'énumérer et d'identifier les périphériques présents. Sur d'autres bus (SPI, i^2c…) ce n'est pas possible et il faut indiquer au noyau quels périphériques sont connectés, ainsi que leurs adresses de communication. On réalise traditionnellement cela **dans le *Device Tree***.

Pendant la phase de prototypage avec un montage volant de ce type, on peut généralement se dispenser de modifier le *Device Tree* et agir sur les entrées `bind` dans l'arborescence `/sys/`. Nous allons imaginer que nous sommes à l'étape de validation d'une carte porteuse et que l'on souhaite ajuster précisément le *Device Tree* du module SOM.



Les connexions sont réalisées comme suit :

- Les broches «Vcc» et «Gnd» du module sont connectées aux sorties «+3.3V» (broche 17) et «Gnd» (broche 25) du Raspberry Pi. Il s'agit des deux fils rouge et marron sur la figure ci-dessus.
- Les broches «SCL» (*Serial Clock*) et «SDA» (*Serial Data*) sont reliées respectivement aux broches 5 (SCL) et 3 (SDA) du Raspberry Pi. Ce sont les fils vert et bleu sur la photo ci-dessus. Il s'agit des deux lignes nécessaires à la communication en i^{2}c.
- Les trois broches «A0», «A1» et «A2» sont reliées à la masse. Elles permettent de fixer les trois bits de poids faible de l'adresse i^{2}c du capteur. Ceci permet de communiquer avec plusieurs capteurs (jusqu'à 8) sur le même bus. En regardant dans la documentation du composant MCP 9808 (page 14) nous déterminons que l'adresse i^{2}c du capteur sera `00011000` en binaire soit `0x18` en hexadécimal.
- La sortie «Alert» n'est pas connectée, elle pourrait servir à déclencher une interruption en cas de dépassement d'un seuil programmé.

Les points importants pour le support dans le noyau sont :

- Bus i^{2}c concerné : le bus accessible sur le port d'extension du Raspberry Pi 4 est «`i2c1`».
- Adresse du *device* sur le bus : «`0x18`».

- *Driver* concerné dans le noyau Linux : cela nous demande un peu de recherche dans le répertoire `Documentation/` des sources du kernel. Le *driver* capable de piloter ce composant et plusieurs autres est «`jc42`» du sous-système *hardware monitoring* (ce capteur peut être utilisé pour surveiller la température interne d'un PC par exemple). Le nom à indiquer dans le *Device Tree* pour identifier le *driver* est : «`edec,jc-42.4-temp`»

Patch sur les sources du noyau

Nous allons faire un *patch* sur le noyau Linux en modifiant l'extrait de *Device Tree* concerné. Pour cela nous devons vérifier quelle version du noyau est compilée par Poky. La commande «`uname -a`» nous a déjà indiqué `4.19.75`.

En cherchant dans le *layer* «`meta-raspberrypi`», nous trouvons une recette `meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.19.bb` contenant :

```
LINUX_VERSION ?= "4.19.75"
LINUX_RPI_BRANCH ?= "rpi-4.19.y"

SRCREV = "642e12d892e694214e387208ebd9feb4a654d287"

require linux-raspberrypi_4.19.inc
```

Cette recette inclut le fichier `meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.19.inc` dont voici le contenu :

```
FILESEXTRAPATHS_prepend := "${THISDIR}/linux-raspberrypi:"

SRC_URI = " \
    git://github.com/raspberrypi/linux.git;branch=${LINUX_RPI_
    "
SRC_URI_append_raspberrypi4-64 = " file://rpi4-64-kernel-misc.

require linux-raspberrypi.inc

LIC_FILES_CHKSUM = "file://COPYING;md5=bbea815ee2795b2f4230826

KERNEL_EXTRA_ARGS_append_rpi = " DTC_FLAGS='-@ -H epapr'"
```

Voici les éléments qui nous intéressent :

- le mode d'obtention du noyau (variable `SRC_URI`) est «`git`»
- l'adresse du dépôt : (`SRC_URI`) `git://github.com/raspberrypi/linux.git`)
- la branche pour `git` (variable `LINUX_RPI_BRANCH`) : «`rpi-4.19.y`»
- le numéro de *commit* (variable `SRCREV`) : `642e12d892e694214e387208ebd9feb4a654d287`

Nous pouvons donc, dans un répertoire de travail, télécharger ces (volumineuses) sources et extraire le numéro de *commit* qui nous intéresse :

```
[build-rpi]$ git  clone  git://github.com/raspberrypi/linux.gi
Clonage dans 'linux'...
remote: Enumerating objects: 7604610, done.
remote: Total 7604610 (delta 0), reused 0 (delta 0), pack-reus
Réception d'objets: 100% (7604610/7604610), 2.22 GiB | 23.70 M
Résolution des deltas: 100% (6352750/6352750), fait.
Extraction des fichiers: 100% (62358/62358), fait.
[build-rpi]$ cd  linux/
[linux]$ git  checkout  642e12d8
Extraction des fichiers: 100% (2567/2567), fait.
Note : extraction de '642e12d8'.
  [...]
```

Nous avons les sources du noyau. Les fichiers sources de *Device Tree* se trouvent dans le sous-répertoire `arch/arm/boot/dts/`. Celui du Raspberry Pi 4 est nommé `bcm2711-rpi-4-b.dts` («`bcm2711`» faisant référence au *System On Chip* Broadcom 2711 utilisé dans cette version du Raspberry Pi).

En examinant le contenu du fichier, on trouve la section

```
  [...]
&i2c1 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c1_pins>;
        clock-frequency = <100000>;
}
  [...]
```

J'édite alors le fichier pour ajouter les lignes suivantes :

```
[linux]$ nano  arch/arm/boot/dts/bcm2711-rpi-4-b.dts
  [...]
&i2c1 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c1_pins>;
        clock-frequency = <100000>;

        status = "okay";
        temp@18 {
                compatible = "jedec,jc-42.4-temp";
                reg = <0x18>;
                #address-cells = <0x1>;
                #size-cells = <0x0>;
        };
};
```

Notre propos n'est pas ici de détailler le contenu du *Device Tree* mais de voir comment intégrer nos modifications au *build* de Yocto. Pour produire un *patch*, le plus simple est de demander à `git` de s'en charger :

```
[linux]$ git  diff  >  ../001-add-i2c-device-in-dts.patch
[linux]$ cd  ..
```

Je peux alors l'outil `recipetool` comme nous l'avions fait dans la séquence II.3 :

```
[build-rpi]$ recipetool  appendsrcfile  ../meta-my-layer/  vir
```

Il me reste à regénérer mon image (après avoir effacé le noyau précédemment compilé pour éviter les *warnings*) et à la charger sur le Raspberry Pi :

```
[build-rpi]$ bitbake  -c  clean  virtual/kernel
[build-rpi]$ bitbake  my-image
```

Si tout se passe bien, je dois avoir dès le démarrage une nouvelle entrée dans le sous-répertoire «`hwmon/`» de `/sys/class` :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyS0

mybox login: root
Password: (linux)
root@mybox:~# ls  /sys/class/hwmon/
hwmon0  hwmon1
```

En effet, normalement le Raspberry Pi 4 n'a qu'une entrée `hwmon0`, dont le contenu est le suivant :

```
root@mybox:~# ls  /sys/class/hwmon/hwmon0/
device  in0_lcrit_alarm  name  power  subsystem  uevent
```

Voyons le contenu de la seconde entrée :

```
root@mybox:~# ls  /sys/class/hwmon/hwmon1/
device  of_node  subsystem   temp1_crit_alarm  temp1_input  te
name    power    temp1_crit  temp1_crit_hyst   temp1_max    te
```

Ceci est typique des capteurs de températures servant à surveiller des seuils.
Voyons la température actuelle (en millidegrés Celsius) :

```
root@mybox:~# cat  /sys/class/hwmon/hwmon1/temp1_input
24750
```

24,75 °C, il fait vraiment trop chaud dans mon bureau ! Je pose le doigt quelques secondes sur le boîtier du capteur :

```
root@mybox:~# cat  /sys/class/hwmon/hwmon1/temp1_input
29062
```

Je retire mon doigt et la température descend doucement :

```
root@mybox:~# cat  /sys/class/hwmon/hwmon1/temp1_input
26250
```

```
root@mybox:~# cat  /sys/class/hwmon/hwmon1/temp1_input
25687
root@mybox:~# cat  /sys/class/hwmon/hwmon1/temp1_input
25000
root@mybox:~#
```

Conclusion

Cette séquence nous aura permis d'intervenir dans deux points critiques de l'image Yocto : le noyau Linux et le *Device Tree*. Aujourd'hui c'est une part importante du travail nécessaire pour supporter une nouvelle carte, celle qui concerne la communication avec le *hardware*.

Nous avons vu au long de ce cours en ligne de nombreux aspects de l'utilisation de Yocto pour produire une image au contenu parfaitement maitrisé.

J'espère que ces documents vous seront utiles. Les lecteurs souhaitant bénéficier d'un support technique pour leur projet, d'une formation spécifique (dans leurs locaux, adaptée à leur matériel personnel, etc.) ou qui préfèrent au contraire suivre un cours classique en format inter-entreprise pourront me retrouver chez Logilin.

«« **sommaire**