Optimisation du temps de boot d'un système Linux embarqué*

Christophe BLAESS

Pour de nombreuses applications embarquées, la durée de démarrage du système est un facteur crucial dans l'interaction avec l'utilisateur. Je me suis intéressé il y a quelques temps à l'optimisation du temps de boot d'un système embarqué conçu par un client. J'ai décidé d'approfondir un peu le sujet pour déterminer quelles séquences de démarrage méritaient un effort d'optimisation et une méthode de travail.

1. Durée de boot

Sur la plupart des systèmes interactifs, la durée de démarrage est une composante importante de la qualité perçue. Chaque seconde passée à attendre impatiemment que le système soit disponible agit négativement sur le ressenti de l'utilisateur.

Pour un serveur ou un poste de travail, certains artifices – afficher une barre de progression, indiquer les tâches en cours d'initialisation, etc. – rendent cette attente moins pénible et font prendre conscience à l'utilisateur de la complexité et de la puissance de l'outil sur lequel il va pouvoir se connecter prochainement.

Lorsqu'il s'agit d'un système embarqué, donc dédié à une tâche particulière, l'utilisateur ne souhaite généralement pas avoir d'information sur le fonctionnement interne de son système et le détail de l'initialisation ne l'intéresse pas. Nous devons réussir à démarrer le plus rapidement possible le code « métier », celui pour lequel est conçu le système embarqué.

Toutefois, ce qui est à prendre en considération n'est pas la durée de boot *complète*, celle au bout de laquelle le système sera totalement opérationnel, mais une durée de boot *subjective*, à l'issue de laquelle l'utilisateur aura la sensation que l'interaction est possible. Il n'est pas nécessaire par exemple que l'interface réseau soit totalement configurée lorsque l'écran d'accueil propose une première liste de choix à l'utilisateur.

Je vais donc m'intéresser ici à une durée de boot que je définis comme le temps s'écoulant entre la mise sous tension (ou l'activation d'une broche *reset*) et le début d'exécution d'un programme utilisateur personnalisé, indépendant du système. Pour que ce programme fonctionne, il faudra simplement que le noyau ait terminé son initialisation, et que l'arborescence des fichiers soit disponible.

> Quels sont nos moyens d'action ?
>
> Lorsque débute la phase d'optimisation d'un système Linux, il faut se fixer des limites d'action, éventuellement des objectifs à atteindre. Je considérerai ici que nous pouvons agir sur les scripts de démarrage du système et sur la configuration du noyau. Je déconseille généralement de modifier le code noyau (par exemple en appliquant des *patches* personnels) car il est important de permettre les mises à jour du kernel si des correctifs de sécurité sont publiés. Sur le long terme, ceci est difficile à garantir lorsque des modifications non-officielles ont été apportées.
>
> En ce qui concerne les objectifs, j'envisage de présenter une première interaction visuelle (un écran d'accueil) une seconde environ après la mise sous tension et de démarrer une application en mode utilisateur moins de deux secondes plus tard.

2. Plate-forme expérimentale

Pour réaliser mon expérience de manière simple et facilement reproductible par les lecteurs, j'ai choisi d'utiliser un Raspberry Pi, et de chronométrer le temps qui s'écoule entre la mise sous tension et le début de l'exécution d'une application utilisateur. En réalité, j'ai même choisi d'utiliser deux Raspberry Pi : le premier est chargé de déclencher le boot du second, puis de mesurer le temps s'écoulant avant que le deuxième Raspberry Pi bascule l'état d'une broche GPIO.

Les mêmes principes d'optimisation du temps de boot sont applicables à n'importe quelle autre plate-forme Linux embarqué. Appelons les deux Raspberry Pi « Test » et « Mesure ».

- *Test* est le système dont nous allons chronométrer la durée de démarrage. Il se trouve initialement dans un

* Cet article est paru dans Open Silicium de décembre 2013, disponible sur `https://boutique.ed-diamond.com/anciens-numeros/535-open-silicium-9.html`

état arrêté logiciellement (après invocation de la commande **`halt`**). Toutefois son alimentation électrique est toujours branchée, ceci permettra de le faire démarrer grâce à une impulsion sur sa broche *Reset*.

- *Mesure* est le Raspberry Pi chargé de déclencher le boot de son compagnon en appliquant une tension appropriée sur la broche de *reset* puis de chronométrer le temps de démarrage.

- Lorsque le Raspberry Pi *Test* aura terminé son initialisation, il notifiera, par l'intermédiaire d'une broche GPIO, le Raspberry Pi *Mesure*. Ce dernier affichera la durée écoulée.

- En nous connectant sur le Raspberry Pi Test sous l'identité *root*, nous pourrons alors lui demander de s'arrêter à nouveau en utilisant la commande **`halt`**, afin de passer à l'expérience suivante.

Cette méthode d'instrumentation présente l'avantage de n'ajouter aucun surplus de code dans le système chronométré et d'être facilement automatisable pour obtenir des séries de mesures successives grâce à des scripts. Notons, qu'il existe également quelques outils permettant de mesurer des temps de démarrage :

- **`CONFIG_PRINTK_TIME`** « *Show timing information on printks* » : cette option que l'on trouve dans le menu « *Kernel hacking* » de la configuration du noyau permet d'ajouter un horodatage au début de chaque ligne affichée par le kernel (visibles après le boot avec la commande **`dmesg`**). Outre la légère surcharge de code, l'inconvénient de cette valeur est qu'elle est initialisée au début de l'exécution du code noyau, pas à la mise sous tension. Le temps de démarrage dû au *bootloader* n'est donc pas pris en compte.

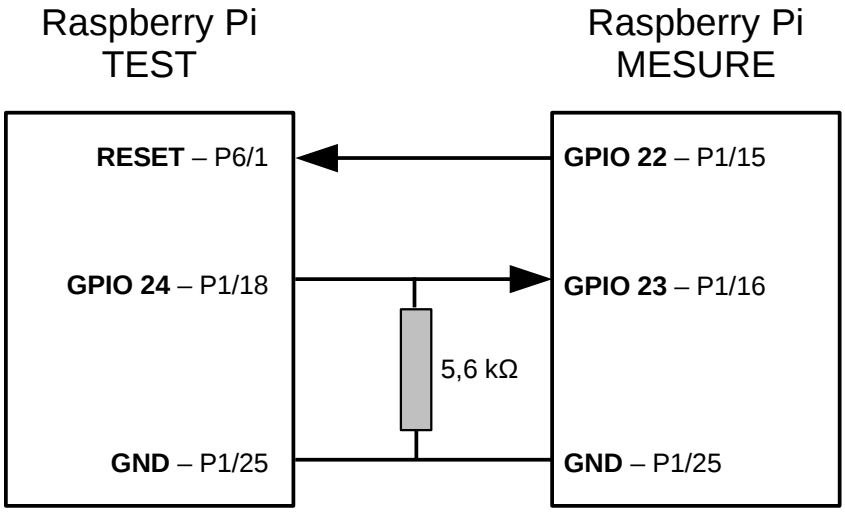
- **`CONFIG_FTRACE`** « *Tracers* » : cette option se trouvant également dans le menu « *Kernel hacking* » permet d'obtenir des traces d'exécution très précises de l'activité du noyau. Bien qu'elle soit souvent citée dans les documents traitant de l'optimisation du boot, je ne la considère pas comme très appropriée pour notre cas car je ne souhaite pas modifier le code du kernel. Elle est surtout très utile lors de l'optimisation du fonctionnement d'un driver développé spécifiquement.

- **`bootchartd`** : ce démon doit être lancé dès l'initialisation du mode utilisateur et présente les activités des différents processus de manière graphique. Cet outil est très utile, mais je ne l'emploie pas beaucoup dans les environnements embarqués. Je le réserve plutôt pour l'optimisation de systèmes complexes (serveurs, outils industriels) contenant de nombreuses tâches dans l'espace utilisateur. On peut considérer que son travail commence là où nous terminerons le nôtre.

Pour provoquer facilement le démarrage du Raspberry Pi *Test*, il suffit d'appliquer une tension nulle suivie d'une une tension de +3.3 V sur la broche *Run* du *system-on-chip* BCM2835. Cette broche est accessible sur la plupart des modèles de *Raspberry Pi* (hormis les tout premiers) via un point de contact nommé P6 se trouvant à gauche du connecteur HDMI (voir photo).

Le fait de court-circuiter les deux points du connecteur P6 ou de relier le point le plus proche du bord de la carte (celui dont la pastille de cuivre est carrée) à la masse temporairement va provoquer le redémarrage du Raspberry Pi.

Voici les branchements réalisés.



La valeur précise de la résistance n'a pas d'importance au-delà de quelques kilo-ohms, elle sert simplement

de *pull-down* : pendant que le Raspberry Pi *Test* redémarre, elle maintient au niveau bas l'entrée GPIO 23 du Raspberry Pi *Mesure*.

3. Paramétrage de l'espace utilisateur

3.1 Application lancée à la fin des scripts d'initialisation

Dans cette première expérience, nous allons invoquer notre code applicatif dans le script qui paraît *a priori* le plus adapté : **`/etc/rc.local`**. Il est destiné à accueillir les commandes et les instructions de démarrage spécifiques pour le système hôte. J'ai choisi d'utiliser un accès aux GPIO par l'intermédiaire du système de fichiers **`/sys`**. Ce mécanisme n'est pas le plus efficace, il existe des bibliothèques accédant directement aux projections des GPIO dans **`/dev/mem`**, mais je trouve que l'accès par **`/sys`** est plus propre, plus correct vis-à-vis du kernel. Nous ne cherchons pas à obtenir une optimisation du temps de boot à la microsecondes près, plutôt des valeurs de l'ordre de la dizaine ou centaine de millisecondes, aussi le retard induit par les appels-système vers **`/sys`** sont-ils négligeables.

Le programme qui va basculer l'état de la broche GPIO est le suivant :

```
// up-gpio.c : sort une impulsion sur la broche 18 (GPIO 24)

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

int main(void)
{
    FILE * fp;

    if ((fp = fopen("/sys/class/gpio/export", "w")) != NULL) {
        fprintf(fp, "24\n");
        fclose(fp);
    }
    if ((fp = fopen("/sys/class/gpio/gpio24/direction", "w")) != NULL) {
        fprintf(fp, "out\n");
        fclose(fp);
    }
    if ((fp = fopen("/sys/class/gpio/gpio24/value", "w"))!= NULL) {
        fprintf(fp, "1\n");
        fclose(fp);
    }
    if ((fp = fopen("/sys/class/gpio/gpio24/value", "w"))!= NULL) {
        fprintf(fp, "0\n");
        fclose(fp);
    }
    if ((fp = fopen("/sys/class/gpio/gpio24/direction", "w")) != NULL) {
        fprintf(fp, "in\n");
        fclose(fp);
    }
    if ((fp = fopen("/sys/class/gpio/unexport", "w")) != NULL) {
        fprintf(fp, "24\n");
        fclose(fp);
    }

    return EXIT_SUCCESS;
}
```

Ce programme est compilé directement sur le Raspberry Pi *Test* et installé dans un répertoire système.

```
(Test)# gcc  up-gpio.c  -o  /usr/local/bin/up-gpio  -Wall
(Test)#
```

Nous ajoutons alors l'invocation dans le script **`/etc/rc.local`** :

```
#!/bin/sh -e
#
# rc.local
[...]

/usr/local/bin/up-gpio

exit 0
```

Arrêtons le Raspberry Pi de Test :

```
(Test)# halt

The system is going down for system halt NOW!MA0) (Thu Oct 10 16:49:46 2013):
```

```
[  203.107550] Power down.
```

Surtout **ne le débranchons pas**, il faut le laisser en attente d'un signal sur sa broche *reset*.

3.2 Programme de chronométrage

Sur le Raspberry Pi *Mesure*, nous allons compiler un petit programme chargé de chronométrer le temps s'écoulant entre l'application du signal de *reset* et la réponse du système de *Test*.

Le programme que nous lancerons sur le Raspberry Pi *Mesure* est le suivant :

```
// chronoboot.c - Programme mesurant le temps écoulé entre la
//      réinitialisation de l'autre Raspberry Pi et la fin de
//      son boot.
//

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

// Sortie sur broche 15 (GPIO 22)
#define CHRONOBOOT_GPIO_OUT 22

// Entrée sur broche 16 (GPIO 23)
#define CHRONOBOOT_GPIO_IN  23

#define LG_BUFFER 256

int gpio_export(int gpio)
{
    FILE * fp;
    if (((fp = fopen("/sys/class/gpio/export", "w")) == NULL)
     || (fprintf(fp, "%d\n", gpio) <= 0)
     || (fclose(fp) != 0)) {
        perror("export");
        return -1;
    }
    return 0;
}


void gpio_unexport(int gpio)
{
    FILE * fp;
    fp = fopen("/sys/class/gpio/unexport", "w");
    fprintf(fp, "%d\n", gpio);
    fclose(fp);
}


int gpio_set_output(int gpio)
{
    FILE * fp;
    char buffer[LG_BUFFER];

    snprintf(buffer, LG_BUFFER, "/sys/class/gpio/gpio%d/direction", gpio);
    if (((fp = fopen(buffer, "w")) == NULL)
     || (fprintf(fp, "out\n") <= 0)
     || (fclose(fp) != 0)) {
        perror("set_output");
        return -1;
    }
    return 0;
}


int gpio_open_fd(int gpio, int flags)
{
    char buffer[LG_BUFFER];
    snprintf(buffer, LG_BUFFER, "/sys/class/gpio/gpio%d/value", gpio);
    return open(buffer, flags);
}
```

```
int gpio_set_value(int fd, int value)
{
    char buffer[LG_BUFFER];
    snprintf(buffer, LG_BUFFER, "%d\n", value);
    return pwrite(fd, buffer, strlen(buffer), 0);
}


int gpio_get_value(int fd)
{
    char buffer[LG_BUFFER];
    int value;
    memset(buffer, 0, LG_BUFFER);
    if (pread(fd, buffer, LG_BUFFER, 0) > 0)
        if (sscanf(buffer, "%d", & value) == 1)
            return value;
    return -1;
}


int main(void)
{
    int fd_in;
    int fd_out;
    int value;
    struct timeval tv;
    struct timeval tv_start;
    long delay;

    // Exporter les GPIO pour qu'elles soient disponibles dans /sys
    if (gpio_export(CHRONOBOOT_GPIO_IN) != 0)
        exit(EXIT_FAILURE);
    if (gpio_export(CHRONOBOOT_GPIO_OUT) != 0) {
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }
    if (gpio_set_output(CHRONOBOOT_GPIO_OUT) != 0) {
        gpio_unexport(CHRONOBOOT_GPIO_OUT);
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }

    fd_in = gpio_open_fd(CHRONOBOOT_GPIO_IN, O_RDONLY);
    if (fd_in < 0) {
        perror("open(gpio_in)");
        gpio_unexport(CHRONOBOOT_GPIO_OUT);
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }

    fd_out = gpio_open_fd(CHRONOBOOT_GPIO_OUT, O_WRONLY);
    if (fd_out < 0) {
        perror("open(gpio_out)");
        close(fd_in);
        gpio_unexport(CHRONOBOOT_GPIO_OUT);
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }

    // Baisser la broche Reset du Raspberry Pi testé.
    if (gpio_set_value(fd_out, 0) < 0) {
        perror("set_value(gpio_out, 0)");
        close(fd_out);
        close(fd_in);
        gpio_unexport(CHRONOBOOT_GPIO_OUT);
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }

    // Attendre 50 µs.
    gettimeofday(& tv_start, NULL);
    do {
        gettimeofday(& tv, NULL);
        // Calcul en microsecondes.
        delay  = tv.tv_sec - tv_start.tv_sec;
        delay *= 1000000;
        delay += tv.tv_usec - tv_start.tv_usec;
    } while (delay < 50);
```

```
    // Relever la broche Reset, le Raspberry Pi testé va démarrer.
    if (gpio_set_value(fd_out, 1) < 0) {
        perror("set_value(gpio_out, 1)");
        close(fd_out);
        close(fd_in);
        gpio_unexport(CHRONOBOOT_GPIO_OUT);
        gpio_unexport(CHRONOBOOT_GPIO_IN);
        exit(EXIT_FAILURE);
    }
    // Lire l'heure de début de boot.
    gettimeofday(& tv_start, NULL);


    for(;;) {
        gettimeofday(& tv, NULL);
        // Attendre que l'autre Raspberry monte la broche GPIO
        value = gpio_get_value(fd_in);
        if (value == 1) {
            // Calculer la durée de boot en microsecondes.
            delay  = tv.tv_sec - tv_start.tv_sec;
            delay *= 1000000;
            delay += tv.tv_usec - tv_start.tv_usec;
            // Et l'afficher.
            printf("%ld\n", delay);
            break;
        }
        if (value < 0) {
            perror("get_value");
            break;
        }
    }
    close(fd_out);
    close(fd_in);
    gpio_unexport(CHRONOBOOT_GPIO_OUT);
    gpio_unexport(CHRONOBOOT_GPIO_IN);

    return 0;
}
```

L'attente active du signal provenant du Raspberry Pi *Test* n'est pas très élégante, mais elle ne dure que quelques secondes. Si la durée avait été vraiment plus longue, j'aurai probablement opté pour une gestion sur interruption avec un petit module personnalisé dans le noyau du Raspberry Pi *Mesure*.

Nous compilons le code et le lançons :

```
(Mesure)# gcc  chronoboot.c  -o  chronoboot  -Wall
(Mesure)# ./chronoboot
22435348
(Mesure)#
```

Le temps mesuré est affiché en microsecondes. Il correspond à 22,4 secondes. C'est très long pour un temps de démarrage de système embarqué, nous allons essayer de réduire cette durée.

Par acquit de conscience, je vais réaliser systématiquement cinq mesures pour chaque expérience (en arrêtant le système *Test* à chaque fois) pour obtenir une valeur moyenne indépendante des petites fluctuations observées (latences de la carte SD, initialisation des périphériques, etc.).

```
(Mesure)# ./chronoboot
22930742
(Mesure)# ./chronoboot
22469826
(Mesure)# ./chronoboot
23322268
(Mesure)# ./chronoboot
22425414
(Mesure)#
```

Par la suite je ne présenterai que la valeur moyenne, pas les lancements de **`chronoboot`**.

La moyenne des durées de démarrage est de 22716720 microsecondes soient **22,7 secondes**.

3.3 Début des scripts d'initialisation

Notre premier effort va consister à lancer le programme applicatif au plus tôt parmi les scripts d'initialisation. Notre choix précédent, le script **`/etc/rc.local`**, était celui qui semblait le plus évident au premier coup d'œil,

mais on peut rechercher un point de démarrage plus tôt.

Sur de nombreux systèmes (dont ceux construits autour de l'outil embarqué Busybox), le premier processus utilisateur du système, **init** (dont le PID vaut 1), exécute tout d'abord le script **/etc/init.d/rcS**. C'est le cas sur notre système Raspbian, **rcS** servant à lancer tous les autres scripts de boot :

```
#! /bin/sh
#
# rcS
#
# Call all S??* scripts in /etc/rcS.d/ in numerical/alphabetical order
#
exec /etc/init.d/rc S
```

Nous pouvons insérer une ligne avant le « **exec** » pour appeler notre application utilisateur.

Toutefois, en ce point, le système est encore très rudimentaire et ne dispose pas encore du contenu du répertoire **/sys** pourtant nécessaire pour accéder aux GPIO dans le programme **up-gpio.c** décrit plus haut. Nous devons donc prendre l'initiative de monter nous même ce système de fichiers. Cela pourrait se faire dans le script **rcS**, avant d'appeler **up-gpio** ainsi :

```
mount none /sys -t sysfs
```

Toutefois cette solution ne m'enchante guère car elle suppose l'exécution d'une commande supplémentaire (**mount**) avec la surcharge liée à l'exécution d'un processus et la lecture du fichier exécutable depuis le disque. Je préfère donc modifier mon programme précédent pour y insérer un appel-système :

```
// mount-sysfs-up-gpio.c : prepare le système de fichiers /sys et
//       active la GIO 24.

#include <stdio.h>
#include <stdlib.h>
#include <sys/mount.h>

int main(void)
{
    FILE * fp;

    mount("none", "/sys", "sysfs", 0, NULL);

    if ((fp = fopen("/sys/class/gpio/export", "w")) != NULL) {
   [...]    // identique à up-gpio.c
```

Ce code est compilé sur le Raspberry Pi *Test* et l'exécutable placé dans **/usr/local/bin**.

Voici donc le contenu du script **/etc/init.d/rcS** modifié

```
#! /bin/sh
#
# rcS
#
# Call all S??* scripts in /etc/rcS.d/ in numerical/alphabetical order
#
/usr/local/bin/mount-sysfs-up-gpio
exec /etc/init.d/rc S
```

Reprenons nos tests avec **chronoboot** sur le Raspberry Pi *Mesure*. La durée moyenne est de **4,330 secondes**.

3.4 Remplacement du processus init

Lorsque le noyau a terminé son initialisation et celles des périphériques dont les drivers sont compilés statiquement, il lance un premier processus utilisateur : **init**. Celui-ci est généralement **/sbin/init**. Toutefois nous pouvons demander, dans les options de boot du kernel, à exécuter un autre processus **init**. Par exemple notre processus **mount-sysfs-up-gpio**.

Le seul problème est que ce dernier devra assurer l'initialisation propre du système (lancer les scripts) puis se mettre en attente pour adopter les processus orphelins et lire leurs statuts de terminaison et éviter de laisser des processus zombies.

La solution la plus simple, une fois notre action réalisée consiste à exécuter le programme **/sbin/init** original en invoquant l'appel-système **execve()** qui va charger ce fichier directement dans la mémoire de notre processus (lui garantissant de conserver ainsi un PID égal à 1). Nous transformerons donc notre programme précédent en :

```
// mount-sysfs-up-gpio-init.c : prepare le système de fichiers /sys,
//       active la GIO 24, puis lance le programme init classique
```

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

int main(int argc, char *argv[], char * env[])
{
  [...]  // identique à mount-sysfs-up-gpio.c

    execve("/sbin/init", argv, env);

    return EXIT_FAILURE;
}
```

Pour accéder aux options de *boot* du kernel sur un Raspberry Pi, nous allons éditer le fichier **cmdline.txt** se trouvant sur la première partition de la carte SD. Ceci est spécifique au *bootloader* du Raspberry Pi, et sur un système embarqué un peu plus traditionnel, on s'intéresserait plutôt aux variables d'environnement de U-Boot.

Dans ce fichier nous trouvons les options qui sont transmises au kernel par le *bootloader*. Ces options se trouvent sur une seule ligne, que l'on qualifie un peu abusivement de *ligne de commande* du noyau. Par défaut, avec la distribution Raspbian nous trouvons les options suivantes :

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

Nous pouvons ajouter, par exemple en début de ligne l'option suivante :

```
init=/usr/local/bin/mount-sysfs-up-gpio-init
```

Une fois le fichier sauvegardé, nous arrêtons le Raspberry Pi *Test* et lançons **chronoboot** sur le Raspberry Pi *Mesure*. La durée de boot moyenne est de **4,101 secondes**.

3.5 Compilation statique de init

Notre processus **mount-sysfs-up-gpio-init** est certes très simple, mais il nécessite le chargement de plusieurs bibliothèques dynamiques, ce qui alourdit son démarrage. Nous pouvons en avoir la confirmation en utilisant la commande **ldd** ;

```
(Test)# ldd  /usr/local/bin/mount-sysfs-up-gpio-init
        /usr/lib/arm-linux-gnueabihf/libcofi_rpi.so (0xb6fc1000)
        libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xb6e88000)
        /lib/ld-linux-armhf.so.3 (0xb6fce000)
(Test)#
```

Nous pouvons essayer de compiler statiquement ce processus et voir si le chargement est plus rapide :

```
(Test)#  gcc    mount-sysfs-up-gpio-init.c    -o    /usr/local/bin/mount-sysfs-up-gpio-init -static  -Wall
(Test)# ldd  /usr/local/bin/mount-sysfs-up-gpio-init
        not a dynamic executable
(Test)#
```

La durée moyenne mesurée est de **3,995 secondes**.

3.6 Suppression des mesures de durées de boucle

Durant son initialisation, le noyau doit déterminer la rapidité du processeur sur lequel il s'exécute, ce qui lui permettra de réaliser par la suite des opérations d'attentes actives très précises grâce à des boucles (utilisées dans les fonctions **ndelay()**, **udelay()** ou **mdelay()** du kernel). Pour ce faire, il programme une interruption horaire et mesure le nombre d'itérations qu'il peut réaliser en un *jiffie*, une unité de temps interne au kernel dont la valeur de l'ordre de la milliseconde est configurable à la compilation du noyau.

Dans les traces du kernel, visibles avec la commande **dmesg**, nous pouvons obtenir la valeur de **lpj** (*loops per jiffy*) mesurée.

```
(Test)# dmesg  |  grep lpj
[    0.001019] Calibrating delay loop... 464.48 BogoMIPS (lpj=2322432)
(Test)#
```

Après avoir effectué un boot et relevé cette valeur, nous pouvons dorénavant la fournir dans les options de boot, ce qui évitera au kernel de devoir la mesurer de nouveau. On indique la valeur en ajoutant

```
lpj=2322432
```

Dans le fichier **cmdline.txt** se trouvant sur la première partition de la carte SD. Au prochain reboot, on

trouve dans les traces du kernel le message suivant :

```
[    0.001024] Calibrating delay loop (skipped) preset value.. 464.48 BogoMIPS (lpj=2322432)
```

indiquant que la valeur a été fixée directement sans calcul.

Attention il peut y avoir des variations sensibles de la valeur de **lpj** entre différentes générations du Raspberry Pi aussi veillez à mesurer la valeur lors d'un boot normal avant de la fixer dans les options de démarrage.

La moyenne des durées de boot est **3,935 secondes**. L'amélioration est faible mais nous progressons toujours...

3.7 Suppression des messages du noyau

Cette action est généralement l'une des premières envisagées lorsque l'on veut démarrer rapidement une machine. Lors du boot, le noyau affiche sur sa console de nombreux messages (résultats d'initialisation de drivers par exemple) ce qui a tendance à ralentir sa progression. Sur le Raspberry Pi, où les traces du kernel sont redirigées vers la console série, l'effet est encore plus important que sur un système où elles sont affichées dans un buffer vidéo.

On peut facilement éliminer l'affichage des messages du noyau en ajoutant l'option **quiet** dans ses options de démarrage (ici dans le fichier **cmdline.txt**). Les traces pourront encore être consultées après le boot à l'aide de la commande **dmesg**. Remarquez que les messages supprimés sont uniquement ceux du kernel, les traces provenant des scripts de démarrage continuent à apparaître sur la console.

Mes mesures fournissent une moyenne de **2,947 secondes** ! Jolie progression, non ?

4. Paramétrage du kernel

L'essentiel des améliorations dans l'espace utilisateur et dans les options de démarrage du noyau ayant été menées, nous allons à présent nous résoudre à travailler avec un noyau personnalisé, que nous allons compiler et ajuster pour optimiser son temps de boot.

La procédure pour compiler un noyau personnel pour le Raspberry Pi depuis un PC Linux a déjà été présentée dans plusieurs articles. En voici un rappel succinct, on considère qu'une chaîne de compilation croisée est présente et qu'un compilateur **arm-linux-gcc** se trouve dans le répertoire **/usr/local/cross/rpi/usr/bin/**. Sinon il faudra ajuster en conséquence le préfixe indiqué dans la variable d'environnement **CROSS_COMPILE**.

```
$ git clone http://github.com/raspberrypi/linux rpi-linux
  [...]
$ cd rpi-linux
$ make ARCH=arm bcmrpi_defconfig
  [...]
$ make ARCH=arm CROSS_COMPILE=/usr/local/cross/rpi/usr/bin/arm-linux
  [...]
$
```

On insère (par exemple en utilisant un adaptateur USB) la carte SD de la distribution Raspbian dans le PC de compilation. Sur mon poste, les deux partitions sont automatiquement montées dans **/media/boot** et **/media/root**. Si les noms où les chemins ne sont pas les mêmes sur votre système ajustez les lignes de commandes ci-dessous.

```
$ sudo make ARCH=arm INSTALL_MOD_PATH=/media/root/ modules_install
  [...]
$ mv /media/boot/kernel.img /media/boot/old-kernel.img
$ cp arch/arm/boot/zImage /media/boot/kernel.img
$ umount /media/*oot
```

Redémarrons le Raspberry Pi *Test*, pour l'arrêter aussitôt avec la commande **halt**, et relancer **chronoboot** sur le Raspberry Pi *Mesure*.

Les valeurs relevées indiquent une moyenne de **2,980 secondes**. La durée s'est légèrement dégradée par rapport au noyau de la distribution Raspbian, mais nous allons essayer d'y remédier.

Il s'ensuit une longue série d'allers-retours entre la compilation du noyau et les mesures de **chronoboot**. Il est très important de ne modifier que peu d'options à la fois, et de bien conserver systématiquement les fichiers **.config** du noyau.

J'ai conservé les fichiers de configuration kernel correspondant aux étapes ci-après, ils sont disponibles sur

mon blog à l'adresse : **http://www.blaess.fr/christophe/articles/files-glmf/**

Le noyau testé ci-dessus pesait 2 859 520 octets et son fichier de configuration était **config-01**.

4.1 Suppression des options de debug et de profiling

Partant du principe que l'optimisation de la durée de boot ne doit se faire qu'une fois le noyau et ses drivers parfaitement au point, j'ai commencé par désactiver différentes options, liées aux fonctionnalités de débogage, avant de recompiler le noyau.

Entre autres j'ai supprimé :

- **CONFIG_PROFILING** (« *General Setup* » -> « *Profiling support* »), **CONFIG_KPROBES**,(« *General Setup* » -> « *Kprobes* ») **CONFIG_AUDIT** (« *General Setup* » -> « *Auditing Support* ») et **CONFIG_FTRACE** (« *Kernel Hacking* » -> « *Tracers* ») qui intègrent des points d'entrée dans le noyau pour des outils de suivi.

- **CONFIG_DETECT_HUNG_TASK**,(« *Kernel Hacking* » -> « *Detect Hung Tasks* »), **CONFIG_KGDB**, (« *Kernel Hacking* » -> « *KGDB : kernel debugger* ») et toutes les options **CONFIG_DEBUG_*** (dans « *Kernel Hacking* ») qui augmentent la taille du noyau et ralentissent légèrement l'exécution.

L'objectif est tout d'abord d'éliminer le code inutile qui charge l'exécution du noyau, et de réduire la taille de l'image à charger. Celle-ci est maintenant de 2 753 088 octets (fichier **config-02**).

La moyenne des durées mesurées vaut **2,836 secondes**.

4.2 Suppression des options inutiles en embarqué

Certaines options du noyau Linux n'ont pas de grande utilité sur un système embarqué, que nous souhaitons volontairement réduit et adapté à un environnement de fonctionnement particulier. J'ai donc désactivé plusieurs fonctionnalités parmi lesquelles :

- **CONFIG_SWAP** (« *General Setup* » -> « *Support for paging of anonymous memory* ») : cette option intègre dans le noyau le support pour l'utilisation éventuelle d'une partition ou d'un fichier de swap. Ceci ne présente pas d'intérêt sur un système embarqué et peut être néfaste pour la durée de vie de la mémoire flash.

- **CONFIG_CGROUPS** (« *General Setup* » -> « *Control Group Support* ») : les *cgroups* sont des options très utiles pour l'administration d'un serveur ou d'une machine multi-utilisateurs en isolant des processus dans des groupes disposant de ressources limitées. Sur un système embarqué, les limitations éventuelles pourront être gérées au cas par cas (avec les commandes **ulimit**, **taskset**, **chrt**, etc.)

- **CONFIG_CPUFREQ** (« *CPU Power Management* » -> « *CPU Frequency scaling* ») : les variations dynamiques de fréquence processeur ne sont pas supportées par le *System-on-chip* du Raspberry Pi.

- **CONFIG_IOSCHED_DEADLINE** et **CONFIG_IOSCHED_CFQ** (dans « *Enable the Block layer* » -> « *IO Schedulers* ») ces ordonnanceurs d'entrées-sorties, utilisés par le sous-système Block, ne présentent pas d'intérêt avec une mémoire flash (pas de déplacement physique) sur un système embarqué (mono-utilisateur).

- **CONFIG_SHMEM** (« *General Setup* » -> « *Use full shmem filesystem* ») : l'utilisation d'un système simplifié pour le partage de mémoire entre processus est suffisant pour la plupart des systèmes embarqués.

Les systèmes de fichiers intégrés dans le noyau ont été élagués pour ne conserver que EXT4 et VFAT, ce qui suffit dans notre cas.

Les protocoles réseau autres que IPv4 et 802.11 ont été éliminés.

On notera que la plupart des options supprimées (systèmes de fichiers, protocoles, etc.) peuvent être réinsérées en les compilant sous forme de modules ce qui n'allonge pas la durée de boot.

Le noyau (**config-03**) pèse 2 077 328octets. La durée moyenne est de **2,680 secondes**.

4.3 Modularisation systématique

Nous avons intérêt à retarder le chargement des drivers qui ne nous sont pas indispensables, en les compilant sous forme de modules. Ceci présente également l'intérêt de réduire la taille du kernel.

J'ai demandé systématiquement la compilation en modules pour tous les drivers qui l'acceptent. Afin d'accélérer le temps de compilation, j'ai également supprimé les drivers supportant du matériel optionnel non compris dans le Raspberry Pi. Par exemple les drivers d'horloges RTC accessibles en SPI ou en I²C sont supprimés. Il est très simples de les ajouter sous forme de modules si le besoin s'en fait sentir.

J'ai conservé les drivers suivants statiquement intégrés dans le noyau :

- **CONFIG_MMC_BLOCK** (« *Device Drivers* » -> « *MMC/SD/SDIO card support* ») et les options relatives aux support de la carte SD sur le Raspberry Pi,

- **CONFIG_USB_DWCOTG** (« *Device drivers* » -> « *USB Support* » -> « *Synopsis DWC host support* ») qui pose des problèmes lors d'une compilation en module, nous en reparlerons plus bas,

- **CONFIG_SERIAL_AMBA_PL011** (« *Devices Drivers* » -> « *Character devices* » -> « *Serial drivers* » -> « *ARM AMBA PL011 serial port support* ») pour disposer des traces du kernel sur le port série si je le désire (en supprimant alors l'option « **quiet** » de la ligne de commande),

- **CONFIG_GPIO_SYSFS** (« *Device Drivers* » -> « *GPIO support* » -> « */sys/class/gpio* ») pour pouvoir accéder facilement aux GPIO depuis l'espace utilisateur,

- **CONFIG_EXT4_FS** (« *File systems* » -> « *The Extended 4 (ext4) filesystem* ») est nécessaire pour monter la partition racine de l'arborescence des fichiers.

La taille du noyau (**config-04**) a légèrement diminué : 1857328 octets, et la moyenne des durées de boot est de **2,614 secondes**.

4.4 Réduction de la taille du noyau

Il est temps de s'occuper de la taille de l'image du kernel. Le temps de chargement en RAM est alourdi par une image trop grosse. Le compilateur Gcc est capable d'optimiser le code qu'il produit pour le rendre le plus rapide possible (option **-O2**) ou le plus compact possible (option **-Os**). En choisissant cette dernière possibilité (« *General Setup* » -> « *Optimise for size* ») on réduit l'image du kernel.

Une fois le noyau compilé, il est compressé et s'auto-décompressera en Ram. Il faut choisir un algorithme de compression du noyau (« *General Setup* » -> « *Kernel Compression Mode* ») qui optimise à la fois la taille de l'image et la vitesse de décompression. J'ai testé les quatre algorithmes proposés :

Algorithme	Configuration	Taille kernel	Durée boot
Gzip	**config-05**	1 711 168 octets	2,631
LZMA	**config-06**	1 291 928 octets	4,022
XZ	**config-07**	1 230 704 octets	3.579
LZO	**config-08**	1 860 696 octets	2.522

L'option LZO n'est pas la plus efficace en terme de réduction de taille du code, mais elle décompresse le noyau très rapidement (**2,522 secondes**). C'est donc celle que nous retiendrons.

4.5 Options avancées

Dans le menu « *General Setup* », un sous-menu « *Configure standard kernel features (expert users)* » permet de modifier certaines options globales pour le noyau, avec des impacts sur plusieurs sous-systèmes. J'y ai désactivé :

- **CONFIG_KALLSYMS** (« *Load all symbols for debugging/ksymoops* ») : la table des symboles alourdit le noyau et n'est pas utile sur un système en production.

- **CONFIG_BUG** (« *BUG() support* ») nous pouvons espérer que sur un système embarqué correctement ajusté, il n'y aura pas de déclenchement des macros **WARN()** ou **BUG()** – qui produisent les traces des *Oops* du kernel – et nous pouvons désactiver leur support.

- **CONFIG_ELF_CORE** (« *Enable ELF core dumps* ») les fichiers **core** produits lors du plantage d'un processus n'ont pas d'intérêt sur un système embarqué. Éliminons la possibilité d'en produire.

Enfin, j'ai réduit (« *General Setup* » -> « *Kernel log buffer size* ») la taille du buffer interne dans lequel le noyau stocke ses messages consultables avec la commande **dmesg**.

Le noyau **(config-09)** a diminué de taille pour occuper 1 442 544 octets et la moyenne des temps de démarrage est de **2,464 secondes**.

4.6 Préemptibilité du kernel

Le noyau Linux *Vanilla* peut être compilé avec un niveau de préemptibilité plus ou mois élevé configurable

dans le menu « *Kernel Features* » -> « *Preemption model* ».

- **CONFIG_PREEMPT_NONE** (« *No Forced Preemption* ») : le code noyau n'est pas préemptible, une fois qu'un appel-système a débuté, il s'exécutera entièrement sans que le thread appelant ne puisse être préempté au profit d'une autre tâche devenue prête entre temps.

- **CONFIG_PREEMPT_VOLUNTARY** (« *Voluntary Kernel Preemption* ») : les appels-système longs appellent régulièrement l'ordonnanceur – via la fonction **schedule()** – pour céder le processeur si une tâche est devenue plus prioritaire que le thread appelant.

- **CONFIG_PREEMPT** (« *Preemptible Kernel* ») : une interruption survenue pendant un appel-système peut à tout moment réveiller une tâche plus prioritaire que celle ayant l'invoqué l'appel-système ; on commutera vers cette nouvelle tâche dès le retour d'interruption en laissant le thread précédent en attente. Le patch *PREEMPT_RT* peut améliorer encore ce comportement en réduisant la latence maximale entre le déclenchement de l'interruption et la commutation.

Option	Configuration	Taille kernel	Durée boot
PREEMPT_NONE	**config-11**	1 398 768 octets	2,504
PREEMPT_VOLUNTARY	**config-10**	1 401 184 octets	2,512
PREEMPT	**config-09**	1 442 544 octets	2,464

Nous conservons donc un noyau préemptible, ce qui améliore d'autant la réactivité pour les entrées-sorties et le comportement temps réel souple du système.

J'ai également testé quelques options comme **CONFIG_NO_HZ** (« *General setup* » -> « *Timers subsystem* » -> « *Tickless System* »), **CONFIG_COMPACTION** (« *Kernel features* » -> « *Allow for memory compaction* »), **CONFIG_CROSS_MEMORY_ATTACH** (« *Kernel features* » -> « *Cross Memory Support* ») dont l'impact sur le temps de boot est resté négligeable.

4.7 Douloureux choix concernant le driver réseau

Dans les drivers spécifiques du Raspberry Pi, il en est un dont la configuration sous forme de module provoque des erreurs de compilation du kernel !

Il s'agit de **CONFIG_USB_DWCOTG** (« *Device drivers* » -> « *USB Support* » -> « *Synopsis DWC host support* ») qui implémente le support USB du *system-on-chip*. Outre le support USB, il permet également d'utiliser l'interface Ethernet (via le driver **smsc95xxx**). Il serait pourtant très intéressant de compiler ce driver sous forme de module, car son initialisation dure près d'une demi-seconde... On pourrait ainsi afficher d'abord un écran d'accueil pour l'utilisateur ou démarrer des tâches urgentes, et retarder l'initialisation en ne chargeant qu'ultérieurement le module.

Faisant exception à ma règle qui consiste à ne pas modifier le code noyau, j'ai préparé un patch qui permet de compiler ce driver en module. Néanmoins le code de ce driver est de piètre qualité (entre autres, il y a des dépendances exagérées entre des modules très différents) et le chargement conduit parfois à des erreur « *kernel panic* »... Le palliatif actuel, sortir le fichier **dwc-otg.ko** des modules du kernel pour s'assurer qu'il ne soit pas chargé avant la fin du boot ne me satisfait pas, et le travail sur ce sujet est toujours en cours.

Le noyau obtenu (**config-13**) mesure 1 289 896 octets et la durée moyenne de démarrage en **1,994** secondes.

Nous devons donc faire un choix entre un *boot* en moins de deux secondes et l'utilisation du réseau Ethernet ainsi que des périphériques USB.

4.8 Ce qui ne marche pas...

Bien entendu, dans la longue liste des expériences menées pour réduire la durée de boot, plusieurs tentatives se sont révélées infructueuses (sans compter celles où le noyau a refusé de booter) :

- placer sur la carte SD un noyau décompressé à l'aide du script **imagetool-uncompressed.py** que l'on peut trouver dans les dépôts kernel du Raspberry Pi. La taille du noyau étant augmentée, le temps de transfert en RAM s'allonge sensiblement, même s'il n'est plus besoin de réaliser l'étape de décompression.

- utiliser un système de fichiers **ext2** plutôt qu'**ext4** : ceci permet d'éviter des écritures supplémentaires sur la carte SD pour la tenue du journal, néanmoins on ne gagne pas de temps de boot.

- utiliser une carte SD plus rapide (classe 10 au lieu de classe 4) : étonnamment cela n'influe pas sur le temps de chargement, les durées sont globalement identiques. Il est possible que l'utilisation d'une carte

autorisant un débit plus élevé améliore la fluidité des accès au système de fichiers après le boot, néanmoins la différence n'est pas perceptible pour le temps de chargement du noyau et le montage du *rootfs*.

- etc

5. Amélioration de la durée subjective de boot

Si l'on s'intéresse à la durée **subjective** de démarrage, c'est à dire le temps au bout duquel l'utilisateur perçoit une réaction du système, plusieurs astuces permettent de donner l'illusion que le système est « vivant » même s'il n'est pas complètement initialisé.

5.1 Remplacement du logo Linux

Depuis la version 2.0 de Linux, en 1996, le fameux pingouin Tux apparaît lorsque le système démarre avec une console dans un *framebuffer*. Les développeurs qui ont adapté le kernel au Raspberry Pi l'ont remplacé par une framboise. Ce logo est affiché très rapidement, environ une seconde après la mise sous tension, et nous pouvons le remplacer afin de faire apparaître un écran personnalisé. Nul n'est besoin en effet de se limiter au format 80x80 initial.

Attention, l'image que nous allons afficher va se trouver incorporée statiquement dans le noyau. Elle doit donc être placée sous une licence compatible avec la GPLv2. Autrement dit, je déconseille fortement d'inclure un logo d'entreprise à ce niveau, sauf à considérer que ce logo peut être librement copié, modifié, redistribué, et exploité dans d'autres situations...

Nous avons le choix entre une image en noir-et-blanc au format PBM (*Portable BitMap*), ce qui peut convenir parfaitement pour affiche un gros titre avec le nom du produit, en 16 couleurs standards ou en 224 couleurs au format PPM (*Portable PixMap*). Pour manipuler ce type de fichiers, il est conseillé d'utiliser les outils du package NetPBM.

Créons avec Gimp une image noir et blanc au format 720x480, contenant un texte très large, par exemple « *Chargement système Linux* ». L'image doit être convertie en mode« *Couleurs indexées* » en utilisant « *Palette noir & blanc (1 bit)* ». Rien ne nous empêche d'utiliser 16 ou 224 couleurs, ou encore un format plus élevé comme 1280x768 mais le temps de chargement sera d'autant plus long que le fichier sera volumineux. Sauvegardons le fichier sous le nom **`message-logo.ppm`** en format « *Brut* ».

La meilleure solution consiste à ajouter votre fichier en plus de ceux déjà existants. Néanmoins, pour simplifier notre expérience, nous pouvons également écraser le fichier **`logo_linux_mono`** original :

```
$ cd  drivers/video/logo/
$ mv  logo_linux_mono.pbm  logo_linux_mono.pbm-original
$ pnmquant  2  ~/message-logo.ppm  |  pnmtoplainpnm  >  logo_linux_mono.pbm
```

Recompilons le noyau en intégrant statiquement les options suivantes du menu « *Device drivers* » -> « *Graphics support* »

- **`CONFIG_FB`** « *Support for frame buffer devices* »

- **`CONFIG_FB_BCM2708`** « *BCM2708 framebuffer support* »

- **`CONFIG_FRAMEBUFFER_CONSOLE`** « *Console display driver support* » -> « *Frame buffer Console support* »

- **`CONFIG_LOGO`** « *Bootup Logo* »

- **`LOGO_LINUX_MONO`** « *Standard black and white Linux logo* »

Sur le Raspberry Pi « Test », il faut éditer le fichier **`config.txt`** se trouvant sur la première partition pour indiquer les dimensions de *framebuffer* désirées en remplissant les lignes suivantes :

```
framebuffer_width=720
framebuffer_height=480
```

Le noyau résultant (**`config-14`**) mesure 1 333 752 octets, et la durée de boot moyenne avant l'activation du programme utilisateur est de 2,062 secondes.

Le logo est présenté sur le *framebuffer* bien avant le démarrage du programme utilisateur (même si le contrôleur vidéo du Raspberry Pi met environ 2 secondes à afficher l'écran).

5.2 Utilisation simple du framebuffer

Depuis l'espace utilisateur également, nous pouvons afficher rapidement une image grâce au *framebuffer*.

Celle-ci venant en remplacement du logo précédent ajoute un élément dynamique accompagnant le démarrage du système. Un choix judicieux d'images complémentaires pourra égayer le boot en attendant que l'application « métier » soit définitivement prête.

Le plus simple consiste à copier le contenu d'un fichier graphique directement dans **`/dev/fb0`**. Toutefois, la configuration directe du *framebuffer* (résolution, profondeur, etc.) est assez compliquée. Je propose donc les opérations suivantes :

- employer sur le Raspberry Pi « *Test* » une visionneuse graphique sur *framebuffer* pour y afficher le contenu d'un fichier de notre choix ;

- sauvegarder dans un fichier le contenu brut du *framebuffer* (en faisant une copie depuis **`/dev/fb0`** ;

- au boot, restaurer simplement le contenu de ce fichier (en le copiant vers **`/dev/fb0`**).

J'ai choisi la visionneuse **`fbvis`**, car elle est très simple à compiler, soit nativement sur la cible, soit en cross-compilation. Voici une compilation native sur Raspberry Pi :

```
(Test)# git  clone  git://repo.or.cz/fbvis.git
Cloning into 'fbvis'...
  [...]
(Test)# cd  fbvis/
(Test)# make
```

Pour une cross-compilation, il suffirait d'éditer le fichier **`Makefile`**, et de remplacer la ligne

```
CC = cc
```

par

```
CC=/usr/local/cross/rpi/bin/arm-linux-gcc
```

en ajustant le chemin et le nom de la chaîne de cross-compilation.

Installons l'utilitaire sur la cible.

```
(Test)# cp  fbvis  /usr/local/bin/
```

Nous copions aussi sur notre cible un fichier graphique (**`RPI.jpg`**) de la dimension choisie.

Affichons l'image, puis sauvegardons le contenu du *framebuffer* :

```
(Test)# echo  q  |  fbvis /usr/local/share/RPI.jpg
 FBVIS:     file:/usr/local/share/RPI.jpg
(Test)# dd  if=/dev/fb0  of=/usr/local/share/RPI.raw
1350+0 records in
1350+0 records out
691200 bytes (691 kB) copied, 0.0453169 s, 15.3 MB/s
(Test)#
```

Nous pouvons noter le volume du transfert (691200 octets) pour optimiser l'appel à **`dd`** lors de la restauration du fichier. Celle-ci peut avoir lieu par exemple dans le script de démarrage **`/etc/init.d/rcS`** en y ajoutant la ligne :

```
dd  if=/usr/local/share/RPI.raw  of=/dev/fb0  bs=691200
```

Naturellement, on peut répéter cette opération à plusieurs reprises, en changeant d'image au gré des différentes étapes d'initialisation restantes. Rappelons qu'à ce niveau, l'application principale a déjà démarré.

Conclusion

Nous avons réussi à remplir plusieurs objectifs :

- une application de l'espace utilisateur démarre et fournit un résultat visible (activation d'une GPIO) moins de deux secondes après la mise sous tension, ceci est suffisant pour une application non-graphique utilisant des entrées-sorties sur ports GPIO, liaisons série, console texte...

- nous pouvons faire afficher par le kernel un écran graphique d'accueil environ une secondes après mise sous tension, ceci au prix d'un retard de quelques centièmes de secondes pour l'application utilisateur ultérieure ;

- l'application utilisateur peut commencer à afficher très rapidement quelques pages graphiques grâce à une écriture directe dans le *framebuffer*, avant d'initialiser son interface (construite par exemple avec DirectFB, EFL, Qt...).

On notera que les meilleures optimisations ont été obtenues en agissant sur les scripts d'initialisation, en rapprochant le point de démarrage du code métier le plus près possible de la fin du boot du kernel. Les améliorations que nous avons obtenues sur le boot du noyau ne jouent que pour une à deux secondes, ce

qui peut être négligeable dans certains cas.

Pour aller plus loin...

Fichiers de configuration et sources des programmes : **http://www.blaess.fr/christophe/articles/files-glmf/**

« *Linux embarqué* », Pierre Ficheux, Éric Bénard, éditions Eyrolles.

« *Linux embarqué* » Gilles Blanc , éditions Pearson.

« *Boot time* » : **http://elinux.org/Boot_Time**

« *Les distributions embarquées pour Raspberry Pi* » Pierre Ficheux, Open Silicium numéro 7.

« *Raspberry Pi from scratch* » Christophe Blaess, Gnu/Linux Magazine France numéros 155 et 158.