Qemu : Comment émuler une nouvelle machine ? Cas de l'APF27

Yvan Roch

Dans la première partie, nous avons abordé deux aspects fondamentaux de Qemu, la recompilation dynamique du code émulé et la gestion du temps. Cependant, nous sommes restés sur le banc des spectateurs. Bien que Qemu 1.0 supporte 27 machines ARM différente, aucune n'est basée sur les SoC Freescale™ de la famille i.MX. Dans ce second volet, nous passerons à la pratique en créant « from scratch » le support de l'émulation d'une nouvelle machine construite autour d'un i.MX27 : la carte Armadeus APF27.

1. Introduction

D'autres sujets périphériques seront aussi abordés. En effet, la connaissance interne du noyau Linux, ou au moins certaines parties, est nécessaire à la création de modules d'émulation matérielle. Nous ferons quelques détours dans le processus de boot de Linux sur une architecture ARM ainsi qu'une petite visite de ses horloges internes.

1.1 D'autres outils bien utiles

En première partie, un certain nombre d'outils ont été présentés. Ils étaient suffisants dans un contexte d'exploration du code source et seront encore très utiles dans un cadre de développement. Cependant, il faut étendre notre boîte à outils si l'on veut mener à bien notre nouvelle tâche.

Dans cette seconde partie, nous développerons des modules d'émulation de matériel qui devront se comporter, d'un point de vue du noyau Linux, comme leurs pendants physiques. Comme nous le verrons, cette tâche peut s'avérer assez difficile, car, contrairement au développement d'application classique, le système n'est pas encore opérationnel lorsque notre code s'exécute. C'est particulièrement le cas de l'émulation de la mémoire qui est l'un des composants à être initialisé en premier par le noyau Linux avant même qu'une console ne soit disponible. Lors de la phase de mise au point et de correction de bugs, qui surviendront inéluctablement, on se retrouve alors « dans le noir » total : le code ne fonctionne pas, mais il est impossible de savoir où, ni pourquoi. Heureusement, Linux et ses options de mise au point vont venir à notre secours. Ces options sont activées par des paramètres passés au noyau et éventuellement en activant certaines options de compilation du noyau. Dans ce cas, une recompilation du noyau sera alors nécessaire pour en bénéficier. Passons les en revue :

– **`loglevel=8`** permet d'obtenir l'affichage de tous les messages émis par le noyau jusqu'aux messages de type **`KERN_DEBUG`**.

– **`bootmem_debug`** permet d'obtenir des messages de débuggage du sous-système **`bootmem`**. Celui-ci est responsable de l'allocation et de la configuration de la mémoire physique.

– **`mminit_loglevel=4`** offre la possibilité d'obtenir tous les messages supplémentaires concernant l'initialisation et la vérification de la mémoire. Il nécessite l'activation de l'option de compilation du noyau **`CONFIG_DEBUG_MEMORY_INIT`** ainsi qu'une recompilation. Les messages produits sont de type **`KERN_DEBUG`**, il faut donc que le paramètre **`loglevel=8`** soit lui aussi passé au noyau.

– **`earlyprintk`** provoque l'affichage des messages du noyau sur une console « précoce ». Par défaut, aucun message du noyau n'est affiché avant que la console système soit opérationnelle. Cette console est dans notre cas un port série et c'est uniquement lorsque ce port série fonctionne que la console système remplit ses offices. Tous les messages précédents sont alors affichés. Si le noyau plante avant que la console système fonctionne, aucun message ne sera visible, y compris ceux qui sont produits avec les paramètres ci-dessus. On est alors « dans le noir ». Un mécanisme est prévu dans le noyau pour palier cette

fâcheuse situation. L'option de compilation **CONFIG_EARLY_PRINTK** permet la création d'une console « précoce » (early console) et le paramètre du noyau **earlyprintk** permet de l'activer. Cette console précoce affichera les messages du noyau avant même que la console système fonctionne.

Qemu dispose d'une option réservée au passage de paramètres au noyau Linux : **-append**. Pour bénéficier des options de débuggage de Linux ci-dessus, il faudra donc ajouter à la ligne de commande de Qemu : **-append "console=ttymxc0 loglevel=8 bootmem_debug mminit_loglevel=4 earlyprintk"**.

Il arrive malheureusement que cela ne suffise pas à déterminer l'origine d'un bug vicieux. Cette fois-ci, c'est Qemu qui va voler à notre secours grâce à l'une de ses fonctionnalités très intéressante : son agent GDB. L'exécution et le débuggage du noyau Linux sous le contrôle de Qemu et GDB ont été largement couverts par l'article de Pierre Ficheux [1] ainsi qu'à la page 281 de son ouvrage [2]. Pour les lecteurs ne disposant pas de cette littérature vivement conseillée, je vais en présenter les grandes lignes.

GDB est conçu pour fonctionner dans certains cas en mode client/serveur. Cette utilisation est particulièrement avantageuse dans un contexte embarqué, lorsque la plate-forme cible ne dispose pas assez de ressources CPU et mémoire pour exécuter l'intégralité de GDB, qui est assez gourmand ; ou bien qu'elle ne comporte pas une interface utilisateur pratique (pas d'interface réseau et un port série unique).

Dans ce mode de fonctionnement, le client est constitué de la partie lourde de GDB, c'est-à-dire le programme GDB lui-même ainsi que le fichier ELF binaire à mettre au point. Ce dernier doit comporter les symboles de débuggage générés par GCC (options **-O0 -g**). Cette partie cliente communique, via le protocole GDB, avec un serveur GDB, autrement appelé agent GDB. Suivant la situation cet agent prendra différentes formes :

– Le programme **gdbserver** compilé pour l'architecture cible. C'est un petit programme d'une centaine de Ko pour sa version 7.1 ARM strippée. Il communique avec le client GDB et exécute sous son contrôle le programme à mettre au point qui n'a pas besoin des symboles de débuggage. Il reste ainsi de taille minimale.

– KGDB, le débuggeur officiel de Linux est aussi un agent GDB, mais il ne rentre pas dans le cadre cet article.

– Qemu qui comporte un agent GDB activable avec l'option **-s**. Dans ce cas, le code invité exécuté par Qemu pourra l'être sous le contrôle d'un GDB client, qui sera dans notre cas le programme GDB pour l'architecture ARM compilé pour la plate-forme x86. Ce dernier est fourni par le BSP Armadeus basé sur Buildroot que nous utiliserons dans la partie mise en pratique.

Avant d'aller plus loin, évitons deux confusions possibles.

– Bien que le noyau Linux soit débuggable avec cette dernière solution, elle n'a rien à voir avec KGDB. D'ailleurs, le noyau Linux n'a pas besoin d'avoir l'option de compilation **CONFIG_KGDB** activée. C'est possible, car comme nous l'avons vu le mois dernier, Qemu contrôle chaque instruction du processeur virtuel, qu'elle soit dans le noyau ou pas.

– Ce n'est pas Qemu qui est débuggué et exécuté sous le contrôle de GDB. C'est le code invité exécuté par le processeur virtuel de Qemu qui l'est. Pour débugger Qemu lui-même, il faut le compiler avec les options de débuggage et l'exécuter sous le contrôle d'un GDB hôte, c'est-à-dire, dans notre cas, un GDB x86.

Donc, pour mettre en œuvre cette solution, nous aurons besoin :

– D'un noyau Linux pour l'architecture invitée compilé avec l'option **CONFIG_DEBUG_INFO**.

– D'un client GDB pour architecture ARM compilé pour x86. Ce dernier est fourni par la chaîne de compilation croisée construite par Buildroot dans le BSP Armadeus.

Voici comment lancer Qemu en activant l'agent GDB :

```
$ qemu-system-arm -M apf27 -kernel apf27-linux.bin -append "console=ttymxc0 loglevel=8
bootmem_debug mminit_loglevel=4 earlyprintk" -initrd apf27-rootfs.cpio -nographic -s -S
```

Les options qui nous intéressent pour l'instant sont **-s** et **-S**. Les autres seront commentées lors des travaux pratiques. **-s** indique à Qemu qu'il doit activer l'agent GDB. **-S** demande à Qemu de ne pas démarrer le processeur virtuel. Il démarrera lorsque le client GDB se connectera.

Maintenant, lançons le client GDB :

```
$ arm-linux-gdb vmlinux
```

arm-linux-gdb est client GDB pour architecture ARM compilé pour x86. Il est situé dans le répertoire **$ARMADEUS_ROOT/buildroot/output/build/staging_dir/usr/bin/**, **$ARMADEUS_ROOT** étant le répertoire racine du BSP Armadeus. **vmlinux** est le fichier binaire contenant le noyau Linux avec ses symboles de débuggage. Il correspond au fichier **apf27-linux.bin** de la commande précédente à la différence que **apf27-linux.bin** est un fichier U-Boot *ulmage*, alors que **vmlinux** est un fichier ELF, seul format accepté par GDB.

Ensuite sous l'invite GDB, les commandes suivantes seront lancées :

```
(gdb) b bootmem_init
(gdb) target remote 127.0.0.1:1234
(gdb) c
```

La première commande placera un breakpoint sur la fonction **bootmem_init()**, admettons que ce soit elle qui nécessite une mise au point. La deuxième connectera le client GDB à l'agent GDB Qemu. La dernière lancera l'exécution du noyau et une bonne séance de débuggage pourra être engagée.

Cette solution est, certes, efficace, mais lourde dans son utilisation, d'autant plus que toute la procédure est à refaire à chaque compilation. Pour terminer sur les outils liés au noyau Linux, voici une description précise des options de compilation nécessaires, pour éviter de se perdre lors de l'exécution de **make menuconfig**.

Toutes les options citées sont dépendantes **CONFIG_DEBUG_KERNEL** que l'on trouve sous ***Kernel hacking ---> [*] Kernel debugging***. **CONFIG_DEBUG_MEMORY_INIT** sera activée avec ***Kernel hacking --->[*] Debug memory initialisation***. Ensuite **CONFIG_EARLY_PRINTK** nécessite **CONFIG_DEBUG_LL** dans ***Kernel hacking --->[*] Kernel low-level debugging functions*** puis ***Kernel hacking ---> [*] Early printk***. Pour finir, **CONFIG_DEBUG_INFO** est activée par ***Kernel hacking ---> [*] Compile the kernel with debug info***.

Après ces ustensiles de dénoyautage, il en reste encore quelques outils à présenter. Tout d'abord, comme le but avoué est de développer un embryon d'émulation de SoC Freescale i.MX27, un document capital est le « *MCIMX27 Multimedia Applications Processor Reference Manual* » [3]. C'est le manuel de référence de l'i.MX27, un magnifique document de plus de 1700 pages quelque peu indigeste. Mais lorsqu'on veut émuler du matériel, il est absolument nécessaire de savoir comment il fonctionne et comment il se comporte. Ainsi la lecture du chapitre 31 sur les « General Purpose Timer (GPT) » semble être un minimum.

Puis la documentation de la carte Armadeus APF27 [4] est aussi un document incontournable. Il donne des informations de mise en œuvre du SoC, comme la cartographie mémoire du système.

Ensuite, la lecture du code source de Qemu [5] est excellente pour la compréhension, l'inspiration et pour ne pas réinventer la roue. Il en est de même pour les archives de la liste de diffusion de développement de Qemu [6]. Bien qu'il n'existe aucun support officiel des SoC Freescale i.MX, j'y ai quand même trouvé, sous licence GPL, deux des trois modules nécessaire à un système minimal.

Une autre lecture indispensable est le code source du noyau Linux et particulièrement des portions de code qui utilisent le matériel que l'on tente d'émuler. Pour reprendre le cas des timers de l'i.MX27, le manuel de référence nous apprend qu'un tel SoC comporte six timers identiques et indépendants qui possèdent deux modes de comptage. En inspectant les sources de Linux en général et plus particulièrement de pilote de l'horloge de la famille i.MX (**arch/arm/plat-mxc/time.c**) on se rend compte qu'il n'utilise qu'un seul de ces timers et, de plus, avec un seul mode de comptage. Notre but étant d'émuler l'i.MX27 pour exécuter un système GNU/Linux, il semble vain d'émulé l'intégralité des timers et de leurs fonctionnalités. Seules celles qui sont utilisées par Linux méritent qu'on s'y attarde, mais pour le savoir, la lecture du code est nécessaire. C'est bien entendu là quelque chose de spécifique à la plateforme que vous émulez ainsi qu'au noyau du système qui sera utilisé et qui devra être adapté le cas échéant.

Pour finir, la lecture du code source de U-Boot, dans sa version patchée et adaptée à l'APF27, fournit des renseignements vitaux sur la configuration du SoC effectuée avant le boot de Linux. Cela concerne particulièrement les zones de mémoire RAM et la pléthore d'horloges de l'i.MX27, qui sans une initialisation adéquate, provoquent un fonctionnement erratique du noyau.

Dans leurs versions spécifiques à la carte APF27, les sources de Linux et de U-Boot sont consultables au sein du BSP Armadeus, respectivement dans les répertoires **$ARMADEUS_ROOT/buildroot/output/build/linux-2.6.38.1** et **$ARMADEUS_ROOT/buildroot/output/build/u-boot-1.3.4**.

2. Les modules de Qemu

Nous voici puissamment outillés, mais avant de descendre dans les bas-fonds du hard, étudions comment Qemu gère et architecture l'émulation matérielle de périphériques. Le mois dernier, nous avons abordé, en détails, l'émulation des processeurs et particulièrement ceux de l'architecture ARM. Le code dédié à cette tâche est situé dans les répertoires **target-ARCH**, où **ARCH** est une architecture particulière comme **arm** ou **m68k**. Le code concernant l'émulation matérielle de machines et de périphériques est placé dans le répertoire **hw**. Il comporte plus 50 modules de machine et 200 modules de périphérique. Un module Qemu est une portion de code autonome, qui remplit une fonctionnalité particulière et qui peut être facilement intégrée au reste de Qemu. Il existe quatre types de modules distincts :

– Les modules de périphériques de type bloc. Ce sont des modules génériques utilisés dans l'implémentation de l'émulation de périphériques de ce type. Ils interviennent, par exemple, dans la gestion des images de disques et fournissent à Qemu une abstraction de celles-ci.

– Les modules de type Qapi. Ce sont des modules de Qemu répondant à la nouvelle Api de communication de Qemu (http://wiki.qemu.org/Features/QAPI), Qapi. Aucun module ne l'utilise actuellement.

– Les modules de type machine. Ils émulent une machine complète telle que l'APF27. Leur rôle est d'instancier et d'initialiser le processeur virtuel de la machine, sa mémoire et tous ses modules de périphériques émulés.

– Les modules de périphériques. Ils émulent un périphérique matériel particulier tel que les ports série de l'i.MX27 ou ses timers.

Dans cet article, nous nous intéresserons uniquement aux deux derniers types. Le module de machine APF27 et le module de périphérique du timer de l'i.MX27 seront étudiés dans les moindres détails. Ils s'enregistrent auprès de Qemu via les macros **`machine_init()`** et **`device_init()`** sans avoir aucune autre modification à faire dans le code de Qemu, si ce n'est l'intégration au système de compilation. Cette élégance est possible grâce à l'attribut GCC **`constructor`** que nous verrons plus loin.

3. Architecture logicielle du système émulé

Un système émulé complet, dans la terminologie de Qemu est appelé une machine. La liste des machines disponibles pour une certaine architecture de processeur peut être obtenue par la commande :

```
$ qemu-system-arm -M ?
Supported machines are:
apf27        Armadeus APF27 Board (ARM926EJ-S)
collie       Collie PDA (SA-1110)
...
versatilepb  ARM Versatile/PB (ARM926EJ-S)
versatileab  ARM Versatile/AB (ARM926EJ-S)
vexpress-a9  ARM Versatile Express for Cortex-A9
z2           Zipit Z2 (PXA27x)
```

Dans la version 1.0 officielle de Qemu, 27 machines ARM sont implémentées. La sortie de console ci-dessus montre l'ajout de la carte APF27. Une machine Qemu est implémentée sous la forme d'un module de machine. C'est le module fédérateur qui crée, dans notre cas, les composants suivants :

– Un ou deux blocs de mémoire RAM de 64 Mo ou 128 Mo.

– Un processeur virtuel de type ARM926EJ-S.

– Un module SoC i.MX27.

– Un module de timer i.MX27.

– Un module de contrôleur d'interruption i.MX27.

– Deux modules UART i.MX27

Ces modules sont tous des modules de périphériques. Comme je l'ai mentionné dans l'introduction, la machine obtenue avec ces différents composants est vraiment minimale et rien ne peut être lui être enlevé. Elle permet simplement d'exécuter un système GNU/Linux basique. Les programmes fondamentaux tels **`init`**, **`bash`** et **`ps`** fonctionneront, mais il est bien évident que ceux utilisant un périphérique qui n'est pas implémenté ne fonctionneront pas. Il manque de nombreux périphériques à la machine émulée APF27 : le contrôleur Flash NAND, les contrôleurs USB, la carte réseau, le contrôleur graphique et bien d'autres encore.

Le module SoC i.MX27 sert uniquement à émuler certains registres de l'i.MX27 pour que le noyau Linux puisse se configurer correctement. Y sont présents :

– Le registre CID ou Chip ID Register. C'est un registre en lecture seule initialisé au démarrage du processeur que Linux utilise pour identifier le modèle exact de SoC.

– Les registres CCM ou Clock Control Module. Ils constituent un ensemble de registres qui servent à paramétrer les horloges physiques de l'i.MX27. Sur une carte réelle, c'est U-Boot qui les configurent pour obtenir les différentes fréquences de références nécessaires au bon fonctionnement du système. Ensuite ces registres sont lus par Linux pour qu'il configure ses différentes horloges internes. Le module Qemu devra donc lui fournir les mêmes valeurs que sur un système réel. Ces paramètres sont importants, car ils conditionnent la source de temps utilisé par le timer système.

Le module de timer émule un timer de l'i.MX27 qui en comporte six. Le noyau Linux utilise le premier comme source de temps pour tout le système. Sans lui, le noyau ne peut pas fonctionner. Il en va de même pour le module de contrôleur d'interruption qui émule le contrôleur d'interruption de l'i.MX27. En effet, sans ce module, les interruptions du timer ne sont pas traitées et celui-ci devient inopérant. Les modules UART émulent le premier et le troisième port série de l'i.MX27. Le premier sera utilisé comme console système. Étant les seuls périphériques d'entée/sortie, ils sortent le système de l'autisme.

Les modules SoC et timer seront complètement présentés et j'en suis le modeste auteur. Le module UART et celui de contrôleur d'interruption ont été trouvés sur la liste de diffusion de développement de Qemu. Leur auteur, Peter Chubb, un docteur en informatique australien, tente d'intégrer le support de la carte KZM à la « main line » de Qemu via une série de patchs [7]. Cette carte est fabriquée par la société japonaise Kyoto Microcomputer Co., qui l'a conçue autour d'un SoC i.MX31. Les UART des SoC i.MX31 et i.MX27 sont identiques, et leurs contrôleurs d'interruption se ressemblent beaucoup, donc les modules développés pour l'i.MX31 sont réutilisables pour l'i.MX27. Ces modules sont sous licence GPL et je remercie chaleureusement Peter Chubb de les avoir publiés. Le module UART est un module de périphérique et la structure de son code est similaire à celle du module timer. Le module de contrôleur d'interruption aurait mérité un approfondissement, mais le nombre de pages du magazine étant limité, le lecteur aura trouvé là un sujet de perfectionnement.

4. La carte principale

La carte APF27 est implémenté sous la forme d'un module de machine Qemu constitué du fichier **hw/apf27.c**. Son rôle est de créer et d'enregistrer une nouvelle machine auprès de Qemu. Cette nouvelle machine sera instanciée grâce à l'option **-M apf27** de la commande **qemu-system-arm**, l'émulateur ARM Qemu. Lors de son initialisation, le module créera et initialisera tous les composants dont il a besoin, puis il chargera l'image binaire du noyau Linux qui a été spécifiée dans la ligne de commande de lancement, avec l'option **-kernel**. Éventuellement, il chargera aussi l'image du disque RAM initial, si celle-ci a été précisée avec l'option **-initrd**. Le système invité sera alors prêt à démarrer. La suite du processus a été détaillée dans la section 2.3 de la première partie de l'article [8].

Si le code présenté dans cet article est dépourvu de tout commentaire, ce n'est pas par souci de mimétisme culturel avec le projet Qemu. C'est simplement pour être plus clair et prendre moins de place. Le patch téléchargeable, présenté dans la section mise en œuvre, contient lui, du code largement commenté.

Voici le code de ce premier module **apf27.c**, précédé de son fichier d'entête **apf27.h** :

```
#ifdef CONFIG_APF27
#define APF27_BOARD_ID                  1698
#define AFP27_DRAM_BANK1_ADDR    0xA0000000
#define AFP27_DRAM_BANK2_ADDR    0xB0000000
#define APF27_DRAM_BANK_SIZE            0x04000000

#define MX27_AITC_BASE_ADDR             0x10040000
#define MX27_AIPI_BASE_ADDR             0x10000000
#define MX27_UART1_BASE_ADDR     (MX27_AIPI_BASE_ADDR + 0x0A000)
#define MX27_UART3_BASE_ADDR     (MX27_AIPI_BASE_ADDR + 0x0C000)
#define MX27_GPT1_BASE_ADDR              (MX27_AIPI_BASE_ADDR + 0x03000)
#define MX27_CCM_BASE_ADDR               (MX27_AIPI_BASE_ADDR + 0x27000)
#endif

#ifdef CONFIG_APF27
#include "sysbus.h"
#include "exec-memory.h"
#include "arm-misc.h"
#include "boards.h"
#include "apf27.h"

static struct arm_boot_info apf27_binfo;

static void apf27_init(ram_addr_t ram_size, const char *boot_device,
             const char *kernel_filename, const char *kernel_cmdline,
             const char *initrd_filename, const char *cpu_model) {
       MemoryRegion *address_space_mem, *ram_bank1, *ram_bank2;
       CPUState *env;
       qemu_irq *cpu_pic;
       DeviceState *dev;
       if (ram_size != APF27_DRAM_BANK_SIZE && ram_size !=APF27_DRAM_BANK_SIZE*2)
       {
              fprintf(stderr, "On APF27, memory size must be %uM or %uM only!!!\n",
                     APF27_DRAM_BANK_SIZE >> 20,
                     (APF27_DRAM_BANK_SIZE*2) >> 20);
              exit(1);
       }
       address_space_mem = get_system_memory();
       ram_bank1 = g_new(MemoryRegion, 1);
       memory_region_init_ram(ram_bank1, NULL, "apf27.ram_bank1",
              APF27_DRAM_BANK_SIZE);
```

```
        memory_region_add_subregion(address_space_mem, AFP27_DRAM_BANK1_ADDR,
                ram_bank1);
        if (ram_size == APF27_DRAM_BANK_SIZE * 2){
                ram_bank2 = g_new(MemoryRegion, 1);
                memory_region_init_ram(ram_bank2, NULL, "apf27.ram_bank2",
                        APF27_DRAM_BANK_SIZE);
                memory_region_add_subregion(address_space_mem,
                        AFP27_DRAM_BANK2_ADDR, ram_bank2);
        }
        env = cpu_init("arm926");
        if (!env) {
                fprintf(stderr, "Unable to find CPU definition\n");
                exit(1);
        }
        cpu_pic = arm_pic_init_cpu(env);
        dev = sysbus_create_varargs("imx_int", MX27_AITC_BASE_ADDR, cpu_pic[0],
                cpu_pic[1], NULL);
        sysbus_create_simple("imx27_timer", MX27_GPT1_BASE_ADDR,
                qdev_get_gpio_in(dev, 26));
        sysbus_create_simple("imx27_soc", MX27_CCM_BASE_ADDR, 0);
        sysbus_create_simple("imx_serial", MX27_UART1_BASE_ADDR,
                qdev_get_gpio_in(dev, 20));
        sysbus_create_simple("imx_serial", MX27_UART3_BASE_ADDR,
                qdev_get_gpio_in(dev, 18));

        apf27_binfo.ram_size = ram_size;
        apf27_binfo.loader_start = AFP27_DRAM_BANK1_ADDR;
        apf27_binfo.board_id = APF27_BOARD_ID;
        apf27_binfo.kernel_filename = kernel_filename;
        apf27_binfo.kernel_cmdline = kernel_cmdline;
        apf27_binfo.initrd_filename = initrd_filename;
        apf27_binfo.nb_cpus = 1;
        arm_load_kernel(env, &apf27_binfo);
}

static QEMUMachine apf27_machine = {
        .name = "apf27",
        .desc = "Armadeus APF27 Board (ARM926EJ-S)",
        .init = apf27_init, };

static void apf27_module_init(void) {
        qemu_register_machine(&apf27_machine);
}

machine_init(apf27_module_init)
#endif
```

On peut remarquer d'abord que la taille du code est modeste, mois de 100 lignes. La définition d'une nouvelle machine n'est pas monstrueuse. Elle est encadrée par une inclusion conditionnelle contrôlée par la constante **`CONFIG_APF27`** qui introduite via l'option **`-D`** de GCC.

Nous débuterons l'analyse par la fin. Tout commence par la macro **`machine_init()`**. C'est bien une macro et pas une fonction. Le point-virgule de fin de ligne est donc superflu, ce qui vaut des commentaires aussi désagréables que laconiques lors de la revue de code sur la liste de diffusion de développement de Qemu. Cette macro étendue donne :

```
static void __attribute__((constructor)) do_qemu_init_apf27_machine_init(void) {
        register_module_init(apf27_module_init, MODULE_INIT_MACHINE);
}
```

On comprend alors l'inutilité du point-virgule, car la macro produit la définition de la fonction **`do_qemu_init_apf27_machine_init()`** qui possède l'attribut GCC **`constructor`**. Lorsqu'une fonction est dotée de cet attribut, elle est automatiquement exécutée au lancement du programme avant la fonction **`main()`**. C'est ainsi que le module de machine est enregistré auprès de Qemu via l'appel de **`register_module_init()`**. Son premier argument est un pointeur sur la fonction d'initialisation du module, **`apf27_module_init()`** et le second spécifie le type de module à enregistrer. Cette opération insère simplement un nouvel élément **`ModuleEntry`** dans une liste double doublement chaînée (*tail queue* en anglais) tous les deux définis dans **`module.c`** :

```
typedef struct ModuleEntry {
    module_init_type type;
    void (*init)(void);
```

```
    QTAILQ_ENTRY(ModuleEntry) node;
} ModuleEntry;
```

Le membre **type** contient le type du module qui est la constante **MODULE_INIT_MACHINE** dans notre cas. Le pointeur de fonction **init** pointe sur la fonction d'initialisation **apf27_module_init()** et le membre **node** sert à gérer la liste. Qemu crée une liste par type de module. Il y aura donc une liste pour les modules de machine et une autre pour les modules de périphérique. L'analyse du fichier binaire de Qemu non strippé produit par la compilation avec la commande **objdump -t arm-softmmu/qemu-system-arm |grep do_qemu_init_** indique qu'il existe plus de 130 fonctions dont le nom commence par **do_qemu_init_**. Elles correspondent à toutes les fonctions d'enregistrement de module. Au lancement de Qemu, lorsque toutes les fonctions ayant l'attribut **constructor** se sont exécutées, la fonction **main()** de **vl.c** est exécutée. Au début de celle-ci **module_call_init(MODULE_INIT_MACHINE)** est appelée. Elle provoque l'exécution de toutes les fonctions d'initialisation des modules de machine, dont **apf27_module_init()**.

*Les structures de données de listes utilisées dans Qemu sont définies dans **qemu-queue.h**. Elles sont fortement inspirées de celles de **sys/queue.h** issues de BSD. Le terme « liste double », traduction de « tail queue », provient directement de la page de manuel française de queue et signifie simplement que les nouveaux éléments peuvent être insérés en tête ou en queue de la liste. « liste double doublement chaînée » n'est donc pas une erreur typographique. Cette page de manuel est contenue dans le paquet **manpages-fr-dev** sur une distribution Debian. Elle est accessible par **man queue**. L'amateur de structures de données sioux en appréciera la lecture et il se délectera de **sys/queue.h**. Bonne gymnastique !*

Cette manière de faire, faisant appel à un attribut spécifique de GCC, peut paraître déroutante au premier abord. En effet sans décortiquer le code, on comprend mal comment le nouveau module est intégré à Qemu. Mais lorsque la lumière est faite sur ce mécanisme, il présente un intérêt majeur. La compilation du fichier **apf27.c** produira un fichier objet **apf27.o**. Il suffira alors simplement que ce fichier objet soit lié au fichier binaire global de Qemu, **qemu-system-arm**, pour que le nouveau module soit intégré à Qemu. Aucun autre fichier ***.c** ou ***.h** n'est à modifier, seul le fichier **Makefile.target** nécessitera une petite adaptation comme nous le verrons dans la partie réservée à la mise en œuvre.

La fonction d'initialisation du module **apf27_module_init()** enregistre une nouvelle machine auprès de Qemu avec l'appel de **qemu_register_machine()**. La structure de type **QEMUMachine**, passée en argument, contient une description de la machine. Le membre **name** correspond à l'argument à passer à l'option **-M** de Qemu pour instancier cette machine, c'est le **-M apf27** évoqué en début de section. La chaîne de caractères pointée par **desc** contient la description de la machine qui est affichée par l'option **-M ?**. Le membre le plus important est certainement le pointeur de fonction **init**. Il pointe sur la fonction d'initialisation de la machine, **apf27_init()** qui ne sera appelée que si une machine APF27 est instanciée. Le pointeur **init** est de type **QEMUMachineInitFunc** défini dans **hw/boards.h** :

```
typedef void QEMUMachineInitFunc(ram_addr_t ram_size,
             const char *boot_device,
             const char *kernel_filename,
             const char *kernel_cmdline,
             const char *initrd_filename,
             const char *cpu_model);
```

Voici une description de son prototype et ses liens avec les options de la ligne de commande de lancement de Qemu :

– **ram_size** est la taille mémoire du système émulé exprimé en octets. La valeur par défaut vaut 128 Mo et elle peut être modifiée avec l'option **-m XXX**, où **XXX** est la taille mémoire souhaitée en Mo.

– **boot_device** contient la liste des périphériques de boot par ordre. Par défaut, elle vaut **cad**, ce qui signifie qu'il faut booter en premier sur le premier disque dur (**c**), puis sur le premier lecteur de disquettes (**a**), puis le premier lecteur de CD-ROM, (**d**). Aïe !!! Je me rends compte en écrivant cela, que les plus jeunes lecteurs, c'est-à-dire ceux qui n'ont, ni la chance d'avoir des cheveux blancs, ni celle d'avoir possédé un PC avec un Mo de RAM tournant sous MS-DOS 4.01, n'ont peut-être pas reconnu que le nommage provient de cet antique système. De plus, ils n'ont peut-être jamais touché une disquette. Certaines choses ont la vie dure, mais qu'ils soient rassurés, ils n'ont pas loupé grand-chose ! Cette valeur peut être modifiée via l'option **-boot order=XYZ**, où **XYZ** est la nouvelle liste de lettres de périphérique. Toute cette prose pour signaler finalement que cet argument n'est pas utilisé, car aucun périphérique de type bloc n'est émulé.

– **kernel_filename** est le chemin du fichier de l'image du noyau Linux à exécuter. Il est renseigné par l'option **-kernel**. Cette image peut être au format natif *zImage* ou dans celui de U-Boot *uImage*.

– **kernel_cmdline** contient les arguments à passer au noyau Linux. Ils seront visibles dans **/proc/cmdline** une fois le noyau opérationnel, et ils sont transmis à Qemu avec l'option **-append**.

– **initrd_filename** est le chemin de l'image de disque RAM initial utilisée éventuellement pour booter le système. L'option **-initrd** permet d'activer son utilisation et d'indiquer le fichier utilisé. Dans notre cas, n'ayant aucun périphérique de type bloc, cette option constitue le seul moyen d'obtenir un système de fichiers racine.

– **cpu_model** contient une chaîne de caractères non nulle uniquement si l'option **-cpu** est utilisée pour paramétrer le modèle ce processeur à utiliser. La liste des processeurs disponible peut être obtenue avec l'option **-cpu ?**. Dans notre cas, cet argument n'est pas utilisé, car le processeur émulé est toujours un ARM926EJ-S.

Ces différents paramètres vont permettre d'initialiser la carte et d'amorcer le système. La taille de la mémoire contenue dans **ram_size** est tout d'abord examinée. Une carte APF27 réelle peut être équipée de la manière suivante en mémoire RAM :

– Soit d'un chip mémoire de 64 Mo.

– Soit de deux chips mémoire de 64 Mo.

– Soit, de manière non standard, de deux chips mémoire de 128 Mo.

Sa taille mémoire est donc de 64 Mo, 128 Mo ou 256 Mo. La taille des chips mémoire émulés est définie par la constante **APF27_DRAM_BANK_SIZE** qui vaut **0x04000000**, c'est-à-dire 64 Mo. Avec cette valeur, les tailles mémoire acceptées seront de 64 Mo (un chip) ou de 128 Mo (deux chips). Si l'on souhaite utiliser des chips de 128 Mo, il faudra affecter la valeur **0x08000000** à cette constante. Les tailles mémoire acceptées seront alors de 128 Mo (un chip) ou de 256 Mo (deux chips).

Pour gérer la mémoire du système invité, Qemu fournit et utilise des structures **MemoryRegion**. Selon un commentaire du code ses membres ne doivent pas être utilisés directement. Ce type de structure permet de définir des zones mémoire RAM, ROM ou d'entrée/sortie. Elle contient toutes les informations sur la zone, telle que son nom, sa taille, son adresse, etc. Les zones peuvent être organisées hiérarchiquement, car chaque zone peut posséder un ensemble de sous-zones ainsi qu'une zone parent.

L'appel à **get_system_memory()** permet d'obtenir le pointeur **address_space_mem** sur une structure **MemoryRegion** correspondant à la totalité de l'espace mémoire du système émulé. Sur une plate-forme 32 bits comme l'architecture ARM, cette espace a une taille de 4 Go. C'est la zone mémoire racine qui est un espace vide dans lequel seront connectées les différentes zones mémoire telles que celles émulant les chips de mémoire RAM ou bien les zones de mémoire d'entrée/sortie des périphériques.

Ensuite, la mémoire pour une nouvelle structure **MemoryRegion** est allouée avec la macro **g_new** de la bibliothèque *Glib*. Cette macro est utilisée pour être cohérent avec le reste du code de Qemu qui utilise la *Glib* pour toutes ses allocations mémoire. La nouvelle structure est pointée par **ram_bank1** et elle constitue la zone mémoire émulant le premier chip de RAM. Cela peut sembler trivial, mais la RAM du système invité est émulée avec de la RAM du système hôte. L'allocation effective de la RAM émulée est réalisée par l'appel à **memory_region_init_ram()** dont voici l'explication de ses arguments :

– **ram_bank1**, vue ci-dessus, accueillera les informations sur la nouvelle zone mémoire allouée.

– Le deuxième pourrait pointer sur une structure **DeviceState** si la RAM appartenait à un périphérique. La zone étant attachée directement à une carte, il vaut **NULL** dans notre cas.

– Le troisième argument est une chaîne de caractères contenant le nom de la zone. Ce nom sera affiché lors du lancement de la commande **info mtree** du moniteur Qemu.

– **APF27_DRAM_BANK_SIZE** spécifie la taille de la zone comme nous l'avons précédemment vu.

Si une erreur survient lors de cet appel, Qemu se termine simplement. La zone mémoire est maintenant disponible et le lancement de la fonction **memory_region_add_subregion()** insère la zone **ram_bank1** dans la zone racine **address_space_mem** à l'adresse **AFP27_DRAM_BANK1_ADDR**. Cette dernière correspond à l'adresse physique du premier chip de RAM de l'APF27 issue de la documentation de la carte [4].

Si la taille mémoire nécessite la création d'une deuxième zone mémoire, les opérations décrites sont répétées avec **ram_bank2** à l'adresse **AFP27_DRAM_BANK2_ADDR**. Le lecteur avisé aura peut-être remarqué que les deux zones mémoire ne sont pas contiguës. Le noyau Linux gère cela très bien, mais Qemu l'accepte beaucoup moins bien... Ce sera le sujet de la section suivante.

Notre système invité est maintenant doté de mémoire et voici le résultat obtenu :

```
(qemu) info mtree
memory
00000000-fffffffe (prio 0): system
  10003000-10003fff (prio 0): imx27_timer
  1000a000-1000afff (prio 0): imx-serial
  1000c000-1000cfff (prio 0): imx-serial
  10027000-10027fff (prio 0): imx27_soc
  10040000-10040fff (prio 0): imx_int
  a0000000-a3ffffff (prio 0): apf27.ram_bank1
  b0000000-b3ffffff (prio 0): apf27.ram_bank2
```

La commande **info mtree** du moniteur Qemu affiche toutes les zones mémoires du système émulé sous la

forme d'un arbre. Au plus haut niveau, se trouve la zone **system**. Elle englobe toutes les autres et les deux dernières lignes correspondent aux deux zones de mémoire RAM. Les autres, telles-que **imx27_timer** ou **imx27_soc** représentent les zones de mémoire d'entrée/sortie des périphériques émulés que nous verrons ensuite.

L'instanciation du processeur virtuel est beaucoup plus simple puisqu'elle consiste simplement en l'appel de **cpu_init("arm926")** qui créer un modèle ARM926EJ-S. Passons à présent aux périphériques émulés. Le premier à être créé est celui dont dépendent les autres, c'est-à-dire le contrôleur d'interruption ou AITC (ARM Interrupt Controller). En effet, les UART et le timer GPT1 nécessitent des lignes d'interruption fournies par l'AITC pour pouvoir être instanciés. Bien que le contrôleur d'interruption de l'i.MX27 et l'implémentation de son émulation soient des sujets passionnants, je ne les détaillerai pas, alors faisons simple mais pas plus.

Un cœur de processeur ARM accepte deux types d'interruptions : les IRQ et les FIQ. Les IRQ (Interrupt Request) sont des interruptions classiques. Les FIQ (Fast Interrupt Request) sont des interruptions rapides. Elles sont prioritaires sur les IRQ, elles peuvent préempter les IRQ et elles ne peuvent pas être préemptées. Les FIQ sont en quelque sorte des « super interruptions » réservées aux traitements très urgents.

Le contrôleur d'interruption de l'i.MX27 comporte 64 entrées d'interruption en provenance des périphériques et deux sorties, IRQ et FIQ, vers le cœur ARM. Chaque entrée d'interruption peut être configurée soit en IRQ, soit en FIQ. Par ailleurs, un cœur ARM a un vecteur d'interruption pour les IRQ à l'adresse **0xFFFF0018**, et un autre pour les FIQ à l'adresse **0xFFFF001C**. Lorsqu'une interruption survient sur l'une de ses 64 entrée, l'AITC la répercute, suivant sa configuration, sur la sortie IRQ ou FIQ. Suivant le cas, le cœur ARM exécute l'instruction située à l'adresse du vecteur correspondant, **0xFFFF0018**, pour une IRQ et **0xFFFF001C** pour une FIQ. Ces adresses contiennent généralement une instruction de saut vers la fonction principale de traitement des interruptions. Celle-ci sauvegarde les registres, lit la source de l'interruption et lance la fonction de traitement spécifique de celle-ci.

*L'architecture ARMv5TEJ comporte sept d'exceptions possibles dont FIQ et IRQ (voir page A2-16 du manuel de référence de l'architecture ARM [9]). Chaque exception possède un vecteur de 32 bits qui permet de lancer la routine de traitement associée à l'exception. Ces vecteurs sont regroupé dans une table. Historiquement, elle était située à l'adresse virtuelle **0x00000000**. Les architectures ARM récentes permettent de placer la table de vecteurs soit à l'adresse **0x00000000**, soit à l'adresse **0xFFFF0000**. Ce choix est fait en manipulant le bit V (bit 13) du registre de contrôle du coprocesseur 15, CP15. La mémoire à l'adresse **0x00000000** étant généralement de la ROM, le noyau Linux utilise une table de vecteur à l'adresse 0xFFFF0000. Il est possible de visualisé simplement cette adresse en cherchant le message suivant dans le journal du noyau avec la commande **dmesg** :*

```
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
```

Le module Qemu d'émulation de l'AITC comporte, lui aussi, 64 entrées venant des périphériques et deux sorties IRQ et FIQ vers le processeur virtuel. Pour gérer les interruptions, Qemu utilise des structures **IRQState** et leur type de pointeur associé **qemu_irq**. Elles contiennent les membres suivant :

– **handler** est pointeur sur la fonction de traitement de l'interruption.

– **opaque** est un pointeur sur une structure de données passée en argument de la fonction **handler**.

– **n** est le numéro de l'interruption.

La fonction **arm_pic_init_cpu()** retourne un tableau contenant deux **qemu_irq**. Le premier correspond à la sortie IRQ et le deuxième à la sortie FIQ. Ces deux interruptions Qemu déclenchent la fonction d'émulation d'interruption du processeur virtuel lorsqu'elles sont activées. Ensuite, l'appel à **sysbus_create_varargs()** instancie le module du contrôleur d'interruption. Le premier est le nom du module Qemu à instancier. Ce nom, nous le verrons plus bas, est déclaré au sein du module de périphérique. Le deuxième est l'adresse physique virtuelle à laquelle il faut implanter le nouveau périphérique, **MX27_AITC_BASE_ADDR**. **sysbus_create_varargs()** est une fonction à nombre variable d'arguments, les deux premiers étant obligatoires. Les suivants désignent les interruptions utilisées par ce périphérique qu'il faut connecter au bus système. Dans notre cas, **cpu_pic[0]** et **cpu_pic[1]**, obtenues précédemment, représentent respectivement IRQ et FIQ. Le dernier argument **NULL** signifie que la liste des interruptions est terminée.La fonction retourne un pointeur sur la structure **DeviceState** du périphérique créé. L'interfaçage entre le processeur virtuel et le contrôleur d'interruption émulé est maintenant opérationnel, occupons-nous des autres périphériques.

Le module de timer est instancié avec l'appel à **sysbus_create_simple()**. Cette fonction remplace **sysbus_create_varargs()** lorsqu'une seule interruption est utilisée par le périphérique. Ses deux premiers arguments sont identiques à ceux de sa grande sœur. Le troisième est l'interruption qu'il faut connecter entre le périphérique et le bus système. Elle doit correspondre à l'une des 64 entrées du contrôleur d'interruption virtuel. La fonction **qdev_get_gpio_in()** permet d'obtenir un pointeur **qemu_irq** sur l'une d'elles. Son premier argument pointeur sur le périphérique du contrôleur d'interruption, le second est le numéro de l'interruption souhaitée, dans notre cas 26. Ce numéro provient de la documentation de l'i.MX27 p 10-27 qui nous apprend que l'interruption du GPT1 est connecté à la ligne d'interruption 26. Ainsi l'interfaçage entre le périphérique générateur d'interruption et le contrôleur d'interruption est réalisée.

Le module SoC et les deux modules d'UART sont créés de la même manière. Le nouveau système pourra être visualisé dans le moniteur de Qemu grâce à la commande **`info qtree`** :

```
(qemu) info qtree
bus: main-system-bus
  type System
  dev: imx_serial, id ""
    irq 1
    mmio 1000c000/00001000
  dev: imx_serial, id ""
    irq 1
    mmio 1000a000/00001000
  dev: imx27_soc, id ""
    irq 0
    mmio 10027000/00001000
  dev: imx27_timer, id ""
    irq 1
    mmio 10003000/00001000
  dev: imx_int, id ""
    gpio-in 64
    irq 2
    mmio 10040000/00001000
```

Elle affiche l'arbre des périphériques connectés au système émulé. Pour chacun d'entre eux sont affichés : le nom de son type, l'adresse de base de la mémoire d'entrée/sortie ainsi que sa taille, le nombre d'interruptions utilisées et le nombre lignes d'entrée.

À présent, la carte APF27 émulée est complètement construite. Il reste à la configurer pour que le système GNU/Linux puisse démarrer. Une structure de type **`arm_boot_info`** est utilisée à cette fin. Elle contient toutes les informations utiles au boot de Linux, c'est-à-dire :

– **`ram_size`** contient la taille totale de la mémoire RAM du système.

– **`loader_start`** est un pointeur sur la première instruction du boot loader à exécuter. Il doit pointer sur une zone de mémoire RAM à laquelle le processeur virtuel a accès.

– **`board_id`** est l'identifiant de la carte APF27 définie dans le noyau Linux. Il peut être trouvé dans le fichier **`arch/arm/tools/mach-type`** du noyau Linux et vaut 1698.

– **`nb_cpus`** est le nombre de processeurs virtuels du système, un seul pour l'APF27.

– **`kernel_filename`** est le chemin du fichier de l'image du noyau Linux à exécuter.

– **`kernel_cmdline`** contient les arguments à passer au noyau Linux.

– **`initrd_filename`** est le chemin de l'image de disque RAM initial utilisée.

Les trois derniers membres correspondent aux arguments passés à la fonction **`apf27_init()`**. Cette structure, ainsi que celle pointant sur le processeur virtuel, sont passées en arguments de la fonction **`arm_load_kernel()`**. Elle effectue les opérations suivantes :

– Le chargement en mémoire du noyau Linux.

– Le chargement en mémoire du disque RAM initial.

– L'initialisation d'un boot loader minimal pour un processeur ARM.

– L'initialisation du processeur virtuel pour que la prochaine instruction exécutée soit la première du boot loader minimal.

Le noyau, le disque RAM et le boot loader sont mappés en mémoire via les fonctions **`rom_add_file_fixed()`** et **`rom_add_blob_fixed()`**. Elles émulent des ROM en copiant dans l'espace mémoire principale du processeur virtuel le contenu d'un fichier ou d'une zone mémoire. Ces ROM peuvent être visualisées avec la commande **`info roms`** du moniteur Qemu :

```
(qemu) info roms
addr=a0000000 size=0x00001c mem=ram name="bootloader"
addr=a0010000 size=0x214280 mem=ram name="apf27-linux.bin"
addr=a0d00000 size=0xa05c00 mem=ram name="apf27-rootfs.cpio"
```

Pour chaque zone de ROM, l'adresse de base, la taille ainsi que le nom sont affichés.

La carte virtuelle est maintenant complètement configurée et lorsque Qemu aura fini l'ensemble de son initialisation, il activera le processeur virtuel et le système démarrera. La description du module de machine arrive à sa fin ou presque…

4.1 Processus de boot de Linux sur ARM et conséquences pour Qemu

Ou presque, en effet, car dans l'état actuel de Qemu, voilà ce que l'on obtient lorsque deux chips de mémoires sont utilisés :

```
$ ./qemu-system-arm -M apf27 -kernel apf27-linux.bin -append "console=ttymxc0
earlyprintk" -nographic -initrd apf27-rootfs.cpio
Uncompressing Linux... done, booting the kernel.
Linux version 2.6.38.1 (yvan@mach) (gcc version 4.4.5 (Buildroot 2010.11) ) #37 PREEMPT
Wed Mar 7 16:17:58 CET 2012
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIVT data cache, VIVT instruction cache
Machine: Armadeus APF27
bootconsole [earlycon0] enabled
Memory policy: ECC disabled, Data cache writeback
kernel BUG at mm/bootmem.c:342!
```

Décevant !!! Le noyau Linux s'arrête au début de son lancement. Il a même l'humilité de signaler qu'il comporte un bug à la ligne 342 du fichier **mm/bootmem.c**. Mais il n'en est rien, et Linux fonctionne très bien sur cette plate-forme ARM, je le sais. C'est l'émulation de la carte qui comporte un bug, mais où ?

Ce noyau a l'option **earlyprintk** activée, car sans elle, seule la première ligne apparaît. Cela nous donne une première piste : le problème se situe dans l'initialisation de la mémoire au moment du boot. Après un certain nombre d'heures passées à chercher, il s'avère que le problème ne provient pas du code de l'émulation de la carte APF27, mais il est situé dans le code générique Qemu de l'architecture ARM. Le boot loader minimal fourni par Qemu ne sait pas gérer plusieurs zones de mémoire non contiguës, or nous sommes précisément dans cette situation :

– Le premier chip de RAM est situé sur la plage **0xA0000000-0xA3FFFFFF**.

– Le second chip de RAM est situé sur la plage **0xB0000000-0xB3FFFFFF**.

Lorsque l'option **-m 64** est utilisée, un seul chip de mémoire est créé et le système démarre normalement. D'ailleurs le fichier **hw/arm_boot.c** contient un commentaire assez éloquent :

```
/* TODO: handle multiple chips on one ATAG list */
```

D'aucuns diront que ce n'est pas un bug, mais plutôt une fonctionnalité manquante. Et puis, la facilité serait de placer les 128 Mo de RAM entre **0xA0000000** et **0xA7FFFFFF**. Mais cela ne correspond pas à la réalité de la carte APF27 et par ailleurs corriger cette lacune va nous donner l'occasion de découvrir le processus de boot d'un noyau Linux sur plate-forme ARM.

On peut souvent médire du BIOS des PC et de son archaïsme. Cependant, il a l'avantage d'exister et de pouvoir transmettre au noyau les informations concernant la mémoire du système de manière assez standard. Sur les plates-formes ARM, la diversité du matériel est trop importante pour obtenir une telle standardisation. C'est pourquoi un boot loader doit être adapté à chaque carte. Il devra effectuer certaines opérations avant d'exécuter le noyau Linux. Le fichier **Documentation/arm/Booting** présent dans les sources de Linux fourni des renseignements sur ce que doit faire un boot loader pour lancer Linux dont voici les grandes lignes :

– 1. Initialiser et configurer la mémoire RAM.

– 2. Initialiser un port série.

– 3. Détecter le type de machine.

– 4. Configurer la liste étiquetée du noyau (tagged list).

– 5. Exécuter l'image du noyau.

Les étapes 1. à 3. sont réalisées lors de l'initialisation de la carte et des périphériques virtuels. Nous avons vu comment la mémoire RAM est mise en place et Qemu utilise les terminaux virtuels (**/dev/pts/X**) comme port série. Le type de machine est défini par la constante **APF27_BOARD_ID**.

La tagged list est une structure de données essentielle dans le processus de boot ARM. Elle est créée par le boot loader et utilisée par le noyau. C'est une liste de tags ATAG mis les uns au bout des autres. Ils sont définis dans le fichier **arch/arm/include/asm/setup.h** des sources du noyau Linux. En voici quelques exemples :

– Le début de liste (**ATAG_CORE**).

– La description des zones de RAM utilisables par Linux (**ATAG_MEM**).

– La taille et l'adresse du disque RAM initial (**ATAG_INITRD2**).

– Les arguments à passer au noyau Linux (**ATAG_CMDLINE**).

– La fin de la liste (**ATAG_NONE**).

La valeur entre parenthèses correspond à la constante d'identifiant du tag. Cette liste n'est pas exhaustive, mais suffisante pour notre cas. Une tagged list doit commencer par un tag **ATAG_CORE** et se finir par un tag **ATAG_NONE**. Entre eux, sont placé les tags qui décrivent au mieux la plate-forme. Voici comment sont définis ces tags dans **arch/arm/include/asm/setup.h** :

```
#define ATAG_NONE   0x00000000
struct tag_header {
       __u32 size;
       __u32 tag;
};
#define ATAG_CORE   0x54410001
struct tag_core {
       __u32 flags;
       __u32 pagesize;
       __u32 rootdev;
};
#define ATAG_MEM    0x54410002
struct tag_mem32 {
       __u32  size;
       __u32  start;
};
…
#define ATAG_INITRD2      0x54420005
struct tag_initrd {
       __u32 start;
       __u32 size;
};
…
#define ATAG_CMDLINE      0x54410009
struct tag_cmdline {
       char   cmdline[1];
};
...
struct tag {
       struct tag_header hdr;
       union {
              struct tag_core           core;
              struct tag_mem32    mem;
              struct tag_videotext      videotext;
              struct tag_ramdisk  ramdisk;
              struct tag_initrd   initrd;
              struct tag_serialnr serialnr;
              struct tag_revision revision;
              struct tag_videolfb videolfb;
              struct tag_cmdline  cmdline;
              struct tag_acorn    acorn;
              struct tag_memclk   memclk;
       } u;
};
```

Un tag est donc constitué d'une structure d'en-tête contenant sa taille et son identifiant de type, suivie d'une structure spécifique à chaque type, dont les noms des membres s'expliquent par eux-mêmes. Le boot loader minimal de Qemu crée la tagged list avec la fonction **set_kernel_args()** du fichier **hw/arm_boot.c**. En voici un extrait qui crée les trois premiers tags :

```
       target_phys_addr_t p;
       p = base + KERNEL_ARGS_ADDR;
       /* ATAG_CORE */
       WRITE_WORD(p, 5);
       WRITE_WORD(p, 0x54410001);
       WRITE_WORD(p, 1);
       WRITE_WORD(p, 0x1000);
       WRITE_WORD(p, 0);
       /* ATAG_MEM */
       /* TODO: handle multiple chips on one ATAG list */
       WRITE_WORD(p, 4);
       WRITE_WORD(p, 0x54410002);
       WRITE_WORD(p, info->ram_size);
       WRITE_WORD(p, info->loader_start);
```

```
        if (initrd_size) {
                /* ATAG_INITRD2 */
                WRITE_WORD(p, 4);
                WRITE_WORD(p, 0x54420005);
                WRITE_WORD(p, info->loader_start + INITRD_LOAD_ADDR);
                WRITE_WORD(p, initrd_size);
        }
```

Chaque tag de type **ATAG_MEM** décrit une zone de mémoire RAM utilisable par le noyau Linux. Il peut y en avoir plusieurs. Dans l'implémentation faite par Qemu, un seul tag **ATAG_MEM** est créé. La taille de la zone décrite s'élève à **info->ram_size** c'est-à-dire 128 Mo à l'adresse **info->loader_start** qui vaut **0xA0000000**. Cette description correspond à la plage **0xA0000000-0xA7FFFFFF,** ce qui ne reflète pas du tout la réalité. En effet, les plages de mémoire RAM honorées par le module APF27 sont **0xA0000000-0xA3FFFFFF** et **0xB0000000-0xB3FFFFFF**. La zone comprise entre **0xA4000000** et **0xA7FFFFFF** est un « trou » et ne contient pas de mémoire opérationnelle. Lorsque Linux démarre, une des premières chose qu'il fait est de vérifier par écriture/lecture les plages mémoires que le boot loader lui a transmises via les tags **ATAG_MEM**. Le test de la zone **0xA4000000-0xA7FFFFFF** échoue et se termine en un triste « *kernel BUG at mm/bootmem.c:342!* ». La solution consiste donc à créer un deuxième tag **ATAG_MEM** pour le deuxième chip mémoire émulé. Voici les modifications à apporter à la fonction **set_kernel_args()** :

```
#ifndef CONFIG_APF27
        WRITE_WORD(p, 4);
        WRITE_WORD(p, 0x54410002);
        WRITE_WORD(p, info->ram_size);
        WRITE_WORD(p, info->loader_start);
#else
        if (info->board_id != APF27_BOARD_ID) {
                WRITE_WORD(p, 4);
                WRITE_WORD(p, 0x54410002);
                WRITE_WORD(p, info->ram_size);
                WRITE_WORD(p, info->loader_start);
        } else {
                WRITE_WORD(p, 4);
                WRITE_WORD(p, 0x54410002);
                WRITE_WORD(p, APF27_DRAM_BANK_SIZE);
                WRITE_WORD(p, AFP27_DRAM_BANK1_ADDR);
                if (info->ram_size == APF27_DRAM_BANK_SIZE * 2){
                        WRITE_WORD(p, 4);
                        WRITE_WORD(p, 0x54410002);
                        WRITE_WORD(p, APF27_DRAM_BANK_SIZE);
                        WRITE_WORD(p, AFP27_DRAM_BANK2_ADDR);
                }
        }
#endif
```

Comme pour le reste du code produit, celui-ci n'est inclus que si la constante **CONFIG_APF27** est définie. Si l'identifiant de la carte ne correspond pas à celui d'une APF27, rien n'est modifié. Sinon un premier tag **ATAG_MEM** est créé. Sa taille est celle un chip mémoire spécifiée dans la constante **APF27_DRAM_BANK_SIZE** et son adresse de base est **0xA0000000**. Si la taille mémoire globale du système nécessite l'émulation d'un deuxième chip, un second tag **ATAG_MEM** est créé décrivant une zone de taille **APF27_DRAM_BANK_SIZE** à l'adresse **0xB0000000**. Ces deux tags permettent de décrire fidèlement le système physique réel et ils fonctionnent avec le module Qemu de l'APF27.

5. Le module SoC et le Module timer système i.MX27

Le sujet principal de cette section est l'étude et l'implémentation de l'émulation de l'un des éléments essentiels d'un système à microprocesseur : le timer système. Dans les grandes lignes, un timer est composant qui permet le comptage des impulsions provenant d'une horloge et qui peut déclencher une interruption matérielle lorsqu'une certaine valeur est atteinte. Il permet donc de mesurer le temps qui s'écoule de manière précise. Les timers de l'i.MX27 sont appelés « General Purpose Timer » ou GPT. Le terme universel signifie qu'ils peuvent être utilisés de nombreuses manières, autres que celles déjà citées, comme dater un événement survenant sur l'un de ses entrées externes. Notre SoC comporte 6 GPT, mais seul le premier, GPT1, est utilisé par le noyau Linux. Il lui sert de source de temps unique.

5.1 Les horloges de l'i.MX27 et celles de Linux

Avant de plonger dans l'implémentation du module d'émulation du timer GPT1 de l'i.MX27, nous allons faire

un petit détour pour rendre visite aux horloges physiques de l'i.MX27 et leurs pendants logiciels dans le noyau Linux. De par leur nombre et la souplesse de leurs configurations, la mise en œuvre de ces horloges peut sembler compliqué au premier abord. En effet, il serait possible de les ignorer et d'obtenir un module qui fonctionne vaguement, pas forcément à la bonne fréquence. Le sentiment serait alors le même que pour le boot ARM de la section précédente, un module qui « tombe en marche ». Mais la source de temps du GPT1 dépend directement de ces horloges et il est, à mon sens, utile de savoir ce qu'il se passe au plus bas niveau. Notre progression va se faire du matériel vers le logiciel.

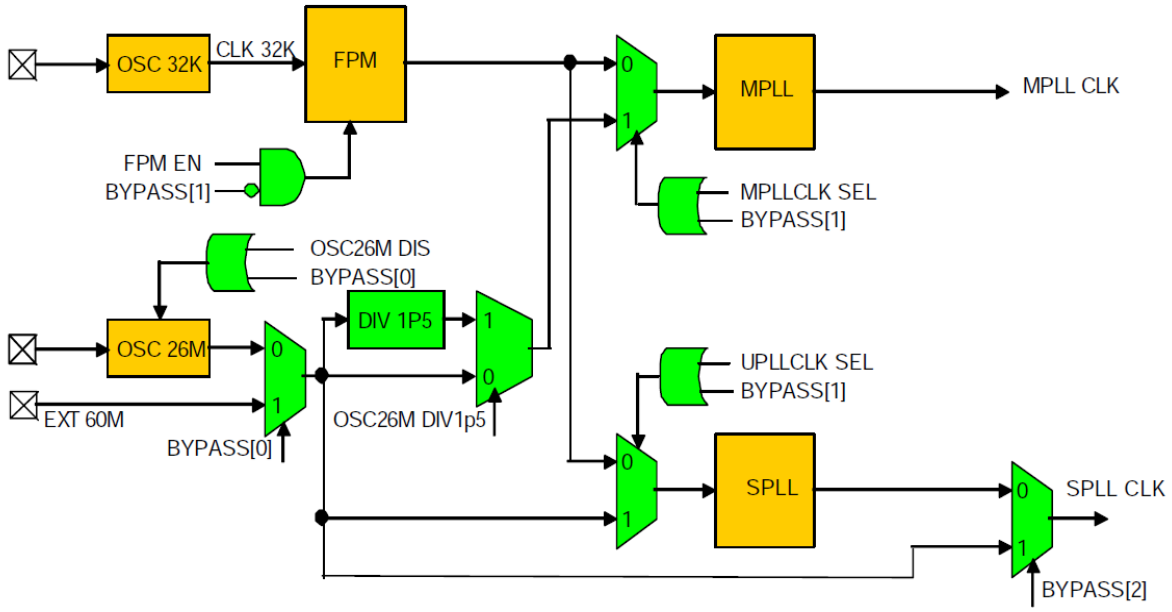

Fig. 1 : Schéma fonctionnel de la distribution des horloges de l'i.MX27 partie 1/2. MCIMX27 Multimedia Applications Processor Reference Manual page 3-2 © Freescale Semiconductor.

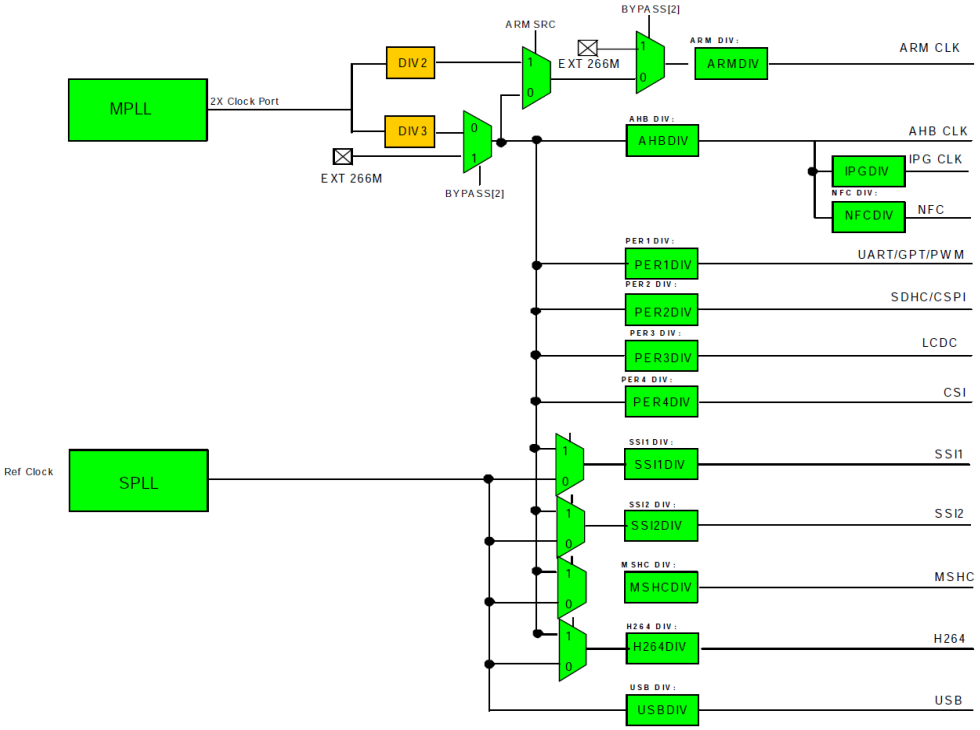

Fig. 2 : Schéma fonctionnel de la distribution des horloges de l'i.MX27 partie 2/2. MCIMX27 Multimedia Applications Processor Reference Manual page 3-3 © Freescale Semiconductor.

Les figures 1 et 2 présentent, d'un point de vue fonctionnel, la distribution des horloges de l'i.MX27. Elles sont documentées au chapitre 3 de [3]. À première vue, le nombre de signaux d'horloges, presque 20, peut étourdir. S'il y en a autant, c'est parce que l'i.MX27 n'est pas un simple processeur, mais un SoC. Mis à part

les mémoires RAM et Flash, il est presque autonome, car il embarque, en plus du processeur ARM926EJ-S, de nombreux périphériques qui nécessitent chacun un signal d'horloge indépendant. En effet, un contrôleur de NAND Flash n'a pas les mêmes besoins qu'un port série.

Tout commence par une source primaire d'horloge. L'i.MX27 accepte un signal à 26 MHz ou à 32,768 kHz. Dans le cas de l'APF27, c'est le quartz à 32,768 kHz, noté Y1 sur son schéma électrique [10] qui le fournit. La fréquence de ce signal est multipliée par 1024 par le pré-multiplicateur de fréquence, noté FPM sur la figure 1, pour obtenir un signal à 33,554 MHz. Ce dernier est utilisé comme source pour les deux principaux sous-systèmes d'horloge que sont :

– Le MPLL, pour MCU/System PLL (*Phase-Locked Loop* ou boucle à phase asservie, en bon français), qui alimente, l'horloge du processeur ARM9, et celles, après une division par 2/3 (DIV3 sur la figure 2), du bus système, du contrôleur de NAND Flash, des UART, des timers (dont notre précieux GPT1), des sorties PWM, du contrôleur graphique, des bus SPI, du contrôleur de carte SD et de l'interface du capteur CMOS,

– Le SPLL, pour Serial Peripheral PLL, qui alimente les horloges des contrôleurs USB, des interfaces SSI, du codec vidéo H.264 et du contrôleur de Memory Stick (ces trois derniers pouvant aussi être alimentés par MPLL).

Chacune des horloges finales dispose son propre diviseur de fréquence dont le facteur est ajustable. La configuration globale des horloges est réalisée via différents registres de l'i.MX27. Voici la description de ceux qui ont un impact sur la configuration des horloges de Linux :

Nom	Description	Valeur
CSCR	Clock Source Control Register	0X4300810D
MPCTL0	MPLL Control Register 0	0X01EF15D5
MPCTL1	MPLL Control Register 1	0X00008000
SPCTL0	SPLL Control Register 0	0X0475206F
SPCTL1	SPLL Control Register 1	0X00000000
PCDR0	Peripheral Clock Divider Register 0	0X12C41083
PCDR1	Peripheral Clock Divider Register 1	0X0707070F

Les valeurs déterminent le fonctionnement de toutes les horloges. Je ne détaillerai pas leurs significations, car cela serait très ennuyeux et dépasserait le cadre de l'article. L'exercice pourra être réalisé en consultant le chapitre 3 de [3]. Par contre, je les cite parce-qu'elles seront utilisées dans le module Qemu SoC décrit plus bas et cette démarche entre dans celle, plus générale, de conception d'un module d'émulation matérielle Qemu. Ensuite, ces valeurs ne sortent pas d'un chapeau magique, mais elles proviennent de U-Boot adapté à l'APF27. Avant d'exécuter le noyau Linux, U-Boot initialise le matériel et particulièrement les horloges en vue de leur utilisation ultérieure. C'est en lisant les fichiers **`u-boot-1.3.4/include/configs/apf27.h`** et **`u-boot-1.3.4/board/armadeus/apf27/lowlevel_init.S`** fournis par le BSP Armadeus (merci Éric), que l'on trouve ces précieuses valeurs. Elles produisent les fréquences suivantes :

– L'horloge de référence, le quartz Y1, tourne à 32,768 kHz.

– La sortie du pré-multiplicateur FPM tourne 33,554 MHz.

– MPLL tourne à 399 MHz.

– SPLL tourne à 299,99937 MHz.

– L'horloge à la sortie de DIV2 (figure 2) tourne à 399 MHz.

– L'horloge du processeur ARM (ARM CLK, figure 2) tourne à 399 MHz.

– L'horloge à la sortie de DIV3 (figure 2) tourne à 266 MHz.

– L'horloge du bus système (AHB CLK, figure 2) tourne à 133 MHz.

– L'horloge PER1 dédiée aux UART, aux sorties PWM et aux GPT, dont notre GPT1, tourne à 16,625 MHz.

À présent que les horloges sont bien identifiées, elles peuvent être représentées sous la forme d'un graphe de dépendances présenté sur la figure 3.

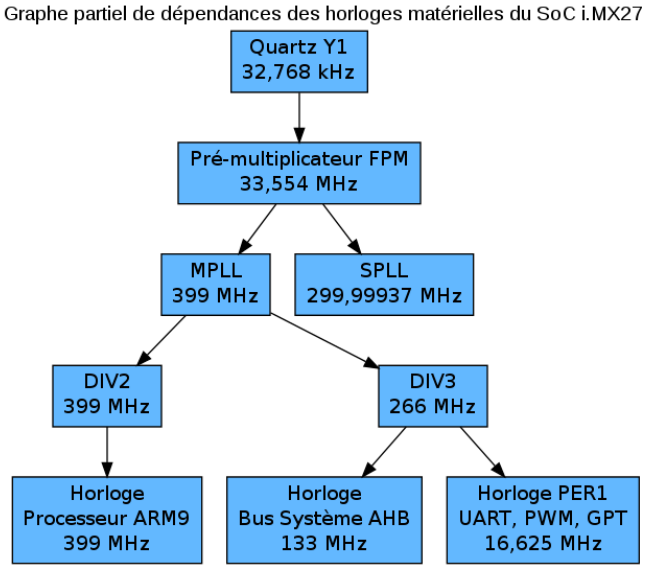

Fig. 3 : Graphe partiel de dépendances des horloges matérielles du SoC i.MX27.

L'aspect matériel des horloges étant éclairci, passons à présent à leurs représentations logicielles. Linux utilise la structure **`clocksource`** (**`include/linux/clocksource.h`**) pour représenter les horloges au plus haut niveau, quelle que soit l'architecture de processeur sous-jacente. Ses champs les plus intéressants sont :

– **`name`**, un pointeur sur une chaîne de caractères contenant son nom en terme humain.

– **`rating`**, un entier représentant la qualité de l'horloge allant de 1, très mauvaise, à 499, excellente.

– **`read`**, un pointeur de fonction retournant la valeur courante du compteur de l'horloge.

– **`mult`** et **`shift`**, deux entiers non signés de 32 bits permettant de convertir la valeur du compteur de l'horloge en nanosecondes.

Il peut y en avoir plusieurs sur un même système, consultables dans le fichier **`/sys/devices/system/clocksource/clocksource0/available_clocksource`**. Cependant, de par la forte dépendance des horloges au matériel, il est de la responsabilité du code spécifique à chaque architecture de créer ces structures **`clocksource`**. Concernant l'architecture ARM i.MX, son code spécifique utilise une structure **`clk`** (**`arch/arm/plat-mxc/include/mach/clock.h`**) pour représenter une horloge matérielle. Ces champs notables sont :

– **`parent`**, un pointeur sur une autre structure **`clk`** indiquant que cette autre horloge est la source de celle-ci.

– **`get_rate`**, un pointeur de fonction retournant la fréquence en hertz de cette horloge. L'appel à la fonction **`clk_get_rate()`** utilisera ce pointeur.

La présence de membre **`parent`** permet d'établir un arbre d'horloges qui est une représentation fidèle de la réalité physique. C'est ce qui est réalisé dans le fichier **`arch/arm/mach-imx/clock-imx27.c`** dont la figure 4 donne une reproduction partielle.

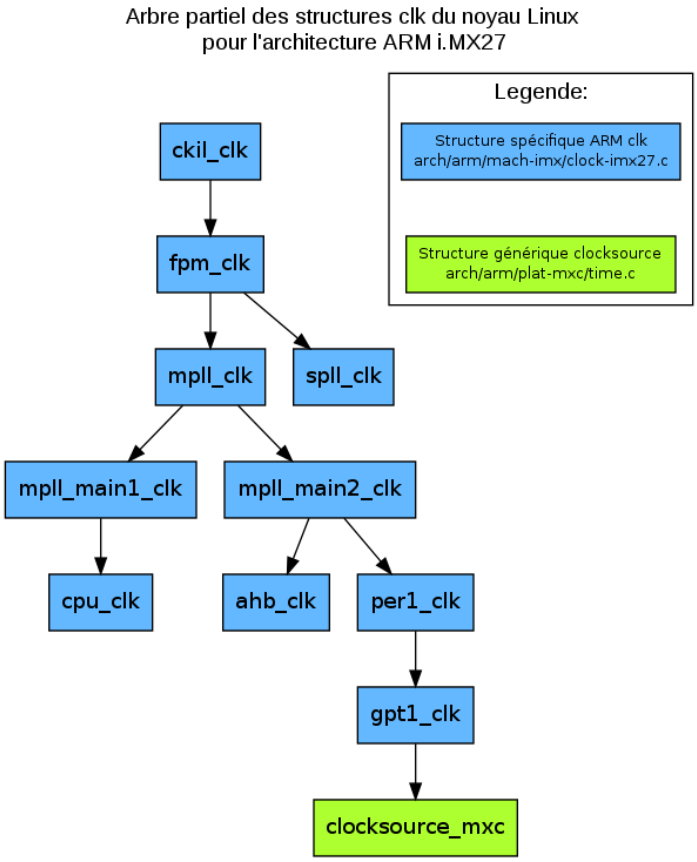

Fig. 3 : Arbre partiel des structures **`clk`** *de l'architecture ARM i.MX27. L'horloge finale, spécifique ARM,* **`gpt1_clk`** *est utilisée comme source pour la source d'horloge globale* **`clocksource_mxc`**.

Le terme partiel est utilisé, car il y en tout plus de 90 structures **`clk`** déclarées dans ce fichier, chaque périphérique (UART1, UART2, …) ayant sa propre horloge. Les développeurs du noyau ont donc créé une représentation fidèle du matériel et les figures 3 et 4 présentent certaines correspondances :

– **`ckil_clk`** : horloge cadencée par le quartz Y1.

– **`fpm_clk`** : Sortie du pré-multiplicateur FPM.

– **`mpll_clk`** : Sortie de MPLL.

– **`spll_clk`** : Sortie de SPLL.

– **`mpll_main1_clk`** : Sortie de DIV2.

– **`mpll_main2_clk`** : Sortie de DIV3.

– **`cpu_clk`** : Horloge du processeur ARM9.

– **`ahb_clk`** : Horloge du bus AHB.

– **`per1_clk`** : Horloge commune aux périphériques UART, GPT et aux sorties PWM.

– **`gpt1_clk`** : Horloge spécifique au GPT1. C'est la même que **`per1_clk`**.

– **`clocksource_mxc`** : Horloge globale utilisée par Linux pour gérer l'intégralité du temps au sein du noyau. Elle n'a pas de pendant dans les horloges physiques de l'i.MX27, mais correspond au registre TCN1 de GPT1.

L'initialisation des horloges commence par l'appel des fonctions **`apf27_timer_init()`** et **`mx27_clocks_init()`**, dans **`arch/arm/mach-imx/apf27.c`**, lors de la mise en place du timer système de la carte APF27. Cette dernière configure les horloges de bas niveau **`clk`**, puis **`mxc_timer_init()`** (**`arch/arm/plat-mxc/time.c`**) est exécutée. Elle prend pour arguments, l'horloge finale **`gpt1_clk`**, dédiée au timer matériel GPT1, ainsi que l'adresse I/O de base de GPT1 et son numéro d'interruption. Elle l'initialise, puis **`mxc_clocksource_init()`** enregistre la source générale d'horloge **`clocksource_mxc`** via **`clocksource_register_hz()`** en utilisant la fréquence de l'horloge de GPT1. Cette structure générique **`clocksource clocksource_mxc`** est déclarée dans le fichier **`arch/arm/plat-mxc/time.c`** :

```
static struct clocksource clocksource_mxc = {
       .name         = "mxc_timer1",
       .rating             = 200,
       .read         = mx1_2_get_cycles,
       .mask         = CLOCKSOURCE_MASK(32),
       .flags        = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

rating correspond à une source de qualité moyenne, **mask** indique que le compteur de l'horloge n'est que sur 32 bits au lieu de 64 bits, taille des représentations de temps au sein du noyau et **CLOCK_SOURCE_IS_CONTINUOUS** signifie qu'aucun pas de comptage n'est omis lors de la progression de l'horloge. Le membre le plus intéressant est certainement **read** qui pointe sur la fonction **mx1_2_get_cycles()**. Cette dernière retourne la valeur courante de l'horloge globale en lisant la valeur courante du compteur GPT1 contenu dans son registre TCN1. Il est incrémenté à la fréquence de l'horloge **gpt1_clk**, c'est-à-dire 16,625 MHz. C'est un registre capital, car c'est lui qui rythme l'intégralité de l'écoulement du temps dans le noyau Linux et donc aussi pour les processus en mode utilisateur. Nous verrons, dans le détail de l'implémentation du module Qemu du timer de l'i.MX27, que c'est principalement la valeur de ce registre que nous devrons émuler.

Quelques **printk()** judicieusement placés dans la fonction **mx27_clocks_init()** permettent de vérifier les propos énoncés. Ayant déjà fait, le mois dernier, l'apologie du **printf()**, je n'aurai pas l'insolence de recommencer avec **printk()**.Pour chaque horloge concernée, ajoutons, par exemple pour la première de toutes, **ckil_clk** :

```
unsigned int c;
c= clk_get_rate(&ckil_clk);
printk ("%s: ckil_clk rate : %d\n",__FUNCTION__,c);
```

La console de Linux affiche alors :

```
mx27_clocks_init: ckil_clk rate :       32768
mx27_clocks_init: fpm_clk rate : 33554432
mx27_clocks_init: mpll_clk rate :       399000080
mx27_clocks_init: spll_clk rate :       299999370
mx27_clocks_init: mpll_main1_clk rate : 399000080
mx27_clocks_init: cpu_clk rate : 399000080
mx27_clocks_init: mpll_main2_clk rate : 266000053
mx27_clocks_init: ahb_clk rate : 133000026
mx27_clocks_init: per1_clk rate :       16625003
mx27_clocks_init: gpt1_clk rate :       16625003
```

Les fréquences affichées sont naturellement exprimées en hertz, et correspondent bien aux valeurs attendues, en respect des valeurs configurées par U-Boot ou simulées par le module SoC i.MX27 Qemu.

Ce passage, en détaillant les horloges internes du noyau Linux, peut sembler s'éloigner du sujet principal, qui est l'émulation matérielle d'un timer de l'i.MX27. Cependant, il n'en est rien. En effet, la connaissance de la fréquence de l'horloge **gpt1_clk** est essentielle pour émuler fidèlement le matériel. Mais celle-ci ne peut être connue qu'en analysant finement la documentation constructeur du SoC, la configuration faite par U-Boot et le fonctionnement interne de Linux. Nous pouvons alors remarquer que pour implémenter sérieusement l'émulation d'un périphérique aussi simple qu'un timer, l'effort de recherche documentaire n'est pas négligeable. C'est le prix à payer pour éviter d'obtenir quelque chose d'approximatif qui « tombe en marche » après de multiples « essais erreurs », méthode chère à nos amis Shadoks [11].

5.2 Le module SoC

Ayant acquis le background nécessaire, passons à présent à l'implémentation du module SoC. Il est abordé en premier, car il est plus simple que le module timer. Il permettra de se familiariser avec les déclarations communes à tous les modules de périphériques. Son code est contenu dans le fichier **hw/imx27_soc.c** qui est apporté par les patchs que nous appliquerons lors de la mise en œuvre. En voici le contenu intégral :

```
#ifdef CONFIG_APF27
#include "sysbus.h"

#define IMX27_SOC_DEVICE_NAME "imx27_soc"
#define IMX27_SOC_DEVICE_DESC "i.MX27 SoC"
#define IMX27_SOC_IO_MEM_SIZE 0x00001000
#define CCM_CSCR    0x0000
#define CCM_MPCTL0  0x0004
#define CCM_MPCTL1  0x0008
#define CCM_SPCTL0  0x000C
#define CCM_SPCTL1  0x0010
```

```
#define CCM_PCDR0   0x0018
#define CCM_PCDR1   0x001C
#define CID               0x0800

typedef struct {
       SysBusDevice busdev;
       MemoryRegion iomem;
       uint32_t ccm_cscr;
       uint32_t ccm_mpctl0;
       uint32_t ccm_mpctl1;
       uint32_t ccm_spctl0;
       uint32_t ccm_spctl1;
       uint32_t ccm_pcdr0;
       uint32_t ccm_pcdr1;
       uint32_t cid;
} imx27_soc_state;

static void imx27_reset_hard(DeviceState *dev) {
       imx27_soc_state *state =
                     FROM_SYSBUS(imx27_soc_state, sysbus_from_qdev(dev));
       state->cid = 0X2882101D;
       state->ccm_cscr = 0X4300810D;
       state->ccm_mpctl0 = 0X01EF15D5;
       state->ccm_mpctl1 = 0X00008000;
       state->ccm_spctl0 = 0X0475206F;
       state->ccm_spctl1 = 0X00000000;
       state->ccm_pcdr0 = 0X12C41083;
       state->ccm_pcdr1 = 0X0707070F;
}
static uint64_t imx27_soc_read(void *opaque, target_phys_addr_t offset,
              unsigned size) {
       imx27_soc_state *state = (imx27_soc_state *) opaque;
       switch (offset) {
              case CCM_CSCR:
                     return state->ccm_cscr;
              case CCM_MPCTL0:
                     return state->ccm_mpctl0;
              case CCM_MPCTL1:
                     return state->ccm_mpctl1;
              case CCM_SPCTL0:
                     return state->ccm_spctl0;
              case CCM_SPCTL1:
                     return state->ccm_spctl1;
              case CCM_PCDR0:
                     return state->ccm_pcdr0;
              case CCM_PCDR1:
                     return state->ccm_pcdr1;
              case CID:
                     return state->cid;
       }
       return 0;
}

static void imx27_soc_write(void *opaque, target_phys_addr_t offset,
              uint64_t value, unsigned size) {

}

static const MemoryRegionOps imx27_soc_ops = {
              .read = imx27_soc_read,
              .write = imx27_soc_write,
              .endianness = DEVICE_NATIVE_ENDIAN,
};

static int imx27_soc_init(SysBusDevice *dev) {
       imx27_soc_state *state = FROM_SYSBUS(imx27_soc_state, dev);
       memory_region_init_io(&state->iomem, &imx27_soc_ops, state,
                     IMX27_SOC_DEVICE_NAME, IMX27_SOC_IO_MEM_SIZE);
       sysbus_init_mmio_region(dev, &state->iomem);
       return 0;
}
```

```
static void imx27_soc_register_device(void) {
       SysBusDeviceInfo *info = g_malloc0(sizeof(*info));
       info->qdev.name = IMX27_SOC_DEVICE_NAME;
       info->qdev.desc = IMX27_SOC_DEVICE_DESC;
       info->qdev.size = sizeof(imx27_soc_state);
       info->qdev.reset = imx27_reset_hard;
       info->init = imx27_soc_init;
       sysbus_register_withprop(info);
}

device_init(imx27_soc_register_device)
#endif
```

Commençons par la dernière ligne. La macro **device_init()** est similaire à **machine_init()** déjà présentée plus haut, mais au lieu d'effectuer l'enregistrement d'un module de type machine, c'est un module de type périphérique qui l'est. Elle est étendue en :

```
static void __attribute__((constructor)) do_qemu_init_imx27_soc_register_device(void) {
       register_module_init(imx27_soc_register_device, MODULE_INIT_DEVICE);
}
```

Comme dans le cas de **machine_init()**, une fonction **do_qemu_init_imx27_soc_register_device()** ayant l'attribut **constructor** est déclarée. Elle sera donc exécutée avant **main()**, et l'appel à **register_module_init()** provoquera

– L'allocation d'une nouvelle structure **ModuleEntry**.

– L'affectation au membre **init** de cette structure de la valeur du pointeur de la fonction d'enregistrement du module **imx27_soc_register_device()**.

– L'insertion de la structure dans la liste doublement chaînée **init_type_list[MODULE_INIT_DEVICE]** réservée aux modules de périphérique.

La suite de l'initialisation des modules intervient durant la phase de lancement de Qemu, dans la fonction **main()**, lorsque **module_call_init(MODULE_INIT_DEVICE)** est appelée. Cet appel survient bien après celui qui concerne les machines (avec **MODULE_INIT_MACHINE** comme argument) et déclenche l'appel successif des fonctions pointées par le membre **init** de toutes les structures **ModuleEntry** de la liste chaînée **init_type_list[MODULE_INIT_DEVICE]**. C'est ainsi que la fonction **imx27_soc_register_device()** sera exécutée.

Une nouvelle structure **SysBusDeviceInfo** est allouée via la fonction Qemu **g_malloc0()** basée sur **mmap()**. Cette structure renferme les informations concernant un périphérique directement connecté au bus système. Contrairement aux périphériques USB ou PCI qui sont connectés à ces bus respectifs, ce sont des périphériques directement connectés au bus d'adresses, de données et de contrôle du processeur. Les anciens périphériques du bus ISA étaient de ce type. En terminologie Linux, de tels périphériques sont appelés des « platform devices ».Le fichier d'en-tête **hw/sysbus.h** contient les déclarations concernant de tels périphériques.

La structure **SysBusDeviceInfo** contient un membre **qdev** de type **DeviceInfo**. Son membre **qdev.name** contient le nom du périphérique qui est utilisé pour en créer une instance dans la définition d'une machine. Le membre **qdev.desc** contient une description textuelle concernant le périphérique. Ces deux informations pourront être visualisées avec la commande **info qdm** du moniteur de Qemu. Vient ensuite le membre **qdev.size** qui est renseigné avec la taille de la structure **imx27_soc_state** qui contiendra les données d'une instance du périphérique. Nous reviendrons un peu plus loin sur cette structure particulière. Le membre **qdev.reset** est un pointeur de fonction sur la fonction de remise à zéro matérielle du périphérique émulé. Il pointe dans notre cas sur **imx27_reset_hard()**. Le membre **init** de **SysBusDeviceInfo** est un pointeur sur la fonction d'initialisation de l'instance du périphérique **imx27_soc_init()**. Cette fonction sera appelée lorsque le périphérique sera instancié dans la définition d'une machine via **sysbus_create_simple()** comme nous l'avons vu précédemment. Une autre manière d'instancier un périphérique est de renseigner l'option **-device imx27_soc** dans la ligne de commande de Qemu. Le nom du périphérique passé en option correspond au champ **qdev.name**.

Pour finir, la fonction **sysbus_register_withprop()** est appelée avec comme argument la structure que nous venons de renseigner. Cette fonction enregistre le périphérique auprès de Qemu. À partir de cet appel, le périphérique est utilisable dans Qemu. La commande du moniteur de Qemu **info qdm** permettra de vérifier sa présence. Peter Maydell, mainteneur de l'architecture ARM de Qemu, conseille d'utiliser **sysbus_register_withprop()** plutôt que **sysbus_register_dev()** pour enregistrer un périphérique auprès de Qemu. Elle nécessite l'allocation manuelle d'une structure **SysBusDeviceInfo**, mais permet d'automatiser l'appel de la fonction de remise à zéro du périphérique via le membre **qdev.reset**.

Lorsque le périphérique sera instancié, la fonction **imx27_soc_init()** sera appelée. Elle prend pour

argument un pointeur sur une structure **SysBusDevice**. Elle contient toutes les informations relatives à cette instance de périphérique connecté au bus système, telle que les interruptions et les plages de mémoire d'entrée sortie. Dans première ligne de la fonction **imx27_soc_init()**, la macro **FROM_SYSBUS** permet de retrouver l'adresse de la structure **imx27_soc_state** propre à cette instance de périphérique à partir du pointeur **dev** de type **SysBusDevice***. Cette structure, défini au début du fichier **hw/imx27_soc.c**, contient les données privées nécessaires au fonctionnement de l'émulation de l'instance du périphérique. Dans notre cas, elle comprendra :

– **SysBusDevice busdev** : Membre **SysBusDevice** dont la présence est obligatoire et en première position.

– **MemoryRegion iomem** : Structure décrivant la zone de mémoire d'entrée/sortie correspondant aux registres émulés. Ce type de structure a déjà été présentée dans la section 4.

– Les membres **ccm_cscr**, **ccm_mpctl0**, **ccm_mpctl1**, **ccm_spctl0**,**ccm_spctl1**, **ccm_pcdr0** et **ccm_pcdr1** sont des entiers non signés de 32 bits. Ils servent à stocker les valeurs des registres émulés du CCM (Clock Control Module) présentés dans le tableau du 5.1. Leur nommage est en correspondance directe.

– **cid** est un entier non signé de 32 bits qui stocke la valeur du registre CID ou Chip ID. Ce registre contient l'identifiant de l'i.MX27, c'est-à-dire la version du SoC, son numéro de référence et l'identifiant du fabricant. Linux le lit via la fonction **mx27_revision()** de **arch/arm/mach-imx/cpu-imx27.c**. La configuration des horloges dépend de la valeur du registre CID. Celle utilisée ici a été extraite d'un i.MX27 réel.

Plusieurs points importants sont cependant à noter au sujet de la structure **imx27_soc_state** et plus généralement sur les structures de données privées d'un module de périphérique Qemu. Leur premier membre doit absolument s'appeler **busdev** et être de type **SysBusDevice**. Ce n'est indiqué nulle part, ni dans la maigre documentation, ni même en commentaire dans le code source. Mais si cette règle n'est pas respectée, des erreurs de compilation ou des problèmes de fonctionnement surviendront. En effet, le pointeur **dev** passé en argument de la fonction **imx27_timer_init()** pointe sur ce membre **busdev** et la macro **FROM_SYSBUS** permet d'obtenir un pointeur sur la structure **imx27_soc_state**. Si le membre **SysBusDevice busdev** est présent en première position, ces deux pointeurs sont alors identiques. Cette construction peut paraître alambiquée, mais elle permet de passer en argument un pointeur sur une structure spécifique, dans notre cas **imx27_soc_state**, dont le reste du code de Qemu n'a pas connaissance, en utilisant un type défini globalement, **SysBusDevice**. Cette contrainte est donc imposée par la macro **FROM_SYSBUS**, dont j'épargnerai l'expansion au lecteur, qui recherche un membre **busdev** au début de la structure **imx27_soc_state** et qui retourne l'adresse de cette dernière. L'absence de l'héritage en C oblige quelques contorsions. Les structures **SysBusDevice** et **imx27_soc_state** sont allouées et renseignées par la fonction **qdev_create_from_info()** de **hw/qdev.c**.

Lorsque le pointeur sur la structure de données privée est retrouvé, la zone de mémoire d'entrées/sorties correspondant aux adresses des registres émulés est allouée avec l'appel de la fonction **memory_region_init_io()**. Son premier argument est le membre **iomem** de notre structure **imx27_soc_state**, déjà présenté. Le second est la structure **imx27_soc_ops** de type **MemoryRegionOps** qui contient les membres suivants :

– **read** est un pointeur de fonction sur une fonction de callback qui sera appelée lorsqu'une opération de lecture sera effectuée sur la zone de mémoire allouée.

– **write** est le pendant de **read** pour les opérations d'écriture.

– **endianness** est le boutisme à utiliser lors des opérations de lecture et d'écriture sur des mots de plus d'un octet de long. **DEVICE_NATIVE_ENDIAN** signifie que les œufs seront mangés du même côté qu'ils le sont sur l'architecture invité. Étant sur une architecture ARM, sous Linux, le boutisme sera donc du little-endian.

Le troisième argument est un pointeur sur l'argument qui sera passé à la fonction **read** et **write**, permettant à ces dernières de retrouver la structure de données privées de l'instance du périphérique. **IMX27_SOC_DEVICE_NAME** pointe sur une chaîne de caractère contenant le nom du périphérique. Ce nom est juste informatif est sera affiché par la commande **info qtree** du moniteur Qemu. **IMX27_GTP_IO_MEM_SIZE** correspond à la taille de la zone mémoire requise. Cette taille est complètement dépendante du matériel et il convient de trouver sa valeur dans la documentation technique du constructeur [3] du périphérique réel. Dans notre cas cette zone à une taille de 4 Ko (0x1000). Il faut remarquer qu'aucune information concernant l'adresse absolue dans l'espace mémoire de la cible n'est fournie. Il y a plusieurs raisons à cela. Un périphérique peut être utilise sur plusieurs machines Qemu différentes et son adresse physique n'est pas obligatoirement la même. Par ailleurs le même périphérique peut exister en plusieurs exemplaires au sein d'une même machine. Leurs adresses de base seront obligatoirement différentes. C'est pour cela que le mapping mémoire est fait au moment où le périphérique est instancié par l'appel:

```
sysbus_create_simple("imx27_soc", MX27_CCM_BASE_ADDR, 0);
```

Le second paramètre **MX27_CCM_BASE_ADDR** fixe l'adresse physique du périphérique émulé. Ce mapping mémoire est fait par la fonction interne **sysbus_mmio_map()**.L'initialisation du module se termine par un appel à **sysbus_init_mmio_region()** qui lie la zone mémoire précédemment réservée à l'instance courante du

module de périphérique.

Si le membre **qdev.reset** de la structure **SysBusDeviceInfo** n'avait pas été renseignée avec **imx27_reset_hard()** et si l'enregistrement du périphérique n'avait pas été faite avec **sysbus_register_withprop()**, il conviendrait d'appeler manuellement, à la fin de la fonction d'initialisation, la fonction de mise à zéro matérielle du périphérique émulé. Cette dernière affecte simplement aux membres de la structure **imx27_soc_state** les valeurs du tableau vues au 5.1 ainsi que la valeur du Chip ID.

À ce stade, le périphérique est utilisable par le système invité émulé par Qemu. Lorsque l'invité accédera aux adresses comprises entre **MX27_CCM_BASE_ADDR** et **MX27_CCM_BASE_ADDR + 0x0fff**, les fonctions **imx27_soc_read()** ou **imx27_soc_write()** seront invoquées. Elles recevront en arguments un pointeur sur la structure **imx27_soc_state** de l'instance courante et le décalage (**offset**) par rapport à l'adresse de base **MX27_CCM_BASE_ADDR**. L'activité à remplir par le module SoC étant triviale, voici les opérations réalisées :

– Les différents registres émulés étant uniquement utilisés en lecture par Linux, la fonction **imx27_soc_write()** retourne sans rien faire.

– Pour la fonction **imx27_soc_read()**, une suite de **case** identifie le registre accédé et retourne la valeur voulue.

5.3 le Module timer système i.MX27

Maintenant que nous connaissons les déclarations « administratives » pour créer un module Qemu, nous pouvons aborder l'émulation d'un périphérique qui fait réellement quelque chose : le timer GTP1 de l'i.MX27.

Mais avant de détailler l'émulation, il est souhaitable faire un peu plus connaissance avec les timers de l'i.MX27 qui peuvent fonctionner selon différents modes. Cependant le noyau Linux n'utilise que le premier timer, GPT1, selon un mode précis. Ce qui n'est pas utilisé par le noyau ne sera pas implémenté.

Un GPT compte les impulsions arrivant sur son entrée de comptage et, après une éventuelle pré-division, il stocke cette valeur dans son registre de comptage TCN. Linux n'utilisant pas de pré-division, ce registre est incrémenté au rythme de l'horloge PER1 (présentée au 5.1) à 16,625 MHz. Un GPT peut réagir à deux types d'événement :

– Une impulsion sur son entrée de capture. La valeur courante du compteur est alors stockée dans le registre de capture, TCR, et une interruption est générée. Cela permet de dater précisément un événement. Linux n'utilise pas cette fonctionnalité.

– Le compteur atteint la valeur de consigne renseignée dans le registre de comparaison TCMP. Une interruption est alors émise et le comptage reprend. Suivant que le GPT est configuré en mode de comptage « restart » ou « free run », le compteur repart de zéro dans le premier cas ou continue à partir de la dernière valeur dans le second. En mode « free run », lorsque le registre de comptage TCN atteint 0xFFFFFFFF, le comptage continue naturellement à partir de 0. Cela se produit environ toutes les 258 secondes avec une horloge à 16,625 MHz. Linux utilise le mode « free run ».

Voyons à présent comme Linux utilise le GPT1. Il commence par l'initialisé dans la fonction, déjà évoquée, **mxc_timer_init()** de **arch/arm/plat-mxc/time.c** :

```
static struct irqaction mxc_timer_irq = {
	.name		= "i.MX Timer Tick",
	.flags		= IRQF_DISABLED | IRQF_TIMER | IRQF_IRQPOLL,
	.handler	= mxc_timer_interrupt,
};
void __init mxc_timer_init(struct clk *timer_clk, void __iomem *base, int irq) {
	uint32_t tctl_val;
	clk_enable(timer_clk);
	timer_base = base;
	__raw_writel(0, timer_base + MXC_TCTL);
	__raw_writel(0, timer_base + MXC_TPRER);
	if (timer_is_v2())
		tctl_val = V2_TCTL_CLK_IPG_32k | V2_TCTL_FRR | V2_TCTL_WAITEN |
MXC_TCTL_TEN;
	else
		tctl_val = MX1_2_TCTL_FRR | MX1_2_TCTL_CLK_PCLK1 | MXC_TCTL_TEN;
	__raw_writel(tctl_val, timer_base + MXC_TCTL);
	mxc_clocksource_init(timer_clk);
	mxc_clockevent_init(timer_clk);
	setup_irq(irq, &mxc_timer_irq);
}
```

clk_enable(timer_clk) active l'horloge **gpt1_clk**. Ensuite le registre du pré-diviseur **MXC_TPRER** est mis à

zéro et le timer est arrêté et placé dans une configuration connue en mettant à zéro son registre de contrôle **MXC_TCTL**. Les GPT de l'i.MX27 étant de version 1, le registre de contrôle prend la valeur **MX1_2_TCTL_FRR | MX1_2_TCTL_CLK_PCLK1 | MXC_TCTL_TEN** :

– **MX1_2_TCTL_FRR** active le mode « free run ».

– **MX1_2_TCTL_CLK_PCLK1** sélectionne l'horloge PER1 comme source de comptage.

– **MXC_TCTL_TEN** lance le timer.

Puis **mxc_clocksource_init()** crée la source globale d'horloge comme nous l'avons déjà vu. **mxc_clockevent_init()** crée une source d'événements d'horloge, de type **clock_event_device**, associée à la source globale d'horloge. Pour être concis, cela a pour effet de créer une source d'événements que l'on peut visualiser dans **/proc/timer_list** :

```
Tick Device: mode:     1
Per CPU device: 0
Clock Event Device: mxc_timer1
 max_delta_ns:   258343851419
 min_delta_ns:   15338
 mult:           71403844
 shift:          32
 mode:           3
 next_event:     6312030000000 nsecs
 set_next_event: mx1_2_set_next_event
 set_mode:       mxc_set_mode
 event_handler:  hrtimer_interrupt
 retries:        0
```

mx1_2_set_next_event, **mxc_set_mode** et **hrtimer_interrupt** sont des noms de fonctions qui sont affichés seulement si l'option de compilation du noyau **CONFIG_KALLSYMS** est activée. Pour finir l'initialisation, la routine d'interruption du timer GPT1 est mise en place avec l'appel de **setup_irq()**.

Le détail des interactions entre le noyau Linux et le GPT1 nécessiterait d'aborder les timers haute résolution de Linux ([12] chapitre 15.4), ce qui sort du cadre de l'article. La présentation sera donc moins précise, mais suffisamment pour bien comprendre le mécanisme.

Lorsque la source d'événements **mxc_timer1** est initialisée, la fonction **mxc_set_mode()** appelle **gpt_irq_enable()** de **arch/arm/plat-mxc/time.c** qui configure le GPT1 pour qu'il génère une interruption lorsque le registre de comptage atteint la valeur du registre de comparaison. Cette interruption déclenche la fonction **mxc_timer_interrupt()** spécifique à l'architecture i.MX. Cette dernière lance la fonction globale d'interruption des timers haute résolution **hrtimer_interrupt()** pointée par le membre **event_handler** de **mxc_timer1**. Elle effectue un certain nombre de tâches dont le réarmement (écrire une nouvelle valeur dans le registre de comparaison) du GPT1 pour qu'il génère l'interruption suivante. Ce réarmement est réalisé par la fonction **mx1_2_set_next_event()**, spécifique à l'architecture i.MX, pointée par le membre **set_next_event** de **mxc_timer1** :

```
static int mx1_2_set_next_event(unsigned long evt, struct clock_event_device *unused)  {
       unsigned long tcmp;
       tcmp = __raw_readl(timer_base + MX1_2_TCN) + evt;
       __raw_writel(tcmp, timer_base + MX1_2_TCMP);
       return (int)(tcmp - __raw_readl(timer_base + MX1_2_TCN)) < 0 ? -ETIME : 0;
}
```

evt représente le délai au bout duquel une interruption devra à nouveau être générée. Elle est fournie par le sous-système générique de gestion de temps de Linux en fonction des besoins. Il n'est pas exprimé en nanosecondes mais en unités d'incrément de temps de l'horloge **gpt1_clk**. La valeur courante du compteur de GPT1 est lue, puis le registre de comparaison est programmé avec cette valeur augmentée de **evt**.

Cette description du fonctionnement réel du GPT1 se révèle extrêmement utile pour imaginer comment implémenter son émulation. D'abord, il faut une source de temps, et **vm_clock**, l'horloge du système invité, largement présenter le mois dernier, nous la fournira. Ensuite, le module doit être capable de déclencher un traitement lorsqu'un délai expire, comme le fait un GPT. Un timer Qemu **QEMUTimer** synchronisé sur **vm_clock** sera parfait. Sa fonction de callback, appelée lors de l'expiration, simulera une interruption matérielle si la configuration courante le requiert (bit 4 du registre de contrôle du GPT1). Cette interruption matérielle nous est offerte par la structure **qemu_irq**. Les éléments principaux de la quincaillerie virtuelle sont là. Il faudra aussi convertir les rythmes des horloges, car **vm_clock** est cadencées à 1 GHz alors que **gpt1_clk** l'est à 16,625 MHz. Ensuite, les registres de contrôle, de comparaison, du compteur et de status devront être simulés en lecture et en écriture.

L'émulation obtenue, en suivant le synoptique ci-dessus, est minimale et très parcellaire, mais elle suffit à la bonne marche de Linux. Il n'est pas du tout sûr qu'elle fonctionne sur d'autres systèmes d'exploitation. Les bases étant posées, passons au détail de l'implémentation dont voici le code :

```
#ifdef CONFIG_APF27
#include "sysbus.h"
#include "qemu-timer.h"

#define IMX27_GPT_DEVICE_NAME "imx27_timer"
#define IMX27_GPT_DEVICE_DESC "i.MX27 General Purpose Timer"
#define IMX27_GTP_IO_MEM_SIZE 0x00001000
#define TIMER_MAX  0xFFFFFFFFUL
#define QEMU_VMCLOCK_FREQ 1000000000UL
#define GPT_FREQ    16625003UL
#define TCTL 0x00
#define TPRER 0x04
#define TCMP 0x08
#define TCR 0x0c
#define TCN 0x10
#define TSTAT 0x14
#define TCTL_TEN (1 << 0)
#define TCTL_COMP_EN (1 << 4)
#define TCTL_FRR (1 << 8)
#define TCTL_SWR (1 << 15)

typedef struct {
       SysBusDevice busdev;
       MemoryRegion iomem;
       qemu_irq irq;
       QEMUTimer *timer;
       uint32_t tctl;
       uint32_t tcmp;
       uint32_t tcn;
       uint32_t tstat;
} imx27_timer_state;

static void imx27_reset_hard(DeviceState *dev) {
       imx27_timer_state *state =
               FROM_SYSBUS(imx27_timer_state, sysbus_from_qdev(dev));
       state->tctl = 0x00000000;
       state->tcmp = 0xFFFFFFFF;
       state->tcn = 0x00000000;
       state->tstat = 0x00000000;
}

static void imx27_reset_soft(imx27_timer_state *state) {
       state->tctl |= 0x00000001;
       state->tcmp = 0xFFFFFFFF;
       state->tcn = 0x00000000;
       state->tstat = 0x00000000;
}

static void imx27_timer_timeout(void *opaque) {
       imx27_timer_state *state = (imx27_timer_state *) opaque;
       if (state->tctl & TCTL_COMP_EN){
              state->tstat = 1;
              qemu_irq_raise(state->irq);
       } else {
              qemu_irq_lower(state->irq);
       }
}

static uint64_t imx27_timer_update_count(imx27_timer_state *state) {
    uint64_t clk = qemu_get_clock_ns(vm_clock);
    state->tcn = ((uint32_t)muldiv64(clk, GPT_FREQ,QEMU_VMCLOCK_FREQ));
    return clk;
}

static void imx27_timer_rearm(imx27_timer_state *state) {
       uint64_t diff;
       uint64_t clk = imx27_timer_update_count(state);
       if (state->tcmp > state->tcn) {
              diff = state->tcmp - state->tcn;
       } else {
              diff = (TIMER_MAX - state->tcn) + state->tcmp;
```

```
        }
        qemu_mod_timer(state->timer, clk + muldiv64(diff, QEMU_VMCLOCK_FREQ,GPT_FREQ));
}

static uint64_t imx27_timer_read(void *opaque, target_phys_addr_t offset,
         unsigned size) {
        imx27_timer_state *state = (imx27_timer_state *) opaque;
        switch (offset) {
                case TCTL:
                        return state->tctl;
                case TCMP:
                        return state->tcmp;
                case TCN:
                        imx27_timer_update_count(state);
                        return state->tcn;
                case TSTAT:
                        return state->tstat;
        }
        return 0;
}

static void imx27_timer_write(void *opaque, target_phys_addr_t offset,
         uint64_t value, unsigned size) {
        imx27_timer_state *state = (imx27_timer_state *) opaque;
        switch (offset) {
                case TCTL:
                        if (value & TCTL_SWR) {
                                imx27_reset_soft(state);
                                return;
                        }
                        if (value & TCTL_TEN) {
                                if (!(state->tctl & TCTL_TEN)) {
                                        imx27_timer_rearm(state);
                                }
                        }
                        if (!(value & TCTL_TEN)) {
                                state->tcn=0;
                        }
                        state->tctl = value;
                        return;
                case TCMP:
                        state->tcmp = value;
                        imx27_timer_rearm(state);
                        return;
                case TSTAT:
                        state->tstat = 0;
                        return;
                default:
                        return;
        }
}

static const MemoryRegionOps imx27_timer_ops = {
        .read = imx27_timer_read,
        .write = imx27_timer_write,
        .endianness = DEVICE_NATIVE_ENDIAN,
};

static int imx27_timer_init(SysBusDevice *dev) {
        imx27_timer_state *state = FROM_SYSBUS(imx27_timer_state, dev);
        memory_region_init_io(&state->iomem, &imx27_timer_ops, state,
                        IMX27_GPT_DEVICE_NAME, IMX27_GTP_IO_MEM_SIZE);
        sysbus_init_mmio_region(dev, &state->iomem);
        sysbus_init_irq(dev, &state->irq);
        state->timer = qemu_new_timer_ns(vm_clock, imx27_timer_timeout, state);
        return 0;
}

static void imx27_timer_register_device(void) {
        SysBusDeviceInfo *info = g_malloc0(sizeof(*info));
        info->qdev.name = IMX27_GPT_DEVICE_NAME;
```

```
        info->qdev.desc = IMX27_GPT_DEVICE_DESC;
        info->qdev.size = sizeof(imx27_timer_state);
        info->qdev.reset = imx27_reset_hard;
        info->init = imx27_timer_init;
        sysbus_register_withprop(info);
}

device_init(imx27_timer_register_device)
#endif
```

Nous ne reviendrons pas sur les opérations d'enregistrement et d'initialisation similaires à celles du module précédent. La structure **imx27_timer_state** est destinée à contenir les données privées du module timer :

– Les membres **busdev** et **iomem** ont le même rôle que dans le module SoC.

– **irq** est un pointeur de type **qemu_irq** déjà présenté dans la section 4. i**rq** est utilisée pour simuler l'interruption matérielle générée par le timer GPT1 lorsque son registre de compteur TCN1 atteint la valeur de son registre de comparaison TCMP1.

– **timer** est un pointeur sur une structure **QEMUTimer**. C'est un timer Qemu qui se déclenche lorsque son temps de consigne est atteint par l'appel d'une fonction de callback. Il est utilisé pour compter le temps qui s'écoule comme le ferait le timer GPT1 d'un i.MX27 physique.

– Les entiers de 32 bits non signés **tctl**, **tcmp**, **tcn** et **tstat** émulent respectivement les registres de contrôle, de comparaison, de comptage et de statut du timer GPT1.

Cette structure sera utilisée par toutes les fonctions implémentant l'émulation. Dans la fonction d'initialisation **imx27_timer_init()**, **sysbus_init_irq()** est utilisée pour initialiser la structure **irq**. Elle est attachée à l'instance courant du périphérique **SysBusDevice**. Lors de l'instanciation du module de périphérique dans un module de machine avec la fonction **sysbus_create_simple()**, le troisième argument passé, dans notre cas 26 selon la documentation constructeur, connectera cette interruption à la ligne correspondante du contrôleur d'interruption.

qemu_new_timer_ns() initialise le timer Qemu utilisé pour simuler le GPT1. Le premier argument fourni l'horloge sur laquelle le timer sera synchronisé, **vm_clock**. Le second est un pointeur de fonction sur la fonction de callback, **imx27_timer_timeout()**, qui sera exécutée lors de l'expiration du timer. Le troisième est un pointeur qui sera passé en argument de cette fonction. C'est bien sûr la structure de données privées de l'instance du périphérique qui sera transmise.

La fonction **imx27_reset_hard()**, appelée lors de l'instanciation du module, affecte aux quatre registres émulés leurs valeurs par défaut fournies par la documentation constructeur. **imx27_reset_soft()** effectue la même chose, sauf que le bit 0 du registre TCTL est préservé selon la même documentation. Elle est appelée lorsque bit 15 SWR de TCTL est positionné à 1.

La fonction d'accès en lecture **imx27_timer_read()** est triviale et retourne simple la valeur du registre demandée, stockée dans la structure **imx27_timer_state**. Cependant, pour le registre du compteur TCN, la fonction de mise à jour du compteur **imx27_timer_update_count()** sera appelée au préalable pour retourner la valeur courante du compteur.

L'accès en écriture, géré par **imx27_timer_write()** est un peu plus complexe, avec un **case** par registre :

– Pour le registre de contrôle TCTL :

- L'écriture d'un 1 sur le bit 15 SWR provoquera l'appel de la remise à zéro logicielle, **imx27_reset_soft().**

- La transition 0 → 1 du bit 0 TEN provoquera le démarrage du timer Qemu via la fonction de réarmement **imx27_timer_rearm()**.

- L'écriture d'un 0 sur bit 0 TEN provoquera la mise à zéro du registre TCTL pour désactiver toute activité du GPT1 émulé.

- Le bit 4, COMP EN, quand il vaut 1, active la génération d'une interruption lorsque le compteur atteint la valeur de comparaison. Il est simplement stocké dans **tctl** et ne nécessite pas de traitement particulier lors de son affectation.

– Pour le registre de comparaison TCMP, l'écriture d'une valeur provoquera son stockage dans le membre **tcmp** de **imx27_timer_state**, puis, le réarmement du timer Qemu avec cette nouvelle valeur sera lancé par **imx27_timer_rearm()**.

– Lorsque le registre de statut TSTAT est accédé en écriture, c'est uniquement pour effacer le bit 0, drapeau signalant un événement de comparaison. Ce registre sera simplement mis à zéro.

imx27_timer_update_count() met à jour le membre **tcn** de **imx27_timer_state** et retourne la valeur de

vm_clock obtenu grâce à **qemu_get_clock_ns()**. Le comptage effectué par **vm_clock** étant au rythme de 1 GHz, il faut le convertir au rythme de **gpt1_clk** à 16,625 MHz. C'est réalisé avec le ratio **GPT_FREQ/QEMU_VMCLOCK_FREQ** et de la fonction de multiplication division 64 bits **muldiv64()** fournie par Qemu. Le résultat subi un cast vers un entier de 32 bits non signé, taille du registre réel du GPT1. Le débordement est ainsi géré naturellement.

La fonction **imx27_timer_rearm()** réarme de timer Qemu pour qu'il se déclenche à nouveau lorsque le compteur atteindra la nouvelle valeur spécifiée dans **tcmp**. Comme le noyau Linux a configuré le timer GPT1 de l'i.MX27 en mode « free run », il est possible que cette nouvelle valeur soit inférieure à la valeur courante du compteur **tcn**. C'est le cas de « roll over », où **tcn** retombe à **0x00000000** après avoir atteint **0xFFFFFFFF**. Suivant le cas, « roll over » ou non, la différence relative entre **tcmp** et **tcn** est stockée dans **diff**. Le timer Qemu est ensuite réarmé avec une nouvelle valeur, située dans le futur, via la fonction **qemu_mod_timer()** qui prend en argument un pointeur sur la structure du timer à modifier et cette valeur. Celle-ci correspond à une date absolue de l'horloge **vm_clock** exprimée en nanosecondes. Elle est calculée en additionnant la valeur courante de **vm_clock** et la valeur **diff** convertie en nanosecondes.

Lorsque cette date est atteinte et que le timer Qemu se déclenche, la fonction de callback de celui-ci, **imx27_timer_timeout()**, est appelée. Si la génération d'interruption de comparaison est activée (bit 4 **TCTL_COMP_EN** de **tctl**) une interruption matérielle est simulée via un appel à **qemu_irq_raise()**. Linux déclenchera alors la fonction de traitement d'interruption du noyau **mxc_timer_interrupt()**. Celle-ci lance les traitements présentés au début de cette section. Parmi eux, la mise à jour du registre de comparaison **tcmp** sera effectuée, ce qui déclenchera la fonction de réarmement **imx27_timer_rearm()**. Cet enchaînement mutuel d'actions entre le noyau Linux et le module Qemu du timer émulé crée un cycle auto-entretenu identique à celui présent sur un système physique. Il est présenté en figure 4.

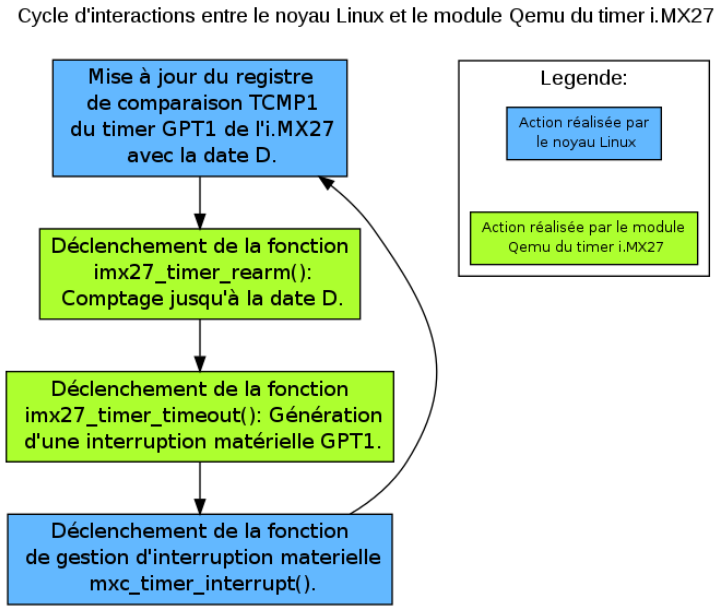

Fig. 4 : Interactions entre le noyau Linux et le module Qemu du timer i.MX27 : Une symbiose purement logicielle.

Nous arrivons au terme de la séquence de dissection. Finalement, le module du timer représente moins de 150 lignes de code et ses fonctions peuvent presque sembler triviales. Cependant, elles suffisent à implémenter un composant absolument essentiel d'un système sur lequel s'exécute Linux. L'élégance de l'architecture de Qemu en est certainement responsable. La difficulté ne provient pas de la complexité du code à produire. Mais avant de le créer, il est indispensable d'avoir parfaitement compris comment le composant fonctionne et la manière avec laquelle il est utilisé. Cette première tâche peut s'avérer longue et complexe. C'est la première difficulté. La seconde provient certainement de notre culture de programmeur. Nous avons l'habitude d'écrire du code qui agit sur du matériel pour obtenir un résultat. Ce matériel a un comportement connu et prévisible. Écrire du code émulant du matériel ressemble à la résolution d'un problème dual : une interruption n'est pas traitée, mais elle est générée. Ce changement de position peut être déroutant au premier abord.

6. Mise en œuvre

À présent que nous disposons de toutes les parties de la nouvelle machine virtuelle, nous allons l'assembler. Cela passe par une légère modification du système de compilation de Qemu basé **make** et un script **configure**. Il faut pouvoir compiler les modules additionnels en ajoutant au fichier **Makefile.target** :

```
QEMU_CFLAGS += "-DCONFIG_APF27"
obj-arm-y += imx_timer.o
obj-arm-y += imx_serial.o
obj-arm-y += imx27_timer.o
obj-arm-y += imx27_soc.o
obj-arm-y += apf27.o
```

La première ligne ajoute **-DCONFIG_APF27** à toutes les commandes GCC lancées. Elle définit la constante **CONFIG_APF27** nécessaire aux inclusions conditionnelles. La directive **obj-arm-y += code.o** ajoute le fichier **code.c** à la liste des dépendances du fichier final **qemu-system-arm** lorsque la cible **arm-softmmu** a été configurée avec le script **configure**.

Les grandes étapes de la mise en œuvre seront :

– Télécharger, patcher, configurer et compiler Qemu avec les nouveaux fichiers et les modifications relatives à la nouvelle machine APF27.

– Installer et configurer un BSP Armadeus pour obtenir les outils permettant la compilation d'un noyau Linux ARM et la construction de son rootfs associé au format *cpio*.

– Modifier la configuration du noyau Linux APF27 pour qu'il fonctionne sur la carte virtuelle minimale APF27.

– Exécuter GNU/Linux sur la carte APF27 émulée par Qemu.

6.1 Prérequis

Avant de s'atteler à la construction de notre système complet, mieux vaut s'assurer que la station de développement possède tous les outils nécessaires. Sur une distribution Debian, cette commande devrait combler d'éventuelles lacunes, c'est la liste des paquets requis à construction du BSP Armadeus et de Qemu :

```
# apt-get install -y libsdl1.2-dev libglib2.0-dev libcurl4-openssl-dev libpng12-dev
libjpeg8-dev libssh2-1-dev libldap2-dev libdirectfb-dev libsvga1-dev build-essential gcc
g++ autoconf automake libtool bison flex gettext patch subversion texinfo wget git-core
libncurses5 libncurses5-dev zlib1g-dev liblzo2-2 liblzo2-dev libacl1 libacl1-dev gawk cvs
curl lzma uuid-dev mercurial
```

Sur d'autres distributions, il faudra chercher les noms des paquets équivalents et les installer avec l'outil ad hoc.

6.2 Patcher et compiler Qemu

Commençons par télécharger et décompresser les sources de Qemu sur le site officiel et l'archive contenant les patchs que j'ai mis en ligne à l'occasion de cette séance de travaux pratiques :

```
$ cd
$ wget http://wiki.qemu.org/download/qemu-1.0.tar.gz
$ wget http://www.embedded-wire.com/qemu-patch-APF27/patch-apf27-qemu-1.0-00.00.00.tgz
$ tar zxvf qemu-1.0.tar.gz
$ tar zxvf patch-apf27-qemu-1.0-00.00.00.tgz
```

Maintenant, appliquons les patchs :

```
$ cd qemu-1.0
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/Makefile.target.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/apf27.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/arm_boot.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/imx27_soc.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/imx27_timer.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/imx_avic.patch
$ patch -p1 < ../patch-apf27-qemu-1.0-00.00.00/imx_serial.patch
```

Les fichiers **hw/apf27.c**, **hw/apf27.**h, **hw/imx27_soc.c**, **hw/imx27_timer.c**, **hw/imx_avic.c**, **hw/imx_serial.c** devraient apparaître. À présent que tous les morceaux sont là, configurons Qemu :

```
$ ./configure --prefix=$HOME/qemu-1.0-APF27 --target-list=arm-softmmu --disable-user
--enable-sdl --extra-cflags=-save-temps --enable-debug
```

Passons en revue ces différents paramètres :

– **`--prefix=$HOME/qemu-1.0-APF27`** est le répertoire dans lequel Qemu sera installé.

– **`--target-list=arm-softmmu`** active l'émulation d'un système ARM complet

– **`--disable-user`** désactive le support en mode utilisateur. Sans cette option, il est possible de lancer l'émulation d'un programme en mode utilisateur sans un système émulé complet.

– **`--enable-sdl`** active le support de la bibliothèque SDL qui permet, par exemple, d'avoir l'émulation d'un frame buffer ou bien un moniteur distinct du terminal.

– **`--extra-cflags=-save-temps`** permet de conserver les fichiers temporaires de GCC (voir première partie). Les expansions de macros obscures seront plus faciles à comprendre ainsi.

– **`--enable-debug`** active la génération des symboles de débuggage pour pouvoir lancer confortablement Qemu sous le contrôle de GDB.

La compilation et l'installation sont lancées par :

```
$ make -j 4 install
```

L'option **`-j 4`** permet de lancer quatre jobs en parallèle et d'utiliser autant de cœurs de votre processeur. La commande qui suit permet de vérifier que Qemu intègre bien la nouvelle machine APF27 :

```
$ ~/qemu-1.0-APF27/bin/qemu-system-arm -M ?|grep apf27
apf27      Armadeus APF27 Board (ARM926EJ-S)
```

6.3 Installer et configurer le BSP Armadeus

Pour pouvoir lancer un système GNU/Linux complet, il nous faut deux éléments : une image de noyau Linux et une image de système de fichiers racine, que nous appellerons *rootfs*. Armadeus fourni pour sa carte APF27 un BSP qui permet de les produire. Un BSP (Board Support Package) est en quelque sorte un SDK (Software Development Kit) spécialisé pour une carte particulière. Celui d'Armadeus est basée sur Buildroot, GCC et la µClibc. Il permet de générer la chaîne de compilation croisée, le boot loader U-Boot, le noyau Linux ainsi que le rootfs. Lors de son utilisation sur une carte APF27, les options par défaut fonctionnent. Cependant, nous aurons besoin de les adapter à notre APF27 virtuel, qui, ne possédant pas de périphérique en mode blocs, n'accepte pas d'image de *rootfs* classique. Elle nécessite une image de disque RAM initial au format *cpio*. Voici comment télécharger et configurer le BSP pour une carte APF27 :

```
$ cd
$ wget http://freefr.dl.sourceforge.net/project/armadeus/armadeus/armadeus-4.1/armadeus-
4.1.tar.bz2
$ tar jxvf armadeus-4.1.tar.bz2
$ cd armadeus-4.1
$ make apf27_defconfig
```

Après de nombreux messages, le menu de configuration de Buildroot apparaît dans le style du *menuconfig* du noyau Linux. Voici les différents points à modifier :

Build options ---> Number of jobs to run simultaneously 4

Permet à Builroot de lancer jusqu'à quatre jobs en parallèle. Sur un seul cœur, le processus complet de compilation prend plus d'une heure. Aujourd'hui, Buildroot gère bien le parallélisme et en fixant ce paramètre à 4, le temps de compilation descend à environ à 20 minutes sur ma machine quadri-coeurs vieillissante.

Toolchain ---> (2.6.38.8) linux version

Spécifie la version des fichiers d'en-tête du noyau Linux à utiliser dans la chaîne de compilation croisée. Nous utiliserons la version la plus ressente pour le BSP 4.1.

Toolchain ---> GDB debugger Version (gdb 7.1) --->

Permet d'obtenir un GDB le plus ressent possible.

System configuration ---> (ttymxc0) Port to run a getty (login prompt) on

Indique quel fichier spécial **`getty`** doit utiliser. Les fichiers spéciaux de ports série ayant changés de nom pour l'architecture i.MX (de **`/dev/ttySMX`** vers **`/dev/ttymxc`**), il faut impérativement le modifier sous peine de n'avoir pas de login.

Filesystem images ---> [ ] jffs2 root filesystem

Filesystem images ---> [ ] ubifs root filesystem

Filesystem images ---> [ ] tar the root filesystem

Filesystem images ---> [*] cpio the root filesystem

Désactive la création des images de *roofs* aux formats *jffs2*, *ubifs* et *tar*, car elles sont inutiles. Active la production d'une image au format *cpio*.

Kernel ---> (2.6.38.8) Kernel version

Indique la version du noyau Linux à compiler.

Tapez *Esc* deux fois pour quitter le menu de configuration de Buildroot et sauvegardez la configuration. Puis lancez le processus avant d'aller prendre un café…

```
$ make
Your Armadeus BSP (version 4.1 for apf27) was successfully built !
    gcc: 4.4.6
    libc: uClibc 0.9.30.3
    busybox: 1.18.5
    U-Boot: 1.3.4
    Linux: 2.6.38.8

Build time: 1249 seconds
```

Voici les précieux fichiers produits dans **buildroot/output/images/** :

– **apf27-linux.bin** est le noyau Linux au format U-Boot *uImage*.

– **apf27-rootfs.cpio** est le rootfs au format *cpio*.

– **apf27-u-boot.bin** est l'image brut du boot loader U-Boot.

Seules les deux premières nous seront utiles, car nous n'utiliserons pas U-Boot, mais c'est cependant possible.

6.4 Modifier la configuration du noyau Linux

La configuration du noyau Linux est lancée, dans le répertoire du BSP, par la commande suivante :

```
$ make linux26-menuconfig
```

Par défaut, la configuration du noyau Linux fournie par le BSP comporte les pilotes de tous les périphériques disponibles dans le SoC i.MX27. La plupart d'entre eux sont directement compilés dans le noyau et il n'est pas possible d'inactiver leur chargement comme cela serait possible avec des modules. Deux de ces pilotes posent problème : celui de la mémoire flash NAND et celui du contrôleur USB. En effet, leur fonction d'initialisation ne rend jamais la main lorsque le périphérique n'est pas présent. La seule solution est alors de les désactiver dans la configuration du noyau :

Device Drivers ---> < > Memory Technology Device (MTD) support --->

Device Drivers ---> [ ] USB support --->

L'autre point à modifier est le support du disque RAM initial qui n'est pas actif par défaut :

General setup ---> [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support

Cela peut être aussi occasion d'activer les options de débuggage présentées au 1.1. La recompilation sera simplement lancée par **make**. Si vous rencontrez des problèmes lors de la construction du BSP, vous pourrez tout d'abord consulter le Wiki d'Armadeus Project [13]. Si vous n'y trouvez pas la solution, vous pourrez envoyer un message sur la liste de diffusion de l'association à l'adresse armadeus-forum@lists.sourceforge.net en indiquant que vous tentez d'utiliser l'émulation Qemu, ainsi que le problème que vous rencontrez. Je me ferai un plaisir de vous répondre.

6.6 Exécuter le système émulé APF27

Tous les ingrédients sont près pour le lancement du système émulé, dont voici l'incantation :

```
cd ~/armadeus-4.1/buildroot/output/images/
~/qemu-1.0-APF27/bin/qemu-system-arm -M apf27 -kernel apf27-linux.bin -append
"console=ttymxc0" -nographic -initrd apf27-rootfs.cpio
Uncompressing Linux... done, booting the kernel.
…
Welcome to the Armadeus development environment.
armadeus login: root
# uname -a
Linux armadeus 2.6.38.8 #2 PREEMPT Sat Mar 10 15:33:19 CET 2012 armv5tejl GNU/Linux
```

Voici la signification des différentes options :

– **`-M apf27`** indique le type de machine à émuler.

– **`-kernel apf27-linux.bin`** spécifie l'image du noyau Linux à utiliser.

– **`-append "console=ttymxc0"`** fournit les arguments du noyau Linux.

– **`-nographic`** stipule qu'il ne faut pas utiliser SDL, car aucun frame buffer n'est émulé.

– **`-initrd apf27-rootfs.cpio`** désigne l'image de disque RAM initial à utiliser.

Après l'affichage des messages d'initialisation du noyau, un prompt de login apparaît. Le compte root ne comporte pas de mot de passe. Les commandes UNIX standard sont alors utilisables.

Conclusion

Nous voici arrivés au terme de la seconde partie de l'exploration de Qemu. Après être passé par son cœur, le TCG, puis par ses horloges, nous sommes arrivés sur un terrain plus expérimental, la création d'une nouvelle machine. Certes, celle-ci est bien limitée et presque autiste. Mais le canevas de base est bien présent. C'est une preuve de plus que dans le domaine du logiciel libre, seule l'imagination limite les possibles. Naturellement c'est au prix de quelques efforts de bonne curiosité. Cette première ébauche de machine ne demande qu'à être complétée par d'autres modules de périphériques comme sa carte Ethernet ou son frame buffer. Un auteur… Pourquoi pas vous ?

Yvan Roch – yvan.roch@gmail.com
Ingénierie et formations sur Linux embarqué et les PKI – http://www.embedded-wire.com

Références

[1] Pierre Ficheux, « Mise au point à distance avec GDB et QEMU », *OpenSilicium* n°1 (janvier 2011)

[2] Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)

[3] Freescale, *MCIMX27 Multimedia Applications Processor Reference Manual*, http://cache.freescale.com/files/32bit/doc/ref_manual/MCIMX27RM.pdf?fpsp=1&WT_TYPE=Reference%20Manuals&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation

[4] Armadeus Systems, APF27 Datasheet, http://www.armadeus.com/_downloads/apf27/documentation/datasheet_apf27.pdf

[5] Code source de Qemu, http://wiki.qemu.org/download/qemu-1.0.tar.gz

[6] qemu-devel Archives, http://lists.nongnu.org/archive/html/qemu-devel/

[7] Peter Chubb, Patch Qemu i.MX31, http://lists.nongnu.org/archive/html/qemu-devel/2011-11/msg03505.html

[8] Yvan Roch, « Qemu : Visite au cœur de l'émulateur », *GNU Linux Magazine France* n° 147 (mars 2012)

[9] ARM Architecture Reference Manual, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html

[10] Armadeus Systems, Schéma électrique de l'APF27, http://www.armadeus.com/_downloads/apf27/hardware/apf27_V1.2.pdf

[11] Et voilà le Shadok : 18e épisode, « Plus ça rate, plus on a de chance que ça marche… », 1

http://www.ina.fr/fictions-et-animations/animation/video/CPF89006216/et-voila-le-shadok-18eme-episode.fr.html

[12] Wolfgang Mauerer, *Linux® Kernel Architecture* , Wiley Publishing, Inc. (2008)

[13] Armadeus Project Wiki, http://www.armadeus.com/wiki/