Systèmes embarqués 1/6

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

8 septembre 2015

Rappels

- Travail sur microcontrôleur : pas de processus, un programme en C converti en assembleur s'exécute sur le processeur
- Unique risque de perturber l'exécution séquentielle du programme : interruptions
- Aucune abstraction, écriture directe dans les registres de configuration du matériel
- Aucun mécanisme de partage des ressources ou de protection de la mémoire entre tâches
- Système d'exploitation (Linux) :
  - mécanismes d'abstraction du matériel (pilotes, `/dev` et `/sys/bus`
  - notion de processus, chaque tâche possède sa mémoire
  - mécanisme d'abstraction des plages mémoires : MMU

Plan

1. Contrôle du matériel depuis l'espace utilisateur, lien entre espaces mémoire virtuelle et matérielle, *buildroot*[1]
2. réseaux IP, TCP, UDP
3. Programmation en espace noyau – pilote dédié et module noyau chargé dynamiquement
4. Diverses interfaces utilisateur-noyau : `/dev`, `/sys/bus`, IIO
5. sources de latences, applications temps-réel, extension Linux au temps réel : Xenomai

1. `make a13_olinuxino_micro_xenomai_defconfig` et `make a13_olinuxino_micro_defconfig`

Accès au matériel depuis l'espace utilisateur

Au travers de `/sys/class`

- échange d'informations en trames ASCII
- configuration au travers du fichier approprié (pas de `ioctl`)
- particulièrement approprié pour une interaction avec l'utilisateur (programmation *shell*)

```
# echo "1" > /sys/class/gpio/gpio12_pg9/value
```

Attention en C : `fseek` à l'offset 0 en `SEEK_SET` pour plusieurs accès sans refaire `fopen` et `fclose`

Accès au matériel depuis l'espace utilisateur

Au travers de `/dev`

- écriture, lecture ou contrôle (`ioctl`)
- passage au travers du module noyau qui implémente les diverses méthodes
- chaque périphérique est identifié par sa classe (*major number*) et son indice (*minor number*)

```
brw-rw----  1 root     disk      8,   0 Feb 28 06:21 sda
brw-rw----  1 root     disk      8,   1 Feb 28 06:21 sda1
brw-rw----  1 root     disk      8,   2 Feb 28 06:21 sda2
brw-rw----  1 root     disk      8,   3 Feb 28 06:21 sda3
[...]
crw-rw----  1 root     dialout   4,  64 Feb 28 07:21 ttyS0
crw-rw----  1 root     dialout   4,  65 Feb 28 07:21 ttyS1
crw-rw----  1 root     dialout   4,  66 Feb 28 07:21 ttyS2
crw-rw----  1 root     dialout   4,  67 Feb 28 07:21 ttyS3
```

En l'absence d'une entrée (par `udev`) dans `/dev` : `mknod`

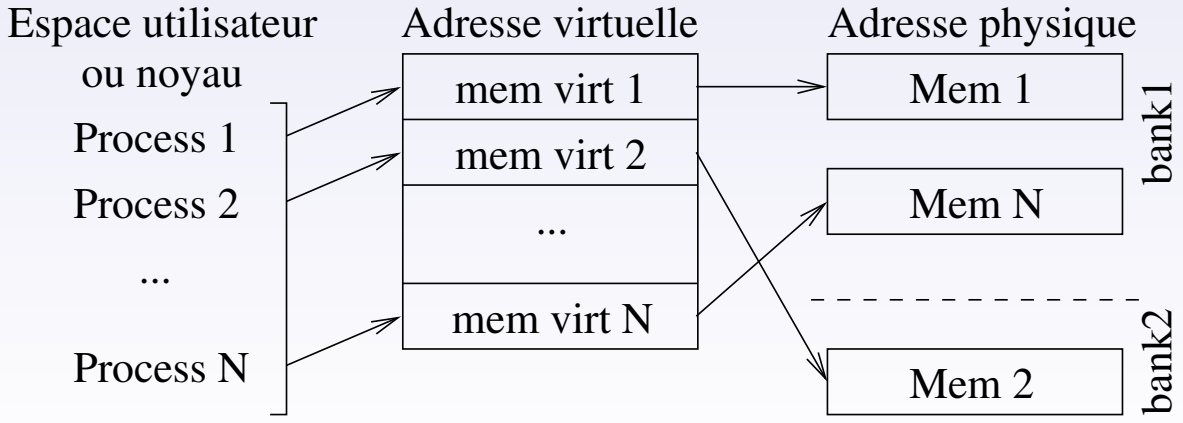
Mémoire virtuelle/mémoire matérielle

Mémoire matérielle

- mémoire matérielle : une adresse sur le bus d'adresse pour identifier un périphérique
- chaque périphérique décode le bus d'adresse pour savoir si le message lui est destiné
- un seul périphérique par adresse physique (sinon, conflit)

Mémoire virtuelle

- chaque processus a sa plage d'adresses
- organisation de la mémoire indépendante des contraintes physiques
- MMU : traduction entre mémoire matérielle et virtuelle



Accès au matériel depuis l'espace utilisateur

Au travers de `/dev/mem`

- avantage : ne pas passer par le noyau (rapide)
- inconvénient : ne pas passer par le noyau (pas de gestion de l'accès aux ressources)

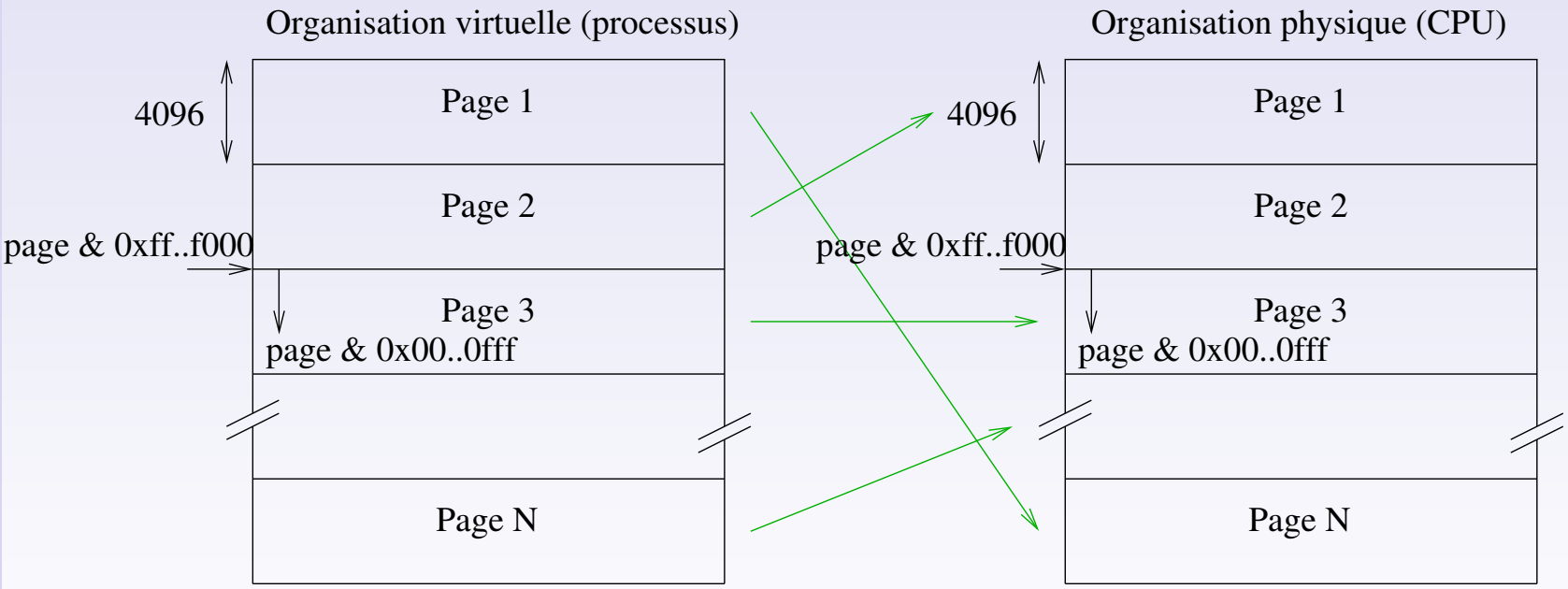
```
#include <fcntl.h>
#include <sys/mman.h>
#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int main(int argc, char **argv) {
 int fd; void *map_base, *virt_addr; unsigned long read_result, writeval;
 off_t target;
 int access_type = 'w'; // Args : { address } [ type [ data ] ]
 target = strtoul(argv[1], 0, 0);
 if(argc > 2) access_type = tolower(argv[2][0]);
 fd = open("/dev/mem", O_RDWR | O_SYNC);
 map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, target & ~MAP_MASK);
 virt_addr = map_base + (target & MAP_MASK);
 switch(access_type) { case 'b': read_result = *((unsigned char *) virt_addr); break;
                       case 'h': read_result = *((unsigned short *)virt_addr); break;
                       case 'w': read_result = *((unsigned long *) virt_addr); break;
 }
 printf("Value at address 0x%X (%p): 0x%X\n", target, virt_addr, read_result);
 if(argc > 3) {
  writeval = strtoul(argv[3], 0, 0);
  switch(access_type) { case 'b': *((unsigned char *) virt_addr) = writeval; break;
                        case 'h': *((unsigned short *)virt_addr) = writeval; break;
                        case 'w': *((unsigned long *) virt_addr) = writeval; break;
               }       }
 munmap(map_base, MAP_SIZE); close(fd); return 0;
}
```

Accès au matériel depuis l'espace utilisateur

Au travers de `/dev/mem`

- avantage : ne pas passer par le noyau (rapide)
- inconvénient : ne pas passer par le noyau (pas de gestion de l'accès aux ressources)



Mémoire virtuelle/mémoire matérielle

Problème identique au niveau du noyau : fonction `ioremap` après avoir réservé une plage d'adresses.

```
#include <linux/io.h>                    // ioremap
#define IO_BASE 0x01c20800
```

et dans la fonction d'initialisation

```
if (request_mem_region(IO_BASE,36*9,"GPIO test")==NULL)
   printk(KERN_ALERT "mem request failed");
jmf_gpio=(u32)ioremap(IO_BASE, 36*9); // 36 octets/port, A-I
writel(0x10,jmf_gpio+36*6+0x04);
```

Pour libérer la ressource :

```
release_mem_region(IO_BASE, 36*9);
```

Adressage en mémoire réelle

```
#define SW_PORTC_IO_BASE 0x01c20800

int sunxi_gpio_init(void) {
    fd = open("/dev/mem", O_RDWR);
    PageSize = sysconf(_SC_PAGESIZE);
    PageMask = (~(PageSize-1));
    addr_start = SW_PORTC_IO_BASE & PageMask;
    addr_offset= SW_PORTC_IO_BASE & ~PageMask;
    gpio_map = (void *)mmap(0, PageSize*2, PROT_READ|PROT_WRITE, MAP_SHARED
    SUNXI_PIO_BASE = (unsigned int)gpio_map;
    SUNXI_PIO_BASE += addr_offset;
}
```

30.2. Port Register List

Module Name	Base Address
PIO	0x01C20800

Register Name	Offset	Description
Pn_CFG0	n*0x24+0x00	Port n Configure Register 0 (n from 0 to 9)
Pn_CFG1	n*0x24+0x04	Port n Configure Register 1 (n from 0 to 9)
Pn_CFG2	n*0x24+0x08	Port n Configure Register 2 (n from 0 to 9)
Pn_CFG3	n*0x24+0x0C	Port n Configure Register 3 (n from 0 to 9)
Pn_DAT	n*0x24+0x10	Port n Data Register (n from 0 to 9)
Pn_DRV0	n*0x24+0x14	Port n Multi-Driving Register 0 (n from 0 to 9)
Pn_DRV1	n*0x24+0x18	Port n Multi-Driving Register 1 (n from 0 to 9)
Pn_PUL0	n*0x24+0x1C	Port n Pull Register 0 (n from 0 to

Adressage en mémoire réelle

Indentification des fonctions de chaque registre (direction, nature du GPIO ...)

30.3.59. PG Data Register

Offset: 0xE8			Register Name: PG_DAT Default Value: 0x0000_0000
Bit	Read/Write	Default	Description
31:12	/	/	/
11:0	R/W	0	PG_DAT

			If the port is configured as input, the corresponding bit is the pin state. If the port is configured as output, the pin state is the same as the corresponding bit. The read bit value is the value setup by software. If the port is configured as functional pin, the undefined value will be read.

30.3.56. PG Configure Register 1

Offset: 0xDC			Register Name: PG_CFG1 Default Value: 0x0000_0000
Bit	Read/Write	Default	Description
31:7	/	/	/
15	/	/	/
14:12	R/W	0	PG11_SELECT 000: Input 001: Output 010: TS1_D7 011: CSI1_D7 100: UART4_RX 101: CSI0_D15 110: Reserved 111: Reserved
11	/	/	/
10:8	R/W	0	PG10_SELECT 000: Input 001: Output 010: TS1_D6 011: CSI1_D6 100: UART4_TX 101: CSI0_D14 110: Reserved 111: Reserved
7	/	/	/
6:4	R/W	0	PG9_SELECT 000: Input 001: Output 010: TS1_D5 011: CSI1_D5 100: UART3_CTS 101: CSI0_D13 110: Reserved 111: Reserved
3	/	/	/
2:0	R/W	0	PG8_SELECT 000: Input 001: Output 010: TS1_D4 011: CSI1_D4 100: UART3_RTS 101: CSI0_D12 110: Reserved 111: Reserved

Base address, espacement des registres ... : A10 User Manual à

`https://dl.linux-sunxi.org/A10/`

Correspondance hard-soft

1. Identifier la broche sur le connecteur GPIO
2. Identifier sa nomenclature carte (Olimex)
3. Identifier sa nomenclature CPU (A13)
4. Trouver les adresses des registres de configuration de la broche dans la datasheet CPU

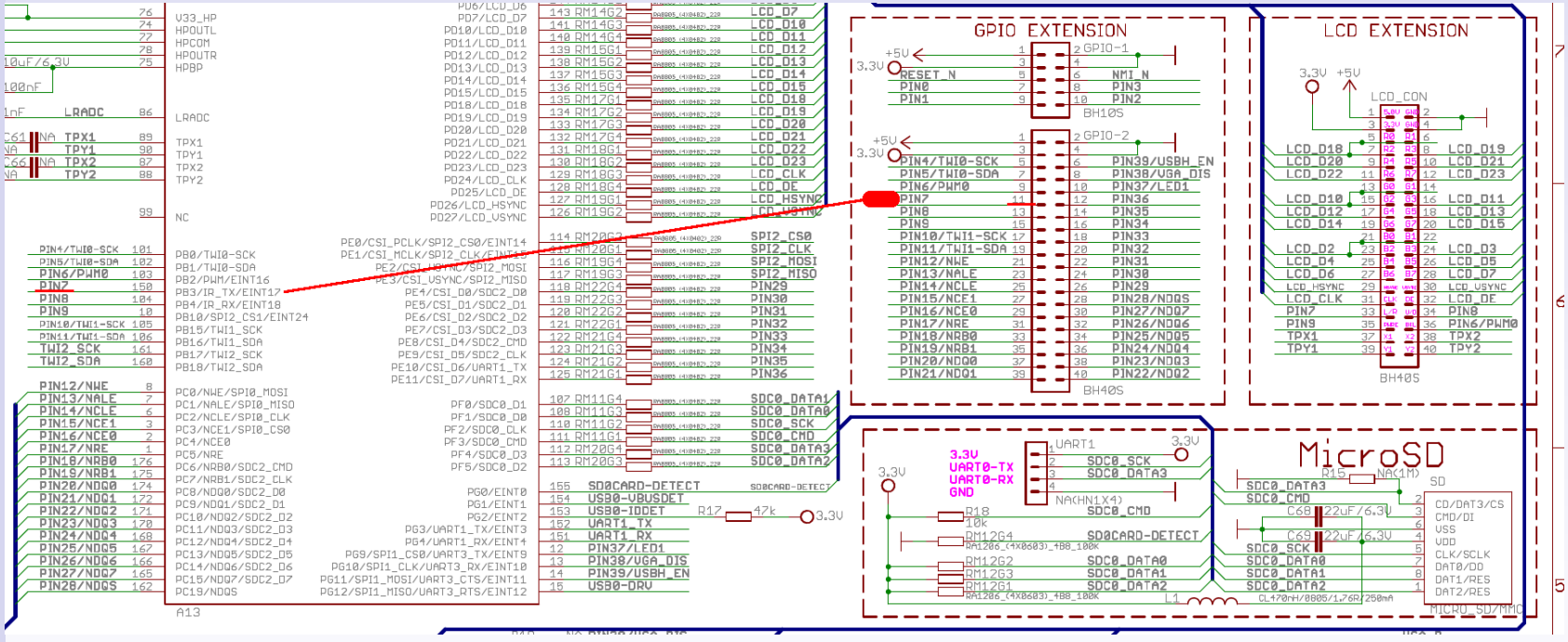


Schéma de la carte Olinuxino A13 :

https://github.com/OLIMEX/OLINUXINO/tree/master/HARDWARE/A13-OLinuXino

Les signaux : prévenir d'un évènement

Équivalent logiciel : les signaux (équivalent aux interruptions)

```
#include <signal.h>

void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);}

int main ( void ) {
    signal (SIGINT, my_handler);
    while (1) {}
}
```

qui donne chaque fois qu'on appuie sur CTRL-C (tuer par `kill -9 PID`)

```
jmfriedt@(none):~$ ./sigint
I got SIGINT, number 2
I got SIGINT, number 2
```

Liste des signaux : `man 7 signal`

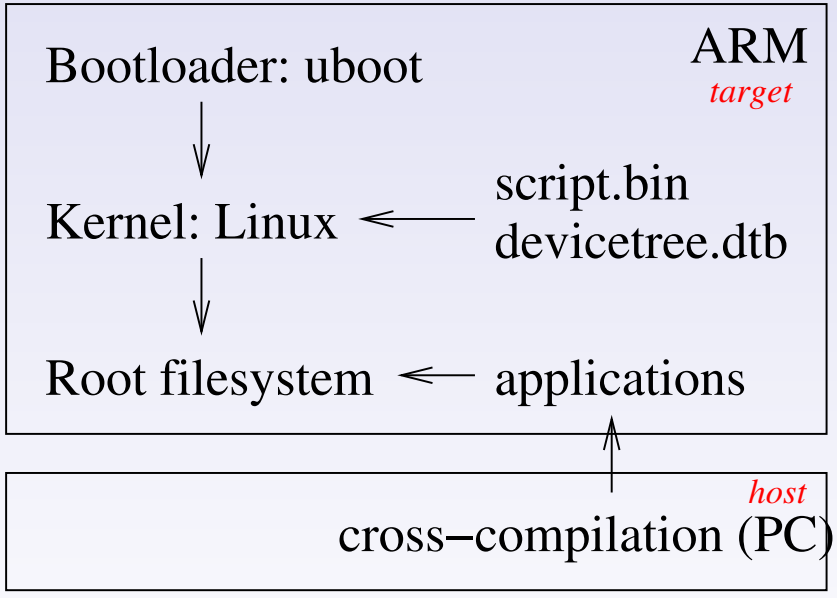
Pourquoi buildroot

Compilation par une chaîne externe binaire incompatible avec notre système

```
$ arm-oe-linux-gnueabi-gcc -mhard-float -c gpio_lib.c
$ arm-oe-linux-gnueabi-gcc -mhard-float -c gpio_sleep.c
$ arm-oe-linux-gnueabi-gcc -mhard-float -o gpio_sleep \
    gpio_sleep.o gpio_lib.o
```

Compilation par la chaîne de buildroot

```
$ arm-linux-gcc -c gpio_sleep.c
$ arm-linux-gcc -c gpio_lib.c
$ arm-linux-gcc -o gpio_sleep \
   gpio_sleep.c gpio_lib.o
```



avec le lien symbolique dans $BR/output/host/usr/bin/

```
.../buildroot-a13-olinuxino-xeno/output/host/usr/bin/arm-linux-gcc
   -> arm-buildroot-linux-uclibcgnueabi-gcc
```

Pourquoi buildroot

À l'exécution, seule la toolchain compatible `buildroot` fournit un exécutable fonctionnel

```
# ./gpio_sleep
^C    <- fonctionne bien
# ./gpio_sleep.openembedded
-sh: ./gpio_sleep.openembedded: not found
```

alors qu'apparemment les fichiers sont deux binaires de nature identique

```
# file gpio_sleep gpio_sleep.openembedded
gpio_sleep:              ELF 32-bit LSB executable, ARM, EABI5 version 1 (S
gpio_sleep.openembedded: ELF 32-bit LSB executable, ARM, EABI5 version 1 (S
```

Pourquoi buildroot

Problème de *bibliothèque chargée dynamiquement* (`.so`)

```
# ldd gpio_sleep.openembedded
checking sub-depends for 'not found'
        libc.so.6 => not found (0x00000000)
        /lib/ld-linux-armhf.so.3 => /lib/ld-linux-armhf.so.3 (0x00000000)
# ldd gpio_sleep
        libc.so.0 => /lib/libc.so.0 (0xb6f14000)
        ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0xb6f6a000)
```

L'erreur `File not found` se comprend par `strace` :

```
# strace -f ./hello
execve("./hello", ["./hello"], [/* 18 vars */]) = -1 ENOENT (No such file o
brk(0x62b000)                                   = 0x62b000
brk(0x62c000)                                   = 0x62c000
write(2, "strace: exec: No such file or di"..., 40strace: exec: No such fil
) = 40
exit_group(1)                                   = ?
+++ exited with 1 +++
```

Pourquoi buildroot

```
# strace ./gpio_sleep
execve("./gpio_sleep", ["./gpio_sleep"], [/* 15 vars */]) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINI
open("/lib/libc.so.0", O_RDONLY)        = 3
fstat(3, {st_mode=S_IFREG|0755, st_size=300808, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINI
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0‘\241\0\0004\0\0\0".
mmap2(NULL, 352256, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f4c0
mmap2(0xb6f4c000, 296428, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 0)
mmap2(0xb6f9c000, 4840, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x4
mmap2(0xb6f9e000, 14240, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_AN
close(3)                                = 0
munmap(0xb6fad000, 4096)                = 0
stat("/lib/ld-uClibc.so.0", {st_mode=S_IFREG|0755, st_size=25304, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINI
set_tls(0xb6fad490, 0xb6fadb30, 0xb6fadb38, 0xb6fad490, 0xb6faff74) = 0
mprotect(0xb6f9c000, 4096, PROT_READ)   = 0
mprotect(0xb6faf000, 4096, PROT_READ)   = 0
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or TCGETS, {B
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or TCGETS, {B
open("/dev/mem", O_RDWR)                = 3
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0x1c20000) = 0xb6fab
close(3)                                = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
nanosleep({1, 0}, 0xbe9bfbd4)           = 0
```

L'OS n'est pas toujours bien

- un OS doit booter : prend du temps et donc de l'énergie
- un OS nécessite de maîtriser des API
- un OS nécessite de la mémoire et de la puissance de calcul

$\Rightarrow$ bien peser les avantages (bibliothèques, contributions externes, stabilité des outils) et les contres avant de faire un choix.

Mise en pratique

Accès au matériel depuis l'espace utilisateur sur carte Olinuxino A13-micro [2]

- cross-compilation dans l'environnement issu de buildroot [3]
- programme C, problème de la MMU

2. https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware
3. http://jmfriedt.free.fr/A13_v1.pdf