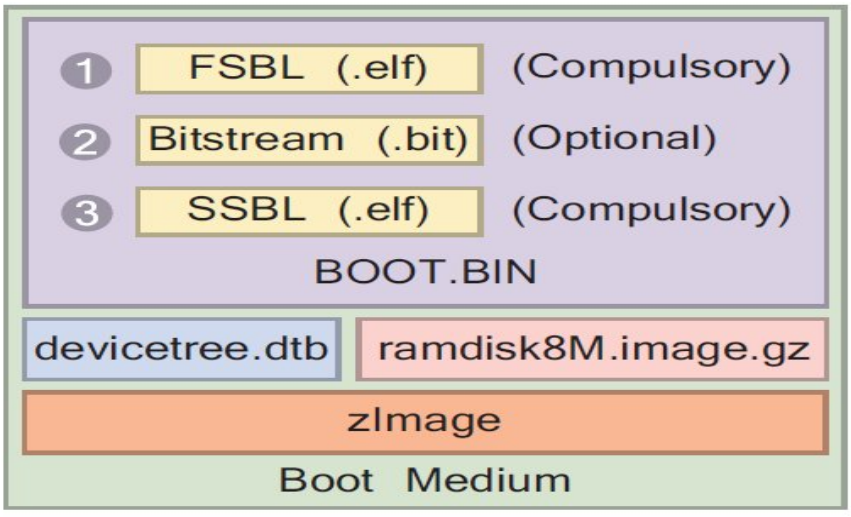
# LINUX Digilent sur la carte ZYBO

La société Digilent fournit une distribution LINUX que l'on peut implémenter sur les cartes ZYNQ ZEDBOARD ou ZYNQ. Le noyau LINUX fourni par XILINX a été patché par la société Digilent afin de prendre en compte les éléments matériels de la carte ZYBO.
Un environnement de cross compilation type UBUNTU est nécessaire afin de compiler le chargeur de noyau U-BOOT le noyau LINUX et d'implanter la distribution LINUX.
La distribution LINUX de la carte ZYBO sera chargée dans la mémoire RAM (ramdisk)

## 1) La Phase de démarrage

Pour pourvoir démarrer la distribution LINUX, plusieurs fichiers sont nécessaires sur la SDCARD :

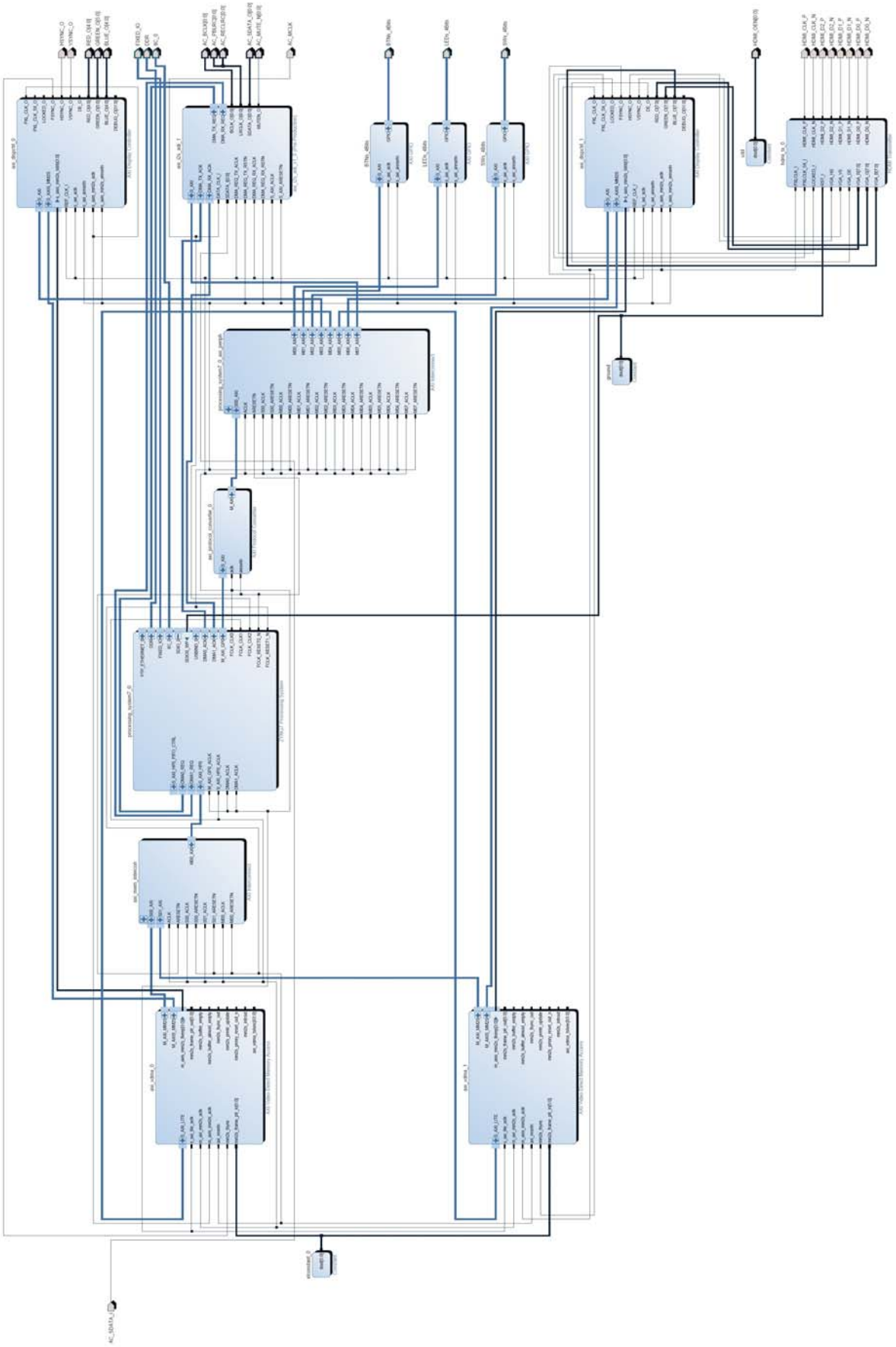
- ✓ Le fichier boot.bin qui intègre le premier chargeur de la société XILINX, la configuration matérielle du FPGA et le programme permettant de lancer le chargeur de boot U-BOOT
- ✓ Le fichier u-boot.elf qui prend en compte les informations de la carte et prépare le système de traitement ARM pour booter LINUX. Une fois la configuration terminée u-boot laissera la main au noyau LINUX pour l'ARM
- ✓ Le fichier ARM Linux (uImage) qui contient le noyau LINUX pour la carte ZYBO.
- ✓ ARM RAMDisk (uramdisk.image.gz) le RAMDisk qui est monté par le noyau et qui contient le système de fichier racine.



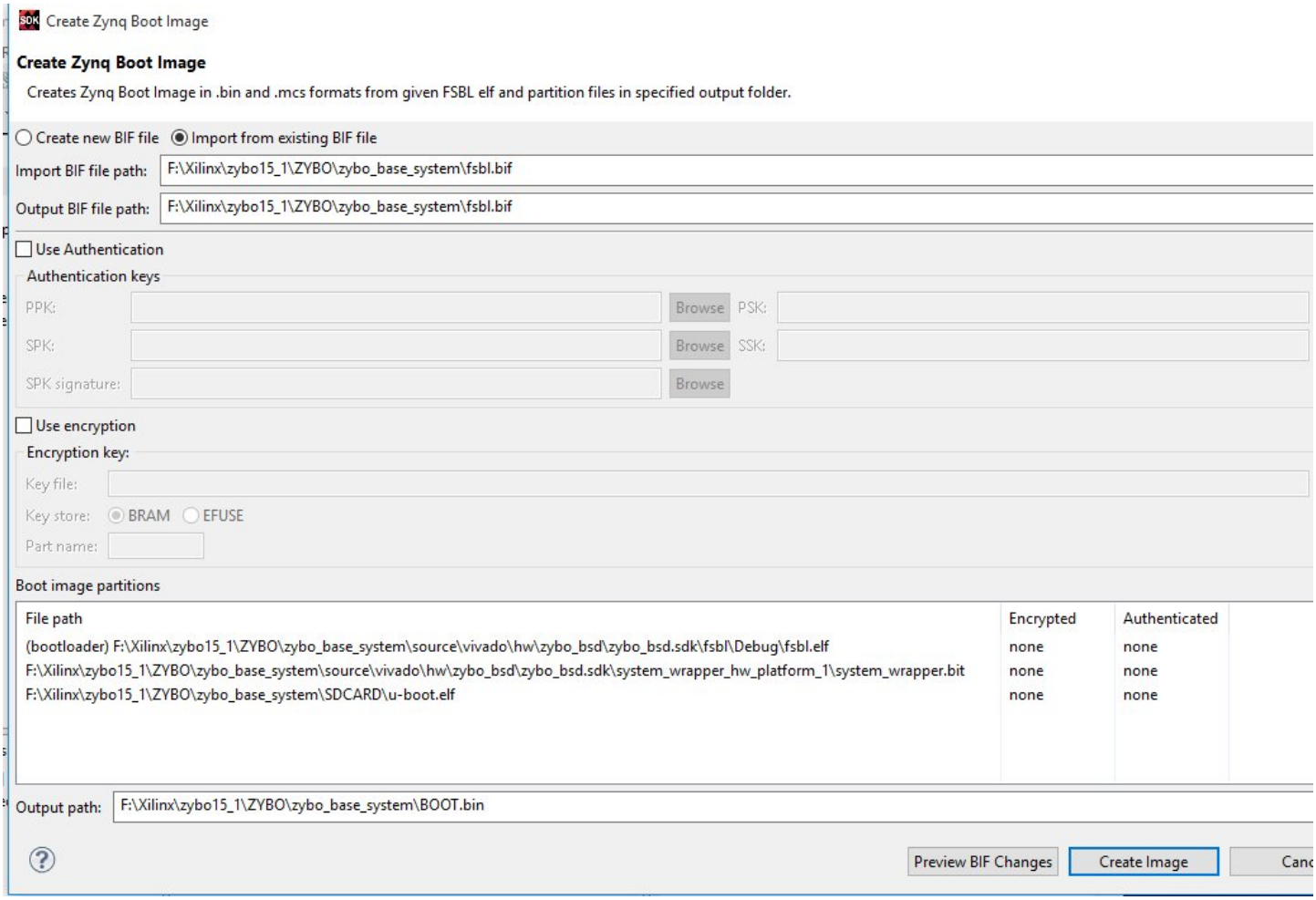
## 2) Configuration du matériel

La société Digilent met à notre disposition un package qui nous permettra de générer le fichier boot.bin. Ce package est nommé Zybo base design system.

Décompressez le fichier zybo_base_system.zip.
Ouvrir le projet sous Vivado, le board design est donné page suivante.
Générez par la suite le fichier bitstream et lancez le logiciel SDK



Sous SDK créez un fichier fsbl puis lancez la création d'un zynq boot image.
Les fichiers doivent être inclus dans l'ordre suivant :




Avant de pouvoir implémenter la distribution LINUX, il nous faut disposer d'un cross compilateur du type ubuntu 64 bits (une version 14.04 LTS suffit). Cette distribution ubuntu peut être lancée d'une machine virtuelle vmware par exemple. Sur Ubuntu on doit installer la suite logicielle vivado design et disposer d'une connexion internet

### 3) Compilation du chargeur de boot

La première étape consiste à installer les librairies qui permettent de lancer des exécutables 32 bits sur un OS 64 bits.

Se mettre en mode root

sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0

## Implantation d'un périphérique MLI
## Sur la distribution LINUX LINARO

Comment intégrer un périphérique sur la carte ZYBO et dans l'environnement logiciel de la distribution LINUX LINARO (Login : root pass : xilinx)?
Pour répondre à cette problématique il nous faut d'abord disposer d'une distribution LINUX LINARO qui puisse s'implémenter sur la carte ZYBO.

### La distribution LINUX LINARO

Le principal inconvénient d'utiliser un système de fichier du type ramdisk est l'impossibilité de modifier ce fichier une fois implantée sur la SDCARD. La distribution LINARO sera implémentée directement sur une partition EXT4 d'une SDCARD.

Récupérez une SDCARD qui dispose d'une partition FAT de 100 MO et d'une partition LINUX du type EXT4

Téléchargez la version linaro-precise-ubuntu-desktop-20121124-560.tar.gz
Sur VMWARE et UBUNTU décompressez l'archive dans un dossier
sudo tar -xvf linaro-precise-ubuntu-desktop-20121124-560.tar.gz -C /home/phil/tmp/linaro

cd /tmp/linaro

sudo mkdir -p /tmp/sd_ext4

cd /binary/boot/filesystem.dir

Montez une partition de la SDCARD sur l'image de la distribution
sudo mount /dev/sdb2 /tmp/sd_ext4

Copiez le système de fichier LINARO sur la partition ext4 de la SDCARD

sudo rsync –a ./ /tmp/sd_ext4

Modification de uboot

Dans le répertoire u-boot-digilent-Dev/include/configs, modifiez le fichier zynq_zybo.h avec les lignes suivantes de façon à lancer le système de fichier sur la SDCARD :

```
"sdboot=if mmcinfo; then " \
                    "run uenvboot; " \
                    "echo Copying Linux from SD to RAM... && " \
                    "fatload mmc 0 0x3000000 ${kernel_image} && " \
                    "fatload mmc 0 0x2A00000 ${devicetree_image} && " \
                    "bootm 0x3000000 - 0x2A00000; " \
            "fi\0" \
```

Le noyau sera kernel_image
Le devicetree sera devicetree_image
u-boot chargera le noyau et le devicetree grâce à la commande bootm 0x3000000 - 0x2A00000
PATH=$PATH:/usr/src/u-boot-Digilent-Dev/tools/
On aurait pu décomposer sous cette forme :

```
u-boot> fatload mmc 0 0x3000000 uImage // charge le noyau
u-boot> fatload mmc 0 0x2A00000 devicetree.dtb // charge le device tree
//u-boot> fatload mmc 0 0x2000000 uramdisk.image.gz // si il y a un RAMDISK
u-boot> bootm 0x3000000 0x2A00000 // boot sur le device tree et le noyau
```

compilez
make CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_zybo_config
make CROSS_COMPILE=arm-xilinx-linux-gnueabi-

Récupérez le travail réalisé lors du projet zybot
https://reference.digilentinc.com/zybo:zybo:zybot

Décompressez les fichiers pour le DTS dans le répertoire zybo_robot_dts

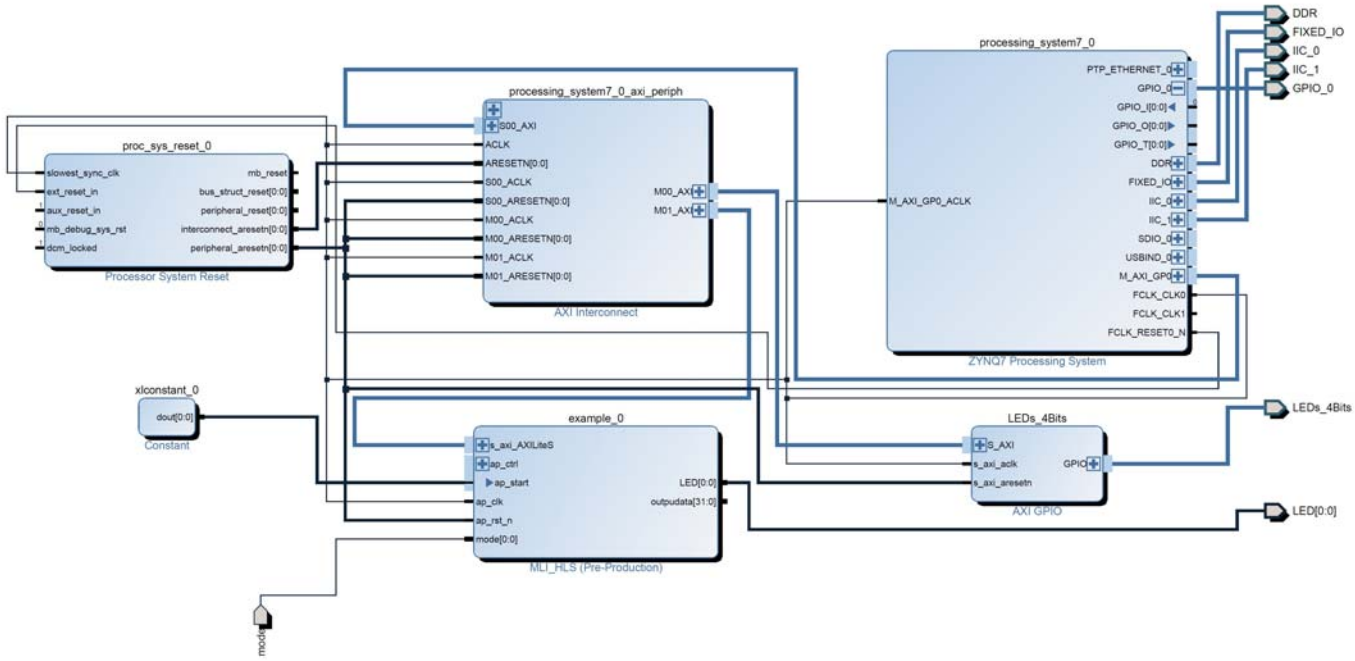
Le devicetree et son fichier system.dts permet de fournir au noyau les arguments suivants permettant le lancement du système de fichier par le noyau

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=1";
     };
```

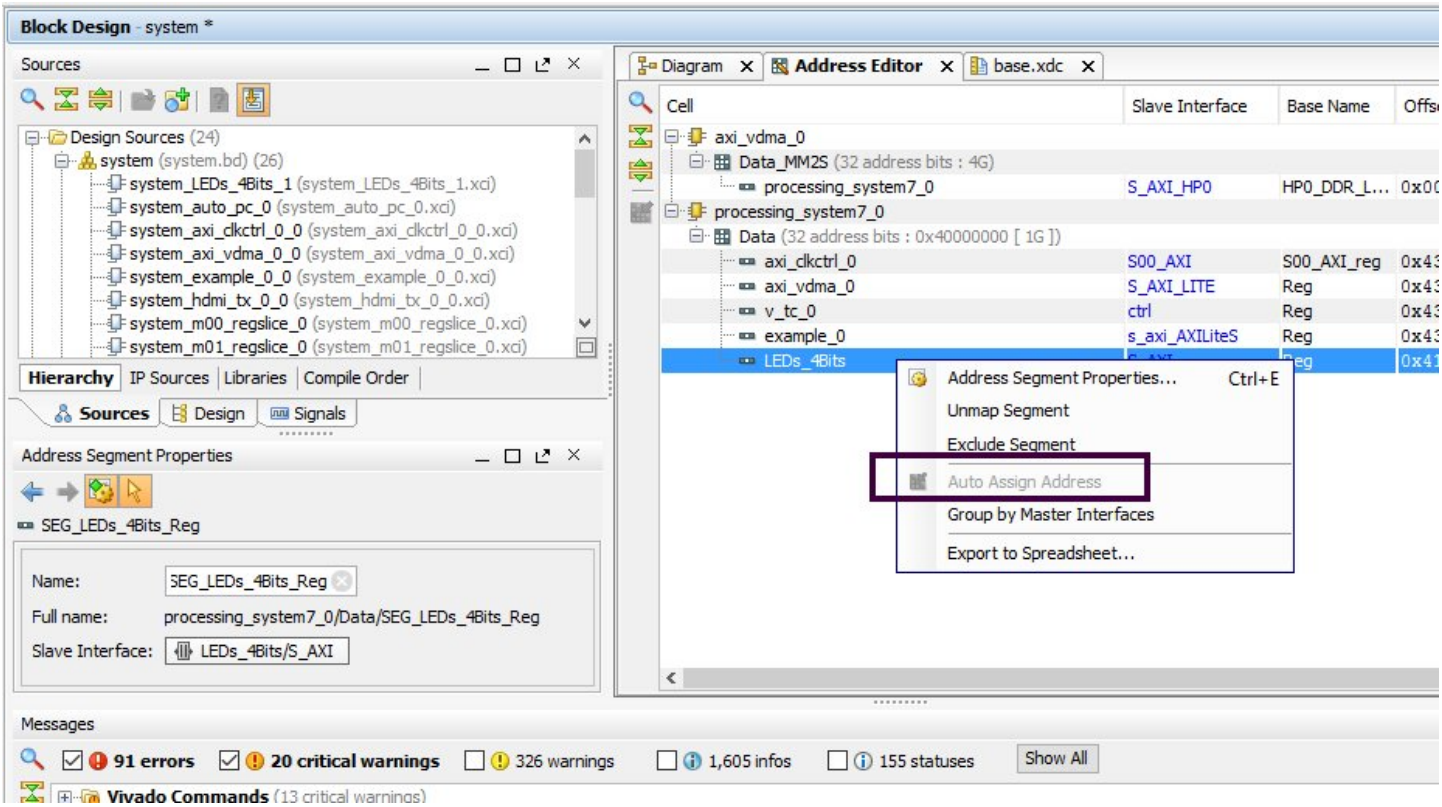
Téléchargez le projet pour vivado puis décompressez les fichiers dans le répertoire zybo_bsd_hdmi

## Implantation matérielle sous Vivado

Dans le répertoire zybo_bsd_hdmi, ouvrir le fichier projet sous vivado et modifiez le schéma de la manière suivante ;




Pour réattribuer les adresses sur le bus AXI, cliquez sur adress editor et cliquez droit sur le périphérique qui n'a pas d'adresse :




Cliquez sur auto assign adress

## Modification du device tree

Le principe du Device Tree est de séparer la description matérielle de notre environnement des sources du noyau. Le noyau se sert du device tree pour mettre en place les pilotes nécessaires pour chaque élément décrit dans le device tree. Le device tree a une structure en arbre avec des branches qui modélisent la hiérarchie des périphériques dans le système
Chaque branche peut avoir un nombre de propriété décrivant différentes propriétés du périphérique, adresses, interruptions, horloges. Le device tree est passé au noyau durant la période de bootage

Un périphérique dans le device tree se déclare de la façon suivante :

La première ligne décrit le nom du périphérique qui sera enregistré par le noyau dans le répertoire proc
La deuxième ligne permet d'identifier le pilote du noyau qui sera associé à ce périphérique (compatible).
La troisième ligne fournit l'adresse du périphérique et la taille de son espace adressable (Reg)
La quatrième ligne et cinquième ligne fournit la gestion de l'interruption (si elle existe) notamment le handler, le numéro ….
Les autres paramètres dépendent du type de périphérique

```
(Name) : (Generic Name)@(Base Address) {
compatible: "(compatibility string)";
reg: < (base address) (size) >;
interrupt-parents: < (interrupt controller phandle) >;
interrupts: < … >;
(param1): "(string value)";
(param2): < (number value, decimal or hexical) >;
…
};
```

Exemple :

```
gpiops: gpio@e000a000 {
compatible = "xlnx,ps7-gpio-1.00.a";
#gpio-cells = <2>;
reg = <0xe000a000 0x1000>;
interrupts = <0x0 0x14 0x0>;
interrupt-parent = <&gic>;
gpio-controller;
};
```

La propriété compatible indique que `ps7-gpio-1.00.a sera le pilote utilisé pour ce type de périphérique. Lors de la création du pilote on associera un nom à celui-ci.`
`Exemple pour le VDMA on aura dans le fichier vdma.c la définition suivante :`

```
static const struct of_device_id xilinx_vdma_of_ids[] = {
	{ .compatible = "xlnx,axi-vdma-1.00.a",},
	{}
};
```

axi-vdma-1.00.a sera le nom du pilote

Si ton périphérique a des fonctionnalités en terme d'interruption tu dois spécifier le numéro d'interruption dans la propriété interrupt (0x14) ainsi que le nom de la fonction d'interruption dans la propriété phandler .Le nom du contrôleur d'interruption sera spécifié dans la propriété interrupt-parent ici gic

Un fichier dts décrira le device tree en utilisant la syntaxe précédente. Un fihcier dts peut utiliser des fichiers dtsi pour décrire des sous-ensembles.
Pour le zynq la logique programmable PL et le PS peuvent se décrire dans des fichiers dtsi
Exemple pour le fichier devicetree.dts :

```
dts-v1/;
/include/ "zynq-7000.dtsi"
/include/ "pl1.dtsi"
/ {
        cpus {
                cpu@0 {
                        operating-points = <650000 1000000 325000 1000000>;
                };
        };
        chosen {
                bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait
devtmpfs.mount=1";
        };
        aliases {
                ethernet0 = &gem0;
                serial0 = &uart1;
        };
        memory {
                device_type = "memory";
                reg = <0x0 0x20000000>;
        };
};
&gem0 {
        //local-mac-address = [00 0a 37 00 00 00];
  phy-handle = <&phy0>;
        phy-mode = "rgmii-id";
        status = "okay";
        xlnx,ptp-enet-clock = <0x6750918>;
        ps7_ethernet_0_mdio: mdio {
                #address-cells = <1>;
                #size-cells = <0>;
    phy0: phy@1 {
                                        compatible = "realtek,RTL8211E";
                                        device_type = "ethernet-phy";
                                        reg = <1>;
                };
        };
};
&gpio0 {
        emio-gpio-width = <1>;
        gpio-mask-high = <0x0>;
        gpio-mask-low = <0x5600>;
};
```

```
&i2c0 {
	clock-frequency = <400000>;
	status = "okay";
};
&i2c1 {
	clock-frequency = <400000>;
	status = "okay";
//   edid: edid@50 {
//                                                  compatible = "at24,24c256";
//                                                  reg = <0x50>;
//                                          };
};
&intc {
	num_cpus = <2>;
	num_interrupts = <96>;
};
&qspi {
	is-dual = <0>;
	num-cs = <1>;
	status = "okay";
};
&sdhci0 {
	clock-frequency = <50000000>;
	status = "okay";
};
&uart1 {
	current-speed = <115200>;
	device_type = "serial";
	port-number = <0>;
	status = "okay";
};
&usb0 {
	dr_mode = "host";
	phy_type = "ulpi";
	status = "okay";
	usb-reset = <&gpio0 46 0>;
};
&clkc {
	fclk-enable = <0x3>;
	ps-clk-frequency = <50000000>;
};
```

Pour le fichier pl1.dtsi

```
/ {
	amba_pl: amba_pl {
		#address-cells = <1>;
		#size-cells = <1>;
		compatible = "simple-bus";
		ranges ;
		LEDs_4Bits: gpio@41210000 {
			#gpio-cells = <2>;
			compatible = "xlnx,xps-gpio-1.00.a";
			gpio-controller ;
			reg = <0x41210000 0x10000>;
```

```
                xlnx,all-inputs = <0x0>;
                xlnx,all-inputs-2 = <0x0>;
                xlnx,all-outputs = <0x1>;
                xlnx,all-outputs-2 = <0x0>;
                xlnx,dout-default = <0x00000000>;
                xlnx,dout-default-2 = <0x00000000>;
                xlnx,gpio-width = <0x4>;
                xlnx,gpio2-width = <0x20>;
                xlnx,interrupt-present = <0x0>;
                xlnx,is-dual = <0x0>;
                xlnx,tri-default = <0xFFFFFFFF>;
                xlnx,tri-default-2 = <0xFFFFFFFF>;}
mymli {

                compatible = "dglnt,mymli";

                reg = <0x43c00000 0x10000>;

           };

     };
           };
```

Le fichier pl1.dtsi est référencé dans le fichier system.dts par (/include/"pl1.dtsi")

On a rajouté notre périphérique à la fin avec son adresse `0x43c00000` et sa taille mémoire `0x10000`

Le compilateur de device tree est utilisé pour obtenir un fichier device tree .dtb à partir du fichier dts ou dtsi. Le fichier dtb sera exploité par le chargeur de noyau ainsi que le noyau.

Dans votre répertoire de travail efface le fichier system.dtb et lancez la commande suivante :

dtc -I dts -O dtb -o system.dtb system.dts
Copiez les fichiers system.dtb et system.dts

Dans le répertoire /usr/src/Linux-Digilent-Dev

Lancez la commande suivante :
sudo ./scripts/dtc/dtc -I dts -O dtb -o ./devicetree.dtb arch/arm/boot/dts/zynq-zybo.dts

Une fois le device tree obtenu il nous reste à associer sur la partition FAT de la SDCARD les fichiers suivants :

- Le boot.bin obtenu sur SDK qui contient le chargeur fsbl, le bitstream, u-boot.elf
- Le fichier devicetree.dtb
- Le noyau de chez digilent que l'on aura compilé pour la carte zybo.

Une fois le système lancé on obtiendra nos périphériques grâce au chemin suivant :

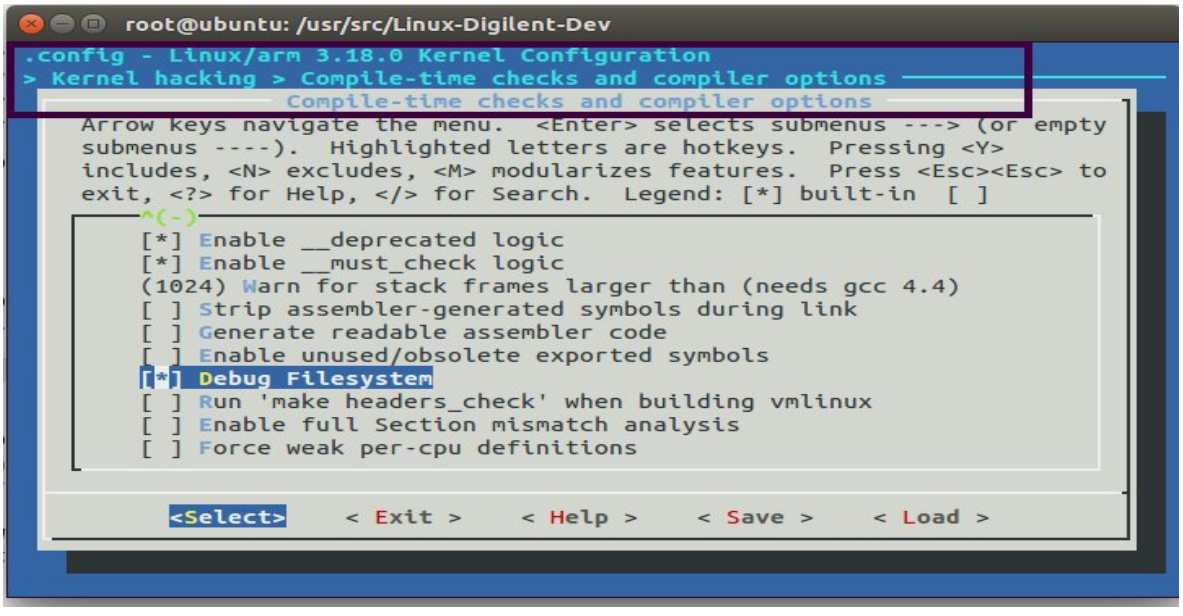
```
root@linaro-ubuntu-desktop:~# cd /proc
root@linaro-ubuntu-desktop:/proc# ls
1     1234  1629  380  623  97           driver        locks         swaps
10    13    1658  394  624  asound       execdomains   meminfo       sys
1030  1335  1668  395  630  buddyinfo    fb            misc          sysvipc
1037  135   1677  396  644  bus          filesystems   modules       thread-self
1038  14    17    4    645  cgroups      fs            mounts        timer_list
1080  15    1740  5    646  cmdline      interrupts    mtd           timer_stats
11    1557  1741  502  649  config.gz    iomem         net           tty
1144  1596  1884  53   7    consoles     ioports       pagetypeinfo  uptime
1147  16    1885  542  713  cpu          irq           partitions    version
1182  1602  19    564  719  cpuinfo      kallsyms      scsi          vmallocinfo
12    1603  1929  595  8    crypto       kmsg          self          vmstat
1206  1613  2     599  9    device-tree  kpagecount    slabinfo      zoneinfo
1225  1614  3     6    92   devices      kpageflags    softirqs
1226  1617  379   610  94   diskstats    loadavg       stat
root@linaro-ubuntu-desktop:/proc# cd device-tree
root@linaro-ubuntu-desktop:/proc/device-tree# ls
#address-cells  aliases  amba_pl  compatible  fixedregulator@0  name
#size-cells     amba     chosen   cpus        memory            pmu
root@linaro-ubuntu-desktop:/proc/device-tree# cd amba_pl
root@linaro-ubuntu-desktop:/proc/device-tree/amba_pl# ls
#address-cells     compatible         gpio@41200000  motor   servo
#size-cells        digilent_encoder   gpio@41210000  name    v-tc@43C20000
clock@0x43C30000  dma@43000000       gpio@41220000  ranges  xilinx_drm
root@linaro-ubuntu-desktop:/proc/device-tree/amba_pl#
```

```
hexdump -C '/proc/device-tree/amba_pl/mymli/compatible'
```

## L'écriture et l'utilisation des pilotes MLI

On peut utiliser le système de fichier debugfs pour communiquer de l'espace utilisateur vers l'espace noyau et notre périphérique. Il faut activer dans le noyau la fonction de debuggage par debugfs. Lancez sur le cross compilateur la configuration du noyau :

En mode root dans le répertoire /usr/src/Linux-digilent
source /opt/Xilinx/Vivado/2014.4/settings64.sh
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- xilinx_zynq_defconfig
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- menuconfig



Recompilez le noyau et copiez le nouveau noyau Uimage sur la partition FAT de la SDCARD :
A la fin du lancement du noyau le système de fichier debugfs existe :

```
root@linaro-ubuntu-desktop:~# cd /sys/kernel/debug
root@linaro-ubuntu-desktop:/sys/kernel/debug# ls
asoc  dma_buf              gpio  memblock  regmap       suspend_stats
bdi   dri                  hid   mmc0      regulator    usb
clk   fault_around_bytes   iio   ras       sleep_time   wakeup_sources
root@linaro-ubuntu-desktop:/sys/kernel/debug#
```

Sur le cross compilateur créer un répertoire qui nous permettre de développer un module. Ce module créera un lien vers le système de fichier debugfs afin de communiquer dans l'espace utilisateur avec la périphérique MLI :
Créez un fichier texte Makefile pour notre module mli_module.ko :

```
obj-m += mli_module.o

all:

	make -C /lib/modules/3.18.0-xilinx-46110-gd627f5d-dirty/build M=$(PWD) modules

clean:

	make -C /lib/modules/3.18.0-xilinx-46110-gd627f5d-dirty/build M=$(PWD) clean
```

Créez un fichier source qui contiendra le code source pour le module:

```
#include	<linux/module.h>
#include	<linux/kernel.h>
#include	<linux/init.h>
#include	<linux/interrupt.h>
#include	<linux/irq.h>
#include	<asm/io.h>
#include	<linux/fs.h>		// Para fops
#include	<linux/signal.h>		// Para señales kernel-userspace
#include	<asm/siginfo.h>		// siginfo
#include	<linux/rcupdate.h>		// rcu_read_lock
#include	<linux/sched.h>	// find_task_by_pid_type
#include	<linux/debugfs.h>
#include	<linux/uaccess.h>		// Para 'copy_from_user' and 'copy_to_user'
MODULE_LICENSE("GPL");
```

```
#define SIG_TEST 44  // Definición de una señal propia (rango de 33 a 64 para real-time signals)

#define IRQ_NUM      61
#define     mli_BASE   0x43C00000
#define     mli_SIZE   0x0000FFFF
#define     TCSR0           0x00000000
#define     TLR0       0x00000004
#define     TCR0       0x00000008
#define     T0INT           0x00000100
#define     mli_TCSR0  0x43C00010 // Offset de $10 pour le périph
#define     mli_TLR0   0x43C00014
#define     mli_TCR0   0x43C00018
unsigned long *pmli_TCSR0;     // Puntero al control and status register del mli 0
unsigned long *pmli_TLR0;      // Puntero al load register del mli 0
unsigned long *pmli_TCR0;      // Puntero al counter register del mli 0

#define DEVICE_NAME "mli"  // Nombre del dispositivo
#define SUCCESS 0      // Valor de retorno Success


struct dentry *file_debugfs;    // Archivo para comunicación kernel-userspace
static int app_pid = 0;// PID de la aplicación en el userspace

// Función write (llamada cuando write() es usado en el espacio de usuario)
ssize_t syscall_write(struct file *flip, const char *buf, size_t length, loff_t *offset)
{
char mymli_phrase[16];
    u32 mymli_value;

    if (length < 9) {
       if (copy_from_user(mymli_phrase, buf, length))
          return -EFAULT;

      //mymli_phrase[count] = '\0';
   }


printk("syscall_write.\n");
//printk("Value received: %u.\n", value);
```

```
printk(" ecriture %x, %x, %x, %x, %x, %x, %x, %x,
\n",mymli_phrase[0],mymli_phrase[1],mymli_phrase[2],mymli_phrase[3],mymli_phrase[4],mymli
_phrase[5],mymli_phrase[6],mymli_phrase[7]);
    //wmb();
        iowrite32(0x000000FE, pmli_TCSR0);

        iowrite32((mymli_phrase[0]<<24)+(mymli_phrase[1]<<16)+(mymli_phrase[2]<<8)+(mymli_ph
rase[3]), pmli_TLR0);
        iowrite32((mymli_phrase[4]<<24)+(mymli_phrase[5]<<16)+(mymli_phrase[6]<<8)+(mymli_ph
rase[7]), pmli_TCR0);
    return length;
 }




// Función read (llamada cuando read() es usado en el espacio de usuario)
ssize_t syscall_read(struct file *flip, char *buf, size_t length, loff_t *offset)
{
unsigned int value;
printk("syscall_read.\n");



        u32 mymli_value;
    mymli_value =(u32) ioread32(pmli_TCSR0);
    printk("la valeur est 0x%08lx \n",mymli_value);
        mymli_value =(u32) ioread32(pmli_TLR0);
printk("la valeur est 0x%08lx \n",mymli_value);
mymli_value =(u32) ioread32(pmli_TCR0);
printk("la valeur est 0x%08lx \n",mymli_value);
if (copy_to_user(buf, &mymli_value, sizeof(unsigned int)) != 0) return -EFAULT; else return 0;
}

// Archivo de operaciones
struct file_operations syscall_fops = {
.read = syscall_read,       // read()
.write = syscall_write,     // write()
};



// Módulo init
static int __init mod_init(void)
```

```
{
printk(KERN_ERR "Init syscall module V1.2 \n");

// Se registran las interrupciones del mli

pmli_TCSR0 = ioremap_nocache(mli_TCSR0,0x4); // Mapeo del control and status register
pmli_TLR0 = ioremap_nocache(mli_TLR0,0x4);  // Mapeo del load register del mli 0
pmli_TCR0 = ioremap_nocache(mli_TCR0,0x4);  // Mapeo del count register del mli 0

//iowrite32(0x000000B0, pmli_TCSR0);     // Carga del TLR
//iowrite32(0x000000D0, pmli_TCSR0);     // Generate mode, downcounter, reload generate

// value, no load, enable IRQ, enable mli

// Se utiliza un archivo de configuración de lectura y escritura para el paso de información
file_debugfs = debugfs_create_file("mli_conf", 0644, NULL, NULL, &syscall_fops);
if (file_debugfs ==0) printk(KERN_ERR "erreur creation mli_conf. \n");

return SUCCESS;
}

// exit module
static void __exit mod_exit(void)
{
debugfs_remove(file_debugfs);
iounmap(pmli_TCSR0);    // Se "des-mapean" los registros del mli 0
iounmap(pmli_TLR0);
iounmap(pmli_TCR0);
//free_irq(IRQ_NUM, NULL); // Se desregistran las interrupciones
printk(KERN_ERR "Exit syscall Module. \n");
}

module_init(mod_init);
module_exit(mod_exit);
MODULE_AUTHOR ("phil");
MODULE_DESCRIPTION("Testdriver for the Xilinx mymli.");
MODULE_LICENSE("GPL v2");
MODULE_ALIAS("custom:syscall");
```

Lancez la compilation du module par le cross compilateur en mode root
source /opt/Xilinx/Vivado/2014.4/settings64.sh
export PATH=/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/:$PATH
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
export ARCH=arm
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-

puis copiez le fichier module mli_module.ko sur la carte zybo.

Pour insérez le nouveau module :
insmod mli_module.ko

```
root@linaro-ubuntu-desktop:/sys/kernel/debug# cd /home/mli/module
root@linaro-ubuntu-desktop:/home/mli/module# ls
Makefile         mli_module.c   mli_module.mod.c  mli_user.c
Makefile~        mli_module.c~  mli_module.mod.o  modules.order
Module.symvers   mli_module.ko  mli_module.o
root@linaro-ubuntu-desktop:/home/mli/module# insmod mli_module.ko
Init syscall module V1.2
root@linaro-ubuntu-desktop:/home/mli/module# lsmod
Module                  Size  Used by
mli_module              1582  0
root@linaro-ubuntu-desktop:/home/mli/module#
```

On peut vérifier si le module a bien rajouté un système de fichier mli_conf dans le debugfs :

```
root@linaro-ubuntu-desktop:/home/mli/module# insmod mli_module.ko
Init syscall module V1.2
root@linaro-ubuntu-desktop:/home/mli/module# lsmod
Module                  Size  Used by
mli_module              1582  0
root@linaro-ubuntu-desktop:/home/mli/module# mount
/dev/root on / type ext4 (rw)
none on /proc type proc (rw,noexec,nosuid,nodev)
none on /sys type sysfs (rw,noexec,nosuid,nodev)
devtmpfs on /dev type devtmpfs (rw,mode=0755)
none on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
none on /sys/kernel/debug type debugfs (rw)
none on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
root@linaro-ubuntu-desktop:/home/mli/module# cd /sys/kernel/debug
root@linaro-ubuntu-desktop:/sys/kernel/debug# ls
asoc      dri                  iio       ras          suspend_stats
bdi       fault_around_bytes   memblock  regmap       usb
clk       gpio                 mli_conf  regulator    wakeup_sources
dma_buf   hid                  mmc0      sleep_time
root@linaro-ubuntu-desktop:/sys/kernel/debug#
```

Il nous reste à développer notre applicatif utilisateur pour communiquer avec le périphérique :
Créez un nouveau répertoire puis un nouveau fichier source mli_user1.c

Complétez le fichier source :

```
#include <stdio.h>
#include <fcntl.h>

int main ( int argc, char **argv )
{
unsigned long valeur;
int configFile ;
char tab[8]={0x02,0xFA,0xF0,0x80,0x05,0xF5,0xE1,0x00};
char buf[4]={0x02};
configFile = open("/sys/kernel/debug/mli_conf", O_RDWR);
if(configFile < 0) {printf("impossible open \n");return -1;}
if (write(configFile, tab, 8) < 0) {printf("impossible write \n");return -1;}
if(read(configFile, buf, 4) < 0) {printf("impossible read \n");return -1;}
valeur=(buf[3]<<24)+(buf[2]<<16)+(buf[1]<<8)+buf[0];
printf("la valeur est de 0x%08lx \n",valeur);
close(configFile);
return 0;
}
```

Compilez :

gcc -o mli_user1 mli_user1.c

Lancez notre applicatif dans le domaine utilisateur :
./mli_user1

Vérifiez

```
root@linaro-ubuntu-desktop:/sys/kernel/debug# gcc -o mli_user1 mli_user1.c^C
root@linaro-ubuntu-desktop:/sys/kernel/debug# cd /home/mli/user
root@linaro-ubuntu-desktop:/home/mli/user# ./mli_user1
la valeur est de 0x05f5e100
root@linaro-ubuntu-desktop:/home/mli/user#
```

Lancez la commande dmesg

```
syscall_write.
 ecriture 2, fa, f0, 80, 5, f5, e1, 0,
syscall_read.
la valeur est 0x000000fe
la valeur est 0x02faf080
la valeur est 0x05f5e100
root@linaro-ubuntu-desktop:/home/mli/user#
```