

# Trabajo Final ISO II: Sistema Implementado en placa ZYB0

Autor: Gonzalo Lavigna  
Agosto 2019

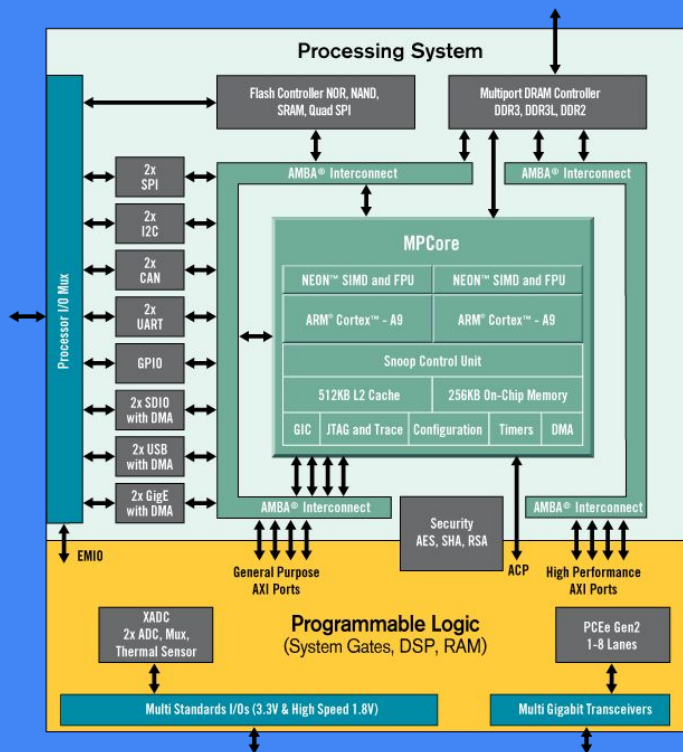
# Kit de desarrollo ZYBO



Figure 1. ZYBO Zynq-7000 development board.

- ZYNQ XC7Z010-1CLG400C
- 512MB x32 DDR3 w/ 1050Mbps bandwidth
- Dual-role (Source/Sink) HDMI port
- 16-bits per pixel VGA source port
- Trimode (1Gbit/100Mbit/10Mbit) Ethernet PHY
- MicroSD slot (supports Linux file system)
- OTG USB 2.0 PHY (supports host and device)
- External EEPROM (programmed with 48-bit globally unique EUI-48/64™ compatible identifier)
- Audio codec with headphone out, microphone and line in jacks
- 128Mb Serial Flash w/ QSPI interface
- On-board JTAG programming and UART to USB converter
- GPIO: 6 pushbuttons, 4 slide switches, 5 LEDs
- Six Pmod connectors (1 processor-dedicated, 1 dual analog/digital, 3 high-speed differential, 1 logic-dedicated)

# ZYNQ



## Processing System (PS)

### ARM Cortex-A9 Based Application Processor Unit (APU)

- 2.5 DMIPS/MHz per CPU
- CPU frequency: Up to 1 GHz
- Coherent multiprocessor support
- ARMv7-A architecture
  - TrustZone® security
  - Thumb®-2 instruction set
- Jazelle® RCT execution Environment Architecture
- NEON™ media-processing engine
- Single and double precision Vector Floating Point Unit (VFPU)
- CoreSight™ and Program Trace Macrocell (PTM)
- Timer and Interrupts
  - Three watchdog timers
  - One global timer
  - Two triple-timer counters

# Trabajo realizado

- Recopilación de fuentes. Preparación del entorno.
- Proyecto de prueba en Vivado (PL).(Xilinx Vivado)
- Generación de FSBL y binario para la lógica programable.(Xilinx SDK)
- Compilación de u-boot.
- Generación de .BIN (FSBL + .BIT + U-BOOT).(Xilinx SDK)
- Compilación kernel de Linux.
- Generación del device tree.(Xilinx SDK)
- Prueba en SD con distribuciones de ubuntu y debian para esta ZYNQ.
- Generación de ROOT\_FS con buildroot.
- Pruebas básicas con el HW implementado en la PL.

# Preparación del entorno

- Se instala todo el entorno de desarrollo de Xilinx Vivado. Versión 2019.1
- Recordar ejecutar el script post instalación con permisos sudo para instalar los driver para el puerto serie por UART.
- Como es una FPGA y tiene para programable y los pines pueden variar de posiciones, hay que encontrar el archivo de constraint de pines. Como digilent que es el proveedor del kit de desarrollo tiene un repositorio con esta información:
  - `git clone https://github.com/Digilent/vivado-boards.git.`
- También hay que decirle al Vivado cuando arranque que busque las placas disponibles en el directorio correspondiente.
  - Generar un archivo .tcl en: `$HOME/.Xilinx/Vivado` llamado `Vivado_init.tcl` el contenido del archivo debe ser : `set_param board.repoPaths [list "/home/glavigna/vivado-boards/new/board_files"]`

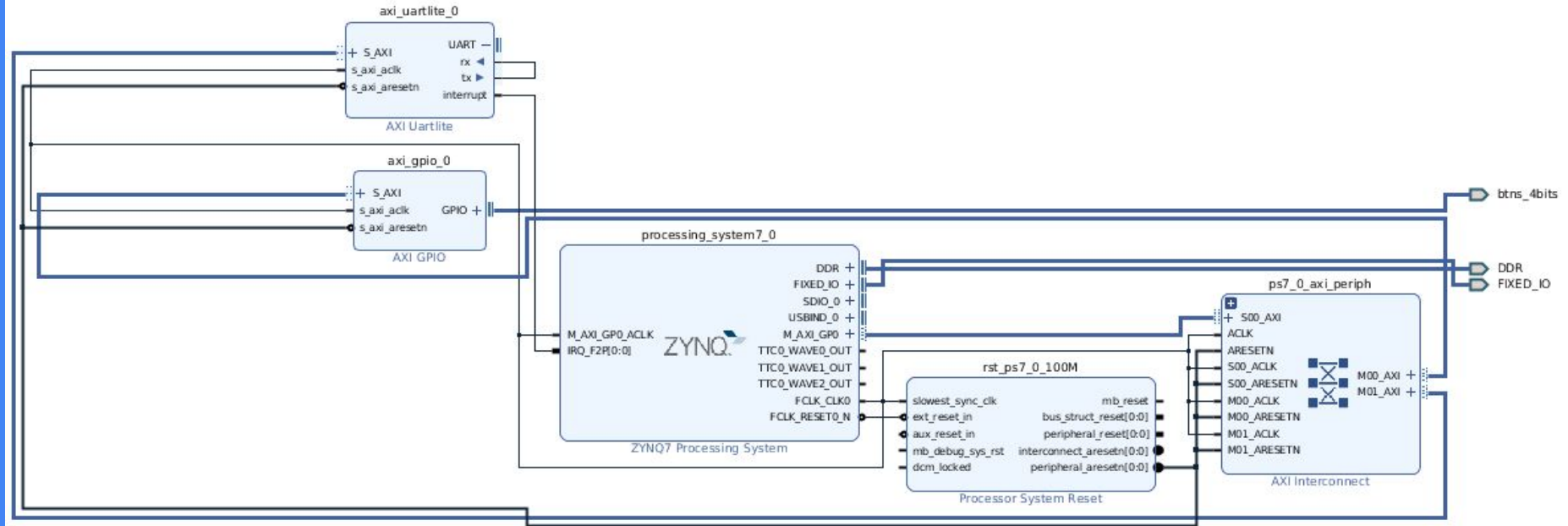
# Preparación del entorno

- Xilinx mantiene versiones del kernel de linux y otras herramientas customizadas para su dispositivos. Se hace un git clone de los siguientes repositorios. (Para todos los repositorios se trabajó con el master):
  - <https://github.com/Xilinx/linux-xlnx.git>. (Kernel de linux con parches y driver específicos).
  - <https://github.com/Xilinx/u-boot-xlnx.git>. (Bootloader con parches de xilinx y drivers específicos).
  - <https://github.com/Xilinx/device-tree-xlnx.git>. (Plugin del sdk de xilinx para poder generar devicetree). Tener en mente que al tener un FPGA el HW puede variar dependiendo de que termine implementando el usuario.
- Para buildroot, se utilizó el repositorio oficial:
  - [git://git.buildroot.net/buildroot](https://git.buildroot.net/buildroot)
- Para el compilador del device tree se utiliza el ejecutable dtc, que esta dentro del kernel de linux.

# Configuración del PATH

- Para todo este trabajo se utilizó el toolchain del SDK de Xilinx. No se generó un toolchain.
- Ir al directorio de instalación del Vivado para agregar el toolchain del SDK de Xilinx al PATH para tener a disposición el toolchain:
  - Ir al directorio: `/tools/Xilinx/Vivado/2019.1/`
  - Ejecutar `→ source settings64.sh`
- Xilinx agrega al path los compiladores de otras dos arquitecturas que soporta:
  - Microblaze (Procesador embebido en lógica programable).
  - ARM 64 bits.
- La que interesa para este trabajo es la ARM de 32 bits para el cortex A-9 que se encuentra dentro de la PS:
- El toolchain con el que se genera todo el trabajo está en:
  - `/tools/Xilinx/SDK/2019.1/gnu/aarch32/linux/gcc-arm-linux-gnueabi/bin/arm-linux-gnueabihf-gcc`

# Projecto VIVADO





# Proyecto VIVADO

- Para el proyecto de Vivado se genera un proyecto simple, en la ZYBO, cuyos archivos de constraint de Digilent.
- Se agrega el ZYNQ Processing System.
- Se agrega unos periféricos conectados al AXI GP → Correr RUN BLOCK AUTOMATION para que el Vivado haga el conexionado del sistema:
  - Un esclavo AXI-MM para manejar los GPIO. (Se conecta a 4 LEDS de la placa).
  - Un esclavo AXI-MM para manejar una UART LITE a 9600 bps, en la cual se conectan internamente el TX con el RX. (Para poder probar un loopback). (En un principio no se conectó la interrupción).
- Address configuradas por el VIVADO. Nos van a servir estos números si queremos acceder manualmente sin usar el driver de liux:

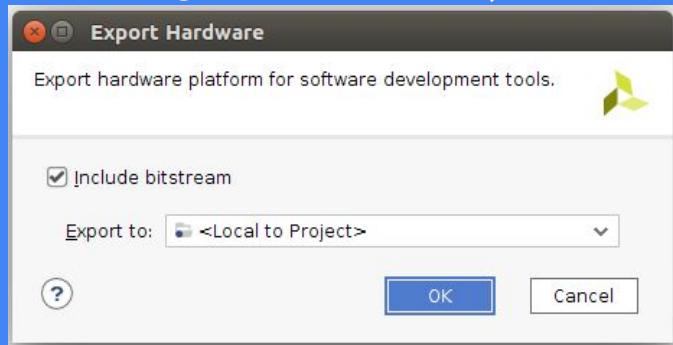
processing_system7_0						
Data (32 address bits : 0x40000000 [ 1G ])						
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	▼	0x4120_FFFF
axi_uartlite_0	S_AXI	Reg	0x42C0_0000	64K	▼	0x42C0_FFFF

# Projecto VIVADO

- Se chequea en el ZYNQ PS la siguiente configuración para que quede cableada la interfaz de Ethernet:

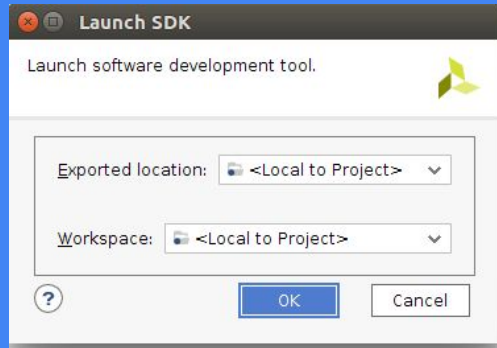


- Una vez configurado el proyecto, realizar la implementación: RUN IMPLEMENTATION. Exporta la definición del HW importante incluir el bitstream. (U-BOOT lo va a grabar en la PL)



# Projecto VIVADO

- Una vez exportado el HW, hay que abrir el SDK. Lo interesante que si cambiar algo de la PL, el SDK lo detecta y vuelve a compilar todos los proyecto. Es importante dejar estas opciones por default, nos conviene que la herramienta haga este trabajo.



- Con el HW generado, la descripción de pines generada, los dispositivos que se hicieron en la PL, trabajamos en el SDK para generar el device tree y el First Stage Bootloader.

# SDK → HW PLATFORM

- Acá lo importante es ver que los 2 esclavos AXI-MM los haya reconocido. También nos indica cómo son los accesos a la memoria, tiene sentido que sean del tipo REG, ya que con accesos simples la PS va a poder controlar estos periféricos.
- También podemos ver en este archivo nos colocó los dispositivos como esclavos AXI, y la posición de memoria coincide con el Address editor del Vivado.
- También identificamos que podemos Usar otros dispositivos que son propios del cortex A-9, como GPIO, I2C, CAN y otras cosas interesantes.

## Design Information

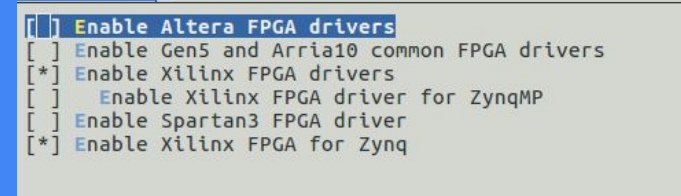
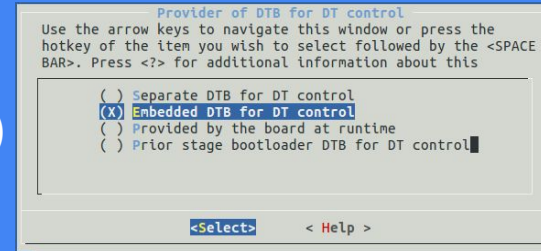
Target FPGA Device: 7z010  
Part: xc7z010clg400-1  
Created With: Vivado 2019.1  
Created On: Fri Aug 16 23:26:23 2019

## Address Map for processor ps7\_cortexa9\_0[0-1]

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
ps7_intc_dist_0	0xf8f01000	0xf8f01fff		REGISTER
ps7_gpio_0	0xe000a000	0xe000afff		REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f0061f		REGISTER
ps7_slcr_0	0xf8000000	0xf8000fff		REGISTER
axi_gpio_0	0x41200000	0x4120ffff	S_AXI	REGISTER
ps7_scuwdt_0	0xf8f00620	0xf8f006ff		REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_scuc_0	0xf8f00000	0xf8f000fc		REGISTER
ps7_qspi_linear_0	0xfc000000	0xfcffffff		FLASH
ps7_pmu_0	0xf8893000	0xf8893fff		REGISTER
ps7_afi_1	0xf8009000	0xf8009fff		REGISTER
axi_uartlite_0	0x42c00000	0x42c0ffff	S_AXI	REGISTER
ps7_afi_0	0xf8008000	0xf8008fff		REGISTER

# U-BOOT

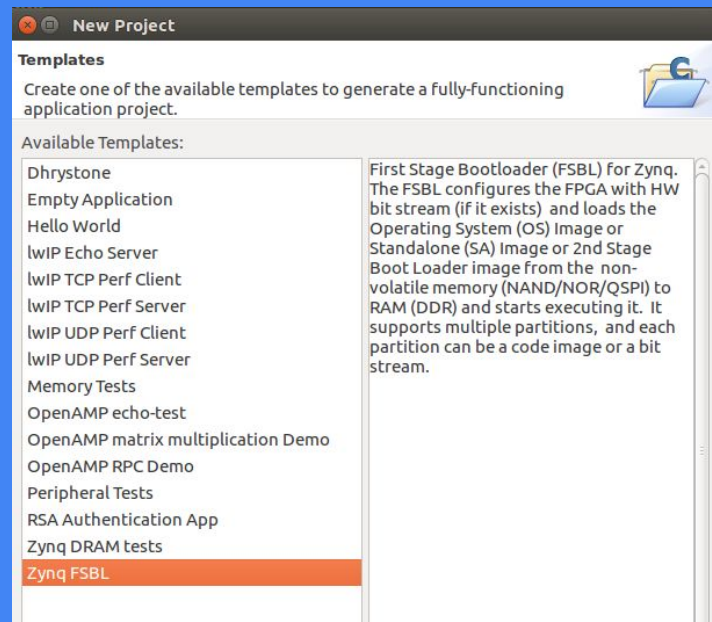
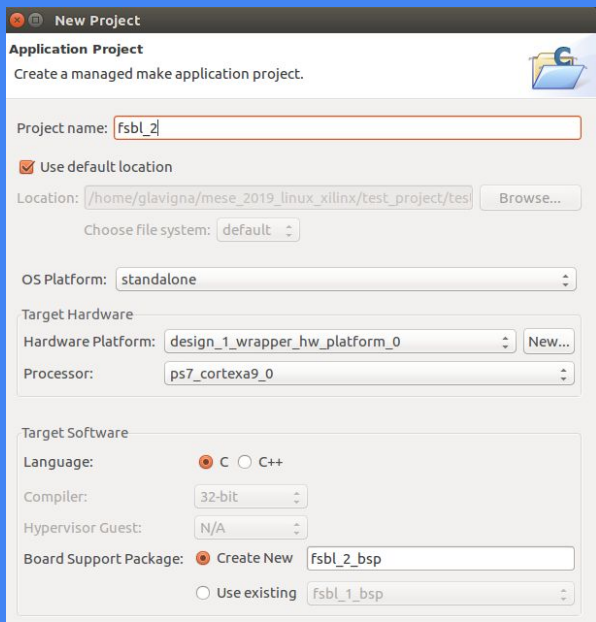
- Este proceso es muy similar al de la beaglebone, tener en mente que trabajamos en el repo de Xilinx. (Esta parte es muy sensible al HW, así que nos sirven que haya una configuración para nuestra placa).
  - `make zynq_zybo_config`
  - `make ARCH=arm menuconfig.(CONFIG_OF_EMBED)`
  - Verificar en Device Drivers / FPGA Support: tener en cuenta que la FPGA, tiene que estar programada antes del Linux, para reconocer los dispositivos en la PL.
  - `make -j12`
  - `cd tools/ --> export PATH=`pwd`: $PATH` (Esto se usa después en el SDK)
  - El SDK no ubica archivos sin extensión, así que hay que hacer un `--> mv u-boot u-boot.elf`



```
(base) glavigna@glavigna-desktop:~/mese_2019_linux_xilinx/u-boot-xlnx$ file u-boot.elf
u-boot.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, not stripped
```

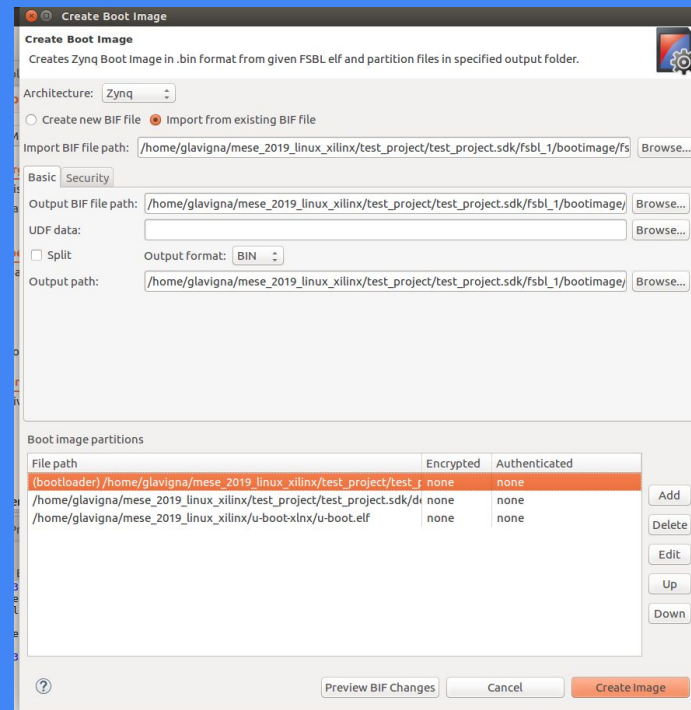
# Generación FSBL y BOOT.bin

- Con el HW que tenemos hay que generar un First Stage Bootloader. File → New Application Project. Seleccionar el template para FSBL ZYNQ.
- Verificar que este el HW que exportamos en el VIVADO. Esto lo utiliza para generar el BSP. Por ejemplo tiene que saber que PS estamos usando.



# Generación FSBL y BOOT.bin

- Con esto nos genera un BSP, con driver básico para hacer el primer stage del booteo.
- Sobre la carpeta del proyecto del SDK generar , click derecho y pulsar “Create Boot Image”:
- Clickear en ADD y agregar:
  - fsbl.elf → Del proyecto FSBL
  - \*\_wrapper.bit → Archivo PL
  - u-boot.elf → Generado anteriormente
- Este archivo BOOT.BIN tiene todo lo necesario para poder bootear el sistema.



# Kernel de Linux

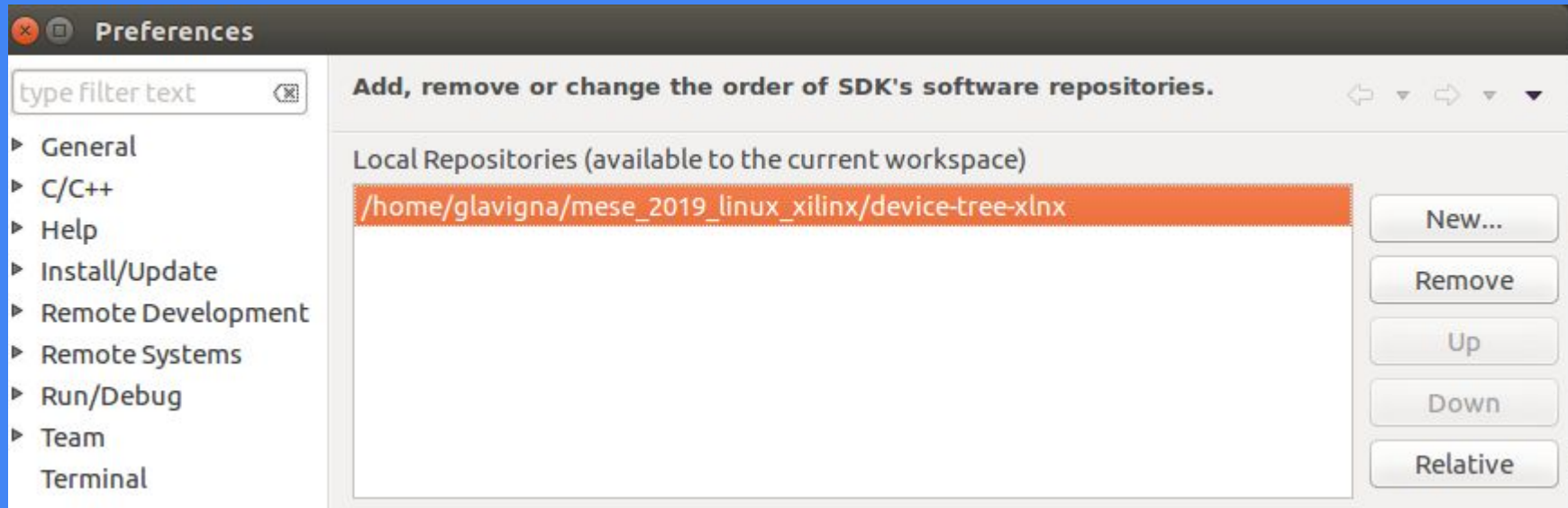
- Vamos al repositorio que bajamos de xilinx:
- `make ARCH=arm xilinx_zynq_defconfig`. (Esta configuración debe servir para todas las placas basadas en ZYNQ, tener en mente que lo más sensible al HW es el u-boot y el fsbl).
- `make ARCH=arm menuconfig`
- En Device Drivers → Character Devices → Serial Drivers (Seleccionar uartlite sino lo coloca como módulo de kernel, y lo queremos desde el arranque).
- `make ARCH=arm UIIMAGE_LOADADDR=0x8000 ulmage -j12`
- Para instalar los módulos de kernel en el ROOT\_FS: → `sudo make ARCH=arm INSTALL_MOD_PATH=../ROOT_FS/ modules_install`

```
(base) glavigna@glavigna-desktop:~/mese_2019_linux_xilinx$ file uImage
uImage: u-boot legacy uImage, Linux-4.19.0-xilinx-00126-gf96ce, Linux/ARM, OS Kernel Image (Not compressed), 4162736 bytes, Sat Aug 17 00:21:53 2019, Load Address: 0x00008000, Entry Point: 0x00008000, Header CRC: 0x5B8545D1, Data CRC: 0x6C72430E
```



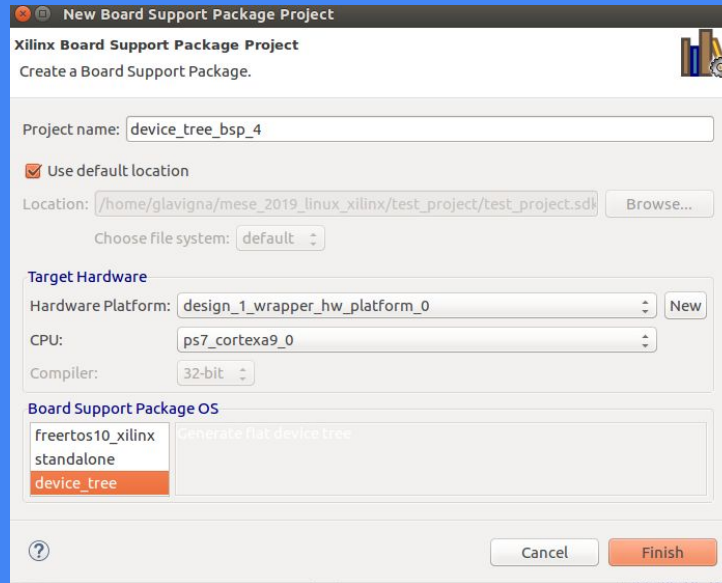
# Generación Device Tree

- Para esto se utiliza el SDK. Se tiene que importar el plugin para poder generar el device tree. Hacer lo siguiente en Xilinx → Repositories:



# Generación Device Tree

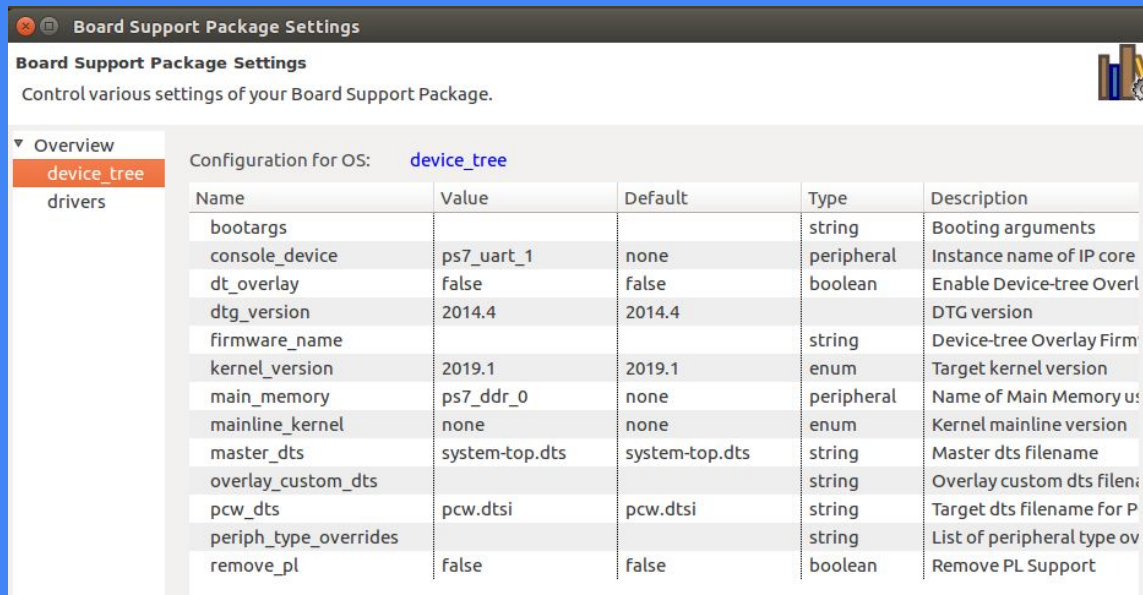
- Después hay que generar el proyecto para generar el device tree:
  - File → New → Board Support Package



- Esto nos genera un device tree, con la especificación del HW en la PL

# Generación Device Tree

- Para que funcione hay que cambiar, hay que apretar en system.mms en el proyecto de device tree que nos generó. “Modify BSP settings”.
- Cambiar el “console device” de axi\_uartlite\_0 a ps7\_uart\_1. (Si no el sistema arranca levantando una consola por la uart en la PL)



The screenshot shows the 'Board Support Package Settings' window. The 'device\_tree' tab is selected in the left sidebar. The main area displays a table of configuration settings for the OS.

Name	Value	Default	Type	Description
bootargs			string	Bootting arguments
console_device	ps7_uart_1	none	peripheral	Instance name of IP core
dt_overlay	false	false	boolean	Enable Device-tree Overl
dtg_version	2014.4	2014.4		DTG version
firmware_name			string	Device-tree Overlay Firm
kernel_version	2019.1	2019.1	enum	Target kernel version
main_memory	ps7_ddr_0	none	peripheral	Name of Main Memory us
mainline_kernel	none	none	enum	Kernel mainline version
master_dts	system-top.dts	system-top.dts	string	Master dts filename
overlay_custom_dts			string	Overlay custom dts filen
pcw_dts	pcw.dtsi	pcw.dtsi	string	Target dts filename for P
periph_type_overrides			string	List of peripheral type ov
remove_pl	false	false	boolean	Remove PL Support

# Generación Device Tree

- Lo interesante es que con la definición del HW, el SDK generó un .dts con el HW donde tenemos los periféricos que se generaron en la PL.
- También hay un archivo con .dts Con todos los dispositivos Que existen en la PS.
- También hay información de los drivers que son compatibles.

```
/ {
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;
        axi_gpio_0: gpio@41200000 {
            #gpio-cells = <3>;
            clock-names = "s_axi_aclk";
            clocks = <&clkc 15>;
            compatible = "xlnx,axi-gpio-2.0", "xlnx,xps-gpio-1.00.a";
            gpio-controller ;
            reg = <0x41200000 0x10000>;
            xlnx,all-inputs = <0x0>;
            xlnx,all-inputs-2 = <0x0>;
            xlnx,all-outputs = <0x0>;
            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>;
        };
        axi_uartlite_0: serial@42c00000 {
            clock-names = "s_axi_aclk";
            clocks = <&clkc 15>;
            compatible = "xlnx,axi-uartlite-2.0", "xlnx,xps-uartlite-1.00.a";
            current-speed = <9600>;
            device_type = "serial";
            interrupt-names = "interrupt";
            interrupt-parent = <&intc>;
            interrupts = <0 29 1>;
            port-number = <1>;
            reg = <0x42c00000 0x10000>;
            xlnx,baudrate = <0x2580>;
            xlnx,data-bits = <0x8>;
            xlnx,odd-parity = <0x0>;
            xlnx,s_axi-aclk-freq-hz-d = "100.0";
            xlnx,use-parity = <0x0>;
        };
    };
};
```

# Compilación Device Tree

- El device tree tiene un bug, que no nos permite compilarlo, no cumple la sintaxis de los device tree. Hay que cambiar en el archivo system-top.dts los “#” por “/.../” . Tiene que quedar como a continuación:

```
/dts-v1;  
/include/ "zynq-7000.dtsi"  
/include/ "pl.dtsi"  
/include/ "pcw.dtsi"  
/ {
```

- Para compilarlo y generar un .dtb hay que ir a la carpeta :
  - cd linux-xlnx/scripts/dtc/
  - ./dtc -I dts -O dtb -o ../../../../devicetree.dtb  
../../../../test\_project/test\_project.sdk/device\_tree\_bsp\_3/system-top.dts
- Fijarse que no genere error, y ahora tenemos un archivo devicetree.dtb.

# ROOT FS → Buildroot

- Para esto utilizamos el toolchain de xilinx, y trabajamos con la rama master de buildroot.
- Primero cargamos una configuración de una placa parecida a la nuestra la ZEBOARD. Si trabajamos por defecto no aparece el cortex A9 con FPU NEON.
  - `make zynq_zed_defconfig`
- Lo importante es pegarle a la configuración del toolchain y ver los header del kernel que tiene el toolchain. (En este caso la 4.14)





```
Toolchain type (External toolchain) --->
*** Toolchain External Options ***
Toolchain (Custom toolchain) --->
Toolchain origin (Pre-installed toolchain) --->
(/tools/Xilinx/SDK/2019.1/gnu/aarch32/lin/gcc-arm-linux-gnueabi) Toolchain path
(arm-linux-gnueabihf) Toolchain prefix
External toolchain gcc version (8.x) --->
External toolchain kernel headers series (4.14.x) --->
External toolchain C library (glibc/eglibc) --->
```

# ROOT FS → Buildroot

- Además le colocamos lo siguiente, para hacer algunas pruebas básicas:
  - Python 2.7 (Con librería pyserial)
  - ssh
  - nano
- Una vez que montamos la SD y cargamos la partición EXT4 en la carpeta ROOT\_FS (Vamos a hablar más adelante), tenemos que hacer lo siguiente:
  - `sudo tar -xvf ../../buildroot/output/images/rootfs.tar`

# Tarjeta SD

- Se formatea con `sudo gparted`, con una partición FAT para el bootloader y otra EXT 4 para el ROOT FS.

Partition		File System	Mount Point	Label	Size	Used	Unused	Flags
unallocated		unallocated			4.00 MiB	—	—	
/dev/sdc1		 fat32	/media/glavigna/BOOT	BOOT	36.00 MiB	7.18 MiB	28.82 MiB	
/dev/sdc2		 ext4	/media/glavigna/ROOT_FS	ROOT_FS	7.46 GiB	593.91 MiB	6.88 GiB	

- `sudo mount /dev/sdb1 ./BOOT`
  - `sudo mount /dev/sdb2 ./ROOT_FS`
- Utilizar el comando `sync`, cuando se escriba en la SD, para asegurarnos que la SD se haya escrito totalmente.
- Para desmontar:
  - `sudo umount /dev/sdc1`
  - `sudo umount /dev/sdc2`



# Tarjeta SD

- En la partición 1 de la SD (FAT 32):
  - Archivo BOOT.BIN (Generado en el proyecto FSBL del SDK) contiene:
    - FSBL
    - Archivo .bit de la FPGA
    - U-boot.elf
  - ulmage Kernel de linux que esta en: linux-xlnx/arch/arm/boot/ulmage)
  - devicetree.dtb (Se genero con ./dttc)
- En la partición 2 de la SD (EXT 4)
  - Descomprimir buildroot/output/images/rootfs.tar
  - Instalar modulos de kernel

# Booteo por SD

- Colocar el switch de ON/OFF en posición de OFF.
- Verificar que JP5 está configurado en SD.
- Conectar cable USB al puerto PROG UART.
- Colocar el switch de ON/OFF en posición de ON.
- Conectar una consola serie con picocom (El USB puede variar):
  - `picocom -b 115200 /dev/ttyUSB1`

```
U-Boot 2019.01-00099-gf65575c (Aug 16 2019 - 04:29:56 -0300)

CPU:      Zynq 7z010
Silicon:  v3.1
Model:    Digilent Zybo board
DRAM:     ECC disabled 512 MiB
MMC:      mmc@e0100000: 0
Loading Environment from SPI Flash... SF: Detected s25fl128s_64k with page size 256 Bytes, erase size 64
KiB, total 16 MiB
OK
In:       serial@e0001000
Out:      serial@e0001000
Err:      serial@e0001000
Net:      ZYNQ GEM: e000b000, phyaddr 0, interface rgmii-id

Warning: ethernet@e000b000 (eth0) using random MAC address - 16:c6:8c:71:c0:1e
eth0: ethernet@e000b000
Hit any key to stop autoboot:  0
Zynq> █
```

# Booteo por SD

- Enviar los siguientes comandos al u-boot:
  - `setenv bootargs console=ttyPS0,115200n8 consoleblank=0 root=/dev/mmcblk0p2 rw rootwait earlyprintk`
  - `fatload mmc 0 0x3000000 ulmage`
  - `fatload mmc 0 0x2A00000 devicetree.dtb`
  - `bootm 0x3000000 - 0x2A00000`
- Vemos que bootea, con la versión de kernel 4.19 que compilamos nosotros.

```
Booting Linux on physical CPU 0x0
Linux version 4.19.0-xilinx-00126-gf96ce67 (glavigna@glavigna-desktop) (gcc version 8.2.0 (GCC)) #1 SMP P
REEMPT Fri Aug 16 21:19:46 -03 2019
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
```

# Booteo por SD

- Vemos en el booteo que levanta los dispositivo que programamos en la FPGA:
- GPIO, en el address especificado en el devicetree.

```
GPIO IRQ not connected
XGpio: gpio@41200000: registered, base is 1020
```

- UARTLITE

```
42c00000.serial: ttyUL1 at MMIO 0x42c00000 (irq = 46, base_baud = 0) is a uartlite
```

- Si hacemos un `ls /dev/` vemos que los dispositivos que programamos en la PL estan montados.

```
# ls /dev
console
cpu_dma_latency
fd
full
gpiochip0
gpiochip1
iio:device0
```

```
tty8
tty9
ttyPS0
ttyUL1
urandom
```

# Booteo por SD

- Vemos que levanto el dhcp, y nos pudimos conectar a nuestro router.

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 16:C6:8C:71:C0:1E
          inet addr:192.168.0.17  Bcast:192.168.0.255  Mask:255.255.255.0
```

- Nos conector por ssh con ssh root@192.168.0.17

```
(base) glavigna@glavigna-desktop:~$ ssh root@192.168.0.17
root@192.168.0.17's password:
# ls
minicom.log
# pwd
/root
# ls /dev
console      port          tty0          tty32         tty56
cpu_dma_latency  ptmx         tty1          tty33         tty57
fd           pts          tty10         tty34         tty58
full         ram0         tty11         tty35         tty59
gpiochip0    ram1         tty12         tty36         tty6
gpiochip1    ram10        tty13         tty37         tty60
iio:device0  ram11        tty14         tty38         tty61
kmsg         ram12        tty15         tty39         tty62
log          ram13        tty16         tty4          tty63
loop-control ram14        tty17         tty40         tty7
loop0        ram15        tty18         tty41         tty8
loop1        ram2         tty19         tty42         tty9
loop2        ram3         tty2          tty43         ttyPS0
loop3        ram4         tty20         tty44         ttyUL1
loop4        ram5         tty21         tty45         urandom
loop5        ram6         tty22         tty46         vcs
loop6        ram7         tty23         tty47         vcs1
loop7        ram8         tty24         tty48         vcsa
mem          ram9         tty25         tty49         vcsa1
memory_bandwidth random        tty26         tty5          vcsu
mncblk0      shm          tty27         tty50         vcsu1
mncblk0p1    snd          tty28         tty51         vga_arbiter
mncblk0p2    stderr       tty29         tty52         watchdog
network_latency stdin         tty3          tty53         watchdog0
network_throughput stdout        tty30         tty54         zero
null         tty          tty31         tty55
```

# Pruebas “muy” simples

- Utilizamos un script muy básico en python, para mover los GPIO (conectado a leds), mostrando que se puede tener acceso a bajo nivel al HW.

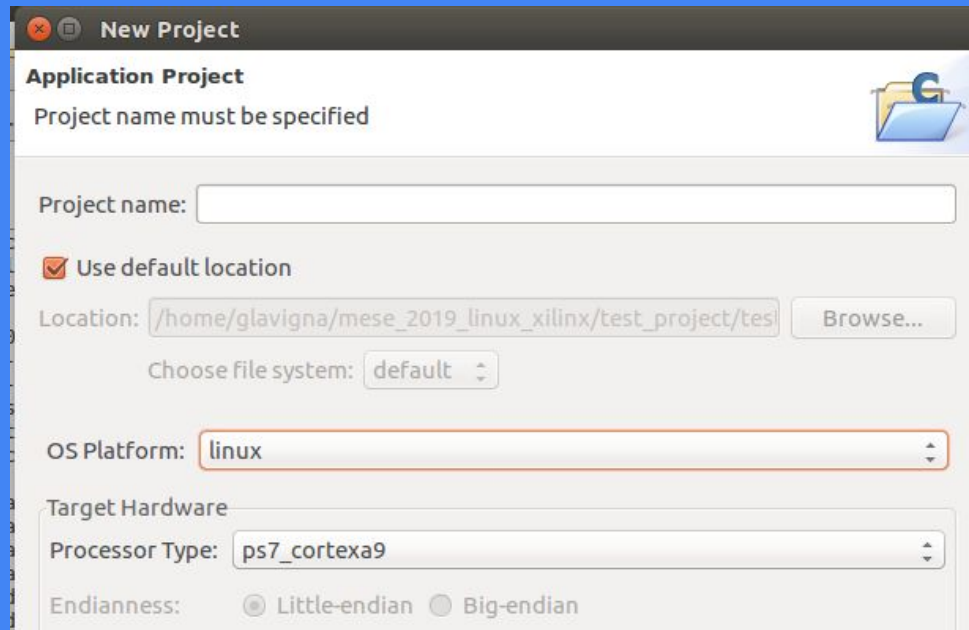
```
glavigna@glavigna-desktop: ~  
GNU nano 4.3 test_gpio.py  
from time import sleep  
import mmap  
  
print("Todos sono mortales")  
with open("/dev/mem", "r+b") as f:  
    mm = mmap.mmap(f.fileno(), 8, offset=0x41200000)  
    mm[4] = chr(0x00)  
    while True:  
        try:  
            mm[0] = chr(0xFF)  
            sleep(0.2)  
            mm[0] = chr(0x00)  
            sleep(0.2)  
        except KeyboardInterrupt:  
            break;  
    mm.close()
```

Read 17 lines

^G Get Help	^O Write Out	^W Where Is	^K Cut Text	^J Justify	^C Cur Pos
^X Exit	^R Read File	^_ Replace	^U Paste Text	^T To Spell	^_ Go To Line

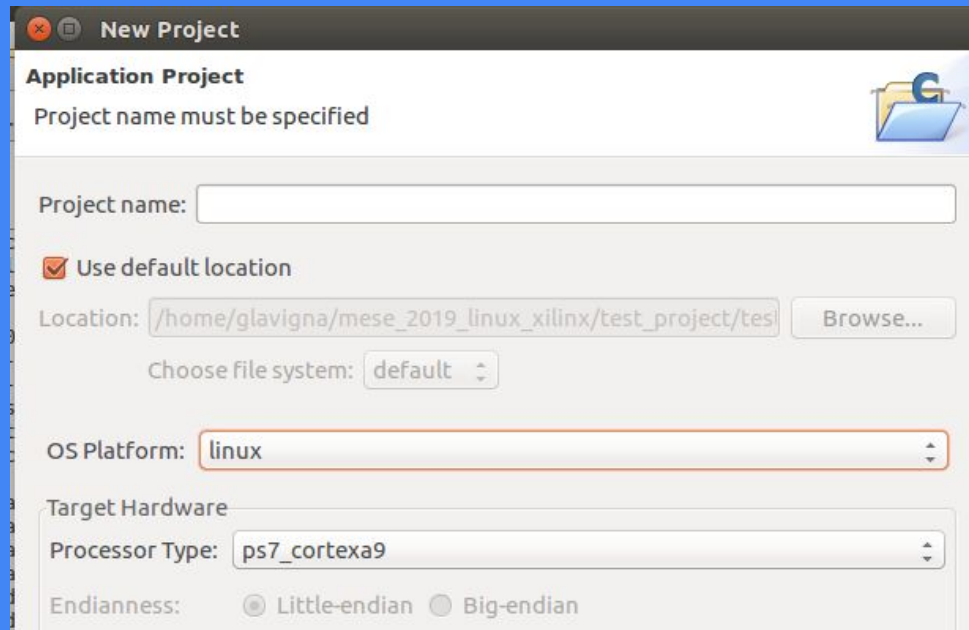
# Pruebas “muy” simples

- En el SDK de xilinx podemos generar aplicaciones de la siguiente manera:
  - File → New → Application Project. (Seleccionar Linux y el Procesador que estamos usando)



# Pruebas “muy” simples

- En el SDK de xilinx podemos generar aplicaciones de la siguiente manera:
  - File → New → Application Project. (Seleccionar Linux y el Procesador que estamos usando)





# Pruebas “muy” simples

- Generamos una aplicación que abra el dispositivo de uart lite y podamos probar el loopback.

```
fd = open("/dev/ttyUL1",O_RDWR | O_NOCTTY | O_NDELAY); /* ttyUSB0 is the FT232 based USB2SERIAL Converter */
                                                    /* O_RDWR Read/Write access to serial port */
                                                    /* O_NOCTTY - No terminal will control the process */
                                                    /* O_NDELAY -Non Blocking Mode,Does not care about- */
                                                    /* -the status of DCD line,Open() returns immediatly */
```

```
bytes_written = write(fd,write_buffer,sizeof(write_buffer));/* use write() to send data to port
/* "fd" - file descriptor pointing to the opened serial port */
/* "write_buffer" - address of the buffer containing data */
/* "sizeof(write_buffer)" - No of bytes to write */

printf("\n %s written to ttyUL1",write_buffer);
printf("\n %d Bytes written to ttyUL1", bytes_written);
printf("\n +-----+\\n\\n");

sleep(2);

bytes_readed = read(fd,read_buffer,4);

read_buffer[16] = 0;
printf("Bytes Leidos = %d Buffer: %s \\r\\n",bytes_readed,read_buffer);

close(fd);/* Close the Serial port */
```

# Pruebas “muy” simples

- Lo corremos y nos da lo siguiente, hay algo con el \r \n que hay que colocarlo, hay que indagar más sobre el driver de UART.
- Vemos que escribimos datos en la UART LITE en la PL que tiene TX y RX conectados dentro de la PL y recibimos lo que mandamos.

```
# ./serial_write.elf

ttyUL1 Opened Successfully
BaudRate = 9600
StopBits = 1
Parity = none
ABC
written to ttyUL1
 6 Bytes written to ttyUL1
+-----+

Bytes Leidos = 4 Bufffer: ABC

# █
```

# Problemas que demoraron mucho tiempo.

- Configuración de U-BOOT para esta placa. (Al principio no booteaba).
- Device Tree de SDK con problemas de sintaxis.
- Configuración de BSP para Device Tree empezaba el booteo por la UART colocada en la PL.
- Se generó un proyecto de VIVADO sin la IRQ de la UART LITE a la PS, lo cual hacía que el linux no levantaba el /dev/ttyUL1.
- Configuración de buildroot para trabajar con el toolchain de Xilinx. (No encontraba cortex A-9 con opción para FPU con NEON). Se bajaron enlatados de UBUNTU y DEBIAN para probar un ROOT FS.
- Direcciones de carga para el Kernel y del Device Tree.
- Comandos a U-BOOT para hacer booteo por la SD.

# Cosas para seguir

- Probar con ARTY-Z7. (Cambio más significativo creo U-BOOT).
- Booteo por TFTP y NFS. Grabar la SD todo el tiempo es un retraso.
- Booteo por QSPI. Útil para un producto que no requiera SD.
- Generación en VIVADO proyecto para VGA, para darle una interfaz gráfica.
- Generación en VIVADO proyecto para HDMI, para darle una interfaz gráfica.
- Escribir drivers en Kernel Space para manejar dispositivos más AD/HOC.
- Manejo de DMAS dentro del LINUX. (Útil para procesamiento de señales).
- Ver de programar la FPGA desde Linux, y forzar una carga de vuelta del device tree sin necesidad de resetear la placa.
- Xilinx recomienda usar PETA LINUX, investigar este entorno.
- Ver cómo compilar otras aplicaciones que no están buildroot.
- Ver si se puede utilizar crosstool-ng para reemplazar el toolchain del SDK.
- Probar con el mainline de LINUX y U-BOOT, capaz para aplicaciones simple que no usen HW complejo funcione igual.
- Replicar proyecto IMD, con el I2C PS y un I2C en la PL.