

Universidad de Granada



Sistemas Críticos

Plataforma criticidad mixta basada en Linux

Marvin Matías Agüero Torales

maguero@correo.ugr.es

Curso 2016-2017

Sumario

Enunciado de la práctica.....	3
Desarrollo.....	3
Selección y configuración de la plataforma de ejecución.....	3
Prerrequisito.....	3
Diseño de una plataforma de ejecución mínima.....	4
Construcción de un Linux empotrado desde cero.....	4
Descarga y configuración por defecto del kernel de Linux.....	4
Ajuste de la configuración y construcción del kernel de Linux.....	5
Construcción del Device Tree Blob.....	5
Tareas Previas.....	5
Generación y construcción del Device Tree Source mediante el SDK.....	5
Construcción del Device Tree Blob a partir del Device Tree Source.....	6
Necesidad de un Root File System.....	6
Construcción de un Root File System.....	7
Tareas Previas.....	7
Configuración mínima y construcción de busybox.....	7
Construimos el rootfs.....	8
Arranque de la placa y ejecución de Linux.....	10
Creación del First Stage Boot Loader mediante el SDK.....	10
Construcción de U-Boot.....	10
Preparación de la imagen de arranque.....	11
Preparación del kernel y el RootFS para ser cargados por U-Boot.....	11
Preparación de la placa.....	11
Inconvenientes.....	12
Opcional.....	13
Conclusiones.....	13
Anexos.....	14

Enunciado de la práctica

En esta práctica se trabajará con la programación de sistemas empotrados de criticidad mixta con Linux. El alumno deberá:

- Diseñar un hardware básico incluyendo algunos periféricos con Vivado.
- Crear el devicetree con SDK a partir del hardware generado en el paso 1.
- Crear la imagen del kernel para la Zybo.
- Crear un sistema de archivos mínimo.
- Generar con SDK el sistema de arranque de la tarjeta.
- Crear tareas críticas y no críticas capaces de modificar GPIOs y/o el contenido de una BRAM desde el espacio de usuario [Opcional]

Además de las actividades realizadas. Incluir los principales problemas resueltos así como los elementos del diseño y formación recibida más interesantes.

Se deberá adjuntar un archivo comprimido con los 4 archivos de la tarjeta microSD necesarios para arrancar el sistema: boot.bin, uImage, uramdisk y devicetree.dtb.

Desarrollo

El desarrollo se hace sobre una portátil HP Envy con procesador i7 3º generación con 8 cores y 8 GB de RAM, con SO Ubuntu 16.04 LTS de 64 bits.

Se siguió el material de la asignatura para llevar acabo este trabajo (González Peñalver, 2015).

Selección y configuración de la plataforma de ejecución

Prerrequisito

Usaremos la plataforma Diligent Zybo y la placa de desarrollo basada en el SoC Zynq de Xilinx.

Como pre-requisitos necesitamos las herramientas de Xilinx: Vivado and SDK Standalone Web Install Client; herramientas de control de versiones: Git; Gmake; Fakeroot para poder construir el RootFS del sistema con permisos de root.

```
XILINX_ROOT=/opt/Xilinx
```

```
PATH=$PATH:$XILINX_ROOT/SDK/2014.4/bin
```

```
PATH=$PATH:$XILINX_ROOT/SDK/2014.4/gnu/arm/lin/bin
```

```
PATH=$PATH:$XILINX_ROOT/Vivado/2014.4/bin
```

```
export PATH
```

```
sudo apt-get -y install git
```

```
sudo ln -s /usr/bin/make /usr/bin/gmake
```

```
sudo apt-get -y install fakeroot
```

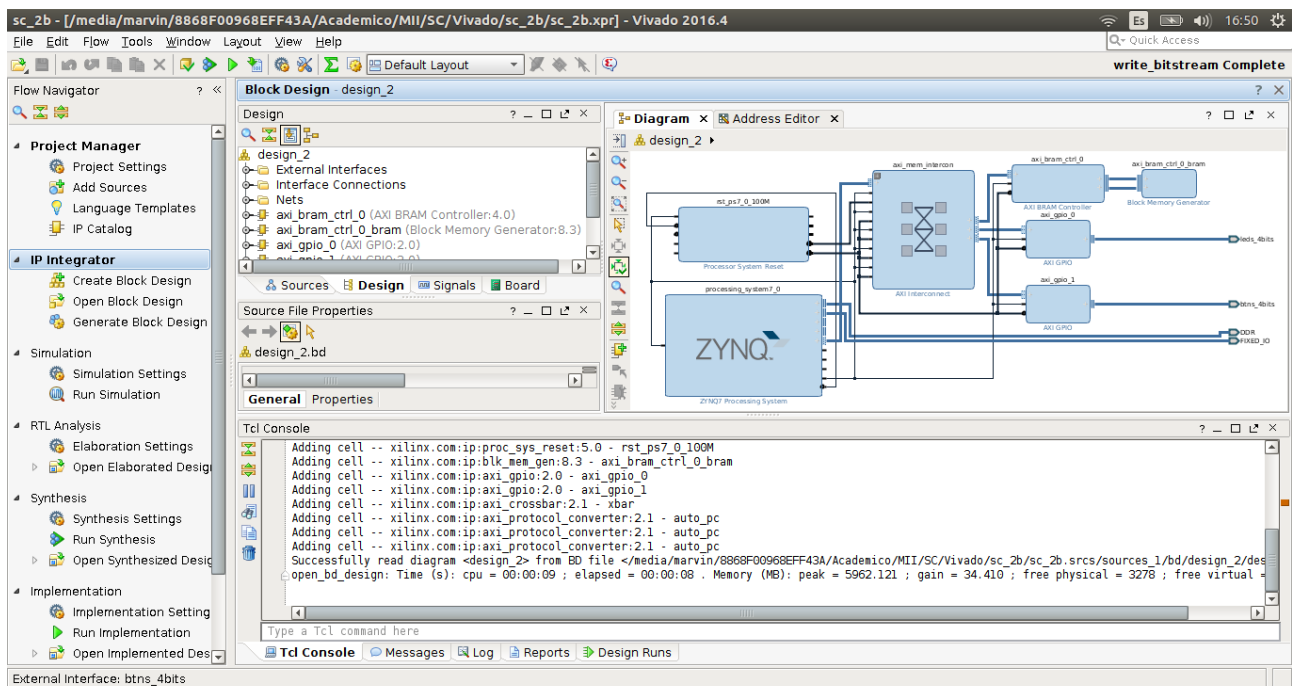
Y configurar el directorio de nuestro proyecto

```
export PRJ_ROOT="${HOME}/zynq-linux"
```

Diseño de una plataforma de ejecución mínima

Creamos un proyecto en Vivado del tipo RTL, Target Language: VHDL, Simulator Language: Mixed, la Board la Zybo de Digilent Inc. Dentro del proyecto, creamos un diseño mínimo: Create block design, añadimos el SoC Zynq de Xilinx: Add IP, automatizamos las conexiones: Run Block Automation, añadimos el controlador de memoria y el GPIO: Add IP, automatizamos las conexiones: Run Connection Automation (en GPIO, led de 4 bits), regeneramos el layout del diseño, fijamos el tamaño máximo de la BRAM a 64K: Address Editor, validamos el diseño: pulsando F6.

Una vez validado el diseño, pasamos a la generación de los ficheros HDL: IP Integrator, Generate Block Design; creamos el Bitstream: Program and Debug, Generate Bitstream.



Construcción de un Linux empotrado desde cero

Descarga y configuración por defecto del kernel de Linux

Obtención de las fuentes

```
KERNEL="Linux-Digilent-Dev"
```

```
DILIGENT_GIT="https://github.com/DigilentInc"
```

```
KERNEL_DIR="${PRJ_ROOT}/${KERNEL}"
```

```
git -C ${PRJ_ROOT} clone -b master-next ${DILIGENT_GIT}/${KERNEL}.git
```

Fijamos el valor de las variables de entorno necesarias

```
export ARCH="arm"
```

```
export HOST="${ARCH}-xilinx-linux-gnueabi"
```

```
export CROSS_COMPILE="/opt/Xilinx/SDK/2016.4/gnu/arm/lin/bin/${HOST}-"
```

Limpiamos restos de compilaciones anteriores

```
cd ${KERNEL_DIR}
```

```
make distclean
```

Configuramos por defecto para la plataforma

```
make xilinx_zynq_defconfig ARCH=arm
```

Ajuste de la configuración y construcción del kernel de Linux

Ajuste de la configuración

```
make menuconfig
```

```
[*] Enable loadable module support
```

```
Kernel Features
```

```
[*] Use the ARM EABI to compile the kernel
```

```
[*] Allow old ABI binaries to run with this kernel
```

Construimos el kernel (antes averiguamos el no. de cores, 8 en mi caso + 1 virtual)

```
grep processor /proc/cpuinfo | wc -l
```

```
nice make -j 9
```

Lo copiamos al directorio de las imágenes

```
mkdir -p ${PRJ_ROOT}/images
```

```
cp arch/arm/boot/zImage ${PRJ_ROOT}/images
```

Construcción del Device Tree Blob

Tareas Previas

Exportamos el diseño de la plataforma de Vivado al SDK

Abrimos el proyecto Vivado creado anteriormente, en File, Export, Export Hardware y lo guardamos dentro del mismo proyecto, luego File, Launch SDK.

Creación de un proyecto HW en el SDK para nuestra plataforma

En el SDK, File, New, Project, Hardware Platform Specification.

Generación y construcción del Device Tree Source mediante el SDK

Descargamos el Device Tree Generator

```
mkdir -p ${PRJ_ROOT}/dts
```

```
git -C ${PRJ_ROOT}/dts clone git://github.com/Xilinx/device-tree-xlnx.git
```

Importamos el repositorio en el SDK

En el SDK, Xiling Tools, Repositories, agregamos la carpeta clonada device-tree-xlnx.
Luego, File, New, Board Support Package con OS device_tree.

Construcción del Device Tree Blob a partir del Device Tree Source

Para ello se debe ejecutar lo siguiente

```
SDK_DEV_TREE="${PRJ_ROOT}/sc_2b/sc_2b.sdk/device_tree_bsp_0"
cd ${SDK_DEV_TREE}
${KERNEL_DIR}/scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb system-top.dts
cp devicetree.dtb ${PRJ_ROOT}/images
```

Necesidad de un Root File System

Para iniciar es necesario un root file system. Creamos un fichero init

```
vi ${PRJ_ROOT}/myinit.c

#include <stdio.h>

int main ()
{
    /* Escribimos el mensaje de saludo */
    printf ("\n");
    printf ("Hola desde la Zybo!\n");
    /* El proceso init nunca debe terminar */
    while (1) { }
    /* Para que no proteste el compilador */
    return 0;
}
```

Creamos un directorio para almacenar el sistema de archivos

```
mkdir -p ${PRJ_ROOT}/rootfs
cd ${PRJ_ROOT}/rootfs
```

Simulamos ser root

```
fakeroot
```

Compilamos nuestro proceso init (en /)

```
${CROSS_COMPILE}gcc -static ${PRJ_ROOT}/myinit.c -o init
```

Creamos la consola

```
mkdir -m 0755 dev
```

```
mknod dev/console c 5 1
```

Creamos el fichero cpio

```
find . | cpio --quiet -o -H newc | gzip > ${PRJ_ROOT}/images/min_rootfs.cpio.gz
```

Dejamos de simular que somos root

```
exit
```

Construcción de un Root File System

Tareas Previas

Pasamos a la construcción de Busybox, creamos el directorio de instalación

```
mkdir -p ${PRJ_ROOT}/sysapps
```

```
cd ${PRJ_ROOT}/sysapps
```

Obtención de las fuentes de busybox

```
export BUSYBOX="busybox-1.22.1"
```

```
wget http://busybox.net/downloads/${BUSYBOX}.tar.bz2
```

```
tar xf ${BUSYBOX}.tar.bz2
```

Librerías necesarias

```
sudo apt-get install lib32z1 lib32ncurses5 /lib32bz2-1.0/ lib32stdc++6
```

Configuración mínima y construcción de busybox

```
cd ${BUSYBOX}
```

```
export ARCH="arm"
```

```
export HOST="${ARCH}-xilinx-linux-gnueabi"
```

```
export CROSS_COMPILE="/opt/Xilinx/SDK/2016.4/gnu/arm/lin/bin/${HOST}-"
```

```
make defconfig
```

```
make menuconfig ARQ=arm
```

```
Busybox Settings ->
```

```
Build Options ->
```

```
[ ] Build BusyBox as a static binary (no shared libs)
```

```
[ ] Force NOMMU build
```

```
[ ] Build with Large File Support (for accessing files > 2 GB)
```

```
(${CROSS_COMPILE}) Cross Compiler prefix
```

```
make -j 9
```

Construimos el rootfs

Creamos un directorio para almacenar el sistema de archivos

```
rm -rf ${PRJ_ROOT}/rootfs  
mkdir -p ${PRJ_ROOT}/rootfs  
cd ${PRJ_ROOT}/rootfs
```

Simulamos ser root

```
fakeroot
```

Creamos los directorios esenciales

```
mkdir -m 0700 root  
mkdir -m 0755 bin dev etc lib proc sbin sys usr var  
mkdir -m 0755 usr/bin usr/sbin usr/lib  
mkdir -m 0755 var/lib var/lock var/log var/run var/www  
mkdir -m 0755 etc/init.d
```

Los ficheros temporales sólo podrán ser borrados por quien los haya creado

```
mkdir -m 1777 tmp var/tmp
```

Instalamos los módulos del kernel

```
cd ${PRJ_ROOT}/${KERNEL}  
export ARCH="arm"  
export HOST="${ARCH}-xilinx-linux-gnueabi"  
export CROSS_COMPILE="/opt/Xilinx/SDK/2016.4/gnu/arm/lin/bin/${HOST}-"  
make INSTALL_MOD_PATH=${PRJ_ROOT}/rootfs/ modules_install
```

Copiamos las bibliotecas de glibc

```
file ./busybox  
XILINX_ROOT=/opt/Xilinx  
cp -r ${XILINX_ROOT}/SDK/2016.4/gnu/arm/lin/${HOST}/libc/lib/* \  
${PRJ_ROOT}/rootfs/lib
```

Quitamos la información de depuración de las bibliotecas

```
arm-xilinx-linux-gnueabi-strip ${PRJ_ROOT}/rootfs/lib/*
```

Instalamos busybox en el Root FS

```
cd ${PRJ_ROOT}/sysapps/${BUSYBOX}  
make CONFIG_PREFIX=${PRJ_ROOT}/rootfs install
```


Enlace para el proceso init

```
cd ${PRJ_ROOT}/rootfs
```

```
ln -s bin/busybox init
```

Poblamos el directorio \${PRJ_ROOT}/rootfs/etc

```
vi ${PRJ_ROOT}/rootfs/etc/profile
```

```
# Fijamos el PATH
```

```
PATH=/bin:/sbin:/usr/bin:/usr/sbin
```

```
export PATH
```

```
vi ${PRJ_ROOT}/rootfs/etc/inittab
```

```
# Fijamos /etc/init.d/rcS como fichero de inicializacion del sistema
```

```
::sysinit:/etc/init.d/rcS
```

```
# Iniciamos una sesión de login en la consola
```

```
::respawn:/sbin/getty 115200 ttyPS0
```

```
# Indicamos que se ejecute /sbin/init si init se reinicia
```

```
::restart:/sbin/init
```

```
# Fijamos /etc/init.d/rcK como fichero de apagado del sistema
```

```
::shutdown:/etc/init.d/rcK
```

Poblamos el directorio /etc

```
vi ${PRJ_ROOT}/rootfs/etc/init.d/rcS
```

```
#!/bin/sh
```

```
hostname -F /etc/hostname
```

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

```
mount -t proc none /proc
```

```
mount -t tmpfs none /tmp
```

```
echo "/sbin/mdev" > /proc/sys/kernel/hotplug
```

```
/sbin/mdev -s
```

```
mkdir -p /dev/pts
```

```
mkdir -p /dev/i2c
```

```
mount -t devpts devpts /dev/pts
```

```
ifconfig eth0 down
```

```
ifconfig eth0 192.168.1.10 up
```

```
telnetd -l /bin/sh
```

```
httpd -h /var/www
```

```
tcpsvd 0:21 ftpd ftpd -w /&
```

```
vi ${PRJ_ROOT}/rootfs/etc/init.d/rcK
```

```
#!/bin/sh
```

```
umount -a -r
```

Hacemos que los scripts de inicialización y apagado sean ejecutables

```
chmod a+x etc/init.d/rcS etc/init.d/rcK
```

Nombre del equipo

```
echo "xilinx_zynq_a9" > etc/hostname
```

Mensaje de bienvenida

```
echo "ARM-Linux desde cero \n \l" > etc/issue
```

Fichero passwd

```
echo "root::0:0:root:/root:/bin/sh" > etc/passwd
```

Creamos el archivo cpio

```
find . | cpio --quiet -o -H newc | gzip > ${PRJ_ROOT}/images/rootfs.cpio.gz
```

Dejamos de simular que somos root

```
exit
```

Arranque de la placa y ejecución de Linux

Creación del First Stage Boot Loader mediante el SDK

En el SDK del proyecto, en File, New, Application Project, Zync FSBL con OS Platform: standalone, Board Support Package: Create New. Al crearse la aplicación, agregar en FSBL_DEBUG la línea

```
#define FSBL_DEBUG_INFO
```

Construcción de U-Boot

Variables de entorno

```
UBOOT="u-boot-Digilent-Dev"
```

```
DILIGENT_GIT="https://github.com/DigilentInc"
```

```
UBOOT_DIR="${PRJ_ROOT}/${UBOOT}"
```

Obtención de las fuentes

```
git -C ${PRJ_ROOT} clone -b master-next ${DILIGENT_GIT}/${UBOOT}.git
```

Limpiamos restos de compilaciones anteriores

```
cd ${UBOOT_DIR}
```

```
make distclean
```

Configuramos y construimos

```
make zynq_zybo_config ARQ=arm
```

```
make -j 9
```

Copiamos el ejecutable de U-Boot al directorio de las imágenes

```
cp ${UBOOT_DIR}/u-boot ${PRJ_ROOT}/images/u-boot.elf
```

Preparación de la imagen de arranque

Generamos la imagen de arranque de la placa mediante el SDK, para ello en el SDK del proyecto, en Xilinx Tools, Create Zynq Boot Image, Create new BIF file. Luego en la misma ventana, en Boot image partitions, Add, añadimos U-Boot (u-boot.elf) a la imagen.

Preparación del kernel y el RootFS para ser cargados por U-Boot

Variables de entorno

```
UBOOT="u-boot-Digilent-Dev"
```

```
UBOOT_DIR="${PRJ_ROOT}/${UBOOT}"
```

```
KERNEL="Linux-Digilent-Dev"
```

```
KERNEL_DIR="${PRJ_ROOT}/${KERNEL}"
```

Generamos una uImage del kernel de Linux

```
PATH=$PATH:${UBOOT_DIR}/tools
```

```
cd ${KERNEL_DIR}
```

```
make -j 9 UIIMAGE_LOADADDR=0x8000 uImage
```

```
cp arch/arm/boot/uImage ${PRJ_ROOT}/images
```

Preparamos la imagen del RootFS para que pueda ser cargada por U-Boot

```
cd ${PRJ_ROOT}/images
```

```
mkimage -A arm -T ramdisk -C gzip -d rootfs.cpio.gz uramdisk.image.gz
```

Preparación de la placa

Variables de entorno

```
SDCARD_DIR="/media/username/sdcardname"
```

Configuramos la tarjeta microSD (copiamos a la primera partición)

```
cd ${PRJ_ROOT}/images
```

```
cp boot.bin ${SDCARD_DIR}
```

```
cp ${PRJ_ROOT}/images/uImage ${SDCARD_DIR}

cp ${PRJ_ROOT}/images/uramdisk.image.gz ${SDCARD_DIR}

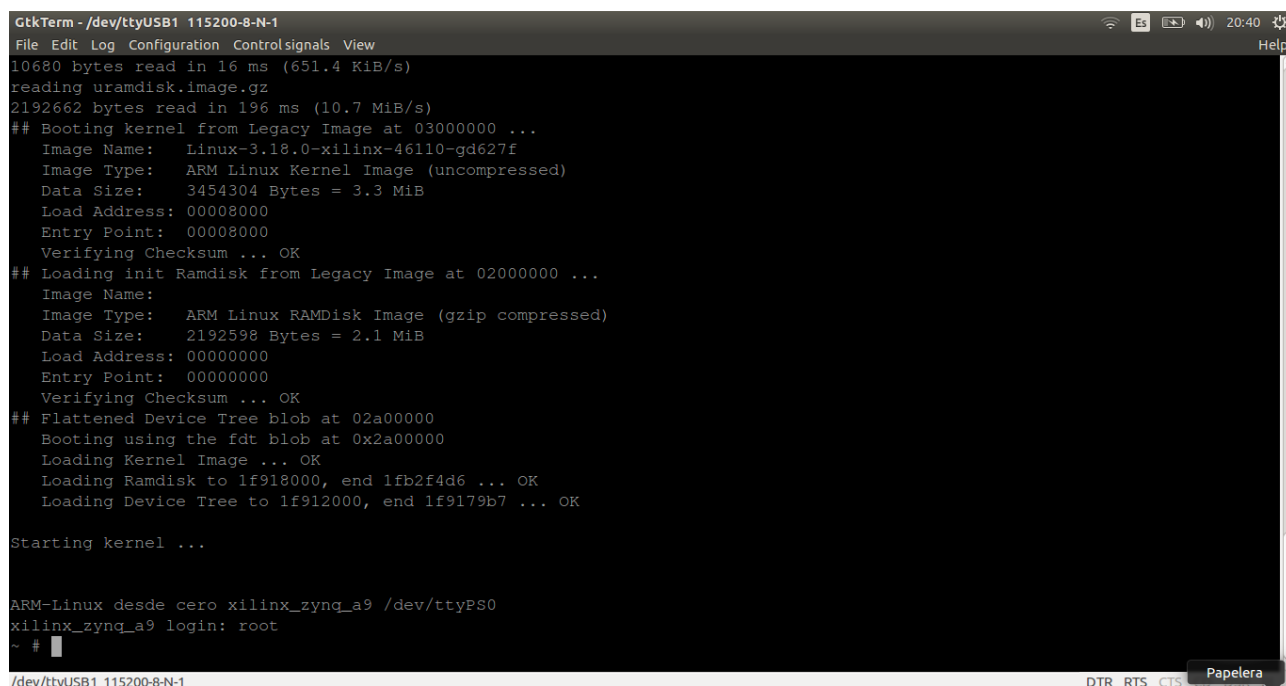
cp ${PRJ_ROOT}/images/devicetree.dtb ${SDCARD_DIR}
```

Encendemos la placa y nos conectamos vía serie, para ello, antes instalamos y configuramos GKTterm

```
sudo apt-get gkterm

sudo gkterm
```

En Configuration, Port, seleccionamos el puerto USB objetivo, y Baud Rate ponemos al máximo



```
GkTerm - /dev/ttyUSB1 115200-8-N-1
File Edit Log Configuration Controlsignals View
10680 bytes read in 16 ms (651.4 KiB/s)
reading uramdisk.image.gz
2192662 bytes read in 196 ms (10.7 MiB/s)
## Booting kernel from Legacy Image at 03000000 ...
  Image Name:   Linux-3.18.0-xilinx-46110-gd627f
  Image Type:   ARM Linux Kernel Image (uncompressed)
  Data Size:    3454304 Bytes = 3.3 MiB
  Load Address: 00008000
  Entry Point:  00008000
  Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 02000000 ...
  Image Name:
  Image Type:   ARM Linux RAMDisk Image (gzip compressed)
  Data Size:    2192598 Bytes = 2.1 MiB
  Load Address: 00000000
  Entry Point:  00000000
  Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
  Booting using the fdt blob at 0x2a00000
  Loading Kernel Image ... OK
  Loading Ramdisk to 1f918000, end 1fb2f4d6 ... OK
  Loading Device Tree to 1f912000, end 1f9179b7 ... OK

Starting kernel ...

ARM-Linux desde cero xilinx_zynq_a9 /dev/ttyPS0
xilinx_zynq_a9 login: root
~ #
```

Inconvenientes

Sólo al empezar al querer crear el proyecto me topé con problemas al generar el Bitstream, la primera instalación del Vivado la había hecho online, por lo que se ve algún archivo se volvió corrupto en el proceso y daba errores. Por lo tanto tuve que volver reinstalar Vivado, esta vez de forma online, con lo que pude solucionar dicho inconveniente.

Para utilizar el CROSS_COMPILE, debemos utilizar el del SDK de Xilinx, Vivado, por lo que hay agregarlo con la ruta completa a la hora de invocarlo, sino no lo encuentra y da errores. También a la hora de hacer make, hay que pasarle como argumento la arquitectura (ARCH=arm).

Otro inconveniente que experimenté, fue que al llenarse la partición de Ubuntu, tuve que mover el proyecto a otra ubicación, al moverlo tuve que rehacerlo, por alguna razón se perdieron las referencias.

Por último al ejecutar el comando siguiente `${KERNEL_DIR}/scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb system.dts`, se debe cambiar `system.dts` por `system-top.dts`.

Opcional

En cuanto a la actividad de crear tareas críticas y no críticas capaces de modificar GPIOs y/o el contenido de una BRAM desde el espacio de usuario, si bien no la desarrollé como tal, pude escribir un simple programa en C, “Hello World”.

Para que el programa funcione en la arquitectura ARM, tenemos que compilarlo para esta, esto lo podemos hacer instalando “gcc”, compilar para arm pero en nuestro ordenador, y añadirlo en el sistema de archivos (generarlo y volcarlo de vuelta al SDCAR) creado para ejecutarlo en el dispositivo (Forray, 2011). Esto se debe a que nuestro Linux es muy ligero y no tiene las facilidades que encontramos en uno estándar.



```
GtkTerm - /dev/ttyUSB1 115200-8-N-1
File Edit Log Configuration Controsignals View
Data Size: 3454304 Bytes = 3.3 MiB
Load Address: 00008000
Entry Point: 00008000
Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 02000000 ...
Image Name:
Image Type: ARM Linux RAMDisk Image (gzip compressed)
Data Size: 2197926 Bytes = 2.1 MiB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
Booting using the fdt blob at 0x2a00000
Loading Kernel Image ... OK
Loading Ramdisk to 1f917000, end 1fb2f9a6 ... OK
Loading Device Tree to 1f911000, end 1f9169b7 ... OK

Starting kernel ...

ARM-Linux desde cero xilinx_zynq_a9 /dev/ttyPS0
xilinx_zynq_a9 login: root
~ # ls
app
~ # cd app
~/app # ls
hello hello.c hello.o noARM
~/app # ./hello
Hola Mundo!~/app #
```

Si quisiéramos crear tareas críticas y no críticas capaces de modificar GPIOs y/o el contenido de una BRAM desde el espacio de usuario, podríamos escribir otro programa en C, valernos de la función nmap, identificando las posiciones en memoria (en el device_tree o en el SDK del proyecto creado anteriormente) y prender o apagar un led por ejemplo.

Conclusiones

En este trabajo pude experimentar con elementos que no había utilizado con anterioridad, aunque hubieron algunos inconvenientes, la formación recibida me pareció de lo más interesante, el poder montar Linux desde cero y de una forma casi personalizada lo valoro mucho, incluso el poder ejecutar un programa desde el espacio de usuario.

Bibliografía

Forray, J. P. (2011, julio 5). Cross Compiler ARM en ubuntu x86. Recuperado 7 de mayo de 2017, a partir de <https://pforray.wordpress.com/2011/07/05/cross-compiler-arm-en-ubuntu-x86/>

González Peñalver, J. (2015). Selección y configuración de un sistema operativo. Universidad de Granada.

Anexos

Archivo comprimido (OK.zip) con lo necesario para arrancar el sistema: boot.bin, uImage, uramdisk y devicetree.dtb.

SDK del proyecto comprimido.