

Laboratorio: Compilación Cruzada

Autor: Víctor Hugo García Ortega

La compilación cruzada consiste en compilar, en una arquitectura, un programa para generar el archivo ejecutable para otra arquitectura diferente a la del procesador donde se compila.

Por ejemplo: Podemos utilizar una computadora de escritorio o portátil con una arquitectura x86 (con procesador INTEL o AMD) y compilar un programa en lenguaje C para un procesador con una arquitectura ARM. Esta es la arquitectura, ARM, en la que se basan los sistemas SoC de Raspberry.

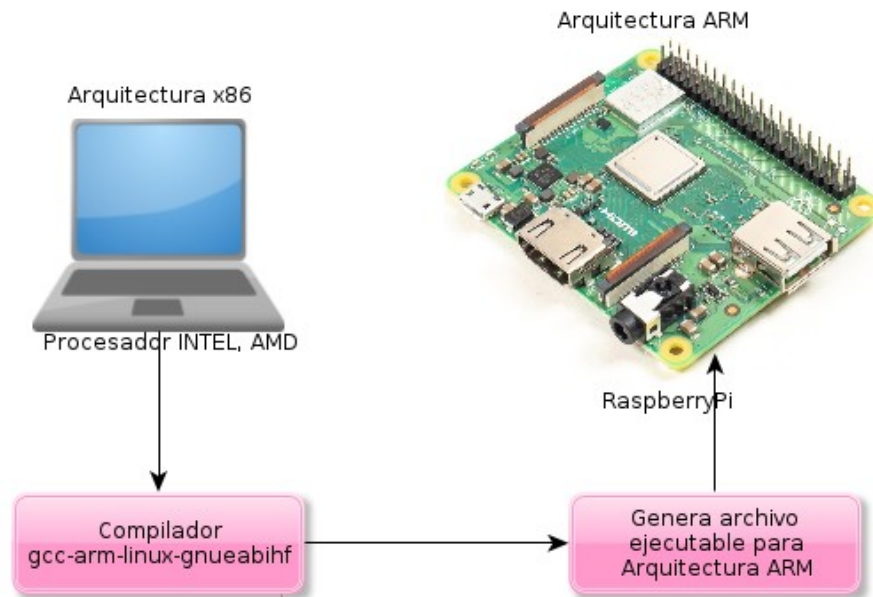


Figura 1: Compilación cruzada

1. Instalación de paquetes

Primero se deben instalar todos los paquetes necesarios para la compilación cruzada.

```
$ sudo apt-get install git bc bison flex libssl-dev make libc6-dev libncurses5-dev lib32z1
```

2. Instalación de la herramienta de compilación

Se debe instalar el compilador de lenguaje C para la arquitectura ARM, llamado "gcc-arm-linux-gnueabi", y sus herramientas, a todo eso se le conoce como "Toolchain". Esto se hace clonando el repositorio de herramientas de RaspberryPi en nuestro "home".

```
$ git clone https://github.com/raspberrypi/tools ~/tools
```

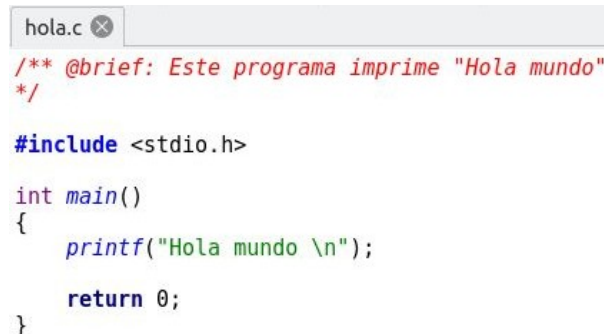
Se debe actualizar de la variable de entorno \$PATH, esto hace que el sistema conozca las ubicaciones de archivos necesarias para la compilación cruzada.

```
$ echo PATH=$PATH:~/tools/arm-bcm2708/arm-linux-gnueabi/bin >> ~/.bashrc  
$ source ~/.bashrc
```

3. Probando la herramienta de compilación.

En este momento podemos compilar un programa en lenguaje C en una computadora con arquitectura x86 y generar el archivo ejecutable para una arquitectura ARM.

Escribir el programa:

A screenshot of a code editor window titled 'hola.c'. The code is written in C and includes a comment, a header inclusion, and a main function that prints 'Hola mundo' and returns 0.

```
hola.c
/** @brief: Este programa imprime "Hola mundo"
 */

#include <stdio.h>

int main()
{
    printf("Hola mundo \n");

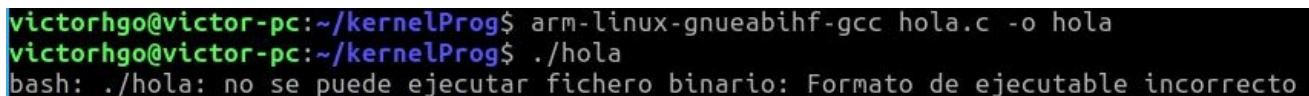
    return 0;
}
```

Figura 2: Programa de prueba

Compilar usando el “toolchain” instalado:

```
$ arm-linux-gnueabi-gcc hola.c -o hola
```

Si se quiere correr el programa ejecutable obtenido después de la compilación cruzada en una arquitectura x86 se obtiene el siguiente error:

A screenshot of a terminal window showing the execution of the hola program on an x86 architecture. The user runs the compiler to create the executable and then runs the executable, which results in an error message.

```
victorhgo@victor-pc:~/kernelProg$ arm-linux-gnueabi-gcc hola.c -o hola
victorhgo@victor-pc:~/kernelProg$ ./hola
bash: ./hola: no se puede ejecutar fichero binario: Formato de ejecutable incorrecto
```

Figura 3: Ejecución del programa de prueba en la arquitectura x86

4. Obteniendo el código fuente de la imagen del kernel de RaspberryPi.

Para realizar este paso clonamos el repositorio:

```
$ git clone --depth=1 https://github.com/raspberrypi/linux
```

Si se omite `--depth=1` se descargará todo el repositorio, incluido el historial completo de todas las ramas, esto llevará mucho más tiempo para descargar el repositorio y ocupará más espacio de almacenamiento.

La forma tradicional de configurar el kernel de linux es usando su menu de configuración el cual se obtiene con su archivo `makefile`.

```
$ cd linux
```

```
$ make menuconfig
```

Con este menú podemos configurar las opciones que se necesitan para optimizar el kernel de Linux. En la parte superior podemos ver la versión del kernel a compilar, 4.19.127.

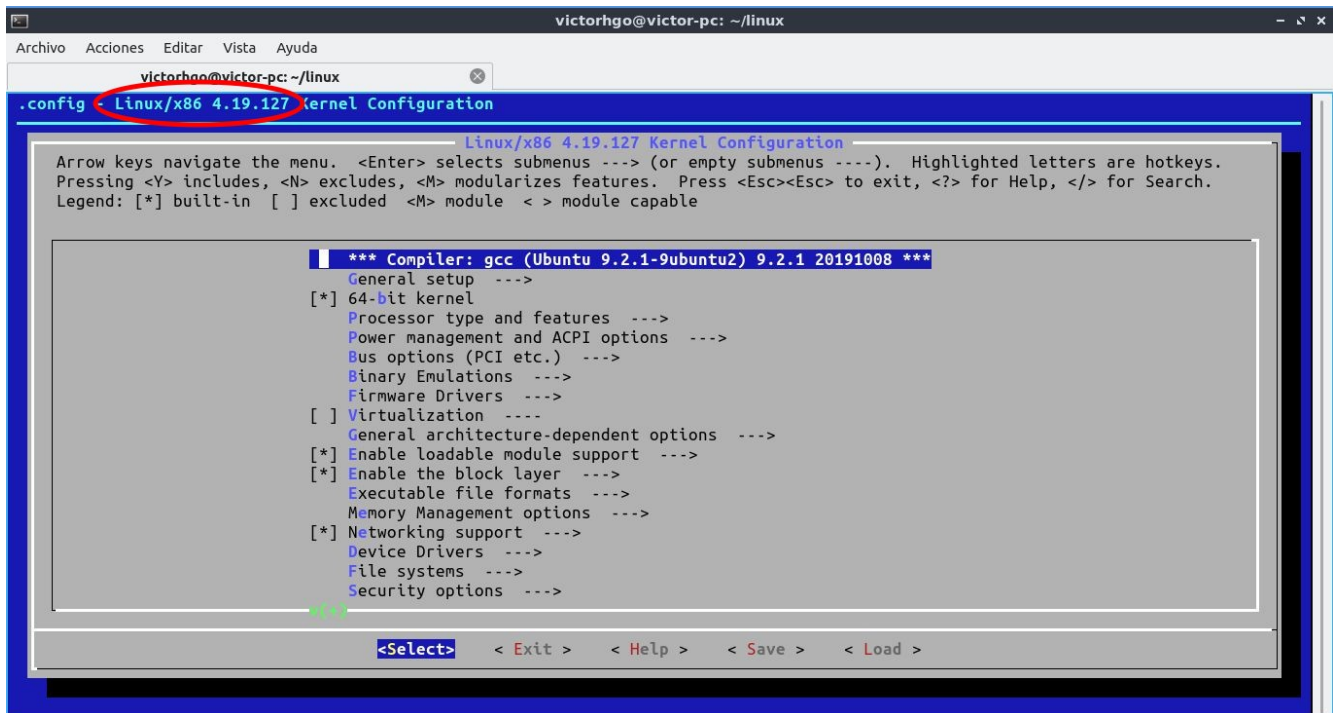


Figura 4: Menu de configuración del kernel

Otra forma para realizar la configuración del kernel para la arquitectura ARM es ejecutar el archivo de configuración ubicado en:

```
$~/linux/arch/arm/configs/bcm2709_defconfig
```

Para las versiones Pi 1, Pi Zero, Pi Zero W, o Módulo de Computo:

```
$ cd linux
$ KERNEL=kernel
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcmrpi_defconfig
```

Para las versiones Pi 2, Pi 3, Pi 3A+, Pi 3B+ o Módulo de Computo 3:

```
$ cd linux
$ KERNEL=kernel7
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

Para la versión Pi 4:

```
$ cd linux
$ KERNEL=kernel7l
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig
```

La variable KERNEL es usada en la sección 7.

5. Reconstruyendo el kernel.

Para reconstruir el kernel del Soc RaspberryPi y actualizarlo a su última versión con todos sus drivers disponibles ejecutamos:

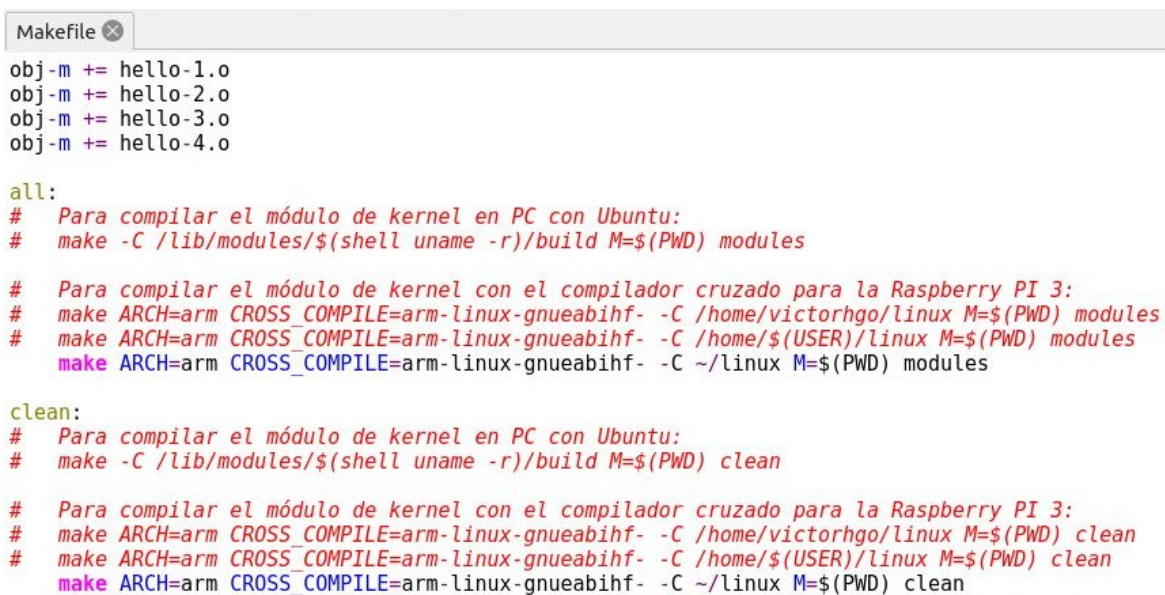
```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

Podemos agregar -jn, donde “n” es el número de procesadores. Con esto la compilación se paralelizará y se realizará mas rápido.

```
$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

6. Realizando compilación cruzada de módulos de kernel.

Para hacer compilación cruzada de módulos de kernel debemos usar el compilador para la arquitectura ARM, “gcc-arm-linux-gnueabihf”, y la herramienta de construcción kbuild del código fuente del kernel de linux para ARM. En los pasos anteriores ya se hizo eso, por lo que procedemos a modificar el archivo Makefile.



```
Makefile
obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o

all:
# Para compilar el módulo de kernel en PC con Ubuntu:
# make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

# Para compilar el módulo de kernel con el compilador cruzado para la Raspberry PI 3:
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /home/victorhgo/linux M=$(PWD) modules
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /home/$(USER)/linux M=$(PWD) modules
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C ~/linux M=$(PWD) modules

clean:
# Para compilar el módulo de kernel en PC con Ubuntu:
# make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

# Para compilar el módulo de kernel con el compilador cruzado para la Raspberry PI 3:
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /home/victorhgo/linux M=$(PWD) clean
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /home/$(USER)/linux M=$(PWD) clean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C ~/linux M=$(PWD) clean
```

Figura 5: Archivo Makefile para compilación cruzada de módulos de kernel

Con este archivo Makefile ya se pueden compilar los módulos de kernel para ARM desde una arquitectura x86, sin embargo, para ejecutarlos en el SoC, la versión del kernel que se compiló en el paso 5 debe ser la misma que la que tiene el SoC donde se desean insertar los módulos de kernel. La versión de kernel se obtiene con:

```
$ uname -r
```

7. Actualizando el kernel de linux en el SoC RaspberryPi.

Una vez que el kernel de linux se encuentra compilado se debe instalar en la memoria SD donde tenemos instalado el Sistema operativo del Soc RaspberryPi. Primero introducimos la memoria SD a nuestra computadora personal. Ubicamos las particiones de la memoria SD en /dev.

```
$ lsblk
```

```
victorhgo@victor-pc:~$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda         8:0    0  477G  0 disk
├─sda1      8:1    0   512M  0 part /boot/efi
├─sda2      8:2    0 468.5G  0 part /
└─sda3      8:3    0    7.9G  0 part [SWAP]
sdb         8:16   1   59.4G  0 disk
├─sdb1      8:17   1   256M  0 part /media/victorhgo/boot
└─sdb2      8:18   1   59.2G  0 part /media/victorhgo/rootfs
sr0        11:0    1 1024M  0 rom
```

Figura 6: Particiones de la memoria SD

Debemos identificar las particiones, en este caso:

sdb1 es boot
sdb2 es rootfs

Procedemos a desmontar las particiones de la memoria SD:

```
$ sudo umount /dev/sdb1
$ sudo umount /dev/sdb2
$ lsblk
```

```
victorhgo@victor-pc:~$ sudo umount /dev/sdb1
[sudo] contraseña para victorhgo:
victorhgo@victor-pc:~$ sudo umount /dev/sdb2
victorhgo@victor-pc:~$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda         8:0    0  477G  0 disk
├─sda1      8:1    0   512M  0 part /boot/efi
├─sda2      8:2    0 468.5G  0 part /
└─sda3      8:3    0    7.9G  0 part [SWAP]
sdb         8:16   1   59.4G  0 disk
├─sdb1      8:17   1   256M  0 part
└─sdb2      8:18   1   59.2G  0 part
sr0        11:0    1 1024M  0 rom
```

Figura 7: Particiones de la memoria SD desmontadas

Después las montamos en dos subdirectorios creados para el boot y el root:

```
$ cd linux
~/linux$ mkdir mnt
~/linux$ mkdir mnt/boot
~/linux$ mkdir mnt/rootfs
~/linux$ sudo mount /dev/sdb1 mnt/boot
~/linux$ sudo mount /dev/sdb2 mnt/rootfs
~/linux$ lsblk
```



```

victorhgo@victor-pc:~$ cd linux
victorhgo@victor-pc:~/linux$ mkdir mnt
victorhgo@victor-pc:~/linux$ mkdir mnt/boot
victorhgo@victor-pc:~/linux$ mkdir mnt/rootfs
victorhgo@victor-pc:~/linux$ sudo mount /dev/sdb1 mnt/boot/
victorhgo@victor-pc:~/linux$ sudo mount /dev/sdb2 mnt/rootfs/
victorhgo@victor-pc:~/linux$ lsblk

```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
sda	8:0	0	477G	0	disk	
├─sda1	8:1	0	512M	0	part	/boot/efi
├─sda2	8:2	0	468.5G	0	part	/
└─sda3	8:3	0	7.9G	0	part	[SWAP]
sdb	8:16	1	59.4G	0	disk	
├─sdb1	8:17	1	256M	0	part	/home/victorhgo/linux/mnt/boot
└─sdb2	8:18	1	59.2G	0	part	/home/victorhgo/linux/mnt/rootfs
sr0	11:0	1	1024M	0	rom	

```

victorhgo@victor-pc:~/linux$

```

Figura 8: Particiones de la memoria SD montadas en las carpetas de mnt

Instalamos todos los módulos compilados del kernel

```
~/linux$ sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- INSTALL_MOD_PATH=mnt/rootfs modules_install
```

```

INSTALL sound/soc/codecs/snd-soc-pcm179x-i2c.ko
INSTALL sound/soc/codecs/snd-soc-pcm186x-i2c.ko
INSTALL sound/soc/codecs/snd-soc-pcm186x.ko
INSTALL sound/soc/codecs/snd-soc-pcm5102a.ko
INSTALL sound/soc/codecs/snd-soc-pcm512x-i2c.ko
INSTALL sound/soc/codecs/snd-soc-pcm512x.ko
INSTALL sound/soc/codecs/snd-soc-sgtl5000.ko
INSTALL sound/soc/codecs/snd-soc-sigmads-i2c.ko
INSTALL sound/soc/codecs/snd-soc-sigmads.ko
INSTALL sound/soc/codecs/snd-soc-spdif-rx.ko
INSTALL sound/soc/codecs/snd-soc-spdif-tx.ko
INSTALL sound/soc/codecs/snd-soc-tas5713.ko
INSTALL sound/soc/codecs/snd-soc-tlv320aic32x4-i2c.ko
INSTALL sound/soc/codecs/snd-soc-tlv320aic32x4.ko
INSTALL sound/soc/codecs/snd-soc-wm-adsp.ko
INSTALL sound/soc/codecs/snd-soc-wm5102.ko
INSTALL sound/soc/codecs/snd-soc-wm8731.ko
INSTALL sound/soc/codecs/snd-soc-wm8741.ko
INSTALL sound/soc/codecs/snd-soc-wm8804-i2c.ko
INSTALL sound/soc/codecs/snd-soc-wm8804.ko
INSTALL sound/soc/generic/snd-soc-audio-graph-card.ko
INSTALL sound/soc/generic/snd-soc-simple-card-utils.ko
INSTALL sound/soc/generic/snd-soc-simple-card.ko
INSTALL sound/soc/snd-soc-core.ko
INSTALL sound/usb/6fire/snd-usb-6fire.ko
INSTALL sound/usb/caiaq/snd-usb-caiaq.ko
INSTALL sound/usb/hiface/snd-usb-hiface.ko
INSTALL sound/usb/misc/snd-ua101.ko
INSTALL sound/usb/snd-usb-audio.ko
INSTALL sound/usb/snd-usbmidi-lib.ko
DEPMOD 4.19.127-v7+
victorhgo@victor-pc:~/linux$

```

Figura 9: Instalación de módulos del kernel

Copiamos la imagen del kernel de linux generada y los archivos de dispositivo:

Aquí usamos la variable KERNEL de la sección 4.

```

~/linux$ sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
~/linux$ sudo cp arch/arm/boot/zImage mnt/boot/$KERNEL.img
~/linux$ sudo cp arch/arm/boot/dts/*.dtb mnt/boot/
~/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/boot/overlays/
~/linux$ sudo cp arch/arm/boot/dts/overlays/README mnt/boot/overlays/
~/linux$ sudo umount mnt/boot
~/linux$ sudo umount mnt/rootfs

```

```

victorhgo@victor-pc:~/linux$ sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
victorhgo@victor-pc:~/linux$ sudo cp arch/arm/boot/zImage mnt/boot/$KERNEL.img
victorhgo@victor-pc:~/linux$ sudo cp arch/arm/boot/dts/*.dtb mnt/boot/
victorhgo@victor-pc:~/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/boot/overlays/
victorhgo@victor-pc:~/linux$ sudo cp arch/arm/boot/dts/overlays/README mnt/boot/overlays/
victorhgo@victor-pc:~/linux$ sudo umount mnt/boot
victorhgo@victor-pc:~/linux$ sudo umount mnt/rootfs
victorhgo@victor-pc:~/linux$

```

Listo!!!, ya solo tenemos que insertar nuestra memoria SD en el SoC de Raspberry y verificar la versión del kernel.

```

victorhgo@victor-pc:~/linux$ ssh pi@192.168.100.200
pi@192.168.100.200's password:
Linux raspberrypi 4.19.127-v7+ #2 SMP Tue Jul 7 02:13:20 CDT 2020 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 7 23:28:41 2020
pi@raspberrypi:~ $ uname -r
4.19.127-v7+
pi@raspberrypi:~ $

```

Figura 10: Kernel actualizado en la RaspberryPi

Referencias

- [1] RaspberryPi, "Kernel Building",
<https://www.raspberrypi.org/documentation/linux/kernel/building.md>, 2020