When compiling a module the -C parameter should point to the source tree where the kernel was built (don't clean it up!). If you built it on the pi its likely in a directory under your home directory.

The build directory under /lib/modules/<version> is a Debian-ism, where a cut-down version of the source tree is provided with just enough context to build modules against. The kernels from the Raspberry Pi Foundation kernels don't ship with a build directory.

They may be a bit out of date, but raspbian provides a kernel as a Debian-style package, which should include the build directory you could use to build kernel modules against.

sudo aptitude install linux-image-rpi-rpfv linux-headers-rpi-rpfv

12

Here are the steps I used to build the [Hello World kernel module](https://github.com/notro/rpi-source/wiki/Examples-on-how-to-build-various-modules" \l "hello-world-example) on Raspbian.

1. Perform sudo rpi-update

See <https://github.com/Hexxeh/rpi-update> for details on rpi-update. You have to be on the latest firmware and associated kernel to be able to perform the next step.

1. Install and run rpi-source to install the source code that built the latest kernel that you are running. This will create the correct entry in /lib/modules for the kernel that you are running. Note: you don't need to be root to run this, however the script will perform certain tasks using sudo and the root password will be requested during the script execution.

Instructions to install rpi-source can be found at <https://github.com/notro/rpi-source/wiki>

Once those steps are performed you should be able to make the Hello World kernel module.

johnma@raspberrypi ~/HelloWorld $ make

make -C /lib/modules/3.12.19+/build M=/home/johnma/HelloWorld modules

make[1]: Entering directory `/home/johnma/linux-c3db7205bcd8988cf7c185e50c8849542554b1f5'

CC [M] /home/johnma/HelloWorld/hello.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/johnma/HelloWorld/hello.mod.o

LD [M] /home/johnma/HelloWorld/hello.ko

make[1]: Leaving directory `/home/johnma/linux-c3db7205bcd8988cf7c185e50c8849542554b1f5'

johnma@raspberrypi ~/HelloWorld $ sudo insmod hello.ko

johnma@raspberrypi ~/HelloWorld $ tail -1 /var/log/syslog

May 15 13:45:39 raspberrypi kernel: [59789.169461] Hello World :)

johnma@raspberrypi ~/HelloWorld $ sudo rmmod hello.ko

johnma@raspberrypi ~/HelloWorld $ tail -1 /var/log/syslog

May 15 13:46:10 raspberrypi kernel: [59819.503503] Goodbye World!

[share](https://stackoverflow.com/a/23685353)[improve this answer](https://stackoverflow.com/posts/23685353/edit)

answered May 15 '14 at 17:51

[HeatfanJohn](https://stackoverflow.com/users/1411277/heatfanjohn)

4,85822238

add a comment

9

You first need kernel headers (and the corresponding kernel binary) to build your module.   
Like Greg said, the raspbian distribution provides the packages :

sudo apt-get install linux-image-rpi-rpfv linux-headers-rpi-rpfv

Then, tell raspbian to boot your newly installed kernel (3.10-3-rpi for me).   
Append this at end of /boot/config.txt and reboot your Pi :

# Parameters to boot on raspbian kernel (linux-image-rpi-rpfv package)

kernel=vmlinuz-3.10-3-rpi

initramfs initrd.img-3.10-3-rpi followkernel

Then, modify your Makefile to point the freshly installed kernel headers :

KERNEL\_HEADERS=/lib/modules/$(shell uname -r)/build

obj-m := hello-1.o

all:

@$(MAKE) -C $(KERNEL\_HEADERS) M=$(PWD) modules

clean:

@$(MAKE) -C $(KERNEL\_HEADERS) M=$(PWD) clean

[share](https://stackoverflow.com/a/22882969)[improve this answer](https://stackoverflow.com/posts/22882969/edit)

[edited Apr 6 '14 at 9:46](https://stackoverflow.com/posts/22882969/revisions)

answered Apr 5 '14 at 15:39



[JayDee](https://stackoverflow.com/users/2488796/jaydee)

9113

add a comment

8

This was a pain. I had to compile and install a kernel mode driver.After long search, i got the headers for pi 2 (3.18.7-v7+) from [here](http://www.niksula.hut.fi/~mhiienka/Rpi/linux-headers-rpi/).

sudo apt-get install dkms build-essential

wget http://www.niksula.hut.fi/~mhiienka/Rpi/linux-headers-rpi/linux-headers-3.18.7-v7%2b\_3.18.7-v7%2b-2\_armhf.deb

sudo dpkg -i linux-headers-3.18.7-v7+\_3.18.7-v7+-2\_armhf.deb

[share](https://stackoverflow.com/a/29354565)[improve this answer](https://stackoverflow.com/posts/29354565/edit)

answered Mar 30 '15 at 19:26



[Ryu\_hayabusa](https://stackoverflow.com/users/1658975/ryu-hayabusa)

2,05311925

add a comment

0

I was working on the exact same sample on my RPI with the exact same kernel. I managed to compile the module on my RPI but when I issued insmod I received an error. I followed the instructions [here](http://bchavez.bitarmory.com/archive/2013/01/16/compiling-kernel-modules-for-raspberry-pi.aspx) on an XUbuntu virtual machine (using my RPI's kernel version 3.6.y) and it worked perfectly. Not sure why compiling directly on the RPI did not work, that will be a problem for another day.

I had to change the Makefile to match the new environment.

obj-m += hello-1.o

all:

make ARCH=arm CROSS\_COMPILE=${CCPREFIX} -C /home/cstick/rpi/linux-rpi-3.6.y M=$(PWD) modules

clean:

make -C /home/cstick/rpi/linux-rpi-3.6.y M=$(PWD) clean

[share](https://stackoverflow.com/a/20269744)[improve this answer](https://stackoverflow.com/posts/20269744/edit)

[edited Nov 29 '13 at 2:33](https://stackoverflow.com/posts/20269744/revisions)

answered Nov 28 '13 at 15:25

[cstick](https://stackoverflow.com/users/2735/cstick)

376310

add a comment

0

I meet with the same problem and just fix it by sudo apt-get install raspberrypi-kernel-headers

On m’a demandé à plusieurs reprises comment compiler les modules pour le noyau du Raspberry Pi que j’ai présenté dans différents articles ou ceux que je propose en illustration de mes sessions de formation. J’emploie généralement une cross-compilation, c’est à dire un compilateur spécifique installé sur un PC pour produire du code pour le processeur à cœur ARM du Raspberry Pi. J’ai déjà présenté cette solution dans plusieurs articles (par exemple [celui-ci](https://www.blaess.fr/christophe/2012/10/12/toolchain-buildroot-pour-raspberry-pi/), [celui-ci](https://www.blaess.fr/christophe/2012/10/19/toolchain-crosstool-ng-pour-raspberry-pi/) ou [la série d’articles](https://www.blaess.fr/christophe/2013/09/10/raspberry-pi-from-scratch-2/) pour Linux Magazine).

Il existe toutefois une autre solution plus simple : la compilation native en utilisant une distribution courante pour Raspberry Pi.

# Problématique

Pour pouvoir compiler un module du noyau, il est indispensable de disposer des fichiers d’en-tête et de configuration du noyau cible. Pour une plate-forme embarquée, cela signifie généralement que l’on réalise la compilation des modules supplémentaires sur la même machine que celle qui a servi à produire le noyau.

Lorsque le noyau précompilé est livré avec le matériel, la mise en œuvre de modules personnalisés peut devenir un véritable casse-tête pour l’intégrateur qui se retrouve pris en sandwich entre différents fournisseurs. C’est pour cela que je préconise toujours d’essayer de disposer de toute la solution de compilation lors de l’industrialisation d’un système embarqué.

Quand nous souhaitons compiler un nouveau module pour un noyau fourni par une distribution Linux, par exemple sur un PC, aucun souci : il suffit d’installer le package contenant les fichiers d’en-tête et de configuration nécessaires (linux-headers sur Debian et dérivés, kernel-devel sur RedHat, etc.) et d’utiliser les outils de compilation standards.

Le problème qui se pose avec les principales distributions pour le Raspberry Pi, notamment la Raspbian que je prendrai en exemple dans cet article, est qu’elles ne fournissent pas les packages nécessaires à la compilation des modules. Il est possible d’installer des packages “alternatifs” fournis par des sources non-officielles, mais tant qu’à faire, autant re-générer le noyau et disposer d’un environnement de compilation de modules parfaitement sain. Cela va prendre un temps de calcul assez conséquent (une dizaine d’heures), mais est à la portée de n’importe quel utilisateur du Raspberry Pi.

# Compilation d’un noyau pour Raspbian

J’ai téléchargé la dernière version de [la distribution Raspbian](http://www.raspbian.org/) disponible à ce jour (2014-01-07-wheezy), mais je pense que les manipulations décrites ci-dessous doivent être réalisables sur toute distribution qui propose l’ensemble des outils de compilation (make, gcc, etc.) natifs sur le Raspberry Pi.

## Téléchargement des sources modifiées pour Raspberry Pi

Tout d’abord, récupérons les sources de Linux incluant les patches spécifiques pour le Raspberry. Attention, il faudra disposer d’environ 2 Go de libre sur la carte SD pour la compilation du noyau. Il peut donc être nécessaire de faire un peu de ménage avant de commencer. Personnellement, j’ai retiré les packages de l’environnement X-Window car sur mon Raspberry Pi de test, je ne me connecte qu’en liaison série ou par le réseau (avec ssh).

Le téléchargement s’effectue ainsi :

[~]$ git clone http://github.com/raspberrypi/linux rpi-kernel

[...]

[~]$

Le transfert (1.6 Go) prend environ une heure.

[~]$ cd rpi-kernel

[rpi-kernel]$

## Configuration du noyau

Vérifions le numéro de version du noyau téléchargé.

[rpi-kernel]$ head Makefile

VERSION = 3

PATCHLEVEL = 10

SUBLEVEL = 32

[...]

[rpi-kernel]$ uname -r

3.10.25+

[rpi-kernel]$

Le noyau téléchargé et celui fourni par Raspbian sont tous les deux de la branche 3.10. Il est donc possible d’utiliser la configuration de notre noyau courant pour compiler le nouveau.

[rpi-kernel]$ zcat /proc/config.gz > ./.config

[rpi-kernel]$

Une autre solution serait de prendre la configuration par défaut pour le Raspberry Pi avec :

[rpi-kernel]$ make bcmrpi\_defconfig

[...]

[rpi-kernel]$

Pour observer et ajuster éventuellement la configuration (notamment en modifiant General Setup -> Local version pour ajouter nos initiales), il nous faut installer le package ncurses-devel afin de profiter des menus semi-graphiques. Toutefois, ce package porte un nom différent dans les distributions Debian/Raspbian.

[rpi-kernel]$ sudo apt-get install libncurses5-dev

[...]

[rpi-kernel]$ make menuconfig

## Compilation

La compilation nécessite l’utilitaire bc non installé par défaut sur Raspbian. Ajoutons-le :

[rpi-kernel]$ sudo apt-get install bc

[...]

[rpi-kernel]$

Une fois la configuration fignolée, on lance la compilation.

[rpi-kernel]$ make

Comptez entre 8 et 10 heures de compilation en fonction de ce que vous avez ajouté ou retiré du noyau, ainsi que de la rapidité de votre carte SD…

## Installation

Le lendemain matin, nous finalisons l’installation.

[rpi-kernel]$ sudo make modules\_install

[...]

On voit alors apparaître l’arborescence /lib/modules/3.10.32 contenant les modules du kernel.

Puis installons le nouveau noyau en prenant garde de conserver le précédent au cas où…

[rpi-kernel]$ sudo cp /boot/kernel.img /boot/kernel.bkup

[rpi-kernel]$ sudo cp arch/arm/boot/zImage /boot/kernel.img

[rpi-kernel]$ sudo reboot

# Essai de l’environnement de compilation

Après avoir redémarré, nous vérifions que le noyau soit bien celui que nous avons généré.

[rpi-kernel]$ uname -r

3.10.32+

[rpi-kernel]$

## Compilation d’un module personnalisé

Copions le fichier suivant dans notre répertoire personnel sur le Raspberry Pi. Il s’agit d’une sorte de “Hello World” pour le noyau, c’est le premier exemple que je présente dans mes formations sur la programmation de drivers Linux.

[khello.c](https://www.blaess.fr/christophe/files/article-2014-03-06/khello.c):

#include <linux/module.h>

static int \_\_init khello\_init (void)

{

printk(KERN\_INFO "%s : HELLO\n", THIS\_MODULE->name);

return 0;

}

static void \_\_exit khello\_exit (void)

{

printk(KERN\_INFO "%s : BYE!\n", THIS\_MODULE->name);

}

module\_init(khello\_init);

module\_exit(khello\_exit);

MODULE\_LICENSE("GPL");

Dans le même répertoire, copions le Makefile suivant. Attention, il est important de respecter les tabulations en début de ligne, il faut s’assurer qu’elles ne sont pas remplacées par des espaces.

[Makefile](https://www.blaess.fr/christophe/files/article-2014-03-06/Makefile):

ifneq (${KERNELRELEASE},)

obj-m = khello.o

else

KERNEL\_DIR ?= /lib/modules/$(shell uname -r)/build

MODULE\_DIR := $(shell pwd)

.PHONY: all

all: modules

.PHONY:modules

modules:

${MAKE} -C ${KERNEL\_DIR} SUBDIRS=${MODULE\_DIR} modules

clean:

rm -f \*.o \*.ko \*.mod.c .\*.o .\*.ko .\*.mod.c .\*.cmd \*~

rm -f Module.symvers Module.markers modules.order

rm -rf .tmp\_versions

endif

Nous pouvons à présent compiler notre module.

[~]$ make

make -C /lib/modules/3.10.32-perso+/build SUBDIRS=/home/pi modules

make[1]: Entering directory `/home/pi/rpi-kernel'

CC [M] /home/pi/khello.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/pi/khello.mod.o

LD [M] /home/pi/khello.ko

make[1]: Leaving directory `/home/pi/rpi-kernel'

[~]$ ls

khello.c khello.mod.c khello.o modules.order rpi-kernel

khello.ko khello.mod.o Makefile Module.symvers

[~]$

## Essai du module

Le module est bien présent, chargeons-le:

[~]$ sudo insmod khello.ko

[~]$ lsmod

Module Size Used by

khello 683 0

snd\_bcm2835 16118 0

snd\_soc\_bcm2708\_i2s 5462 0

regmap\_mmio 2778 1 snd\_soc\_bcm2708\_i2s

snd\_soc\_core 132437 1 snd\_soc\_bcm2708\_i2s

[...]

[~]$

Le module est bien chargé dans le kernel, vérifions si son message d’initialisation est visible.

[~]$ dmesg |tail

[ 3.029926] smsc95xx v1.0.4

[ 3.108995] smsc95xx 1-1.1:1.0 eth0: register 'smsc95xx' at usb-bcm2708\_usb-1.1, smsc95xx USB 2.0 Ethernet, b8:27:eb:c9:e5:8d

[ 3.953891] udevd[157]: starting version 175

[ 6.215100] bcm2708-i2s bcm2708-i2s.0: Failed to create debugfs directory

[ 11.279103] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)

[ 11.765049] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)

[ 20.017728] smsc95xx 1-1.1:1.0 eth0: hardware isn't capable of remote wakeup

[ 21.550910] smsc95xx 1-1.1:1.0 eth0: link up, 100Mbps, full-duplex, lpa 0x45E1

[ 24.950826] Adding 102396k swap on /var/swap. Priority:-1 extents:1 across:102396k SSFS

[ 5697.509567] khello : HELLO

[~]$

Nous allons retirer le module de la mémoire du kernel.

[~]$ sudo rmmod khello

pi@raspberrypi:~$ dmesg |tail

[ 3.108995] smsc95xx 1-1.1:1.0 eth0: register 'smsc95xx' at usb-bcm2708\_usb-1.1, smsc95xx USB 2.0 Ethernet, b8:27:eb:c9:e5:8d

[ 3.953891] udevd[157]: starting version 175

[ 6.215100] bcm2708-i2s bcm2708-i2s.0: Failed to create debugfs directory

[ 11.279103] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)

[ 11.765049] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)

[ 20.017728] smsc95xx 1-1.1:1.0 eth0: hardware isn't capable of remote wakeup

[ 21.550910] smsc95xx 1-1.1:1.0 eth0: link up, 100Mbps, full-duplex, lpa 0x45E1

[ 24.950826] Adding 102396k swap on /var/swap. Priority:-1 extents:1 across:102396k SSFS

[ 5697.509567] khello : HELLO

[ 5760.410545] khello : BYE!

[~]$

# Conclusion

Nous voyons qu’il est tout à fait possible d’utiliser un Raspberry Pi pour réaliser des compilations natives de modules du noyau. Cela réclame un temps de calcul important, mais rien ne nous oblige à rester à proximité. L’intérêt est alors de disposer d’une petite plate-forme embarquée conviviale (grâce à l’ensemble des applications fournies par la distribution) que nous pouvons utiliser pour développer facilement nos propres modules noyau.