**GPIO kernel module for arm linux**

I want to build a kernel module that will toggle in a loop a gpio.

Instead doing it using the sysfs.

I would like to start from a Makefile and from the environment to compile and load our module.

The boards are raspberry pi 3 or beaglebone

The compile have the buildroot compiler already installed in output/host/usr/bin

arm-linux-gnueabihf=

**The Makefile for arm linux.**

obj-m := gpiok.o

KERNELDIR := /home/adsteam/buildroot/BBB/buildroot-2017.05.1/output/build/linux-c17b3de1577dd4e223ee6cc923a1d22b3e1e261b

all:

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- -C $(KERNELDIR) M=$(PWD) modules

clean:

rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions

Notes:

The KERNELDIR is from buildroot. So the long hash number is the commit version of the linux kernel.

**A X68 Makefile:**

obj-m := gpiok.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

We will start from simple module to more advanced:

#include <linux/module.h> // included for all kernel modules

#include <linux/kernel.h> // included for KERN\_INFO

#include <linux/init.h> // included for \_\_init and \_\_exit macros

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Eli Arad");

MODULE\_DESCRIPTION("Toggle led in raspberry pi 3 using module");

MODULE\_VERSION("1.0");

//static char \*gpionum = "gpionum";

int gpionum = 0;

module\_param(gpionum, int, S\_IRUGO); ///< Param desc. charp = char ptr, S\_IRUGO can be read/not changed

MODULE\_PARM\_DESC(gpionum, "The gpio number to toggle"); ///< parameter description

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

{

printk(KERN\_INFO "Hello world!\n");

printk(KERN\_INFO "gpio num %d\n" , gpionum);

return 0; // Non-zero return means that the module couldn't be loaded.

}

static void \_\_exit gpiok\_cleanup(void)

{

printk(KERN\_INFO "Cleaning up module.\n");

}

module\_init(gpiok\_init);

module\_exit(gpiok\_cleanup);

**2.6. Passing Command Line Arguments to a Module**

<http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html>

We can pass parameters to the kernel from type charp( char pointer) to short , int and array.

The initial module contains parameter name gpionum from type int.

To load the module type

Insmod gpiok

Or

Insmod gpiok gpionum=31

Using dmesg we can see the printk:

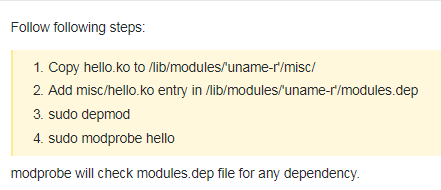
[1820153.716673] Hello world!

[1820153.716679] gpio num 31

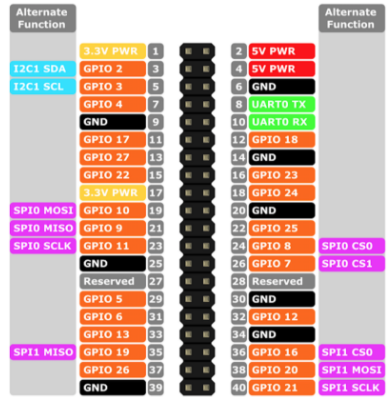
**How to load using modprobe.**

modprobe check for dependencies in kernel modules and load all the dependencies if exists

To be able to load the module using modprobe:



**Gpio of raspberry pi 3**



Before we access the gpio, let’s make sure it is connected properly and we can see it using the sysfs gpio.

**cd /sys/class/gpio**

**echo 17 > export**

**where 17 is the gpio number**

**echo out > gpio17/direction**

**echo 1 > gpio17/value**

Gpio 17 that we select to toggle is physical pin 11

Let’s write a bash script to toggle the gpio:

while true

do

echo 1 > gpio17/value

sleep 1

echo 0 > gpio17/value

done

And we see the gpio toggle.

**Toggle the gpio in our linux kernel module**



<https://lwn.net/Articles/532714/>

It is also possible to pass GPIO numbers through platform data or a device tree.

GPIOs must be allocated before use

int gpio\_request(unsigned int gpio, const char \*label);

The gpio parameter indicates which GPIO is required, while label associates a string with it that can later appear in sysfs.

A zero return code indicates success

A GPIO can be returned to the system

void gpio\_free(unsigned int gpio);

Direction:

int gpio\_direction\_input(unsigned int gpio);

int gpio\_direction\_output(unsigned int gpio, int value);

For input GPIOs, the current value can be read with:

int gpio\_get\_value(unsigned int gpio);

To set value:

void gpio\_set\_value(unsigned int gpio, int value);

Interrupt:

int gpio\_to\_irq(unsigned int gpio);

The value for gpio is the return handle from the function gpio\_request and not the gpio pin itself.

If interrupt is associated it will return back and then can be hook using

request\_irq()

 The GPIO subsystem is able to represent GPIO lines via a sysfs hierarchy, allowing user space to query (and possibly modify) them. Kernel code can cause a specific GPIO to appear in sysfs with:

int gpio\_export(unsigned int gpio, bool direction\_may\_change);

The direction\_may\_change parameter controls whether user space is allowed to change the direction of the GPIO; in many cases, allowing that control would be asking for bad things to happen to the system as a whole. A GPIO can be removed from sysfs with gpio\_unexport() or given another name with gpio\_export\_link().

Our new init function will now be:

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

{

printk(KERN\_INFO "Hello world!\n");

printk(KERN\_INFO "gpio num %d\n" , gpionum);

if (gpionum != 0)

{

gpio\_led = gpio\_request(gpionum , "MyLed");

if (gpio\_led != 0)

{

printk(KERN\_INFO "Gpio allocation failed %d\n" , gpio\_led);

}

else

{

printk(KERN\_INFO "Gpio allocation ok for %d\n", gpionum);

gpio\_direction\_output(gpionum, 1);

gpio\_export(gpionum, 0);

}

}

return 0; // Non-zero return means that the module couldn't be loaded.

}