

Operating Systems for Embedded Systems

Developing a loadable kernel module to manage GPIOs in i.MX53QSB

Massimo Violante, Politecnico di Torino, Dip. Automatica e Informatica
massimo.violante@polito.it

Purpose

The purpose of this document is to describe how to create a minimal loadable kernel module capable of sensing the i.MX53QSB GPIO buttons to be used in the lab for the course “Operating System for Embedded System”.

Introduction

The i.MX53QSB sets available two GPIOs attached to push buttons: USER1 corresponding to GPIO_2_14, and USER2 corresponding to GPIO_2_15.

When the Freescale BSP is used (e.g., Linux kernel 2.6.35.3), a built-in driver already manages USER1 and USER2. In order to write a custom module using them, capable of reacting to the interrupts they generate, we should remove the built-in driver, and for this purpose the Linux kernel must be configured disabling the following options:

```
CONFIG_KEYBOARD_GPIO
CONFIG_KEYBOARD_GPIO_MODULE
```

When the new kernel is ready, we can proceed with the following steps.

The loadable kernel module

In this lab exercise a simple module is used that prints a message every time the USER1 button is pressed.

The code of the module is the following.

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/irq.h>
#include <linux/interrupt.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <linux/gpio.h>

#define MOD_LICENSE "GPL"
#define MOD_AUTHOR "Massimo Violante <massimo.violante@polito.it>"
#define MOD_DESCRIPTION "Simple char device driver"

#define MODULE_NAME      "simple_driver"

MODULE_LICENSE (MOD_LICENSE);
MODULE_DESCRIPTION (MOD_DESCRIPTION);
MODULE_AUTHOR (MOD_AUTHOR);

unsigned int    irq_num = -1;
unsigned int    irq_counter = 0;

static int __init init_routine(void);
static void __exit exit_routine(void);

module_init(init_routine);
```

```

module_exit(exit_routine);

static irqreturn_t myirq_handler( int irq, void *dev_id )
{
    printk( KERN_INFO "%s: entering interrupt handler\n", MODULE_NAME );
    irq_counter++;
    printk( KERN_INFO "%s: IRQ received (%d times)\n", MODULE_NAME, irq_counter );
    return( IRQ_HANDLED );
}

static int __init init_routine(void)
{
    int      result;

    if( gpio_is_valid( 46 ) )
        gpio_free( 46 );
    else
    {
        printk( KERN_INFO "%s: invalid gpio 46\n", MODULE_NAME );
        return( result );
    }

    result = gpio_request( 46, MODULE_NAME );
    if( result )
    {
        printk( KERN_INFO "%s: cannot get gpio 46\n", MODULE_NAME );
        return( result );
    }
    result = gpio_direction_input( 46 );
    if( result )
    {
        printk( KERN_INFO "%s: cannot gpio 46 direction\n", MODULE_NAME );
        return( result );
    }

    irq_num = gpio_to_irq( 46 );

    printk(KERN_INFO "%s: registering for IRQ %d\n", MODULE_NAME, irq_num );

    result = request_irq( irq_num, myirq_handler, IRQF_TRIGGER_FALLING,MODULE_NAME,NULL );
    if( result )
    {
        printk( KERN_INFO "%s: error %d\n", MODULE_NAME, result );
        return( result );
    }

    gpio_free( 46 );

    printk( KERN_INFO "%s: registered IRQ %d\n", MODULE_NAME, irq_num );
    return 0;
}

static void __exit exit_routine(void)
{
    free_irq( irq_num, NULL );
}

```

The core of the module is the initialization function (`init_routine`), which programs the GPIO and registers the custom interrupt handler (`myirq_handler`).

In details, the operations performed are the following:

1. First, we need to check whether the integer number associated to the GPIO is a valid one. This is done using the function call `gpio_is_valid(46)`. The integer number associated to the GPIO is computer using the following formula: $\text{GPIO}_{x,y} = (x-1)*32+y$. For USER1, GPIO_2_14, the associated integer number is **46**.
2. If it is valid, we set it free so that we can configure it. This is done using the function call: `gpio_free(46)`.
3. We now claim the GPIO, and we set it as input using the sequence of function calls: `gpio_request(46, MODULE_NAME)`, `gpio_direction_input(46)`.

4. If everything proceeded without errors, we can now ask the Linux kernel to provide the interrupt number associated to the selected GPIO. This is done using the function call: `gpio_to_irq(46)`
5. We can not register our custom interrupt routine to the proper interrupt `request_irq(irq_num, myirq_handler, IRQF_TRIGGER_FALLING, MODULE_NAME, NULL)`, and we can release the GPIO: `gpio_free(46)`.

The `Makefile` is the following, where `<path to kernel>` is the directory where the Linux kernel is stored.

```
obj-m:= simple_driver.o

PWD = $(shell pwd)

default: simple_driver.ko

simple_driver.ko: simple_driver.c
    make M=$(PWD) -C <path to kernel> ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
clean:
    rm *.o *.ko
```

Testing the module

Once the module has been cross-compiler and copied over the target, we can insert it in the kernel:

```
insmod simple_driver.ko
```

If everything is fine, we should get the following Linux kernel log messages:

```
simple_driver: registering for IRQ 174
simple_driver: registered IRQ 174
```

Now, every time we press USER1 button we get the following Linux kernel log message:

```
simple_driver: entering interrupt handler
simple_driver: IRQ received (x times)
```

Where `x` is an integer that increments every time we press the button.