**1.**

**a)**

Interrupts können keinen Speicher allokieren außer mit GFP\_ATOMIC. Der Grund dafür ist, dass Interrupts keine Aktionen ausführen die können die in den *sleep* Zustand gehen würden.

**b)**

In der *Top Half* werden kurze Operationen ausgeführt und in der *Bottom Half* können längere Operationen ausgeführt werden. In Gegensatz zur *Top Half* sind in der *Bottom Half* Interrupts aktiviert.

Zwei mögliche Arten von *Bottom Halves* sind *Tasklets* und *Workqueues*. *Tasklets* sind schneller, müssen aber *atomic* sein. *Workqueues* haben eine höhere Latenz, sie können dafür aber auch schlafen.

**2.**

In der Datei sind die verschiedenen Interrupts zu sehen. Der Interrupt mit dem Namen *leds* ist der Interrupt aus der Übung. 47 ist die Interrupt Nummer. 13 ist die Anzahl der Aufrufe des Interrupts.

root@colibri:/var/tmp# cat /proc/interrupts

CPU0

3: 0 ohci\_hcd:usb1

8: 0 MMC card detect

21: 4 BTUART

22: 1089 FFUART

23: 2 pxa2xx-mci

25: 0 DMA

26: 62699 PXA Timer Tick

47: 13 leds

116: 0 PCMCIA CD

146: 17639 eth0

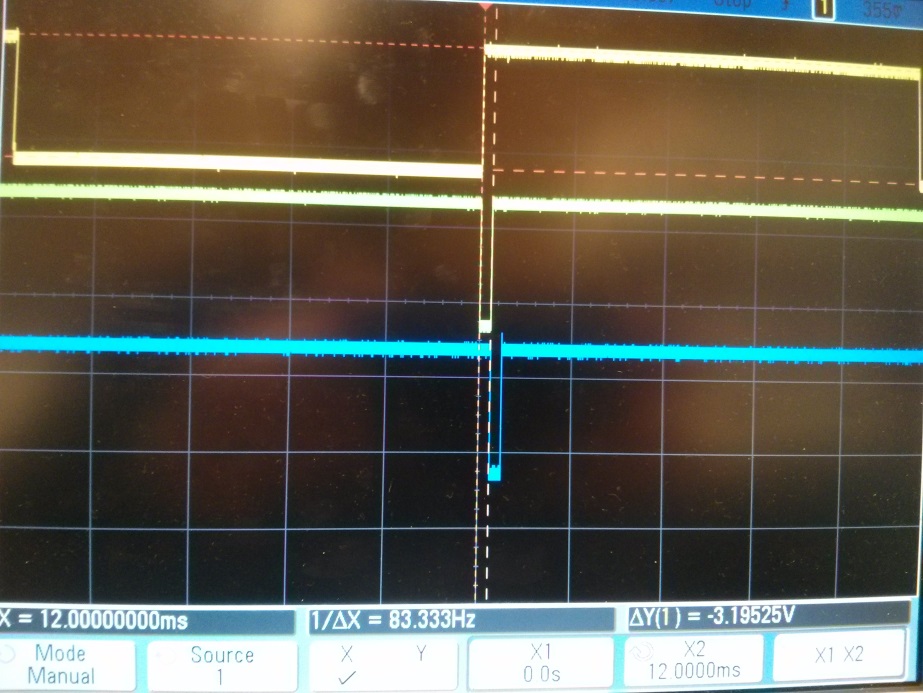
Err: 0

**3.**

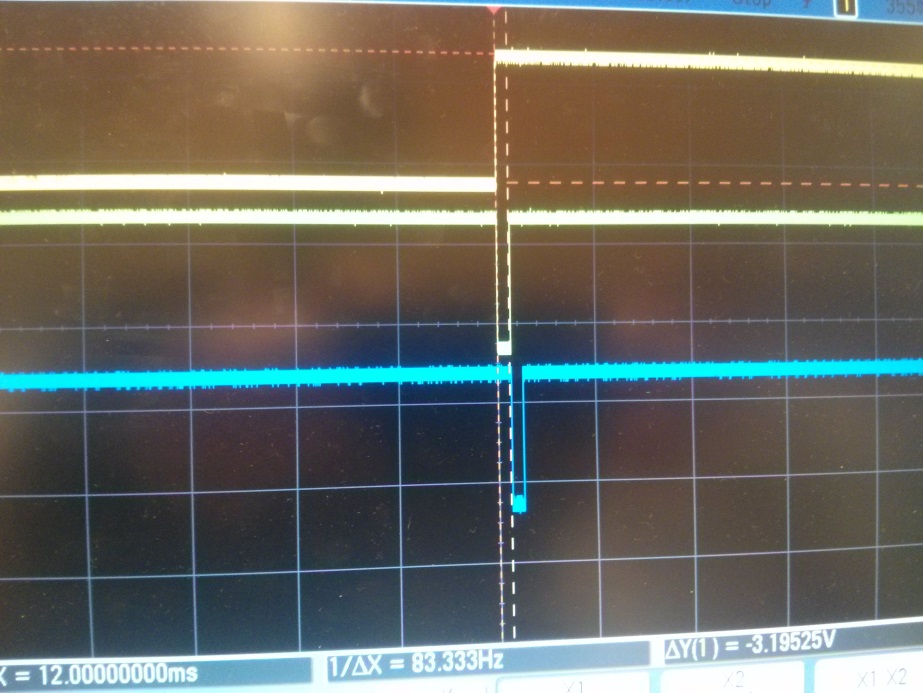
**Tasklet ohne Belastung:**

****

**Tasklet mit Belastung:**



**Workqueue ohne Belastung:**

****

**Workqueue mit Belastung:**

****

Bei allen vier Messungen wurden keine Unterschiede festgestellt. Die Latenz zwischen Signal und Interrupt war praktisch 0 und die Latenz zwischen Signal und Tasklet/Workqueue war 12ms.

Erwartet hätten wir uns, dass die Workqueue unter Last länger dauert, da sie vom Scheduler unterbrochen werden kann.

Die Bilder wurden mit einem Handy abfotografiert, da der USB Stick vom Oszilloskop nicht erkannt wurde.

**Sourcecode:**

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/ioport.h>

#include <asm/io.h>

#include <linux/cdev.h>

#include "gpio\_defs.h"

#include "hwaccess.h"

#include <asm/arch/irqs.h>

#include <asm/uaccess.h>

#include <linux/interrupt.h>

#define DEVICE\_NAME "leds"

#define MINOR\_NUMBER 0

#define NUMBER\_OF\_DEVICES 1

#define SWITCH0\_IRQ IRQ\_GPIO(15)

MODULE\_LICENSE("Dual BSD/GPL");

struct gpio\_dev\_t gpio\_dev;

static struct work\_struct ledDriver\_wq;

static ssize\_t ledDriver\_read(struct file \*filp, /\* see include/linux/fs.h \*/

char \*buffer, /\* buffer to fill with data \*/

size\_t length, /\* length of the buffer \*/

loff\_t \* offset)

{

buffer[0] = readSwitches(gpio\_dev.vaddr);

return 1;

}

static ssize\_t ledDriver\_write(struct file \*filp, const char \*buffer, size\_t length, loff\_t \* offset)

{

setLeds(gpio\_dev.vaddr, buffer[0]);

return 0;

}

struct file\_operations ledDriver\_fops = {

read: ledDriver\_read,

write: ledDriver\_write

};

void ledDriver\_do\_tasklet(unsigned long unused) {

setLeds(gpio\_dev.vaddr, 0x02);

printk("Led 2 set\n");

setLeds(gpio\_dev.vaddr, 0x00);

}

void ledDriver\_do\_wq(unsigned long unused) {

setLeds(gpio\_dev.vaddr, 0x04);

printk("Led 3 set!\n");

setLeds(gpio\_dev.vaddr, 0x00);

}

DECLARE\_TASKLET(ledDriver\_tasklet, ledDriver\_do\_tasklet, 0);

irqreturn\_t led\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

setLeds(gpio\_dev.vaddr, 0x01);

setLeds(gpio\_dev.vaddr, 0x00);

//tasklet\_schedule(&ledDriver\_tasklet);

schedule\_work(&ledDriver\_wq);

return IRQ\_HANDLED;

}

static int ledDriver\_init(void)

{

if (request\_mem\_region(GPIO\_BASE, sizeof(GpioRegs), DEVICE\_NAME) == NULL)

{

printk(KERN\_ALERT "Error requesting memory\n");

return -1;

}

gpio\_dev.vaddr = (GpioRegs\*)ioremap\_nocache(GPIO\_BASE, sizeof(GpioRegs));

if (alloc\_chrdev\_region(&gpio\_dev.first, MINOR\_NUMBER, NUMBER\_OF\_DEVICES, DEVICE\_NAME))

{

goto fail\_alloc\_chrdev;

}

cdev\_init(&gpio\_dev.cdev, &ledDriver\_fops);

if (cdev\_add(&gpio\_dev.cdev, gpio\_dev.first, NUMBER\_OF\_DEVICES))

{

goto fail\_cdev\_add;

}

initLeds(gpio\_dev.vaddr);

initSwitches(gpio\_dev.vaddr);

INIT\_WORK(&ledDriver\_wq, (void (\*)(void \*))ledDriver\_do\_wq, NULL);

int ledIrq = SWITCH0\_IRQ;

if (ledIrq >= 0)

{

set\_irq\_type(SWITCH0\_IRQ, IRQT\_RISING);

int result = request\_irq(ledIrq, led\_interrupt, SA\_INTERRUPT, DEVICE\_NAME, NULL);

if (result)

{

printk(KERN\_INFO "led: can't get assigned irq %i\n", ledIrq);

ledIrq = -1;

}

}

return 0;

fail\_cdev\_add:

unregister\_chrdev\_region(gpio\_dev.first, NUMBER\_OF\_DEVICES);

fail\_alloc\_chrdev:

iounmap((void\*)gpio\_dev.vaddr);

release\_mem\_region(GPIO\_BASE, sizeof(GpioRegs));

return -1;

}

static void ledDriver\_exit(void)

{

cdev\_del(&gpio\_dev.cdev);

unregister\_chrdev\_region(gpio\_dev.first, NUMBER\_OF\_DEVICES);

iounmap(gpio\_dev.vaddr);

release\_mem\_region(GPIO\_BASE, sizeof(GpioRegs));

}

module\_init(ledDriver\_init);

module\_exit(ledDriver\_exit);