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.

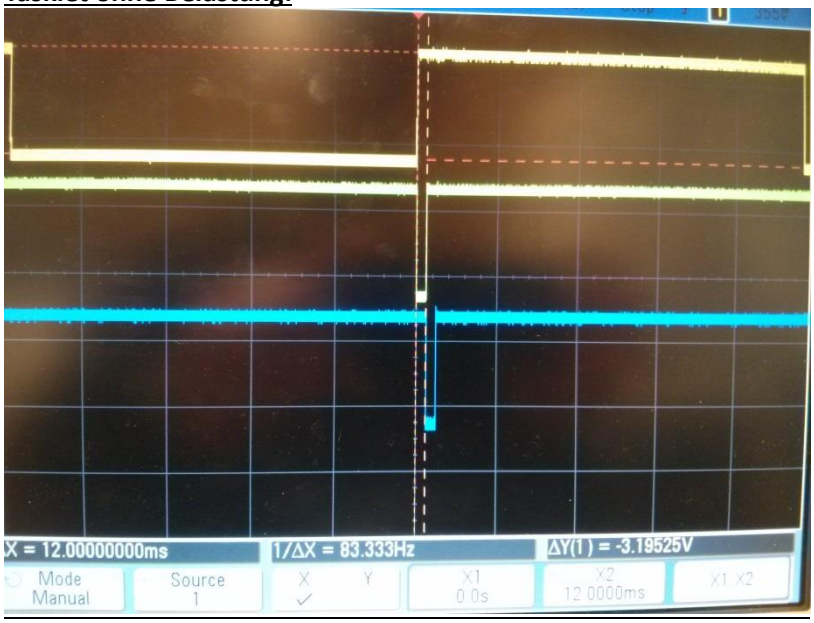
2.

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.

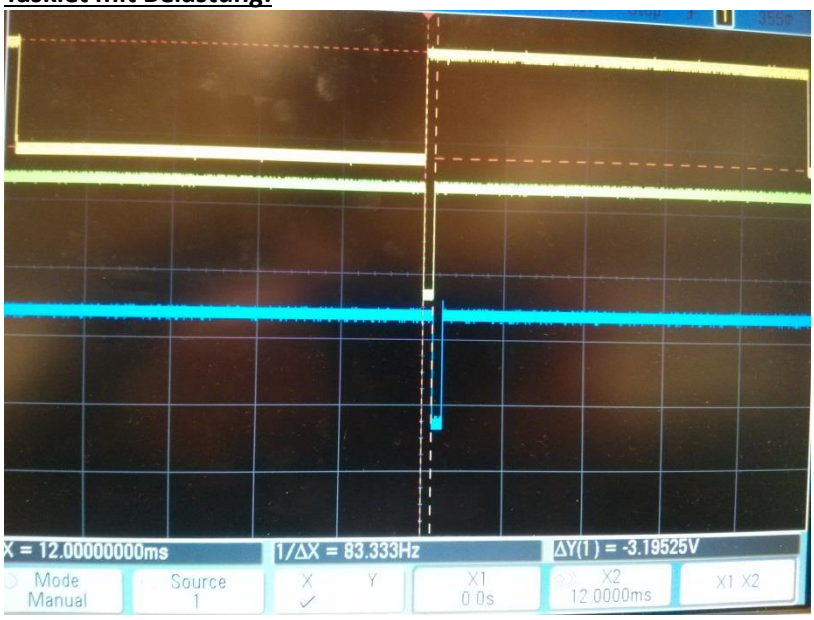
```
root@colibri:/var/tmp# cat /proc/interrupts
           CPU0
  3:          0   ohci_hcd:usb1
  8:          0   MMC card detect
 21:          4   BTUART
 22:       1089   FFUART
 23:          2   pxa2xx-mci
 25:          0   DMA
 26:      62699   PXA Timer Tick
 47:         13   leds
116:          0   PCMCIA CD
146:      17639   eth0
Err:          0
```

3.

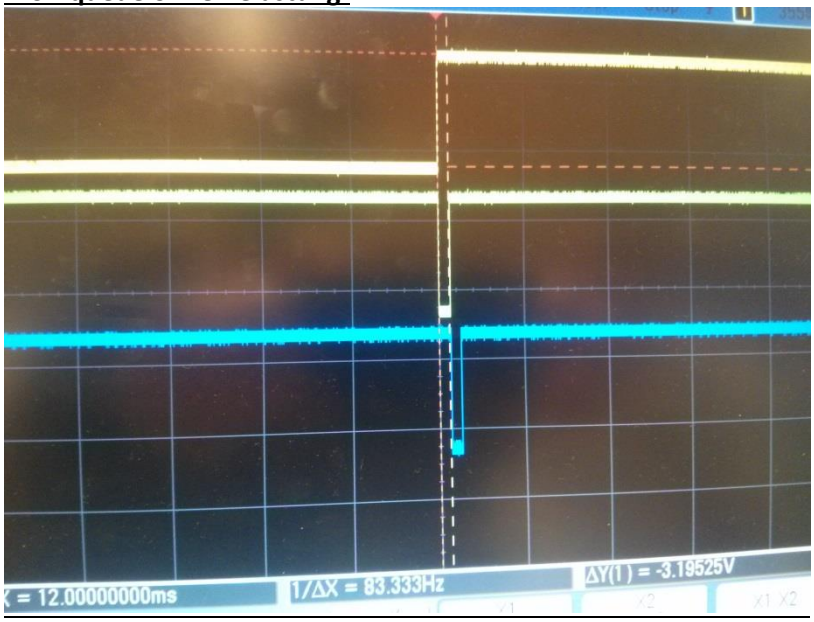
Tasklet ohne Belastung:



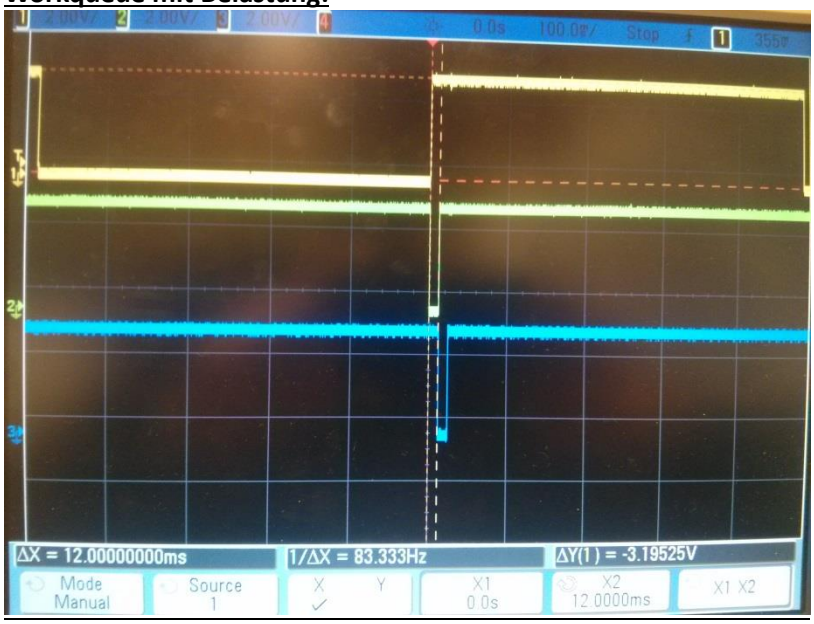
Tasklet mit Belastung:



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, DE-
VICE_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);
```