

Hardware

Hans Buchmann FHNW/IME

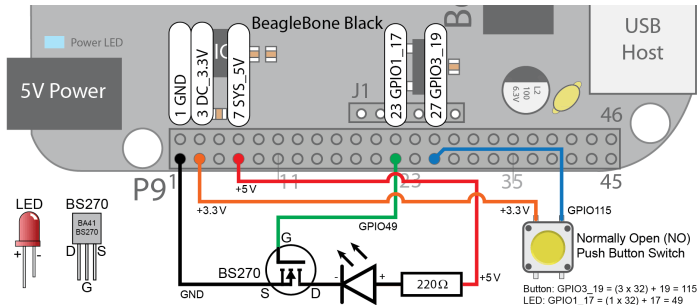
22. Mai 2018

Um was geht es ?

Hardware

- ▶ *user-space* → *kernel-space* → *HW*
 - ▶ via *sysfs*
- ▶ *HW* → *kernel-space* → *user-space*
 - ▶ Interrupts

Unsere Hardware



©derekmolloy.ie/kernel-gpio-programming-buttons-and-leds

Teil I

user-space

Vom *user-space* aus

In Verzeichnis `/sys/class/gpio`

- ▶ Output `gpio49`
 - ▶ `cd gpio49`
 - ▶ `echo out > direction`
 - ▶ `echo 1 > value echo 0 > value`
- ▶ Input `gpio115`
 - ▶ entsprechend

- ▶ blink
- ▶ Polling
 - ▶ read switch set led

Teil II

LKM

Loadable Kernel Module (LKM)

gpio-0.c

- ▶ eigenes Verzeichnis in `sys: my-hw`
- ▶ einen File `led` mit `rw`:
 - ▶ write: `echo x > /sys/my-hw/led`
 - `x=1` set led
 - `x=0` clear led
 - `x=t|T` toggle ledandere Werte von `x` ändern nichts
 - ▶ read: `cat /sys/my-hw/led`
 - `1` led on
 - `0` led off

Was Sie brauchen

Pin 49 ist Output

- ▶ `gpio_request`
- ▶ `gpio_free`
- ▶ `gpio_direction_output`
- ▶ `gpio_set_value`

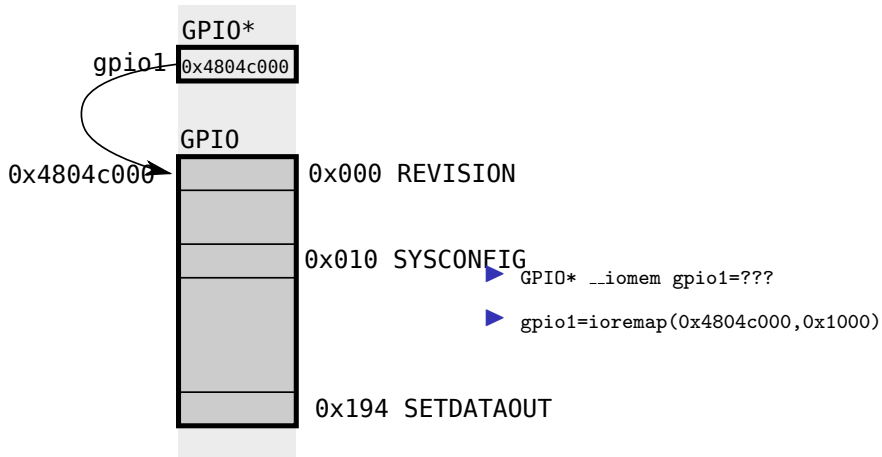
Direkter Zugriff auf die *HW*

- ▶ die Register
- ▶ als Memory
- ▶ Siehe `src/gpio.h`

Informationen

▶ `cat /proc/iomem`

Register in Memory



Die einzelnen Bits die Operationen

setBit Operation |
clrBit Operation &

Loadable Kernel Module (LKM)

gpio-1.c

- ▶ Manual spruh731.pdf Abschnitt 25
- ▶ gpio.h die einzelnen Register:
 - REVISION für Test
 - OE OutputEnable
 - DATAIN Data In
 - CLEARDATAOUT Löscht Bit
 - SETDATAOUT Setzt Bit
- ▶ ioremap
- ▶ iounmap

Outline

Callback Hollywood Prinzip:

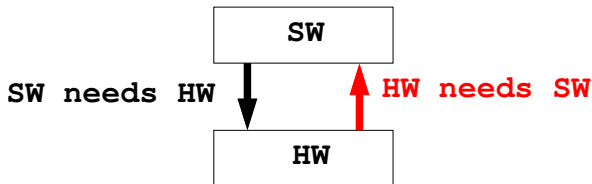
don't call us, we'll call you.

- ▶ die Hardware ruft zurück

Interrupts Der Schritt zur Echtzeit

- ▶ Die Hardware ruft Software auf
- ▶ Elementare Anwendung vom *callback*
- ▶ Ereignisgesteuert: *event driven*

SW braucht HW und umgekehrt



Beispiel

- SW→HW
 - ▶ Setze LED
 - ▶ Schreibe Buchstabe
- HW→SW
 - ▶ Ein *timer* tick
 - ▶ Tastatur Taste gedrückt

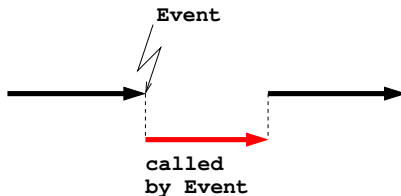
Remark: Beide Richtungen $SW \rightarrow HW$ und $HW \rightarrow SW$ sind wichtig

Asynchron jederzeit

Normale Programmausführung: Thread



Interrupt



jederzeit Zwischen zwei Instruktionen
unsichtbar vom Thread aus gesehen

Informationen

▶ `cat /proc/interrupts`

Ziel

- ▶ Interrupt erkennen
- ▶ Switch
 - ▶ drücken: erzeugt interrupt: `IRQF_TRIGGER_RISING`
 - ▶ loslassen: erzeugt interrupt: `IRQF_TRIGGER_FALLING`

gpio-2.c
schrittweise

- ▶ gpio_direction_input
- ▶ gpio_to_irq
- ▶ request_irq

Der Handler

```
static irqreturn_t onSWI(int id, void* d)
{
    printk("onSWI\n"); /* debug */
    /* do something */
    return IRQ_HANDLED;
}
```

foreground-background

Tasklet: die Verbindung

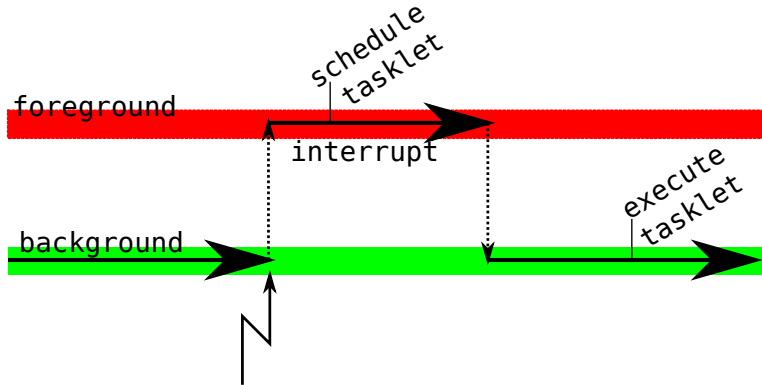
foreground der/die Interrupts

background die normale Ausführung

tasklet Verbindung foreground → background

Tasklet

foreground → background



gpio-3.c
schrittweise

- ▶ Definition global `tasklet_struct`
- ▶ Initialisation `tasklet_init`
- ▶ Scheduling im *foreground* `tasklet_schedule`

Der Handler

```
/* *not* called in interrupt */  
static void onSWITasklet(unsigned long data)  
{  
    /* do something */  
}
```

Teil III

kernel-space → *user-space*
Semaphore & Co.

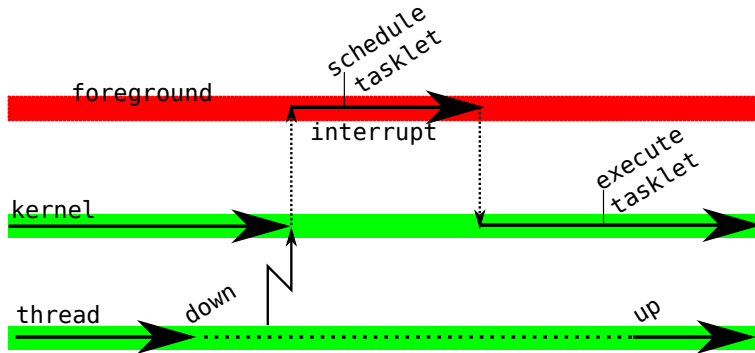
Semaphore



©David Ingham

- ▶ down: Signal zu
 ▶ warten
- ▶ up: Signal offen
 ▶ fahren

Tasklet-Semaphore



gpio-4.c

schrittweise

- ▶ Definition global `semaphore`
- ▶ Initialisation
`sema_init` anfangs geschlossen
- ▶ Im Tasklet `up` öffne
- ▶ Im `get_swi` down schliesse