

# Hardware

Hans Buchmann FHNW/IME

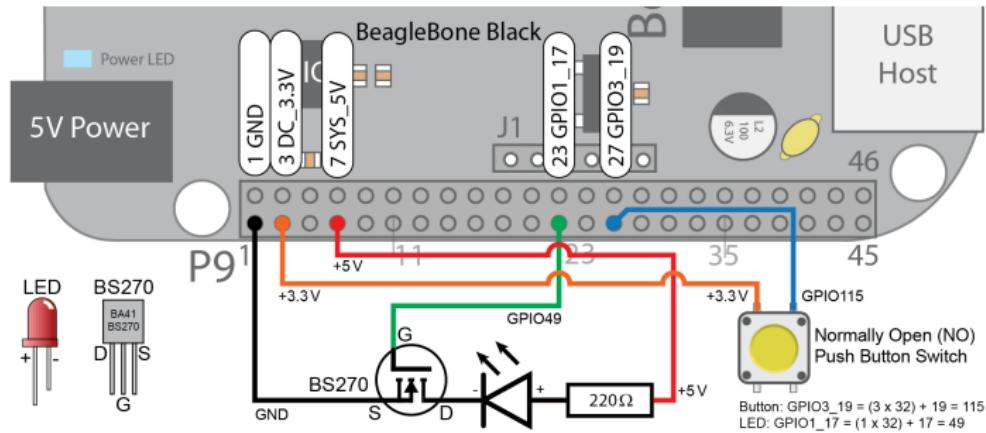
12. Juni 2018

Um was geht es ?

Hardware

- ▶  $\text{user-space} \rightarrow \text{kernel-space} \rightarrow \text{HW}$ 
  - ▶ via *sysfs*
- ▶  $\text{HW} \rightarrow \text{kernel-space} \rightarrow \text{user-space}$ 
  - ▶ 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

# Aufgabe

## Skript

- ▶ 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 led
    - andere 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`

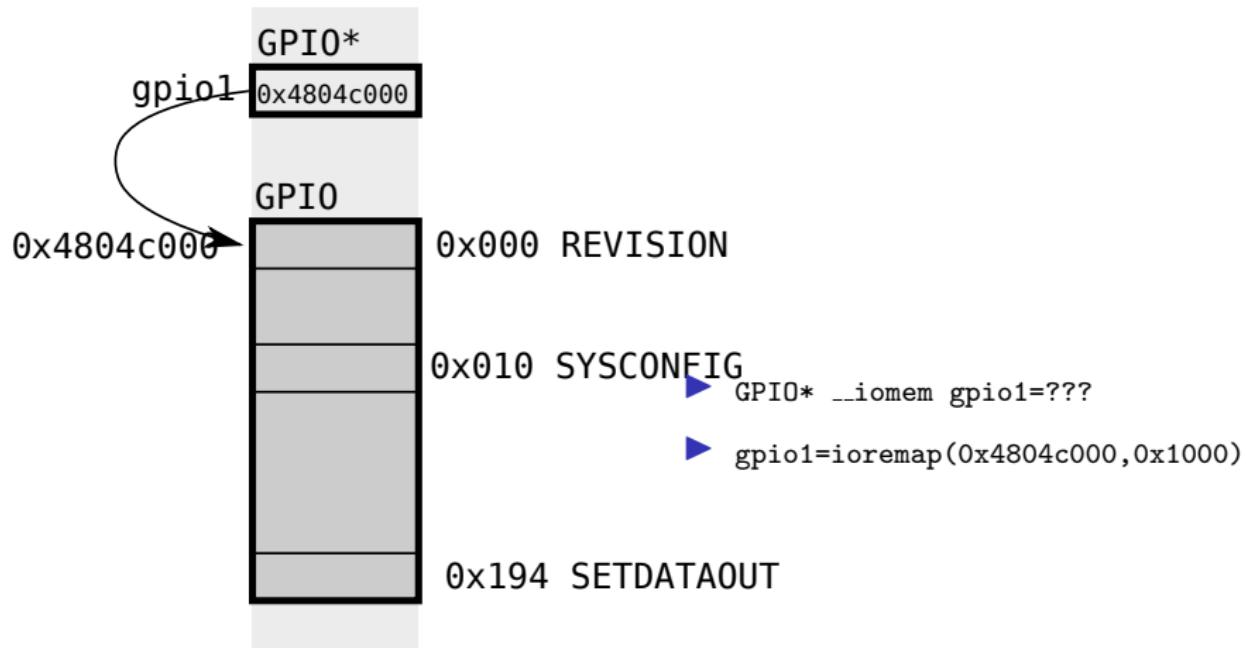
## Direkter Zugriff auf die HW

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

## Informationen

▶ cat /proc/iomem

## Register im 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:

`REVSION` für Test

`OE` OutputEnable

`DATAIN` Data In

`CLEARDATAOUT` Löscht Bit

`SETDATAOUT` Setzt Bit

- ▶ `ioremap`
- ▶ `iounmap`

## Outline

Callback Hollywood Prinzip:

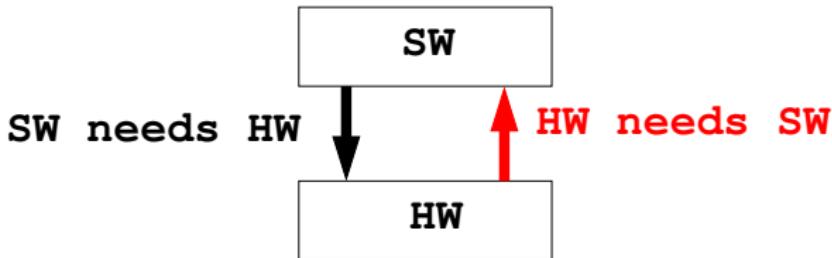
*don't call us, we'll call you.*

- ▶ 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

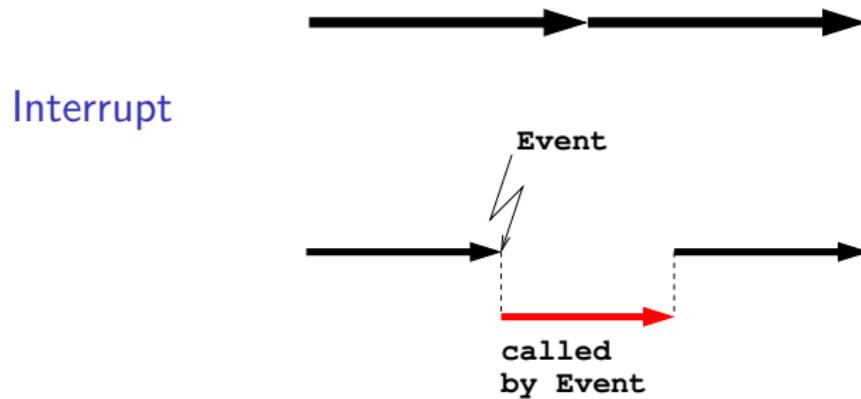
HW→SW

- ▶ Ein *timer* tick
- ▶ Tastatur Taste gedrückt

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

Asynchron  
jederzeit

## Normale Programmausführung: Thread



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 schriftweise

- ▶ 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

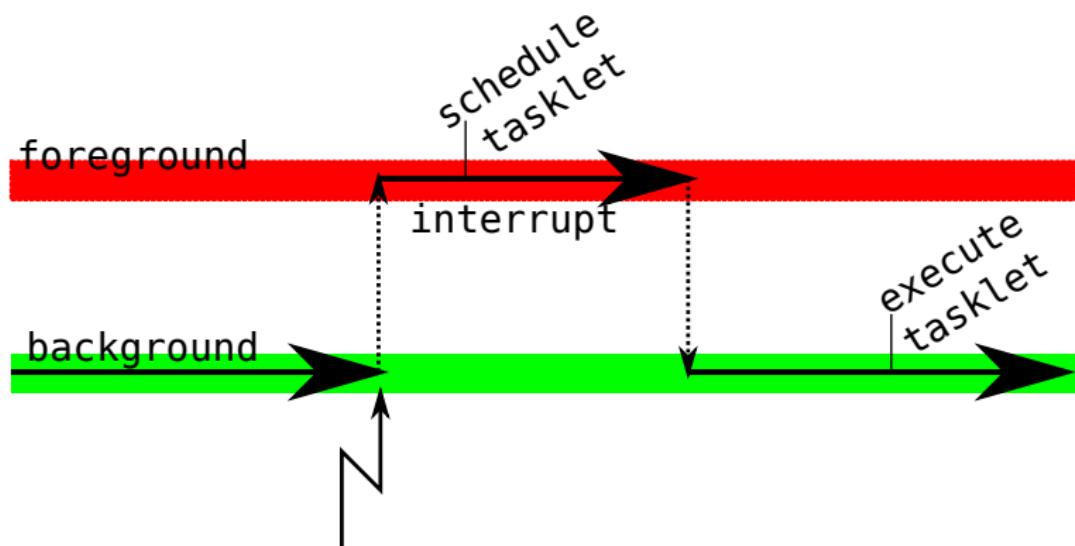
foreground der/die Interrupts

background die normale Ausführung

tasklet Verbindung foreground → background

## Tasklet

foreground → background



## gpio-3.c schriftweise

- ▶ 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.

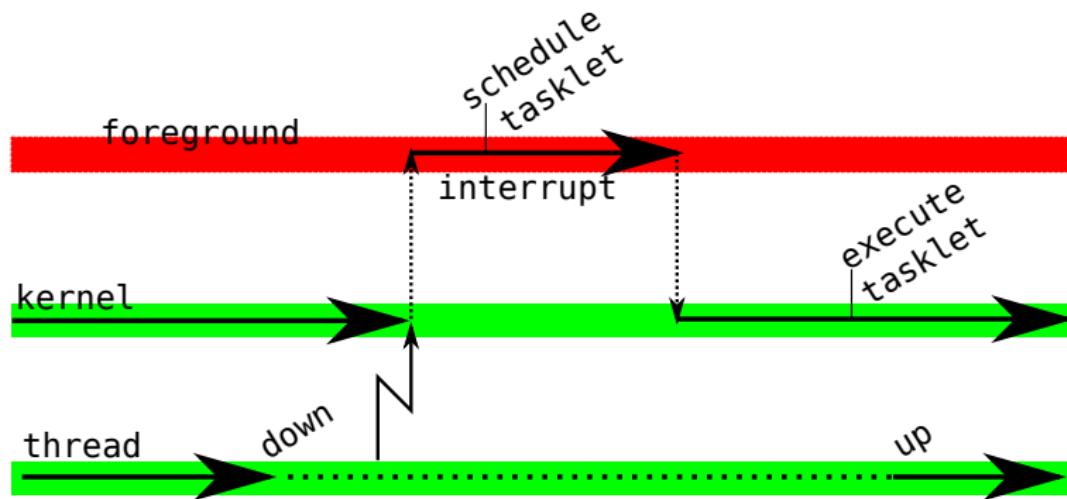
## Semaphore



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

©upload.wikimedia.org/wikipedia/commons/0/0b/BhfEpfenhofen\_Ausfahrssignale\_Talaufw

## Tasklet-Semaphore

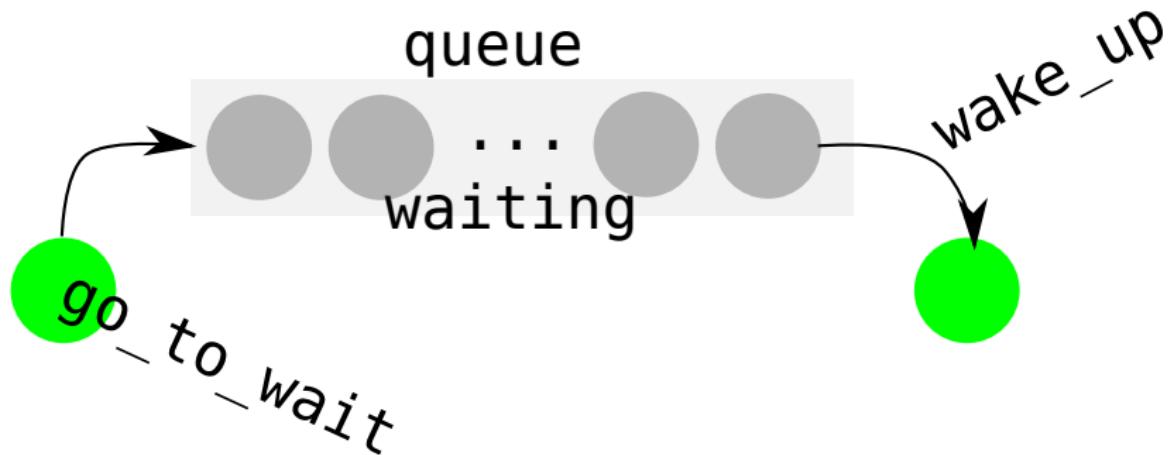


## gpio-4.c schrittweise

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

## Wait queue

Task/processe warten

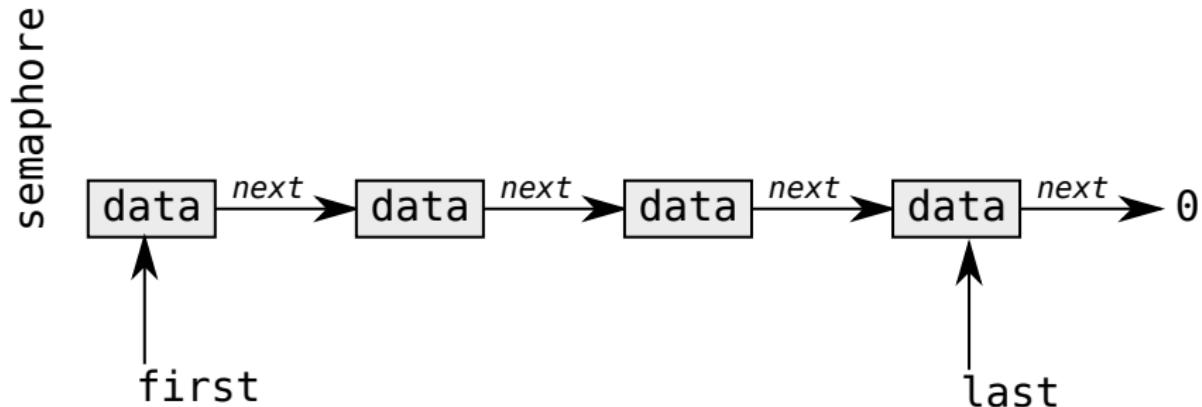


- running thread
- waiting thread

## gpio-5.c schriftweise

- ▶ Definition global
  - ▶ `wait_queue_head`
  - ▶ `swiN` die Anzahl Schaltungen von `swi`
- ▶ go to wait
  - ▶ `wait_event_interruptible`
- ▶ wakeup
  - ▶ `wake_up_interruptible`

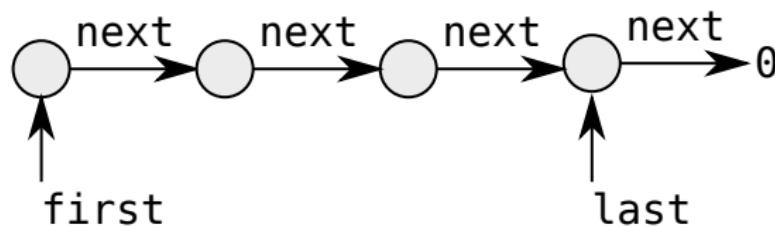
## Mit eigener *Semaphore-queue*



- ▶ put
  - ▶ füge hinten an put: data: Zustand des Schalters: on|off
  - ▶ öffne Semaphore
- ▶ get
  - ▶ warte am Semaphore
  - ▶ hole get

## Die Queue

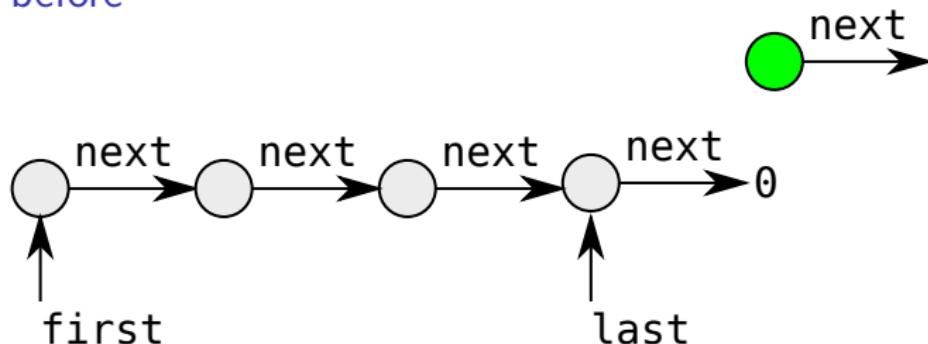
einfach verlinkte Liste



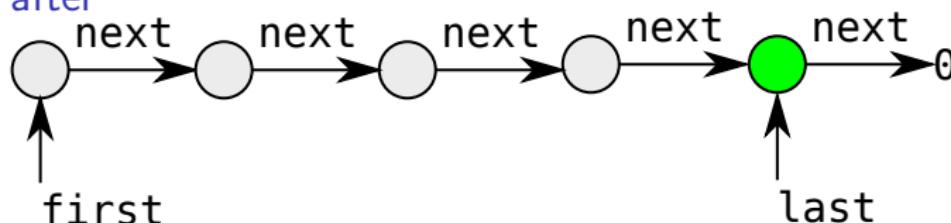
## Die Queue

put

before

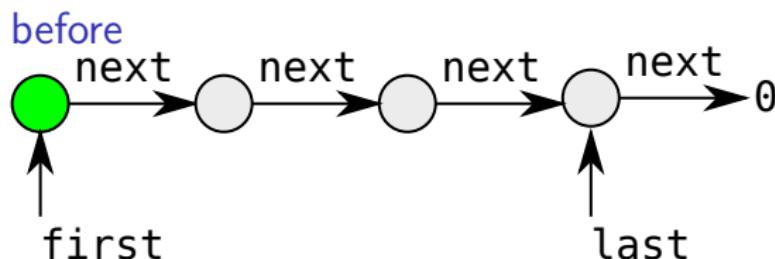


after

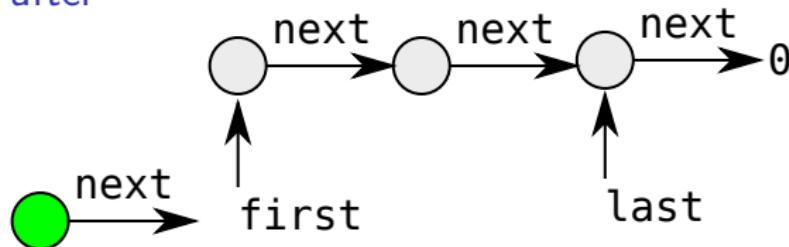


## Die Queue

get



after



## Verlinkte Listen

Kommen immer wieder vor

- ▶ im userspace
- ▶ struct
- ▶ init\_
- ▶ put\_
- ▶ get\_
- ▶ Dynamisches Memory
  - ▶ malloc
  - ▶ free

## Die Semaphore-Queue

- ▶ `struct` für Semaphore-Queue
- ▶ `init_`
- ▶ `put_`
- ▶ `get_`
- ▶ Dynamisches Memory
  - ▶ `kmalloc`
  - ▶ `kfree`

## TODO's

Im Zusammenhang mit den Interrupts

- ▶ mehr Informationen in `buf` von `get_swi` zurückgeben:

```
struct SWIInfo
{
    time_t    when;
    unsigned  nbr; //the number of switching
    unsigned  state; //on|off
};
```

- ▶ Queue als *ringbuffer*

# C++ gemacht mit C

- ▶ src/queue-a-la-cpp.c

## SWIInfo

```
typedef struct SWIInfo
{
    time_t      when;
    unsigned     nbr;
    unsigned     state;
} SWIInfo;
```

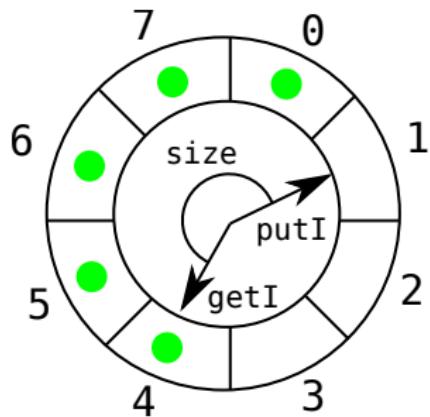
- ▶ Die Zeit
  - ▶ Globale Variable: jiffies

## gpio-8.c

### Vorgehen

- ▶ currentE
  - ▶ der aktuelle Eintrag: gesetzt von onSWI
- ▶ tasklet call-back
  - ▶ **kopiert** currentE in den Ringbuffer: put
- ▶ get\_swi
  - ▶ **kopiert** vom Ringbuffer in den buf:get

## Ringbuffer



- ▶ put
- ▶ get
- ▶ C
  - ▶ ring-buffer.c
- ▶ C++
  - ▶ ring-buffer-cpp.cc

## Teil IV

*user-space C++*

## C++ im *user-space*

- ▶ `led.cc` funktioniert das Prinzip ?
- ▶ `blink.cc` *user-space* thread
- ▶ `swi.cc` warten auf *event*

## led.cc

std::ofstream

- ▶ Constructor/Instance
  - ▶ std::ofstream led(*name*)
- ▶ write
  - ▶ led<<*value*
- ▶ Destructor
  - ▶ schliesst den File

blink.cc

## Anwendung *user-space threads*

- ▶ thread
- ▶ Zeitintervalle

## Warten auf Daten vom *kernel-space*

- ▶ SWIInfo

```
struct SWIInfo
{
    unsigned          when;
    unsigned          nbr;
    unsigned          state;
};
```

- ▶ istream/read