

Systemnahe Programmierung in C (SPiC)

Teil D Betriebssystemabstraktionen

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPiC

Ereignisbehandlung

- Bei einem **Peripheriegerät** tritt ein Ereignis (⚡) auf → 14-5
 - Signal an einem Port-Pin wechselt von *low* auf *high*
 - Ein *Timer* ist abgelaufen
 - Ein A/D-Wandler hat einen neuen Wert vorliegen
 - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
 - **Polling:** Das **Programm** überprüft den Zustand regelmäßig und ruft ggf. eine Bearbeitungsfunktion auf.
 - **Interrupt:** Gerät „meldet“ sich beim **Prozessor**, der daraufhin in eine Bearbeitungsfunktion verzweigt.

Überblick: Teil D Betriebssystemabstraktionen

15 Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme

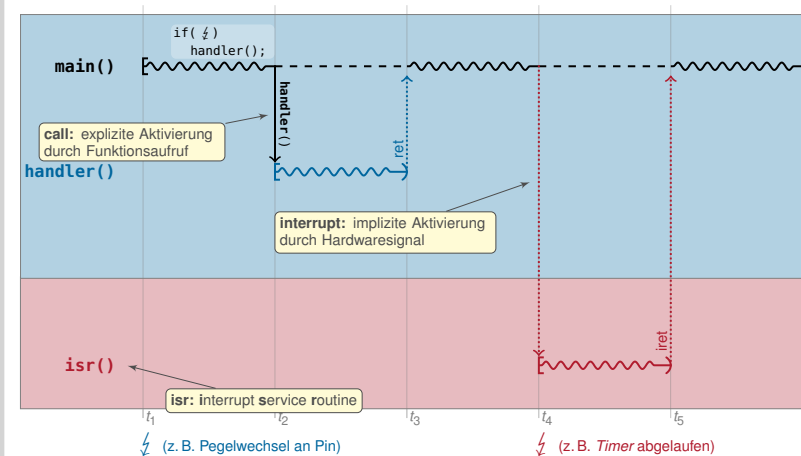
18 Dateisysteme

19 Programme und Prozesse

20 Speicherorganisation

21 Nebenläufige Prozesse

Interrupt ↦ Funktionsaufruf „von außen“

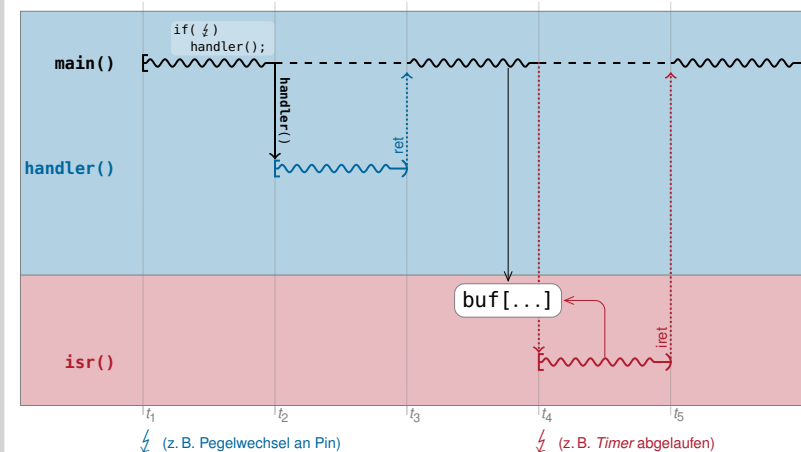


Polling vs. Interrupts – Vor- und Nachteile

- Polling (→ „Taktgesteuertes System“)
 - Ereignisbearbeitung erfolgt **synchron** zum Programmablauf
 - Ereigniserkennung über das Programm „verstreut“ (Trennung der Belange)
 - „Verschwendung“ von Prozessorzeit (falls anderweitig verwendbar)
 - Hochfrequentes Pollen $\leadsto$ hohe Prozessorlast $\leadsto$ **hoher Energieverbrauch**
 - + Implizite Datenkonsistenz durch festen, sequentiellen Programmablauf
 - + Programmverhalten gut vorhersagbar
- Interrupts (→ „Ereignisgesteuertes System“)
 - Ereignisbearbeitung erfolgt **asynchron** zum Programmablauf
 - + Ereignisbearbeitung kann im Programmtext gut separiert werden
 - + Prozessor wird nur beansprucht, wenn Ereignis tatsächlich eintritt
 - Höhere Komplexität durch Nebenläufigkeit $\leadsto$ Synchronisation erforderlich
 - Programmverhalten **schwer vorhersagbar**

Beide Verfahren bieten spezifische Vor- und Nachteile
 $\leadsto$ Auswahl anhand des konkreten Anwendungsszenarios

Interrupt $\mapsto$ unvorhersagbarer Aufruf „von außen“

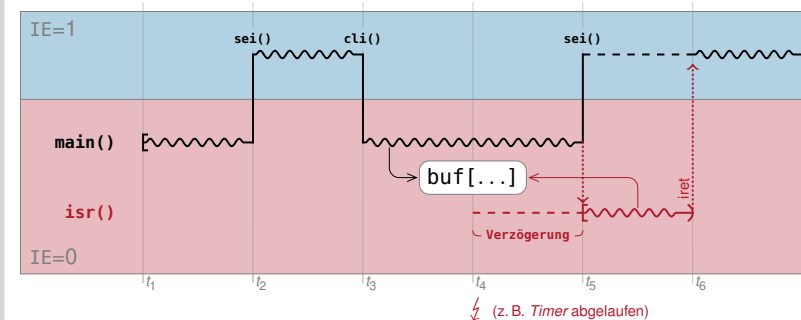


Interruptsperrn

- Zustellung von Interrupts kann softwareseitig **gesperrt** werden
 - Wird benötigt zur **Synchronisation** mit ISRs
 - Einzelne ISR: Bit in gerätespezifischem Steuerregister
 - Alle ISRs: Bit (**IE**, **I**nterrupt **E**nable) im Statusregister der CPU
- Auflaufende IRQs werden (üblicherweise) gepuffert
 - Maximal einer pro Quelle!
 - Bei längeren Sperrzeiten können IRQs verloren gehen!
- Das **IE**-Bit wird beeinflusst durch:
 - Prozessor-Befehle: `cli`: $IE \leftarrow 0$ (*clear interrupt*, IRQs gesperrt)
`sei`: $IE \leftarrow 1$ (*set interrupt*, IRQs erlaubt)
 - Nach einem RESET: $IE=0 \leadsto$ IRQs sind zu Beginn des Hauptprogramms gesperrt
 - Bei Betreten einer ISR: $IE=0 \leadsto$ IRQs sind während der Interruptbearbeitung gesperrt

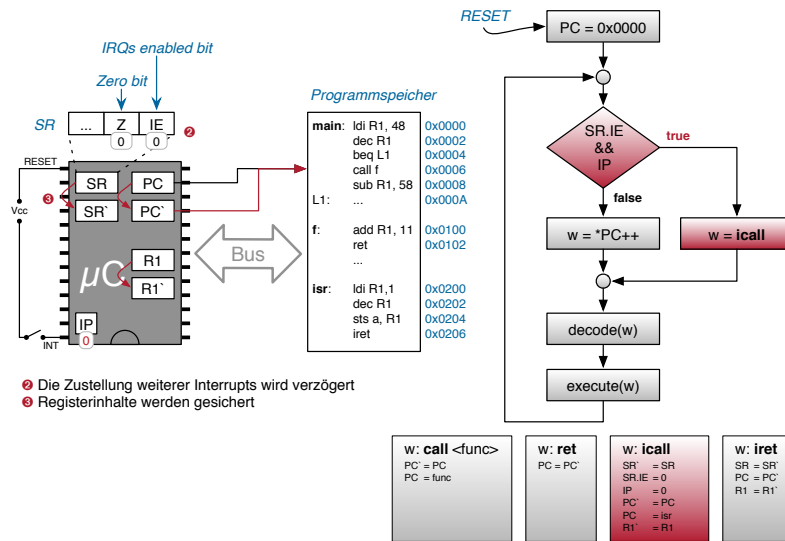
IRQ $\mapsto$ Interrupt
 ReQuest

Interruptsperrn: Beispiel

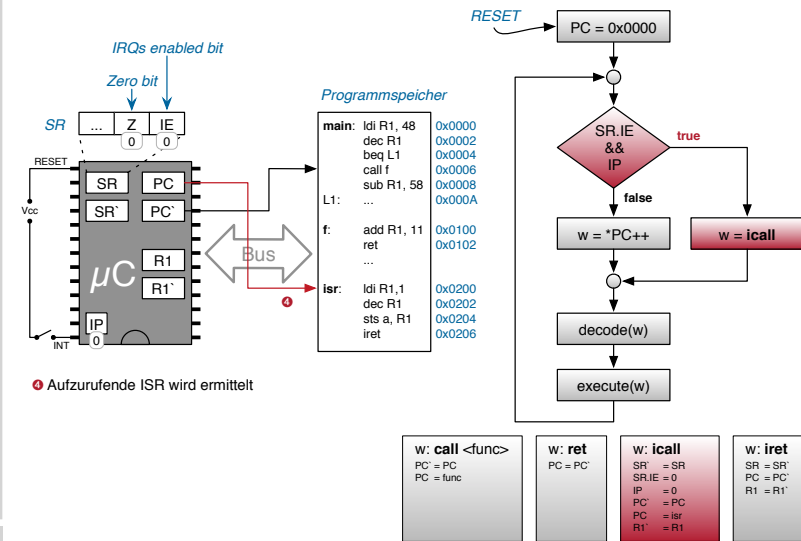


- t_1 Zu Beginn von `main()` sind IRQs gesperrt ($IE=0$)
- t_2, t_3 Mit `sei()` / `cli()` werden IRQs freigegeben ($IE=1$) / erneut gesperrt
- t_4 aber $IE=0 \leadsto$ Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t_5 `main()` gibt IRQs frei ($IE=1$) $\leadsto$ gepufferter IRQ „schlägt durch“
- t_5-t_6 Während der ISR-Bearbeitung sind die IRQs gesperrt ($IE=0$)
- t_6 Unterbrochenes `main()` wird fortgesetzt

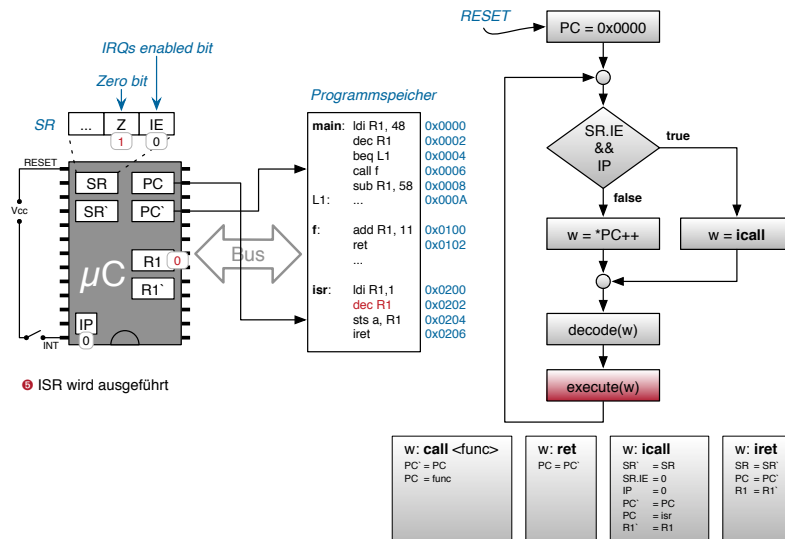
Ablauf eines Interrupts – Details



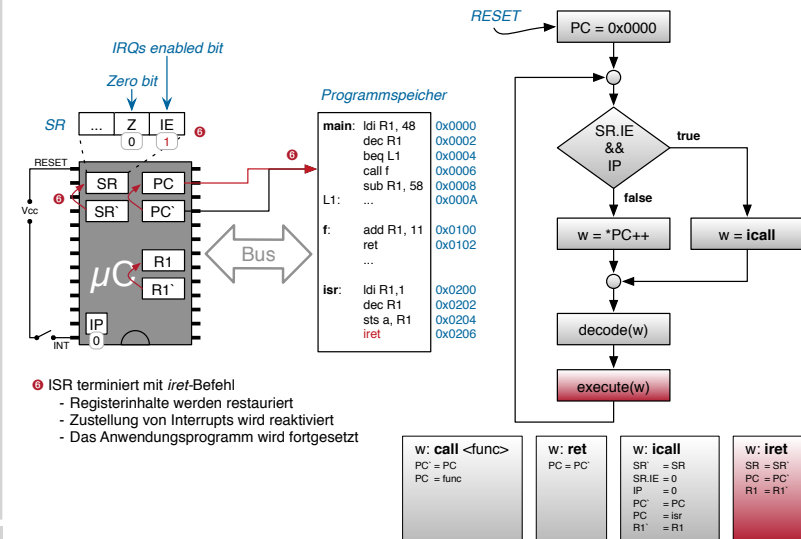
Ablauf eines Interrupts – Details



Ablauf eines Interrupts – Details

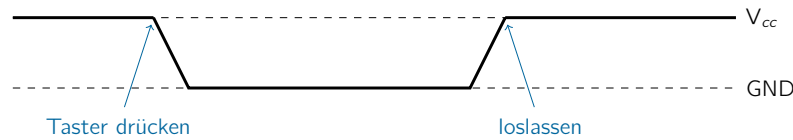


Ablauf eines Interrupts – Details



Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines **idealisierten** Tasters (*active low*)



- Flankengesteuerter Interrupt
 - Interrupt wird durch den Pegelwechsel (Flanke) ausgelöst
 - Häufig ist konfigurierbar, welche Flanke (steigend/fallend/beide) einen Interrupt auslösen soll
- Pegelgesteuerter Interrupt
 - Interrupt wird immer wieder ausgelöst, so lange der Pegel anliegt

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega32 (IRQ $\mapsto$ *Interrupt ReQuest*) [1, S. 45]

- 21 IRQ-Quellen
- einzelde-/aktivierbar
- IRQ $\leadsto$ Sprung an Vektor-Adresse

Vector No.	Program Address ⁽¹⁾	Source	Interrupt Definition
1	\$000 ⁽¹⁾	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$002	INT0	External Interrupt Request 0
3	\$004	INT1	External Interrupt Request 1
4	\$006	INT2	External Interrupt Request 2
5	\$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	\$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	\$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	\$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	\$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	\$012	TIMER1 OVF	Timer/Counter1 Overflow
11	\$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	\$016	TIMER0 OVF	Timer/Counter0 Overflow
13	\$018	SPI, STC	Serial Transfer Complete
14	\$01A	USART, RXC	USART, Rx Complete
15	\$01C	USART, UDRE	USART Data Register Empty
16	\$01E	USART, TXC	USART, Tx Complete
17	\$020	ADC	ADC Conversion Complete
18	\$022	EE_RDY	EEPROM Ready
19	\$024	ANA_COMP	Analog Comparator
20	\$026	TWI	Two-wire Serial Interface
21	\$028	SPM_RDY	Store Program Memory Ready

- Verschaltung SPiCboard ($\hookrightarrow$ [14-14] $\hookrightarrow$ [2-4])
 - INT0 $\mapsto$ PD2 $\mapsto$ Button0 (hardwareseitig entprellt)
 - INT1 $\mapsto$ PD3 $\mapsto$ Button1

Externe Interrupts: Register

- Steuerregister für INT0 und INT1

- GICR **General Interrupt Control Register:** Legt fest, ob die Quellen INT_i IRQs auslösen (Bit INT_i=1) oder deaktiviert sind (Bit INT_i=0) [1, S. 71]

7	6	5	4	3	2	1	0
INT1	INT0	INT2	-	-	-	IVSEL	IVCE
R/W	R/W	R/W	R	R	R	R/W	R/W

- MCUCR **MCU Control Register:** Legt für externe Interrupts INT0 und INT1 fest, wodurch ein IRQ ausgelöst wird (Flanken-/Pegelsteuerung) [1, S. 69]

7	6	5	4	3	2	1	0
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Jeweils zwei *Interrupt-Sense-Control*-Bits (ISC_{i0} und ISC_{i1}) steuern dabei die Auslöser (Tabelle für INT1, für INT0 gilt entsprechendes):

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Externe Interrupts: Verwendung

- Schritt 1:** Installation der *Interrupt-Service-Routine*

- ISR in Hochsprache $\leadsto$ Registerinhalte sichern und wiederherstellen
- Unterstützung durch die *avrlibc*: Makro `ISR( SOURCE_vect )` (Modul `avr/interrupt.h`)

```
#include <avr/interrupt.h>
#include <avr/io.h>

ISR( INT1_vect ) { // invoked for every INT1 IRQ
    static uint8_t counter = 0;
    sb_7seg_showNumber( counter++ );
    if( counter == 100 ) counter = 0;
}

void main() {
    ... // setup
}
```

Externe Interrupts: Verwendung (Forts.)

■ Schritt 2: Konfigurieren der Interrupt-Steuerung

- Steuerregister dem Wunsch entsprechend initialisieren
- Unterstützung durch die avrlibc: Makros für Bit-Indizes (Modul avr/interrupt.h und avr/io.h)

```
...
void main() {
    DDRD  &= ~(1<<PD3);           // PD3: input with pull-up
    PORTD |= (1<<PD3);
    MCUCR &= ~(1<<ISC10 | 1<<ISC11); // INT1: IRQ on level=low
    GICR  |= (1<<INT1);           // INT1: enable
    ...
    sei();                        // global IRQ enable
    ...
}
```

■ Schritt 3: Interrupts global zulassen

- Nach Abschluss der Geräteinitialisierung
- Unterstützung durch die avrlibc: Befehl sei() (Modul avr/interrupt.h)

Externe Interrupts: Verwendung (Forts.)

■ Schritt 4: Wenn nichts zu tun, den Stromsparmodus betreten

- Die sleep-Instruktion hält die CPU an, bis ein IRQ eintrifft
 - In diesem Zustand wird nur sehr wenig Strom verbraucht
- Unterstützung durch die avrlibc (Modul avr/sleep.h):
 - sleep_enable() / sleep_disable(): Sleep-Modus erlauben / verbieten
 - sleep_cpu(): Sleep-Modus betreten



```
#include <avr/sleep.h>
...
void main() {
    ...
    sei();                        // global IRQ enable
    while(1) {
        sleep_enable();
        sleep_cpu();             // wait for IRQ
        sleep_disable();
    }
}
```

Atmel empfiehlt die Verwendung von sleep_enable() und sleep_disable() in dieser Form, um das Risiko eines „versehentlichen“ Betreten des Sleep-Zustands (z. B. durch Programmierfehler oder Bit-Kipper in der Hardware) zu minimieren.

Nebenläufigkeit

Definition: Nebenläufigkeit

Zwei Programmausführungen A und B sind nebenläufig (A|B), wenn für einzelne Instruktionen a aus A und b aus B nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird (a, b oder b, a).

■ Nebenläufigkeit tritt auf durch

- Interrupts
 - IRQs können ein Programm an „beliebiger Stelle“ unterbrechen
- Echt-parallele Abläufe (durch die Hardware)
 - andere CPU / Peripherie greift „jederzeit“ auf den Speicher zu
- Quasi-parallele Abläufe (z. B. Fäden in einem Betriebssystem)
 - Betriebssystem kann „jederzeit“ den Prozessor entziehen

■ Problem: Nebenläufige Zugriffe auf gemeinsamen Zustand

Nebenläufigkeitsprobleme

■ Szenario

- Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
- Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
static volatile uint16_t cars; // photo sensor is connected
                               // to INT2

void main() {
    while(1) {
        waitsec( 60 );
        send( cars );
        cars = 0;
    }
}

ISR(INT2_vect){
    cars++;
}
```

■ Wo ist hier das Problem?

- Sowohl main() als auch ISR lesen und schreiben cars
 - Potentielle Lost-Update-Anomalie
- Größe der Variable cars übersteigt die Registerbreite
 - Potentielle Read-Write-Anomalie

Nebenläufigkeitsprobleme (Forts.)

- Wo sind hier die Probleme?
 - **Lost-Update**: Sowohl `main()` als auch ISR lesen und schreiben `cars`
 - **Read-Write**: Größe der Variable `cars` übersteigt die Registerbreite
- Wird oft erst auf der **Assemblerebene** deutlich

```
void main() {  
    ...  
    send( cars );  
    cars = 0;  
    ...  
}  
  
// photosensor is connected  
// to INT2  
ISR(INT2_vect){  
    cars++;  
}
```

```
main:  
...  
lds r24,cars  
lds r25,cars+1  
rcall send  
sts cars+1, __zero_reg__  
sts cars, __zero_reg__  
...  
  
INT2_vect:  
...  
; save regs  
lds r24,cars ; load cars.lo  
lds r25,cars+1 ; load cars.hi  
adiw r24,1 ; add (16 bit)  
sts cars+1,r25 ; store cars.hi  
sts cars,r24 ; store cars.lo  
...  
; restore regs
```

15-IRQ: 2016-05-20



Nebenläufigkeitsprobleme: *Lost-Update*-Anomalie

```
main:  
...  
lds r24,cars  
lds r25,cars+1  
rcall send  
sts cars+1, __zero_reg__  
sts cars, __zero_reg__  
...  
  
INT2_vect:  
...  
; save regs  
lds r24,cars  
lds r25,cars+1  
adiw r24,1  
sts cars+1,r25  
sts cars,r24  
...  
; restore regs
```

- Sei `cars=5` und an **dieser Stelle** tritt der IRQ (⚡) auf
 - `main` hat den Wert von `cars` (5) bereits in Register gelesen (Register → lokale Variable)
 - `INT2_vect` wird ausgeführt
 - Register werden gerettet
 - `cars` wird inkrementiert → `cars=6`
 - Register werden wiederhergestellt
 - `main` übergibt den **veralteten Wert** von `cars` (5) an `send`
 - `main` nullt `cars` → 1 Auto ist „verloren“ gegangen

15-IRQ: 2016-05-20



Nebenläufigkeitsprobleme: *Read-Write*-Anomalie

```
main:  
...  
lds r24,cars  
lds r25,cars+1  
rcall send  
sts cars+1, __zero_reg__  
sts cars, __zero_reg__  
...  
  
INT2_vect:  
...  
; save regs  
lds r24,cars  
lds r25,cars+1  
adiw r24,1  
sts cars+1,r25  
sts cars,r24  
...  
; restore regs
```

- Sei `cars=255` und an **dieser Stelle** tritt der IRQ (⚡) auf
 - `main` hat bereits `cars=255` Autos mit `send` gemeldet
 - `main` hat bereits das **High-Byte** von `cars` genullt → `cars=255`, `cars.lo=255`, `cars.hi=0`
 - `INT2_vect` wird ausgeführt
 - `cars` wird gelesen und inkrementiert, **Überlauf ins High-Byte**
 - `cars=256`, `cars.lo=0`, `cars.hi=1`
 - `main` nullt das **Low-Byte** von `cars`
 - `cars=256`, `cars.lo=0`, `cars.hi=1`
 - Beim nächsten `send` werden **255 Autos zu viel gemeldet**

15-IRQ: 2016-05-20



Interruptsperrn: Datenflussanomalien verhindern

```
void main() {  
    while(1) {  
        waitsec( 60 );  
        cli();  
        send( cars );  
        cars = 0;  
        sei();  
    }  
}
```

- Wo genau ist das **kritische Gebiet**?
 - **Lesen** von `cars` und **Nullen** von `cars` müssen atomar ausgeführt werden
 - Dies kann hier mit **Interruptsperrn** erreicht werden
 - ISR unterbricht `main`, aber nie umgekehrt → asymmetrische Synchronisation
 - Achtung: Interruptsperrn sollten **so kurz wie möglich** sein
 - Wie lange braucht die Funktion `send` hier?
 - Kann man `send` aus dem kritischen Gebiet herausziehen?

15-IRQ: 2016-05-20



Nebenläufigkeitsprobleme (Forts.)

■ Szenario, Teil 2 (Funktion waitsec())

- Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
- Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
void waitsec( uint8_t sec ) {  
    ...           // setup timer  
    sleep_enable();  
    event = 0;  
    while( !event ) { // wait for event  
        sleep_cpu(); // until next irq  
    }  
    sleep_disable();  
}
```

```
static volatile int8_t event;  
  
// TIMER1 ISR  
// triggers when  
// waitsec() expires  
ISR(TIMER1_COMPA_vect) {  
    event = 1;  
}
```

■ Wo ist hier das Problem?

- **Test, ob nichts zu tun ist**, gefolgt von **Schlafen, bis etwas zu tun ist**
→ Potentielle *Lost-Wakeup*-Anomalie



Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

```
void waitsec( uint8_t sec ) {  
    ...           // setup timer  
    sleep_enable();  
    event = 0;  
    while( !event ) {  
        sleep_cpu();  
    }  
    sleep_disable();  
}
```

```
static volatile int8_t event;  
  
// TIMER1 ISR  
// triggers when  
// waitsec() expires  
ISR(TIMER1_COMPA_vect) {  
    event = 1;  
}
```

■ Angenommen, an **dieser Stelle** tritt der Timer-IRQ (⚡) auf

- waitsec hat bereits festgestellt, dass event **nicht gesetzt** ist
- ISR wird ausgeführt → event **wird gesetzt**
- Obwohl event gesetzt ist, wird der **Schlafzustand betreten**
→ Falls kein weiterer IRQ kommt, **Dornröschenschlaf**



Lost-Wakeup: Dornröschenschlaf verhindern

```
1 void waitsec( uint8_t sec ) {  
2     ...           // setup timer  
3     sleep_enable();  
4     event = 0;  
5     cli();  
6     while( !event ) {  
7         sei();           // kritisches Gebiet  
8         sleep_cpu();  
9         cli();  
10    }  
11    sei();  
12    sleep_disable();  
13 }
```

```
static volatile int8_t event;  
  
// TIMER1 ISR  
// triggers when  
// waitsec() expires  
ISR(TIMER1_COMPA_vect) {  
    event = 1;  
}
```

■ Wo genau ist das **kritische Gebiet**?

- Test auf Vorbedingung und Betreten des Schlafzustands (Kann man das durch Interruptsperrern absichern?)
- Problem: Vor sleep_cpu() müssen IRQs freigegeben werden!
- Funktioniert dank spezieller Hardwareunterstützung:
→ Befehlssequenz sei, sleep wird von der CPU **atomar** ausgeführt



Zusammenfassung

- Interruptbearbeitung erfolgt **asynchron** zum Programmablauf
 - Unerwartet → Zustandssicherung im Interrupt-Handler erforderlich
 - Quelle von Nebenläufigkeit → **Synchronisation erforderlich**
- Synchronisationsmaßnahmen
 - Gemeinsame Zustandsvariablen als **volatile** deklarieren (immer)
 - Zustellung von Interrupts sperren: cli, sei (bei nichtatomaren Zugriffen, die mehr als einen Maschinenbefehl erfordern)
 - Bei **längeren Sperrzeiten** können IRQs **verloren gehen!**
- Nebenläufigkeit durch Interrupts ist eine **sehr große Fehlerquelle**
 - *Lost-Update* und *Lost-Wakeup* Probleme
 - indeterministisch → durch Testen schwer zu fassen
- Wichtig zur Beherrschbarkeit: **Modularisierung** → 12-7
 - Interrupthandler und Zugriffsfunktionen auf gemeinsamen Zustand (**static** Variablen!) in eigenem Modul kapseln.

