

Systemnahe Programmierung in C

20 Unterbrechungen – Nebenläufigkeit

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

<http://sys.cs.fau.de/lehre/ss24>



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;

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

```
// photo sensor is connected
// to INT2

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

- Wo ist hier das Problem?



Nebenläufigkeitsprobleme

■ Szenario

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

```
static volatile uint16_t cars;

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

```
// photo sensor is connected
// to INT2

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

■ Wo ist hier das Problem?

- Sowohl `main()` als auch `ISR` **lesen und schreiben** `cars`
→ Potentielle *Lost-Update*-Anomalie



Nebenläufigkeitsprobleme

■ Szenario

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

```
static volatile uint16_t cars;

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

```
// photo sensor is connected
// to INT2

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(void) {  
    ...  
    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
```



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
```



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



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)



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



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



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**



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
```



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



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



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



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



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**



Interruptsperrn: Datenflussanomalien verhindern

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

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



Interruptsperrn: Datenflussanomalien verhindern

```
void main(void) {  
    while(1) {  
        waitsec(60);
```

```
        send(cars);  
        cars = 0;
```

kritisches Gebiet

```
    }  
}
```

- Wo genau ist das **kritische Gebiet**?
 - Lesen von cars und Nullen von cars müssen atomar ausgeführt werden



Interruptsperrn: Datenflussanomalien verhindern

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

kritisches Gebiet

- 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 \leadsto asymmetrische Synchronisation



Interruptsperrn: Datenflussanomalien verhindern

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

kritisches Gebiet

- 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 \rightsquigarrow 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?



- 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?



- 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



- 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



- 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



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



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 \rightsquigarrow event **wird gesetzt**



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  
6     while (! event) {  
7  
8         sleep_cpu();  
9  
10    }  
11  
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**?



Lost-Wakeup: Dornröschenschlaf verhindern

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



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!



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 \rightsquigarrow Zustandssicherung im Interrupt-Handler erforderlich
 - Quelle von Nebenläufigkeit \rightsquigarrow **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 \rightsquigarrow durch Testen schwer zu fassen
- Wichtig zur Beherrschbarkeit: **Modularisierung** \leftrightarrow 12-7
 - Interrupthandler und Zugriffsfunktionen auf gemeinsamen Zustand (**static** Variablen!) in eigenem Modul kapseln.

