System-Level Programming

20 Interrupts – Concurrency

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

Concurrency

Definition: Concurrency

Two executions A and B of a program are considered to be concurrent $(A|B)$, if for every single instruction a of A and b of B it is not determined, whether a or b is executed first (a, b or b, a).

- Concurrency is induced by
  - Interrupts
    $\leadsto$ IRQs can interrupt a program at an "arbitrary point"
  - Real-parallel sequences (by the hardware)
    $\leadsto$ other CPU / peripheral devices access the memory at "anytime"
  - Quasi-parallel sequences (e. g., threads in an operating system)
    $\leadsto$ OS can preempt tasks "anytime"
- **Problem:** Concurrent access to a **shared state**

Problems with Concurrency

- Scenario
  - a light gate at the entrance of a parking lot counts cars
  - every 60 seconds, the value is transferred to security agency

```
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++;
}
```

- Where does the problem occur?

Problems with Concurrency

- Scenario
  - a light gate at the entrance of a parking lot counts cars
  - every 60 seconds, the value is transferred to security agency

```
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++;
}
```

- Where does the problem occur?
  - both `main()` as well as `ISR` **read and write** `cars`
    $\rightsquigarrow$ potential *lost-update anomaly*

Problems with Concurrency

- Scenario
  - a light gate at the entrance of a parking lot counts cars
  - every 60 seconds, the value is transferred to security agency

```
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++;
}
```

- Where does the problem occur?
  - both `main()` as well as `ISR` read and write `cars`
    ⇝ potential *lost-update anomaly*
  - size of the variable `cars` **is larger than one register**
    ⇝ potential *read-write* anomaly

Problems with Concurrency (continued)

- Where are the problems here?
  - lost-update: both `main()` as well as `ISR` read and write `cars`
  - read-write: size of the variable `cars` is larger than one register
- problem only becomes obvious when looking at the assembly level

```
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
```

Concurrency Problems: *Lost-Update* Anomaly

```
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
```


Concurrency Problems: *Lost-Update* Anomaly

```
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
```

- Let `cars`=5 and let the IRQ () occur at **this point**

Concurrency Problems: *Lost-Update* Anomaly

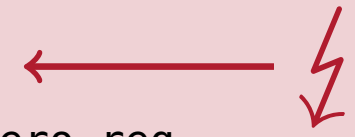
```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

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

- Let `cars`=5 and let the IRQ (↯) occur at **this point**
  - `main` already read the value of `cars` (5) from the register (register $\mapsto$ local variable)

Concurrency Problems: *Lost-Update* Anomaly

```
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
```

- Let `cars`=5 and let the IRQ () occur at **this point**
  - `main` already read the value of `cars` (5) from the register (register $\mapsto$ local variable)
  - `INT2_vect` is executed
    - registers are saved
    - `cars` is incremented $\rightsquigarrow$ `cars`=6
    - registers are restored

Concurrency Problems: *Lost-Update* Anomaly

```
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
```

- Let `cars`=5 and let the IRQ (⚡) occur at **this point**
  - `main` already read the value of `cars` (5) from the register (register $\mapsto$ local variable)
  - `INT2_vect` is executed
    - registers are saved
    - `cars` is incremented $\rightsquigarrow$ `cars`=6
    - registers are restored
  - `main` passes the **old value** of `cars` (5) to `send`

Concurrency Problems: *Lost-Update* Anomaly

```
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
```

- Let `cars`=5 and let the IRQ (↯) occur at **this point**
  - `main` already read the value of `cars` (5) from the register (register ↦ local variable)
  - `INT2_vect` is executed
    - registers are saved
    - `cars` is incremented ⇝ `cars`=6
    - registers are restored
  - `main` passes the **old value** of `cars` (5) to `send`
  - `main` sets `cars` to zero ⇝ **1 car is "lost"**

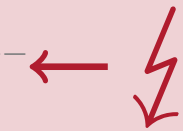
Concurrency Problems: *Read-Write* Anomaly

```
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
```


Concurrency Problems: *Read-Write* Anomaly

```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

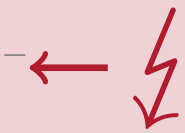


```
INT2_vect:
  ...                ; save regs
  lds r24,cars
  lds r25,cars+1
  adiw r24,1
  sts cars+1,r25
  sts cars,r24
  ...                ; restore regs
```

- Let `cars`=255 and let the IRQ () occur at **this point**

Concurrency Problems: *Read-Write* Anomaly

```
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
```

- Let `cars`=255 and let the IRQ (↯) occur at **this point**
  - `main` has already transmitted `cars`=255 with `send`

Concurrency Problems: *Read-Write* Anomaly

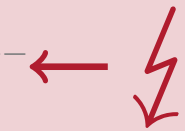
```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

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

- Let `cars`=255 and let the IRQ (↯) occur at **this point**
  - `main` has already transmitted `cars`=255 with `send`
  - `main` has already set the high byte of `cars` to zero
    ⇝ `cars`=255, `cars.lo`=255, `cars.hi`=0

Concurrency Problems: *Read-Write* Anomaly

```
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
```

- Let `cars`=255 and let the IRQ (ϟ) occur at **this point**
  - `main` has already transmitted `cars`=255 with `send`
  - `main` has already set the high byte of `cars` to zero
    ⇝ `cars`=255, `cars.lo`=255, `cars.hi`=0
  - `INT2_vect` is executed
    ⇝ `cars` is read and incremented, **overflow in the high byte**
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1

Concurrency Problems: *Read-Write* Anomaly

```
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
```

- Let `cars`=255 and let the IRQ (↯) occur at **this point**
  - `main` has already transmitted `cars`=255 with `send`
  - `main` has already set the high byte of `cars` to zero
    ⇝ `cars`=255, `cars.lo`=255, `cars.hi`=0
  - `INT2_vect` is executed
    ⇝ `cars` is read and incremented, **overflow in the high byte**
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
  - `main` sets the low byte of `cars` to zero
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
    ⇝ During the next `send`, `main` will transmit **too many cars (255 cars)**

Interrupt Locks: Avoid Data-Flow Anomalies

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

    send(cars);
    cars = 0;

  }
}
```

- Where exactly is the **critical region**?

Interrupt Locks: Avoid Data-Flow Anomalies

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

    send(cars);
    cars = 0;          critical region

  }
}
```

- Where exactly is the **critical region**?
  - Reading of `cars` and setting it to zero have to be executed atomically

Interrupt Locks: Avoid Data-Flow Anomalies

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

critical region

- Where exactly is the **critical region**?
  - Reading of `cars` and setting it to zero have to be executed atomically
  - This can be forced by using **interrupt locks**
    - `ISR` interrupts `main`, never the other way round
      ⇝ asymmetric synchronization (also unilateral synchronization)

Interrupt Locks: Avoid Data-Flow Anomalies

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

critical region

- Where exactly is the **critical region**?
  - Reading of `cars` and setting it to zero have to be executed atomically
  - This can be forced by using **interrupt locks**
    - `ISR` interrupts `main`, never the other way round
      ⇝ asymmetric synchronization (also unilateral synchronization)
  - Attention: keep regions with blocked interrupts **as short as possible**
    - How long does the function `send` take?
    - Can `send` be excluded from the critical region?

Problems with Concurrency (continued)

- Scenario, part 2 (function `waitsec()`)
  - a light gate at the entrance of a parking lot should count cars
  - every 60 seconds, the value is transferred to security agency

```
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;
}
```

- Where exactly does the problem occur?

Problems with Concurrency (continued)

- Scenario, part 2 (function `waitsec()`)
  - a light gate at the entrance of a parking lot should count cars
  - every 60 seconds, the value is transferred to security agency

```
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;
}
```

- Where exactly does the problem occur?
  - **Test, whether sth. is to be done**, followed by sleeping until there is sth. to do

Problems with Concurrency (continued)

- Scenario, part 2 (function `waitsec()`)
  - a light gate at the entrance of a parking lot should count cars
  - every 60 seconds, the value is transferred to security agency

```
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;
}
```

- Where exactly does the problem occur?
  - Test, whether sth. is to be done, followed by **sleeping until there is sth. to do**

Problems with Concurrency (continued)

- Scenario, part 2 (function `waitsec()`)
  - a light gate at the entrance of a parking lot should count cars
  - every 60 seconds, the value is transferred to security agency

```
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;
}
```

- Where exactly does the problem occur?
  - Test, whether sth. is to be done, followed by sleeping until there is sth. to do
    ⇝ Potential lost-wakeup anomaly

Concurrency Problems: *Lost-Wakeup*-Anomaly

```
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;
}
```

- Suppose, at **this point** a timer-IRQ (ϟ) occurs

Concurrency Problems: *Lost-Wakeup*-Anomaly

```
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;
}
```

- Suppose, at **this point** a timer-IRQ (ϟ) occurs
  - `waitsec` already determined that `event` is not set

Concurrency Problems: *Lost-Wakeup*-Anomaly

```
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;
}
```

- Suppose, at **this point** a timer-IRQ (↯) occurs
  - `waitsec` already determined that `event` is not set
  - `ISR` is executed ⇝ `event` **is set to 1**

Concurrency Problems: *Lost-Wakeup*-Anomaly

```
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;
}
```

- Suppose, at **this point** a timer-IRQ (↯) occurs
  - `waitsec` already determined that `event` is not set
  - `ISR` is executed ⇝ `event` **is set to 1**
  - Even though `event` is set to 1, the sleep state is entered
    ⇝ If no further IRQ occurs, **sleeping forever**

Lost-Wakeup: Prevention of Deep Sleep

```
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;
}
```

- Where exactly is the **critical region** located?

Lost-Wakeup: Prevention of Deep Sleep

```
1  void waitsec(uint8_t sec) {
2    ...                  // setup timer
3    sleep_enable();
4    event = 0;
5
6    while (! event) {
7                              critical region
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;
}
```

- Where exactly is the **critical region** located?
  - evaluation of the condition and entry of the sleeping state
    (Can *this* be solved by interrupt blocking?)

Lost-Wakeup: Prevention of Deep Sleep

```
1  void waitsec(uint8_t sec) {
2    ...                  // setup timer
3    sleep_enable();
4    event = 0;
5    cli();
6    while (! event) {
7      sei();                    critical region
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;
}
```

- Where exactly is the **critical region** located?
  - evaluation of the condition and entry of the sleeping state (Can *this* be solved by interrupt blocking?)
  - problem: the IRQs have to be unblocked prior to `sleep_cpu()`!

Lost-Wakeup: Prevention of Deep Sleep

```
1  void waitsec(uint8_t sec) {
2    ...                  // setup timer
3    sleep_enable();
4    event = 0;
5    cli();
6    while (! event) {
7      sei();                    critical region
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;
}
```

- Where exactly is the **critical region** located?
  - evaluation of the condition and entry of the sleeping state (Can *this* be solved by interrupt blocking?)
  - problem: the IRQs have to be unblocked prior to `sleep_cpu()`!
  - works thanks to specific **hardware support**:
    ⇝ sequence `sei`, `sleep` is executed as an **atomic instruction**

Summary

- Handling of interrupts is asynchronous to the program flow
  - unexpected $\leadsto$ current state has to be saved in the interrupt handler
  - source of concurrency $\leadsto$ synchronization required
- Measures for synchronization
  - shared variables shall (always) be declared as `volatile`
  - blocking arrival of interrupts: `cli`, `sei` (when working with non-atomic accesses that translate to more than one machine instruction)
  - Locking for longer times leads to the loss of IRQs!
- Concurrency induced by interrupts is **enormous source for errors**
  - *lost-update* and *lost-wakeup* problems
  - indeterministic $\leadsto$ cannot efficiently be tested for
- Important for complexity management: **modularization** $\hookrightarrow$ 12–7
  - Interrupt handler and functions accessing a shared state (`static` variables!) should be encapsulated in their own module