U5 Fortgeschrittene AVR-Programmierung

- Interrupts
- Synchronisation mit Unterbrechungsbehandlungen
- Stromsparmodi des AVR

U5-1 Externe Interrupts des AVR-µC

1 Funktionsweise (vgl. VL 15-7)

(1) Gerät löst Interrupt aus und setzt entsprechendes Flag
(2) Sind die Interrupts aktiviert und der Interrupt nicht maskiert unterbricht der Interruptcontroller die aktuelle Ausführung
(3) weitere Interrupts werden deaktiviert
(4) aktuelle Position im Programm wird gesichert
(5) Eintrag im Interrupt-Vektor ermitteln
(6) Die Adresse im Interrupt-Vektor wird angesprungen
(= Sprung in den Interrupt-Handler)
(7) am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts

2 Konfiguration des externen Interrupts

- Externe Interrupts durch Pegel(änderung) an bestimmten I/O-Pins
  - ATmega32: 3 Quellen an den Pins PD2, PD3 und PB2
- Pegel- oder flankengesteuert
  - ➤ Abhängig von der jeweiligen Interruptquelle
- Konfiguration über Bits
- Beispiel: Externer Interrupt 2 (INT2)

ISC2	IRQ bei:
0	Fallender Flanke
1	Steigender Flanke

- Dokumentation im ATmega32-Datenblatt
  - Interruptbehandlung allgemein: S. 45-49
  - Externe Interrupts: S. 69-72

2 Konfiguration des externen Interrupts (2)

- Beim ATmega32 verteilen sich die Interrupt Sense Control (`ISC`)-Bits zur Konfiguration der externen Interrupts auf zwei Register:
  - MCU Control Register (`MCUCR`)
  - MCU Control and Status Register (`MCUCSR`)
- Position der ISC-Bits in den Registern durch Makros definiert `ISCn0` und `ISCn1` (`INT0` und `INT1`) oder `ISC2` (`INT2`)
- Beispiel: `INT2` bei ATmega32 für fallende Flanke konfigurieren

```
/* die ISCs für INT2 befinden sich im MCUCSR */
MCUCSR &= ~(1<<ISC2);     /* ISC2 löschen */
```

2 (De-) Aktivieren von Interrupts

- Interrupts können durch die speziellen Maschinenbefehle `sei` und `cli` aktiviert bzw. deaktiviert werden. Die Bibliothek avr-libc bietet hierfür Makros an:
  (`#include <avr/interrupt.h>`)
  - `sei()` (Set Interrupt Flag) - lässt Interrupts zu
  - `cli()` (Clear Interrupt Flag) - blockiert alle Interrupts
- Innerhalb eines Interrupt-Handlers sind automatisch alle Interrupts blockiert, beim Verlassen werden sie wieder deblockiert
- Beim Start des µC sind die Interrupts abgeschaltet

3 (De-) Maskieren von Interrupts

- Das Auslösen einzelner Interrupts kann separat aktiviert (=demaskiert) werden
- Für externe Interrupts ist folgendes Register zuständig:
  - ATmega32: General Interrupt Control Register (`GICR`)
- Die Bitpositionen in diesem Register sind durch Makros `INT`*n* definiert
- Ein gesetztes Bit aktiviert den jeweiligen Interrupt
- Beispiel: Interrupt 2 aktivieren

```
GICR |= (1<<INT2);      /* demaskiere Interrupt 2 */
```


4 Interrupt-Handler

- Installieren eines Interrupt-Handlers wird durch C-Bibliothek unterstützt
- Makro `ISR` (Interrupt Service Routine) zur Definition einer Handler-Funktion (`#include <avr/interrupt.h>`)
- Als Parameter gibt man dem Makro den gewünschten Vektor an, z. B. `INT2_vect` für den externen Interrupt 2
  - verfügbare Vektoren: siehe avr-libc-Doku zu `avr/interrupt.h`
  - verlinkt im Doku-Bereich auf der SPiC-Webseite
- Beispiel: Handler für Interrupt 2 implementieren

```
#include <avr/interrupt.h>
static uint16_t zaehler = 0;

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


5 Implementierung von Interruptbehandlungen

- Interrupts werden durch ein Flag in einem Statusregister vermerkt
- Beim Betreten der Interruptbehandlung wird das Flag zurückgesetzt
- Verlust von Interrupts
  - Wird das Flag vor der Behandlung mehrfach gesetzt, wird der Interrupt dennoch nur einmal behandelt
- Ursachen für den Verlust von Interrupts
  - Während einer Interruptbehandlung sind andere Interrupts gesperrt
  - Interruptsperren zur Synchronisation von kritischem Abschnitten
- Das Problem ist generell nicht zu verhindern
  ☞ Risikominimierung: Interruptbehandlungen sollten möglichst kurz sein
- `sei` sollte niemals in einer Interruptbehandlung ausgeführt werden
  - potentiell endlos geschachtelte Interruptbehandlung
  - Stackoverflow möglich (Vorlesung,vorraussichtlich Kapitel 17)

U5-2 Das `volatile`-Schlüsselwort

- Hauptprogramm wartet auf ein Ereignis, das durch einen Interrupt gemeldet wird (dieser setzt **event** auf **1**)
- Aktive Warteschleife wartet, bis **event != 0**
- Der Compiler erkennt, dass **event** innerhalb der Warteschleife nicht verändert wird
  - ➤ der Wert von **event** wird nur einmal **vor** der Warteschleife aus dem Speicher in ein Prozessorregister geladen
  - ➤ dann wird nur noch der Wert im Register betrachtet
  - ➤ Endlosschleife

```
static uint8_t event = 0;
ISR (INT0_vect) { event = 1; }

void main(void) {
    while(1) {
        while(event == 0) { /* warte auf Event */ }
        /* bearbeite Event */
```


U5-2 Das `volatile`-Schlüsselwort (2)

- Lösung: Unterdrücken der Optimierung mit dem `volatile`-Schlüsselwort

```
static volatile uint8_t event = 0;
ISR (INT0_vect) { event = 1; }

void main(void) {
    while(1) {
        while(event == 0) { /* warte auf Event */ }
        /* bearbeite Event */
    }
}
```

- Wert wird bei jedem Lesezugriff erneut aus dem Speicher geladen

1 Verwendung von volatile

- Fehlendes `volatile` kann zu unerwartetem Programmablauf führen
- Unnötige Verwendung von `volatile` unterbindet Optimierungen des Compilers und führt zu schlechterem Maschinencode
- Korrekte Verwendung von `volatile` ist Aufgabe des Programmierers!
- Verwendung von `volatile` so selten wie möglich, aber so oft wie nötig

U5-3 Synchronisation mit Interrupt-Handlern

- Unterbrechungsbehandlungen führen zu **Nebenläufigkeit**
- Nicht-atomare Modifikation von gemeinsamen Daten kann zu Inkonsistenzen führen
- Einseitige Unterbrechung: Interrupt-Handler unterbricht Hauptprogramm
- Lösung: Synchronisationsprimitiven
  - ➤ hier: zeitweilige Deaktivierung der Interruptbehandlung

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5		
2			
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2			
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6	6	4	6
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
  - ➤ Inkrementierung in der Unterbrechungsbehandlung
  - ➤ Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile uint8_t zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */


/* C-Anweisung: zaehler++ */
4 lds r25, zaehler
5 inc r25
6 sts zaehler, r25
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6	6	4	6
3	4	4	-

2 Beispiel 2: 16-bit-Zugriffe

- Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile uint16_t zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r22, zaehler
2 lds r23, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	
3-7		
2		

2 Beispiel 2: 16-bit-Zugriffe

- Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile uint16_t zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r22, zaehler
2 lds r23, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7		
2		

2 Beispiel 2: 16-bit-Zugriffe

- Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile uint16_t zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r22, zaehler
2 lds r23, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7	0x0100	0x??ff
2		

2 Beispiel 2: 16-bit-Zugriffe

- Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile uint16_t zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r22, zaehler
2 lds r23, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7	0x0100	0x??ff
2	0x0100	0x01ff

- Weitere Problemszenarien?
- Nebenläufige Zugriffe auf Werte >8-bit müssen i.d.R. geschützt werden!

3 Sperren der Unterbrechungsbehandlung beim AVR

- Viele weitere Nebenläufigkeitsprobleme möglich
  - Problemanalyse durch den Anwendungsprogrammierer
  - Vorsicht bei gemeinsamen Daten nebenläufiger Kontrollflüsse
  - Auswahl geeigneter Synchronisationsprimitive
- Lösung hier: Einseitiger Ausschluss durch Sperren der Interrupts
  - Sperrung aller Interrupts (`sei()`, `cli()`)
  - Maskieren einzelner Interrupts (`GICR`-Register)
- Problem: Interrupts während der Sperrung gehen evtl. verloren
  - Kritische Abschnitte sollten so kurz wie möglich gehalten werden.

U5-4 Stromsparmodi von AVR-Prozessoren

- AVR-basierte Geräte oft batteriebetrieben (z.B. Sensorknoten)
- Energiesparen kann die Lebensdauer drastisch erhöhen
- AVR-Prozessoren unterstützen unterschiedliche Powersave-Modi
  - ➤ Deaktivierung funktionaler Einheiten
  - ➤ Unterschiede in der "Tiefe" des Schlafes
  - ➤ Nur aktive funktionale Einheiten können die CPU aufwecken
- Standard-Modus: Idle
  - ◆ CPU-Takt wird angehalten
  - ◆ Keine Zugriffe auf den Speicher
  - ◆ Hardware (Timer, externe Interrupts, ADC, etc) sind weiter aktiv
- Dokumentation im ATmega32-Datenblatt, S. 33-37

1 Nutzung der Sleep-Modi

- Unterstützung aus der avr-libc:
  (`#include <avr/sleep.h>`)
  - ◆ `sleep_enable()` - aktiviert den Sleep-Modus
  - ◆ `sleep_cpu()` - setzt das Gerät in den Sleep-Modus
  - ◆ `sleep_disable()` - deaktiviert den Sleep-Modus
  - ◆ `set_sleep_mode(uint8_t mode)` - stellt den zu verwendenden Modus ein
- Dokumentation von `avr/sleep.h` in avr-libc-Dokumentation
  ☞ verlinkt im Doku-Bereich auf der SPiC-Webseite
- Beispiel

```
#include <avr/sleep.h>
set_sleep_mode(SLEEP_MODE_IDLE); /* Idle-Modus verwenden */
sleep_enable(); /* Sleep-Modus aktivieren */
sleep_cpu(); /* Sleep-Modus betreten */
sleep_disable(); /* "Empfohlen": Sleep-Modus danach deaktivieren */
```


2 Passives Warten auf Unterbrechungen

- Polling bei passivem Warten nicht möglich (warum?)
- Beispiel:

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();
    while(1) {
        while(event == 0) { /* Warte auf Event */
            sleep_cpu();
        }

        /* bearbeite Event */
    }
}
```

- Synchronisation erforderlich?

2 Passives Warten auf Unterbrechungen

- Was passiert, wenn der Interrupt wie unten gezeigt eintrifft?
- Beispiel:

```
volatile static uint8_t event = 0;
ISR (INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();
    while(1) {
        while(event==0) { /* Warte auf Event */
                Interrupt!
            sleep_cpu();
        }

        /* bearbeite Event */
    }
}
```

2 Dornröschenschlaf vermeiden

- Atomarität von Wartebedingungsprüfung und Sleep-Modus-Aktivierung
- Beispiel:

```
volatile static uint8_t event = 0;
ISR (INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();
    while(1) {
        cli();
        while(event==0) { /* Warte auf Event */

            sei();                    kritischer Abschnitt
            sleep_cpu();
            cli();
        }
        sei();
        /* bearbeite Event */
```

- `sei` und die Folgeanweisung werden atomar ausgeführt (notwendig?)