

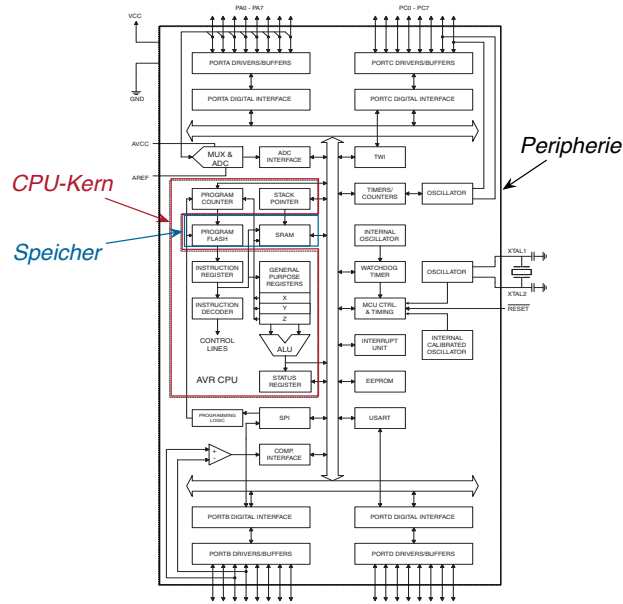
Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

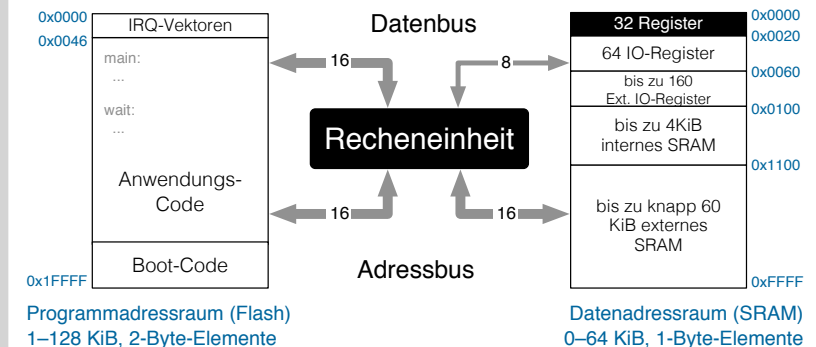
Beispiel ATmega32: Blockschaltbild



Was ist ein μ -Controller?

- **μ -Controller** := Prozessor + Speicher + Peripherie
 - Faktisch ein Ein-Chip-Computersystem $\rightarrow$ SoC (*System-on-a-Chip*)
 - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher $\leadsto$ kostengünstiges Systemdesign
- Wesentliches Merkmal ist die (reichlich) enthaltene Peripherie
 - Timer/Counter (Zeiten/Ereignisse messen und zählen)
 - Ports (digitale Ein-/Ausgabe), A/D-Wandler (analoge Eingabe)
 - PWM-Generatoren (pseudo-analoge Ausgabe)
 - Bus-Systeme: SPI, RS-232, CAN, Ethernet, MLI, I²C, ...
 - ...
- Die Abgrenzungen sind fließend: Prozessor $\longleftrightarrow \mu$ C $\longleftrightarrow$ SoC
 - AMD64-CPU's haben ebenfalls eingebaute Timer, Speicher (Caches), ...
 - Einige μ C erreichen die Geschwindigkeit „großer Prozessoren“

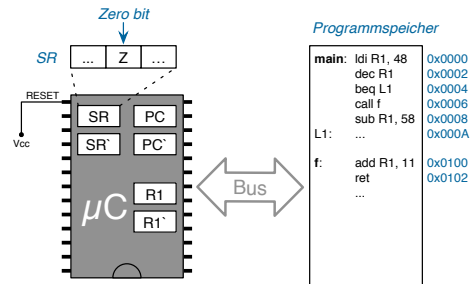
Beispiel ATmega-Familie: CPU-Architektur



- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingebettet $\leadsto$ ansprechbar wie globale Variablen

Zum Vergleich: PC basiert auf von-Neumann-Architektur [$\hookrightarrow$ GDI, 18-10] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

Wie arbeitet ein Prozessor?

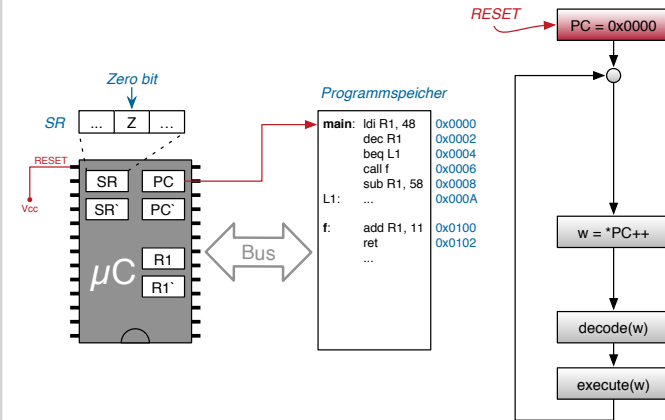


Programmspeicher

```
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1    0x0004
      call f    0x0006
      sub R1, 58 0x0008
      ...      0x000A
f:     add R1, 11 0x0100
      ret      0x0102
```

- Hier am Beispiel eines sehr einfachen Pseudoprozessors
 - Nur zwei Vielweckregister (R1 und R2)
 - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
 - Kein Datenspeicher, kein Stapel → Programm arbeitet nur auf Registern

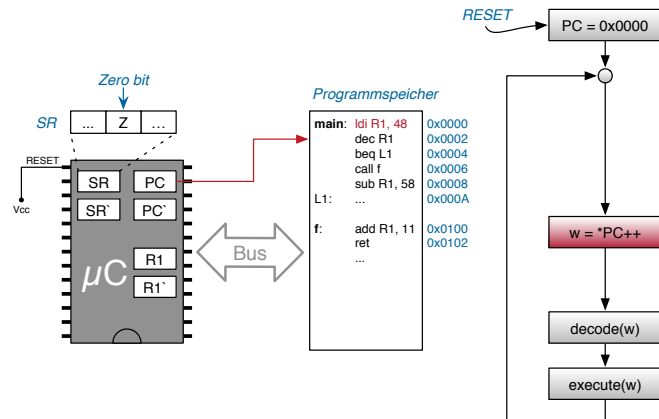
Wie arbeitet ein Prozessor?



Programmspeicher

```
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1    0x0004
      call f    0x0006
      sub R1, 58 0x0008
      ...      0x000A
f:     add R1, 11 0x0100
      ret      0x0102
```

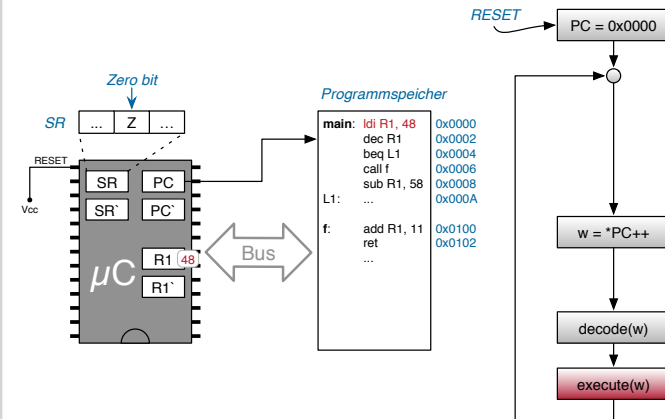
Wie arbeitet ein Prozessor?



Programmspeicher

```
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1    0x0004
      call f    0x0006
      sub R1, 58 0x0008
      ...      0x000A
f:     add R1, 11 0x0100
      ret      0x0102
```

Wie arbeitet ein Prozessor?



Programmspeicher

```
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1    0x0004
      call f    0x0006
      sub R1, 58 0x0008
      ...      0x000A
f:     add R1, 11 0x0100
      ret      0x0102
```


Peripheriegeräte: Beispiele

- Auswahl von typischen Peripheriegeräten in einem μ -Controller
 - Timer/Counter Zählregister, die mit konfigurierbarer Frequenz (Timer) oder durch externe Signale (Counter) erhöht werden und bei konfigurierbarem Zählwert einen Interrupt auslösen.
 - Watchdog-Timer Timer, der regelmäßig neu beschrieben werden muss oder sonst einen RESET auslöst („Totmannknopf“).
 - (A)synchrone serielle Schnittstelle Bausteine zur seriellen (bitweisen) Übertragung von Daten mit synchronem (z. B. RS-232) oder asynchronem (z. B. I²C) Protokoll.
 - A/D-Wandler Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V $\rightarrow$ 10-Bit-Zahl).
 - PWM-Generatoren Bausteine zur Generierung von pulsweiten-modulierten Signalen (pseudo-analoge Ausgabe).
 - Ports Gruppen von üblicherweise 8 Anschlüssen, die auf GND oder Vcc gesetzt werden können oder deren Zustand abgefragt werden kann. $\rightarrow$ 14–12

14-MC: 2015-12-02



Peripheriegeräte – Register

- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
 - Memory-mapped: Register sind in den Adressraum eingebettet; der Zugriff erfolgt über die Speicherbefehle des Prozessors (load, store)
(Die meisten μ C)
 - Port-basiert: Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle in- und out-Befehle
(x86-basierte PCs)
- Die Registeradressen stehen in der Hardware-Dokumentation

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
\$3F (\$SF)	SREG	I	T	H	S	V	N	Z	C	8
\$3E (\$SE)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	11
\$3D (\$SD)	SPL	–	–	–	–	SP4	SP3	SP2	SP1	11
\$3C (\$SC)	OCR0	Timer/Counter0 Output Compare Register								86
\$12 (\$2)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
\$11 (\$1)	DDRD	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0	67
\$10 (\$0)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1, S. 334]

14-MC: 2015-12-02



Peripheriegeräte – Register (Forts.)

- Memory-mapped Register ermöglichen einen komfortablen Zugriff
 - Register $\rightarrow$ Speicher $\rightarrow$ Variable
 - Alle C-Operatoren stehen direkt zur Verfügung (z. B. PORTD++)
- Syntaktisch wird der Zugriff oft durch Makros erleichtert:

```
#define PORTD ( * (volatile uint8_t*) ( 0x12 ) )
```

Adresse: int
Adresse: volatile uint8_t* (Cast $\rightarrow$ 7–17)
Wert: volatile uint8_t (Dereferenzierung $\rightarrow$ 13–4)

PORTD ist damit (syntaktisch) äquivalent zu einer volatile uint8_t-Variablen, die an Adresse 0x12 liegt

- Beispiel

```
#define PORTD (*(volatile uint8_t*)(0x12))  
  
PORTD |= (1<<7); // set D.7  
uint8_t *pReg = &PORTD; // get pointer to PORTD  
*pReg &= ~(1<<7); // use pointer to clear D.7
```

14-MC: 2015-12-02



Registerzugriff und Nebenläufigkeit

- Peripheriegeräte arbeiten **nebenläufig** zur Software
 $\rightarrow$ Wert in einem Hardwareregister kann sich **jederzeit ändern**
- Dies widerspricht einer Annahme des Compilers
 - Variablenzugriffe erfolgen **nur** durch die aktuell ausgeführte Funktion
 $\rightarrow$ Variablen können in Registern zwischengespeichert werden

```
// C code  
#define PIND (*(uint8_t*)(0x10))  
void foo(void) {  
    ...  
    if( !(PIND & 0x2) ) {  
        // button0 pressed  
        ...  
    }  
    if( !(PIND & 0x4) ) {  
        // button 1 pressed  
        ...  
    }  
}  
  
// Resulting assembly code  
foo:  
    lds    r24, 0x0010 // PIND->r24  
    sbrc  r24, 1      // test bit 1  
    rjmp  L1          // button0 pressed  
    ...  
L1:  
    sbrc  r24, 2      // test bit 2  
    rjmp  L2          // button 1 pressed  
    ...  
L2:  
    ret  
  
PIND wird nicht erneut aus dem Speicher geladen. Der Compiler nimmt an, dass der Wert in r24 aktuell ist.
```

14-MC: 2015-12-02



Der volatile-Typmodifizierer

- **Lösung:** Variable `volatile` („flüchtig, unbeständig“) deklarieren
 - Compiler hält Variable nur so kurz wie möglich im Register
 - ~ Wert wird unmittelbar vor Verwendung gelesen
 - ~ Wert wird unmittelbar nach Veränderung zurückgeschrieben

```
// C code                                     // Resulting assembly code
#define PIND \
    (*(volatile uint8_t*)(0x10))
void foo(void) {
    ...
    if( !(PIND & 0x2) ) {
        // button0 pressed
        ...
    }
    if( !(PIND & 0x4) ) {
        // button 1 pressed
        ...
    }
}

foo:
    lds r24, 0x0010 // PIND->r24
    sbrc r24, 1     // test bit 1
    rjmp L1
    // button0 pressed
    ...
L1:
    lds r24, 0x0010 // PIND->r24
    sbrc r24, 2     // test bit 2
    rjmp L2
    ...
L2:
    ret
```

PIND ist `volatile` und wird deshalb vor dem Test erneut aus dem Speicher geladen.

Der volatile-Typmodifizierer (Forts.)

- Die `volatile`-Semantik verhindert viele Code-Optimierungen (insbesondere das Entfernen von **scheinbar unnützem Code**)
- Kann ausgenutzt werden, um aktives Warten zu implementieren:

```
// C code                                     // Resulting assembly code
void wait( void ){
    for( uint16_t i = 0; i<0xffff; i++)
}

// wait:
// compiler has optimized
// "unnneeded" loop
ret
```

volatile!

Achtung: volatile → \$\$\$

Die Verwendung von `volatile` verursacht erhebliche **Kosten**

- Werte können nicht mehr in Registern gehalten werden
- Viele Code-Optimierungen können nicht durchgeführt werden

Regel: `volatile` wird nur in **begründeten Fällen** verwendet

Peripheriegeräte: Ports

- **Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
 - Digitaler Ausgang: Bitwert → Spannungspegel an µC-Pin
 - Digitaler Eingang: Spannungspegel an µC-Pin → Bitwert
 - Externer Interrupt: Spannungspegel an µC-Pin → Bitwert (bei Pegelwechsel) ~ Prozessor führt Interruptprogramm aus
- Die Funktion ist üblicherweise pro Pin konfigurierbar
 - Eingang
 - Ausgang
 - Externer Interrupt (nur bei bestimmten Eingängen)
 - Alternative Funktion (Pin wird von anderem Gerät verwendet)

Beispiel ATmega32: Port/Pin-Belegung

PDIP			
(XCK/T0) PB0	1	40	PA0 (ADC0)
(T1) PB1	2	39	PA1 (ADC1)
(INT2/AIN0) PB2	3	38	PA2 (ADC2)
(OC0/AIN1) PB3	4	37	PA3 (ADC3)
(SS) PB4	5	36	PA4 (ADC4)
(MOSI) PB5	6	35	PA5 (ADC5)
(MISO) PB6	7	34	PA6 (ADC6)
(SCK) PB7	8	33	PA7 (ADC7)
RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7 (TOSC2)
XTAL1	13	28	PC6 (TOSC1)
(RXD) PD0	14	27	PC5 (TDI)
(TXD) PD1	15	26	PC4 (TDO)
(INT0) PD2	16	25	PC3 (TMS)
(INT1) PD3	17	24	PC2 (TCK)
(OC1B) PD4	18	23	PC1 (SDA)
(OC1A) PD5	19	22	PC0 (SCL)
(ICP1) PD6	20	21	PD7 (OC2)

Aus **Kostengründen** ist nahezu jeder Pin **doppelt belegt**, die Konfiguration der gewünschten Funktion erfolgt durch die **Software**.

Beim SPiCboard werden z. B. **Pins 39-40 als ADCs konfiguriert**, um Poti und Photosensor anzuschließen.
PORTA steht daher **nicht zur Verfügung**.

Beispiel ATmega32: Port-Register

- Pro Port x sind drei Register definiert (Beispiel für $x = D$)

- DDRx** **Data Direction Register:** Legt für jeden Pin i fest, ob er als Eingang (Bit $i=0$) oder als Ausgang (Bit $i=1$) verwendet wird.

7	6	5	4	3	2	1	0
DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- PORTx** **Data Register:** Ist Pin i als Ausgang konfiguriert, so legt Bit i den Pegel fest (0=GND sink, 1=Vcc source). Ist Pin i als Eingang konfiguriert, so aktiviert Bit i den internen Pull-Up-Widerstand (1=aktiv).

7	6	5	4	3	2	1	0
PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- PINx** **Input Register:** Bit i repräsentiert den Pegel an Pin i (1=high, 0=low), unabhängig von der Konfiguration als Ein-/Ausgang.

7	6	5	4	3	2	1	0
PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
R	R	R	R	R	R	R	R

Verwendungsbeispiele: $\rightarrow$ 3-4 und $\rightarrow$ 3-7 [1, S.66]

14-MC: 2015-12-02



Strukturen: Motivation

- Jeder Port wird durch *drei* globale Variablen verwaltet

- Es wäre besser diese **zusammen zu fassen**
- „problembezogene Abstraktionen“
- „Trennung der Belange“

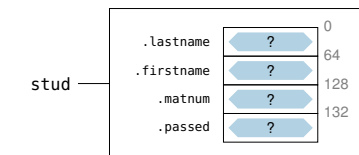
$\rightarrow$ 4-1
 $\rightarrow$ 12-4

- Dies geht in C mit **Verbundtypen** (Strukturen)

```
// Structure declaration
struct Student {
    char lastname[64];
    char firstname[64];
    long matnum;
    int passed;
};

// Variable definition
struct Student stud;
```

Ein **Strukturtyp** fasst eine Menge von Daten zu einem gemeinsamen Typ zusammen.
Die Datenelemente werden **hintereinander** im Speicher abgelegt.



14-MC: 2015-12-02



Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden $\rightarrow$ 13-8

```
struct Student {
    char lastname[64];
    char firstname[64];
    long matnum;
    int passed;
};
```

```
struct Student stud = { "Meier", "Hans",
                        4711, 0 };
```

Die Initialisierer werden nur über ihre Reihenfolge, nicht über ihren Bezeichner zugewiesen.
 $\leadsto$ **Potentielle Fehlerquelle** bei Änderungen!

- Analog zur Definition von **enum**-Typen kann man mit **typedef** die Verwendung vereinfachen $\rightarrow$ 6-8

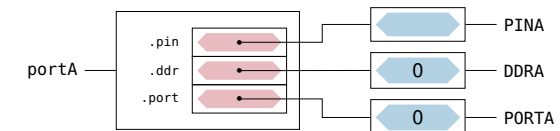
```
typedef struct {
    volatile uint8_t *pin;
    volatile uint8_t *ddr;
    volatile uint8_t *port;
} port_t;
```

```
port_t portA = { &PINA, &DDRA, &PORTA };
port_t portD = { &PIND, &DDRD, &PORTD };
```

14-MC: 2015-12-02



Strukturen: Elementzugriff



- Auf Strukturelemente wird mit dem **.-Operator** zugegriffen $\approx$ Java

```
port_t portA = { &PINA, &DDRA, &PORTA };
```

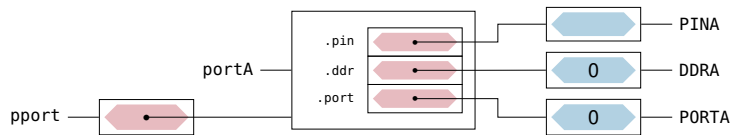
```
*portA.port = 0; // clear all pins
*portA.ddr = 0xff; // set all to input
```

Beachte: **.** hat eine höhere Priorität als *****

14-MC: 2015-12-02



Strukturen: Elementzugriff



- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

```
port_t * pport = &portA; // p --> portA

*(pport).port = 0; // clear all pins
*(pport).ddr = 0xff; // set all to output
```

- Mit dem `->`-Operator lässt sich dies vereinfachen `s->m` $\equiv$ `(*s).m`

```
port_t * pport = &portA; // p --> portA

pport->port = 0; // clear all pins
pport->ddr = 0xff; // set all to output
```

`->` hat **ebenfalls** eine höhere Priorität als `*`

Strukturen als Funktionsparameter

- Im Gegensatz zu Arrays werden Strukturen *by-value* übergeben

```
void initPort( port_t p ){
    *p.port = 0; // clear all pins
    *p.ddr = 0xff; // set all to output

    p.port = &PORTD; // no effect, p is local variable
}

void main(){ initPort( portA ); ... }
```

- Bei größeren Strukturen wird das **sehr ineffizient**
 - Z. B. Student ($\leftrightarrow$ 14-15): Jedes mal 134 Byte allozieren und kopieren
 - Besser man übergibt einen Zeiger auf eine konstante Struktur

```
void initPort( const port_t *p ){
    *p->port = 0; // clear all pins
    *p->ddr = 0xff; // set all to output

    // p->port = &PORTD; compile-time error, *p is const!
}

void main(){ initPort( &portA ); ... }
```

Bit-Strukturen: Bitfelder

- Strukturelemente können auf Bit-Granularität festgelegt werden
 - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
 - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen

- Beispiel

- MCUCR

MCU Control Register: Steuert Power-Management-Funktionen und Auslöser für externe Interrupt-Quellen INT0 und INT1. [1, S. 36+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

```
typedef struct {
    uint8_t ISC0 : 2; // bit 0-1: interrupt sense control INT0
    uint8_t ISC1 : 2; // bit 2-3: interrupt sense control INT1
    uint8_t SM : 3; // bit 4-6: sleep mode to enter on sleep
    uint8_t SE : 1; // bit 7 : sleep enable
} MCUCR_t;
```

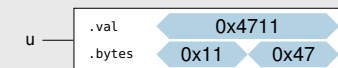
Unions

- In einer Struktur liegen die Elemente *hintereinander* im Speicher, in einer Union hingegen *übereinander*
 - Wert im Speicher lässt sich verschieden (Typ)-interpretieren
 - Nützlich für bitweise Typ-Casts
- Beispiel

```
void main(){
    union {
        uint16_t val;
        uint8_t bytes[2];
    } u;

    u.val = 0x4711;

    // show high-byte
    sb_7seg_showHexNumber( u.bytes[1] );
    ...
    // show low-byte
    sb_7seg_showHexNumber( u.bytes[0] );
    ...
}
```



47

11

Unions und Bit-Strukturen: Anwendungsbeispiel

- Unions werden oft mit Bit-Feldern kombiniert, um ein Register wahlweise „im Ganzen“ oder bitweise ansprechen zu können

```
typedef union {
    volatile uint8_t reg; // complete register
    volatile struct {
        uint8_t ISC0 : 2; // components
        uint8_t ISC1 : 2;
        uint8_t SM : 3;
        uint8_t SE : 1;
    };
} MCUCR_t;

void foo( void ) {
    MCUCR_t *mcucr = (MCUCR_t *) (0x35);
    uint8_t oldval = mcucr->reg; // save register
    ...
    mcucr->ISC0 = 2; // use register
    mcucr->SE = 1; // ...
    ...
    mcucr->reg = oldval; // restore register
}
```

