

# Systemnahe Programmierung in C

## 17 $\mu$ C-Systemarchitektur – Peripherie

**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>



# Was ist ein $\mu$ -Controller?

- **$\mu$ -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  $\rightsquigarrow$  kostengünstiges Systemdesign
- Wesentliches Merkmal sind die (reichlich) enthaltenen Ein-/Ausgabe-Komponenten (Peripherie)
- Die Abgrenzungen sind fließend: Prozessor  $\longleftrightarrow$   $\mu$ C  $\longleftrightarrow$  SoC
  - AMD64-CPU's haben ebenfalls eingebaute Timer, Speicher (Caches), ...
  - Einige  $\mu$ C erreichen die Geschwindigkeit „großer Prozessoren“

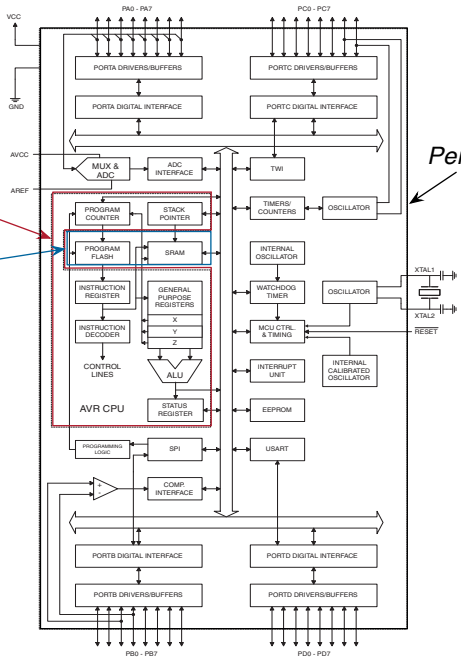


# Beispiel ATmega32: Blockschaltbild

CPU-Kern

Speicher

Peripherie



- **Peripheriegerät:** Hardwarekomponente, die sich „außerhalb“ der Zentraleinheit eines Computers befindet
  - Traditionell (PC): Tastatur, Bildschirm, ...  
(→ physisch „außerhalb“)
  - Allgemeiner: Hardwarefunktionen, die nicht direkt im Befehlssatz des Prozessors abgebildet sind  
(→ logisch „außerhalb“)
- Peripheriebausteine werden über **I/O-Register** angesprochen
  - Kontrollregister: Befehle an / Zustand der Peripherie wird durch **Bitmuster** kodiert (z. B. **DDRD** beim ATmega)
  - Datenregister: Dienen dem eigentlichen Datenaustausch (z. B. **PORTD**, **PIND** beim ATmega)
  - Register sind häufig für entweder nur Lesezugriffe (*read-only*) oder nur Schreibzugriffe (*write-only*) zugelassen



- Auswahl von typischen Peripheriegeräten in einem  $\mu$ -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. I<sup>2</sup>C) Protokoll.
  - A/D-Wandler Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V  $\leftrightarrow$  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.  $\leftrightarrow$  17-11
  - Bus-Systeme SPI, RS-232, CAN, Ethernet, MLI, I<sup>2</sup>C, ...



- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
  - Memory-mapped: Register sind in den Adressraum eingeblenet; (Die meisten  $\mu C$ ) der Zugriff erfolgt über die Speicherbefehle des Prozessors (`load`, `store`)
  - Port-basiert: Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle `in`- und `out`-Befehle



# Peripheriegeräte – Register

- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
  - Memory-mapped: Register sind in den Adressraum eingebunden; (Die meisten  $\mu C$ ) der Zugriff erfolgt über die Speicherbefehle des Prozessors (load, store)
  - Port-basiert: Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle in- und out-Befehle
- 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 (\$5F)	SREG	I	T	H	S	V	N	Z	C	8
\$3E (\$5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	11
\$3D (\$5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
\$3C (\$5C)	OCR0	Timer/Counter0 Output Compare Register								86
\$12 (\$32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
\$11 (\$31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	67
\$10 (\$30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1]



- 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
DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
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/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Verwendungsbeispiele:  $\leftrightarrow$  3-7 und  $\leftrightarrow$  3-11 [1]





- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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 )
```



- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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
```



- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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  $\mapsto$  7-19)



- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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  $\leftrightarrow$  7-19)

Wert: volatile uint8\_t (Dereferenzierung  $\leftrightarrow$  13-4)



- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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  $\leftrightarrow$  7-19)

Wert: volatile uint8\_t (Dereferenzierung  $\leftrightarrow$  13-4)

PORTD ist damit (syntaktisch) äquivalent zu einer volatile uint8\_t-Variablen, die an Adresse 0x12 liegt



- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register  $\mapsto$  Speicher  $\mapsto$  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  $\leftrightarrow$  7-19)

Wert: volatile uint8\_t (Dereferenzierung  $\leftrightarrow$  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
```



# Registerzugriff und Nebenläufigkeit

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



# Registerzugriff und Nebenläufigkeit

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

```
// C code                                     // Resulting assembly code
#define PIND (*(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:
    sbrc  r24, 2      // test bit 2
    rjmp  L2
    ...
L2:
    ret
```





# Registerzugriff und Nebenläufigkeit

- Peripheriegeräte arbeiten **nebenläufig** zur Software  
↪ Wert in einem Hardwareregister kann sich **jederzeit ändern**
- Dies widerspricht einer Annahme des Compilers
  - Variablenzugriffe erfolgen **nur** durch die aktuell ausgeführte Funktion  
↪ 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
    ...
L2:
    ret
```

PIND wird nicht erneut aus dem Speicher geladen. Der Compiler nimmt an, dass der Wert in r24 aktuell ist.



- **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



# 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
#define PIND \
    (*(volatile 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:
    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.



- 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) {                          wait:
    for (uint16_t i = 0; i < 0xffff; i++); // compiler has optimized
}                                           // "unneeded" loop
                                           ret
```



- 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) {                          wait:
    for (uint16_t i = 0; i < 0xffff; i++); // compiler has optimized
}                                           // "unneeded" loop
                                           ret
```

**volatile!**



# 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) {                          wait:
    for (uint16_t i = 0; i < 0xffff; i++); // compiler has optimized
}                                           // "unneeded" 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

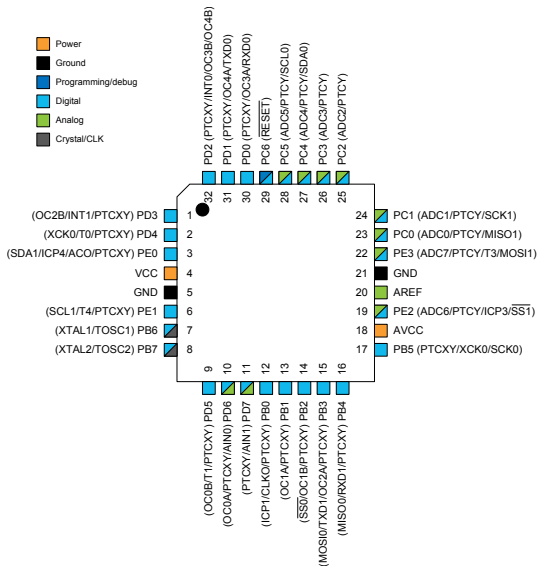


# Peripheriegeräte: Ports

- **Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
  - Digitaler Ausgang: Bitwert  $\mapsto$  Spannungspegel an  $\mu$ C-Pin
  - Digitaler Eingang: Spannungspegel an  $\mu$ C-Pin  $\mapsto$  Bitwert
  - Externer Interrupt: Spannungspegel an  $\mu$ C-Pin  $\mapsto$  Bitwert  
(bei Pegelwechsel)  $\rightsquigarrow$  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 ATmega328PB: Port/Pin-Belegung



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

Beim SPiCboard werden z. B. **Pins 23–24 als ADCs konfiguriert**, um Poti und Photosensor anzuschließen.

Diese Pins stehen daher für PORTC **nicht zur Verfügung**.

