

# Microcontroller Kurs Programmieren

# General Purpose Input Output (GPIO)

- Jeder der Pins der vier I/O Ports kann als Eingabe- oder Ausgabe-leitung benutzt werden.
- Dazu wird das Data Direction Register (DDRx) benutzt.
  - Zum Programmieren stehen zur Verfügung DDRA, DDRB, DDRC, DDRD
  - Wird ein Bit im DDRx eine 0 geschrieben wird die entsprechende Leitung als Eingabeleitung (Input) geschaltet.
  - Wird ein Bit auf 1 gesetzt wird die entsprechende Leitung zur Ausgabeleitung (Output).
  - Die Anweisung DDRD=0xFF; macht alle Leitungen am Port D zu Ausgangsleitungen
  - Die Anweisung DDRC=0xC; macht die Leitungen PC2 und PC3 zu Ausgabeleitungen, alle anderen Leitungen sind damit als Eingänge geschaltet
  - Beim Einschalten des Prozessors sind alle Leitungen automatisch als Input aktiv.

# Output

- Eine Ausgabeeleitung kann entweder die Prozessorspannung ausgeben(high) oder auf Ground gezogen werden(low).
- Um die Ausgabeeleitung zu schalten muss das entsprechende Bit im PORTx Register gesetzt werden.
  - Es stehen zur Verfügung die Register PORTA,PORTB,PORTC,PORTD
  - Die Anweisung PORTD=0xFF; schaltet alle Ausgabeeleitungen im PORTD auf high (Voraussetzung DDRD= 0xFF)
  - Die Anweisung PORTD=0xF0; schaltet die Ausgabeeleitungen PD4..PD7 auf high und PD0 .. PD3 auf low.

# Input

- Von einer Eingabeleitung kann gelesen werden ob eine Spannung anliegt (high) oder die Leitung auf Ground liegt (low).
- Der Zustand einer Leitung kann mit dem Register PINx gelesen werden
  - Es stehen zur Verfügung die Register PINA,PINB,PINC,PINE
  - Die Anweisung `PORTD=PINA;` schaltet jede Ausgabeleitung von PD0 .. PD7 auf den Status der auf der Leitungen PA0..PA7  
!!! wichtig `DDRD=0xFF` und `DDRA=0` !!!!
- Mit dem Register PORTx kann ein interner Pullup Widerstand eingeschaltet werden (siehe Hardware Beschreibung)

# GPIO Zusammenfassung

	Input	Output
DDRx	Um eine Leitung als Input zu konfigurieren, muss in das entsprechende Bit eine 0 geschrieben werden.	Um eine Leitung als Output zu konfigurieren, muss in das entsprechende Bit eine 1 geschrieben werden.
PORTx	Schaltet den Pullup ein/aus.	Schaltet den Ausgang ein (high/1) bzw. aus(low/0).
PINx	Wird benutzt zum Lesen ob ein Signal am Eingang anliegt.	Wird nicht benutzt.

# Logische Operatoren

- Im Prozessor kennen wir in der Regel zwei Zustände; Spannung an (high , 1) oder Spannung aus (low , 0)
  - Etwas ähnlich haben wir z.B. In der Aussagenlogik. Ein Satz kann entweder wahr oder falsch sein.
- Um mit diesen Systemen umzugehen benutzen wir drei sogenannte Operatoren.
- Beim Programmieren benötigen wir
  - sowohl Binäroperatoren, die auf einzelne Bits eines Bytes wirken; z.B: bei I/O Leitungen, wir haben immer nur alle 8 bit auf einmal
  - als auch aussagenlogische Operatoren, die nur zwischen false und true unterscheiden; z.B.: bei if – Anweisungen oder in while Schleifen. Dabei ist der Wert 0 == false und ein Wert ungleich 0 immer true.

# Not / Nicht

- Der Operator Not negiert den Wert von A

A	Not A
0/false/low	1/high/true
1/high/true	0/false/low

- Als Binäroperator wird das Zeichen `~` verwendet
  - Es gilt `~0xF0` ist gleich `0x0F`
- Als ausagenlogischer Operator wird das Zeichen `!` verwendet.

# And / und

A	B	A and B
0/false/low	0/false/low	0/false/low
0/false/low	1/high/true	0/false/low
1/high/true	0/false/low	0/false/low
1/high/true	1/high/true	1/high/true

- Als Binäroperator wird das Zeichen & oder das Wort bitand verwendet
  - z.B.: 0xF3 bitand 0xFC ist gleich 0xF0;
- Als aussagenlogischer Operator wird das Zeichen && oder das Wort and verwendet
  - z.B.: true and false ist gleich false

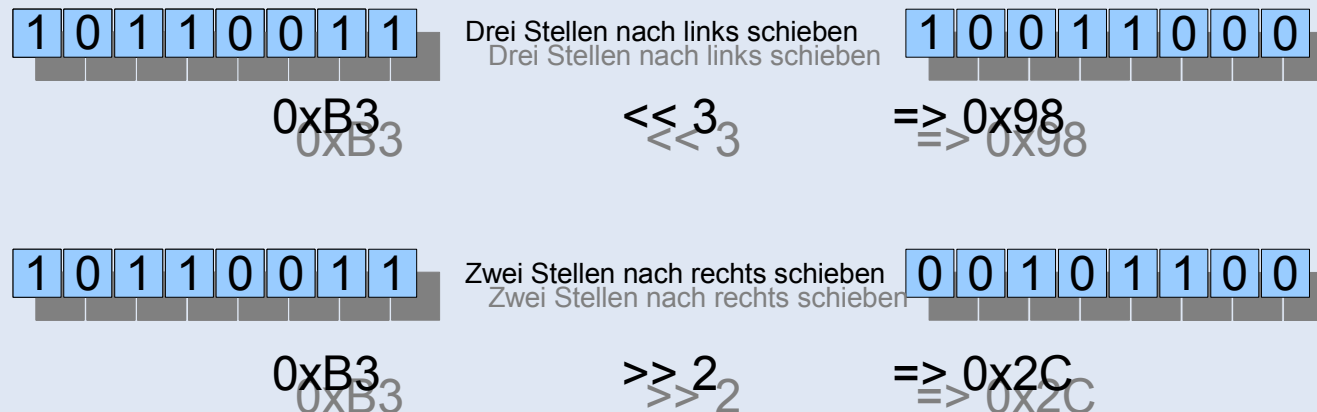
# OR / oder

A	B	A or B
0/false/low	0/false/low	0/false/low
0/false/low	1/high/true	1/high/true
1/high/true	0/false/low	1/high/true
1/high/true	1/high/true	1/high/true

- Als Binäroperator wird das Zeichen | oder das Wort bitor verwendet
  - z.B.: 0xF3 bitor 0xFC ist gleich 0xFF;
- Als aussagenlogischer Operator wird das Zeichen || oder das Wort or verwendet
  - z.B.: true or false ist gleich true

# Bitshift Operatoren

- Es gibt noch zwei weitere wichtige Operationen im Umgang mit Bits
  - Schieben nach links ( shift left; <<)
  - Schieben nach rechts (shift right; >>)
- Mit diesen Operatoren kann man bits innerhalb einer Zahl verschieben.



# Variablen

- Um in einem Programm Daten zwischenspeichern benötigen wir Variablen.
- In C/C++ muss jede Variable vor der Benutzung definiert werden.
  - ```
uint8_t i=PINC; // definiert eine ganzzahlige Variable vom typ uint8_t
                // und weist ihr den Wert von PINC zu
i=i+1;          // zählt i um eins hoch (alternative ++i)
PORTD=i;       // weist den Wert von i auf PORTD zu
```
- Gültigkeitsbereich von Variablen
  - Variablen sind nur innerhalb des Blocks { ... } gültig in dem sie definiert sind.
  - Variablen die Ausserhalb aller Blöcke definiert sind gelten für das gesamte Programm

# Ganzzahlige Datentypen

- Jede Variable braucht einen Datentyp, der die Art des Inhalts beschreibt.
  - Standardtypen für ganze Zahlen sind: char, short, int, long, unsigned char, unsigned short, unsigned int und unsigned long
  - Wir benutzen die neuen Typen für ganze Zahlen:

| Typ      | Beschreibung                             | Wertebereich                                 |
|----------|------------------------------------------|----------------------------------------------|
| int8_t   | Ganzzahliger 8-bit wert mit Vorzeichen   | -128 .. +127; $-2^7 .. +(2^7 - 1)$           |
| uint8_t  | Ganzzahliger 8-bit wert ohne Vorzeichen  | 0 .. 255; $0 .. (2^8 - 1)$                   |
| int16_t  | Ganzzahliger 16-bit wert mit Vorzeichen  | -32768 .. +32767; $-2^{15} .. +(2^{15} - 1)$ |
| uint16_t | Ganzzahliger 16-bit wert ohne Vorzeichen | 0 .. 65535; $0 .. (2^{16} - 1)$              |
| int32_t  | Ganzzahliger 32-bit wert mit Vorzeichen  | $-2^{31} .. +(2^{31} - 1)$                   |
| uint32_t | Ganzzahliger 32-bit wert ohne Vorzeichen | $0 .. (2^{32} - 1)$                          |

# Weitere Datentypen

- Gleitkommatypen
  - float, double
  - Spielen bei Mikrocontrollerprogrammierung keine grosse Rolle, da der Prozessor keine Gleitkommazahlen unterstützt. Deshalb müssen die emuliert werden. Das führt zu großen und langsamen Programmen.
- Zeiger (Pointer)
  - Zeiger auf Speicherstellen. Werden später bei Bedarf erklärt.
- Void
  - Leerer Datentyp. Wird nur verwendet im Zusammenhang mit Pointern und zur Rückgabe bei Funktionen.

# Funktionen

- Eine Funktion ist ein Block im Programm, der von anderer Stelle aufgerufen werden kann.
- Eine Funktion hat einen
  - eindeutigen Namen (mean),
  - eine Rückgabewert (uint8\_t),
  - Parameter (uint8\_t a und uint8\_t b),
  - und einen Block zur Implementierung.
- Eine Funktion wird kann jetzt von jeder Stelle des Programmes aufgerufen werden, die hinter der Funktionsdeklaration liegt.

```
uint8_t mean(uint8_t a, uint8_t b)
{
    uint16_t c= a; // !!! wertebereich
    c=c+b;
    c=c/2;
    return c;
}
```

```
int main()
{
    ...
    uint8_t x=mean(250,254);
    ...
    return 0;
}
```

# Funktionen

- Soll eine Funktion keinen Wert zurückgeben, ist der Typ des Rückgabewertes void
- Die Funktion "int main()" wird beim Start eines Programmes immer automatisch aufgerufen
- Es ist nicht möglich Funktionen innerhalb von Funktionen zu implementieren.

# Timer

- Aufgabe: bringe die LEDs am PORT D zum Blinken.
  - 1. Lösungsmöglichkeit Schleife im Programm

z.B:

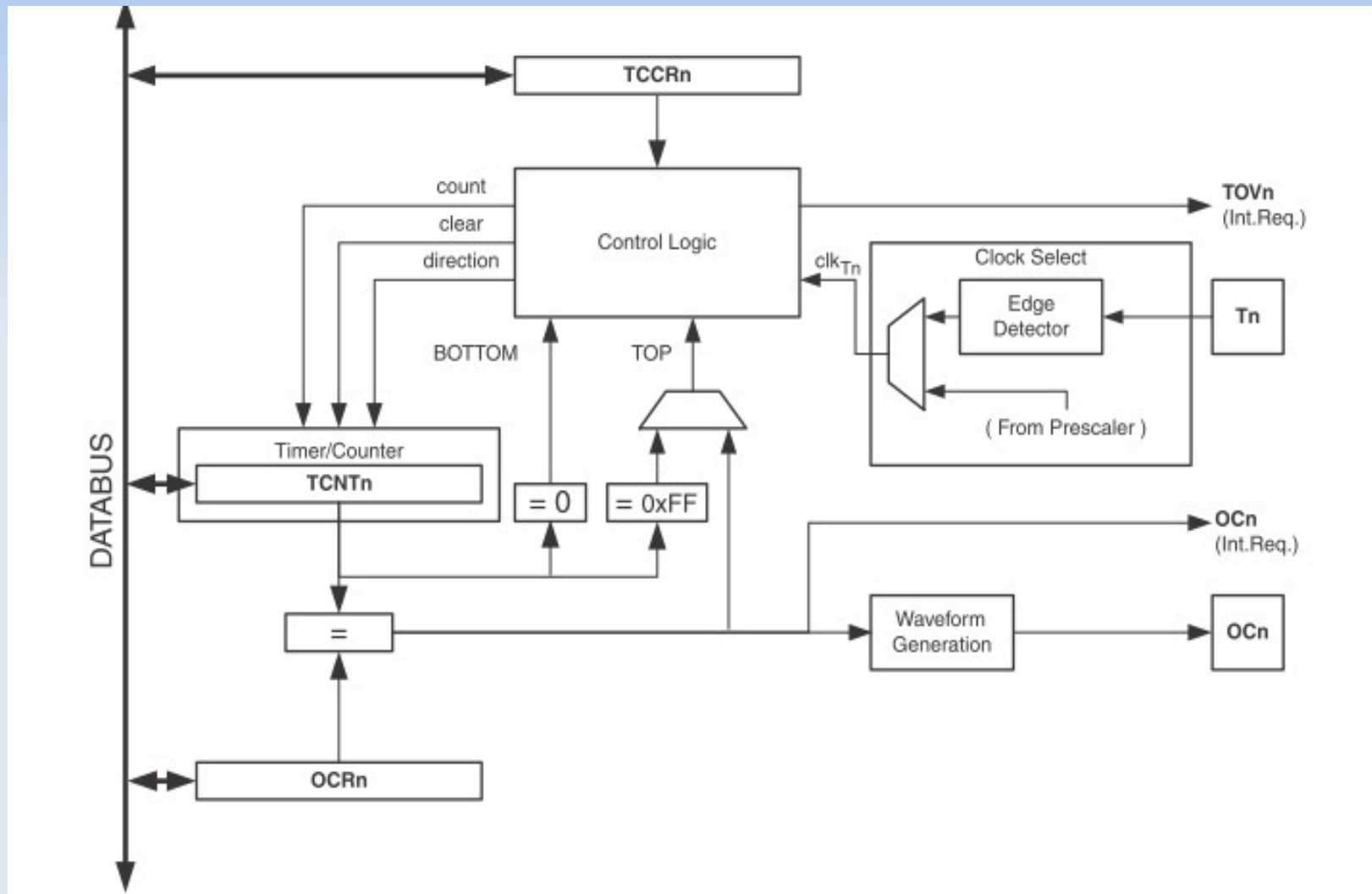
```
#include <avr/io.h>

int main()
{
    DDRD=0xFF; // schaltet alle
                //pins von Port D auf Ausgabe (ouput)
    PORTD=0xFF; // schaltet alle pins von PortD ein
    while(true) // endlosschleife
    {
        uint32_t i;
        for(i=0;i<100000;++i)// Leere Schleife zum Warten
        {}
        PORTD=~PORTD;
    }
    return 0;
}
```

# Timer

- Nachteile dieser Lösung; Zeit ist schwierig zu ermitteln
- Die atmega Prozessoren haben eingebaute Timer.
  - Der ATMEGA16 hat zwei 8-bit und einen 16-bit Timer
- Timer sind sehr vielfältig, damit auch kompliziert. Wir starten bei einer einfachen Benutzung und machen später ein paar schwierigere Versuche
- Wichtige Informationen finden sich im Datasheet:  
[http://www.atmel.com/dyn/resources/prod\\_documents/2466.pdf](http://www.atmel.com/dyn/resources/prod_documents/2466.pdf)

# Timer Übersicht



# Nutzung des 16 bit Timers

- Aufgabe: bringe die LEDs am PORT D zum Blinken.
  - 2. Lösungsmöglichkeit  
Schleife im Programm  
mit Timer  
z.B:

```
#include <avr/io.h>

int main()
{
    DDRD=0xFF; // schaltet alle
                //pins von Port D auf Ausgabe (ouput)
    TCCR1B=(1<<CS12) bitor (1<<CS10);
    // setzt die clock source auf den Prozessor clock
    //mit einem prescaler von 1024
    while(true) // endlosschleife
    {
        TCNT1=0; // setzt den Counter auf 0
        while (TCNT1<7812) // wartet bis der
            // counter den Wert 7812 erreicht hat
        {}
        PORTD=~PORTD;
    }
    return 0;
}
```

# Interrupts

- Bisher besteht immer noch die größte Aufgabe des Prozessors darin zu warten. Wir können den Prozessor nicht wirklich während diese Zeit für andere Aufgaben nutzen, da wir ja immer das TCNT1 register lesen müssen.
- Lösung für das Problem bieten Interrupts
- Interrupts unterbrechen die normale Programmausführung des Prozessors und rufen eine vorher definierte Funktion auf
- Nachdem die Funktion beendet ist, setzt der Prozessor das Programm an genau der Stelle fort an der er es unterbrochen hatte.
- Wir können die Timer als Quelle für Interrupts verwenden.
- Genaue Beschreibung der für den jeweiligen Prozessor möglichen interrupt Funktionen findet ihr in avr-libc dokumentation im WinAVR Verzeichnis

# Die Interrupt Funktion

- Die Interrupt Funktion wird definiert durch `ISR(isr_vector)`
- Wir werden den Timer so programmieren, dass der TCNT1 bis zum Wert 7812 läuft, dann der Prozessor den Interrupt auslöst und den TCNT1 wieder zurücksetzt auf 0 (CTC Mode).
- Dazu benötigen wir den `isr_vector` `TIMER1_COMPA_vect`

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_COMPA_vect)
{
    PORTD=~PORTD;
}
```

Wichtig : `#include <avr/interrupt.h>` nicht vergessen  
Wichtig : `#include <avr/interrupt.h>` nicht vergessen

# Initialisierung des Interrupt

- Das komplizierte folgt jetzt:  
Die Timer muss so konfiguriert werden, dass er den Interrupt auslöst. Dabei hilft nur das Datasheet ( oder eine Vorlage).

```
int main()
{
    DDRD=0xFF; // schaltet alle PORTD auf output
    TCCR1B=(1<<WGM12) // setzt den CTC mode
        bitor (1<<CS12) bitor (1<<CS10); // setzt die clock source auf den Prozessor clock
        //mit einem prescaler von 1024
    OCR1A=7812; // setzt den Wert bis zu dem gezählt wird
    TIMSK=(1<<OCIE1A); // schaltet den Timer interrupt frei
    sei(); // schaltet interrupts im prozessor ein
    while(true) // endlosschleife
    {
        // hhhmmm nothing to do
    }
    return 0;
}
```

# volatile

- Der Prozessor hat Register, in denen er die Daten ablegt mit denen er arbeitet. Braucht er die Daten nicht mehr oder braucht er die Register für andere Daten, schreibt er sie wieder in den Speicher. Braucht er sie wieder liest er sie zuerst aus dem Speicher in ein Register.
- Problem:
  - Wird der Wert einer Variablen in einem Programm verändert, heisst das noch nicht, dass dieser Wert auch schon im Speicher steht.
  - Greift eine Interrupt Service Routine auf diese Variable zu holt sie sich den Wert aus dem Speicher.
- Lösung:
  - Wird eine Variable mit dem Schlüsselwort *volatile* deklariert, muss sie der Prozessor vor den Benutzung immer aus dem Speicher lesen und nach der Veränderung immer in den Speicher zurückschreiben.
  - Beispiel: `volatile uint8_t i=0;`

# Interrupts ein-/aus-schalten

- Man kann im Prozessor Interrupts grundsätzlich ein- und aus- schalten. Das wird gemacht indem im Register SREG das bit 7 (I) gesetzt (eingeschaltet) oder gelöscht (ausgeschaltet) wird.
- Dazu gibt es zwei Funktionen
  - sei() zum Einschalten
  - cli() zum Ausschalten
- Beim Start des Prozessors sind die Interrupts ausgeschaltet.
- Problem:
  - Schreibt oder liest ein Programm eine 16-bit Variable sind bei einem 8-bit Prozessor dafür zwei Operationen notwendig.
  - Grundsätzlich kann bei eingeschalteten Interrupts zwischen den beiden Lese/Schreibzugriffen die Interrupt Service Routine aufgerufen werden.
  - Wird die Variable auch in der Interrupt Service Routine benutzt, kann beliebiger Unsinn entstehen
  - Deshalb ist es wichtig in diesem Fall im Programm die Interrupts zuerst auszuschalten und nach dem Lesen/Schreiben die Interrupts wieder einzuschalten.

# Interrupts ein-/aus-schalten

- Beispiel:

```
cli();  
++mySpecialValue;  
sei();
```

- In komplexeren Programmen kann man sich nicht immer sicher sein, dass die Interrupts tatsächlich gerade eingeschaltet sind, wenn man sie ausschalten will. Blindes Einschalten kann dann zu Problemen führen.

- Deshalb bietet sich die folgende Implementierung an:

```
uint8_t sreg = SREG; // sichere SREG  
cli(); // disable Interrupts  
++mySpecialValue;  
SREG=sreg; //stelle SREG wieder her
```

# Aufgabe

- Stoppe das Blinken der LEDs solange eine deiner Tasten gedrückt ist. (Tip: TIMSK Register)
- Lasse die LEDs nicht blinken sondern binär zählen von 0 bis 60.
- Programmiere eine Stoppuhr
  - 1. nur mit start/stop
  - 2. mit reset funktion