

Microcontroller Teil 3 Bitmanipulation

DI (FH) Andreas Pötscher

HTL Litec

Werden SFRs Werte direkt zugewiesen, werden dabei immer alle 8 Bit neu überschrieben.

```
DDRA = 0x0f;
```

überschreibt alle 8 Bit auf 0000 1111.

Sollen aber nur gezielt ein oder mehrere Bits überschrieben werden, gibt es die Möglichkeit dies mit Bitweisen Operatoren zu realisieren.

- ▶ Die Operatoren `&`, `|`, `^` und `~` sind bitweise Operatoren. (Im Gegensatz zu den logischen Operatoren `&&`, `||` und `!`, die bei Bedingungen in Schleifen oder if-Verzweigungen zur Anwendung kommen). Jedes Bit des einen Operanden wird mit dem entsprechenden Bit des anderen Operanden verknüpft.

- ▶ Die Operatoren `&`, `|`, `^` und `~` sind bitweise Operatoren. (Im Gegensatz zu den logischen Operatoren `&&`, `||` und `!`, die bei Bedingungen in Schleifen oder `if`-Verzweigungen zur Anwendung kommen). Jedes Bit des einen Operanden wird mit dem entsprechenden Bit des anderen Operanden verknüpft.
- ▶ Z.B. `a & m`

- ▶ Die Operatoren `&`, `|`, `^` und `~` sind bitweise Operatoren. (Im Gegensatz zu den logischen Operatoren `&&`, `||` und `!`, die bei Bedingungen in Schleifen oder `if`-Verzweigungen zur Anwendung kommen). Jedes Bit des einen Operanden wird mit dem entsprechenden Bit des anderen Operanden verknüpft.
- ▶ Z.B. `a & m`
- ▶ Beim Verändern von einzelnen Bits einer Variable wird diese meist mit einer zweiten Variable mit einer bitweisen Verknüpfung verarbeitet. Die zweite Variable die zur Veränderung dient heißt Bitmaske oder kurz Maske.

Beim bitweisen not ~ werden alle Bits einzeln invertiert. Aus 0 wird 1 und aus 1 wird 0.

► `uint8_t a = 0xA5; //1010 0101`

Beim bitweisen not ~ werden alle Bits einzeln invertiert. Aus 0 wird 1 und aus 1 wird 0.

```
▶ uint8_t a = 0xA5;           //1010 0101  
▶ uint8_t notA = ~a;          //0101 1010
```

Beim bitweisen und & werden alle Bits einzeln und verknüpft. Es müssen beide Bits 1 sein damit das Ergebnis 1 ist.

► `uint8_t a = 0xA5; //1010 0101`

Beim bitweisen und & werden alle Bits einzeln und verknüpft. Es müssen beide Bits 1 sein damit das Ergebnis 1 ist.

- ▶ `uint8_t a = 0xA5;` `//1010 0101`
- ▶ `uint8_t m = 0x0F;` `//0000 1111`

Beim bitweisen und & werden alle Bits einzeln und verknüpft. Es müssen beide Bits 1 sein damit das Ergebnis 1 ist.

- ▶ `uint8_t a = 0xA5; //1010 0101`
- ▶ `uint8_t m = 0x0F; //0000 1111`
- ▶ `uint8_t aAndM = a&m; //0000 0101`

Mit dem bitweisen und können gezielt einzelne Bits gelöscht (auf 0 gesetzt) werden.
Wenn Z.B. Das Bit 2 auf 0 gesetzt werden soll kann das so aussehen.

► `uint8_t a = 0xA5; //1010 0101`

Mit dem bitweisen und können gezielt einzelne Bits gelöscht (auf 0 gesetzt) werden.
Wenn Z.B. Das Bit 2 auf 0 gesetzt werden soll kann das so aussehen.

- ▶ `uint8_t a = 0xA5; //1010 0101`
- ▶ `uint8_t m = 0xFB; //1111 1011`

Mit dem bitweisen und können gezielt einzelne Bits gelöscht (auf 0 gesetzt) werden.
Wenn Z.B. Das Bit 2 auf 0 gesetzt werden soll kann das so aussehen.

```
▶ uint8_t a = 0xA5;           //1010 0101
▶ uint8_t m = 0xFB;           //1111 1011
▶ uint8_t aAndM = a&m;         //1010 0001
```

Beim bitweisen oder | werden alle Bits einzeln oder verknüpft. Es muss eines der beiden Bits 1 sein damit das Ergebnis 1 ist.

► `uint8_t a = 0xA5; //1010 0101`

Beim bitweisen oder `|` werden alle Bits einzeln oder verknüpft. Es muss eines der beiden Bits 1 sein damit das Ergebnis 1 ist.

- ▶ `uint8_t a = 0xA5;` `//1010 0101`
- ▶ `uint8_t m = 0x0F;` `//0000 1111`

Beim bitweisen oder `|` werden alle Bits einzeln oder verknüpft. Es muss eines der beiden Bits 1 sein damit das Ergebnis 1 ist.

- ▶ `uint8_t a = 0xA5;` `//1010 0101`
- ▶ `uint8_t m = 0x0F;` `//0000 1111`
- ▶ `uint8_t aOrM = a|m;` `//1010 1111`

Mit dem bitweisen oder können gezielt einzelne Bits gesetzt (auf 1 gesetzt) werden.
Wenn Z.B. Das Bit 3 auf 1 gesetzt werden soll kann das so aussehen.

► `uint8_t a = 0xA5; //1010 0101`

Mit dem bitweisen oder können gezielt einzelne Bits gesetzt (auf 1 gesetzt) werden.
Wenn Z.B. Das Bit 3 auf 1 gesetzt werden soll kann das so aussehen.

```
▶ uint8_t a = 0xA5;           //1010 0101
▶ uint8_t m = 0x08;           //0000 1000
```

Mit dem bitweisen oder können gezielt einzelne Bits gesetzt (auf 1 gesetzt) werden.
Wenn Z.B. Das Bit 3 auf 1 gesetzt werden soll kann das so aussehen.

```
▶ uint8_t a = 0xA5;           //1010 0101
▶ uint8_t m = 0x08;           //0000 1000
▶ uint8_t aAndM = a|m;        //1010 1101
```

Beim bitweisen exklusiv-oder \wedge werden alle Bits einzeln exklusiv-oder verknüpft. Es müssen beide Bits ungleich sein damit das Ergebnis 1 ist.

► `uint8_t a = 0xA5; //1010 0101`

Beim bitweisen exklusiv-oder \wedge werden alle Bits einzeln exklusiv-oder verknüpft. Es müssen beide Bits ungleich sein damit das Ergebnis 1 ist.

- ▶ `uint8_t a = 0xA5;` `//1010 0101`
- ▶ `uint8_t m = 0x0F;` `//0000 1111`

Beim bitweisen exklusiv-oder \wedge werden alle Bits einzeln exklusiv-oder verknüpft. Es müssen beide Bits ungleich sein damit das Ergebnis 1 ist.

```
▶ uint8_t a = 0xA5;           //1010 0101
▶ uint8_t m = 0x0F;           //0000 1111
▶ uint8_t aXorM = a^m;        //1010 1010
```

Mit dem bitweisen exklusiv-oder können gezielt einzelne Bits getoggelt (invertiert) werden. Wenn Z.B. Die Bits 2 und 3 getoggelt werden sollen kann das so aussehen.

► `uint8_t a = 0xA5; //1010 0101`

Mit dem bitweisen exklusiv-oder können gezielt einzelne Bits getoggelt (invertiert) werden. Wenn Z.B. Die Bits 2 und 3 getoggelt werden sollen kann das so aussehen.

```
▶ uint8_t a = 0xA5;           //1010 0101  
▶ uint8_t m = 0x0C;           //0000 1100
```


Mit dem bitweisen exklusiv-oder können gezielt einzelne Bits getoggelt (invertiert) werden. Wenn Z.B. Die Bits 2 und 3 getoggelt werden sollen kann das so aussehen.

```
▶ uint8_t a = 0xA5;           //1010 0101
▶ uint8_t m = 0x0C;           //0000 1100
▶ uint8_t aXorM = a^m;        //1010 1001
```

- ▶ **a & m** Beim bitweisen **und** werden alle Bits die in der Maske 0 sind im Ergebnis 0. Alle anderen Bits bleiben unverändert.

- ▶ $a \ \& \ m$ Beim bitweisen **und** werden alle Bits die in der Maske 0 sind im Ergebnis 0. Alle anderen Bits bleiben unverändert.
- ▶ $a \ | \ m$ Beim bitweisen **oder** werden alle Bits die in der Maske 1 sind im Ergebnis 1. Alle anderen Bits bleiben unverändert.

- ▶ $a \& m$ Beim bitweisen **und** werden alle Bits die in der Maske 0 sind im Ergebnis 0. Alle anderen Bits bleiben unverändert.
- ▶ $a | m$ Beim bitweisen **oder** werden alle Bits die in der Maske 1 sind im Ergebnis 1. Alle anderen Bits bleiben unverändert.
- ▶ $a \wedge m$ Beim bitweisen **xor** werden alle Bits die in der Maske 1 sind im Ergebnis invertiert. Alle anderen Bits bleiben unverändert.

Um die für die Bitmanipulation benötigte Maske in möglichst gut lesbarer Form erzeugen zu können, sind Schiebebefehle nötig. Diese können wieder anhand von Beispielen am besten erklärt werden.

Beim Linksschieben oder Shift-Left << werden alle Bits um eine bestimmte Anzahl von Stellen nach links geschoben. Auf der rechten Seite wird mit 0 aufgefüllt.

- ▶ `uint8_t a = 0xC7; //1100 0111`
- ▶ `uint8_t result = a << 1; //1000 1110`

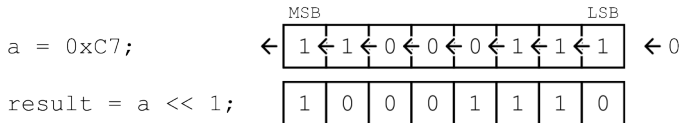


Figure 1: Shift Left

Beim Rechtsschieben oder Shift-Right >> werden alle Bits um eine bestimmte Anzahl von Stellen nach rechts geschoben. Auf der linken Seite wird mit 0 aufgefüllt.

- ▶ `uint8_t a = 0xC7; //1100 0111`
- ▶ `uint8_t n = 2;`
- ▶ `uint8_t result = a >> n; //0011 0001`

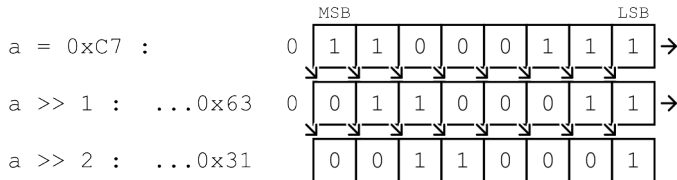


Figure 2: Shift Right

Das Setzen (auf logisch 1 setzen) von Bits erfolgt mittels bitweiser Oder-Verknüpfung mit einer Maske. In der Maske sind die zu setzenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gesetzt werden. Alle anderen bleiben unverändert.

► `DDRB |= 0x46;`

Das Setzen (auf logisch 1 setzen) von Bits erfolgt mittels bitweiser Oder-Verknüpfung mit einer Maske. In der Maske sind die zu setzenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gesetzt werden. Alle anderen bleiben unverändert.

▶ `DDRB |= 0x46;`

▶ `//DDRB` `xxxx xxxx`

Beispiel. Es sollen die Bits 6, 2 und 1 gesetzt werden. Alle anderen bleiben unverändert.

```
▶ DDRB |= 0x46;
▶ //DDRB          xxxx  xxxx
▶ //mask 0x46     0100  0110
```

Das Setzen (auf logisch 1 setzen) von Bits erfolgt mittels bitweiser Oder-Verknüpfung mit einer Maske. In der Maske sind die zu setzenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gesetzt werden. Alle anderen bleiben unverändert.

```

▶ DDRB |= 0x46;
▶ //DDRB          xxxx xxxx
▶ //mask 0x46      0100 0110
▶ //DDRB (nach DDRB |= mask) x1xx x1xx

```

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);  
DDRB |= mask;
```

► //0x01

0000 0001

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100
▶ //0x01 << 1	0000 0010

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100
▶ //0x01 << 1	0000 0010
▶ //(0x01<<6) (0x01<<2) (0x01<<1)	0100 0110


```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01		0	0	0	0	0	0	0	1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0
DDRB = 0x01<<3									1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0
DDRB = 0x01<<3								0	1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0
DDRB = 0x01<<3							1	0	1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0
DDRB = 0x01<<3						1	1	0	1

```
DDRB |= 0x01 << 3;
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<3		0	0	0	0	1	0	0	0
DDRB = 0x01<<3		1	1	0	0	1	1	0	1

Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einser-Komplement (Negation, Not-Verknüpfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gelöscht werden. Alle anderen bleiben unverändert.

► `DDRB &= ~(0x46);`

Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einser-Komplement (Negation, Not-Verknüpfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gelöscht werden. Alle anderen bleiben unverändert.

- ▶ `DDRB &= ~(0x46);`
- ▶ `//mask 0x46` 0100 0110

Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einser-Komplement (Negation, Not-Verknüpfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gelöscht werden. Alle anderen bleiben unverändert.

- ▶ `DDRB &= ~(0x46);`
- ▶ `//mask 0x46` 0100 0110
- ▶ `//~mask 0x46` 1011 1001

Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einser-Komplement (Negation, Not-Verknüpfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gelöscht werden. Alle anderen bleiben unverändert.

▶ <code>DDRB &= ~(0x46);</code>	
▶ <code>//mask 0x46</code>	<code>0100 0110</code>
▶ <code>//~mask 0x46</code>	<code>1011 1001</code>
▶ <code>//DDRB</code>	<code>xxxx xxxx</code>

Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einer-Komplement (Negation, Not-Verknüpfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.

Beispiel. Es sollen die Bits 6, 2 und 1 gelöscht werden. Alle anderen bleiben unverändert.

- ▶ `DDRB &= ~(0x46);`
- ▶ `//mask 0x46` 0100 0110
- ▶ `//~mask 0x46` 1011 1001
- ▶ `//DDRB` xxxx xxxx
- ▶ `//DDRB(nach DDRB &= ~mask)` x0xx x00x

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

► //0x01

0000 0001

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

- | | |
|---------------|-----------|
| ▶ //0x01 | 0000 0001 |
| ▶ //0x01 << 6 | 0100 0000 |
| ▶ //0x01 << 2 | 0000 0100 |

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100
▶ //0x01 << 1	0000 0010

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100
▶ //0x01 << 1	0000 0010
▶ //(0x01<<6) (0x01<<2) (0x01<<1)	0100 0110

Wie beim setzen kann auch hier die Bitmaske mit Bitshifting erzeugt werden. Wichtig ist dabei die Negation und die Klammern. Damit auch die Ausführungsreihenfolge stimmt.

```
uint8_t mask = ~((0x01<<6) | (0x01<<2) | (0x01<<1));  
DDRB |= mask;
```

▶ //0x01	0000 0001
▶ //0x01 << 6	0100 0000
▶ //0x01 << 2	0000 0100
▶ //0x01 << 1	0000 0010
▶ //(0x01<<6) (0x01<<2) (0x01<<1)	0100 0110
▶ //~((0x01<<6) (0x01<<2) (0x01<<1))	1011 1001

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01		0	0	0	0	0	0	0	1

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
0x01<<2		0	0	0	0	0	1	0	0

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
~(0x01<<2)		1	1	1	1	1	0	1	1

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
~(0x01<<2)		1	1	1	1	1	0	1	1
DDRB &= ~(0x01<<2)									1


```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
~(0x01<<2)		1	1	1	1	1	0	1	1
DDRB &= ~(0x01<<2)								0	1

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
~(0x01<<2)		1	1	1	1	1	0	1	1
DDRB &= ~(0x01<<2)							0	0	1

```
DDRB &= ~(0x01 << 2);
```

Ausgeführte Befehle	hex	7	6	5	4	3	2	1	0
DDRB	0xC5	1	1	0	0	0	1	0	1
~(0x01<<2)		1	1	1	1	1	0	1	1
DDRB &= ~(0x01<<2)		1	1	0	0	0	0	0	1

Unter Toggeln eines Bits versteht man, das Bit zu ändern. Ist das Bit vor dem Toggeln „0“ so ist es nach dem Toggeln „1“. Ist das Bit vor dem Toggeln hingegen „1“, so ist es nachher „0“.

Beispiel. Es sollen die Bits 6, 2 und 1 getoggelt werden. Alle anderen bleiben unverändert.

► `PORTB ^= 0x46;`

Unter Toggeln eines Bits versteht man, das Bit zu ändern. Ist das Bit vor dem Toggeln „0“ so ist es nach dem Toggeln „1“. Ist das Bit vor dem Toggeln hingegen „1“, so ist es nachher „0“.

Beispiel. Es sollen die Bits 6, 2 und 1 getoggelt werden. Alle anderen bleiben unverändert.

- ▶ `PORTB ^= 0x46;`
- ▶ `//PORTB` XXXX XXXX

Unter Toggeln eines Bits versteht man, das Bit zu ändern. Ist das Bit vor dem Toggeln „0“ so ist es nach dem Toggeln „1“. Ist das Bit vor dem Toggeln hingegen „1“, so ist es nachher „0“.

Beispiel. Es sollen die Bits 6, 2 und 1 getoggelt werden. Alle anderen bleiben unverändert.

- ```
▶ PORTB ^= 0x46;
▶ //PORTB xxxx xxxx
▶ //mask 0x46 0100 0110
```

Unter Toggeln eines Bits versteht man, das Bit zu ändern. Ist das Bit vor dem Toggeln „0“ so ist es nach dem Toggeln „1“. Ist das Bit vor dem Toggeln hingegen „1“, so ist es nachher „0“.

Beispiel. Es sollen die Bits 6, 2 und 1 getoggelt werden. Alle anderen bleiben unverändert.

- ▶ PORTB ^= 0x46;
- ▶ //PORTB                      xxxx  xxxx
- ▶ //mask 0x46                 0100  0110
- ▶ //PORTB (nach PORTB ^= mask) x!xx x!!x (*! bedeutet, dass das Bit getoggelt wurde*)

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
PORTB ^= mask;
```

► //0x01

0000 0001



Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
PORTB ^= mask;
```

|               |           |
|---------------|-----------|
| ▶ //0x01      | 0000 0001 |
| ▶ //0x01 << 6 | 0100 0000 |

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
PORTB ^= mask;
```

|               |           |
|---------------|-----------|
| ▶ //0x01      | 0000 0001 |
| ▶ //0x01 << 6 | 0100 0000 |
| ▶ //0x01 << 2 | 0000 0100 |

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
PORTB ^= mask;
```

|               |           |
|---------------|-----------|
| ▶ //0x01      | 0000 0001 |
| ▶ //0x01 << 6 | 0100 0000 |
| ▶ //0x01 << 2 | 0000 0100 |
| ▶ //0x01 << 1 | 0000 0010 |

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
PORTB ^= mask;
```

|                                       |           |
|---------------------------------------|-----------|
| ▶ //0x01                              | 0000 0001 |
| ▶ //0x01 << 6                         | 0100 0000 |
| ▶ //0x01 << 2                         | 0000 0100 |
| ▶ //0x01 << 1                         | 0000 0010 |
| ▶ //(0x01<<6)   (0x01<<2)   (0x01<<1) | 0100 0110 |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| DDRB                | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------------|------|---|---|---|---|---|---|---|---|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------------|------|---|---|---|---|---|---|---|---|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| DDRB ^= mask          |      |   |   |   |   |   |   |   | 1 |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1        | 0        |
|-----------------------|------|---|---|---|---|---|---|----------|----------|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1 | 0        | 1        |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1 | 1 | 0        | 0        |
| DDRB ^= mask          |      |   |   |   |   |   |   | <b>0</b> | <b>1</b> |



```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3 | 2        | 1        | 0        |
|-----------------------|------|---|---|---|---|---|----------|----------|----------|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1        | 0        | 1        |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1 | 1        | 0        | 0        |
| DDRB ^= mask          |      |   |   |   |   |   | <b>0</b> | <b>0</b> | <b>1</b> |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------------|------|---|---|---|---|---|---|---|---|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| DDRB ^= mask          |      |   |   |   |   | 1 | 0 | 0 | 1 |

```
uint8_t mask = (0x01 << 2) | (0x01 << 3);
DDRB ^= mask;
```

| Ausgeführte Befehle   | hex  | 7 | 6 | 5 | 4 | 3        | 2        | 1 | 0 |
|-----------------------|------|---|---|---|---|----------|----------|---|---|
| DDRB                  | 0xC5 | 1 | 1 | 0 | 0 | 0        | 1        | 0 | 1 |
| (0x01<<2)   (0x01<<3) |      | 0 | 0 | 0 | 0 | 1        | 1        | 0 | 0 |
| DDRB ^= mask          |      | 1 | 1 | 0 | 0 | <b>1</b> | <b>0</b> | 0 | 1 |

*Anstelle von „Testen“ spricht man auch vom „Einlesen“ von Bits. Meistens möchte man in Abhängigkeit davon, ob ein bestimmtes Bit in einem Special-Function-Register oder in einer Variable logisch 1 bzw. logisch 0 ist, einen Anweisungsblock ausführen, oder eben nicht ausführen. Es handelt sich also um eine if-Anweisung. Das Testen erfolgt so, dass die nicht zu testenden Bits ausmaskiert (auf 0 gesetzt) werden.*

Am Pin PB4 sind ein Taster und ein externer pull-down-Widerstand angeschlossen. Wenn der Taster gedrückt ist, liegt am Pin PB4 ein High-Spannungspegel an, und Bit 4 des Registers PINB ist somit „1“. Bei offenem Taster ist der Spannungspegel an PB4 Low und Bit 4 von PINB ist „0“.

```
if(PINB & (0x01 << 4))
{
 //Code Abschnitt der ausgeführt wird, wenn an PIN PB4 ein High Pegel
 //anliegt und somit Bit 4 vom Register PINB gleich 1 ist.
}
else
{
 //Code Abschnitt der ausgeführt wird, wenn an PIN PB4 ein Low Pegel
 //anliegt und somit Bit 4 vom Register PINB gleich 0 ist.
}
```

Taster (PB4) betätigt

```
PINB & (0x01 << 4);
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

Taster (PB4) betätigt

```
PINB & (0x01 << 4);
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| (0x01<<4)           |      | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Taster (PB4) betätigt

```
PINB & (0x01 << 4);
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| (0x01<<4)           |      | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| PINB &= mask        |      | 0 | 0 | 0 |   | 0 | 0 | 0 | 0 |



Taster (PB4) betätigt

```
PINB & (0x01 << 4);
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4        | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|----------|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1        | 0 | 1 | 0 | 1 |
| (0x01<<4)           |      | 0 | 0 | 0 | 1        | 0 | 0 | 0 | 0 |
| PINB &= mask        | 0x10 | 0 | 0 | 0 | <b>1</b> | 0 | 0 | 0 | 0 |

Taster (PB4) nicht betätigt

```
PINB & (0x01 << 4);
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4        | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|----------|---|---|---|---|
| PINB                | 0x85 | 1 | 0 | 0 | 0        | 0 | 1 | 0 | 1 |
| (0x01<<4)           |      | 0 | 0 | 0 | 1        | 0 | 0 | 0 | 0 |
| PINB &= mask        | 0x00 | 0 | 0 | 0 | <b>0</b> | 0 | 0 | 0 | 0 |

Um das Programm übersichtlich und wartbar zu gestalten empfiehlt sich die Zustände der GPIO Pins nach dem Einlesen direkt in eine Variable zu speichern. Nach dem vorher beschriebenen Muster sieht das so aus.

```
uint8_t bit = PINB & (0x01 << 4);
```

Die Variable bit ist je nach dem Zustand von Bit 4 von PINB nach diesen Zeilen entweder 00000000=0, also logisch falsch, oder 00010000=16, also ungleich null und somit logisch wahr.

Eine zweite sehr kurze und elegante Variante ist das Verschieben des Registers, dass das gesuchte Bit an der Stelle 0 steht und dann mit 0x01 ausmaskiert wird. Damit ist bei gesetztem Bit das Ergebnis 1 und bei nicht gesetztem Bit 0.

```
uint8_t bit = PINB >> 4 & 0x01;
```

```
▶ //PINB xxxb xxxx
```

Eine zweite sehr kurze und elegante Variante ist das Verschieben des Registers, dass das gesuchte Bit an der Stelle 0 steht und dann mit 0x01 ausmaskiert wird. Damit ist bei gesetztem Bit das Ergebnis 1 und bei nicht gesetztem Bit 0.

```
uint8_t bit = PINB >> 4 & 0x01;
```

```

▶ //PINB xxxb xxxx
▶ //Nach PIN B >> 4 0000 xxxb

```

Eine zweite sehr kurze und elegante Variante ist das verschieben des Registers, dass das gesuchte Bit an der Stelle 0 steht und dann mit 0x01 ausmaskiert wird. Damit ist bei gesetzten Bit das Ergebnis 1 und bei nicht gesetzten Bit 0.

```
uint8_t bit = PINB >> 4 & 0x01;
```

- |                     |                   |
|---------------------|-------------------|
| ▶ //PINB            | xxx <b>b</b> xxxx |
| ▶ //Nach PIN B >> 4 | 0000 xxx <b>b</b> |
| ▶ //Nach & 0x01     | 0000 000 <b>b</b> |

Taster (PB4) betätigt

```
uint8_t bit = PINB >> 4 & 0x01;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

Taster (PB4) betätigt

```
uint8_t bit = PINB >> 4 & 0x01;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| PINB >> 4           | 0x09 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |



Taster (PB4) betätigt

```
uint8_t bit = PINB >> 4 & 0x01;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| PINB >> 4           | 0x09 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| PINB >> 4 & 0x01    |      | 0 | 0 | 0 | 0 | 0 | 0 | 0 |   |

Taster (PB4) betätigt

```
uint8_t bit = PINB >> 4 & 0x01;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0        |
|---------------------|------|---|---|---|---|---|---|---|----------|
| PINB                | 0x95 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1        |
| PINB >> 4           | 0x09 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1        |
| PINB >> 4 & 0x01    | 0x01 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | <b>1</b> |

Taster (PB4) nicht betätigt

```
uint8_t bit = PINB >> 4 & 0x01;
```

| Ausgeführte Befehle | hex  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------|------|---|---|---|---|---|---|---|---|
| PINB                | 0x85 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| PINB >> 4           | 0x08 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| PINB >> 4 & 0x01    | 0x00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Setzen von Bits

```
uint8_t bitNo;
PORTA |= 0x01 << bitNo;
```

## Löschen von Bits

```
uint8_t bitNo;
PORTA &= ~(0x01 << bitNo);
```

## Toggeln von Bits

```
uint8_t bitNo;
PORTA ^= 0x01 << bitNo;
```

## Auslesen von Bits

```
uint8_t bitNo;
uint8_t bit = PINA >> bitNo & 0x01;

if(bit)
{
 //...
}
```