Digitalelektronik 2

Bitmuster darstellen

Stefan Rothe

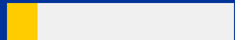
2015–03–01

Rechtliche Hinweise

Dieses Werk von Thomas Jampen und Stefan Rothe steht unter einer *Creative Commons Attribution-Non-Commercial-ShareAlike*-Lizenz.

Zudem verzichten die Autoren auf sämtliche Urheberrechtsansprüche für die in diesem Werk enthaltenen Quelltexte.

Rechtliche Hinweise



Stellenwertsysteme

In einem **Stellenwertsystem** bestimmt die Position (Stelle) eines Symbols dessen Wert:

Beispiel: 10:53:28 Uhr

Bedeutung: Stunden, Minuten, Sekunden

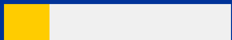
Umrechnung: $10 \cdot 60^2 + 53 \cdot 60 + 28$ Sekunden nach Mitternacht

Vorgehen:

- Jede Zahl wird mit einer Potenz der Basis multipliziert.
- Die Potenz ergibt sich aus der Position der Zahl.

Genau so verhält es sich mit unserem Zahlensystem, dem Dezimalsystem:

$$927 = 9 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$$



Das Binärsystem

Da ein Bit nur 2 Werte annehmen kann, wird in der Informatik oft das **Binärsystem** (auch Dualsystem genannt) eingesetzt.

Beispiel: 1101_b

Basis: 2

Umrechnung ins Dezimalsystem: $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$

Damit binäre Zahlen nicht mit Dezimalzahlen verwechselt werden, wird den Binärzahlen ein kleines, tiefstehendes **b** angehängt.

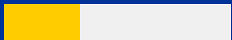
Aufgabe 4: Binär $\leftrightarrow$ Dezimal

Schreiben Sie die Binärdarstellung der folgenden Zahlen auf:

- $1 = 1_b$ $3 = 11_b$ $7 = 111_b$
- $14 = 1110_b$ $15 = 1111_b$ $16 = 10000_b$
- $32 = 100000_b$ $64 = 1000000_b$ $128 = 10000000_b$

Schreiben Sie die Dezimaldarstellung der folgenden Zahlen in Binärdarstellung auf:

- $1_b = 1$ $10_b = 2$ $100_b = 4$
- $1001_b = 9$ $10001_b = 17$ $100001_b = 33$
- $101010_b = 42$ $101011_b = 43$ $101100_b = 44$



Arduino: Typen für ganze Zahlen

Typ	Grösse	Minimum	Maximum
`byte`	8 Bit	0	255
`int`	16 Bit	–32’768	32’767
`unsigned int`	16 Bit	0	65’535
`long`	32 Bit	-2’147’483’648	2’147’483’647
`unsigned long`	32 Bit	0	4’294’967’295

Achtung: Diese Typen und Grössenangaben sind nur für die Arduino-Umgebung gültig. Die vorhandenen Typen und deren Grössen hängen in C/C++ vom Compiler und dem Prozessor ab.

Bitweises UND

```
x & y
```

- Führt für jedes Bit von **`x`** und **`y`** eine logische UND-Operation durch
- Ein Bit im Resultat wird gesetzt, wenn es sowohl bei **`x`** als auch bei **`y`** gesetzt ist
- englisch: *bitwise AND*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
y	0	1	1	0	0	1	0	1
x & y	0	1	0	0	0	1	0	0

```
byte x =  92;
byte y = 101;
byte z = x & y; // z == 68
```

Bitweises ODER

```
x | y
```

- Führt für jedes Bit von **x** und **y** eine logische ODER-Operation durch
- Ein Bit im Resultat wird gesetzt, wenn es bei **x** oder bei **y** gesetzt ist
- englisch: *bitwise OR*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
y	0	1	1	0	0	1	0	1
x | y	0	1	1	1	1	1	0	1

```
byte x =  92;
byte y = 101;
byte z = x | y; // z == 125
```

Bitweises NICHT

```
~x
```

- Führt für jedes Bit von **x** eine logische NICHT-Operation durch
- Ein Bit im Resultat wird gesetzt, wenn es bei **x** **nicht** gesetzt ist
- englisch: *bitwise NOT*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
~x	1	0	1	0	0	0	1	1

```
byte x =  92;
byte y = ~x; // y == 163
```

Bitweises exklusives ODER

```
x ^ y
```

- Führt für jedes Bit von **x** und **y** eine logische exklusiv-ODER-Operation durch
- Ein Bit im Resultat wird gesetzt, wenn es bei **x** oder bei **y**, jedoch **nicht bei beiden** gesetzt ist.
- englisch: *bitwise XOR*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
y	0	1	1	0	0	1	0	1
x ^ y	0	0	1	1	1	0	0	1

```
byte x =  92;
byte y = 101;
byte z = x ^ y; // z == 57
```

Bitweises Schieben nach links

```
x << a
```

- Schiebt jedes Bit von **x** um **a** Stellen nach **links**
- Rechts wird mit ungesetzten Bits (0) aufgefüllt
- Identisch zur Multiplikation mit 2^a, falls die obersten a Bits nicht gesetzt sind
- englisch: *bitwise shift left*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
x << 1	1	0	1	1	1	0	0	0
x << 2	0	1	1	1	0	0	0	0

```
byte x =  92;
byte y = x << 1; // y == 184
byte z = x << 2; // z == 112
```

Bitweises Schieben nach rechts

```
x >> a
```

- Schiebt jedes Bit von **x** um **a** Stellen nach **rechts**
- Links wird mit ungesetzten Bits (0) aufgefüllt
- Identisch zur Division durch 2^a
- englisch: *bitwise shift right*

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
x >> 1	0	0	1	0	1	1	1	0
x >> 2	0	0	0	1	0	1	1	1

```
byte x =  92;
byte y = x >> 1; // y == 46
byte z = x >> 2; // z == 23
```

Bitweise Operationen

Ausdruck	Bedeutung
`x & y`	bitweises UND
`x	y`	bitweises ODER
`x ^ y`	bitweises exklusives ODER
`~x`	bitweises NICHT
`x << a`	bitweises Schieben nach links
`x >> b`	bitweises Schieben nach rechts

Bit überprüfen

Durch eine Kombination von bitweisen Operationen kann überprüft werden, ob bei einer ganzzahligen Variable ein bestimmtes Bit gesetzt ist:

```
if ((x & (1 << i)) != 0) {
    // i. Bit von x gesetzt
}
else {
    // i. Bit von x nicht gesetzt
}
```

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
1 << 0	0	0	0	0	0	0	0	1
x & (1 << 0)	0	0	0	0	0	0	0	0

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	1	1	0	0
1 << 3	0	0	0	0	1	0	0	0
x & (1 << 3)	0	0	0	0	1	0	0	0

Aufgabe 5: Bitmuster darstellen

a. Erstellen Sie basierend auf dem Lauflicht eine Schaltung und ein Programm, welches das Bitmuster einer `byte`-Variable mit acht Leuchtdioden darstellt.

b. Lassen Sie die Variable von 0 bis 255 zählen.

Beispielsweise sollte der Wert 210 folgendes LED-Muster ergeben:

1 1 0 1 0 0 1 0