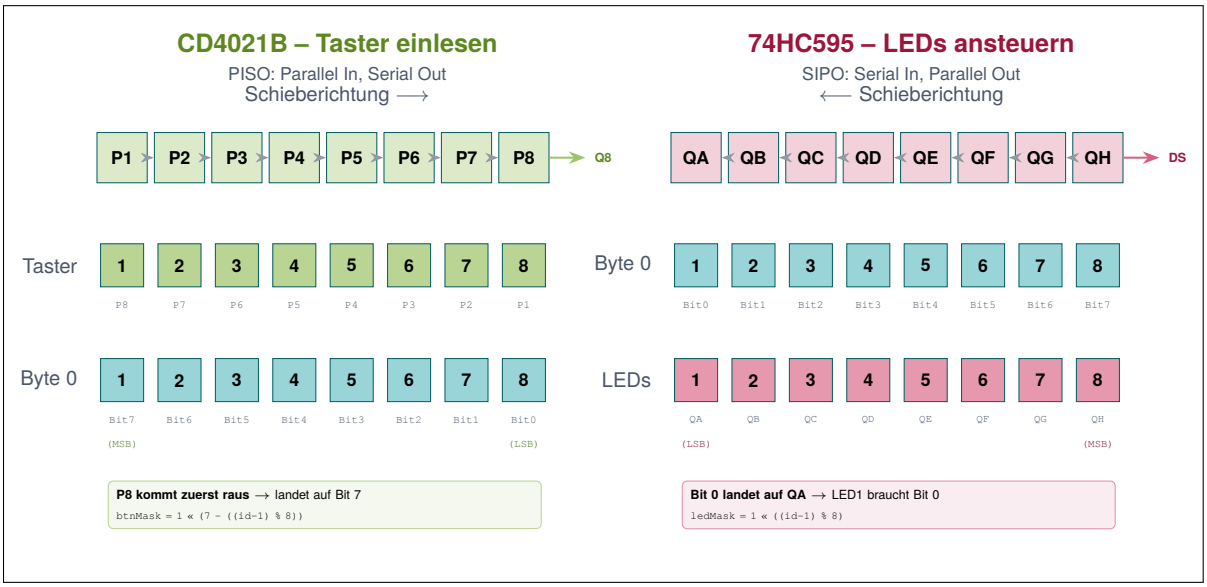
Bit-Mapping bei Schieberegistern

CD4021B (Input) & 74HC595 (Output)

MSB vs. LSB – Warum die Asymmetrie?

ESP32-S3 • Selection Panel • SPI-Kommunikation



Keywords

CD4021B • 74HC595 • PISO/SIPO • MSB/LSB • Bit-Masken

Stand: 11.01.2026
Version 1.0.0

Jan Unger
Tools: Claude 4.5, ChatGPT 5.2

Inhaltsverzeichnis

1 Grundlagen: Zahlensysteme und Bit-Operationen

Bevor wir uns mit Schieberegistern beschäftigen, müssen wir verstehen, wie Computer mit Zahlen arbeiten.

1.1 Zahlensysteme: Dezimal, Binär, Hexadezimal

Computer arbeiten intern mit Bits – Einsen und Nullen. Für uns Menschen ist das unübersichtlich, deshalb nutzen wir verschiedene Schreibweisen für dieselbe Zahl.

1.1.1 Die drei Darstellungen

Tabelle 1: Zahlensysteme im Vergleich

System	Basis	Ziffern	Präfix in C/C++
Dezimal	10	0–9	(keiner)
Binär	2	0, 1	`0b`
Hexadezimal	16	0–9, A–F	`0x`

Nibble-Schreibweise

Ein **Nibble** ist eine Gruppe von 4 Bits. Wir schreiben Binärzahlen mit einem Punkt als Trenner, um die Lesbarkeit zu erhöhen:

`0b0001.0000` statt `0b00010000`

Ein Nibble entspricht genau einer Hex-Ziffer – das macht die Umrechnung einfach.

1.1.2 Umrechnungstabelle: Dezimal – Binär – Hexadezimal

Tabelle 2: Umrechnungstabelle für ein Nibble (4 Bit)

Dezimal	Binär	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

1.1.3 Beispiel: Die Zahl 16

```
Dezimal:  16
Binaer:   0b0001.0000  (1 Nibble = 0001, 2. Nibble = 0000)
Hex:      0x10         (1. Hex-Ziffer = 1, 2. Hex-Ziffer = 0)
```

Die Umrechnung Binär → Hex funktioniert nibbleweise:

1.2 Datentypen und Wertebereiche (C++ / ESP32)

In der Embedded-Programmierung ist es wichtig, den **exakten Speicherbedarf** eines Datentyps zu kennen. C++ bietet dafür Typen mit garantierter Bitbreite.

1.2.1 Ganzzahl-Typen mit fester Breite

Tabelle 3: Integer-Datentypen mit fester Bitbreite (stdint.h)

Typ	Bits	Wertebereich	Hex-Bereich
`uint8_t`	8	0 bis 255	`0x00` – `0xFF`
`int8_t`	8	−128 bis +127	`0x80` – `0x7F`
`uint16_t`	16	0 bis 65.535	`0x0000` – `0xFFFF`
`int16_t`	16	−32.768 bis +32.767	`0x8000` – `0x7FFF`
`uint32_t`	32	0 bis 4.294.967.295	`0x00000000` – `0xFFFFFFFF`
`int32_t`	32	±2.147.483.647	`0x80000000` – `0x7FFFFFFF`

Namenskonvention

- `u` = **unsigned** (vorzeichenlos, nur positive Werte)
- Zahl = **Bitbreite** (8, 16, 32, 64)
- `_t` = **type** (Kennzeichnung als Typedef)

Beispiel: `uint8_t` = unsigned integer, 8 Bit, type

1.2.2 Warum `uint8_t` für Schieberegister?

Ein Schieberegister verarbeitet genau **8 Bits** – ein Byte. Der Typ `uint8_t` passt perfekt:

```
#include <stdint.h>  // oder <cstdint> in modernem C++

uint8_t buttonState = 0;      // 8 Taster = 8 Bits = 1 Byte
uint8_t ledState[2] = {0, 0}; // 10 LEDs = 10 Bits = 2 Bytes
```

Tabelle 4: Vergleich: `uint8_t` vs. alternative Typen

Typ	Bits	Garantiert?	Empfehlung
`uint8_t`	8	Ja, immer exakt 8 Bit	**Empfohlen**
`unsigned char`	8	Ja, mindestens 8 Bit	OK, aber weniger explizit
`byte`	8	Nur in Arduino-Framework	Nicht portabel
`int`	32	Plattformabhängig	Zu groß, verschwendet RAM

1.2.3 ESP32-S3: Speicher und Performance

Der ESP32-S3 ist ein 32-Bit-Mikrocontroller. Das bedeutet:

Tabelle 5: ESP32-S3: Datentypen und Speicher

Typ	Speicher	Anwendung
`uint8_t`	1 Byte	Schieberegister, GPIO-Zustände
`uint16_t`	2 Bytes	ADC-Werte (12 Bit), Timer
`uint32_t`	4 Bytes	Zeitstempel (`millis()`), Adressen
`bool`	1 Byte	Flags, Zustände (true/false)
`float`	4 Bytes	Sensorwerte mit Nachkommastellen

Überlauf beachten!

Wenn ein Wert den Wertebereich überschreitet, springt er auf den Anfang zurück:

```
uint8_t x = 255;
x = x + 1;        // x ist jetzt 0, nicht 256!

uint8_t y = 0;
y = y - 1;        // y ist jetzt 255, nicht -1!
```

Dies nennt man **Überlauf** (Overflow) bzw. **Unterlauf** (Underflow).

1.2.4 Literal-Suffixe für Klarheit

In C++ kann man Zahlen mit Suffixen versehen, um den Typ explizit anzugeben:

```
uint8_t mask = 1u << 4;       // 'u' = unsigned (wichtig bei Shifts!)
uint32_t bigNum = 1000000UL;  // 'UL' = unsigned long (32 Bit)
```

Tabelle 6: Literal-Suffixe in C++

Suffix	Typ	Beispiel
(keiner)	`int`	`42`
`u` / `U`	`unsigned int`	`42u`
`l` / `L`	`long`	`42L`
`ul` / `UL`	`unsigned long`	`42UL`
`f` / `F`	`float`	`3.14f`

1.2.5 Warum `1u` bei Bit-Operationen?

Das `u` hinter einer Zahl bedeutet **unsigned** (vorzeichenlos). Bei Bit-Operationen ist das entscheidend wichtig.

Tabelle 7: Vergleich: `1` vs. `1u` bei Shift-Operationen

Code	Typ von `1`	Bedeutung
`1 « 4`	`int` (signed)	Vorzeichen-behaftet, kann negativ werden
`1u « 4`	`unsigned int`	Nur positiv, sicheres Bit-Shifting

Das Problem mit signed Integers

Bei `signed` Typen ist das höchste Bit das **Vorzeichen-Bit**:

```
int8_t (signed, 8 Bit):
+---+---+---+---+---+---+---+---+
| V | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  V = Vorzeichen-Bit
+---+---+---+---+---+---+---+---+
  ^
  |
  0 = positiv (+0 bis +127)
  1 = negativ (-128 bis -1)

uint8_t (unsigned, 8 Bit):
+---+---+---+---+---+---+---+---+
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  Alle Bits fuer den Wert
+---+---+---+---+---+---+---+---+
  ^
  |
  Kein Vorzeichen, Wert 0 bis 255
```

Konkretes Beispiel: Was passiert bei `1 « 7`?

```
1 // OHNE 'u' - problematisch:
2 int result = 1 << 7;      // 1 << 7 = 128
3 int8_t x = result;        // 128 passt nicht in int8_t (-128 bis +127)!
4                           // Ergebnis: x = -128 (Ueberlauf ins Vorzeichen-Bit)
5
6 // MIT 'u' - sicher:
7 unsigned int result = 1u << 7;  // 1u << 7 = 128
8 uint8_t y = result;             // 128 passt in uint8_t (0 bis 255)
9                                 // Ergebnis: y = 128 ✓
```

Undefined Behavior vermeiden!

Ein Linksshift, der das Vorzeichen-Bit eines `signed` Typs verändert, ist in C++ **undefiniertes Verhalten**. Der Compiler kann alles machen – das Programm kann abstürzen, falsche Werte liefern, oder zufällig funktionieren.

Best Practice: Immer `1u` bei Bit-Operationen

Verwende **immer** `1u` statt `1` bei Shift-Operationen:

```
// Bit-Masken erzeugen:
uint8_t mask = 1u << 7;         // ✓ Sicher
uint8_t mask = 1 << 7;          // × Riskant

// Bits setzen:
ledState |= (1u << bitPos);     // ✓ Sicher
ledState |= (1 << bitPos);      // × Riskant

// Bits loeschen:
ledState &= ~(1u << bitPos);    // ✓ Sicher
ledState &= ~(1 << bitPos);     // × Riskant
```

Merkregel: Siehst du « oder », schreib `1u` statt `1`.

1.3 Skalierung: Von 8 auf 100+ Bits

Was passiert, wenn wir nicht nur 8 LEDs haben, sondern 100? Brauchen wir einen `uint100_t`? Die Antwort: Nein – wir verwenden **Arrays**.

1.3.1 Das Problem: Standard-Typen enden bei 64 Bit

Tabelle 8: Verfügbare Integer-Typen in C++

Typ	Bits	Status
`uint8_t`	8	✓ Standard
`uint16_t`	16	✓ Standard
`uint32_t`	32	✓ Standard
`uint64_t`	64	✓ Standard (Maximum!)

Der ESP32-S3 ist ein **32-Bit-Prozessor** – er kann maximal 32 Bits in einem Rechenschritt verarbeiten. Für größere Datenmengen müssen wir die Arbeit aufteilen.

1.3.2 Die Lösung: Arrays von Bytes

Statt eines riesigen Integer-Typs verwenden wir ein **Array aus `uint8_t`**:

```
// Berechnung: Wie viele Bytes brauchen wir?
// 100 Bits / 8 Bits pro Byte = 12.5 -> aufrunden = 13 Bytes

// 100 LEDs = 100 Bits = 13 Bytes (13 x 8 = 104 Bits, davon 100 genutzt)
uint8_t ledState[13] = {0};

```

```
// 100 Taster = 100 Bits = 13 Bytes
uint8_t buttonState[13] = {0};
```

Formel: Anzahl Bytes berechnen

$$\text{Bytes} = \left\lceil \frac{\text{Anzahl Bits}}{8} \right\rceil = \frac{\text{Bits} + 7}{8} \quad \text{(ganzzahlig)}$$

In C++:

```
#define NUM_LEDS 100
#define NUM_BYTES ((NUM_LEDS + 7) / 8)  // = 13
uint8_t ledState[NUM_BYTES] = {0};
```

1.3.3 Die Formeln skalieren automatisch

Die gleichen Formeln funktionieren für 10, 100 oder 1000 LEDs:

```
// LED einschalten (funktioniert fuer jede Anzahl!)
void setLed(uint8_t id) {
    uint8_t byteIndex = (id - 1) / 8;     // Welches Byte?
    uint8_t bitPosition = (id - 1) % 8;   // Welches Bit im Byte?
    ledState[byteIndex] |= (1u << bitPosition);
}

// LED ausschalten
void clearLed(uint8_t id) {
    uint8_t byteIndex = (id - 1) / 8;
    uint8_t bitPosition = (id - 1) % 8;
    ledState[byteIndex] &= ~(1u << bitPosition);
}

// LED-Status abfragen
bool getLed(uint8_t id) {
    uint8_t byteIndex = (id - 1) / 8;
    uint8_t bitPosition = (id - 1) % 8;
    return ledState[byteIndex] & (1u << bitPosition);
}
```

1.3.4 Beispiel: LED 87 einschalten

```
id = 87

Schritt 1: byteIndex = (87 - 1) / 8 = 86 / 8 = 10
           -> LED 87 liegt in Byte 10

Schritt 2: bitPosition = (87 - 1) % 8 = 86 % 8 = 6
           -> LED 87 liegt auf Bit 6 (innerhalb Byte 10)
```

```
Schritt 3: mask = 1u << 6 = 0b0100.0000 = 0x40

Schritt 4: ledState[10] |= 0x40
           -> Bit 6 in Byte 10 wird gesetzt
```

Tabelle 9: Beispiele: Byte-Index und Bit-Position für verschiedene LEDs

LED	(id-1)	Byte	Bit	Maske
1	0	0	0	0b0000.0001
8	7	0	7	0b1000.0000
9	8	1	0	0b0000.0001
16	15	1	7	0b1000.0000
50	49	6	1	0b0000.0010
87	86	10	6	0b0100.0000
100	99	12	3	0b0000.1000

1.3.5 Hardware: Kaskadierung der Schieberegister

Für 100 LEDs brauchen wir 13 Schieberegister (74HC595) in Reihe:

```
ESP32 ---> SR1 ---> SR2 ---> SR3 ---> ... ---> SR13
            |         |         |                  |
          LED       LED       LED                LED
          1-8       9-16     17-24              97-104
```

Die Daten werden als **zusammenhängendes Array** gesendet:

```
#define NUM_LEDS 100
#define NUM_BYTES ((NUM_LEDS + 7) / 8)  // 13

uint8_t ledState[NUM_BYTES] = {0};

void updateShiftRegisters() {
    digitalWrite(LATCH_PIN, LOW);
    // Bytes rueckwaerts senden (letztes Byte zuerst)
    for (int i = NUM_BYTES - 1; i >= 0; i--) {
        SPI.transfer(ledState[i]);
    }
    digitalWrite(LATCH_PIN, HIGH);
}
```

Merkregel: Skalierung

- **Bits → Bytes:** `(bits + 7) / 8`
- **ID → Byte-Index:** `(id - 1) / 8`
- **ID → Bit-Position:** `(id - 1) % 8`
- **Maske:** `1u << bitPosition`

Diese Formeln funktionieren für **jede** Anzahl von LEDs oder Tastern!

1.4 Bit-Positionen in einem Byte

Ein Byte besteht aus 8 Bits. Jedes Bit hat eine **Position** (0–7) und einen **Stellenwert** (Potenz von 2).

Tabelle 10: Bit-Positionen und Stellenwerte in einem Byte

Position	7	6	5	4	3	2	1	0
Stellenwert	128	64	32	16	8	4	2	1
Potenz	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bezeichnung	MSB							LSB

- **MSB** (Most Significant Bit) = Bit 7 = höchstwertiges Bit
- **LSB** (Least Significant Bit) = Bit 0 = niedrigstwertiges Bit

1.5 Der Modulo-Operator (%)

Der Modulo-Operator berechnet den **Rest einer Division**. Wenn wir `a % b` schreiben, fragen wir: „Wie viel bleibt übrig, wenn ich `a` durch `b` teile?“

1.5.1 Grundprinzip

```
a % b = Rest von (a / b)

Beispiel: 13 % 5
  13 / 5 = 2 Rest 3
  Also: 13 % 5 = 3
```

1.5.2 Warum `bit % 8 = 4`?

Wenn wir LED 5 ansteuern wollen, rechnen wir:

```
selected_id = 5
bit = selected_id - 1 = 4    // nullbasierter Index (0..9 statt 1..10)

bit % 8 = 4 % 8 = ?
```

Die Rechnung im Detail:

1. Division: $4 \div 8 = 0$ (ganzzahlig, da $4 < 8$)
2. Multiplikation zurück: $0 \times 8 = 0$
3. Rest: $4 - 0 = 4$
4. Ergebnis: `4 % 8 = 4`

Faustregel

Wenn die Zahl **kleiner** als der Divisor ist, ist das Ergebnis die Zahl selbst:

`4 % 8 = 4` (weil $4 < 8$)

1.5.3 Wozu brauchen wir `% 8`?

Der Modulo-Operator mit 8 gibt uns die **Position innerhalb eines Bytes**:

Tabelle 11: Modulo 8: Position innerhalb des Bytes

bit	bit % 8	Bedeutung
0	0	Bit 0 in Byte 0
1	1	Bit 1 in Byte 0
2	2	Bit 2 in Byte 0
...	...	...
7	7	Bit 7 in Byte 0
8	0	Bit 0 in Byte 1 (!)
9	1	Bit 1 in Byte 1

Ab Bit 8 „springt“ der Wert wieder auf 0 – wir sind im nächsten Byte.

1.6 Der Linksshift-Operator («)

Der Operator « verschiebt alle Bits einer Zahl nach links. Dabei werden rechts Nullen nachgefüllt.

1.6.1 Grundprinzip

```
a << n  bedeutet:  Verschiebe alle Bits von a um n Positionen nach links
                   Sprich: "a Linksshift n" oder "a um n nach links"

Beispiel: 1 << 4
  Start:   0b0000.0001   (die Zahl 1)
  Shift:        <-----   (4 Positionen nach links)
  Result:  0b0001.0000   (die Zahl 16)
```

1.6.2 Schritt für Schritt: `1 « 4`

Tabelle 12: Linksshift: `1 « 4` Schritt für Schritt

Operation	Binär (nibbleweise)	Dezimal	Potenz
Start: `1`	`0b0000.0001`	1	2^0
`1 « 1`	`0b0000.0010`	2	2^1
`1 « 2`	`0b0000.0100`	4	2^2
`1 « 3`	`0b0000.1000`	8	2^3
`1 « 4`	`0b0001.0000`	16	2^4

Mathematischer Zusammenhang

Linksshift um n Positionen entspricht einer Multiplikation mit 2^n:

$$1 \ll 4 = 1 \times 2^4 = 1 \times 16 = 16 = \texttt{0x10}$$

1.7 Der Rechtsshift-Operator (»)

Der Operator » ist das Gegenstück zum Linksshift – er verschiebt alle Bits nach **rechts**. Dabei fallen Bits rechts heraus und links werden Nullen nachgefüllt.

1.7.1 Grundprinzip

```
a >> n  bedeutet:  Verschiebe alle Bits von a um n Positionen nach rechts
                   Sprich: "a Rechtsshift n" oder "a um n nach rechts"

Beispiel: 16 >> 2
  Start:   0b0001.0000   (die Zahl 16)
  Shift:   <------       (2 Positionen nach rechts)
  Result:  0b0000.0100   (die Zahl 4)
```

1.7.2 Schritt für Schritt: `16 » n`

Tabelle 13: Rechtsshift: `16 » n` Schritt für Schritt

Operation	Binär (nibbleweise)	Dezimal	Division
Start: `16`	`0b0001.0000`	16	$16 \div 2^0 = 16$
`16 » 1`	`0b0000.1000`	8	$16 \div 2^1 = 8$
`16 » 2`	`0b0000.0100`	4	$16 \div 2^2 = 4$
`16 » 3`	`0b0000.0010`	2	$16 \div 2^3 = 2$
`16 » 4`	`0b0000.0001`	1	$16 \div 2^4 = 1$
`16 » 5`	`0b0000.0000`	0	$16 \div 2^5 = 0$ (abgerundet)

Mathematischer Zusammenhang

Rechtsshift um n Positionen entspricht einer **ganzzahligen Division** durch 2^n:

$$16 \gg 2 = 16 \div 2^2 = 16 \div 4 = 4$$

Achtung: Das Ergebnis wird abgerundet (keine Nachkommastellen).

1.8 Vergleich: Linksshift vs. Rechtsshift

Tabelle 14: Gegenüberstellung der Shift-Operatoren

	Linksshift «	Rechtsshift »
Symbol	« (Doppel-Kleiner)	» (Doppel-Größer)
Richtung	Bits wandern nach links	Bits wandern nach rechts
Auffüllen	Rechts mit Nullen	Links mit Nullen
Mathematik	Multiplikation mit 2^n	Division durch 2^n
Beispiel	`1 « 4 = 16`	`16 » 2 = 4`

1.8.1 Wann braucht man welchen Operator?

Tabelle 15: Typische Anwendungsfälle für Shift-Operatoren

Aufgabe	Operator	Beispiel
Bit-Maske erzeugen	«	`1 « 4` → Maske für Bit 4
Schnelle Multiplikation	«	`x « 3` → $x \times 8$
Byte-Index berechnen	»	`bit » 3` → `bit / 8`
Schnelle Division	»	`x » 2` → $x \div 4$
Bits extrahieren	»	High-Nibble: `x » 4`

1.8.2 Praxisbeispiel: Byte-Index mit Rechtsshift

Im Schieberegister-Code berechnen wir den Byte-Index mit Division:

```
1 uint8_t byteIndex = bit / 8;   // Welches Byte?
```

Der Compiler ersetzt dies intern durch einen Rechtsshift, weil Division durch 8 = Division durch 2^3:

```
1 uint8_t byteIndex = bit >> 3;  // Identisch, aber schneller
```

Tabelle 16: Byte-Index-Berechnung: Division vs. Rechtsshift

bit	bit / 8	bit » 3	Byte
0	0	0	Byte 0
4	0	0	Byte 0
7	0	0	Byte 0
8	1	1	Byte 1
15	1	1	Byte 1
16	2	2	Byte 2

Merkregel: Die Shift-Operatoren

« zeigt nach **links** → Bits nach links → **Multiplikation**
» zeigt nach **rechts** → Bits nach rechts → **Division**

1.8.3 Warum eine Bit-Maske?

Eine **Bit-Maske** ist ein Byte, bei dem genau ein Bit gesetzt ist. Sie dient dazu, ein einzelnes Bit in einem Byte zu manipulieren.

```
1 << 0  =  0b0000.0001  =  0x01   // Maske fuer Bit 0
1 << 1  =  0b0000.0010  =  0x02   // Maske fuer Bit 1
1 << 2  =  0b0000.0100  =  0x04   // Maske fuer Bit 2
1 << 3  =  0b0000.1000  =  0x08   // Maske fuer Bit 3
1 << 4  =  0b0001.0000  =  0x10   // Maske fuer Bit 4
1 << 5  =  0b0010.0000  =  0x20   // Maske fuer Bit 5
1 << 6  =  0b0100.0000  =  0x40   // Maske fuer Bit 6
1 << 7  =  0b1000.0000  =  0x80   // Maske fuer Bit 7
```

1.9 Bitweise logische Operatoren

Um einzelne Bits zu manipulieren, brauchen wir neben den Shift-Operatoren auch **logische Operatoren**. Diese arbeiten Bit für Bit – daher „bitweise“.

1.9.1 Übersicht: Die vier Operatoren

Tabelle 17: Bitweise logische Operatoren in C++

Operator	Name	Sprechweise	Anwendung
`|`	OR	„Oder“	Bits setzen
`&`	AND	„Und“	Bits prüfen / maskieren
`~`	NOT	„Nicht“	Bits invertieren
`^`	XOR	„Exklusiv-Oder“	Bits umschalten (toggle)

1.9.2 Wahrheitstabellen

Jeder Operator hat eine **Wahrheitstabelle**, die zeigt, was bei jeder Kombination von Eingabe-Bits herauskommt.

Tabelle 18: Wahrheitstabelle: OR (`|`)

A	B	A | B	Regel
0	0	0	Beide 0 → Ergebnis 0
0	1	1	Mindestens einer 1 → Ergebnis 1
1	0	1	Mindestens einer 1 → Ergebnis 1
1	1	1	Mindestens einer 1 → Ergebnis 1

Merkregel OR

OR liefert 1, wenn **mindestens ein** Eingang 1 ist.
Eselsbrücke: „Oder“ – einer **oder** der andere (oder beide).

Tabelle 19: Wahrheitstabelle: AND (`&`)

A	B	A & B	Regel
0	0	0	Nicht beide 1 → Ergebnis 0
0	1	0	Nicht beide 1 → Ergebnis 0
1	0	0	Nicht beide 1 → Ergebnis 0
1	1	1	Beide 1 → Ergebnis 1

Merkregel AND

AND liefert 1, wenn **beide** Eingänge 1 sind.
Eselsbrücke: „Und“ – dieser **und** jener müssen wahr sein.

Tabelle 20: Wahrheitstabelle: NOT (`~`)

A	~A	Regel
0	1	Aus 0 wird 1
1	0	Aus 1 wird 0

Merkregel NOT

NOT **invertiert** jedes Bit – aus 0 wird 1, aus 1 wird 0.
Eselsbrücke: „Nicht“ – das Gegenteil.

Tabelle 21: Wahrheitstabelle: XOR (`^`)

A	B	A ^ B	Regel
0	0	0	Gleich → Ergebnis 0
0	1	1	Unterschiedlich → Ergebnis 1
1	0	1	Unterschiedlich → Ergebnis 1
1	1	0	Gleich → Ergebnis 0

Merkregel XOR

XOR liefert 1, wenn die Eingänge **unterschiedlich** sind.
Eselsbrücke: „Entweder-Oder“ – aber nicht beide!

1.10 Bits manipulieren: Setzen, Löschen, Umschalten, Prüfen

Mit Bit-Masken und logischen Operatoren können wir einzelne Bits gezielt verändern, ohne die anderen Bits zu beeinflussen.

1.10.1 Bit setzen mit OR (`|=`)

Um ein Bit auf 1 zu setzen, verwenden wir **OR mit der Maske**:

```
1 ledState |= (1u << bitPos);   // Bit setzen
```

Warum funktioniert das?

```
Beispiel: Bit 4 setzen in ledState = 0b0000.0000

ledState:     0b0000.0000     (vorher)
Maske:        0b0001.0000     (1u << 4)
              -----------  OR
Ergebnis:     0b0001.0000     (nachher: Bit 4 ist jetzt 1)
```

Was passiert mit den anderen Bits?

```
Beispiel: Bit 4 setzen, aber Bit 7 ist schon 1

ledState:     0b1000.0000     (vorher: Bit 7 = 1)
Maske:        0b0001.0000     (1u << 4)
              -----------  OR
Ergebnis:     0b1001.0000     (nachher: Bit 7 bleibt, Bit 4 neu)

Bit fuer Bit betrachtet:
Position:     7 6 5 4 3 2 1 0
ledState:     1 0 0 0 0 0 0 0
Maske:        0 0 0 1 0 0 0 0
              ---------------  OR (mindestens einer 1 -> 1)
Ergebnis:     1 0 0 1 0 0 0 0
              ^     ^
              |     Neu gesetzt
              Unveraendert!
```

OR zum Setzen

`x | 0 = x` (bleibt gleich) und `x | 1 = 1` (wird 1)
Deshalb: OR mit einer Maske **setzt** das Masken-Bit, ohne andere zu verändern.

1.10.2 Bit löschen mit AND und NOT (`&= ~`)

Um ein Bit auf 0 zu setzen (zu „löschen“), verwenden wir **AND mit der invertierten Maske**:

```
ledState &= ~(1u << bitPos);  // Bit loeschen
```

Schritt für Schritt

```
Beispiel: Bit 4 loeschen in ledState = 0b0001.0000

Schritt 1: Maske erzeugen
  1u << 4  =  0b0001.0000

Schritt 2: Maske invertieren mit NOT (~)
  ~(1u << 4)  =  ~0b0001.0000  =  0b1110.1111

Schritt 3: AND mit invertierter Maske
  ledState:       0b0001.0000     (vorher: Bit 4 = 1)
  ~Maske:         0b1110.1111
                  -----------  AND
  Ergebnis:       0b0000.0000     (nachher: Bit 4 = 0)
```

Was passiert mit den anderen Bits?

```
Beispiel: Bit 4 loeschen, aber andere Bits behalten

ledState:     0b1001.0011     (vorher: mehrere Bits gesetzt)
~Maske:       0b1110.1111     (~(1u << 4))
              -----------  AND
Ergebnis:     0b1000.0011     (nachher: nur Bit 4 geloescht)

Bit fuer Bit betrachtet:
Position:     7 6 5 4 3 2 1 0
ledState:     1 0 0 1 0 0 1 1
~Maske:       1 1 1 0 1 1 1 1
              ---------------  AND (beide 1 -> 1)
Ergebnis:     1 0 0 0 0 0 1 1
                  ^
                  Geloescht (0 AND x = 0)
              ^         ^ ^
              Unveraendert (1 AND x = x)
```

AND zum Löschen

`x & 1 = x` (bleibt gleich) und `x & 0 = 0` (wird 0)
Deshalb: AND mit einer **invertierten** Maske **löscht** das Masken-Bit.

1.10.3 Bit umschalten (toggle) mit XOR (^=)

Um ein Bit umzuschalten (0→1 oder 1→0), verwenden wir **XOR mit der Maske**:

```
ledState ^= (1u << bitPos);   // Bit umschalten
```

Warum funktioniert das?

```
Beispiel 1: Bit 4 ist 0 -> wird 1
ledState:     0b0000.0000
Maske:        0b0001.0000
              -----------  XOR
Ergebnis:     0b0001.0000     (0 XOR 1 = 1)

Beispiel 2: Bit 4 ist 1 -> wird 0
ledState:     0b0001.0000
Maske:        0b0001.0000
              -----------  XOR
Ergebnis:     0b0000.0000     (1 XOR 1 = 0)
```

XOR zum Umschalten

`x ^ 0 = x` (bleibt gleich) und `x ^ 1 = ~x` (wird invertiert)
Deshalb: XOR mit einer Maske **togglet** das Masken-Bit.

1.10.4 Bit prüfen mit AND (&)

Um zu prüfen, ob ein Bit gesetzt ist, verwenden wir **AND mit der Maske**:

```
1 if (ledState & (1u << bitPos)) {
2     // Bit ist gesetzt (= 1)
3 } else {
4     // Bit ist nicht gesetzt (= 0)
5 }
```

Warum funktioniert das?

```
Beispiel 1: Bit 4 ist gesetzt
ledState:     0b0001.0000
Maske:        0b0001.0000
              -----------  AND
Ergebnis:     0b0001.0000  =  16  (nicht 0 -> true)

Beispiel 2: Bit 4 ist nicht gesetzt
ledState:     0b0000.0000
Maske:        0b0001.0000
              -----------  AND
Ergebnis:     0b0000.0000  =  0   (0 -> false)
```

AND zum Prüfen

AND mit einer Maske liefert:

- **Nicht 0** (true), wenn das Bit gesetzt ist
- **0** (false), wenn das Bit nicht gesetzt ist

1.11 Zusammenfassung: Bit-Operationen

Tabelle 22: Die vier Bit-Operationen auf einen Blick

Operation	Code	Erklärung
Bit setzen	`x |= (1u << n);`	OR: 0→1, 1→1
Bit löschen	`x &= ~(1u << n);`	AND mit NOT: 1→0, 0→0
Bit umschalten	`x ^= (1u << n);`	XOR: 0→1, 1→0
Bit prüfen	`if (x & (1u << n))`	AND: ergibt 0 oder ≠0

```
// Komplettes Beispiel: LED-Steuerung
#define LED_PIN 4

uint8_t ledState = 0;

// LED einschalten
ledState |= (1u << LED_PIN);      // Bit 4 setzen

// LED ausschalten
ledState &= ~(1u << LED_PIN);     // Bit 4 loeschen

// LED umschalten
ledState ^= (1u << LED_PIN);      // Bit 4 togglen

// LED-Status pruefen
if (ledState & (1u << LED_PIN)) {
    Serial.println("LED ist AN");
} else {
    Serial.println("LED ist AUS");
}
```

1.12 Zusammenspiel: LED 5 ansteuern

Schauen wir uns die komplette Rechnung an:

```
// Ziel: LED 5 einschalten
uint8_t selected_id = 5;

// Schritt 1: Nullbasierter Index
```

```
uint8_t bit = selected_id - 1;  // bit = 4

// Schritt 2: Welches Byte?
uint8_t byteIndex = bit / 8;    // 4 / 8 = 0 (erstes Byte)

// Schritt 3: Welche Position im Byte?
uint8_t bitPosition = bit % 8;  // 4 % 8 = 4 (fuenftes Bit, da 0-basiert)

// Schritt 4: Bit-Maske erzeugen
uint8_t mask = 1u << bitPosition;  // 1 << 4 = 0b0001.0000 = 0x10

// Schritt 5: Bit setzen (mit OR)
ledState[byteIndex] |= mask;
```

1.12.1 Visualisierung

```
LED-Nummern:     [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]
                  |    |    |    |    |    |    |    |
Bit-Positionen:   0    1    2    3    4    5    6    7

Byte vorher:   0b0000.0000
Maske:         0b0001.0000  (1 << 4)
               -----------
Byte nachher:  0b0001.0000  (LED 5 ist an)
```

1.13 Komplette Umrechnungstabelle für LEDs 1–10

Tabelle 23: 74HC595: Maskenberechnung mit nibbleweiser Darstellung

LED	bit	Byte	Bit-Pos	Binär (nibbleweise)	Hex
1	0	0	0	0b0000.0001	0x01
2	1	0	1	0b0000.0010	0x02
3	2	0	2	0b0000.0100	0x04
4	3	0	3	0b0000.1000	0x08
5	4	0	4	0b0001.0000	0x10
6	5	0	5	0b0010.0000	0x20
7	6	0	6	0b0100.0000	0x40
8	7	0	7	0b1000.0000	0x80
9	8	1	0	0b0000.0001	0x01
10	9	1	1	0b0000.0010	0x02

Merkregel: Hex-Werte der Bit-Masken

Die Hex-Werte für Bit-Masken folgen einem einfachen Muster:

1	2	4	8	16	32	64	128
Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
`0x01`	`0x02`	`0x04`	`0x08`	`0x10`	`0x20`	`0x40`	`0x80`

2 Bit-Mapping bei Schieberegistern: MSB vs. LSB

Warum ordnet der CD4021 die Taster „rückwärts“ (Bit 7→0), während der 74HC595 die LEDs „vorwärts“ (Bit 0→7) ansteuert?

2.1 Das Grundprinzip – Wie Schieberegister arbeiten

2.1.1 Was ist ein Flip-Flop?

Ein **Flip-Flop** ist die kleinste Speichereinheit in der Digitaltechnik – es kann genau **ein Bit** speichern (0 oder 1). Man kann es sich wie einen Lichtschalter vorstellen: Er ist entweder AN (1) oder AUS (0) und behält seinen Zustand, bis er aktiv umgeschaltet wird.

Tabelle 24: Flip-Flop: Die digitale Speicherzelle

Eigenschaft	Beschreibung
Speicherkapazität	Genau 1 Bit (0 oder 1)
Zustandsänderung	Nur bei Taktflanke (Clock)
Verhalten	Behält Zustand bis zum nächsten Takt
Symbol	D-Flip-Flop (D = Data)

2.1.2 Das Schieberegister als Flip-Flop-Kette

Ein **Schieberegister** ist eine Reihenschaltung von Flip-Flops. Jedes Flip-Flop gibt bei jedem Taktimpuls seinen Inhalt an das nächste weiter:

```
8-Bit Schieberegister = 8 Flip-Flops in Reihe

+------+    +------+    +------+    +------+
| FF 0 |--->| FF 1 |--->| FF 2 |--->| FF 3 |--->  ...  ---> Ausgang
+------+    +------+    +------+    +------+
   ^
Eingang
```

Flip-Flops pro Chip

- **CD4021B:** 8 Flip-Flops (8-Bit Schieberegister)
- **74HC595:** 8 Flip-Flops + 8 Ausgangs-Latches (8-Bit)
- **Kaskadiert:** 13 Chips × 8 = 104 Flip-Flops für 100 LEDs

2.1.3 Die Becherkette-Analogie

Bei jedem Taktimpuls wandert jedes Bit um eine Position weiter – wie eine **Becherkette**, bei der jeder Becher seinen Inhalt an den Nachbarn weitergibt:

```
Becherkette mit 4 Bechern:

Takt 0:  [A] [_] [_] [_]   <- A wird eingefuellt
Takt 1:  [B] [A] [_] [_]   <- A wandert weiter, B kommt nach
Takt 2:  [C] [B] [A] [_]   <- Kette fuellt sich
Takt 3:  [D] [C] [B] [A]   <- Kette voll
Takt 4:  [E] [D] [C] [B]   <- A faellt am Ende heraus!
```

Merkregel

Schieberegister = Becherkette

- Jeder Becher = 1 Flip-Flop = 1 Bit Speicher
- Jeder Takt = Alle Becher kippen gleichzeitig nach rechts
- Was links reinkommt, fällt irgendwann rechts raus

Die entscheidende Frage: **Wo kommt das Bit herein, und wo kommt es heraus?**

2.2 CD4021 – Vom Taster zum Mikrocontroller

2.2.1 Was der Chip tut

Der CD4021 ist ein **Parallel-zu-Seriell-Wandler** (PISO: Parallel In, Serial Out). Er hat 8 parallele Eingänge (P1–P8), an denen die Taster hängen. Auf Kommando (`PS`-Pin = Load) übernimmt er alle 8 Zustände gleichzeitig in sein internes Register. Danach schiebt er sie Bit für Bit über den seriellen Ausgang (Q8) zum ESP32.

2.2.2 Die Hardware-Realität

Schauen wir ins Datenblatt: Der CD4021 gibt **P8 zuerst** aus, dann P7, dann P6 ... und P1 zuletzt. Die interne Schiebekette sieht so aus:

```
Schieberichtung: ->

+-----+   +-----+   +-----+   +-----+   +-----+   +-----+   +-----+   +-----+
| P1  |-->| P2  |-->| P3  |-->| P4  |-->| P5  |-->| P6  |-->| P7  |-->| P8  |--> Q8
+-----+   +-----+   +-----+   +-----+   +-----+   +-----+   +-----+   +-----+
                                                                      ^
                                                                Serieller Ausgang
```

Nach dem LOAD-Signal liegt P8 direkt am Ausgang Q8 bereit. Mit jedem Taktimpuls rückt die Kette nach rechts:

Tabelle 25: CD4021: Bit-Reihenfolge am seriellen Ausgang Q8

Takt	Bit an Q8	Position im empfangenen Byte
1	P8	Bit 7 (MSB, zuerst empfangen)
2	P7	Bit 6
3	P6	Bit 5
4	P5	Bit 4
5	P4	Bit 3
6	P3	Bit 2
7	P2	Bit 1
8	P1	Bit 0 (LSB, zuletzt empfangen)

Ergebnis

Das empfangene Byte hat das Format `[P8][P7][P6][P5][P4][P3][P2][P1]` – das höchstwertige Bit (MSB) kam zuerst.

2.2.3 Die Verdrahtungskonvention im Projekt

Die Taster sind so angeschlossen, dass **Taster 1 an P8** hängt, **Taster 8 an P1**:

```
Physische Verdrahtung:
+---------+---------+---------+---------+---------+---------+---------+---------+
| Taster 1| Taster 2| Taster 3| Taster 4| Taster 5| Taster 6| Taster 7| Taster 8|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     |         |         |         |         |         |         |         |
     v         v         v         v         v         v         v         v
    P8        P7        P6        P5        P4        P3        P2        P1
```

Warum diese Verdrahtung? Damit die Taster-Nummer direkt der Bit-Position im empfangenen Byte entspricht:

- Taster 1 → P8 → kommt zuerst raus → landet auf Bit 7
- Taster 2 → P7 → kommt als zweites → landet auf Bit 6
- ...
- Taster 8 → P1 → kommt zuletzt → landet auf Bit 0

2.2.4 Die Formel im Code

```
1 // id = 1..8, wir wollen Bit 7..0
2 uint8_t btnMask_msb(uint8_t id) {
```

```
3     return 1u << (7 - ((id - 1) % 8));
4 }
```

Schritt für Schritt für Taster 3:

1. `id = 3`
2. `(id - 1) = 2` → nullbasierter Index
3. `(id - 1) % 8 = 2` → Position innerhalb eines Bytes
4. `7 - 2 = 5` → Bit-Position (von MSB gezählt)
5. `1 « 5 = 0b0010.0000 = 0x20` → Maske

Tabelle 26: CD4021: Maskenberechnung für Taster 1–8

Button ID	`(id-1) % 8`	`7 - …`	Binär (nibbleweise)	Hex
1	0	7	`0b1000.0000`	`0x80`
2	1	6	`0b0100.0000`	`0x40`
3	2	5	`0b0010.0000`	`0x20`
4	3	4	`0b0001.0000`	`0x10`
5	4	3	`0b0000.1000`	`0x08`
6	5	2	`0b0000.0100`	`0x04`
7	6	1	`0b0000.0010`	`0x02`
8	7	0	`0b0000.0001`	`0x01`

Für Taster 9–10 (zweiter CD4021, Byte 1) wiederholt sich das Muster:

Tabelle 27: CD4021: Masken für Taster 9–10 (Byte 1)

Button ID	Byte	Bit	Maske
9	1	7	`0x80`
10	1	6	`0x40`

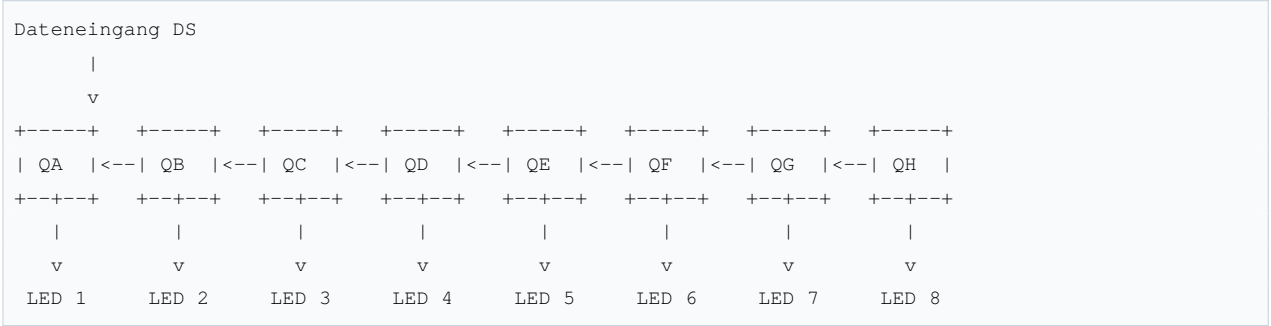
2.3 74HC595 – Vom Mikrocontroller zur LED

2.3.1 Was der Chip tut

Der 74HC595 ist ein **Seriell-zu-Parallel-Wandler** (SIPO: Serial In, Parallel Out) – das Gegenstück zum CD4021. Er empfängt Bits seriell vom ESP32 und gibt sie parallel an 8 Ausgängen (QA–QH) aus.

2.3.2 Die Hardware-Realität

Das **erste gesendete Bit** wandert durch die gesamte Kette und landet am Ende auf **QH**. Das **letzte Bit** bleibt am Anfang auf **QA**.



Wenn wir ein Byte mit `MSBFIRST` senden:

Tabelle 28: 74HC595: Bit-Verteilung beim Senden mit MSBFIRST

Takt	Gesendetes Bit	Wandert nach
1	Bit 7 (MSB)	→ QH
2	Bit 6	→ QG
3	Bit 5	→ QF
4	Bit 4	→ QE
5	Bit 3	→ QD
6	Bit 2	→ QC
7	Bit 1	→ QB
8	Bit 0 (LSB)	→ QA

Ergebnis

Bit 0 landet auf QA, Bit 7 landet auf QH.

2.3.3 Die Verdrahtungskonvention im Projekt

Die LEDs hängen so, dass **LED 1 an QA**, **LED 8 an QH**:

- LED 1 → QA → gesteuert durch Bit 0
- LED 2 → QB → gesteuert durch Bit 1
- ...
- LED 8 → QH → gesteuert durch Bit 7

2.3.4 Die Formel im Code

```
// id = 1..10, wir wollen Bit 0..9
void setOneHot(uint8_t selected_id) {
    uint8_t bit = selected_id - 1;  // 0..9
    ledState[bit / 8] |= (1u << (bit % 8));
}
```

Schritt für Schritt für LED 5:

1. `selected_id = 5`
2. `bit = 5 - 1 = 4` → nullbasierter Index
3. `bit / 8 = 0` → Byte-Index (erstes Byte)
4. `bit % 8 = 4` → Bit-Position innerhalb des Bytes
5. `1 « 4 = 0b0001.0000 = 0x10` → Maske

Tabelle 29: 74HC595: Maskenberechnung für LEDs 1–10

LED ID	`bit`	Byte	Bit-Pos	Binär (nibbleweise)	Hex
1	0	0	0	`0b0000.0001`	`0x01`
2	1	0	1	`0b0000.0010`	`0x02`
3	2	0	2	`0b0000.0100`	`0x04`
4	3	0	3	`0b0000.1000`	`0x08`
5	4	0	4	`0b0001.0000`	`0x10`
6	5	0	5	`0b0010.0000`	`0x20`
7	6	0	6	`0b0100.0000`	`0x40`
8	7	0	7	`0b1000.0000`	`0x80`
9	8	1	0	`0b0000.0001`	`0x01`
10	9	1	1	`0b0000.0010`	`0x02`

2.4 Warum die Asymmetrie?

Die unterschiedliche Bit-Reihenfolge ist kein Fehler – sie ergibt sich aus der **Schieberichtung** der beiden Chips:

Tabelle 30: Vergleich der Schieberichtung beider Chips

Chip	Typ	Schieberichtung	Erstes Bit	Letztes Bit
CD4021	PISO	P1 → P2 → ...→ Q8	P8 → Bit 7 (MSB)	P1 → Bit 0 (LSB)
74HC595	SIPO	DS → QH → ...→ QA	Bit 7 → QH	Bit 0 → QA

Beide Chips verwenden `MSBFIRST` als SPI-Modus. Das bedeutet:

- **CD4021:** Der Chip gibt MSB zuerst aus → wir empfangen MSB zuerst → BTN1 liegt auf Bit 7
- **74HC595:** Wir senden MSB zuerst → MSB wandert nach hinten (QH) → LED1 braucht Bit 0

2.5 Zusammenfassung: Das Bit-Layout

CD4021 (Taster einlesen) – MSB first

```
Byte 0: [BTN1][BTN2][BTN3][BTN4][BTN5][BTN6][BTN7][BTN8]
         Bit7  Bit6  Bit5  Bit4  Bit3  Bit2  Bit1  Bit0
        (MSB)                                   (LSB)

Byte 1: [BTN9][BTN10][ - ][ - ][ - ][ - ][ - ][ - ]
         Bit7  Bit6  (ungenutzt)
```

74HC595 (LEDs ansteuern) – LSB mapping

```
Byte 0: [LED1][LED2][LED3][LED4][LED5][LED6][LED7][LED8]
         Bit0  Bit1  Bit2  Bit3  Bit4  Bit5  Bit6  Bit7
        (LSB)                                   (MSB)

Byte 1: [LED9][LED10][ - ][ - ][ - ][ - ][ - ][ - ]
         Bit0  Bit1  (ungenutzt)
```

2.5.1 Die Formeln auf einen Blick

```
1 // Taster: MSB-first -> Bit 7 zuerst
2 uint8_t btnMask  = 1u << (7 - ((id - 1) % 8));
3 uint8_t btnByte  = (id - 1) / 8;
4
5 // LEDs: LSB-mapping -> Bit 0 zuerst
6 uint8_t ledMask  = 1u << ((id - 1) % 8);
7 uint8_t ledByte  = (id - 1) / 8;
```

Fazit

Die Software kompensiert die Hardware-bedingte Asymmetrie durch unterschiedliche Masken-Berechnungen – und am Ende passt **Taster i zu LED i**.