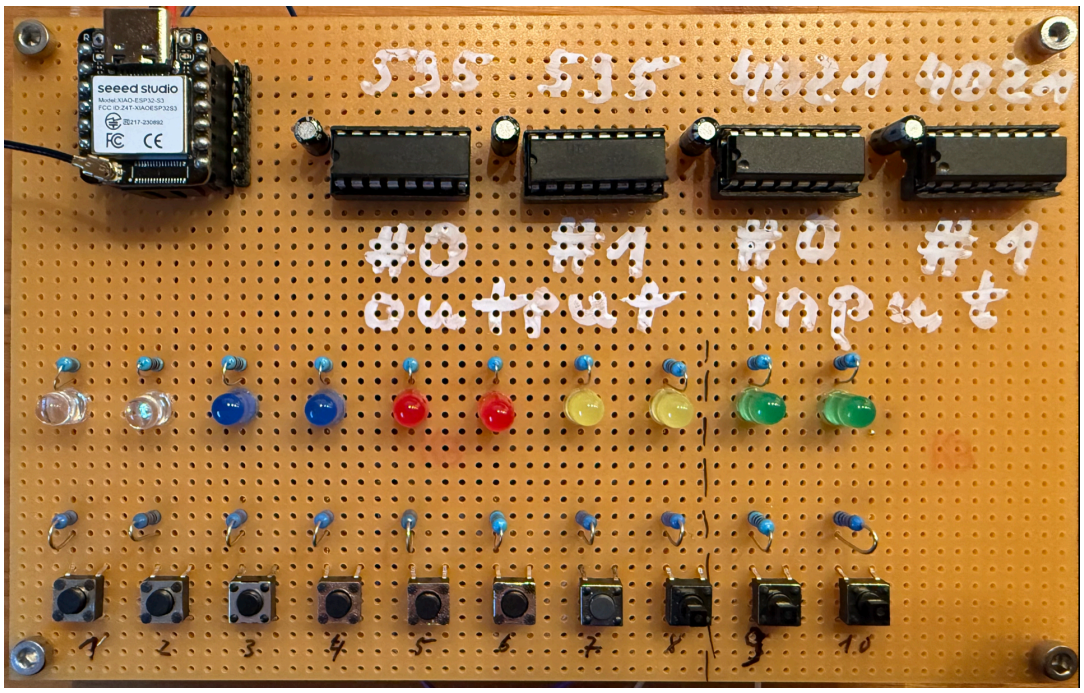
Schieberegister

74HC595 + CD4021BE

Serial-In/Parallel-Out • Parallel-In/Serial-Out

I/O-Erweiterung für Mikrocontroller



Themen

SIPO • PISO • SPI • Kaskadierung • Debouncing • Bit-Mapping

Stand: 1. Januar 2026
Technische Dokumentation

Jan Unger
Version: 1.1.0

Inhaltsverzeichnis

1 Schieberegister: 74HC595 und CD4021BE

1.1 Motivation: Warum Schieberegister?

Stellen wir uns ein konkretes Problem vor: Ein ESP32-S3 soll 100 LEDs steuern und 100 Taster einlesen. Direkt verdrahtet bräuchten wir 200 GPIO-Pins – der ESP32-S3 hat aber nur etwa 20 nutzbare. Die Lösung liegt in der **seriellen Kommunikation**: Wir wandeln parallele Signale in serielle um und umgekehrt.

Die Analogie dazu: Ein Güterzug. Statt 100 LKWs gleichzeitig durch eine enge Straße zu schicken (100 Spuren nötig), reihen wir die Container hintereinander auf einem Gleis auf. Am Ziel werden sie wieder parallel verteilt.

Tabelle 1: Übersicht der Schieberegister-Bausteine

Baustein	Richtung	Typ	Funktion
74HC595	MCU → Außenwelt	SIPO	Serial-In, Parallel-Out
CD4021B	Außenwelt → MCU	PISO	Parallel-In, Serial-Out

1.2 Der 74HC595 – Daten hinausschreiben

1.2.1 Architektur

Wenn wir das Blockdiagramm im Datenblatt betrachten, erkennen wir drei Stufen:

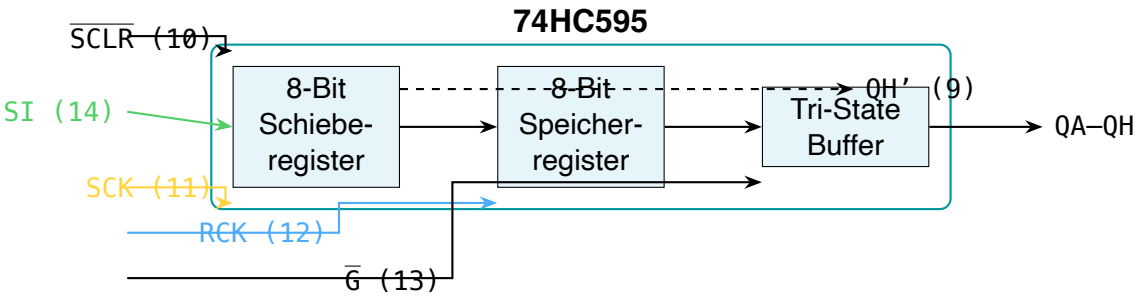


Abbildung 1: Blockdiagramm des 74HC595

Das Prinzip: Daten werden Bit für Bit in das Schieberegister getaktet. Erst wenn alle 8 Bits drin sind, kopiert ein Latch-Impuls den gesamten Inhalt ins Speicherregister – und die Ausgänge ändern sich gleichzeitig.

1.2.2 Signalbeschreibung

Tabelle 2: 74HC595 Pinbelegung und Funktion

Pin	Symbol	Funktion	Aktiv	Beschreibung
14	SI	Serial Input	–	Serieller Dateneingang
11	SCK	Shift Clock	↑	Positive Flanke übernimmt SI
12	RCK	Register Clock	↑	Kopiert Schieberegister → Speicher
10	$\overline{\text{SCLR}}$	Shift Clear	LOW	Löscht das Schieberegister
13	$\overline{\text{G}}$	Output Enable	LOW	Aktiviert die Ausgänge
9	QH′	Serial Out	–	Für Kaskadierung

1.2.3 Timing-Sequenz

Um das Muster `0b10110001` auszugeben, gehen wir wie folgt vor:

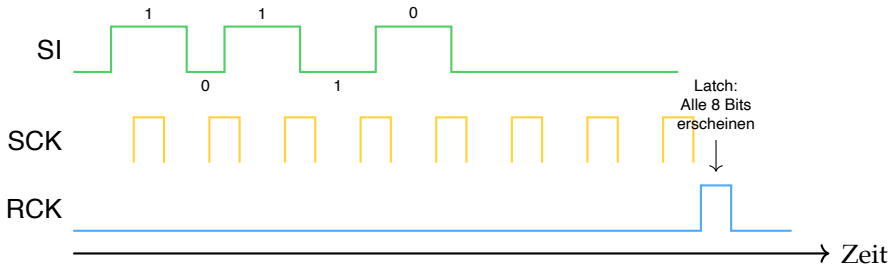


Abbildung 2: Timing-Diagramm: Daten in den 74HC595 schieben

Kritischer Punkt

Die Ausgänge ändern sich erst bei der RCK-Flanke – nicht während des Schiebens. Das verhindert „Geisterbilder“ auf den LEDs.

1.2.4 Kaskadierung

Der `QH'`-Ausgang (Pin 9) liefert das „herausgeschobene“ Bit. Verbinden wir `QH'` des ersten ICs mit `SI` des zweiten, entsteht eine 16-Bit-Kette:

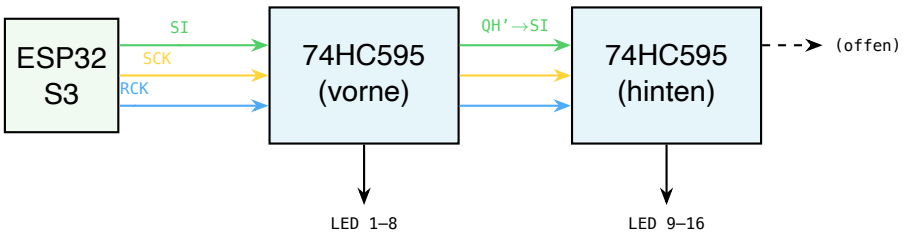


Abbildung 3: Kaskadierung zweier 74HC595

Reihenfolge beachten

Wir schieben zuerst die Bits für das **hintere** IC, dann für das vordere. Das erste Bit „rutscht“ durch bis zum Ende der Kette.

1.3 Der CD4021B – Daten hereinlesen

1.3.1 Architektur

Der CD4021B arbeitet spiegelverkehrt: Er liest 8 parallele Eingänge gleichzeitig ein und schiebt sie seriell heraus.

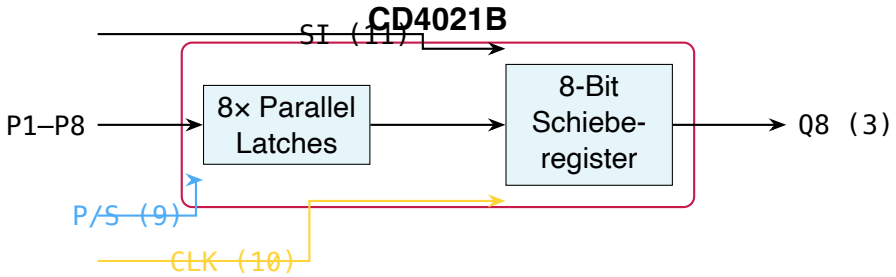


Abbildung 4: Blockdiagramm des CD4021B

1.3.2 Die zwei Betriebsmodi

Der P/S-Pin (Parallel/Serial) steuert das Verhalten:

Tabelle 3: CD4021B Betriebsmodi

P/S	Modus	Aktion bei CLK ↑
HIGH	Parallel Load	Alle 8 Eingänge werden in die Latches übernommen
LOW	Serial Shift	Daten werden um 1 Position geschoben

1.3.3 Timing-Sequenz

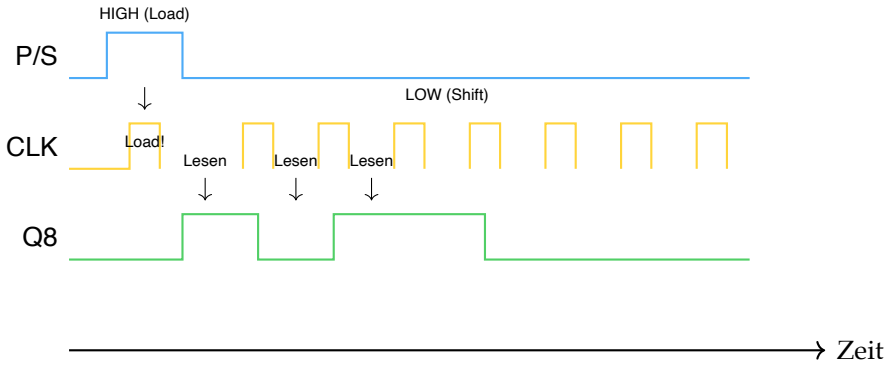


Abbildung 5: Timing-Diagramm: Daten aus dem CD4021B lesen

Kritische Lese-Sequenz

Nach dem Load-Puls liegt `Q8` **sofort** am Ausgang – noch vor dem ersten Clock. Die korrekte Sequenz ist:

1. P/S = HIGH → CLK ↑ (Load)
2. P/S = LOW
3. **Q8 lesen** (erstes Bit!)
4. CLK ↑ (Shift)
5. Q8 lesen (zweites Bit)
6. …wiederholen

1.4 Firmware-Implementierung

1.4.1 Konfiguration

```
constexpr uint8_t NUM_LEDS = 10;
constexpr uint8_t NUM_BTNS = 10;
constexpr uint8_t SHIFT_BITS = 16;   // 2x 8-bit Shiftregister
constexpr uint32_t DEBOUNCE_MS = 30; // Entprellen
constexpr uint32_t POLL_MS = 1;      // Poll-Rate begrenzen
```

Listing 1: Zentrale Konfiguration

Warum `constexpr`? Diese Werte werden zur Compile-Zeit ausgewertet – kein RAM-Verbrauch, kein Laufzeit-Overhead. Bei Skalierung auf 100× ändern wir nur diese Zeilen.

1.4.2 LED-Ausgabe: Der 74HC595-Treiber

```
inline void write595(uint8_t u3, uint8_t u2) {
    digitalWrite(LED_LATCH_PIN, LOW);                    // Latch sperren
    shiftOut(LED_DATA_PIN, LED_CLOCK_PIN, MSBFIRST, u3); // IC hinten
    shiftOut(LED_DATA_PIN, LED_CLOCK_PIN, MSBFIRST, u2); // IC vorne
    digitalWrite(LED_LATCH_PIN, HIGH);                   // Latch freigeben
}
```

Listing 2: 74HC595 Schreibfunktion

Reihenfolge erklärt: Das zuerst gesendete Byte (`u3`) „rutscht“ durch das vordere IC hindurch ins hintere. Das zweite Byte (`u2`) bleibt im vorderen IC. Deshalb: hinten zuerst, vorne zuletzt.

1.4.3 Taster-Einlesen: Der CD4021B-Treiber

```
inline uint16_t read4021Stream16() {
    // 1) Parallel Load
    digitalWrite(BTN_LOAD_PIN, HIGH);
    delayMicroseconds(2);  // t_WH: min 160 ns @ 5V
    digitalWrite(BTN_LOAD_PIN, LOW);
    delayMicroseconds(2);  // t_REM: Warten bis Q8 stabil

    // 2) Serial Shift: MSB-first
    uint16_t v = 0;
    for (uint8_t i = 0; i < SHIFT_BITS; i++) {
        // ERST lesen (Q8 enthaelt aktuelles Bit)
        v <<= 1;
        v |= (digitalRead(BTN_DATA_PIN) ? 1u : 0u);

        // DANN clocken (schiebt naechstes Bit nach Q8)
        if (i < (SHIFT_BITS - 1)) {
            digitalWrite(BTN_CLOCK_PIN, HIGH);
            delayMicroseconds(1);
            digitalWrite(BTN_CLOCK_PIN, LOW);
```

```
            delayMicroseconds(1);
        }
    }
    return v;
}
```

Listing 3: CD4021B Lesefunktion (korrigiert)

Die kritische Stelle

Wir lesen **vor** dem Clock-Puls, nicht danach. Nach dem Parallel Load liegt das erste Bit bereits an Q8. Der Clock schiebt das nächste Bit herbei.

1.4.4 Bit-Mapping: Hardware-Abstraktion

```
constexpr uint8_t BTN_BIT_MAP[NUM_BTNS] = {
    15, // BTN1  -> Bit 15 (IC#1, Q8)
    12, // BTN2  -> Bit 12 (IC#1, Q5)
    13, // BTN3  -> Bit 13 (IC#1, Q6)
    // ... weitere Mappings
};

inline uint16_t readButtonsPressedMask() {
    const uint16_t stream = read4021Stream16();

    uint16_t pressed = 0;
    for (uint8_t i = 0; i < NUM_BTNS; i++) {
        const uint8_t sb = BTN_BIT_MAP[i];              // Physische Position
        const bool level = ((stream >> sb) & 1u) != 0;  // Bit extrahieren
        const bool isPressed = BTN_ACTIVE_LOW ? (!level) : level;
        if (isPressed)
            pressed |= (static_cast<uint16_t>(1u) << i); // Logische Position
    }
    return pressed;
}
```

Listing 4: Mapping physisch → logisch

Warum ein Mapping? Die physische Verdrahtung entspricht selten der logischen Numerierung. Statt im Code zu rechnen, definieren wir eine Lookup-Tabelle. Bei Verdrahtungsänderungen passen wir nur diese Tabelle an.

1.4.5 Debouncing

Mechanische Taster prellen – sie schalten beim Drücken mehrfach ein und aus (typisch 5 ms bis 20 ms). Die Lösung: Wir warten, bis der Zustand **stabil** bleibt.

```
inline void buttonsTask(uint32_t nowMs) {
    const uint16_t raw = readButtonsPressedMask();

    // Rohaenderung: Zeitpunkt merken
    if (raw != g_btnRaw) {
        g_btnRaw = raw;
        g_btnChangedAt = nowMs;  // Timer neu starten
    }

    // Wenn lange genug stabil -> uebernehmen
    if ((nowMs - g_btnChangedAt) >= DEBOUNCE_MS &&
        g_btnStable != g_btnRaw) {
        const uint16_t prev = g_btnStable;
        g_btnStable = g_btnRaw;

        // Rising Edge: 0 -> 1 (Taste wurde gedrueckt)
        const uint16_t rising = (g_btnStable & ~prev);
        if (rising) {
            // Toggle-Logik hier...
        }
    }
}
```

Listing 5: Debouncing mit Flankenerkennung

1.5 Timing-Analyse

1.5.1 Sind unsere Delays ausreichend?

Tabelle 4: Timing-Parameter CD4021B (bei 5 V)

Parameter	Symbol	Min	Unser Delay	Status
P/S Pulse Width	t_{WH}	80 ns	2000 ns	✓
P/S Removal Time	t_{REM}	140 ns	2000 ns	✓
Clock Pulse Width	t_W	40 ns	1000 ns	✓

Bei 3,3 V (statt 5 V) verlängern sich alle Zeiten um etwa Faktor 2. Unsere Delays haben einen Sicherheitsfaktor von ca. 25× – mehr als ausreichend.

1.5.2 Gesamtzeit für einen Durchlauf

$$t_{\text{Load}} = 2\,\mu\text{s} + 2\,\mu\text{s} = 4\,\mu\text{s} \tag{1}$$

$$t_{\text{Shift}} = 16 \times (\text{digitalRead} + 2\,\mu\text{s}) \approx 16 \times 3\,\mu\text{s} = 48\,\mu\text{s} \tag{2}$$

$$t_{\text{Summe}} \approx 52\,\mu\text{s pro Scan} \tag{3}$$

Bei `POLL_MS` = 1 ms haben wir ca. 950 μs Reserve.

1.6 Skalierung auf 100×

Für 100 LEDs und 100 Taster brauchen wir 13 ICs je Typ ($13 \times 8 = 104 \geq 100$).

1.6.1 Änderungen im Code

```
// Vorher (10x)
constexpr uint8_t NUM_LEDS = 10;
constexpr uint8_t SHIFT_BITS = 16;
static uint16_t g_ledState = 0;

// Nachher (100x)
constexpr uint8_t NUM_LEDS = 100;
constexpr uint8_t NUM_SHIFT_BYTES = 13;  // Aufrunden: (100+7)/8
static uint8_t g_ledState[NUM_SHIFT_BYTES] = {0};
```

Listing 6: Skalierung auf 100×

1.6.2 Optimierung: Hardware-SPI

Bei 100× wird Bit-Banging zum Flaschenhals. Der ESP32-S3 hat Hardware-SPI:

Tabelle 5: Geschwindigkeitsvergleich

Methode	**16 Bits**	**104 Bits**
`shiftOut()`	≈64 µs	≈416 µs
Hardware-SPI @ 4 MHz	≈4 µs	≈26 µs
Faktor	16×	16×

1.7 Zusammenfassung

Die Kombination aus 74HC595 (Ausgänge) und CD4021B (Eingänge) löst das Pin-Multiplex-Problem elegant:

Tabelle 6: Vergleich: 10× vs. 100× System

Aspekt	**10× (aktuell)**	**100× (Ziel)**	**Änderung**
GPIO-Pins	6	6	keine
ICs	4	26	+22
Scan-Zeit	≈100 µs	≈500 µs	5×
Code-Änderungen	–	Konstanten + Arrays	minimal

Das Grundprinzip bleibt identisch – nur die Skalierung ändert sich. Genau das macht gutes Hardware-Abstraktions-Design aus.