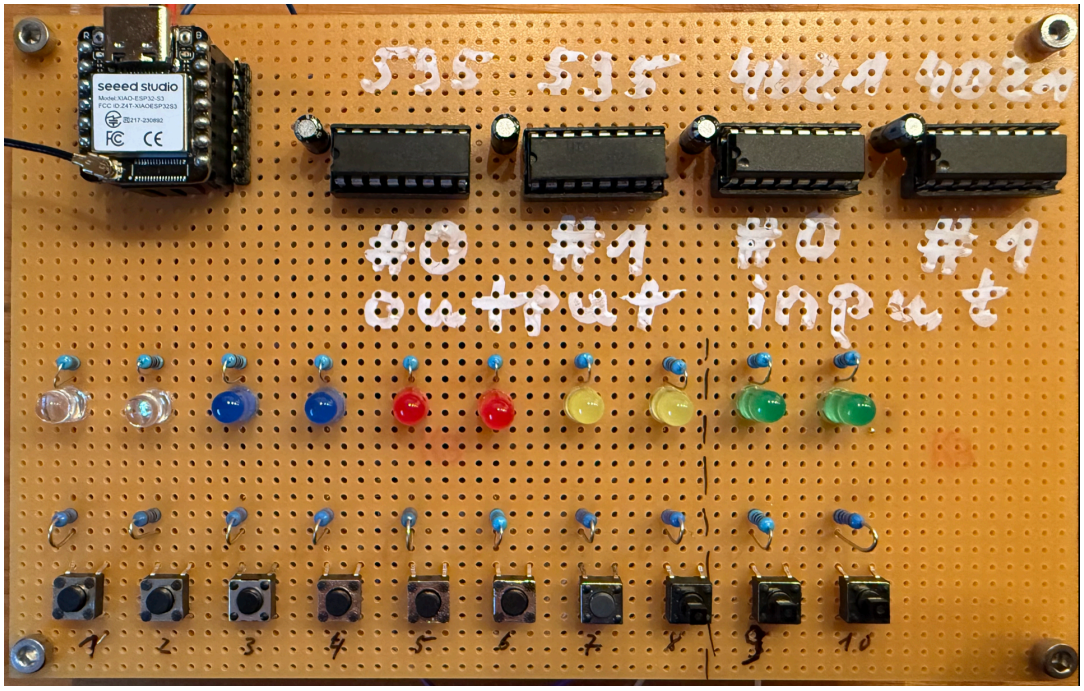
Schieberegister

74HC595 + CD4021B

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

I/O-Erweiterung für Mikrocontroller



Themen

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

Stand: 08.01.2026
Technische Dokumentation

Jan Unger
Version: 2.5.2

Inhaltsverzeichnis

1 Schieberegister: 74HC595 und CD4021B

1.1 Motivation: Warum Schieberegister?

Ein ESP32-S3 soll 100 LEDs steuern und 100 Taster einlesen. Direkt verdrahtet bräuchten wir 200 GPIO-Pins – doch der ESP32-S3 bietet nur etwa 20 nutzbare. Wie lösen wir dieses Missverhältnis?

Die Antwort liegt in der **seriellen Kommunikation**: Wir wandeln parallele Signale in serielle um und umgekehrt. Die Analogie: Statt 100 LKWs gleichzeitig durch eine enge Straße zu schicken (100 Spuren nötig), reihen wir die Container hintereinander auf einem Gleis auf. Am Ziel verteilen wir sie wieder parallel.

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

Werfen wir einen Blick auf das Blockdiagramm in Abbildung 1. Wir erkennen drei Stufen, die wie ein Staffellauf funktionieren:

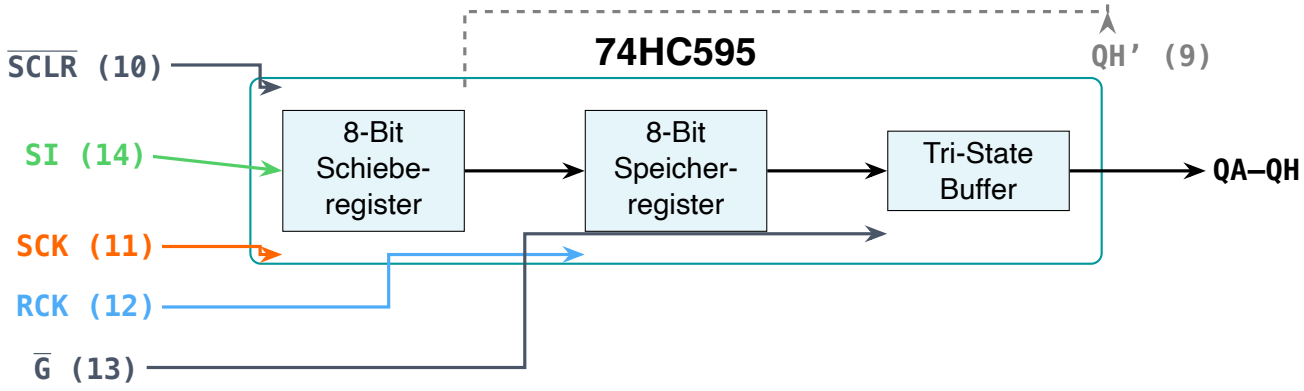


Abbildung 1: Blockdiagramm des 74HC595

Das Prinzip dahinter: Daten wandern Bit für Bit in das Schieberegister. Erst wenn alle 8 Bits angekommen sind, kopiert ein Latch-Impuls den gesamten Inhalt ins Speicher-

register – und die Ausgänge ändern sich *gleichzeitig*. Diese Entkopplung verhindert Flackern während des Schiebens.

1.2.2 Signalbeschreibung

Welche Signale brauchen wir, um den 74HC595 anzusteuern? Tabelle 2 zeigt die wichtigsten Pins:

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

Doch wie sieht das Zusammenspiel dieser Signale konkret aus? Um das Muster `0b10110001` auszugeben, durchlaufen wir folgende Sequenz – das Timing-Diagramm in Abbildung 2 macht den Ablauf sichtbar:

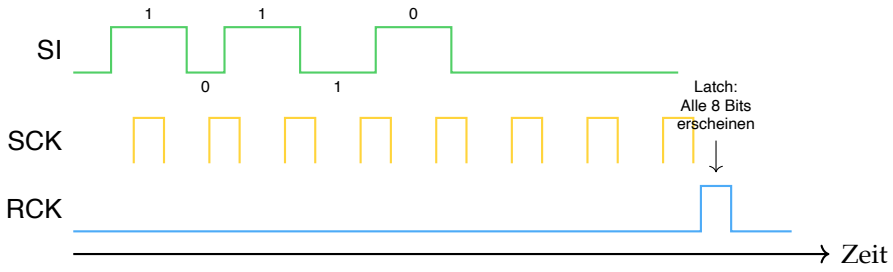


Abbildung 2: Timing-Diagramm: Daten in den 74HC595 schieben

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

Was aber, wenn 8 Ausgänge nicht reichen? 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 zeigt das Prinzip:

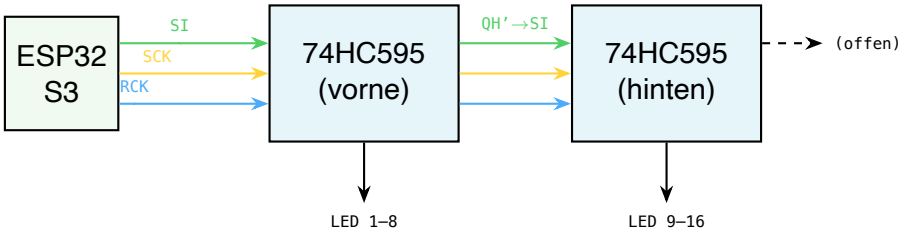


Abbildung 3: Kaskadierung zweier 74HC595

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 zum 74HC595: Er liest 8 parallele Eingänge gleichzeitig ein und schiebt sie seriell heraus. Das Blockdiagramm in Abbildung 4 verdeutlicht den zweistufigen Aufbau:

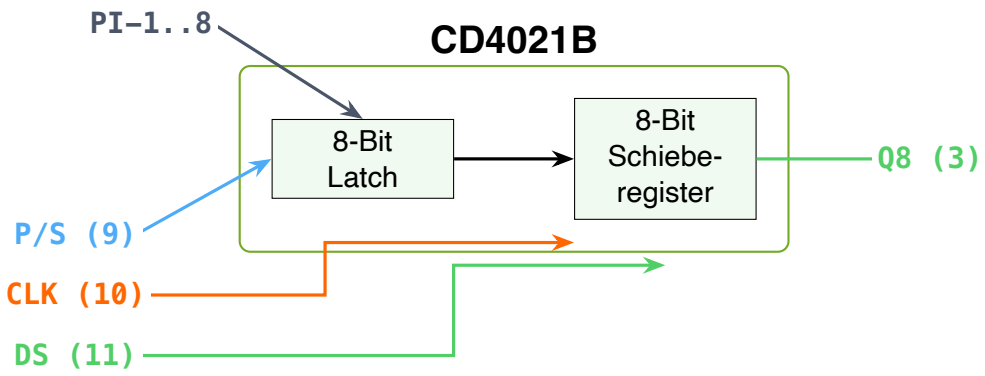


Abbildung 4: Blockdiagramm des CD4021B

1.3.2 Signalbeschreibung

Tabelle 3: CD4021B Pinbelegung und Funktion

Pin	Symbol	Funktion	Aktiv	Beschreibung
1,4–7,13–15	PI-8..1	Parallel Inputs	–	8 parallele Eingänge
9	P/S	Parallel/Serial	HIGH	HIGH = Load, LOW = Shift
10	CLK	Clock	↑	Positive Flanke schiebt
11	DS	Serial Input	–	Für Kaskadierung
3	Q8	Serial Output	–	Serieller Datenausgang

Invertierte Logik!

Der CD4021B hat eine **invertierte** Load-Logik: P/S = HIGH für Parallel Load, P/S = LOW für Serial Shift. Das ist das Gegenteil dessen, was man intuitiv erwarten würde!

1.3.3 Timing-Sequenz und First-Bit-Problem

Beim CD4021B liegt nach dem Parallel Load bereits das erste Bit (PI-1, MSB) an Q8 an – **bevor** wir den ersten Clock-Puls geben. Abbildung 5 zeigt diesen kritischen Ablauf:

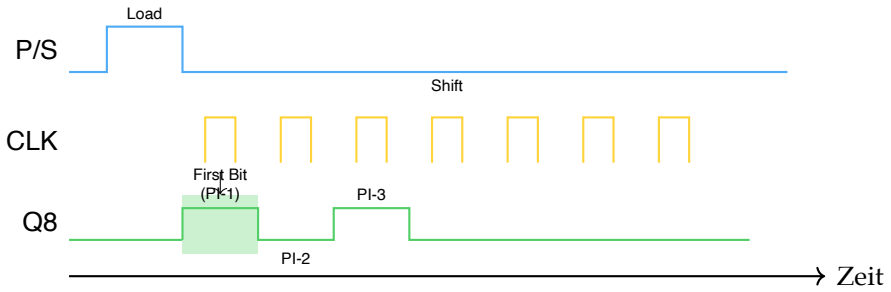


Abbildung 5: CD4021B Timing: First Bit liegt sofort nach Load an Q8

First-Bit-Rescue

Das erste Bit muss **vor** dem SPI-Transfer gelesen werden! Nach dem Parallel Load (P/S = HIGH → LOW) liegt PI-1 bereits an Q8. Der erste Clock-Puls würde es „wegschieben“, bevor wir es lesen können.

1.3.4 Bit-Reihenfolge: MSB-first

Der CD4021B gibt die Bits in MSB-first-Reihenfolge aus:

- Nach Load: PI-1 (MSB) an Q8
- Nach 1. Clock: PI-2 an Q8
- Nach 7. Clock: PI-8 (LSB) an Q8

Hardware-Verdrahtung beachten

Die physische Verdrahtung bestimmt das Bit-Mapping:

- **BTN 1** → PI-8 (Pin 1) → erscheint als Bit 0 im Datenstrom
- **BTN 8** → PI-1 (Pin 7) → erscheint als Bit 7 im Datenstrom

Die Firmware abstrahiert dies mit: `btn_bit(id) = 7 - ((id - 1) % 8)`

1.3.5 Kaskadierung

Für mehr als 8 Eingänge verbinden wir `Q8` des hinteren ICs mit `DS` des vorderen:

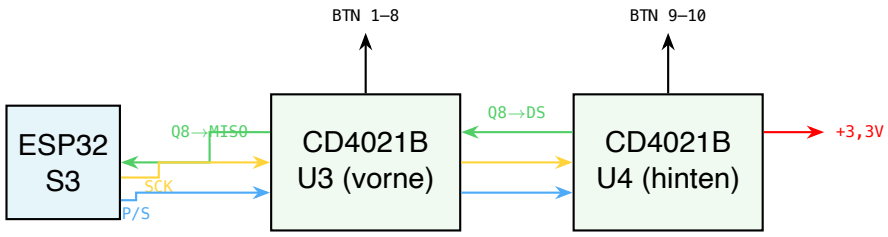


Abbildung 6: Kaskadierung: CD4021B U4 (hinten) → U3 (vorne) → ESP32

DS des letzten ICs auf VCC!

Der `DS`-Eingang des letzten ICs in der Kette **muss** auf +3,3V (VCC) gelegt werden! CMOS-Eingänge dürfen niemals floaten – das führt zu zufälligen Werten und erhöhtem Stromverbrauch.

1.4 Firmware-Implementierung

1.4.1 ESP32-S3 Pinbelegung

Wir nutzen einen gemeinsamen SPI-Bus für beide IC-Typen, mit separaten Steuerleitungen:

Tabelle 4: ESP32-S3 XIAO Pinbelegung

Signal	Pin	Ziel	SPI-Mode
MOSI	D10	74HC595 SER (Pin 14)	MODE0, 1 MHz
SCK	D8	Alle CLK (gemeinsam)	–
RCK	D0	74HC595 RCLK (Pin 12)	–
OE	D2	74HC595 OE (Pin 13)	PWM optional
MISO	D9	CD4021B Q8 (Pin 3)	MODE1, 500 kHz
P/S	D1	CD4021B P/S (Pin 9)	–

1.4.2 CD4021B mit First-Bit-Rescue

```
uint16_t cd4021_read(uint8_t* buffer, size_t numBytes) {
    // 1) Parallel Load: P/S HIGH -> LOW
    digitalWrite(BTN_LOAD_PIN, HIGH);
    delayMicroseconds(2);
    digitalWrite(BTN_LOAD_PIN, LOW);
    delayMicroseconds(2);

    // 2) First Bit Rescue: PI-1 liegt bereits an Q8!
    uint8_t firstBit = digitalRead(BTN_DATA_PIN);

    // 3) SPI Transfer (restliche Bits)
    SPI.beginTransaction(SPISettings(500000, MSBFIRST,
       SPI_MODE1));
    SPI.transfer(buffer, numBytes);
    SPI.endTransaction();

    // 4) First Bit einsetzen (MSB von Byte 0)
    buffer[0] = (buffer[0] >> 1) | (firstBit << 7);

    return combineBytes(buffer, numBytes);
}
```

Listing 1: CD4021B First-Bit-Rescue

Die kritische Stelle

Wir lesen **vor** dem SPI-Transfer, nicht danach. Nach dem Parallel Load liegt das erste Bit bereits an `Q8`. Der SPI-Clock würde das nächste Bit herbei schieben und das erste Bit wäre verloren.

1.4.3 Bit-Mapping: Hardware-Abstraktion

```
// Firmware-Abstraktion: btn_bit(id) = 7 - ((id - 1) % 8)
// BTN 1 -> PI-8 (Pin 1) -> Bit 0 im Datenstrom
// BTN 8 -> PI-1 (Pin 7) -> Bit 7 im Datenstrom

inline bool isButtonPressed(uint16_t stream, uint8_t btnId) {
    uint8_t byteIdx = (btnId - 1) / 8;
    uint8_t bitPos = 7 - ((btnId - 1) % 8);
    uint8_t mask = 1 << bitPos;

    // Active-Low: gedrückt = 0
    return ((stream >> (byteIdx * 8)) & mask) == 0;
}
```

Listing 2: Mapping physisch → logisch

Warum ein Mapping? Die physische Verdrahtung (BTN 1 → PI-8) entspricht nicht der logischen Nummerierung. Statt im Code zu rechnen, definieren wir eine klare Formel. Bei Verdrahtungsänderungen passen wir nur diese Formel an – der restliche Code bleibt unberührt.

1.4.4 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 3: Debouncing mit Flankenerkennung

1.5 Timing-Analyse

Funktioniert unser Code schnell genug? Schauen wir uns die Zahlen an.

1.5.1 Sind unsere Delays ausreichend?

Tabelle 5: 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

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

$$t_{\text{SPI}} = 16\ \text{Bits} \times \frac{1}{500\,\text{kHz}} = 32\,\mu\text{s} \tag{2}$$

$$t_{\text{Summe}} \approx 40\,\mu\text{s pro Scan (Hardware-SPI)} \tag{3}$$

Bei `POLL_MS` $= 5\,$ms haben wir ca. $4960\,\mu$s Reserve – das System ist weit vom Limit entfernt.

1.6 Skalierung auf 100×

Für 100 LEDs und 100 Taster brauchen wir 13 ICs je Typ ($13 \times 8 = 104 \geq 100$). Doch was ändert sich im Code?

1.6.1 Änderungen im Code

```
// Vorher (10x)
constexpr uint8_t NUM_LEDS = 10;
constexpr uint8_t NUM_SHIFT_BYTES = 2;
static uint8_t g_ledState[NUM_SHIFT_BYTES] = {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 4: Skalierung auf 100×

1.6.2 Transfer-Zeiten bei 100×

Tabelle 6: Geschwindigkeitsvergleich: 10× vs. 100×

Komponente	**10× (16 Bits)**	**100× (104 Bits)**
74HC595 @ 1 MHz	≈16 μs	≈104 μs
CD4021B @ 500 kHz	≈32 μs	≈208 μs
Gesamt	≈50 μs	≈320 μs

Selbst bei 100× bleibt die Scan-Zeit unter 500 µs – bei einem 5 ms-Polling-Intervall haben wir noch 90% Reserve.

Hardware-SPI ist Pflicht

Bei 100× ist Hardware-SPI Pflicht. Bit-Banging würde ca. 16× länger dauern und wäre nicht mehr praktikabel.

1.7 Zusammenfassung

Die Kombination aus 74HC595 (Ausgänge) und CD4021B (Eingänge) löst das Pin-Multiplex-Problem elegant. Tabelle 7 fasst den Vergleich zwischen dem aktuellen 10×-System und dem Ziel-System mit 100× zusammen:

Tabelle 7: Vergleich: 10× vs. 100× System

Aspekt	10× (aktuell)	100× (Ziel)	Änderung
GPIO-Pins	6	6	keine
ICs	4	26	+22
Scan-Zeit	≈50 µs	≈320 µs	6×
Code-Änderungen	–	Konstanten + Arrays	minimal

Das Grundprinzip bleibt identisch – nur die Skalierung ändert sich. Genau das macht gutes Hardware-Abstraktions-Design aus: Die Architektur skaliert, ohne dass wir das Rad neu erfinden müssen.