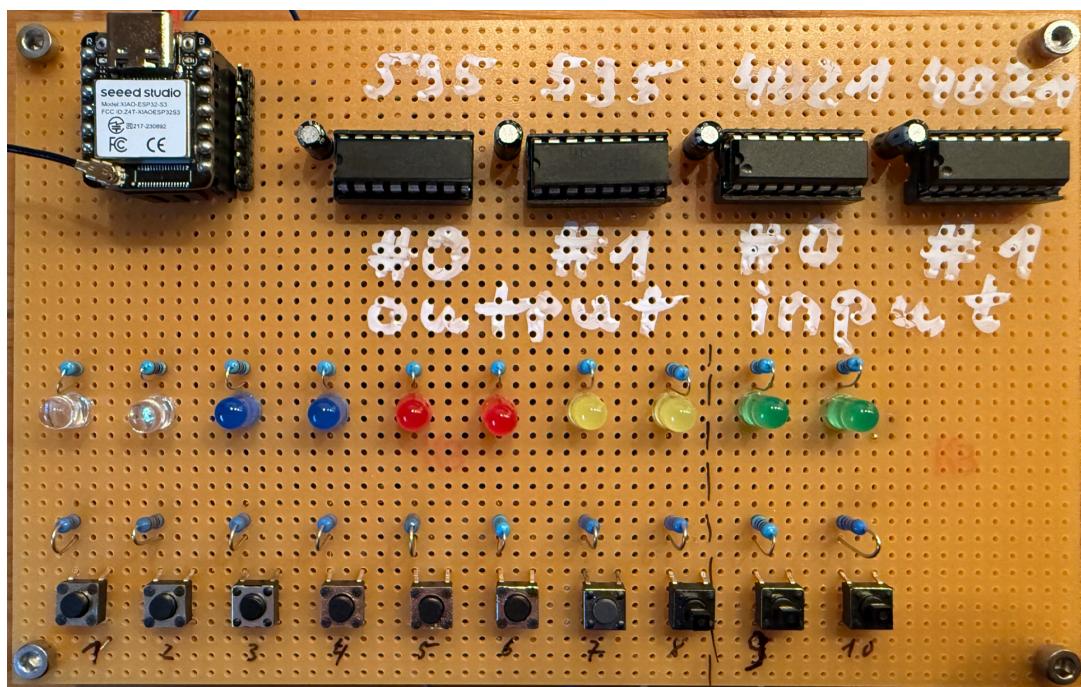


Schieberegister

74HC595 + CD4021BE

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

I/O-Erweiterung für Mikrocontroller



Themen

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

Inhaltsverzeichnis

1 Schieberegister: 74HC595 und CD4021BE	1
1.1 Motivation: Warum Schieberegister?	1
1.2 Der 74HC595 – Daten hinausschreiben	1
1.3 Der CD4021B – Daten hereinlesen	3
1.4 Firmware-Implementierung	4
1.5 Timing-Analyse	8
1.6 Skalierung auf 100×	9
1.7 Zusammenfassung	10

1 Schieberegister: 74HC595 und CD4021BE

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:

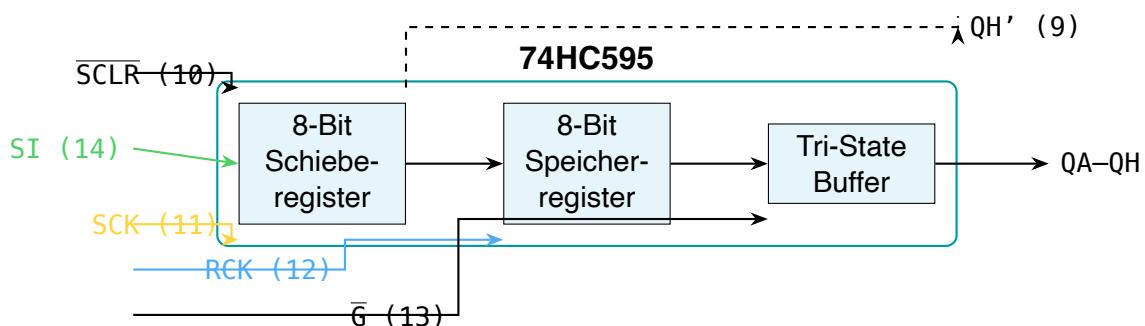


Abbildung 1: Blockdiagramm des 74HC595

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 Speicherregister – 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	SCLR	Shift Clear	LOW	Löscht das Schieberegister
13	\bar{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:

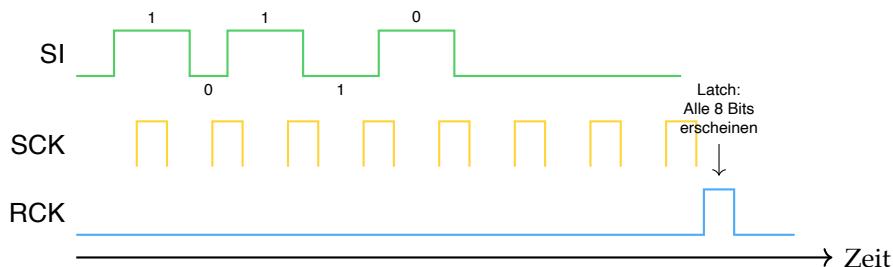


Abbildung 2: Timing-Diagramm: Daten in den 74HC595 schieben

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:

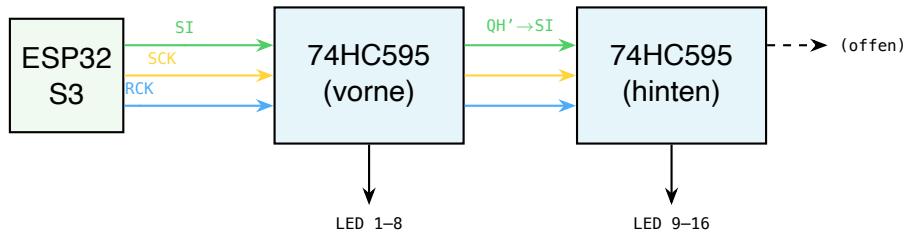


Abbildung 3: Kaskadierung zweier 74HC595

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:

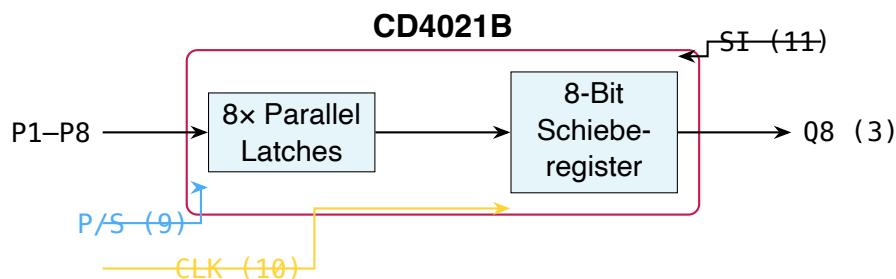


Abbildung 4: Blockdiagramm des CD4021B

1.3.2 Die zwei Betriebsmodi

Der P/S-Pin (Parallel/Serial) steuert, wie sich der Baustein verhält. Tabelle 3 zeigt die beiden Modi:

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

Hier wird es knifflig – und ein Blick auf Abbildung 5 zeigt warum: Nach dem Load-Puls liegt das erste Bit **sofort** am Ausgang, noch bevor wir den ersten Clock senden.

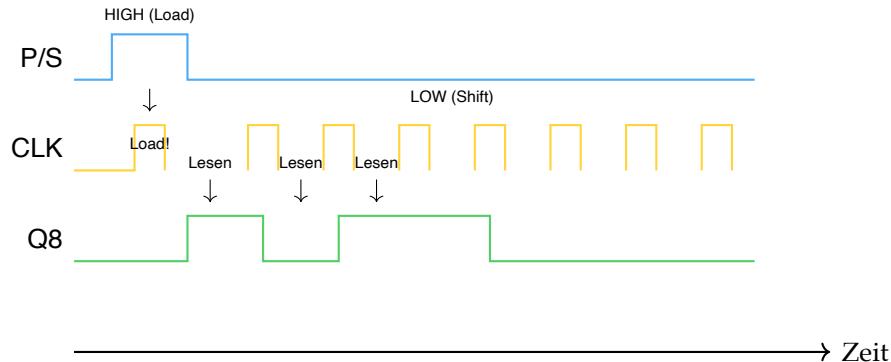


Abbildung 5: Timing-Diagramm: Daten aus dem CD4021B lesen

Kritische Lese-Sequenz

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

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

Warum kein Hardware-SPI für CD4021B?

SPI sendet immer zuerst einen Clock, dann liest es. Der CD4021B liefert das erste Bit aber **vor** dem ersten Clock:

SPI-Sequenz: Clock → Lesen → Clock → Lesen → ...

CD4021B braucht: **Lesen** → Clock → Lesen → Clock → ...

Hardware-SPI würde das erste Bit verpassen. Deshalb bleibt CD4021B bei Bit-Banging.

1.4 Firmware-Implementierung

Wie setzen wir das Ganze in Code um? Beginnen wir mit der zentralen Konfiguration.

1.4.1 Konfiguration

```

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

```

Listing 1: Zentrale Konfiguration

Warum **constexpr**? Der Compiler wertet diese Werte zur Compile-Zeit aus – kein RAM-Verbrauch, kein Laufzeit-Overhead. Bei einer Skalierung auf 100× ändern wir nur diese Zeilen.

1.4.2 LED-Ausgabe: Der 74HC595-Treiber

```

1 inline void write595(uint8_t u3, uint8_t u2) {
2     digitalWrite(LED_LATCH_PIN, LOW);                                // Latch
3     sperren
4     shiftOut(LED_DATA_PIN, LED_CLOCK_PIN, MSBFIRST, u3); // IC
hinten
5     shiftOut(LED_DATA_PIN, LED_CLOCK_PIN, MSBFIRST, u2); // IC
vorne
6     digitalWrite(LED_LATCH_PIN, HIGH);                                // Latch
freigeben
}

```

Listing 2: 74HC595 Schreibfunktion

Die Reihenfolge ist entscheidend: 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

```

1 inline uint16_t read4021Stream16() {
2     // 1) Parallel Load
3     digitalWrite(BTN_LOAD_PIN, HIGH);
4     delayMicroseconds(2); // t_WH: min 160 ns @ 5V
5     digitalWrite(BTN_LOAD_PIN, LOW);

```

```

6     delayMicroseconds(2); // t_Rem: Warten bis Q8 stabil
7
8     // 2) Serial Shift: MSB-first
9     uint16_t v = 0;
10    for (uint8_t i = 0; i < SHIFT_BITS; i++) {
11        // ERST lesen (Q8 enthaelt aktuelles Bit)
12        v <<= 1;
13        v |= (digitalRead(BTN_DATA_PIN) ? 1u : 0u);
14
15        // DANN clocken (schiebt naechstes Bit nach Q8)
16        if (i < (SHIFT_BITS - 1)) {
17            digitalWrite(BTN_CLOCK_PIN, HIGH);
18            delayMicroseconds(1);
19            digitalWrite(BTN_CLOCK_PIN, LOW);
20            delayMicroseconds(1);
21        }
22    }
23    return v;
24 }
```

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

```

1 constexpr uint8_t BTN_BIT_MAP[NUM_BTNS] = {
2     15, // BTN1 -> Bit 15 (IC#1, Q8)
3     12, // BTN2 -> Bit 12 (IC#1, Q5)
4     13, // BTN3 -> Bit 13 (IC#1, Q6)
5     // ... weitere Mappings
6 };
7
8 inline uint16_t readButtonsPressedMask() {
9     const uint16_t stream = read4021Stream16();
10
11     uint16_t pressed = 0;
```

```

12     for (uint8_t i = 0; i < NUM_BTNS; i++) {
13         const uint8_t sb = BTN_BIT_MAP[i];           // 
14             Physische Position
15         const bool level = ((stream >> sb) & 1u) != 0; // Bit
16             extrahieren
17         const bool isPressed = BTN_ACTIVE_LOW ? (!level) : level;
18         if (isPressed)
19             pressed |= (static_cast<uint16_t>(1u) << i); // 
20                 Logische Position
21     }
22     return pressed;
23 }
```

Listing 4: Mapping physisch → logisch

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

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.

```

1  inline void buttonsTask(uint32_t nowMs) {
2      const uint16_t raw = readButtonsPressedMask();
3
4      // Rohaenderung: Zeitpunkt merken
5      if (raw != g_btnRaw) {
6          g_btnRaw = raw;
7          g_btnChangedAt = nowMs; // Timer neu starten
8      }
9
10     // Wenn lange genug stabil -> uebernehmen
11     if ((nowMs - g_btnChangedAt) >= DEBOUNCE_MS &&
12         g_btnStable != g_btnRaw) {
13         const uint16_t prev = g_btnStable;
14         g_btnStable = g_btnRaw;
15
16         // Rising Edge: 0 -> 1 (Taste wurde gedrueckt)
```

```

17     const uint16_t rising = (g_btnStable & ~prev);
18     if (rising) {
19         // Toggle-Logik hier...
20     }
21 }
22 }
```

Listing 5: 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 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

$$t_{\text{Load}} = 2 \mu\text{s} + 2 \mu\text{s} = 4 \mu\text{s} \quad (1)$$

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

$$t_{\text{Summe}} \approx 52 \mu\text{s} \text{ pro Scan} \quad (3)$$

Bei POLL_MS = 1 ms haben wir ca. 950 μs Reserve – das System ist alles andere als am Limit.

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

```

1 // Vorher (10x)
2 constexpr uint8_t NUM_LEDS = 10;
3 constexpr uint8_t SHIFT_BITS = 16;
4 static uint16_t g_ledState = 0;
5
6 // Nachher (100x)
7 constexpr uint8_t NUM_LEDS = 100;
8 constexpr uint8_t NUM_SHIFT_BYTES = 13; // Aufrunden: (100+7)/8
9 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 bietet Hardware-SPI – wie viel schneller ist das? Tabelle 5 zeigt den Vergleich:

Tabelle 5: Geschwindigkeitsvergleich

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

Hardware-SPI bringt einen Faktor 16 – das macht bei 100× den Unterschied zwischen „knapp“ und „entspannt“.

Umverdrahtung für Hardware-SPI:

Tabelle 6: Pin-Änderungen für Hardware-SPI

Signal	Aktuell (10×)	Hardware-SPI (100×)	Bemerkung
SER (Data)	D0	D10 (MOSI)	umverdrahten
SRCLK (Clock)	D1	D8 (SCK)	umverdrahten
RCLK (Latch)	D2	D2	bleibt

CD4021B bleibt bei Bit-Banging

Hardware-SPI können wir nur für die 74HC595 (Ausgänge) verwenden. Der CD4021B (Eingänge) bleibt bei Bit-Banging, da SPI das erste Bit nach dem Parallel Load verpassen würde.

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	≈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: Die Architektur skaliert, ohne dass wir das Rad neu erfinden müssen.