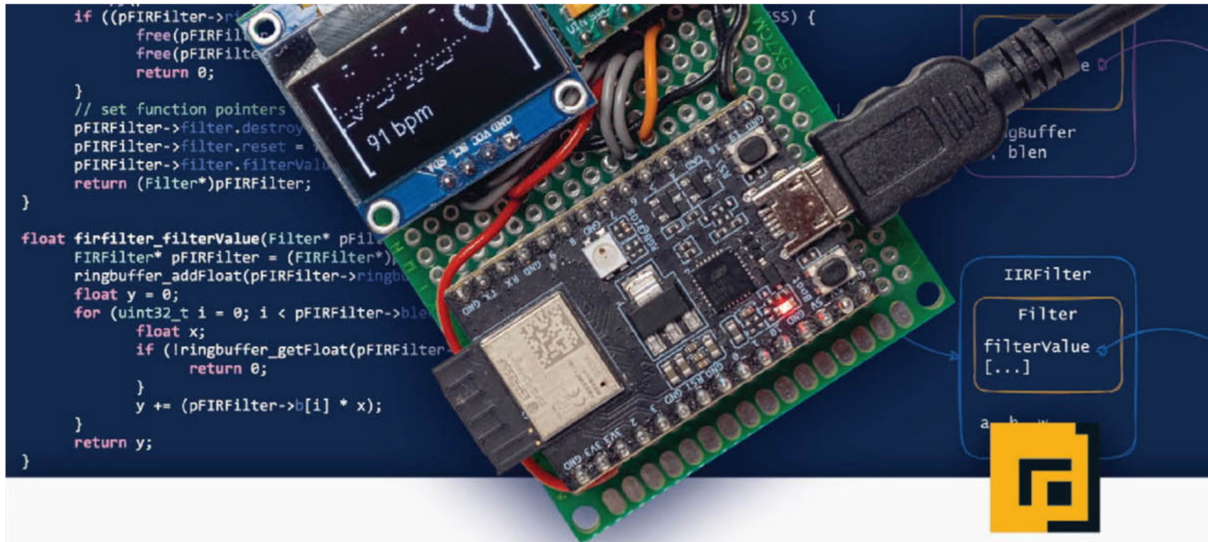

Patrick Ritschel

Embedded Systems mit RISC-V und ESP32-C3

Eine praktische Einführung in Architektur, Peripherie und eingebettete Programmierung

dpunkt.verlag

Inhalt

Cover

Über den Autor

Titel

Impressum

Vorwort

Inhaltsverzeichnis

I Mikrocontrollergrundlagen

1 Einleitung

1.1 Ziel des Buchs

1.2 Struktur des Buches

1.3 Zielpublikum

1.4 Gebrauchsanweisung

1.4.1 Konventionen

2 Hallo, Welt!

2.1 Wahl der Programmiersprache

2.2 Benötigte Komponenten für die Applikationsentwicklung

2.2.1 Development Board

2.2.2 Software für die Entwicklung

2.3 Die erste Applikation

3 Der Mikroprozessor

3.1 Prozessorarchitektur

3.1.1 Eine kleine Aufgabe

3.1.2 Die Registerbank

3.1.3 Die Arithmetic Logic Unit (ALU)

3.1.4 Datenspeicher

3.1.5 Befehlsspeicher
3.1.6 Steuerwerk
3.1.7 Weitere Einheiten
3.1.8 Der Prozessor
3.1.9 Pipeline
3.2 Instruction Set Architecture
3.2.1 RISC-V
3.2.2 sum_up_n in Assembler
3.2.3 sum_up_n-Maschinensprache
3.3 Performance
3.3.1 Control and Status Registers
3.3.2 Funktionsaufruf
3.3.3 Optimierung des Codes
3.3.4 Änderung des Verfahrens
4 Der Mikrocontroller
4.1 Aufbau eines Mikrocontrollers
4.1.1 Test des Zufallszahlengenerators
4.1.2 Das Bussystem
4.1.3 ESP32-C3 Memory Map
4.2 Speicher
4.2.1 Speichertechnologien
4.2.2 Speicherzugriffe in Software
4.2.3 Cache
4.2.4 Linker
4.3 Peripheriemodule
4.3.1 Peripheriezugriff
4.3.2 Durchführung des Zufallszahlentests
4.3.3 Informationen der Hersteller

4.3.4 Speicherlayout der Peripherie
4.3.5 Bits als Schalter
4.4 Bitmaskierung
4.4.1 Klassische Aussagenlogik
4.4.2 Bitweise Operatoren in C
4.4.3 Bitmaskierung
4.5 Zusammenfassung
II Peripheriemodule
5 Digitale Ein-/Ausgabe
5.1 Peripherie
5.2 Projekt Pulsoximeter
5.3 Elektrotechnische Grundlagen
5.3.1 Strom und Spannung
5.3.2 Widerstand und Ohm’sches Gesetz
5.3.3 Halbleiter und Diode
5.3.4 Schaltungsaufbau »LED an Batterie«
5.4 LED schalten
5.4.1 Transistor
5.4.2 Logische Funktionen mit CMOS
5.4.3 GPIO-Modul
5.4.4 Schaltungsaufbau ESP32-C3 mit LEDs
5.4.5 Pin-Multiplexing
5.4.6 Set-/Reset-Register
5.4.7 Bitfeld und Union in C
5.4.8 Gesamtes Modul kapseln
5.4.9 API des Herstellers
5.4.10 Oszilloskop als Hilfsmittel
5.4.11 Kondensator

5.4.12 Leistung, Arbeit, Batterielebensdauer
5.5 Taster anschließen
5.5.1 GPIO Eingangssignalpfad
6 Interrupts und Exceptions
6.1 Exceptions und Interrupts
6.1.1 RISC-V-Ausnahmebehandlung
6.1.2 Aktivierung des Interrupts
6.1.3 Exception Handler
6.2 Schichtenarchitektur und Callback
6.2.1 Schichtenarchitektur
6.2.2 Callbacks
6.3 Interrupt bei Tastendruck
6.4 Sourcecodeverwaltung
6.4.1 Module in Unterverzeichnissen
6.4.2 Komponentenmodell des ESP-IDF
6.4.3 Versionsverwaltung
7 Externe Komponenten digital anschließen
7.1 Display ansteuern
7.2 Konfiguration im ESP-IDF
7.3 I2C-Protokoll
7.3.1 SMBus
7.4 SPI-Schnittstelle
7.4.1 Bit-Banging
7.4.2 DMA: Direct Memory Access
7.4.3 Dateispeicherung auf SD-Karten
7.5 WS2812B
7.6 Weitere Kommunikationsschnittstellen
7.6.1 Serielle Schnittstelle, RS-232

7.6.2 I2S
7.6.3 CAN
7.6.4 Funkschnittstellen
8 Analoge Werte verarbeiten
8.1 Die Welt ist analog
8.1.1 Abtastung (Sampling)
8.1.2 Analog-Digital-Wandlung
8.1.3 Messen am Spannungsteiler
8.2 Werte filtern
8.2.1 Filterimplementierung
8.3 Den Herzschlag erkennen
8.3.1 Diskrete Fourier-Transformation
8.4 Die Zeit messen
8.4.1 Taktgeber
8.5 Das Timer-Modul
8.5.1 Timer des ESP32-C3
8.5.2 Systemzeit und Kalenderzeit
8.5.3 Zeitsynchronisierung
8.5.4 Pulsweitenmodulation (PWM)
8.5.5 Weitere Komponenten
8.6 Zusammenfassung
III Embedded System
9 Embedded Betriebssystem
9.1 Embedded Applikationsmodell
9.2 Multitasking
9.3 Echtzeitbetriebssystem
9.3.1 FreeRTOS
9.4 Nebenläufigkeit

9.4.1 Semaphor

9.4.2 Kritische Region

9.4.3 Deadlock

9.4.4 Producer/Consumer

9.4.5 Message-Queue

9.4.6 Mutex und Signalisierung

9.4.7 Prioritätenbasiertes Scheduling

9.5 Systemkontext

9.6 Gerätetreiber

9.6.1 POSIX-Standard

10 Internet der Dinge

10.1 Internet

10.1.1 Wi-Fi-Konfiguration

10.1.2 Berkeley Sockets

10.1.3 UDP

10.1.4 TCP

10.1.5 Datenformate

10.1.6 Header

10.2 Cloud-Zugriff

10.2.1 REST und CoAP

10.2.2 MQTT-Protokoll

10.2.3 Webserver

10.3 Bluetooth

10.3.1 NimBLE Stack

10.3.2 Generic Access Profile (GAP)

10.3.3 GATT-Profil und ATT-Protokoll

10.4 Power-Management

10.4.1 Sleep Modes

10.4.2 Power-Management-Algorithmus

11 Schlusswort

IV Anhang

A Webseite zum Buch

A.1 Material zum ESP32-C3 und ESP-IDF

A.2 Beispiele des Buchs

A.3 Übungsbeispiele

A.4 Errata

Literaturverzeichnis

Index

4 Der Mikrocontroller

> *»The unknown was always so attractive to me … and still is.«*
>
> HEDY LAMARR

Im vorigen Kapitel wurde der Mikroprozessor als zentrale Komponente eines Rechners erläutert. Für den Betrieb in einem Embedded System werden weitere Komponenten wie Speicher und I/O Interfaces benötigt. In diesem Kapitel wird der Aufbau eines Mikrocontrollers, der viele dieser Komponenten auf einem Chip vereinigt, besprochen.

Anhand eines Beispiels, das erst auf den Speicher zugreift und dann erweitert wird, um den Hardware-Zufallszahlengenerator zu verwenden, werden der Aufbau eines Mikrocontrollers, der Zugriff auf die Peripherie über Memory-Mapped I/O und die Anwendung in der Programmiersprache C gezeigt.

4.1 Aufbau eines Mikrocontrollers

Um den Anforderungen an Embedded Systeme gerecht zu werden, wird als zentrale rechnende Komponente oft ein Mikrocontroller eingesetzt, der, salopp formuliert, die gemeinsame Unterbringung eines Mikroprozessors mit Peripheriekomponenten auf einem einzigen Chip ist. Man spricht hier auch von einem SoC (System-on-a-Chip).

Das Hardwaredesign der Zielplattform wird dadurch stark vereinfacht, dass einzelne Komponenten wie zum Beispiel der RAM-Speicher des Systems nicht als separater Chip integriert werden müssen, sondern im Mikrocontroller bereits vorhanden sind. Um zu verstehen, welche Auswirkungen das Design und die Architektur eines Mikrocontrollers auf die Programmerstellung haben, ist es sinnvoll, dessen Aufbau näher anzusehen.

Abb. 4–1 zeigt den typischen schematischen Aufbau eines Mikrocontrollers in einem Blockschaltbild. Die einzelnen Komponenten werden hier im Überblick besprochen, einer detaillierteren Besprechung widmen sich dieses Kapitel und der gesamte Teil II des Buchs.

Ein zentraler Bestandteil, der aber nur einen Teil der Siliziumfläche ausmacht, ist der Mikroprozessor (grün markiert, siehe Kapitel 3), der zentrale Berechnungs-

und Steuerungsaufgaben übernimmt.

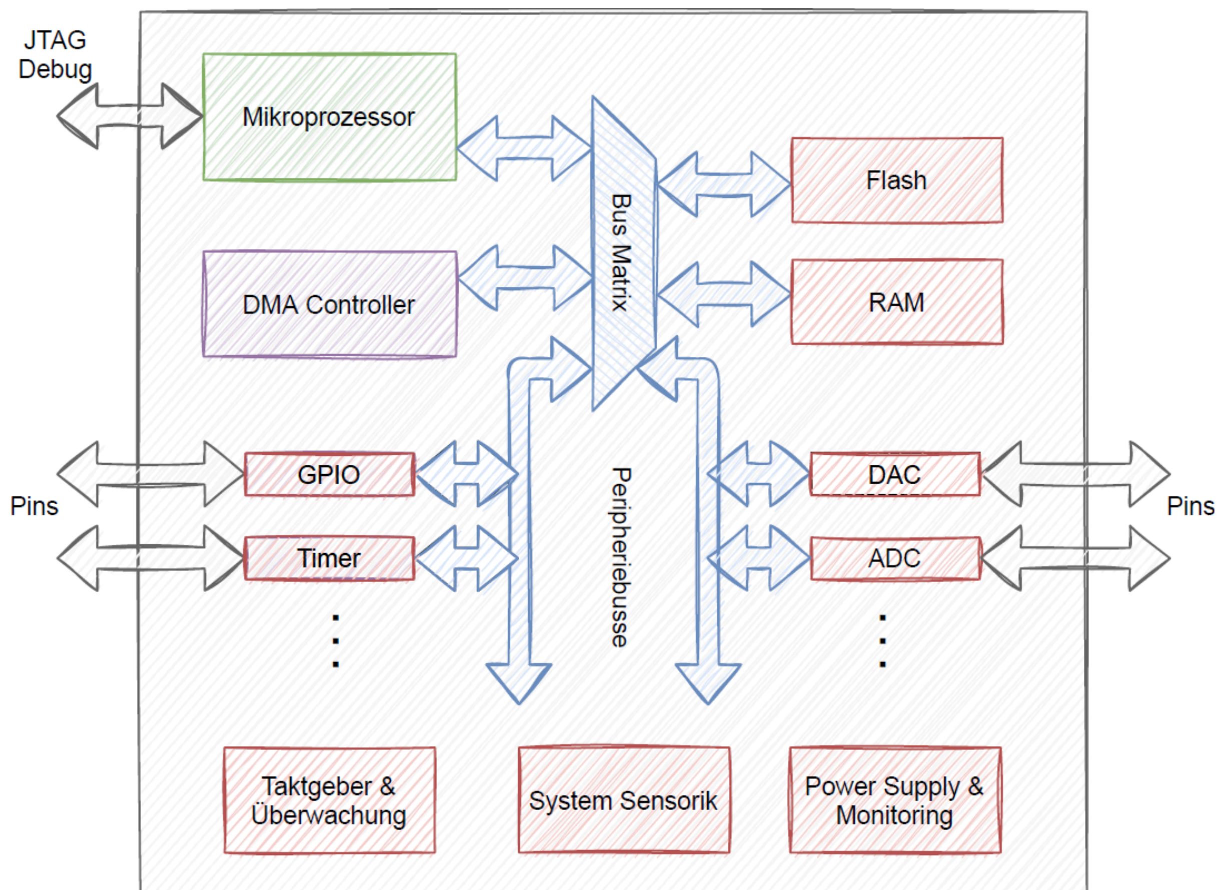

Abb. 4–1 *Schematischer Aufbau eines Mikrocontrollers im Blockschaltbild*

Verschiedene Module, die spezialisierte Tätigkeiten übernehmen können, sind rot eingezeichnet. Diese »Peripherie« (Abschnitt 4.3) beinhaltet Module wie GPIO (Abschnitt 5.4.3) zum Setzen und Lesen externer digitaler Signale, Timer/Counter zum Zählen von Ereignissen (Abschnitt 8.5), DAC/ADC zur Messung und Erzeugung externer analoger Signale (Kapitel 8) und viele mehr. Die grauen Pfeile zu den Chipkontakten (»Pins«) bzw. zum JTAG Interface (Abschnitt 2.2) symbolisieren die externen Schnittstellen. Zur Peripherie zählen auch Speicher wie RAM und Flash (Abschnitt 4.2).

Damit die CPU auf die Peripherie zugreifen kann, kommunizieren sie über ein Bussystem (Abschnitt 4.1.2), an das sie angeschlossen sind, miteinander. Das hierarchisch strukturierte Bussystem moderner Embedded Systeme ist blau eingezeichnet. Der Bus arbeitet in der Zuteilung der Teilnehmer mit Adressen: Jedem Teilnehmer ist ein eigener Adressbereich zugeordnet, auf dem er erreichbar ist. Die Adressierung selbst erfolgt mit der Granularität einzelner Bytes. Die CPU greift als Master, die anderen Module als Slaves auf den Bus zu. Der Master greift steuernd auf den Bus zu, die Slaves dürfen nur antworten.

Schwierig wird es, wenn mehrere Teilnehmer als Master auf einen Bus zugreifen möchten. Der lila eingezeichnete DMA-Controller (»Direct Memory Access«, Abschnitt 7.4.2) ist eine Komponente, die zur Entlastung der CPU auf Peripherie zugreifen kann. So ist es beispielsweise möglich, dass der DMA-Controller Daten von einer Kommunikationsschnittstelle in das RAM überträgt, während der Prozessor »schläft« oder anderen Tätigkeiten nachgeht.

Das Bussystem muss in diesem Fall unterstützen, dass mehrere Master gleichzeitig zugreifen können. In der Abbildung ermöglicht die »Bus Matrix« die Arbitrierung, also den gleichzeitigen Zugriff zum Durchschleusen der Daten zwischen den Mastern und den Slaves, oder die Reihung des Zugriffs, wenn mehrere Master auf dieselben Busse zugreifen möchten.

Der Mikrocontroller beherbergt noch weitere Komponenten, die nicht zwingend an den Bus angeschlossen sein müssen. Überwachung und Steuerung von Größen wie Prozessortakt (Abschnitt 8.4.1), Temperatur und Spannung garantieren ein reibungsloses Funktionieren der Hardware.

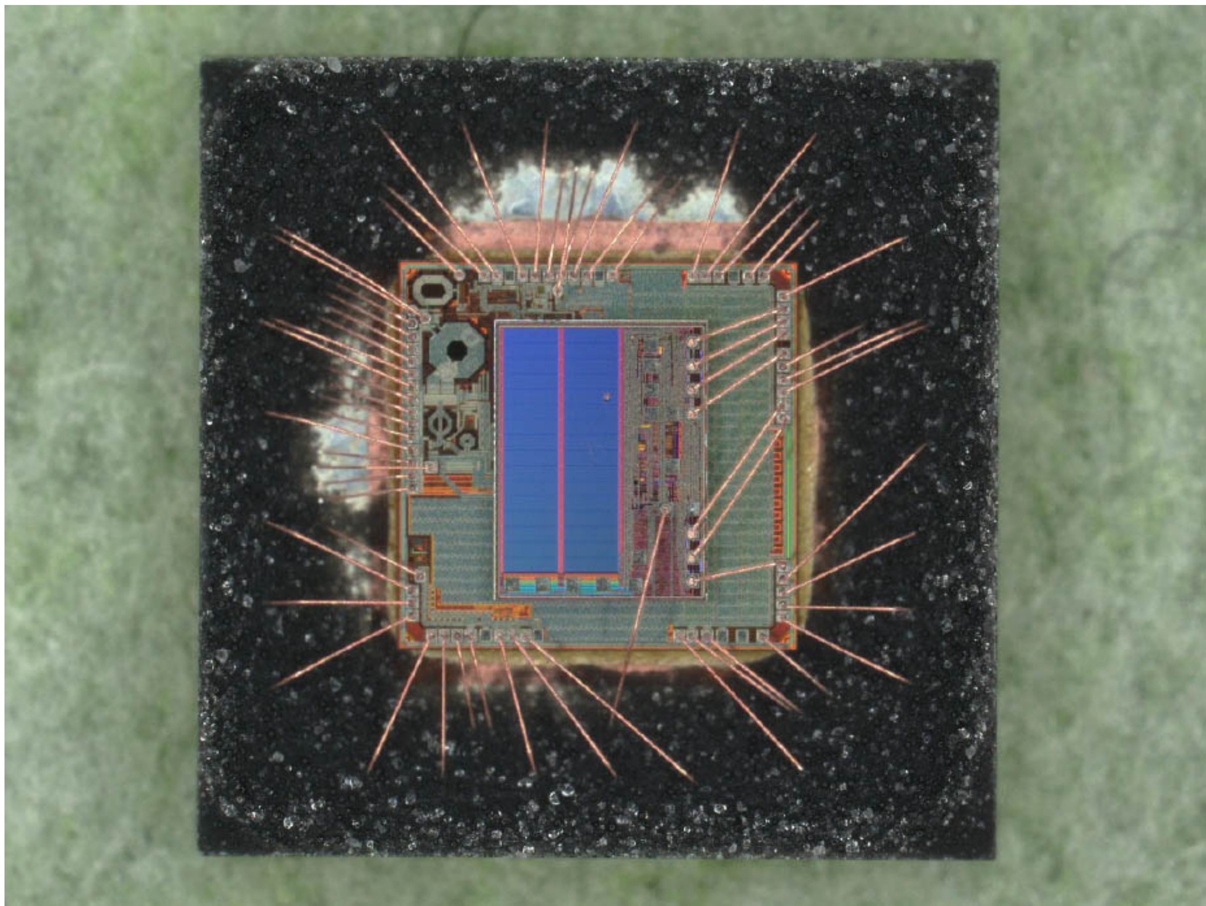

Abb. 4–2 *Fotografie des Silizium-Die des Mikrocontrollers ESP32-C2, Quelle: The ESP Journal [59]*

Abb. 4–2 zeigt die vergrößerte Aufnahme eines ESP32-C2-Mikrocontroller-Chips. Auf den Die mit dem Mikrocontroller ist der Flash-Speicherchip obenauf gestapelt. Die einzelnen funktionalen Einheiten sind optisch zu erkennen,

gleichförmig gemusterte Areale sind Speicher. Über die »Bond-Drähte« sind die Chips miteinander und mit den Kontakten des Gehäuses verbunden.

4.1.1 Test des Zufallszahlengenerators

In diesem Kapitel wird der Zugriff über den Bus auf den Speicher sowie auf den eingebauten Zufallszahlengenerator anhand eines Beispiels gezeigt. Mittels χ^2-Test lässt sich die Verteilung der Zufallszahlen prüfen.

Der χ^2-Test (Chi-Quadrat-Test) wird oft verwendet, um vorliegende Daten auf eine bestimmte Verteilung zu prüfen. So kann man beispielsweise testen, ob die produzierten Werkstücke in einem Fertigungsprozess normalverteilt sind, was oft ein Qualitätskriterium für einen guten Prozess ist. Abb. 4–3 zeigt einen solchen exemplarischen Fertigungsprozess mit vertikal eingetragenen Stückzahlen und horizontal eingetragenen Abweichungen vom Zielprodukt.

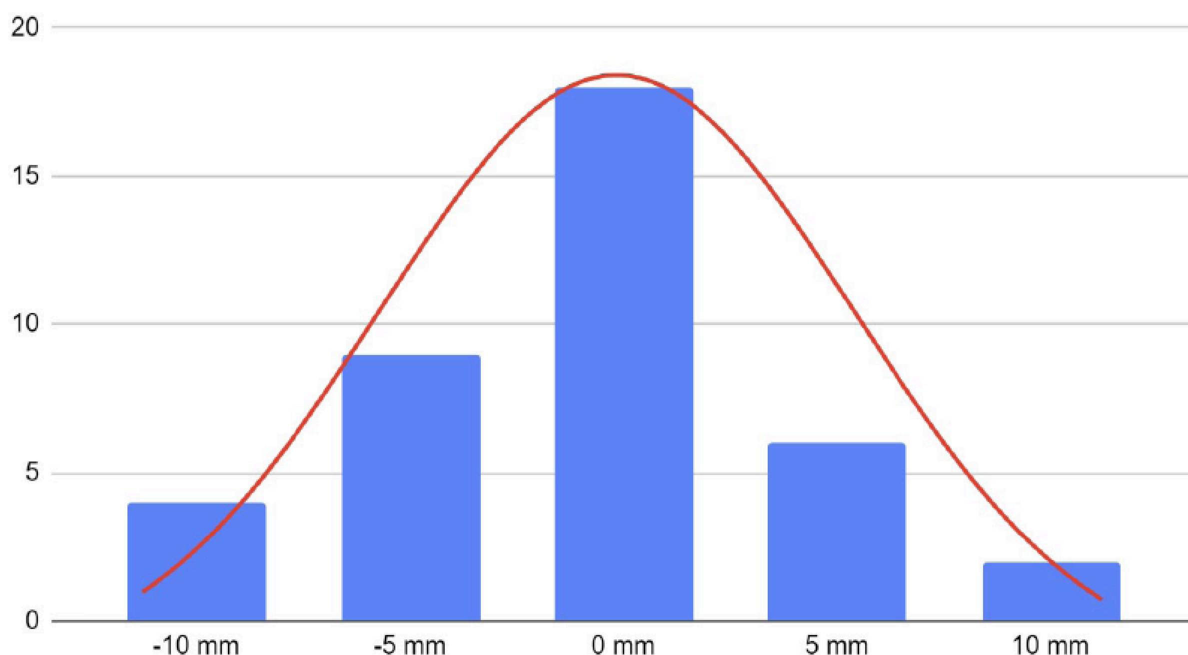

Abb. 4–3 *Normalverteilung von Werkstücken in einem Fertigungsprozess*

Ohne beim Chi-Quadrat(χ^2)-Test zu sehr ins Detail zu gehen, soll dessen Vorgehensweise kurz skizziert werden. Das zu untersuchende statistische Merkmal X wird in m Kategorien eingeteilt, das heißt, es wird ein Histogramm mit den Häufigkeiten N_j erstellt. Für jede Kategorie wird die erwartete Häufigkeit n_{0j} berechnet und anschließend die Prüfgröße X^2 als Größe der Abweichung wie folgt ermittelt:

$$X^2 = \sum_{i=1}^{m} \frac{(N_i - n_{0i})^2}{n_{0i}}$$

Die »Nullhypothese« H_0, die besagt, dass X eine zu überprüfende Verteilung aufweist, wird bei einem Signifikanzniveau α abgelehnt, wenn $X^2 > \chi^2_{(1-\alpha;m-1)}$. Die χ^2-Quantile mit entsprechenden Freiheitsgraden $m - 1$ werden üblicherweise einer Tabelle entnommen [64].

Im Beispielprojekt dieses Kapitels soll geprüft werden, ob ermittelte Zufallszahlen gleichverteilt sind und damit in etwa gleich oft vorkommen. Vor dem Zugriff auf den eingebauten Zufallszahlengenerator, um ihn zu prüfen, wird ein realer Spielewürfel oft geworfen und die Ergebnisse werden notiert. Abb. 4–4 zeigt das Ergebnis von 30 durchgeführten Würfen.

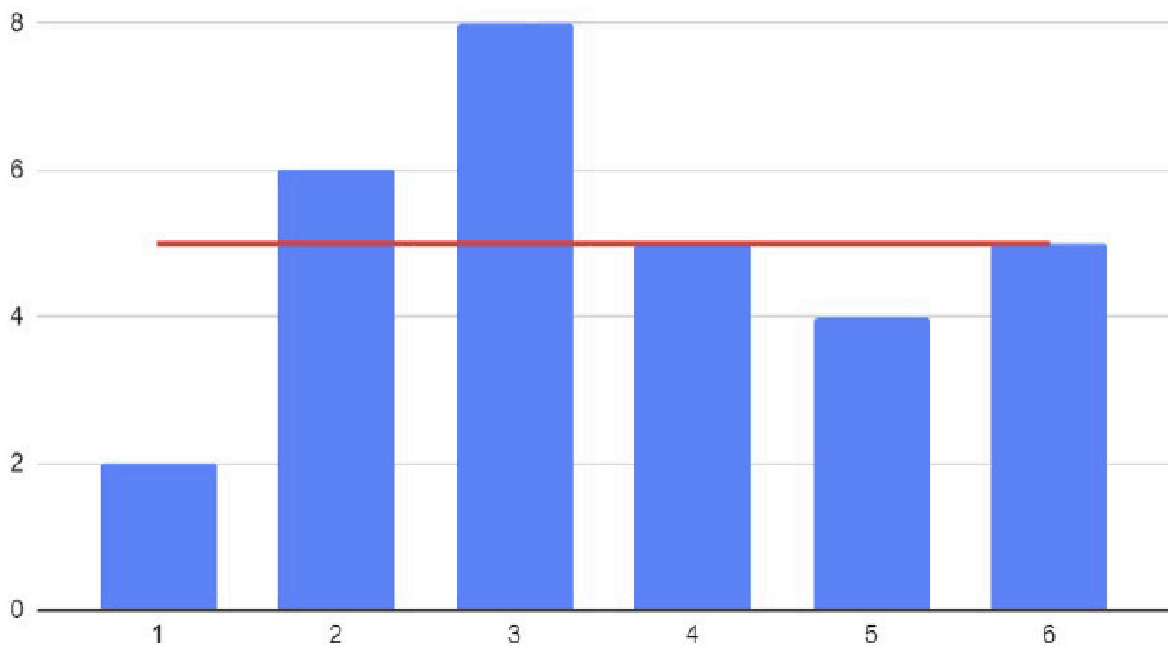

Abb. 4–4 *30 Würfe eines Würfels sollen auf Gleichverteilung geprüft werden.*

In Listing 4.1, das den χ^2-Test mit $\alpha = 10\%$ implementiert, sind die Würfe im Array `N` eingetragen. Bei der Implementierung wurden die Variablen gemäß der Formel benannt, allerdings als Funktionsparameter C-typisch in Kleinbuchstaben. Includes sind nicht und der Funktionsaufruf ist nur schematisch wiedergegeben.

```
#define M               6

#define OBSERVATIONS    30
```

```c
#define SQUARE(x)           ((x) * (x))

static const uint32_t N[M] = { 2, 6, 8, 5, 4, 5 };

bool equalDistChi2(const uint32_t n[], uint32_t m, uint32_t n0,
        uint32_t chi2) {

    uint32_t squaresum = 0;

    for (int i = 0; i < m; i += 1) {

        squaresum += SQUARE(n[i] - n0);

    }

    uint32_t x2 = squaresum / n0;

    return (x2 <= chi2);

}

bool ok = equalDistChi2(N, M, (OBSERVATIONS / M), 9); // Aufruf
```

***Listing 4.1** Implementierung des χ^2-Tests für die Prüfung der Gleichverteilung*

Der Code ist sehr direkt umgesetzt und bedarf nur wenig Erklärung. Einerseits erfolgt die Division durch n_{0i} nicht in jedem Schleifendurchgang, da bei der Gleichverteilung alle erwarteten Häufigkeiten gleich n_0 sind und die Division aus der Schleife gehoben werden kann, was den Berechnungsaufwand reduziert:

$$\forall_i : n_{0i} = n_0 \Rightarrow X^2 = \sum_{i=1}^{m} \frac{(N_i - n_0)^2}{n_0} = \frac{\sum_{i=1}^{m}(N_i - n_0)^2}{n_0}$$

Andererseits erfolgen die Berechnungen zur Steigerung der Performanz ganzzahlig. Dies bedeutet zwar eine geringere Genauigkeit, spielt aber für den Einsatzzweck eine untergeordnete Rolle. Das damit neu eingeführte Problem eines

Überlaufs beim Quadrieren oder der Summation sollte sich nicht auswirken, da die Werte im Histogramm diese Größenordnung nicht erreichen.

Die Performanz und Einfachheit des χ^2-Tests ist der hauptsächliche Grund seines Einsatzes. Er sollte aber nur verwendet werden, wenn genügend Daten zur Analyse zur Verfügung stehen. Konkret sollte jeder Eintrag im Histogramm mindestens den Wert 5 haben. Für die Prüfung von schwächer besetzten Datenreihen gibt es in der Statistik ausgereiftere, aber aufwendigere Tests.

In Abschnitt 3.1 wurden hauptsächlich die 32 Register der RISC-V CPU und die internen CSRs verwendet. Der Befehlsspeicher und der Datenspeicher waren Komponenten, die über Adressen gelesen und beschrieben werden konnten, aber nicht weiter betrachtet wurden. Diese Sicht soll hier anhand des Feldzugriffs weiter detailliert werden.

Der Hauptunterschied dieses Beispiels zu dem Summationsprogramm in Kapitel 3 ist die Verwendung des Arrays `N`, das als globale Variable im Datensegment untergebracht wird. Der Zugriff erfolgt über den Assemblerbefehl `lw` mit der berechneten Adresse n+i·4. Die Adresse des ersten Array-Eintrags ist `n` und hat den Wert 0x3c022990. Das C-Statement `*(n + i)` entspricht daher in Pointer-Arithmetik dem Array-Zugriff `n[i]`.

Der Speicher, auf den zugegriffen wird, liegt außerhalb der CPU. Um dort Daten zu lesen oder zu schreiben, werden sie über das Bussystem mit ihrer Adresse angesprochen.

4.1.2 Das Bussystem

Über den Steuerbus wird die Kommunikation gesteuert.

Ein Bus dient dazu, zwischen mehreren Teilnehmern, die direkt über dasselbe Übertragungsmedium physisch verbunden sind, Daten auszutauschen. Abb. 4–5 zeigt ein Bussystem, also eine Verknüpfung mehrerer Busse, aus Daten-, Adress- und Steuerbus. In der Folge werden Bus und Bussystem austauschbar verwendet. Die Busse im System sind parallel ausgeführt. Bei diesen Bussen werden die einzelnen Bits auf jeweils eigenen Leitungen parallel, und damit gleichzeitig, übertragen. An einen Bus sind im einfachsten Fall ein Master und mehrere Slaves angeschlossen. Slaves sind Speicher, I/O-Module, Kommunikationsmodule und weitere.

Die Steuerung der Kommunikation erfolgt über einzelne Signalleitungen auf dem Steuerbus. Der Master bestimmt über Signale wie »Read« und »Write«, ob die Slaves auf die Anfrage Daten bereitstellen oder entgegennehmen sollen. Taktsignale dienen der Synchronisation der Teilnehmer. Die Übernahme der Adressen und Daten geschieht beispielsweise bei einem definierten Übergang des

Taktsignals, also synchron zu einer Flanke des Takts. Master und Slaves haben auch die Möglichkeit, Bestätigungen (ACK, acknowledgement), priorisierte Anfragen (IRQ, interrupt requests) und Weiteres zu signalisieren. Zudem wird auch die Arbitrierung, also die Zugriffsabfolge beim Vorhandensein mehrerer Master über Signale auf dem Steuerbus durchgeführt. In vielen, aber nicht allen Bussystemen gibt es auch die Möglichkeit, die Blockgröße des Transfers festzulegen.

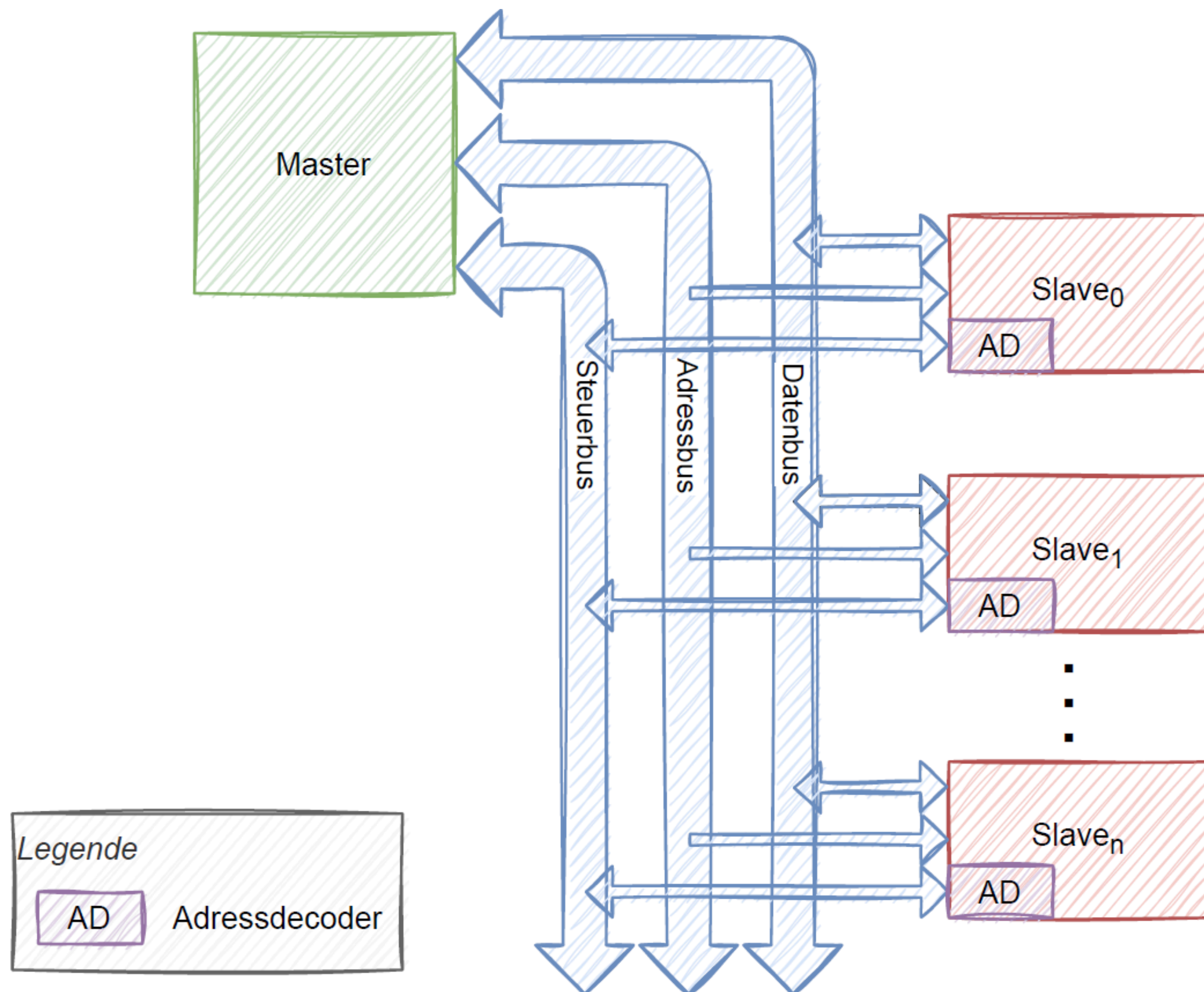

Abb. 4–5 *Schema eines parallelen Bussystems*

Auf dem Datenbus werden die Nutzdaten vom Master zu den Slaves und von den Slaves zum Master übertragen. In einer 32 Bit breiten Architektur werden üblicherweise Vielfache von 32 Bit übertragen, um ein oder mehrere Wörter in einem einzigen Transfer zu übermitteln.

Der Master legt für einen Zugriff die Adresse einer Speicherstelle, die er ansprechen möchte, auf den Adressbus. Üblicherweise kann jedes Byte einzeln adressiert werden, was bei 32 Bit einen Adressraum von 2^{32} B = 4 GiB ergibt. Den angeschlossenen Slaves werden jeweils Teilbereiche des Adressraums zugeordnet.

Per Adressdecoder stellt ein Slave fest, ob der adressierte Speicher in seinem privaten Adressraum liegt.

Eine »Memory Map«, wie diese beispielhaft in Tab. 4–1 gezeigt ist, dient dazu, die Adressbereiche der Busteilnehmer aufzulisten. Legt der Master beispielsweise die Adresse 0x00020020 auf den Adressbus, ist $Slave_1$ Ziel des Transfers. Der Adressdekoder (AD) dieses Slaves subtrahiert nun seine Basisadresse 0x00020000 und greift in seinem privaten Adressraum auf den Offset 0x20 zu, während die anderen Slaves den Bustransfer ignorieren.

Tab. 4–1 *Beispielhafte Memory Map mit drei Busteilnehmern*

Modul	**Adressbereich**
$Slave_0$	`0x00010000 – 0x0001FFFF`
$Slave_1$	`0x00020000 – 0x00020FFF`
$Slave_2$	`0x00040300 – 0x000403FF`

Im Adressraum muss nicht zwingend jede Adresse gültig sein, wie auch in diesem Beispiel nur ein kleiner Bereich des Adressraums »belegt« ist. Wird auf einen nicht zugeordneten Bereich zugegriffen, bleibt die Antwort eines Slaves aus. Der Master signalisiert in diesem Fall einen »Bus Fault«.

Der Durchsatz des Busses hängt wesentlich vom Takt ab, mit dem die Daten übertragen werden. Dieser Bustakt lässt sich per Software beeinflussen. Die an den Bus angeschlossenen Module (Slaves), die einen eigenen Takt benötigen, können ihren Takt vom Bus entnehmen. Um ein solches Modul in den Stromsparmodus zu versetzen, kann der Takt des gesamten Busses oder betreffender Module beim sogenannten »Clock Gating« abgeschaltet werden.

Nach dem Systemstart sind bei Mikrocontrollern viele Module in diesem Stromsparmodus, um den Stromverbrauch vom Start weg niedrig zu halten. Um ein solches Modul zu verwenden, muss der Takt programmtechnisch über das Power-Management eingeschaltet werden. Der Bustakt `RTC20M_CLK` wird exemplarisch in Abschnitt 4.3.5 eingeschaltet, um Rauschen für den Zufallszahlengenerator bereitzustellen.

Manche Busse, wie der RISC-V IBUS zum Zugriff auf Befehle, unterstützen nur aligned access (siehe Abschnitt 3.1.4). Andere Busse, wie der RISC-V DBUS zum Zugriff auf Daten, bieten des einfacheren CPU-Designs halber auch »misaligned access« an.

Busse mit mehreren Mastern (»Multimaster«) sind auch möglich, allerdings sind diese komplexer und damit aufwendiger. In Abb. 4–1 ist ein solcher Multimaster-Bus als Hochgeschwindigkeits-Bus-Matrix eingezeichnet. An dieser Bus-Matrix (auch »Bridge«) sind dann einzelne Peripheriebusse mit niedrigerer Geschwindigkeit, ein Mikroprozessor und ein DMA-Controller angeschlossen.

4.1.3 ESP32-C3 Memory Map

Das Bussystem sowie der 4 GiB große Adressraum sind für alle Mikrocontroller der ESP32-C3-Familie gleich aufgebaut. Abb. 4–6 zeigt den schematischen Aufbau. Der Mikroprozessor RV32IMC greift im Instruction Memory (siehe Abschnitt 3.1.5) über den IBUS (Instruction Bus) 4-Byte aligned lesend auf den IRAM-Speicherbereich (blau markiert) oder die Peripherie (orange) zu. Der Zugriff im Data Memory (siehe Abschnitt 3.1.4) erfolgt 1-/2-/4-Byte aligned über den DBUS (Data Bus) auf den DRAM-Speicherbereich oder die Peripherie.

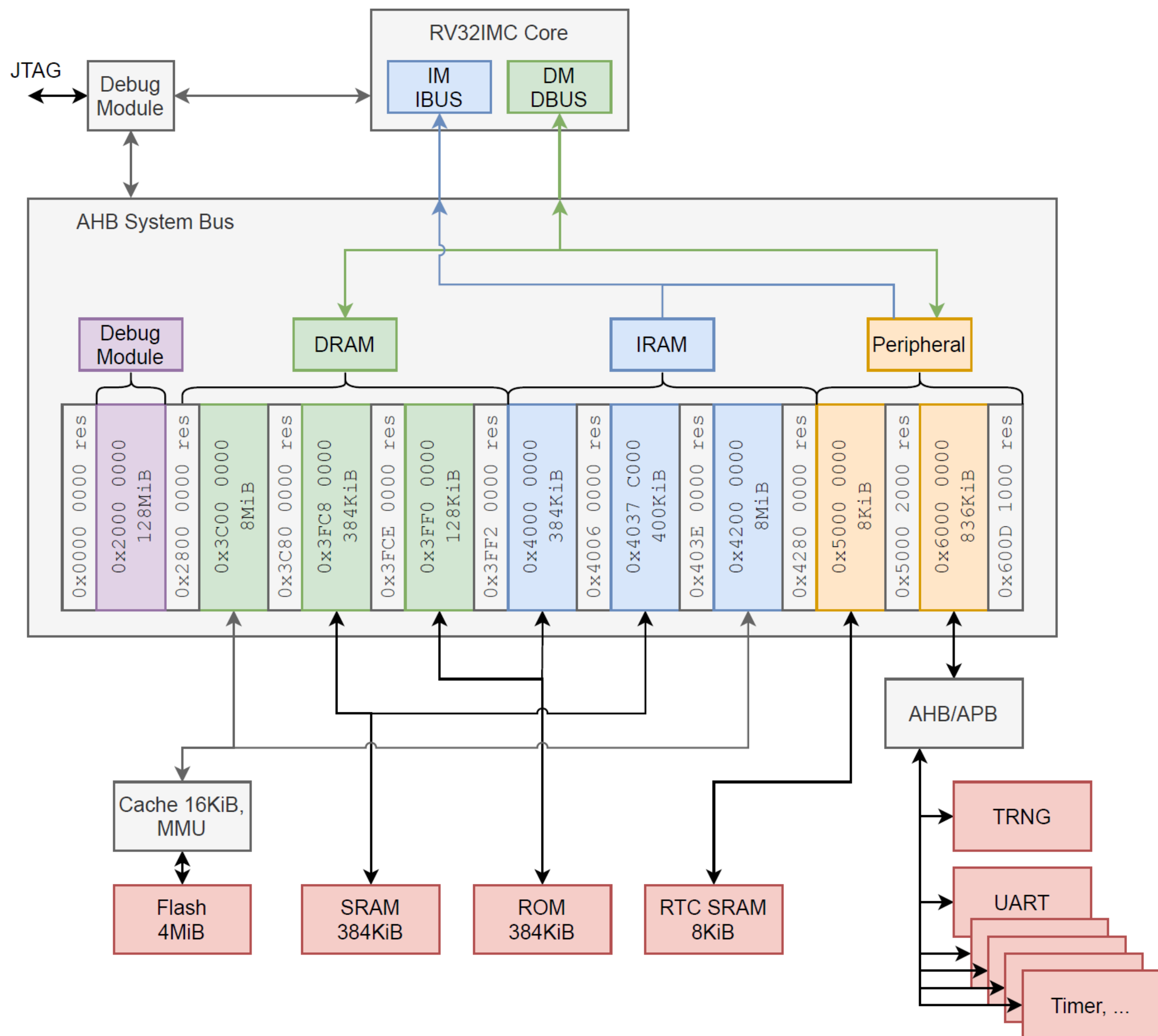

Abb. 4–6 *Schematischer Aufbau des ESP32-C3-Adress- und -Bussystems*

Im Speicherbereich der Peripherie liegt einerseits der Speicher der RTC (»Real Time Clock«). Diese 8 KiB RAM werden noch im Tiefschlaf mit Strom versorgt, sodass der Inhalt dieses Speichers im Gegensatz zum restlichen RAM über diese Phase und einen Reset hinaus erhalten bleibt. Beim Aufwachen muss bei einem System mit tiefstem Schlaf der Systemstatus dann mit Hilfe dieses Speichers mit entsprechendem zeitlichen Aufwand wiederhergestellt werden. Weitere Peripherie wie Kommunikationsschnittstellen, kryptografische Coprozessoren, Timer usw. sind im Speicherbereich der Peripherie untergebracht (siehe Abschnitt 4.3).

Als Bussystem kommt der schnelle AHB (Advanced High-Performance Bus) zum Einsatz. Die langsamere Peripherie ist über den einfacheren APB (Advanced Peripheral Bus), der seinerseits über eine Bridge mit dem AHB verbunden ist, angeschlossen. Diese Busse sind in dem offenen AMBA (Advanced Microcontroller Bus Architecture) Standard [3] für die Verbindung innerhalb von Chips definiert und können von den Herstellern lizenzfrei verwendet werden. Die zuverlässigen AMBA-Busse, zu denen auch noch AXI (Advanced Extensible Interface) und ATP

(Advanced Trace Bus) gehören, erfreuen sich deshalb großer Beliebtheit und sind weit verbreitet.

Das Debug Module in der Abbildung wird über JTAG angesprochen. Es kommuniziert einerseits mit dem Mikroprozessor und kann andererseits direkt auf den Bus zugreifen. Auf diese Weise ist es möglich, beim Debuggen direkt auf die Register, Speicher und Peripheriemodule lesend und schreibend zuzugreifen.

4.2 Speicher

Im System sind verschiedene Arten von Speicher untergebracht, die jeweils ihre Eigenheiten, Vor- und Nachteile aufweisen. Für Embedded-Software-Entwickler:innen ist es wichtig, den Speicher korrekt einzusetzen, um ein effizientes und langlebiges System zu kreieren.

4.2.1 Speichertechnologien

Die Hauptunterscheidung in der Namensgebung ist zwischen RAM (»Random Access Memory«, Speicher mit wahlfreiem Zugriff) und ROM (»Read Only Memory«, Nur-Lese-Speicher), wobei diese Bezeichnungen leicht irreführend sind. Der wahlfreie Zugriff bezieht sich bei RAM oft auf ganze Wörter, es sind also nicht immer einzelne Bytes ansprechbar, weshalb RAM auch schon in RWRAM (»Read-Write RAM«) umbenannt werden sollte. Bei PCs wird auch der Arbeitsspeicher (Hauptspeicher) als RAM bezeichnet, weil hierfür hauptsächlich DRAM zum Einsatz kommt. Im Kontext von Embedded Systemen ist mit RAM die eingesetzte Technologie und mit Hauptspeicher der zur Verfügung stehende Hauptspeicher gemeint. ROMs auf der anderen Seite sind als PROM (»Programmable ROM«) auch ein Mal schreibbar.

Eine weitere Unterscheidung der Technologien liegt in deren Datenerhaltung nach Abschaltung der Stromzufuhr. Volatile Technologien verlieren den Inhalt, wohingegen persistente (beziehungsweise non-volatile) Technologien diesen behalten. Im Folgenden werden gebräuchliche Speichertechnologien, die auch in Tabelle 4–2 zusammengefasst sind, besprochen.

SRAM SRAM (»Static RAM«) ist ein volatiler Speicher, der seinen Zustand in einer bistabilen Kippstufe («Flipflop«) je Bit speichert. Die gängige Implementierung erfordert sechs Transistoren pro Bit, was im Verhältnis zu anderen Speichertechnologien sehr viel Siliziumfläche einnimmt und damit teuer ist. Vorteile von SRAM sind die hohe Geschwindigkeit und Stabilität sowie ein niedriger Stromverbrauch für die Datenerhaltung. Haupteinsatzgebiete sind als Arbeitsspeicher in Embedded Systemen, für den Aufbau von Registern in

Prozessoren und Peripheriemodulen sowie als Caches (siehe Abschnitt 4.2.3). Die hohe Geschwindigkeit steht im Vordergrund beim Einsatz in CPU-Speicher und Caches, da der Zugriff innerhalb eines Taktes erfolgen kann. Der ESP32-C3 bringt 400 KiB SRAM als Arbeitsspeicher mit, von denen 16 KiB als Cache für den Flash-Speicher verwendet werden können. Die Echtzeituhr (RTC) bietet noch zusätzlich 8 KiB SRAM, die ebenso als Arbeitsspeicher verwendet werden können.

DRAM Dynamischer RAM ist ein volatiler Speicher, der die Datenbits in einzelnen Kondensatoren (siehe Abschnitt 5.4.11) speichert. Der große Vorteil dieser Technologie gegenüber SRAM ist die kleinere Siliziumfläche, die ein Speicherbit einnimmt. Für die Implementierung einer Speicherzelle genügen ein Transistor und ein Kondensator.

Ein Bit wird als Ladung eines Kondensators gespeichert. Diese Ladung kann sich über Leckströme ändern, wodurch der Speicherinhalt mit fortlaufender Zeit verloren geht. Um dem entgegenzuwirken, muss der Speicher zyklisch ausgelesen und neu beschrieben werden. Diesen etwa alle 40 ms durchzuführenden Refresh führt ein eigener DRAM-Controller durch. Nachteile dieser Technologie gegenüber SRAM sind der erhöhte Stromverbrauch und die niedrigere Geschwindigkeit, die sich einerseits aus dem Zugriff über den DRAM-Controller, andererseits aus der Optimierung auf Packungsdichte statt Geschwindigkeit ergibt.

DRAM findet vor allem Anwendung, wenn große Arbeitsspeicher im Bereich mehrerer GiB wie bei PCs und Smartphones benötigt werden. Der hier eingesetzte SDRAM (»Synchronous DRAM«) wird durch den Bus getaktet und wird als DDR-SDRAM in Steckmodulbauweise hergestellt. Für den mobilen Einsatz gibt es die stromsparende »Low-Power«-Variante LP-SDRAM.

Tab. 4–2 *Verbreitete Speichertechnologien im Vergleich*

Technologie	Persistenz	Vorteile	Nachteile
SRAM	volatil	schnell, stromsparend	teuer
DRAM	volatil	günstig	langsam, stromhungrig
Masken-ROM	non-volatil	am günstigsten, schnell	Änderungen unmöglich
PROM	non-volatil	schnell lesbar	einmalig beschreibbar, teuer
EPROM	non-volatil	löschbar	Programmer und Löschgerät nötig
EEPROM	non-volatil	vielfach schreibbar	begrenzte Schreib-/Löschzyklen, langsam
FRAM	non-volatil	schnell, RAM-Ersatz, lange haltbar	sehr teuer
ReRAM	non-volatil	stromsparend	begrenzte Schreibzyklen

Eine weitere Variante, PSRAM (»Pseudostatisches RAM«), beinhaltet den Refresh Controller und lässt sich wie SRAM ansteuern. Damit kann ein SRAM-Hardwarebaustein einfach durch einen günstigeren PSRAM-Baustein ersetzt werden. Im ESP32-C3-MINI-Modul ist kein DRAM verbaut.

Masken-ROM Beim Masken-ROM werden die einzelnen Bits bereits bei der Herstellung »fest verdrahtet«, also in der Belichtungsmaske untergebracht. Ein Bit wird hier prinzipiell durch Vorhandensein (»1«) oder Fehlen (»0«) einer Diode in einer Zeilen-/Spaltenmatrix kodiert. Dieser Speicher ist der günstigste in der Massenherstellung, sein Inhalt kann jedoch nicht mehr geändert werden. Die Hersteller bringen im ROM Basisfunktionalitäten wie einen Bootloader oder Bibliotheken unter. Dabei wird darauf geachtet, dass Patches über anderen persistenten Speicher wie Flash aufgebracht werden können, um im Falle eines Fehlers nicht die gesamte Produktionscharge unbrauchbar zu machen. Auf den Mikrocontrollern der ESP32-C3-Familie sind 128 KiB ROM mit Bootloader und Kernfunktionen untergebracht.

PROM Ein PROM (»Programmable ROM«) hat denselben Grundaufbau wie ein Masken-ROM. Im Unterschied dazu werden alle Dioden oder alternativ Transistoren bestückt und bei jeder Diode wird eine Leitung als Schwachstelle

ausgeführt. Die Daten des Speichers bestehen somit aus gesetzten Bits. Eine Schwachstelle kann beim Programmieren mit einem erhöhten Strom durchgebrannt werden, womit aus einer »1« irreversibel eine »0« wird.

Das PROM in dieser Form findet nur noch wenig Verwendung. Manchmal wird es noch als »Sicherung« eingesetzt. Soll beispielsweise bei einem Embedded System die Programmierung, JTAG Debugging, Auslesen des Speichers usw. abgeschaltet werden, werden entsprechende Sicherungen durchgebrannt. Eine Alternative zu PROM für diesen Zweck ist der Einsatz von EPROM- oder EEPROM-Speicher ohne Löschfunktion. Die 512 Byte große eFuse des ESP32C3 ist ein solcher »OTP«-Speicher, also »One-Time Programmable«.

Die Verwendung der eFuse ist auf der Espressif Webseite [12] beschrieben.

EPROM Wird der Gedanke des PROM weitergeführt, sodass der Speicher nicht nur ein Mal beschrieben, sondern auch wieder gelöscht werden kann, resultiert dies in einem EPROM (»Erasable PROM«). Wie im Schema in Abb. 4–8 wird unter dem Gate eines Feldeffekt-Transistors (FET, siehe Abschnitt 5.4.1) ein zweites Gate in einem Isolator untergebracht. Ist dieses »Floating Gate« mit Elektronen geladen, sperrt der Transistor. Im anderen Fall ist der Transistor leitend.

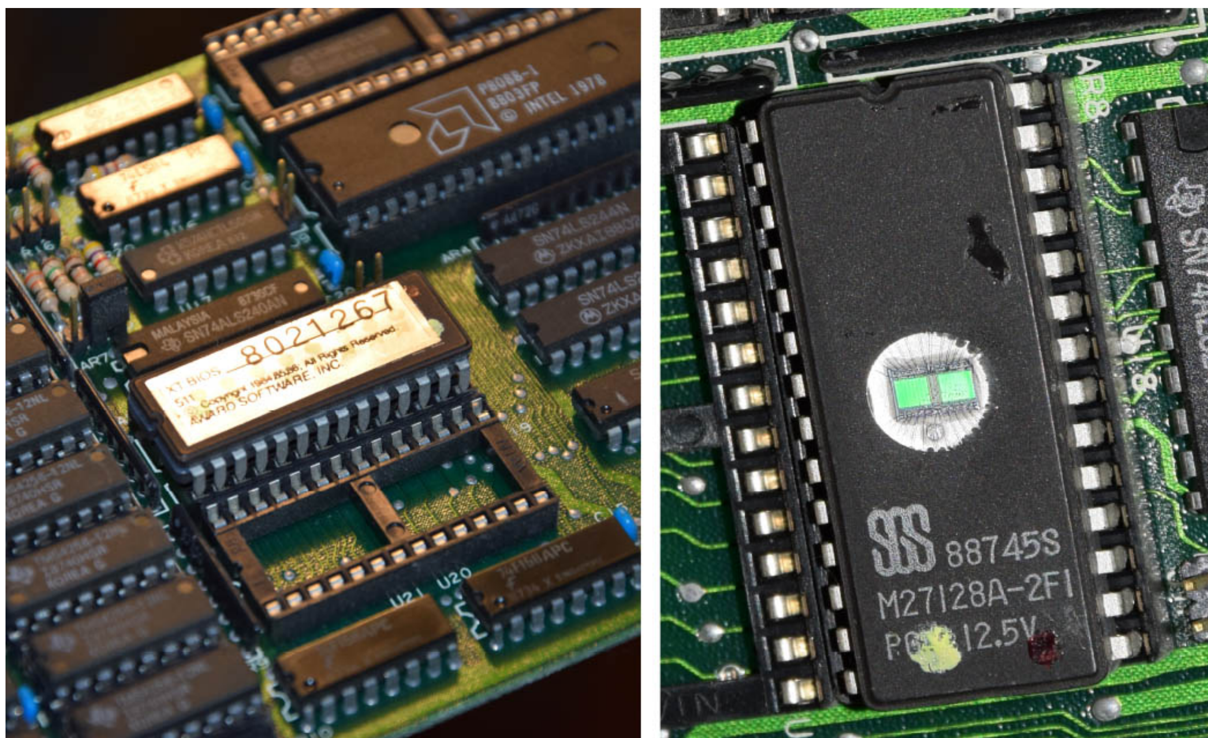

***Abb. 4–7** EPROM mit dem BIOS eines PC-XT-Nachbaus links mit Schutzfolie, rechts mit freigelegtem Fenster (Quelle: eigenes Foto)*

Wird das EPROM für etwa 20 Minuten UV-Licht ausgesetzt, ionisiert die Strahlung die Isolierschicht durch ein Fenster im Chipgehäuse, worauf die Elektronen vom Floating Gate entweichen. Das EPROM wird somit zur Gänze

gelöscht. Zur Programmierung wird eine hohe Spannung (bis zu 25 V) an Gate und Drain angelegt, im resultierenden Lawinendurchbruch überwinden Elektronen die dünne Isolierschicht und sammeln sich am Floating Gate. Dieser Vorgang dauert im Bereich einer Millisekunde.

Das Löschen erfolgt in einem eigenen Löschgerät, einer UV-Leuchte, die wegen der schädlichen Strahlung in einer Box untergebracht ist. Die Programmierung erfolgt in einem eigenen EPROM-Programmer, der die nötige hohe Spannung anlegen kann. Es kann davon ausgegangen werden, dass ein programmiertes EPROM den Inhalt mindestens 10 Jahre zuverlässig behalten kann. Da aber Programmieren und Löschen sehr umständlich sind, wurde diese Technologie nahezu vollständig von der folgend beschriebenen EEPROM-Technologie abgelöst.

Abb. 4–7 zeigt ein EPROM mit dem BIOS eines PC XT aus dem Jahr 1986. Die Alufolie schützt den Baustein im linken Bild vor Licht. Im rechten Bild wurde die Alufolie entfernt und das Fenster freigelegt.

1 Angström (1 Å) entspricht 0,1 nm.

EEPROM/Flash Eine Erweiterung des EPROM ist das »Electrically Erasable« PROM, das elektrisch gelöscht und beschrieben werden kann (siehe Abb. 4–8). Das Floating Gate ist mit einer Isolationsschicht von ungefähr 100 Å Dicke sehr nahe am Substrat. Dies macht es möglich, dass die Elektronen beim Löschen und Programmieren den Isolator über den quantenmechanischen Tunneleffekt passieren. Die angelegte Programmierspannung ist dabei niedriger als beim EPROM und kann üblicherweise auf dem Chip generiert werden.

Das Laden und Entladen des Floating Gate dauert verhältnismäßig lang, im Bereich mehrerer Millisekunden. Der Zustand wird dann je nach Hersteller zwischen 10 und 100 Jahren gehalten. Zu beachten ist hierbei, dass eine höhere Temperatur einen negativen Einfluss auf die Haltbarkeit hat. Des Weiteren wirkt das Tunneln der Elektronen zerstörerisch auf den Isolator. Jeder Schreib-/Löschzyklus hinterlässt seine Spuren und erniedrigt die Datenerhaltungszeit. Die maximale Anzahl an garantiert möglichen Zyklen liegt im Bereich 10.000 bis 1.000.000. Das Auslesen ist sehr schnell und hat nur einen geringen Einfluss auf die gespeicherte Ladung.

Flash-Speicher entspricht vom Aufbau her dem EEPROM mit dem Unterschied, dass hier nicht einzelne Bytes geschrieben werden, sondern größere Bereiche (»Pages«) gleichzeitig. Da sie auch das Problem der Abnutzung (»Wearout«) haben, werden in eigenen Controllern »Wear Leveling«-Algorithmen eingesetzt. Wird derselbe Block adressiert und geschrieben, nimmt der Controller einen unterschiedlichen physischen Block, um beim Schreiben den gesamten Speicher möglichst gleichmäßig abzunutzen.

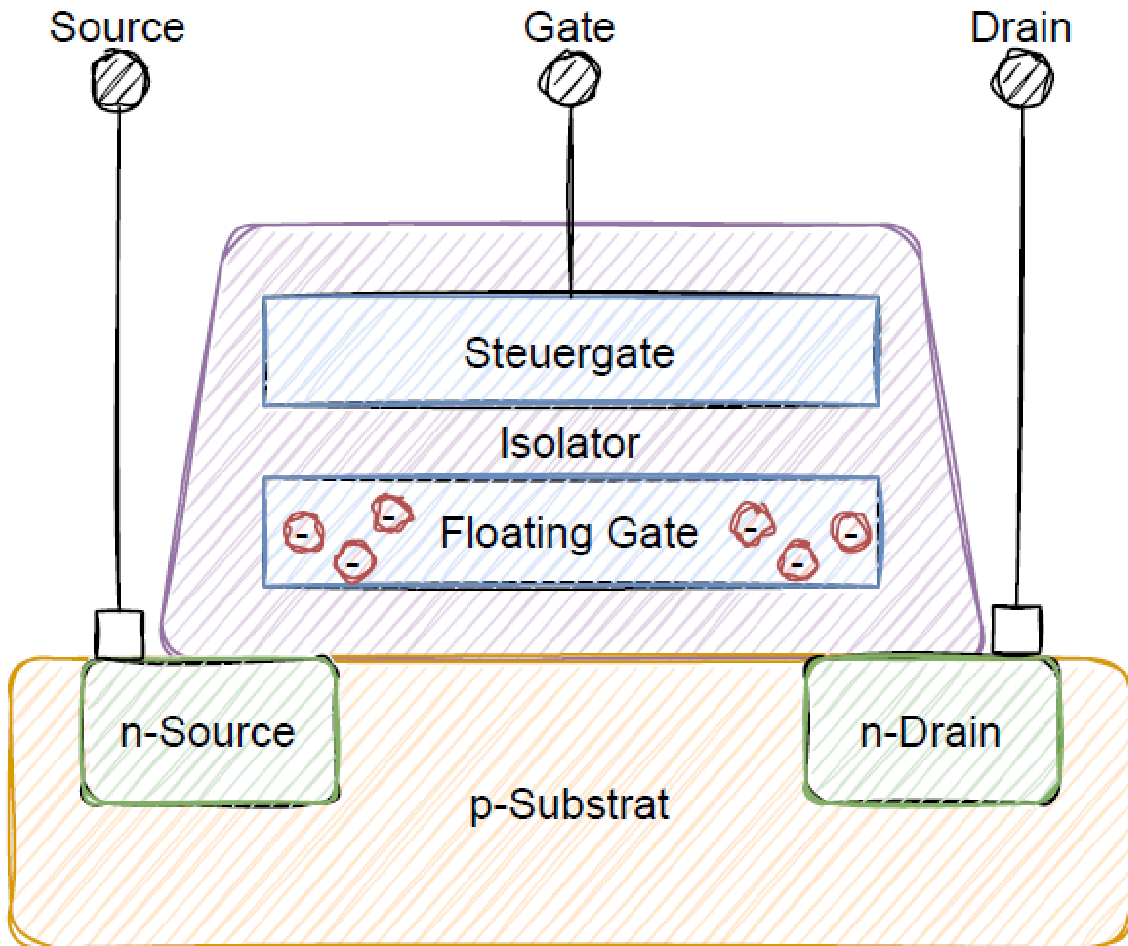

Abb. 4–8 *Schematischer Aufbau einer EPROM- oder EEPROM-Zelle*

Flash beziehungsweise EEPROM wird in modernen Systemen häufig zur persistenten Datenhaltung eingesetzt: SSDs, das sind auf Flash basierte Laufwerke, haben die mechanischen Festplatten weitgehend ersetzt. Mobile Geräte wie Fotoapparate, Smartphones usw. verwenden SD-Karten mit der Flash-Technologie.

Der ESP32C3FN4-Controller besitzt einen über ein SPI-Interface (siehe Abschnitt 7.4) verbundenen externen Flash-Speicher mit 4 MiB Größe. Auf ihm werden Programme, konstante Daten und optional ein Filesystem gespeichert. Der lesende Zugriff erfolgt über den Bus durch einen normalen Lesezugriff. Der schreibende Zugriff kann üblicherweise nicht so einfach durchgeführt werden. Ein separater Flash-Controller oder spezieller Zugriff mit entsprechend einzuhaltendem Timing muss durchgeführt werden. Bei vielen Systemen ist es auch nicht möglich, während eines schreibenden Zugriffs auf den Flash-Programmcode, der im selben Flash gespeichert ist, auszuführen. Routinen, die während des Speicherns ausgeführt werden sollen, werden deshalb im RAM (oder auf einer unabhängigen »Bank« des Flashs) untergebracht.

FRAM Bei dieser persistenten Speichertechnologie werden einzelne Bits durch die Lage von Atomen in einer Kristallstruktur gespeichert. Die aus der

Atombewegung resultierende elektrische Polarisierung kann gemessen werden.

FRAM bietet gegenüber EEPROM enorme Vorteile. Ein Schreibzugriff liegt mit 150 ns im Bereich von langsamem RAM, und mit etwa 10^{12} Lese-/Schreibzyklen ist der Wearout praktisch vernachlässigbar, weshalb FRAM auch als Hauptspeicher eingesetzt werden kann. Der Speicher ist eigentlich recht unempfindlich, hohe Temperaturen zerstören aber die gespeicherten Daten. Da FRAM nicht die Speicherdichte von Flash aufweist und dadurch verhältnismäßig teuer ist, ist FRAM derzeit wenig verbreitet.

Die MSP430FRxxx-Mikrocontroller von Texas Instruments beinhalten bis zu 256 KiB FRAM (»Ferroelectric RAM«). Bausteine mit 512 KiB FRAM sind bei Fujitsu mit verschiedenen Schnittstellen ausgestattet erhältlich.

ReRAM Das persistente RRAM/ReRAM (»Resistive RAM«) speichert die Daten in der programmierbaren Änderung des elektrischen Widerstands eines leitfähigen Dielektrikums. Diese Technologie, deren größter Vorteil ein sehr niedriger Stromverbrauch beim Auslesen ist, ist in Mikrocontrollern der Panasonic MN101L-Familie und als separate Speicher mit etwa 512 KiB erhältlich. Mit einer maximalen Zahl von 1.000.000 Schreibzyklen hat der Speicher einen deutlichen Wearout und ist wie EEPROM nicht als RAM-Arbeitsspeicher im System einsetzbar. Für eingebettete Applikationen mit hohen Ansprüchen an die Lebensdauer einer Batterie, wie Armbanduhren oder Sensoren in der Medizintechnik, ist ReRAM eine gute Speicheralternative.

Weitere Technologien Es existiert eine große Palette an weiteren Technologien, die keine wesentliche Bedeutung für Embedded Systeme mehr oder noch nicht erlangt haben. Mechanische Festplatten, die die Informationen auf sich drehenden magnetischen Scheiben speichern, wurden in Kleinstrechnern nie eingesetzt und verlieren auch in PCs und Servern an Bedeutung. CDROMs, DVDs und Blue-Ray Discs sowie Bandlaufwerke werden hauptsächlich für Backups eingesetzt.

Es befindet sich eine Reihe an persistenten Speichertechnologien in der (Weiter-)Entwicklung wie die erwähnten FRAM und RRAM, aber auch MRAM und PRAM. Sie nutzen unterschiedliche elektrophysikalische Effekte, um die Daten als Einzelbits (»SLC, Single Level Cell«) oder Multibits (»MLC, Multi Level Cell«) zu speichern. Die Skalierbarkeit und Massenproduktion von Speichern mit hoher Haltbarkeit in ansprechenden Größen und zu konkurrenzfähigen Preisen sind Gegenstand aktiver Weiterentwicklung.

4.2.2 Speicherzugriffe in Software

Auf die verschiedenen Speicher wird einerseits zugegriffen, um die auszuführenden Befehle zu laden, und andererseits, um dort Daten zu halten. In

der Software werden die Daten in Variablen abgebildet, die vom Compiler automatisch angelegt und zerstört werden. Dies ist bei globalen und lokalen Variablen sowie Funktionsparametern der Fall. Es besteht auch die Möglichkeit, den Speicher selbst zu verwalten. Dieser dynamische Speicher muss dann explizit angefordert und freigegeben werden.

Speicherlayout

In C-Programmen werden Elemente, die statisch Speicher benötigen, in einem Speicherlayout angeordnet. Grob unterschieden werden in der klassischen C-Sicht, wie Abb. 4–9 links zeigt, das Textsegment und das Datensegment. Im Textsegment oder auch Codesegment, das nur les- und nicht beschreibbar ist, werden der kompilierte Programmcode sowie Konstanten und Sprungtabellen untergebracht.

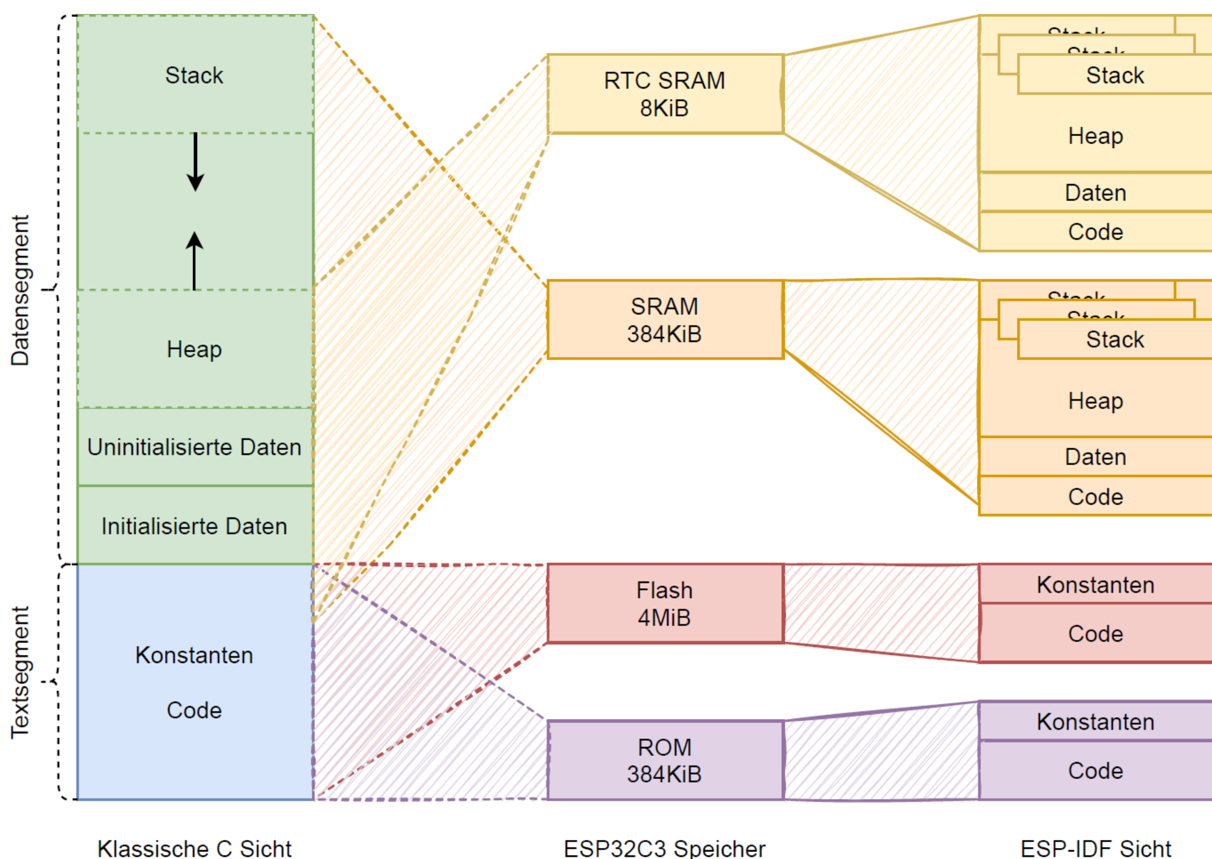

Abb. 4–9 *Segmente von Applikationen und Zuordnung zum Speicher*

Das Datensegment fasst einen uninitialisierten, einen initialisierte Bereich, den Heap und den Stack zusammen. Globale Variablen, die nicht oder mit 0 initialisiert werden, kommen in den uninitialisierten Bereich. Dieser Bereich, der aus historischen Gründen `BSS` (`Block Started by Symbol` war in den 1950er Jahren ein hierfür benutzter Pseudoassemblerbefehl) heißt, wird beim Programmstart mit dem Wert 0 belegt. In Embedded Systemen gibt es auch den

Bereich `noinit`, der tatsächlich uninitialisiert ist, um den Speicherinhalt über einen Systemreset hinweg zu erhalten (siehe Abschnitt 4.2.4).

Im initialisierten Bereich werden globale Variablen, die mit einem Wert ungleich 0 initialisiert werden, abgelegt. Die Initialisierungswerte werden beim Start aus dem Konstantenbereich kopiert. Aus diesem Grund benötigen initialisierte Variablen zusätzlich Platz im Konstantenbereich.

Der Stack, der von der höheren Adresse zur niederen wächst, wird ebenso im Datensegment angelegt. Auf dem Stack werden lokale Variablen, Funktionsparameter und Rücksprungadressen abgelegt, sofern diese nicht in Registern gespeichert sind.

Dem Stack entgegen wächst der »Heap« von der niederen zur höheren Adresse. Dieser Speicher wird als dynamischer Speicher in den Funktionen des Moduls `malloc.h` verwendet. Abschnitt 4.2.2 beschäftigt sich eingehender mit der Heap-Implementierung des ESPIDF. In C++ werden auf dem Heap-Objekte und -Daten gespeichert, die mit `new()` angelegt werden.

In der Abb. 4–9 ist die Zuordnung der Segmente zu den Speichern des ESP32-C3 in der Bildmitte ersichtlich. Das ROM und der Flash nehmen Code und Daten auf. Im SRAM und im RTC SRAM wird hauptsächlich das Datensegment, aber auch Code untergebracht (die Ausführung von Code im RTC SRAM ist beim Aufwachen aus dem Tiefschlafmodus zwingend).

Da beim ESP-IDF ein Multitasking-Betriebssystem und damit mehrere Tasks eingesetzt werden, muss nicht nur die Applikation als Task, sondern ebenso das Betriebssystem im Speicher untergebracht werden. Der Linker (siehe Abschnitt 4.2.4) ordnet die Elemente zur Compile-Zeit den einzelnen Speichern zu. In der Abbildung ist diese Zuordnung rechts dargestellt. Auffallend ist, dass der Code auf alle vier Speicher verteilt werden kann. Da der flüchtige SRAM den auszuführenden Code beim Systemstart aber noch nicht enthält, wird er zusätzlich im Flash abgelegt und beim Starten in den SRAM geladen.

Daten können bei Bedarf im RTC SRAM abgelegt werden, im Standardfall wird hier aber der SRAM bevorzugt. Der Heap wird auf alle verfügbaren volatilen Speicher verteilt. Für jeden Task wird beim Anlegen ein eigener Stack im Heap reserviert. Aus diesem Grund können sich die Stacks ebenso auf alle verfügbaren volatilen Speicher verteilen. Bevor näher auf den dynamischen Speicher eingegangen wird, werden die Speicherzugriffe auf Instruktionen und Daten über die entsprechenden Busse betrachtet.

Zugriff auf Instruktionen

Der Instruction Memory ist in RISC-V über den IBUS (Instruction Bus) an den Speicher angebunden. Maschinenbefehle werden zur Ausführung über diesen Bus, auf den 4-Byte aligned zugegriffen werden muss, transportiert.

Beim Booten des Systems wird zunächst der Bootloader, der im ROM gespeichert ist, ausgeführt. Wie Abb. 4–6 zeigt, erfolgt der Zugriff hierfür ab Adresse 0x40000000.

»Wait States« oder »Wait Cycles sind Takte, die der Mikroprozessor auf einen Speicherzugriff warten muss.

Im regulären Anwendungsfall verzweigt der Bootloader in die Applikation (Betriebssystem und auszuführende Programme), die auf dem Flash gespeichert sind. Dies wird durch den Sprung an eine Adresse ab 0x42000000 durchgeführt. Da der Flash-Speicher über SPI angesprochen wird und dementsprechend langsamer ausgelesen werden kann als der Mikroprozessor dies erfordert, wird ein Cache zwischengeschaltet. Ein Cache ist ein schneller Zwischenspeicher, der die meistbenutzten Daten eines langsameren Speichers vorhält (siehe Abschnitt 4.2.3). Dies erlaubt die Ausführung von Code aus dem Cache, auf den ohne »Wait States«, also ohne die Notwendigkeit, auf die Befehle zu warten, zugegriffen werden kann.

Per IBUS-Zugriff an Adressen ab 0x4037C000 kann der Prozessor ein Programm, oder häufiger Programmteile wie einzelne Funktionen, im SRAM ausführen. Dies kann notwendig sein, wenn zeitkritische Handlungen (Reaktion auf externe Ereignisse, zeitgebundene Tätigkeiten, …) ausgeführt werden sollen obwohl gerade Daten in den Flash persistiert werden. Interrupt-Service-Routinen, siehe Abschnitt 6.1, werden deshalb meist im SRAM platziert.

Zugriff auf Daten

Wenn über LOAD/STORE-Befehle auf den Data Memory zugegriffen wird, erfolgen diese Zugriffe über den DBUS. Wie dies auch beim IBUS der Fall ist, werden diese Zugriffe dann je nach Zieladresse auf die verschiedenen physischen Speicher ausgeführt.

Lokale Variablen, Rückgabewerte und Rücksprungadressen werden, wie in Abschnitt 3.3.2 besprochen, nach Möglichkeit in Registern abgelegt. Dieser schnellste Speicher ist jedoch auch klein und rasch erschöpft. Wenn nicht genügend Registerplatz vorhanden ist, wird der Stack zum Sichern von Registern beim Aufruf von Unterfunktionen verwendet. Dieser liegt im RAM, im `sum_up_n`-Beispiel im Abschnitt 3.3.2 an Adresse 0x3FC8EEA0. An dieser Adresse wird über den DBUS auf den schnellen SRAM zugegriffen. Der Zugriff auf das SRAM erfolgt

beim ESP32-C3 ohne Wait States, weshalb hier auch kein Cache vorgeschaltet wird.

Das unveränderliche Array `static const uint32_t N[M]` des χ^2-Test-Beispiels wird an die Adresse 0x3c022990, die per DBUS in den Flash geleitet wird, gelegt. Da der Flash verhältnismäßig groß, seine Programmierung aber zeitaufwendig und mit einem Wearout verbunden ist, macht es Sinn, dass dieser Speicher für solche Konstanten, aber nicht für Variablen verwendet wird. Die Konstanten werden beim Kompilieren festgelegt, ihr Inhalt kann ohne »Programmiertricks« nicht geändert werden. Konstanten, die im Flash liegen, dürfen auch mit diesen Tricks, die bei findigen C-Tüftlern aber mitunter zu finden sind, nicht verändert werden: Der Type Cast in einen beschreibbaren Pointer `uint32_t* pN = (uint32_t*)N` mit anschließender Inhaltszuweisung, beispielsweise `pN[0] = 7`, führt zu einem Programmabsturz, da der Flash auf diese Weise nicht beschrieben werden kann (»Store access fault«). Abstürze führen beim ESP-System zu einer Fehlerausgabe mit anschließendem Reset.

Wenn das Feld nicht konstant angelegt wird, in diesem Fall global `static uint32_t N[M]`, wird es im Beispiel ab Adresse 0x3FC8A0BC, und somit über den DBUS ansprechbar im SRAM, abgelegt. Globale Variablen werden vom Compiler im Datensegment abgelegt. Auch Gedächtnisvariablen, also lokale Variablen, die mit dem Schlüsselwort `static` versehen sind, werden im Datensegment untergebracht. Wird das Array aber lokal definiert, kommt es auf dem Stack zu liegen.

In allen Fällen wird ein Array-Eintrag mit der Assemblerfolge

```
1  slli a5, a4, 0x2
2  add a5, a5, a0
3  lw a5, 0(a5)
```

Listing 4.2 *Laden eines Array-Eintrags*

geladen. Register `a0` enthält die Basisadresse und Register `a4` den Index des Arrays. Da der Basisdatentyp des Feldes `uint32_t` ist und damit 4 Byte groß, muss der Offset mit 4 multipliziert werden. In Zeile 1 wird dieser Offset ins Array über eine Shift-Left-Operation berechnet. Ein Links-Shift um 2 Bit entspricht einer Multiplikation mit $2^2 = 4$ (siehe Abschnitt 4.4.2). In Zeile 2 wird der berechnete Offset zur Basisadresse des Arrays addiert, und in Zeile 3 wird auf diesen Offset per

Load-Befehl zugegriffen. Das Schreiben der Variablen läuft analog über den Store-Befehl.

Dynamischer Speicher

In Programmen ist die Größe der zu verarbeitenden Daten oft nicht zur Compile-Zeit (i.e. statisch) bekannt. Somit kann eine exakte Dimensionierung von Arrays nur schwer abgeschätzt werden.

Als Lösung können Arrays einerseits statisch mit der Maximalgröße initialisiert werden. Dies ist aber mitunter unpraktisch. Soll beispielsweise eine Audiodatei geladen werden, muss Speicher für die größtmögliche Datei vorreserviert werden. Und beim Empfang von Datenpaketen über eine Netzwerkschnittstelle müssen Puffer für die maximale Anzahl an gleichzeitig zu verarbeitenden Paketen in maximaler Paketgröße angelegt werden. Für zu sendende Pakete verhält es sich genauso.

Tab. 4–3 *Statischer vs. dynamischer Speicher*

Speicherung	Vor-/Nachteile
statisch	△ Speicherprobleme zur Compile-Zeit △ deterministisch ▽ Maximum an Speicher notwendig
dynamisch	△ speichereffizient ▽ Speicherprobleme zur Laufzeit ▽ Fragmentierung ▽ indeterministische Dauer bei Allokation und Freigabe

Problematisch ist, dass mit der statischen Reservierung alle Speicher vorreserviert werden müssen, auch wenn sie nicht gleichzeitig verwendet werden. Wird also im Beispiel nicht kommuniziert, während die Audiodatei benötigt wird, belegen die Speicher dennoch den maximalen Platz. Damit muss der Speicher des Systems größer dimensioniert werden, als dies eigentlich praktisch notwendig wäre.

Der große Vorteil der statischen Reservierung ist hingegen, dass bereits zur Compile-Zeit erkannt wird, wenn zu wenig Speicher vorhanden ist. Bei erfolgreicher Kompilierung steht der Speicher dann zur Laufzeit zur Verfügung und kann keine unvorhergesehenen Speicherprobleme verursachen.

Eine Alternative zur statischen Reservierung ist die Verwendung von dynamischem Speicher. Dabei wird der Heap, der ein großer vorreservierter oder bei vielen Systemen ein wachsender Speicher ist, zur Laufzeit in Blöcke angeforderter Größe unterteilt. So kann beispielsweise ein Speicherblock der Größe einer Audiodatei angefordert werden, um diese in den Speicher zu laden. Oder beim Empfang und Senden von Paketen kann der Speicher paketweise angefordert und verwendet werden.

Aus der C-Programmierung ist die Verwendung der Funktionen aus dem Modul `malloc.h`, wie `malloc()`, `calloc()`, `realloc()` und `free()` bekannt. Die Allokationsfunktionen geben einen Zeiger auf den reservierten Bereich zurück, der dann mit `free()` wiederum freigegeben werden muss.

Das typische Muster bei der Verwendung dieser Funktionen ist in Listing 4.3 wiedergegeben. Die Funktion `testRNG()` stellt ein Gerüst dar, um den χ^2-Test aus dem Beispiel in Listing 4.1 zu kapseln, um in der Folge den Hardware-Zufallszahlengenerator zu testen. Der Übergabeparameter `m` gibt die Größe des Histogramms an. Da die Größe übergeben wird, muss das Array dynamisch erstellt (oder statisch für alle erwarteten Fälle groß genug dimensioniert) werden.

```
1  Status_t testRNG(uint32_t observations, uint32_t m) {
2      uint32_t* n = malloc(m * sizeof(uint32_t));
3      if (n == NULL) {
4          return Status_OutOfMemory;
5      }
6      memset(n, 0, m * sizeof(uint32_t));
7      // fetch observations numbers from the random number
8      // generator and run Chi-Square-Test
```

```
9       free(n);

10      n = NULL;

11      return Status_Ok;

12  }
```

Listing 4.3 *Muster für die Verwendung von* `malloc()` *und* `free()`

In Zeile 2 wird das Array mit `m` Elementen allokiert. Da die `malloc()`-Funktion die Anzahl an Bytes als Parameter erhält, wird die tatsächliche Größe des Arrays durch Multiplikation mit der Größe des Basisdatentyps, `sizeof(uint32_t)`, berechnet. Nach der Erzeugung muss das Array initialisiert werden. In Zeile 6 wird die Funktion `memset()` dafür eingesetzt. Alternativ erledigt die Funktion `calloc()` die Allokation mit Initialisierung. Die Anzahl und Größe der Elemente werden dieser Funktion mit zwei separaten Parametern übergeben, wie die Änderung in Listing 4.6 zeigt. Das Array `n` kann nun wie gewohnt verwendet werden, beispielsweise `n[i] = value`. Nach der Verwendung muss der Speicher wieder wie in Zeile 8 freigegeben werden.

Der Zeiger auf den Datenbereich wird gemäß üblicher Programmierrichtlinien auf `NULL` gesetzt (Zeile 9), um einen »Dangling Pointer«, also einen Zeiger auf einen freigegebenen Speicher oder ein gelöschtes Objekt, zu vermeiden. Ein Zeiger auf `NULL`, in `stddef.h` definiert mit `(void *)0`, führt statt des Zugriffs auf ungültigen Speicher zu einer Laufzeit-Exception (beim Leseversuch ein »Load access fault«, beim Schreibversuch ein »Store access fault«).

Die Funktionen zum dynamischen Speichermanagement haben keine deterministische Ausführungsdauer. Dies liegt an der blockweisen Speicherung auf dem Heap. Es gibt zwar verschiedene Verfahren der Reservierung und Freigabe (wie "First-Fit", "Best-Fit", "Next-Fit", "Buddy System"), die sich in Aufwand, Speicherausnutzung und Effizienz unterscheiden, doch grundsätzlich haben alle dieselben Eigenschaften. Eine ausführliche Beschreibung findet sich beispielsweise in [31, Kapitel 8], ein kurzer Überblick folgt hier.

In Abb. 4–10 Teil a) ist ein Teil des Heaps mit reservierten Blöcken eingezeichnet. Jeder Block hat einen Header, in dem steht, ob der Block frei oder vergeben ist, sowie die Größe und bei sicheren Speichermanagern eine Checksumme und mehr. Die von `malloc()` zurückgegebenen Pointer zeigen hinter den Header auf den reservierten Datenbereich.

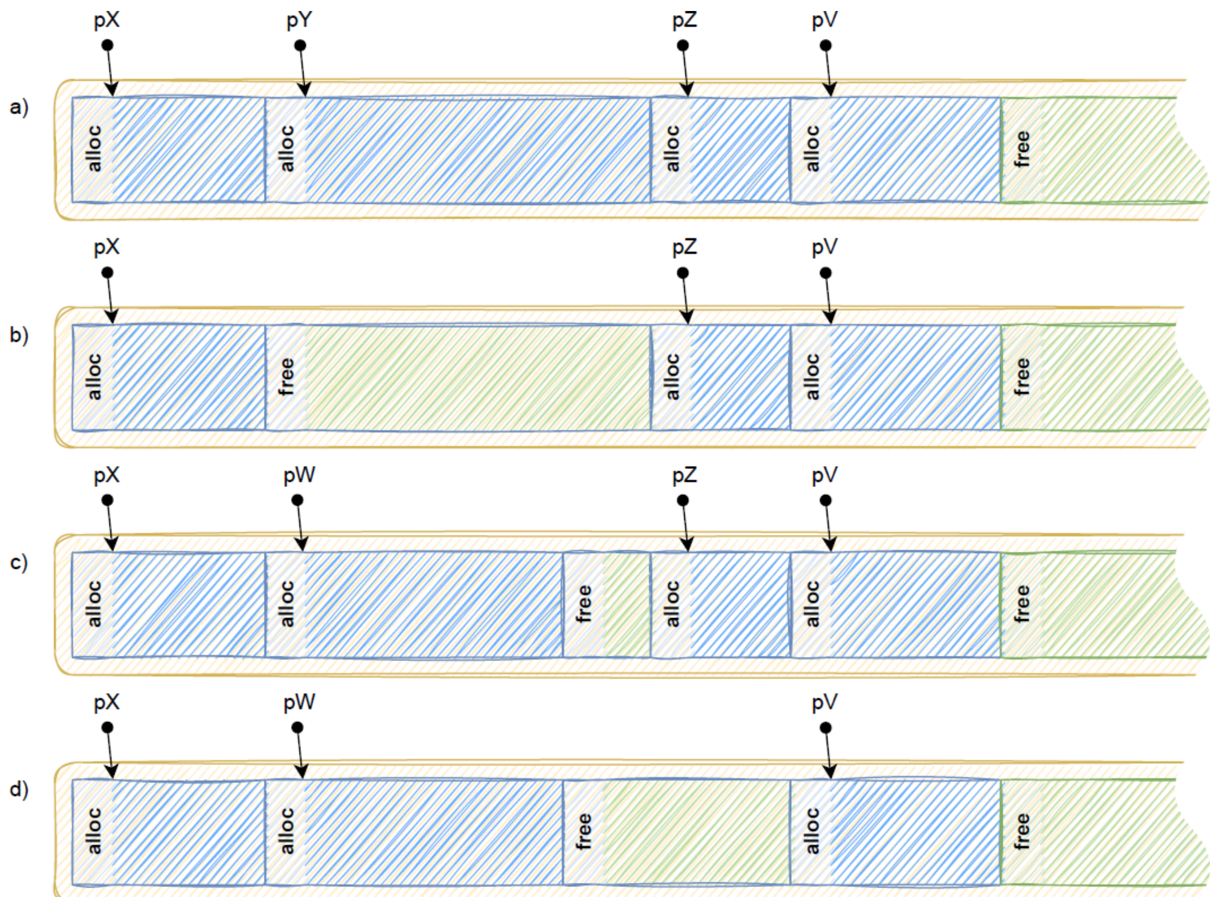

Abb. 4–10 *Blockaufbau des dynamischen Speichers*
b) nach `free(pY)` *eines Blocks*
c) nach `malloc()`
d) nach `free(pZ)`

Die Zeiger `pX`, `pY`, `pZ`, `pV` in der Abbildung wurden nacheinander mit `malloc()` per »First-Fit«-Algorithmus reserviert. Bei diesem Verfahren wird der erste freie Speicherblock, der groß genug für den angeforderten Inhalt ist, vergeben. Dabei wird er in einen reservierten und einen freien Block zerteilt.

Abb. 4–10 b) zeigt den Speicher nach Löschen des zweiten Blocks mit `free(pY)`. Der zugehörige Speicherblock wird dabei im Header auf frei gesetzt. Die meisten Speichermanager löschen den Speicherinhalt nicht, sondern belassen ihn bei den aktuellen Werten.

In der Abbildung Teil c) wurde ein neuer Speicherblock per `pW = malloc(count)` angelegt. Da der Block kleiner angefordert wird, als der freie Block war, kann der zuvor vom größeren Block `pY` belegte Speicher verwendet werden. Dabei entstehen der reservierte Block und ein kleiner freier Block. Dieser kleine Block kann nun zu klein für weitere Allokationen sein, weshalb dann größere freie Blöcke herangezogen werden. In der Folge kann es sein, dass solche kleinen Blöcke, die nicht weiter genutzt werden, über den Speicher verteilt werden. Diese »externe Fragmentierung« kann besonders bei Systemen mit

kleinen Speichern wie Embedded Systemen, zu Laufzeitproblemen führen: Nach langer Zeit erfolgreicher Ausführung kann eine Allokation fehlschlagen. Aus diesem Grund ist gerade in der embedded Programmierung die Testung der `malloc()`-Rückgabe auf `NULL` wichtig.

Auf Systemen mit Festplatten kann die externe Fragmentierung durch eine »Defragmentierung« behoben werden. Dabei werden örtlich zusammengehörende Blöcke auch örtlich nahe auf der Platte verschoben. Freier Speicherplatz wird ebenso zusammengezogen. Eine Defragmentierung ist im dynamischen Speichermanagement nicht möglich, da Pointer auf die reservierten Speicher existieren. Würden in der Abbildung die Blöcke, auf die `pZ` und `pV` zeigen, nach vorne verschoben, würden die Pointer ungültig (»wild pointer«) werden. Eine Änderung der Pointer beim Defragmentieren ist in C nicht möglich, da die Pointer (und Kopien der Pointer, also »aliased pointer«) vom System nicht korrigiert werden können.

Bei der Freigabe eines Blockes wird geprüft, ob auch benachbarte freie Blöcke existieren. Wenn dies wie in der Abbildung Teil d) nach `free(pZ)` der Fall ist, werden diese Blöcke zu einem Block zusammengefasst, um der Fragmentierung entgegenzuarbeiten. Aus der Beschreibung des Verfahrens ist ersichtlich, dass sowohl die Allokation eines Blockes als auch die Deallokation zeitlich nicht deterministisch sind.

Da die Speicherprobleme bei der Verwendung von dynamischen Speicher selten und schwer zu lokalisieren sind, wird statischer Speicher in Programmierstandards für Embedded Systeme, wie dem weit verbreiteten MISRA-C Standard, dynamischem Speicher vorgezogen. Vollständige Informationen zu MISRA sind auf der MISRA-Webseite [41] nachlesbar.

Interne Fragmentierung Die »interne Fragmentierung« ist weniger problematisch als die oben beschriebene externe Fragmentierung. Bei der Reservierung eines Blocks kann es sein, dass ein Aufteilen des Blockes nicht möglich ist, da zu wenig Platz für einen zusätzlichen Header bleibt. In diesem und verschiedenen anderen Fällen wird tatsächlich mehr Speicher reserviert, als angefordert wurde. Da dieser Speicher dann zwar vorhanden und reserviert ist, aber nicht verwendet wird, ist er für das System verloren. Man spricht hier auch, wie beim Abfall eines Materialzuschnitts, von »Verschnitt«.

Auch bei statisch reserviertem Speicher gibt es Verschnitt:

- Ein Array wird in C mit statisch fixer Größe angelegt. Oft sind Arrays aber zu groß reserviert und während der gesamten Laufzeit nicht zur Gänze gefüllt.
- Variablen im Datensegment und auf dem Stack werden aligned positioniert. Werden bei einem 32-bittigen Alignment beispielsweise eine

`uint32_t`-, dann eine `uint8_t`- und eine `int32_t`-Variable angelegt, werden die 32-bittigen Variablen an Adressen, die durch 4 teilbar sind, gelegt. Dies bedeutet, dass die dazwischenliegende 8-bittige Variable mit drei Byte aufgefüllt wird. Dieses Auffüllen wird »Padding« genannt. Im Grunde ist es dem Compiler aber möglich, die Reihenfolge der Variablen zur Optimierung des Verschnitts zu ändern.

- Ebenso werden Strukturen (`struct`) intern mit entsprechenden Paddings aligned. Hier hat der Compiler die Möglichkeit der optimierten Umordnung nicht:

```
struct StructA {
    uint8_t a;
    uint32_t b;
    uint8_t c;
    uint32_t d;
    uint16_t e;
};
```

Eine Bestimmung der Größe mit `sizeof(struct StructA)` liefert 20 Byte zurück, da nach den Komponenten `a` und `c` je drei und nach `e` zwei Padding-Bytes angehängt werden. Die manuelle Umordnung der Komponenten

```
struct StructB {
    uint8_t a;
    uint8_t c;
```

```
        uint16_t e;

        uint32_t b;

        uint32_t d;

    };
```

liefert den optimalen Speicherbedarf von 12 Byte.

Zusammenfassend ist es ratsam, bei der Notwendigkeit effizienter Speicherausnutzung gerade bei der verstärkten Variante des letzten Falles, nämlich bei Arrays von Strukturen, auf die interne Fragmentierung zu achten.

Von einer generellen Anpassung des Compiler-Alignments über `#pragma pack(1)` ist aber abzuraten, da dies zwar die interne Fragmentierung vermeidet, aber zu Lasten der Performance geht. Eine gewisse Speicherverschwendung sowohl durch externe als auch durch interne Fragmentierung ist also nicht sinnvoll vermeidbar und muss deshalb in das Systemdesign eingerechnet werden.

Besonderheiten des ESP-IDF Im ESP-IDF kann wie gewohnt die Funktion `malloc()` zur Speicherreservierung verwendet werden. Als Voreinstellung für `malloc()` wird ein 8-Bit aligned Speicher zurückgegeben. Im Framework stehen jedoch mehrere Heaps mit unterschiedlichen Speichermöglichkeiten zur Verfügung, aus denen über die Funktion `heap_caps_malloc()` gewählt werden kann.

Diese Funktion bietet die Möglichkeit, Angaben über den zu reservierenden Speicher zu machen. Konstanten wie `MALLOC_CAP_EXEC` für Speicher, der ausführbaren Code beinhalten kann, oder `MALLOC_CAP_DMA` zur Verwendung mit Modulen mit Direct Memory Access (siehe Abschnitt 7.4.2) sind im Modul `esp_heap_caps.h` untergebracht. Da es auch Heap-Speicher im 32-Bit aligned adressierbaren IRAM gibt, dient das Flag `MALLOC_CAP_8BIT` zur Angabe, dass 8-Bit aligned Speicher (z.B. DRAM) angefordert werden soll.

Mit `heap_caps_get_free_size(MALLOC_CAP_DEFAULT)` kann die Größe des freien Heap-Speichers ermittelt werden. Unabhängig von der Art der Reservierung wird der Speicher mittels `free()` freigegeben.

4.2.3 Cache

Im Pipeline-Beispiel in Abschnitt 3.1.9 war Berta für das Bemalen von Vogelhäuschen in einer Farbe zuständig. Bei der Bestellung konnte die gewünschte Farbe aus sehr vielen ausgewählt werden. Zum Bemalen holte Berta die jeweilige Farbe aus dem Lager, bemalte das Häuschen und brachte die Farbe wieder zurück. Der Transport dauerte fast so lange wie das Malen selbst. Wenn Berta ein zweites Häuschen in derselben Farbe bemalen sollte, ließ sie die Farbe am Arbeitsplatz und sparte so viel Aufwand ein. Im Laufe der Zeit stellte sich heraus, dass über 80% der Häuschen in Rot oder Blau angemalt werden sollten, weshalb Berta beschloss, diese beiden am häufigsten verwendeten Farben am Platz zu belassen. Vom Prinzip her nicht weit hergeholt wird solches Caching in der Praxis oft eingesetzt, wenn ein lokaler Zugriff wiederholt stattfindet und der entfernte Zugriff erhöhten Aufwand bedeutet.

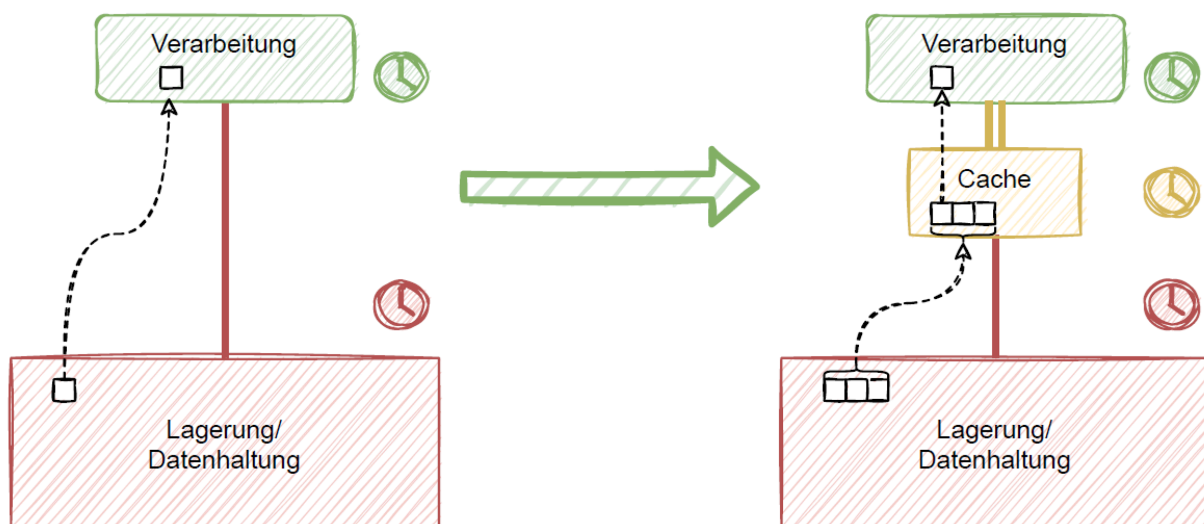

***Abb. 4–11** Einführung eines Cache zur Erhöhung der Performanz*

Schematisch ist die Einführung eines solchen Zwischenspeichers in Abb. 4–11 dargestellt. Im linken Teil der Abbildung wird die Verarbeitung durch den langsamen Zugriff auf die Lagerung bzw. Datenhaltung gebremst. Durch Einführung eines temporären Zwischenspeichers bzw. Cache in der Abbildung rechts, auf den sehr schnell zugegriffen werden kann, kann die Verarbeitung im Regelfall ungebremst stattfinden. Nur wenn das angeforderte Gut bzw. die angeforderten Daten nicht im Cache gelagert sind, muss dieses langsam aus der großen Lagerung nachgefordert werden, was die Verarbeitung in diesem Fall entsprechend bremst.

Dieses Verfahren funktioniert gut, wenn das Prinzip der zeitlichen Lokalität (»Temporal Locality of Reference«) zutrifft. Dieses Lokalitätsprinzip besagt, dass auf Bereiche, auf die zugegriffen wurde, in zeitlicher Nähe mit hoher Wahrscheinlichkeit wieder zugegriffen wird. Im Beispiel der Bemalung von

Vogelhäuschen diente dieses Prinzip der Argumentation des Cache. Es erscheint auch in der Datenverarbeitung plausibel, dass ein bearbeitetes Datum in der Folge weiterverarbeitet wird. Ebenso bewirkt eine Schleife im Code die wiederholte Ausführung derselben Befehle.

Das zweite Lokalitätsprinzip, die räumliche Lokalität (»Spatial Locality of Reference«), besagt, dass auf Daten in unmittelbarer räumlicher Nähe aktuell verarbeiteter Daten wiederum zugegriffen wird. Aus diesem Grund wird bei einem Cache nicht nur das fehlende Datum, sondern ein ganzer Datenbereich (ein »Block« bzw. eine »Line«) nachgeladen. In der Abbildung rechts symbolisieren dies die drei Datenkästchen, die per Mengenklammer zusammengefasst in den Cache geladen werden. Augenscheinlich gilt dieses Prinzip beispielsweise bei der sequenziell indizierten Abarbeitung von Arrays, aber auch bei der sequenziellen Ausführung von Befehlen.

In der technischen Anwendung werden Caches im Speichersystem dann eingesetzt, wenn der lokale Zugriff mit Verarbeitung bedeutend schneller ist als der Zugriff auf den entfernten Speicher. Dies ist beispielsweise bei PC-Systemen zwischen Registerbank und RAM, zwischen RAM und Festplatte, aber auch zwischen Webbrowser und Webserver der Fall.

Wenn verschiedene Speicherkomponenten unterschiedlicher Geschwindigkeiten aufeinander zugreifen, verwendet man auf jeder Ebene einer solchen Speicherhierarchie einen Cache. Abb. 4–12 zeigt die Abhängigkeit grafisch von oben nach unten in einer (Stufen-) Pyramide.

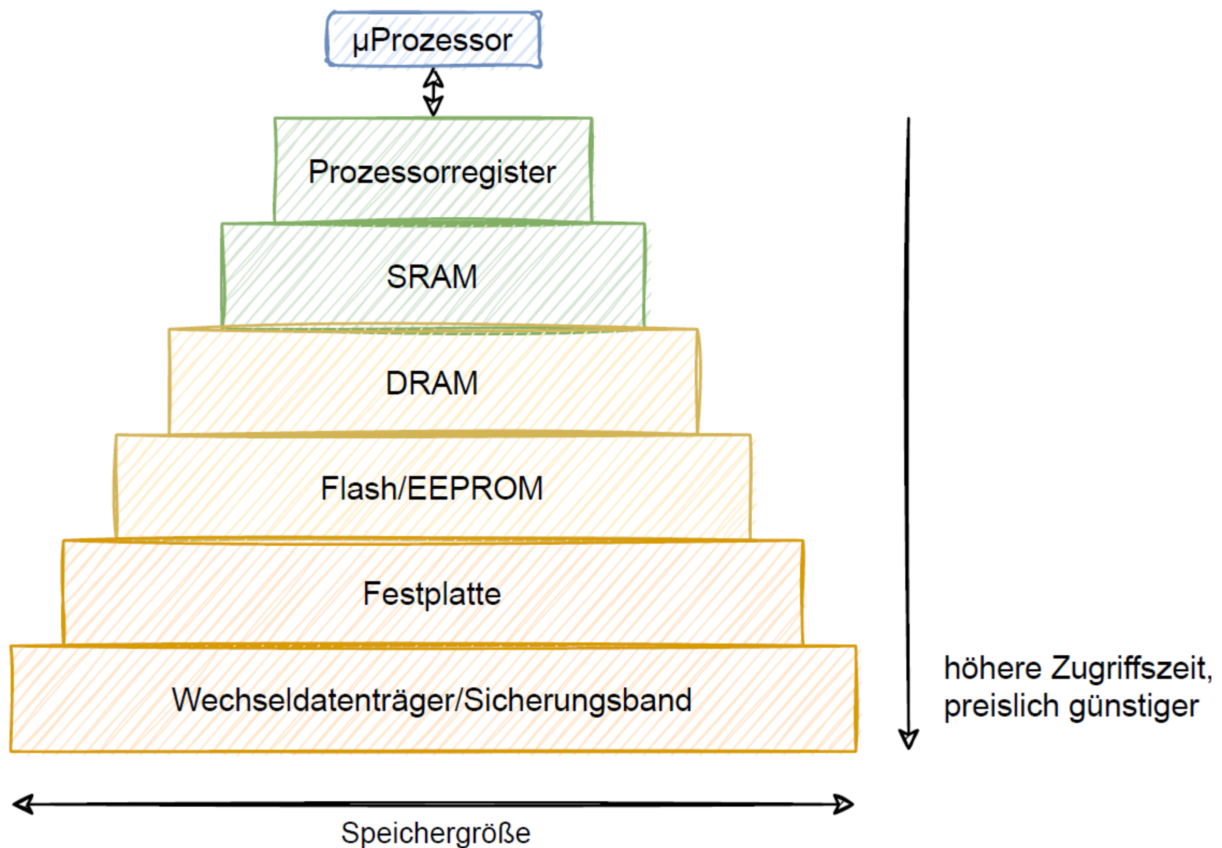

Abb. 4–12 *Speicherhierarchie eines PC-Systems*

Wenn Caches eingefügt werden, werden diese von oben nach unten als First-/Second-/Third-/ … Level Cache bzw. L1, L2, L3, … bezeichnet. In modernen PCs sind die ersten drei Cache-Level mit den Cores direkt auf dem Silizium untergebracht. Zur Festplatte wird ein Cache im RAM zur Vermeidung von Buszugriffen und ein Cache in der Festplatte zur weiteren Pufferung von Lese-/Schreibzugriffen angebracht.

In Embedded Systemen werden weniger Caches eingesetzt, da der RAM oft zur Gänze als schneller SRAM ausgeführt wird und weniger Ebenen in der Speicherhierarchie vorhanden sind.

Ein Cache arbeitet dann gut, wenn seine Gesamtgröße und Blockgröße gut auf die Lokalität des Codes abgestimmt sind. In diesem Fall ist das Verhältnis der Zugriffe auf vorhandene Daten (*Hit*) zu Zugriffen auf nachzuladende Daten (*Miss*), die *Hitrate* = $Hitrate = \frac{Hit}{Hit+Miss}$ hoch. Die *Missrate* ergibt sich als 1 − *Hitrate*. Die Zeit des Zugriffs bei einem Hit (*Hittime*) ergibt sich aus der Dauer des Feststellens, ob die angeforderten Daten im Cache sind, und der Dauer der Rückgabe der Daten. Im Falle eines Miss kommt noch die *MissPenalty*, die Dauer des Nachladens, Ersetzens und Zurücklieferns der Daten hinzu. Damit ergibt sich die durchschnittliche Zeit für einen Speicherzugriff als *Accesstime* = *Hittime* + *Missrate* · *MissPenalty*.

Cache-Organisation

In der Literatur finden sich drei Organisationsarten von Caches, die sich hauptsächlich auf die Hitrate und den Implementierungsaufwand und die Hittime auswirken. Schematisch sind diese Arten für einen Cache mit 16 KiB und einer Blockgröße von 32 Byte in Abb. 4–13 bis 4–15 dargestellt.

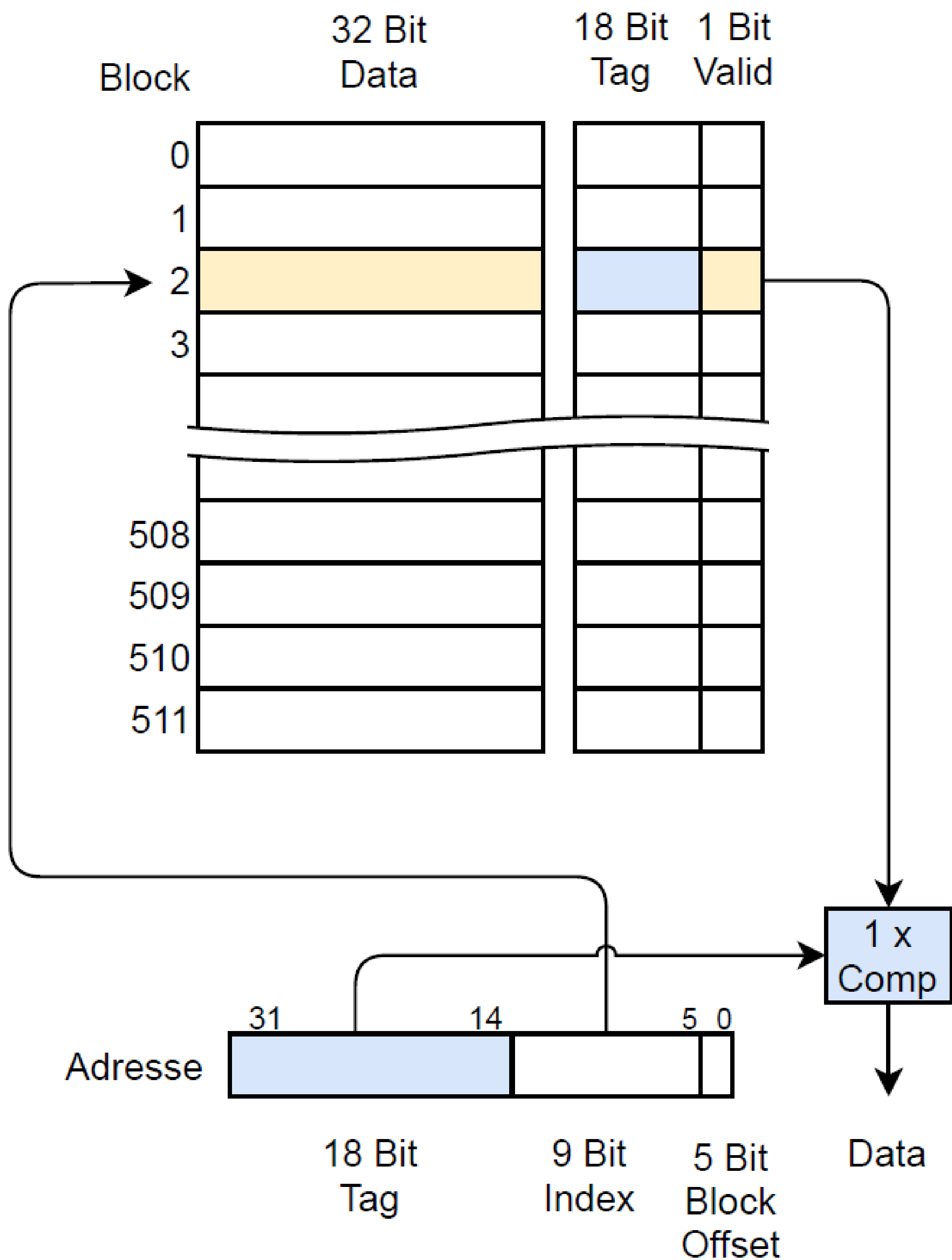

Abb. 4–13 *Ein 16-KiB-Cache mit 32-Byte-Blöcken als direct-mapped Cache*

Direct-mapped Cache Den einfachsten Aufbau hat der »direkt abgebildete« (»direct-mapped«) Cache. Die Adresse des Zugriffs wird in Tag, Index und Block Offset unterteilt. Im Beispiel benötigt der Block Offset, der ein Byte innerhalb eines Blocks adressiert, 5 Bit, da ein Block $32 = 2^5$ Byte umfasst.

Der Block selbst wird durch den Index adressiert, und zwar *Cacheline*(*Address*) $\equiv \frac{Address}{2^{Blocksize}} \bmod \frac{Cachesize}{Blocksize} \equiv \frac{Address}{32} \bmod 512$. Damit werden beispielsweise die Adresse 0x42 auf Zeile 2 abgebildet. Nach dem Auslesen des Blocks wird dann über den Block Offset, in diesem Beispiel 2 oder 3, auf das betreffende Datenbyte zugegriffen.

Durch die Restklassenbildung werden verschiedene Adressen auf denselben Block abgebildet. Ein Schreibzugriff an 0x4042 wird beispielsweise auch auf Zeile 2 abgebildet. Eine solche Kollision führt zum Überschreiben des gespeicherten Blocks für Adresse 0x42. Wird wechselweise auf diese Adressen zugegriffen, entstehen Kollisionen im Cache, auch wenn noch viele andere Blöcke frei wären. Diesen Hauptnachteil von direkt abgebildeten Caches versuchen die anderen Organsationsformen zu vermeiden.

Damit beim Lesen eindeutig bestimmt werden kann, ob der im Cache gespeicherte Block der Adresse entspricht, wird der obere Bereich der Adresse, das »Tag«, zur Cache-Zeile gespeichert. Beim Auslesen wird das gespeicherte Tag mit dem Tag der zu lesenden Adresse über einen Komparator verglichen. Die Speicherung der Tags bedeutet für den exemplarischen Cache einen Zusatzbedarf von $512 \cdot 18$ Bit $\approx 1,2$ KiB.

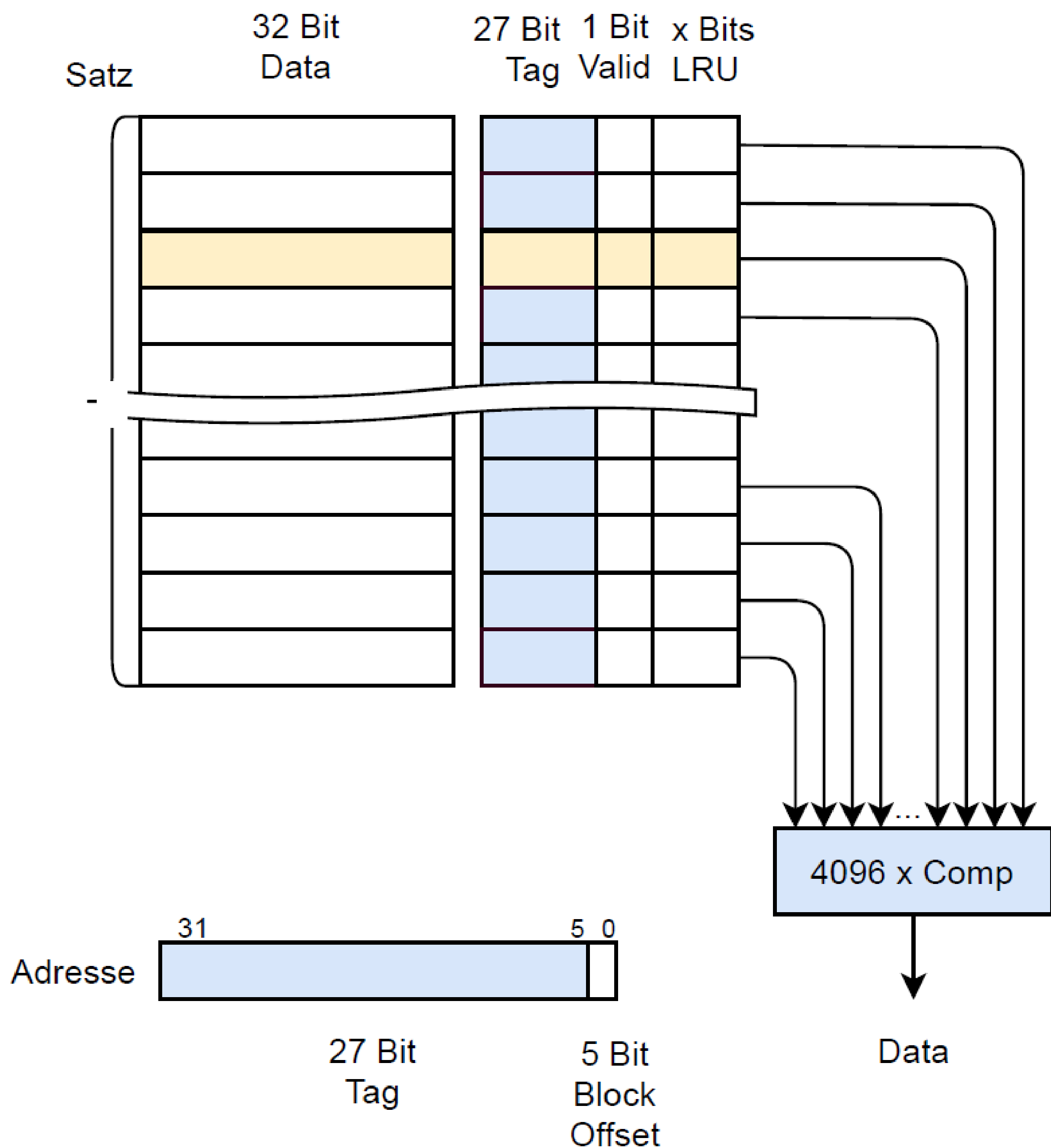

Abb. 4–14 *Ein 16-KiB-Cache mit 32-Byte-Blöcken als fully associative Cache*

Da der Cache zu Beginn leer ist, wird bei jeder Zeile zusätzlich ein Valid-Bit mitgespeichert, das angibt, ob die entsprechende Zeile einen gültigen Block enthält. Wenn dies nicht der Fall ist, wird auf den Speicher durchgegriffen.

Fully associative Cache Beim vollassoziativen Cache erfolgt der Zugriff auf die einzelnen Zeilen im Gegensatz zum direkt abgebildeten nicht über einen aus der Adresse abgeleiteten Index. Vielmehr werden die Tags aller Einträge mit dem Tag der Adresse verglichen.

Vorteilhaft ist hierbei, dass ein Eintrag in einer beliebigen Zeile gespeichert werden kann und Kollisionen damit mit einer guten Ersetzungsstrategie am besten

vermieden werden können. Dieser Vorteil geht aber zu Lasten eines hohen Implementierungsaufwands, da für jede Zeile ein eigener Komparator und ein entsprechend großer Multiplexer notwendig werden.

Zusätzlich zum gestiegenen Speicheraufwand für die Tags muss eine geeignete Ersetzungsstrategie mit entsprechendem Speicher implementiert werden. Abschließend wird auch die Hittime drastisch erhöht.

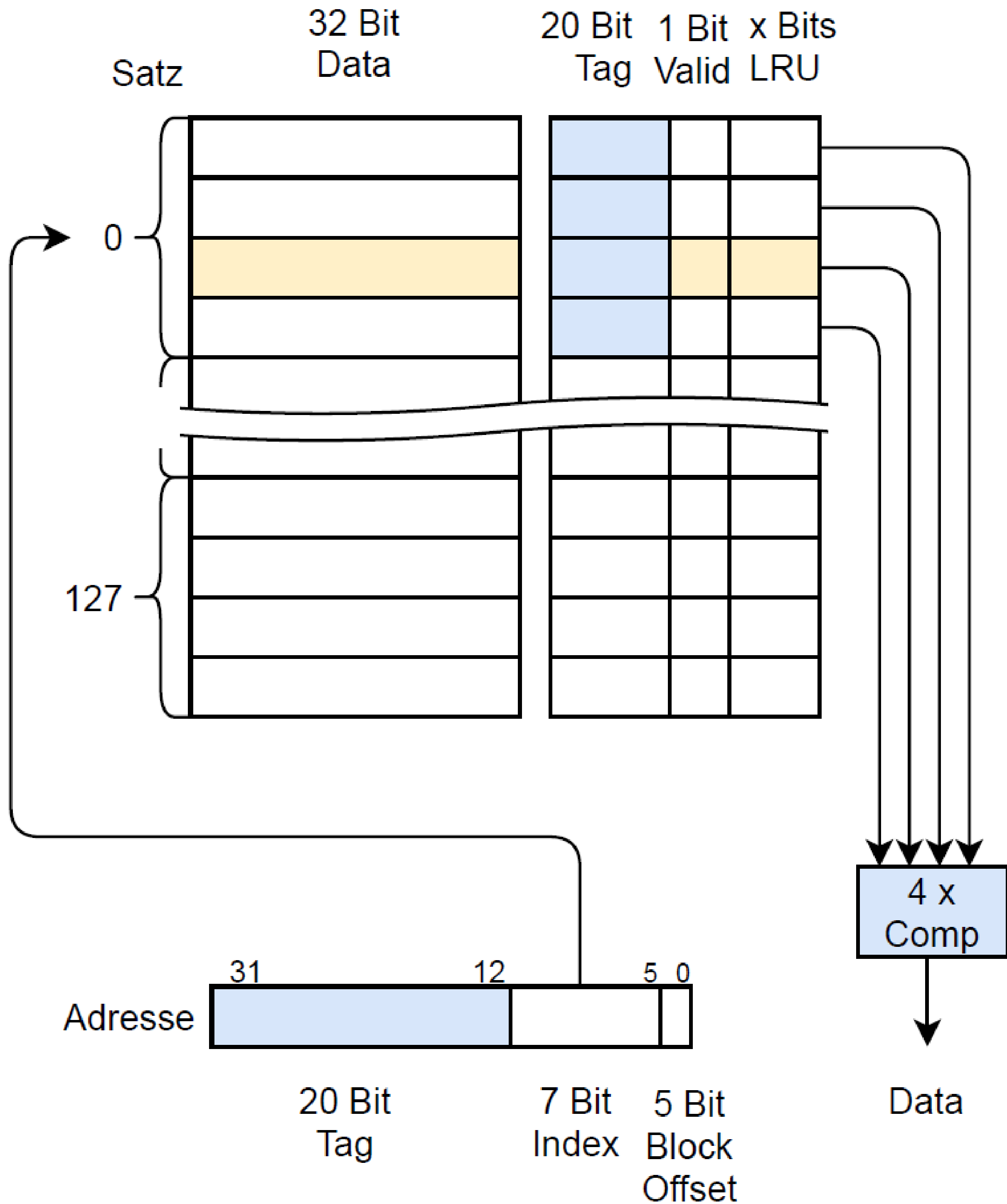

Abb. 4–15 *Ein 16-KiB-Cache mit 32-Byte-Blöcken als 4-way set-associative Cache*

Set-associative Cache Der *n*-Weg-satzassoziative Cache fasst jeweils *n* der insgesamt *m* Blöcke zu seinem Satz zusammen, resultierend in $\frac{m}{n}$, im Beispiel $\frac{512}{4}$ = 128 Sätzen. Die einzelnen Sätze werden über einen aus der Adresse abgeleiteten Index adressiert. Innerhalb eines Satzes wird über *n* Komparatoren mit dem Tag verglichen. Ein Satz verhält sich also intern vollassoziativ, weshalb Kollisionen innerhalb eines Satzes durch geeignete Ersetzungsstrategien möglichst vermieden werden.

Ein direkt abgebildeter Cache ist damit als Spezialfall 1-Wegsatzassoziativ und ein vollassoziativer Cache *m*-Weg-satzassoziativ zu sehen. In der Praxis bietet der *n*-Weg-satzassoziative Cache einen guten Kompromiss aus Kollisionsvermeidung, Speicherbedarf, Implementierungsaufwand und Hittime. Typische Assoziativitäten sind $n = 2$, $n = 4$ und $n = 8$.

Ersetzungsstrategie

Beim Nachladen eines Blocks in assoziativen Caches muss entschieden werden, welcher Block ersetzt werden soll. Naheliegend ist das Ersetzen eines als ungültig markierten Blocks.

Wenn aber das Set voll ist, wäre es am vernünftigsten, den Eintrag zu ersetzen, der am längsten nicht mehr gebraucht wird. Dafür müsste aber ein unmöglicher Blick in die Zukunft stattfinden.

Aufgrund der zeitlichen Lokalität werden üblicherweise »Least Recently Used«(LRU)-Algorithmen eingesetzt. Bei einem 2-Weg-assoziativen Cache kann über ein Bit entschieden werden, welcher der beiden Datensätze relativ zueinander zuletzt verwendet wurde, indem bei diesem Datensatz das LRU-Bit gesetzt und beim anderen gelöscht wird.

Eine Aufzeichnung der Zugriffe bei höherer Assoziativität wird zunehmend speicher-, zeit- und implementierungsaufwendig. Aus diesem Grund wird hier auf die LRU-Exaktheit verzichtet und beispielsweise Tree-based Pseudo LRU, das 1 Bit pro Eintrag benötigt, eingesetzt. Eine weitere Alternative ist das Ersetzen eines zufälligen Eintrags, was den Determinismus des Systems allerdings stört. Die einfacheren Ersetzungsstrategien bewähren sich in der Praxis durch wenig schlechtere Missrates bei deutlich geringerem Ressourcenaufwand.

Konsistenz

Wenn ein System parallel auf den Speicher zugreifende Einheiten wie CPUs, Caches oder DMA-Controller (siehe Abschnitt 7.4.2) beherbergt, können Probleme

mit der Datenkonsistenz auftreten. In diesem Fall haben nicht alle Speicherkopien denselben Inhalt.

In Abb. 4–16 ist dieses Problem schematisch dargestellt. Angenommen, ein Block des Hauptspeichers hat den arbiträren Wert AAAA. Cache A lud diesen Datenblock, um die Daten zu bearbeiten, und speicherte somit eine Kopie. Ein im Anschluss ausgeführter Schreibzugriff änderte die Kopie auf den Inhalt BBBB. Um festzuhalten, dass der Block verändert wurde, wird dies in der Cache-Zeile mit einem »Dirty«-Bit markiert. Der Datenblock im Cache ist damit nicht mehr mit dem Original im Speicher konsistent.

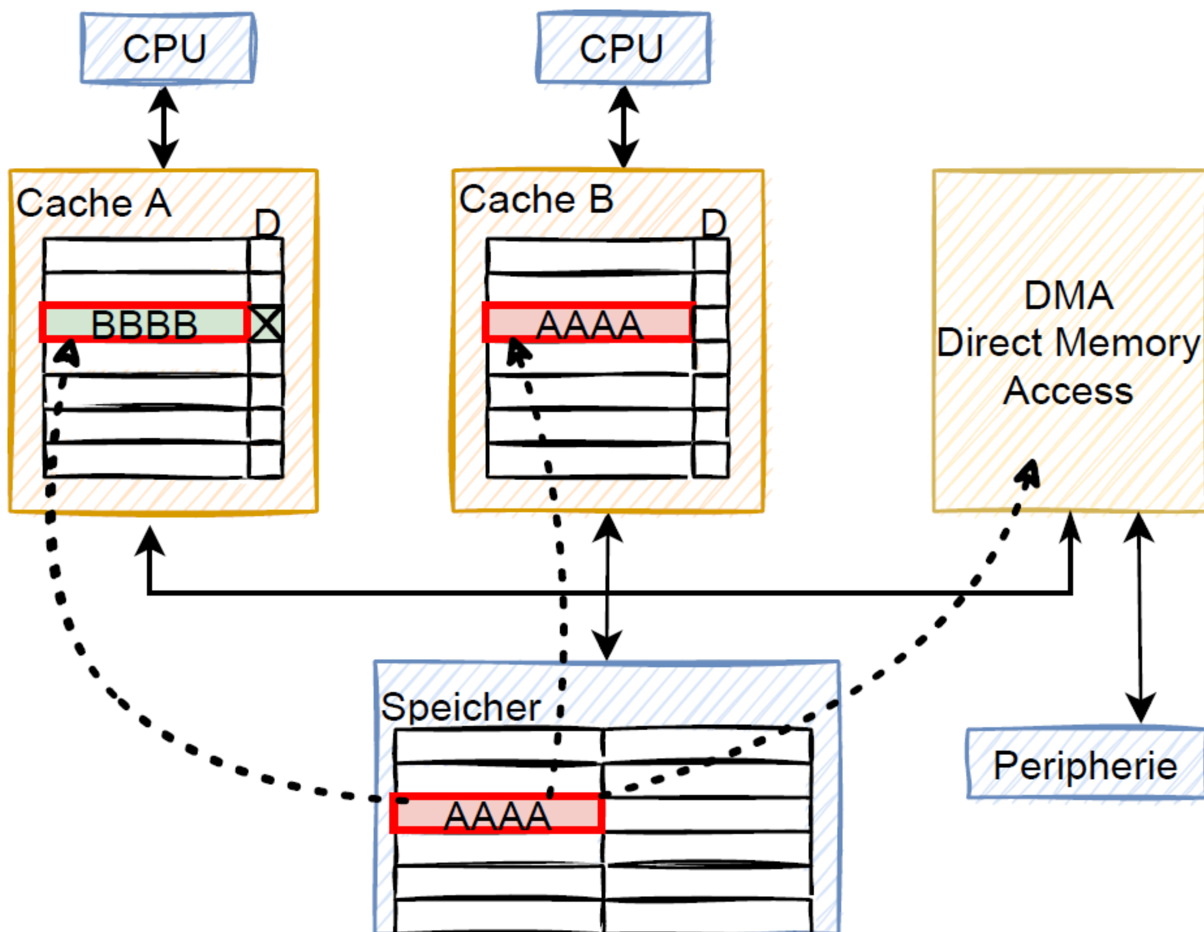

Abb. 4–16 *Cache B ist inkonsistent, da ein Datenblock in Cache A geändert wurde. Zugriffe von Cache B und DMA erfolgen inkohärent.*

Dies stellt kein Problem dar, sofern nicht weitere parallele Kopien existieren oder parallele Zugriffe stattfinden. In der Abbildung führt ein Zugriff des DMA-Controllers nun zu veralteten, im schlimmsten Fall ungültigen Daten. Auch Cache B und die auf diesen zugreifende CPU haben die Aktualisierung aus Cache A nicht.

Um parallele Datenbearbeitung mit solchen inkonsistenten Kopien zu ermöglichen, muss Kohärenz garantiert werden. Kohärenzmechanismen garantieren, dass Zugriffe immer auf die letzte Änderung erfolgen, auch wenn lokal inkonsistente Kopien existieren.

Der Cache des ESP32-C3

Der ESP32-C3 hat für den Zugriff auf den externen Speicher (i.e. Flash) einen 8-Weg-satzassoziativen Cache mit 16 KiB und einer Blockgröße von 32 Byte. Für Zugriffe auf den schnellen RAM wird kein Cache benötigt. Auf die Peripherie wird ebenso ohne Cache zugegriffen. Ein Cache hätte beim Zufallszahlengenerator das Ergebnis, dass immer dieselben lokal im Cache gespeicherten Zufallszahlen zurückgeliefert würden. Als Folge des direkten Zugriffs erfolgt der periphere Datentransfer gegebenenfalls mit Wait States.

Designbedingt gibt es in diesem System kein Cache-Konsistenzproblem, da der Cache generell nur lesende Zugriffe unterstützt. Im Betrieb mit üblicher Speicherkonfiguration, also mit externem Flash-Speicher, wird nur lesend über den IBUS oder den DBUS zugegriffen. Wenn Anfragen über IBUS und DBUS gleichzeitig gestellt werden, kann der Cache diese allerdings nur sequenziell mit entsprechenden Wait States abarbeiten. Für eine Konfiguration mit externem RAM ist der Cache nicht vorgesehen.

Der Cache ist konfigurierbar und lässt sich abschalten. In diesem Fall stehen 16 KiB mehr RAM, aber eben kein Cache zur Verfügung.

In der Linguistik und Kryptoanalyse sind Histogramme ein wichtiges Indiz für die Sprache bzw. Art der Kodierung.

Um das Verhalten des Cache zu analysieren, kann eine Applikation, wie in Listing 4.4 schematisch wiedergegeben, herangezogen werden. Hier wird ein Histogramm über einen Text, das Nibelungenlied, berechnet. Dieses wird, da `const` deklariert, im Flash abgelegt. Das Array `histogram` hat 256 Einträge, um sämtliche ASCII-Zeichen inklusive den erweiterten 8-Bit-Bereich abzudecken.

Um den Cache an seine Grenzen zu bringen, wird der Text in Sprüngen der Blockgröße von 32 Byte gelesen, in der ersten Runde an Offsets $0 + i \cdot 32$, in der zweiten $1 + i \cdot 32$ usw. (Codezeilen 12–16).

```
1   #define TESTSIZE              (1 * 1024)
2   #define BLOCKSIZE             32
3   const char gNibelungenlied[]=
4      "Aventivre von den nibelvngen: Vns ist in alten "
5      "maeren wunders vil geseit von heleden lobebaeren "
```

```
6       "von grozer arebeit […]";
7   // […]
8   uint32_t histogram[256] = { 0 };
9   uint32_t offs = 0;
10  for (uint32_t i = 0; i < TESTSIZE; i += 1) {
11      histogram[(unsigned char)gNibelungenlied[offs]] += 1;
12      offs += BLOCKSIZE;
13      if (offs >= TESTSIZE) {
14          offs -= TESTSIZE;
15          offs += 1;
16      }
17  }
18  // […]
```

Listing 4.4 *Histogramm über die Buchstaben des Nibelungenlieds*

Die Messergebnisse mit Arrays unterschiedlicher `TESTSIZE` sind in der Grafik Abb. 4–17 zusammengefasst. Die ausgeführten Instruktionen steigen mit der `TESTSIZE` weitestgehend linear an. Die Einbrüche an den Größen $n \cdot 4$ lassen sich durch effizienteren Code erklären: Jeder Testlauf wurde separat kompiliert, und die Offsetabfrage wird in diesen Fällen vom Compiler performanter übersetzt.

Wesentlicher ist die Betrachtung der gemessenen Taktzyklen. Der Cache arbeitet bis etwa 15 KiB höchst effizient, um dann bei 16 KiB zu steigen. Ab 17 KiB kann der

Cache nicht mehr sinnvoll arbeiten.

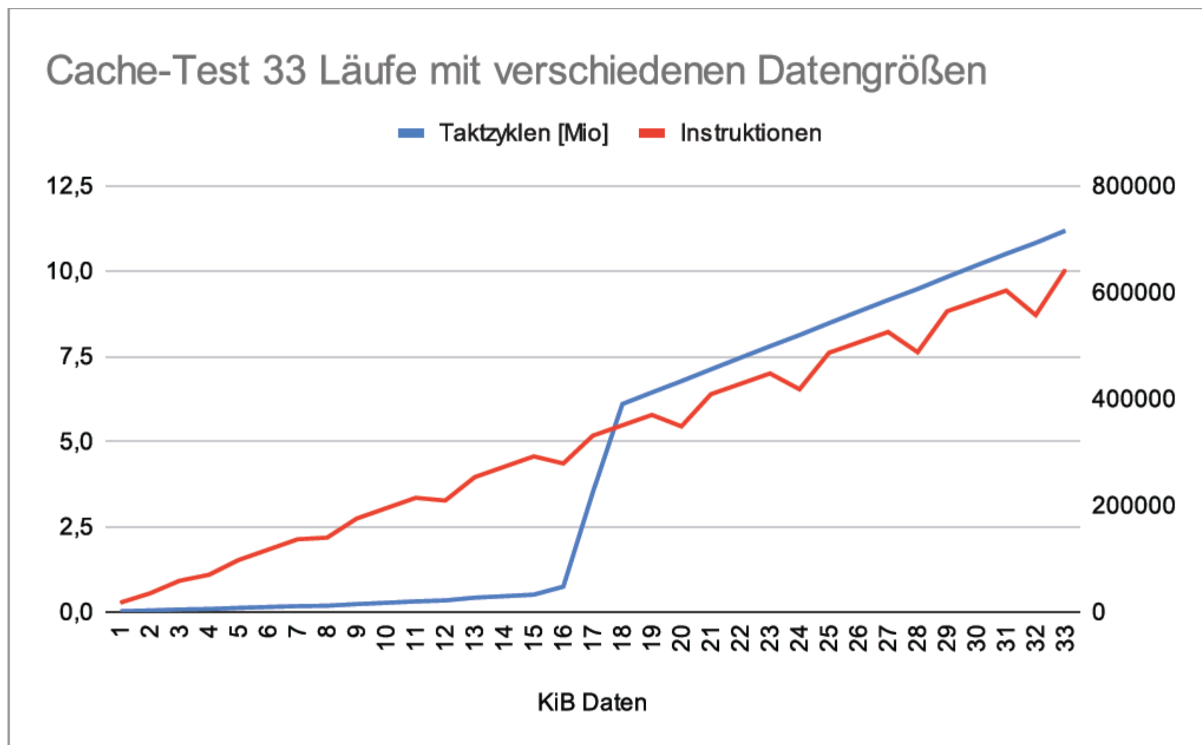

Abb. 4–17 *Messergebnisse der Histogrammapplikation in Listing 4.4*

Wird der Text aber sequenziell durchgearbeitet, tritt das Cache-Problem nicht in dieser Form auf, da dann jeweils 32 Zugriffe auf derselben Page getätigt werden. Noch effizienter ist das Arbeiten auf dem Text im SRAM, da in dem Fall weder nach dem Programmstart auf den langsamen externen Flash zugegriffen noch der Cache verwendet werden muss.

Zusammenfassend arbeitet der Cache für den Flash auf dem ESP32-C3 sehr gut für Instruktionen und Daten. Wenn allerdings auf relativ große Tabellen stark verteilt zugegriffen wird, kann eine Speicherung im SRAM sinnvoll sein. Der SRAM arbeitet ohne Cache und ohne Wait States, ist in seiner Größe gegenüber dem Flash allerdings stark eingeschränkt.

Operationen mit großen Matrizen, wie beispielsweise eine Matrizenmultiplikation, stellen bei der Implementierung für Systeme mit Caches eine Herausforderung dar.

4.2.4 Linker

Die Anordnung von Code und Daten im Speicher übernimmt der Linker anhand von Regeln, die im Linker Script definiert sind. Über dieses Skript wird gesteuert, wie die Segmente bzw. »Sections« den verschiedenen Speicherarten zugeordnet werden.

Wenn eine globale Variable beispielsweise im `.noinit`-Segment untergebracht wird, wird ihr Inhalt beim Reset nicht mit dem Standardwert 0 belegt. Auf diese Weise kann eine Variable über einen Reset hinweg ihren Wert behalten.

Das entsprechende GCC-Statement ist

```
__attribute__((section(".noinit"))) uint32_t gKeepValue;
```

Im ESP-IDF sind die vordefinierten Segmente in der Header-Datei `esp_attr.h` untergebracht. Obiges Statement entspricht

```
__NOINIT_ATTR uint32_t gKeepValue2;
```

Noch nützlicher ist die Deklaration als

```
RTC_NOINIT_ATTR uint32_t gDeepKeepValue;
```

In diesem Fall wird die Variable im RAM der Echtzeituhr (RTC) abgelegt. Dies hat den Vorteil, dass der Inhalt auch in den diversen Stromsparmodi nicht verloren geht. Soll der Inhalt aber über das Entfernen der Energieversorgung hinaus persistent sein, muss er im Flash abgelegt werden.

Um die Platzierungen von Code und Daten überprüfen zu können, legt der Linker eine `<projektname>.map`-Datei im `build`-Verzeichnis an. In dieser Datei ist ersichtlich, dass die Variable `gDeepKeepValue` im `.rtc_noinit`-Bereich abgelegt wurde:

```
.rtc_noinit    0x50000010    0x4              main.c.obj
               0x50000010    gDeepKeepValue
```

Zur Zuordnung der Segmente zum physischen Speicher sind im Skript `memory.ld` die Speicherregionen namentlich definiert. Der Ausschnitt

```
MEMORY
```

```
{
  rtc_iram_seg(RWX) : org = 0x50000000, len = 0x2000 - 0
}
REGION_ALIAS("rtc_data_location", rtc_iram_seg );
```

legt fest, dass der benannte Speicherbereich `rtc_iram_seg` an der physischen Adresse 0x50000000 liegt und 0x2000 B groß ist. Der Speicher darf gelesen(`R`) und beschrieben(`W`) werden. Außerdem darf in diesem Adressbereich auch ausführbarer Code(`X`) platziert werden. Der Speicherbereich kann alternativ über den Namen `rtc_data_location` angesprochen werden.

Das Skript `sections.ld` beinhaltet Regeln zur Zuordnung der Segmente in den benannten physischen Speicher, wie der folgende Ausschnitt mit der Regel für das Segment `rtc_noinit` zeigt.

Zeilen 1 und 8 definieren, dass das Segment `rtc_noinit` im Speicher `rtc_data_location` untergebracht werden soll. Eine spezielle Bedeutung hat der alleinstehende Punkt, der in den Zeilen 3, 4, 6 und 7 Verwendung findet. Er stellt die aktuelle Adresse dar, an der der Linker arbeitet. Startend bei 0 wird er mit jedem platzierten Element um die Größe des Elements hochgezählt. Wird beispielsweise das Element A mit der Größe 128 B zu Beginn platziert, kommt es an der Adresse 0 zu liegen und `.` erhält den Wert 128. Wird Element B mit der Größe 220 B im Anschluss platziert, erhält es die Adresse 128, und der `.` wird auf 348 erhöht.

Mit dem `ALIGN`-Statement wird die Adresse auf die nächste durch den gegebenen Wert teilbare Adresse erhöht. In Zeile 5 werden Segmente wie `.rtc_noinit` und `.rtc_noinit.5` über Wildcards gelinkt.

```
1  .rtc_noinit (NOLOAD):
2  {
3    . = ALIGN(4);
4    _rtc_noinit_start = ABSOLUTE(.);
```

```
5     *(.rtc_noinit .rtc_noinit.*)
6     . = ALIGN(4) ;
7     _rtc_noinit_end = ABSOLUTE (.);
8   } > rtc_data_location
```

In den Zeilen 4 und 7 wird auf den `.` lesend zugegriffen. Dadurch stehen dem Linker `rtc_noinit_start` und `rtc_noinit_end` als Symbole zur Verfügung. Es ist möglich, vom C Code aus auf Linkersymbole zuzugreifen, um beispielsweise Daten über die Speicherbelegung zu verwenden. Die beiden Symbole können direkt als Namen für `extern` deklarierte Variable verwendet werden:

```
extern uint32_t _rtc_noinit_start;
extern uint32_t _rtc_noinit_end;
```

Der Zugriff auf die Adresse der Variablen gibt dann deren Platzierung im Speicher zurück. Das Codestück

```
uint32_t* pRTCNoinitStart = &_rtc_noinit_start;
uint32_t* pRTCNoinitEnd = &_rtc_noinit_end;
printf("rtc noinit start 0x%8p, rtc noinit end 0x%8p\n",
pRTCNoinitStart, pRTCNoinitEnd);
```

zeigt dieselben Adressen wie das `.map`-File. Die Adresse 0x50000010 liegt im RTC-SRAM, wie dies auch in Abb. 4–6 dargestellt ist. Auf diese Adresse wird über das Peripheriesystem zugegriffen.

4.3 Peripheriemodule

Als Peripherie werden Komponenten bezeichnet, die außerhalb der CPU liegen. Hier sind Komponenten untergebracht, die, hauptsächlich um die CPU zu

entlasten, Spezialaufgaben mit den unterschiedlichsten Anforderungen lösen.

Zu den sehr unterschiedlichen Aufgaben gehören die Ein- und Ausgabe von digitalen und analogen Signalen, die Kommunikation über verschiedene Protokolle, das Zählen von Ereignissen und »echter« Zeit, die effiziente Sicherung mit kryptografischen Methoden, das Überwachen des Systems zur frühzeitigen Fehlererkennung und vieles mehr.

Da diese Module über einen standardisierten Bus an die CPU angeschlossen sind, können dieselben Module von den Herstellern in Mikrocontrollern mit verschiedenen Kernen integriert werden. Der ESP32-S3 hat beispielsweise viele Komponenten mit dem ESP32-C3 gemein, auch ein analoges Speichermodell der Peripherie, jedoch zwei Xtensa-LX7-CPUs. Oft kaufen die Hersteller die Module auch von Drittanbietern zu und integrieren diese in ihrem Silizium.

In diesem Abschnitt wird der allgemeine Zugriff auf ein Peripheriemodul anhand des Zufallszahlengenerators betrachtet. Teil II des Buchs beschreibt die Einsatzmöglichkeiten und die Programmierung einer Vielzahl an Modulen, wie sie auch im ESP32-C3 implementiert sind.

4.3.1 Peripheriezugriff

Wie in den Abschnitten 4.1.2 und 4.1.3 besprochen, erfolgt der Zugriff auf die Peripherie über das Bussystem. Das Peripheriemodul hat einen über die Memory Map zugewiesenen Speicherbereich. Busadressen in diesem Bereich werden vom Adressdekoder angenommen und in lokale Offsets übersetzt. Auf diese Weise wird auf spezielle Speicherzellen, die ebenso wie der innere CPU-Speicher »Register« heißen, für den Datenaustausch zugegriffen.

Üblicherweise haben Module mehrere Register zum Setzen und Lesen von Zuständen sowie zur Datenübertragung. Der im Vergleich mit anderen Modulen einfach anzusprechende Zufallszahlengenerator (RNG, Random Number Generator) ist in Abb. 4–18 dargestellt.

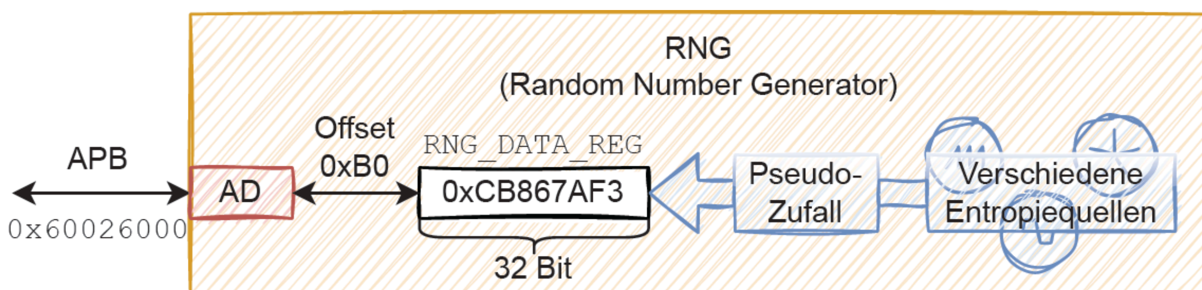

Abb. 4–18 *Schema des Zufallszahlengenerators mit Adressierung*

Das thermische Rauschen und die Laufungenauigkeit eines Taktgenerators werden gemessen und als Entropiequelle für einen Pseudo-Zufallszahlengenerator verwendet. Auf diese Weise werden fortwährend nicht deterministische Zufallszahlen im Register `RNG_DATA_REG` abgelegt.

Lesezugriff

Die CPU kann eine 32-Bit-Zufallszahl aus diesem Register auslesen. Die Basisadresse des RNG-Moduls ist 0x60026000, der Offset des Registers `RNG_DATA_REG` ist 0xB0.

Im Gegensatz dazu wird bei »Port-Mapped I/O« für die Peripheriegeräte ein eigener kleiner Speicherbereich definiert, auf den mit spezialisierten Instruktionen, beispielsweise `in` und `out`, zugegriffen wird. Dies hat hauptsächlich den Vorteil, dass kein Hauptspeicherbereich wegfällt.

Die Adresse ist im Reference Manual angeführt [26, Kapitel 24]. Dass das Bussystem mit den Adressen der Peripherieregister im Adressraum des Hauptspeichers untergebracht ist und der Zugriff über diese Adressen erfolgt, wird »Memory-Mapped I/O« genannt.

Der große Vorteil von Memory-Mapped I/O ist hingegen der einfache Zugriff aus einer Hochsprache wie C oder C++, ohne dafür Assembler oder intrinsische Funktionen (siehe Abschnitt 3.3.1) verwenden zu müssen. Über Pointer kann man die Register typsicher ansprechen, wie in Listing 4.5 durchgeführt.

```
1  #define RNG_BASE 0x60026000
2  #define RNG_DATA_REG_OFFS 0xB0
3
4  volatile uint32_t* pRngDataReg =
5      ↪ (volatile uint32_t*)
6      ↪ (RNG_BASE | RNG_DATA_REG_OFFS);
7
8  inline uint32_t nextRand() {
```

```
 9      return *pRngDataReg;
10  }
```

Listing 4.5 *Zugriff auf den Zufallszahlengenerator per Pointer*

In Zeile 1 wird die Basisadresse des RNG-Moduls, in Zeile 2 der Offset des Registers `RNG_DATA_REG` definiert. In Zeile 4 wird ein Pointer auf die Adresse `(RNG_BASE | RNG_DATA_REG_OFFS)` als `uint32_t`-Pointer angelegt, da es sich um ein 32-Bit-Register handelt. Der explizite Typecast auf den Pointertyp ist notwendig, da die Zuweisung der Adresse auf `pRngDataReg` von Integer auf Integer-Pointer nicht zuweisungskompatibel ist.

Der Qualifizierer `volatile` ist an dieser Stelle notwendig, um dem Compiler mitzuteilen, dass es sich um eine Variable handelt, deren Wert sich ohne einen dem Compiler bekannten Zugriff ändern kann. Dies ist möglich, wenn parallele Ausführungen wie andere Tasks (siehe Abschnitt 9.4.2) oder Interrupt-Service-Routinen (siehe Abschnitt 6.1), aber auch, wie im aktuellen Fall, die Hardware die Variable ändern. Ohne `volatile` würde der optimierende Compiler den zweiten Zugriff der Folge

```
uint32_t rnd1 = *pRngDataReg;
uint32_t rnd2 = *pRngDataReg;
```

nicht zwingend durchführen müssen. Es genügt, ihn nur einmal durchzuführen, da der Wert nach dem ersten Auslesen vermeintlich bekannt ist. Die Optimierung des Compilers endet darin, dass nur ein Zugriff durchgeführt wird und damit `rnd1` denselben Wert wie `rnd2` erhält.

In den Zeilen 8–10 ist der Zugriff als `inline`-Funktion implementiert. Dem Compiler wird also vorgeschlagen, den Inhalt der Funktion an die Stelle des Aufrufs zu kopieren. Der Zugriff selbst ist eine simple Dereferenzierung des Pointers, die in einem Load-Befehl resultiert.

4.3.2 Durchführung des Zufallszahlentests

Wie am Eingang des Kapitels in Abschnitt 4.1.1 erwähnt, soll der Zufallszahlengenerator mittels χ^2-Test auf Gleichverteilung funktional geprüft

werden. Hierfür wird die Funktion `testRNG()` verwendet, mit der Erweiterung um den Zugriff auf den Zufallszahlengenerator in Listing 4.6. Wie in C üblich liefert die Funktion als Rückgabe einen Status. Dieser ist über den Enumerationstyp `Status_t` definiert. Da das Histogramm eine beliebige Größe erreichen kann, wird der Speicher dynamisch angelegt.

```
1   Status_t testRNG(uint32_t observations, uint32_t m) {
2       uint32_t* n = calloc(m, sizeof(uint32_t));
3       if (n == NULL) {
4           return Status_OutOfMemory;
5       }
6       for (int i = 0; i < observations; i += 1) {
7           n[nextRand() % m] += 1;
8       }
9       Status_t status =
10       ↪ equalDistChi2(n, m, (observations / m),
11       ↪ chiSquared(m - 1, ChiSquaredAlpha_10_percent));
12      free(n);
13      n = NULL;
14      return status;
15  }
```

Listing 4.6 Funktion zum Testen des Zufallszahlengenerators, Erweiterung von Listing 4.3

In der Schleife Zeile 6–8 werden `observations` Zufallszahlen vom RNG-Modul abgeholt und per Modulo-Operation dem Histogramm zugeteilt. Anschließend wird der χ^2-Test durchgeführt, das dynamische Array gelöscht und der Status an den Aufrufer zurückgegeben. Die Ausführungen mit `observations = 100` und 10.000.000 zeigen, dass die Verteilung der Gleichverteilung entspricht. Die grafische Auswertung beider Läufe ist in Abb. 4–19 dargestellt.

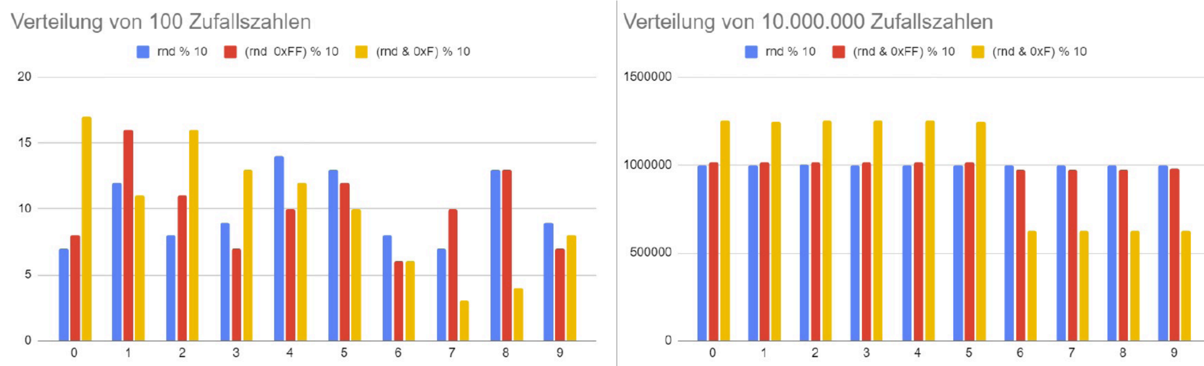

***Abb. 4–19** Verteilung von 100 (links) und 10.000.000 Zufallszahlen (rechts) mit unterschiedlicher Histogrammbildung*

Richtet man das Augenmerk auf die blauen Säulen, lässt sich erahnen, dass aber eine Schiefe vorliegt: Die Werte für die Intervalle 0 bis 5 liegen leicht über 1.000.000, die für 6 bis 9 leicht darunter. Die roten Säulen zeigen, dass das Problem stärker wird, wenn nur ein Byte der Zufallszahl verwendet wird, also in Codezeile 7 `(nextRand() & 0xFF) % m`. In diesem Fall ist die Schiefe so stark, dass der χ^2-Test für die rechte Verteilung fehlschlägt. Wird statt eines Bytes nur ein Nibble der Zufallszahl und damit (gelbe Balken) `(nextRand() & 0x0F) % m` herangezogen, schlägt der χ^2-Test für beide Verteilungen fehl.

Dies bedeutet aber nicht, dass der Zufallszahlengenerator unbrauchbare Zahlen liefert. Jedoch ist die Restklasseneinteilung ungültig, da die Zahlen den Histogrammsäulen nicht ohne Rest zuordenbar sind. Im extremen dritten Fall werden die Zahlen 0 bis 15 in die Klassen 0 bis 10 eingeteilt. Damit werden die Paare (0, 10) der Klasse 0, (1, 11) der Klasse 1 usw. zugeordnet. Die Klasse 6 erhält aber nur 7, Klasse 7 die 7, …

In der Praxis führt dieser Fehler zu leichter erratbaren Zufallszahlen, was ein kryptografisches System erheblich schwächt. Es ist ersichtlich, dass für die Implementierung kryptografischer Algorithmen eigenes Expertenwissen notwendig ist.

Auch die reine Anwendung der Algorithmen sollte nicht ohne Bedacht durchgeführt werden. Das diesbezügliche englische Standardwerk »Applied

Cryptography« [55] von Bruce Schneier und das deutsche Werk »Kryptografie« [54] sind gute Referenzen für detaillierte Informationen.

Zufall generieren

Zufall bedeutet im Allgemeinen, dass für ein Ereignis keine kausale Erklärung gefunden werden kann. In der informationstechnischen Anwendung ist aber wichtig, dass eine zufällige Zahl nicht vorausgesagt werden kann, sie also von anderen generierten Zahlen unabhängig ist. Weiters sollen die Zahlen einer definierten Verteilung entsprechen.

In der Praxis werden oft Pseudozufallszahlen verwendet. Aus einer »Seed«, einer initialen Zahl, werden fortwährend Zufallszahlen generiert. Diese deterministische Zahlenfolge kann mit gleicher Seed wiederholt werden. In PCs wird als Seed oft die aktuelle Zeit, die nicht zufällig ist, verwendet. Viele Algorithmen in der Kryptografie hängen aber von sicheren, nicht erratbaren Zufallszahlen ab. Ein Angreifer kann sonst versuchen, einen Schlüssel durch mehrfaches Ausprobieren der Möglichkeiten eines Zeitintervalls herauszufinden.

Einen Ausweg in sicheren Systemen liefern nichtdeterministische Zufallszahlengeneratoren. Diese leiten die »Entropie« aus physikalischen Effekten wie thermischem oder atmosphärischem Rauschen, radioaktivem Zerfall, schwankender Diodenspannung, ungenauen (mitunter im Produktionsprozess absichtlich verunreinigten) Taktgebern und Ähnlichem mehr ab.

Um die Güte eines Zufallszahlengenerators zu prüfen, reicht der χ^2-Test nicht aus. Hierfür werden spezielle Applikationen wie die Dieharder random number test suite (https://webhome.phy.duke.edu/~rgb/General/dieharder.php) eingesetzt.

Im laufenden Betrieb eines sicheren Systems sollte aber immer wieder geprüft werden, ob die Hardware noch vernünftige Zufallszahlen liefert oder defekt ist oder manipuliert wurde. Hierfür kommt in der Praxis der χ^2-Test zum Zug.

4.3.3 Informationen der Hersteller

Die Hersteller von Mikrocontrollern stellen eine Fülle an Materialien zur Verfügung. Die Dokumente umfassen oft mehrere tausend Seiten. Verschiedene Versionen umfangreicher APIs werden durch Application Notes und Demos ergänzt.

In der folgenden Auflistung werden die Arten der Informationsquellen beschrieben, sodass sie sinnvoll für den jeweiligen Zweck genutzt werden können.

Data Sheet Das Datenblatt listet die wesentlichen Merkmale eines konkreten Mikrocontrollers auf. Dazu zählen Speichergrößen, Gehäusetypen, Belegungen der Anschlüsse, implementierte Peripheriemodule. Außerdem werden die physikalischen und technischen Details wie minimale und maximale Versorgungsspannung, Fähigkeiten der Treiber für die Anschlüsse, mögliche Taktraten, der mögliche Temperaturbereich, maximale Kommunikationsgeschwindigkeit usw. angeführt. Damit ist dieses Dokument

wichtig bei der Auswahl des Bausteins und hauptsächlich für das Schaltungsdesign notwendig.

Reference Manual Dieses, auch »Family Guide« genannte Dokument ist die wichtigste Quelle für die embedded Entwicklung. Hier sind die Funktionalitäten beschrieben, die für eine ganze Familie von Mikrocontrollern gleich sind. Dies sind Informationen zur CPU, dem Speicherlayout, den Debug-Möglichkeiten sowie zur Funktionsweise und zu den bereitgestellten Registern der Peripherie. Das Reference Manual ist aufgrund seines Detaillierungsgrades sehr umfangreich. Es dient als Nachschlagewerk bei der Inbetriebnahme eines Peripheriemoduls.

Errata Im Laufe des Lebenszyklus einer Mikrocontrollerversion werden Implementierungsfehler und Abweichungen von Data Sheet und Reference Manual entdeckt. Diese werden in den Errata dokumentiert und, wenn möglich, wird zu jedem Problem ein Workaround beschrieben. Dieses Dokument ist eine wesentliche Informationsquelle bei Entwicklung, Test und Vertrieb von Embedded Systemen. Es sollte daher immer detailliert durchgearbeitet werden.

Application Notes Um eine Hilfestellung bei der Entwicklung zu leisten, stellen die Hersteller Ausarbeitungen von Miniprojekten zur Verfügung. Diese zeigen beispielsweise die Verwendung der Schnittstellenmodule zur Kommunikation, zur Audioverarbeitung, zum Auslesen von Sensoren und weitere. Die Verwendung der Application Notes sollte mit Bedacht geschehen, da die Qualität hier manchmal nicht auf produktionsbereitem Niveau liegt.

Development Boards und Demos Zur leichteren Einarbeitung in ein Mikrocontrollersystem bieten die Hersteller verschiedene Development Boards mit zugehöriger Demosoftware an. Im Gegensatz zu den Application Notes handelt es sich bei den Demos im Allgemeinen um größere Projekte, die mehrere Komponenten verwenden. Das in diesem Buch verwendete Board ist in Abschnitt 2.2.1 beschrieben.

Software Frameworks und APIs Damit die Ansteuerungssoftware für die Peripherie nicht selbst entwickelt werden muss, bieten die Hersteller fertige Software zum Download an. Diese »Software Frameworks«, die meist auch die Integration in ein embedded Betriebssystem beinhalten, und die Peripheral APIs sind aber oft bewusst proprietär gehalten. Aufgrund dieser Herstellerabhängigkeit eignen sie sich wenig zur Entwicklung portabler Software. In diesem Buch wird die Hersteller-API zum ESP32-C3 in Abschnitt 5.4.9 eingeführt und von da an für die Beispiele verwendet.

Ein Application Programming Interface (API) ist eine Sammlung von Routinen, Datentypen und Protokollen zum Erstellen einer Software.

Material zur CPU Zur eingesetzten CPU gibt es separates Material, das direkt vom CPU-Hersteller bezogen werden kann. Im Normalfall wird dies aber nicht benötigt, da die CPU durch die Hochsprache und den Compiler abstrahiert wird.

Compiler-Handbuch Dieses Dokument enthält Informationen zur Arbeitsweise des C/C++-Compilers, des Assemblers und des Linkers. Neben der Auflistung der Parameter für diese Werkzeuge finden auch Klarstellungen der C-Implementierung Platz. Die Breite von `enum`-Typen, `char`- und `int`-Variablen, die Byte Order, die Platzierung in Strukturen und Bitfeldern, die Parameterübergabe an Funktionen und vieles mehr sind bei der Implementierung von Embedded Systemen von Interesse.

Anhang A.1 fasst das von Espressif speziell zum ESP32-C3 zur Verfügung gestellte Material zusammen.

4.3.4 Speicherlayout der Peripherie

In der Memory Map des ESP32-C3 (siehe Abschnitt 4.1.3) ist der Adressbereich 0x6000000–0x600D0FFF für den Memory-Mapped-I/O-Zugriff auf die Peripherie vorgesehen. In diesen 836 KiB sind die Module, wie in Abb. 4–20 ersichtlich, mit Registern untergebracht.

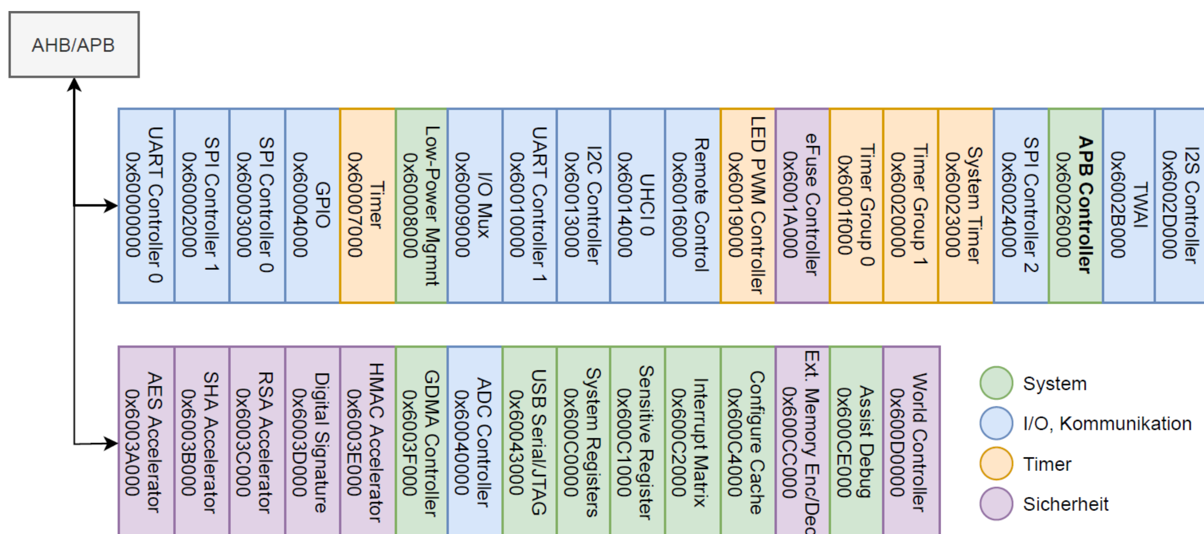

Abb. 4–20 *Speicherlayout der Peripherie*

Den Modulen sind jeweils 4 KiB Adressraum zugeordnet (mit Ausnahme der 32 KiB umfassenden Cache-Konfiguration), was den Adressdecoder vereinfacht: Der Register-Offset entspricht den unteren 12 Bit der Adresse.

In der Abbildung wurden die Module nach ihrem Einsatzzweck eingefärbt. Die Systemmodule (grün) dienen der Konfiguration des Systems und interner Module.

Blau eingefärbt sind Module für digitale und analoge Ein-/Ausgabe und für Kommunikation über I2C, SPI usw. Die orangen Timer-Module dienen der Messung und Generierung zeit- oder ereignisabhängiger Signale. Die Sicherheitsmodule (lila) schließlich implementieren kryptografische Algorithmen oder sicherheitsrelevante Mechanismen.

Der APB-Controller ist herausgehoben, da in diesem Modul auch das Register `RNG_DATA_REG` des Zufallszahlengenerators untergebracht ist, was in der vorliegenden Version des Reference Manual [26, 23] leider nicht ersichtlich ist.

4.3.5 Bits als Schalter

Im Reference Manual des ESP32-C3, Kapitel 23, ist erwähnt, dass der Zufallszahlengenerator mit Entropie aus verschiedenen Quellen versorgt werden kann und zumindest aus einer versorgt werden sollte. Es wird vorgeschlagen, den asynchronen Takt `RTC20M_CLK` einzuschalten, um dessen Metastabilität als Zufallsquelle zu nutzen. Um solches Schalten zu ermöglichen, werden die Inhalte der Register nicht zwingend als ganze Wörter interpretiert. Es ist auch oft der Fall, dass einzelne Bits als Schalter verwendet oder kleine Bitgruppen inhaltlich zusammengefasst werden.

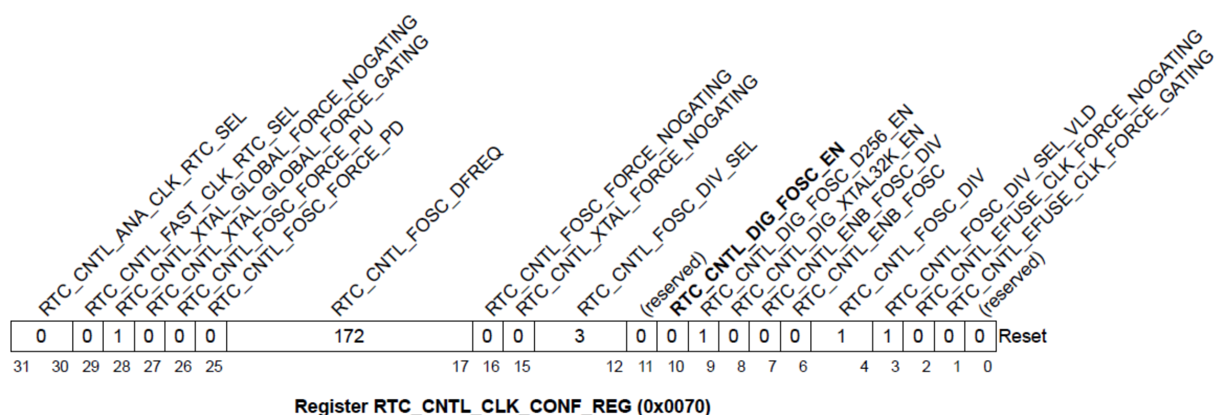

Abb. 4–21 *Ein Register mit Schaltern und Bitgruppen, wie es im Reference Manual dargestellt wird [26, Abb. 9.25]*

In Abb. 4–21 ist ein Register abgebildet, wie es im Reference Manual dargestellt wird. Die Bits 0–31 sind von rechts nach links angeordnet und einzeln oder in Gruppen benannt. Die Werte, mit denen die Gruppen bei einem System-Reset initialisiert werden, sind in der »Reset«-Zeile angegeben. Die Bedeutung der Gruppen ist im Datenblatt nachfolgend kurz erklärt. Im selben Kapitel vorangehend ist eine detaillierte Beschreibung der Funktionsweise und Verwendung derRegister und Registergruppen.

Konkret ist im Beispiel das Register `RTC_CNTL_CLK_CONF_REG` für Einstellungen der Echtzeituhr (RTC, Real-Time Clock) dargestellt [26, Abb. 9.25].

Der Schalter RTC_CNTL_DIG_FOSC_EN dient hierbei zum Ein-/Ausschalten des Taktes RTC20M_CLK.

Das herausgehobene Bit 10 muss gesetzt werden, um den Takt einzuschalten. Ein Schreiben des Bits über die Anweisung Zeile 9 in Listing 4.7 schaltet zwar den Takt ein, löscht aber alle anderen Werte auf 0, mit entsprechend schaltendem Effekt.

```
1  #define LOWPOWER_MGR_BASE 0x60008000
2  #define RTC_CNTL_CLK_CONF_REG 0x70
3
4  volatile uint32_t* pRtcCntlClkConfReg =
5       ↪ (volatile uint32_t*)
6       ↪ (LOWPOWER_MGR_BASE | RTC_CNTL_CLK_CONF_REG);
7
8  inline void switchOnRtc20MClk() {
9     *pRtcCntlClkConfReg = 0x00000400; // set Bit 10
10 }
```

Listing 4.7 *Schreiben eines Registers, analog zu Listing 4.5*

Dieses Problem tritt durch den wortweisen Zugriff auf: Es muss immer ein ganzes Wort gelesen oder geschrieben werden. Der Zugriff auf einzelne Bits ist nicht ohne Weiteres möglich.

Als Lösung bieten manche Controller, beispielsweise ARM Cortex M3/M4 mit »Bit-Banding«, einen bitweisen Zugriff auf spezielle Bereiche des Peripheriespeichers an. Hierbei wird ein Bereich des Speichers zusätzlich bitweise auf den wortweisen Speicher abgebildet, sodass jedes Bit ein eigenes Adresswort erhält. Für ein 32-Bit-Wort werden also 32 Wörter im Adressbereich benötigt.

Dieses Verfahren hat sich in der Breite nicht durchgesetzt und ist auch in RISC-V nicht implementiert.

4.4 Bitmaskierung

In der breiten Praxis wird »Bitmaskierung« zur Manipulation von einzelnen Bits und Bitgruppen eingesetzt. Beim Zugriff wird ein Wort aus einem Peripherieregister in ein CPU-Register gelesen, die notwendige Änderung an den Bits vorgenommen und das geänderte Wort final an das Peripherieregister zurück übertragen.

Die Grundlage dafür stellen die logischen Operatoren bereit, die im Folgenden besprochen werden.

4.4.1 Klassische Aussagenlogik

> *»Wenn alle Stricke reißen – häng' ich mich auf.«*
>
> JOHANN NEPOMUK NESTROY

Die klassische Aussagenlogik beschäftigt sich mit der Verknüpfung von Aussagen, denen ein Wahrheitswert (*wahr*/*true* oder *falsch*/*false*) zugeordnet wird, was wiederum in einem Wahrheitswert resultiert.

Beispiele für elementare, also nicht weiter zerlegbare Aussagen sind:

- A_1: Ein KiB sind 1024 Byte ⇒ *true*
- A_2: 5 ist durch 2 teilbar ⇒ *false*

Aussagen, die im Kontext nicht zu *wahr* oder *falsch* ausgewertet werden können, sind ungültig. »Das Wasser ist angenehm warm« ist ohne weitere Definition eine solche Aussage.

Eine Verneinung (Negation) dreht den Wahrheitswert einer Aussage über das Wort »nicht« in ihr Gegenteil um: Ein KiB sind *nicht* 1024 Byte ⇒ *false*. Man verwendet auch die Schreibweisen $NOT A_1$, $\neg A_1$ oder $\overline{A_1}$.

Die für dieses Buch wesentlichen »Junktoren« sind die Verknüpfungen *AND, OR* und *XOR*. Es sind jeweils verschiedene Symbole für diese Junktoren in Gebrauch, von denen hier die üblichen $\wedge$, $\vee$ und $\oplus$ Einsatz finden.

Tab. 4–4 *Wahrheitstabelle für AND ($\wedge$), OR ($\vee$) und XOR ($\oplus$)*

A	B	$A \wedge B$	$A \vee B$	$A \oplus B$
$false$	$false$	$false$	$false$	$false$
$false$	$true$	$false$	$true$	$true$
$true$	$false$	$false$	$true$	$true$
$true$	$true$	$true$	$true$	$false$

Die Wahrheitstabelle 4–4 definiert diese Verknüpfungen. Sie ist dabei beispielsweise so zu lesen: Sind beide Aussagen *A* und *B* *true*, so ist auch die *AND*-verknüpfte Gesamtaussage *A* ⋀ *B* *true*.

Ist aber (mindestens) eine der beiden Aussagen *false*, so ist auch die verknüpfte Aussage *false*:

$$\underbrace{Ein\ KiB\ sind\ 1024\ Byte}_{true} \wedge \underbrace{Ein\ MiB\ sind\ 1024^2\ Byte}_{true} \Rightarrow true$$

$$\underbrace{Ich\ habe\ Kleingeld\ dabei}_{true} \wedge \underbrace{Ich\ habe\ nur\ Scheine\ dabei}_{false} \Rightarrow false$$

Im Grunde ist dieses Wissen bei grundlegender Programmiererfahrung aus der Formulierung von Bedingungen bekannt und wird daher nur oberflächlich betrachtet. Die Wahrheitswerte werden von den Aussagen losgelöst in booleschen Variablen gespeichert und weiterverarbeitet. In der Programmiersprache C gibt es keinen eigenen Datentyp für Wahrheitswerte, wie beispielsweise `boolean` in Java. Stattdessen wird für die Speicherung der Basistyp `int` verwendet. *false* ist mit dem Wert 0 belegt und jeder andere Wert (positiv und negativ) kodiert *true*. Damit zwei beliebige wahre Werte miteinander verglichen werden können, wurde in C99 der Datentyp `_Bool` eingeführt, der aber weiterhin eine Integerzahl ist. Mit dem Include `stdbool.h` können die Definitionen `bool` (auf `_Bool`), `true` (auf 1) und `false` (auf 0) portabel verwendet werden. In C++ sind `bool`, `true` und `false` typisiert und Teil der Sprache.

Als logische Operatoren stehen `!` (NOT), && (AND) und || (OR) zur Verfügung. Bei der Verarbeitung logischer Ausdrücke wendet der Compiler die Kurzschlussauswertung an:

Der Ausdruck `if (x && isEven(x))` führt den Funktionsaufruf nur aus, wenn `x` wahr ist, also einen Wert ≠ 0 hat. Ist aber `x` = 0, kann der Ausdruck nicht mehr wahr werden, weshalb auch die Funktion `isEven()` nicht aufgerufen wird. Die Kurzschlussauswertung spart zwar Rechenzeit, macht das Programmieren von Seiteneffekten, also der Änderung globaler Variablen in Funktionen, aber noch gefährlicher.

4.4.2 Bitweise Operatoren in C

Das Zweierkomplement wird zu Darstellung negativer Integerzahlen benutzt. Mit ihm sind Subtraktionen durch Additionen rechenbar [68].

Die Programmiersprache C unterstützt neben den logischen auch bitweise Operatoren für Integer-Datentypen. Die Operanden werden bei diesen Operationen nicht als *true* und *false* interpretiert, sondern jedes Bit separat. Bei einer Verknüpfung der beiden Variablen `a` und `b` wird Bit a_0 mit Bit b_0, Bit a_1 mit Bit b_1, allgemein Bit a_i mit Bit b_i verknüpft. Tabelle 4–5 stellt die in C verfügbaren Verknüpfungsoperatoren AND(&), OR(|), XOR(^) und NOT bzw. Komplement(~) anhand eines Beispiels mit arbiträren 8-Bit-Zahlen dar.

Zusätzlich sind die Shift-Operatoren < < und > > definiert. `x < < n` schiebt den Inhalt von `x` um `n` Stellen nach links, 0-Bits nachschiebend. Diese Shift-Operation entspricht damit mathematisch für positive und negative Zahlen in Zweierkomplementdarstellung einer Multiplikation mit 2^n. Ein Nach-Links-Schieben der 16-Bit-Zahl 589_{dez} (0x024D) um 5 ist beispielsweise

$$0000001001001101_{bin} \Rightarrow 0100100110100000_{bin}$$

und ergibt 188480_{dez} (0x49A0). Bits, die links aus dem Stellenbereich der Variable geschoben werden, gehen verloren. Solche Overflows werden in C generell nicht erkannt oder behandelt.

Tab. 4–5 *Beispiel der Verknüpfung von 8-Bit-Zahlen für bitweises AND(&), OR(|), XOR(^) und NOT(~)*

Statement	Bitmuster							
	b_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
uint8_t a = 0xCA	1	1	0	0	1	0	1	0
uint8_t b = 0x43	0	1	0	0	0	0	1	1
a & b	0	1	0	0	0	0	1	0
a | b	1	1	0	0	1	0	1	1
a^b	1	0	0	0	1	0	0	1
~a	0	0	1	1	0	1	0	1

Analog entspricht ein Schieben nach rechts x > > n einer Division durch 2^n. Das Verhalten für negative Zahlen ist aber compilerabhängig. Üblicherweise wird bei einer vorzeichenbehafteten Zahl ein arithmetisches Shift implementiert. Dabei wird das höchstwertige Bit nachgeshiftet. Bei vorzeichenlosen Datentypen wird ein logisches Shift durchgeführt, also der Wert 0 links eingeschoben.

Handelt es sich bei der Zahl 0x8324 beispielsweise um die vorzeichenbehaftete Zahl -31964_{dez}, die um 5 Bit nach rechts geschoben wird, also

$1000001100100100_{bin} \Rightarrow 1111110000011001_{bin}$

resultiert dies in der Zahl -999_{dez} (0xFC19). Da es sich um eine Ganzzahldivision handelt, wird hinter dem Komma nicht gerundet, sondern abgeschnitten. Wird dieselbe Zahl 0x8324 vorzeichenlos gespeichert, ist der Wert 33572_{dez}, und das Schieben

$1000001100100100_{bin} \Rightarrow 0000010000011001_{bin}$

resultiert in 1049_{dez} (0x0419) und damit wieder in einer korrekten Division.

4.4.3 Bitmaskierung

Bei der näheren Betrachtung der Tabelle 4–5 fällt auf, dass bei der AND-Operation diejenigen Bits im Ergebnis gesetzt sind, die auch in beiden Operanden gesetzt waren. Alle anderen Bits sind gelöscht.

Außerdem ist ersichtlich, dass im Falle der OR-Operation diejenigen Bits gesetzt sind, die in einem der beiden Operanden gesetzt sind. Nur Bits, die in beiden Operanden 0 sind, sind auch im Ergebnis gelöscht.

Bits setzen per OR Bei der Bitmaskierung werden diese Operationen auf eine Maske und die Zieldaten angewendet. Da das 10. Bit im `RtcCntlClkConf Register` gesetzt werden soll, um den besprochenen Takt einzuschalten, ohne die anderen Bits zu verändern, wird die Maske

$$0000010000000000_{bin} = 0x0400$$

mit dem Ziel ODER-verknüpft:

```
*pRtcCntlClkConfReg |= 0x00000400 // set Bit 10
```

Der Leserlichkeit halber sollte die Maske nicht direkt als Zahl angegeben oder definiert werden, sondern über den Shift-Operator erzeugt werden:

```
#define RTC_CNTL_DIG_FOSC_EN_BIT  10
#define RTC_CNTL_DIG_FOSC_EN
    ↪       (0x1 << RTC_CNTL_DIG_FOSC_EN_BIT)
```

Auf diese Weise ist der Code zum einen leserlicher und zum anderen besser wartbar, da es genügt, die Bitdefinitionen anzupassen, anstatt die Zahlen des gesamten Codes durchzusehen und gegebenenfalls zu ändern. Das Modul `esp_bit_defs.h` definiert das Makro `BIT(nr)`, das den Shift implementiert, sowie `BIT0` (0x00000001), `BIT1` (0x00000002), ... zur einfachen Verwendung.

Gerade für Neulinge ist der Registerzugriff über Pointer ungewohnt, weshalb die Möglichkeit, die Dereferenzierung in der Registerdefinition unterzubringen, in der Praxis oft anzutreffen ist:

```
#define rtcCntlClkConfReg
    ↪ *((volatile uint32_t*)
    ↪ (LOWPOWER_MGR_BASE | RTC_CNTL_CLK_CONF_REG))
```

Mit dieser Definition wird keine globale Variable für den Registerzugriff angelegt, was in diesem Fall auch unnötig ist. Die Variable ist dann vorteilhaft, wenn die Zieladresse des Zugriffs dynamisch geändert werden soll. In Abschnitt 5.4.8 wird

gezeigt, wie I/O-Module über Strukturpointer angesprochen und an Funktionen übergeben werden können. Dann zeigt sich auch die Mächtigkeit von Memory-Mapped I/O.

Mit obigen Definitionen schaltet das folgende Statement den Takt zur Generierung der Entropie für den Zufallszahlengenerator ein.

```
rtcCntlClkConfReg |= RTC_CNTL_DIG_FOSC_EN
```

Bits löschen per AND Analog zum Setzen der Bits können auch Bits gelöscht werden. Da die Bits gelöscht werden, die in der Maske 0 sind, muss die Maske bitweise invertiert werden.

Die Zeile

```
rtcCntlClkConfReg &= ~RTC_CNTL_DIG_FOSC_EN
```

löscht das Bit wiederum, ohne die anderen Bits zu beeinflussen. Der Takt wird somit wieder abgeschaltet.

Bits umschalten per XOR Seltener kommt der XOR-Operator zum Einsatz. Seine Anwendung schaltet die in der Maske gesetzten Bits um. Man spricht hier neudeutsch von »toggeln«, aus 0 wird 1, aus 1 wird 0. Verwendung findet dieses Toggeln beispielsweise bei der Implementierung des Blinkens einer LED.

Read-Modify-Write

Aufgrund der modernen Load-Store-Architektur erfolgt der ändernde Registerzugriff in drei Schritten, die in Abb. 4–22 dargestellt sind. In *1. Read* wird das Peripherieregister mit einem Load-Befehl über das Bussystem in ein Prozessorregister geladen. Im Schritt *2. Modify* wird das Register lokal in der CPU verändert. Schließlich wird in *3. Write* das geänderte Prozessorregister per Store-Befehl auf das Peripherieregister übertragen.

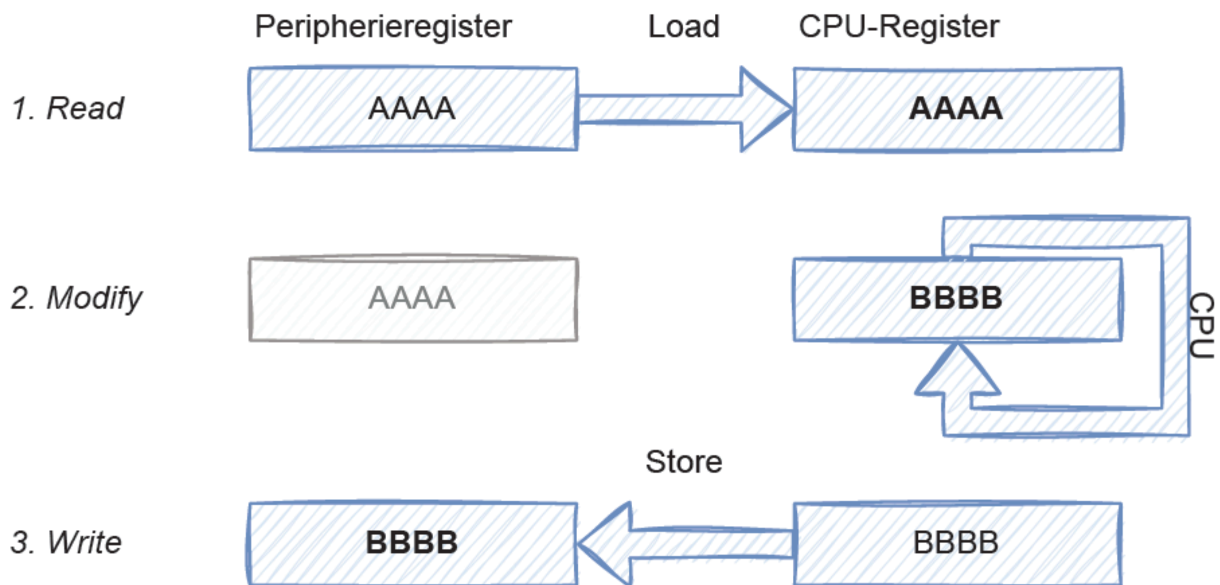

Abb. 4–22 *Das Ändern erfolgt in den drei Schritten Read, Modify, Write.*

Im Normalfall stellt dies kein Problem dar. Wenn es allerdings auf niedrigste Latenz beim Schalten ankommt, kann die Verzögerung problematisch sein. Ebenso kann es problematisch sein, wenn sich der Status eines Registers in der Zeit zwischen Auslesen und Schreiben extern ändert. Die externe Änderung würde durch die Inkonsistenz verloren gehen.

Für diese Fälle stellen manche Peripheriemodule, wie GPIO (General Purpose Input/Output, siehe Abschnitt 5.4.6), spezielle Register zur Verfügung, die die Maskierung selbst übernehmen. In einem Einschaltregister beispielsweise wird beim Setzen eine OR-Verknüpfung durchgeführt. Damit werden dann alle Ausgänge mit gesetzten Bits eingeschaltet, während die anderen unverändert bleiben. Somit ist bei dieser Registerart nur ein Schreiben notwendig.

Atomare Prozessorinstruktionen

In ISAs werden eigene RMV-Befehle definiert, die Read-Modify-Write-Zyklen atomar, also ohne Unterbrechung, durchführen können. Diese Befehle werden in der Systemprogrammierung zur Inter-Prozess-Kommunikation beispielsweise in Semaphoren eingesetzt.

Für die RISC-V ISA existiert die Erweiterung »A« (Atomic Instructions), die atomare Befehle für Swap-, Add-, And-, Or-, Max- und Min-Berechnungen bereitstellt. Diese Erweiterung ist im ESP32-C3 nicht implementiert.

Ändern mehrerer Bits Wenn mehrere Bits gesetzt werden sollen, ist es sinnvoll, die Änderungen gleichzeitig auf das Peripherieregister durchzuführen, anstatt mehrere Read-Modify-Write-Zyklen vorzunehmen.

Das gleichzeitige Setzen oder Löschen mehrerer Bits erfolgt durch OR-Verknüpfung der jeweiligen Bits, beispielsweise:

```
REG |= (BIT5 | BIT9 | BIT28); // Bits 5,9,28 setzen
REG &= (BIT5 | BIT9 | BIT31); // Bits 5,9,28 löschen
REG ^= (BIT5 | BIT9 | BIT31); // Bits 5,9,28 toggeln
```

Die Peripherieregister enthalten nicht nur 1/0-Schalter, sondern auch Werte, die mehrere Bits umfassen können. Das `RtcCntlClkConf`-Register in Abb. 4–21 fasst beispielsweise die 8 Bits 17–24 als `RTC_CNTL_FOSC_DFREQ` zum Einstellen der Taktgeschwindigkeit zusammen.

Möchte man diesen Wert von 172 (10101100_{bin}) auf 177 (10110001_{bin}) ändern, ohne die anderen Bits zu beeinflussen, müssen die 8 Bits erst gelöscht werden, bevor der neue Wert gesetzt wird. Listing 4.8 implementiert diese Vorgehensweise.

Die Definition in Zeile 3 generiert eine Maske, die die Bits 17–24 gesetzt hat. Dazu werden die acht gesetzten Bits in `MASK_8BIT` erzeugt, indem erst das Bitmuster 100000000_{bin} durch `1 < < 8` und durch Subtraktion von 1 das Muster 11111111_{bin} erzeugt wird. Dieses Muster wird in Zeile 3 um 17 Bit nach links geschoben, um die Maske $00000001111111100000000000000000_{bin}$ zu erhalten.

In der Funktion `setRtcFOscDFreq()` wird das Peripherieregister in die lokale Variable `reg` gelesen. In Zeile 7 werden die acht Bit des Zielwerts auf 0 gesetzt, ohne die anderen Bits zu beeinflussen. In der folgenden Zeile 8 wird der Zielwert an die richtige Stelle im Bitmuster geschoben und in `reg` per *OR*-Verknüpfung gesetzt. Abschließend wird das Peripherieregister mit dem geänderten Wert beschrieben.

```
1  #define RTC_CNTL_FOSC_DFREQ_BIT 17
2  #define MASK_8BIT ((1 << 8) - 1)
3  #define RTC_CNTL_FOSC_DFREQ_MASK
4      ↪ (MASK_8BIT << RTC_CNTL_FOSC_DFREQ_BIT)
5  void setRtcFOscDFreq(uint8_t dfreq) {
6     uint32_t reg = rtcCntlClkConfReg;
```

```
 7      reg &= ~RTC_CNTL_FOSC_DFREQ_MASK;
 8      reg |= (dfreq << RTC_CNTL_FOSC_DFREQ_BIT);
 9      rtcCntlClkConfReg = reg;
10  }
```

Listing 4.8 *Ändern des Wertes in Bits 17–24 per Maskierungsoperation*

Bitmaskierung ist in der Programmiersprache C nicht der einzige Weg zur bitweisen Manipulation. Auf die Verwendung von »Bitfeldern« mit den Vor- und Nachteilen wird in Abschnitt 5.4.7 näher eingegangen.

4.5 Zusammenfassung

Im ersten Teil wurden die Grundlagen der Architektur und Verwendung von Mikrocontrollern besprochen. Neben dem Aufbau der CPU, ihrer ISA und der Funktionsweise spielt die Programmierung in C eine tragende Rolle. Die Besprechung des Bussystems mit dem Zugriff auf die Peripherie über Memory-Mapped I/O und Einzelbitzugriff per Bitmaskierung rundet die Grundlagen ab.

Im nächsten Teil wird auf die einzelnen Peripheriemodule, die physikalische Interaktion mit der Umwelt sowie Programmiermodelle, die hier zum Einsatz kommen, näher eingegangen.

Index

A

ABI, Application Binary Interface 65
Absoluter Sprung 36
Abtastrate 222
Abtasttheorem 223
Abtastung 222
Active-high 164
Active-low 164
ADC, Analog-to-Digital Converter 224
Adressbus 79
Adressraum 79
ADT, Abstrakter Datentyp 232
AHB, Advanced High-Performance Bus 82
Aliasing 223
Aligned Zugriff 32
Alignment 98
ALU, Arithmetic Logic Unit 30
AMBA, Advanced Microcontroller Bus Architecture 82
Analog 221
AND 120
APB, Advanced Peripheral Bus 82
API, Application Programming Interface 115
Application Notes 115
Applikationsschicht 186, 288
Arbeit 161
Arbeitsspeicher 82
Arithmetische Datentypen in C 25
ARP, Alert Response Protocol 205

Assembler 26
Asynchron 171
Asynchrone serielle Schnittstelle 215
Atomar 180
Atomare Prozessorinstruktionen 124
Atomizität 180
Aufruf einer ISR 174

B

Bandpassfilter 229
Batterielebensdauer 162
Befehlssatzarchitektur 48
Befehlsspeicher 35
Berkeley Socket 290
Betriebssystem 257
Big-Endian 295
Bipolartransistor 139
Bit-Banging 208
Bitfeld 152
Bitmaskierung 118
Bits löschen per AND 123
Bits setzen per OR 122
Bits umschalten per XOR 123
Bitweise Operatoren in C 120
BLE Scanner App 313
BLE, Bluetooth Low-Energy 307
Blockschaltbild 15
Bluetooth 307
Branch Prediction 47
Breadboard 138
Brown-Out Detection 135
Bussystem 74, 78
Busy Loop 148, 167
Byte Order 295

C

Cache, direct-mapped 101

Cache, Ersetzungsstrategie 104
Cache, ESP32-C3 105
Cache, fully associative 103
Cache, Konsistenz 104
Cache, set-associative 103
Cacheorganisation 101
Caching 98
Call-by-Reference 63
Call-by-Value 63
CAN, Controller Area Network 219
Canary Value 264
Capture Mode 242
Chi-Quadrat(χ^2)-Test 76
CISC, Complex Instruction Set Computer 48
Clock Gating 314
Clock Stretching 201
Cloud 303
CMOS-Technologie 142
CMSIS, Common Microcontroller Software Interface Standard 156
CoAP, Constrained Application Protocol 303
Codesegment 89
Compare Mode 242
Compile-Zeit 17
Compiler-Handbuch 116
Control Hazard 47
Control Unit 36
CPU, Central Processing Unit 23
CRC, Cyclic Redundancy Check 301
Critical Section 180
Cross-Platform Development 17
CSR, Control and Status Register Instructions 59

D

DAC, Digital-to-Analog Converter 225
Dangling Pointer 94
Data Hazard 47

Data Sheet 114
datapath 38
Datenblatt 114
Datenbus 79
Datenpfad 38
Datensegment 89
Datenspeicher 32
DC-Offset 236
Deadlock 268, 269
Deep-sleep Mode 315
Defragmentierung 96
DER, Distinguished Encoding Rules 296
Development Board 13, 115
Development Toolchain 16
DFT, Diskrete Fourier-Transformation 237
Digital 221
Diode 135
Direkte Ansteuerung 151
Disassembly 25
DMA, Direct Memory Access 209
DMA-Controller 75
DMX, Digital Multiplex 219
Dominanter Pegel 146
DRAM, Dynamic RAM 83
DSP, Digital Signal Processor 229
Duty Cycle 246
Dynamisch vs. statisch 17
Dynamischer Callback 188
Dynamischer Speicher 92
Dynamisches Speichermanagement 94

E

Echtzeitbetriebssystem 257
Edge-triggered Interrupt 184
EEPROM, Electrically Erasable PROM 86
Eingebettetes System 10

Elektrische Kapazität 159
Embedded System 10
Entwicklungsboard 13
EPROM, Erasable PROM 85
Errata 115
ESP Privilege Separation 280
ESP-IDF 18
ESP32-C3-DevKitM-1 13, 213
Event Handler 180
Event Loop Library 276
Exception 173
Exception Handler 180
Exception Handling 172
F
Feldeffekttransistor 140
Fensterfunktion 238
FFT, Fast-Fourier-Transformation 239
Filesystem 209
FIR-Filter 229
FlexiPlot 228
Fließkommazahlen 25
Flusskontrolle 217
Fragmentierung, extern 96
Fragmentierung, intern 96
FRAM, Ferroelectric RAM 88
FreeRTOS 258
Frequenzauflösung 238
Function Pointer 188, 190, 233
Funktionsaufruf 63
G
GAP, Generic Access Profile 309
Gateway 287
GATT, Generic Attribute Profile 310
Gerätetreiber 281
Git 193

Gleichanteil 236
Gleitender Mittelwert 230
Glitch 165
GPIO, General Purpose Input/Output 144
Grenzfrequenz 223
H
HAL, Hardware Abstraction Layer 186
Halbduplex 207
Halbleiter 135
Hann-Window 239
Harvard-Architektur 35
Hauptspeicher 82
Hazard 46
Header 298
Heap 94
Hexadezimalzahlen 53
Histogramm 106
Hochpassfilter 229
Host-System 12
Hysterese 164
I
I^2C, Inter-Integrated Circuit Protokoll 200
I^2S 219
IC, Integrated Circuit 23, 48
IDE, Integrated Development Environment 18
IEC-Präfix 31
IIR-Filter 229
Implementierung 17
Inline Assembler 60
Inlining 70
Instruction Memory 90
Instrumentierung 62
Integrierte Entwicklungsumgebung 18
Internetschicht 286
Interrupt 173

Intrinsische Funktion 62
Inverse Ansteuerung 151
IoT, Internet of Things 285
IP-Protokollstapel 286
IRQ, Interrupt Request 172
ISA, Instruction Set Architecture 48
ISD, In-System Debugging 12, 18
ISR, Interrupt Service Routine 173, 180
J
JSON, JavaScript Object Notation 297
JTAG, Join Test Action Group 13
K
Kalenderzeit 244
Komponentenmodell des ESP-IDF 191
Kondensator 159
Konfiguration 200
Konstantengenerator 37
Kooperatives Multitasking 255
Kritische Region 180, 266, 267
Kurzschlussauswertung 120
L
Laufzeit 17
LCD, Liquid Crystal Display 197
Leck(Leakage)-Effekt 238
LED, Light-Emitting Diode 136
Leistung 161
Level Conversion 248
Level-triggered Interrupt 184
Light-sleep Mode 315
Linker 90, 107
Linker Script 107
Little-Endian 295
Load/Store-Architektur 34
Lokalität, räumlich 99
Lokalität, zeitlich 99

Lost update problem 265
M
MAC, Media Access Control 286
Machine Mode 279
Masken-ROM 84
Matrixdisplay 196
MAX30102 203, 235
Memory Map 79
Memory Map, ESP32-C3 81
Memory-Mapped I/O 111, 155
Message Buffer 274
Message-Queue 272
Mikrocontroller 73
Mikroprozessor 23
Misaligned Zugriff 32
Modbus 219
MOSFET, Metal Oxid Semiconductor Field Effect Transistor 140
MQTT, Message Queueing Telemetry Transport 304
Multimeter 138
Multiplexer 38
Multitasking 255
Mutex 275
N
Nebenläufigkeit 179, 265
Netzzugangsschicht 286
NMOS Low-Side Switch 142
NOT 120
Notification 275
O
Ohmscher Widerstand 134
Ohmsches Gesetz 134
OLED, Organic LED 197
Open-Drain 145
Optimierung 68
Optimierung, Compiler 69

OR 120
Oszilloskop 158, 167
Oversampling 165

P

Pad 144
Padding 65, 97, 302
Paritätsbit 216
PC, Program Counter 35
PCM, Puls-Code-Modulation 225
Performance 58
Peripherie 74, 109, 129
Peripheriemodul 110
Pin-Multiplexing 149
Pipeline 44
PMOS High-Side Switch 143
PMP, Physical Memory Protection 280
Polarform 238
Polling 172, 254
Polymorphie 235
Port-Mapped I/O 111
POSIX-Standard 282
Power Gating 315
Power Management 314
Power-Management-Algorithmus 316
Prellen eines Tasters 167
Prescaler 242
Priority based nested Interrupt Handling 182
Prioritätenbasiertes Scheduling 276
Prioritätsinversion 278
Prioritätsvererbung 278
Privilege Level 279
Privileged Architecture 60
Producer/Consumer-Problem 270
Programmiersprache C 11
PROM, Programmable ROM 85

Prozess 256
Präemptives Multitasking 255
Pseudoassemblerbefehl 36
Pseudozufallszahl 114
Publish 304
Pull-Konfiguration 142
Pull-up-Widerstand 145
Pulsoximeter 130
Push-Konfiguration 143
Push-Pull 145
PWM, Pulsweitenmodulation 246

Q

QoS, Quality of Service 305
Quantisierungsfehler 225
Quarz 240
Queue 230

R

ra, Return Address Register 66
Race Condition 266
RAM, Random Access Memory 82
Read-Modify-Write 123
Reference Manual 115
Register 27, 110
Registerbank 27
Rekursion 264
Relativer Sprung 36
ReRAM, Resistive RAM 88
REST, Representational State Transfer 303
Rezessiver Pegel 146
RFU, Reserved for Future Use 153
Ringpuffer 231
RISC, Reduced Instruction Set Computer 28
RISC-V Instruction Set Architecture 49
RISC-V Integer Register 65
RISC-V Privileged Architecture 279

RISC-V-Ausnahmebehandlung 174
RNG, Random Number Generator 110
ROM, Read Only Memory 82
Round-Robin 256
Roundtrip 287
Router 286
RS-232 214
RS-232-Transceiver 216
RS-485 218
RV32I 50
RVC, Standard Extension for Compressed Instructions 55
RVM, Standard Extension for Integer Multiplication and Division 54
Rücksprung aus einer ISR 177

S

Sampling 222
Samplingrate 222
Schaltplan 133, 195, 213
Scheduler 255
Scheduling-Strategie 256
Schichtenarchitektur 185
Schmitt-Trigger 164
SD-Karte 209
Segmentdisplay 197
Selbstentladung 162
Semaphor 266
Serielle Schnittstelle 214
Serviceschicht 186
Servomotor 247
Set-/Reset-Register 152
SI-Präfix 31
Simplex 207
Sleep Mode 315
SMBus, System Management Bus 205
Software Interrupt 280
Spannung 133

Spannungsteiler 225
Speicherhierarchie 100
Speicherlayout 89
Speicherlayout der Peripherie 116
SPI, Serial Peripheral Interface 206
Spike 165
Sprungvorhersage 47
Spurious Interrupt 183
SRAM, Static RAM 83
Stack 63, 90
Stack Overflow 262
Stacküberlauf 262
Starvation 270
Statisch vs. dynamisch 17
Statischer Callback 187
Steckplatine 138
Steuerwerk 36
Stream Buffer 274
Strom 132
Strom-Spannung-Kennlinie 136
Subscribe 304
Superskalare Architektur 47
Supervisor Mode 280
System Call 280
Systick 178

T

Target-System 12
Task 255
Task-Erzeugung 260
Task-Zustände 259
Taster anschließen 163
Tastgrad 246
TCB, Task Control Block 256
Terminalprogramm 19
Textsegment 89

Thread 255
Tiefpassfilter 223, 229
Timeout-Parameter 270
Timer-Modul 242
TLV, Tag-Length-Value 296
Transistor 139
Transmission Control Protocol 292
Transportschicht 287
TWI, Two-Wire Interface 201
U
UART, Universal Asynchronous Receiver/Transmitter 217
UDP, User Datagram Protocol 290
Unaligned Zugriff 32
UND-Gatter 38
union 154
Unixzeit 244
USB-to-Serial-Converter 215
User Mode 280
UUID, Universally Unique Identifier 310
V
Vectored Interrupt Handling 175
Verschnitt 96
Versionsverwaltung 192
volatile 111, 268
Vollduplex 207
Von-Neumann-Architektur 35
Vorwiderstand 137
W
Wahrheitstabelle 119
Watchdog-Timer 277
Wear Leveling-Algorithmus 86
Wearout 86
Webserver 306
Wettlaufsituation 266
Wi-Fi-Modul 288

Widerstandsreihe 137
Wired-AND Schaltung 145
Wired-OR Schaltung 146
Wireshark 293
WS2812B-Controller 212
X
XML, Extensible Markup Language 298
XOR-Checksumme 300
Y
Y2K38-Problem 244
Yield 255
Z
Zeitsynchronisierung 244
Zufall 114
Zufallszahlengenerator 76, 110