Gliederung der Vorlesung

1. Einleitung
2. Power Management
3. Parallele I/O
4. Serielle Schnittstelle
    1. I^2C
    2. USART
5. Timer
6. Interrupt Handling
7. Signalverarbeitung
8. Regelungen
    1. Software Interrupt
9. Leitungscodes
10. Weiterführende Themen

Markt für Mikrocontroller

ARM (Advanced Risc Machines) definiert CPU-Kerne in Cortex-Architekturen, die den Markt für Mikrocontroller dominieren.

- Cortex-A: Application (betriebssystembasierte Anwendungen)
- Cortex-R: Realtime (Echtzeitanwendungen)
- Cortex-M: Microcontroller (Cores für Mikrocontroller, Nachfolger **ARM7**)

- Mehr als 90% aller mobilen Geräte (Smartphones, Tablets, …) beinhalten einen ARM Prozessor
- 12.2.2007: 5 Milliarden ARM Prozessoren in mobilen Geräten
- Okt. 2010: 19 Milliarden ARM Prozessoren
- 2013: 50 Milliarden ARM Prozessoren Cellebrating 50 billion chips
- 2022: 230 Milliarden ARM Prozessoren
- 2024: wurden ca. 7 Milliarden ARM-Prozessoren ausgeliefert

ARM-Architekturen und -familien

Architektur	Familie(n)	Erscheinungsjahr	Takt
ARMv1	ARM1	1985	4 MHz
ARMv2	ARM2, ARM3	1986, 1989	8–25 MHz
ARMv3	ARM6, ARM7	1991, 1993	12–40 MHz
ARMv4	ARM7TDMI, ARM8,	1995	16,8–75 MHz,
	StrongARM	1997	203–206 MHz
	ARM9TDMI		180 MHz
ARMv5	ARM7EJ, ARM9E, ARM10E,	2002	104–369 MHz
	XScale		133–1250 MHz
ARMv6	ARM11,	2002	400–772 MHz
	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1		bis 200 MHz
ARMv7	ARM Cortex-M3, ARM Cortex-M4	2004	
	ARM Cortex-A (A8, A9, A5, A15, A7 und A12),	2005	bis 2 GHz
	ARM Cortex-R		
ARMv8	ARM Cortex-A53, ARM Cortex-A57	2013	3 GHz

Quelle: Wikipedia.de (ARM Architektur)

Die Familien sind nach unterschiedlichen Kriterien entworfen:

z.B. minimaler Stromverbrauch, maximale Rechenleistung, ab ARMv6 auch Multi-Core-Designs

ARM7TDMI, Cortex-M0+ und Cortex-M3 werden in dieser Veranstaltung vorgestellt.

ARMv9 wurde als Weiterentwicklung von ARMv8 2021 eingeführt.

Mikroprozessor / Mikrocontroller

- **Mikroprozessor: Mikrorechner auf einem Chip**
  - Bausteine des Prozessors auf einem Mikrochip
  - Integrierter Schaltkreis (IC) vereinigt
- **Mikrocontroller (MC): zusätzlich Peripherie integriert**
  - Für spezielle Anwendungsfälle zugeschnitten
  - Meist Steuerungs- oder Kommunikationsaufgaben
  - Anwendung oft einmal programmiert und für die Lebensdauer des Mikrocontrollers auf diesem ausgeführt
  - Anwendungsfelder sind breit gestreut
  - Oft unsichtbar in uns umgebenden Geräten verborgen
- **Oft Verzicht auf sonst übliche Prozessorbausteine bei MCs**
  - Speicherverwaltungseinheit (Memory Management Unit MMU)
  - Virtualisierungsfunktionen (wie Intel VT, AMD Virtualization)
  - Spezialbefehle für Vektorverarbeitung, Hyper-Threading, …

Anwendungsfelder

Klassische „Embedded“ Anwendungen

- im Haushalt
  - die Steuerung der Kaffeemaschine,
  - der Waschmaschine,
  - des Telefons,
  - des Staubsaugers,
  - des Fernsehers, ...
- in der KFZ Technik
  - das Motormanagement,
  - das Antiblockiersystem,
  - das Stabilitätsprogramm,
  - die Traktionskontrolle,
  - diverse Fahrassistenten, ...
- in der Automatisierung
  - das Steuern und Regeln von Prozessen,
  - das Überwachen von Prozessen,
  - das Regeln von Materialflüssen,
  - die Steuerung von Fertigungs- und Produktionsanlagen, ...

Begriffe - Embedded Systems

Ein **eingebettetes System** benötigt **Sensoren** und **Aktoren**, um mit seiner Umgebung wechselwirken zu können.

Weitere Anwendungsfelder

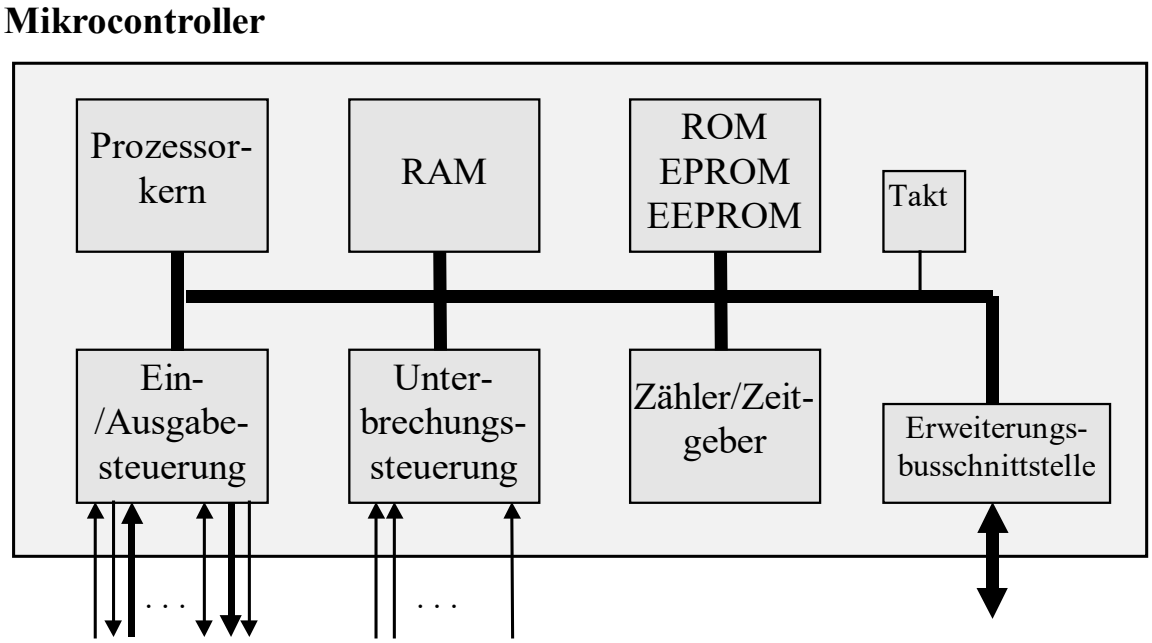
„Embedded“-Anforderungen in anderen Bereichen

- Mobile Devices
  - Embedded-Anforderungen an Rechner mit User-Front-End
  - Multitouch Displays
  - GPS-Sensoren,
  - Funkprotokolle,
  - Lage und Beschleunigung,
  - Fingerabdruck, …
- IT-Infrastruktur
  - Embedded-Anforderungen an IT-Komponenten
  - Festplatten-Controller
  - RAID-Controller
  - Netzwerk-Router
  - Netzwerkkomponenten zur Verschlüsselung/Kompression
  - Bandroboter, …

Mikrocontroller / System-on-Chip (SoC)

Abgrenzung zu Mikroprozessoren

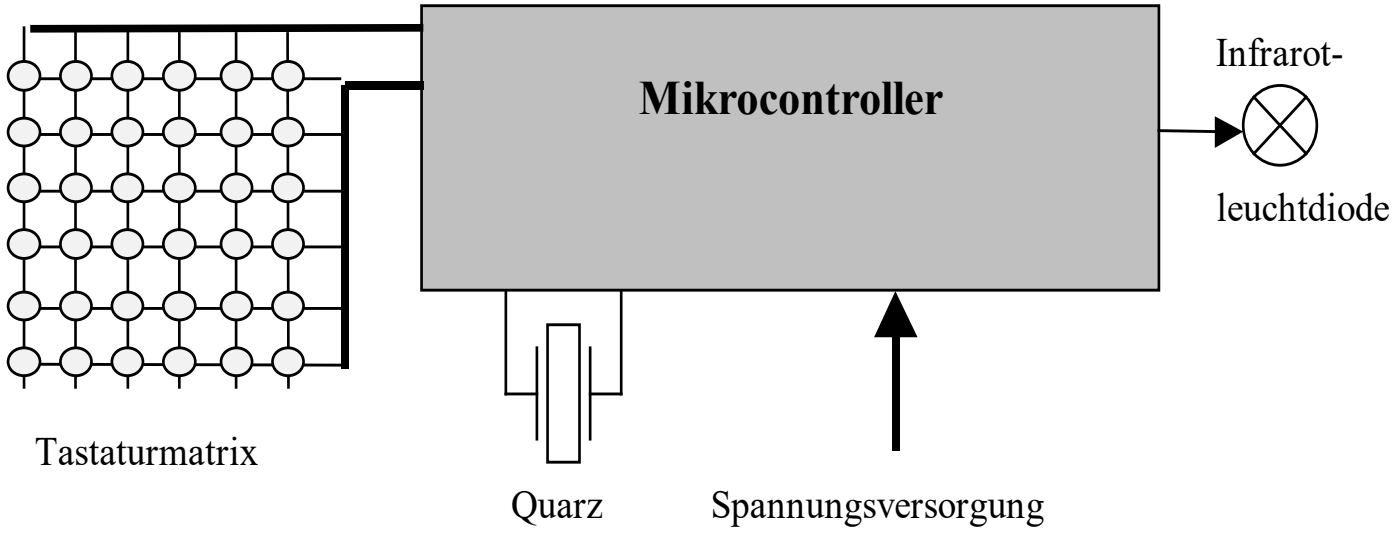
Ein **Mikrocontroller** ist ein Ein-Chip-Mikrorechner **mit** aufgabenspezifischer Peripherie.



Für komplexe Chips wird oft der Begriff **System-on-Chip** (Ein-Chip-System) verwendet. Sie integrieren alle oder fast alle Bestandteile eines Systems in sich.

Mikrocontroller - Designziele

Ziel:	Möglichst wenige externe Bausteine für eine Steuerungsaufgabe
Idealfall:	Mikrocontroller, Quarz, Stromversorgung sowie ggf. Treiber und ein Bedienfeld
Beispiel:	Fernbedienung



Mikrocontroller - Speicher

- integrierter Festwert- und Schreiblesespeicher
- Aufnahme von Programmen und Daten
- Vorteil: Einsparung von Anschlüssen und Dekodierlogik bei vollständiger interner Speicherung
- Größe und Typ des Speichers unterscheiden oft verschiedene Untertypen desselben Mikrocontrollers
- z.B. je nach Stückzahl der Anwendung unterschiedlicher Typ des Festwertspeichers FLASH (ROM, PROM, EPROM, EEPROM)

Mikrocontroller - Schnittstellen

Serielle und parallele Ein-/Ausgabekanäle

- grundlegende digitale Schnittstellen eines Mikrocontrollers
- seriell oder parallel
- synchron oder asynchron

AD/DA-Wandler

- grundlegende analoge Schnittstellen eines Mikrocontrollers
- Anschluss analoger Sensoren und Aktoren
- Auflösung und Wandlungszeit sind die wichtigsten Größen
- AD-Wandler sind häufiger anzutreffen als DA-Wandler

Mikrocontroller – Zähler und Zeitgeber

- Zähler (Counter) und Zeitgeber (Timer) sind im Echtzeitbereich ein wichtiges Hilfsmittel.
- **Echtzeiteigenschaften** entstehen, wenn ein System auf Ereignisse in einem vorgegebenen Zeitrahmen reagieren muss (**Reaktionszeit**). Bei der Zuverlässigkeit der Einhaltung von Echtzeitbedingungen unterscheidet man
  - **harte** Echtzeitbedingungen (hard real-time): garantierte Einhaltung einer Reaktionszeit
  - **weiche** Echtzeitbedingungen (soft real-time): statistische Garantie der Einhaltung einer Reaktionszeit; typischerweise wird die Zeit eingehalten. Dies ist aber nicht für alle Fälle garantiert.
- Zähler und Zeitgeber sind für eine Vielzahl unterschiedlich komplexer Anwendungen einsetzbar wie
  - Zählen von Ereignissen, Messen von Zeiten, Pulsweitenmodulation, Frequenz- oder Drehzahlmessung, …

Mikrocontroller - Selbstüberwachung

Watchdog

- „Wachhund“ zur Überwachung der Programmaktivitäten eines Mikrocontrollers
- Programm muss in regelmäßigen Abständen Lebenszeichen liefern
- Bleiben diese aus, so nimmt der Wachhund einen Fehler im Programmablauf an => **Reset**

Vergleiche: Lokführer müssen alle 30 Sekunden einen Knopf im Führerstand betätigen.

Mikrocontroller - Interrupts

Unterbrechungen (*Interrupts*)

- Unterbrechung des Programmablaufs bei Ereignissen
- Schnelle, vorhersagbare Reaktion auf Ereignisse
- Insbesondere wichtig bei Echtzeitanwendungen
- Behandlung eines Ereignisses durch eine **Interrupt-Service-Routine**
- Mikrocontroller kennen meist **externe** Unterbrechungsquellen (Eingangssignale) und **interne** Unterbrechungsquellen (Zähler, Zeitgeber, E/A-Kanäle, ...)

Mikrocontroller – DMA

Direkter Speicherzugriff - Direct Memory Access (DMA)

- Direkter Datentransfer zwischen Peripherie und Speicher ohne Beteiligung des Prozessorkerns
- Höhere Datenraten durch spezielle Transferhardware
- Entlastung des Prozessorkerns
- Prozessorkern muss lediglich die Randbedingungen des Transfers festlegen
- Erfordert eine Logik, die gleichzeitige Zugriffe durch den Prozessorkern und die Peripheriebausteine verhindert
- Meist in Mikrocontrollern gehobener Leistungsklasse zu finden

Mikrocontroller - Anforderungen

Allgemeines:

ein möglichst vollständiges Mikrocomputersystem auf einem Chip
also Mikrocontrollerkern (Core), Speicher, Peripherie

Nichtfunktionale Eigenschaften:

Kosten: maximale funktionelle Integration bei möglichst geringen Systemkosten

Performance: möglichst hohe Verarbeitungsleistung

Energieverbrauch: möglichst geringer Strombedarf insbesondere in Bereichen ohne stationäre Stromversorgung

Zuverlässigkeit: Betrieb über lange Laufzeiten in kritischen Umgebungen (Temperatur, Feuchtigkeit, elektromagnetische Verträglichkeit, Schwankungen in der Spannungsversorgung, Strahlungsresistenz, …)

Wartbarkeit: dauerhafter Betrieb häufig ohne Möglichkeit der Aktualisierung

Mikrocontroller – Designaspekte

Standardmikrocontroller (AT91M63200, SAM3X8E)

abhängig vom Design universell einsetzbar
ohne spezifisches Marktsegment

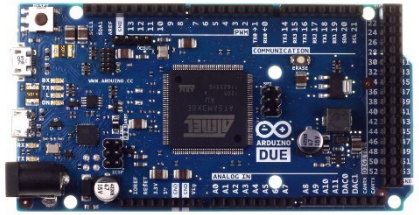
Kundenorientierte Mikrocontroller (RP2040)

in Einschränkung universell einsetzbar; zugeschnitten auf bestimmtes Marktsegment; hohe Stückzahl für konkrete Projekte

Ein MC besteht aus

dem Mikrocontrollerkern (Core): On-Chip integrierte CPU beinhaltet komplexes Steuerwerk, ALU, Register, Bussystem
und der benötigten On-Chip Peripherie:
Taktoszillator, **IO Ports, Timer**, A/D D/A Wandler, **Interrupt Controller**, Watchdog Timer, Schnittstellen (**USART**,...)

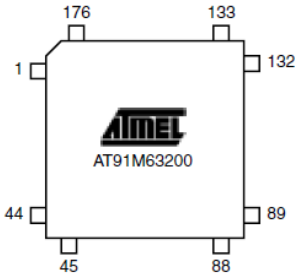
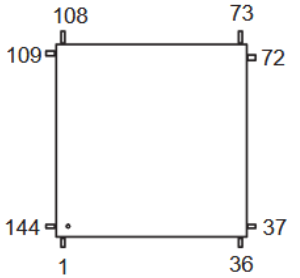
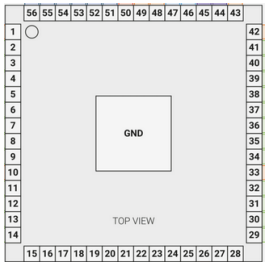
Mikrocontroller – Laborhardware

Mikrocontroller (μC)	Board	Generation Familie	Boardansicht
Atmel AT91M63200 (abgekürzt AT91)	AT91EB63 (5V) *(Folien A)*	ARMv4 ARM7TDMI	
Atmel SAM3X8E (abgekürzt SAM3X)	Arduino Due (3,3V) *(Folien B)*	ARMv7 ARM Cortex M3	
Raspberry RP2040 (abgekürzt RP2040)	Raspberry Pi Pico (3,3V) *(Folien C)*	ARMv6 ARM Cortex M0+(2x)	

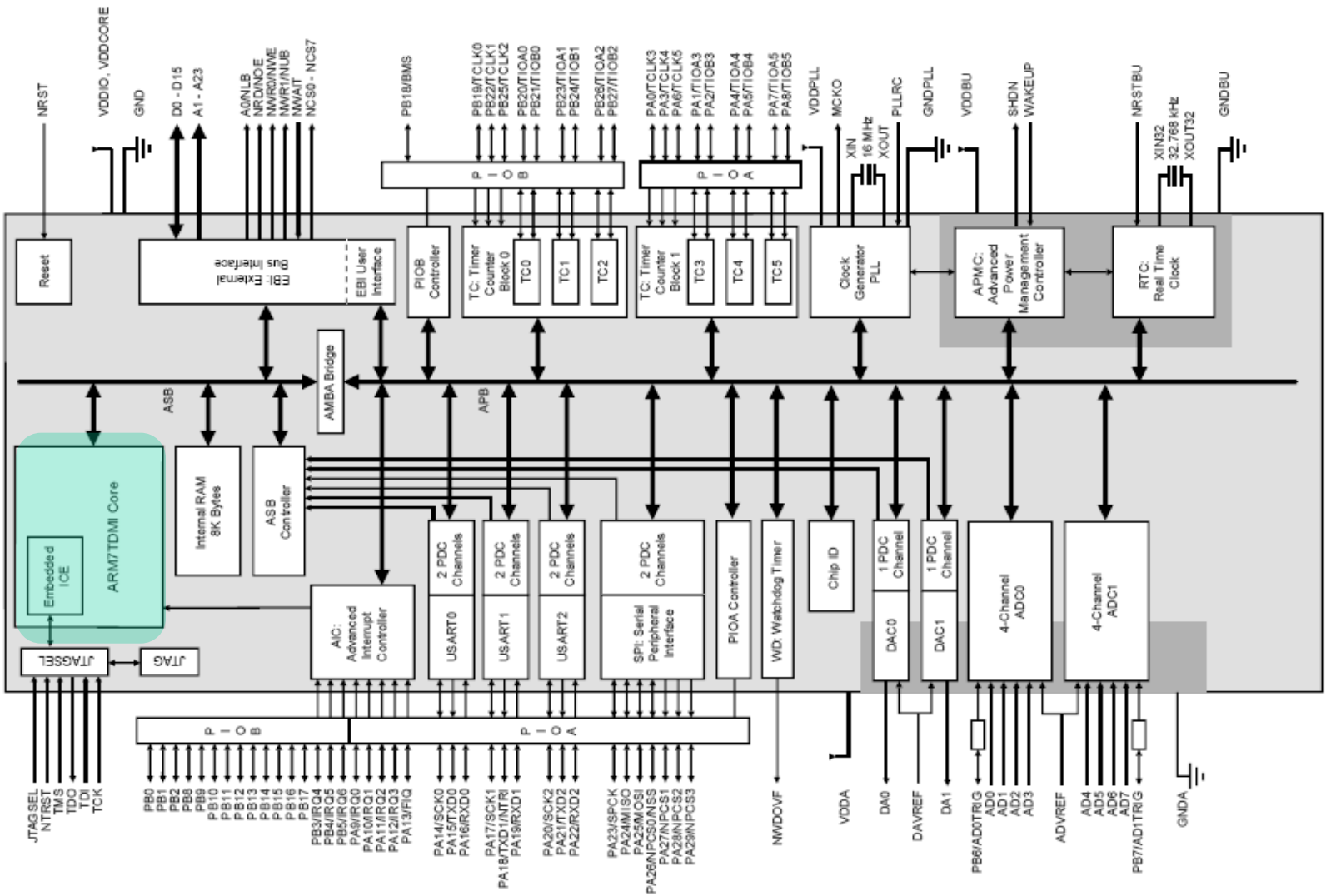
Nomenklatur in dieser Veranstaltung:

Die Namen von Mikrocontroller und Board werden teilweise synonym verwendet. Board-spezifische Folien sind im Dateinamen mit Buchstaben (A, B, C) oder Kombinationen (AB) gekennzeichnet. In diesem Abschnitt sind alle Boardspezifika zur Einführung und zum Vergleich enthalten.

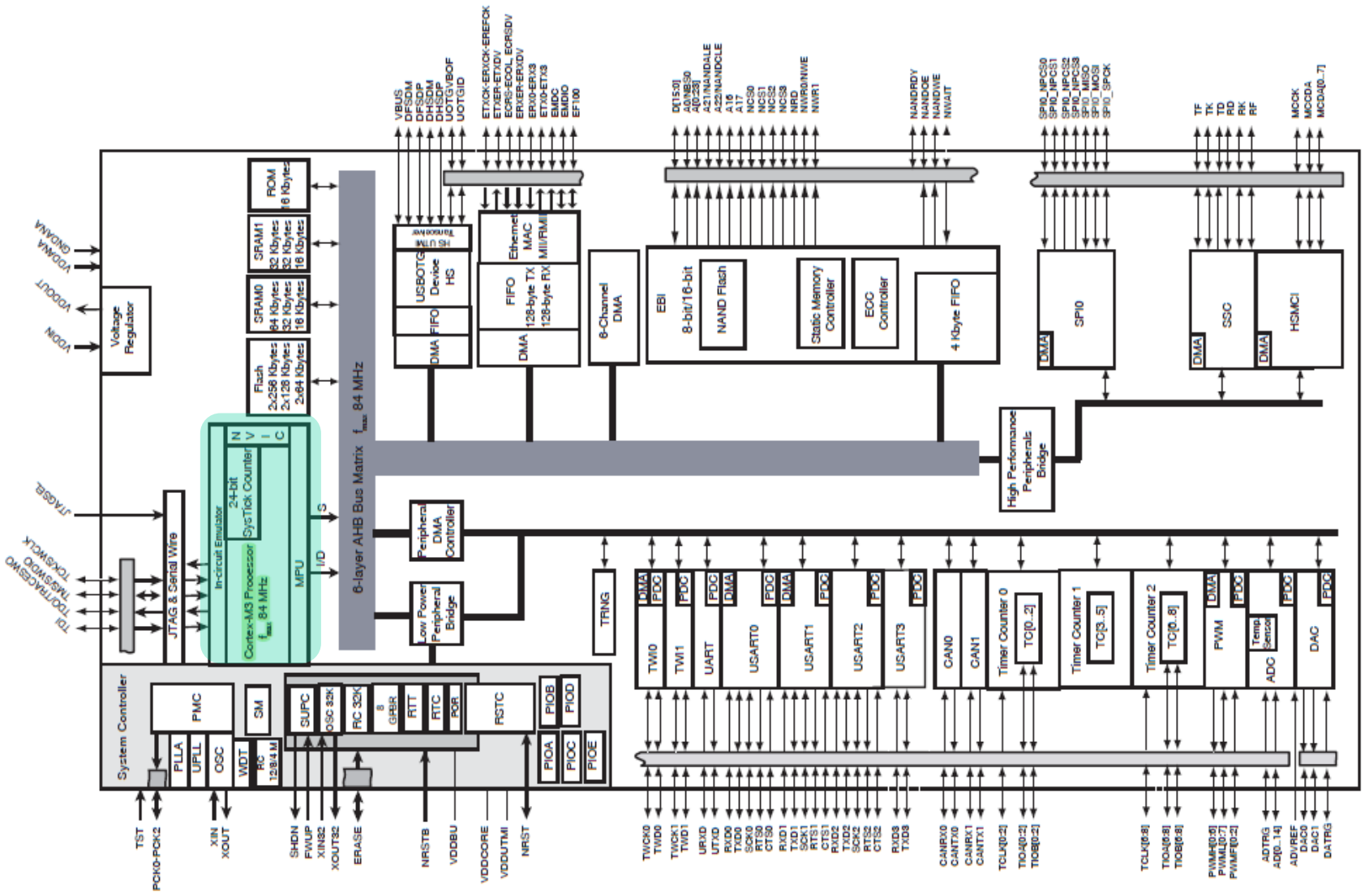
Mikrocontroller –Angaben zu den Mikrocontrollern

μC	CPU/Mem	Schnittstellen (Auswahl)	Pin Layout
AT91	1 Kern, 25 MHz (max), 32 Bit ARM und 16 Bit Thumb-1 2K RAM (intern) 256K RAM, 2M Flash (Board)	58 Portpins 2x 32-Bit PIO Ports 6x 16-Bit Timer/Counter 3x USART, SPI (Serial Peripheral Interface), Watchdog Timer	
SAM3X	1 Kern, 84 MHz (max), 16 Bit ARM Thumb-2 Instructions 96KB SRAM, 512KB Flash (intern)	63 Portpins 2x USB 2.0, 3x USART, 1x UART, 2x TWI (I^2C), 9x 32-Bit Timer/Counter, ADC/DAC, SPI, 10/100MBit Ethernet, 2 CAN Controller	
RP2040	2 Kerne, 133 MHz (max) 32 Bit ARM und 16 Bit Thumb-1 und -2	26 Portpins 2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC, 16 × PWM, USB 1.1	

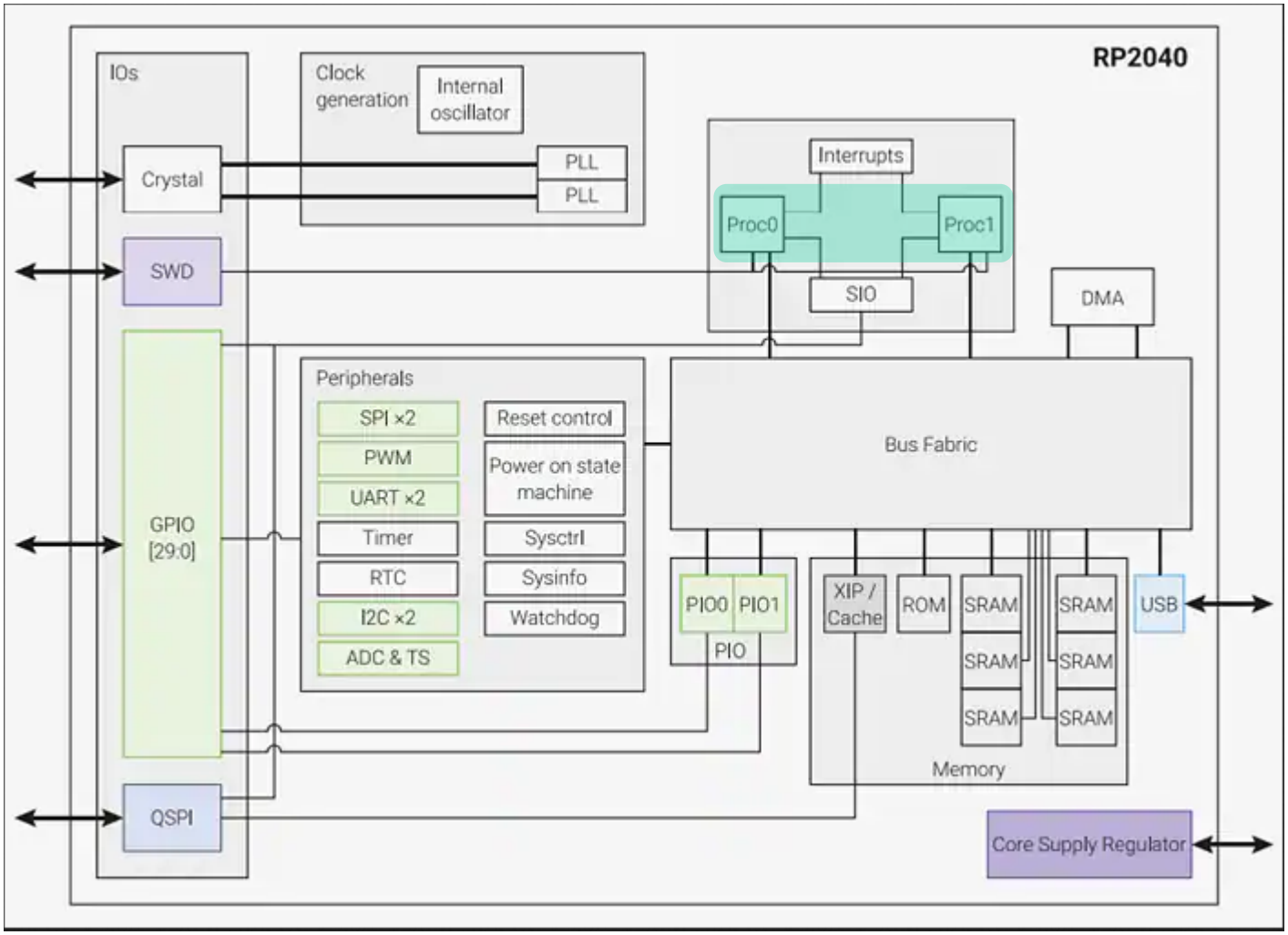
Blockdiagramm - ATMEL AT91M63200



Blockdiagramm - ATMEL SAM3X

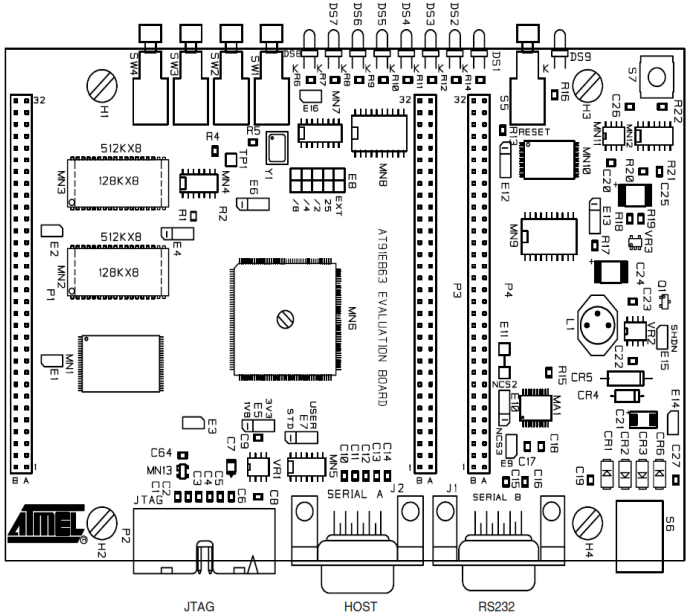


Blockdiagramm - Raspberry RP2040

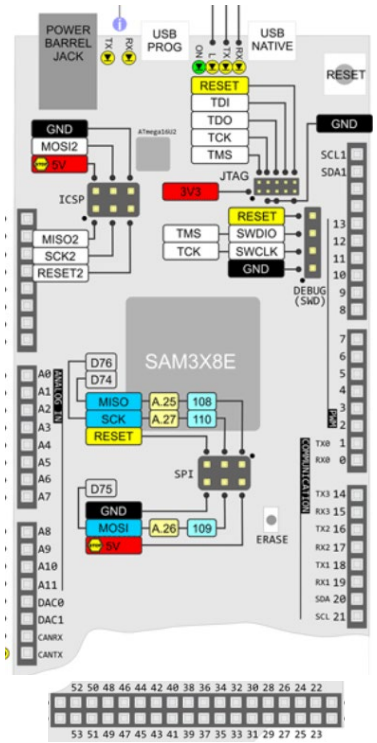


Blockdiagramm - Board Layout

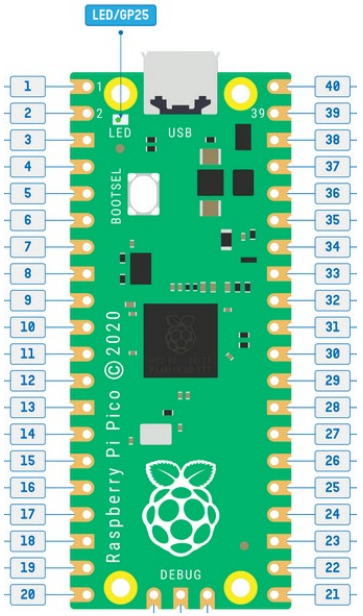
Atmel AT91EB63



Arduino Due



Raspberry Pi Pico



Programmierung der Peripherie

Der Prozessorkern muss seine Peripheriebausteine ansprechen und steuern können. Wie geht das?

Memory Mapped I/O
(z.B. ARM-Architektur)
Die Peripherie wird in den Adressraum des Hauptspeichers eingeblendet und direkt über das Lesen und Schreiben an die Adressen gesteuert. Die Inhalte an den Adressen sind die Inhalte der Register der E/A Bausteine.

Die Übersicht über alle eingeblendeten (mapped) Bausteine gibt die **Memory Map** des Mikrocontrollers.

Spezielle E/A Maschinenbefehle
(z.B. Intel x86 Architektur)
Die Peripherie wird über E/A-Ports angesprochen, die wiederum über Spezialbefehle (IN/OUT) ausgewählt und beschrieben oder gelesen werden.

Programmierbeispiel:

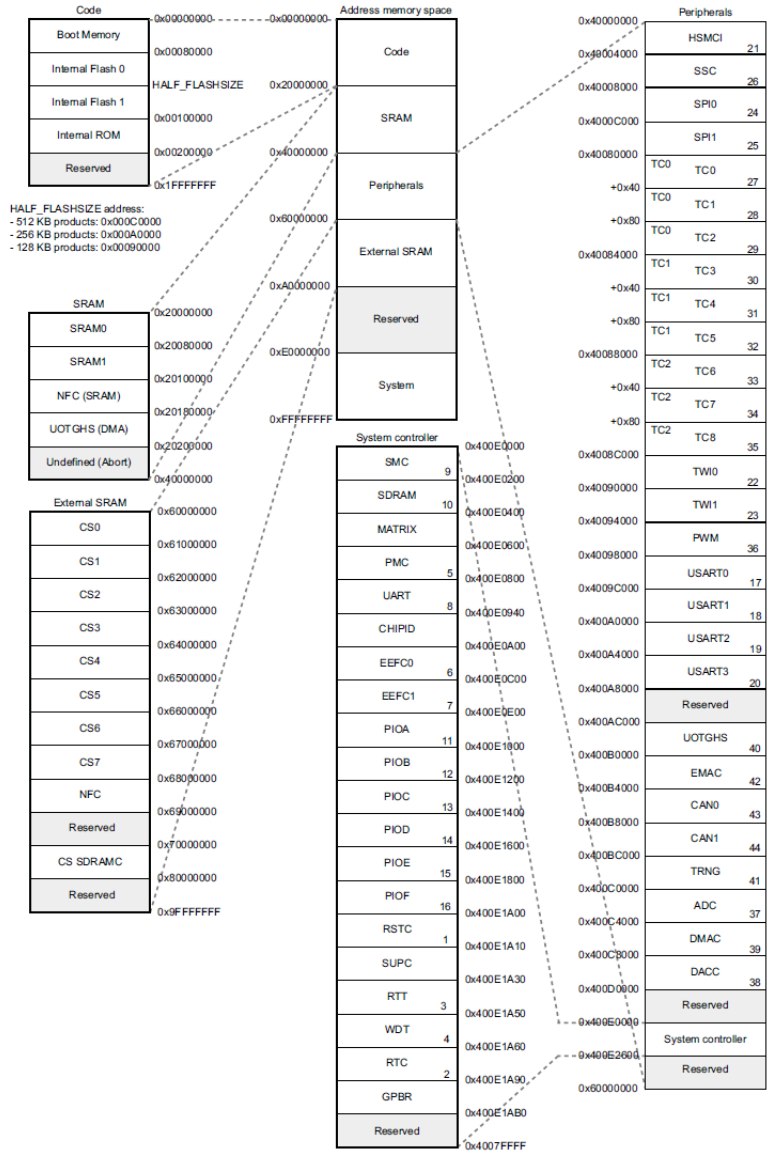
```
// Gib an dem Baustein mit der
// E/A Adresse 20H den Wert 1 aus
MOV AX, 1    // Lade den Wert 1 in
             // Register AX
OUT 20H, AX  // Spezialbefehl, lade
              // den Inhalt des
              //Registers AX
             // an den E/A Port 20H
```

AT91 Memory Map

Device	Name	Base
AIC	Advanced Interrupt Controller	0xFFFFF000
Reserved		
WD	Watchdog Timer	0xFFFF8000
PMC	Power Management Controller	0xFFFF4000
PIO	Parallel I/O Controller B	0xFFFF0000
PIO	Parallel I/O Controller A	0xFFFEC000
Reserved		
TC1	Timer/Counter Channels 3, 4, 5	0xFFFD4000
TC0	Timer/Counter Channels 0, 1, 2	0xFFFD0000
Reserved		
USART2	USART 2	0xFFFC8000
USART1	USART 1	0xFFFC4000
USART0	USART 0	0xFFFC0000
SPI	SPI	0xFFFBC000
Reserved		
SF	Special Function	0xFFF00000
Reserved		
EBI	External Bus Interface	0xFFE00000
Reserved		

SAM3X Memory Map

Figure 7-1. SAM3X/A Product Mapping



SAM3X Memory Map

Beispiele von im Praktikum verwendeten Komponenten

Device	Name	Base Address
PMC	Power Management Controller	0x400E0600
PIOA	Parallel I/O Controller A	0x400E0E00
PIOB	Parallel I/O Controller B	0x400E1000
PIOC	Parallel I/O Controller C	0x400E1200
PIOD	Parallel I/O Controller D	0x400E1400
TC0	Timer Counter Block 0 (TC0, TC1, TC2)	0x40080000
TC2	Timer Counter Block 2 (TC6, TC7, TC8)	0x40088000
USART1	Serielle Schnittstelle 1	0x4009C000
….	….	….

RP2040 Memory Map (Ausschnitt Manual)

ROM	0x00000000
XIP	0x10000000
SRAM	0x20000000
APB Peripherals	0x40000000
AHB-Lite Peripherals	0x50000000
IOPORT Registers	0xd0000000
Cortex-M0+ internal registers	0xe0000000

AHB-Lite peripherals:

DMA_BASE	0x50000000

USB has a DPRAM at its base followed by registers:

USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000

Remaining AHB-Lite peripherals:

PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
XIP_AUX_BASE	0x50400000

IOPORT Peripherals:

SIO_BASE	0xd0000000

Cortex-M0+ Internal Peripherals:

PPB_BASE	0xe0000000

APB Peripherals:

SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40004000
CLOCKS_BASE	0x40008000
RESETS_BASE	0x4000c000
PSM_BASE	0x40010000
IO_BANK0_BASE	0x40014000
IO_QSPI_BASE	0x40018000
PADS_BANK0_BASE	0x4001c000
PADS_QSPI_BASE	0x40020000
XOSC_BASE	0x40024000
PLL_SYS_BASE	0x40028000
PLL_USB_BASE	0x4002c000
BUSCTRL_BASE	0x40030000
UART0_BASE	0x40034000
UART1_BASE	0x40038000
SPI0_BASE	0x4003c000
SPI1_BASE	0x40040000
I2C0_BASE	0x40044000
I2C1_BASE	0x40048000
ADC_BASE	0x4004c000
PWM_BASE	0x40050000
TIMER_BASE	0x40054000
WATCHDOG_BASE	0x40058000
RTC_BASE	0x4005c000

Speicheraufteilung

- .text
  - legt einen Textbereich an
- .data
  - legt einen Datenbereich an
- .comm symbol, size
  - legt ein Symbol in die globale bss Section für uninitialisierte Daten
- .global
  - nimmt ein symbol in die globale Symboltabelle auf

- .word Ausdruck
  - legt einen intitialisierten Speicher an
- align #Bits
  - sorgt dafür, daß die nachfolgende Anweisung auf einer Speicherstelle steht, deren unterste #Bits 0 sind
- label:
  - legt ein Programmlabel (Marke) an
- .end
  - das Ende des Programms

Sichtbarkeit von HW-Eigenschaften in C

- Zusammengesetzte Datenstrukturen
  - Alignment der Elemente auf Zugriffsgrenzen
- Deklaration von Variablen als **`volatile`**
  - engl. flüchtig, beweglich
  - Speicherinhalte, die sich ohne Einfluss des Prozessors verändern
- Deklaration von Variablen als **`static`**
  - Zugreifbarkeit wie bei einer globalen Variable aber mit Reduktion der Sichtbarkeit auf Modulgrenzen (bei C++ Einschränkung der Sichtbarkeit auf Klasse)
- Schiebeoperationen
  - Links-Shift: **`1 << 14`** schiebt eine „1“ auf das 14. Bit in einem Register
  - Rechts-Shift **`0x0F000000 >> 24`** wird zu 0x0000000F

Sichtbarkeit von HW-Eigenschaften in C

- Arithmetische Operatoren für UND / ODER
  - Die Operatoren `&|` verknüpfen Operanden **bitweise**
  - Der Operator ~ invertiert Bitmuster **bitweise**
    (anders als die logischen Operatoren &&, || und !)
  - Der Operator ^ implementiert das **bitweise** XOR
- Parameterübergabe an Funktionen
  - C kennt „Call by value“ und „Call by reference“
    -> value = Wert
    -> reference = Zeiger = Adresse
  - ARM Procedure Call Standard (APCS) definiert, wie Daten zwischen aufrufender und aufgerufener Funktion ausgetauscht werden
  - Möglichkeit der Übergabe per Register und Stack
  - Rückgabewerte mittels **`return`**

Hardwarenahe Deklarationen in C

- Hardwarebausteine, die in einen Speicherbereich „gemapped“ sind, werden als zusammengesetzter Datentyp (struct) deklariert.
- Adressen werden als Konstanten bekannt gemacht.

```
// Typische Inhalte von Deklarationsdateien (Header-Files)
// Vergleichen Sie mit den Inhalten des Verzeichnisses h
// im Praktikum

// Variablentyp fuer Register mit 32 Bit Breite bei einer
// 32-Bit-Prozessorarchitektur
typedef volatile unsigned int register32;

// Deklaration einer Struktur, die einer Register-Map entspricht
typedef struct {
        register32 REGISTER0;
        register32 REGISTER1;
} structBaustein;
```

```
// Praeprozessordirektive fuer eine Konstante, die die
// absolute Basisadresse eines Bausteins darstellt
#DEFINE baustein_basisadresse ((structBaustein*) 0x00FF0000)
```

Sichtbarkeit von HW-Eigenschaften in C

- Zeiger und Adressen
  - Der Begriff des Zeigers (engl. Pointer) in C entspricht einer Speicheradresse
- Beispiele für die Verwendung von Adressen

```
// Direkte Zuweisung benötigt Cast-Operator
volatile unsigned int *reg = (???*) 0xFF000000;
```

-
```
// mit Deklarationsdateien
structBaustein *bausteinA = baustein_basisadresse;
bausteinA->REGISTER0 = 0;
```

- Einzelne Bits setzen

```
// Funktion einzelner Bits mittels Präprozessordirektiven festlegen
#DEFINE BAUSTEIN_LED1 (1 << 12)
#DEFINE BAUSTEIN_LED2 (1 << 13)

// das Bit 12 setzen ohne andere Bits zu verändern
bausteinA->REGISTER0 = bausteinA->REGISTER0|BAUSTEIN_LED1;
// oder in Kurzform auch so
bausteinA->REGISTER0 |= BAUSTEIN_LED1;
```

Registerfelder in C

- Häufig werden mehrere Register als Vektor von Registern zusammengefasst.

```
// Deklaration einer Struktur, die einen Registervektor enthaelt
typedef struct {
    register16 REGISTER0;    // Offset 0
    // Alignment auf 32-Bit-Grenze
    register16 reserved0;    // Offset 2
   // Deklaration eines Vektors aus 32-Bit-Registern
    register32 VECTOR[4];    // Offset [4, 8, 12, 16]
} structBaustein; // Groesse im Speicher insgesamt 20 Byte

// Funktionsbeschreibung eines Registers in einem Vektor
#DEFINE VEC_TIMER0 0
#DEFINE VEC_TIMER1 1
#DEFINE VEC_INT0 2
#DEFINE VEC_INT1 3
…
// Zeiger auf den Baustein
structBaustein *baustein1 = baustein_basisadresse;
// Zuweisung zu dem Element VEC_TIMER1 des Vektors
baustein1->VECTOR[VEC_TIMER1] = 0;
```

Board Support Packages

Jedes Board mit Mikrocontroller erfordert spezifische Angaben und Programmcode:

- Adressen der Peripheriekomponenten in der Memory Map
- Datenstrukturen zur Beschreibung der Programmierschnittstelle einer Peripheriekomponente und der enthaltenen Register
- Mikrocontrollerspezifische Konstanten für die Bedeutung von Bits in Registern
- Boardspezifische Konstanten zur Definition der Verdrahtung der angeschlossenen Sensoren und Aktoren an die Anschlüsse des Mikrocontrollers
- Startupcode und grundsätzliche Funktionen zur Initialisierung, bevor das Anwendungsprogramm also main() aufgerufen wird.
- Weitere Hilfsmittel zur vereinfachten Programmierung

Das Board Support Package (BSP) eines Herstellers liefert all dieses in den jeweils unterstützten Programmiersprachen zu einem Board.

Optimierungsstufen bei Compilern

- Unterschiedliche Optimierungen können auf Basis des gleichen Quelltextes zu sehr unterschiedlichem Maschinencode führen.
  Optimierungsanalysen stellen fest, ob Ergebnisse weiterverwendet werden und die Abhängigkeiten hierzu.

- Allerdings erfolgen die Analysen rein auf Quelltextbasis, d.h. es wird nur der Ablauf analysiert, den der Prozessorkern durchführt. Andere Einflüsse werden nicht erkannt.

- Optimierungsstufen sollten bei hardwarenaher Programmierung vorsichtig eingesetzt werden.

Optimierungsstufen bei Compilern

gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

Quelle: https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Syntax

```
$ gcc -Olevel [options] [source files] [object files] [-o output file]
```

Überblick: GCC-Optimierungsstufen -O(ptimize)*X*

- -O0
  - Keine Optimierung Schnell zu kompilieren, perfekt zum Debuggen (Zeilen-Mapping bleibt stabil).
  - Größerer, langsamerer Code; oft mehr Speicherzugriffe.
- -Og
  - Debugging mit „sinnvollen“ Optimierungen (bessere Laufzeit als -O0, Debuggability bleibt akzeptabel).
- -O1
  - Leichte Optimierung (tote-Code-Elimination, einfache Schleifen- und Peephole-Optimierungen).
  - Meist ohne Risiko für Codegröße/Kompilierzeit.
- -O2
  - Standard für Release in vielen Projekten.
  - Breites Bündel (Inlining, Schleifenoptimierung, Common Subexpression Elimination (CSE), Code-Motion, partielle Vektorisierung).
  - Gute Balance aus Speed/Größe/Stabilität.

Überblick: GCC-Optimierungsstufen

- -O3
  - Aggressiv: mehr Inlining, Loop-Unrolling, (Auto-)Vektorisierung.
  - Bessere Laufzeit, aber längere Kompilierzeit, u. U. größere Binaries.
- -Os
  - Optimieren für kleine Codegröße (wichtig bei Flash-Budget). Kann Geschwindigkeit manchmal verschlechtern.
- -Oz*
  - „Extra size-optimiert“ (primär bei Clang verbreitet; GCC unterstützt es in neueren Versionen). Nur nutzen, wenn es verfügbar ist und Größe kritisch ist.
- -Ofast
  - Wie -O3, aber u. a. mit -ffast-math (lockert IEEE-Regeln). Nur, wenn das numerisch unkritisch ist.

Überblick: GCC-Optimierungsstufen

- Weitere „Hebel“ für Embedded Systems:
  - Link-Time-Optimization: -flto (linker muss es können) → globale Inlining/Optimierung.
  - Ziel-CPU/FPU angeben: -mcpu=cortex-a9 -mfpu=neon -mfloat-abi=hard (oder passend zur Plattform). Ohne korrekte -mcpu/-mfpu bremst ihr Auto-Vektorisierung aus.

Optimierung – volatile

volatile Typqualifikator *(Quelle: Wikipedia)*

Volatile ist ein Zusatz bei der Deklaration von Variablen in C und C++.

Dadurch wird festgelegt, dass sich der Wert der Variablen außerhalb des aktuellen Programmkontextes ändern kann, beispielsweise durch externe Hardware.

Bei der Generierung des Maschinen-Codes aus einem in C oder C++ geschriebenen Programm **verhindert** die Kennzeichnung einer Variablen als volatile eine in diesem Fall die Funktionalität beeinträchtigende **Optimierung**, so dass das Programm immer auf den tatsächlich in der Hardware vorhandenen Wert zugreift.

Beispiel in C

Ohne den Zusatz volatile könnte der Compiler die Schleife im folgenden Programmausschnitt durch eine einfache Endlosschleife ersetzen und die Variable ***status*** „wegoptimieren“:

```
static volatile int status = 0;
void poll_status( void ) { while ( status == 0 ) ; }
```

Beschreibung – volatile

volatile *declarator* *(Quelle: MSDN Library)*

The volatile keyword is a type qualifier used to declare that an object can be modified in the program by something **other than statements**, such as the operating system, **the hardware**, or a concurrently executing thread.

The following example declares a volatile integer nVint whose value can be modified by external processes:

int volatile nVint;

Objects declared as volatile are **not used in optimizations** because their value can change at any time. The system always reads the current value of a volatile object at the point it is requested, even if the previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

One use of the volatile qualifier is to provide access to memory locations used by asynchronous processes such as **interrupt handlers**.

Optimierung – Stufe 0

C und Assembler bei der Optimierungsstufe –O0

01-volatile\01-volatile.c | boot\boot_ice.S

```
1  // Beispiel zur Wirkung des Keywords volatile
2
3  // Definition von globalen Variablen
4  int globvar1;
5  volatile int globvar2;
6
7
8  // Einsprung
9  int main(void) {
10     int locvar1;
11
12     globvar1 = 0xAAFFAAFF;
13     globvar2 = 0xFFAAFFAA;
14
15     locvar1 = globvar1;
16     locvar1 = globvar2;
17
18     globvar1 = locvar1;
19     globvar2 = locvar1;
20
21     return 0;
22 }
```

Disassembly

Function:

Frame start: 02040000

```
0x020001D0  push  {r11}              ; (str r11, [sp, #-4]!)
0x020001D4  add   r11, sp, #0
0x020001D8  sub   sp, sp, #12
0x020001DC  ldr   r3, [pc, #84]      ; 0x2000238 <main+104>
0x020001E0  ldr   r2, [pc, #84]      ; 0x200023c <main+108>
0x020001E4  str   r2, [r3]
0x020001E8  ldr   r3, [pc, #80]      ; 0x2000240 <main+112>
0x020001EC  ldr   r2, [pc, #80]      ; 0x2000244 <main+116>
0x020001F0  str   r2, [r3]
0x020001F4  ldr   r3, [pc, #60]      ; 0x2000238 <main+104>
0x020001F8  ldr   r3, [r3]
0x020001FC  str   r3, [r11, #-8]
0x02000200  ldr   r3, [pc, #56]      ; 0x2000240 <main+112>
0x02000204  ldr   r3, [r3]
0x02000208  str   r3, [r11, #-8]
0x0200020C  ldr   r3, [pc, #36]      ; 0x2000238 <main+104>
0x02000210  ldr   r2, [r11, #-8]
0x02000214  str   r2, [r3]
0x02000218  ldr   r3, [pc, #32]      ; 0x2000240 <main+112>
0x0200021C  ldr   r2, [r11, #-8]
0x02000220  str   r2, [r3]
0x02000224  mov   r3, #0
0x02000228  mov   r0, r3
0x0200022C  add   sp, r11, #0
0x02000230  pop   {r11}
0x02000234  bx    lr
```

Optimierung – Stufe 3

C und Assembler bei der Optimierungsstufe –O3

01-volatile\01-volatile.c | 01-volatile\makefile | boot\boot_ice.S

```
1  // Beispiel zur Wirkung des Keywords volatile
2
3  // Definition von globalen Variablen
4  int globvar1;
5  volatile int globvar2;
6
7
8  // Einsprung
9  int main(void) {
10     int locvar1;
11
12     globvar1 = 0xAAFFAAFF;
13     globvar2 = 0xFFAAFFAA;
14
15     locvar1 = globvar1;
16     locvar1 = globvar2;
17
18     globvar1 = locvar1;
19     globvar2 = locvar1;
20
21     return 0;
22 }
23
```

Disassembly

Function:

Frame start: 02040000

```
0x02000000      ldr     r3, [pc, #28]   ; 0x2000024 <main+36>
0x02000004      ldr     r2, [pc, #28]   ; 0x2000028 <main+40>
0x02000008      str     r2, [r3]
0x0200000C      ldr     r1, [pc, #24]   ; 0x200002c <main+44>
0x02000010      ldr     r2, [r3]
0x02000014      mov     r0, #0
0x02000018      str     r2, [r1]
0x0200001C      str     r2, [r3]
0x02000020      bx      lr
```

Optimierung – Beispiel

Interpretation des Assemblercode –O3

```
0x02000000      ldr     r3, [pc, #28]
0x02000004      ldr     r2, [pc, #28]
0x02000008      str     r2, [r3]
0x0200000C      ldr     r1, [pc, #24]
0x02000010      ldr     r2, [r3]
0x02000014      mov     r0, #0
0x02000018      str     r2, [r1]
0x0200001C      str     r2, [r3]
0x02000020      bx      lr
```

Als „Pseudo“-C Notation

```
C: globvar2 = 0xFFAAFFAA;
        // &globvar2
        [R3] = *[0x2000024]
        // 0xFFAAFFAA
        [R2] = *[0x2000028]
        Speichern der Konstanten
        *[R3] = [R2]
```

```
// Laden der Adresse von globvar1 in R1
[R1] = &globvar1;
```

```
0x02000020:  e12fff1e 02008200 ffaaffaa 02008204
```

Memory Dump ab 0x02000020

Interpretation des Speicherinhalts

```
0x2000024 => Adresse &globvar2
0x2000028 => Konstante
0xFFAAFFAA
0x200002C => Adresse &globvar1
```

```
// => Erneutes Laden des Inhalts von
// globvar2, falls dieser sich geaendert
// haben sollte: C: locvar1 = globvar2;
[R2] = *[R3]
```

```
// Rückgabewert return 0;
```

```
C: globvar1 = locvar1;
*[R1] = [R2]
C: globvar2 = locvar1;
*[R3] = [R2]
```